1. Redis高可用概述

在 Redis 中，实现 **高可用** 的技术主要包括 **持久化**、**复制**、**哨兵** 和 **集群**，下面简单说明它们的作用，以及解决了什么样的问题：

**持久化**：持久化是 **最简单的** 高可用方法。它的主要作用是 **数据备份**，即将数据存储在 **硬盘**，保证数据不会因进程退出而丢失。

复制：复制是高可用 Redis 的基础，哨兵 和 集群 都是在 复制基础 上实现高可用的。复制主要实现了数据的多机备份以及对于读操作的负载均衡和简单的故障恢复。缺陷是故障恢复无法自动化、写操作无法负载均衡、存储能力受到单机的限制。

哨兵：在复制的基础上，哨兵实现了 自动化 的 故障恢复。缺陷是 写操作 无法 负载均衡，存储能力 受到 单机 的限制。

**集群**：通过集群，Redis 解决了 **写操作** 无法 **负载均衡** 以及 **存储能力** 受到 **单机限制** 的问题，实现了较为 **完善** 的 **高可用方案**。

## 2. Redis Sentinel的基本概念

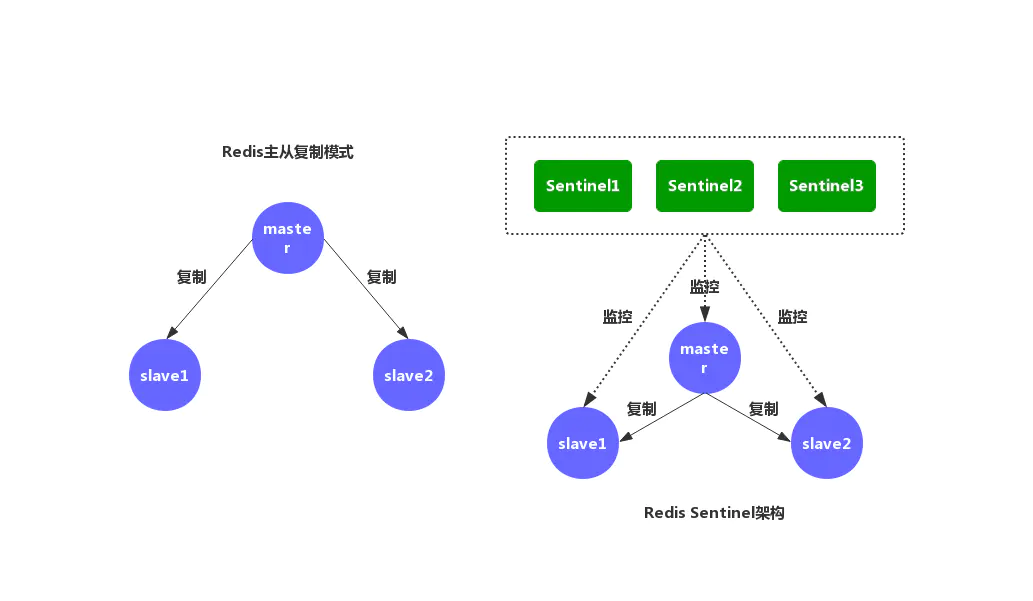
Redis Sentinel 是 Redis 高可用 的实现方案。

Sentinel 是一个管理多个 Redis 实例的工具，它可以实现对 Redis 的 监控、通知、自动故障转移。下面先对 Redis Sentinel 的 基本概念 进行简单的介绍。

基本名词说明：

|  |  |  |
| --- | --- | --- |
| Redis数据节点 | 主节点和从节点 | 主节点和从节点的进程 |
| 主节点(master) | Redis主数据库 | 一个独立的Redis进程 |
| 从节点(slave) | Redis从数据库 | 一个独立的Redis进程 |
| Sentinel节点 | 监控Redis数据节点 | 一个独立的Sentinel进程 |
| Sentinel节点集合 | 若干Sentinel节点的抽象组合 | 若干Sentinel节点进程 |
| Redis Sentinel | Redis高可用实现方案 | Sentinel节点集合和Redis数据节点进程 |
| 应用客户端 | 泛指一个或多个客户端 | 一个或者多个客户端进程或者线程 |

如图所示，Redis 的 主从复制模式 和 Sentinel 高可用架构 的示意图：



## 3. Redis主从复制的问题

Redis 主从复制 可将 主节点 数据同步给 从节点，从节点此时有两个作用：

1. 一旦 主节点宕机，从节点 作为 主节点 的 备份 可以随时顶上来。
2. 扩展 主节点 的 读能力，分担主节点读压力。

主从复制 同时存在以下几个问题：

1. 一旦 主节点宕机，从节点 晋升成 主节点，同时需要修改 应用方 的 主节点地址，还需要命令所有 从节点 去 复制 新的主节点，整个过程需要 人工干预。
2. 主节点 的 写能力 受到 单机的限制。
3. 主节点 的 存储能力 受到 单机的限制。

原生复制 的弊端在早期的版本中也会比较突出，比如：Redis 复制中断 后，从节点 会发起 psync 。此时如果 同步不成功，则会进行 全量同步，主库 执行 全量备份 的同时，可能会造成毫秒或秒级的 卡顿。

## 4. Redis Sentinel深入探究

### 4.1. Redis Sentinel的架构

### 4.2. Redis Sentinel的主要功能

Sentinel 的主要功能包括 主节点存活检测、主从运行情况检测、自动故障转移 （failover）、主从切换。

Redis 的 Sentinel 最小配置是 一主一从。

Redis 的 Sentinel系统可以用来管理多个 Redis 服务器，该系统可以执行以下四个任务：

1.监控

Sentinel 会不断的检查 主服务器 和 从服务器 是否正常运行。

2.通知

当被监控的某个 Redis 服务器出现问题，Sentinel 通过 API 脚本 向 管理员 或者其他的 应用程序 发送通知。

3.自动故障转移

当 主节点 不能正常工作时，Sentinel 会开始一次 自动的 故障转移操作，它会将与 失效主节点 是 主从关系 的其中一个 从节点 升级为新的 主节点，并且将其他的 从节点 指向 新的主节点。

1. 配置提供者

在 Redis Sentinel 模式下，客户端应用 在初始化时连接的是 Sentinel 节点集合，从中获取 主节点 的信息。

### 4.3. 主观下线和客观下线

默认情况下，每个 Sentinel 节点会以 每秒一次 的频率对 Redis 节点和 其它 的 Sentinel 节点发送 PING 命令，并通过节点的 回复 来判断节点是否在线。

1.主观下线

主观下线 适用于所有 主节点 和 从节点。如果在 down-after-milliseconds 毫秒内，Sentinel 没有收到 目标节点 的有效回复，则会判定 该节点 为 主观下线。

2.客观下线

客观下线 只适用于 主节点。如果 主节点 出现故障，Sentinel 节点会通过 sentinel is-master-down-by-addr 命令，向其它 Sentinel 节点询问对该节点的 状态判断。如果超过 <quorum> 个数的节点判定 主节点 不可达，则该 Sentinel 节点会判断 主节点 为 客观下线。

### 4.4. Sentinel的通信命令

Sentinel 节点连接一个 Redis 实例的时候，会创建 cmd 和 pub/sub 两个 连接。Sentinel 通过 cmd 连接给 Redis 发送命令，通过 pub/sub 连接到 Redis 实例上的其他 Sentinel 实例。

Sentinel 与 Redis 主节点 和 从节点 交互的命令，主要包括：

PING

Sentinel 向 Redis 节点发送 PING 命令，检查节点的状态

INFO

Sentinel 向 Redis 节点发送 INFO 命令，获取它的 从节点信息

PUBLISH

Sentinel 向其监控的 Redis 节点 \_\_sentinel\_\_:hello 这个 channel 发布 自己的信息 及 主节点 相关的配置

SUBSCRIBE

Sentinel 通过订阅 Redis 主节点 和 从节点 的 \_\_sentinel\_\_:hello 这个 channnel，获取正在监控相同服务的其他 Sentinel 节点

Sentinel 与 Sentinel 交互的命令，主要包括：

PING

Sentinel 向其他 Sentinel 节点发送 PING 命令，检查节点的状态

SENTINEL:is-master-down-by-addr

和其他 Sentinel 协商 主节点 的状态，如果 主节点 处于 SDOWN 状态，则投票自动选出新的 主节点

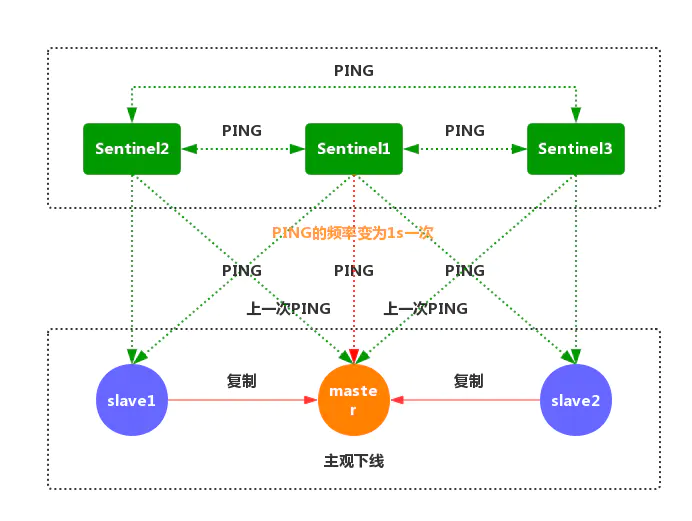
### 4.5. Redis Sentinel的工作原理

每个 Sentinel 节点都需要 定期执行 以下任务：

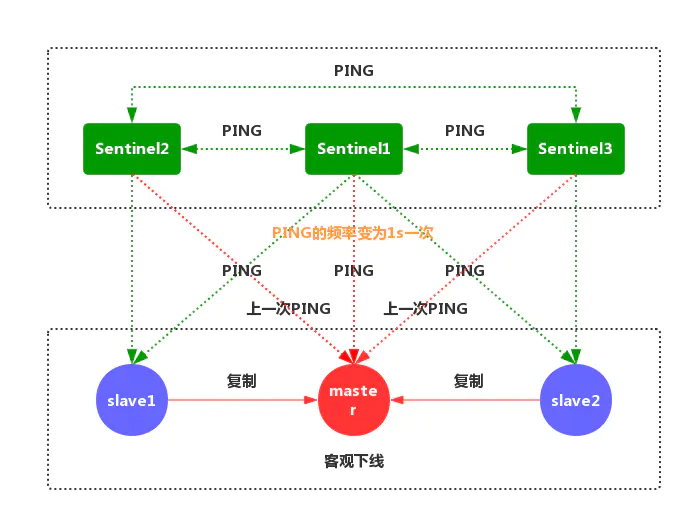
1.每个 Sentinel 以 每秒钟 一次的频率，向它所知的 主服务器、从服务器 以及其他 Sentinel 实例 发送一个 PING 命令。

2.如果一个 实例（instance）距离 最后一次 有效回复 PING 命令的时间超过 down-after-milliseconds 所指定的值，那么这个实例会被 Sentinel 标记为 主观下线。

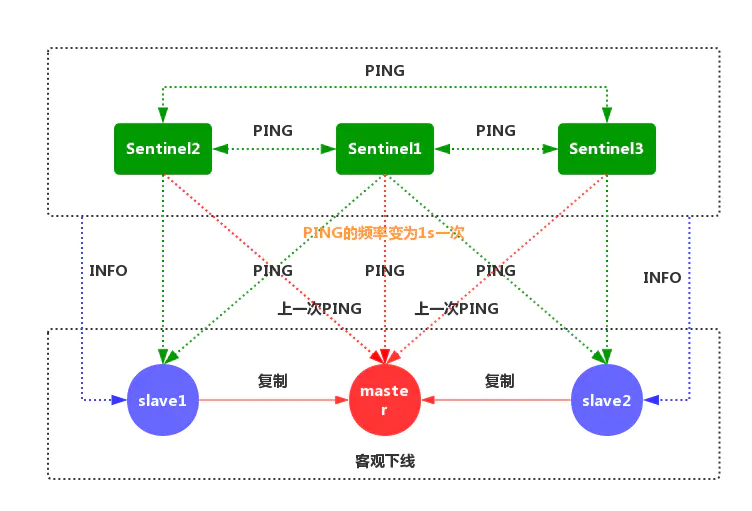
3.如果一个 主服务器 被标记为 主观下线，那么正在 监视 这个 主服务器 的所有 Sentinel 节点，要以 每秒一次 的频率确认 主服务器 的确进入了 主观下线 状态。



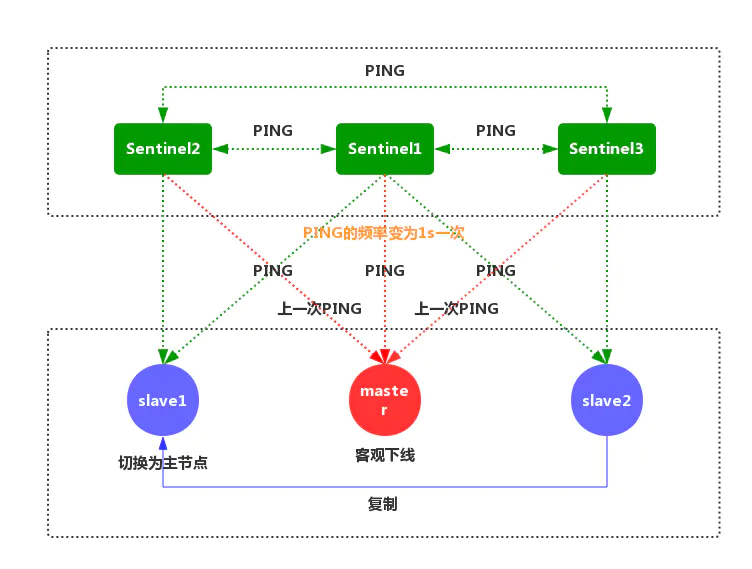
4.如果一个 主服务器 被标记为 主观下线，并且有 足够数量 的 Sentinel（至少要达到 配置文件 指定的数量）在指定的 时间范围 内同意这一判断，那么这个 主服务器 被标记为 客观下线。



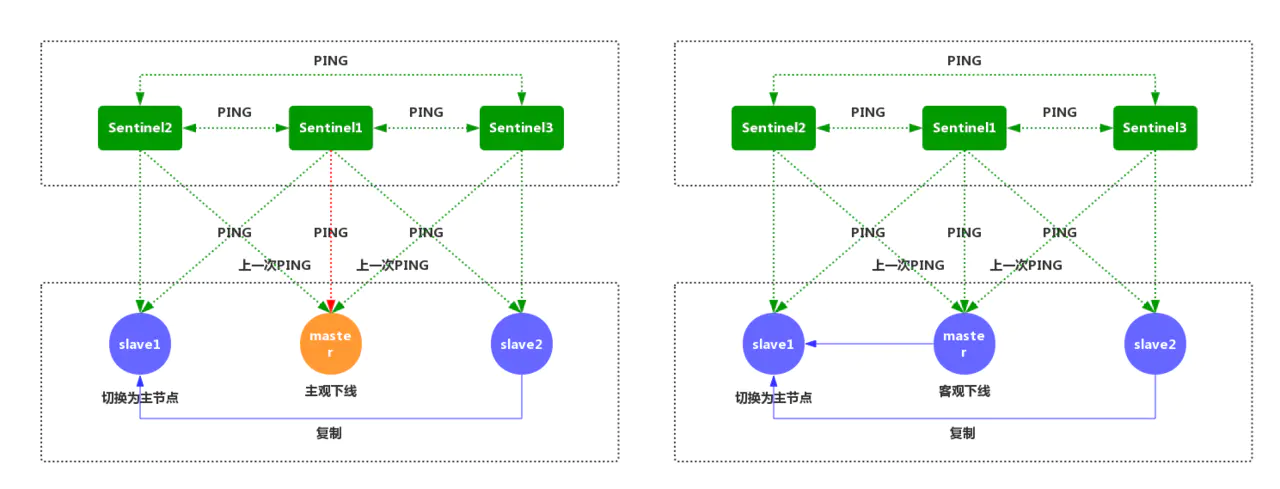
5.在一般情况下， 每个 Sentinel 会以每 10 秒一次的频率，向它已知的所有 主服务器 和 从服务器 发送 INFO 命令。当一个 主服务器 被 Sentinel 标记为 客观下线 时，Sentinel 向 下线主服务器 的所有 从服务器 发送 INFO 命令的频率，会从 10 秒一次改为 每秒一次。



6.Sentinel 和其他 Sentinel 协商 主节点 的状态，如果 主节点 处于 SDOWN 状态，则投票自动选出新的 主节点。将剩余的 从节点 指向 新的主节点 进行 数据复制。



7.当没有足够数量的 Sentinel 同意 主服务器 下线时， 主服务器 的 客观下线状态 就会被移除。当 主服务器 重新向 Sentinel 的 PING 命令返回 有效回复 时，主服务器 的 主观下线状态 就会被移除。



注意：一个有效的 PING 回复可以是：+PONG、-LOADING 或者 -MASTERDOWN。如果 服务器 返回除以上三种回复之外的其他回复，又或者在 指定时间 内没有回复 PING 命令， 那么 Sentinel 认为服务器返回的回复 无效（non-valid）。

## 5. Redis Sentinel搭建

5.1. Redis Sentinel的部署须知

一个稳健的 Redis Sentinel 集群，应该使用至少 三个 Sentinel 实例，并且保证讲这些实例放到 不同的机器 上，甚至不同的 物理区域。

Sentinel 无法保证 强一致性。

常见的 客户端应用库 都支持 Sentinel。

Sentinel 需要通过不断的 测试 和 观察，才能保证高可用。

下面主要介绍哨兵模式的搭建以及代码测试

搭建哨兵模式目录结构

└── redis

├── docker-compose.yaml

├── redis.conf

├── redis\_slave.conf

└── sentinel.conf

docker-compose.yaml

version: '3'

services:

redis-master:

image: redis:latest

container\_name: redis

restart: always

ports:

- 6379:6379

- 26379:26379

volumes:

- .\redis.conf:/usr/local/etc/redis/redis.conf:rw

- .\sentinel.conf:/usr/local/etc/redis/sentinel.conf:rw

networks:

my-web-networks:

ipv4\_address: 172.25.0.210

command:

- /bin/sh

- -c

- |

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

redis-server /usr/local/etc/redis/redis.conf

redis-Sentinel-slave1:

image: redis:latest

ports:

- 6380:6379

- 26380:26379

volumes:

- .\redis\_slave.conf:/usr/local/etc/redis/redis.conf:rw

- .\sentinel.conf:/usr/local/etc/redis/sentinel.conf:rw

networks:

my-web-networks:

ipv4\_address: 172.25.0.211

depends\_on:

- redis-master

command:

- /bin/sh

- -c

- |

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

redis-server /usr/local/etc/redis/redis.conf

redis-Sentinel-slave2:

image: redis:latest

ports:

- 6381:6379

- 26381:26379

volumes:

- .\redis\_slave.conf:/usr/local/etc/redis/redis.conf:rw

- .\sentinel.conf:/usr/local/etc/redis/sentinel.conf:rw

networks:

my-web-networks:

ipv4\_address: 172.25.0.212

depends\_on:

- redis-master

command:

- /bin/sh

- -c

- |

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

redis-server /usr/local/etc/redis/redis.conf

networks:

my-web-networks:

external: true

redis.conf

bind 0.0.0.0

protected-mode no

port 6379

timeout 0

save 900 1 # 900s内至少一次写操作则执行bgsave进行RDB持久化

save 300 10

save 60 10000

rdbcompression yes

dbfilename dump.rdb

dir /data

appendonly yes

appendfsync everysec

requirepass zhendao\_zhendao123

redis\_salve.conf

bind 0.0.0.0

protected-mode no

port 6379

timeout 0

save 900 1 # 900s内至少一次写操作则执行bgsave进行RDB持久化

save 300 10

save 60 10000

rdbcompression yes

dbfilename dump.rdb

dir /data

appendonly yes

appendfsync everysec

requirepass zhendao\_zhendao123

slaveof 172.25.0.210 6379

masterauth zhendao\_zhendao123

sentinel.conf

# 以后台守护进程的方式启动

daemonize yes

bind 0.0.0.0

port 26379

# 配置监听的主服务器，这里sentinel monitor代表监控，mymaster代表服务器的名称，可以自定义，192.168.91.134 代表监控的主服务器，6379代表端口，2代表只有两个或两个以上的哨兵认为主服务器不可用的时候，才会进行failover操作。

sentinel monitor mymaster 172.25.0.210 6379 2

# sentinel author-pa/requss定义服务的密码，mymaster是服务名称，123456是Redis服务器密码

# sentinel auth-pass <master-name> <password>

sentinel auth-pass mymaster zhendao\_zhendao123

在redis 目录下执行

docker-compose up -d

代码目录如下

Api/

|-- bin

| |-- cli

| `-- class

| |-- autoload.php

| |-- lib

| |-- MyRedis.php

| `-- Sentinel.php

|

`-- public

`-- testRedisSentinel.php

autoload.php

<?php

spl\_autoload\_register(function($classname){

$classnameForPath = implode(DIRECTORY\_SEPARATOR,explode('\\', $classname));

$file\_path = \_\_DIR\_\_.DIRECTORY\_SEPARATOR.$classnameForPath.'.php';

if(file\_exists($file\_path)){

require $file\_path;

}

});

MyRedis.php

<?php

namespace lib;

use lib\Sentinel;

class MyRedis{

private static $redisInstace = []; //redis对象

private static $sentinel = null; //sentinel对象

private static $readMode = 1; //1.主从均匀随机读，2.读只请求从，3.读写都走主

private static $AllNodeNum = 3; //总节点个数

private function \_\_construct(){ //不允许实利化

}

private function \_\_clone(){ //不允许克隆

}

public static function init($sentinelConfig){

self::$readMode = $sentinelConfig['readMode'];

self::$AllNodeNum = $sentinelConfig['AllNodeNum'];

Sentinel::setConfig($sentinelConfig);

}

public static function run($type){

$sentinel = self::$sentinel;

if($type == 'slave'){

$config = Sentinel::getSlaveConfig();

}else{

$config = Sentinel::getMasterConfig();

}

return self::getInstace($config);

}

public static function getInstace($config,$name=''){

$params = !empty($name) ? $name : md5(json\_encode($config));

if(!isset(self::$redisInstace[$params])){

self::$redisInstace[$params] = static::connectRedis($config);

}

return self::$redisInstace[$params];

}

private static function connectRedis($config) {

var\_dump($config);

$redis = new \Redis();

$port = $config['port'] ? $config['port'] : 6379;

$host = $config['host'] ? $config['host'] : '127.0.0.1';

$redis->connect($host, $port);

if ($config['password']) {

$redis->auth($config['password']);

}

return $redis;

}

public static function \_\_callStatic($name,$param) {

$type = 'master';

if(self::$readMode == 3){

$type = 'master';

}else{

if(self::$readMode == 1){

$n = rand(1,self::$AllNodeNum);

if($n == self::$AllNodeNum){

$type = 'master';

}else{

if(in\_array($name, ['get','hget','hmget']) && self::$AllNodeNum>1){

$type = 'slave';

}

}

}else{

if(in\_array($name, ['get','hget','hmget']) && self::$AllNodeNum>1){

$type = 'slave';

}

}

}

echo $type.PHP\_EOL;

return call\_user\_func\_array(array(self::run($type),$name),$param);

}

public static function gzSet($redis,$redis\_key,$redis\_data,$expire=0){

$redis->set($redis\_key, gzcompress(json\_encode($redis\_data)));

if($expire){

$redis->expire($redis\_key,$expire);

}

}

public static function gzGet($redis,$redis\_key){

$redis\_data = $redis->get($redis\_key);

if($redis\_data){

$redis\_data = gzuncompress($redis\_data);

$redis\_data = json\_decode($redis\_data,true);

}

return $redis\_data;

}

}

Sentinel.php

<?php

namespace lib;

class Sentinel{

private static $sentinelConfig = null; //sentinel配置

private function \_\_construct(){ //不允许实利化

}

private function \_\_clone(){ //不允许克隆

}

public static function setConfig($sentinelConfig){

self::$sentinelConfig = $sentinelConfig;

}

public static function getMasterConfig(){

if(empty(self::$sentinelConfig)){

exit('please use sentinel::setInstace($sentinelConfig); before sentinel::getMasterConfig()');

}

foreach (self::$sentinelConfig['sentinel\_config'] as $k => $v) {

//初始化redis对象

$redis = new \Redis();

try {

$status = @$redis->connect($v['host'], $v['port'],1);

if(!$status){

continue;

}

} catch (\Exception $exc) {

continue;

}

$result = $redis->rawCommand('SENTINEL', 'master', self::$sentinelConfig['master\_name']);

$master\_ip = $result[3];

$master\_port = $result[5];

return ['host'=>$master\_ip,'port'=>$master\_port,'password'=>self::$sentinelConfig['password']];

}

}

public static function getSlaveConfig(){

$allSlavesConfig = self::getSlaveAllConfig();

$num = count($allSlavesConfig);

$rand = rand(1,$num);

return $allSlavesConfig[$rand-1];

}

public static function getSlaveAllConfig(){

if(empty(self::$sentinelConfig)){

exit('please use sentinel::setInstace($sentinelConfig); before sentinel::getMasterConfig()');

}

foreach (self::$sentinelConfig['sentinel\_config'] as $k => $v) {

//初始化redis对象

$redis = new \Redis();

try {

$status = @$redis->connect($v['host'], $v['port'],1);

if(!$status){

continue;

}

} catch (\Exception $exc) {

continue;

}

$result = $redis->rawCommand('SENTINEL', 'slaves', self::$sentinelConfig['master\_name']);

$slaves = [];

foreach ($result as $k => $v) {

if($v[9]=='slave'){

$slaves\_ip = $v[3];

$slaves\_port = $v[5];

$slaves[] = ['host'=>$slaves\_ip,'port'=>$slaves\_port,'password'=>self::$sentinelConfig['password']];

}

}

return $slaves;

}

}

}

测试程序为 testRedisSentinel.php

<?php

include dirname(\_\_DIR\_\_) . '/bin/cli/class/autoload.php';

$config = ['password'=>'zhendao\_zhendao123',

'master\_name'=>'mymaster',

'AllNodeNum'=>3,

'readMode' => 1, //1.主从均匀随机读，2.读只请求从，3.读写都走主

'sentinel\_config'=>[

['host'=>'172.25.0.210','port'=>26379],

['host'=>'172.25.0.211','port'=>26379],

['host'=>'172.25.0.212','port'=>26379]

]

];

lib\MyRedis::init($config);

lib\MyRedis::set('k1','v1');

$res = lib\MyRedis::get('k1');

var\_dump($res);die;

这个测试没问题

这个时候主动将主容器退出，过一段时间它会被标记为一个退出的从，并重新选举了一个主，再次连接上了，原来的主还是从，一个正常的从库。

结论：在我们臻推宝系统，还没有非常大规模的数据，所以不需要搭建集群，使用复制或者哨兵模式就足够了，为了高可用推荐使用哨兵模式。

部署及测试代码是自己原创

本文原理性介绍参考了 [juejin.cn/...](https://juejin.cn/post/6844903663362637832)