Contents

1 概述

1.1 Redis

Redis 是一个开源（BSD 许可）的，内存中的数据结构存储系统，它可以用作数据库、缓存和消息中间件. 它支持多种类型的数据结构，如字符串（strings），散列（hashes），列表（lists），集合（sets），有序集合（sorted sets）与范围查询，bitmaps，hyperloglogs 和地理空间（geospatial）索引半径查询. Redis 内置了复制（replication），LUA 脚本（Lua scripting），LRU 驱动事件（LRU eviction），事务（transactions）和不同级别的磁盘持久化（persistence），并通过 Redis 哨兵（Sentinel）和自动分区（Cluster）提供高可用性（high availability）.

Redis 是一个典型的 NoSQL 数据库服务器，使用 ANSI C 编写并且能在绝大多数 Linux 系统上运行，官方推荐使用 Linux 部署. 单线程运行但是具有强大的性能，Redis 拥有丰富的客户端库，比如 hiredis(C)、Jedis(JAVA)、redis-py(Python) 等，您可以使用大多数的编程语言来使用 Redis.

Redis 的常用使用方式有缓存和存储两种，前者类似于 memcache，能够容忍部分数据的丢失，主要用来加速访问；后者类似于常规数据库，主要用于数据存储. 所以对于不同的应用场景需要有不同的部署方式和持久化方式.

Redis 与 memcache 对比
1. Redis 支持丰富的数据结构
2. Redis 支持主从复制
3. Redis 数据可以持久化到存储上
4. Redis 可以用来作为数据库存储，而不仅仅是缓存

5. memcache 多线程，Redis 单线程，只能利用单核 CPU
6. Redis 支持 sentinel 机制的高可用
7. Redis 拥有丰富的客户端库

1.2 UPRedis

UPRedis 是银联 Redis 团队基于开源 Redis 定制开发的缓存数据库产品，主要包括：

- 基于 Redis 2.8.23 版本开发的 UPRedis
- 基于 Twemproxy 0.4.0 版本开发的 UPRedis Proxy
- 基于 hiredis 开发的 C 语言客户端库 UPRedis-Api-c
- 基于 jedis 开发的 Java 客户端库 UPRedis-Api-Java

同时维护了 hiredis(C)、jedis(JAVA) 客户端库。

1.3 Twemproxy

Twemproxy 是由 Twitter 开源的支持 Redis 和 Memcache 协议的快速、轻量代理服务器;
可以接受多个应用的请求，按照路由规则，转发到后台各个服务器，再原路返回; 很好地解决了单个 Redis/Memcache 实例的承载能力问题;
同时也支持 pipeline 和分片机制;

1.4 UPRedis Proxy

UPRedis Proxy 是基于 Twemproxy 0.4.0 定制开发的版本；并在此基础上增加如下功能：

- 参数动态生效 (目前只支持:servers，black_list，white_list，sentinels 以及 dbpms 参数项生效)
- 主从切换感知并实时生效
- 多进程
- 管理端口和命令
- 配置文件密码加密
- 接入控制
- DBPM 认证

1.5 关于本文档

本文档全面介绍了 UPRedis 的相关特性；

主要包括如下几节：

- 版本：版本列表、功能点列表、安装包
- 功能：UPRedis、UPRedis Proxy、数据迁移等相关的功能与使用限制介绍
- 性能：基准性能、各场景下的性能、性能优化建议
- 复制：复制介绍、各种复制结构、复制性能
- 持久化：持久化介绍、持久化推荐配置
- 安全：认证使用、接入控制
- 管理：基本信息查看、日志查看
- 开发：C 语言的代码 demo、开发建议
- 迁移：离线/在线数据迁移，扩缩容

本文档可以作为开发人员，运维人员，Redis 开发者的参考文档。

2 版本

2.1 查看版本号

2.1.1 UPRedis

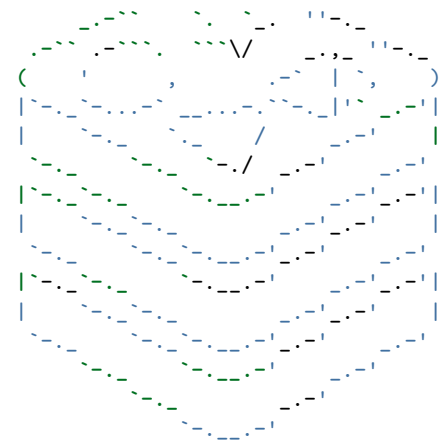
UPRedis 版本号 (Redis 的版本号) 可以在客户端执行命令查询：

```
127.0.0.1:3301> info server
# Server
redis_version:2.8.23
upredis_version:1.2.0
```

UPRedis 版本号 (Redis 的版本号) 也可以在启动 Redis 的终端上查看：

```
$./redis-server ../etc/redis.conf-3301

                _.-
            _.-``__ ''-._
```



```
UPRedis-2.8.23-1.2.0 (00000000/0) 64 bit

Running in stand alone mode
Port: 3301
PID: 12548


      http://redis.io
```

```
# Server started, UPRedis version 2.8.23-1.2.0
```

2.1.2 UPRedis Proxy

UPRedis Proxy 版本号 (Twemproxy 的版本号) 可以在管理客户端执行命令查询:

```
127.0.0.1:22222> stats
{
    "service":      "upredis-proxy",
    "source":       "vm_60b11",
    "version":      "0.4.0",
    "upversion":    "1.3.0",
    ...
}
```

UPRedis Proxy 版本号 (Twemproxy 的版本号) 也可以在启动 UPRedis Proxy 的终端上查看:

```
$ ./src/upredis-proxy -c ./etc/upredis-proxy.yml-22201 -n 4 -s 22222
[2017-07-05 10:27:34.772]    37112| 203|nc.c:nc_print_run       > upredis-proxy-1.3.0 based on nutcrac
```

2.2 建议版本

目前版本 2.1.1，作为推荐版本:

- UPRedis
  服务端: upredis-1.2.0-sles11sp2-x86_64.tar.gz
  客户端: upredis-client-1.2.0-sles11sp2-x86_64.tar.gz

- UPRedis Proxy
  服务端：`upredis-proxy-2.1.1-sles11sp2-x86_64.tar.gz`
- UPRedis Tool
  工具：`upredis-tool-2.1.0-sles11sp2-x86_64.tar.gz`

  另外提供 sles11sp3、sles11sp4、centos7 版本供应用根据环境选用
  目前 centos7 与 upel 版本都以 centos7 打包

2.3 API 版本

2.3.1 hiredis

```
-------------------------------------
| upredis-devel       [recommended]
-------------------------------------
|    groupId:          com.unionpay.common.cpackages.sles11sp2.64
|    artifactId:       upredis-devel
|    version:          1.1.0-0
|    summary:          libhiredis and libae compiled from upredis, based on Redis 2.8.23
|    homepage:         http://172.17.249.122/upredis/upredis
|    source:           internally developed
|    organization:     UPREDIS Team
|    maintainer:       unionpay
|    maintainerEmail:  wangsibing@unionpay.com
|    license:          BSD licensed
|    date:             2017-03-06 17:26:31
-------------------------------------
```

2.3.2 upredis-api-c

```
-------------------------------------
| upredis-api-c       [recommended]
-------------------------------------
|    groupId:          com.unionpay.common.cpackages.sles11sp2.64
|    artifactId:       upredis-api-c
|    version:          1.1.0-0
|    summary:          libupredis-api-c compiled from hiredis
|    homepage:         http://172.17.249.122/upredis/upredis
|    source:           internally developed
|    organization:     UPREDIS Team
|    maintainer:       unionpay
|    maintainerEmail:  lijun@unionpay.com
|    license:          BSD licensed
|    date:             2017-03-06 17:44:04
-------------------------------------
```

2.3.3 upredis-api-java

upredis-api-java 版本:1.1.0-0

```
<dependency>
    <groupId>com.unionpay.upredis</groupId>
    <artifactId>upredis-api-java</artifactId>
    <version>1.1.0</version>
</dependency>
```

2.3.4 jedis

jedis 版本：2.8.1

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.8.1</version>
</dependency>
```

2.4 版本列表

以下版本说明只针对功能点做简单说明，具体内容请参加《UPRedis 参考手册》;
在这里描述的 UPRedis 版本历史，按版本逆序排列。

2.4.1 UPRedis-2.1.1

主题：bugfix

发布测试：20180928 发布生产：20181030

缺陷修复：

- (I37-proxy)：修复 upredis-proxy MGET SCRIPT 命令在后端异常情况下断链问题.

2.4.2 UPRedis-2.1.0

主题：扩缩容及分片优化

发布测试：20180820 发布生产：20180830

功能优化：

- (I10-tool)：rmt 支持 eaval
- (I34-proxy)：proxy 添加 java_hashcode_abs 哈希算法

缺陷修复：

- (I36-proxy)：修复广播命令 (script load 等) 与 ketama 分片方式不兼容

2.4.3 UPRedis-2.0.0

主题：异地同步

发布测试：20180615 发布生产：20180720

功能优化：

- (I29-redis)：aof-binlog
- (I30-proxy)：support script load

2.4.4 UPRedis-1.4.0

主题：支持 java String.hashCode 哈希和 jump Consistent 分布

功能优化：

- (I30-proxy)：add javahashcode jump

2.4.5 UPRedis-1.3.0

主题：读写分离

发布日期：20180402

功能优化：

- (I29-proxy)：读写分离

2.4.6 UPRedis-1.2.0

主题：同步复制

发布日期：20170922

功能优化：

- (I16-redis)：同步复制

2.4.7 UPRedis-1.1.0

主题：高可管理

发布测试：2017-06-13
发布生产：2017-07-12

功能优化：

- (I10-redis)：实现 UPRedis 密码加密存储
- (I11-redis)：UPRedis 访问 dbpm 进行密码验证
- (I12-redis)：支持 auth_s 密码验证
- (I13-redis)：支持管理端口
- (I3-redis)：benchmark 优化
- (I9-redis)：编译脚本优化
- (I8-proxy)：黑白名单
- (I4-proxy)：参数动态生效
- (I9-proxy)：密码加密存储
- (I10-proxy)：支持 dbpm 认证
- (I6-proxy)：多进程
- (I5-proxy)：集成 monitor
- (I7-proxy)：管理及状态监控
- (I11-proxy)：脚本优化
- (I9-api-c)：UPRedis-Api-C 重构
- (I1-api-java)：UPRedis-Api-Java 重构
- (I1-tool)：在线数据迁移
- (I1-tool)：离线数据迁移

缺陷修复：

- 无

2.4.8 UPRedis-1.0.3

主题：缺陷修复

发布生产：2017-4-7

功能优化：

- 无

缺陷修复：

- (I1-redis)：修复 UPRedis Monitor 配置文件重写缺陷

2.4.9 UPRedis-1.0.2

主题：缺陷修复

发布生产：2016-12-26

功能优化：

- (I4-redis)：UPRedis 银联版本号显示

缺陷修复：

- (I5-redis)：修复 UPRedis 编译问题导致 coredump

2.4.10 UPRedis-1.0.0

主题：组件化

发布生产：2016-08-02

功能优化：

- (I00-api-c)：UPRedis-Api-C
- (I00-tool)：UPRedis 主从切换监控及 Twemproxy 配置文件修改

3 功能

3.1 内容简介

本手册简要描述 UPRedis 的功能，包括如下内容：

- UPRedis
- UPRedis Proxy
- HA(高可用)
- 数据迁移
- 使用限制

  更多详细内容，请参见相应的redis 文档。

3.2 UPRedis

3.2.1 安全

3.2.1.1 配置项

```
requirepass foobared
或者
requirepass_s 1b58ee375b42e41f0e48ef2ff27d10a5b1f6924a9acdcdba7cae868e7adce6bf
或者
dbpm "failover://127.0.0.1:12345:5000:db:usr,127.0.0.1:12346:5000:db:usr"
```

3.2.1.2 使用

使用客户端连接服务器时，输入

```
$ AUTH foobared
或者
$ AUTH_S 1b58ee375b42e41f0e48ef2ff27d10a5b1f6924a9acdcdba7cae868e7adce6bf
```

完成密码验证。

如果是 slave 服务器，还需设置配置文件中的 masterauth 或者 masterauth_s 或者 dbpm 属性

> 注意：主库也要配置 masterauth 或者 masterauth_s 或者 dbpm 属性
> 以备主备切换

更多安全信息，请查看《**upredis-guide-**安全》

3.2.2 日志

3.2.2.1 配置项

```
/* 日志文件名称，该值为空时会定向到终端 */
logfile ""

/*
 * 日志级别，支持 debug、verbose、notice、warning，默认为 verbose
 * debug 记录很多信息，用于开发和测试
 * varbose 有用的信息，不像 debug 会记录那么多
 * notice 普通的 verbose，常用于生产环境
 * warning 只有非常重要或者严重的信息会记录到日志
 */
loglevel notice
```

3.2.3 慢日志

UPRedis 的慢日志查询功能用于记录执行时间超过给定时长的命令请求，
用户可以通过这个功能产生的日志来监控和优化查询速度。

3.2.3.1 配置项

```
/* 记录执行时间超过设置时间的命令，单位微秒 */
slowlog-log-slower-than 10000

/* 最大存储多少条，无限制，但是消耗内存 */
slowlog-max-len 128
```

3.2.3.2 查看慢日志

在客户端中使用如下语句打印指定数量的日志

```
SLOWLOG GET [number]
```

3.2.4 监控

请查看《**upredis-guide-**管理》

3.2.5 持久化

请查看《**upredis-guide-**持久化》

3.2.6 复制

请查看《**upredis-guide-**复制》

3.2.7 异地同步

请查看《**upredis-guide-**异地同步》

3.2.8 pipeline

3.2.8.1 介绍

upredis 本身是一个 C/S 结构的数据结构服务器，其对外提供查询命令的方式通常是阻塞的，命令序列如下：

```
Client: INCR X
Server: 1
Client: INCR X
Server: 2
```

```
Client: INCR X
Server: 3
```

如上序列的问题在于单条命令的执行时间远小于网络传输时间，因此 UPRedis 提供了 pipeline 的功能来尽可能的减小这种阻塞和网络传输时间带来的性能损失。

pipeline 允许客户端一次发送多条命令，而服务端顺序执行这些命令并最终返回数据。

pipeline 的典型命令序列如下：

```
Client: INCR X
Client: INCR X
Client: INCR X
Server: 1
Server: 2
Server: 3
```

3.2.8.2 使用

请查看《**upredis-guide-**开发》

3.2.9 pub/sub 机制

3.2.9.1 介绍

UPRedis 使用 SUBSCRIBE、UNSUBSCRIBE 和 PUBLISH 等命令实现了发布与订阅信息机制，在这个实现中，发送者（发送信息的客户端）不是将信息直接发送给特定的接收者，而是将信息广播给所有订阅该频道的订阅者。

3.2.9.2 频道订阅

订阅：

```
SUBSCRIBE [channel] [channel ...]
```

发布：

```
PUBLISH [channel] [message]
```

退订：

```
UNSUBSCRIBE [channel] [channel ...]
```

3.2.9.3 模式订阅

UPRedis 的 pub/sub 机制支持模式匹配，用户可以使用 `PSUBSCRIBE` 和 `PUNSUBSCRIBE` 来订阅符合某个模式 (pattern) 的频道。

```
PSUBSCRIBE   foo*
PUNSUBSCRIBE foo*
```

UPRedis 也支持其他一些通配符，包括 "* ？ […]" 等等。

3.2.9.4 keyspace 通知

keyspace notification 基于 Pub/sub 机制，提供了一套 key 值空间的观察与管理接口。

```
$ redis-cli config set notify-keyspace-events KEA
$ redis-cli --csv psubscribe '__key*'
```

则在另一个客户端执行命令时，该客户端收到如下消息：

```
"pmessage","__key*","__keyspace@0__:foo","set"
"pmessage","__key*","__keyevent@0__:set","foo"
```

3.2.10 expire 机制

3.2.10.1 介绍

UPRedis 的一个重要特性是 expire(过期) 机制，它允许给一个 key 值设定超时时间。这对于会话 (session) 类信息的存储格外便利。

在设定某个 key 值的超时时间后，如下命令会修改或更新超时时间：

- DEL
- SET
- GETSET
- *STORE 类
- PERSIST 将会重置该 key 值为不过期的 key 值
- RENAME 如果仅仅是改名，则该超时时间仍然有效；如果是 RENAME key1 key2，则 key2 的超时时间失效。
- EXPIRE 重置超时时间为新的超时时间

3.2.10.2 使用

使用 expire 机制的典型案例如下：

```
$ SET key "Hello"
OK
$ EXPIRE mykey 10
(integer) 1
$ TTL mykey
(integer) 10
$ SET mykey "Hello World"
OK
$ TTL mykey
(integer) -1
```

3.2.10.3 expire 时间精度

expire 所设定的时间精度在当前版本不大于一毫秒。
此外，超时时间依赖于系统时间，因此系统时间必须保持稳定。

3.2.11 事务

3.2.11.1 介绍

UPRedis 的事务提供了一种将多个命令请求打包，然后一次性、按顺序地执行多个命令的机制，并且在事务执行期间，服务器不会中断事务而改去执行其他客户端的命令请求，只有事务中所有的命令都执行完毕，才会处理其他客户端的命令请求。

1. 并不具备完整的 ACID 属性
2. 不支持事务回滚
3. 存在部分成功部分失败的情况
4. 一般不建议使用

事务包含五条命令：

```
MULTI      开启事务
EXEC       执行事务
DISCARD    取消事务
WATCH      监视一个或多个 key
UNWATCH    取消监视一个或多个 key
```

3.2.11.2 使用举例

```
WATCH key /* 监视键 key */
MULTI  /* 事务开始 */
SET key value
```

```
get key
EXEC /* 执行事务 */
```

3.2.11.3 事务处理中的问题

3.2.11.3.1 事务中断

示例事务中可能在两个地方出现错误：exec 前与 exec 后。对于 exec 前出现的错误，UPRedis 会丢弃该事务中所有命令；对于 exec 后所出现的错误，所有命令都将被执行，然后依次返回命令错误信息。如：

exec 执行前:

```
$ MULTI
OK
$ incr a b c
(error) ERR wrong number of arguments for 'incr' command
```

exec 执行后:

```
$ MULTI
OK
$ set a 4
QUEUED
$ lpop a
QUEUED
$ exec
1) OK
2) (error) WRONGTYPE Operation against a key holding the wrong kind of value
```

3.2.11.3.2 回滚

UPRedis 当前不支持事务回滚，原因包括：UPRedis 在事务中出现的错误都是可预见的用户端错误；以及 UPRedis 更重视快速与简单。

3.2.11.3.3 watch

UPRedis 中的 watch 命令实质是一个 check-and-set(简化的 compare-and-swap)：事务中有任何被 watch 的值，则该事务终止并返回空值，所有 queued 的命令均不会执行。示例：

```
$ watch a
OK
$ incr a
(integer) 3
```

```
$ multi
OK
$ set a 1
QUEUED
$ set b 10
QUEUED
$ exec
(nil)
$ get b
"1"
$ get a
"3"
```

在事务执行后 (无论成功与否)，被 watch 的值均被置为 unwatched。

3.3 UPRedis Proxy

3.3.1 安全

3.3.1.1 配置项

```
redis_auth: foobared
或者
redis_auth_s 1b58ee375b42e41f0e48ef2ff27d10a5b1f6924a9acdcdba7cae868e7adce6bf
或者
dbpms:
 - IP1:PORT1:db1:usr1
 - IP2:PORT2:db2:usr2
```

```
white_list:
- '*.*.*.*'
black_list:
- 172.17.100.100
```

3.3.1.2 使用

使用客户端连接服务器时，输入

```
$ AUTH foobared
或者
$ AUTH_S 1b58ee375b42e41f0e48ef2ff27d10a5b1f6924a9acdcdba7cae868e7adce6bf
```

完成密码验证。

更多安全信息，请查看《**upredis-guide-**安全》

3.3.2 日志

UPRedis Proxy 的日志需要在启动时通过命令行指定，请查阅启动部分的说明。

日志格式如下：

```
upredis-proxy starting...
[2017-07-05 11:29:41.536]    43311| 203|nc.c:nc_print_run       > upredis-proxy-1.3.0 based on nutcrac
[2017-07-05 11:29:41.536]    43311| 208|nc.c:nc_print_run       > run, rabbit run / dig that hole, for
```

3.3.3 交易日志

报文日志记录 twemproxy 接收的所有报文

```
-r, --reqestlog          : write the log to a file REQ_xxx  (记录报文日志)
```

3.3.4 故障恢复

建议设置 auto_eject_hosts 为 false，
server 故障时，proxy 会间隔 timeout+server_retry_timeout(均可配置) 秒去尝试重连 server，
在 server 故障期间，所有路由到故障 server 的请求都会回复 server timeout。

不建议设置 auto_to_eject_hosts 为 true，因为会导致分片错乱。

3.3.5 分片

配置文件中关于分片的参数选项如下：

```
# 存在 ketama、modula、random 和 jump 四种可选的配置
# ketama: ketama 一致性 hash 算法，会根据服务器构造出一个 hash ring，
#    并为 ring 上的节点分配 hash 范围。ketama 的优势在于单个节点添加、删除之后，
#    会最大程度上保持整个群集中缓存的 key 值可以被重用
# modula: 就是根据 key 值得 hash 值取模，根据取模的结果选择对应的服务器
# random: 不论 key 的 hash 值是多少，随机选择一个服务器
# jump: goole jump consistent hash(upredis-proxy-1.4.0 新增)
# 一般建议应用使用 modula 模式
distribution: modula
```

hash 算法用于将输入的 key 转换为 hash 值，根据 hash 值落入的分片算法分配的各 server 的 hash 值区间，决定路由到哪个 server。

hash_tag 用于限制使用 key 中哪些部分计算 hash 值:
比如 hash_tag=“{}”，输入 set ___{Union}__Pay___ value，那么 hash 算法只会使用 “Union” 来计算 hash 值。

UPRedis Proxy 目前不支持分片规则在线变更

3.3.6 管理端口

UPRedis Proxy 在 twemproxy 原有状态端口 (通过-s 指定，默认 22222) 的基础上，增加了管理功能。

客户端连接管理端口，通过发送管理命令，能够实现参数重载，数据迁移 (需配合迁移工具)，状态信息查看，关闭 UPRedis Proxy 等功能。

3.3.7 管理命令

Service	Function	concurrent
reload_redis	配置servers项出现增删改，执行本命令，使修改生效	No
reload_sentinel	配置sentinels项出现增删改，执行本命令，使修改生效	No
stats	查询proxy的状态信息 (所有 worker 状态信息的总和)	No
stats_all	查询每个worker的状态信息	No
migrate_start	设置proxy的状态为 MIGRATING，拒绝所有请求 (部分管理命令除外)	No
migrate_end	设置proxy的数据迁移状态为 MIGRATED，接收所有请求	No
shutdown	关闭proxy	Yes

处于数据迁移状态，管理端口只接收 migrate_end、reload_redis 和 shutdown。

正常情况，除了 shutdown，其他命令都不能并发。

3.3.8 多进程

UPRedis Proxy 支持多进程，通过启动参数指定需要启动的 worker 的数量，默认 1，最大支持 99。

多进程的性能 请查看《**upredis-guide-**性能》

3.3.9 主备切换

UPRedis Proxy 配置文件中 sentinels 配置项指定主从切换消息源。

```
sentinels:
  - 127.0.0.1:3501
  - 127.0.0.1:3502
  - 127.0.0.1:3503
```

主从切换发生后:
1.UPRedis Proxy 的 monitor 模块接收 sentinel 发送的切换信息
2. 修改配置文件
3. 进行参数重载

3.3.10 读写分离

UPRedis Proxy 配置文件中 separate_read_write 配置项开启读写分离功能, 使用读写分离功能能够显著提升 Proxy 读取性能。

```
    separate_read_write: read_slaves
```

separate_read_write 共有三种模式:

- off: 默认模式，关闭读写分离
- read_slaves: 读请求只转发到 slave（除非所有 slave 都不可用）
- read_both: 读请求转发到 slave 和 master

3.4 HA(高可用)

3.4.1 sentinel

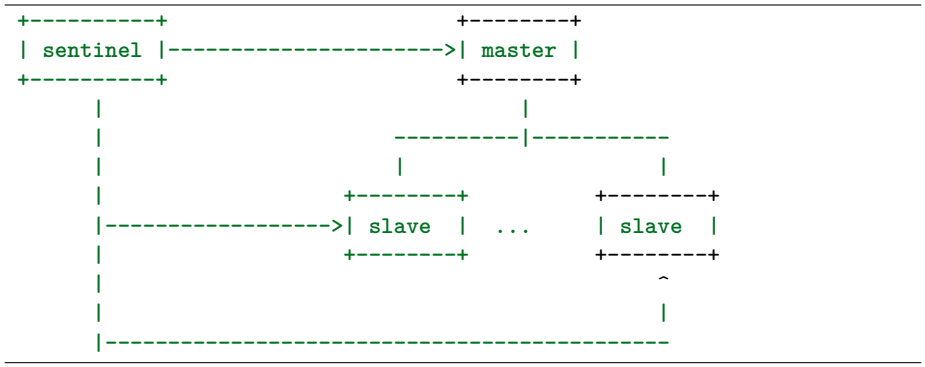
3.4.1.1 介绍

sentinel 是 Redis 官方推荐的 Redis 高可用解决方案，具备监控，通知，自动故障转移，配置提供者的功能。

具体功能介绍如下:

- 监控（Monitoring），sentinel 时刻监控着 Redis master-slave 是否正常运行。
- 通知（Notification），sentinel 可以通过 api 来通知管理员，被监控的 Redis master-slave 出现了问题。
- 自动故障转移（Automatic failover），当 Redis master 出现故障不可用状态，sentinel 会开始一次故障转移，将其中一个 slave 提升为新的 master，其他的 slave 将重新配置使用新的 master 同步，并使用 Redis 的服务器应用程序在连接时收到使用新的地址连接。
- 配置提供者（Configuration provider），sentinel 作为在集群中的权威来源，客户端连接到 sentinel 来获取某个服务的当前 Redis 主服务器的地址和其他信息。当故障转移发生时，sentinel 会报告新地址。

3.4.1.2 部署



3.4.1.3 监控状态

+reset-master ： 主服务器已被重置。
+slave ： 一个新的从服务器已经被 **sentinel** 识别并关联。
+failover-state-reconf-slaves ： 故障转移状态切换到了 **reconf-slaves** 状态。
+failover-detected ： 另一个 **sentinel** 开始了一次故障转移操作，或者一个 **slave** 转换成了 **master**。
+slave-reconf-sent ： 作为 **leader** 的 **sentinel** 向实例发送了 **SLAVEOF** 命令，为实例设置新的主服务器。
+slave-reconf-inprog ： 实例正在将自己设置为指定 **master** 的 **slave**，但相应的同步过程仍未完成。
+slave-reconf-done ： 从服务器已经成功完成对新 **master** 的同步。
-dup-sentinel ： 对给定主服务器进行监视的一个或多个 **sentinel** 因为重复出现而被移除 —— 当 **Sentinel** 实例重启的时候，
+sentinel ： 一个监视给定主服务器的新 **sentinel** 被加入到 **sentinel** 集群中。
+sdown ： 给定的实例现在处于主观下线状态。
-sdown ： 给定的实例已经不再处于主观下线状态。
+odown ： 给定的实例现在处于客观下线状态。
-odown ： 给定的实例已经不再处于客观下线状态。
+new-epoch ： 当前的纪元（**epoch**）已经被更新。
+try-failover ： 一个新的故障迁移操作正在执行中，等待被大多数 **sentinel** 选中。
+elected-leader ： 赢得指定纪元的选举，可以进行故障迁移操作了。
+failover-state-select-slave ： 故障转移操作现在处于 **select-slave** 状态——**sentinel** 正在寻找可以升级为主服务器的
no-good-slave ： **Sentinel** 操作未能找到适合进行升级的从服务器。**Sentinel** 会在一段时间之后再次尝试寻找合适的从服务器来
selected-slave ： **Sentinel** 顺利找到适合进行升级的从服务器。
failover-state-send-slaveof-noone ： **sentinel** 正在将指定的从服务器升级为主服务器，等待升级功能完成。
failover-end-for-timeout ： 故障转移因为超时而中止。
failover-end ： 故障转移操作顺利完成。所有从服务器都开始复制新的主服务器了。
+switch-master ： 配置变更，主服务器的 **IP** 和地址已经改变。
+tilt ： 进入 **tilt** 模式。
-tilt ： 退出 **tilt** 模式。

3.4.1.4 主从切换

一次完整的主从切换的日志:

```
[3254] 21 Oct 14:25:27.489 # +try-failover master mymaster 127.0.0.1 6379
[3254] 21 Oct 14:25:27.490 # +vote-for-leader c92ad0bd8f50530dfd657e469ba1ed5a82e21f05 2
[3254] 21 Oct 14:25:27.492 # 127.0.0.1:26381 voted for c92ad0bd8f50530dfd657e469ba1ed5a82e21f05 2
[3254] 21 Oct 14:25:27.493 # 127.0.0.1:26379 voted for c92ad0bd8f50530dfd657e469ba1ed5a82e21f05 2
[3254] 21 Oct 14:25:27.542 # +elected-leader master mymaster 127.0.0.1 6379
[3254] 21 Oct 14:25:27.542 # +failover-state-select-slave master mymaster 127.0.0.1 6379
[3254] 21 Oct 14:25:27.619 # +selected-slave slave 127.0.0.1:6380 127.0.0.1 6380 @ mymaster 127.0.0.1
[3254] 21 Oct 14:25:27.619 * +failover-state-send-slaveof-noone slave 127.0.0.1:6380 127.0.0.1 6380 @
[3254] 21 Oct 14:25:27.681 * +failover-state-wait-promotion slave 127.0.0.1:6380 127.0.0.1 6380 @ myma
[3254] 21 Oct 14:25:28.508 # +promoted-slave slave 127.0.0.1:6380 127.0.0.1 6380 @ mymaster 127.0.0.1
[3254] 21 Oct 14:25:28.508 # +failover-state-reconf-slaves master mymaster 127.0.0.1 6379
[3254] 21 Oct 14:25:28.582 * +slave-reconf-sent slave 127.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0.0
[3254] 21 Oct 14:25:29.534 * +slave-reconf-inprog slave 127.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0
[3254] 21 Oct 14:25:29.535 * +slave-reconf-done slave 127.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0.0
[3254] 21 Oct 14:25:29.586 # -odown master mymaster 127.0.0.1 6379
[3254] 21 Oct 14:25:29.586 # +failover-end master mymaster 127.0.0.1 6379
[3254] 21 Oct 14:25:29.587 # +switch-master mymaster 127.0.0.1 6379 127.0.0.1 6380
```

当出现 +try-failover 时，表示 sentinel 正在做主从切换，当出现 failover-end 时，表示故障转移完成。

3.4.1.5 启动

3.4.1.5.1 通过 **redis-server** 启动

```
./redis-server --sentinel [配置文件]
```

3.4.1.5.2 通过 **redis-sentinel** 启动

```
./redis-server [配置文件]
```

3.4.2 UPRedis Proxy

UPRedis Proxy 在高可用管理上也是一个重要的参与者，
主要功能是订阅多个 sentinel 的主从切换信息，然后在线修改 redis-server 的信息，以达到对应用透明的主备切换效果；

具体信息参考本章 UPRedis Proxy 功能-主备切换

3.5 数据迁移

3.5.1 离线数据迁移

请查看《upredis-guide-迁移》

3.5.2 在线数据迁移

请查看《upredis-guide-迁移》

3.6 使用限制

3.6.1 UPRedis

3.6.1.1 UPRedis 密码优先级

密码的设置有三种方式:

1. 明文密码：requirepass foobared
2. 密文密码:requirepass_s 1b58ee375b42e41f0e48ef2ff27d10a5b1f6924a9acdcdba7cae868e7adce6bf
3. 访问 DBPM 获取密码:dbpm “failover://172.18.64.196:7000:5000:dbname:usrname,172.18.64.196:8000:5000

密码优先级：密文 > 明文 >DBPM, 不建议配置明文密码

密文密码采用 sha256 加密，获取方式为：“echo -n foobared | sha256sum”

DBPM 配置支持最多两台，格式为：failover://ip1:port1:timeout1:db1:usr1,...

3.6.2 UPRedis Proxy

3.6.2.1 UPRedis Proxy 密码优先级

密码的设置有三种方式:

1. 明文密码：redis_auth: foobared
2. 密文密码:redis_auth_s: 1b58ee375b42e41f0e48ef2ff27d10a5b1f6924a9acdcdba7cae868e7adce6bf
3. dbpm 信息

```
dbpms:
  - IP1:PORT1:db1:usr1
  - IP2:PORT2:db2:usr2
```

密码优先级：密文 > 明文 >DBPM, 不建议配置明文密码

密文密码采用 sha256 加密，获取方式为：“echo -n foobared | sha256sum”

DBPM 配置支持最多两台

3.6.2.2 UPRedis Proxy 管理端口不支持 pipeline

如果向管理端口以 pipeline 的形式发送请求，UPRedis Proxy 会断开与前端的连接，并记录日志：

```
mgm command don't support pipeline
```

3.6.2.3 UPRedis Proxy 接收到不支持的命令或者参数不足的命令，会断开与前端连接

比如发送：

*1
$4
ssss

或者
*1
$3
set

UPRedis Proxy 接收到类似的请求，都会断开前端连接，并记录日志：

```
[2017-07-01 19:32:36.378]    22046|1099|nc_redis.c:redis_parse_*> parsed unsupported command 'sss'
[2017-07-01 19:32:36.378]    22046|1740|nc_redis.c:redis_parse_*> parsed bad req 6 res 1 type 0 state
00000000  2a 31 0d 0a 24 33 0d 0a  73 73 73 0d 0a            |*1..$3..sss..|
[2017-07-01 19:32:36.378]    22046| 797|nc_core.c:core_close    > close client:redis 10 '127.0.0.1:359

[2017-07-01 19:33:25.138]    22232|1740|nc_redis.c:redis_parse_*> parsed bad req 6 res 1 type 70 state
00000000  2a 31 0d 0a 24 33 0d 0a  73 65 74 0d 0a            |*1..$3..set..|
[2017-07-01 19:33:25.138]    22232| 797|nc_core.c:core_close    > close client:redis 10 '127.0.0.1:445
```

3.6.2.4 UPRedis Proxy 内存管理，pipeline 限制

UPRedis Proxy 的内存，一旦申请，就不会释放，而 pipeline 模式会导致大规模的内存分配，所以需要重点评估 pipline 模式下的内存需求

```
单个连接需要的内存 =*2*mbuf_size*P

mubf_size可以通过-m 启动参数配置，默认 16KB
P表示一次 pipeline 中的命令数量
```

例如:一个客户端一次 10000 个指令的 pipeline,需要分配的内存是 10000*16K*2=250M

因此一般不建议大规模的 pipeline，一般以几十个命令 pipeline 最佳；

3.6.2.5 UPRedis Proxy 不支持多 auth 的 pipeline

如果一次 pipeline 发送类似以下的请求，UPRedis Proxy 不支持

```
auth
<多个请求>
auth
<多个请求>
```

对于上面请求，只有第一个 auth 才认为是有效认证。

如果第一个 auth 认证通过，则之后的所有请求（不支持的命令或者参数不足的命令除外）都会去后端。

如果第一个 auth 认证不通过，则之后的所有请求都会失败，对于其他的 auth 会回复：

```
-ERR unsupport multi-auth pipeline\r\n
```

3.6.2.6 sentinel 信息筛选问题

UPRedis Proxy 对于相同的切换消息只执行一次。

UPRedis Proxy 会监听配置文件中配置的所有 sentinel 的 +switch-master 频道，
当多个主 redis server 宕机时，每个 sentinel 的 +switch-master 频道中都会出现多条切换消息，
由于网络原因，不同 sentinel 发送的消息 UPRedis Proxy 接收可能有先后（**sentinel** 间的消息有先后，**sentinel** 内部的消息是按序的），
这样会出现一个现象：

日志一开始提示切换成功，然后提示过滤掉重复的切换消息，最后提示切换失败。

最后提示切换失败，是因为切换成功后，配置文件已经更新，但是切换消息还是旧的信息，提示切换失败。

如果出现切换失败的日志，需要分析是不是属于上述情况

3.6.2.7 UPRedis Proxy 支持在线变更的配置项

```
servers

sentinels

dbpms

black_list

white_list
```

3.6.2.8 UPRedis Proxy DBPM 在线变更仅对新连接有效

如果在线变更 DBPM，仅影响新的连接，旧的连接不受影响

3.6.2.9 UPRedis Proxy 重连后端的时间间隔

如果 auto_eject_hosts 设置为 true，后端重连时间间隔为 server_retry_timeout

如果 auto_eject_hosts 设置为 false,后端重连时间间隔为 server_retry_timeout+timeout

3.6.2.10 UPRedis Proxy auth 说明

针对每一条前端连接，UPRedis Proxy 只认为第一个 auth 的结果是有效认证。

如果第一个 auth 认证通过，则之后的所有请求 (不支持的命令或者参数不足的命令除外) 都会去后端。

auth 认证通过，并且后端认证也通过，假如再次发送一个 **auth** ，会导致后端某个 **redis server** 需要重新认证。

如果第一个 auth 认证不通过，则之后的所有请求都会失败。

3.6.2.11 UPRedis Proxy 通讯 unix socket 说明

UPRedis Proxy 启动后，会在当前执行目录下生成一个 tmp 文件夹 (当前执行目录需要有写权限)，用于存放管理进程与工作进程的通讯 unix domain socket。程序运行期间，不允许对 tmp 文件夹进行任何操作 (如果误删除，会导致 worker 在重启时，找不到 socket 文件而退出)。程序退出后，会自动删除 tmp 文件夹下的 unix domain socket。

3.6.2.12 管理命令并发限制

处于数据迁移状态 (migrating)，管理端口只接收 migrate_end、reload_redis 和 shutdown。

正常情况，除了 shutdown，其他命令都不能并发。

3.6.3 UPRedis Tool

3.6.3.1 redis-migrate-tool

3.6.3.1.1 使用场景

数据源：**RDB**

1.rdb ————————>UPRedis Proxy(rdb 数据迁移到带 proxy 的 redis 集群)

- 一个 rdb 文件——>UPRedis Proxy
- 多个 rdb 文件——>UPRedis Proxy

2.rdb ————————>single(rdb 数据迁移到一个或者多个 redis server)

- 一个 rdb 文件——> 一个 redis server
- 一个 rdb 文件——> 多个 redis server(不支持)
- 多个 rdb 文件——> 一个 redis server
- 多个 rdb 文件——> 多个 redis server(不支持)

3.rdb ————————>rdb(不支持)

4.rdb ————————>aof(不支持)

数据源：**AOF**

1.aof————————->upredis proxy(aof 数据迁移到带 proxy 的 redis 集群)

- 一个 aof 文件——>UPRedis Proxy
- 多个 aof 文件——>UPRedis Proxy

2.aof ————————>single(aof 数据迁移到一个或者多个 redis server)

- 一个 aof 文件——> 一个 redis server
- 一个 aof 文件——> 多个 redis server(不支持)
- 多个 aof 文件——> 一个 redis server
- 多个 aof 文件——> 多个 redis server(不支持)

3.aof ————————>rdb(不支持)

4.aof ————————>aof(不支持)

数据源：**single**

1.single————————->UPRedis Proxy(单个或多个 redis 服务器的数据迁移到带 proxy 的 redis 集群)

- 一个 redis server——>UPRedis Proxy
- 多个 redis server——>UPRedis Proxy

2.single————————>single(一个或者多个 redis server 迁移到一个或者多个 redis server)

- 一个 redis server——> 一个 redis server
- 一个 redis server——> 多个 redis server(不支持)
- 多个 redis server——> 一个 redis server
- 多个 redis server——> 多个 redis server(不支持)

3.single————————>rdb

- 一个 redis server——>rdb
- 多个 redis server——>rdb(不支持, 生成多个文件)

4.single————————>aof(不支持)

数据源：**UPRedis Proxy**

1.UPRedis Proxy——>UPRedis Proxy(带 proxy 的 redis 集群的数据迁移到带 proxy 的 redis 集群)

注意：如果数据源中存在不属于数据源 **proxy** 分片算法分片的数据，这部分数据不会被迁移 **(**但是在线生成的数据会被迁移**)**

2.UPRedis Proxy——>single(带 proxy 的 redis 集群的数据迁移到一个或者多个 redis server)

- UPRedis Proxy—> 一个 redis server
- UPRedis Proxy—> 多个 redis server(不支持)

3.UPRedis Proxy——>rdb(不支持, 生成多个文件, 根据源端 redis 的数量)

4.UPRedis Proxy——>aof(不支持)

3.6.3.1.2 配置文件说明

如果迁移集群类型中有 twemproxy，rmt 配置文件中的配置项需要与对应 UPRedis Proxy 的配置项一致，需要一致的配置项:

```
servers
redis_auth
hash
hash_tag
distribution
```

threads 配置项表示 rmt 启动的读写线程数量，默认等于机器的 cpu 核心数，如果数据源是多个 **rdb** 文件或者 **aof** 文件，需要保证 **threads** 的数量大于文件个数 **(**最好大于文件个数的两倍**)**，如果小于，会导致部分文件不迁移。
性能

3.7 单机性能

3.7.1 基准性能

本节介绍 UPRedis 的基准性能测试

- 数据量 1500 万
- 数据大小 512Byte
- 局域网跨机，UPRedis 部署在物理机，UPRedis Proxy 部署在虚拟机

```
+-----------+-----------+-----------+-----------+-----------+
|           |        UPRedis        |    UPRedis-pieline    |
+           +-----------+-----------+-----------+-----------+
|           |   QPS     |  latency  |    QPS    |  latency  |
+-----------+-----------+-----------+-----------+-----------+
|   SET     |   110000  |    235    |   505050  |   1361    |
+-----------+-----------+-----------+-----------+-----------+
```

GET	120000	240	713775	1331
HSET	106000	239	822667	1286
INCR	120000	243	802568	1172
LPUSH	120000	235	536509	1372
LPOP	120000	241	628140	1325
SADD	120000	241	930752	1145
SPOP	125000	312	1335648	1155

1. latency 单位是微秒
2. latency 的测试条件是固定 100 个并发，10000TPS 时的延迟
3. 物理机和虚拟机 CPU 等硬件条件基本一致 (CPU 核心数和存储除外)，具体信息详见测试相关说明
4. 由于 UPRedis 都是单进程单线程程序，所以 CPU 的核心数对性能不造成影响，CPU 型号会有影响
5. 上述测试数据都是在保证带宽、内存足够的条件下获取的
6. 上述测试数据的瓶颈都是 CPU，其中 UPRedis 测试数据是 UPRedis CPU 100%;
7. 测试环境中 UPRedis 采用单实例 (一个进程)
8. UPRedis Proxy 以 pipeline 的形式转发请求，因此使用 UPRedis Proxy 可以使得 Redis 性能超过 10W
9. UPRedis-proxy 单进程性能大约 8W QPS，在 6 进程时达到 Redis-Pipeline 的极限性能 50W QPS

3.7.2 各种部署下的性能

本节介绍 UPRedis 在各种部署下的性能对比

- 数据量 1500 万
- 数据大小 512Byte
- 局域网跨机，UPRedis 物理机，Proxy 虚拟机
- 数据分析

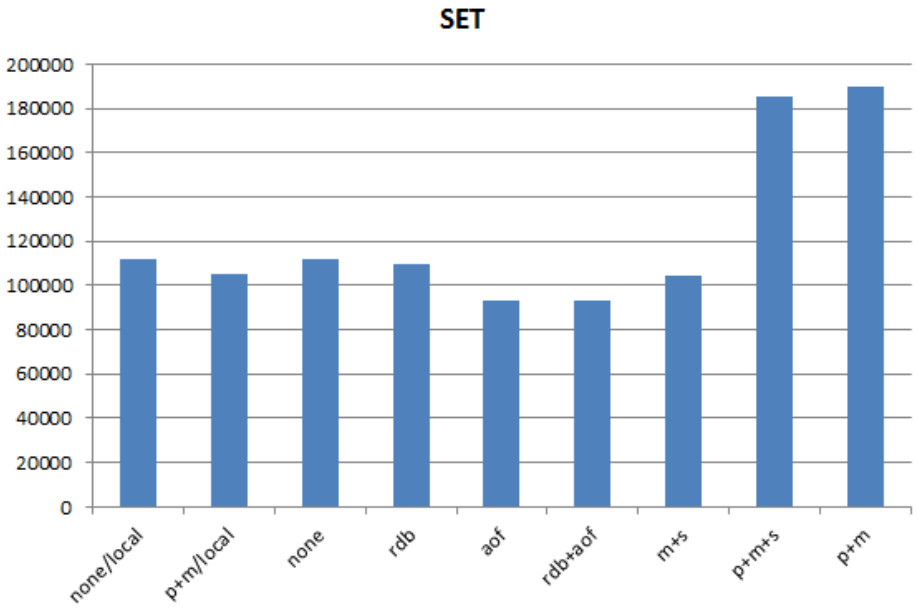


Figure 1: 各种部署下的性能-set

- none 和 rdb 性能相近，说明合适的 rdb 方式几乎不影响性能

- aof 会有一定的性能影响
- 每加挂一个 slave 都会拖慢 master 的性能，所以避免过多的 slave，一般不超过 4 台

- 由于 UPRedis Proxy 性能瓶颈，所以导致通过 Proxy 后性能都有一定的下降
- 但是 UPRedis Proxy 能提供诸如分片、高可用等功能，所以一般建议采用 UPRedis + UPRedis Proxy 部署
- 测试环境中 UPRedis 采用单实例 (一个进程)，UPRedis Proxy 也采用单实例 (四个工作进程)
- UPRedis Proxy 以 pipeline 的形式转发请求，并且多个工作进程增加了 UPRedis 一次事件循环处理的事件数量，从而增加了 UPRedis 的处理能力

- 数据分析
- get 由于不涉及写，所以持久化方式和 slave 都不会带来额外的开销
- 测试环境中 UPRedis 采用单实例 (一个进程)，UPRedis Proxy 也采用单实例 (四个工作进程)
- UPRedis Proxy 以 pipeline 的形式转发请求，并且多个工作进程增加了 UPRedis 一次事件循环处理的事件数量，从而增加了 UPRedis 的处理能力

  备注:
  none:redis server alone
  local：本地连接

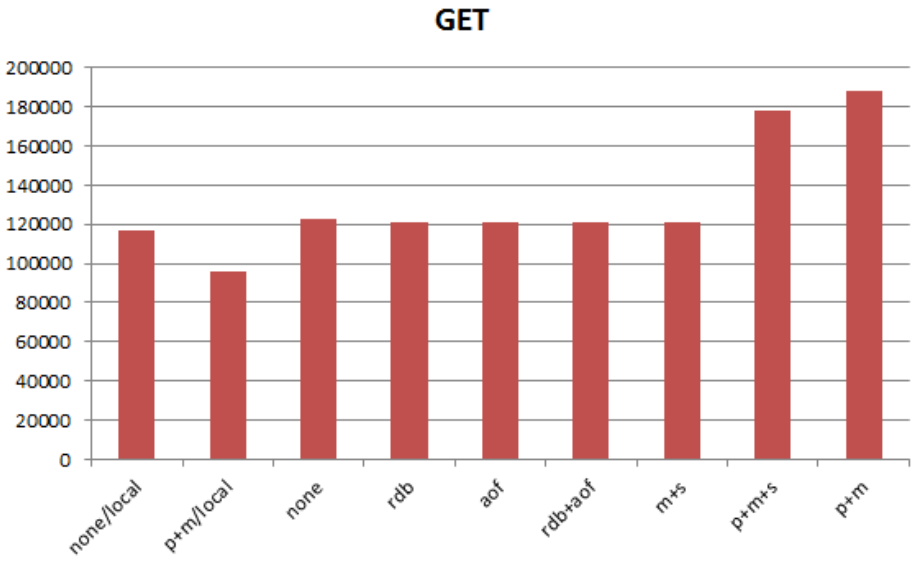


Figure 2: 各种部署下的性能-get

rdb：开启了 RDB 持久化
aof：开启了 AOF 持久化
rdb+aof：开启了 RDB 和 AOF 持久化
m+s：一主一从部署
p+m+s：UPRedis Proxy+ 一主一从部署
p+m：UPRedis Proxy+ 一主部署

3.7.3 value 大小对性能影响

本节介绍 UPRedis 的 value 大小对性能的影响

- 分别测试了 value 大小为 50、128、256、512、1024、2048、4096、8192 场景下的性能
- 结论：随着 value 的增大，性能缓慢下降

3.7.4 并发数对性能影响

本节介绍 UPRedis 的客户端数量对性能的影响
- 分别测试了客户端数量为 50、128、256、512、1024、2048、4096、8192 场景下的性能

- 结论：get/set 在并发数 128-2048 性能达到峰值

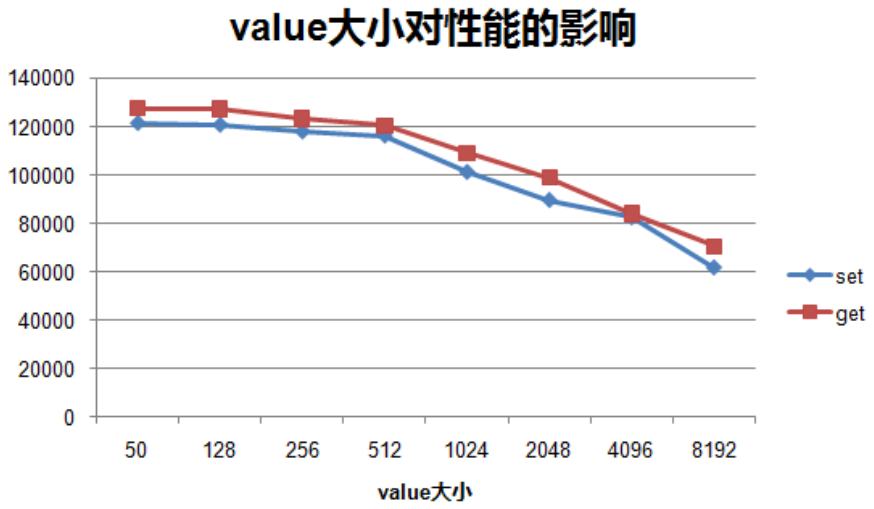


Figure 3: value 大小对性能影响

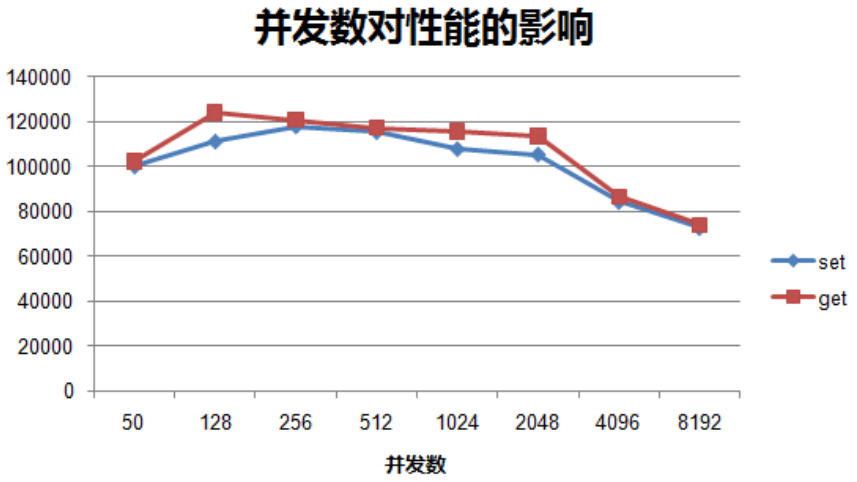


Figure 4: 并发数对性能影响-set

3.7.5 持久化方式对性能影响

本节介绍 UPRedis 的持久化方式对性能的影响
- 分别测试了不使用持久化 (memory)、rdb、rdb+aof(everysec)、aof(everysec)、aof(always) 五种方式
- 物理机

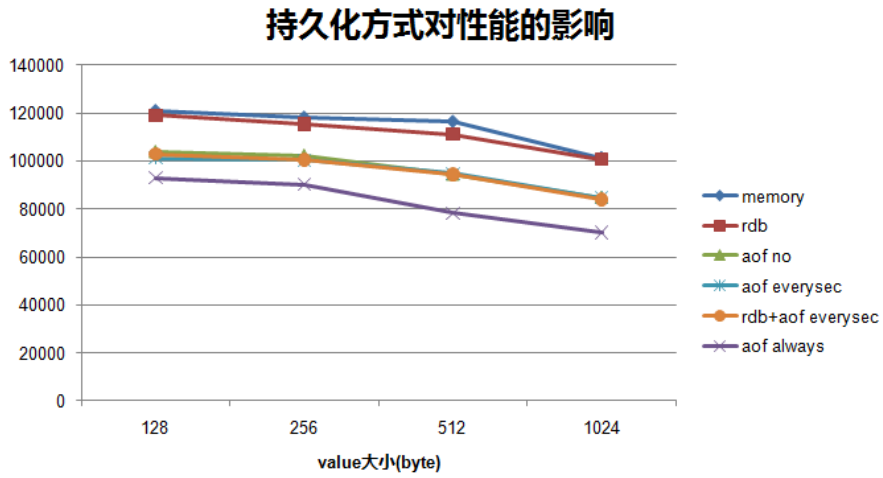


Figure 5: 持久化方式对性能影响

- 结论:
  - rdb 几乎不影响性能;
  - aof(everysec、no) 会有 15% 的性能损耗;
  - aof(always) 会有 30% 的性能损耗

3.7.6 虚拟机和物理机存储对性能的影响

本节介绍 UPRedis 由于虚拟机的存储对性能的影响
- 分别测试了 AOF 持久化 no、everysec、always 三种方式
- 关闭 aofrewrite
- 存储全部选用内置盘，暂无外置盘数据
- 物理机性能采用了绑定 CPU

- 结论:
  - no、everysec 方式性能比物理机低 15-20%
  - always 方式性能比物理机低 50%

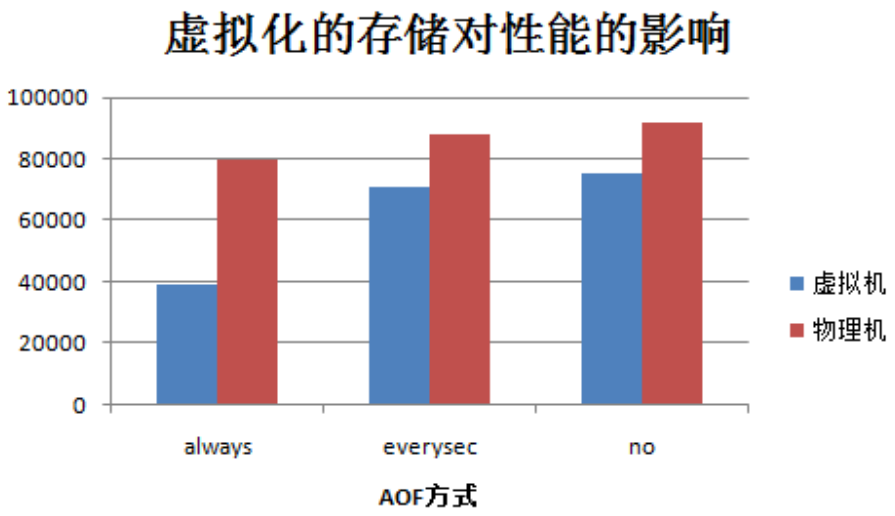


Figure 6: 虚拟化存储对性能的影响

3.8 UPRedis Proxy 性能

3.8.0.1 worker 数量对性能影响

	TPS	
worker num	set	get
1	50000	50000
2	90000	87000
4	180000	170000
8	280000	290000

- 结论:
- 在机器资源足够的条件下，UPRedis Proxy 的性能随着进程数量的增长，呈线性增长

3.8.0.2 读写分离对性能的影响

读写分离测试结果显示，在多个 slave 的情况下，UPRedis Proxy 的读性能大致与 slave 的数量成正比。也就是如果开启读写分离:

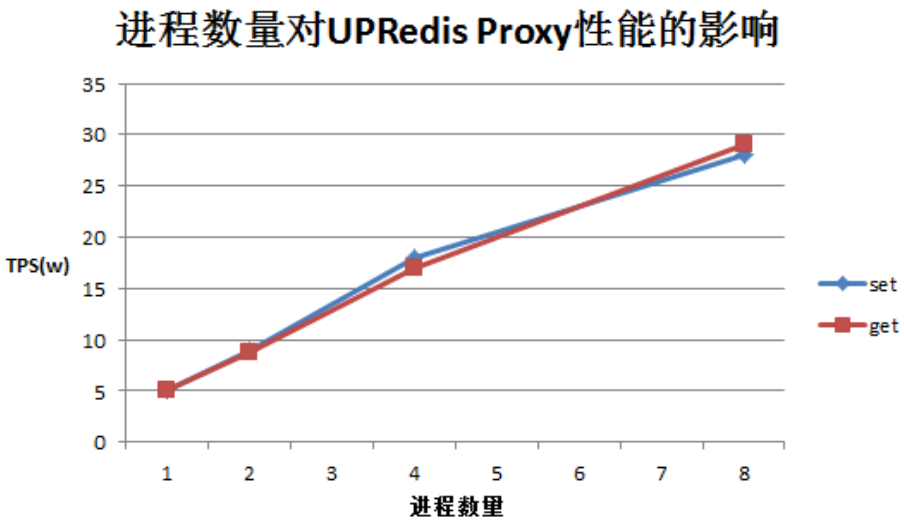


Figure 7: worker 数量对性能的影响

- 1 主 1 从，读取性能最高可到 100WQPS (50WQPS * 2)；
- 1 主 2 从，读取性能最高 150WQPS (50WQPS *3)。

3.9 性能优化建议

3.9.1 服务器性能优化

3.9.1.0.1 物理内存

- 保证物理内存充足，尽量不要使用 swap
- 设置 maxmemery，当达到最大内存时可以拒绝写

3.9.1.0.2 持久化方式

- rdb 和 aof(everysec) 并不会带来很大的性能损耗，应用可以根据情况选择使用
- aof(always) 有较大性能损耗，在对一致性要求较高的场景可以使用
- 可以选用高速存储来加速持久化过程

3.9.1.0.3 主从结构

- 从机过多会拖慢主机的速度，必要时可以选择级联复制

3.9.2 开发性能优化

3.9.2.0.1 大查询

- 原则上不允许类似于遍历所有 key 的大查询

3.9.2.0.2 pipeline

- 多次请求可以通过 pipeline 的方式合并成一次请求
- 另外采用 UPRedis Proxy 架构可以通过 Proxy 进行 Pipeline，增大 Redis 的性能极限

3.10 测试相关说明

3.10.1 测试环境

- 物理机

```
CPU:  Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz
二级缓存：20M Cache
内存：128G
磁盘：内置盘96G
网络：千兆网
主机：172.17.140.227(物理机，32C)
```

- 虚拟机

```
CPU:  Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz
二级缓存：20M Cache
内存：12G
磁盘：内置盘80G
网络：万兆网
主机：172.17.140.138/9(虚拟机，4C)
```

3.10.2 实例配置

默认配置

3.10.3 测试工具

redis 自带的性能测试工具 `redis-benchmark`

`memtier_benchmark` 是 Redis Labs 推出的一款命令行工具，它能够产生各种各样的流量模式，可以对 Memcached 和 Redis 实例进行基准测试。这个工具提供了丰富的自定义选项和报表功能，通过命令行界面就能够轻松地使用。

3.10.4 测试案例

```
./redis-benchmark -h 127.0.0.1 -p 22228 -n clients -t set,get -r
keyrange -d data
```

```
./memtier_benchmark -s 172.21.101.91 -p 44401 -n 15000000 -c 512 -t
32 --ratio 0:1 -a foobared -d 16 -R --key-maximum=10000
```

4 复制

4.1 介绍

为了防止数据丢失，upredis 提供复制功能，支持同步复制和异步复制，即 UPRedis 支持主从同步，数据可以从主服务器向任意数量的从服务器上同步，从服务器可以是关联其他从服务器的主服务器。

复制也是实现高可用性（HA）最常用的方法。

4.2 复制类型

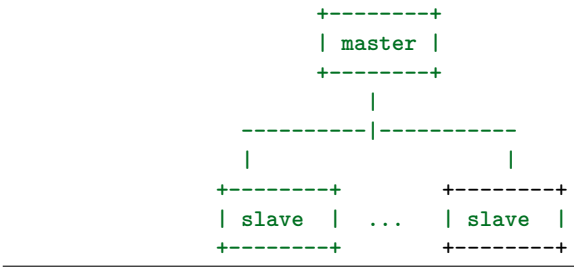
复制方式分为异步复制和同步复制，通过配置文件配置。

- 复制过程分为全量复制和增量复制
- 从服务器与主服务器第一次连接时，采用全量复制
- 如果从服务器中途断链，再次连接后，会根据缺失内容的大小 (repl-backlog-size 影响) 决定选择全量复制还是部分复制

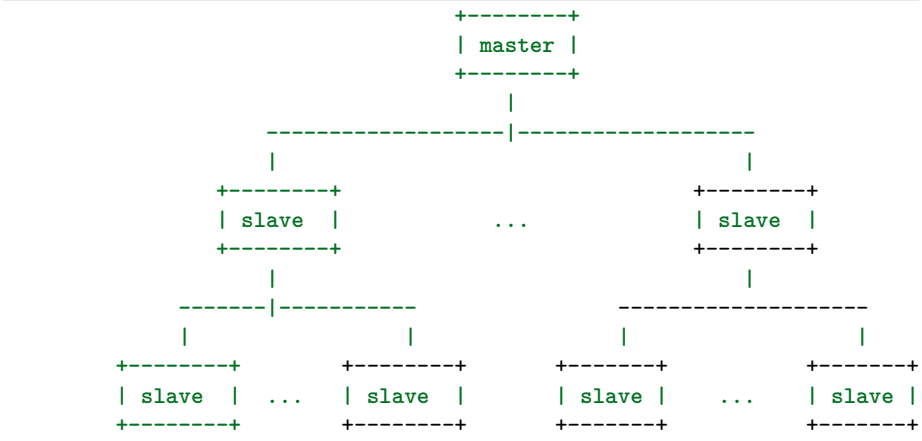
4.3 复制的拓扑结构

4.3.1 主备

4.3.2 标准的主从复制



4.3.3 级联复制



强烈不建议使用级联复制

4.4 配置项

```
# 主从复制，设置该数据库为其他数据库的从数据库。
# 设置当本机为 slave 服务时，设置 master 服务的 IP 地址及端口，在 Redis 启动时，
# 它会自动从 master 进行数据同步
slaveof <masterip> <masterport>

# 当 master 服务设置了密码保护时 (用 requirepass 制定的密码)
```

```
# slave 服务连接 master 的密码
masterauth <master-password>
masterauth_s <master-password-sha256>
dbpm <dbpm_info>

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

# slave 是否允许写操作
slave-read-only yes

# 如下俩选项均为无盘同步相关，建议使用默认选项
repl-diskless-sync no
repl-diskless-sync-delay 5

# 从库会按照一个时间间隔向主库发送 PINGs. 可以通过 repl-ping-slave-period
# 设置这个时间间隔，默认是 10 秒
repl-ping-slave-period 10

# repl-timeout 设置主库批量数据传输时间或者 ping 回复时间间隔，默认值是 60 秒
# 一定要确保 repl-timeout 大于 repl-ping-slave-period
repl-timeout 60

# 对于 master 与 slave 物理邻近且非第一次同步 (或第一次同步数据量较小不涉及数据迁移)
# 的情况下，关闭该选项
repl-disable-tcp-nodelay no

# master 复制存储临时数据大小. 该数值越大，则允许的 slave 断开时长越久.
# 该选项与增量同步策略相关
repl-backlog-size 100mb

# master 释放 backlog 时间，单位为秒
repl-backlog-ttl 3600

# redis-sentinel 根据该值决定主从切换时应提升哪个 slave，越低越好.
# 但 0 表示不适合担任 master
slave-priority 100

# master 在较少 slave 且延迟很高的情况下拒绝接收数据写入.
# 该种情况用于防止 slave 不够导致的数据备份不足.
# 默认不开启
#min-slaves-to-write 3
#min-slaves-max-lag 10

# 同步复制配置项
```

```
# synchronous <numslaves> <timeout> <desync>
# 第一个参数表示响应的 slave 的数量，需要小于等于实际配置的 slave 的数量，
# 如果设置数量等于实际配置的 slave 的数量，则需要所有的 slave 能够应答，才认为同步复制成功
# 如果设置数量小于实际配置的 slave 的数量，则则只需要有设置数量的 slave 能够应答，即可认为同步复制成功
# 第二个参数表示超时时间，第三个参数表示是否降级为异步复制
# 当能够应答的 slave 数量小于<numslaves>，并且 desync 设置为 1，则认为同步复制降级为异步复制
# 如果 desync 设置为 0，则不做降级，执行写命令会收到失败应答
# timeout，单位是毫秒，默认为 0，表示永不超时
# desync，取值 0 或 1，默认 0，表示不降级为异步复制
synchronous 1 500 1
```

4.5 读写分离与数据持久化

4.5.1 读写分离配置

slave 配置文件中默认 `slave-read-only` 选项开启为 yes，此时 slave 为只读模式。

如果 master 配置文件中开启了密码认证，则需要密码认证选项。

当多个 slave 同时连接到 master 时，master 只做一次 `bgsave`，并将生成的 rdb 文件发送给这些 slave，并随后将缓存的写操作同步到 slave 中。

4.5.2 数据持久化与一致性

在配置 master-slave 集群时，为了尽可能的保证数据备份的安全性与数据一致性 (UPRedis 的复制是异步的)，master 应考虑开启如下选项:

```
min-slaves-to-write 3
min-slaves-max-lag 10
```

这两个选项用于防止 slave 不够导致的数据备份不足。

> 对于 master-slave 模式，不允许出现 master 关闭持久化时再对 master 的 UPRedis 进程进行守护的行为，会导致严重的数据丢失: 因为新拉起来的 UPRedis 会没有任何数据，然后会被主从同步到从库，从而导致备份数据的丢失。

4.6 复制部署架构推荐

应用应评估自身数据量与实际部署环境内存的大小

1. 比如 500 万条平均数据大小为 1KB 的应用，尽管 UPRedis 会对其进行优化和压缩，仍应估计其占用内存不小于 5GB

2. 由于 UPRedis 同步与 Bgsave 所需要 fork 子进程的原因，单机内存 10GB 以内的部署环境就不适合该应用数据库

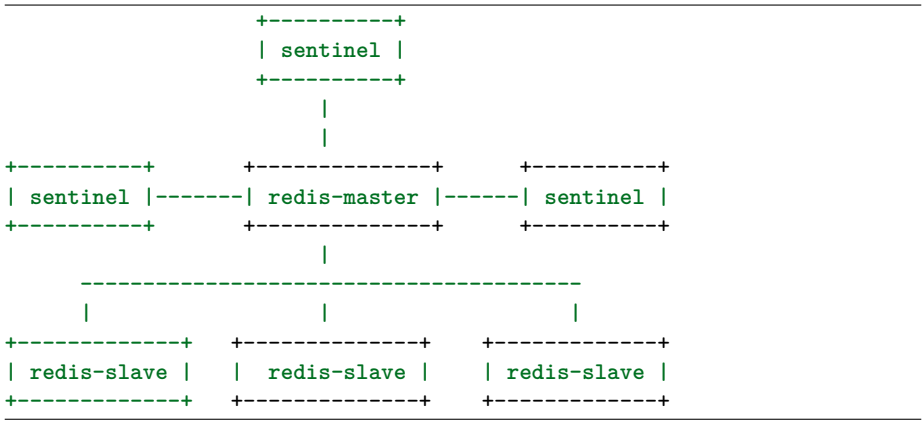
3. 如果出现数据量特别大的情形，则应考虑分片策略，部署多 master-slave 集群，分别存储部分数据

4.6.1 单 master-slave 集群模式

建议使用一主多从的模式，配置从服务器为只读模式
同时配置 3 台以上的 sentinel 对该集群进行监控

sentinel 相关内容请查阅《upredis-guide-功能》

示例拓扑图如下:

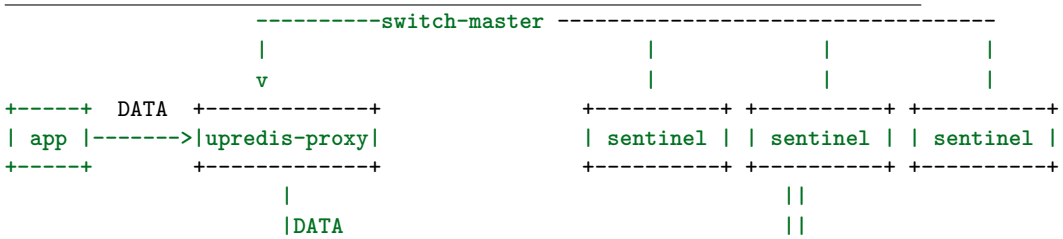


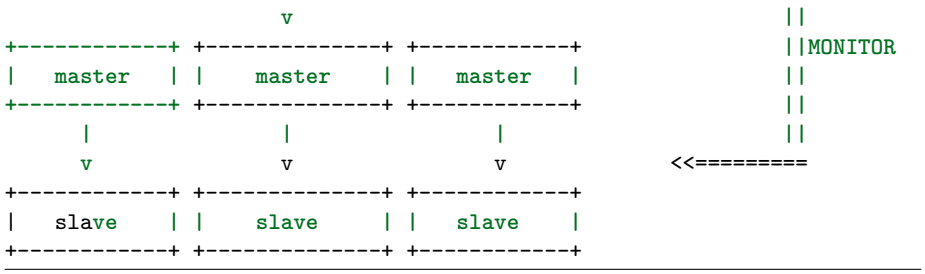
4.6.2 搭配 UPRedis Proxy 与配套高可用工具使用

UPRedis Proxy 是应用和 UPRedis 交互的中间件产品，用于解决分片、高可用、集群管理等问题。

UPRedis Proxy 相关内容请查阅 upredis-guide-功能文档。

示例拓扑图如下:





5 持久化

UPRedis 是一种内存数据库，由于内存中的数据时刻面临丢失的危险，UPRedis 提供了两种持久化机制，及时将内存中的数据写入存储。

- RDB
  在指定时间间隔内利用内存快照的方式把内存数据定期写入存储。
  RDB 保存某个时间点的数据集，若 UPRedis 意外停止工作会丢失从上次快照开始新写入数据；
- AOF
  利用写日志的方式，每执行一批更改 UPRedis 数据的命令，就在日志里附加上该命令并保存在存储上

  UPRedis 默认关闭 RDB 模式，关闭 AOF 模式
  如果对数据安全要求极高，无法承担任何数据丢失的后果，AOF 模式就变成了持久化的首选
- 推荐配置
  - 在压力小的情况下建议都打开，充分保证数据的完整性 (官方建议)
  - 在压力大的情况下建议都关闭，至少 RDB 需要关闭
  - 在缓存场景下，对数据完整性要求低，建议都关闭
  - 在存储场景下，对数据完整性要求高，建议都打开
  - 建议打开主从复制，部分持久化操作转移到从库执行
  - AOF 方式建议选择默认的 everysec，对数据完整性要求极高，可以选用 always，但是要注意性能会下降 30%
- AOF-BINLOGUPRedis-2.0 使用了 AOF-BINLOG 特性替换了 AOF 特性，并且异地数据同步依赖于 AOF-BINLOG 特性。如果需要使用异地数据同步功能，则必须主库开启 AOF-BINLOG。

AOF-BINLOG 存储格式与 AOF 相同，但是存储了时间戳、opid、server-id 等额外信息。

5.1 RDB

5.1.1 介绍

将 UPRedis 内存的一个快照保存到一个 RDB 文件中，以实现数据恢复

5.1.2 配置

```
/*
 * 指出在多长时间内，有多少次更新操作，就将数据同步到数据文件 rdb。相当于条件触发
 * 抓取快照，这个可以多个条件配合
 * 比如默认配置文件中的设置，就设置了三个条件：
 * save 900 1  900 秒内至少有 1 个 key 被改变
 * save 300 10  300 秒内至少有 300 个 key 被改变
 * save 60 10000  60 秒内至少有 10000 个 key 被改变
 */
save <second> <changes>

/* bgsave 失败时是否拒绝写入以保证数据一致性，建议开启 */
stop-writes-on-bgsave-error yes

/* 存储至本地数据库时（持久化到 rdb 文件）是否压缩数据，默认为 yes */
rdbcompression yes

/* rdb 文件是否使用 checksum 方式来检验完整性 */
rdbchecksum yes

/* rdb 文件名称 */
dbfilename dump.rdb-3301

/*
 * 工作目录
 * 数据库镜像备份的文件放置的路径。
 * 这里的路径跟文件名要分开配置是因为 redis 在进行备份时，
 * 先会将当前数据库的状态写入到一个临时文件中，等备份完成时，
 * 再把该临时文件替换为上面所指定的文件，
 * 而这里的临时文件和上面所配置的备份文件都会放在这个指定的路径当中。
 *
 * AOF 文件也会存放在这个目录下面
 *
 * 注意这里必须指定一个目录而不是文件
 */
dir ./
```

5.1.3 命令

- 同步保存

```
$ SAVE
```

会阻塞服务器, 不建议使用

- 异步保存

```
$ BGSAVE
```

5.1.4 持久化性能

5.1.4.1 性能测试环境

```
CPU:  Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz
二级缓存：20M Cache
内存：128G
磁盘：内置盘96G
网络：千兆网，loop
主机：172.21.101.91(物理机，32C)
```

5.1.4.2 数据

SAVE 和 BGSAVE 每保存 1GB 数据耗时情况:

```
+--------+---------+-----------+
| memory | SAVE(s) | BGSAVE(s) |
+--------+---------+-----------+
| 1GB    | 3.10    | 3.42      |
+--------+---------+-----------+
```

5.1.5 数据恢复性能

每装载 1GB 内存数据，AOF 和 RDB 持久化的耗时情况

```
+--------+---------+-----------+
| data   | AOF(s)  | RDB(s)    |
+--------+---------+-----------+
| 1GB    | 4.30    | 5.40      |
+--------+---------+-----------+
```

5.1.6 优缺点

5.1.6.1 优点

1. RDB 是一个非常紧凑（compact）的文件，它保存了 UPRedis 在某个时间点上的数据集
2. RDB 可以最大化 UPRedis 的性能
3. RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快 (官方说法，与测试数据不符)

5.1.6.2 缺点

1. 一旦发生故障停机，你就可能会丢失好几分钟的数据
2. 如果需要保存的数据集比较大，可能造成一段时间的服务不可用

5.2 AOF

AOF 适用于 UPRedis-1.X 版本，UPRedis-2.0 采用了 AOF-BINLOG 替换了 AOF 特性。

5.2.1 介绍

将 UPRedis 所有的写命令追加到 AOF 文件中，以实现数据恢复

5.2.2 配置

```
# 默认情况下，redis 会在后台异步的把数据库镜像备份到磁盘，
# 但是该备份是非常耗时的，而且备份也不能很频繁，如果发生诸如断电、拔插头等状
# 况，那么将造成比较大范围的数据丢失。
# 所以 redis 提供了另外一种更加高效的数据库备份及灾难恢复方式。
# 开启 append only 模式之后，redis 会把所接收到的每一次写操作请求都追加到
# appendonly.aof 文件中，当 redis 重新启动时，会从该文件恢复出之前的状态。
# 但是这样会造成 appendonly.aof 文件过大，所以 redis 还支持了 BGREWRITEAOF 指令，
# 对 appendonly.aof 进行重新整理。
# 可以同时开启 asynchronous dumps 和 AOF
appendonly yes

# AOF 文件名称 (默认: "appendonly.aof")
appendfilename appendonly.aof-3301

# Redis 支持三种同步 AOF 文件的策略:
# no: 不进行同步，系统去操作 . Faster.
# always: always 表示每次有写操作都进行同步. Slow, Safest.
# everysec: 表示对写操作进行累积，每秒同步一次. Compromise.
```

```
# 默认是"everysec"，按照速度和安全折中这是最好的。
# 如果想让 Redis 能更高效的运行，你也可以设置为"no"，让操作系统决定什么时候去执行
# 或者相反想让数据更安全你也可以设置为"always"
# 如果不确定就用 "everysec".
appendfsync everysec

# AOF 策略设置为 always 或者 everysec 时，后台处理进程 (后台保存或者 AOF 日志重写)
# 会执行大量的 I/O 操作
# 在某些 Linux 配置中会阻止过长的 fsync() 请求。注意现在没有任何修复，
# 即使 fsync 在另外一个线程进行处理
# 为了减缓这个问题，可以设置下面这个参数 no-appendfsync-on-rewrite
no-appendfsync-on-rewrite no

# AOF 自动重写
# 当 AOF 文件增长到一定大小的时候 Redis 能够调用 BGREWRITEAOF 对日志文件进行重写
# 它是这样工作的：Redis 会记住上次进行些日志后文件的大小 (如果从开机以来还没进行过
# 重写，那日子大小在开机的时候确定)
# 基础大小会同现在的大小进行比较。如果现在的大小比基础大小大制定的百分比，重写功能
# 将启动
# 同时需要指定一个最小大小用于 AOF 重写，这个用于阻止即使文件很小但是增长幅度很大也
# 去重写 AOF 文件的情况
# 设置 percentage 为 0 就关闭这个特性
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb

# aof 文件不完整时，upredis 是否允许载入被截断的数据
aof-load-truncated yes
```

5.2.3 命令

5.2.3.1 文件重写

随着时间的推移，AOF 文件占用空间会越来越大;
为了减少 AOF 文件的磁盘空间占用，需要 AOF 重写

5.2.3.2 redis 主动 AOF 重写:

```
# aof 重写限制
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
```

- 只有当 AOF 文件大小超过 `auto-aof-rewrite-min-size` 才会触发重写
- 当 AOF 文件大小超过上次重写后的文件大小的 100% 时会触发重写

5.2.3.3 手工 AOF 重写:

```
$ BGREWRITEAOF
```

5.2.4 持久化性能

5.2.4.1 性能测试环境

以之前测试环境一致

5.2.4.2 数据

BGRWRITEAOF 每重写 1GB 内存数据耗时情况:

```
+--------+----------------+
| memory | BGRWRITEAOF(s) |
+--------+----------------+
| 1GB    | 5.486          |
+--------+----------------+
```

5.2.5 数据恢复性能

参考 RDB 装载性能数据结果

5.2.6 优缺点

5.2.6.1 优点

1. 使用 AOF 持久化会让 UPRedis 变得非常耐久（much more durable）
2. AOF 文件是一个只进行追加操作的日志文件
3. UPRedis 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
4. AOF 文件分析很轻松，因为 UPRedis 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 UPRRedis 协议的格式保存

5.2.6.2 缺点

1. 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
2. 根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB

5.3 AOF-BINLOG

5.3.1 持久化-AOF-BINLOG

5.3.1.1 介绍

UPRedis-2.0 采用 AOF-BINLOG 替换了官方的 AOF 机制，以实现数据异地同步和复制优化。

与 AOF 机制类似，所有的写命令追加到 AOF-BINLOG 文件中，以实现数据恢复

5.3.1.2 配置项

```
# UPRedis-2.0 采用 AOF-BINLOG 替换了 AOF 机制，但是保留了 appendonly, appendfsync 两个配置选项。
#
# 默认情况下，redis 会在后台异步的把数据库镜像备份到磁盘，但是该备份是非常耗时的，而且备份也不能很频繁，如果发生诸如拉
# 所以 redis 提供了另外一种更加高效的数据库备份及灾难恢复方式。
# 开启 AOF-BINLOG 模式之后，redis 会把所接收到的每一次写操作请求都追加到 appendonly-inc-<timestamp>.aof 文件中
# 但是这样会造成 appendonly.aof 文件过大，所以 AOF-BINLOG 还支持通过 BGSAVE 指令，将当前 redis 数据归纳到 rdb 文
# NOTE: AOF-BINLOG 机制使用 BGSAVE(而不是 BGREWRITEAOF) 命令整理 aof 文件，BGREWRITEAOF 仍然可用于将当前 red
# 可以同时开启 RDB 和 AOF-BINLOG。
appendonly no

# AOF 文件名称格式为: "appendonly-inc-<timestamp>.aof"

# Redis 支持三种同步 AOF-BINLOG 文件的策略:
# no: 不进行同步，系统去操作 . Faster.
# always: always 表示每次有写操作都进行同步. Slow, Safest.
# everysec: 表示对写操作进行累积，每秒同步一次. Compromise.
# 默认是"everysec"，按照速度和安全折中这是最好的。
# 如果想让 Redis 能更高效的运行，你也可以设置为"no"，让操作系统决定什么时候去执行
# 或者相反想让数据更安全你也可以设置为"always"
# 如果不确定就用 "everysec".
appendfsync everysec

# AOF-BINLOG 策略设置为 always 或者 everysec 时，后台处理进程 (后台保存或者 AOF-BINLOG 日志重写) 会执行大量的 I
# 在某些 Linux 配置中会阻止过长的 fsync() 请求。注意现在没有任何修复，即使 fsync 在另外一个线程进行处理
# 为了减缓这个问题，可以设置下面这个参数:
rdb-save-incremental-fsync yes

# server-id 是一个 redis 分片（包括 Master 和 Slave）的标识，表示了 AOF-BINLOG 每个命令产生的源头。
# 需要注意：同一个分片的 Master 和 Slave 必须使用相同的 server-id，不同分片的 Master，Slave 必须
# 使用不同的 server-id。server-id 使用数字格式，默认的 server-id 是 12345678。
server-id 12345678

# aof-psync 特性：aof-psync-state 用于开启 aof-psync 特性，当 PSYNC 不成功时，redis 将会尝试使用 AOF-PSYNC 复
# 从 AOF-BINLOG 中查找命令历史。因此复制的优先级为：PSYNC > AOF-PSYNC > FULLRESYNC。
# aof-psync-state 的默认值为 yes。
aof-psync-state yes
```

```
# 自动 AOF-BINLOG 切换: aof-max-sisze 定义 AOF-BINLOG 文件的最大大小，如果当前使用的 AOF-BINLOG 文件大小超过 a
# UPRedis 将切换到下一个 AOF-BINLOG 文件。
aof-max-size 128mb

# 自动 AOF-BINLOG 文件清除：当开启 auto-purge-aof 之后，如果 AOF-BINLOG 文件的总量超过 aof-max-keep-size，
# UPRedis 将自动清除最老的 AOF-BINLOG 文件。默认 aof-max-keep-size 为 5GB。
auto-purge-aof yes
aof-max-keep-size 5gb

# AOF-BINLOG 自动重写 -- cron-bgsave
# 当 AOF-BINLOG 文件增长到一定大小的时候 Redis 能够调用 BGSAVE 对日志文件进行重写
# 它是这样工作的：Redis 会记住上次进行些日志后文件的大小 (如果从开机以来还没进行过重写，那日子大小在开机的时候确定)
# 基础大小会同现在的大小进行比较。如果现在的大小比基础大小大指定的百分比，重写功能将启动
# 同时需要指定一个最小大小用于 AOF 重写，这个用于阻止即使文件很小但是增长幅度很大也去重写 AOF 文件的情况
auto-cron-bgsave yes
cron-bgsave-rewrite-percentage 100
cron-bgsave-rewrite-min-size 512mb
```

5.3.1.3 AOF-BINLOG 操作

通过 UPRedis 接口可以完成 AOF-BINLOG 切换、重写、清除等功能。

5.3.1.3.1 AOF-BINLOG 文件切换

为了方便从 AOF-BINLOG 文件中快速地找到对应的日志，AOF-BINLOG 日志文件的大小必须保持在一个比较合理的范围。也就是超过特定大小之后，AOF-BINLOG 文件应该切换到下一个。切换的方式目前包括手动和自动两种:

a) 手工切换

```
# 手动切换 AOF-BINLOG 文件，即使当前 AOF—BINLOG 文件大小没有超过 aof-max-size
aofflush
```

b) 自动切换

```
# 当前 AOF-BINLOG 超过 aof-max-size 大小将切换到新的 AOF-BINLOG 文件
aof-max-size 128mb
```

5.3.1.3.2 AOF-BINLOG 文件重写

随着时间的推移，AOF-BINLOG 文件占用空间会越来越大; 为了减少 AOF-BINLOG 文件的磁盘空间占用，需要 AOF-BINLOG 重写

a) 手工重写

```
# 在打开 AOF-BINLOG 情况下执行 BGSAVE，UPRedis 把当前全量数据快照存储在 dump.rdb 文件
# 中，并重新生成 rdb.index 文件（保存了 dump.rdb 与 AOF-BINLOG 的对应关系）。
BGSAVE
```

b) 自动重写

```
# AOF-BINLOG 重写条件：开启 auto-cron-bgsave 情况下，AOF-BINLOG 文件距离上一次重写
# 增长超过 100%（2 倍）且总量大于 512mb 触发自动重写。
auto-cron-bgsave yes
cron-bgsave-rewrite-percentage 100
cron-bgsave-rewrite-min-size 512mb
```

AOF-BINLOG 重写存在和 RDB 文件生成一样的造成内存和存储冲高的风险

5.3.1.3.3 AOF-BINLOG 文件清除

a) 手工清除

```
# 清除比<aof-binlog-filename> 旧的 AOF-BINLOG 文件。
purgeaofto <aof-binlog-filename>
```

b) 自动清除

```
# 如果当前 AOF-BINLOG 总量超过 aof-max-keep-size，则清除旧的 AOF-BINLOG
auto-purge-aof yes
aof-max-keep-size 5gb
```

purgeaofto 命令有时执行不成功，这通常是因为当前被 purge 的 AOF-BINLOG 被没有被保存到 dump.rdb 文件中，只要执行 SAVE(或者 BGSAVE) 就可以将 AOF-BINLOG 手动 purge 掉了。

5.3.1.4 数据保存性能

5.3.1.4.1 测试环境

与之前测试环境一致

5.3.1.4.2 测试结果:

BGSAVE 保存 1GB 数据耗时情况:

```
+--------+---------+-----------+
| memory | SAVE(s) | BGSAVE(s) |
```

```
+--------+---------+-----------+
| 1GB    | 3.10    | 3.42      |
+--------+---------+-----------+
```

5.3.1.5 AOF-BINLOG 的优缺点

5.3.1.5.1 优点

1. 使用 AOF-BINLOG 持久化会让 UPRedis 变得非常耐久(much more durable)
2. AOF-BINLOG 文件是一个只进行追加操作的日志文件
3. UPRedis 可以在 AOF-BINLOG 文件体积变得过大时，自动地在后台对 AOF-BINOOG 进行重写
4. 开启 AOF-BINLOG, UPRedis 可以支持异地数据同步和 AOF-PSYNC 复制

5.3.1.5.2 缺点

1. 对于相同的数据集来说，AOF-BINLOG 文件的体积通常要大于 RDB 文件的体积
2. 根据所使用的 fsync 策略，AOF-BINLOG 的速度可能会慢于 RDB

5.3.1.6 AOF-BINLOG 使用注意

异地数据同步依赖于 AOF-BINLOG

master 必须开启 AOF-BINLOG(appendonly yes)，master 和 slave 才能产生 AOF-BINLOG 文件

一个分片中的 master 和 slave 必须 server-id 相同，否则无法建立复制

因为 AOF-BINLOG 特性包含的 opget 命令从 aof-binlog 文件中读取命令日志，为了防止老旧数据的影响 slave 启动之前或者 master 回切之前需要清理掉之前的 aof-binlog 文件，以免造成新老数据 opid 不连续。

6 安全

UPRedis 没有用户/账号的概念，只有一个轻量级的 AUTH 密码认证。
UPRedis Proxy 除了密码认证，还增加了前端黑白名单控制的功能。

6.1 UPRedis 端

6.1.1 认证

6.1.1.1 配置项

```
requirepass foobared
或者
requirepass_s 1b58ee375b42e41f0e48ef2ff27d10a5b1f6924a9acdcdba7cae868e7adce6bf
或者
dbpm "failover://127.0.0.1:12345:5000:db:usr,127.0.0.1:12346:5000:db:usr"
```

6.1.1.2 使用

使用客户端连接服务器时，输入

```
$ AUTH foobared
或者
$ AUTH_S 1b58ee375b42e41f0e48ef2ff27d10a5b1f6924a9acdcdba7cae868e7adce6bf
```

> 注意：主库也要配置 masterauth 或者 masterauth_s 或者 dbpm 属性
> 以备主备切换

6.1.1.3 说明

开启该选项时, 仍可以连接到 UPRedis, 但需要使用 auth/auth_s 命令或在命令行指定-a 参数才能进一步的操作数据;

密码的设置有三种方式:

- 明文密码：requirepass foobared
- 密文密码:requirepass_s 1b58ee375b42e41f0e48ef2ff27d10a5b1f6924a9acdcdba7cae868e7adce6bf
- 访问 DBPM 获取密码:dbpm “failover://127.0.0.1:12345:5000:db:usr,127.0.0.1:12346:5000:db:usr”

主从复制密码设置有三种方式:

- 明文密码：masterauth foobared
- 密文密码:masterauth_s 1b58ee375b42e41f0e48ef2ff27d10a5b1f6924a9acdcdba7cae868e7adce6bf
- 访问 DBPM 获取密码:dbpm “failover://127.0.0.1:12345:5000:db:usr,127.0.0.1:12346:5000:db:usr”

其他说明:

- 密码优先级：密文 > 明文 >DBPM
- 密文密码采用 sha256 加密，获取方式为：“echo -n foobared | sha256sum”
- 不建议采用明文密码
- DBPM 配置支持最多两台，格式为: failover://ip1:port1:timeout1:db1:usr1,...
- 如果采用了 DBPM，那么只用配置一次就可以，主库密码和主从复制密码都从 DBPM 获取
- sentinel 的密码配置目前仍然是明文密码配置在配置文件中

6.1.2 连接数控制

6.1.2.1 配置项

```
maxclients 50000
```

6.1.2.2 说明

UPRedis 控制最大连接数，默认最大 50000 个连接；

6.1.3 重命名

为防止一些恶意或疏漏的开发行为，如 `config set` 与 `flushall` 等操作，可以通过在 redis-server 端执行如下命令来杜绝：

```
redis-cli rename-command CONFIG ""
redis-cli rename-command flushall ""
```

也需要合理规划重命名，保留一定的可管理型

6.2 UPRedis Proxy 端

6.2.1 认证

6.2.1.1 配置项

```
redis_auth: foobared
或者
redis_auth_s: 1b58ee375b42e41f0e48ef2ff27d10a5b1f6924a9acdcdba7cae868e7adce6bf
或者
dbpms:
 - IP1:PORT1:db1:usr1
 - IP2:PORT2:db2:usr2
```

6.2.1.2 使用

使用客户端连接服务器时，输入

```
$ AUTH foobared
或者
$ AUTH_S 1b58ee375b42e41f0e48ef2ff27d10a5b1f6924a9acdcdba7cae868e7adce6bf
```

6.2.1.3 说明

密码的设置有三种方式:

- 明文密码：redis_auth: foobared
- 密文密码:redis_auth_s 1b58ee375b42e41f0e48ef2ff27d10a5b1f6924a9acdcdba7cae868e7adce6bf
- 访问 DBPM 获取密码

其他说明:

- 密码优先级：密文 > 明文 >DBPM
- 密文密码采用 sha256 加密，获取方式为：“echo -n foobared | sha256sum”
- 不建议采用明文密码
- DBPM 配置支持最多两台，格式为:

```
dbpms:
 - IP1:PORT1:db1:usr1
 - IP2:PORT2:db2:usr2
```

6.2.1.4 其他说明

- UPRedis Proxy 如果连接多个 UPRedis,则所有 UPRedis 的密码需要保持一致;
- 客户端对 UPRedis Proxy 的认证和 UPRedis Proxy 对 UPRedis 的认证是分离的，独立的;
- **UPRedis Proxy** 和 **UPRedis** 密码必须配置一致
- 管理端口没有密码，无需认证

6.2.2 黑白名单

6.2.2.1 配置项

```
white_list:
- '*.*.*.*'
black_list:
- 172.17.100.100
- 172.17.200.*
```

6.2.2.2 使用

控制访问用户的 IP

支持通配,但是通配符后面不允许再次出现非通配信息,例如:`172.17.1.*1`

6.2.2.3 说明

- IP 在白名单中且不在黑名单中才可以访问
- 什么都不配置则表示全部通过
- 单配白名单则只有在白名单中才可以通过
- 单配黑名单则只有不在黑名单中才可以通过
- 如果配置了黑名单或者白名单，则所有 ipv6 地址不能通过
- 所有通过 unix socket 的连接都可以通过

6.2.3 连接数控制

6.2.3.1 配置项

```
client_connections: 50000
```

6.2.3.2 说明

UPRedis Proxy 连接数控制目前没有生效；

6.2.4 流控

UPRedis Proxy 目前不支持流量控制 (TPS 或者网络流量)

6.3 客户端

客户端的安全控制详见 请查看《**upredis-guide-**开发》

7 管理

UPRedis 的管理分为三方面，系统级需求与日常管理命令以及工具

7.1 系统级管理需求

- 配置进程使用内存
  `sysctl vm.overcommit_memory=1`

- 禁用大内存页面
  /sys/kernel/mm/transparent_hugepage/enabled
- 查看/procs/swaps 确保 swap 文件大小
- 如果需要限制 UPRedis 服务使用内存，请在 redis.conf 中显式的配置 maxmemory
- 评估数据量大小，确保系统可用内存至少为数据量大小的 2 倍
  当数据大小超出这个限制时，应考虑使用分片策略
- 开启复制时，请确保 master 不会出现持久化策略关闭 + 自动重启的情形，这样会导致严重的数据丢失

7.2 日常管理命令-UPRedis

UPRedis 的日常管理依赖于 redis-cli 命令行工具。
当前 redis-cli 命令行工具的参数选项请使用--help 参数查看

7.2.1 info 命令

通过在客户端发送

```
$ INFO [section]
```

获取服务器全部或部分信息。

信息类型可以分为以下 8 类:

7.2.1.1 server

记录了 UPRedis 服务器的信息

```
redis_version : UPRedis 服务器版本-Redis 版本
upredis_version:1.3.0  UPRedis 服务器版本-UP 版本
redis_git_sha1 : Git SHA1
redis_git_dirty : Git dirty flag
os : UPRedis 服务器的宿主操作系统
arch_bits : 架构（32 或 64 位）
multiplexing_api : UPRedis 所使用的事件处理机制
gcc_version : 编译 UPRedis 时所使用的 GCC 版本
process_id : 服务器进程的 PID
run_id : UPRedis 服务器的随机标识符（用于 Sentinel 和集群）
tcp_port : TCP/IP 监听端口
uptime_in_seconds : 自 UPRedis 服务器启动以来，经过的秒数
uptime_in_days : 自 UPRedis 服务器启动以来，经过的天数
lru_clock : 以分钟为单位进行自增的时钟，用于 LRU 管理
```

7.2.1.2 clients

记录了已连接客户端的信息

---

connected_clients : 已连接客户端的数量（不包括通过从属服务器连接的客户端）
client_longest_output_list : 当前连接的客户端当中，最长的输出列表
client_longest_input_buf : 当前连接的客户端当中，最大输入缓存
blocked_clients : 正在等待阻塞命令（BLPOP、BRPOP、BRPOPLPUSH）的客户端的数量

7.2.1.3 memory

记录了服务器的内存信息

---

used_memory : 由 UPRedis 分配器分配的内存总量，以字节（byte）为单位
used_memory_human : 以人类可读的格式返回 UPRedis 分配的内存总量
used_memory_rss : 从操作系统的角度，返回 UPRedis 已分配的内存总量
（俗称常驻集大小）。这个值和 top 、 ps 等命令的输出一致。
used_memory_peak : UPRedis 的内存消耗峰值（以字节为单位）
used_memory_peak_human : 以人类可读的格式返回 UPRedis 的内存消耗峰值
used_memory_lua : Lua 引擎所使用的内存大小（以字节为单位）
mem_fragmentation_ratio : used_memory_rss 和 used_memory 之间的比率
mem_allocator : 在编译时指定的， UPRedis 所使用的内存分配器。
可以是 libc 、 jemalloc 或者 tcmalloc

7.2.1.4 persistence

记录了跟 RDB 持久化和 AOF 持久化有关的信息

---

loading : 一个标志值，记录了服务器是否正在载入持久化文件。
rdb_changes_since_last_save : 距离最近一次成功创建持久化文件之后，经过了多少秒。
rdb_bgsave_in_progress : 一个标志值，记录了服务器是否正在创建 RDB 文件。
rdb_last_save_time : 最近一次成功创建 RDB 文件的 UNIX 时间戳。
rdb_last_bgsave_status : 一个标志值，记录了最近一次创建 RDB 文件的结果是成功还是失败。
rdb_last_bgsave_time_sec : 记录了最近一次创建 RDB 文件耗费的秒数。
rdb_current_bgsave_time_sec : 如果服务器正在创建 RDB 文件，
那么这个域记录的就是当前的创建操作已经耗费的秒数。
aof_enabled : 一个标志值，记录了 AOF 是否处于打开状态。
aof_rewrite_in_progress : 一个标志值，记录了服务器是否正在创建 AOF 文件。
aof_rewrite_scheduled : 一个标志值，记录了在 RDB 文件创建完毕之后，
是否需要执行预约的 AOF 重写操作。
aof_last_rewrite_time_sec : 最近一次创建 AOF 文件耗费的时长。
aof_current_rewrite_time_sec : 如果服务器正在创建 AOF 文件，
那么这个域记录的就是当前的创建操作已经耗费的秒数。
aof_last_bgrewrite_status : 一个标志值，记录了最近一次创建
AOF 文件的结果是成功还是失败。

如果 AOF 持久化功能处于开启状态，那么这个部分还会加上以下域：

aof_current_size : AOF 文件目前的大小。
aof_base_size : 服务器启动时或者 AOF 重写最近一次执行之后，AOF 文件的大小。
aof_pending_rewrite : 一个标志值，记录了是否有 AOF
        重写操作在等待 RDB 文件创建完毕之后执行。
aof_buffer_length : AOF 缓冲区的大小。
aof_rewrite_buffer_length : AOF 重写缓冲区的大小。
aof_pending_bio_fsync : 后台 I/O 队列里面，等待执行的 fsync 调用数量。
aof_delayed_fsync : 被延迟的 fsync 调用数量。

7.2.1.5 stats

记录了一般统计信息

total_connections_received : 服务器已接受的连接请求数量。
total_commands_processed : 服务器已执行的命令数量。
instantaneous_ops_per_sec : 服务器每秒钟执行的命令数量。
rejected_connections : 因为最大客户端数量限制而被拒绝的连接请求数量。
expired_keys : 因为过期而被自动删除的数据库键数量。
evicted_keys : 因为最大内存容量限制而被驱逐（evict）的键数量。
keyspace_hits : 查找数据库键成功的次数。
keyspace_misses : 查找数据库键失败的次数。
pubsub_channels : 目前被订阅的频道数量。
pubsub_patterns : 目前被订阅的模式数量。
latest_fork_usec : 最近一次 fork() 操作耗费的毫秒数。

7.2.1.6 replication

记录了主/从复制信息

role : 如果当前服务器没有在复制任何其他服务器，那么这个域的值就是 master；
        否则的话，这个域的值就是 slave 。注意，在创建复制链的时候，
        一个从服务器也可能是另一个服务器的主服务器。

如果当前服务器是一个从服务器的话，那么这个部分还会加上以下域：

master_host : 主服务器的 IP 地址。
master_port : 主服务器的 TCP 监听端口号。
master_link_status : 复制连接当前的状态，up 表示连接正常，down 表示连接断开。
master_last_io_seconds_ago : 距离最近一次与主服务器进行通信已经过去了多少秒钟。
master_sync_in_progress : 一个标志值，记录了主服务器是否正在与这个从服务器进行同步。

如果同步操作正在进行，那么这个部分还会加上以下域：

master_sync_left_bytes : 距离同步完成还缺少多少字节数据。
master_sync_last_io_seconds_ago : 距离最近一次因为 SYNC 操作而进行 I/O 已经过去了多少秒。

如果主从服务器之间的连接处于断线状态，那么这个部分还会加上以下域：

master_link_down_since_seconds ： 主从服务器连接断开了多少秒。

以下是一些总会出现的域：

connected_slaves ： 已连接的从服务器数量。

对于每个从服务器，都会添加以下一行信息：

slaveXXX ： ID、IP 地址、端口号、连接状态

7.2.1.7 cpu

记录了 CPU 的计算量统计信息

```
used_cpu_sys : UPRedis 服务器耗费的系统 CPU
used_cpu_user : UPRedis 服务器耗费的用户 CPU
used_cpu_sys_children : 后台进程耗费的系统 CPU
used_cpu_user_children : 后台进程耗费的用户 CPU
```

7.2.1.8 keyspace

记录了数据库相关的统计信息

```
dbXXX:keys=XXX,expires=XXX,avg_ttl=XXX
```

7.2.2 常用管理命令

7.2.2.1 连接管理

查看当前连接

```
$ client list
```

关闭某一连接

```
$ client kill ip:port
```

> 由于 UPRedis 是单线程处理，如果该连接正在执行命令，则会等待命令执行完之后才会关闭

如针对所有客户端连接、slave 连接或 pubsub 连接等，可以使用如下命令关闭连接：

```
$ client kill type normal
$ client kill type slave
$ client kill type pubsub
```

连接默认是无名的，但在部分使用场景需要使用连接名称，如使用 UPRedis 构建消息队列时，可以根据连接负责的任务，为生产者和消费者分别设置不同的名字。使用如下方式命名连接:

```
$ client setname xxxxxx
```

同时可以使用如下方式获取连接命名:

```
$ client getname
```

7.2.2.2 数据备份

如果想要以 rdb 文件方式在本地进行数据备份:

```
$ bgsave
```

随着时间的推移，AOF 文件占用空间会越来越大;
为了减少 AOF 文件的磁盘空间占用，需要 AOF 重写
如果想要进行 AOF 重写:

```
$ bgrewriteaof
```

redis-cli 使用–rdb 参数来远程备份 redis-server 数据, 该种方式类似于 slave 到 master 的 SYNC 操作。

```
redis-cli --rdb /path/backup.rdb
```

这种备份并不保证一定成功，如果使用 cron job 的方式来执行数据备份，建议脚本进行返回值检查。

7.2.2.3 配置参数动态生效

UPRedis 使用如下命令来修改当前已经运行的 redis-server

```
$ config set parameter value
```

并使用如下方式刷新到配置文件

```
$ config rewrite
```

需要重置 info stat 中某些统计数据时：

```
$ config resetstat
```

7.2.2.4 数据清理

UPRedis 允许针对某个 db 执行数据清理操作：

```
$ select 1 # 切换到数据库 1
$ flushdb  # 清理数据
```

也可以清理当前 redis-server 不同 db 下的所有数据：

```
$ flushall
```

注：该操作应在数据备份后再执行，此外，应注意 select db 的返回值结果。

7.2.2.5 LRU 模拟与部署参数估定

UPRedis 在很多应用场景中被部署为一个 LRU 缓存，在这种情况下需要估计缓存的命中率，redis-cli 提供了`--lru-test` 参数来对命中率进行测试。

```
redis-cli --lru-test 1000000
```

1000000 即为预分配 LRU 缓存大小。

7.2.3 监控命令

7.2.3.1 键值空间统计

UPRedis 可以使用如下来获取当前内存中键值数：

```
$ dbsize
```

或

```
$ info keyspace
```

后者还提供了过期数据数与平均 ttl 等信息。

7.2.3.2 进程数据收集

查看当前进程使用内存、连接数、key 值数等信息

```
redis-cli --stat -i 1
```

以上为每秒一次将统计数据打印到标准输出，可以重定向到文件中，也可以使用-i 参数指定统计时间间隔 (单位为秒)。
典型的示例如下：

```
------- data ------ --------------------- load -------------------- - child -
keys       mem      clients blocked requests            connections
6          2.07M    3       0       83 (+0)             17
```

7.2.3.3 检查当前键值大小

查看当前 UPRedis 进程内存中最大 value 大小。

```
redis-cli --bigkeys
```

7.2.3.4 寻找模式匹配键值

针对寻找符合某个模式的键值，有两种方式。

使用--scan 与--pattern 参数。

```
redis-cli --scan --pattern '*'
```

或者在 redis-cli 命令行下使用 keys 命令。

```
$ keys *
```

7.2.3.5 监视当前执行命令

UPRedis 可以通过 redis-cli 实时监控当前执行命令，使用了 pub/sub 机制。

```
$ monitor
```

在生产上不要开启 **monitor**，非常消耗内存，影响 **UPRedis** 性能

7.2.3.6 命令执行延迟监控

redis-cli 对命令执行延迟监控提供了--latency 参数。

```
redis-cli --latency
```

如果要查看一段时间内的命令执行延迟：

```
redis-cli --latency-history -i 15
```

-i 参数指定了收集延迟数据的间隔，默认为 15s。

此外，`redis-cli` 还允许收集关于系统本身“固有延迟”数据。

```
redis-cli --intrinsic-latency 5
```

如上命令收集了 5 秒内运行 `redis-cli` 所在机器的系统执行命令所花费时间。建议仅当 `redis-server` 所在机器上出现明显的延迟问题，并排除不是由 `redis-server` 引起时再运行检查。

7.2.3.7 slow-log 监控

建议打开慢查询日志，对性能影响很小，可以在 UPRedis 配置文件中设定，也可以使用如下命令进行查询和配置:

```
$ config get slowlog-log-slower-than
$ config set slowlog-log-slower-than 1000
```

要查看 slow-log:

```
$ slowlog get
```

典型的运行结果如下：

```
1) 1) (integer) 0 # 唯一的日志标识符
   2) (integer) 1469250091 # 命令执行 unix 时间戳
   3) (integer) 573069 # 命令执行时间
   4) 1) "SET" # 命令细节
      2) "lru:78352\n"
      3) "val"
```

要查看当前 slow-log 数量：

```
$ slowlog len
```

清空 slow-log:

```
$ slowlog reset
```

7.3 日常管理命令-UPRedis Proxy

7.3.1 进程查看

```
ps -ef | grep upredis-proxy
```

分为三种进程:

主进程
upredis-proxy: master process pid: xxx
用于守护工作进程和管理进程

管理进程
upredis-proxy: management process stats port: xxxx
用于监控主从切换、重载配置文件以及接收、执行管理命令

工作进程
upredis-proxy: worker process
用于传递交易及执行管理命令

7.3.2 管理端口

UPRedis Proxy 默认启动管理线程 (端口默认:22222)，如果需要禁用，通过-S 启动项指定。

目前支持的管理命令:

Service	Function	concur
reload_redis	配置文件中的servers项出现增删改，执行本命令，使修改生效	No
reload_sentinel	配置文件中的sentinels项出现增删改，执行本命令，使修改生效	No
stats	查询proxy的状态信息 (所有 worker 状态信息的总和)	
stats_all	查询每个worker的状态信息	No
migrate_start	设置proxy的数据迁移状态为 MIGRATING，拒绝所有请求 (部分管理命令除外)	N
migrate_end	设置proxy的数据迁移状态为 MIGRATED，接收所有请求	No
shutdown	关闭proxy	Yes

处于数据迁移状态，管理端口只接收 migrate_end、reload_redis 和 shutdown。

正常情况，除了 shutdown，其他命令都不能并发。

7.3.2.1 参数重载命令

reload_redis 用于使工作进程的配置文件生效，支持生效的配置项有：

```
servers

black_list

white_list

dbpms
```

reload_sentinel 用于使管理进程的配置生效，支持生效的配置项有：

```
sentinels
```

7.3.2.2 数据迁移命令

migrate_start 和 migrate_end 命令用于配合 redis-migrate-tool 使用。

执行 migrate_start 后，proxy 处于数据迁移状态，所有交易请求均被拒绝。

数据迁移结束后，需要执行 migrate_end，结束数据迁移状态。

7.3.2.3 状态查看命令

查看 UPRedis Proxy 状态的命令有两个，stats 和 stats_all

连接管理端口，发送 stats 命令，会收到以下回复：

```
{
        "service":      "upredis-proxy",
        "source":       "mysql-miaohao1",
        "version":      "0.4.0",
        "upversion":    "1.3.0",
        "uptime":       "435",
        "timestamp":    "1499330173",
        "total_connections":    "4",
        "curr_connections":     "4",
        "alpha":        {
                "client_eof":   "0",
                "client_err":   "0",
                "client_connections":   "0",
                "server_ejects":        "0",
                "forward_error":        "0",
                "fragments":    "0",
```

```
                "server1":      {
                        "server_eof":   "0",
                        "server_err":   "0",
                        "server_timedout":      "0",
                        "server_connections":   "0",
                        "server_ejected_at":    "0",
                        "requests":     "0",
                        "request_bytes":        "0",
                        "responses":    "0",
                        "response_bytes":       "0",
                        "in_queue":     "1",
                        "in_queue_bytes":       "0",
                        "out_queue":    "0",
                        "out_queue_bytes":      "0"
                }
        }
}
```

每项的具体含义请通过./upredis-proxy -D 查看。

```
pool stats:
  client_eof            "# eof on client connections(断线的客户端的数量)"
  client_err            "# errors on client connections(出错客户端的数量)"
  client_connections    "# active client connections(可用客户端的数量)"
  server_ejects         "# times backend server was ejected(后端 server 内隔离的次数)"
  forward_error         "# times we encountered a forwarding error(遇到请求传递错误的次数)"
  fragments             "# fragments created from a multi-vector request(类似 mset 命令的子请求数)"

server stats:
  server_eof            "# eof on server connections(断线的 server 的数量)"
  server_err            "# errors on server connections(出错 server 的数量)"
  server_timedout       "# timeouts on server connections(超时 server 的数量)"
  server_connections    "# active server connections(可用 server 的数量)"
  server_ejected_at     "timestamp when server was ejected in usec since epoch(server 被隔离的时间戳)"
  requests              "# requests(请求数)"
  request_bytes         "total request bytes(请求大小)"
  responses             "# responses(响应数)"
  response_bytes        "total response bytes(响应大小)"
  in_queue              "# requests in incoming queue(in_queue 数量)"
  in_queue_bytes        "current request bytes in incoming queue(in_queue 大小)"
  out_queue             "# requests in outgoing queue(out_queue 数量)"
  out_queue_bytes       "current request bytes in outgoing queue(out_queue 大小)"
```

连接管理端口，发送 stats_all 命令，会收到以下回复:

```
{"workerpid":"9410", "service":"upredis-proxy", "source":"mysql-miaohao1", "version":"0.4.0", "upversi
```

启动了多少个 worker，stats_all 就会返回多少个 worker 的状态信息。

7.3.2.4 关闭命令

shutdown 命令用于关闭 UPRedis Proxy

7.4 工具

7.4.1 redis-benchmark

UPRedis 自带的测试工具，可以模拟 N 个客户端同时发出 M 个请求。

7.4.1.1 启动

```
Usage: redis-benchmark [-h <host>] [-p <port>] [-c <clients>] [-n <requests]> [-k <boolean>]

 -h <hostname>      服务器 ip (default 127.0.0.1)
 -p <port>          服务器 port (default 6379)
 -s <socket>        服务器 socket (overrides host and port)
 -a <password>      服务器密码
 -c <clients>       并发客户端数量 (default 50)
 -n <requests>      请求数量 (default 100000)
 -d <size>          set/get 的值得大小，单位字节 (default 2)
 -dbnum <db>        选择分库 (default 0)
 -k <boolean>       1=keep alive 0=reconnect (default 1)
 -r <keyspacelen>   使用随机的 key 和随机的 value 测试 set/get/incr 命令
 -P <numreq>        Pipeline <numreq> requests. Default 1 (no pipeline).
 -q                 只展示 query/sec
 --csv              CSV 格式输出
 -l                 Loop. Run the tests forever
 -t <tests>         只测试<tests>中的命令
 -I                 Idle mode. Just open N idle connections and wait.
 --tps              设置发送的 TPS 大小
```

7.4.1.2 使用举例

```
./redis-benchmark -t set,lpush -n 100000 -q
```

7.4.1.3 限制

redis-benchmark 由于其单进程但线程的限制，只适合压单台 upredis 或者性能和单台 upredis 接近的应用，当测试对象性能超出后，需要启动多台 redis-benchmark 以防止客户端压力不足；

redis-benchmark 没有对错误的处理，比如认证失败，没有数据命中等，因此测试需要注意是否真正的访问了数据，而不是出错；

redis 官方推荐另外一款多线程压测工具：`memtier_benchmark`

memtier_benchmark
安装与使用

7.4.2 redis-check-dump

用于检查 RDB 文件是否合法，修复错误的 RDB 文件

使用举例：

```
./redis-check-dump [RDB 文件]  /* 检查 RDB 文件是否合法 */
./redis-check-dump --fix [RDB 文件]  /* 修复 RDB 文件 */
```

7.4.3 redis-check-aof

用于检查 AOF 文件是否合法，修复错误的 AOF 文件

使用举例：

```
./redis-check-aof [AOF 文件] /* 检查 AOF 文件是否合法 */
./redis-check-aof --fix [AOF 文件]  /* 修复 AOF 文件 */
```

8 开发

8.1 介绍

目前 UPRedis 支持绝大部分语言编写的客户端，如 c 语言的 hiredis，java 的 jedis，python 的 redis-py 等等，详细的客户端支持情况，请查阅redis 客户端

8.2 Hiredis

Hiredis 是 UPRedis 的一个轻量 C 语言客户端，提供了对 UPRedis 操作语句支持的接口。除了支持发送命令和接收应答/应答数据，它还提供了对应答数据的解析操作。

8.2.1 同步 API

```
/* 连接数据库 */
redisContext *redisConnect(const char *ip, int port);

/* 发送命令到 UPRedis */
```

```
void *redisCommand(redisContext *c, const char *format, ...);

/* 清理请求结构 */
void freeReplyObject(void *reply);
```

8.2.2 同步 API 编程举例:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <hiredis.h>

int main() {
    unsigned int j;
    redisContext *c;
    redisReply *reply;
    const char *hostname = "127.0.0.1";
    int port = 3301;

    c = redisConnect(hostname, port);
    if (c == NULL || c->err) {
        if (c) {
            printf("Connection error: %s\n", c->errstr);
            redisFree(c);
        } else {
            printf("Connection error: can't allocate redis context\n");
        }
        exit(1);
    }

    /* PING server */
    reply = redisCommand(c,"PING");
    printf("PING: %s\n", reply->str);
    freeReplyObject(reply);

    /* Set a key */
    reply = redisCommand(c,"SET %s %s", "foo", "hello world");
    printf("SET: %s\n", reply->str);
    freeReplyObject(reply);

    /* Pipeline */
    redisAppendCommand(context, "SET foo bar");
    redisAppendCommand(context, "GET foo");
    redisGetReply(context, &reply); //reply for SET
    freeReplyObject(reply);
    redisGetReply(context, &reply); //reply for GET
```

```
    freeReplyObject(reply);

    /* Disconnects and frees the context */
    redisFree(c);

    return 0;
}
```

8.2.3 异步 API

```
/* 用于建立异步连接 */
redisAsyncContext *redisAsyncConnect(const char *ip, int port);

/* 设置连接回调函数，回调函数形式：void callback(const redisAsyncContext *c, int status) */
int redisAsyncSetConnectCallback(redisAsyncContext *ac, redisConnectCallback *fn);

/* 设置断开连接回调函数，回调函数形式：void callback(const redisAsyncContext *c, int status) */
int redisAsyncSetDisconnectCallback(redisAsyncContext *ac, redisDisconnectCallback *fn);

/* 断开异步连接 */
void redisAsyncDisconnect(redisAsyncContext *ac);

/* 释放建立连接时，创建的 redisAsyncContext 结构 */
void redisAsyncFree(redisAsyncContext *ac);

/* 发送 Redis 命令，需要实现一个回调函数来出来命令的返回，fn 是回调函数的地址，回调函数形式：void callback(red
int redisAsyncCommand(redisAsyncContext *ac, redisCallbackFn *fn, void *privdata, const char *form
```

8.2.4 异步 API 编程举例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>

#include <hiredis.h>
#include <async.h>
#include <adapters\ae.h>

/* Put event loop in the global scope, so it can be explicitly stopped */
static aeEventLoop *loop;

void getCallback(redisAsyncContext *c, void *r, void *privdata) {
    redisReply *reply = r;
    if (reply == NULL) return;
```

```
        printf("argv[%s]: %s\n", (char*)privdata, reply->str);

        /* Disconnect after receiving the reply to GET */
        redisAsyncDisconnect(c);
    }

    void connectCallback(const redisAsyncContext *c, int status) {
        if (status != REDIS_OK) {
            printf("Error: %s\n", c->errstr);
            aeStop(loop);
            return;
        }

        printf("Connected...\n");
    }

    void disconnectCallback(const redisAsyncContext *c, int status) {
        if (status != REDIS_OK) {
            printf("Error: %s\n", c->errstr);
            aeStop(loop);
            return;
        }

        printf("Disconnected...\n");
    }

    int main (int argc, char **argv) {

        signal(SIGPIPE, SIG_IGN);

        loop = aeCreateEventLoop(1024 * 10);

        redisAsyncContext *c = redisAsyncConnect("127.0.0.1", 3301);
        if (c->err) {
            /* Let *c leak for now... */
            printf("Error: %s\n", c->errstr);
            return 1;
        }

        loop = aeCreateEventLoop(64);

        redisAeAttach(loop, c);
        redisAsyncSetConnectCallback(c,connectCallback);
        redisAsyncSetDisconnectCallback(c,disconnectCallback);
        redisAsyncCommand(c, NULL, NULL, "SET key %b", argv[argc-1], strlen(argv[argc-1]));
        redisAsyncCommand(c, getCallback, (char*)"end-1", "GET key");
        aeMain(loop);
        return 0;
}
```

8.3 UPRedis-Api-C

8.3.1 配置选项设置

1. 函数设置

```
int upredis_set_option(upredis_t *uprds, int key, void *value);
```

如果需要设置**dbpm**:

```
char *dbpminfo = "failover://172.18.64.196:7000:5000:dbname:usrname,172.18.64.196:8000:5000:dbname:usr
int ret  = upredis_set_option(uprds, UPREDIS_OPT_DBPM_INFO, (void *)dbpminfo);
```

2. 配置文件

```
[servers_info]
servers=loadbalance://IP1:PORT1,IP2:PORT2,IP3:PORT3
svrs_timeout = 5000
encryption = 1
max_conn = 50000
conns_per_server = 100
dbpm_info = failover://172.18.64.196:7000:5000:dbname:usrname,172.18.64.196:8000:5000:dbname:usrname
```

8.3.2 初始化连接池

1. 传参

```
int upredis_init(upredis_t *uprd, char *servers, char *passwd);
```

2. 配置文件

```
int upredis_init_from_cfg(upredis_t *uprds, char *config_file, char *section, char *passwd);
```

8.3.3 获取连接

```
conn_t *upredis_open_conn(upredis_t *uprds); /* 获取短连接 */
conn_t *upredis_open_persistent_conn(upredis_t *uprds); /* 获取长连接 */
```

8.3.4 执行 **redis** 命令

```
redisReply *upredis_command(conn_t *conn, char *format, ...);
```

8.3.5 关闭连接

```
void upredis_close_conn(upredis_t *uprds, conn_t *conn);
```

8.3.6 销毁句柄

```
int upredis_destroy_handle(upredis_t *uprds);
```

8.3.7 参数重载

1. 传参

```
int upredis_reload(upredis_t *uprds, char *servers, char *passwd);
```

2. 配置文件

```
int upredis_reload_from_cfg(upredis_t *uprds, char *config_file, char *section, char *passwd);
```

8.3.8 API

```
    /* 创建 upredis 句柄 */
    upredis_t *upredis_create_handle();

    /* 设置配置选项 */
    int upredis_set_option(upredis_t *uprds, int key, void *value);

    /* 获取配置选项 */
    int upredis_get_option(upredis_t *uprds, int key, void *value);

    /* 销毁 upredis 句柄 */
    void upredis_destroy_handle(upredis_t *uprds);

    /* 根据设置的配置项初始化 upredis 句柄 */
    int upredis_init(upredis_t *uprds, char *redis_info, char *passwd);

    /* 根据配置文件初始化 upredis 句柄 */
    int upredis_init_from_cfg(upredis_t *uprds, char *cfg_file, char *section, char *passwd);

    /* 根据设置的配置选项进行配置重载 */
    int upredis_reload(upredis_t *uprds, char *redis_info, char *passwd);
```

```
/* 根据配置文件进行配置重载 */
int upredis_reload_from_cfg(upredis_t *uprds, char *cfg_file, char *section, char *passwd);

/* 获取一个长连接 */
upredis_conn_t *upredis_open_persistent_conn(upredis_t *uprds);

/* 获取一个短连接 */
upredis_conn_t *upredis_open_conn(upredis_t *uprds);

/* 关闭一个连接 */
int upredis_close_conn(upredis_t *uprds, upredis_conn_t *conn);

/* 与 redis 建立连接 */
upredis_conn_t *upredis_connect_with_auth(char *ip, int port, int timeout, char *passwd, int encrp

/* 获取 redisContext 结构体 */
redisContext *upredis_get_rediscontext(upredis_conn_t *conn);

/* 获取连接池中的不同计数，通过 key 指定 */
int upredis_get_pool_info(upredis_t *uprds, char *ip, int port, int key);

/* 获取 conn 连接的 server 的 ip 和 port */
int upredis_get_server_ip_port(upredis_conn_t *conn,char *ip, int *port);
```

对原生**hiredis** api 进行的封装：

```
void upredis_free_reply_object(void *reply);

int upredis_vformat_command(char **target, const char *format, va_list ap);

int upredis_format_command(char **target, const char *format, ...);

int upredis_format_command_argv(char **target, int argc, const char **argv, const size_t *argvlen)

upredis_conn_t *upredis_connect(const char *ip, int port);

upredis_conn_t *upredis_connect_with_timeout(const char *ip, int port, const struct timeval tv);

upredis_conn_t *upredis_connect_non_block(const char *ip, int port);

upredis_conn_t *upredis_connect_bind_non_block(const char *ip, int port, const char *source_addr);

upredis_conn_t *upredis_connect_unix(const char *path);

upredis_conn_t *upredis_connect_unix_with_timeout(const char *path, const struct timeval tv);

upredis_conn_t *upredis_connect_unix_non_block(const char *path);

upredis_conn_t *upredis_connect_fd(int fd);
```

```
int upredis_set_timeout(redisContext *c, const struct timeval tv);

int upredis_enable_keep_alive(redisContext *c);

void upredis_free(redisContext *c);

int upredis_free_keep_fd(redisContext *c);

int upredis_get_reply(upredis_conn_t *conn, redisReply **reply);

int upredis_append_formatted_command(upredis_conn_t *conn, const char *cmd, size_t len);

int upredis_vappend_command(upredis_conn_t *conn, const char *format, va_list ap);

int upredis_append_command(upredis_conn_t *conn, const char *format, ...);

int upredis_append_command_argv(upredis_conn_t *conn, int argc, const char **argv, const size_t *a

redisReply *upredis_vcommand(upredis_conn_t *conn, const char *format, va_list ap);

redisReply *upredis_command(upredis_conn_t *conn, const char *format, ...);

redisReply *upredis_command_argv(upredis_conn_t *conn, int argc, const char **argv, const size_t *
```

8.3.9 编程举例

```
main thread:

    #include "upredis-api.h"

    upredis_t *uprds = upredis_create_handle(void);

    int encrp_type = 0;
    upredis_set_option(uprds, UPREDIS_OPT_MAX_CONN, (void*)&max_conn, sizeof(int));

    char *dbpminfo = "failover://172.18.64.196:7000:5000:dbname:usrname,172.18.64.196:8000:5000:dbname
    upredis_set_option(uprds, UPREDIS_OPT_DBPM_INFO, (void*)dbpminfo, strlen(dbpminfo));

    char *svr = "loadbalance://127.0.0.1:3301,127.0.0.1:3302,127.0.0.1:3303,127.0.0.1:3304";
    upredis_init(uprds, svr, NULL);
    //upredis_init_from_cfg(uprds, "/PATH/TO/CONFIG/conf.ini", "servers_info", NULL);

    create work thread...

    exit:
    upredis_destroy_handle(uprds);
```

```
worker thread exampl1:
    while(1) {
        //... ...
        conn_t *conn = upredis_open_conn(uprds);
        redisReply *reply = upredis_command(conn, "get %s", key1);
        //... ...
        upredis_free_reply_object(reply);
        upredis_close_conn(uprds, conn);
    }


worker thread exampl2:

    conn_t *conn = upredis_open_persistent_conn(uprds);
    while(1) {
        //... ...
        redisReply *reply = upredis_command(conn, "get %s", key1);
        //... ...
        upredis_free_reply_object(reply);
    }
    upredis_close_conn(uprds, conn);

reload notify:

    // Whenever you modify the servers, you can reload it without stoping the transaction or the apps.

    //get the reload notify
    //... ...
    char *svr = "loadbalance://127.0.0.1:3301,127.0.0.1:3302,127.0.0.1:3305,127.0.0.1:3306";
    upredis_reload(uprds, svr, NULL);
    //upredis_reload_from_cfg(uprds, "/PATH/TO/CONFIG/conf.ini", "servers_info", NULL);
```

8.4 Pipeline

对于 UPRedis 而言，pipeline 处理一个客户端打包发送多条请求的情况 (客户端主动将多条命令打包)。

下面以 hiredis 为例，说明如何使用 pipeline

8.4.1 API

```
    /* 组织需要发送的请求 */
    void redisAppendCommand(redisContext *c, const char *format, ...);
```

```
/* 组织需要发送的请求 */
void redisAppendCommandArgv(redisContext *c, int argc,
    const char **argv, const size_t *argvlen);

/* 发送请求或者接受响应 */
int redisGetReply(redisContext *c, void **reply);
```

8.4.2 编程举例

```
redisReply *reply;
/* 将请求打包 */
redisAppendCommand(context, "SET foo bar");
redisAppendCommand(context, "GET foo");
/* 发送请求，并接收响应 */
redisGetReply(context, &reply); //reply for SET
freeReplyObject(reply);
/* 接收响应 */
redisGetReply(context, &reply); //reply for GET
freeReplyObject(reply);
```

8.5 Auth

8.5.1 API

```
void *redisCommand(redisContext *c, const char *format, ...);
```

8.5.2 编程举例

```
redisReply *reply;
reply = redisCommand(context, "AUTH foobar");
```

```
redisReply *reply;
reply = redisCommand(context, "AUTH_S 1b58ee375b42e41f0e48ef2ff27d10a5b1f6924a9acdcdba7cae868e7adc
```

在 UPRedis-Api-C 中可以通过配置 `encryption = 1` 来达到在传输中使用加密的密码;

```
[servers_info]
encryption = 1
```

9 迁移

9.1 迁移工具

redis-migrate-tool 是由唯品会开源的 Redis 集群迁移工具，基于 redis 复制机制。具有以下特点：

特点：

- 快速。
- 多线程。
- 基于 redis 复制。
- 实时迁移。
- 迁移过程中，源集群不影响对外提供服务。
- 异构迁移。
- 支持 Twemproxy 集群，redis cluster 集群，rdb 文件和 aof 文件。
- 过滤功能。
- 当目标集群是 Twemproxy，数据会跳过 Twemproxy 直接导入到后端的 redis。
- 迁移状态显示。
- 完善的数据抽样校验。

9.2 使用场景

9.2.1 数据源：**RDB**

1.rdb ——————>UPRedis Proxy(rdb 数据迁移到 UPRedis Proxy 集群)

- 一个 rdb 文件——>UPRedis Proxy
- 多个 rdb 文件——>UPRedis Proxy

2.rdb ——————>single(rdb 数据迁移到一个或者多个 UPRedis)

- 一个 rdb 文件——> 一个 UPRedis
- 一个 rdb 文件——> 多个 UPRedis(不支持)
- 多个 rdb 文件——> 一个 UPRedis
- 多个 rdb 文件——> 多个 UPRedis(不支持)

3.rdb ——————>rdb(不支持)

4.rdb ——————>aof(不支持)

9.2.2 数据源：**AOF**

1.aof——————->UPRedis Proxy(aof 数据迁移到 UPRedis Proxy 集群)

- 一个 aof 文件——>UPRedis Proxy
- 多个 aof 文件——>UPRedis Proxy

2.aof ————————>single(aof 数据迁移到一个或者多个 UPRedis)

- 一个 aof 文件——> 一个 UPRedis
- 一个 aof 文件——> 多个 UPRedis(不支持)
- 多个 aof 文件——> 一个 UPRedis
- 多个 aof 文件——> 多个 UPRedis(不支持)

3.aof ————————>rdb(不支持)

4.aof ————————>aof(不支持)

9.2.3 数据源：single

1.single————————->UPRedis Proxy(单个或多个 UPRedis 服务器的数据迁移到 UPRedis Proxy 集群)

- 一个 UPRedis——>UPRedis Proxy
- 多个 UPRedis——>UPRedis Proxy

2.single————————>single(一个或者多个 UPRedis 迁移到一个或者多个 UPRedis)

- 一个 UPRedis——> 一个 UPRedis
- 一个 UPRedis——> 多个 UPRedis(不支持)
- 多个 UPRedis——> 一个 UPRedis
- 多个 UPRedis——> 多个 UPRedis(不支持)

3.single————————>rdb

- 一个 UPRedis——>rdb
- 多个 UPRedis——>rdb(不支持, 生成多个文件)

4.single————————>aof(不支持)

9.2.4 数据源：UPRedis Proxy

1.UPRedis Proxy——>UPRedis Proxy(UPRedis Proxy 集群的数据迁移到 UPRedis Proxy 集群)

注意：如果数据源中存在不属于数据源 **UPRedis Proxy** 分片算法分片的数据，这部分数据不会被迁移 **(**但是在线生成的数据会被迁移**)**

2.UPRedis Proxy——>single(UPRedis Proxy 集群的数据迁移到一个或者多个 UPRedis)

- UPRedis Proxy——> 一个 UPRedis
- UPRedis Proxy——> 多个 UPRedis(不支持)

3.UPRedis Proxy——>rdb(不支持, 生成多个文件, 根据源端 UPRedis 的数量)

4.UPRedis Proxy——>aof(不支持)

9.2.5 配置文件说明

配置文件分为三个部分：source，target 以及 common。

9.2.5.1 source 或者 target 配置项

type	数据的来源或者数据迁移的集群类型，包括： single(一个或者多个 UPRedis) twemproxy(UPRedis Proxy) redis cluster rdb file aof file
servers	数据来源或者去向列表，如果是twemproxy，UPRedis Proxy 一致
redis_auth	密码，与UPRedis或者 UPRedis Proxy 的密码一致
timeout	读写UPRedis服务器的超时时间，只对 source 有效，默认 120 秒
hash	与UPRedis Proxy 的配置一致
hash_tag	与UPRedis Proxy 的配置一致
distribution	与UPRedis Proxy 的配置一致

9.2.5.2 common 的配置项

listen	redis-migrate-tool的监听 ip 和 port，默认：127.0.0.1:8888(无需修改)
max_clients	最大客户端连接数，默认：100(无需修改)
threads	最大线程数，默认：等于cpu核心数，如果数据源是文件，最大线程数应该大于文件数量
step	解析请求时使用，设置越大，迁移越快，但是越耗资源，默认：1(无需修改)
mbuf_size	请求的缓存大小，默认：512字节 (无需修改)
source_safe	不允许同时迁移多个UPRedis，必须设置为 true
dir	工作目录，用于存储临时的rdb文件
filter	过滤不符合patten的 key，默认 NULL

9.2.6 配置文件举例

9.2.6.1 rdb—->UPRedis Proxy

```
[source]
type: rdb file
servers:
 - /data/redis/dump.rdb-3301
 - /data/redis/dump.rdb-3302

[target]
type: twemproxy
redis_auth: foobared
hash: fnv1a_64
hash_tag: "{}"
distribution: modula
servers:
 - 127.0.0.1:3301:1 server1
 - 127.0.0.1:3302:1 server2
 - 127.0.0.1:3303:1 server3
 - 127.0.0.1:3304:1 server4

[common]
listen: 0.0.0.0:8888
step: 1
mbuf_size: 512
source_safe: true
```

9.2.6.2 single—->UPRedis Proxy

```
[source]
type: single
redis_auth: foobared
servers:
 - 127.0.0.1:3301
 - 127.0.0.1:3302

[target]
type: twemproxy
redis_auth: foobared
hash: fnv1a_64
hash_tag: "{}"
distribution: modula
servers:
 - 127.0.0.1:3305:1 server1
 - 127.0.0.1:3306:1 server2
 - 127.0.0.1:3307:1 server3
```

```
 - 127.0.0.1:3308:1 server4

[common]
listen: 0.0.0.0:8888
step: 1
mbuf_size: 1024
source_safe: true
```

9.2.6.3 UPRedis Proxy—->UPRedis Proxy

```
[source]
type: twemproxy
redis_auth: foobared
hash: fnv1a_64
hash_tag: "{}"
distribution: modula
servers:
 - 127.0.0.1:3301:1 server1
 - 127.0.0.1:3302:1 server2

[target]
type: twemproxy
redis_auth: foobared
hash: fnv1a_64
hash_tag: "{}"
distribution: modula
servers:
 - 127.0.0.1:3305:1 server1
 - 127.0.0.1:3306:1 server2
 - 127.0.0.1:3307:1 server3
 - 127.0.0.1:3308:1 server4

[common]
listen: 0.0.0.0:8888
step: 1
mbuf_size: 1024
source_safe: true
```

更多配置文件请参考《redis-migrate-tool 参考手册》

9.2.7 注意事项

- 扩缩容使用的新旧机器不能有交集
- 进行扩缩容前，确保旧机器的硬盘空间至少能生成 UPRedis 的 rdb 文件
- 以下命令不支持数据迁移:

```
RENAME
RENAMENX
RPOPLPUSH
BRPOPLPUSH
FLUSHALL
FLUSHDB
BITOP
MOVE
GEORADIUS
GEORADIUSBYMEMBER
EVAL
EVALSHA
SCRIPT
PFMERGE
```

9.3 离线扩缩容

9.3.1 离线扩容

9.3.1.1 step0: 说明

以 UPRedis Proxy—->UPRedis Proxy 为例，说明扩容的使用方法。

```
                                  +--------------------+
                                  | 172.21.101.91:3301 |
+----------------------+          +--------------------+
| 172.17.240.227:3301 |           | 172.21.101.91:3302 |
+----------------------+  ------>  +--------------------+
| 172.17.240.227:3302 |           | 172.21.101.91:3303 |
+----------------------+          +--------------------+
                                  | 172.21.101.91:3304 |
                                  +--------------------+
```

将旧集群（172.17.140.227:3301, 172.17.140.227:3302）的数据向新集群（172.21.101.91:3301, 172.21.101.91:3302, 172.21.101.91:3303, 172.21.101.91:3304）进行迁移。

9.3.1.2 step1: 启动新 redis 集群

```
redis-server /path/to/redis.conf-3301
redis-server /path/to/redis.conf-3302
redis-server /path/to/redis.conf-3303
redis-server /path/to/redis.conf-3304
```

9.3.1.3 step2: 准备 rmt 配置文件

根据新旧集群的配置，准备 rmt 配置文件：其中 [source] 与旧集群的 proxy 配置一致，[target] 与新集群一致，其他配置请参考 rmt 的配置说明。

```
[source]
type: twemproxy
redis_auth: foobared
hash: fnv1a_64
hash_tag: "{}"
distribution: modula
servers:
   - 172.17.140.227:3301:1 server1
   - 172.17.140.227:3302:1 server2

[target]
type: twemproxy
redis_auth: foobared
hash: fnv1a_64
hash_tag: "{}"
distribution: modula
servers:
   - 172.21.101.91:3301:1 server1
   - 172.21.101.91:3302:1 server2
   - 172.21.101.91:3303:1 server3
   - 172.21.101.91:3304:1 server4

[common]
listen: 127.0.0.1:8888
step: 1
mbuf_size: 1024
source_safe: true
```

9.3.1.4 step3: 启动 rmt, 开始数据迁移

```
./redis-migrate-tool -c rmt.conf -o log -d
```

9.3.1.5 step4: 监测 rmt 状态，并作相应处理

查看日志，如果出现：

```
All nodes' rdb file parsed finished for this write thread(0)
```

表示 rmt 转移数据完成，可以关闭 rmt；否则需要清空新集群，重新进行数据迁移。

也可以使用 `redis-cli` 连接到 `redis-migrate-tool`(默认 `8888`)，并执行 `info` 命令，查看当前 `rmt` 运行状态。

```
$redis-cli -h 127.0.0.1 -p 8888

127.0.0.1:8888> info
# Server
version:0.1.0
os:Linux 2.6.32-573.12.1.el6.x86_64 x86_64
multiplexing_api:epoll
gcc_version:4.4.7
process_id:9199
tcp_port:8888
uptime_in_seconds:1662
uptime_in_days:0
config_file:/ect/rmt.conf

# Clients
connected_clients:1
max_clients_limit:100
total_connections_received:3

# Memory
mem_allocator:jemalloc-4.0.4

# Group
source_nodes_count:32
target_nodes_count:48

# Stats
all_rdb_received:1
all_rdb_parsed:1
all_aof_loaded:0
rdb_received_count:2
rdb_parsed_count:2
aof_loaded_count:0
total_msgs_recv:7753587
total_msgs_sent:7753587
total_net_input_bytes:234636318
total_net_output_bytes:255384129
total_net_input_bytes_human:223.77M
total_net_output_bytes_human:243.55M
total_mbufs_inqueue:0
total_msgs_outqueue:0
```

9.3.1.6 **step5:** 数据一致性检查

rmt 完成数据迁移后，关闭 rmt 工具。

检查新旧集群数据是否一致:

```
./redis-migrate-tool -c rmt.conf -o log -C "redis_check [key 的数量]"
```

关闭旧集群，使用新集群提供 redis 服务。

9.3.2 离线缩容

与离线扩容类似，具体请参考《redis-migrate-tool 参考手册》

9.4 在线扩缩容

在线扩缩容使用 rmt-bootstrap 脚本，该脚本调用 rmt 数据迁移，UPRedis Proxy 参数动态生效等特性完成在线扩缩容。

注意：在线扩缩容，只适用于 **UPRedis Proxy-1.1.0** 及以上版本

9.4.1 在线扩容

9.4.1.1 step0: 说明

以 UPRedis Proxy—->UPRedis Proxy 为例，说明扩容的使用方法。

```
                                +--------------------+
                                | 172.21.101.91:3301 |
+----------------------+        +--------------------+
| 172.17.240.227:3301 |         | 172.21.101.91:3302 |
+---------------------+  ------> +--------------------+
| 172.17.240.227:3302 |         | 172.21.101.91:3303 |
+----------------------+        +--------------------+
                                | 172.21.101.91:3304 |
                                +--------------------+
```

将旧集群（172.17.140.227:3301，172.17.140.227:3302）的数据向新集群（172.21.101.91:3301, 172.21.101.91:3302, 172.21.101.91:3303, 172.21.101.91:3304）进行迁移。

9.4.1.2 step1: 启动新 redis 集群

```
redis-server /path/to/redis.conf-3301
redis-server /path/to/redis.conf-3302
redis-server /path/to/redis.conf-3303
redis-server /path/to/redis.conf-3304
```

9.4.1.3 step2: 准备 UPRedis Proxy 配置文件

将所有的 UPRedis Proxy 的配置文件修改为新集群配置:

```
alpha:
  listen: 172.21.101.91:22201
  hash: fnv1a_64
  redis_auth: foobared
  hash_tag: "{}"
  distribution: modula
  auto_eject_hosts: false
  timeout: 400
  redis: true
  servers:
   - 172.21.101.91:3301:1 server1
   - 172.21.101.91:3302:1 server2
   - 172.21.101.91:3303:1 server3
   - 172.21.101.91:3304:1 server4
```

9.4.1.4 step3: 准备 rmt 配置文件

根据新旧集群的配置，准备 rmt 配置文件：其中 [source] 与旧集群的 proxy 配置一致，[target] 与新集群一致，其他配置请参考 rmt 的配置说明。

```
[source]
type: twemproxy
redis_auth: foobared
hash: fnv1a_64
hash_tag: "{}"
distribution: modula
servers:
   - 172.17.140.227:3301:1 server1
   - 172.17.140.227:3302:1 server2

[target]
type: twemproxy
redis_auth: foobared
hash: fnv1a_64
hash_tag: "{}"
distribution: modula
servers:
   - 172.21.101.91:3301:1 server1
   - 172.21.101.91:3302:1 server2
   - 172.21.101.91:3303:1 server3
   - 172.21.101.91:3304:1 server4

[common]
listen: 127.0.0.1:8888
```

```
step: 1
mbuf_size: 1024
source_safe: true
```

9.4.1.5 step4: 启动 rmt-bootstrap.sh 脚本，开始数据迁移

```
./rmt-bootstrap.sh -c rmt.conf -o log -p 172.21.101.91:22221,172.21.101.91:22222  d ./
```

rmt-bootstrap.sh脚本的参数说明如下：

```
 Usage: rmt-bootstrap.sh [-h]
       rmt-bootstrap.sh [-c <conf>] [-o <log>] [-d <dir>] [-t <tps>] [-p <proxies>]

      -h           帮助信息
      -c <conf>      redis-migrate-tool 配置文件 (default:<dir>/rmt.conf)
      -o <log>       redis-migrate-tool 日志文件 (default:<dir>/log)
      -d <dir>       包含 redis-migrate-tool 和 redis-cil 的工作目标 (default: ./)
      -t <tps>       允许 UPRedis Proxy 生效新配置的应用的 tps (default 0)
      -p <proxies>   UPRedis Proxy 的 ip 和管理端口: ip:port,ip:port
```

特别说明：-t 参数指定迁移时候的应用写入 tps，当写入 tps 低于设定值时才会启动迁移 (停止交易、更新 UPRedis Proxy 配置、恢复交易)，否则将一直等待；如果不设置，默认为 0，那么只有当 tps 将为 0 的时候，才会启动数据迁移。

9.4.1.6 step5: 监测 rmt-bootstrap.sh 状态，并作相应处理

正常的扩缩容输出如下：

```
[rmt] migrating, TPS: 0,  recv/sent:0B/0B
[rmt] migrating, TPS: 0,  recv/sent:0B/0B
...
[rmt] migrating, TPS: 0,  recv/sent:152.05M/0B
[rmt] migrating, TPS: 70713,  recv/sent:322.60M/809.83M
...
[rmt] migrating, TPS: 37746,  recv/sent:631.22M/5.04G
[rmt] migrate done. TPS: 22, recv/sent:631.22M/5.04G
[rmt] PHASE 1: t_parse 199397ms, t_rmt 207557 ms
[rmt] 172.21.101.91:22221 migrate_start
[rmt] 开始数据迁移
[rmt] 172.21.101.91:22222 migrate_start
[rmt] 开始数据迁移
[rmt] 172.21.101.91:22221 reload_redis
[rmt] reload finished
[rmt] 172.21.101.91:22221 migrate_end
[rmt] 结束数据迁移
[rmt] 172.21.101.91:22222 reload_redis
[rmt] reload finished
```

```
[rmt] 172.21.101.91:22222 migrate_end
[rmt] 结束数据迁移
[rmt] 扩缩容成功
[rmt] PHASE 2: t_migrate 443 ms
```

a) 如果出现扩缩容成功说明集群已经完成扩缩容，需要操作人员确认数据完整性，并检查应用是否正常。

b) 如果出现:

```
[rmt] ERROR: 扩缩容失败!（所有 UPRedis Proxy 均切换失败，保持原状态）
[rmt] ERROR: 请运维人员介入，并尝试重新切换
```

说明扩缩容失败，需要进一步分析日志，查明原因。

c) 如果出现:

```
[rmt] ERROR: 扩缩容失败!（部分 UPRedis Proxy 切换失败）
[rmt] ERROR: 请运维人员介入，并尽量切换到新集群
```

说明部分 UPRedis Proxy 参数动态生效失败，需要分析日志并查明原因。
待查明原因后，需要再次启动数据迁移前，需要手工停止 `redis-migrate-tool` 进程。

> 此时数据迁移进程没有退出，原因是：需要将部分访问老集群的交易 (写操作) 同步到新的集群，减少数据丢失; 但是新集群的写入操作不会同步到老的集群，仍然存在数据不一致的风险; 所以需要尽快确定是使用新的集群还是老的集群，停止或者修改出问题的 UPRedis Proxy。

d) 如果出现:

```
redis-migrate-tools 进程 (30953) 存在，请关闭后重试
```

说明脚本启动 `redis-migrate-tools` 失败，需要确定是否有相同进程存在并占用相同端口;

9.4.1.7 step6: 数据一致性检查

rmt 完成数据迁移后, 检查新旧集群数据是否一致:

```
./redis-migrate-tool -c rmt.conf -o log -C "redis_check [key 的数量]"
```

关闭旧集群，使用新集群提供 redis 服务。

9.4.2 在线缩容

> 与在线扩容类似，具体请参考《redis-migrate-tool 参考手册》

10 读写分离

10.1 适用场景

读写分离试用与读多写少，对于读取性能要求较高，并且能忍受 ms 级的读写不一致的场景。

10.2 配置

```
separate-read-write: off (off, read-slaves, read-both)

off: 不开启读写分离模式（default）
read-slaves：读请求只在 slaves 中负载；在所有 slave 均不可用时，读写请求都发送到 master
read-both: 读取请求在 slave 和 master 中负载
```

10.3 性能

模式	slave 数量					
	0	1	2	3	4	5
read_slaves	10217	9774	17370	23320	34809	43404
read_both	9802	17828	25929	33790	40594	45779

读写分离测试结果显示，在多个 slave 的情况下，UPRedis Proxy 的读性能基本与 slave 数量成正比。

也就是开启读写分离：1 主 1 从，读取性能最高可到 100WQPS；1 主 2 从读取性能最高 150WQPS。

由于 Proxy 架构下，需要 6 个 benchmark，UPRedis Proxy8 进程才能将 Redis 压到 100%，同时还需要使用万兆网卡。由于测试环境资源有限，以上测试使用 cpulimit 将 redis 耗费 CPU 限制到 10% 时测试的数据。

10.4 实现原理

UPRedis Proxy 区分读写命令，将写命令路由到 master，将读取命令路由到对应的 slave。master 下属的 slave 列表 Proxy 通过发送 info replication 命令获取；UPRedis

Proxy 在各 slave 之间进行按照批次 (默认 128 消息一批次) 在 slave 之间进行负载均衡。

10.5 注意事项

读写不一致:

开启读写分离，假如客户端连续执行 set k v1; set k v2; get k; 由于复制延迟, 客户端 get 命令的结果可能是 nil, v1, v2。

11 异地同步

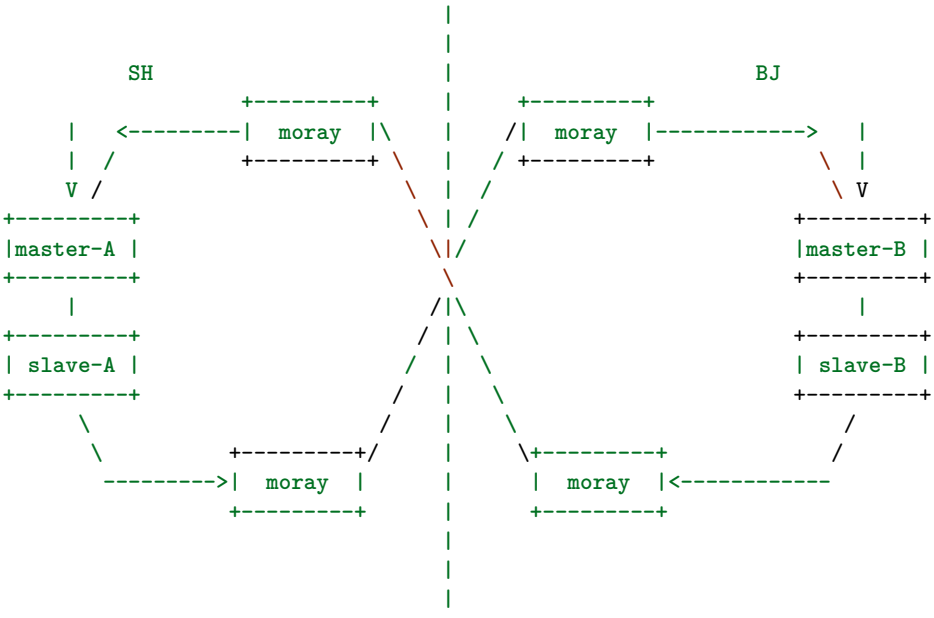
UPRedis-2.0 支持 UPRedis 数据异地同步，可用于异地双活或者灾备。

异地同步功能依赖于 UPRedis 内核的 AOF-BINLOG 特性和数据转移工具 (MORAY)。

通过 Moray 进行异地数据同步，对应用透明，无需额外开发并且对数据库性能影响较小。

> 关于 MORAY 的详细文档详见《Moray 参考手册》

11.1 原理



以 SH 和 BJ 两中心为例，说明 UPRedis 在 1 主 1 备，异地双活架构实现数据同步过程：

a) Moray 通过 opget 命令，从 SH 中心的 slave-A 获取需要日志
b) Moray 将数据从 SH 中心传输到 BJ 中心
c) BJ 中心 Moray 将日志回放到 master-B

从 BJ–>SH 的同步过程与 SH–>BJ 的同步过程类似。

11.2 设计思路

a) 数据一致性

修改 Redis 内核，实现 AOF-BINLOG 机制；通过 server-id 避免循环复制，通过 opid 过滤重复执行。

b) 高性能

支持数据过滤筛选、减少同步数据量；报文压缩，减少网络带宽；按照 PIPELINE 发送，提高收发效率；

设计性能 10W QPS，流量 400Mbps，压缩后 <40Mbps

c) 高可用

订阅 sentinel failover 信息，redis 主从切换不影响异地同步

11.3 场景

UPRedis 数据同步可用于异地双活或者异地容灾场景。

应用两中心双活的情况下，如果可以从业务上将数据拆分到不同中心，异地同步功能可以实现异地容灾；应用两中心双活的情况下，如果无法从业务上将数据拆分到不同中心，异地同步功能可以实现异地双活（有数据冲突的风险，参考注意事项）；

全渠道试点系统采用的是异地双活架构，全渠道系统使用 UPRedis-2.0 与 Moray 同步用户权限控制数据：

11.4 配置

```
# opget-max-count 用于限制 opget 命令单次能获取的日志条数。从 slave 中能获取的最大日志条数为
# opget-max-count，从 master 中能获取的最大日志条数为 opget-max-count/10。 默认 opget-max-count
# 为 10000，因此从 master 单次最多能获取 1000 条日志，从 slave 中单次最多能获取 10000 条日志。
opget-max-count 10000

# 当从 master 中获取操作日志，在线 slave 数量至少大于 opget-master-min-slave 才能从 master 中获取日志。
# 默认为 0。
opget-master-min-slaves 0
```

同步工具 Moray 相关的配置详见《Moray 参考手册》

11.5 使用方法

UPRedis 实现异地同步的接口命令，同步工具（比如 Moray）通过调用这些命令实现异地同步功能.

11.5.1 接口命令

命令语法

```
"opget <startopid> [count <count>] [matchdb <db>]* [matchkey <key>]* [matchid <serverid>]* [skipflags
"opapply" --> OK/ERR
"getopidbyaof <aof-filename>" --> <aof-first-opid>
"purgeaofto <aof-filename>" --> <aof-purged-count>
"aofflush" --> OK/ERR
```

命令说明

```
opget -- 用于从 ** 源 redis** 中拉取（并过滤）aof-binlog 文件中记录的命令（包括 opinfo 和 cmd），拉取到的结果可以
opapply -- 标记客户端为 opapply 客户端（客户端应用 opget 结果之前，需要使用 opapply 命令，否则无法识别 opinfo 命
getopidbyaof -- 用于从 aof 文件中获取 aof-first-opid。在搭建异地数据转移关系，确定转移起点时使用该命令获取 opget
purgeaofto -- 用于主动 purge aof 文件
aofflush -- 用于主动切换 aof 文件
```

11.5.2 同步流程

以下以 Moray 的同步为例说明同步的流程:

```
a) moray-redis-FE（前端：从源 UPRedis 获取日志）

redis-moray-FE 连接 redis-slave-A
while (1) {
    cmds = opget start_opid
    moray_send(cmds)
}

b) moray-redis-BE（后端：向目标 UPRedis 应用日志）

opapply
while (1) {
    cmds = moray_recv()
```

```
    send cmds to redis-master-B
}
```

11.6 部署

a) 不同的中心 UPRedis 架构必须一致（比如 SH 中心 4 分片，那么 BJ 也必须是 4 分片）
b) 异地同步依赖于 AOF-BINLOG 特性，主从库都需要开启 AOF-BINLOG（相对于不开启 AOF-BINLOG，性能下降约 15%）
c) 考虑到异地同步对主库性能可能有影响，Moray 默认从库 AOF 数据
d) 如果两个中心的 UPRedis 含有多个分片，则每个分片都需要部署 Moray 同步工具

关于 Moray 的部署方法，详见 <>

另外，UPRedis 提供了 opget/opapply 等接口命令支持异地同步，应用可以选择 Moray 进行异地同步，也可以调用这些接口实现 UPRedis 异地同步。

11.7 性能

通过 Moray 异地数据同步的性能 10W TPS，流量 400Mbps，压缩后 <40Mbps，延迟时间为两中心之间的网络延迟。

11.8 注意事项

由于异地数据同步存在以下限制，因此请注意合理地使用异地同步功能:

a) 数据过滤

异地之间的网络带宽有限，因此不需要同步的数据，可以通过 opget 命令的各种过滤开关滤除。

b) 数据一致性

异地之间数据同步存在网络延迟，因此可能出现时序混乱造成的数据不一致，应用在使用异地同步功能时应该考虑数据不一致造成的影响。

case-1(其中含有 * 标记的是同步到异地的数据）

```
SH              BJ
-----------   ----------
set k V-sh
                set k V-bj
*set k V-bj     *set k V-sh
```

预期最终 SH、BJ 中心的结果为 V-bj，但是由于时序问题造成 SH 为 V-bj，BJ 为 V-sh。

case-2

```
SH              BJ
-----------   ----------
set k v
                del k
*del k          lpush k 1
*lpush k v

                *set k v
```

预期最终 k 的结果 `[1] (list` 类型，含有数据 `1)`，但是在 BJ 执行 `set k v` 将出现 ERR。

11.9 FAQ

1) 全量数据如何开始同步，能否从某一个时间点开始同步?

全量数据通过主从复制或者拷贝 AOF-BINLOG 文件的方式同步到异地，然后再从当前的 opid 开始同步。目前异地同步只支持从某个 opid 开始，可以通过 getopidbyaof 命令查到 aof 文件对应的 opid。

2) 主从复制出现 gap，发生 failover 切换到从，是否会丢失数据?

gap 中的数据将丢失（UPRedis 默认使用异步复制，无法保证主从一致）。

3) Moray 本身采用怎样的高可用方案? 如果 Moray 所在的机器挂了会造成什么影响?

Moray 本身采用冷备方案：Moray 的同步进度存储在集中存储，如果 Moray 所在的机器挂了，则通过 agent 在冷备机器上启动一个新的 Moray 进程。

4) 目前有些 redis 集群的分片数据量较多，每个分片搭建需要搭建来去两个方案的 moray 组件，搭建复制比较麻烦，能否简化运维?

该功能通过 Moray 提供的管理监控实现。

5) upredis-moray 异地数据同步存在延迟，那么延迟是多少?

Moray 的异地数据同步延迟为异地之间的网络延迟 (20-30ms)。

6) 如果应用同时写同一个 key，会产生什么结果? 能否检测到数据冲突，并提供配置选项配置数据冲突处理策略?

会产生数据冲突，结果不确定; 目前无法检测到数据冲突。

7) upredis-moray 的典型应用场景有哪些，使用的限制有哪些，应该在文档中显式说明

参见以上文档