RedisVIP讲义-持久化、集群

一、今日教学目标

1. 持久化机制
2. Redis集群

三、教学过程

Redis持久化

为什么需要持久化

我们在第一节课的时候就讲过，Redis的数据是存放在内存的，那么存放在内存的话，如果每次关闭或者机器断电，那么我们的数据就丢失了！！

所以，Redis如果要被别人用，那么这个问题肯定要解决，也就是我们需要把数据进行持久化。

我们内存的数据要同步到磁盘，然后我们Redis启动的时候，由于内存中没有数据，需要从磁盘中加载数据，然后基于内存去操作。

这样就既能满足基于内存的操作，保证性能，同时，也能尽可能的防止数据丢失。

持久化的方式有哪些

我们看下官网的介绍：

https://redis.io/docs/manual/persistence/

我们发现重要的就2种方式：RDB跟AOF

RDB

RDB 是 Redis 默认的持久化方案。RDB快照（Redis Database），当满足一定条件的时候，会把当前内存中的数据写入磁盘，生成一个快照文件 dump.rdb（默认）

可配置RDB文件的文件名与路径：

```
# The filename where to dump the DB
dbfilename dump.rdb   //dunmp.rdb文件名

dir ./   //文件路径
```

那么RDB入什么时候触发?

什么时候触发

自动触发

1. 配置触发

   ```
   save 900 1    900s检查一次，至少有1个key被修改就触发
   save 300 10    300s检查一次，至少有10个key被修改就触发
   save 60 10000    60s检查一次，至少有10000个key被修改
   ```

2. shutdown正常关闭

   任何组件，再正常关闭的时候，都会去完成该完成的事情，比如Mysql中的Redolog持久化 正常关闭的时候也会去持久化。

3. flushall指令触发

   数据清空指令会触发RDB操作，并且是触发一个空的RDB文件，所以，如果在没有开启其他的持久化的时候，flushall是可以删库跑路的，在生产环境慎用。

手动触发

1. save 主线程去进行备份，备份期间不会去处理其他的指令，其他指令必须等待
2. bgsave 子线程去进行备份，其他指令正常执行。

操作实例

1.我们首先添加一份数据并且备份到RDB

```
redis> set k1 1
redis> set k2 2
redis> set k3 3
```

2.查看数据是否存在

```
redis>keys *
```

3.我们进行shutdown操作触发RDB快照

```
redis> shutdown
```

4.对现有RDB数据进行备份cp

```
redis>cp dump.rdb dump.rdb.bak
```

5.启动redis

```
redis> src/redis-server& redis.conf
```

发现数据都还在

6.现在模拟数据丢失

```
redis> flushall
```

7.停服务器再启动

```
redis> shutdown

redis> src/redis-server& redis.conf
```

8.发现数据已经丢失，我们现在就要从我们备份的数据恢复，先关闭

```
redis> shutdown
```

9.删除原RDB备份数据

```
redis>rm dump.rdb
```

10.将备份数据改名为dump.rdb

```
mv dump.rdb.bak dump.rdb
```

11.重启服务

```
src/redis-server& redis.conf
```

我们发现数据已经恢复

怎么备份

1.新起子线程，子线程会将当前Redis的数据写入一个临时文件

2.当临时文件写完成后，会替换旧的RDB文件。

当Redis启动的时候，如果只开启了RDB持久化，会从RDB文件中加载数据。

RDB的优势与不足

优势

官网介绍：

RDB advantages

- RDB is a very compact single-file point-in-time representation of your Redis data. RDB files are perfect for backups. For instance you may want to archive your RDB files every hour for the latest 24 hours, and to save an RDB snapshot every day for 30 days. This allows you to easily restore different versions of the data set in case of disasters.
- RDB is very good for disaster recovery, being a single compact file that can be transferred to far data centers, or onto Amazon S3 (possibly encrypted).
- RDB maximizes Redis performances since the only work the Redis parent process needs to do in order to persist is forking a child that will do all the rest. The parent process will never perform disk I/O or alike.
- RDB allows faster restarts with big datasets compared to AOF.
- On replicas, RDB supports partial resynchronizations after restarts and failovers.

1. 是个非常紧凑型的文件，非常适合备份与灾难恢复。
2. 最大限度的提升了性能，会fork一个子进程，父进程永远不会产于磁盘IO或者类似操作。
3. 更快的重启。

简单总结一句话，RDB恢复与备份都非常的快。

不足

官网介绍：

RDB disadvantages

- RDB is NOT good if you need to minimize the chance of data loss in case Redis stops working (for example after a power outage). You can configure different *save points* where an RDB is produced (for instance after at least five minutes and 100 writes against the data set, you can have multiple save points). However you'll usually create an RDB snapshot every five minutes or more, so in case of Redis stopping working without a correct shutdown for any reason you should be prepared to lose the latest minutes of data.
- RDB needs to fork() often in order to persist on disk using a child process. fork() can be time consuming if the dataset is big, and may result in Redis stopping serving clients for some milliseconds or even for one second if the dataset is very big and the CPU performance is not great. AOF also needs to fork() but less frequently and you can tune how often you want to rewrite your logs without any trade-off on durability.

1.数据安全性不是很高，因为是根据配置的时间来备份，假如每5分钟备份一次，也会有5分钟数据的丢失

2.经常fork子进程，所以比较耗CPU，对CPU不是很友好。

AOF

由于RDB的数据可靠性非常低，所以Redis又提供了另外一种持久化方案：Append Only File　简称：AOF

AOF默认是关闭的，你可以再配置文件中进行开启：

```
appendonly no    //默认关闭，可以进行开启

# The name of the append only file (default:
"appendonly.aof")

appendfilename "appendonly.aof"     //AOF文件名
```

追加文件，即每次更改的命令都会附加到我的AOF文件中。

同步机制

AOF,会记录每个写的操作，那么问题来了？我难道每次操作命令又得跟磁盘交互呢？不跟mysql一样每次需要跟磁盘交互呢？

当然不行，所以，redis支持几种策略，由你们自己来决定要不要每次都跟磁盘交互

```
# appendfsync always    表示每次写入都执行fsync(刷新)函数    性能会非常非常慢    但是非常安全
appendfsync everysec    每秒执行一次fsync函数    可能丢失1s的数据
# appendfsync no    由操作系统保证数据同步到磁盘，速度最快    你的数据只需要交给操作系统就行
```

默认1s一次，最多有1s丢失

官网介绍：

How durable is the append only file?

You can configure how many times Redis will `fsync` data on disk. There are three options:

- `appendfsync always`: `fsync` every time new commands are appended to the AOF. Very very slow, very safe. Note that the commands are appended to the AOF after a batch of commands from multiple clients or a pipeline are executed, so it means a single write and a single fsync (before sending the replies).
- `appendfsync everysec`: `fsync` every second. Fast enough (since version 2.4 likely to be as fast as snapshotting), and you may lose 1 second of data if there is a disaster.
- `appendfsync no`: Never `fsync`, just put your data in the hands of the Operating System. The faster and less safe method. Normally Linux will flush data every 30 seconds with this configuration, but it's up to the kernel's exact tuning.

The suggested (and default) policy is to `fsync` every second. It is both fast and relatively safe. The `always` policy is very slow in practice, but it supports group commit, so if there are multiple parallel writes Redis will try to perform a single `fsync` operation.

重写机制

由于AOF是追加的形式，所以文件会越来越大，越大的话，数据加载越慢。所以我们需要对AOF文件进行重写。

何为重写

比如 我们的incr指令，假如我们incr了100次，现在数据是100，但是我们的aof文件中会有100条incr指令，但是我们发现这个100条指令用处不大，假如我们能把最新的内存里的数据保存下来的话。

所以，重写就是做了这么一件事情，把当前内存的数据重写下来，然后把之前的追加的文件删除。

重写流程

在Redis7之前：

1. Redis fork一个子进程，在一个临时文件中写入新的AOF （当前内存的数据生成的新的AOF）
2. 那么在写入新的AOF的时候，主进程还会有指令进入，那么主进程会在内存缓存区中累计新的指令 （但是同时也会写在旧的AOF文件中，就算重写失败，也不会导致AOF损坏或者数据丢失）
3. 如果子进程重写完成，父进程会收到完成信号，并且把内存缓存中的指令追加到新的AOF文件中
4. 替换旧的AOF文件 ，并且将新的指令附加到重写好的AOF文件中。

在Redis7之后，AOF文件不再是一个，所以会有临时清单的概念。 (可以暂时不用了解)

1. 子进程开始在一个临时文件中写入新的基础 AOF。
2. 父级打开一个新的增量 AOF 文件以继续写入更新。如果重写失败，旧的基础和增量文件（如果有的话）加上这个新打开的增量文件就代表了完整的更新数据集，所以我们是安全的。
3. 当子进程完成基础文件的重写后，父进程收到一个信号，并使用新打开的增量文件和子进程生成的基础文件来构建临时清单，并将其持久化。
4. 现在 Redis 对清单文件进行原子交换，以便此 AOF 重写的结果生效。Redis 还会清理旧的基础文件和任何未使用的增量文件。

但是重写是把当前内存的数据，写入一个新的AOF文件，如果当前数据比较大，然后以指令的形式写入的话，会很慢很慢。

所以在4.0之后，在重写的时候是采用RDB的方式去生成新的AOF文件，然后后续追加的指令，还是以指令追加的形式追加的文件末尾。

```
aof-use-rdb-preamble yes   //是否开启RDB与AOF混合模式
```

什么时候重写

配置文件redis.conf

```
# 重写触发机制
auto-aof-rewrite-percentage 100

auto-aof-rewrite-min-size 64mb   就算达到了第一个百分比的大小，也必须大于 64M
```

举例说明上述配置的意义

在 aof 文件小于64mb的时候不进行重写，当到达64mb的时候，就重写一次。重写后的 aof 文件可能是40mb。上面配置了auto-aof-rewrite-percentag为100，即 aof 文件到了80mb的时候，进行重写。

AOF的优势与不足

优势

1.安全性高，就算默认的持久化同步机制，也最多只会导致1s丢失。

2.AOF由于某些原因，比如磁盘满了等导致追加失败，也能通过redis-check-aof 工具来修复

```
//Usage: ./redis-check-aof [--fix] <file.aof>
[root@localhost src]# ./redis-check-aof --fix  append
```

3.格式都是追加的日志，所以可读性更高

不足

1. 数据集一般比RDB大
2. 持久化跟数据加载比RDB更慢
3. 在7.0之前，重写的时候，因为重写的时候，新的指令会缓存在内存区，所以会导致大量的内存使用
4. 并且重写期间，会跟磁盘进行2次IO，一个是写入老的AOF文件，一个是写入新的AOF文件

Redis集群

Redis主从

为什么要有主从?

1. 故障恢复 主挂了或者数据丢失了，我从还会有数据冗余
2. 负载均衡，流量分发 我们可以主写，从库读，减少单实例的读写压力
3. 高可用 我们等下讲的集群 等等，都是基于主从去实现的

安装

安装见预习资料的安装文档

我们现在安装了一个主从，来看下我们的主从信息。

主：192.168.8.129 6379

```
127.0.0.1:6379> info replication
# Replication
role:master  //角色
connected_slaves:1 //从节点数量
slave0:ip=192.168.8.127,port=6379,state=online,offset=7889
9,lag=1  //从节点的信息 状态  偏移量
master_replid:04f4969ab63ce124e870fa1e4920942a5b3448e7
 //# master启动时生成的40位16进制的随机字符串，用来标识master节点
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:78899 //mater已写入的偏移量
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576  //缓冲区大小
repl_backlog_first_byte_offset:1
repl_backlog_histlen:78899 //缓冲区的数据已有大小（是个环形，跟
RedoLog一样会覆盖）
```

从：192.168.8.127 6379 和192.168.8.128：6379

```
127.0.0.1:6379> info replication
# Replication
role:slave  //角色
master_host:192.168.8.129  //主节点IP
master_port:6379  //主节点端口
master_link_status:up  //连接状态  up是正常同步连接状态 down表
示复制端口
master_last_io_seconds_ago:1  //主库多少秒没有发送数据到从库 0-
10
master_sync_in_progress:0  //是否正在跟主服务同步
slave_repl_offset:163  //从节点偏移量
slave_priority:100  //选举时成为主节点的优先级  越大优先级越高 0
不会成为主节点
```

```
slave_read_only:1  //是否为只读从库    如果不是只读，则能自己进行数据写入，默认是只读
connected_slaves:0  //连接的从库实例
master_replid:04f4969ab63ce124e870fa1e4920942a5b3448e7
//master启动时生成的40位16进制的随机字符串，用来标识master节点
master_replid2:0000000000000000000000000000000000000000
 //slave切换master之后，会生成了自己的master标识，之前的master节点的标识存到了master_replid2的位置
master_repl_offset:163 //已写入偏移量
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576  //复制积压的缓存区大小
repl_backlog_first_byte_offset:1
repl_backlog_histlen:163
```

主从数据同步

我们发现，我们的信息中，有个slave_read_only，如果是1 代表只读，也就是默认，当然也可以配置成可写，但是写的数据不会同步到主库，需要自己去保证数据一致性，可能从库写的数据会被主库的数据操作覆盖。出现在从库set一个值后，获取到的是主库覆盖的数据

例如：

1.从库

```
127.0.0.1:6379> set  k1 2  //从库设置k1  2
```

2.主库

```
127.0.0.1:6379> set k1 3   //主库设置k1为3    并且会同步到从库
```

3.从库 从库得到的是主库设置的值

```
127.0.0.1:6379> get k1
"3"
```

所以，我们的从库，一般只用于只读，不会用于数据写入。

那么问题来了，我们的从库的数据怎么跟主库同步的？同步过程是怎么样的？或者主从数据保证一致性的？

建立连接

当首次成为主节点的从节点时，执行replicaof ip port命令的时候就会保存主服务器的IP与端口

并且与主服务器建立连接，接收主节点返回的命令

判断主节点是否有密码 如果有 进行权限校验

保证主从之前保存了各自的信息，并正常连接。

slave发起同步master数据指令

在slave的serverCron方法调用replicationCron方法，里面会发起跟master的数据同步

```
run_with_period(1000) replicationCron();
```

全量同步

1. master服务器收到slave的命令后（psync），判断slave传给我的master_replid 是否跟我的replid一致，如果不一致或者传的是个空的，那么就需要全量同步。

   备注：4.0之前，用的是runId,但是runId每次实例重启时都会改变，会导致主从切换或者重启的时候，都需要全量同步，所以4.0之后采用replid

   replid有master_replid跟master_replid2，当主从切换的时候，我升级为master的slave会继承之前master的replid，所以不需要每次主从切换，都触发全量同步。

   同时slave会把master的master_replid持久化到磁盘。
2. slave首次关联master，从主同步数据，slave肯定是没有主的replid,所以需要进行全量同步。
3. 进行全量同步

1. master开始执行bgsave，生成一个RDB文件，并且把RDB文件传输给我们的slave，同时把master的replid以及offerset（master的数据进度，处理完命令后，都会写入自身的offerset）
2. slave接收到rdb文件后，清空slave自己内存中的数据，然后用rdb来加载数据，这样保证了slave拿到的数据是master生成rdb时候的最新数据。
3. 由于master生成RDB文件是用的bgsave生成，所以，在生成文件的时候，是可以接收新的指令的。那么这些指令，我们需要找一个地方保存，等到slave加载完RDB文件以后要同步给slave。
    1. 在master生成rdb文件期间，会接收新的指令，这些新的指令会保存在一个内存区间，这个内存区间就是replication_buffer。

       我们可以通过以下方式设置replication_buffer的大小

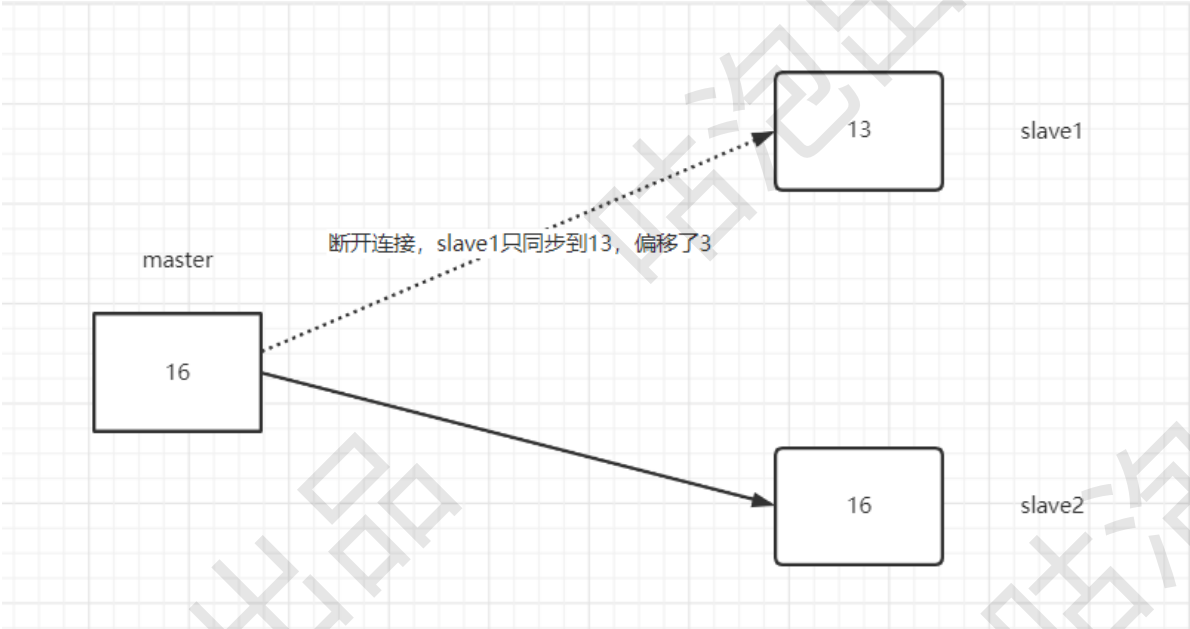
       ```
       client-output-buffer-limit replica 256mb 64mb 60
           256mb   硬性限制，大于256M断开连接
           64mb 60   软限制   超过64M 并且超过了60s还没进行同步内存数据就会断开连接
       ```

    2. 这个空间不能太小，如果太小，为了数据安全，会关闭跟从库的网络连接。再次连接得重新全量同步，但是问题还在，会导致无限的在同步

增量(部分)同步

为什么需要增量同步?

我们刚才有说过，每个节点都会保存数据的偏移量（从0开始，随着数据的添加递增）。那么就有可能出现slave跟master网络断开一小会，然后发起数据同步的场景，如图:



slave1由于网络断开，偏移量跟master差了3。

那么当slave1重新跟master连接后，同样的会去跟master连接触发数据同步。但是这个时候还需不需要全量同步？我们肯定是尽可能的减少全量同步。

所以，当master收到slave的指令时：

1. 判断slave1传给我的master_replid 是否跟master的replid一致，由于之前已经连接过保过了master的replid，满足条件。
2. 所以我希望只同步断开连接后没有同步到的数据，必须，我slave1只差了3的数据。那么我需要找到这个3的数据。所以master中有个另外的积压缓存（replication_backlog_buffer）。

   我们可以设置replication_backlog_buffer的大小：

```
# The bigger the replication backlog, the longer the
time the replica can be 复制积压越大，复制副本的时间越长
# disconnected and later be able to perform a partial
resynchronization.  已断开连接，稍后可以执行部分重新同步。
#
# The backlog is only allocated once there is at least
a replica connected. 只有在至少连接了一个副本后，才会分配积压工作。
#
# repl-backlog-size 1mb
```

我们也不可能无限制的往里面写数据， replication_backlog_buffer的数据是会覆盖的。

3. 所以，我们slave1跟master相差的3条数据可能会被覆盖，如果覆盖了，触发全量，如果没有覆盖，即能找到相差的3条数据。增量即可。

指令同步

master写入的指令，异步同步至slave，如果有slave，写入replication_backlog_buffer

流程图

https://www.processon.com/view/link/62cd1c400e3e74070446a5b2

Redis sentinel哨兵

主从，我们虽然解决了比如负载、数据备份等问题。

但是，我们发现我们master挂了，slave是不会直接升级为主，必须手动把slave升级为主。

我们看master挂了后，slave只是状态变更为down

```
127.0.0.1:6379> info replication
# Replication
role:slave
master_host:192.168.8.129
master_port:6379
master_link_status:down
master_last_io_seconds_ago:-1
master_sync_in_progress:0
slave_repl_offset:323482
master_link_down_since_seconds:147
slave_priority:100
slave_read_only:0
connected_slaves:0
master_replid:04f4969ab63ce124e870fa1e4920942a5b3448e7
```

```
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:323482
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:214059
repl_backlog_histlen:109424
```

我们不希望master挂了后,还需要人半夜来解决这件事情，来启动slave容灾。所以就有了我们的哨兵集群，以及我们后面的cluster集群。

官网地址：https://redis.io/docs/manual/sentinel

我们先看下官网对Sentinel的介绍。

High availability for non-clustered Redis

Redis Sentinel provides high availability for Redis when not using Redis Cluster.

Redis Sentinel also provides other collateral tasks such as monitoring, notifications and acts as a configuration provider for clients.

This is the full list of Sentinel capabilities at a macroscopic level (i.e. the *big picture*):

- **Monitoring**. Sentinel constantly checks if your master and replica instances are working as expected.
- **Notification**. Sentinel can notify the system administrator, or other computer programs, via an API, that something is wrong with one of the monitored Redis instances.
- **Automatic failover**. If a master is not working as expected, Sentinel can start a failover process where a replica is promoted to master, the other additional replicas are reconfigured to use the new master, and the applications using the Redis server are informed about the new address to use when connecting.

- **Configuration provider**. Sentinel acts as a source of authority for clients service discovery: clients connect to Sentinels in order to ask for the address of the current Redis master responsible for a given service. If a failover occurs, Sentinels will report the new address

简单翻译下：

Redis sentinel在不适用Cluster集群的时候，为Redis提供了高可用性。

并且提供了监测、通知、自动故障转移、配置提供等功能。

监控：能够监控我的Redis各实例是否正常工作

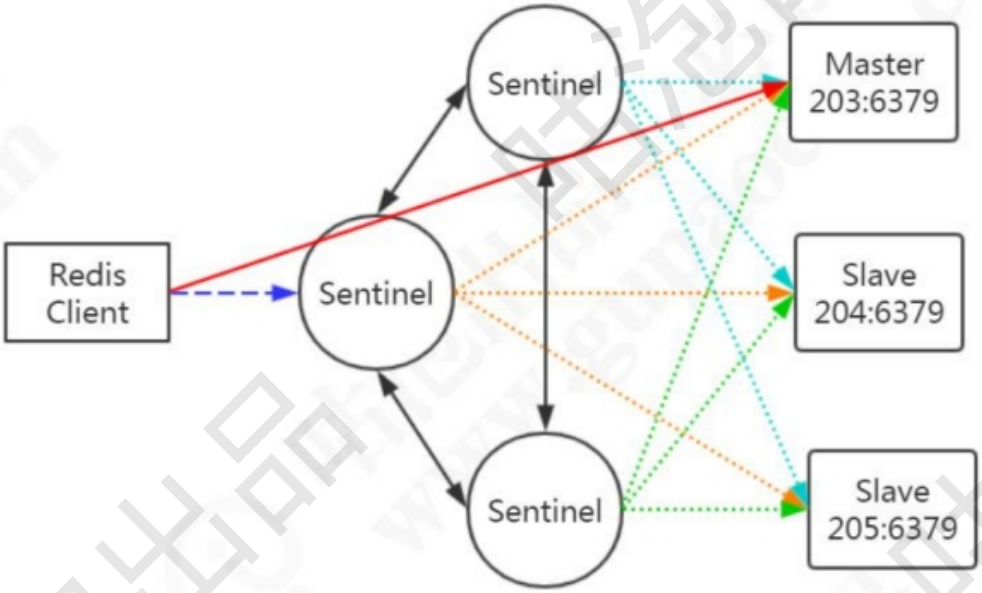
通知：如果Redis的实例出现问题，能够通知给其他实例以及sentinel

自动故障转移：当我的master宕机，slave可以自动升级为master

配置提供：sentinel可以提供Redis的master实例地址，那么客户端只需要跟sentinel进行连接，master挂了后会提供新的master

从sentinel的名字我们就能看出来，哨兵。哨兵就比如我们中南海的门卫，是来保证我们整个中南海的安全，监控中南海是否安全。所以哨兵应该是独立于我们Redis服务外的一个监控服务。哨兵是为了保证其他人的安全的。

其实sentinel其实是独立于Redis服务的单独的服务，并且他们之间是相互通信的



安装

安装见预习资料的安装文档

基于安装文档，我们知道了，sentinel能够实现我们的主从切换。

客户端的使用

没有sentinel

```
spring.redis.port=6379      --主服务器端口
spring.redis.host=192.168.8.127    --主服务器地址
```

假如主节点宕机

访问接口：http://localhost:8080/list

```
2022-08-04 20:47:19.569  INFO 24604 --- [xecutorLoop-1-5] i.l.core.protocol.ConnectionWatchdog     : Reconnecting, last destination was 192.168.8.127:6379
2022-08-04 20:47:19.569  INFO 24604 --- [xecutorLoop-1-4] i.l.core.protocol.ConnectionWatchdog     : Reconnecting, last destination was 192.168.8.127:6379
2022-08-04 20:47:19.695 ERROR 24604 --- [nio-8080-exec-2] o.a.c.c.C.[.[.[/].[dispatcherServlet]    : Servlet.service() for servlet [dispatcherServlet] in context with path []
 threw exception [Request processing failed; nested exception is org.springframework.dao.QueryTimeoutException: Redis command timed out; nested exception is io.lettuce.core
 .RedisCommandTimeoutException: Command timed out after 1 minute(s)] with root cause
```

超过时间后，没法进行重连

使用sentinel

```
spring.redis.sentinel.nodes=192.168.8.128:26379,192.168.8.127:26379,192.168.8.129:26379  //sentinel的节点地址
spring.redis.sentinel.master=mymaster    //配置的mater的名称
```

当主节点宕机

访问接口：http://localhost:8080/list

我们发现当主节点宕机后，我们会自动重连到升级为master的slave节点

哨兵故障转移流程流程

那么我们的哨兵到底怎么去发现我们的master是挂了的？挂了后让slave变成master的整个流程是咋样的？

首先，我们要知道，sentinel与sentinel之间，以及sentinel与master/slave之间都会进行通信。

发现master故障

官网说明：https://redis.io/docs/manual/sentinel/#sdown-and-odown-failure-state

1. 当我们某个sentinel 跟master通信时（默认1s发送ping），发现我在一定时间内（down-after-milliseconds） 没有收到master的有效的回复。这个时候这个sentinel就会人为master是不可用，但是有多个sentinel，它现在只有1个人觉得宕机了，这个时候不会触发故障转移，

只会标记一个状态，这个状态就是SDOWN（Subjectively Down condition），也就是我们讲的主观下线。
2. SDOWN时，不会触发故障转移，会去询问其他的sentinel，其他的sentinel是否能连上master，如果超过Quorum(法定人数)的sentinel都认为master不可用，都标记SDOWN状态，这个时候，master可能就真的是down了。那么就会将master标为ODOWN（Objectively Down condition 客观下线)

进行故障转移

1. 当状态为ODWON的时候，我们就需要去触发故障转移，但是有这么多的sentinel，我们需要选一个sentinel去做故障转移这件事情，并且这个sentinel在做故障转移的时候，其他sentinel不能进行故障转移。
2. 所以，我们需要选举一个sentinel来做这件事情:其中这个选举过程有2个因素。
    1. Quorum如果小于等于一半，那么必须超过半数的sentinel授权，你才能去做故障迁移，比如5台 sentinel，你配置的Quorum=2，那么选举的时候必须有3（5台的一半以上）人同意
    2. Quorum如果大于一半，那么必须Quorum的sentinel授权，故障迁移才能启动。

选哪个slave来变成master

1. **与master的断开连接时间**

   如果slave与主服务器断开的连接时间超过主服务器配置的超时时间（down-after-milliseconds）的十倍，被认为不适合成为master。直接去除资格
2. **优先级**

   https://redis.io/docs/manual/sentinel/#replica-selection-and-priority

   配置 replica-priority，replica-priority越小，优先级越高，但是配置为0的时候，永远没有资格升为master
3. **已复制的偏移量**

   比较slave的赋值的数据偏移量，数据最新的优先升级为master

4. **Run ID （每个实例启动都会有个Run ID ）**

   通过info server可以查看。

Sentinel导致的数据一致性问题解析

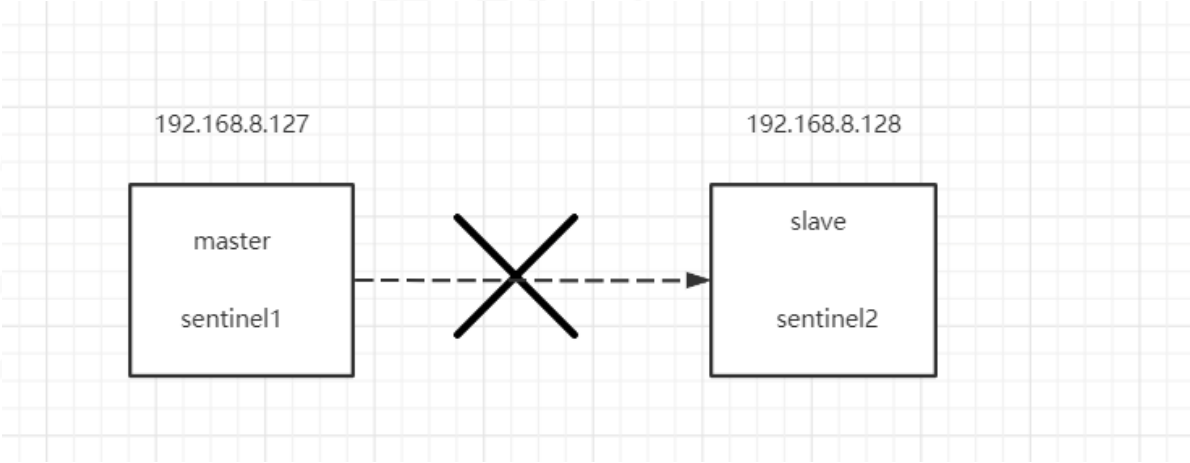
官方建议配置至少3个sentinel。

官方也有相关说明：https://redis.io/docs/manual/sentinel/#example-1-just-two-sentinels-dont-do-this

原因官方也有说明：

1.如果只有1个sentinel实例，则这个实例挂了就不能保证sentinel的高可用性。

2.如果只有2个sentinel实例



如果，我配置了2个sentinel ，分别在127跟128。那么有啥问题?

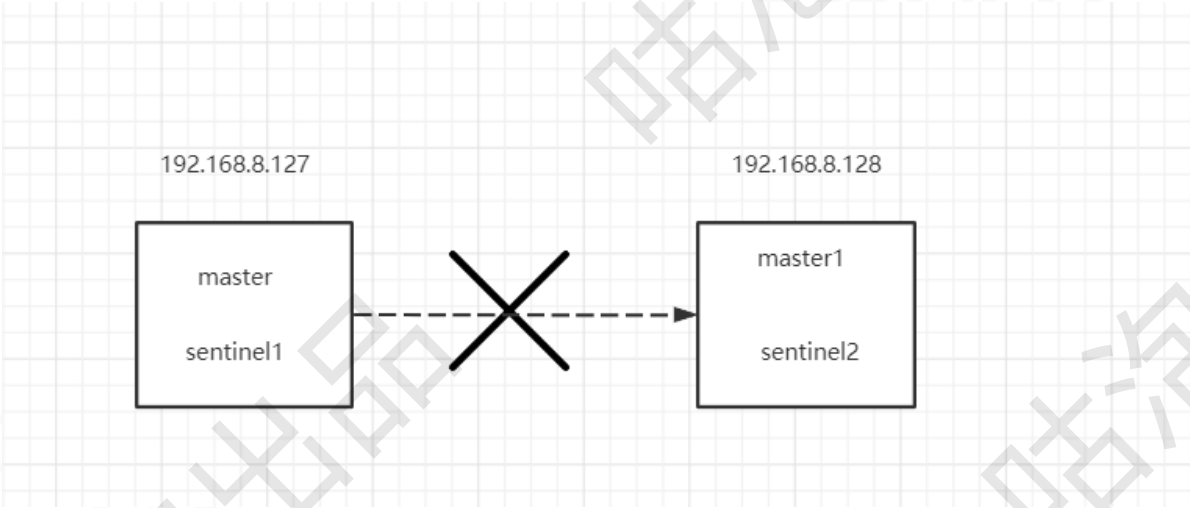
假如，127跟128的网络断开，但是他们各自还是能正常运行的。也就是我们发生了分区容错。

假如我们配置的quorum=1；可能发生：

1.sentinel2检测到master不可用，因为127跟128网络断开，这个时候会触发主观下线，同时，sentinel2只能连接到1个sentinel，也满足半数以上原则（只有1个sentinel2）。

2.sentinel2开启故障转移，将128的slave升级为master,就出现了2个master，这个情况也叫作脑裂。并且客户端会连接到2个master。 （客户端连的是sentinel集群，所以sentinel1连到127.sentinel2 连到128）

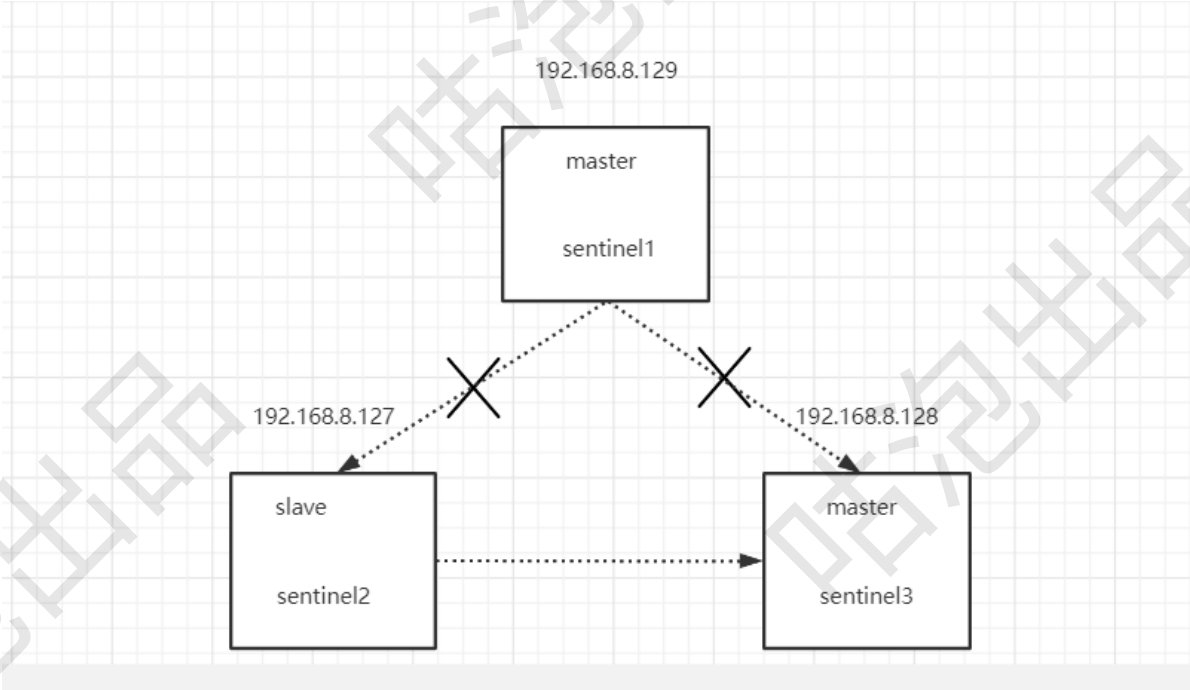
3.2个master都会写入数据，当网络恢复后，127会变成从，127的数据会从128去拿取，这个时候127的数据就会丢失。



脑裂问题

脑裂问题其实就是我会有2个master,client会从不同的master写数据，从而在master恢复的时候会导致数据丢失。

所以，只要发生分区容错性，不管多少节点都会出现。比如3个节点。



初始129是master,假如129网络断开，跟127.128连接断开后，128sentinel发起故障转移。发现sentinel的个数超过一半，能够发起故障转移。将128升级为master，导致128.129同时2个master并可用。

解决方案：

在Redis.cfg文件中有2个配置：

```
min-replicas-to-write 1          至少有1个从节点同步到我主节点的数据，但是由于是异步同步，所以是最终一致性  不会确保有数据写入
min-replicas-max-lag 10          判断上面1个的延迟时间必须小于等于10s
```

按上述场景

129由于没有slave同步，不满足我配置的要求，不能进行数据写入。尽量减少数据一致性问题。

为什么一般部署奇数台？ 因为4跟3 能容许挂掉的机器数量是一样的，都是1台

RedisCluster

我们发现sentinel提供了比如监控、自动故障转移、客户端配置等高可用的方案，但是没有分片功能。

何为分片：就是我希望把数据分布到不同的节点。这样如果某些节点异常，其他数据能正常提供服务，跟我们微服务的思想很相似。

所以，我们的cluster提供了比Sentinel更高级的集群方案。我们看官网的介绍

So in practical terms, what do you get with Redis Cluster?

- The ability to **automatically split your dataset among multiple nodes**.

- The ability to **continue operations when a subset of the nodes are experiencing failures** or are unable to communicate with the rest of the cluster.

简单翻译下，我们提供

1.多个节点之间的数据拆分，也就是我们的数据分片

2.当某些节点遇到故障的时候，其他的节点还能继续服务。

hash slot(虚拟槽)

那么如何进行分片？分片要做的事情就是要把不同的数据放到不同的实例里面去。就相当于我们分表，不同的数据放到不同的表里。

普通取模问题

那么我们经常用到的方法有：

取模 直接取模实例数，得到key的hash值，取模实例数，假如实例是3，那么取模后得到0-2的值，每个值代表一个实例。

但是这种取模有一个问题，假如我做了实例的扩容与缩容，那么全部数据要进行迁移。

假如我之前是3台，扩容到4台，那么所有的数据都必须重新rehash。

所以，Redis里面提出了一个Hash槽，也叫作虚拟槽的概念。什么是虚拟槽，其实就是虚拟节点。

Redis cluster中有16384个虚拟槽。

我们的key会跟槽对应，怎么对应?

我们会根据key 通过CRC16 取模 16383 得到一个0到16383的值 计算公式：slot = CRC16(key) & 16383，得到的值，就代表这个key在哪个虚拟槽。

举例：

假如

```
set k1 1:
CRC16(k1) & 16383=11901
set k2 1:
CRC16(k2) & 16383=10
set k3 1:
CRC16(k3) & 16383=6666
```

那么我们k1对应的槽是11901　　k2对应的槽是10　　k3对应的槽是6666

key跟槽的关系是根据key算出来的，后续不能变动。

如果想把相关key放入一个虚拟槽,也就是一个实例节点，我们可以采用{},那么就只会根据{}里面的内容计算hash槽!

比如:

huihui{18} 跟 james{18} 就会在一个虚拟槽

那么key怎么去放到我们的真实节点?

我们的虚拟槽会跟我们的真实节点进行对应，这个关系是可以变更的。可以设置某个实例节点包含哪些虚拟槽，但是至少得有3个主实例节点，主节点可以有自己的从节点。

假如，我们现在有6台实例，三主三从。 主跟槽的对应关系如下

master1　0-5460虚拟槽

master2　5461-10922虚拟槽

master3 10923-16383虚拟槽

我们可以知道，k1 的虚拟槽是10001 ，所以放入master3,k2的虚拟槽是10 放入master1 ,k3的虚拟槽为666，放入master2。

从节点的数据全部来源于主，所以k1放入master3的从，以此对应。

我们可以通过指令

```
127.0.0.1:6380> cluster nodes   查看当前节点的虚拟槽信息
```

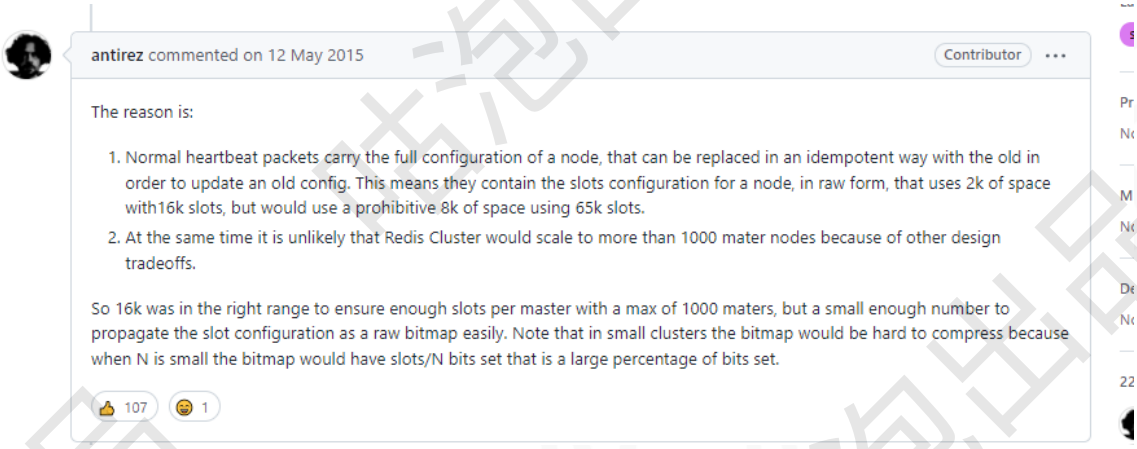
那么每次节点的扩容与缩容，只需要改变节点跟虚拟槽的关系即可，不需要全部变动。

节点移动、宕机故障转移、扩容缩容

见安装实战文档

为什么虚拟槽是16384?

作者的答复 https://github.com/redis/redis/issues/2576



antirez commented on 12 May 2015 Contributor

The reason is:

1. Normal heartbeat packets carry the full configuration of a node, that can be replaced in an idempotent way with the old in order to update an old config. This means they contain the slots configuration for a node, in raw form, that uses 2k of space with16k slots, but would use a prohibitive 8k of space using 65k slots.
2. At the same time it is unlikely that Redis Cluster would scale to more than 1000 mater nodes because of other design tradeoffs.

So 16k was in the right range to ensure enough slots per master with a max of 1000 maters, but a small enough number to propagate the slot configuration as a raw bitmap easily. Note that in small clusters the bitmap would be hard to compress because when N is small the bitmap would have slots/N bits set that is a large percentage of bits set.

107 1

翻译下：

- 正常的心跳包携带节点的完整配置，可以用旧的方式以幂等的方式替换，以更新旧的配置。这意味着它们包含原始形式的节点的插槽配置，该节点使用 2k 空间和 16k 插槽，但使用 65k 插槽会使用令人望而却步的 8k 空间。
- 同时，由于其他设计权衡，Redis Cluster 不太可能扩展到 1000 多个主节点。

所以 16k 在正确的范围内，以确保每个 master 有足够的插槽，最多 1000 个 maters，但数量足够小，可以轻松地将插槽配置作为原始位图传播。请注意，在小型集群中，位图将难以压缩，因为当 N 较小时，位图将设置插槽/N 位，这占位设置的很大百分比。

我们发现，他们cluster节点是会相互心跳监测，进行数据交互的。我们看下交互的数据头

cluster.h文件下

```
typedef struct {
    char sig[4];        /* Signature "RCmb" (Redis Cluster
message bus). */
    uint32_t totlen;    /* Total length of this message */
    uint16_t ver;       /* Protocol version, currently set
to 1. */
    uint16_t port;      /* TCP base port number. */
    uint16_t type;      /* Message type */
    uint16_t count;     /* Only used for some kind of
messages. */
    uint64_t currentEpoch;  /* The epoch accordingly to
the sending node. */
    uint64_t configEpoch;   /* The config epoch if it's a
master, or the last
                               epoch advertised by its
master if it is a
                               slave. */
    uint64_t offset;    /* Master replication offset if
node is a master or
                           processed replication offset if
node is a slave. */
    char sender[CLUSTER_NAMELEN]; /* Name of the sender
node */
    unsigned char myslots[CLUSTER_SLOTS/8];
    char slaveof[CLUSTER_NAMELEN];
    char myip[NET_IP_STR_LEN];    /* Sender IP, if not all
zeroed. */
    char notused1[34];  /* 34 bytes reserved for future
usage. */
    uint16_t cport;      /* Sender TCP cluster bus port */
```

```
    uint16_t flags;      /* Sender node flags */
    unsigned char state; /* Cluster state from the POV of
the sender */
    unsigned char mflags[3]; /* Message flags:
CLUSTERMSG_FLAG[012]_... */
    union clusterMsgData data;
} clusterMsg;
```

我们发现有个char类型的　unsigned char myslots[CLUSTER_SLOTS/8];

其中大小为当前的槽的数量/8

16384的话大小为：16384/8　一个char在c语言中大小被定义为1Byte。所以中大小为16384/8/1024=2kb

但是假如使用65K的虚拟槽，为什么要举例65k

因为crc16算法得到的hash值是16bit　最大的值是65536、通过计算能达到8k，数据传输会很慢。

并且一般场景下，16384个节点已经能满足我们的需求。所以取了一个16384，提升了我们的传输性能。

四、课后总结

五、下节课预告

Redis场景及复习！