

分布式缓存

Redis集群



黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

单点Redis的问题



数据丢失问题

Redis是内存存储，服务重启可能会丢失数据

单点Redis的问题



并发能力问题

单节点Redis并发能力虽然不错，但也无法满足如618这样的高并发场景

单点Redis的问题



故障恢复问题

如果Redis宕机，则服务不可用，需要一种自动的故障恢复手段

单点Redis的问题



存储能力问题

Redis基于内存，单节点能存储的数据量难以满足海量数据需求



Redis是内存存储

单点Redis的问题

数据丢失问题

实现Redis数据持久化

01

02

并发能力问题

搭建主从集群，实现读写分离

04

03

存储能力问题

搭建分片集群，利用插槽机制实现动态扩容

故障恢复问题

利用Redis哨兵，实现健康检测和自动恢复



目录

Contents

- ◆ Redis持久化
- ◆ Redis主从
- ◆ Redis哨兵
- ◆ Redis分片集群



Redis持久化

- RDB持久化
- AOF持久化



目录

Contents

- ◆ RDB持久化
- ◆ AOF持久化

RDB

RDB全称Redis Database Backup file（Redis数据备份文件），也被叫做Redis数据快照。简单来说就是把内存中的所有数据都记录到磁盘中。当Redis实例故障重启后，从磁盘读取快照文件，恢复数据。

快照文件称为RDB文件，默认是保存在当前运行目录。

```
[root@localhost ~]# redis-cli  
127.0.0.1:6379> save #由Redis主进程来执行RDB，会阻塞所有命令
```

Redis停机时会执行一次RDB。

RDB

首先需要在Linux系统中安装一个Redis，如果尚未安装的同学，可以参考课前资料《Redis集群.md》：



Redis集群.md

RDB

Redis内部有触发RDB的机制，可以在redis.conf文件中找到，格式如下：

```
# 900秒内，如果至少有1个key被修改，则执行bgsave，如果是save ""则表示禁用RDB
save 900 1
save 300 10
save 60 10000
```

RDB的其它配置也可以在redis.conf文件中设置：

```
# 是否压缩，建议不开启，压缩也会消耗cpu，磁盘的话不值钱
rdbcompression yes

# RDB文件名称
dbfilename dump.rdb

# 文件保存的路径目录
dir ./
```

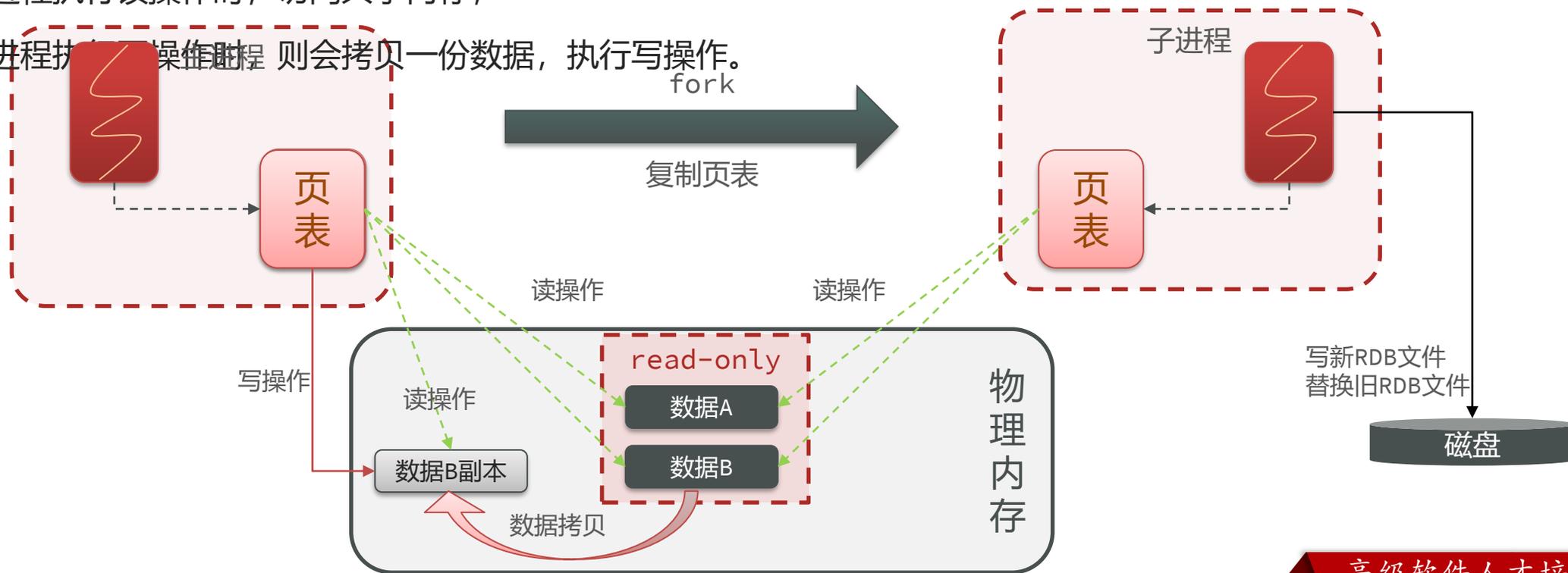
RDB

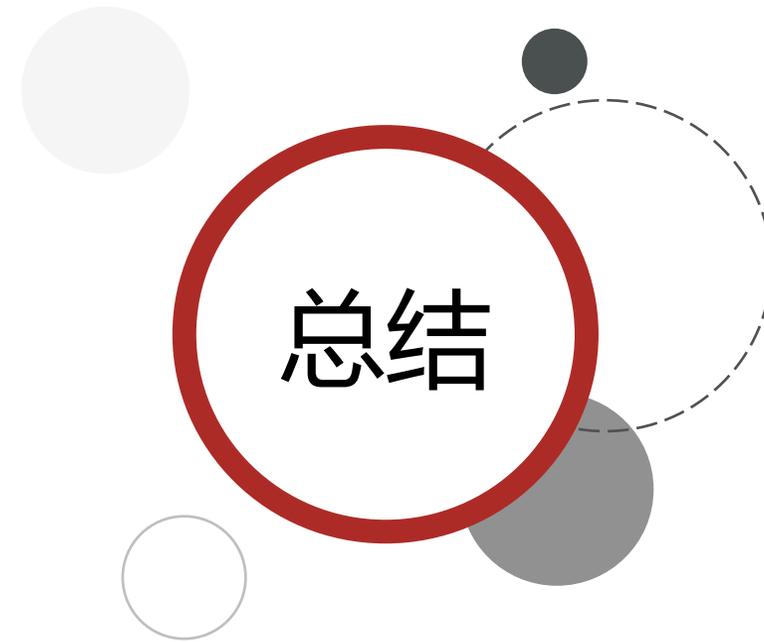
bgsave开始时fork主进程得到子进程，子进程共享主进程的内存数据。完成fork后读取内存数据并写入 RDB 文件

。

fork采用的是copy-on-write技术：

- 当主进程执行读操作时，访问共享内存；
- 当主进程执行写操作时，则会拷贝一份数据，执行写操作。





总结

RDB方式bgsave的基本流程?

- fork主进程得到一个子进程，共享内存空间
- 子进程读取内存数据并写入新的RDB文件
- 用新RDB文件替换旧的RDB文件。

RDB会在什么时候执行? save 60 1000代表什么含义?

- 默认是服务停止时。
- 代表60秒内至少执行1000次修改则触发RDB

RDB的缺点?

- RDB执行间隔时间长，两次RDB之间写入数据有丢失的风险
- fork子进程、压缩、写出RDB文件都比较耗时



目录

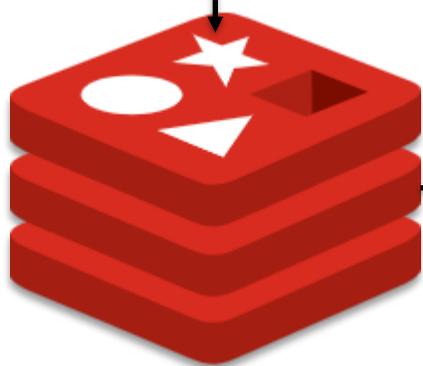
Contents

- ◆ RDB持久化
- ◆ AOF持久化

AOF

AOF全称为Append Only File（追加文件）。Redis处理的每一个写命令都会记录在AOF文件，可以看做是命令日志文件。

```
[root@localhost redis-6.2.4]# redis-cli  
127.0.0.1:6379> set num 123  
OK
```



```
$3  
set  
$3  
num  
$3  
123
```

AOF

AOF

AOF默认是关闭的，需要修改redis.conf配置文件来开启AOF：

```
# 是否开启AOF功能，默认是no  
appendonly yes  
# AOF文件的名称  
appendfilename "appendonly.aof"
```

AOF的命令记录的频率也可以通过redis.conf文件来配：

```
# 表示每执行一次写命令，立即记录到AOF文件  
appendfsync always  
# 写命令执行完先放入AOF缓冲区，然后表示每隔1秒将缓冲区数据写到AOF文件，是默认方案  
appendfsync everysec  
# 写命令执行完先放入AOF缓冲区，由操作系统决定何时将缓冲区内容写回磁盘  
appendfsync no
```

配置项	刷盘时机	优点	缺点
Always	同步刷盘	可靠性高，几乎不丢数据	性能影响大
everysec	每秒刷盘	性能适中	最多丢失1秒数据
no	操作系统控制	性能最好	可靠性较差，可能丢失大量数据

AOF

因为是记录命令，AOF文件会比RDB文件大的多。而且AOF会记录对同一个key的多次写操作，但只有最后一次写操作才有意义。通过执行**bgrewriteaof**命令，可以让AOF文件执行重写功能，用最少的命令达到相同效果。



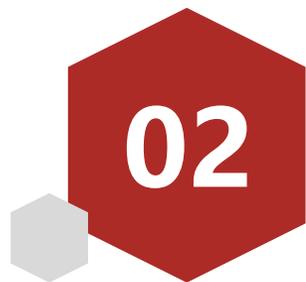
Redis也会在触发阈值时自动去重写AOF文件。阈值也可以在redis.conf中配置：

```
# AOF文件比上次文件 增长超过多少百分比则触发重写
auto-aof-rewrite-percentage 100
# AOF文件体积最小多大以上才触发重写
auto-aof-rewrite-min-size 64mb
```

AOF

RDB和AOF各有自己的优缺点，如果对数据安全性要求较高，在实际开发中往往会**结合**两者来使用。

	RDB	AOF
持久化方式	定时对整个内存做快照	记录每一次执行的命令
数据完整性	不完整，两次备份之间会丢失	相对完整，取决于刷盘策略
文件大小	会有压缩，文件体积小	记录命令，文件体积很大
宕机恢复速度	很快	慢
数据恢复优先级	低，因为数据完整性不如AOF	高，因为数据完整性更高
系统资源占用	高，大量CPU和内存消耗	低，主要是磁盘IO资源 但AOF重写时会占用大量CPU和内存资源
使用场景	可以容忍数分钟的数据丢失，追求更快的启动速度	对数据安全性要求较高常见



Redis主从

- 搭建主从架构
- 主从数据同步原理



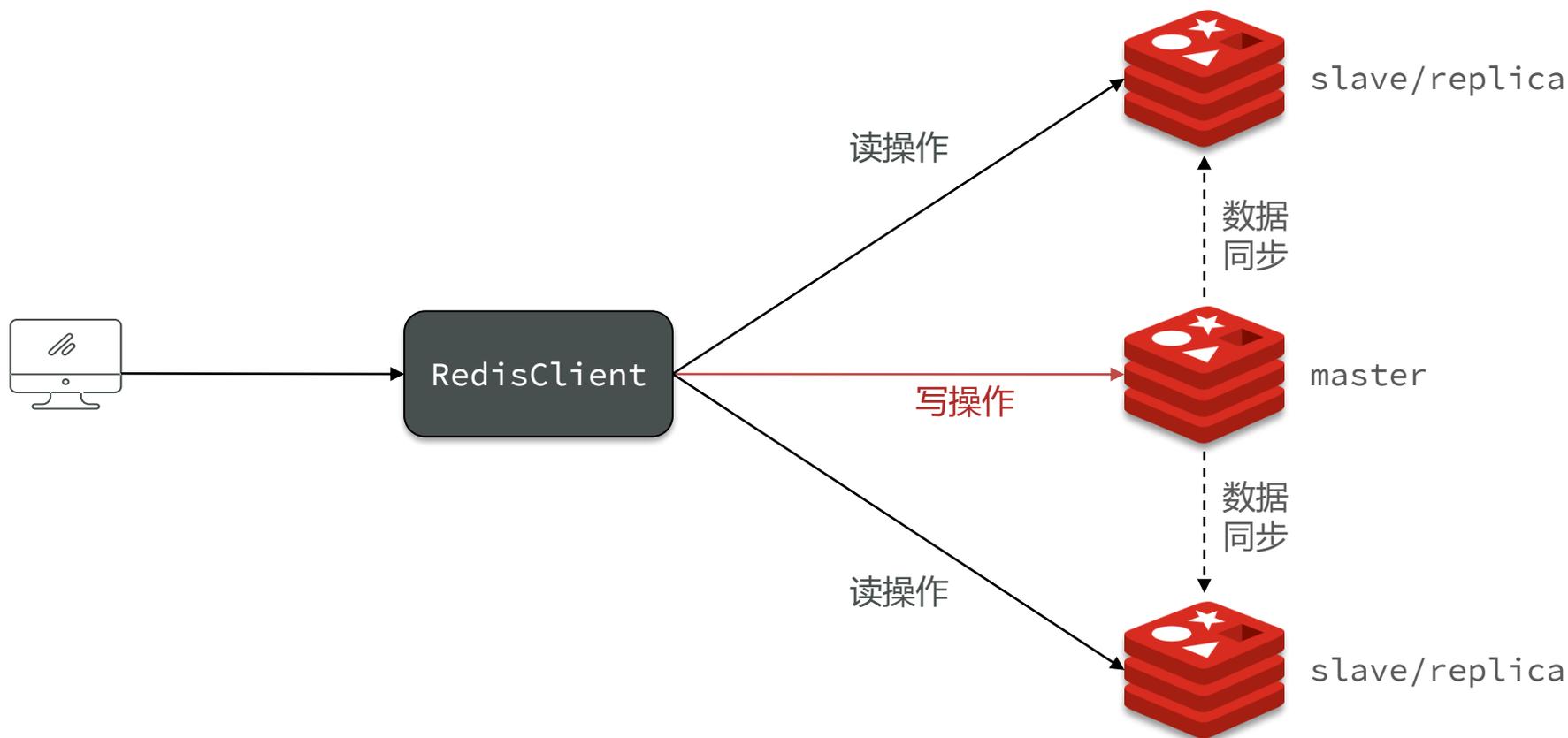
目录

Contents

- ◆ 搭建主从架构
- ◆ 主从数据同步原理

搭建主从架构

单节点Redis的并发能力是有上限的，要进一步提高Redis的并发能力，就需要搭建主从集群，实现读写分离。

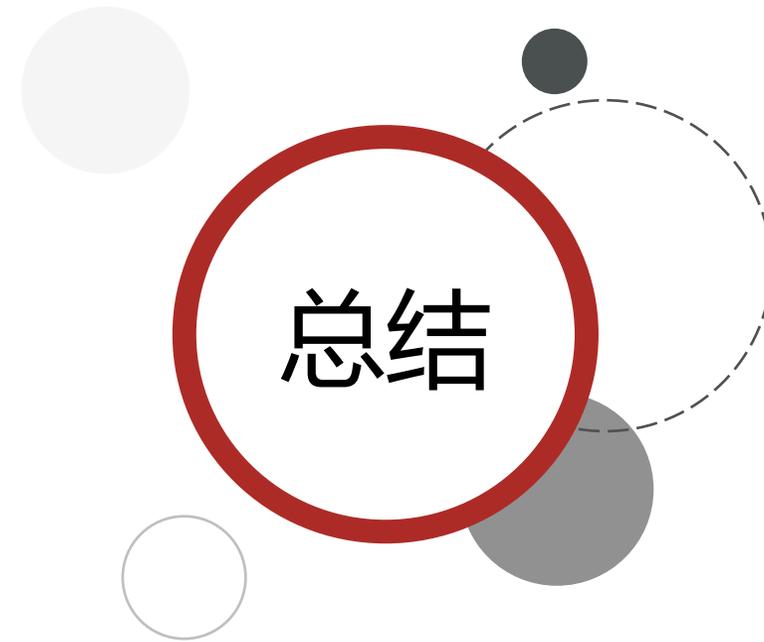


搭建主从架构

具体搭建流程参考课前资料《Redis集群.md》：



Redis集群.md



总结

假设有A、B两个Redis实例，如何让B作为A的slave节点?

- 在B节点执行命令：slaveof A的IP A的port



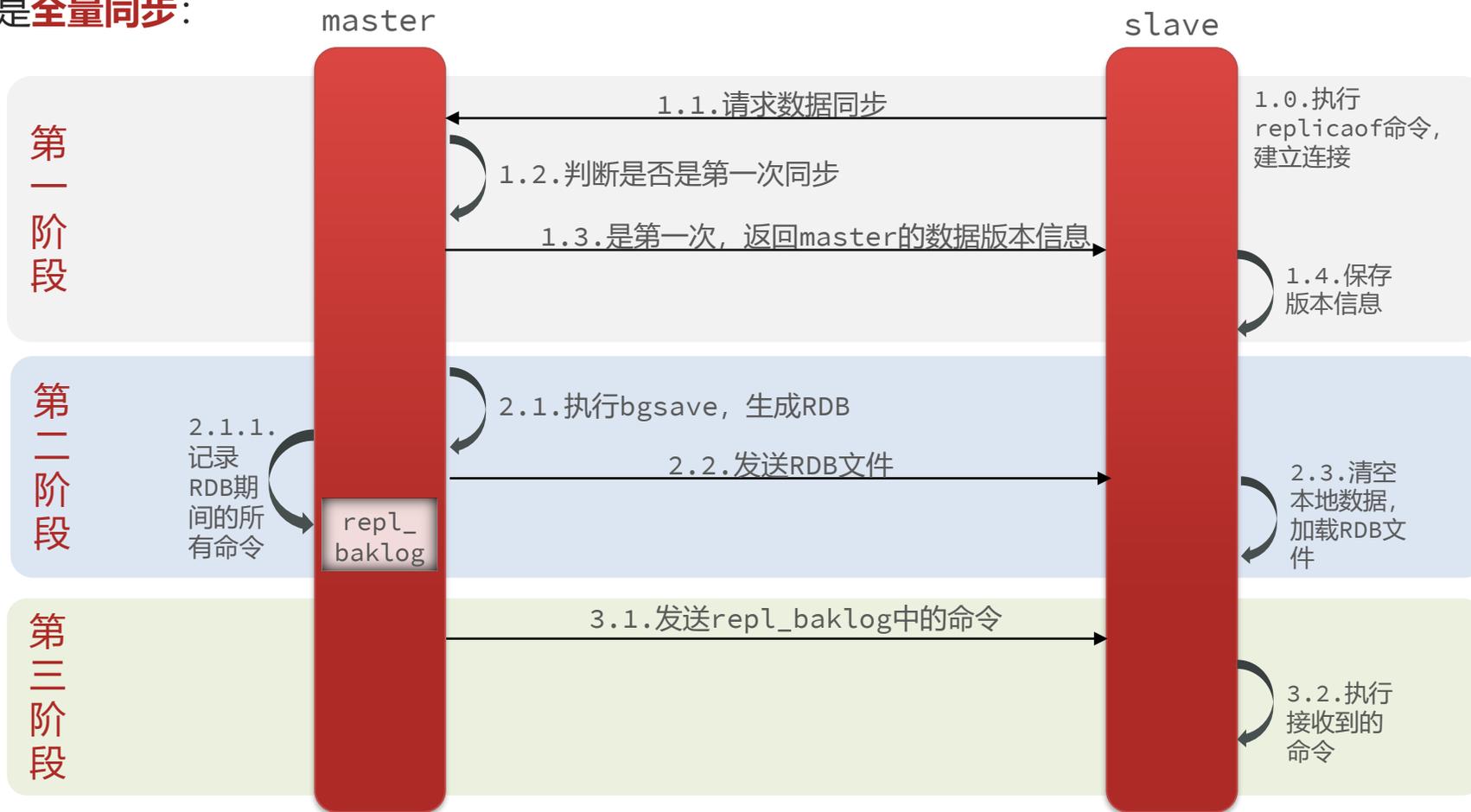
目录

Contents

- ◆ 搭建主从架构
- ◆ 数据同步原理

数据同步原理

主从第一次同步是**全量同步**:



数据同步原理

master如何判断slave是不是第一次来同步数据？这里会用到两个很重要的概念：

- **Replication Id**：简称replid，是数据集的标记，id一致则说明是同一数据集。每一个master都有唯一的replid，slave则会继承master节点的replid
- **offset**：偏移量，随着记录在repl_baklog中的数据增多而逐渐增大。slave完成同步时也会记录当前同步的offset。如果slave的offset小于master的offset，说明slave数据落后于master，需要更新。

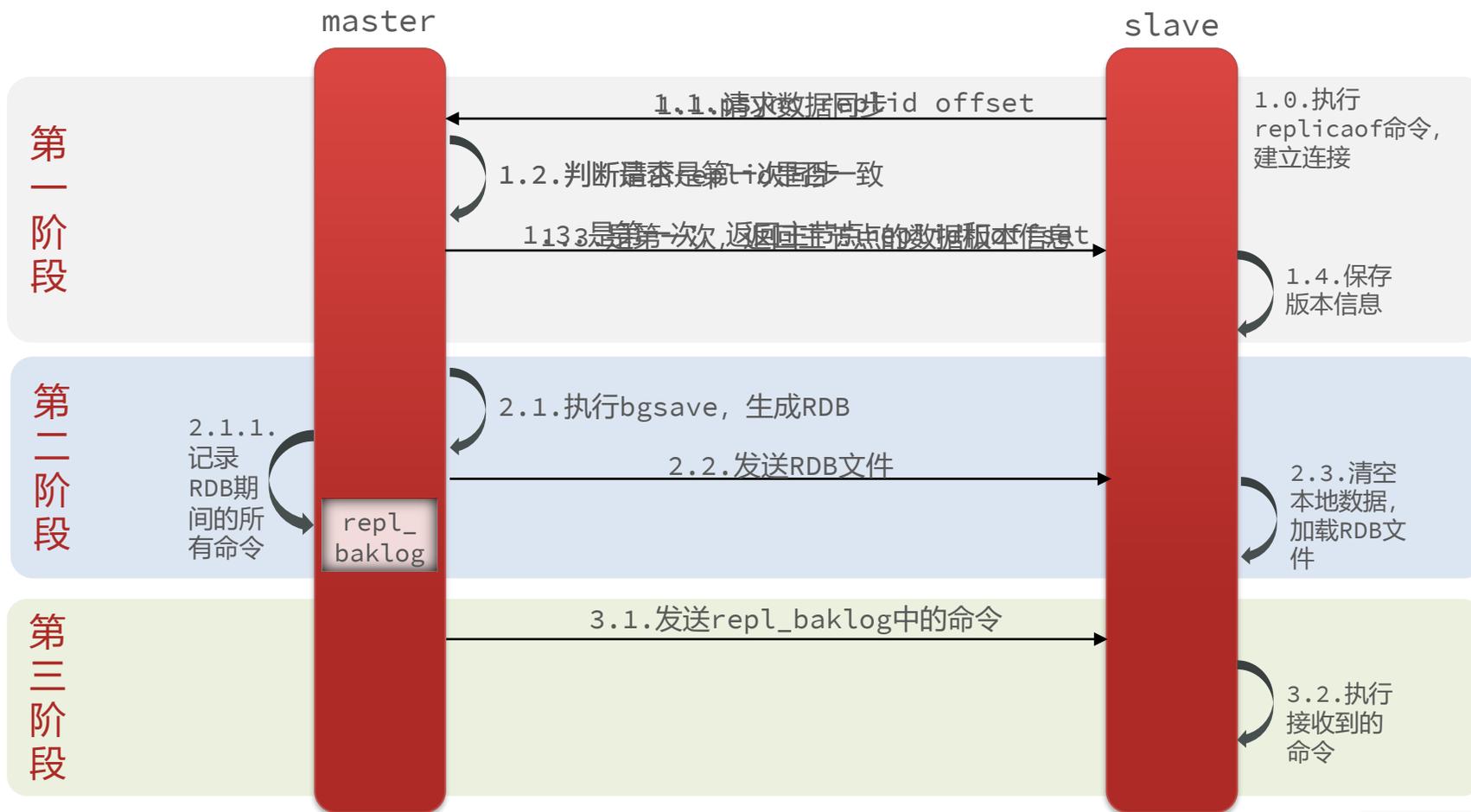
因此slave做数据同步，必须向master声明自己的**replication id** 和**offset**，master才可以判断到底需要同步哪些数据

思考一下

master如何判断slave节点是不是第一次来做数据同步？

数据同步原理

主从第一次同步是**全量同步**





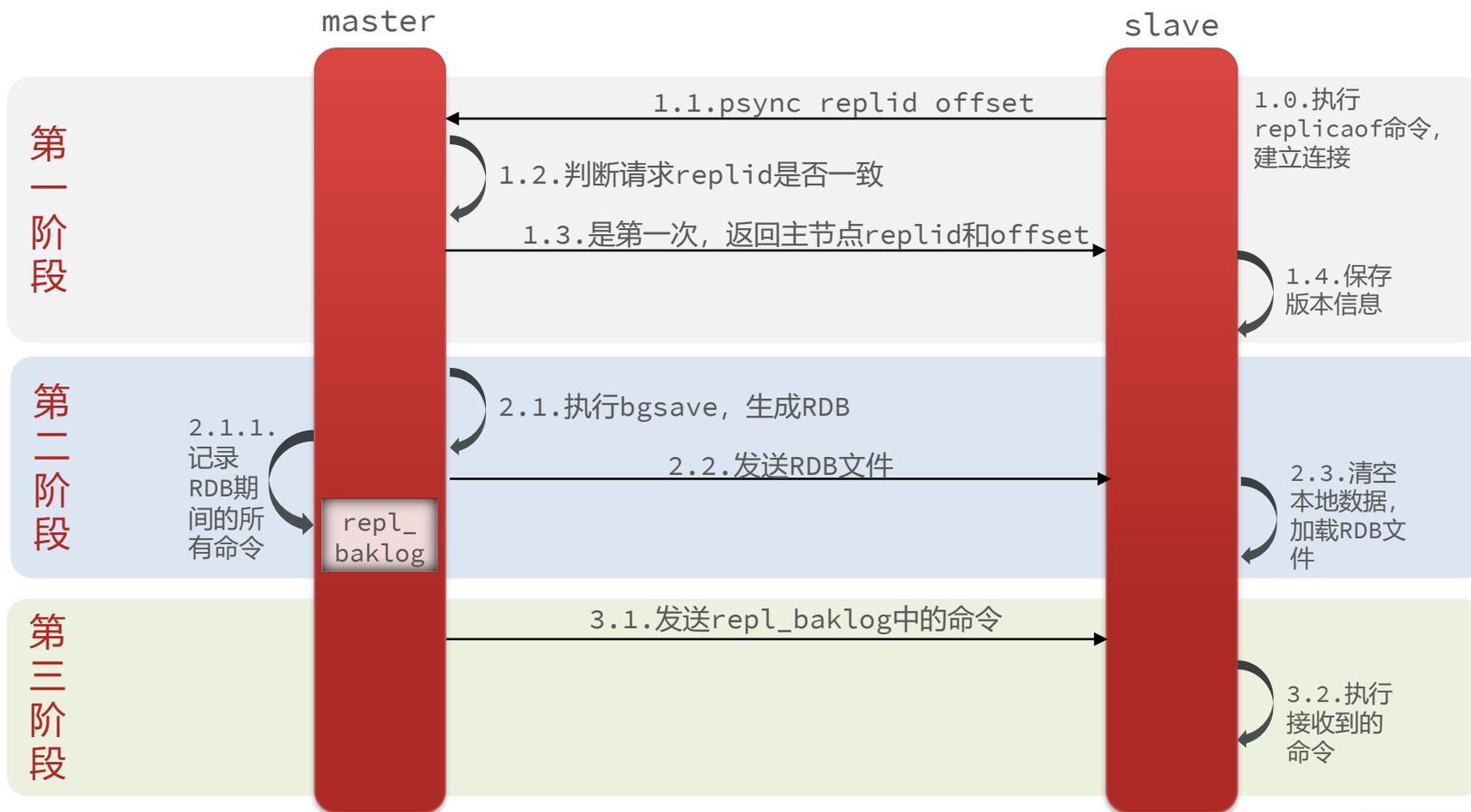
总结

简述全量同步的流程?

- slave节点请求增量同步
- master节点判断replid, 发现不一致, 拒绝增量同步
- master将完整内存数据生成RDB, 发送RDB到slave
- slave清空本地数据, 加载master的RDB
- master将RDB期间的命令记录在repl_baklog, 并持续将log中的命令发送给slave
- slave执行接收到的命令, 保持与master之间的同步

数据同步原理

主从第一次同步是**全量同步**



数据同步原理

主从第一次同步是**全量同步**，但如果slave重启后同步，则执行**增量同步**



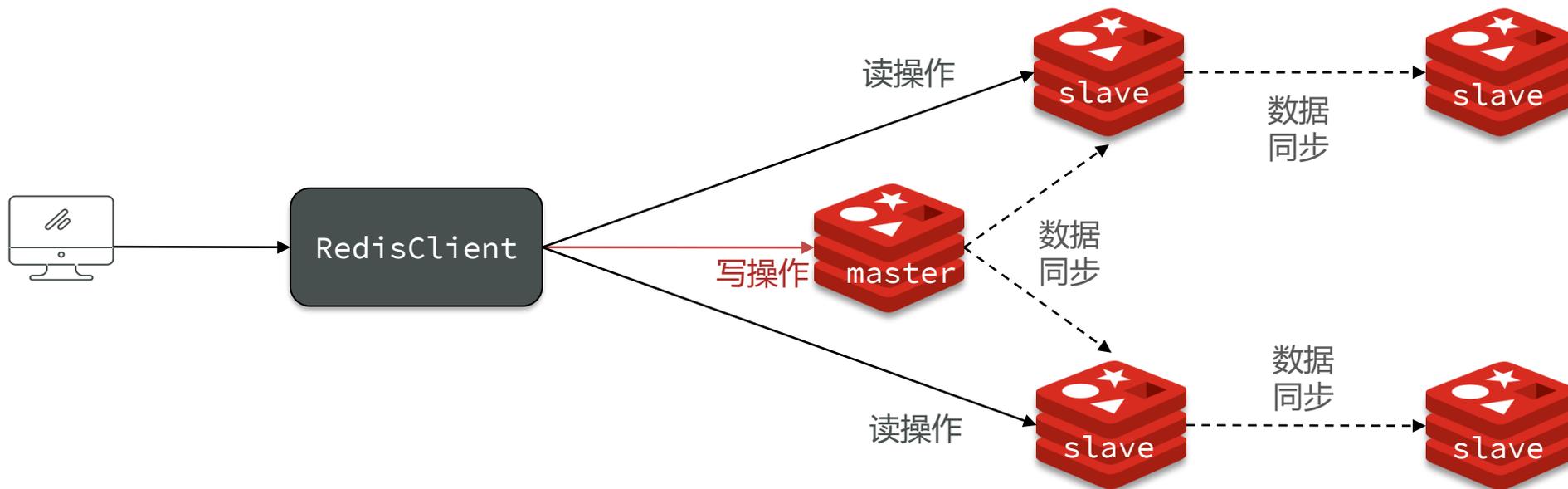
注意

repl_baklog大小有上限，写满后会覆盖最早的数据。如果slave断开时间过久，导致尚未备份的数据被覆盖，则无法基于log做增量同步，只能再次全量同步。

数据同步原理

可以从以下几个方面来优化Redis主从就集群:

- 在master中配置repl-diskless-sync yes启用无磁盘复制，避免全量同步时的磁盘IO。
- Redis单节点上的内存占用不要太大，减少RDB导致的过多磁盘IO
- 适当提高repl_baklog的大小，发现slave宕机时尽快实现故障恢复，尽可能避免全量同步
- 限制一个master上的slave节点数量，如果实在是太多slave，则可以采用主-从-从链式结构，减少master压力





总结

简述全量同步和增量同步区别?

- 全量同步: master将完整内存数据生成RDB, 发送RDB到slave。后续命令则记录在repl_baklog, 逐个发送给slave。
- 增量同步: slave提交自己的offset到master, master获取repl_baklog中从offset之后的命令给slave

什么时候执行全量同步?

- slave节点第一次连接master节点时
- slave节点断开时间太久, repl_baklog中的offset已经被覆盖时

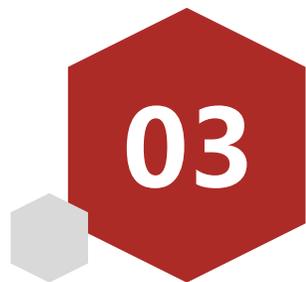
什么时候执行增量同步?

- slave节点断开又恢复, 并且在repl_baklog中能找到offset时



思考

- slave节点宕机恢复后可以找master节点同步数据，
那master节点宕机怎么办？



Redis哨兵

- 哨兵的作用和原理
- 搭建哨兵集群
- RedisTemplate的哨兵模式



目录

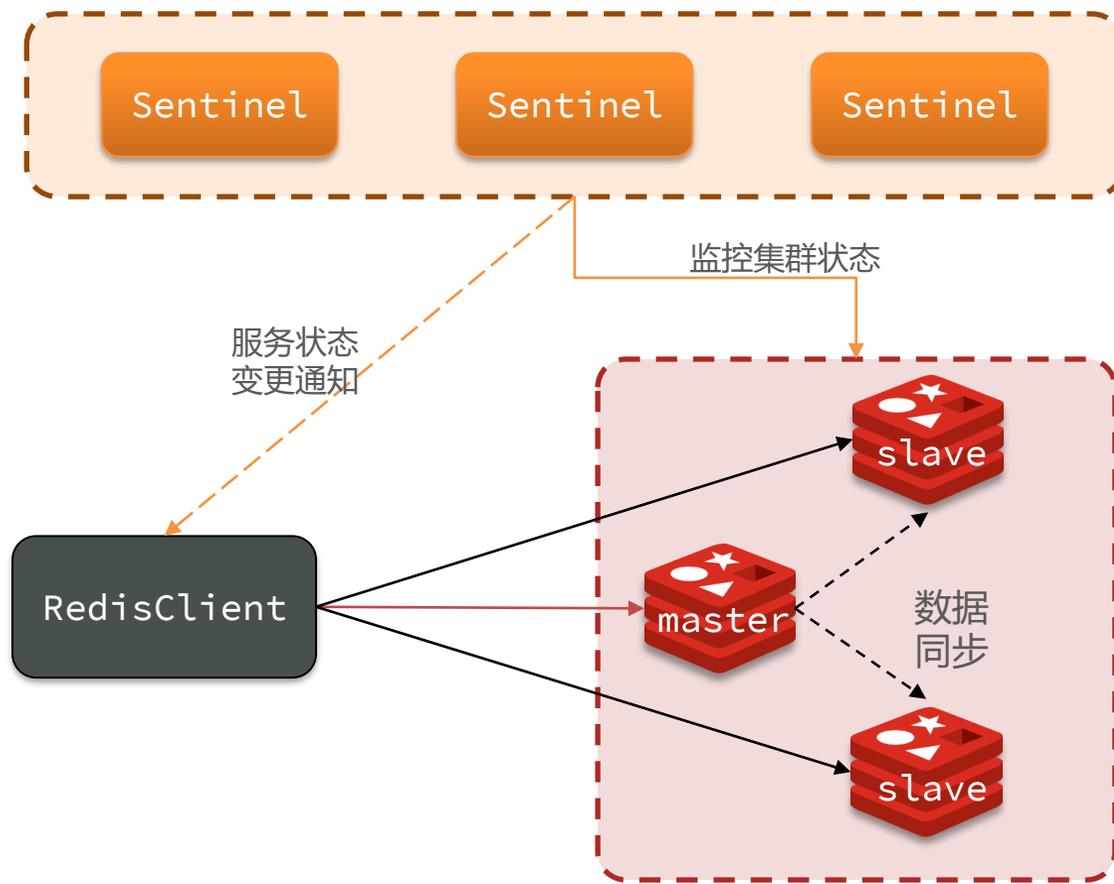
Contents

- ◆ 哨兵的作用和原理
- ◆ 搭建哨兵集群
- ◆ RedisTemplate的哨兵模式

哨兵的作用

Redis提供了哨兵（Sentinel）机制来实现主从集群的自动故障恢复。哨兵的结构和作用如下：

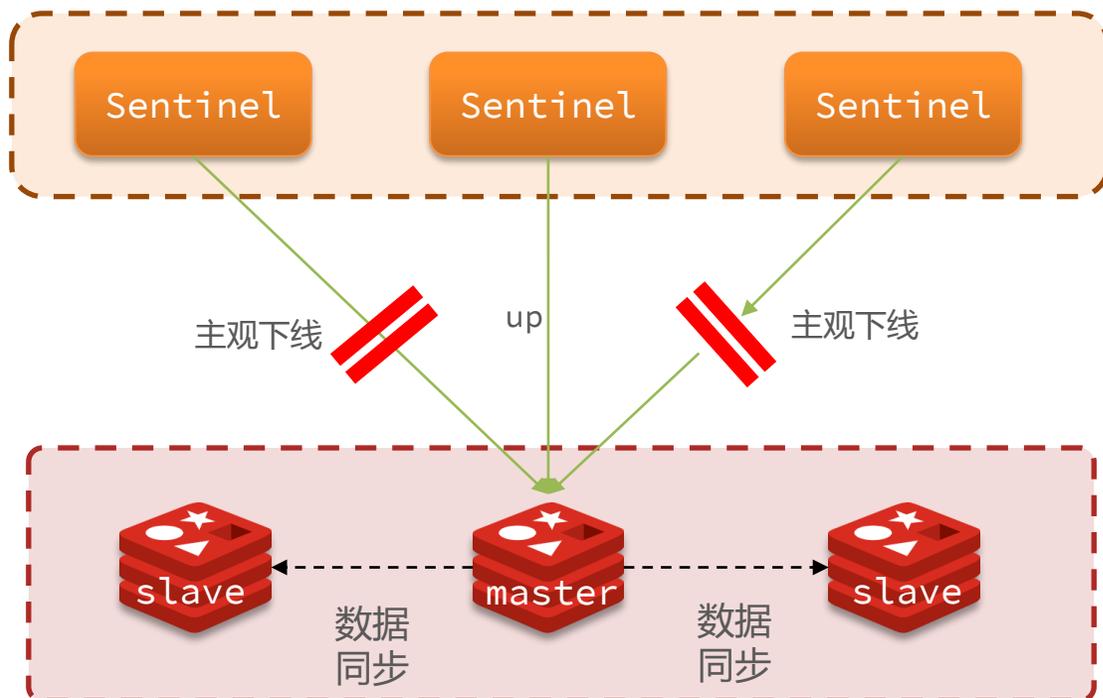
- **监控**：Sentinel 会不断检查您的master和slave 是否按预期工作
- **自动故障恢复**：如果master故障，Sentinel会将一个slave提升为master。当故障实例恢复后也以新的master为主
- **通知**：Sentinel充当Redis客户端的服务发现来源，当集群发生故障转移时，会将最新信息推送给Redis的客户端



服务状态监控

Sentinel基于心跳机制监测服务状态，每隔1秒向集群的每个实例发送ping命令：

- 主观下线：如果某sentinel节点发现某实例未在规定时间内响应，则认为该实例**主观下线**。
- 客观下线：若超过指定数量（quorum）的sentinel都认为该实例主观下线，则该实例**客观下线**。quorum值最好超过Sentinel实例数量的一半。



选举新的master

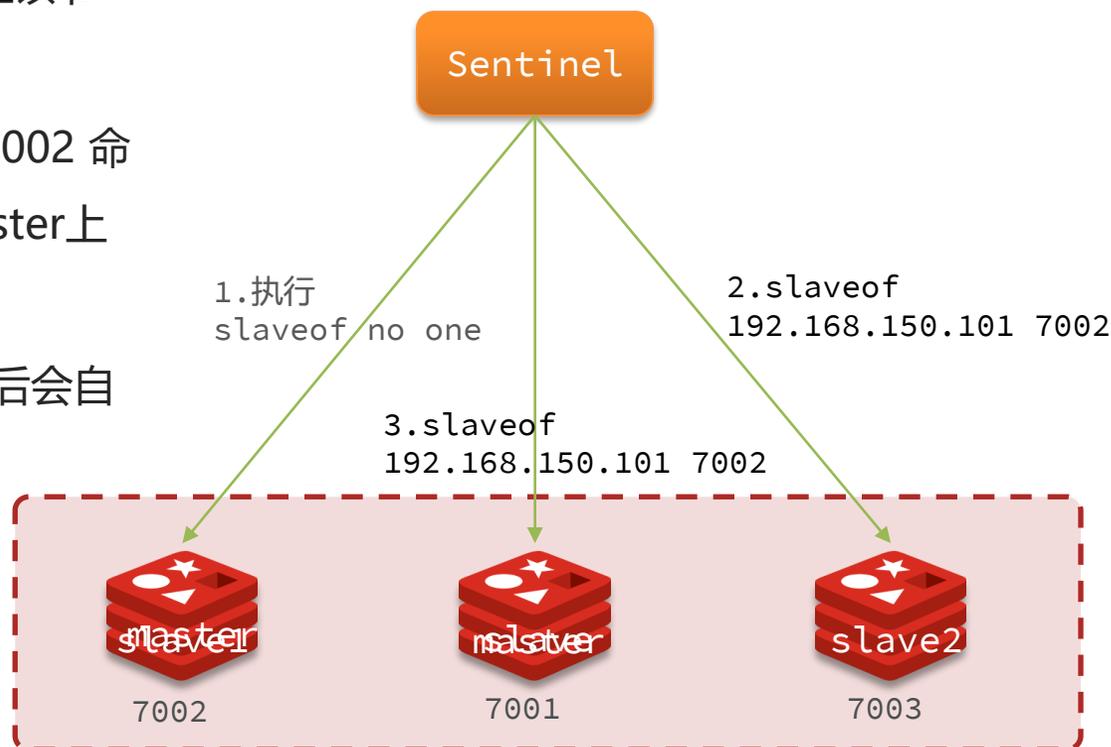
一旦发现master故障，sentinel需要在salve中选择一个作为新的master，选择依据是这样的：

- 首先会判断slave节点与master节点断开时间长短，如果超过指定值 ($\text{down-after-milliseconds} * 10$) 则会排除该slave节点
- 然后判断slave节点的slave-priority值，越小优先级越高，如果是0则永不参与选举
- 如果slave-prority一样，则判断slave节点的offset值，越大说明数据越新，优先级越高
- 最后是判断slave节点的运行id大小，越小优先级越高。

如何实现故障转移

当选中了其中一个slave为新的master后（例如slave1），故障的转移的步骤如下：

- sentinel给备选的slave1节点发送slaveof no one命令，让该节点成为master
- sentinel给所有其它slave发送slaveof 192.168.150.101 7002 命令，让这些slave成为新master的从节点，开始从新的master上同步数据。
- 最后，sentinel将故障节点标记为slave，当故障节点恢复后会自动成为新的master的slave节点





总结

Sentinel的三个作用是什么?

- 监控
- 故障转移
- 通知

Sentinel如何判断一个redis实例是否健康?

- 每隔1秒发送一次ping命令，如果超过一定时间没有相向则认为主观下线
- 如果大多数sentinel都认为实例主观下线，则判定服务下线

故障转移步骤有哪些?

- 首先选定一个slave作为新的master，执行slaveof no one
- 然后让所有节点都执行slaveof 新master
- 修改故障节点，执行slaveof 新master



目录

Contents

- ◆ 哨兵的作用和原理
- ◆ 搭建哨兵集群
- ◆ RedisTemplate的哨兵模式

搭建哨兵架构

具体搭建流程参考课前资料《Redis集群.md》：



Redis集群.md



目录

Contents

- ◆ 哨兵的作用和原理
- ◆ 搭建哨兵集群
- ◆ RedisTemplate的哨兵模式

RedisTemplate的哨兵模式

在Sentinel集群监管下的Redis主从集群，其节点会因为自动故障转移而发生变化，Redis的客户端必须感知这种变化，及时更新连接信息。Spring的RedisTemplate底层利用lettuce实现了节点的感知和自动切换。

首先，我们引入课前资料提供的Demo工程：



redis-demo

RedisTemplate的哨兵模式

1. 在pom文件中引入redis的starter依赖:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

2. 然后在配置文件application.yml中指定sentinel相关信息:

```
spring:
  redis:
    sentinel:
      master: mymaster # 指定master名称
      nodes: # 指定redis-sentinel集群信息
        - 192.168.150.101:27001
        - 192.168.150.101:27002
        - 192.168.150.101:27003
```

RedisTemplate的哨兵模式

3. 配置主从读写分离

```
@Bean
public LettuceClientConfigurationBuilderCustomizer configurationBuilderCustomizer(){
    return configBuilder -> configBuilder.readFrom(ReadFrom.REPLICA_PREFERRED);
}
```

这里的ReadFrom是配置Redis的读取策略，是一个枚举，包括下面选择：

- MASTER：从主节点读取
- MASTER_PREFERRED：优先从master节点读取，master不可用才读取replica
- REPLICA：从slave (replica) 节点读取
- REPLICA_PREFERRED：优先从slave (replica) 节点读取，所有的slave都不可用才读取master



Redis分片集群

- 搭建分片集群
- 散列插槽
- 集群伸缩
- 故障转移
- RedisTemplate访问分片集群



目录

Contents

- ◆ 搭建分片集群
- ◆ 散列插槽
- ◆ 集群伸缩
- ◆ 故障转移
- ◆ RedisTemplate访问分片集群

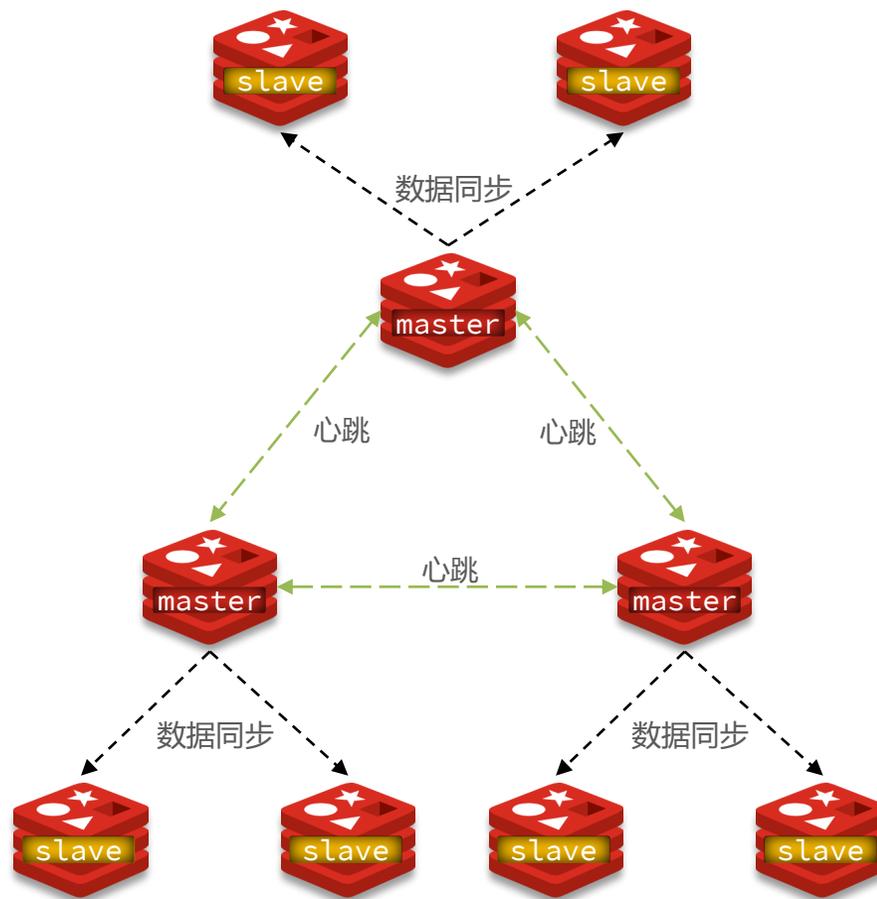
分片集群结构

主从和哨兵可以解决高可用、高并发读的问题。但是依然有两个问题没有解决：

- 海量数据存储问题
- 高并发写的问题

使用分片集群可以解决上述问题，分片集群特征：

- 集群中有多个master，每个master保存不同数据
- 每个master都可以有多个slave节点
- master之间通过ping监测彼此健康状态
- 客户端请求可以访问集群任意节点，最终都会被转发到正确节点



搭建分片集群

具体搭建流程参考课前资料《Redis集群.md》：



Redis集群.md



目录

Contents

- ◆ 搭建分片集群
- ◆ 散列插槽
- ◆ 集群伸缩
- ◆ 故障转移
- ◆ RedisTemplate访问分片集群

散列插槽

Redis会把每一个master节点映射到0~16383共16384个插槽 (hash slot) 上，查看集群信息时就能看到：

```
M: f5fc58defbebb957e47fb0d8327a09dc4f1678f5 192.168.150.101:7001
  slots:[0-5460] (5461 slots) master
M: afaaa70d6528fc72490e0f3f7b32731a12c12bb8 192.168.150.101:7002
  slots:[5461-10922] (5462 slots) master
M: 1c00e5f9e158b169f199f15884ab43bc433b1a06 192.168.150.101:7003
  slots:[10923-16383] (5461 slots) master
```

数据key不是与节点绑定，而是与插槽绑定。redis会根据key的有效部分计算插槽值，分两种情况：

- key中包含"{}"，且 "{}" 中至少包含1个字符， "{}" 中的部分是有效部分
- key中不包含 "{}" ，整个key都是有效部分

例如：key是num，那么就根据num计算，如果是{itcast}num，则根据itcast计算。计算方式是利用CRC16算法得到一个hash值，然后对16384取余，得到的结果就是slot值。

```
127.0.0.1:7001> set a 1
-> Redirected to slot [15495] located at 192.168.150.101:7003
OK
192.168.150.101:7003> get num
-> Redirected to slot [2765] located at 192.168.150.101:7001
"123"
```



总结

Redis如何判断某个key应该在哪个实例?

- 将16384个插槽分配到不同的实例
- 根据key的有效部分计算哈希值，对16384取余
- 余数作为插槽，寻找插槽所在实例即可

如何将同一类数据固定的保存在同一个Redis实例?

- 这一类数据使用相同的有效部分，例如key都以{typeId}为前缀



目录

Contents

- ◆ 搭建分片集群
- ◆ 散列插槽
- ◆ 集群伸缩
- ◆ 故障转移
- ◆ RedisTemplate访问分片集群

添加一个节点到集群

redis-cli --cluster提供了很多操作集群的命令，可以通过下面方式查看：

```
[root@localhost ~]# redis-cli --cluster help
Cluster Manager Commands:
  create      host1:port1 ... hostN:portN
              --cluster-replicas <arg>
  check      host:port
              --cluster-search-multiple-owners
  info       host:port
  fix        host:port
              --cluster-search-multiple-owners
              --cluster-fix-with-unreachable-masters
```

比如，添加节点的命令：

```
[root@localhost ~]# redis-cli --cluster help
Cluster Manager Commands:
  create      host1:port1 ... hostN:portN
              --cluster-replicas <arg>
  add-node   new_host:new_port existing_host:existing_port
              --cluster-slave
              --cluster-master-id <arg>
```

案例

向集群中添加一个新的master节点，并向其中存储 num = 10

需求：

- 启动一个新的redis实例，端口为7004
- 添加7004到之前的集群，并作为一个master节点
- 给7004节点分配插槽，使得num这个key可以存储到7004实例

练习

- 删除集群中的一个节点

需求:

- 删除7004这个实例



目录

Contents

- ◆ 搭建分片集群
- ◆ 散列插槽
- ◆ 集群伸缩
- ◆ 故障转移
- ◆ RedisTemplate访问分片集群

故障转移

当集群中有一个master宕机会发生什么呢?

1. 首先是该实例与其它实例失去连接
2. 然后是疑似宕机:

```
1fa6d68d590827c24c237b1c490b78e5c7fe2ca9 192.168.150.101:8003@18003 slave f5fc58defbebb957e47fb0d8327a09dc4f1678f5 0 1625207711535 8 connected
f5fc58defbebb957e47fb0d8327a09dc4f1678f5 192.168.150.101:7001@17001 myself,master - 0 1625207710000 8 connected 0-5460
afaaa70d6528fc72490e0f3f7b32731a12c12bb8 192.168.150.101:7002@17002 master, fail? - 1625207705198 1625207703000 10 disconnected 5461-10922
6ec60fb5afd950a465f05c8024bf8f75d809b014 192.168.150.101:8002@18002 slave 1c00e5f9e158b169f199f15884ab43bc433b1a06 0 1625207710000 3 connected
1c00e5f9e158b169f199f15884ab43bc433b1a06 192.168.150.101:7003@17003 master - 0 1625207711000 3 connected 10923-16383
7b6d5ffc9a985d614dc5aeb2ee3abac1adfd3e22 192.168.150.101:8001@18001 slave afaaa70d6528fc72490e0f3f7b32731a12c12bb8 0 1625207709420 10 connected
```

3. 最后是确定下线，自动提升一个slave为新的master:

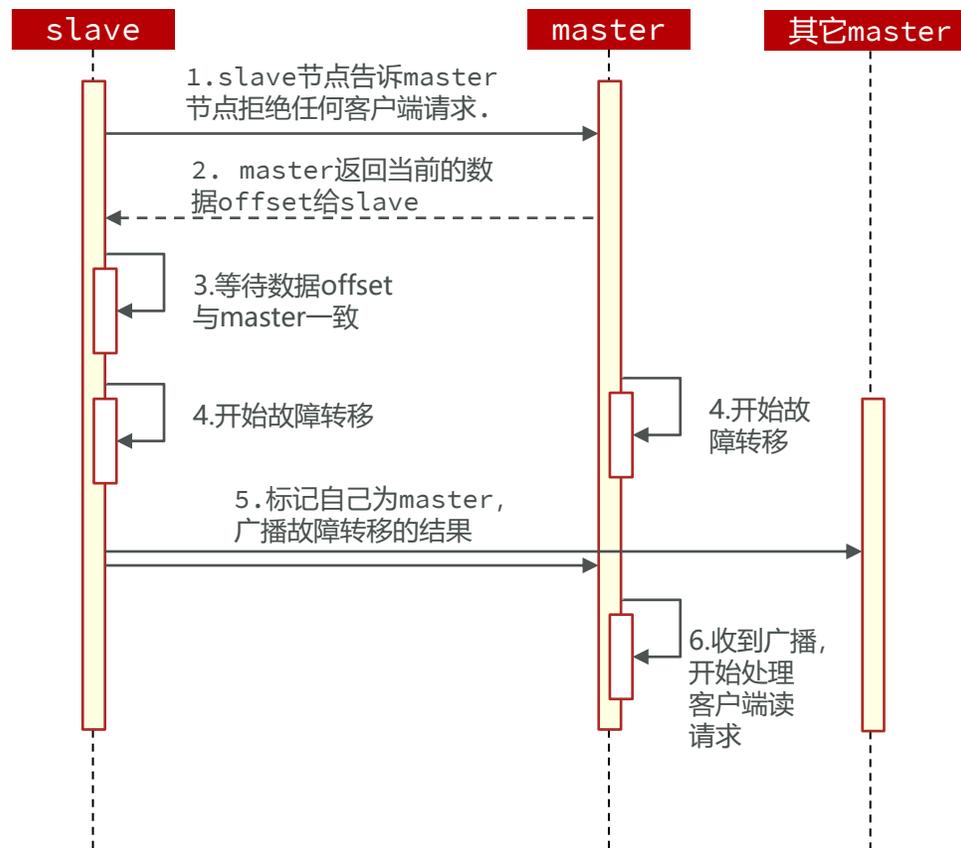
```
1fa6d68d590827c24c237b1c490b78e5c7fe2ca9 192.168.150.101:8003@18003 slave f5fc58defbebb957e47fb0d8327a09dc4f1678f5 0 1625208023157 8
f5fc58defbebb957e47fb0d8327a09dc4f1678f5 192.168.150.101:7001@17001 myself,master - 0 1625208022000 8 connected 0-5460
afaaa70d6528fc72490e0f3f7b32731a12c12bb8 192.168.150.101:7002@17002 master, fail - 1625207705198 1625207703000 10 disconnected
6ec60fb5afd950a465f05c8024bf8f75d809b014 192.168.150.101:8002@18002 slave 1c00e5f9e158b169f199f15884ab43bc433b1a06 0 1625208021035 3
1c00e5f9e158b169f199f15884ab43bc433b1a06 192.168.150.101:7003@17003 master - 0 1625208022084 3 connected 10923-16383
7b6d5ffc9a985d614dc5aeb2ee3abac1adfd3e22 192.168.150.101:8001@18001 master - 0 1625208023000 11 connected 5461-10922
```

数据迁移

利用cluster failover命令可以手动让集群中的某个master宕机，切换到执行cluster failover命令的这个slave节点，实现无感知的数据迁移。其流程如下：

手动的Failover支持三种不同模式：

- 缺省：默认的流程，如图1~6步
- force：省略了对offset的一致性校验
- takeover：直接执行第5步，忽略数据一致性、忽略master状态和其它master的意见



案例

在7002这个slave节点执行手动故障转移，重新夺回master地位

步骤如下：

1. 利用redis-cli连接7002这个节点
2. 执行cluster failover命令



目录

Contents

- ◆ 搭建分片集群
- ◆ 散列插槽
- ◆ 集群伸缩
- ◆ 故障转移
- ◆ RedisTemplate访问分片集群

RedisTemplate访问分片集群

RedisTemplate底层同样基于lettuce实现了分片集群的支持，而使用的步骤与哨兵模式基本一致：

1. 引入redis的starter依赖
2. 配置分片集群地址
3. 配置读写分离

与哨兵模式相比，其中只有分片集群的配置方式略有差异，如下：

```
spring:
  redis:
    cluster:
      nodes: # 指定分片集群的每一个节点信息
        - 192.168.150.101:7001
        - 192.168.150.101:7002
        - 192.168.150.101:7003
        - 192.168.150.101:8001
        - 192.168.150.101:8002
        - 192.168.150.101:8003
```



思考

Redis的分片集群、Elasticsearch的分片集群有哪些共同的特征呢?



传智教育旗下高端IT教育品牌