[1. 文档控制 4](#_Toc7282)

[1.1. 修改记录 4](#_Toc4671)

[1.2. 分发 4](#_Toc2962)

[1.3. 文档格式说明 4](#_Toc4502)

[2. 目录 6](#_Toc25340)

[1. Redis相关概念 10](#_Toc22568)

[1.1. 数据类型 11](#_Toc16765)

[1.2. 内存存储 11](#_Toc23582)

[1.3. 持久化 12](#_Toc13514)

[1.4. 主从同步 12](#_Toc27594)

[1.5. 性能 12](#_Toc5039)

[1.6. 提供API的语言 12](#_Toc17790)

[2. Redis设计规范 13](#_Toc17161)

[2.1. 软件版本介绍 13](#_Toc9006)

[2.2. 目前稳定版本 13](#_Toc862)

[2.3. Redis与其他数据库和软件区别 13](#_Toc35)

[2.4. Redis使用理由 14](#_Toc5187)

[2.4.1. 取最新的N个数据的操作 15](#_Toc20810)

[2.4.2. 排行榜应用，取TOP操作 15](#_Toc9206)

[2.4.3. 需要精准设定过期时间的应用 15](#_Toc27867)

[2.4.4. 计数器应用 15](#_Toc29917)

[2.4.5. Uniq操作，获取某时间所有数据排重值 16](#_Toc1142)

[2.4.6. 实时系统，反垃圾系统 16](#_Toc12220)

[2.4.7. Pub/Sub构建实时消息系统 16](#_Toc27879)

[2.4.8. 构建队列系统 16](#_Toc20708)

[2.4.9. 缓存 16](#_Toc2340)

[2.5. Redis 持久化 16](#_Toc20401)

[2.5.1. Redis不同级别持久化 16](#_Toc6737)

[2.5.2. 不同级别的优缺点 17](#_Toc23305)

[2.5.3. 如何选择持久化方式 17](#_Toc1227)

[2.6. 复制 18](#_Toc6841)

[2.6.1. 架构图 18](#_Toc23168)

[2.6.2. Resid复制特性 18](#_Toc9449)

[2.6.3. Redis复制原理 19](#_Toc15652)

[2.6.4. 适用场景 19](#_Toc4092)

[2.7. 主从复制keepalived高可用方案 20](#_Toc32521)

[2.7.1. 设计思路 20](#_Toc3218)

[2.7.2. 架构图 20](#_Toc24473)

[2.7.3. 工作原理 20](#_Toc4237)

[2.7.4. 使用场景 21](#_Toc26824)

[2.8. Redis 集群 21](#_Toc9027)

[2.8.1. 背景 21](#_Toc23803)

[2.8.2. Redis官方集群方案 Redis Cluster 21](#_Toc7242)

[2.8.3. Redis+TwemProxy（nutcracker）集群方案 24](#_Toc24880)

[2.8.4. Redis Codis集群方案 26](#_Toc29044)

[2.9. 安全 28](#_Toc11635)

[2.9.1. 设置访问规则 28](#_Toc22019)

[2.9.2. 设置数据库密码 28](#_Toc7647)

[2.9.3. 重新命名命令 29](#_Toc15382)

[2.10. 备份redis数据 29](#_Toc4283)

[2.11. 容灾备份 30](#_Toc4830)

[3. Redis安装部署规范 30](#_Toc8821)

[3.1. Redis主从配置及keepalived实现Redis高可用 30](#_Toc12060)

[3.1.1. 架构图： 31](#_Toc7821)

[3.1.2. 环境说明 31](#_Toc30969)

[3.1.3. 主机环境配置 32](#_Toc29863)

[3.1.4. 安装redis keepalived 32](#_Toc4410)

[3.1.5. 配置redis主从 33](#_Toc15196)

[3.1.6. 配置keepalived高可用 36](#_Toc29010)

[3.1.7. 创建redis监控脚本 38](#_Toc22159)

[3.2. Redis+TwemProxy（nutcracker）集群 41](#_Toc25837)

[3.2.1. 架构图 42](#_Toc17731)

[3.2.2. 环境说明 42](#_Toc28900)

[3.2.3. 主机环境配置 43](#_Toc31558)

[3.2.4. Redis主从搭建 43](#_Toc6843)

[3.2.5. nutcracker 安装部署 43](#_Toc3387)

[3.3. Redis Codis集群安装部署 49](#_Toc23827)

[3.3.1. 关于安装部署 50](#_Toc27061)

[4. Redis操作规范 51](#_Toc14524)

[4.1. 操作数据库 51](#_Toc28816)

[4.1.1. 插入数据 51](#_Toc3914)

[4.1.2. 查询数据 51](#_Toc424)

[4.1.3. 删除键值 51](#_Toc7001)

[4.1.4. 验证键是否存在 51](#_Toc9878)

[4.2. Redis 数据类型及操作 52](#_Toc15331)

[4.2.1. strings 类型及操作 52](#_Toc22308)

[4.2.2. hashes 类型及操作 59](#_Toc13140)

[4.2.3. lists 类型及操作 63](#_Toc5784)

[4.2.4. sets 类型及操作 68](#_Toc16455)

[4.2.5. sorted sets 类型及操作 74](#_Toc22523)

[4.3. Redis 键值相关命令 80](#_Toc25865)

[4.3.1. Keys 80](#_Toc18961)

[4.3.2. exists 81](#_Toc7254)

[4.3.3. del 81](#_Toc28573)

[4.3.4. expire 81](#_Toc27891)

[4.3.5. move 82](#_Toc20254)

[4.3.6. persist 82](#_Toc6921)

[4.3.7. randomkey 83](#_Toc7473)

[4.3.8. rename 83](#_Toc28017)

[4.3.9. type 84](#_Toc7988)

[4.4. 服务器相关命令 84](#_Toc28245)

[4.4.1. Ping 84](#_Toc12261)

[4.4.2. echo 84](#_Toc2506)

[4.4.3. Select 85](#_Toc24078)

[4.4.4. quit 85](#_Toc16979)

[4.4.5. Dbsize 85](#_Toc7554)

[4.4.6. info 86](#_Toc12911)

[4.4.7. monitor 86](#_Toc1258)

[4.4.8. config get 86](#_Toc2975)

[4.4.9. flushdb 87](#_Toc10051)

[4.4.10. flushall 87](#_Toc1665)

[5. Redis开发规范 88](#_Toc19957)

[5.1. 冷热数据分离，不要将所有数据全部都放到Redis中 88](#_Toc5374)

[5.2. 不同的业务数据要分开存储 88](#_Toc16615)

[5.3. 存储的Key一定要设置超时时间 88](#_Toc28307)

[5.4. 对于必须要存储的大文本数据一定要压缩后存储 88](#_Toc29700)

[5.5. 线上Redis禁止使用Keys正则匹配操作 88](#_Toc4163)

[5.6. 可靠的消息队列服务 88](#_Toc28165)

[5.7. 谨慎全量操作Hash、Set等集合结构 89](#_Toc21681)

[5.8. 根据业务场景合理使用不同的数据结构类型 89](#_Toc6788)

**Redis规划与操作规范**

|  |  |
| --- | --- |
| 作者： | 广州云本开源软件有限公司 |
| 创建时间： | 2017年10月22日 |
| 最新更新时间： |  |
| 版本： | 1.0 |

# 文档控制

## 修改记录

| **日期** | **作者** | **版本** | **修改参考** |
| --- | --- | --- | --- |
|  |  |  |  |
| 2017-10-22 |  | 1．0 |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

## 分发

| **编号** | **名称** | **地点** |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

## 文档格式说明

本文档采用以下格式：

* 命令、代码等指令型内容：

execute command 1

# 目录

[1. 文档控制 4](#_Toc8658)

[1.1. 修改记录 4](#_Toc16750)

[1.2. 分发 4](#_Toc19453)

[1.3. 文档格式说明 4](#_Toc668)

[2. 目录 6](#_Toc25743)

[1. Redis相关概念 10](#_Toc21301)

[1.1. 数据类型 11](#_Toc6137)

[1.2. 内存存储 11](#_Toc27896)

[1.3. 持久化 12](#_Toc17349)

[1.4. 主从同步 12](#_Toc16109)

[1.5. 性能 12](#_Toc3069)

[1.6. 提供API的语言 12](#_Toc29095)

[2. Redis设计规范 13](#_Toc201)

[2.1. 软件版本介绍 13](#_Toc24301)

[2.2. 目前稳定版本 13](#_Toc32278)

[2.3. Redis与其他数据库和软件区别 13](#_Toc8015)

[2.4. Redis使用理由 14](#_Toc32057)

[2.4.1. 取最新的N个数据的操作 15](#_Toc9644)

[2.4.2. 排行榜应用，取TOP操作 15](#_Toc24512)

[2.4.3. 需要精准设定过期时间的应用 15](#_Toc17967)

[2.4.4. 计数器应用 15](#_Toc19495)

[2.4.5. Uniq操作，获取某时间所有数据排重值 16](#_Toc14498)

[2.4.6. 实时系统，反垃圾系统 16](#_Toc14716)

[2.4.7. Pub/Sub构建实时消息系统 16](#_Toc24507)

[2.4.8. 构建队列系统 16](#_Toc16259)

[2.4.9. 缓存 16](#_Toc462)

[2.5. Redis 持久化 16](#_Toc19063)

[2.5.1. Redis不同级别持久化 16](#_Toc25745)

[2.5.2. 不同级别的优缺点 17](#_Toc4596)

[2.5.3. 如何选择持久化方式 17](#_Toc20202)

[2.6. 复制 18](#_Toc24603)

[2.6.1. 架构图 18](#_Toc19966)

[2.6.2. Resid复制特性 18](#_Toc29405)

[2.6.3. Redis复制原理 19](#_Toc32728)

[2.6.4. 适用场景 19](#_Toc20623)

[2.7. 主从复制keepalived高可用方案 20](#_Toc30753)

[2.7.1. 设计思路 20](#_Toc29241)

[2.7.2. 架构图 20](#_Toc14269)

[2.7.3. 工作原理 20](#_Toc31033)

[2.7.4. 使用场景 21](#_Toc13911)

[2.8. Redis 集群 21](#_Toc4921)

[2.8.1. 背景 21](#_Toc13585)

[2.8.2. Redis官方集群方案 Redis Cluster 21](#_Toc1666)

[2.8.3. Redis+TwemProxy（nutcracker）集群方案 24](#_Toc3063)

[2.8.4. Redis Codis集群方案 26](#_Toc29230)

[2.9. 安全 28](#_Toc28658)

[2.9.1. 设置访问规则 28](#_Toc26841)

[2.9.2. 设置数据库密码 28](#_Toc23872)

[2.9.3. 重新命名命令 29](#_Toc426)

[2.10. 备份redis数据 29](#_Toc29374)

[2.11. 容灾备份 30](#_Toc6906)

[3. Redis安装部署规范 30](#_Toc11407)

[3.1. Redis主从配置及keepalived实现Redis高可用 30](#_Toc8930)

[3.1.1. 架构图： 31](#_Toc8416)

[3.1.2. 环境说明 31](#_Toc7536)

[3.1.3. 主机环境配置 32](#_Toc14293)

[3.1.4. 安装redis keepalived 32](#_Toc7511)

[3.1.5. 配置redis主从 33](#_Toc24554)

[3.1.6. 配置keepalived高可用 36](#_Toc19288)

[3.1.7. 创建redis监控脚本 38](#_Toc2633)

[3.2. Redis+TwemProxy（nutcracker）集群 41](#_Toc14774)

[3.2.1. 架构图 42](#_Toc22245)

[3.2.2. 环境说明 42](#_Toc27753)

[3.2.3. 主机环境配置 43](#_Toc22106)

[3.2.4. Redis主从搭建 43](#_Toc1020)

[3.2.5. nutcracker 安装部署 43](#_Toc14308)

[3.3. Redis Codis集群安装部署 49](#_Toc10289)

[3.3.1. 关于安装部署 50](#_Toc11240)

[4. Redis操作规范 51](#_Toc30941)

[4.1. 操作数据库 51](#_Toc18400)

[4.1.1. 插入数据 51](#_Toc16033)

[4.1.2. 查询数据 51](#_Toc24243)

[4.1.3. 删除键值 51](#_Toc8807)

[4.1.4. 验证键是否存在 51](#_Toc6335)

[4.2. Redis 数据类型及操作 52](#_Toc10700)

[4.2.1. strings 类型及操作 52](#_Toc12564)

[4.2.2. hashes 类型及操作 59](#_Toc26595)

[4.2.3. lists 类型及操作 63](#_Toc10678)

[4.2.4. sets 类型及操作 68](#_Toc3308)

[4.2.5. sorted sets 类型及操作 74](#_Toc5867)

[4.3. Redis 键值相关命令 80](#_Toc5372)

[4.3.1. Keys 80](#_Toc25041)

[4.3.2. exists 81](#_Toc5286)

[4.3.3. del 81](#_Toc9432)

[4.3.4. expire 81](#_Toc23587)

[4.3.5. move 82](#_Toc6352)

[4.3.6. persist 82](#_Toc32686)

[4.3.7. randomkey 83](#_Toc22577)

[4.3.8. rename 83](#_Toc8757)

[4.3.9. type 84](#_Toc11758)

[4.4. 服务器相关命令 84](#_Toc24612)

[4.4.1. Ping 84](#_Toc20814)

[4.4.2. echo 84](#_Toc3739)

[4.4.3. Select 85](#_Toc27588)

[4.4.4. quit 85](#_Toc32462)

[4.4.5. Dbsize 85](#_Toc18706)

[4.4.6. info 86](#_Toc5961)

[4.4.7. monitor 86](#_Toc16393)

[4.4.8. config get 86](#_Toc31820)

[4.4.9. flushdb 87](#_Toc4587)

[4.4.10. flushall 87](#_Toc26631)

[5. Redis开发规范 88](#_Toc32313)

[5.1. 冷热数据分离，不要将所有数据全部都放到Redis中 88](#_Toc6594)

[5.2. 不同的业务数据要分开存储 88](#_Toc30143)

[5.3. 存储的Key一定要设置超时时间 88](#_Toc15153)

[5.4. 对于必须要存储的大文本数据一定要压缩后存储 88](#_Toc28431)

[5.5. 线上Redis禁止使用Keys正则匹配操作 88](#_Toc21896)

[5.6. 可靠的消息队列服务 88](#_Toc12242)

[5.7. 谨慎全量操作Hash、Set等集合结构 89](#_Toc31753)

[5.8. 根据业务场景合理使用不同的数据结构类型 89](#_Toc14736)

[1. 文档控制 4](#_Toc24614)

[1.1. 修改记录 4](#_Toc13536)

[1.2. 分发 4](#_Toc19133)

[1.3. 文档格式说明 4](#_Toc19956)

[2. 目录 6](#_Toc13079)

[1. Redis相关概念 12](#_Toc16329)

[1.1. 数据类型 13](#_Toc3092)

[1.2. 内存存储 14](#_Toc7164)

[1.3. 持久化 14](#_Toc27161)

[1.4. 主从同步 14](#_Toc12726)

[1.5. 性能 14](#_Toc691)

[1.6. 提供API的语言 14](#_Toc7983)

[2. Redis设计规范 15](#_Toc18070)

[2.1. 软件版本介绍 15](#_Toc6202)

[2.2. 目前稳定版本 15](#_Toc20596)

[2.3. Redis与其他数据库和软件区别 16](#_Toc21066)

[2.4. Redis使用理由 16](#_Toc30570)

[2.4.1. 取最新的N个数据的操作 17](#_Toc18835)

[2.4.2. 排行榜应用，取TOP操作 17](#_Toc2599)

[2.4.3. 需要精准设定过期时间的应用 17](#_Toc28722)

[2.4.4. 计数器应用 18](#_Toc8179)

[2.4.5. Uniq操作，获取某时间所有数据排重值 18](#_Toc8056)

[2.4.6. 实时系统，反垃圾系统 18](#_Toc5951)

[2.4.7. Pub/Sub构建实时消息系统 18](#_Toc9434)

[2.4.8. 构建队列系统 18](#_Toc21863)

[2.4.9. 缓存 18](#_Toc20242)

[2.5. Redis 持久化 18](#_Toc14795)

[2.5.1. Redis不同级别持久化 18](#_Toc17767)

[2.5.2. 不同级别的优缺点 19](#_Toc1643)

[2.5.3. 如何选择持久化方式 20](#_Toc584)

[2.6. 复制 20](#_Toc21160)

[2.6.1. 架构图 20](#_Toc21394)

[2.6.2. Resid复制特性 21](#_Toc225)

[2.6.3. Redis复制原理 21](#_Toc5695)

[2.6.4. 适用场景 21](#_Toc13813)

[2.7. 主从复制keepalived高可用方案 22](#_Toc20558)

[2.7.1. 设计思路 22](#_Toc2561)

[2.7.2. 架构图 22](#_Toc8993)

[2.7.3. 工作原理 22](#_Toc9160)

[2.7.4. 使用场景 23](#_Toc8619)

[2.8. Redis 集群 23](#_Toc15531)

[2.8.1. 背景 23](#_Toc12574)

[2.8.2. Redis官方集群方案 Redis Cluster 23](#_Toc14623)

[2.8.3. Redis+TwemProxy（nutcracker）集群方案 26](#_Toc4091)

[2.8.4. Redis Codis集群方案 28](#_Toc8699)

[2.9. 安全 30](#_Toc16049)

[2.9.1. 设置访问规则 30](#_Toc27930)

[2.9.2. 设置数据库密码 30](#_Toc7731)

[2.9.3. 重新命名命令 31](#_Toc14367)

[2.10. 备份redis数据 31](#_Toc31781)

[2.11. 容灾备份 32](#_Toc28263)

[3. Redis安装部署规范 32](#_Toc16788)

[3.1. Redis主从配置及keepalived实现Redis高可用 32](#_Toc23397)

[3.1.1. 架构图： 33](#_Toc24663)

[3.1.2. 环境说明 33](#_Toc29949)

[3.1.3. 主机环境配置 34](#_Toc15498)

[3.1.4. 安装redis keepalived 34](#_Toc17816)

[3.1.5. 配置redis主从 35](#_Toc31421)

[3.1.6. 配置keepalived高可用 38](#_Toc23565)

[3.1.7. 创建redis监控脚本 40](#_Toc8404)

[3.2. Redis+TwemProxy（nutcracker）集群 43](#_Toc6029)

[3.2.1. 架构图 44](#_Toc9895)

[3.2.2. 环境说明 44](#_Toc6197)

[3.2.3. 主机环境配置 45](#_Toc15589)

[3.2.4. Redis主从搭建 45](#_Toc8441)

[3.2.5. nutcracker 安装部署 45](#_Toc965)

[3.3. Redis Codis集群安装部署 51](#_Toc24056)

[3.3.1. 关于安装部署 52](#_Toc30858)

[4. Redis操作规范 53](#_Toc23173)

[4.1. 操作数据库 53](#_Toc22506)

[4.1.1. 插入数据 53](#_Toc3179)

[4.1.2. 查询数据 53](#_Toc25806)

[4.1.3. 删除键值 53](#_Toc17957)

[4.1.4. 验证键是否存在 53](#_Toc15887)

[4.2. Redis 数据类型及操作 54](#_Toc12336)

[4.2.1. strings 类型及操作 54](#_Toc795)

[4.2.2. hashes 类型及操作 61](#_Toc10661)

[4.2.3. lists 类型及操作 65](#_Toc18075)

[4.2.4. sets 类型及操作 70](#_Toc12968)

[4.2.5. sorted sets 类型及操作 76](#_Toc1953)

[4.3. Redis 键值相关命令 82](#_Toc22405)

[4.3.1. Keys 82](#_Toc27874)

[4.3.2. exists 83](#_Toc2405)

[4.3.3. del 83](#_Toc6907)

[4.3.4. expire 83](#_Toc31238)

[4.3.5. move 84](#_Toc1515)

[4.3.6. persist 84](#_Toc31198)

[4.3.7. randomkey 85](#_Toc24336)

[4.3.8. rename 85](#_Toc5253)

[4.3.9. type 86](#_Toc3190)

[4.4. 服务器相关命令 86](#_Toc15160)

[4.4.1. Ping 86](#_Toc12040)

[4.4.2. echo 86](#_Toc28221)

[4.4.3. Select 87](#_Toc20832)

[4.4.4. quit 87](#_Toc13217)

[4.4.5. Dbsize 87](#_Toc24599)

[4.4.6. info 88](#_Toc15995)

[4.4.7. monitor 88](#_Toc8108)

[4.4.8. config get 88](#_Toc20464)

[4.4.9. flushdb 89](#_Toc10445)

[4.4.10. flushall 89](#_Toc29099)

[5. Redis开发规范 90](#_Toc11109)

[5.1. 冷热数据分离，不要将所有数据全部都放到Redis中 90](#_Toc21803)

[5.2. 不同的业务数据要分开存储 90](#_Toc22318)

[5.3. 存储的Key一定要设置超时时间 90](#_Toc1682)

[5.4. 对于必须要存储的大文本数据一定要压缩后存储 90](#_Toc32205)

[5.5. 线上Redis禁止使用Keys正则匹配操作 90](#_Toc22230)

[5.6. 可靠的消息队列服务 90](#_Toc25275)

[5.7. 谨慎全量操作Hash、Set等集合结构 91](#_Toc16511)

[5.8. 根据业务场景合理使用不同的数据结构类型 91](#_Toc3976)

# Redis相关概念

* Redis是一个key-value存储系统，和Memacached类似，它支持存储value类型相对更多，包括string（字符串）、list（链表）、set（集合）和zset（有序集合）。这些数据类型都支持push/pop、add/remove及取交集和差集及更丰富的操作，而且这些操作是原子性的。在此基础上，Redis支持各种不同方式的排序。与memcached一样，为了保证效率，数据都是缓存在内存中的。区别的是Redsi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。
* Key-value store注重对海量数据存取的性能、分布式、扩展性支持上。并不需要传统关系型数据库的一些特性，例如：schema、失误、完整sql查询支持等等。因此分布式环境下的性能相对传统的关系型数据库有较大的提升。

**Key-value数据库分为很多种类，具体如下图：**



## 数据类型

* 作为key-value型数据库，Redis也提供了键（key）和键值（value）的映射关系。但是，除了常规的数值和字符串,Redis的键值还可以是以下三种形式之一：
* List（列表）
* Set（集合）
* Sotted sets（有序集合）
* Hashes（哈希表）

键值的数据类型界定了该兼职支持的操作。Redis支持诸如列表、集合或者有序集合的交集、并集、查集等高级原子操作：同时，如果键值的类型是普通的数字，Redis则提供自增等原子操作。

## 内存存储

Redis数据库中的所有数据都存储在内存中，优于内存的读写熟读远快于硬盘，因此Redis在性能上对比其他的基于硬盘存储的数据库有非常明显的优势，在一台普通的笔记本上，Redis可以在一秒内读写超过十万个键值。

## 持久化

通常，Redis将数据存储于内存中，或被配置为使用虚拟内存。将数据存储在内存中也有问题，例如，程序退出后内存中的数据会丢失，不过Redis提供了对持久化的支持，通常两种方式可以试下数据持久化：使用截图的方式，将内存中的数据不断写入键盘；或使用类似mysql的日志方式，记录每次更新的日志。前者性能较高，但是可能会引起一定程度的数据丢失；后者相反。

## 主从同步

Redis支持将数据同步到多台从库上，这种特性对提高读取性能非常有益。

## 性能

相比需要依赖磁盘记录每个更新的数据库，基于内存的特性无疑给Redis带来了非常优秀的性能。读写操作之间有显著的性能差异。

## 提供API的语言

* C
* C++
* C#
* Clojure
* Common lisp
* Erlang
* Haskell
* Java
* Javascript
* Lua
* Objective-c
* Perl
* Php
* Python
* Ruby
* Scala
* Go
* tcl

# Redis设计规范

## 软件版本介绍

* Redis 版本号采用标准惯例：主版本号.副版本号.补丁级别,一个副版本号就标记为一个标准发行版本，例如 1.2，2.0，2.2，2.4，2.6，2.8，奇数的副版本号用来表示非标准版本,例如2.9.x发行版本是Redis 3.0标准版本的非标准发行版本。

## 目前稳定版本

* 目前稳定版本为4.0

**做了以下改进：**

Redis 4.0 was released as GA in July 2017. Contains several big improvements: a modules system, much better replication (PSYNC2), improvements to eviction policies, threaded DEL/FLUSH, mixed RDB+AOF format, Raspberry Pi support as primary platform, the new MEMORY command, Redis Cluster support for Nat/Docker, active memory defragmentation, memory usage and performance improvements, much faster Redis Cluster key creation, many other smaller features and a number of behavior fixed.

|  |  |  |
| --- | --- | --- |
| 发布时间 | 版本号 | 下载链接 |
| 2017年7月 | redis-4.0.0.tar.gz | http://download.redis.io/releases/redis-4.0.0.tar.gz |

## Redis与其他数据库和软件区别

表2.3-1展示了一部分在功能上与Redis有重叠的数据库服务器和缓存服务器，从这个表可以看出Redis与这些数据库及软件之间的区别

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 名称 | 类型 | 数据存储选项 | 查询类型 | 附加功能 |
| Redis | 使用内存存储（in-memory）的非关系型数据库 | 字符串、列表、集合、散列、有序集合 | 每种数据类型都有自己的专属命令，另外还有批量操作和不完全的事物支持 | 发布与订阅，主从复制，持久化，客户端分片、脚本（存储过程，stored procedure） |
| Memcached | 使用内存存储的键值缓存 | 键值之间的映射 | 创建命令、读取命令、更新命令、删除命及其其他几个命令 | 为提升性能而设的多线服务器 |
| Mysql | 关系数据库 | 每个数据库可以包含多个表，每个表可以包含多个行；可以处理多个表的视图；支持空间和第三方扩展 | SELECT、INSERT、UPDATE、DELETE、函数、存储过程 | 支持ACID性质，主从复制和主主复制 |
| PostgreSQL | 关系数据库 | 每个数据库可以包含多个表，每个表可以包含多个行；支持空间和第三方扩展；支持可定制类型 | SELECT、INSERT、UPDATE、DELETE、内置函数、自定义的存储过程 | 支持ACID性质，主从复制，由第三方支持的多主复制 |
| MongoDB | 使用硬盘存储（on-disk）的非关系型文档存储 | 每个数据库可以包含多个表，每个表可以包含多个无schema的BSON文档 | 创建命令、读取命令、更新命令、删除命令、条件查询等 | 支持map-redure操作，主从复制，分片，空间索引 |

表2.3-1

## Redis使用理由

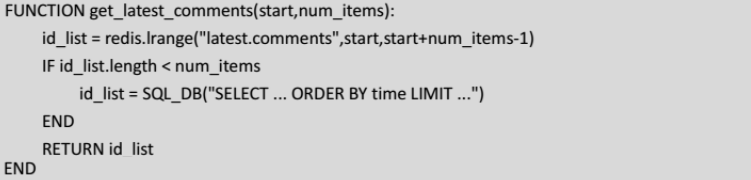
* 其实对于企业选型Memcache、Redis而言，更多还是应该看业务使用场景（因为Memcache、Redis两者都具有足够高的性能和稳定性）。假若业务场景需要用到持久化缓存功能、或者支持多种数据结构的缓存功能，那么Redis则是最佳选择。
* Redis集群解决方式也优于Memcache，Memcache在客户端一致性hash的集群解决方案，Redis采用无中心的服务器端集群解决方案
* 综上所述：为了让缓存系统能够支持更多的业务场景，选择Redis会更优。（目前也越来越多的厂商选择Redis）。

**下面是Redis适用的一些场景：**

### 取最新的N个数据的操作

比如典型的取你网站的更新文章，通过下面方式，我们可以将最新的5000条评论的ID放在Redis的List集合中，并将超出集合部分从数据库获取。

使用LPUSH latest.comments<ID>命令，向list集合中插入数据，插入完成后再用LTRIM latest.comments 0 5000 命令时期永远只保存最近5000个ID，然后我们在客户端获取某一页评论时可以使用下面的逻辑





如果你还有不同的筛选维度，比如某个分类的最新N条，那么你可以再建一个按此类分类的List，只存ID的话，Redis是非常高效的。

### 排行榜应用，取TOP操作

这个需求与商店的需求不同之处在于，前面操作以时间为权重，这个是以某个条件为权重，比如按顶的次数排序，这个时候就需要sorted set出马了，将你安排的值设置成sorted set 的score，将具体的数据设置成相应的value，每次只需要执行一条ZADD命令即可。

### 需要精准设定过期时间的应用

比如你可以把上面说到的sorted set的score值设置成过期时间的时间戳，那么久可以简单的通过过期时间排序，定时清除过期数据了，不仅是清除Redis中的过期数据，还可以把Redis里这个过期时间当成是对数据库中数据的索引，用Redis来找出那些数据需要过期删除，然后再准确的从数据库中删除相应的记录。

### 计数器应用

Redis的命令都是原子性的，你可以轻松的利用INCR,DECR命令来构建计数器系统。

### Uniq操作，获取某时间所有数据排重值

这个使用Redis的set数据结构最合适了，只需要不断地将数据往set中扔就行，set意为集合，所以会自动排重

### 实时系统，反垃圾系统

通过上面说到的set功能，你可以知道一个终端用户是否进行了某个操作，可以找到其操作的集合并进行分析统计对比等。

### Pub/Sub构建实时消息系统

Redis的Pub/Sub系统可以构建实时的消息系统，比如很多用Pub/Sub构建的实时聊天系统的例子

### 构建队列系统

使用list可以都贱队列系统，使用sorted set甚至可以构建有优先级的队列系统。

### 缓存

性能优于Memcached，数据结构多样化。

## Redis 持久化

### Redis不同级别持久化

**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.

### 不同级别的优缺点

|  |  |  |
| --- | --- | --- |
| 级别名称 | 优点 | 缺点 |
| RDB | RDB是一个非常紧凑的文件,保存了某个时间点得数据集,非常适用于数据集的备份;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持久化方式可以最大化redis的性能;恢复大的数据集的时候，RDB方式会更快一些. | 虽然可以配置不同的save时间点，但是完整保存数据集还是很繁重，Redis在意外停止工作的情况下，可能会丢失几分钟数据。 |
| AOF | AOF 会让你的Redis更加耐久，使用不同的fsync策略，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，可以使用redis-check-aof工具修复；  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；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。 | 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）。 |

### 如何选择持久化方式

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。

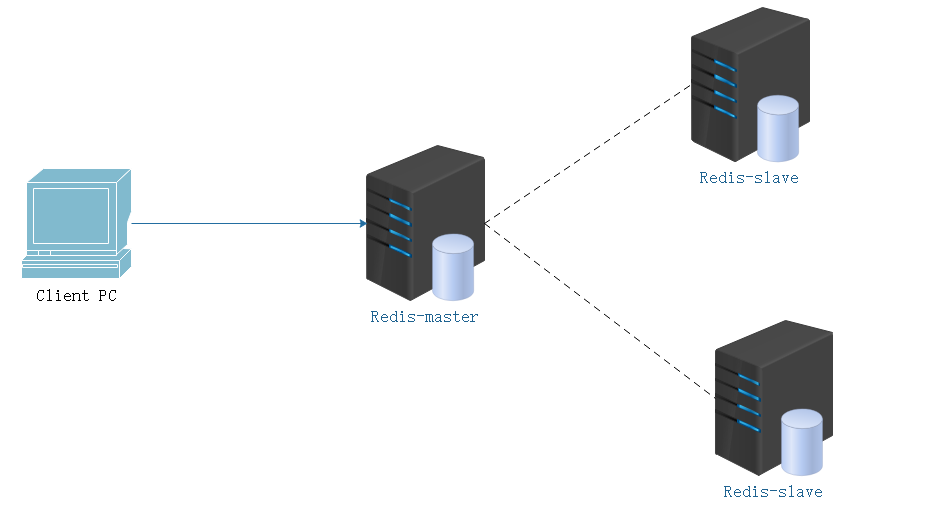
如果你非常关心你的数据， 但仍然可以承受数分钟以内的数据丢失， 那么你可以只使用 RDB 持久化。

不推荐AOF 持久化这种方式： 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 。

## 复制

* 通过持久化功能，Redis保证了即使在服务器从启的情况下也不会损失（或少量损失）数据。但是由于数据是存储在一台服务器上的，如果这台服务器的硬盘出现故障，也会导致数据丢失，为了避免单点故障，我们希望将数据库复制多个副本以部署在不同的服务器上，即使有一台出现故障其他服务器依然可以继续提供服务。这就要求当一台服务器上的数据更新后，可以将更新的数据同步到其他服务器上，Redis提供了复制（replication）功能可以自动实现同步的过程。

### 架构图



### Resid复制特性

* redis复制的非常重要特性：
* 一个Master可以有多个Slaves。
* Slaves能过接口其他slave的链接，除了可以接受同一个master下面slaves的链接以外，还可以接受同一个结构图中的其他slaves的链接。
* Redis复制是在master段是非阻塞的，这就意味着master在同一个或多个slave端执行同步的时候还可以接受查询。
* 复制在slave端也是非阻塞的，假设你在redis.conf中配置redis这个功能，当slave在执行的新的同步时，它仍可以用旧的数据信息来提供查询，否则，你可以配置当redis slaves去master失去联系是，slave会给发送一个客户端错误。
* 为了有多个slaves可以做只读查询，复制可以重复2次，甚至多次，具有可扩展性（例如：slaves对话与重复的排序操作，有多份数据冗余就相对简单了）。
* 通过复制可以避免master全量写硬盘的消耗：只要配置 master 的配置文件redis.conf来“避免保存”（注释掉所有”save”命令），然后连接一个用来持久化数据的slave即可。但是这样要确保masters 不会自动重启.

### Redis复制原理

* 如果设置了一个slave，不管是在第一次链接还是重新链接master的时候，slave会发送一个同步命令 然后master开始后台保存，收集所有对修改数据的命令。当后台保存完成，master会将这个数据文件传送到slave，然后保存在磁盘，加载到内存中；master接着发送收集到的所有的修改数据的命令，这好比一个流命令，是redis协议本身来实现的。
* 当master和slave因一些故障当机时，slaves会自动的重链，如果master收到多个slave的同步请求，master会执行一个后台保存，以确保所有的slaves都是正常的。
* 当master和slave能够维持链接，就会有一个完整的同步进行。

### 适用场景

该方案没有负载均衡高可用功能，但很大程度增强的数据的一致性且资源利用率为最大比；对并发要求不高，且对高可用需求不大可选该方案。

## 主从复制keepalived高可用方案

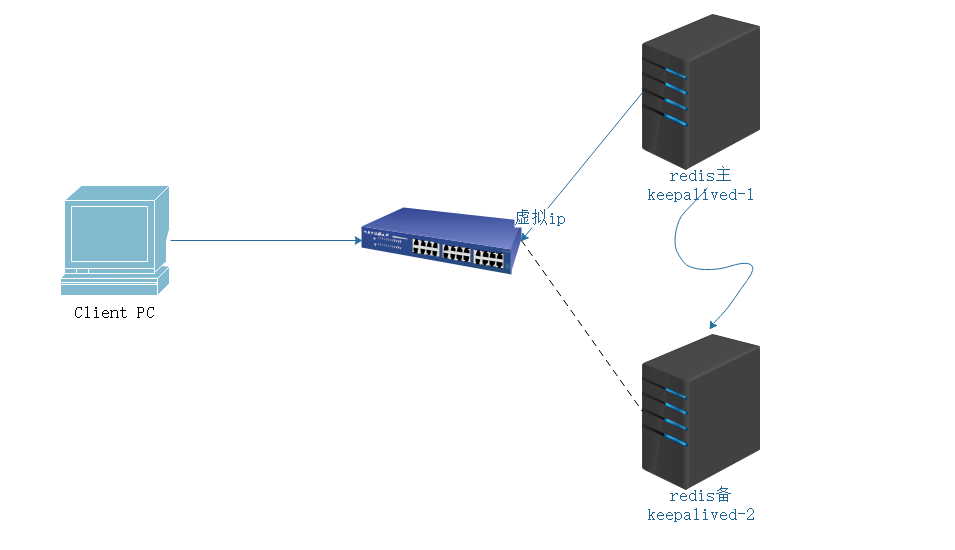
### 设计思路

当 Master 与 Slave 均运作正常时, Master负责服务，Slave负责Standby；  
当 Master 挂掉，Slave 正常时, Slave接管服务，同时关闭主从复制功能；  
当 Master 恢复正常，则从Slave同步数据，同步数据之后关闭主从复制功能，恢复Master身份，于此同时Slave等待Master同步数据完成之后，恢复Slave身份。  
然后依次循环。

注意：

该需要在Master与Slave上都开启本地化策略，否则在互相自动切换的过程中，未开启本地化的一方会将另一方的数据清空，造成数据完全丢失。

### 架构图



### 工作原理

* 该方案在Redis的master-slave方案上增加了keepalived实现Redis高可用，keepalived的状态监测脚本包含了四个脚本 分别是：
* 当keepalived进入master状态，那么它将执行notifiy\_master脚本，该脚本的内容为执行命令将本机Redis设置slave状态，等待10秒完成master-slave数据同步再设置本机为master状态。
* 当keepalived进入slave状态，那么它将执行notify\_backup脚本，等待15秒，等待数据被对方同步完成再切换为slaved状态。
* 当keepalived进入紧急情况，那么将会执行脚本notify\_fault
* 当keepalived进入终止状态将会执行脚本notify\_stop

### 使用场景

由于该方案在Redis master-salve的基础上增加了高可用性，故如果对Redis并发压力不是很大，又需要服务的高可用，那么选择该方案可以将资源利用率达到最大，且最大性质的保证了数据的一致性。

## Redis 集群

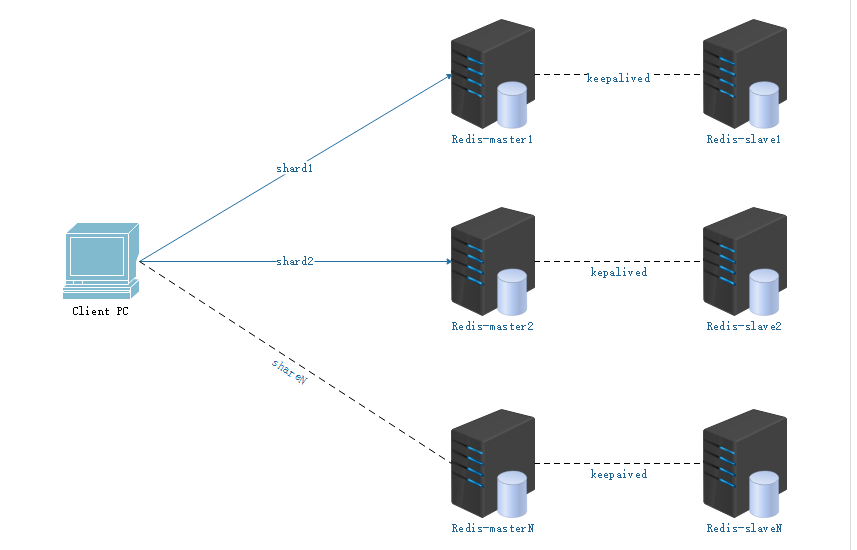
### 背景

* 长期以来，Redis本身仅支持单实例，内存一般最多10~20GB。这无法支撑大型线上业务系统的需求。而且也造成资源的利用率过低——毕竟现在服务器内存动辄100~200GB。
* 为解决单机承载能力不足的问题，各大互联网企业纷纷出手，“自助式”地实现了集群机制。在这些非官方集群解决方案中，物理上把数据“分片”（sharding）存储在多个Redis实例，一般情况下，每一“片”是一个Redis实例。

### Redis官方集群方案 Redis Cluster

* Redis 集群是一个提供在多个Redis间节点间共享数据的程序集
* Redis集群并不支持处理多个keys的命令,因为这需要在不同的节点间移动数据,从而达不到像Redis那样的性能,在高负载的情况下可能会导致不可预料的错误.
* Redis 集群通过分区来提供一定程度的可用性,在实际环境中当某个节点宕机或者不可达的情况下继续处理命令
* 优点：
* 自动分割数据到不同的节点上
* 整个集群的部分节点失败或者不可达的情况下能够继续处理命令。

#### 架构图



#### Redis 集群的数据分片

* Redis 集群没有使用一致性hash, 而是引入了哈希槽的概念.
* Redis 集群有16384个哈希槽,每个key通过CRC16校验后对16384取模来决定放置哪个槽.集群的每个节点负责一部分hash槽,举个例子,比如当前集群有3个节点,那么:
* 节点 A 包含 0 到 5500号哈希槽.
* 节点 B 包含5501 到 11000 号哈希槽.
* 节点 C 包含11001 到 16384号哈希槽.

这种结构很容易添加或者删除节点. 比如如果我想新添加个节点D, 我需要从节点 A, B, C中得部分槽到D上. 如果我像移除节点A,需要将A中得槽移到B和C节点上,然后将没有任何槽的A节点从集群中移除即可. 由于从一个节点将哈希槽移动到另一个节点并不会停止服务,所以无论添加删除或者改变某个节点的哈希槽的数量都不会造成集群不可用的状态.

#### Redis 集群的主从复制模型

* 为了使在部分节点失败或者大部分节点无法通信的情况下集群仍然可用，所以集群使用了主从复制模型,每个节点都会有N-1个复制品.
* 在我们例子中具有A，B，C三个节点的集群,在没有复制模型的情况下,如果节点B失败了，那么整个集群就会以为缺少5501-11000这个范围的槽而不可用.
* 然而如果在集群创建的时候（或者过一段时间）我们为每个节点添加一个从节点A1，B1，C1,那么整个集群便有三个master节点和三个slave节点组成，这样在节点B失败后，集群便会选举B1为新的主节点继续服务，整个集群便不会因为槽找不到而不可用了
* 不过当B和B1 都失败后，集群是不可用的.

#### Redis集群缺点 一致性保证

* Redis 并不能保证数据的强一致性. 这意味这在实际中集群在特定的条件下可能会丢失写操作

**第一个原因是因为集群是用了异步复制. 写操作过程:**

* 客户端向主节点B写入一条命令.
* 主节点B向客户端回复命令状态.
* 主节点将写操作复制给他得从节点 B1, B2 和 B3.
* **在性能和一致性之间做出权衡：**
* 主节点对命令的复制工作发生在返回命令回复之 后， 因为如果每次处理命令请求都需要等待复制操作完成的话， 那么主节点处理命令请求的速度将极大地降低
* Redis 集群可能会在将来提供同步写的方法。 Redis 集群另外一种可能会丢失命令的情况是集群出现了网络分区， 并且一个客户端与至少包括一个主节点在内的少数实例被孤立。

#### 方案优缺点

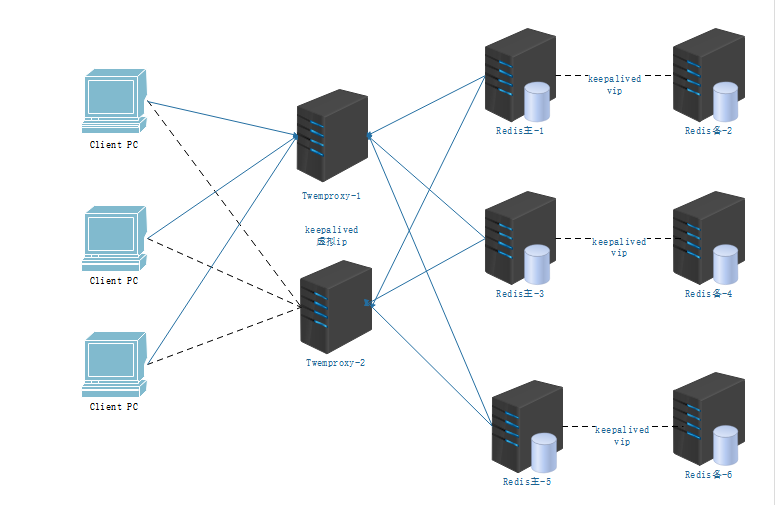
reids-cluster在redis3.0中推出，支持Redis分布式集群部署模式。采用无中心分布式架构。所有的redis节点彼此互联(PING-PONG机制),内部使用二进制协议优化传输速度和带宽.节点的fail是通过集群中超过半数的节点检测失效时才生效.客户端与redis节点直连,不需要中间proxy层.客户端不需要连接集群所有节点,连接集群中任何一个可用节点即可，减少了代理层，大大提高了性能。redis-cluster把所有的物理节点映射到[0-16383]slot上,cluster 负责维护node<->slot<->key之间的关系。目前Jedis已经支持Redis-cluster。从计算架构或者性能方面无疑Redis-cluster是最佳的选择方案。

* 虽然Redis-cluster从方案选型上面比较占据优势，但是由于Redis-cluster刚推出不久，虽然官方宣传已经发布的是文档版本，但稳定性方面还有待验证

### Redis+TwemProxy（nutcracker）集群方案

* Twemproxy 又称nutcracker ，是一个memcache、Redis协议的轻量级代理，一个用于sharding 的中间件。有了Twemproxy，客户端不直接访问Redis服务器，而是通过twemproxy 代理中间件间接访问。 Twemproxy 为 Twitter 开源产品，简单来说，Twemproxy是Twitter开发的一个redis代理proxy，类似于nginx的反向代理或者mysql的代理工具，如amoeba。Twemproxy通过引入一个代理层，可以将其后端的多台Redis或Memcached实例进行统一管理与分配，使应用程序只需要在Twemproxy上进行操作，而不用关心后面具体有多少个真实的Redis或Memcached存储。

#### 架构图



#### ****Twemproxy的特性****

1. 支持失败节点自动删除

   可以设置重新连接该节点的时间

   可以设置连接多少次之后删除该节点

1. 支持设置HashTag

   通过HashTag可以自己设定将两个key哈希到同一个实例上去

1. 减少与redis的直接连接数

   保持与redis的长连接

   减少了客户端直接与服务器连接的连接数量

1. 自动分片到后端多个redis实例上

   多种hash算法：md5、crc16、crc32 、crc32a、fnv1\_64、fnv1a\_64、fnv1\_32、fnv1a\_32、hsieh、murmur、jenkins

   多种分片算法：ketama(一致性hash算法的一种实现)、modula、random

   可以设置后端实例的权重

1. 避免单点问题

   可以平行部署多个代理层,通过HAProxy做负载均衡，将redis的读写分散到多个twemproxy上。

1. 支持状态监控

   可设置状态监控ip和端口，访问ip和端口可以得到一个json格式的状态信息串

   可设置监控信息刷新间隔时间

1. 使用 pipelining 处理请求和响应

   连接复用，内存复用

   将多个连接请求，组成reids pipelining统一向redis请求

1. 并不是支持所有redis命令

   不支持redis的事务操作

   使用SIDFF, SDIFFSTORE, SINTER, SINTERSTORE, SMOVE, SUNION and SUNIONSTORE命令需要保证key都在同一个分片上。

**注意：**

通过Twemproxy可以使用多台服务器来水平扩张redis服务，可以有效的避免单点故障问题。虽然使用Twemproxy需要更多的硬件资源和在redis性能有一定的损失（twitter测试约20%），但是能够提高整个系统的高可用

#### 方案优缺点

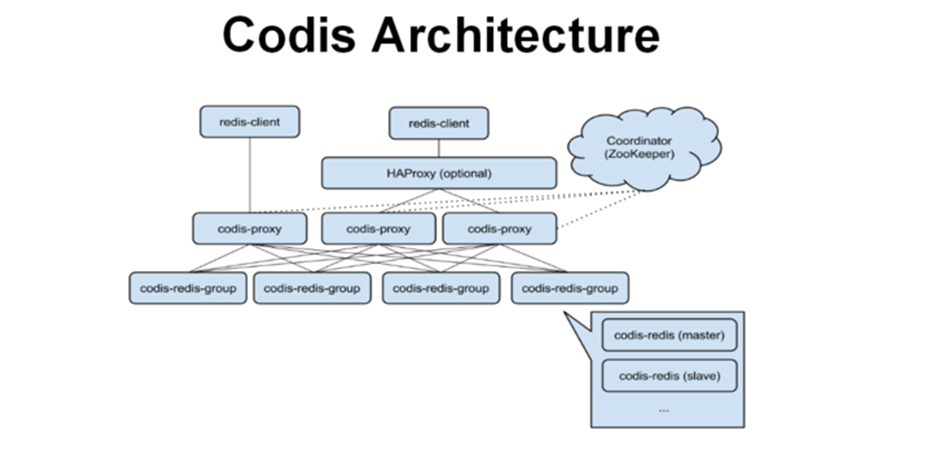
* Twemproxy是一种代理分片机制，由Twitter开源。Twemproxy作为代理，可接受来自多个程序的访问，按照路由规则，转发给后台的各个Redis服务器，再原路返回。这个方案顺理成章地解决了单个Redis实例承载能力的问题。当然，Twemproxy本身也是单点，需要用Keepalived做高可用方案。这么些年来，Twemproxy是应用范围最广、稳定性最高、最久经考验的分布式中间件。只是，他还有诸多不方便之处。Twemproxy最大的痛点在于，无法平滑地扩容/缩容。这样增加了运维难度：业务量突增，需增加Redis服务器；业务量萎缩，需要减少Redis服务器。但对Twemproxy而言，基本上都很难操作。或者说，Twemproxy更加像服务器端静态sharding。有时为了规避业务量突增导致的扩容需求，甚至被迫新开一个基于Twemproxy的Redis集群。Twemproxy另一个痛点是，运维不友好，甚至没有控制面板。

### Redis Codis集群方案

* Codis由豌豆荚于2014年11月开源，基于Go和C开发，是近期涌现的、国人开发的优秀开源软件之一。现已广泛用于豌豆荚的各种Redis业务场景.
* 从各种压力测试来看，稳定性符合高效运维的要求。性能更是改善很多，最初比Twemproxy慢20%；现在比Twemproxy快近100%（条件：多实例，一般Value长度）。

#### 组件说明

* **codis-proxy:**客户端连接的Redis代理服务，本身实现了Redis协议，表现很像原生的Redis（就像Twemproxy）。一个业务可以部署多个codis-proxy，其本身是无状态的。
* **codis-config：**Codis的管理工具，支持添加/删除Redis节点、添加/删除Proxy节点、发起数据迁移等操作。codis-config自带了一个http server，会启动一个dashboard，用户可以在浏览器上观察Codis集群的运行状态。
* **codis-server：**Codis项目维护的一个Redis分支，加入了slot的支持和原子的数据迁移指令。
* **ZooKeeper：**Codis依赖ZooKeeper来存放数据路由表和codis-proxy节点的元信息，codis-config发起的命令会通过 ZooKeeper同步到各个存活的codis-proxy



#### 方案优缺点

* Codis具有可视化运维管理界面。Codis无疑是为解决Twemproxy缺点而出的新解决方案。因此综合方面会优于Twemproxy很多。目前也越来越多公司选择Codis。Codis引入了Group的概念，每个Group包括1个Redis Master及至少1个Redis Slave，这是和Twemproxy的区别之一。这样做的好处是，如果当前Master有问题，则运维人员可通过Dashboard“自助式”切换到Slave，而不需要小心翼翼地修改程序配置文件。为支持数据热迁移（Auto Rebalance），出品方修改了Redis Server源码，并称之为Codis Server。Codis采用预先分片（Pre-Sharding）机制，事先规定好了，分成1024个slots（也就是说，最多能支持后端1024个Codis Server），这些路由信息保存在ZooKeeper中。

## 安全

* Redis的安全设计是在“Redis运行在可信环境”这个前提下做出的，在生产环境运行时不能允许外网直接连接到Redis服务器上，而是应该通过应用程序进行中转，运行在可信的环境中是保证Redis安全的最重要的办法。

### 设置访问规则

* Redis的默认设置会接受来自任何地址发来的请求，即使在任何一个拥有公网IP的服务器上启动Redis服务器，都可以被外网其他服务器访问到，要更改这一设置，在配置文件中修改bind参数，如只允许本机连接Redis，可以更改参数：

bind 127.0.0.1

**注意：**Bind参数只能绑定一个地址，如果想要更自由的设置访问规则则需要防火墙来完成

### 设置数据库密码

* 可以设置密码提高安全性，通过配置文件中的requirepass参数为Redis设置一个密码。例如：

Requirepass TAFK(@~JI^XALQ(sYh5xTwTn5d$s7JF

客户端每次连接到Redis都需要发送密码，否则Redis会拒接执行客户端发来的命令 例如：

redis> GET foo

(error) ERR operation not permitted

#发送密码需要使用AUTH命令：

redis> AUTH TAFK(@~JI^XALQ(sYh5xTwTn5d$s7JF

OK

#之后就可以执行任何命令了：

Redis> GET foo

“1”

**注意：**由于Redis性能极高 所以输入错误不会主动延迟，故可以通过穷举破解，故可以把密码设置的更复杂。

配置Redis复制时如果主数据库设置了密码，则需要在从配置文件中通过masterauth参数设置主数据库的密码，使从连接主时自动使用AUTH命令认证。

### 重新命名命令

Redis支持在配置文件中将命令重命名，如果将FLUSHALL 命令重命名weight一个比较复杂的名字，以保证只有自己的应用可以使用该命令。例如：

Rename-command FLUSHALL oyfekmjvmwxq5a9c8usofuo369x0it2k

#如果希望直接禁用某个命令可以将命令重命名weight空白字符：

Rename-command FLUSGALL “”

## 备份redis数据

* 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，这样就保证了无论何时， 复制 RDB 文件都是绝对安全的。

**备份方式：**

* 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。

一些实用又便宜的容灾备份方法：

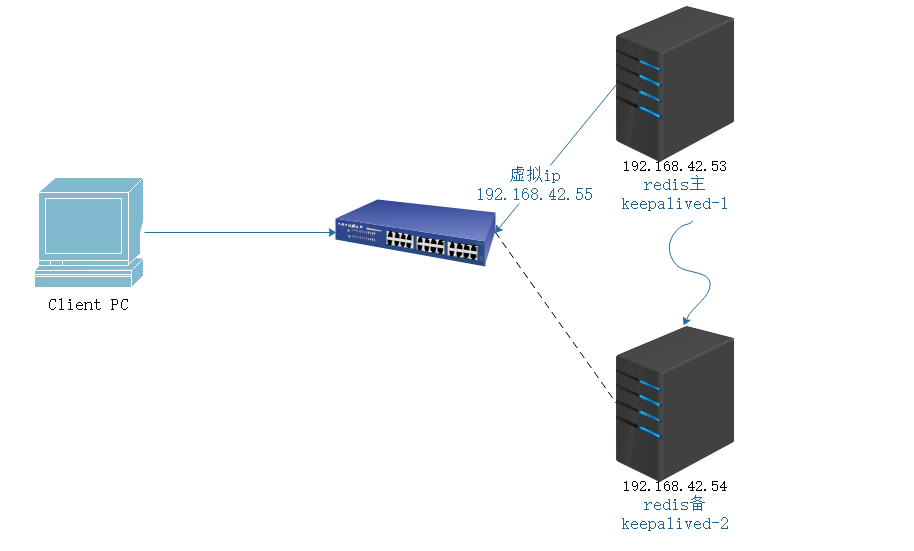
* Amazon S3 ，以及其他类似 S3 的服务，是一个构建灾难备份系统的好地方。 最简单的方法就是将你的每小时或者每日 RDB 备份加密并传送到 S3 。 对数据的加密可以通过 gpg -c 命令来完成（对称加密模式）。 记得把你的密码放到几个不同的、安全的地方去。 同时使用多个储存服务来保存数据文件，可以提升数据的安全性。
* 传送快照可以使用 SCP 来完成（SSH 的组件）。 以下是简单并且安全的传送方法： 买一个离你的数据中心非常远的 VPS ， 装上 SSH ， 创建一个无口令的 SSH 客户端 key ， 并将这个 key 添加到 VPS 的 authorized\_keys 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， 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）失灵时通知报警**

# Redis安装部署规范

## Redis主从配置及keepalived实现Redis高可用

### 架构图：



### 环境说明

**系统环境和软件版本号：**

|  |  |
| --- | --- |
| 项目 | 参数 |
| 系统版本 | Redhat 7.2 64位 |
| 内核版本 | jdk 1.8.0\_111 64位 |
| Redis版本 | redis-4.0.0 |
| Keepalived版本 | Keepalived v1.3.5 |

### 主机环境配置

#### 主机名解析

* 在各主机/etc/hosts中配置主机名解析，两台均执行：

#vim /etc/hosts

127.0.0.1 localhost localhost.localdomain localhost4 localhost4.localdomain4

::1 localhost localhost.localdomain localhost6 localhost6.localdomain6

192.168.42.53 osb53

192.168.42.54 osb54

#### 安全策略

* 两台主机均selinux/iptables设置:

sed -i 's/SELINUX=.\*/SELINUX=disabled/' /etc/selinux/config

setenforce 0

systemctl stop firewalld

systemctl disable firewalld

### 安装redis keepalived

#### yum 安装

* 本地有配置redis源两台主机直接yum安装即可

yum -y install redis keepalived

#### 二进制包安装

* 两台主机下载目前redis最稳定版本

wget http://download.redis.io/releases/redis-4.0.0.tar.gz

* 两台主机下载keepalived二进制安装包

wget http://www.keepalived.org/software/keepalived-1.3.5.tar.gz

#### 编译安装redis

* **编译安装redis：**

#两台主机执行：

tar axf /opt/redis-4.0.0.tar.gz

cd redis-4.0.0

make && make install

* **编译安装keepalived**

##两台主机执行：

tar zxvf keepalived-1.3.5.tar.gz

./configure --prefix=/usr/local/keepalived

make && make install

### 配置redis主从

#### 配置redis

* 复制redis配置文件到/etc目录：

cp /opt/redis-4.0.0/redis.conf /etc/redis

* 修改配置文件使redis-server可以在后台启动

sed -i s/daemonize no/daemonize yes/ /etc/redis.conf

* Init.d 启动脚本：

vim /etc/init.d/redis-server

#!/usr/bin/env bash

#redis start up the redis server daemon

#chkconfig: 345 99 99

#description: redis service in /etc/init.d/redis

#chkconfig --add redis or chkconfig --list redis

# service redis start or service redis stop

#processname: redis-server

#config: /etc/redis.conf

PATH=/usr/local/bin:/sbin:/usr/bin:/bin

REDISPORT=6379

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

REDIS\_CLI=/usr/local/bin/redis-cli

PIDFILE=/var/run/redis.pid

CONF="/etc/redis.conf"

#make sure some dir exist

if [ ! -d /var/lib/redis ] ;then

mkdir -p /var/lib/redis

mkdir -p /var/log/redis

fi

case "$1" in

status)

ps -A|grep redis

;;

start)

if [ -f $PIDFILE ]

then

echo "$PIDFILE exists, process is already running or crashed"

else

echo "Starting Redis server..."

${EXEC} ${CONF}

fi

if [ "$?"="0" ]

then

echo "Redis is running..."

fi

;;

stop)

if [ ! -f $PIDFILE ]

then

echo "$PIDFILE does not exist, process is not running"

else

PID=$(cat $PIDFILE)

echo "Stopping ..."

$REDIS\_CLI -p $REDISPORT SHUTDOWN

while [ -x ${PIDFILE} ]

do

echo "Waiting for Redis to shutdown ..."

sleep 1

done

echo "Redis stopped"

fi

;;

restart|force-reload)

${0} stop

${0} start

;;

\*)

echo "Usage: /etc/init.d/redis {start|stop|restart|force-reload}" >&2

exit 1

esac

* 赋权启动：

chmod 755 /etc/init.d/redis-server

chkconfig --add redis-server

service redis-server start/stop/restart

#### 从服务器配置Redis

* 从服务器配置一样，只不过更改/etx/redis.cof中slaveof修改为：

slaveof 192.168.42.54 6379

* 重启配置生效

/etc/init.d/redis-server restart

#### 测试Redis主从复制

#主服务器

redis-cli -p 6379 set hello word

OK

#从服务器

redis-cli -p 6379 get hello

"word"

#主服务器

redis-cli -p 6379 set hello2 word2

OK

#从服务器

redis-cli -p 6379 get hello2

"word2"

redis-cli -p 6379 set hello world

(error) READONLY You can't write against a read only slave.

#成功配置主从redi服务器，又要配置中有一条从服务器是只读的，所以从服务器没法设置数据，只可以读取数据。

### 配置keepalived高可用

#### 配置keepalived

* 复制相应配置文件

cp /usr/local/keepalived/sbin/keepalived /usr/sbin/

cp /usr/local/keepalived/etc/sysconfig/keepalived /etc/sysconfig/

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

mkdir -p /etc/keepalived

cp /usr/local/keepalived/etc/keepalived/keepalived.conf /etc/keepalived

#### 修改配置文件

* **Master192.168.42.53上配置文件：**

#vim /etc/keepalived/keepalived.conf

! Configuration File for keepalived

global\_defs {

router\_id redis1

}

vrrp\_script chk\_redis {

script "/etc/keepalived/scripts/redis\_check.sh"

interval 2

}

vrrp\_instance Redis {

state BACKUP

interface eth0

virtual\_router\_id 51

priority 100

advert\_int 1

nopreempt

unicast\_src\_ip 192.168.42.53

unicast\_peer {

192.168.42.54

}

authentication {

auth\_type PASS

auth\_pass 112233

}

track\_script {

chk\_redis

}

virtual\_ipaddress {

192.168.42.55

}

notify\_master /etc/keepalived/scripts/redis\_master.sh

notify\_backup /etc/keepalived/scripts/redis\_backup.sh

notify\_fault /etc/keepalived/scripts/redis\_fault.sh

notify\_stop /etc/keepalived/scripts/redis\_stop.sh

}

* **Slave 192.168.42.54上配置文件：**

#vim /etc/keepalived/kepalived.conf

! Configuration File for keepalived

global\_defs {

router\_id redis2

}

vrrp\_script chk\_redis {

script "/etc/keepalived/scripts/redis\_check.sh"

interval 2

}

vrrp\_instance Redis {

state BACKUP

interface eth0

virtual\_router\_id 51

priority 100

advert\_int 1

# nopreempt

unicast\_src\_ip 192.168.42.54

unicast\_peer {

192.168.42.53

}

authentication {

auth\_type PASS

auth\_pass 112233

}

track\_script {

chk\_redis

}

virtual\_ipaddress {

192.168.42.55

}

notify\_master /etc/keepalived/scripts/redis\_master.sh

notify\_backup /etc/keepalived/scripts/redis\_backup.sh

notify\_fault /etc/keepalived/scripts/redis\_fault.sh

notify\_stop /etc/keepalived/scripts/redis\_stop.sh

}

### 创建redis监控脚本

* 创建存放脚本目录：

mkdir -pv /etc/keepalived/scripts

#### 编写状态监控脚本

* 在master/slave创建redis状态监控脚本

vim /etc/keepalived/scripts/redis\_check.sh

#!/bin/bash

ALIVE=`/usr/local/bin/redis-cli PING`

if [ "$ALIVE" == "PONG" ]

then

echo $ALIVE

exit 0

else

echo $ALIVE

exit 1

fi

#### 编写关键脚本

ll /etc/keepalived/scripts/

-rwxr-xr-x 1 root root 891 10月 18 16:07 keepalived-redis-state.log

-rwxr-xr-x 1 755 root 457 10月 17 16:59 redis\_backup.sh

-rwxr-xr-x 1 755 root 182 10月 17 16:41 redis\_fault.sh

-rwxr-xr-x 1 755 root 580 10月 18 09:19 redis\_master.sh

-rwxr-xr-x 1 755 root 162 10月 17 16:46 redis\_stop.sh

* Keepalived在转换状态时会依照状态来执行脚本：
* 当进入master状态时会执行redis\_master.sh
* 当进入backup状态时会执行redis\_backup.sh
* 当发现异常情况时会执行redis\_fault.sh
* 当进入当程序终止时则会执行redis\_stop.sh
* redis\_master.sh脚本

vim redis\_master.sh

#!/bin/bash

#当进入master状态会呼叫notify\_master

redis\_cli="/usr/local/bin/redis-cli"

log\_file="/etc/keepalived/scripts/keepalived-redis-state.log"

echo "[master]" >> $log\_file

date >> $log\_file

echo "Being master...." >>$log\_file 2>&1

echo "Run SLAVEOF cmd....." >>$log\_file

######设置本机53为54的从机

$redis\_cli SLAVEOF 192.168.42.54 6379 >> $log\_file 2>&1q

#延迟10秒以后待数据同步完成后再取消同步状态

sleep 10

echo "Run SLAVEOF NO ONE cmd...." >>$log\_file

######设置为本地为Redis的主机

$redis\_cli SLAVEOF NO ONE >> $log\_file 2>&1

* redis\_backup.sh

vim redis\_backup.sh

#!/bin/bash

#当进入BACKUP状态会呼叫notify\_backup

redis\_cli="/usr/local/bin/redis-cli"

log\_file="/etc/keepalived/scripts/keepalived-redis-state.log"

echo "[backup]" >>$log\_file

date >> $log\_file

echo "Being slave......" >> $log\_file 2>&1

#延迟15秒待数据被对方同步完成之后再切换主从角色

sleep 15

echo "Run SLAVEOF cmd...." >>$log\_file

######设置本机53为54的从机

$redis\_cli SLAVEOF 192.168.42.54 6379 >>log\_file 2>&1

* redis\_fault.sh

vim redis\_fault.sh

#!/bin/bash

#当发现异常情况时进入Fault状态呼叫notify\_fault

log\_file=/etc/keepalived/scripts/keepalived-redis-state.log

echo "[fault]" >>$log\_file

date >> $log\_file

* redis\_stop.sh

vim redis\_stop.sh

#!/bin/bash

#当程序终止时，则呼叫notify\_stop

log\_file=/etc/keepalived/scripts/keepalived-redis-state.log

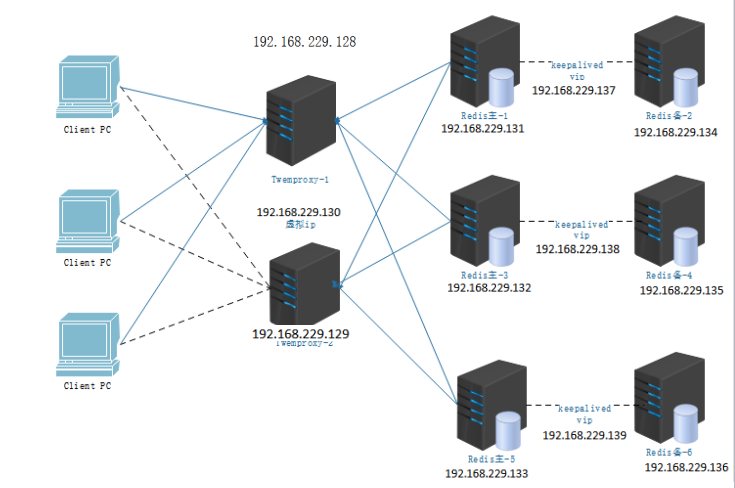
echo "[stop]" >>$log\_file

date >>$log\_file

## Redis+TwemProxy（nutcracker）集群

* Twemproxy 又称nutcracker ，是一个memcache、Redis协议的轻量级代理，一个用于sharding 的中间件。有了Twemproxy，客户端不直接访问Redis服务器，而是通过twemproxy 代理中间件间接访问。 Twemproxy 为 Twitter 开源产品，简单来说，Twemproxy是Twitter开发的一个redis代理proxy，类似于nginx的反向代理或者mysql的代理工具，如amoeba。Twemproxy通过引入一个代理层，可以将其后端的多台Redis或Memcached实例进行统一管理与分配，使应用程序只需要在Twemproxy上进行操作，而不用关心后面具体有多少个真实的Redis或Memcached存储。

### 架构图



### 环境说明

**系统环境和软件版本号：**

|  |  |
| --- | --- |
| 项目 | 参数 |
| 系统版本 | Redhat 7.2 64位 |
| 内核版本 | 3.10.0-327.el7.x86\_64 |
| Redis版本 | redis-4.0.0 |
| Keepalived版本 | Keepalived v1.3.5 |
| Twemproxy | nutcracker-0.4.0 |

使用redis+nutcracker+keepalive实现内存高可用解决方案

一 搭建redis环境. 操作在两台机器上完成相同的工作 (memcached)

### 主机环境配置

#### 主机名解析

* 在各主机/etc/hosts中配置主机名解析，两台均执行：

#vim /etc/hosts

127.0.0.1 localhost localhost.localdomain localhost4 localhost4.localdomain4

::1 localhost localhost.localdomain localhost6 localhost6.localdomain6

192.168.229.128 node1

192.168.229.129 node2

#### 安全策略

* 两台主机均selinux/iptables设置:

sed -i 's/SELINUX=.\*/SELINUX=disabled/' /etc/selinux/config

setenforce 0

systemctl stop firewalld

systemctl disable firewalld

### Redis主从搭建

* 可参考3.1 Redis主从配置及keepalived实现Redis高可用

### nutcracker 安装部署

#### 安装包下载

* 将安装包分别拷贝到192.168.229.128 和 192.168.229.129 两台机器中的/usr/local/soft/ 下

#编译安装

cd /usr/local/soft/

tar -xzvf nutcracker-0.4.0.tar.gz

cd nutcracker-0.4.0

./configure

make && make install

* 复制配置文件：

cd /usr/local/soft/nutcracker-0.4.0/conf

cp nutcracker.yml /etc/nutcracker

* 编辑配置文件：

#vim /etc/nutcracker/nutcracker.yml (配置文件只需要写这些就行)

alpha:

listen: 0.0.0.0:22121

hash: fnv1a\_64

hash\_tag: "{}"

distribution: ketama

auto\_eject\_hosts: true

timeout: 400

redis: true

server\_retry\_timeout: 300000

server\_failure\_limit: 3

servers: -----三台redis服务器的地址和端口

#### 测试nutcracker配置文件是否正常

nutcracker –t -c /etc/nutcracker/nutcracker.yml

* **Linux显示内容:**

nutcracker: configuration file 'conf/nutcracker.yml' syntax is ok

#显示以上内容表示负载成功！

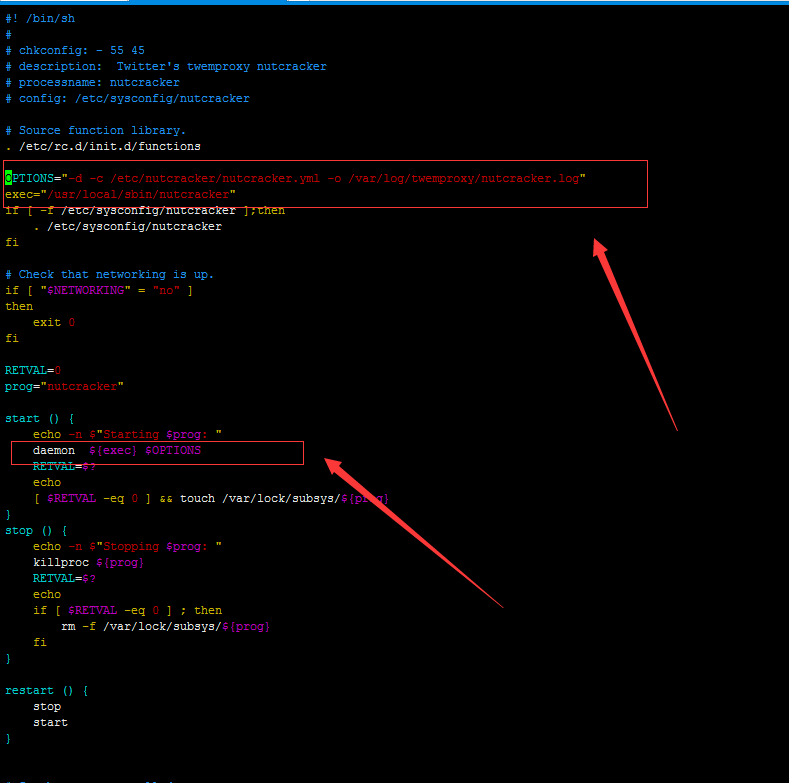
#### 配置chkconfig 以及service

cd /usr/local/soft/nutcracker-0.4.0/scripts

cp nutcracker.init /etc/init.d/nutcracker

chmod 777 /etc/init.d/nutcracker

vim /etc/init.d/nutcracker



修改成上述文件的样子，注意红色框内为改变的地方

第一个红色框：

OPTIONS="-d -c /etc/nutcracker/nutcracker.yml -o /var/log/twemproxy/nutcracker.log"

exec="/usr/local/sbin/nutcracker"

第二个红色框:

daemon ${exec} $OPTIONS

* 设置开机启动项

Chkconfig --add nutcracker

Chkconfig nutcracker on

Chkconfig --list nutcracker

* linux显示内容：

nutcracker 0:off 1:off 2:on  3:on  4:on 5:on 6:off

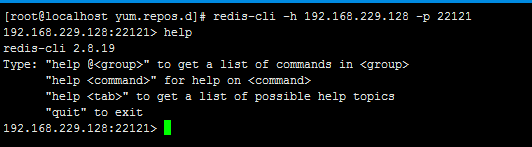
如显示以上内容，则证明nutcracker安装成功

#### 测试nutcracker是否成功为redis代理,

service nutcracker start

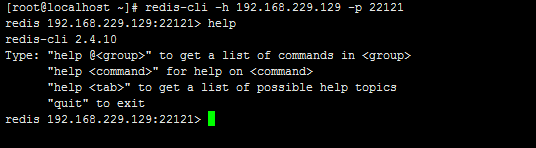
redis-cli  -h 192.168.229.128 -p 22121

#在192.168.229.128上执行的动作



redis-cli  -h 192.168.229.129 -p 22121

#在192.168.229.129上执行的动作



#### keepalived安装

* 以下操作在两台装有nutcracker的机器上执行相同的命令

**Keepalived安装参考安装部署规范**

#### 配置keepalived脚本

编写nutcracker监控脚本

mkdir /etc/keepalived/script

cd /etc/keepalived/script/

vim twemproxy\_check.sh

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

#!/bin/bash

#To install check twemproxy status automatically

#Made by liujun,2014/12/31

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

#Define system environment PATH

export PATH=$PATH

flag=$(pidof nutcracker)

if [ "$flag" == "" ];then

/etc/init.d/nutcracker start

sleep 3

flag\_wait=$(pidof nutcracker)

if [ "$flag\_wait" == "" ];then

/etc/init.d/keepalived stop

fi

fi

* 编写192.268.229.128上的keepalived配置（配置该台keepalive为Master）

vim /etc/keepalived/keepalived.conf

global\_defs {

router\_id LVS\_01

}

vrrp\_script twemproxy\_check {

script "/etc/keepalived/scripts/twemproxy\_check.sh"

interval 2

weight 2

}

vrrp\_instance VI\_1 {

state MASTER

interface eth0 --可能需要修改

virtual\_router\_id 51

priority 200

advert\_int 1

authentication {

auth\_type PASS

auth\_pass 1111

}

virtual\_ipaddress {

10.72.38.112 //vip 虚拟IP

}

track\_script {

twemproxy\_check

}

}

* 编写192.268.229.129上的keepalived配置（配置该台keepalive为slave）

global\_defs {

router\_id LVS\_02

}

vrrp\_script twemproxy\_check {

script "/etc/keepalived/scripts/twemproxy\_check.sh"

interval 2

weight 2

}

vrrp\_instance VI\_1 {

state BACKUP

interface eth0

virtual\_router\_id 51

priority 150

advert\_int 1

authentication {

auth\_type PASS

auth\_pass 1111

}

virtual\_ipaddress {

192.168.229.190

}

track\_script {

twemproxy\_check

}

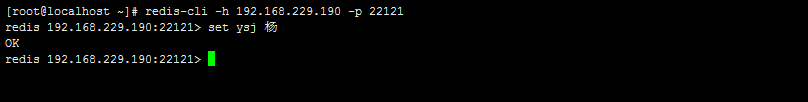
}

#### 启动两台机器上的keepalived

service keepalived start --这条语句分别在两台机器上执行

#### 测试集群环境下的redis是否能正常存储数据

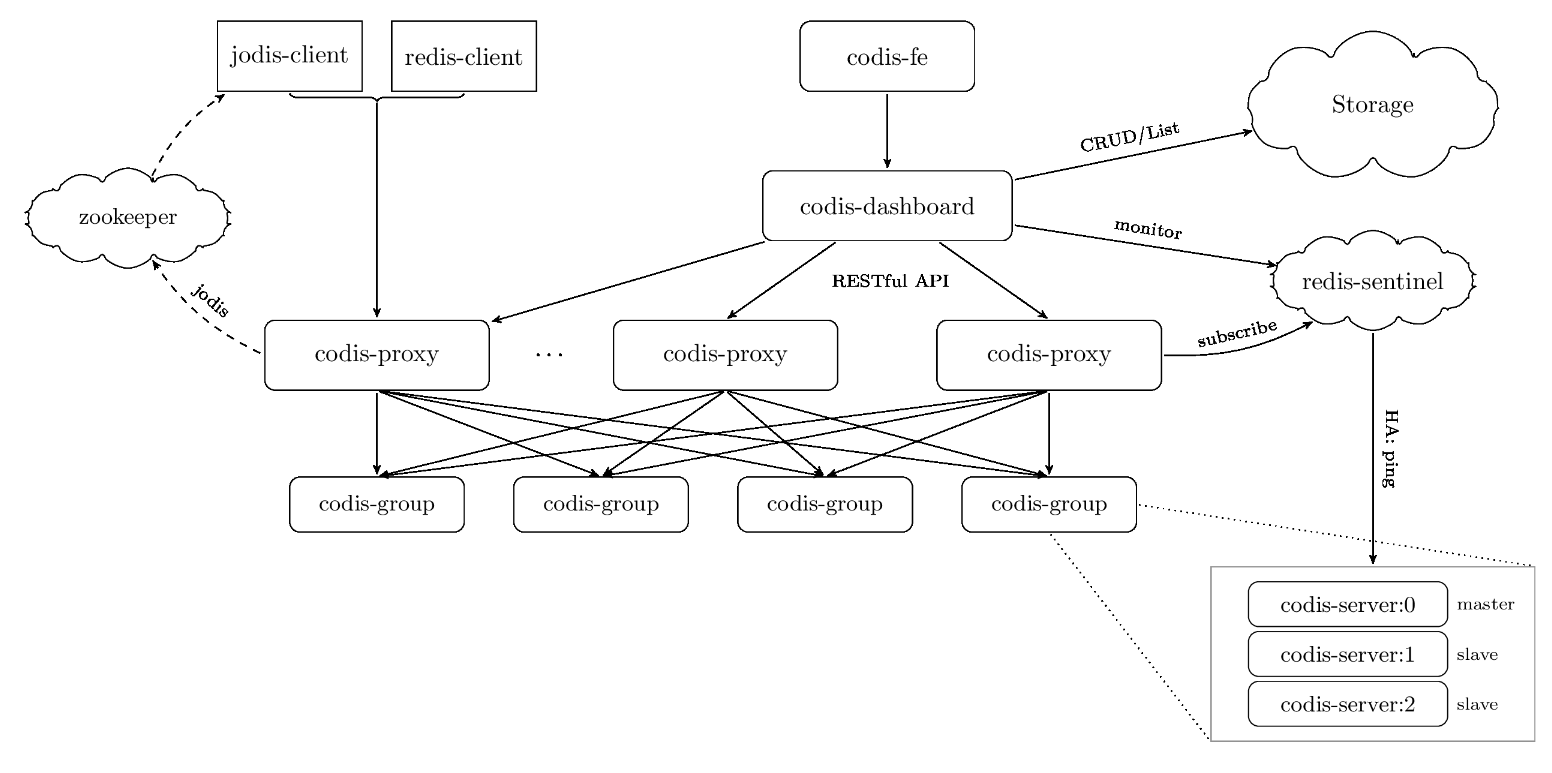
redis-cli  -h 192.168.229.190 -p 22121 --注释192.168.229.190为VIP



**注意：本文中采用的192.168.229.128与192.168.229.129两个IP不做解释，实际安装，可按照不同的机器的IP做相应的更换。本文中的IP只帮助理解实际安装过程**

## Redis Codis集群安装部署

* Codis 是一个分布式 Redis 解决方案, 对于上层的应用来说, 连接到 Codis Proxy 和连接原生的 Redis Server 没有显著区别 (不支持的命令列表), 上层应用可以像使用单机的 Redis 一样使用, Codis 底层会处理请求的转发, 不停机的数据迁移等工作, 所有后边的一切事情, 对于前面的客户端来说是透明的, 可以简单的认为后边连接的是一个内存无限大的 Redis 服务，codis集群是目前生产环境用的最多以及性价比最高的集群。
* **架构图如下：**



### 关于安装部署

**由于安装部署规范较长，故添加至附件** codis3.2安装配置手册（详细）.mht

# Redis操作规范

* Redis提供了丰富的命令（command）对数据和各种数据类型进行操作

## 操作数据库

* 下面描述操作一下数据库的一些基本命令

### 插入数据

redis 127.0.0.1:6379> set name wwl

OK

#设置一个 key-value对

### 查询数据

redis 127.0.0.1:6379> get name

"wwl"

#取出 key 所对应的 value

### 删除键值

redis 127.0.0.1:6379> del name

#删除这个 key 及对应的 value

### 验证键是否存在

redis 127.0.0.1:6379> exists name

(integer) 0

#其中 0，代表此 key 不存在； 1 代表存在

## Redis 数据类型及操作

### strings 类型及操作

* string 是最简单的类型，你可以理解成与 Memcached 是一模一样的类型，一个 key 对应一个value，其上支持的操作与 Memcached 的操作类似。但它的功能更丰富。
* string 类型是二进制安全的。意思是 redis 的 string 可以包含任何数据， 比如 jpg 图片或者序列化的对象。 从内部实现来看其实 string 可以看作 byte 数组，最大上限是 1G 字节。

**下面是string 类型的定义:**

struct sdshdr {

long len;

long free;

char buf[];

};

* len 是 buf 数组的长度。
* free 是数组中剩余可用字节数， 由此可以理解为什么 string 类型是二进制安全的了， 因为它本质上就是个 byte 数组， 当然可以包含任何数据了
* buf 是个 char 数组用于存贮实际的字符串内容， 其实 char 和 c#中的 byte 是等价的，都是一个字节。
* 另外 string 类型可以被部分命令按 int 处理.比如 incr 等命令， 如果只用 string 类型， redis 就可以被看作加上持久化特性的 memcached。当然 redis 对 string 类型的操作比 memcached 还是多很多的。

**具体操作方法如下：**

#### set

* 设置 key 对应的值为 string 类型的 value

**例如我们添加一个 name= HongWan 的键值对：**

redis 127.0.0.1:6379> set name HongWan

OK

redis 127.0.0.1:6379>

#### setnx

* 设置 key 对应的值为 string 类型的 value。 如果 key 已经存在，返回 0， nx 是 not exist 的意思

**例如我们添加一个 name= HongWan\_new 的键值对：**

redis 127.0.0.1:6379> get name

"HongWan"www.ChinaDBA.net 中国 DBA 超级论坛

23

redis 127.0.0.1:6379> setnx name HongWan\_new

(integer) 0

redis 127.0.0.1:6379> get name

"HongWan"

redis 127.0.0.1:6379>

#由于原来 name 有一个对应的值，所以本次的修改不生效，且返回码是 0。

#### setex

* 设置 key 对应的值为 string 类型的 value，并指定此键值对应的有效期

**例如我们添加一个 haircolor= red 的键值对，并指定它的有效期是 10 秒**：

redis 127.0.0.1:6379> setex haircolor 10 red

OK

redis 127.0.0.1:6379> get haircolor

"red"

redis 127.0.0.1:6379> get haircolor

(nil)

redis 127.0.0.1:6379>

#可见由于最后一次的调用是 10 秒以后了，所以取不到 haicolor 这个键对应的值。

#### setrange

* 设置指定 key 的 value 值的子字符串。

**例如我们希望将 HongWan 的 126 邮箱替换为 gmail 邮箱：**

redis 127.0.0.1:6379> get name

"HongWan@126.com"

redis 127.0.0.1:6379> setrange name 8 gmail.com

(integer) 17

redis 127.0.0.1:6379> get name

"HongWan@gmail.com"

redis 127.0.0.1:6379>

#其中的 8 是指从下标为 8（包含 8）的字符开始替换

#### mset

* 一次设置多个 key 的值，成功返回 ok 表示所有的值都设置了，失败返回 0 表示没有任何值被设置

redis 127.0.0.1:6379> mset key1 HongWan1 key2 HongWan2

OK

redis 127.0.0.1:6379> get key1

"HongWan1"

redis 127.0.0.1:6379> get key2www.ChinaDBA.net 中国 DBA 超级论坛

24

"HongWan2"

redis 127.0.0.1:6379>

#### msetnx

* 一次设置多个 key 的值，成功返回 ok 表示所有的值都设置了，失败返回 0 表示没有任何值被设置， 但是不会覆盖已经存在的 key

redis 127.0.0.1:6379> get key1

"HongWan1"

redis 127.0.0.1:6379> get key2

"HongWan2"

redis 127.0.0.1:6379> msetnx key2 HongWan2\_new key3 HongWan3

(integer) 0

redis 127.0.0.1:6379> get key2

"HongWan2"

redis 127.0.0.1:6379> get key3

(nil)

#可以看出如果这条命令返回 0，那么里面操作都会回滚，都不会被执行。

#### get

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

**例如我们获取一个库中存在的键 name，可以很快得到它对应的 value：**

redis 127.0.0.1:6379> get name

"HongWan"

redis 127.0.0.1:6379>

#我们获取一个库中不存在的键 name1，那么它会返回一个 nil 以表时无此键值对

redis 127.0.0.1:6379> get name1

(nil)

redis 127.0.0.1:6379>

#### getset

* 设置 key 的值，并返回 key 的旧值。

redis 127.0.0.1:6379> get name

"HongWan"

redis 127.0.0.1:6379> getset name HongWan\_new

"HongWan"

redis 127.0.0.1:6379> get name

"HongWan\_new"

redis 127.0.0.1:6379>

**接下来我们看一下如果 key 不存的时候的结果：**

redis 127.0.0.1:6379> getset name1 aaa

(nil)

redis 127.0.0.1:6379>

#可见，如果 key 不存在，那么将返回 nil

#### getrange

* 获取指定 key 的 value 值的子字符串。

**具体样例如下:**

redis 127.0.0.1:6379> get name

"HongWan@126.com"

redis 127.0.0.1:6379> getrange name 0 6

"HongWan"

redis 127.0.0.1:6379>

#字符串左面下标是从 0 开始的

redis 127.0.0.1:6379> getrange name -7 -1

"126.com"

redis 127.0.0.1:6379>

#字符串右面下标是从-1 开始的

redis 127.0.0.1:6379> getrange name 7 100

"@126.com"

redis 127.0.0.1:6379>

#当下标超出字符串长度时，将默认为是同方向的最大下标

#### mget

* 一次获取多个 key 的值，如果对应 key 不存在，则对应返回 nil。

**具体样例如下:**

redis 127.0.0.1:6379> mget key1 key2 key3

1) "HongWan1"

2) "HongWan2"

3) (nil)

redis 127.0.0.1:6379>

#key3 由于没有这个键定义，所以返回 nil。

#### incr

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

redis 127.0.0.1:6379> set age 20

OK

redis 127.0.0.1:6379> incr age

(integer) 21

redis 127.0.0.1:6379> get age

"21"

redis 127.0.0.1:6379>

#### incrby

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

redis 127.0.0.1:6379> get age

"21"

redis 127.0.0.1:6379> incrby age 5

(integer) 26

redis 127.0.0.1:6379> get name

"HongWan@gmail.com"

redis 127.0.0.1:6379> get age

"26"

redis 127.0.0.1:6379>

#### decr

* 对 key 的值做的是减减操作， decr 一个不存在 key，则设置 key 为-1

redis 127.0.0.1:6379> get age

"26"

redis 127.0.0.1:6379> decr age

(integer) 25

redis 127.0.0.1:6379> get age

"25"

redis 127.0.0.1:6379>

#### decrby

* 同 decr，减指定值

redis 127.0.0.1:6379> get age

"25"

redis 127.0.0.1:6379> decrby age 5

(integer) 20

redis 127.0.0.1:6379> get age

"20"

redis 127.0.0.1:6379>

#decrby 完全是为了可读性，我们完全可以通过 incrby 一个负值来实现同样效果，反之一样。

redis 127.0.0.1:6379> get age

"20"

redis 127.0.0.1:6379> incrby age -5

(integer) 15

redis 127.0.0.1:6379> get age

"15"

redis 127.0.0.1:6379>

#### append

* 给指定 key 的字符串值追加 value,返回新字符串值的长度

**例如我们向 name 的值追加一个@126.com 字符串**：

redis 127.0.0.1:6379> append name @126.com

(integer) 15

redis 127.0.0.1:6379> get name

"HongWan@126.com"

redis 127.0.0.1:6379>

#### strlen

* 取指定 key 的 value 值的长度

redis 127.0.0.1:6379> get name

"HongWan\_new"

redis 127.0.0.1:6379> strlen name

(integer) 11

redis 127.0.0.1:6379> get age

"15"

redis 127.0.0.1:6379> strlen age

(integer) 2

redis 127.0.0.1:6379>

### hashes 类型及操作

* Redis hash 是一个 string 类型的 field 和 value 的映射表.它的添加、删除操作都是 O(1) （平均）。

hash 特别适合用于存储对象。相较于将对象的每个字段存成单个 string 类型。将一个对象存储在 hash 类型中会占用更少的内存，并且可以更方便的存取整个对象。省内存的原因是新建一个 hash 对象时开始是用 zipmap（又称为 small hash）来存储的。这个 zipmap 其实并不是 hash table，但是 zipmap 相比正常的 hash 实现可以节省不少 hash 本身需要的一些元数据存储开销。尽管 zipmap 的添加，删除，查找都是 O(n)，但是由于一般对象的 field 数量都不

太多。所以使用 zipmap 也是很快的,也就是说添加删除平均还是 O(1)。如果 field 或者 value的大小超出一定限制后， Redis 会在内部自动将 zipmap 替换成正常的 hash 实现.

**这个限制可以在配置文件中指定：**

* hash-max-zipmap-entries 64 #配置字段最多 64 个
* hash-max-zipmap-value 512 #配置 value 最大为 512 字节

#### hset

* 设置 hash field 为指定值，如果 key 不存在，则先创建。

redis 127.0.0.1:6379> hset myhash field1 Hello

(integer) 1

redis 127.0.0.1:6379>

#### hsetnx

* 设置 hash field 为指定值，如果 key 不存在，则先创建。 如果 field 已经存在，返回 0， nx 是not exist 的意思

redis 127.0.0.1:6379> hsetnx myhash field "Hello"

(integer) 1

redis 127.0.0.1:6379> hsetnx myhash field "Hello"

(integer) 0

redis 127.0.0.1:6379>

#第一次执行是成功的，但第二次执行相同的命令失败，原因是 field 已经存在了。

#### hmset

* 同时设置 hash 的多个 field。

redis 127.0.0.1:6379> hmset myhash field1 Hello field2 World

OK

redis 127.0.0.1:6379>

#### hget

* 获取指定的 hash field

redis 127.0.0.1:6379> hget myhash field1

"Hello"

redis 127.0.0.1:6379> hget myhash field2

"World"

redis 127.0.0.1:6379> hget myhash field3

(nil)

redis 127.0.0.1:6379>

#由于数据库没有 field3，所以取到的是一个空值 nil

#### hmget

* 获取全部指定的 hash filed。

redis 127.0.0.1:6379> hmget myhash field1 field2 field3

1) "Hello"

2) "World"

3) (nil)

redis 127.0.0.1:6379>

#由于数据库没有 field3，所以取到的是一个空值 nil

#### hincrby

* 指定的 hash filed 加上给定值

redis 127.0.0.1:6379> hset myhash field3 20

(integer) 1

redis 127.0.0.1:6379> hget myhash field3

"20"

redis 127.0.0.1:6379> hincrby myhash field3 -8

(integer) 12

redis 127.0.0.1:6379> hget myhash field3

"12"

redis 127.0.0.1:6379>

#在本例中我们将 field3 的值从 20 降到了 12，即做了一个减 8 的操作。

#### hexists

* 测试指定 field 是否存在

redis 127.0.0.1:6379> hexists myhash field1

(integer) 1

redis 127.0.0.1:6379> hexists myhash field9

(integer) 0

redis 127.0.0.1:6379>

#通过上例可以说明 field1 存在，但 field9 是不存在的。

#### hlen

* 返回指定 hash 的 field 数量

redis 127.0.0.1:6379> hlen myhash

redis 127.0.0.1:6379>

#通过上例可以看到 myhash 中有 4 个 field。

#### hdel

* 返回指定 hash 的 field 数量

redis 127.0.0.1:6379> hlen myhash

(integer) 4

redis 127.0.0.1:6379> hdel myhash field1

(integer) 1

redis 127.0.0.1:6379> hlen myhash

(integer) 3

redis 127.0.0.1:6379>

#### hkeys

* 返回 hash 的所有 field

redis 127.0.0.1:6379> hkeys myhash

1) "field2"

2) "field"

3) "field3"

redis 127.0.0.1:6379>

#说明这个 hash 中有 3 个 field

#### hvals

* 返回 hash 的所有 value

redis 127.0.0.1:6379> hvals myhash

1) "World"

2) "Hello"

3) "12"

redis 127.0.0.1:6379>

#说明这个 hash 中有 3 个 field

#### hgetall

* 获取某个 hash 中全部的 filed 及 value

redis 127.0.0.1:6379> hgetall myhash

1) "field2"

2) "World"

3) "field"www.ChinaDBA.net 中国 DBA 超级论坛

31

4) "Hello"

5) "field3"

6) "12"

redis 127.0.0.1:6379>

* 可见，一下子将 myhash 中所有的 field 及对应的 value 都取出来了。

### lists 类型及操作

* list 是一个链表结构，主要功能是 push、 pop、获取一个范围的所有值等等， 操作中 key 理解为链表的名字。
* Redis 的 list 类型其实就是一个每个子元素都是 string 类型的双向链表。链表的最大长度是(2的 32 次方)。我们可以通过 push,pop 操作从链表的头部或者尾部添加删除元素。这使得 list既可以用作栈，也可以用作队列。有意思的是 list 的 pop 操作还有阻塞版本的，当我们[lr]pop 一个 list 对象时，如果 list 是空，或者不存在，会立即返回 nil。但是阻塞版本的 b[lr]pop 可以则可以阻塞，当然可以加超时时间，超时后也会返回 nil。为什么要阻塞版本的 pop 呢，主要是为了避免轮询。

#### lpush

* 在 key 对应 list 的头部添加字符串元素

redis 127.0.0.1:6379> lpush mylist "world"

(integer) 1

redis 127.0.0.1:6379> lpush mylist "hello"

(integer) 2

redis 127.0.0.1:6379> lrange mylist 0 -1

1) "hello"

2) "world"

redis 127.0.0.1:6379>

#在此处我们先插入了一个 world，然后在 world 的头部插入了一个 hello。其中 lrange 是用于取 mylist 的内容。

#### rpush

* 在 key 对应 list 的尾部添加字符串元素

redis 127.0.0.1:6379> rpush mylist2 "hello"

(integer) 1

redis 127.0.0.1:6379> rpush mylist2 "world"

(integer) 2

redis 127.0.0.1:6379> lrange mylist2 0 -1

1) "hello"

2) "world"

redis 127.0.0.1:6379>

#在此处我们先插入了一个 hello，然后在 hello 的尾部插入了一个 world。

#### linsert

* 在 key 对应 list 的特定位置之前或之后添加字符串元素

redis 127.0.0.1:6379> rpush mylist3 "hello"

(integer) 1

redis 127.0.0.1:6379> rpush mylist3 "world"

(integer) 2

redis 127.0.0.1:6379> linsert mylist3 before "world" "there"

(integer) 3

redis 127.0.0.1:6379> lrange mylist3 0 -1

1) "hello"

2) "there"

3) "world"

redis 127.0.0.1:6379>

#在此处我们先插入了一个 hello，然后在 hello 的尾部插入了一个 world，然后又在 world 的前面插入了 there。

#### lset

* 设置 list 中指定下标的元素值(下标从 0 开始)

redis 127.0.0.1:6379> rpush mylist4 "one"

(integer) 1

redis 127.0.0.1:6379> rpush mylist4 "two"

(integer) 2

redis 127.0.0.1:6379> rpush mylist4 "three"

(integer) 3

redis 127.0.0.1:6379> lset mylist4 0 "four"

OK

redis 127.0.0.1:6379> lset mylist4 -2 "five"

OK

redis 127.0.0.1:6379> lrange mylist4 0 -1

1) "four"

2) "five"

3) "three"

redis 127.0.0.1:6379>

#在此处我们依次插入了 one,two,three，然后将标是 0 的值设置为 four，再将下标是-2 的值设置为 five。

#### lrem

* 从 key 对应 list 中删除 count 个和 value 相同的元素。

count>0 时，按从头到尾的顺序删除，

**具体如下:**

redis 127.0.0.1:6379> rpush mylist5 "hello"

(integer) 1

redis 127.0.0.1:6379> rpush mylist5 "hello"

(integer) 2

redis 127.0.0.1:6379> rpush mylist5 "foo"

(integer) 3

redis 127.0.0.1:6379> rpush mylist5 "hello"

(integer) 4

redis 127.0.0.1:6379> lrem mylist5 2 "hello"

(integer) 2

redis 127.0.0.1:6379> lrange mylist5 0 -1

1) "foo"

2) "hello"

redis 127.0.0.1:6379>

* count<0 时，按从尾到头的顺序删除

**具体如下:**

redis 127.0.0.1:6379> rpush mylist6 "hello"

(integer) 1

redis 127.0.0.1:6379> rpush mylist6 "hello"

(integer) 2

redis 127.0.0.1:6379> rpush mylist6 "foo"

(integer) 3

redis 127.0.0.1:6379> rpush mylist6 "hello"

(integer) 4

redis 127.0.0.1:6379> lrem mylist6 -2 "hello"

(integer) 2

redis 127.0.0.1:6379> lrange mylist6 0 -1

1) "hello"

2) "foo"

redis 127.0.0.1:6379>

* count=0 时，删除全部

**具体如下:**

redis 127.0.0.1:6379> rpush mylist7 "hello"

(integer) 1

redis 127.0.0.1:6379> rpush mylist7 "hello"

(integer) 2

redis 127.0.0.1:6379> rpush mylist7 "foo"

(integer) 3

redis 127.0.0.1:6379> rpush mylist7 "hello"

(integer) 4

redis 127.0.0.1:6379> lrem mylist7 0 "hello"

(integer) 3

redis 127.0.0.1:6379> lrange mylist7 0 -1

1) "foo"

redis 127.0.0.1:6379>

#### ltrim

* 保留指定 key 的值范围内的数据

redis 127.0.0.1:6379> rpush mylist8 "one"

(integer) 1

redis 127.0.0.1:6379> rpush mylist8 "two"

(integer) 2

redis 127.0.0.1:6379> rpush mylist8 "three"

(integer) 3

redis 127.0.0.1:6379> rpush mylist8 "four"

(integer) 4

redis 127.0.0.1:6379> ltrim mylist8 1 -1

OK

redis 127.0.0.1:6379> lrange mylist8 0 -1

1) "two"

2) "three"

3) "four"

redis 127.0.0.1:6379>

#### lpop

* 从 list 的头部删除元素，并返回删除元素

redis 127.0.0.1:6379> lrange mylist 0 -1

1) "hello"

2) "world"

redis 127.0.0.1:6379> lpop mylist

"hello"

redis 127.0.0.1:6379> lrange mylist 0 -1

1) "world"

redis 127.0.0.1:6379>

#### rpop

* 从 list 的尾部删除元素，并返回删除元素

redis 127.0.0.1:6379> lrange mylist2 0 -1

1) "hello"

2) "world"

redis 127.0.0.1:6379> rpop mylist2

"world"

redis 127.0.0.1:6379> lrange mylist2 0 -1

1) "hello"

redis 127.0.0.1:6379>

#### rpoplpush

* 从第一个 list 的尾部移除元素并添加到第二个 list 的头部,最后返回被移除的元素值，整个操作是原子的.如果第一个 list 是空或者不存在返回 nil

redis 127.0.0.1:6379> lrange mylist5 0 -1

1) "three"

2) "foo"

3) "hello"

redis 127.0.0.1:6379> lrange mylist6 0 -1

1) "hello"

2) "foo"

redis 127.0.0.1:6379> rpoplpush mylist5 mylist6

"hello"

redis 127.0.0.1:6379> lrange mylist5 0 -1

1) "three"

2) "foo"

redis 127.0.0.1:6379> lrange mylist6 0 -1

1) "hello"

2) "hello"

3) "foo"

redis 127.0.0.1:6379>

#### lindex

* 返回名称为 key 的 list 中 index 位置的元素

redis 127.0.0.1:6379> lrange mylist5 0 -1

1) "three"

2) "foo"

redis 127.0.0.1:6379> lindex mylist5 0

"three"

redis 127.0.0.1:6379> lindex mylist5 1

"foo"

redis 127.0.0.1:6379>

#### llen

* 返回 key 对应 list 的长度

redis 127.0.0.1:6379> llen mylist5

(integer) 2

redis 127.0.0.1:6379>

### sets 类型及操作

* set 是集合，和我们数学中的集合概念相似，对集合的操作有添加删除元素，有对多个集合求交并差等操作， 操作中 key 理解为集合的名字。
* set 的是通过 hash table 实现的，所以添加、删除和查找的复杂度都是 O(1)。 hash table 会随着添加或者删除自动的调整大小。需要注意的是调整 hash table 大小时候需要同步（获取写锁）会阻塞其他读写操作，可能 不久后就会改用跳表（skip list）来实现，跳表已经在 sorted set 中使用了。关于 set 集合类型除了基本的添加删除操作，其他有用的操作还包含集合的取并集(union)，交集(intersection)，差集(difference)。通过这些操作可以很容易的实现 sns中的好友推荐和 blog 的 tag 功能。

**下面详细介绍 set 相关命令:**

#### sadd

* 向名称为 key 的 set 中添加元素

redis 127.0.0.1:6379> sadd myset "hello"

(integer) 1

redis 127.0.0.1:6379> sadd myset "world"

(integer) 1

redis 127.0.0.1:6379> sadd myset "world"

(integer) 0

redis 127.0.0.1:6379> smembers myset

1) "world"

2) "hello"

redis 127.0.0.1:6379>

#本例中，我们向 myset 中添加了三个元素，但由于第三个元素跟第二个元素是相同的，所以第三个元素没有添加成功，最后我们用 smembers 来查看 myset 中的所有元素。

#### srem

* 删除名称为 key 的 set 中的元素 member

redis 127.0.0.1:6379> sadd myset2 "one"

(integer) 1

redis 127.0.0.1:6379> sadd myset2 "two"

(integer) 1

redis 127.0.0.1:6379> sadd myset2 "three"

(integer) 1

redis 127.0.0.1:6379> srem myset2 "one"

(integer) 1

redis 127.0.0.1:6379> srem myset2 "four"

(integer) 0

redis 127.0.0.1:6379> smembers myset2

1) "three"

2) "two"

redis 127.0.0.1:6379>

#本例中，我们向 myset2 中添加了三个元素后，再调用 srem 来删除 one 和 four，但由于元素中没有 four 所以，此条 srem 命令执行失败。

#### spop

* 随机返回并删除名称为 key 的 set 中一个元素

redis 127.0.0.1:6379> sadd myset3 "one"

(integer) 1

redis 127.0.0.1:6379> sadd myset3 "two"

(integer) 1

redis 127.0.0.1:6379> sadd myset3 "three"

(integer) 1

redis 127.0.0.1:6379> spop myset3

"three"

redis 127.0.0.1:6379> smembers myset3

1) "two"

2) "one"

redis 127.0.0.1:6379>

#本例中，我们向 myset3 中添加了三个元素后，再调用 spop 来随机删除一个元素，可以看到three 元素被删除了。

#### sdiff

* 返回所有给定 key 与第一个 key 的差集

redis 127.0.0.1:6379> smembers myset2

1) "three"

2) "two"

redis 127.0.0.1:6379> smembers myset3

1) "two"

2) "one"

redis 127.0.0.1:6379> sdiff myset2 myset3

1) "three"

redis 127.0.0.1:6379>

#本例中，我们可以看到 myset2 中的元素与 myset3 中不同的只是 three，所以只有 three 被查出来了，而不是 three 和 one，因为 one 是 myset3 的元素。

**我们也可以将 myset2 和 myset3 换个顺序来看一下结果:**

redis 127.0.0.1:6379> sdiff myset3 myset2

1) "one"

redis 127.0.0.1:6379>

#这个结果中只显示了， myset3 中的元素与 myset2 中不同的元素。

#### sdiffstore

* 返回所有给定 key 与第一个 key 的差集，并将结果存为另一个 key

redis 127.0.0.1:6379> smembers myset2

1) "three"

2) "two"

redis 127.0.0.1:6379> smembers myset3

1) "two"

2) "one"

redis 127.0.0.1:6379> sdiffstore myset4 myset2 myset3

(integer) 1

redis 127.0.0.1:6379> smembers myset4

1) "three"

redis 127.0.0.1:6379>

#### sinter

* 返回所有给定 key 的交集

redis 127.0.0.1:6379> smembers myset2

1) "three"

2) "two"

redis 127.0.0.1:6379> smembers myset3

1) "two"

2) "one"

redis 127.0.0.1:6379> sinter myset2 myset3

1) "two"

redis 127.0.0.1:6379>

#通过本例的结果可以看出, myset2 和 myset3 的交集 two 被查出来了

#### sinterstore

* 返回所有给定 key 的交集，并将结果存为另一个 key

redis 127.0.0.1:6379> smembers myset2

1) "three"

2) "two"

redis 127.0.0.1:6379> smembers myset3

1) "two"

2) "one"

redis 127.0.0.1:6379> sinterstore myset5 myset2 myset3

(integer) 1

redis 127.0.0.1:6379> smembers myset5

1) "two"

redis 127.0.0.1:6379>

#通过本例的结果可以看出, myset2 和 myset3 的交集被保存到 myset5 中了

#### sunion

* 返回所有给定 key 的并集

redis 127.0.0.1:6379> smembers myset2

1) "three"

2) "two"

redis 127.0.0.1:6379> smembers myset3

1) "two"

2) "one"

redis 127.0.0.1:6379> sunion myset2 myset3

1) "three"

2) "one"

3) "two"

redis 127.0.0.1:6379>

#通过本例的结果可以看出, myset2 和 myset3 的并集被查出来了

#### sunionstore

* 返回所有给定 key 的并集，并将结果存为另一个 key

redis 127.0.0.1:6379> smembers myset2

1) "three"

2) "two"

redis 127.0.0.1:6379> smembers myset3

1) "two"

2) "one"

redis 127.0.0.1:6379> sunionstore myset6 myset2 myset3

(integer) 3

redis 127.0.0.1:6379> smembers myset6

1) "three"

2) "one"

3) "two"

redis 127.0.0.1:6379>

#通过本例的结果可以看出, myset2 和 myset3 的并集被保存到 myset6 中了

#### smove

* 从第一个 key 对应的 set 中移除 member 并添加到第二个对应 set 中

redis 127.0.0.1:6379> smembers myset2

1) "three"

2) "two"

redis 127.0.0.1:6379> smembers myset3

1) "two"

2) "one"

redis 127.0.0.1:6379> smove myset2 myset7 three

(integer) 1

redis 127.0.0.1:6379> smembers myset7

1) "three"

redis 127.0.0.1:6379>

#通过本例可以看到， myset2 的 three 被移到 myset7 中了

#### scard

* 返回名称为 key 的 set 的元素个数

redis 127.0.0.1:6379> scard myset2

(integer) 1

redis 127.0.0.1:6379>

#通过本例可以看到， myset2 的成员数量为 1

#### sismember

* 测试 member 是否是名称为 key 的 set 的元素

redis 127.0.0.1:6379> smembers myset2

1) "two"

redis 127.0.0.1:6379> sismember myset2 two

(integer) 1

redis 127.0.0.1:6379> sismember myset2 one

(integer) 0

redis 127.0.0.1:6379>

#通过本例可以看到， two 是 myset2 的成员，而 one 不是。

#### srandmember

* 随机返回名称为 key 的 set 的一个元素，但是不删除元素

redis 127.0.0.1:6379> smembers myset3

1) "two"

2) "one"

redis 127.0.0.1:6379> srandmember myset3

"two"

redis 127.0.0.1:6379> srandmember myset3

"one"

redis 127.0.0.1:6379>

### sorted sets 类型及操作

* sorted set 是 set 的一个升级版本， 它在 set 的基础上增加了一个顺序属性，这一属性在添加修改元素的时候可以指定，每次指定后， zset 会自动重新按新的值调整顺序。可以理解为有两列的 mysql 表，一列存 value，一列存顺序。操作中 key 理解为 zset 的名字。
* 和 set 一样 sorted set 也是 string 类型元素的集合，不同的是每个元素都会关联一个 double类型的 score。 sorted set 的实现是 skip list 和 hash table 的混合体。
* 当元素被添加到集合中时，一个元素到 score 的映射被添加到 hash table 中，所以给定一个元素获取 score 的开销是 O(1),另一个 score 到元素的映射被添加到 skip list，并按照 score 排序，所以就可以有序的获取集合中的元素。添加，删除操作开销都是 O(log(N))和 skip list 的开销一致,redis 的 skip list 实现用的是双向链表,这样就可以逆序从尾部取元素。 sorted set 最经常的使用方式应该是作为索引来使用.我们可以把要排序的字段作为 score 存储，对象的 id当元素存储。

**下面是 sorted set 相关命令：**

#### zadd

* 向名称为 key 的 zset 中添加元素 member， score 用于排序。如果该元素已经存在，则根据score 更新该元素的顺序

redis 127.0.0.1:6379> zadd myzset 1 "one"

(integer) 1

redis 127.0.0.1:6379> zadd myzset 2 "two"

(integer) 1

redis 127.0.0.1:6379> zadd myzset 3 "two"

(integer) 0

redis 127.0.0.1:6379> zrange myzset 0 -1 withscores

1) "one"

2) "1"

3) "two"

4) "3"

redis 127.0.0.1:6379>

#本例中我们向 myzset 中添加了 one 和 two，并且 two 被设置了 2 次，那么将以最后一次的设置为准，最后我们将所有元素都显示出来并显示出了元素的 score。

#### zrem

* 删除名称为 key 的 zset 中的元素 member

redis 127.0.0.1:6379> zrange myzset 0 -1 withscores

1) "one"

2) "1"

3) "two"

4) "3"

redis 127.0.0.1:6379> zrem myzset two

(integer) 1

redis 127.0.0.1:6379> zrange myzset 0 -1 withscores

1) "one"

2) "1"

redis 127.0.0.1:6379>

#可以看到 two 被删除了

#### zincrby

* 如果在名称为 key 的 zset 中已经存在元素 member，则该元素的 score 增加 increment；否则向集合中添加该元素，其 score 的值为 increment

redis 127.0.0.1:6379> zadd myzset2 1 "one"

(integer) 1

redis 127.0.0.1:6379> zadd myzset2 2 "two"

(integer) 1

redis 127.0.0.1:6379> zincrby myzset2 2 "one"

"3"

redis 127.0.0.1:6379> zrange myzset2 0 -1 withscores

1) "two"

2) "2"

3) "one"

4) "3"

redis 127.0.0.1:6379>

#本例中将 one 的 score 从 1 增加了 2，增加到了 3

#### zrank

* 返回名称为 key 的 zset 中 member 元素的排名(按 score 从小到大排序)即下标

redis 127.0.0.1:6379> zrange myzset3 0 -1 withscores

1) "one"

2) "1"

3) "two"

4) "2"

5) "three"

6) "3"

7) "five"

8) "5"

redis 127.0.0.1:6379> zrank myzset3 two

(integer) 1

redis 127.0.0.1:6379>

#本例中将 two 的下标是 1，我这里取的是下标，而不是 score

#### zrevrank

* 返回名称为 key 的 zset 中 member 元素的排名(按 score 从大到小排序)即下标

redis 127.0.0.1:6379> zrange myzset3 0 -1 withscores

1) "one"

2) "1"

3) "two"

4) "2"

5) "three"

6) "3"

7) "five"

8) "5"

redis 127.0.0.1:6379> zrevrank myzset3 two

(integer) 2

redis 127.0.0.1:6379>

#按从大到小排序的话 two 是第三个元素，下标是 2

#### zrevrange

* 返回名称为 key 的 zset（按 score 从大到小排序）中的 index 从 start 到 end 的所有元素

redis 127.0.0.1:6379> zrevrange myzset3 0 -1 withscores

1) "five"

2) "5"

3) "three"

4) "3"

5) "two"

6) "2"

7) "one"

8) "1"

redis 127.0.0.1:6379>

#首先按 score 从大到小排序，再取出全部元素

#### zrangebyscore

* 返回集合中 score 在给定区间的元素

redis 127.0.0.1:6379> zrange myzset3 0 -1 withscores

1) "one"

2) "1"

3) "two"

4) "2"

5) "three"

6) "3"

7) "five"

8) "5"

redis 127.0.0.1:6379> zrangebyscore myzset3 2 3 withscores

1) "two"

2) "2"

3) "three"

4) "3"

redis 127.0.0.1:6379>

#本例中，返回了 score 在 2~3 区间的元素

#### zcount

* 返回集合中 score 在给定区间的数量

redis 127.0.0.1:6379> zrange myzset3 0 -1 withscores

1) "one"

2) "1"

3) "two"

4) "2"

5) "three"

6) "3"

7) "five"

8) "5"

redis 127.0.0.1:6379> zcount myzset3 2 3

(integer) 2

redis 127.0.0.1:6379>

#本例中，计算了 score 在 2~3 之间的元素数目

#### zcard

* 返回集合中元素个数

redis 127.0.0.1:6379> zrange myzset3 0 -1 withscores

1) "one"

2) "1"

3) "two"

4) "2"

5) "three"

6) "3"

7) "five"

8) "5"

redis 127.0.0.1:6379> zcard myzset3

(integer) 4

redis 127.0.0.1:6379>

#从本例看出 myzset3 这个集全的元素数量是 4

#### zscore

* 返回给定元素对应的 score

redis 127.0.0.1:6379> zrange myzset3 0 -1 withscores

1) "one"

2) "1"

3) "two"

4) "2"

5) "three"

6) "3"

7) "five"

8) "5"

redis 127.0.0.1:6379> zscore myzset3 two

"2"

redis 127.0.0.1:6379>

#此例中我们成功的将 two 的 score 取出来了。

#### zremrangebyrank

* 删除集合中排名在给定区间的元素

redis 127.0.0.1:6379> zrange myzset3 0 -1 withscores

1) "one"

2) "1"

3) "two"

4) "2"

5) "three"

6) "3"

7) "five"

8) "5"

redis 127.0.0.1:6379> zremrangebyrank myzset3 3 3

(integer) 1

redis 127.0.0.1:6379> zrange myzset3 0 -1 withscores

1) "one"

2) "1"

3) "two"

4) "2"

5) "three"

6) "3"

redis 127.0.0.1:6379>

#在本例中我们将 myzset3 中按从小到大排序结果的下标为 3 的元素删除了。

#### zremrangebyscore

* 删除集合中 score 在给定区间的元素

redis 127.0.0.1:6379> zrange myzset3 0 -1 withscores

1) "one"

2) "1"

3) "two"

4) "2"

5) "three"

6) "3"

redis 127.0.0.1:6379> zremrangebyscore myzset3 1 2

(integer) 2

redis 127.0.0.1:6379> zrange myzset3 0 -1 withscores

1) "three"

2) "3"

redis 127.0.0.1:6379>

#在本例中我们将 myzset3 中按从小到大排序结果的 score 在 1~2 之间的元素删除了。

## Redis 键值相关命令

### Keys

* 返回满足给定pattern的所有key

[root@redis1 ~]# redis-cli -p 192.168.42.53 -p 6379

127.0.0.1:6379> keys \*

1) "hello2"

2) "a"

3) "hell06"

4) "hello3"

5) "hello"

6) "hell03"

7) "testms"

8) "a9"

9) "name"

10) "a1"

#用表达式\*，代表所有的key

127.0.0.1:6379> keys hello\*

1) "hello2"

2) "hello3"

3) "hello"

#表达式hello\*，代表取出所有以hello开头的key

### exists

* 确认一个key是否存在

127.0.0.1:6379> exists TEST

(integer) 0

127.0.0.1:6379> exists hello

(integer) 1

#从结果看数据库中不存在TEST这个key，但是hello这个key是存在的

### del

* 删除一个key

127.0.0.1:6379> del hello

(integer) 1

127.0.0.1:6379> del hello

(integer) 0

#从结果看数据库已经删除hello这个key，再次删除提示删除为0

### expire

* 设置一个key的过期时间（单位：秒）

redis 127.0.0.1:6379> expire addr 10

(integer) 1

redis 127.0.0.1:6379> ttl addr

(integer) 8

redis 127.0.0.1:6379> ttl addr

(integer) 1

redis 127.0.0.1:6379> ttl addr

(integer) -1

redis 127.0.0.1:6379

#在本例中，我们设置addr这个key的过期时间是10秒，然后我们不断的用ttl来获取这个key的有效时长，直至-1说明此值已过期

### move

* 将当前数据库的key转移到其他数据库中

127.0.0.1:6379> select 0

OK

127.0.0.1:6379> set age 30

OK

127.0.0.1:6379> get age

"30"

127.0.0.1:6379> move age 1

(integer) 1

127.0.0.1:6379> get age

(nil)

127.0.0.1:6379> select 1

OK

127.0.0.1:6379[1]> get age

"30"

#在本例中，我们先显示的选择了数据库0，然后在这个库中设置一个key，接下来我们将这个key从数据库0转移到数据库1，之后我们确认数据库0中无此key了，但在数据库1中存在这个key，说明我们转移成功了

### persist

* 移除给定key的过期时间

redis 127.0.0.1:6379[1]> expire age 300

(integer) 1

redis 127.0.0.1:6379[1]> ttl age

(integer) 294

redis 127.0.0.1:6379[1]> persist age

(integer) 1

redis 127.0.0.1:6379[1]> ttl age

(integer) -1

redis 127.0.0.1:6379[1]>

#在这个例子中，我们手动的将未到过期时间的key，成功设置为过期

### randomkey

随机返回key空间的一个key

127.0.0.1:6379> randomkey

"a9"

127.0.0.1:6379> randomkey

"hello3"

#通过结果可以看出获取的key的规则是随机的

### rename

* 重命名key

127.0.0.1:6379> set age 30

OK

127.0.0.1:6379> keys \*

1) "age"

127.0.0.1:6379> RENAME age age\_new

OK

127.0.0.1:6379> keys \*

1) "age\_new"

#age成功的被改名为age\_new

### 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

redis 127.0.0.1:6379>

#这个方法可以非常简单的判断出值的类型

## 服务器相关命令

### Ping

* 测试连接是否存活

redis 127.0.0.1:6379> ping

PONG

//执行下面命令之前，我们停止 redis 服务器

redis 127.0.0.1:6379> ping

Could not connect to Redis at 127.0.0.1:6379: Connection refused

//执行下面命令之前，我们启动 redis 服务器

not connected> ping

PONG

redis 127.0.0.1:6379>

* 第一个 ping 时，说明此连接正常
* 第二个 ping 之前，我们将 redis 服务器停止，那么 ping 是失败的
* 第三个 ping 之前，我们将 redis 服务器启动，那么 ping 是成功的

### echo

* 在命令行打印一些内容

127.0.0.1:6379> ECHO nihao

"nihao"

### Select

* 选择数据库，Redis数据库编号默认从0~15，我们可以选择任意一个数据库来进行数据的获取

127.0.0.1:6379[1]> SELECT 1

OK

127.0.0.1:6379[1]> SELECT 16

(error) ERR invalid DB index

#当选择16时，保存，说明没有编号为16的这个数据库

### quit

* 退出连接

127.0.0.1:6379> quit

[root@redis1 ~]#

### Dbsize

返回当前数据库中key的数目

127.0.0.1:6379> keys \*

1) "hello"

2) "age\_new"

127.0.0.1:6379> DBSIZE

(integer) 2

#结果说明此库中有2个key

### info

* 获取服务器的信息和统计

redis 127.0.0.1:6379> info

redis\_version:2.2.12

redis\_git\_sha1:00000000

redis\_git\_dirty:0

arch\_bits:32

multiplexing\_api:epoll

process\_id:28480

uptime\_in\_seconds:2515

uptime\_in\_days:0

.

.

.

#此结果用于说明服务器的基础信息，包括版本、启动时间等。

### monitor

* 调试命令

redis 127.0.0.1:6379> monitor

1339518083.107412 [0 127.0.0.1:60866] "keys" "\*"

1339518087.877697 [0 127.0.0.1:60866] "dbsize"

1339518090.420270 [0 127.0.0.1:60866] "set" "x" "6"

1339518096.506257 [0 127.0.0.1:60866] "get" "x"

1339518099.363765 [0 127.0.0.1:60866] "del" "x"

1339518100.544926 [0 127.0.0.1:60866] "get" "x"

#它能帮助我们了解在数据库上发生了什么操作

### config get

* 获取服务器配置信息

redis 127.0.0.1:6379> config get dir

1) "dir"

2) "/root/4setup/redis-2.2.12"

redis 127.0.0.1:6379>

#本例中我们获取了 dir 这个参数配置的值，如果想获取全部参数据的配置值也很简单，只需

执行” config get \*”即可将全部的值都显示出来。

### flushdb

* 删除当前选择数据库中的所有 key、

redis 127.0.0.1:6379> dbsize

(integer) 18

redis 127.0.0.1:6379> flushdb

OK

redis 127.0.0.1:6379> dbsize

(integer) 0

redis 127.0.0.1:6379>

#在本例中我们将 0 号数据库中的 key 都清除了。53

### flushall

* 删除所有数据库中的所有 key

redis 127.0.0.1:6379[1]> dbsize

(integer) 1

redis 127.0.0.1:6379[1]> select 0

OK

redis 127.0.0.1:6379> flushall

OK

redis 127.0.0.1:6379> select 1

OK

redis 127.0.0.1:6379[1]> dbsize

(integer) 0

redis 127.0.0.1:6379[1]>

#在本例中我们先查看了一个 1 号数据库中有一个 key，然后我切换到 0 号库执行 flushall 命令，

结果 1 号库中的 key 也被清除了，说是此命令工作正常

# Redis开发规范

## **冷热数据分离，不要将所有数据全部都放到Redis中**

* 虽然Redis支持持久化，但是Redis的数据存储全部都是在内存中的，成本昂贵。建议根据业务只将高频热数据存储到Redis中【QPS大于5000】，对于低频冷数据可以使用MySQL/ElasticSearch/MongoDB等基于磁盘的存储方式，不仅节省内存成本，而且数据量小在操作时速度更快、效率更高！

## **不同的业务数据要分开存储**

* 不要将不相关的业务数据都放到一个Redis实例中，建议新业务申请新的单独实例。因为Redis为单线程处理，独立存储会减少不同业务相互操作的影响，提高请求响应速度；同时也避免单个实例内存数据量膨胀过大，在出现异常情况时可以更快恢复服务！

## **存储的Key一定要设置超时时间**

* 如果应用将Redis定位为缓存Cache使用，对于存放的Key一定要设置超时时间！因为若不设置，这些Key会一直占用内存不释放，造成极大的浪费，而且随着时间的推移会导致内存占用越来越大，直到达到服务器内存上限！另外Key的超时长短要根据业务综合评估，而不是越长越好！

## **对于必须要存储的大文本数据一定要压缩后存储**

* 对于大文本【超过500字节】写入到Redis时，一定要压缩后存储！大文本数据存入Redis，除了带来极大的内存占用外，在访问量高时，很容易就会将网卡流量占满，进而造成整个服务器上的所有服务不可用，并引发雪崩效应，造成各个系统瘫痪！

## **线上Redis禁止使用Keys正则匹配操作**

* Redis是单线程处理，在线上KEY数量较多时，操作效率极低【时间复杂度为O(N)】，该命令一旦执行会严重阻塞线上其它命令的正常请求，而且在高QPS情况下会直接造成Redis服务崩溃！如果有类似需求，请使用scan命令代替！

## **可靠的消息队列服务**

* Redis List经常被用于消息队列服务。假设消费者程序在从队列中取出消息后立刻崩溃，但由于该消息已经被取出且没有被正常处理，那么可以认为该消息已经丢失，由此可能会导致业务数据丢失，或业务状态不一致等现象发生。为了避免这种情况，Redis提供了RPOPLPUSH命令，消费者程序会原子性的从主消息队列中取出消息并将其插入到备份队列中，直到消费者程序完成正常的处理逻辑后再将该消息从备份队列中删除。同时还可以提供一个守护进程，当发现备份队列中的消息过期时，可以重新将其再放回到主消息队列中，以便其它的消费者程序继续处理。

## **谨慎全量操作Hash、Set等集合结构**

* 在使用HASH结构存储对象属性时，开始只有有限的十几个field，往往使用HGETALL获取所有成员，效率也很高，但是随着业务发展，会将field扩张到上百个甚至几百个，此时还使用HGETALL会出现效率急剧下降、网卡频繁打满等问题【时间复杂度O(N)】,此时建议根据业务拆分为多个Hash结构；或者如果大部分都是获取所有属性的操作,可以将所有属性序列化为一个STRING类型存储！同样在使用SMEMBERS操作SET结构类型时也是相同的情况！

## **根据业务场景合理使用不同的数据结构类型**

* 目前Redis支持的数据库结构类型较多：字符串（String），哈希（Hash），列表（List），集合（Set），有序集合（Sorted Set）, Bitmap, HyperLogLog和地理空间索引（geospatial）等,需要根据业务场景选择合适的类型，常见的如：String可以用作普通的K-V、计数类；Hash可以用作对象如商品、经纪人等，包含较多属性的信息；List可以用作消息队列、粉丝/关注列表等；Set可以用于推荐；Sorted Set可以用于排行榜等！