课前准备

- 准备redis安装包

课堂主题

Redis持久化、Redis主从复制、Redis哨兵机制、Redis集群原理

课堂目标

- 掌握redis的RDB和AOF的原理和选型
- 理解Redis主从复制原理和哨兵原理
- 能够配置Redis主从+哨兵
- 理解RedisCluster的原理和容错机制
- 能够配置RedisCluster并使用

知识要点

Redis持久化

Redis是一个**内存**数据库，为了保证数据的持久性，它提供了两种持久化方案：

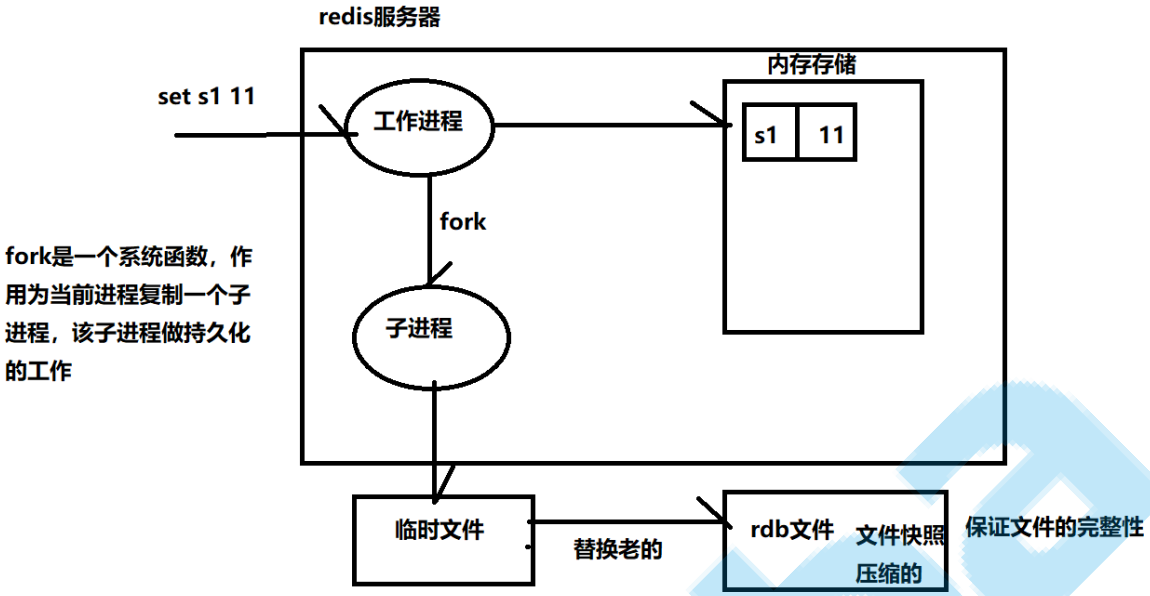
RDB方式（默认）

RDB方式是通过**快照**（`snapshotting`）完成的，当**符合一定条件**时Redis会自动将内存中的数据进行快照并持久化到硬盘。

触发快照的时机

1. 符合自定义配置的快照规则 redis.conf
2. 执行save或者bgsave命令
3. 执行flushall命令
4. 执行主从复制操作 (第一次)

原理图



设置快照规则

save 多少秒内 数据变了多少

save "" : 不使用RDB存储

save 900 1： 表示15分钟（900秒钟）内至少1个键被更改则进行快照。 save 300 10： 表示5分钟（300秒）内至少10个键被更改则进行快照。 save 60 10000： 表示1分钟内至少10000个键被更改则进行快照。

漏斗型

注意事项

1. `Redis` 在进行快照的过程中不会修改 `RDB` 文件，只有快照结束后才会将旧的文件替换成新的，也就是说任何时候 `RDB` 文件都是完整的。
2. 这就使得我们可以通过定时备份 `RDB` 文件来实现 `Redis` 数据库的备份， `RDB` 文件是 经过压缩的二进制文件，占用的空间会小于内存中的数据，更加利于传输。

RDB优缺点

- **缺点：**使用 `RDB` 方式实现持久化，一旦 `Redis` **异常退出，就会丢失最后一次快照以后更改的所有数据。**这个时候我们就需要根据具体的应用场景，通过组合设置自动快照条件的方式来将可能发生的数据损失控制在能够接受范围。如果数据相对来说比较重要，希望将损失降到最小，则可以使用 `AOF` 方式进行持久化
- **优点：** `RDB` 可以最大化 `Redis` 的性能：父进程在保存 `RDB` 文件时唯一要做的就是 `fork` 出一个子进程，然后这个子进程就会处理接下来的所有保存工作，父进程无需执行任何磁盘 `I/O` 操作。同时这个也是一个缺点，如果数据集比较大的时候， `fork` 可以能比较耗时，造成服务器在一段时间内停止处理客户端的请求；

AOF方式

默认情况下`Redis`没有开启`AOF`（`append only file`）方式的持久化。

开启`AOF`持久化后，每执行一条会**更改`Redis`中的数据的命令**，`Redis`就会将该命令写入硬盘中的`AOF`文件，这一过程显然**会降低`Redis`的性能**，但大部分情况下这个影响是能够接受的，另外使**用较快的硬盘可以提高`AOF`的性能。**

配置 redis.conf

```
# 可以通过修改redis.conf配置文件中的appendonly参数开启
appendonly yes

# AOF文件的保存位置和RDB文件的位置相同，都是通过dir参数设置的。
dir   ./

# 默认的文件名是appendonly.aof，可以通过appendfilename参数修改
appendfilename   appendonly.aof
```

RESP

Redis客户端使用RESP（Redis的序列化协议）协议与Redis的服务器端进行通信。 虽然该协议是专门为Redis设计的，但是该协议也可以用于其他 客户端-服务器 （Client-Server）软件项目。

1、间隔符号，在Linux下是\r\n，在Windows下是\n

2、简单字符串 Simple Strings, 以 "+"加号 开头

3、错误 Errors, 以"-"减号 开头

4、整数型 Integer， 以 ":" 冒号开头

5、大字符串类型 Bulk Strings, 以 "$"美元符号开头，长度限制512M

6、数组类型 Arrays，以 "*"星号开头

用SET命令来举例说明RESP协议的格式。

```
redis> SET mykey "Hello"
"OK"
```

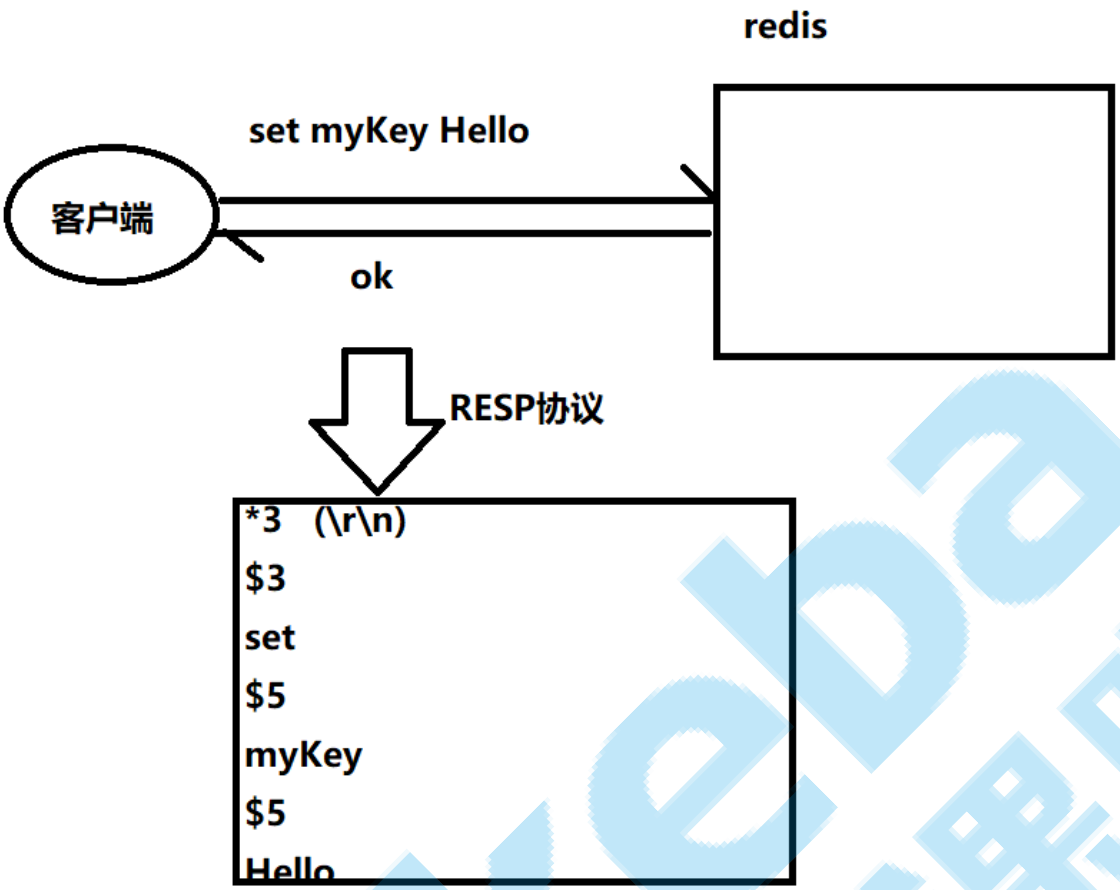
实际发送的请求数据：

```
*3\r\n$3\r\nSET\r\n$5\r\nmykey\r\n$5\r\nHello\r\n

*3
$3
SET
$5
mykey
$5
Hello
```

实际收到的响应数据：

```
+OK\r\n
```



AOF文件中存储的是redis的命令

原理

Redis 将所有对数据库进行过写入的命令（及其参数）记录到 AOF 文件， 以此达到记录数据库状态的目的， 为了方便起见， 我们称呼这种记录过程为同步。

同步命令到 AOF 文件的整个过程可以分为三个阶段：

命令传播：Redis 将执行完的命令、命令的参数、命令的参数个数等信息发送到 AOF 程序中。 缓存追加：AOF 程序根据接收到的命令数据，将命令转换为网络通讯协议的格式，然后将协议内容追加到服务器的 AOF 缓存中。 文件写入和保存：AOF 缓存中的内容被写入到 AOF 文件末尾，如果设定的 AOF 保存条件被满足的话， fsync 函数或者 fdatasync 函数会被调用，将写入的内容真正地保存到磁盘中。

命令传播

当一个 Redis 客户端需要执行命令时， 它通过网络连接， 将协议文本发送给 Redis 服务器。服务器在接到客户端的请求之后， 它会根据协议文本的内容， 选择适当的命令函数， 并将各个参数从字符串文本转换为 Redis 字符串对象（`StringObject`）。每当命令函数成功执行之后， 命令参数都会被传播到 AOF 程序。

缓存追加

当命令被传播到 AOF 程序之后， 程序会根据命令以及命令的参数， 将命令从字符串对象转换回原来的协议文本。协议文本生成之后， 它会被追加到 `redis.h/redisServer` 结构的 `aof_buf` 末尾。

`redisServer` 结构维持着 Redis 服务器的状态， `aof_buf` 域则保存着所有等待写入到 AOF 文件的协议文本。

文件写入和保存

每当服务器常规任务函数被执行、 或者事件处理器被执行时， aof.c/flushAppendOnlyFile 函数都会被调用， 这个函数执行以下两个工作：

WRITE：根据条件，将 aof_buf 中的缓存写入到 AOF 文件。

SAVE：根据条件，调用 fsync 或 fdatasync 函数，将 AOF 文件保存到磁盘中。

AOF 保存模式

Redis 目前支持三种 AOF 保存模式，它们分别是：

AOF_FSYNC_NO ： 不保存。 AOF_FSYNC_EVERYSEC ： 每一秒钟保存一次。 （默认）
AOF_FSYNC_ALWAYS ： 每执行一个命令保存一次。 （不推荐） 以下三个小节将分别讨论这三种保存模式。

不保存

在这种模式下， 每次调用 flushAppendOnlyFile 函数， WRITE 都会被执行， 但 SAVE 会被略过。

在这种模式下， SAVE 只会在以下任意一种情况中被执行：

Redis 被关闭 AOF 功能被关闭 系统的写缓存被刷新（可能是缓存已经被写满，或者定期保存操作被执行） 这三种情况下的 SAVE 操作都会引起 Redis 主进程阻塞。

每一秒钟保存一次

在这种模式中， SAVE 原则上每隔一秒钟就会执行一次， 因为 SAVE 操作是由后台子线程调用的， 所以它不会引起服务器主进程阻塞。

每执行一个命令保存一次

在这种模式下，每次执行完一个命令之后， WRITE 和 SAVE 都会被执行。

另外，因为 SAVE 是由 Redis 主进程执行的，所以在 SAVE 执行期间，主进程会被阻塞，不能接受命令请求。

AOF 保存模式对性能和安全性的影响

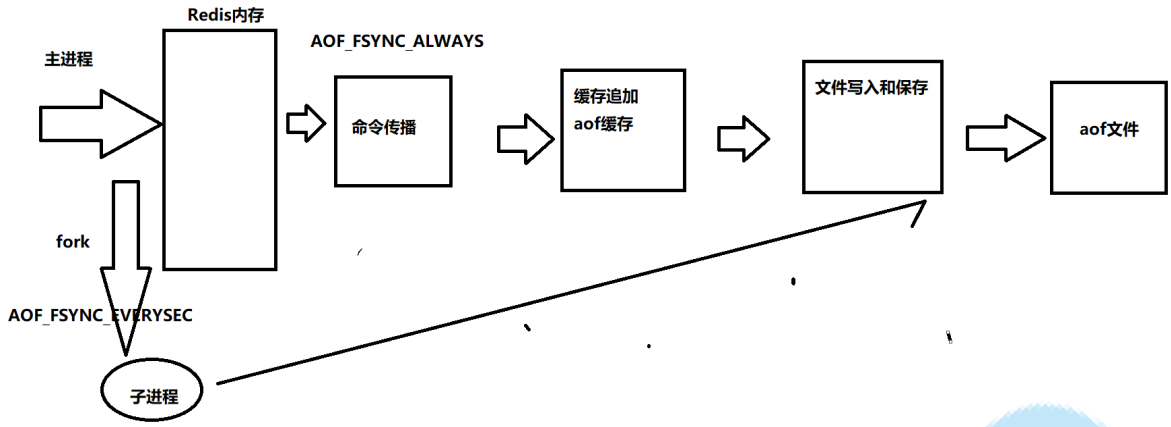
对于三种 AOF 保存模式， 它们对服务器主进程的阻塞情况如下：

不保存（AOF_FSYNC_NO）：写入和保存都由主进程执行，两个操作都会阻塞主进程。 每一秒钟保存一次（AOF_FSYNC_EVERYSEC）：写入操作由主进程执行，阻塞主进程。保存操作由子线程执行，不直接阻塞主进程，但保存操作完成的快慢会影响写入操作的阻塞时长。 每执行一个命令保存一次（AOF_FSYNC_ALWAYS）：和模式 1 一样。 因为阻塞操作会让 Redis 主进程无法持续处理请求， 所以一般说来， 阻塞操作执行得越少、完成得越快， Redis 的性能就越好。

模式 1 的保存操作只会在AOF 关闭或 Redis 关闭时执行， 或者由操作系统触发， 在一般情况下， 这种模式只需要为写入阻塞， 因此它的写入性能要比后面两种模式要高， 当然， 这种性能的提高是以降低安全性为代价的： 在这种模式下， 如果运行的中途发生停机， 那么丢失数据的数量由操作系统的缓存冲洗策略决定。

模式 2 在性能方面要优于模式 3 ， 并且在通常情况下， 这种模式最多丢失不多于 2 秒的数据， 所以它的安全性要高于模式 1 ， 这是一种兼顾性能和安全性的保存方案。

模式 3 的安全性是最高的， 但性能也是最差的， 因为服务器必须阻塞直到命令信息被写入并保存到磁盘之后， 才能继续处理请求。



综合起来，三种 AOF 模式的操作特性可以总结如下：

模式	WRITE 是否阻塞?	SAVE 是否阻塞?	停机时丢失的数据量
AOF_FSYNC_NO	阻塞	阻塞	操作系统最后一次对 AOF 文件触发 SAVE 操作之后的数据。
AOF_FSYNC_EVERYSEC	阻塞	不阻塞	一般情况下不超过 2 秒钟的数据。
AOF_FSYNC_ALWAYS	阻塞	阻塞	最多只丢失一个命令的数据。

AOF重写原理（优化AOF文件）

Redis 可以在 AOF **文件体积变得过大**时，自动地在后台（Fork子进程）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**。 所谓的"重写"其实是一个有歧义的词语， 实际上， AOF 重写并不需要对原有的 AOF 文件进行任何写入和读取， 它针对的是数据库中键的当前值。

举例如下：

set s1 11

set s1 22 -------> set s1 33

set s1 33

没有优化的：

set s1 11

set s1 22

set s1 33

优化后：

set s1 33

lpush list1 1 2 3

lpush list1 4 5 6 --------> list1 1 2 3 4 5 6

优化后

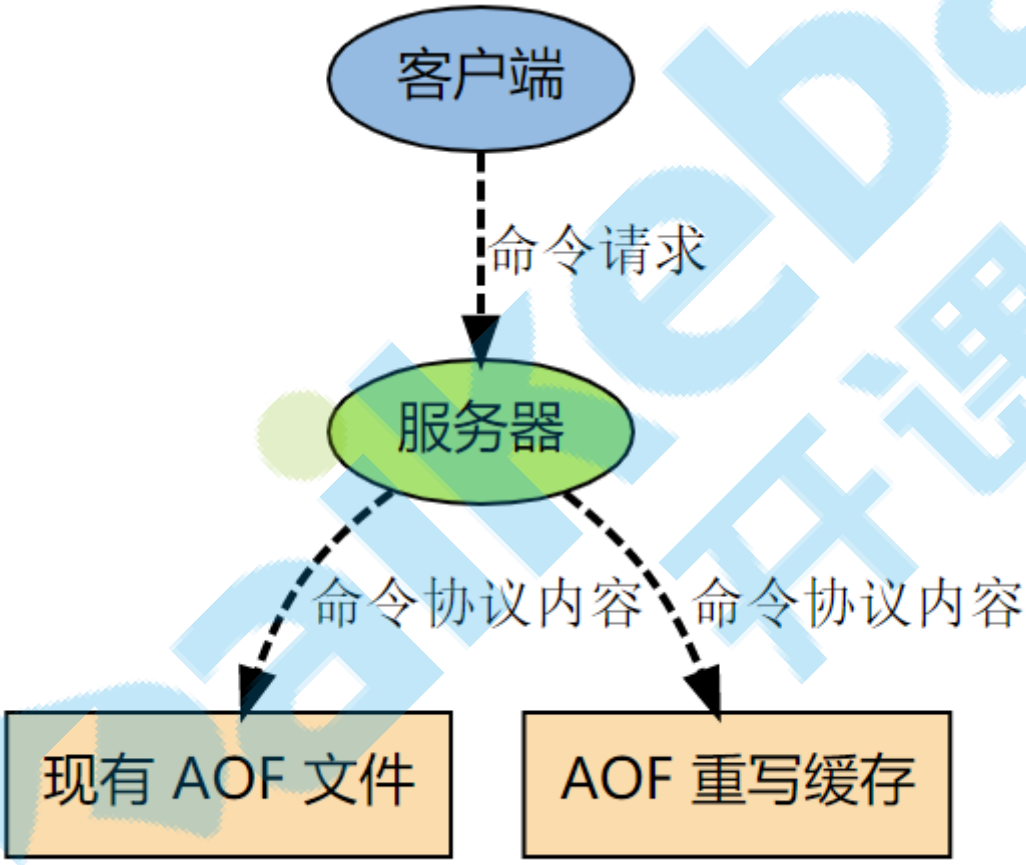
lpush list1 1 2 3 4 5 6

Redis 不希望 AOF 重写造成服务器无法处理请求， 所以 Redis 决定将 AOF 重写程序放到（后台）子进程里执行， 这样处理的最大好处是：

1、子进程进行 AOF 重写期间，主进程可以继续处理命令请求。 2、子进程带有主进程的数据副本，使用子进程而不是线程，可以在避免锁的情况下，保证数据的安全性。

不过， 使用子进程也有一个问题需要解决： 因为子进程在进行 AOF 重写期间， 主进程还需要继续处理命令， 而新的命令可能对现有的数据进行修改， 这会让当前数据库的数据和重写后的 AOF 文件中的数据不一致。

为了解决这个问题， Redis 增加了一个 AOF 重写缓存， 这个缓存在 fork 出子进程之后开始启用， Redis 主进程在接到新的写命令之后， 除了会将这个写命令的协议内容追加到现有的 AOF 文件之外，还会追加到这个缓存中。



重写过程分析（整个重写操作是绝对安全的）：

`Redis` 在创建**新** `AOF` **文件**的过程中，会继续将命令追加到**现有的** `AOF` **文件**里面，即使重写过程中发生停机，现有的 `AOF` 文件也不会丢失。 而一旦**新** `AOF` **文件**创建完毕，`Redis` 就会从**旧** `AOF` **文件**切换到**新** `AOF` **文件**，并开始对**新** `AOF` **文件**进行追加操作。

当子进程在执行 AOF 重写时， 主进程需要执行以下三个工作：

处理命令请求。 将写命令追加到现有的 AOF 文件中。 将写命令追加到 AOF 重写缓存中。 这样一来可以保证：

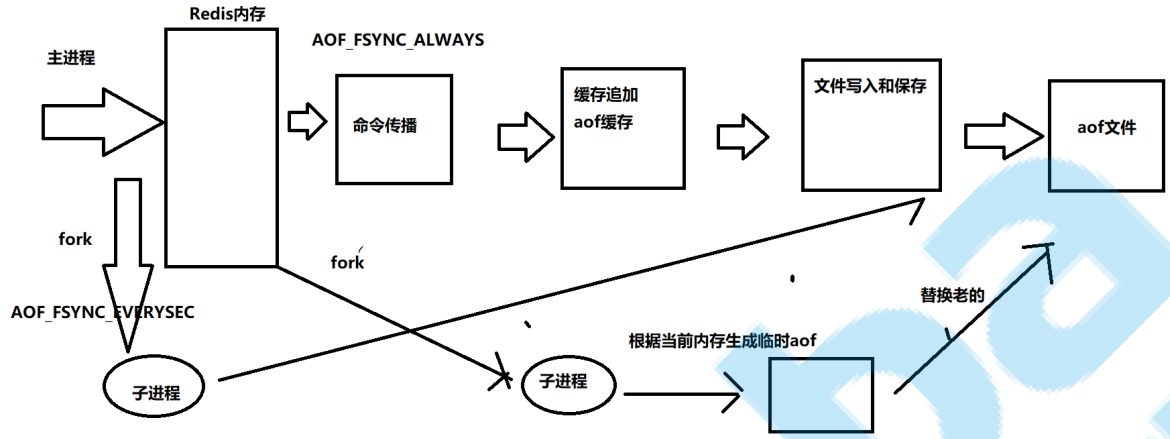
现有的 AOF 功能会继续执行，即使在 AOF 重写期间发生停机，也不会有任何数据丢失。 所有对数据库进行修改的命令都会被记录到 AOF 重写缓存中。 当子进程完成 AOF 重写之后， 它会向父进程发送一个完成信号， 父进程在接到完成信号之后， 会调用一个信号处理函数， 并完成以下工作：

将 AOF 重写缓存中的内容全部写入到新 AOF 文件中。 对新的 AOF 文件进行改名，覆盖原有的 AOF 文件。 当步骤 1 执行完毕之后， 现有 AOF 文件、新 AOF 文件和数据库三者的状态就完全一致了。

当步骤 2 执行完毕之后， 程序就完成了新旧两个 AOF 文件的交替。

这个信号处理函数执行完毕之后， 主进程就可以继续像往常一样接受命令请求了。 在整个 AOF 后台重写过程中， 只有最后的写入缓存和改名操作会造成主进程阻塞， 在其他时候， AOF 后台重写都不会对主进程造成阻塞， 这将 AOF 重写对性能造成的影响降到了最低。

以上就是 AOF 后台重写， 也即是 BGREWRITEAOF 命令(AOF重写)的工作原理。



优化触发条件：

```
# 表示当前aof文件大小超过上一次aof文件大小的百分之多少的时候会进行重写。如果之前没有重写过，以启动时aof文件大小为准
auto-aof-rewrite-percentage 100

# 限制允许重写最小aof文件大小，也就是文件大小小于64mb的时候，不需要进行优化
auto-aof-rewrite-min-size 64mb
```

Redis应用场景

内存数据库 不存在DB中 登录信息、浏览记录、购物车

缓存服务器 缓存DB信息减少DB压力，商品数据信息 代码表

session存储 多个应用服务器

任务队列 list 秒杀 请求限流

分布式锁 set nx

应用排行 zset

数据过期 冷热数据 expire

如何选择RDB和AOF

内存数据库 rdb（redis database）+aof 数据不能丢

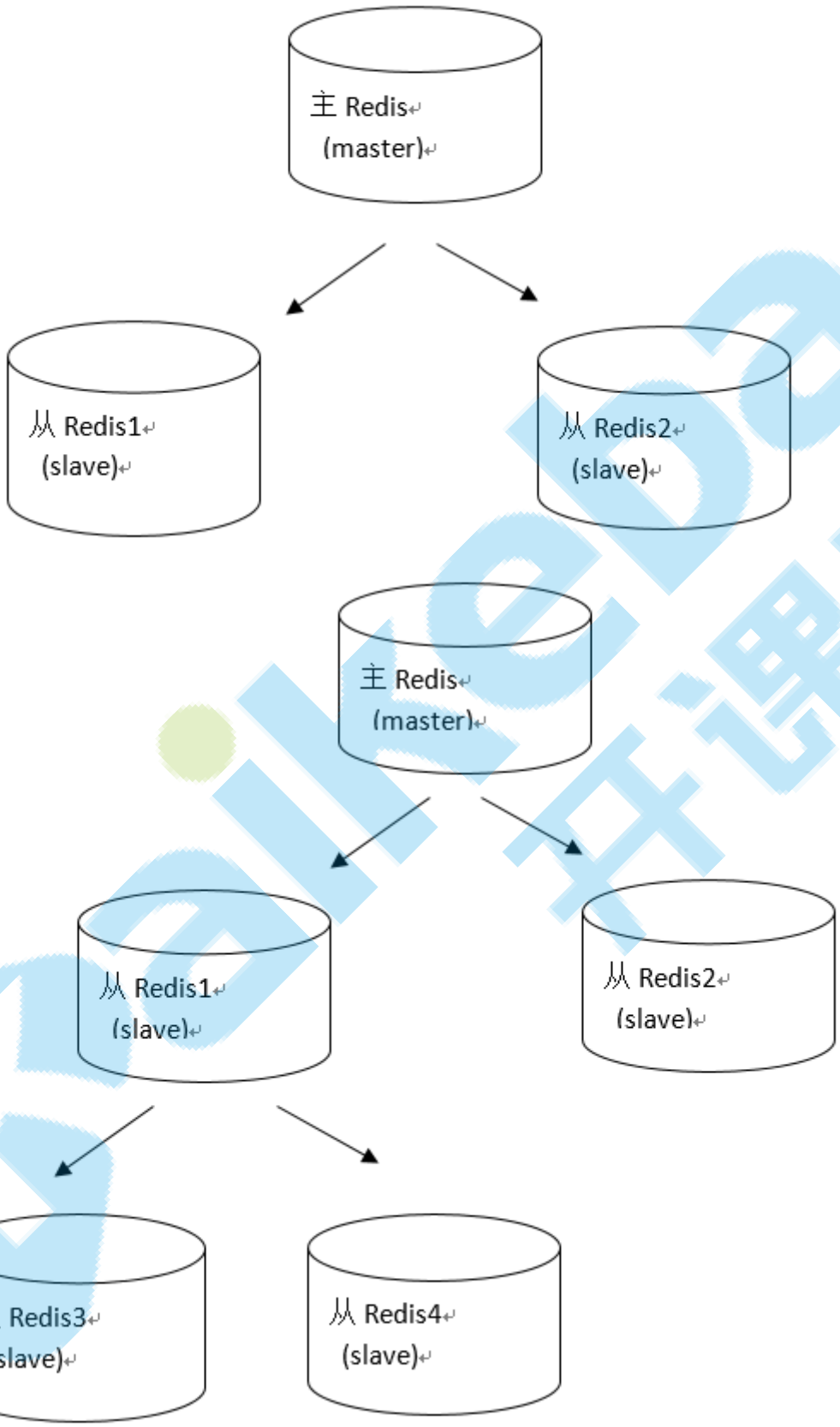
缓存服务器 rdb

不建议 只使用 aof (性能差)

恢复时： 先aof再rdb

Redis主从复制

什么是主从复制



主对外从对内，主可写从不可写

主挂了，从不可为主

主从配置

主Redis配置

无需特殊配置

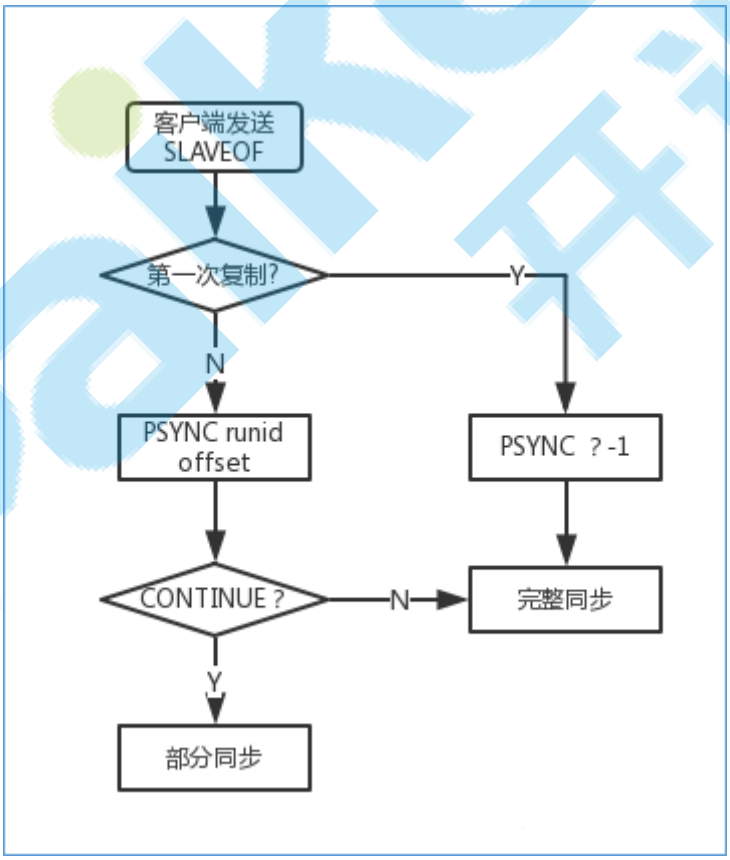
从Redis配置

修改从服务器上的`redis.conf`文件：

```
# slaveof <masterip> <masterport>
# 表示当前【从服务器】对应的【主服务器】的IP是192.168.10.135，端口是6379。
slaveof 127.0.0.1 6379
```

实现原理

- `Redis`的主从同步，分为**全量同步**和**增量同步**。
- 只有从机第一次连接上主机是**全量同步**。
- 断线重连有可能触发**全量同步**也有可能是**增量同步**（`master`判断`runid`是否一致）。



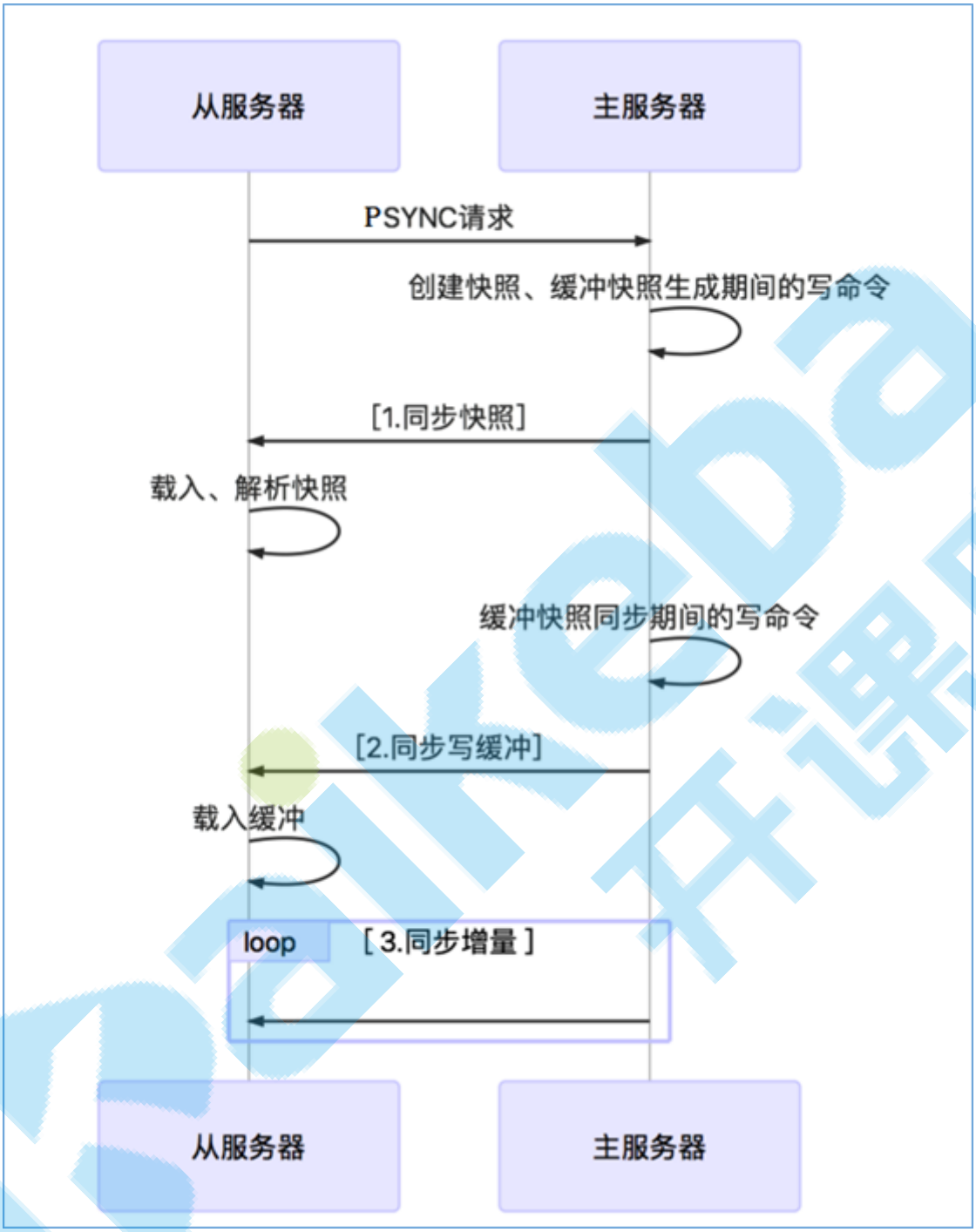
- 除此之外的情况都是**增量同步**。

全量同步

`Redis`的全量同步过程主要分三个阶段：

- **同步快照阶段：**`Master`创建并发送**快照**RDB给`Slave`，`Slave`载入并解析快照。`Master`同时将此阶段所产生的新的写命令存储到缓冲区。

- **同步写缓冲阶段：** `Master` 向 `Slave` 同步存储在缓冲区的写操作命令。
- **同步增量阶段：** `Master` 向 `Slave` 同步写操作命令。



增量同步

- Redis `增量同步主要指**` Slave `完成初始化后开始正常工作时，` Master `发生的写操作同步到` Slave `的过程**。
- 通常情况下，`Master` 每执行一个写命令就会向 `Slave` 发送相同的**写命令**，然后 `Slave` 接收并执行。

Redis哨兵机制

Redis主从复制的缺点：没有办法对master进行动态选举（master宕机后，需要重新选举master），需要使用Sentinel机制完成动态选举。

简介

Redis的哨兵模式到了2.8`版本之后

Sentinel(哨兵)进程是用于**监控Redis集群中Master主服务器工作的状态

在Master主服务器发生故障的时候，可以实现Master和Slave服务器的切换，保证系统的高可用（HA）

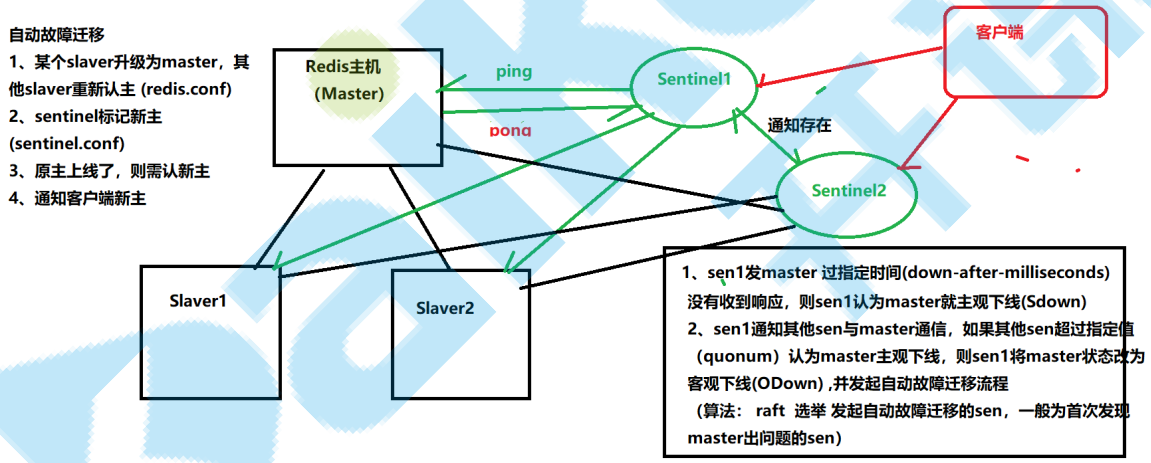
哨兵进程的作用

监控(Monitoring):哨兵(sentinel)会不断地检查你的Master和Slave是否运作正常。

提醒(Notification)： 当被监控的某个Redis节点出现问题时,哨兵(sentinel)可以通过API向管理员或者其他应用程序发送通知。

自动故障迁移(Automatic failover)：**当**一个Master不能正常工作**时，哨兵(sentinel)会开始**一次自动故障迁移操作

故障判定原理分析



1. 每个Sentinel（哨兵）进程以**每秒钟一次**的频率向整个集群中的Master主服务器，Slave从服务器以及其他Sentinel（哨兵）进程发送一个PING命令。
2. 如果一个实例（instance）距离最后一次有效回复PING命令的时间超过down-after-milliseconds选项所指定的值，则这个实例会被Sentinel（哨兵）进程标记为**主观下线**（SDOWN）。
3. 如果一个Master主服务器被标记为主观下线（SDOWN），则正在监视这个Master主服务器的**所有Sentinel（哨兵）**进程要以每秒一次的频率**确认Master主服务器**的确**进入了主观下线状态。**
4. 当**有足够数量的Sentinel（哨兵）**进程（大于等于配置文件指定的值）在指定的时间范围内确认Master主服务器进入了主观下线状态（SDOWN），则Master主服务器会被标记为**客观下线（ODOWN）**。
5. 在一般情况下，每个Sentinel（哨兵）进程会以每10秒一次的频率向集群中的所有Master主服务器、Slave从服务器发送INFO命令。
6. 当Master主服务器被Sentinel（哨兵）进程标记为**客观下线（ODOWN）**时，Sentinel（哨兵）进程向下线的Master主服务器的所有Slave从服务器发送INFO命令的频率会从10秒一次改为每秒一次。

7. 若没有足够数量的 `Sentinel`（哨兵）进程同意 `Master` 主服务器下线， `Master` 主服务器的客观下线状态就会被移除。若 `Master` 主服务器重新向 `Sentinel`（哨兵）进程发送 `PING` 命令返回有效回复， `Master` 主服务器的主观下线状态就会被移除。

自动故障迁移

- 它会将失效 `Master` 的其中一个 `Slave` 升级为新的 `Master`，并让失效 `Master` 的其他 `Slave` 改为复制新的 `Master`；
- 当客户端试图连接失效的 `Master` 时，集群也会向客户端返回新 `Master` 的地址，使得集群可以使用现在的 `Master` 替换失效 `Master`。
- `Master` 和 `Slave` 服务器切换后， `Master` 的 `redis.conf`、 `Slave` 的 `redis.conf` 和 `sentinel.conf` 的配置文件的内容都会发生相应的改变，即， `Master` 主服务器的 `redis.conf` 配置文件中会多一行 `slaveof` 的配置， `sentinel.conf` 的监控目标会随之调换。

案例演示

- 修改从机的 `sentinel.conf`：

```
# 哨兵sentinel监控的redis主节点的 ip port
# master-name  可以自己命名的主节点名字 只能由字母A-z、数字0-9 、这三个字符".-_"组成。
# quorum 当这些quorum个数sentinel哨兵认为master主节点失联 那么这时 客观上认为主节点失联了
# sentinel monitor <master-name> <master ip> <master port> <quorum>
sentinel monitor mymaster 192.168.10.133 6379 1
```

- **其他配置项说明**

`sentinel.conf`

```
# 哨兵sentinel实例运行的端口 默认26379
port 26379

# 哨兵sentinel的工作目录
dir /tmp

# 哨兵sentinel监控的redis主节点的 ip port
# master-name  可以自己命名的主节点名字 只能由字母A-z、数字0-9 、这三个字符".-_"组成。
# quorum 当这些quorum个数sentinel哨兵认为master主节点失联 那么这时 客观上认为主节点失联了
# sentinel monitor <master-name> <ip> <redis-port> <quorum>
sentinel monitor mymaster 127.0.0.1 6379 1

# 当在Redis实例中开启了requirepass foobared 授权密码 这样所有连接Redis实例的客户端都要提供密码
# 设置哨兵sentinel 连接主从的密码 注意必须为主从设置一样的验证密码
# sentinel auth-pass <master-name> <password>
sentinel auth-pass mymaster MySUPER--secret-0123passw0rd


# 指定多少毫秒之后 主节点没有应答哨兵sentinel 此时 哨兵主观上认为主节点下线 默认30秒
# sentinel down-after-milliseconds <master-name> <milliseconds>
sentinel down-after-milliseconds mymaster 30000

# 这个配置项指定了在发生failover主备切换时最多可以有多少个slave同时对新的master进行 同步，这个数字越小，完成failover所需的时间就越长，
```

```
但是如果这个数字越大，就意味着越 多的slave因为replication而不可用。
可以通过将这个值设为 1 来保证每次只有一个slave 处于不能处理命令请求的状态。
# sentinel parallel-syncs <master-name> <numslaves>
sentinel parallel-syncs mymaster 1



# 故障转移的超时时间 failover-timeout 可以用在以下这些方面：
#1. 同一个sentinel对同一个master两次failover之间的间隔时间。
#2. 当一个slave从一个错误的master那里同步数据开始计算时间。直到slave被纠正为向正确的master那里同步数据时。
#3.当想要取消一个正在进行的failover所需要的时间。
#4.当进行failover时，配置所有slaves指向新的master所需的最大时间。不过，即使过了这个超时，slaves依然会被正确配置为指向master，但是就不按parallel-syncs所配置的规则来了
# 默认三分钟
# sentinel failover-timeout <master-name> <milliseconds>
sentinel failover-timeout mymaster 180000

# SCRIPTS EXECUTION

#配置当某一事件发生时所需要执行的脚本，可以通过脚本来通知管理员，例如当系统运行不正常时发邮件通知相关人员。
#对于脚本的运行结果有以下规则：
#若脚本执行后返回1，那么该脚本稍后将会被再次执行，重复次数目前默认为10
#若脚本执行后返回2，或者比2更高的一个返回值，脚本将不会重复执行。
#如果脚本在执行过程中由于收到系统中断信号被终止了，则同返回值为1时的行为相同。
#一个脚本的最大执行时间为60s，如果超过这个时间，脚本将会被一个SIGKILL信号终止，之后重新执行。

#通知型脚本:当sentinel有任何警告级别的事件发生时（比如说redis实例的主观失效和客观失效等等），将会去调用这个脚本，这时这个脚本应该通过邮件，SMS等方式去通知系统管理员关于系统不正常运行的信息。调用该脚本时，将传给脚本两个参数，一个是事件的类型，一个是事件的描述。
#如果sentinel.conf配置文件中配置了这个脚本路径，那么必须保证这个脚本存在于这个路径，并且是可执行的，否则sentinel无法正常启动成功。
#通知脚本
# sentinel notification-script <master-name> <script-path>
sentinel notification-script mymaster /var/redis/notify.sh

# 客户端重新配置主节点参数脚本
# 当一个master由于failover而发生改变时，这个脚本将会被调用，通知相关的客户端关于master地址已经发生改变的信息。
# 以下参数将会在调用脚本时传给脚本:
# <master-name> <role> <state> <from-ip> <from-port> <to-ip> <to-port>
# 目前<state>总是“failover”,
# <role>是“leader”或者“observer”中的一个。
# 参数 from-ip, from-port, to-ip, to-port是用来和旧的master和新的master(即旧的slave)通信的
# 这个脚本应该是通用的，能被多次调用，不是针对性的。
# sentinel client-reconfig-script <master-name> <script-path>
 sentinel client-reconfig-script mymaster /var/redis/reconfig.sh
```

- 通过 `redis-sentinel` 启动哨兵服务

```
./redis-sentinel sentinel.conf
```

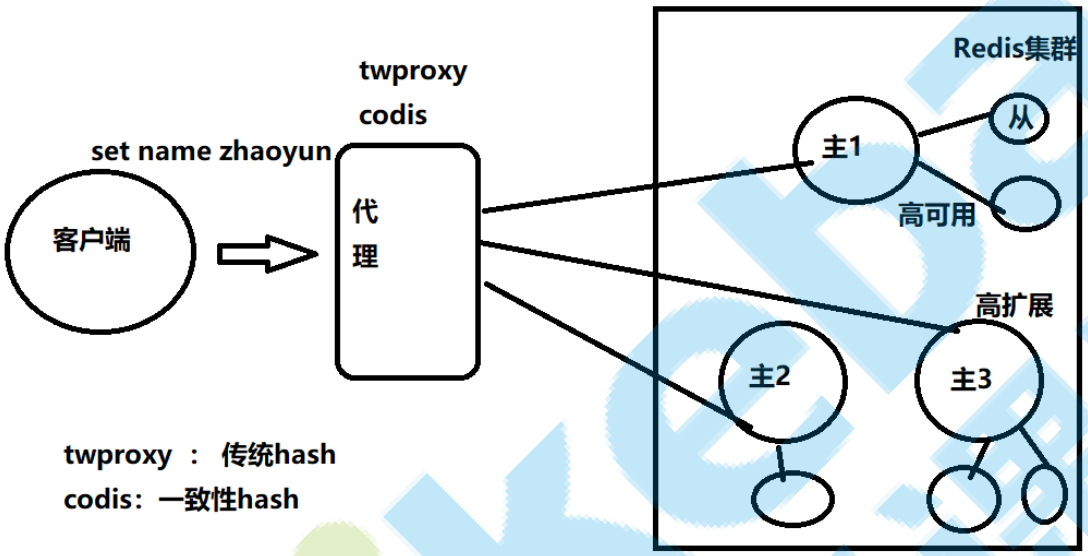
Redis集群

Redis的集群策略

twproxy

codis（豌豆荚）

代理方案



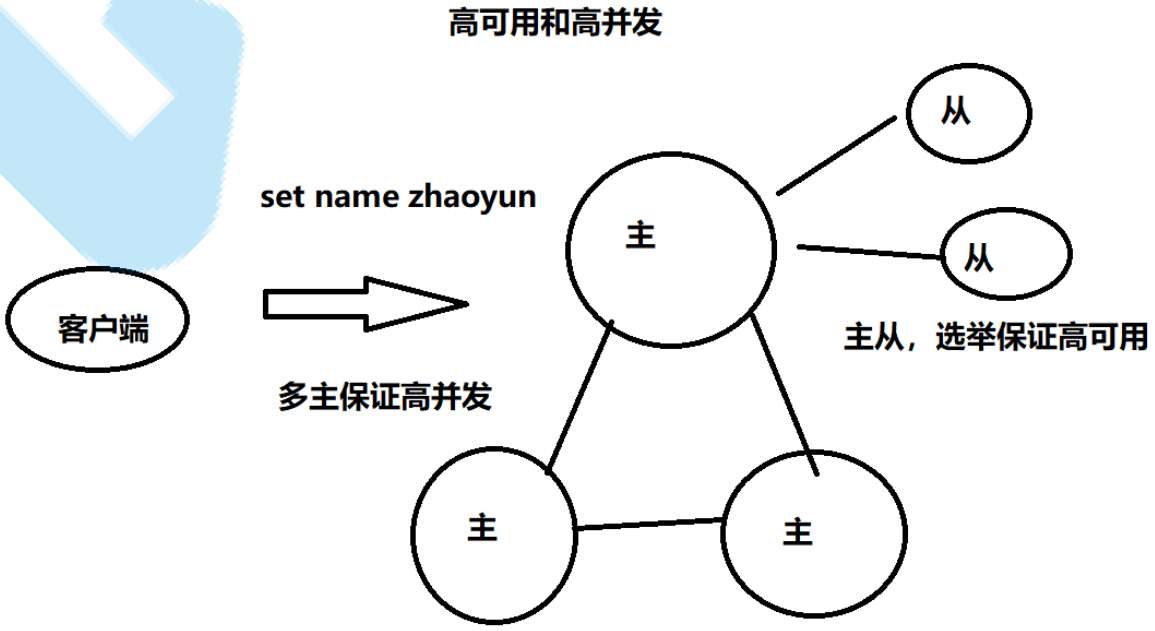
缺点：健壮性相对差，性能相对差

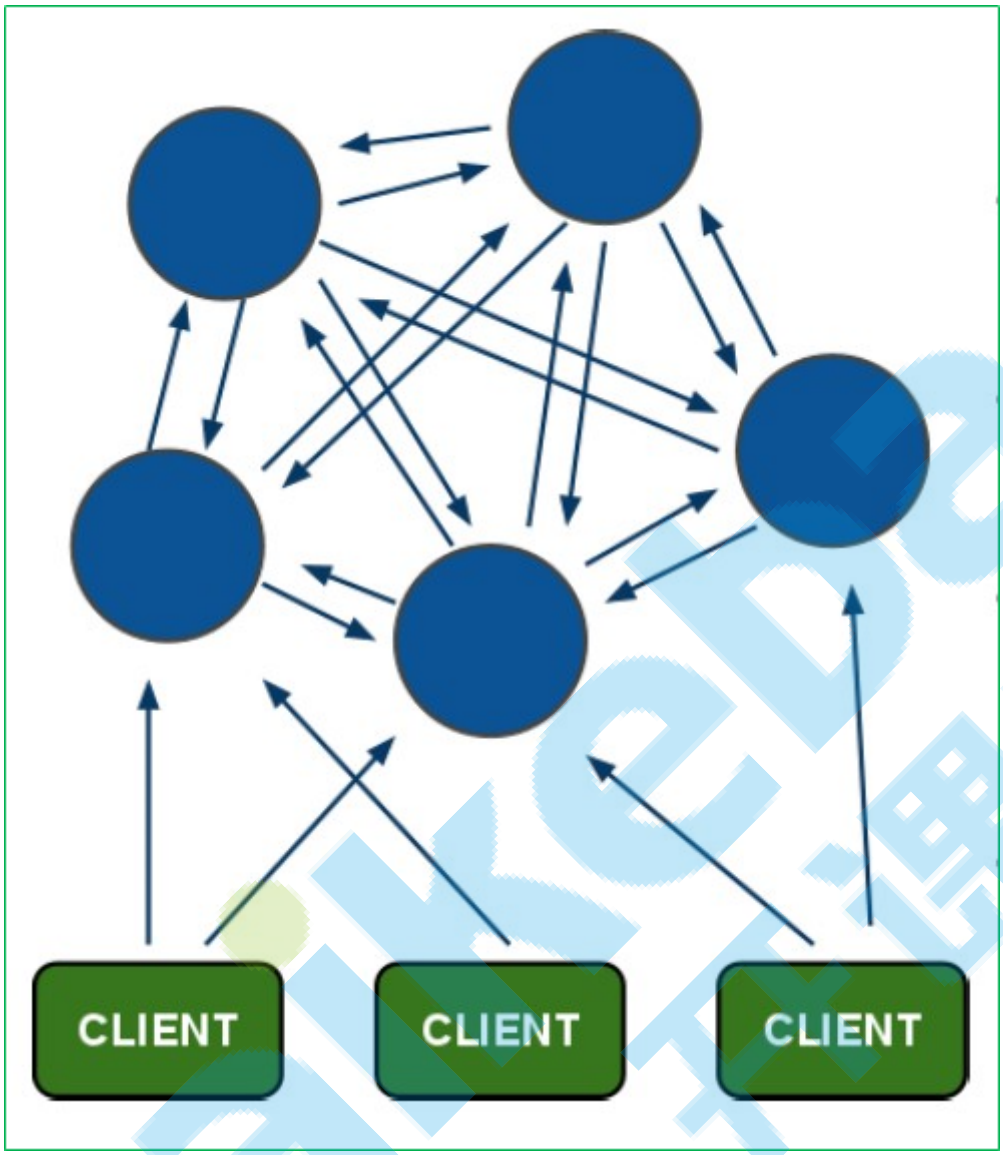
Redis-cluster架构图

Redis3以后，官方的集群方案 Redis-Cluster

Redis3 使用lua脚本实现

Redis5 直接实现





架构细节:

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

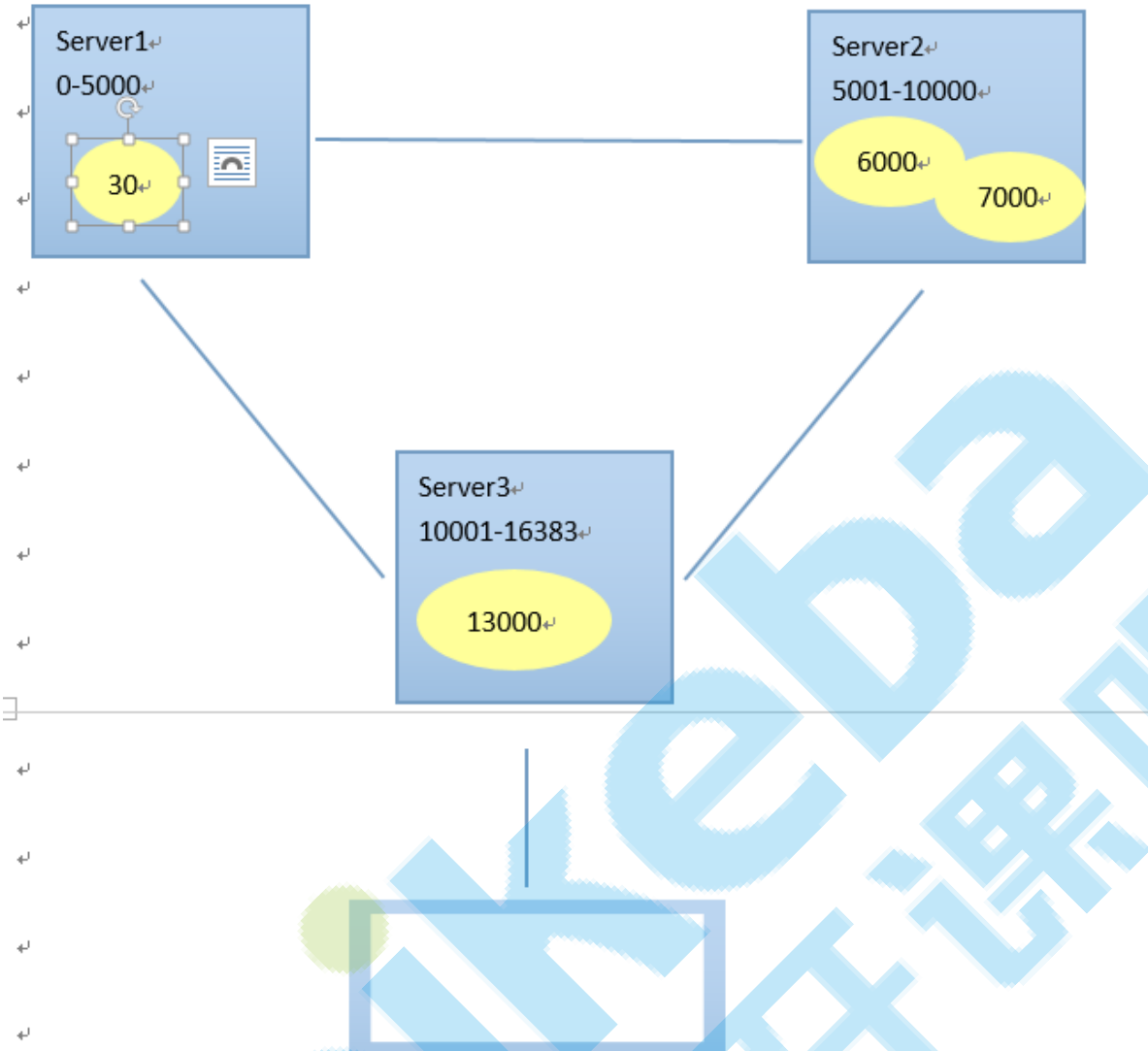
(2)节点的fail是通过集群中超过半数的节点检测失效时才生效.

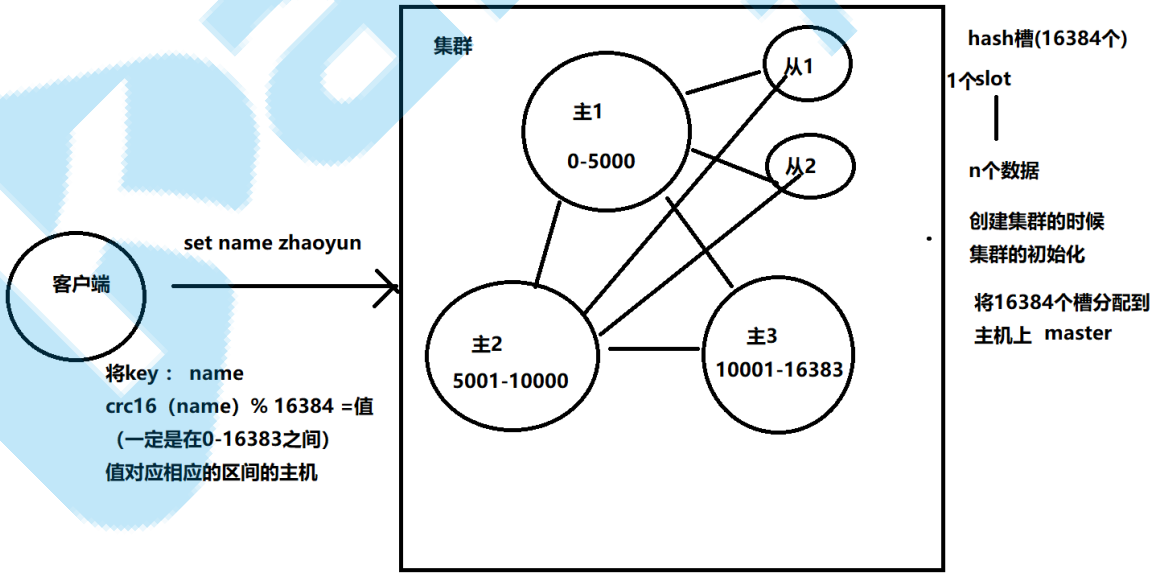
(3)客户端与redis节点直连,不需要中间proxy层.客户端不需要连接集群所有节点,连接集群中任何一个可用节点即可

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

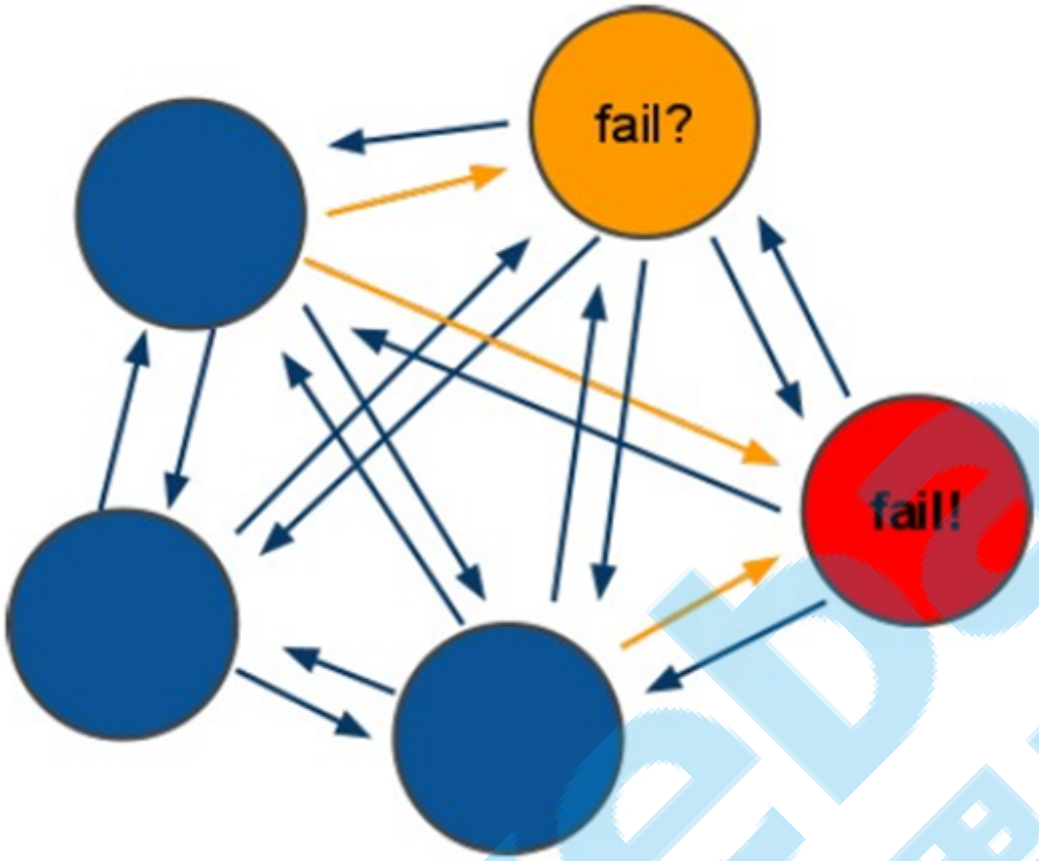
```
Redis 集群中内置了 16384个哈希槽，当需要在 Redis 集群中放置一个 key-value 时，redis 先对 key 使用 crc16 算法算出一个结果，然后把结果对 16384 求余数，这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽，redis 会根据节点数量大致均等的将哈希槽映射到不同的节点
```

示例如下:





Redis-cluster投票:容错



1、主节点投票，如果超过半数的主都认为某主down了，则该主就down了 （主选择单数）

2、主节点投票，选出挂了的主的从升级为主

注：

集群挂了的情况：

1、半数的主挂了，不能投票生效，则集群挂了

2、挂了的主机的从也挂了，造成slot槽分配不连续（16384不能完全分配），集群就挂了

安装RedisCluster

> Redis集群最少需要三台主服务器，三台从服务器。
>
> 端口号分别为：7001~7006

- 第一步：创建7001实例，并编辑redis.conf文件，修改port为7001。

  注意：创建实例，即拷贝单机版安装时，生成的bin目录，为7001目录。

```
# Accept connections on the specified port, default is 6379.
# If port 0 is specified Redis will not listen on a TCP socket.
port 7001
```

- 第二步：修改redis.conf配置文件，打开cluster-enable yes

- 第三步：复制7001，创建7002~7006实例，**注意端口修改。**
- 第四步：创建start.sh，启动所有的实例

```
cd 7001
./redis-server redis.conf
cd ..

cd 7002
./redis-server redis.conf
cd ..

cd 7003
./redis-server redis.conf
cd ..

cd 7004
./redis-server redis.conf
cd ..

cd 7005
./redis-server redis.conf
cd ..

cd 7006
./redis-server redis.conf
cd ..
```

chomd u+x start.sh

- 第五步：创建Redis集群

```
[root@localhost 7001]# ./redis-cli --cluster create 127.0.0.1:7001
127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005 127.0.0.1:7006 -
-cluster-replicas 1
>>> Performing hash slots allocation on 6 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922
Master[2] -> Slots 10923 - 16383
Adding replica 127.0.0.1:7005 to 127.0.0.1:7001
Adding replica 127.0.0.1:7006 to 127.0.0.1:7002
Adding replica 127.0.0.1:7004 to 127.0.0.1:7003
```

```
>>> Trying to optimize slaves allocation for anti-affinity
[WARNING] Some slaves are in the same host as their master
M: af559fc6c82c83dc39d07e2dfe59046d16b6a429 127.0.0.1:7001
   slots:[0-5460] (5461 slots) master
M: 068b678923ad0858002e906040b0fef6fff8dda4 127.0.0.1:7002
   slots:[5461-10922] (5462 slots) master
M: d277cd2984639747a17ca79428602480b28ef070 127.0.0.1:7003
   slots:[10923-16383] (5461 slots) master
S: 51c3ebdd0911dd6564040c7e20b9ae69cabb0425 127.0.0.1:7004
   replicates af559fc6c82c83dc39d07e2dfe59046d16b6a429
S: e7b1f1962de2a1ffef2bf1ac5d94574b2e4d67d8 127.0.0.1:7005
   replicates 068b678923ad0858002e906040b0fef6fff8dda4
S: 78dfe773eaa817fb69a405a3863f5b8fcf3e172f 127.0.0.1:7006
   replicates d277cd2984639747a17ca79428602480b28ef070
Can I set the above configuration? (type 'yes' to accept): yes
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join
....
>>> Performing Cluster Check (using node 127.0.0.1:7001)
M: af559fc6c82c83dc39d07e2dfe59046d16b6a429 127.0.0.1:7001
   slots:[0-5460] (5461 slots) master
   1 additional replica(s)
M: d277cd2984639747a17ca79428602480b28ef070 127.0.0.1:7003
   slots:[10923-16383] (5461 slots) master
   1 additional replica(s)
S: e7b1f1962de2a1ffef2bf1ac5d94574b2e4d67d8 127.0.0.1:7005
   slots: (0 slots) slave
   replicates 068b678923ad0858002e906040b0fef6fff8dda4
M: 068b678923ad0858002e906040b0fef6fff8dda4 127.0.0.1:7002
   slots:[5461-10922] (5462 slots) master
   1 additional replica(s)
S: 51c3ebdd0911dd6564040c7e20b9ae69cabb0425 127.0.0.1:7004
   slots: (0 slots) slave
   replicates af559fc6c82c83dc39d07e2dfe59046d16b6a429
S: 78dfe773eaa817fb69a405a3863f5b8fcf3e172f 127.0.0.1:7006
   slots: (0 slots) slave
   replicates d277cd2984639747a17ca79428602480b28ef070
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
[root@localhost-0723 redis]#
```

命令客户端连接集群

命令：

```
./redis-cli  -h 127.0.0.1 -p 7001 -c
```

注意：**-c** 表示是以redis集群方式进行连接

```
[root@localhost redis-cluster]# cd 7001
[root@localhost 7001]# ./redis-cli  -h 127.0.0.1 -p 7001 -c
127.0.0.1:7001> set name1 aaa
-> Redirected to slot [12933] located at 127.0.0.1:7003
OK
127.0.0.1:7003>
```

查看集群的命令

- 查看集群状态

```
127.0.0.1:7003> cluster info
cluster_state:ok
cluster_slots_assigned:16384
cluster_slots_ok:16384
cluster_slots_pfail:0
cluster_slots_fail:0
cluster_known_nodes:6
cluster_size:3
cluster_current_epoch:6
cluster_my_epoch:3
cluster_stats_messages_sent:926
cluster_stats_messages_received:926
```

- 查看集群中的节点:

```
127.0.0.1:7003> cluster nodes
d277cd2984639747a17ca79428602480b28ef070 127.0.0.1:7003@17003 myself,master - 0
1570457306000 3 connected 10923-16383
af559fc6c82c83dc39d07e2dfe59046d16b6a429 127.0.0.1:7001@17001 master - 0
1570457307597 1 connected 0-5460
e7b1f1962de2a1ffef2bf1ac5d94574b2e4d67d8 127.0.0.1:7005@17005 slave
068b678923ad0858002e906040b0fef6fff8dda4 0 1570457308605 5 connected
068b678923ad0858002e906040b0fef6fff8dda4 127.0.0.1:7002@17002 master - 0
1570457309614 2 connected 5461-10922
51c3ebdd0911dd6564040c7e20b9ae69cabb0425 127.0.0.1:7004@17004 slave
af559fc6c82c83dc39d07e2dfe59046d16b6a429 0 1570457307000 4 connected
78dfe773eaa817fb69a405a3863f5b8fcf3e172f 127.0.0.1:7006@17006 slave
d277cd2984639747a17ca79428602480b28ef070 0 1570457309000 6 connected
127.0.0.1:7003>
```

维护节点

集群创建成功后可以继续向集群中添加节点

添加主节点

- 先创建7007节点
- 添加7007结点作为新节点,并启动

  执行命令:

```
[root@localhost 7007]# ./redis-cli --cluster add-node 127.0.0.1:7007
127.0.0.1:7001
>>> Adding node 127.0.0.1:7007 to cluster 127.0.0.1:7001
>>> Performing Cluster Check (using node 127.0.0.1:7001)
M: af559fc6c82c83dc39d07e2dfe59046d16b6a429 127.0.0.1:7001
   slots:[0-5460] (5461 slots) master
   1 additional replica(s)
M: d277cd2984639747a17ca79428602480b28ef070 127.0.0.1:7003
   slots:[10923-16383] (5461 slots) master
   1 additional replica(s)
S: e7b1f1962de2a1ffef2bf1ac5d94574b2e4d67d8 127.0.0.1:7005
   slots: (0 slots) slave
   replicates 068b678923ad0858002e906040b0fef6fff8dda4
M: 068b678923ad0858002e906040b0fef6fff8dda4 127.0.0.1:7002
   slots:[5461-10922] (5462 slots) master
   1 additional replica(s)
S: 51c3ebdd0911dd6564040c7e20b9ae69cabb0425 127.0.0.1:7004
   slots: (0 slots) slave
   replicates af559fc6c82c83dc39d07e2dfe59046d16b6a429
S: 78dfe773eaa817fb69a405a3863f5b8fcf3e172f 127.0.0.1:7006
   slots: (0 slots) slave
   replicates d277cd2984639747a17ca79428602480b28ef070
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
>>> Send CLUSTER MEET to node 127.0.0.1:7007 to make it join the cluster.
[OK] New node added correctly.
```

- 查看集群结点发现7007已添加到集群中

```
127.0.0.1:7001> cluster nodes
d277cd2984639747a17ca79428602480b28ef070 127.0.0.1:7003@17003 master - 0
1570457568602 3 connected 10923-16383
50b073163bc4058e89d285dc5dfc42a0d1a222f2 127.0.0.1:7007@17007 master - 0
1570457567000 0 connected
e7b1f1962de2a1ffef2bf1ac5d94574b2e4d67d8 127.0.0.1:7005@17005 slave
068b678923ad0858002e906040b0fef6fff8dda4 0 1570457569609 5 connected
068b678923ad0858002e906040b0fef6fff8dda4 127.0.0.1:7002@17002 master - 0
1570457566000 2 connected 5461-10922
51c3ebdd0911dd6564040c7e20b9ae69cabb0425 127.0.0.1:7004@17004 slave
af559fc6c82c83dc39d07e2dfe59046d16b6a429 0 1570457567000 4 connected
af559fc6c82c83dc39d07e2dfe59046d16b6a429 127.0.0.1:7001@17001 myself,master - 0
1570457567000 1 connected 0-5460
78dfe773eaa817fb69a405a3863f5b8fcf3e172f 127.0.0.1:7006@17006 slave
d277cd2984639747a17ca79428602480b28ef070 0 1570457567593 6 connected
```

hash槽重新分配（数据迁移）

添加完主节点需要对主节点进行hash槽分配，这样该主节才可以存储数据。

- **查看集群中槽占用情况**

```
cluster nodes
```

redis集群有16384个槽，集群中的每个结点分配自已槽，通过查看集群结点可以看到槽占用情况。

```
127.0.0.1:7001> cluster nodes
d277cd2984639747a17ca79428602480b28ef070 127.0.0.1:7003@17003 master - 0
1570457568602 3 connected 10923-16383
50b073163bc4058e89d285dc5dfc42a0d1a222f2 127.0.0.1:7007@17007 master - 0
1570457567000 0 connected
e7b1f1962de2a1ffef2bf1ac5d94574b2e4d67d8 127.0.0.1:7005@17005 slave
068b678923ad0858002e906040b0fef6fff8dda4 0 1570457569609 5 connected
068b678923ad0858002e906040b0fef6fff8dda4 127.0.0.1:7002@17002 master - 0
1570457566000 2 connected 5461-10922
51c3ebdd0911dd6564040c7e20b9ae69cabb0425 127.0.0.1:7004@17004 slave
af559fc6c82c83dc39d07e2dfe59046d16b6a429 0 1570457567000 4 connected
af559fc6c82c83dc39d07e2dfe59046d16b6a429 127.0.0.1:7001@17001 myself,master - 0
1570457567000 1 connected 0-5460
78dfe773eaa817fb69a405a3863f5b8fcf3e172f 127.0.0.1:7006@17006 slave
d277cd2984639747a17ca79428602480b28ef070 0 1570457567593 6 connected
```

给刚添加的7007结点分配槽

- **第一步：连接上集群（连接集群中任意一个可用结点都行）**

```
[root@localhost 7007]# ./redis-cli --cluster reshard 127.0.0.1:7007
>>> Performing Cluster Check (using node 127.0.0.1:7007)
M: 50b073163bc4058e89d285dc5dfc42a0d1a222f2 127.0.0.1:7007
   slots: (0 slots) master
S: 51c3ebdd0911dd6564040c7e20b9ae69cabb0425 127.0.0.1:7004
   slots: (0 slots) slave
   replicates af559fc6c82c83dc39d07e2dfe59046d16b6a429
S: 78dfe773eaa817fb69a405a3863f5b8fcf3e172f 127.0.0.1:7006
   slots: (0 slots) slave
   replicates d277cd2984639747a17ca79428602480b28ef070
S: e7b1f1962de2a1ffef2bf1ac5d94574b2e4d67d8 127.0.0.1:7005
   slots: (0 slots) slave
   replicates 068b678923ad0858002e906040b0fef6fff8dda4
M: af559fc6c82c83dc39d07e2dfe59046d16b6a429 127.0.0.1:7001
   slots:[0-5460] (5461 slots) master
   1 additional replica(s)
M: 068b678923ad0858002e906040b0fef6fff8dda4 127.0.0.1:7002
   slots:[5461-10922] (5462 slots) master
   1 additional replica(s)
M: d277cd2984639747a17ca79428602480b28ef070 127.0.0.1:7003
   slots:[10923-16383] (5461 slots) master
   1 additional replica(s)
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
```

- **第二步：输入要分配的槽数量**

```
How many slots do you want to move (from 1 to 16384)? 3000
```

输入：**3000**，表示要给目标节点分配3000个槽

- **第三步：输入接收槽的结点id**

```
What is the receiving node ID?
```

输入：50b073163bc4058e89d285dc5dfc42a0d1a222f2

```
PS：这里准备给7007分配槽，通过cluster nodes查看7007结点id为：

50b073163bc4058e89d285dc5dfc42a0d1a222f2
```

- **第四步：输入源结点id**

```
Please enter all the source node IDs.
  Type 'all' to use all the nodes as source nodes for the hash slots.
  Type 'done' once you entered all the source nodes IDs.
```

输入：**all**

- **第五步：输入yes开始移动槽到目标结点id**

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

输入：**yes**

```
Moving slot 11899 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11900 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11901 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11902 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11903 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11904 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11905 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11906 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11907 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11908 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11909 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11910 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11911 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11912 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11913 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11914 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11915 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11916 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11917 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11918 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11919 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11920 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11921 from 127.0.0.1:7003 to 127.0.0.1:7007:
```

查看结果

```
127.0.0.1:7001> cluster nodes
d277cd2984639747a17ca79428602480b28ef070 127.0.0.1:7003@17003 master - 0
1570458285557 3 connected 11922-16383
50b073163bc4058e89d285dc5dfc42a0d1a222f2 127.0.0.1:7007@17007 master - 0
1570458284000 7 connected 0-998 5461-6461 10923-11921
e7b1f1962de2a1ffef2bf1ac5d94574b2e4d67d8 127.0.0.1:7005@17005 slave
068b678923ad0858002e906040b0fef6fff8dda4 0 1570458283000 5 connected
068b678923ad0858002e906040b0fef6fff8dda4 127.0.0.1:7002@17002 master - 0
1570458284546 2 connected 6462-10922
51c3ebdd0911dd6564040c7e20b9ae69cabb0425 127.0.0.1:7004@17004 slave
af559fc6c82c83dc39d07e2dfe59046d16b6a429 0 1570458283538 4 connected
af559fc6c82c83dc39d07e2dfe59046d16b6a429 127.0.0.1:7001@17001 myself,master - 0
1570458283000 1 connected 999-5460
78dfe773eaa817fb69a405a3863f5b8fcf3e172f 127.0.0.1:7006@17006 slave
d277cd2984639747a17ca79428602480b28ef070 0 1570458284000 6 connected
```

添加从节点

- **添加7008从结点，将7008作为7007的从结点**

命令：

```
./redis-cli --cluster add-node 新节点的ip和端口   旧节点ip和端口 --cluster-slave --
cluster-master-id 主节点id
```

例如：

```
./redis-cli --cluster add-node 127.0.0.1:7008 127.0.0.1:7007 --cluster-slave --
cluster-master-id 50b073163bc4058e89d285dc5dfc42a0d1a222f2
```

50b073163bc4058e89d285dc5dfc42a0d1a222f2是7007结点的id，可通过cluster nodes查看。

```
[root@localhost 7008]# ./redis-cli --cluster add-node 127.0.0.1:7008
127.0.0.1:7007 --cluster-slave --cluster-master-id
50b073163bc4058e89d285dc5dfc42a0d1a222f2
>>> Adding node 127.0.0.1:7008 to cluster 127.0.0.1:7007
>>> Performing Cluster Check (using node 127.0.0.1:7007)
M: 50b073163bc4058e89d285dc5dfc42a0d1a222f2 127.0.0.1:7007
   slots:[0-998],[5461-6461],[10923-11921] (2999 slots) master
S: 51c3ebdd0911dd6564040c7e20b9ae69cabb0425 127.0.0.1:7004
   slots: (0 slots) slave
   replicates af559fc6c82c83dc39d07e2dfe59046d16b6a429
S: 78dfe773eaa817fb69a405a3863f5b8fcf3e172f 127.0.0.1:7006
   slots: (0 slots) slave
   replicates d277cd2984639747a17ca79428602480b28ef070
S: e7b1f1962de2a1ffef2bf1ac5d94574b2e4d67d8 127.0.0.1:7005
   slots: (0 slots) slave
   replicates 068b678923ad0858002e906040b0fef6fff8dda4
M: af559fc6c82c83dc39d07e2dfe59046d16b6a429 127.0.0.1:7001
   slots:[999-5460] (4462 slots) master
   1 additional replica(s)
M: 068b678923ad0858002e906040b0fef6fff8dda4 127.0.0.1:7002
   slots:[6462-10922] (4461 slots) master
```

```
   1 additional replica(s)
M: d277cd2984639747a17ca79428602480b28ef070 127.0.0.1:7003
   slots:[11922-16383] (4462 slots) master
   1 additional replica(s)
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
>>> Send CLUSTER MEET to node 127.0.0.1:7008 to make it join the cluster.
Waiting for the cluster to join
.....
>>> Configure node as replica of 127.0.0.1:7007.
[OK] New node added correctly.
```

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

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

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

- **查看集群中的结点，刚添加的7008为7007的从节点：**

```
[root@localhost 7008]# ./redis-cli  -h 127.0.0.1 -p 7001 -c
127.0.0.1:7001> cluster nodes
d277cd2984639747a17ca79428602480b28ef070 127.0.0.1:7003@17003 master - 0
1570458650720 3 connected 11922-16383
c3272565847bf9be8ae0194f7fb833db40b98ac4 127.0.0.1:7008@17008 slave
50b073163bc4058e89d285dc5dfc42a0d1a222f2 0 1570458648710 7 connected
50b073163bc4058e89d285dc5dfc42a0d1a222f2 127.0.0.1:7007@17007 master - 0
1570458649000 7 connected 0-998 5461-6461 10923-11921
e7b1f1962de2a1ffef2bf1ac5d94574b2e4d67d8 127.0.0.1:7005@17005 slave
068b678923ad0858002e906040b0fef6fff8dda4 0 1570458650000 5 connected
068b678923ad0858002e906040b0fef6fff8dda4 127.0.0.1:7002@17002 master - 0
1570458649715 2 connected 6462-10922
51c3ebdd0911dd6564040c7e20b9ae69cabb0425 127.0.0.1:7004@17004 slave
af559fc6c82c83dc39d07e2dfe59046d16b6a429 0 1570458648000 4 connected
af559fc6c82c83dc39d07e2dfe59046d16b6a429 127.0.0.1:7001@17001 myself,master - 0
1570458650000 1 connected 999-5460
78dfe773eaa817fb69a405a3863f5b8fcf3e172f 127.0.0.1:7006@17006 slave
d277cd2984639747a17ca79428602480b28ef070 0 1570458651725 6 connected
127.0.0.1:7001>
```

删除结点

命令：

```
./redis-cli --cluster del-node 127.0.0.1:7008
41592e62b83a8455f07f7797f1d5c071cffedb50
```

删除已经占有hash槽的结点会失败，报错如下：

```
[ERR] Node 127.0.0.1:7008 is not empty! Reshard data away and try again.
```

需要将该结点占用的hash槽分配出去（参考hash槽重新分配章节）。

Jedis连接集群

需要开启防火墙，或者直接关闭防火墙。

```
service iptables stop
```

代码实现

创建JedisCluster类连接redis集群。

```
@Test
public void testJedisCluster() throws Exception {
    //创建一连接，JedisCluster对象,在系统中是单例存在
    Set<HostAndPort> nodes = new HashSet<>();
    nodes.add(new HostAndPort("192.168.10.133", 7001));
    nodes.add(new HostAndPort("192.168.10.133", 7002));
    nodes.add(new HostAndPort("192.168.10.133", 7003));
    nodes.add(new HostAndPort("192.168.10.133", 7004));
    nodes.add(new HostAndPort("192.168.10.133", 7005));
    nodes.add(new HostAndPort("192.168.10.133", 7006));
    JedisCluster cluster = new JedisCluster(nodes);
    //执行JedisCluster对象中的方法，方法和redis一一对应。
    cluster.set("cluster-test", "my jedis cluster test");
    String result = cluster.get("cluster-test");
    System.out.println(result);
    //程序结束时需要关闭JedisCluster对象
    cluster.close();
}
```

使用spring

Ø 配置applicationContext.xml

```
<!-- 连接池配置 -->
<bean id="jedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig">
    <!-- 最大连接数 -->
    <property name="maxTotal" value="30" />
    <!-- 最大空闲连接数 -->
    <property name="maxIdle" value="10" />
    <!-- 每次释放连接的最大数目 -->
    <property name="numTestsPerEvictionRun" value="1024" />
    <!-- 释放连接的扫描间隔（毫秒） -->
    <property name="timeBetweenEvictionRunsMillis" value="30000" />
    <!-- 连接最小空闲时间 -->
    <property name="minEvictableIdleTimeMillis" value="1800000" />
    <!-- 连接空闲多久后释放, 当空闲时间>该值 且 空闲连接>最大空闲连接数 时直接释放 -->
    <property name="softMinEvictableIdleTimeMillis" value="10000" />
    <!-- 获取连接时的最大等待毫秒数,小于零:阻塞不确定的时间,默认-1 -->
```

```
    <property name="maxWaitMillis" value="1500" />
    <!-- 在获取连接的时候检查有效性, 默认false -->
    <property name="testOnBorrow" value="true" />
    <!-- 在空闲时检查有效性, 默认false -->
    <property name="testWhileIdle" value="true" />
    <!-- 连接耗尽时是否阻塞, false报异常,ture阻塞直到超时, 默认true -->
    <property name="blockWhenExhausted" value="false" />
</bean>
<!-- redis集群 -->
<bean id="jedisCluster" class="redis.clients.jedis.JedisCluster">
    <constructor-arg index="0">
        <set>
            <bean class="redis.clients.jedis.HostAndPort">
                <constructor-arg index="0" value="192.168.101.3"></constructor-
arg>
                <constructor-arg index="1" value="7001"></constructor-arg>
            </bean>
            <bean class="redis.clients.jedis.HostAndPort">
                <constructor-arg index="0" value="192.168.101.3"></constructor-
arg>
                <constructor-arg index="1" value="7002"></constructor-arg>
            </bean>
            <bean class="redis.clients.jedis.HostAndPort">
                <constructor-arg index="0" value="192.168.101.3"></constructor-
arg>
                <constructor-arg index="1" value="7003"></constructor-arg>
            </bean>
            <bean class="redis.clients.jedis.HostAndPort">
                <constructor-arg index="0" value="192.168.101.3"></constructor-
arg>
                <constructor-arg index="1" value="7004"></constructor-arg>
            </bean>
            <bean class="redis.clients.jedis.HostAndPort">
                <constructor-arg index="0" value="192.168.101.3"></constructor-
arg>
                <constructor-arg index="1" value="7005"></constructor-arg>
            </bean>
            <bean class="redis.clients.jedis.HostAndPort">
                <constructor-arg index="0" value="192.168.101.3"></constructor-
arg>
                <constructor-arg index="1" value="7006"></constructor-arg>
            </bean>
        </set>
    </constructor-arg>
    <constructor-arg index="1" ref="jedisPoolConfig"></constructor-arg>
</bean>
```

```
```

Ø 测试代码

```
private ApplicationContext applicationContext;
    @Before
```

```
    public void init() {
        applicationContext = new ClassPathXmlApplicationContext(
                "classpath:applicationContext.xml");
    }

    // redis集群
    @Test
    public void testJedisCluster() {
        JedisCluster jedisCluster = (JedisCluster) applicationContext
                .getBean("jedisCluster");

        jedisCluster.set("name", "zhangsan");
        String value = jedisCluster.get("name");
        System.out.println(value);
    }
```