Redis 多机特性工作原理简介

黄健宏（huangz）

视频共分为三个部分，分别讲述 Redis 的三个多机特性。

1. 复制（replication）
2. Sentinel
3. 集群（cluster）

复制

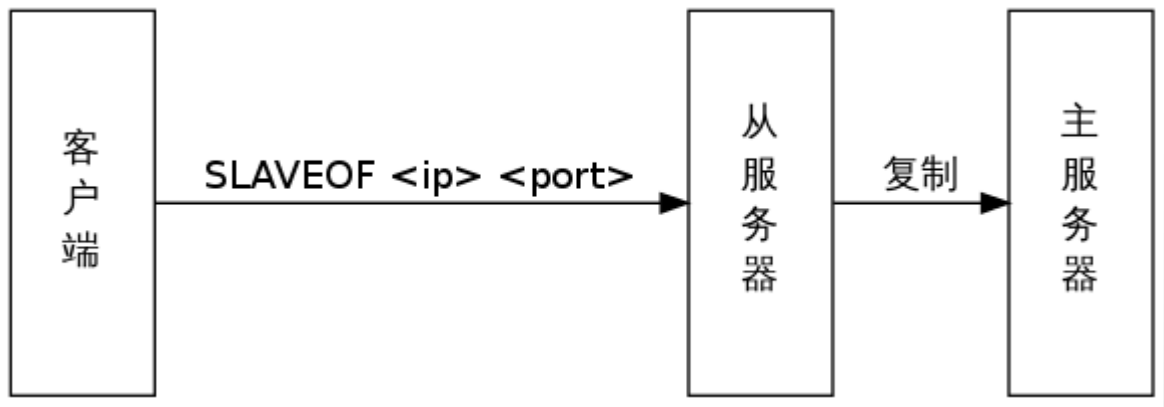
SLAVEOF 、SYNC 和 PSYNC、一致性

复制

SLAVEOF 命令的实现原理

SLAVEOF <ip> <port>

将一个服务器(从服务器)变成为另一个服务器(主服务器)的复制品。



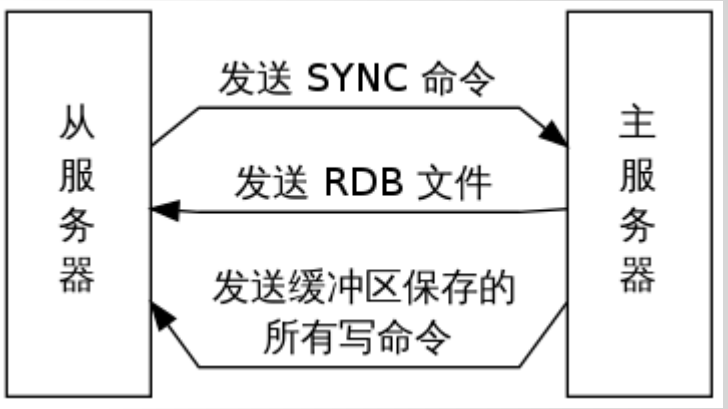
复制的执行步骤(4/5):同步

1. 从服务器向主服务器发送 SYNC 命令。

2.主服务器调用 BGSAVE，创建一个 RDB 文件，并使用缓冲区记录接下来执行的所有写命令。

3. 主服务器向从服务器发送 RDB 文件，从服务器接收并载入该文件。

4. 主服务器将缓冲区储存的所有写命令发送给从服务器执行。



SYNC 命令执行示例

时间	主服务器	从服务器
T0	服务器启动。	服务器启动。
T1	执行 `SET k1 v1` 。	
T2	执行 `SET k2 v2` 。	
T3	执行 `SET k3 v3` 。	
T4		向主服务器发送 *SYNC* 命令。
T5	接收到从服务器发来的 *SYNC* 命令， 执行 *BGSAVE* 命令， 创建包含键 `k1` 、 `k2` 、 `k3` 的 RDB 文件， 并使用缓冲区记录接下来执行的所有写命令。	
T6	执行 `SET k4 v4` ， 并将这个命令记录到缓冲区里面。	
T7	执行 `SET k5 v5` ， 并将这个命令记录到缓冲区里面。	
T8	*BGSAVE* 命令执行完毕， 向从服务器发送 RDB 文件。	
T9		接收并载入主服务器发来的 RDB 文件， 获得 `k1` 、 `k2` 、 `k3` 三个键。
T10	向从服务器发送缓冲区中保存的写命令 `SET k4 v4` 和 `SET k5 v5` 。	
T11		接收并执行主服务器发来的两个 *SET* 命令， 得到 `k4` 和 `k5` 两个键。
T12	同步完成， 现在主从服务器两者的数据库都包含了键 `k1` 、 `k2` 、 `k3` 、 `k4` 和 `k5` 。	同步完成， 现在主从服务器两者的数据库都包含了键 `k1` 、 `k2` 、 `k3` 、 `k4` 和 `k5` 。

但是……

执行 SYNC 命令之后，主从服务器的数据库状态将**达到一致状态**（两个数据库都储存了相同的数据），但这种一致性只是**暂时性的**，因为一旦主服务器**执行了新的写命令**，主从服务器的数据库又会变得**不一致**。

不一致性出现的例子

同步执行完毕之后

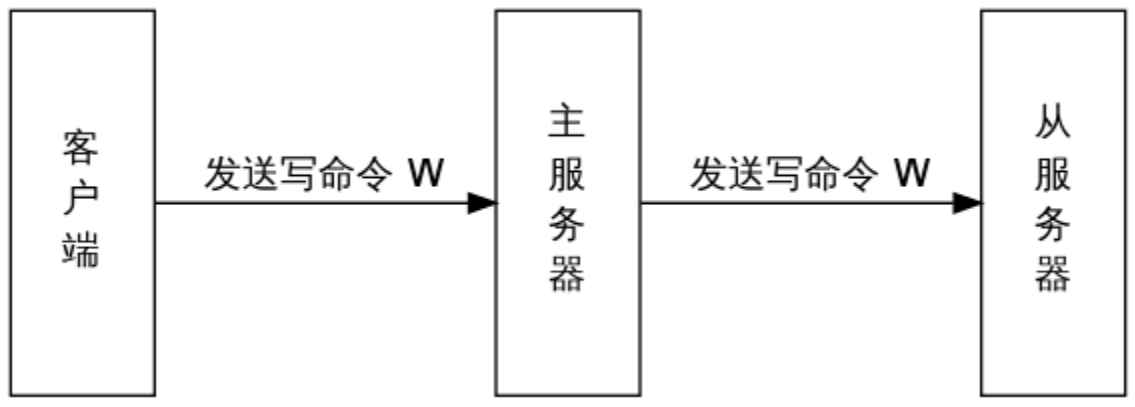
主服务器和从服务器
k1
k2
k3
k4
k5

主服务器执行 SET k6 v6 之后

主服务器	从服务器
k1	k1
k2	k2
k3	k3
k4	k4
k5	k5
k6	**N/A**

复制的执行步骤(5/5):命令传播

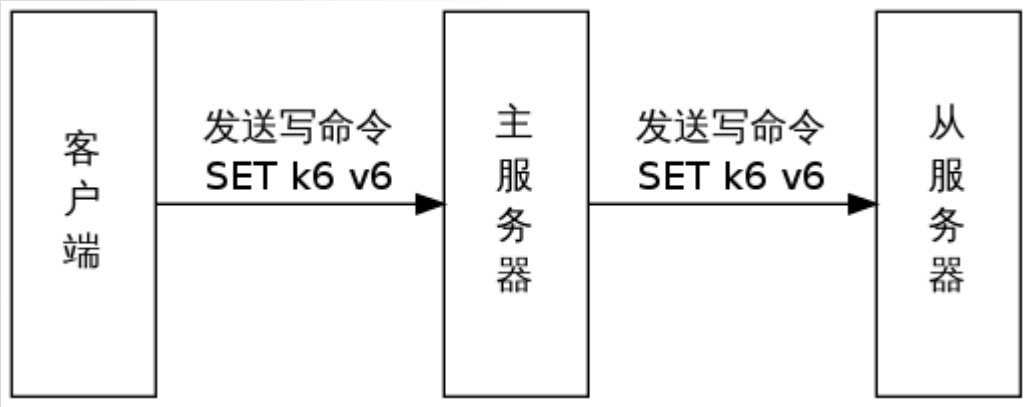
5.在主从服务器完成同步之后，主服务器每执行一个写命令，它都会将**被执行的写命令发送给从服务器执行**，这个操作被称为“命令传播”(command propagate)。



注意！
命令传播是一个持续的过程:只要复制仍在继续，命令传播就会一直进行，使得主从服务器的状态可以一直保持一致。

命令传播示例

执行并发送 SET k6 v6 之后



主服务器	从服务器
k1	k1
k2	k2
k3	k3
k4	k4
k5	k5
k6	**k6**

复制

SYNC 和 PSYNC 的区别

从 Redis 2.8 开始，Redis 使用 PSYNC 命令代替 SYNC 命令。

PSYNC 比起 SYNC 的最大改进在于 PSYNC 实现了部分同步（partial sync）特性：

在主从服务器断线并重连时，只要条件允许，PSYNC 可以让主服务器只向从服务器同步**断线期间缺失的数据**，而不用重新向从服务器同步**整个数据库**。

SYNC 处理断线重连

时间	主服务器	从服务器
T0	主从服务器完成同步。	主从服务器完成同步。
T1	执行并传播 `SET k1 v1` 。	执行主服务器传来的 `SET k1 v1` 。
T2	执行并传播 `SET k2 v2` 。	执行主服务器传来的 `SET k2 v2` 。
...	...	...
T10085	执行并传播 `SET k10085 v10085` 。	执行主服务器传来的 `SET k10085 v10085` 。
T10086	执行并传播 `SET k10086 v10086` 。	执行主服务器传来的 `SET k10086 v10086` 。
T10087	主从服务器连接断开。	主从服务器连接断开。
T10088	执行 `SET k10087 v10087` 。	断线中，尝试重新连接主服务器。
T10089	执行 `SET k10088 v10088` 。	断线中，尝试重新连接主服务器。
T10090	执行 `SET k10089 v10089` 。	断线中，尝试重新连接主服务器。
T10091	主从服务器重新连接。	主从服务器重新连接。
T10092		向主服务器发送 *SYNC* 命令。
T10093	接收到从服务器发来的 *SYNC* 命令， 执行 *BGSAVE* 命令， 创建包含键 `k1` 至键 `k10089` 的 RDB 文件， 并使用缓冲区记录接下来执行的所有写命令。	
T10094	*BGSAVE* 命令执行完毕，向从服务器发送 RDB 文件。	
T10095		接收并载入主服务器发来的 RDB 文件， 获得键 `k1` 至键 `k10089` 。
T10096	因为在 *BGSAVE* 命令执行期间， 主服务器没有执行任何写命令， 所以跳过发送缓冲区包含的写命令这一步。	
T10097	主从服务器再次完成同步。	主从服务器再次完成同步。

从服务器在断线之前已经拥有主服务器的绝大部分数据，**要让主从服务器重新回到一致状态，真正需要向从服务器发送的是 k10087 、k10088 、k10089 这三个键的数据。**

SYNC 命令的做法 —— 将主服务器的整个数据库重新同步给从服务器—— 是极度浪费的！

PSYNC 处理断线重连

时间	主服务器	从服务器
T0	主从服务器完成同步。	主从服务器完成同步。
T1	执行并传播 `SET k1 v1` 。	执行主服务器传来的 `SET k1 v1` 。
T2	执行并传播 `SET k2 v2` 。	执行主服务器传来的 `SET k2 v2` 。
...	...	...
T10085	执行并传播 `SET k10085 v10085` 。	执行主服务器传来的 `SET k10085 v10085` 。
T10086	执行并传播 `SET k10086 v10086` 。	执行主服务器传来的 `SET k10086 v10086` 。
T10087	主从服务器连接断开。	主从服务器连接断开。
T10088	执行 `SET k10087 v10087` 。	断线中，尝试重新连接主服务器。
T10089	执行 `SET k10088 v10088` 。	断线中，尝试重新连接主服务器。
T10090	执行 `SET k10089 v10089` 。	断线中，尝试重新连接主服务器。
T10091	主从服务器重新连接。	主从服务器重新连接。
T10092		向主服务器发送 `PSYNC` 命令。
T10093	向从服务器返回 `+CONTINUE` 回复，表示执行部分重同步。	
T10094		接收 `+CONTINUE` 回复，准备执行部分重同步。
T10095	向从服务器发送 `SET k10087 v10087` 、 `SET k10088 v10088` 、 `SET k10089 v10089` 三个命令。	
T10096		接收并执行主服务器传来的三个 *SET* 命令。
T10097	主从服务器再次完成同步。	主从服务器再次完成同步。

PSYNC 只会将**从服务器断线期间缺失的数据**发送给从服务器。

两个例子的情况是相同的，但 SYNC 需要发送包含**整个数据库**的 RDB 文件，而 PSYNC 只需要发送**三个命令**。

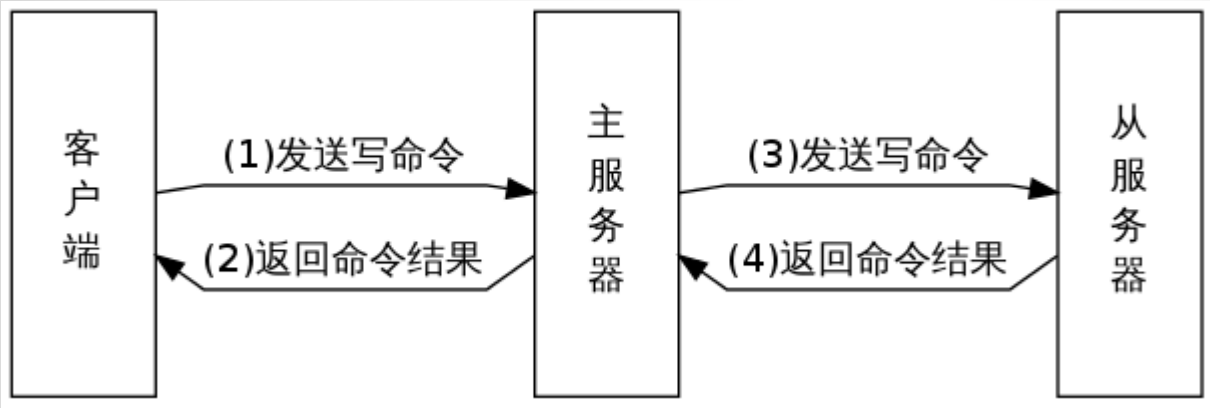
结论

如果主从服务器所处的网络环境并不那么好的话（经常断线），那么请尽量使用 Redis 2.8 或以上版本，这可以节约**大量网络资源、计算资源、内存资源** —— 这些消耗都来自**重复创建和传输 RDB 文件**。

复制

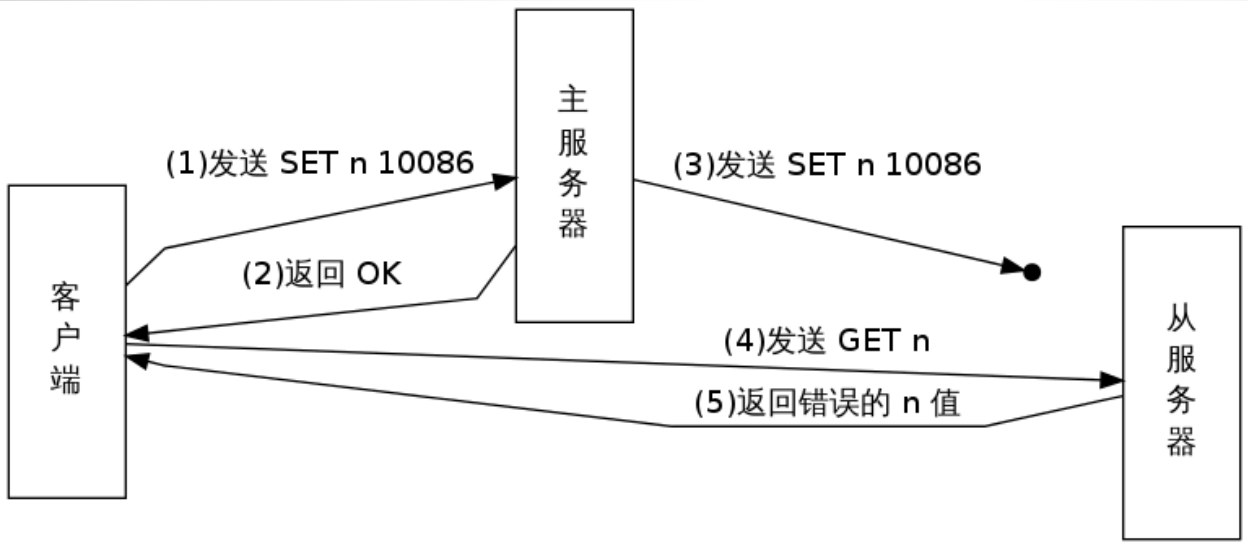
一致性问题

命令传播需要花费一段时间来完成



在主服务器执行完步骤(1)发送的命令之后、直到从服务器执行完步骤(3)发送的命令之前的这段时间里面，主从服务器状态**并不一致**。

现在，考虑这样一种情况……



客户端与主服务器的连接非常快，而主从服务器之间的连接却非常慢。

客户端向主服务器发送了命令 SET n 10086，并在获得返回 OK 之后，向从服务器发送 GET n 。

但是这时主服务器传播的 SET n 10086 因为网络原因仍然未到达从服务器，那么客户端获得的键 n 的值将是错误的（过期的）。

结论

- Redis 目前的复制实现只保证**最终一致性**，而不是**强一致性**。
- 如果程序**不能够容忍过期数据**，那么就不要读取从服务器。
- 未来：unstable 分支中的 WAIT 命令 —— 阻塞最多 N 秒，直到客户端之前发送的命令已经被复制到至少 M 个从服务器，**实现同步复制效果**。
  http://antirez.com/news/66

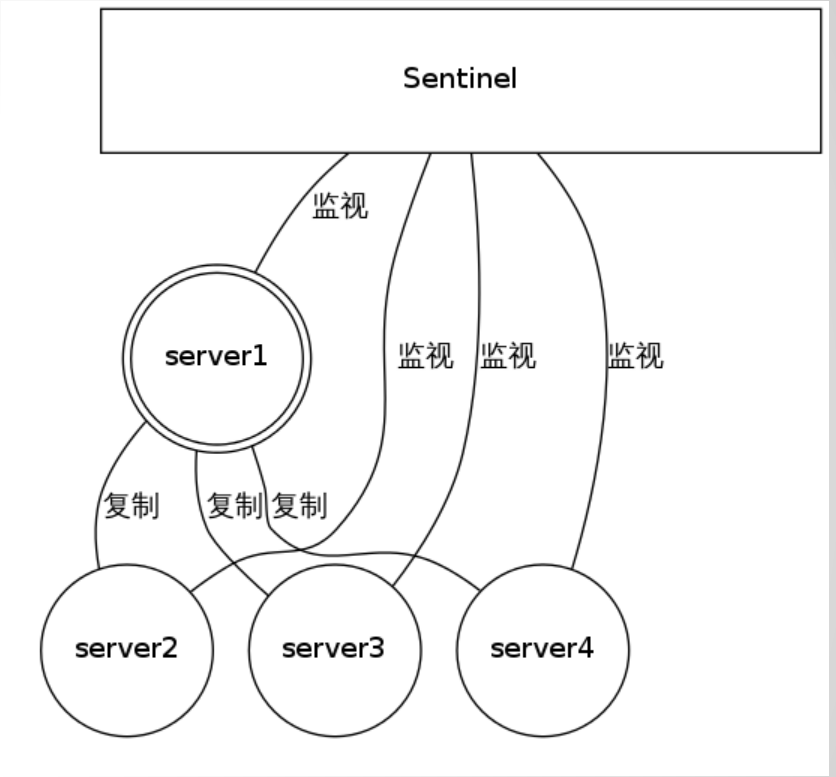
Sentinel

构造、实例监视、下线判断、故障转移

Sentinel 的构造

Sentinel 是一个监视器，它可以根据被监视实例的**身份**和**状态**来判断应该执行何种动作。

```
DEF sentinel():
  while sentinel_is_running():
    FOR instance IN all_instances:
      instance.check_status()              # 检查实例的当前状态
      IF instance.role == MASTER:      # 实例是一个主服务器
        IF instance.status == FAIL: handle_master_down()
        IF instance.status == OK: ...
      IF instance.role == SLAVE:          # 实例是一个从服务器
        …
      IF instance.role == SENTINEL:   # 实例是一个 Sentinel
      ...
```



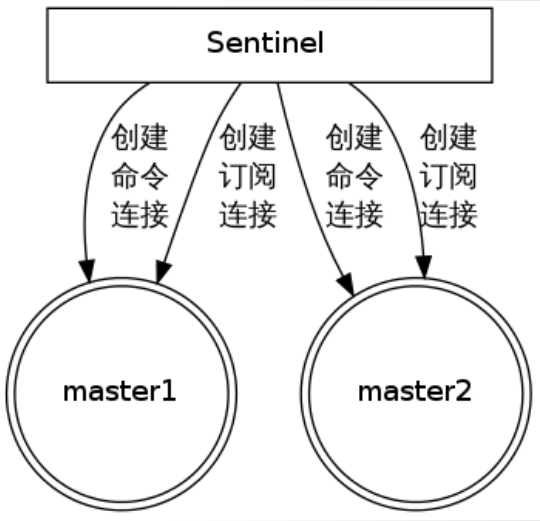
被监视的实例可以是主服务器、从服务器、或者其他 Sentinel 。

监视主服务器

Sentinel 通过用户给定的配置文件来发现主服务器。

```
############ master1 configure ##############
sentinel monitor master1 127.0.0.1 6379 2
sentinel down-after-milliseconds master1 30000
sentinel parallel-syncs master1 1
sentinel failover-timeout master1 900000

############ master2 configure ##############
sentinel monitor master2 127.0.0.1 12345 5
sentinel down-after-milliseconds master2 50000
sentinel parallel-syncs master2 5
sentinel failover-timeout master2 450000
```



Sentinel 会与被监视的主服务器创建两个网络连接：

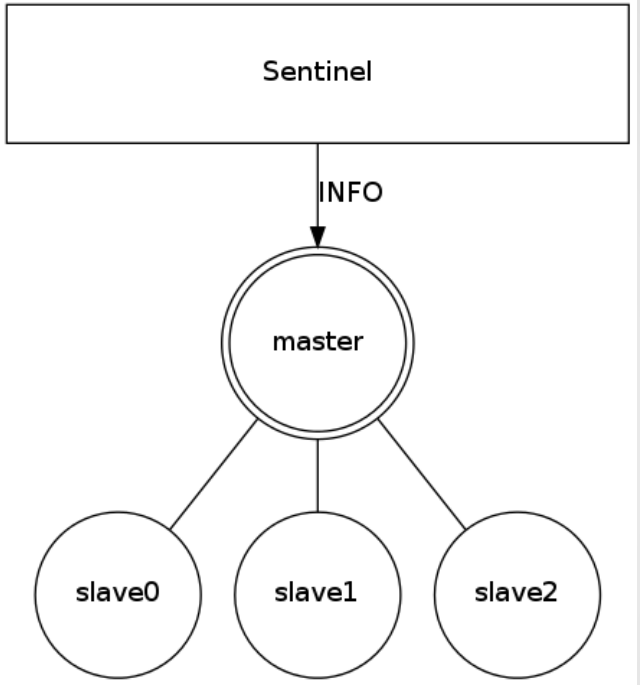
命令连接用于向主服务器发送命令。

订阅连接用于订阅指定的频道，从而发现监视同一主服务器的其他 Sentinel（细节稍后介绍）。

监视从服务器

Sentinel 通过向主服务器发送 INFO 命令来自动获得所有从服务器的地址。

```
redis>INFO
# other section ...
# Replication
role:master
...
slave0:ip=127.0.0.1,port=11111,state=online,offset=43,lag=0
slave1:ip=127.0.0.1,port=22222,state=online,offset=43,lag=0
slave2:ip=127.0.0.1,port=33333,state=online,offset=43,lag=0
…
# other section ...
```

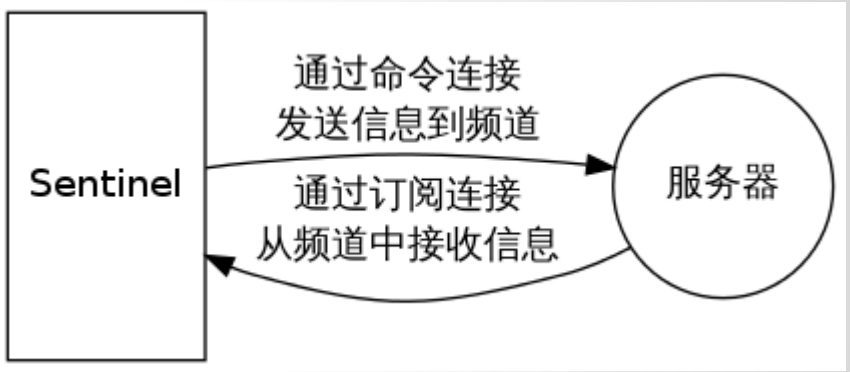


跟主服务器一样，Sentinel 会与每个被发现的从服务器创建命令连接和订阅连接。

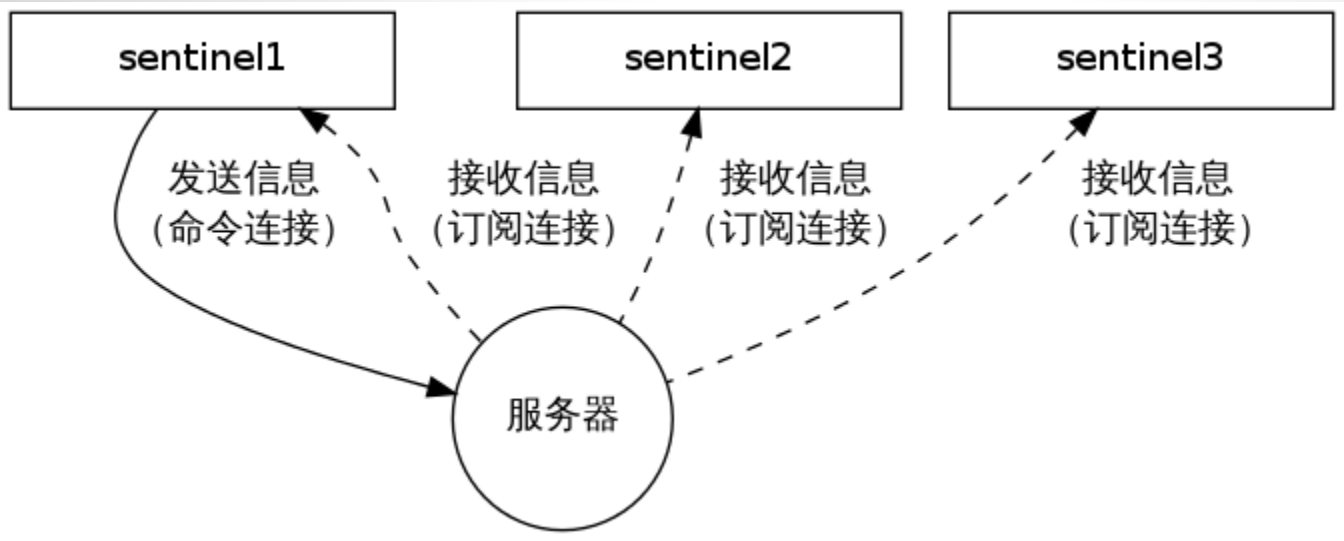
发现其他 Sentinel

Sentinel 会通过**命令连接**向被监视的主从服务器发送 HELLO 信息，以此来向其他 Sentinel 宣告自己的存在。

与此同时，Sentinel 会通过**订阅连接**收听其他 Sentinel 的 HELLO 信息，以此来发现监视同一个主服务器的其他 Sentinel 。



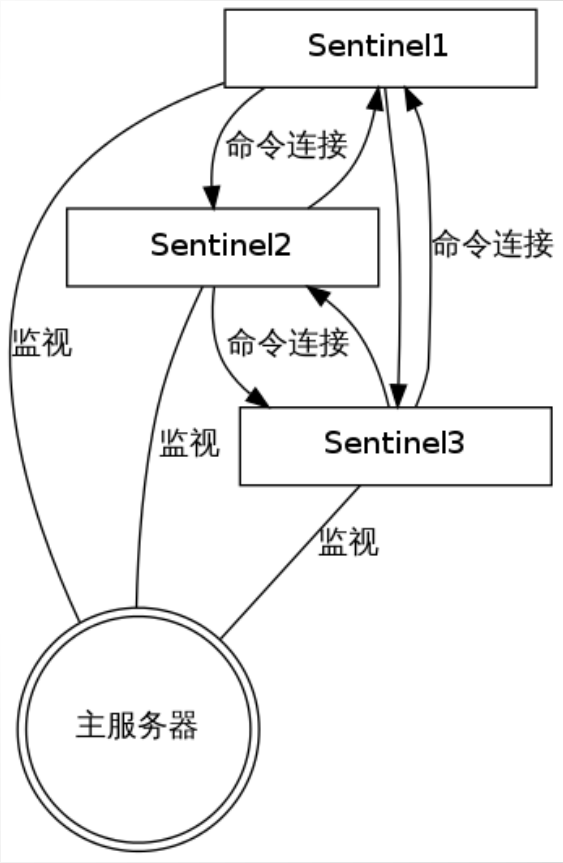
例子:发现其他 Sentinel



sentinel1 通过发送 HELLO 信息来让 sentinel2 和 sentinel3 发现自己，其他两个 sentinel 也会进行类似的操作。

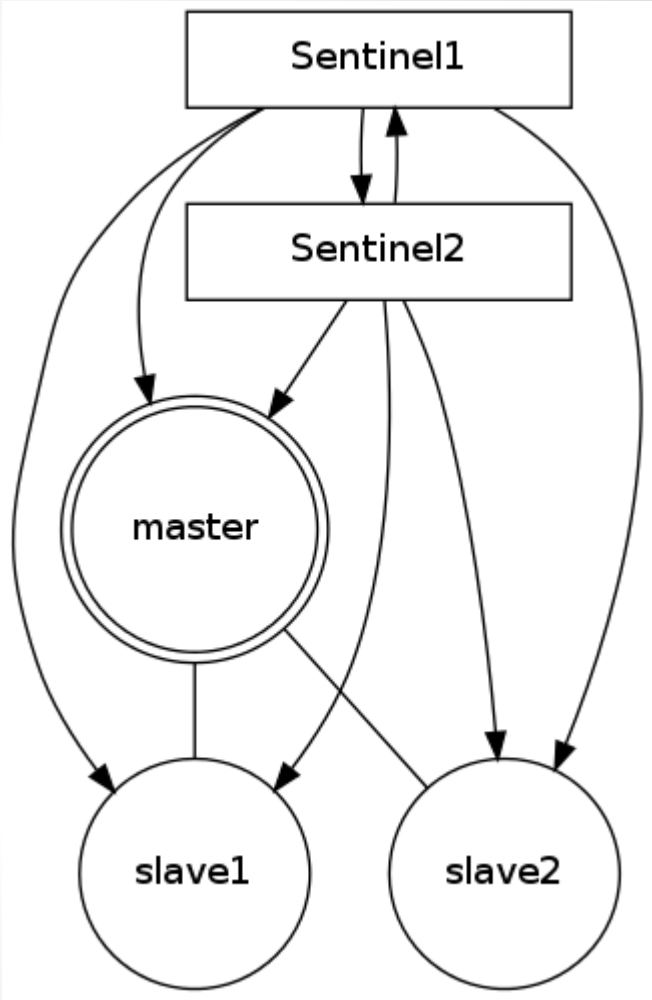
监视其他 SentienI

- Sentinel 之间会互相创建**命令连接**，用于进行通信。
- 因为已经有主从服务器作为发送和接收 HELLO 信息的中介，所以 Sentinel 之间**不会**创建订阅连接。



检测实例的状态

Sentinel 使用 PING 命令来检测实例的状态：如果实例在指定的时间内没有返回，或者返回错误的回复，那么该实例会被 Sentinel 判断为下线。



下线状态（1/2）

从服务器和其他 Sentinel 的下线状态为 PFAIL（probably fail）：

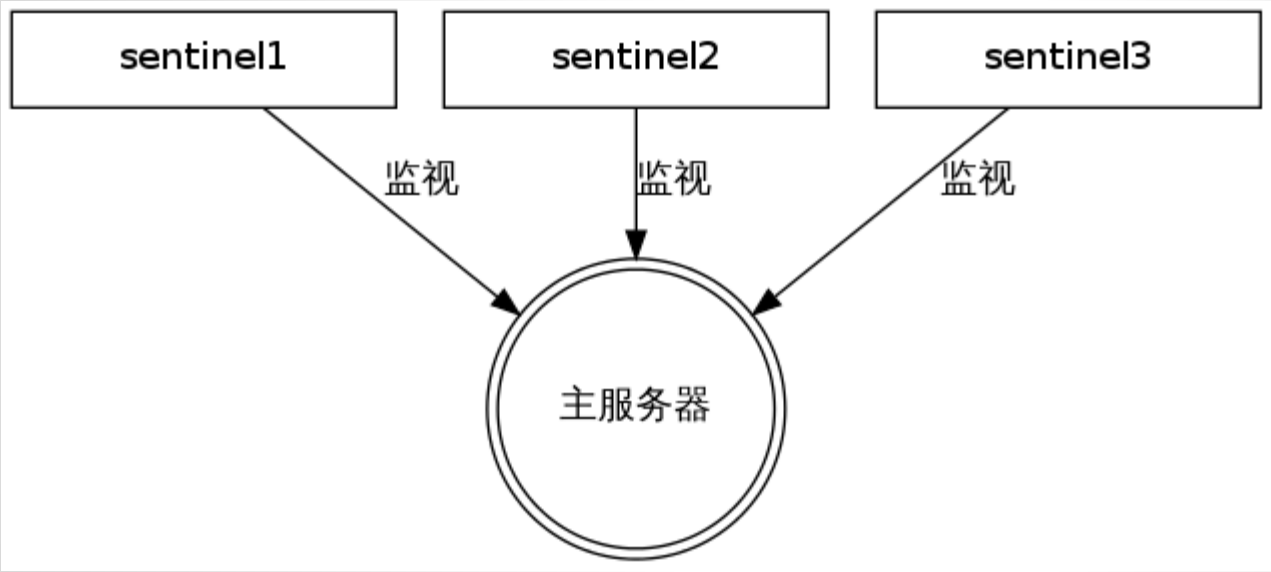
- Sentinel **不会**与这个状态下的其他 Sentinel 交换信息。
- 这个状态的从服务器也**不会**在故障转移（failover，细节稍后介绍）时被选为新的主服务器。

下线状态（2/2）

主服务器的下线状态分为 PFAIL 和 FAIL 两级：

- **PFAIL 状态**：当前 Sentinel 认为主服务器已下线。
- **FAIL 状态**：通过多个 Sentinel 互相通信，确认主服务器已经下线。
- 换言之，对于主服务器来说，FAIL 才是**真正的**下线状态。
- sentinel monitor <name> <ip> <port> **<quorum>**

示例:检测主服务器下线



监视同一个主服务器的多个 Sentinel 之间会互相交流信息，如果总共有 <quorum> 个 Sentinel 认为主服务器已下线（PFAIL），那么这个主服务器将被 Sentienl 判断为 FAIL 状态。

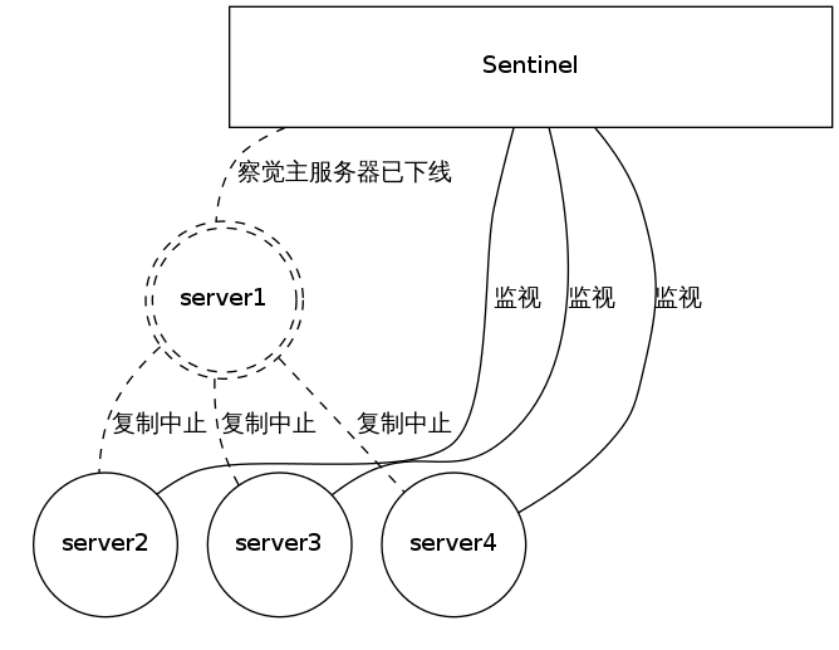
一旦主服务器被判断为 FAIL，Sentinel 就会对下线主服务器进行**故障转移**。

步骤：

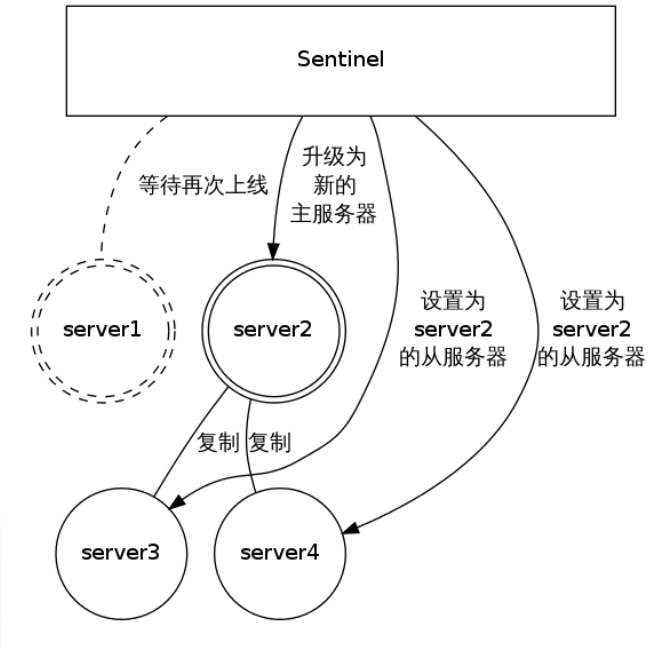
1. 在下线主服务器的所有从服务器中选出一个作为新的主服务器，被选中的从服务器是所有从服务器中**数据最新**的。
2. Sentinel 向被选中的从服务器发送 **SLAVEOF NO ONE**，将它升级为主服务器。
3. 向其他从服务器发送 **SLAVEOF <new-master-addr> <new-master-port>**，让其他从服务器开始复制新的主服务器。

故障转移示例

主服务器 FAIL



进行故障转移



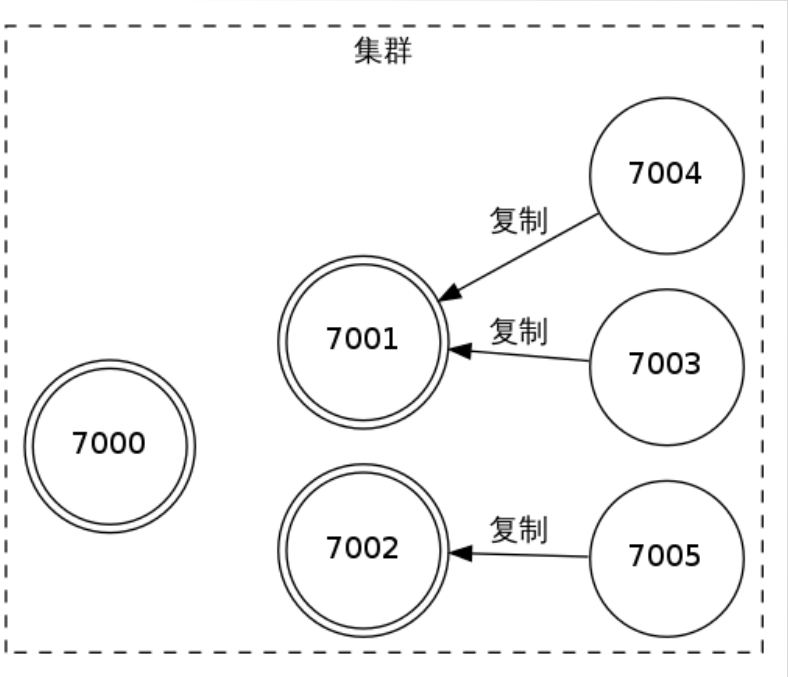
集群

节点、槽、命令执行、复制和故障转移

集群的构成

一个集群由一个或多个**节点(node)**组成，其中**主节点(master)**负责储存键值对，而**从节点(slave)**则负责复制主节点。

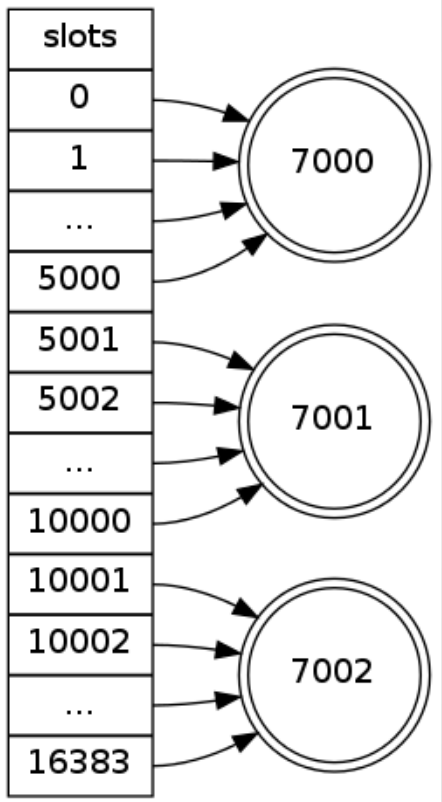
每个节点互相连接。



一个包含三个主节点，三个从节点的集群。

分片(sharding)

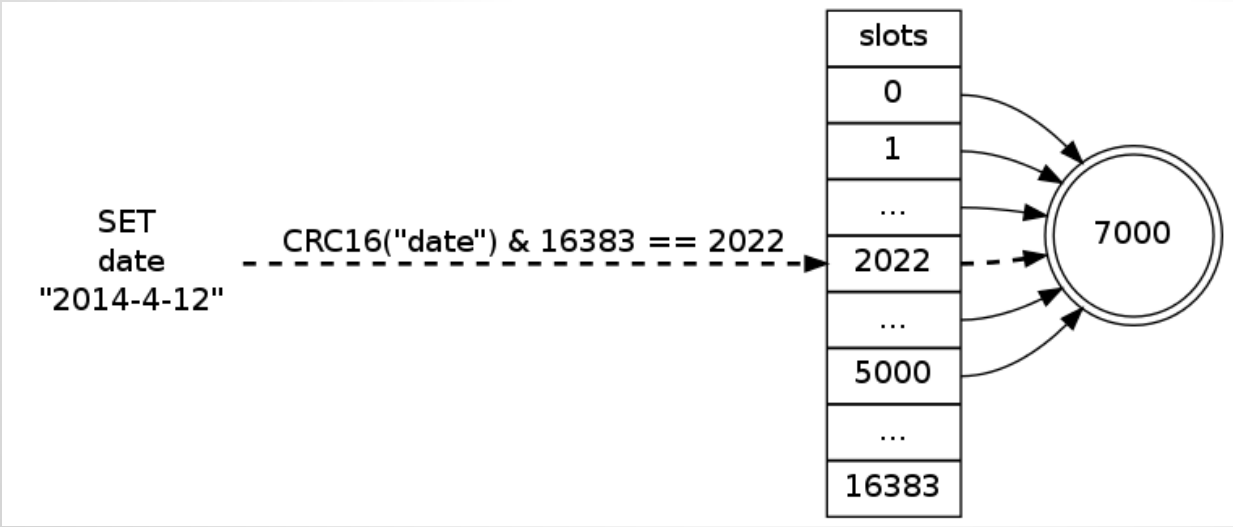
- 集群将整个数据库分为16384(2 的 14 次方)个槽。
- 每个主节点可以负责处理 0 个至 16384 个槽。
- 当 16384 个槽都有主节点负责处理时，集群进入上线状态。



将 16384 个槽分别指派给 7000 、7001 和 7002 这三个主节点。

槽的计算公式

集群使用公式 **CRC16(key) & 16383** 计算键 key 属于哪个槽。

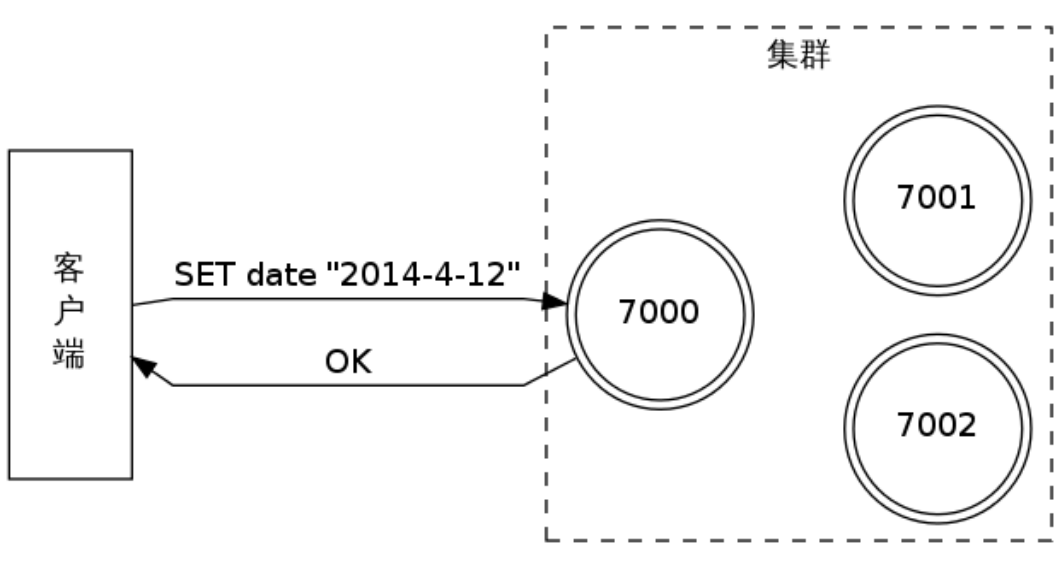


在集群里面执行命令的两种情况

1. 命令发送到了正确的节点**(命令要处理的键所在的槽正好是由接收命令的节点负责)**，那么该节点执行命令，就像单机 Redis 服务器一样。

2. 命令发送到了错误的节点**(接收到命令的节点并非处理键所在槽的节点)**，那么节点将向客户端返回一个**转向(redirection)错误**，告知客户端应该到哪个节点去执行这个命令，客户端会根据错误信息，重新向正确的节点发送命令。

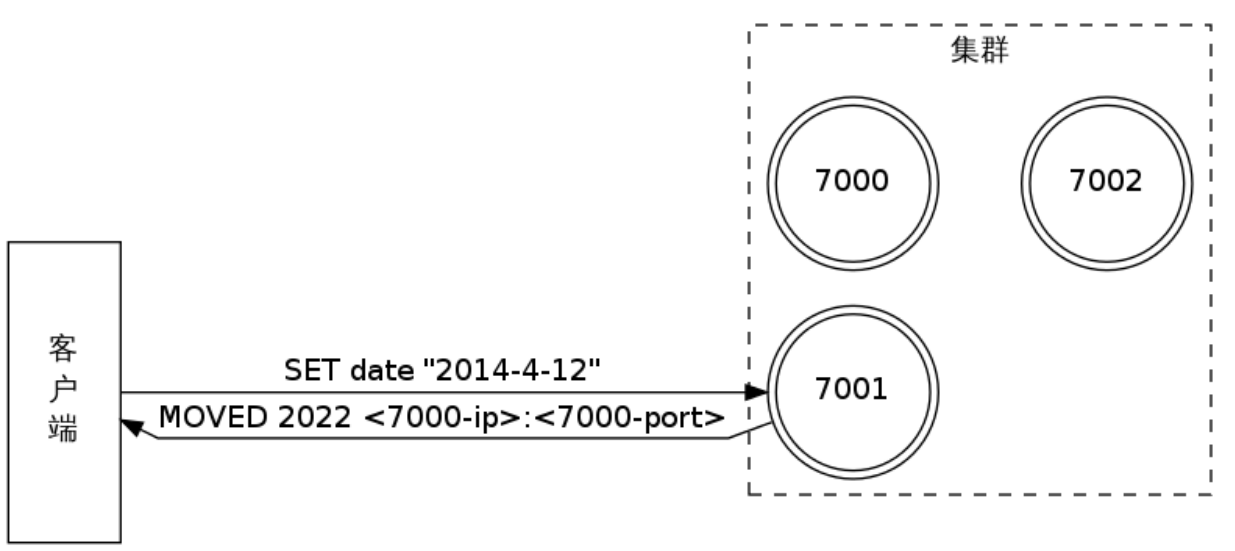
情况一:命令发送给正确的节点



7000:槽 0 ～ 5000
7001:槽 5001 ～ 10000
7002:槽 10001 ～ 16383

键 date 位于 2022 槽，该槽由节点 7000 负责，命令会直接执行。

情况二:命令发送给了错误的节点(1/2)

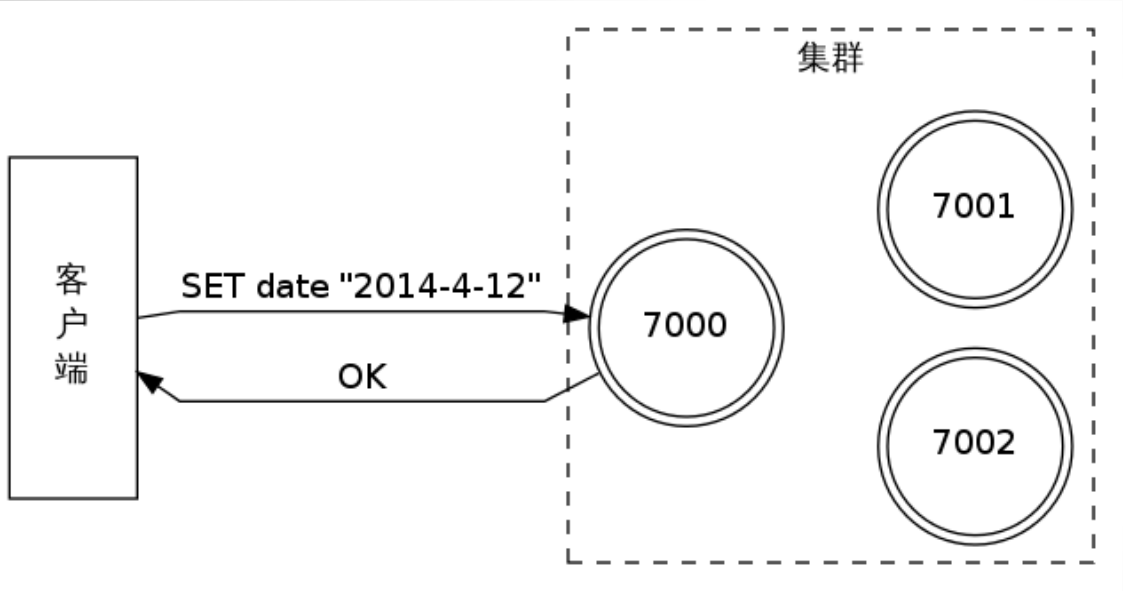


7000:槽 0 ~ 5000
7001:槽 5001 ~ 10000
7002:槽 10001 ~ 16383

转向错误格式:
MOVED <slot> <ip>:<port>

键 date 所在的槽 2022 并非由节点 7001 负责,7001 返回转向错误。

情况二:命令发送给了错误的节点(2/2)

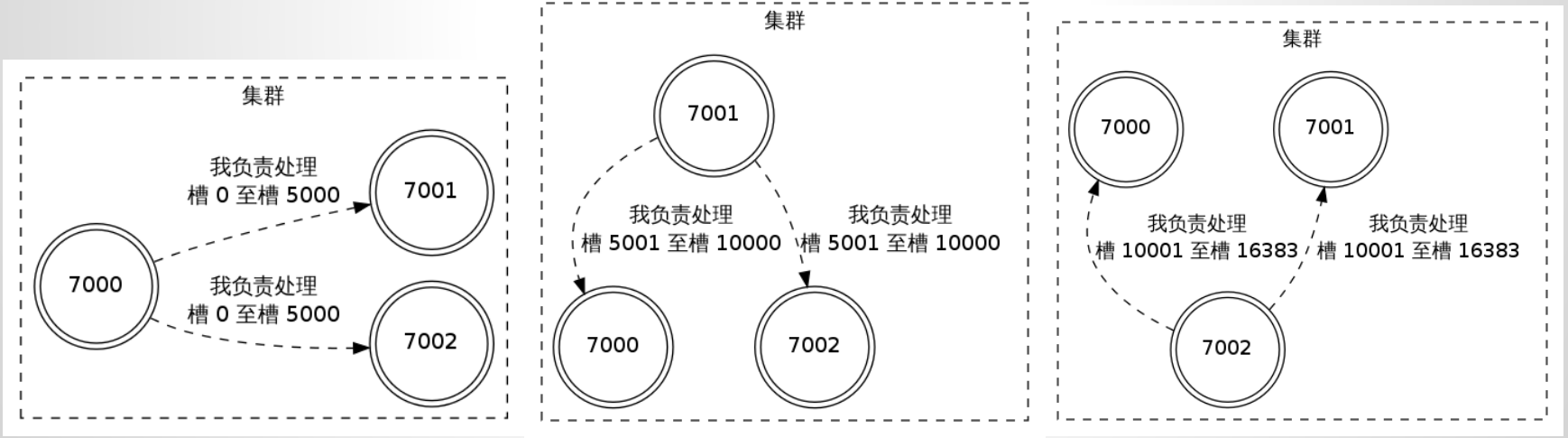


7000:槽 0 ～ 5000
7001:槽 5001 ～ 10000
7002:槽 10001 ～ 16383

客户端根据转向错误的指引，转向至节点 7000，并重新发送命令。

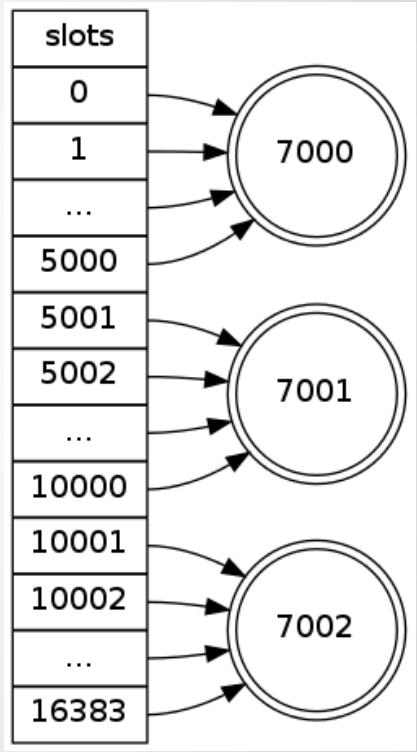
转向错误的实现(1/2)

集群中的节点会**互相告知对方**，自己负责处理哪些槽。



转向错误的实现(2/2)

- 集群中的**每个节点**都会**记录 16384 个槽分别由哪个节点负责**，从而形成一个“槽表”(slot table)。
- 节点在接到命令时，会**检查键所在的槽是否由本节点处理**，如果不是的话，就根据槽号查找槽表，提取出正确节点的地址信息，然后返回转向错误。



集群节点的复制

集群的复制特性**重用了 SLAVEOF 命令的代码**，所以集群节点的复制行为和 SLAVEOF 命令的复制行为**完全相同**。

集群主节点的故障转移

- 集群的下线检测和故障转移等功能的实现没有重用 Sentinel 的代码，但它们的**运作原理基本一样**。
- 和独立运行的 Sentinel 不同，集群对主从节点的下线检测、以及对主节点的故障转移操作，都是由主节点本身负责的，这也即是说，在集群模式里面，**主节点兼顾了 Sentinel 的功能**。
- Redis Sentinel 和集群节点的 Sentinel 特性的**模式不同**使得它们很难直接重用整个代码基，但将来共用一些函数应该还是没有问题的。

谢谢观看！ :）

通过 www.huangz.me 联系我