

day10-Redis面试篇

经过前几天的学习，大家已经掌握了微服务相关技术的实际应用，能够应对企业开发的要求了。不过大家都知道在IT领域往往都是面试造火箭，实际工作拧螺丝。

为了更好的应对面试，让大家能拿到更高的offer，我们接下来就讲讲“造火箭”的事情。

接下来的内容主要包括以下几方面：

Redis高级：

- Redis主从
- Redis哨兵
- Redis分片集群
- Redis数据结构
- Redis内存回收
- Redis缓存一致性

微服务高级：

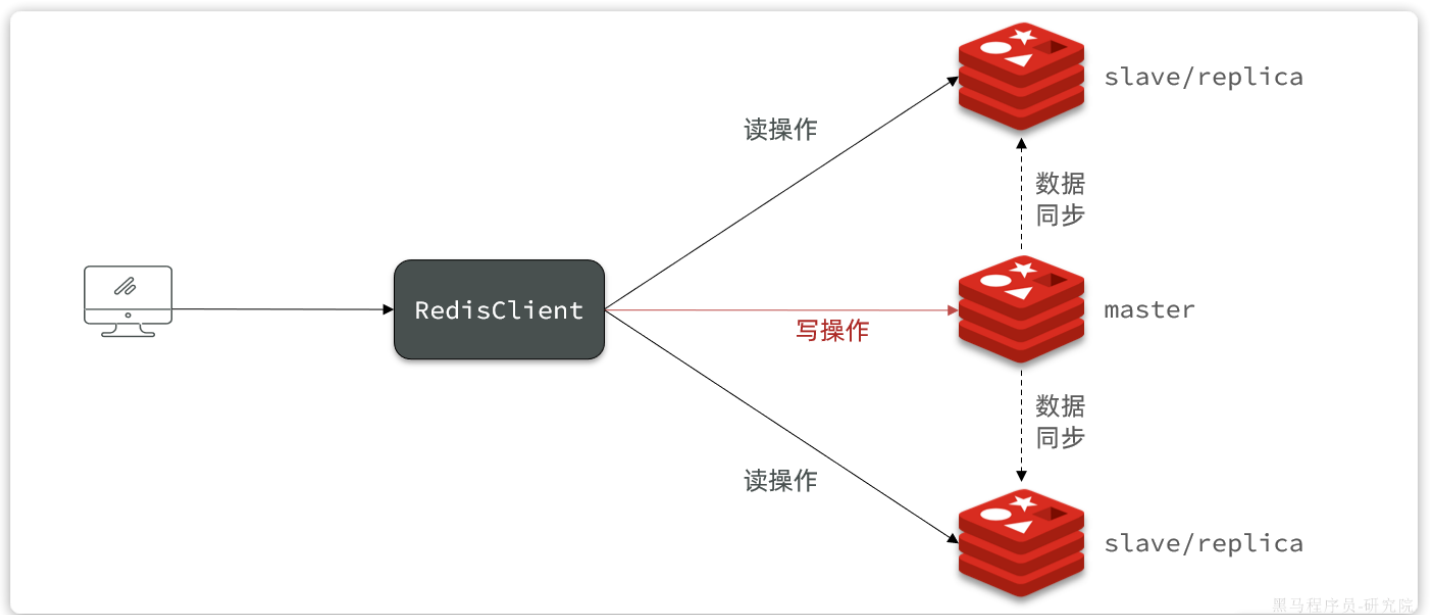
- Eureka和Nacos对比
- Ribbon和SpringCloudLoadBalancer
- Hystix和Sentinel
- 限流算法

1.Redis主从

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。

1.1.主从集群结构

下图就是一个简单的Redis主从集群结构：



如图所示，集群中有一个master节点、两个slave节点（现在叫replica）。当我们通过Redis的Java客户端访问主从集群时，应该做好路由：

- 如果是写操作，应该访问master节点，master会自动将数据同步给两个slave节点
- 如果是读操作，建议访问各个slave节点，从而分担并发压力

1.2.搭建主从集群



我们会在同一个虚拟机中利用3个Docker容器来搭建主从集群，容器信息如下：

容器名	角色	IP	映射端口
r1	master	192.168.150.101	7001
r2	slave	192.168.150.101	7002
r3	slave	192.168.150.101	7003

1.2.1.启动多个Redis实例

我们利用课前资料提供的docker-compose文件来构建主从集群：

新加卷 (D:) > 课程资料 > 服务框架 > day10-Redis高级 > 资料 >

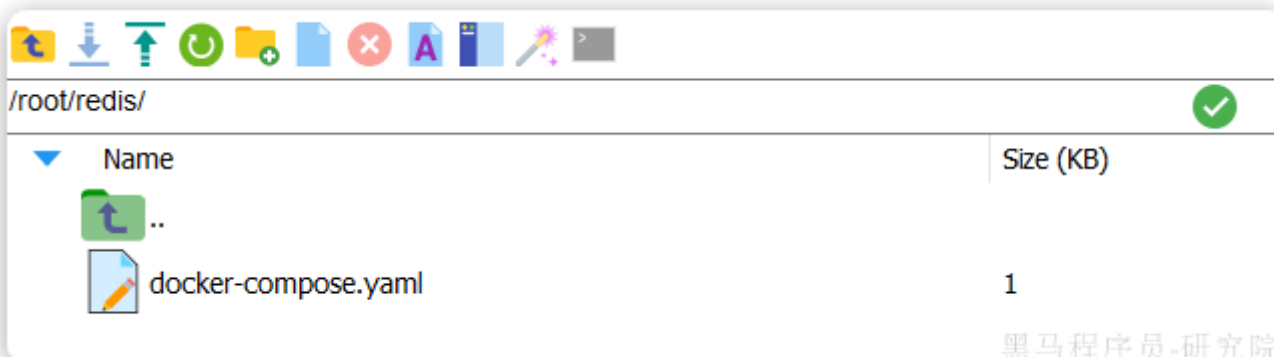
名称	类型	大小
 docker-compose.yaml	Yaml 源文件	1 KB
 sentinel.conf	CONF 文件	1 KB

黑马程序员-研究院

文件内容如下：

```
1 version: "3.2"
2
3 services:
4   r1:
5     image: redis
6     container_name: r1
7     network_mode: "host"
8     entrypoint: ["redis-server", "--port", "7001"]
9   r2:
10    image: redis
11    container_name: r2
12    network_mode: "host"
13    entrypoint: ["redis-server", "--port", "7002"]
14   r3:
15     image: redis
16     container_name: r3
17     network_mode: "host"
18     entrypoint: ["redis-server", "--port", "7003"]
```

将其上传至虚拟机的 `/root/redis` 目录下：



黑马程序员-研究院

执行命令，运行集群：

```
1 docker compose up -d
```

结果：

```
[root@heima redis]# docker-compose up -d
[+] Running 4/4
✔ Network redis-cluster Created 0.3s
✔ Container r3 Started 0.6s
✔ Container r1 Started 0.8s
✔ Container r2 Started 0.7s
```

查看docker容器，发现都正常启动了：

```
[root@heima redis]# dps
CONTAINER ID    NAMES    STATUS          PORTS
234a1587e7d8    r3       Up 2 seconds
e902eaacd1c8    r1       Up 2 seconds
395dce10cc66    r2       Up 2 seconds
```

由于采用的是host模式，我们看不到端口映射。不过能直接在宿主机通过ps命令查看到Redis进程：

```
[root@heima redis]# ps -ef | grep redis
root      50200   50146   1 17:54 ?        00:00:00 redis-server *:7001
root      50230   50160   0 17:54 ?        00:00:00 redis-server *:7003
root      50240   50173   0 17:54 ?        00:00:00 redis-server *:7002
root      50407    1850   0 17:54 pts/0    00:00:00 grep --color=auto redis
```

1.2.2.建立集群

虽然我们启动了3个Redis实例，但是它们并没有形成主从关系。我们需要通过命令来配置主从关系：

```
1 # Redis5.0以前
2 slaveof <masterip> <masterport>
3 # Redis5.0以后
4 replicaof <masterip> <masterport>
```

有临时和永久两种模式：

- 永久生效：在redis.conf文件中利用 `slaveof` 命令指定 `master` 节点

- 临时生效：直接利用redis-cli控制台输入 `slaveof` 命令，指定 `master` 节点

我们测试临时模式，首先连接 `r2`，让其以 `r1` 为master

```
1 # 连接r2
2 docker exec -it r2 redis-cli -p 7002
3 # 认r1主，也就是7001
4 slaveof 192.168.150.101 7001
```

然后连接 `r3`，让其以 `r1` 为master

```
1 # 连接r3
2 docker exec -it r3 redis-cli -p 7003
3 # 认r1主，也就是7001
4 slaveof 192.168.150.101 7001
```

然后连接 `r1`，查看集群状态：

```
1 # 连接r1
2 docker exec -it r1 redis-cli -p 7001
3 # 查看集群状态
4 info replication
```

结果如下：

```
1 127.0.0.1:7001> info replication
2 # Replication
3 role:master
4 connected_slaves:2
5 slave0:ip=192.168.150.101,port=7002,state=online,offset=140,lag=1
6 slave1:ip=192.168.150.101,port=7003,state=online,offset=140,lag=1
7 master_failover_state:no-failover
8 master_replid:16d90568498908b322178ca12078114e6c518b86
9 master_replid2:0000000000000000000000000000000000000000
10 master_repl_offset:140
11 second_repl_offset:-1
```

```
12 repl_backlog_active:1
13 repl_backlog_size:1048576
14 repl_backlog_first_byte_offset:1
15 repl_backlog_histlen:140
```

可以看到，当前节点 `r1:7001` 的角色是 `master`，有两个slave与其连接：

- `slave0`：port 是 `7002`，也就是 `r2` 节点
- `slave1`：port 是 `7003`，也就是 `r3` 节点

1.2.3.测试

依次在 `r1`、`r2`、`r3` 节点上执行下面命令：

```
1 set num 123
2
3 get num
```

你会发现，只有在 `r1` 这个节点上可以执行 `set` 命令（**写操作**），其它两个节点只能执行 `get` 命令（**读操作**）。也就是说读写操作已经分离了。

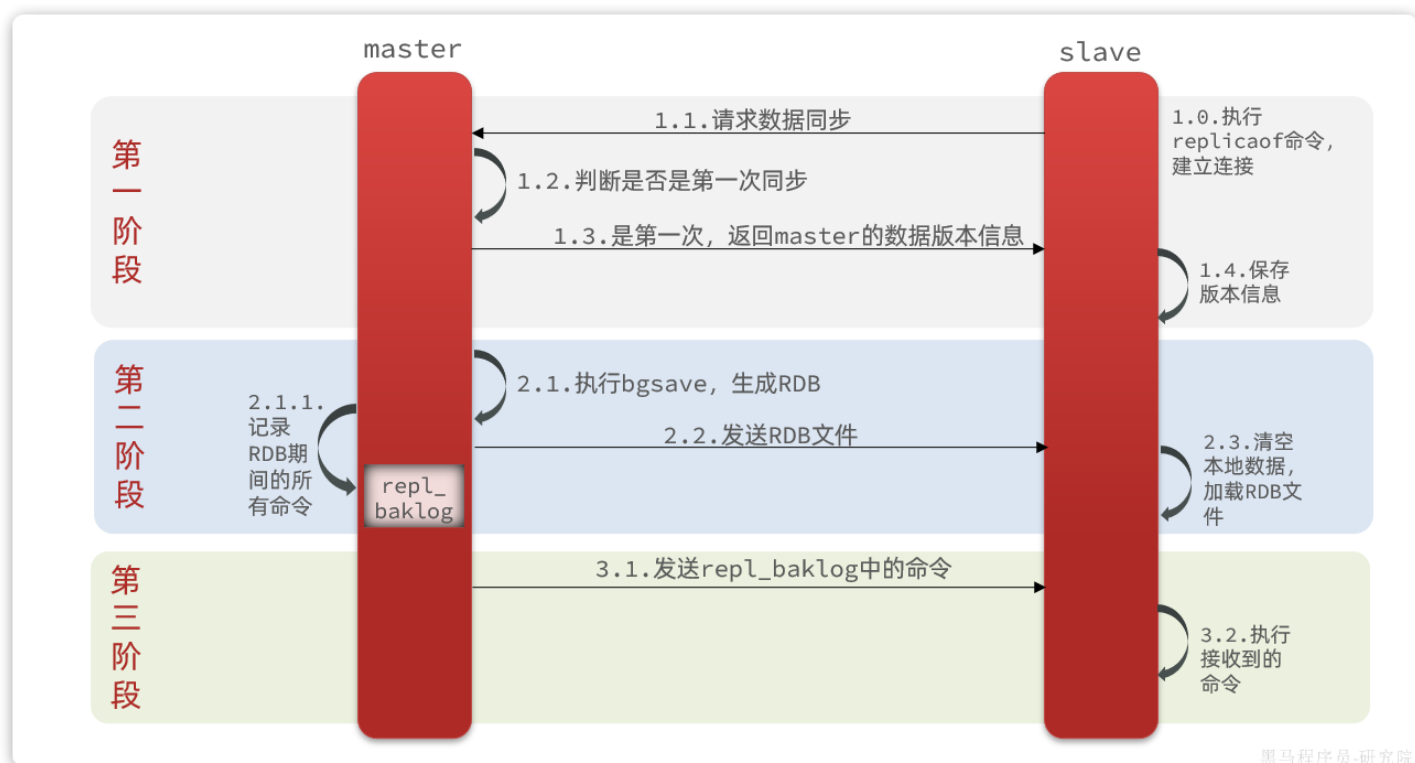
1.3.主从同步原理

在刚才的主从测试中，我们发现 `r1` 上写入Redis的数据，在 `r2` 和 `r3` 上也能看到，这说明主从之间确实完成了数据同步。

那么这个同步是如何完成的呢？

1.3.1.全量同步

主从第一次建立连接时，会执行**全量同步**，将master节点的所有数据都拷贝给slave节点，流程：



这里有一个问题，`master` 如何得知 `slave` 是否是第一次来同步呢？

有几个概念，可以作为判断依据：

- **Replication Id**：简称 `replid`，是数据集的标记，`replid`一致则是同一数据集。每个 `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。

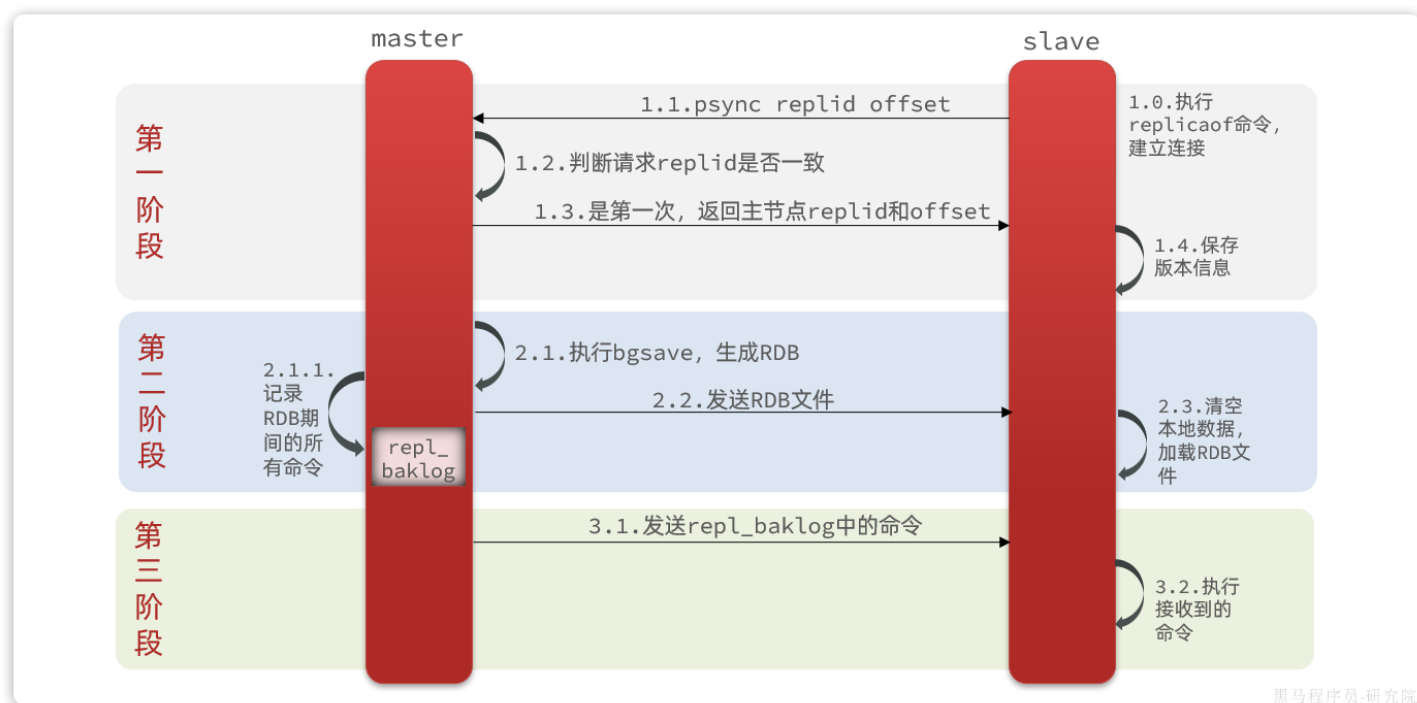
由于我们在执行 `slaveof` 命令之前，所有redis节点都是 `master`，有自己的 `replid` 和 `offset`。

当我们第一次执行 `slaveof` 命令，与 `master` 建立主从关系时，发送的 `replid` 和 `offset` 是自己的，与 `master` 肯定不一致。

`master` 判断发现 `slave` 发送来的 `replid` 与自己的不一致，说明这是一个全新的slave，就知道要做全量同步了。

master 会将自己的 replid 和 offset 都发送给这个 slave，slave 保存这些信息到本地。自此以后 slave 的 replid 就与 master 一致了。

因此，master判断一个节点是否是第一次同步的依据，就是看replid是否一致。流程如图：



完整流程描述：

- 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

来看下 r1 节点的运行日志：


```

1:M 13 Jul 2023 09:59:38.558 * Replica 192.168.150.101:7002 asks for synchronization
1:M 13 Jul 2023 09:59:38.559 * Partial resynchronization not accepted: Replication ID mismatch (Replica asked for '9989c4df3aacb1e7575becdbf57ac8b799daefb1', my replication IDs are '201425741cd10b48031bdce22ddbfc713b36408f' and '0000000000000000000000000000000000000000000000000000000000000000')
1:M 13 Jul 2023 09:59:38.559 * Replication backlog created, my new replication IDs are '16d90568498908b322178ca12078114e6c518b86' and '0000000000000000000000000000000000000000000000000000000000000000'
1:M 13 Jul 2023 09:59:38.559 * Starting BGSAVE for SYNC with target: disk
1:M 13 Jul 2023 09:59:38.559 * Background saving started by pid 11
11:C 13 Jul 2023 09:59:38.594 * DB saved on disk
11:C 13 Jul 2023 09:59:38.594 * RDB: 4 MB of memory used by copy-on-write
1:M 13 Jul 2023 09:59:38.609 * Background saving terminated with success
1:M 13 Jul 2023 09:59:38.609 * Synchronization with replica 192.168.150.101:7002 succeeded
1:M 13 Jul 2023 10:01:09.513 * Replica 192.168.150.101:7003 asks for synchronization

```

黑马程序员-研究院

再看下 r2 节点执行 replicaof 命令时的日志：

```

1:S 13 Jul 2023 09:59:38.558 * Connecting to MASTER 192.168.150.101:7001
1:S 13 Jul 2023 09:59:38.558 * MASTER <-> REPLICA sync started
1:S 13 Jul 2023 09:59:38.558 * REPLICAOF 192.168.150.101:7001 enabled (use r request from 'id=3 addr=127.0.0.1:55130 laddr=127.0.0.1:7002 fd=8 name=age=11 idle=0 flags=N db=0 sub=0 psub=0 multi=-1 qbuf=49 qbuf-free=40905 aargv-mem=26 obl=0 oll=0 omem=0 tot-mem=61490 events=r cmd=slaveof user=default redir=-1')
1:S 13 Jul 2023 09:59:38.558 * Non blocking connect for SYNC fired the event.
1:S 13 Jul 2023 09:59:38.559 * Master replied to PING, replication can continue...
1:S 13 Jul 2023 09:59:38.559 * Trying a partial resynchronization (request 9989c4df3aacb1e7575becdbf57ac8b799daefb1:1).
1:S 13 Jul 2023 09:59:38.559 * Full resync from master: 16d90568498908b322178ca12078114e6c518b86:0
1:S 13 Jul 2023 09:59:38.593 * Discarding previously cached master state.
1:S 13 Jul 2023 09:59:38.593 * MASTER <-> REPLICA sync: receiving 175 bytes from master to disk
1:S 13 Jul 2023 09:59:38.610 * MASTER <-> REPLICA sync: Flushing old data
1:S 13 Jul 2023 09:59:38.610 * MASTER <-> REPLICA sync: Loading DB in memory
1:S 13 Jul 2023 09:59:38.610 * Loading RDB produced by version 6.2.6
1:S 13 Jul 2023 09:59:38.610 * RDB age 0 seconds
1:S 13 Jul 2023 09:59:38.610 * RDB memory usage when created 1.83 Mb
1:S 13 Jul 2023 09:59:38.610 # Done loading RDB, keys loaded: 0, keys expired: 0.
1:S 13 Jul 2023 09:59:38.610 * MASTER <-> REPLICA sync: Finished with success

```

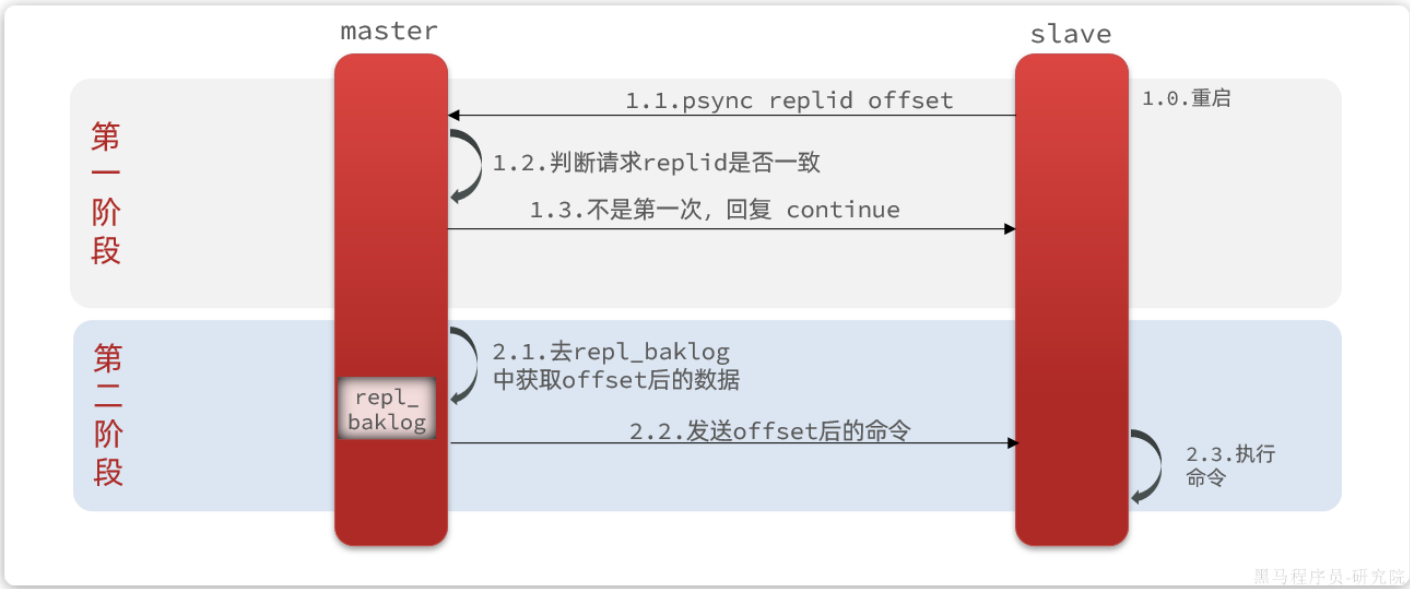
黑马程序员-研究院

与我们描述的完全一致。

1.3.2.增量同步

全量同步需要先做RDB，然后将RDB文件通过网络传输个slave，成本太高了。因此除了第一次做全量同步，其它大多数时候slave与master都是做**增量同步**。

什么是增量同步？就是只更新slave与master存在差异的部分数据。如图：



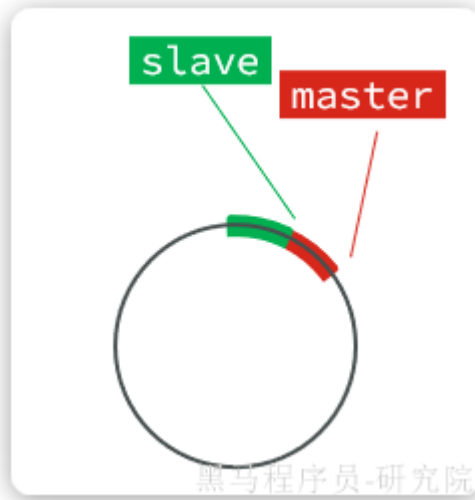
那么master怎么知道slave与自己的数据差异在哪里呢？

1.3.3.repl_baklog原理

master怎么知道slave与自己的数据差异在哪里呢？

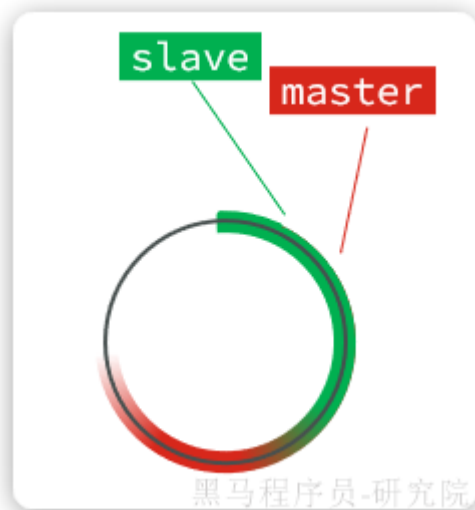
这就要说到全量同步时的 `repl_baklog` 文件了。这个文件是一个固定大小的数组，只不过数组是环形，也就是说**角标到达数组末尾后，会再次从0开始读写**，这样数组头部的数据就会被覆盖。

`repl_baklog` 中会记录Redis处理过的命令及 `offset`，包括master当前的 `offset`，和slave已经拷贝到的 `offset`：

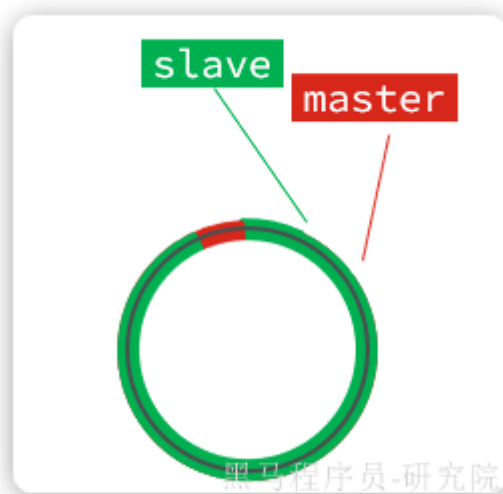


slave与master的offset之间的差异，就是slave需要增量拷贝的数据了。

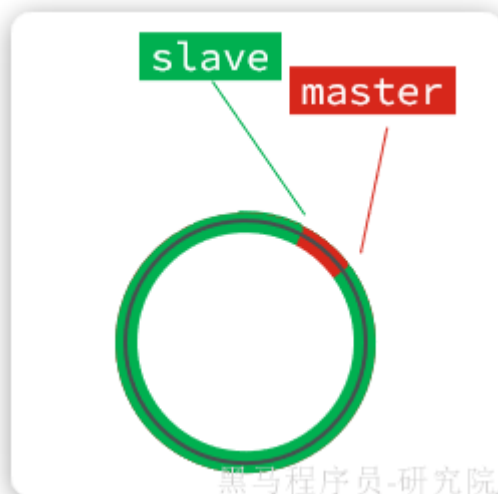
随着不断有数据写入，master的offset逐渐变大，slave也不断的拷贝，追赶master的offset：



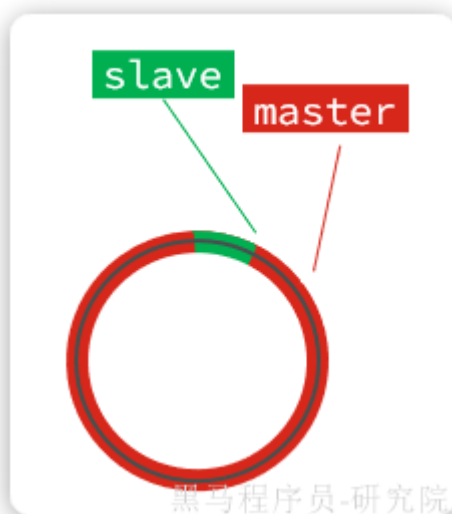
直到数组被填满：



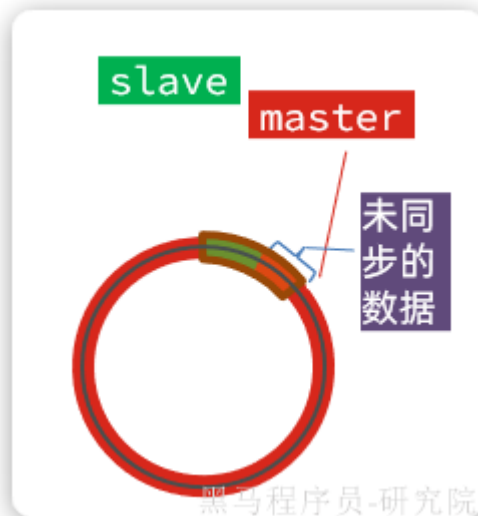
此时，如果有新的数据写入，就会覆盖数组中的旧数据。不过，旧的数据只要是绿色的，说明是已经被同步到slave的数据，即便被覆盖了也没什么影响。因为未同步的仅仅是红色部分：



但是，如果slave出现网络阻塞，导致master的 `offset` 远远超过了slave的 `offset`：



如果master继续写入新数据，master的 `offset` 就会覆盖 `repl_baklog` 中旧的数据，直到将slave现在的 `offset` 也覆盖：



棕色框中的红色部分，就是尚未同步，但是却已经被覆盖的数据。此时如果slave恢复，需要同步，却发现自己的 `offset` 都没有了，无法完成增量同步了。只能做**全量同步**。

`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 `repl_baklog` 做增量同步，只能再次全量同步。

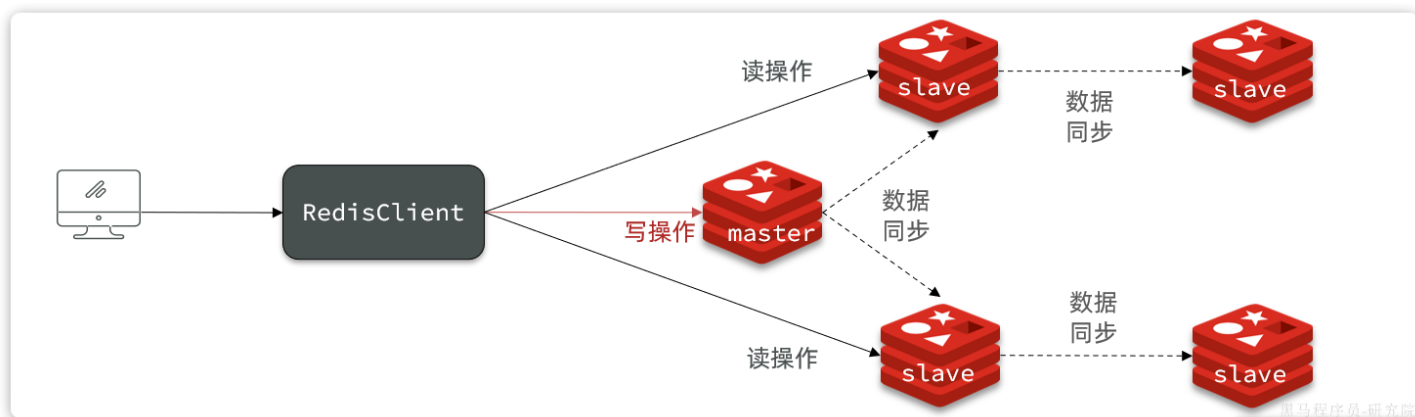
1.4.主从同步优化

主从同步可以保证主从数据的一致性，非常重要。

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

`主-从-从` 架构图：



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` 中找到offset时

2.Redis哨兵

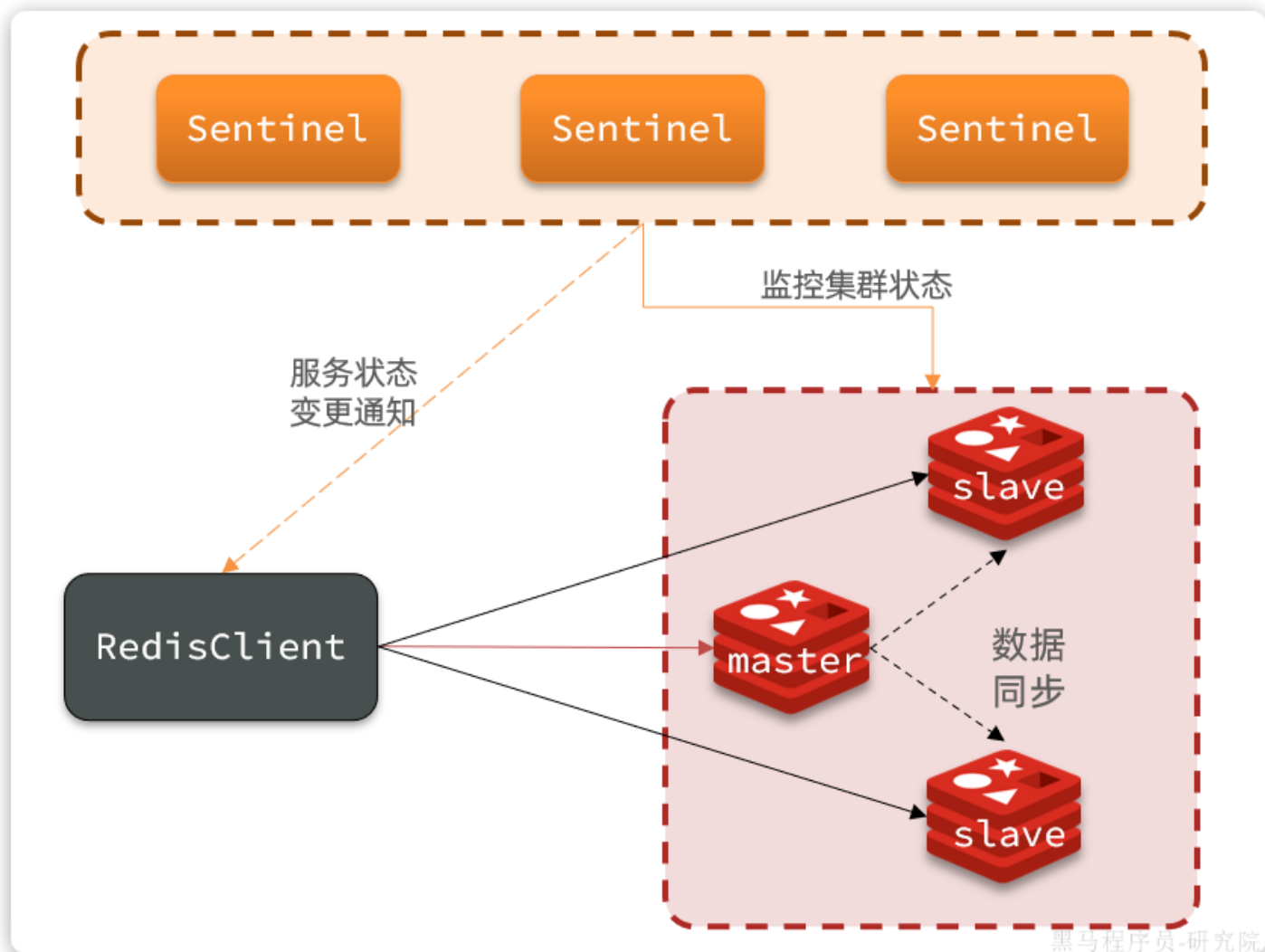
主从结构中master节点的作用非常重要，一旦故障就会导致集群不可用。那么有什么办法能保证主从集群的高可用性呢？

2.1.哨兵工作原理

Redis提供了 `哨兵` (`Sentinel`) 机制来监控主从集群监控状态，确保集群的高可用性。

2.1.1.哨兵作用

哨兵集群作用原理图：



哨兵的作用如下：

- **状态监控：** Sentinel 会不断检查您的 master 和 slave 是否按预期工作
- **故障恢复 (failover)：** 如果 master 故障， Sentinel 会将一个 slave 提升为 master。当故障实例恢复后会成为 slave
- **状态通知：** Sentinel 充当 Redis 客户端的服务发现来源，当集群发生 failover 时，会将最新集群信息推送给 Redis 的客户端

那么问题来了， Sentinel 怎么知道一个Redis节点是否宕机呢？

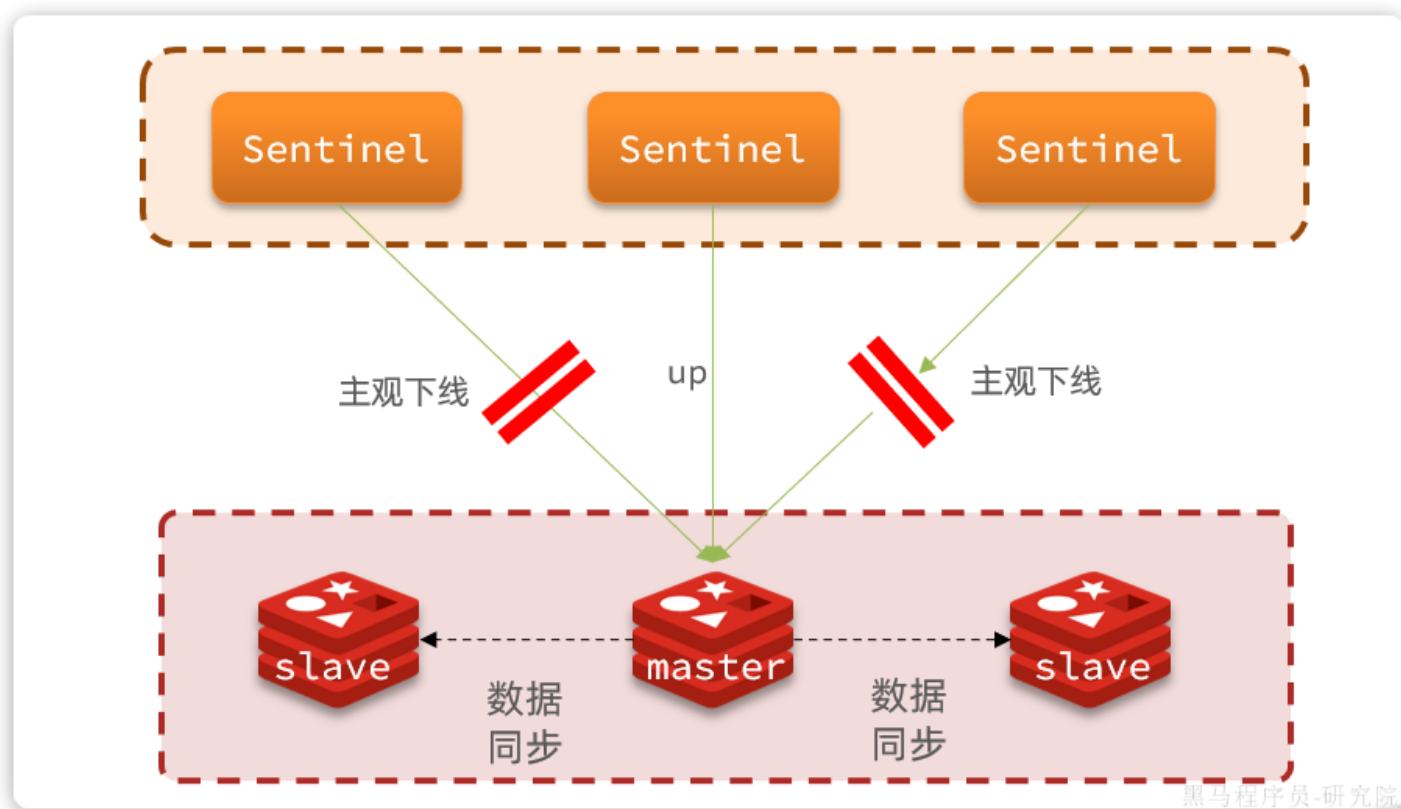
2.1.2.状态监控

Sentinel 基于心跳机制监测服务状态，每隔1秒向集群的每个节点发送ping命令，并通过实例的响应结果来做出判断：

- **主观下线 (sdown)：** 如果某sentinel节点发现某Redis节点未在规定时间内响应，则认为该节点主观下线。

- **客观下线(odown)**: 若超过指定数量（通过 `quorum` 设置）的sentinel都认为该节点主观下线，则该节点客观下线。quorum值最好超过Sentinel节点数量的一半，Sentinel节点数量至少3台。

如图:



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 `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（默认都是1）。
- 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 `run_id` 大小，越小优先级越高（通过 `info server` 可以查看 `run_id`）。

对应的官方文档如下：

 <https://redis.io/docs/management/sentinel/#replica-selection-and-priority>

High availability with Redis Sentinel

High availability for non-clustered Redis

问题来了，当选出一个新的master后，该如何实现身份切换呢？

大概分为两步：

- 在多个 `sentinel` 中选举一个 `leader`
- 由 `leader` 执行 `failover`

2.1.3.选举leader

首先，Sentinel集群要选出一个执行 `failover` 的Sentinel节点，可以成为 `leader`。要成为 `leader` 要满足两个条件：

- 最先获得超过半数的投票
- 获得的投票数不小于 `quorum` 值

而sentinel投票的原则有两条：

- 优先投票给目前得票最多的
- 如果目前没有任何节点的票，就投给自己

比如有3个sentinel节点，`s1`、`s2`、`s3`，假如 `s2` 先投票：

- 此时发现没有任何人在投票，那就投给自己。`s2` 得1票
- 接着 `s1` 和 `s3` 开始投票，发现目前 `s2` 票最多，于是也投给 `s2`，`s2` 得3票
- `s2` 称为 `leader`，开始故障转移

不难看出，**谁先投票，谁就会称为leader**，那什么时候会触发投票呢？

答案是**第一个确认master客观下线的人会立刻发起投票，一定会成为leader**。

OK，`sentinel` 找到 `leader` 以后，该如何完成 `failover` 呢？

2.1.4.failover

我们举个例子，有一个集群，初始状态下7001为 `master`，7002和7003为 `slave`：

Sentinel



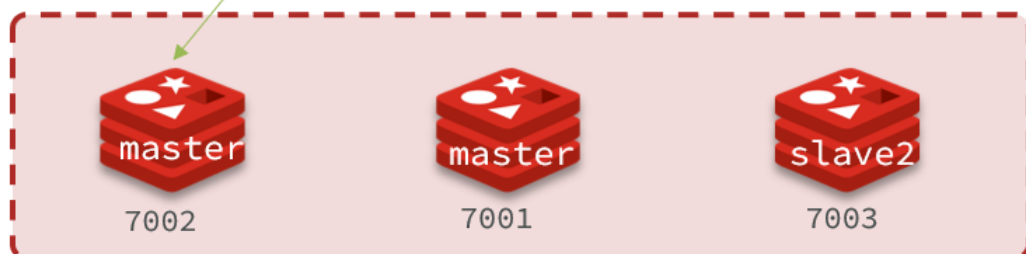
黑马程序员-研究院

假如master发生故障，slave1当选。则故障转移的流程如下：

- 1) `sentinel` 给备选的 `slave1` 节点发送 `slaveof no one` 命令，让该节点成为 `master`

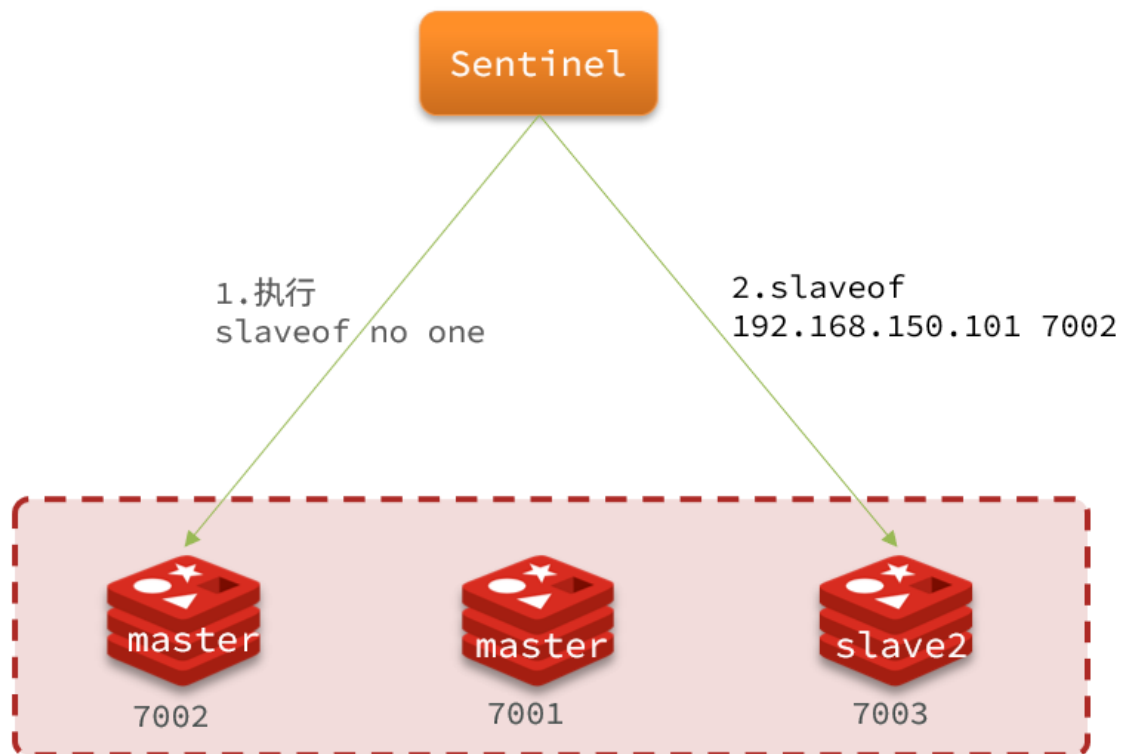
Sentinel

1. 执行
slaveof no one



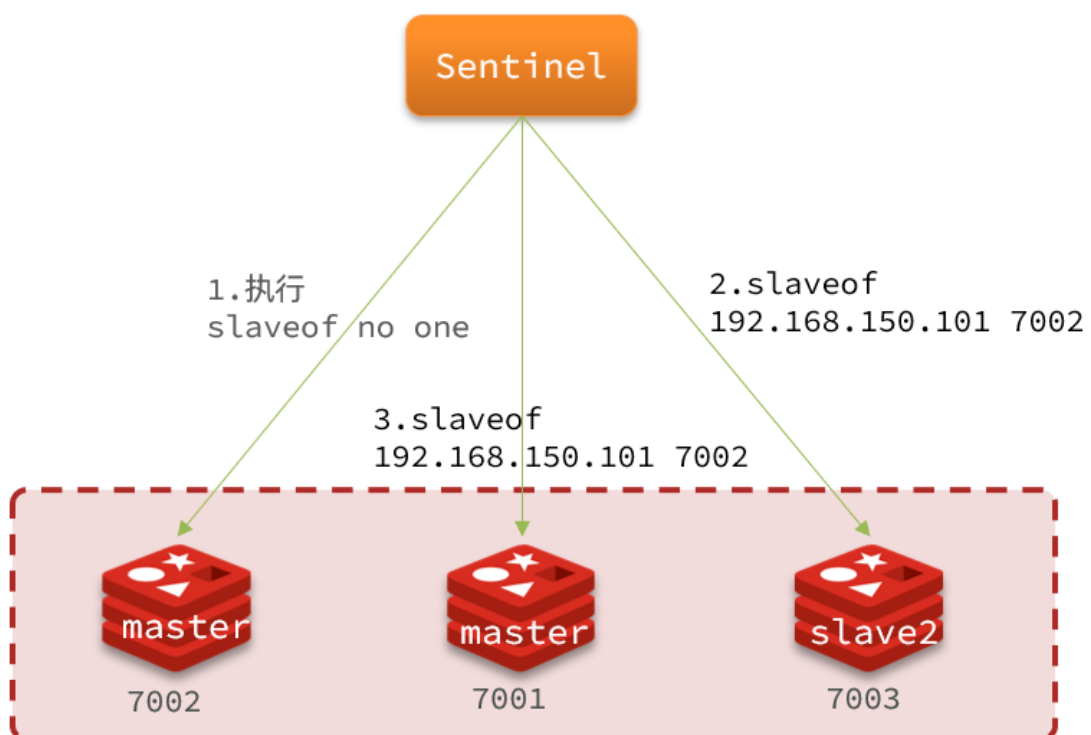
黑马程序员-研究院

- 2) `sentinel` 给所有其它 `slave` 发送 `slaveof 192.168.150.101 7002` 命令，让这些节点成为新 `master`，也就是 7002 的 `slave` 节点，开始从新的 `master` 上同步数据。



黑马程序员-研究院

3) 最后，当故障节点恢复后会接收到哨兵信号，执行 `slaveof 192.168.150.101 7002` 命令，成为 slave：



黑马程序员-研究院



2.2.搭建哨兵集群

首先，我们停掉之前的redis集群：

```
1 # 老版本DockerCompose
2 docker-compose down
3
4 # 新版本Docker
5 docker compose down
```

然后，我们找到课前资料提供的sentinel.conf文件：

新加卷 (D:) > 课程资料 > 服务框架 > day10-Redis高级 > 资料 >

名称	类型	大小
 docker-compose.yaml	Yaml 源文件	1 KB
 sentinel.conf	CONF 文件	1 KB

黑马程序员-研究院

其内容如下：

```
1 sentinel announce-ip "192.168.150.101"
2 sentinel monitor hmaster 192.168.150.101 7001 2
3 sentinel down-after-milliseconds hmaster 5000
4 sentinel failover-timeout hmaster 60000
```

说明：

- `sentinel announce-ip "192.168.150.101"`：声明当前sentinel的ip
- `sentinel monitor hmaster 192.168.150.101 7001 2`：指定集群的主节点信息
 - `hmaster`：主节点名称，自定义，任意写
 - `192.168.150.101 7001`：主节点的ip和端口
 - `2`：认定 `master` 下线时的 `quorum` 值
- `sentinel down-after-milliseconds hmaster 5000`：声明master节点超时多久后被标记下线
- `sentinel failover-timeout hmaster 60000`：在第一次故障转移失败后多久再次重试

我们在虚拟机的 `/root/redis` 目录下新建3个文件夹：`s1`、`s2`、`s3`：



将课前资料提供的 `sentinel.conf` 文件分别拷贝一份到3个文件夹中。

接着修改 `docker-compose.yaml` 文件，内容如下：

```
1 version: "3.2"
2
3 services:
4   r1:
5     image: redis
6     container_name: r1
7     network_mode: "host"
8     entrypoint: ["redis-server", "--port", "7001"]
9   r2:
10    image: redis
11    container_name: r2
12    network_mode: "host"
13    entrypoint: ["redis-server", "--port", "7002", "--slaveof",
14    "192.168.150.101", "7001"]
15  r3:
16    image: redis
17    container_name: r3
18    network_mode: "host"
19    entrypoint: ["redis-server", "--port", "7003", "--slaveof",
20    "192.168.150.101", "7001"]
21  s1:
22    image: redis
23    container_name: s1
24    volumes:
25      - /root/redis/s1:/etc/redis
```

```
24     network_mode: "host"
25     entrypoint: ["redis-sentinel", "/etc/redis/sentinel.conf", "--port",
    "27001"]
26     s2:
27     image: redis
28     container_name: s2
29     volumes:
30     - /root/redis/s2:/etc/redis
31     network_mode: "host"
32     entrypoint: ["redis-sentinel", "/etc/redis/sentinel.conf", "--port",
    "27002"]
33     s3:
34     image: redis
35     container_name: s3
36     volumes:
37     - /root/redis/s3:/etc/redis
38     network_mode: "host"
39     entrypoint: ["redis-sentinel", "/etc/redis/sentinel.conf", "--port",
    "27003"]
```

直接运行命令，启动集群：

```
1 docker-compose up -d
```

运行结果：

```
[root@heima redis]# docker-compose up -d
[+] Running 6/6
✔ Container s1   Started
✔ Container s2   Started
✔ Container r1   Started
✔ Container r2   Started
✔ Container s3   Started
✔ Container r3   Started
```

黑马程序员-研究院

我们以s1节点为例，查看其运行日志：

```
1 # Sentinel ID is 8e91bd24ea8e5eb2aee38f1cf796dcb26bb88acf
2 # +monitor master hmaster 192.168.150.101 7001 quorum 2
3 * +slave slave 192.168.150.101:7003 192.168.150.101 7003 @ hmaster
    192.168.150.101 7001
```

```
4 * +sentinel sentinel 5bafeb97fc16a82b431c339f67b015a51dad5e4f 192.168.150.101
27002 @ hmaster 192.168.150.101 7001
5 * +sentinel sentinel 56546568a2f7977da36abd3d2d7324c6c3f06b8d 192.168.150.101
27003 @ hmaster 192.168.150.101 7001
6 * +slave slave 192.168.150.101:7002 192.168.150.101 7002 @ hmaster
192.168.150.101 7001
```

可以看到 sentinel 已经联系到了 7001 这个节点，并且与其它几个哨兵也建立了链接。哨兵信息如下：

- 27001：Sentinel ID 是 8e91bd24ea8e5eb2aee38f1cf796dcb26bb88acf
- 27002：Sentinel ID 是 5bafeb97fc16a82b431c339f67b015a51dad5e4f
- 27003：Sentinel ID 是 56546568a2f7977da36abd3d2d7324c6c3f06b8d

2.3.演示failover

接下来，我们演示一下当主节点故障时，哨兵是如何完成集群故障恢复（failover）的。

我们连接 7001 这个 master 节点，然后通过命令让其休眠60秒，模拟宕机：

```
1 # 连接7001这个master节点，通过sleep模拟服务宕机，60秒后自动恢复
2 docker exec -it r1 redis-cli -p 7001 DEBUG sleep 60
```

稍微等待一段时间后，会发现sentinel节点触发了 failover：

```
1: X 07 Jul 2023 08:45:02.701 # +sdown master hmaster 192.168.150.101 7001
1: X 07 Jul 2023 08:45:02.846 # +new-epoch 1
1: X 07 Jul 2023 08:45:02.847 # +vote-for-leader 5bafeb97fc16a82b431c339f67b015a51dad5e4f 1
1: X 07 Jul 2023 08:45:03.866 # +odown master hmaster 192.168.150.101 7001 #quorum 3/2
1: X 07 Jul 2023 08:45:03.933 # Next failover delay: I will not start a failover before Fri Jul 7 08:47:03 2023
1: X 07 Jul 2023 08:45:03.933 # +config-update-from sentinel 5bafeb97fc16a82b431c339f67b015a51dad5e4f
192.168.150.101 27002 @ hmaster 192.168.150.101 7001
1: X 07 Jul 2023 08:45:03.933 # +switch-master hmaster 192.168.150.101 7001 192.168.150.101 7002
1: X 07 Jul 2023 08:45:03.933 # +slave slave 192.168.150.101:7003 192.168.150.101 7003 @ hmaster 192.168.150.101 7002
1: X 07 Jul 2023 08:45:03.933 # +slave slave 192.168.150.101:7001 192.168.150.101 7001 @ hmaster 192.168.150.101 7002
1: X 07 Jul 2023 08:45:08.959 # +sdown slave 192.168.150.101:7001 192.168.150.101 7001 @ hmaster 192.168.150.101 7002
1: X 07 Jul 2023 08:45:27.233 # -sdown slave 192.168.150.101:7001 192.168.150.101 7001 @ hmaster 192.168.150.101 7002

1: X 07 Jul 2023 08:45:02.846 # +new-epoch 1
1: X 07 Jul 2023 08:45:02.848 # +vote-for-leader 5bafeb97fc16a82b431c339f67b015a51dad5e4f 1
1: X 07 Jul 2023 08:45:03.024 # +sdown master hmaster 192.168.150.101 7001
1: X 07 Jul 2023 08:45:03.932 # +odown master hmaster 192.168.150.101 7001 #quorum 3/2
1: X 07 Jul 2023 08:45:03.932 # Next failover delay: I will not start a failover before Fri Jul 7 08:47:03 2023
1: X 07 Jul 2023 08:45:03.932 # +config-update-from sentinel 5bafeb97fc16a82b431c339f67b015a51dad5e4f
192.168.150.101 27002 @ hmaster 192.168.150.101 7001
1: X 07 Jul 2023 08:45:03.932 # +switch-master hmaster 192.168.150.101 7001 192.168.150.101 7002
1: X 07 Jul 2023 08:45:03.932 # +slave slave 192.168.150.101:7003 192.168.150.101 7003 @ hmaster 192.168.150.101 7002
1: X 07 Jul 2023 08:45:03.932 # +slave slave 192.168.150.101:7001 192.168.150.101 7001 @ hmaster 192.168.150.101 7002
1: X 07 Jul 2023 08:45:08.983 # +sdown slave 192.168.150.101:7001 192.168.150.101 7001 @ hmaster 192.168.150.101 7002
1: X 07 Jul 2023 08:45:27.264 # -sdown slave 192.168.150.101:7001 192.168.150.101 7001 @ hmaster 192.168.150.101 7002

1: X 07 Jul 2023 08:45:02.750 # +sdown master hmaster 192.168.150.101 7001
1: X 07 Jul 2023 08:45:02.842 # +odown master hmaster 192.168.150.101 7001 #quorum 2/2
1: X 07 Jul 2023 08:45:02.842 # +new-epoch 1
1: X 07 Jul 2023 08:45:02.842 # +try-failover master hmaster 192.168.150.101 7001
1: X 07 Jul 2023 08:45:02.844 # +vote-for-leader 5bafeb97fc16a82b431c339f67b015a51dad5e4f 1
1: X 07 Jul 2023 08:45:02.848 # 8e91bd24ea8e5eb2aee38f1cf796dcb26bb88acf voted for 5bafeb97fc16a82b431c339f67b015a51dad5e4f 1
1: X 07 Jul 2023 08:45:02.848 # 56546568a2f7977da36abd3d2d7324c6c3f06b8d voted for 5bafeb97fc16a82b431c339f67b015a51dad5e4f 1
1: X 07 Jul 2023 08:45:02.911 # +failover-state-select-slave master hmaster 192.168.150.101 7001
1: X 07 Jul 2023 08:45:02.994 # +selected-slave slave 192.168.150.101:7002 192.168.150.101 7002 @ hmaster 192.168.150.101 7001
1: X 07 Jul 2023 08:45:02.994 # +failover-state-send-slaveof-noone slave 192.168.150.101:7002 192.168.150.101 7001
1: X 07 Jul 2023 08:45:02.994 # +failover-state-reconf-slaves master hmaster 192.168.150.101 7001
1: X 07 Jul 2023 08:45:04.965 # +failover-end master hmaster 192.168.150.101 7001
1: X 07 Jul 2023 08:45:04.965 # +switch-master hmaster 192.168.150.101 7001 192.168.150.101 7002
1: X 07 Jul 2023 08:45:04.966 # +slave slave 192.168.150.101:7003 192.168.150.101 7003 @ hmaster 192.168.150.101 7002
1: X 07 Jul 2023 08:45:04.966 # +slave slave 192.168.150.101:7001 192.168.150.101 7001 @ hmaster 192.168.150.101 7002
1: X 07 Jul 2023 08:45:09.987 # +sdown slave 192.168.150.101:7001 192.168.150.101 7001 @ hmaster 192.168.150.101 7002
1: X 07 Jul 2023 08:45:27.255 # -sdown slave 192.168.150.101:7001 192.168.150.101 7001 @ hmaster 192.168.150.101 7002
```

2.4.总结

Sentinel的三个作用是什么？

- 集群监控
- 故障恢复
- 状态通知

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（`sdown`）
- 如果大多数sentinel都认为实例主观下线，则判定服务客观下线（`odown`）

故障转移步骤有哪些？

- 首先要在 `sentinel` 中选出一个 `leader`，由leader执行 `failover`
- 选定一个 `slave` 作为新的 `master`，执行 `slaveof noone`，切换到master模式
- 然后让所有节点都执行 `slaveof` 新master
- 修改故障节点配置，添加 `slaveof` 新master

sentinel选举leader的依据是什么？

- 票数超过sentinel节点数量1半
- 票数超过quorum数量
- 一般情况下最先发起failover的节点会当选

sentinel从slave中选取master的依据是什么？

- 首先会判断slave节点与master节点断开时间长短，如果超过 `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（默认都是1）。
- 如果 `slave-priority` 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 `run_id` 大小，越小优先级越高（通过 `info server` 可以查看 `run_id`）。

2.5.RedisTemplate连接哨兵集群（自学）

分为三步：

- 1) 引入依赖
- 2) 配置哨兵地址
- 3) 配置读写分离

2.5.1.引入依赖

就是SpringDataRedis的依赖：

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

2.5.2.配置哨兵地址

连接哨兵集群与传统单点模式不同，不再需要设置每一个redis的地址，而是直接指定哨兵地址：

```
1 spring:
2     redis:
3         sentinel:
4             master: hmaster # 集群名
5             nodes: # 哨兵地址列表
6                 - 192.168.150.101:27001
7                 - 192.168.150.101:27002
8                 - 192.168.150.101:27003
```

2.5.3.配置读写分离

最后，还要配置读写分离，让java客户端将写请求发送到master节点，读请求发送到slave节点。定义一个bean即可：

```
1 @Bean
```

```
2 public LettuceClientConfigurationBuilderCustomizer
   clientConfigurationBuilderCustomizer(){
3     return clientConfigurationBuilder ->
       clientConfigurationBuilder.readFrom(ReadFrom.REPLICA_PREFERRED);
4 }
```

这个bean中配置的就是读写策略，包括四种：

- MASTER：从主节点读取
- MASTER_PREFERRED：优先从 master 节点读取，master 不可用才读取 slave
- REPLICAS：从 slave 节点读取
- REPLICAS_PREFERRED：优先从 slave 节点读取，所有的 slave 都不可用才读取 master

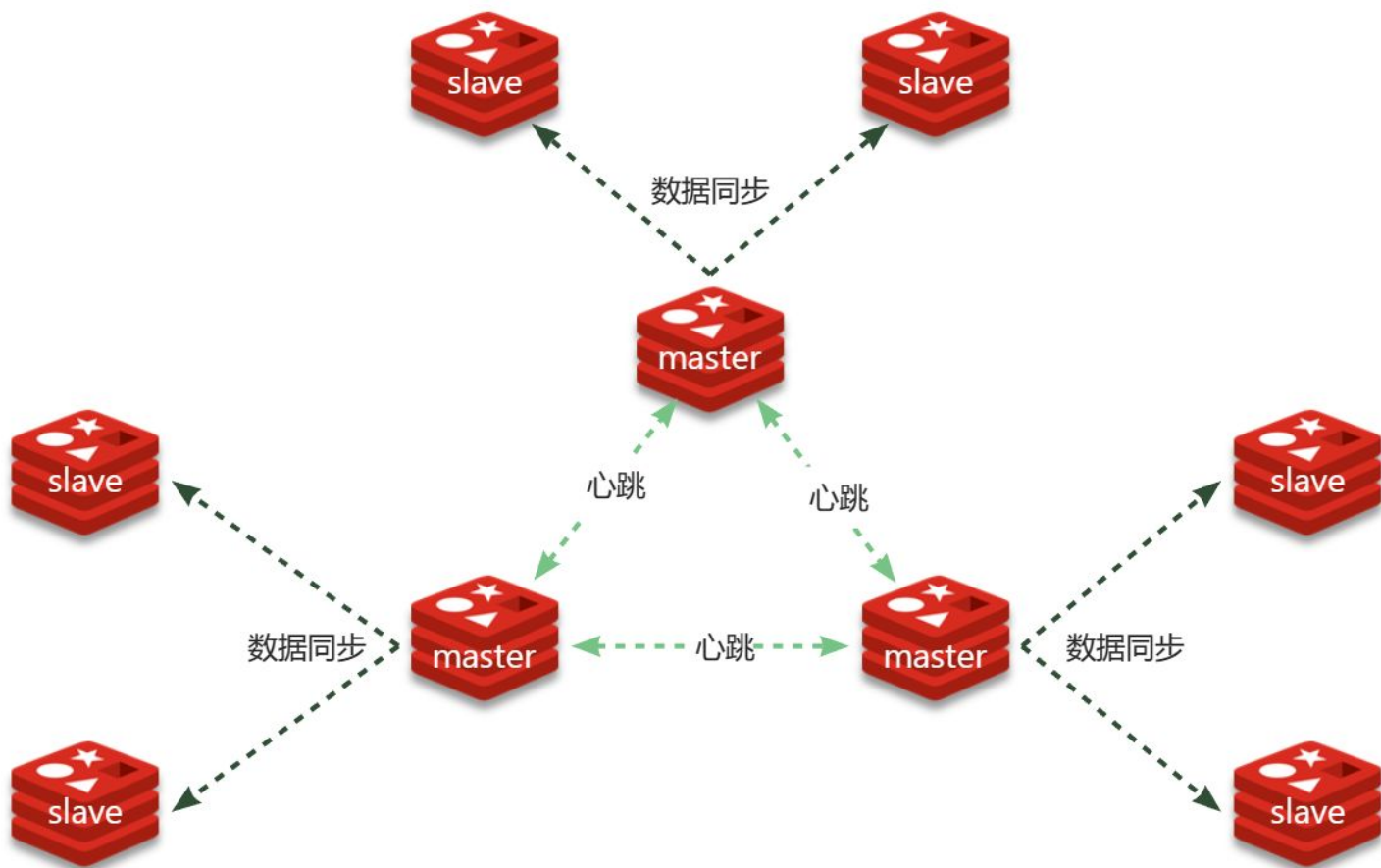
3.Redis分片集群

主从模式可以解决高可用、高并发读的问题。但依然有两个问题没有解决：

- 海量数据存储
- 高并发写

要解决这两个问题就需要用到分片集群了。分片的意思，就是把数据拆分存储到不同节点，这样整个集群的存储数据量就更大了。

Redis分片集群的结构如图：

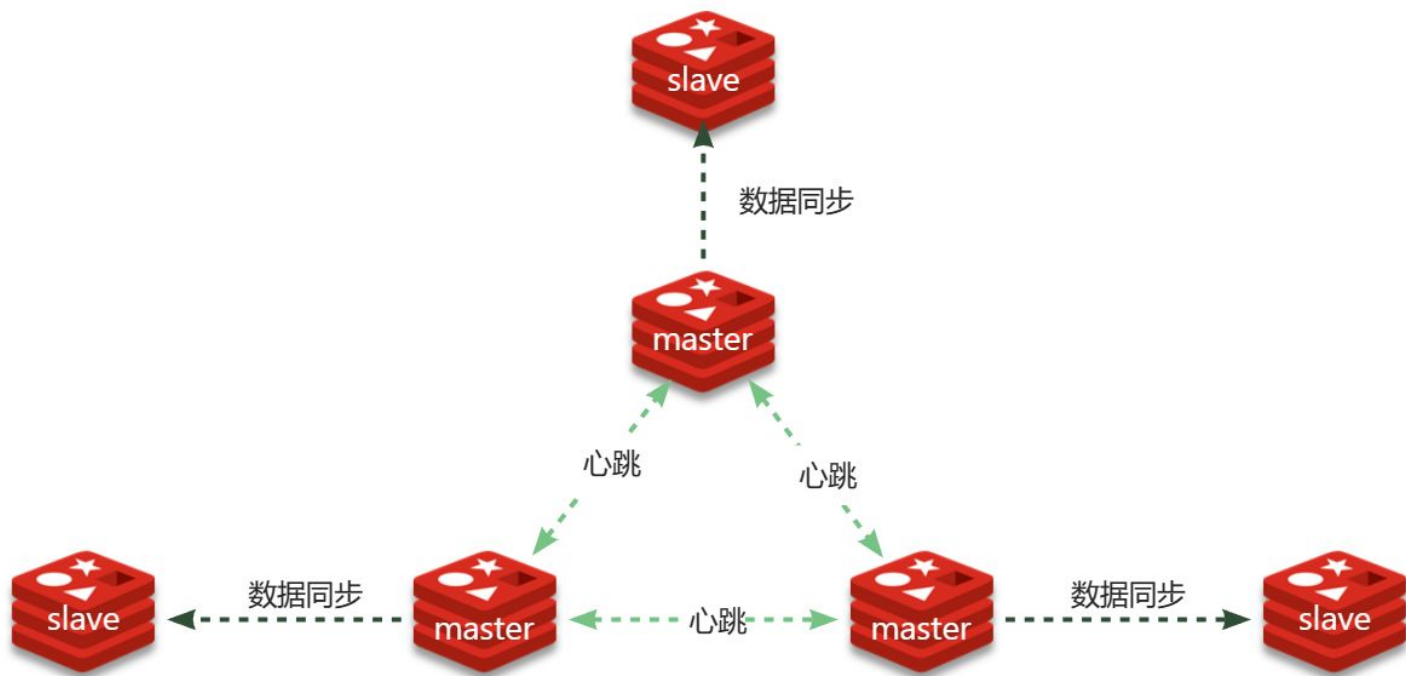


分片集群特征：

- 集群中有多个master，每个master保存不同分片数据，解决海量数据存储问题
- 每个master都可以有多个slave节点，确保高可用
- master之间通过ping监测彼此健康状态，类似哨兵作用
- 客户端请求可以访问集群任意节点，最终都会被转发到数据所在节点

3.1.搭建分片集群

Redis分片集群最少也需要3个master节点，由于我们的机器性能有限，我们只给每个master配置1个slave，形成最小的分片集群：



计划部署的节点信息如下：

容器名	角色	IP	映射端口
r1	master	192.168.150.101	7001
r2	master	192.168.150.101	7002
r3	master	192.168.150.101	7003
r4	slave	192.168.150.101	7004
r5	slave	192.168.150.101	7005
r6	slave	192.168.150.101	7006

3.1.1.集群配置

分片集群中的Redis节点必须开启集群模式，一般在配置文件中添加下面参数：

```
1 port 7000
2 cluster-enabled yes
3 cluster-config-file nodes.conf
4 cluster-node-timeout 5000
5 appendonly yes
```

其中有3个我们没见过的参数：

- `cluster-enabled`：是否开启集群模式
- `cluster-config-file`：集群模式的配置文件名称，无需手动创建，由集群自动维护
- `cluster-node-timeout`：集群中节点之间心跳超时时间

一般搭建部署集群肯定是给每个节点都配置上述参数，不过考虑到我们计划用 `docker-compose` 部署，因此可以直接在启动命令中指定参数，偷个懒。

在虚拟机的 `/root` 目录下新建一个 `redis-cluster` 目录，然后在其中新建一个 `docker-compose.yaml` 文件，内容如下：

```
1 version: "3.2"
2
3 services:
4   r1:
5     image: redis
6     container_name: r1
7     network_mode: "host"
8     entrypoint: ["redis-server", "--port", "7001", "--cluster-enabled", "yes",
9       "--cluster-config-file", "node.conf"]
10  r2:
11    image: redis
12    container_name: r2
13    network_mode: "host"
14    entrypoint: ["redis-server", "--port", "7002", "--cluster-enabled", "yes",
15      "--cluster-config-file", "node.conf"]
16  r3:
17    image: redis
18    container_name: r3
19    network_mode: "host"
20    entrypoint: ["redis-server", "--port", "7003", "--cluster-enabled", "yes",
21      "--cluster-config-file", "node.conf"]
22  r4:
23    image: redis
24    container_name: r4
25    network_mode: "host"
26    entrypoint: ["redis-server", "--port", "7004", "--cluster-enabled", "yes",
27      "--cluster-config-file", "node.conf"]
28  r5:
29    image: redis
```

```
26     container_name: r5
27     network_mode: "host"
28     entrypoint: ["redis-server", "--port", "7005", "--cluster-enabled", "yes",
    "--cluster-config-file", "node.conf"]
29   r6:
30     image: redis
31     container_name: r6
32     network_mode: "host"
33     entrypoint: ["redis-server", "--port", "7006", "--cluster-enabled", "yes",
    "--cluster-config-file", "node.conf"]
```

注意：使用Docker部署Redis集群，network模式必须采用host

3.1.2.启动集群

进入 `/root/redis-cluster` 目录，使用命令启动redis：

```
1 docker-compose up -d
```

启动成功，可以通过命令查看启动进程：

```
1 ps -ef | grep redis
2 # 结果:
3 root      4822    4743  0 14:29 ?        00:00:02 redis-server *:7002 [cluster]
4 root      4827    4745  0 14:29 ?        00:00:01 redis-server *:7005 [cluster]
5 root      4897    4778  0 14:29 ?        00:00:01 redis-server *:7004 [cluster]
6 root      4903    4759  0 14:29 ?        00:00:01 redis-server *:7006 [cluster]
7 root      4905    4775  0 14:29 ?        00:00:02 redis-server *:7001 [cluster]
8 root      4912    4732  0 14:29 ?        00:00:01 redis-server *:7003 [cluster]
```

可以发现每个redis节点都以cluster模式运行。不过节点与节点之间并未建立连接。

接下来，我们使用命令创建集群：

```
1 # 进入任意节点容器
2 docker exec -it r1 bash
3 # 然后，执行命令
```

```
4 redis-cli --cluster create --cluster-replicas 1 \  
5 192.168.150.101:7001 192.168.150.101:7002 192.168.150.101:7003 \  
6 192.168.150.101:7004 192.168.150.101:7005 192.168.150.101:7006
```

命令说明：

- `redis-cli --cluster`：代表集群操作命令
- `create`：代表是创建集群
- `--cluster-replicas 1`：指定集群中每个 `master` 的副本个数为1
 - 此时 `节点总数 ÷ (replicas + 1)` 得到的就是 `master` 的数量 `n`。因此节点列表中的前 `n` 个节点就是 `master`，其它节点都是 `slave` 节点，随机分配到不同 `master`

输入命令后控制台会弹出下面的信息：

```
root@heima:/data# redis-cli --cluster create --cluster-replicas 1 \  
192.168.150.101:7001 192.168.150.101:7002 192.168.150.101:7003 \  
192.168.150.101:7004 192.168.150.101:7005 192.168.150.101:7006  
>>> Performing hash slots allocation on 6 nodes...  
Master[0] -> Slots 0 - 5460  
Master[1] -> Slots 5461 - 10922  
Master[2] -> Slots 10923 - 16383  
Adding replica 192.168.150.101:7005 to 192.168.150.101:7001  
Adding replica 192.168.150.101:7006 to 192.168.150.101:7002  
Adding replica 192.168.150.101:7004 to 192.168.150.101:7003  
>>> Trying to optimize slaves allocation for anti-affinity  
[WARNING] Some slaves are in the same host as their master  
M: 9756f40e2e48b3f5c729bcd9cff1cdf6bcda134f 192.168.150.101:7001  
slots:[0-5460] (5461 slots) master  
M: 997f139639b18b5831647d61535e134ccb862fa0 192.168.150.101:7002  
slots:[5461-10922] (5462 slots) master  
M: 2859b1a79b5b06b74bf28a22bece1f8c02ad5083 192.168.150.101:7003  
slots:[10923-16383] (5461 slots) master  
S: 2260eb3836c0adbdef6f197000feada8b2391f8b 192.168.150.101:7004  
replicates 2859b1a79b5b06b74bf28a22bece1f8c02ad5083  
S: bab9e1671d78925378cc377fc3de5be67ce152cd 192.168.150.101:7005  
replicates 9756f40e2e48b3f5c729bcd9cff1cdf6bcda134f  
S: 8b165e9895bef59db4df49736647c69ffa4a018a 192.168.150.101:7006  
replicates 997f139639b18b5831647d61535e134ccb862fa0  
Can I set the above configuration? (type 'yes' to accept): yes
```

黑马程序员-研究院

这里展示了集群中 `master` 与 `slave` 节点分配情况，并询问你是否同意。节点信息如下：

- 7001 是 `master`，节点 `id` 后6位是 `da134f`
- 7002 是 `master`，节点 `id` 后6位是 `862fa0`
- 7003 是 `master`，节点 `id` 后6位是 `ad5083`

- 7004 是 slave，节点 id 后6位是 391f8b，认 ad5083 (7003) 为 master
- 7005 是 slave，节点 id 后6位是 e152cd，认 da134f (7001) 为 master
- 7006 是 slave，节点 id 后6位是 4a018a，认 862fa0 (7002) 为 master

输入 `yes` 然后回车。会发现集群开始创建，并输出下列信息：

```
>>> 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 192.168.150.101:7001)
M: 9756f40e2e48b3f5c729bcd9cff1cdf6bcda134f 192.168.150.101:7001
  slots:[0-5460] (5461 slots) master
  1 additional replica(s)
S: 8b165e9895bef59db4df49736647c69ffa4a018a 192.168.150.101:7006
  slots: (0 slots) slave
  replicates 997f139639b18b5831647d61535e134ccb862fa0
S: bab9e1671d78925378cc377fc3de5be67ce152cd 192.168.150.101:7005
  slots: (0 slots) slave
  replicates 9756f40e2e48b3f5c729bcd9cff1cdf6bcda134f
M: 2859b1a79b5b06b74bf28a22bece1f8c02ad5083 192.168.150.101:7003
  slots:[10923-16383] (5461 slots) master
  1 additional replica(s)
S: 2260eb3836c0adbdef6f197000feada8b2391f8b 192.168.150.101:7004
  slots: (0 slots) slave
  replicates 2859b1a79b5b06b74bf28a22bece1f8c02ad5083
M: 997f139639b18b5831647d61535e134ccb862fa0 192.168.150.101:7002
  slots:[5461-10922] (5462 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.
root@heima:/data#
```

黑马程序员-研究院

接着，我们可以通过命令查看集群状态：

```
1 redis-cli -p 7001 cluster nodes
```

结果：


```
root@heima:/data# redis-cli -p 7001 cluster nodes
8b165e9895bef59db4df49736647c69ffa4a018a 192.168.150.101:7006@17006 slave 997f139639b18b5831647d61535e134ccb862fa0 0 1688800470000 2 connected
bab9e1671d78925378cc377fc3de5be67ce152cd 192.168.150.101:7005@17005 slave 9756f40e2e48b3f5c729bcd9cff1cdf6bcdal34f 0 1688800470835 1 connected
2859b1a79b5b06b74bf28a22bece1f8c02ad5083 192.168.150.101:7003@17003 master - 0 1688800468000 3 connected 10923-16383
9756f40e2e48b3f5c729bcd9cff1cdf6bcdal34f 192.168.150.101:7001@17001 myself,master - 0 1688800469000 1 connected 0-5460
2260eb3836c0adbdef6f197000feada8b2391f8b 192.168.150.101:7004@17004 slave 2859b1a79b5b06b74bf28a22bece1f8c02ad5083 0 1688800468825 3 connected
997f139639b18b5831647d61535e134ccb862fa0 192.168.150.101:7002@17002 master - 0 1688800471840 2 connected 5461-10922
```

黑马程序员-研究院

3.2.散列插槽

数据要分片存储到不同的Redis节点，肯定需要有分片的依据，这样下次查询的时候才能知道去哪个节点查询。很多数据分片都会采用一致性hash算法。而Redis则是利用散列插槽（`hash slot`）的方式实现数据分片。

详见官方文档：

 <https://redis.io/docs/management/scaling/#redis-cluster-101>

Scale with Redis Cluster

Horizontal scaling with Redis Cluster

在Redis集群中，共有16384个 `hash slots`，集群中的每一个master节点都会分配一定数量的 `hash slots`。具体的分配在集群创建时就已经指定了：

```
root@heima:/data# redis-cli --cluster create --cluster-replicas 1 \
192.168.150.101:7001 192.168.150.101:7002 192.168.150.101:7003 \
192.168.150.101:7004 192.168.150.101:7005 192.168.150.101:7006
>>> Performing hash slots allocation on 6 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922
Master[2] -> Slots 10923 - 16383
Adding replica 192.168.150.101:7005 to 192.168.150.101:7001
Adding replica 192.168.150.101:7006 to 192.168.150.101:7002
Adding replica 192.168.150.101:7004 to 192.168.150.101:7003
```

黑马程序员-研究院

如图中所示：

- Master[0]，本例中就是7001节点，分配到的插槽是0~5460
- Master[1]，本例中就是7002节点，分配到的插槽是5461~10922
- Master[2]，本例中就是7003节点，分配到的插槽是10923~16383

当我们读写数据时，Redis基于 `CRC16` 算法对 `key` 做 `hash` 运算，得到的结果与 `16384` 取余，就计算出了这个 `key` 的 `slot` 值。然后到 `slot` 所在的Redis节点执行读写操作。

不过 `hash slot` 的计算也分两种情况：

- 当 `key` 中包含 `{}` 时，根据 `{}` 之间的字符串计算 `hash slot`
- 当 `key` 中不包含 `{}` 时，则根据整个 `key` 字符串计算 `hash slot`

例如：

- `key` 是 `user`，则根据 `user` 来计算 `hash slot`
- `key` 是 `user:{age}`，则根据 `age` 来计算 `hash slot`

我们来测试一下，先于 `7001` 建立连接：

```
1 # 进入容器
2 docker exec -it r1 bash
3 # 进入redis-cli
4 redis-cli -p 7001
5 # 测试
6 set user jack
```

会发现报错了：

```
root@heima:/data# redis-cli -p 7001
127.0.0.1:7001>
127.0.0.1:7001> set user jack
(error) MOVED 5474 192.168.150.101:7002
127.0.0.1:7001>
```

黑马程序员-研究院

提示我们 `MOVED 5474`，其实就是经过计算，得出 `user` 这个 `key` 的 `hash slot` 是 `5474`，而 `5474` 是在 `7002` 节点，不能在 `7001` 上写入！！

说好的任意节点都可以读写呢？

这是因为我们连接的方式有问题，连接集群时，要加 `-c` 参数：

```
1 # 通过7001连接集群
2 redis-cli -c -p 7001
3 # 存入数据
4 set user jack
```

结果如下：

```
root@heima:/data# redis-cli -c -p 7001
127.0.0.1:7001>
127.0.0.1:7001> set user jack
-> Redirected to slot [5474] located at 192.168.150.101:7002
OK
192.168.150.101:7002>
```

黑马程序员-研究院

可以看到，客户端自动跳转到了 5474 这个 slot 所在的 7002 节点。

现在，我们添加一个新的key，这次加上 {}：

```
1 # 试一下key中带{}
2 set user:{age} 21
3
4 # 再试一下key中不带{}
5 set age 20
```

结果如下：

```
192.168.150.101:7002>
192.168.150.101:7002> set user:{age} 21
-> Redirected to slot [741] located at 192.168.150.101:7001
OK
192.168.150.101:7001>
192.168.150.101:7001> get user
-> Redirected to slot [5474] located at 192.168.150.101:7002
"jack"
192.168.150.101:7002> set age 20
-> Redirected to slot [741] located at 192.168.150.101:7001
OK
192.168.150.101:7001>
```

黑马程序员-研究院

可以看到 user:{age} 和 age 计算出的 slot 都是 741。

3.3.故障转移

分片集群的节点之间会互相通过ping的方式做心跳检测，超时未回应的节点会被标记为下线状态。当发现master下线时，会将这个master的某个slave提升为master。

我们先打开一个控制台窗口，利用命令监测集群状态：

```
1 watch docker exec -it r1 redis-cli -p 7001 cluster nodes
```

命令前面的watch可以每隔一段时间刷新执行结果，方便我们实时监控集群状态变化。

接着，我们故技重施，利用命令让某个master节点休眠。比如这里我们让 7002 节点休眠，打开一个新的ssh控制台，输入下面命令：

```
1 docker exec -it r2 redis-cli -p 7002 DEBUG sleep 30
```

可以观察到，集群发现7002宕机，标记为下线：

```
Every 2.0s: docker exec -it r1 redis-cli -p 7001 cluster nodes Sat Jul 8 15:58:43
8b165e9895bef59db4df49736647c69ffa4a018a 192.168.150.101:7006@17006 slave 997f139639b18b5831647d61535e134ccb862fa0 0 1688803119947 2 connected
bab9e1671d78925378cc377fc3de5be67ce152cd 192.168.150.101:7005@17005 slave 9756f40e2e48b3f5c729bcd9cfff1cdf6bcd134f 0 1688803120000 1 connected
2859b1a79b5b06b74bf28a22bece1f8c02ad5083 192.168.150.101:7003@17003 master - 0 1688803122000 3 connected 10923-16383
9756f40e2e48b3f5c729bcd9cfff1cdf6bcd134f 192.168.150.101:7001@17001 myself,master - 0 1688803121000 1 connected 0-5460
2260eb3836c0adbdef6f197000feada8b2391f8b 192.168.150.101:7004@17004 slave 2859b1a79b5b06b74bf28a22bece1f8c02ad5083 0 1688803122964 3 connected
997f139639b18b5831647d61535e134ccb862fa0 192.168.150.101:7002@17002 master,fail? - 1688803107875 1688803104000 2 connected 5461-10922
```

黑马程序员-研究院

过了一段时间后，7002原本的小弟7006变成了 master：

```
Every 2.0s: docker exec -it r1 redis-cli -p 7001 cluster nodes Sat Jul 8 15:59
8b165e9895bef59db4df49736647c69ffa4a018a 192.168.150.101:7006@17006 master - 0 1688803179000 7 connected 5461-109220
bab9e1671d78925378cc377fc3de5be67ce152cd 192.168.150.101:7005@17005 slave 9756f40e2e48b3f5c729bcd9cfff1cdf6bcd134f 0 1688803179000 1 connected
2859b1a79b5b06b74bf28a22bece1f8c02ad5083 192.168.150.101:7003@17003 master - 0 1688803180299 3 connected 10923-16383
9756f40e2e48b3f5c729bcd9cfff1cdf6bcd134f 192.168.150.101:7001@17001 myself,master - 0 1688803177000 1 connected 0-5460
2260eb3836c0adbdef6f197000feada8b2391f8b 192.168.150.101:7004@17004 slave 2859b1a79b5b06b74bf28a22bece1f8c02ad5083 0 1688803179294 3 connected
997f139639b18b5831647d61535e134ccb862fa0 192.168.150.101:7002@17002 slave 8b165e9895bef59db4df49736647c69ffa4a018a 0 1688803178289 7 connected
```

黑马程序员-研究院

而7002被标记为 slave，而且其 master 正好是7006，主从地位互换。

3.4.总结

Redis分片集群如何判断某个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？

- Redis计算key的插槽值时会判断key中是否包含 `{}`，如果有则基于 `{}` 内的字符计算插槽
- 数据的key中可以加入 `{类型}`，例如key都以 `{typeId}` 为前缀，这样同类型数据计算的插槽一定相同

3.5.Java客户端连接分片集群（选学）

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，参考 2.5节：

- 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：

```
1 spring:
2   redis:
3     cluster:
4       nodes:
5         - 192.168.150.101:7001
6         - 192.168.150.101:7002
7         - 192.168.150.101:7003
8         - 192.168.150.101:8001
9         - 192.168.150.101:8002
10        - 192.168.150.101:8003
```

4.Redis数据结构

我们常用的Redis数据类型有5种，分别是：

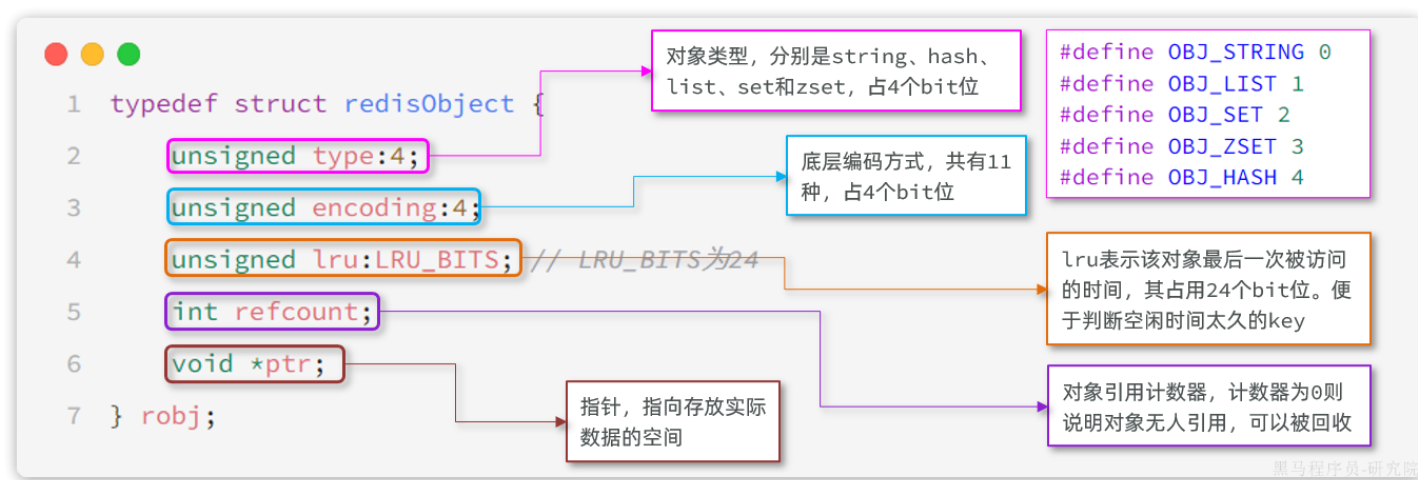
- String
- List
- Set
- SortedSet
- Hash

还有一些高级数据类型，比如Bitmap、HyperLogLog、GEO等，其底层都是基于上述5种基本数据类型。因此在Redis的源码中，其实只有5种数据类型。

4.1.RedisObject

不管是任何一种数据类型，最终都会封装为RedisObject格式，它是一种结构体，C语言中的一种结构，可以理解为Java中的类。

结构大概是这样的：



可以看到整个结构体中并不包含真实的数据，仅仅是对象头信息，内存占用的大小为 $4+4+24+32+64 = 128\text{bit}$

也就是16字节，然后指针 `ptr` 指针指向的才是真实数据存储的内存地址。所以RedisObject的内存开销是很大的。

属性中的 `encoding` 就是当前对象底层采用的**数据结构或编码方式**，可选的有11种之多：

编号	编码方式	说明
0	OBJ_ENCODING_RAW	raw编码动态字符串
1	OBJ_ENCODING_INT	long类型的整数的字符串
2	OBJ_ENCODING_HT	hash表（也叫dict）
3	OBJ_ENCODING_ZIPMAP	已废弃
4	OBJ_ENCODING_LINKEDLIST	双端链表
5	OBJ_ENCODING_ZIPLIST	压缩列表
6	OBJ_ENCODING_INTSET	整数集合

7	OBJ_ENCODING_SKIPLIST	跳表
8	OBJ_ENCODING_EMBSTR	embstr编码的动态字符串
9	OBJ_ENCODING_QUICKLIST	快速列表
10	OBJ_ENCODING_STREAM	Stream流
11	OBJ_ENCODING_LISTPACK	紧凑列表

Redis中的5种不同的数据类型采用的底层数据结构和编码方式如下：

数据类型	编码方式
STRING	int、embstr、raw
LIST	LinkedList和ZipList (3.2以前)、QuickList (3.2以后)
SET	intset、HT
ZSET	ZipList (7.0以前)、Listpack (7.0以后)、HT、SkipList
HASH	ZipList (7.0以前)、Listpack (7.0以后)、HT

这些数据类型比较复杂，我们**重点讲解几个面试会问的**，其它的大家可以查看黑马程序员发布的Redis专业课程：

<https://player.bilibili.com/player.html?bvid=BV1cr4y1671t&p=145&page=145&autoplay=0>

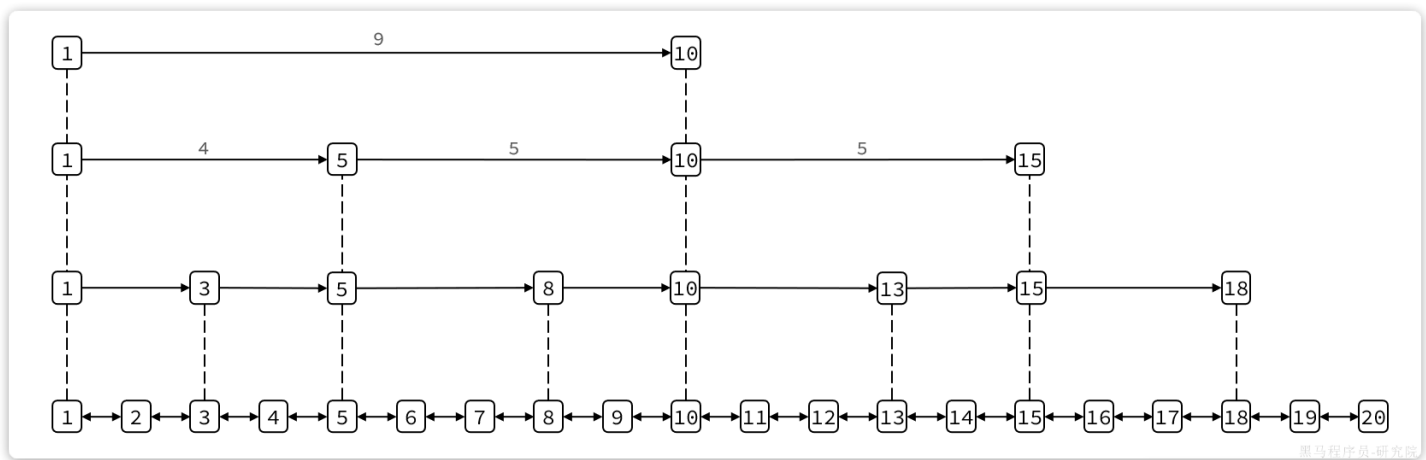
4.2.SkipList

SkipList（跳表）首先是链表，但与传统链表相比有几点差异：

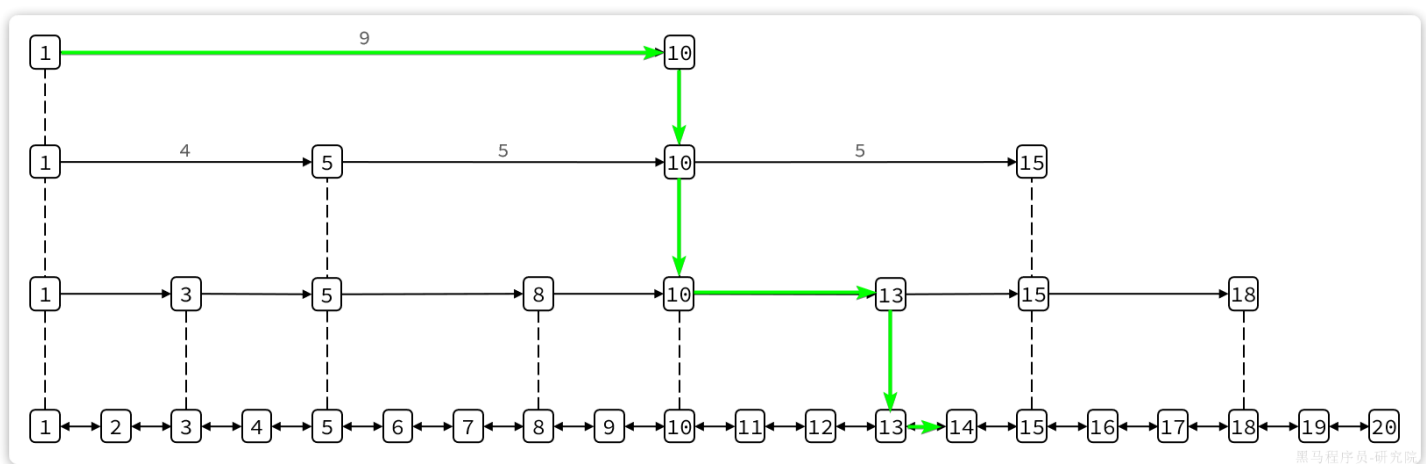
- 元素按照升序排列存储
- 节点可能包含多个指针，指针跨度不同。

传统链表只有指向前后元素的指针，因此只能顺序依次访问。如果查找的元素在链表中间，查询的效率会比较低。而SkipList则不同，它内部包含跨度不同的多级指针，可以让我们跳跃查找链表中间的元素，效率非常高。

其结构如图：



我们可以看到1号元素就有指向3、5、10的多个指针，查询时就可以跳跃查找。例如我们要找大小为14的元素，查找的流程是这样的：



- 首先找元素1节点最高级指针，也就是4级指针，起始元素大小为1，指针跨度为9，可以判断出目标元素大小为10。由于14比10大，肯定要从10这个元素向下接着找。
- 找到10这个元素，发现10这个元素的最高级指针跨度为5，判断出目标元素大小为15，大于14，需要判断下级指针
- 10这个元素的2级指针跨度为3，判断出目标元素为13，小于14，因此要基于元素13接着找
- 13这个元素最高级指针跨度为2，判断出目标元素为15，比14大，需要判断下级指针。
- 13的下级指针跨度为1，因此目标元素是14，刚好与目标一致，找到。

这种多级指针的查询方式就避免了传统链表的逐个遍历导致的查询效率下降问题。在对有序数据做随机查询和排序时效率非常高。

跳表的结构体如下：

```
1 typedef struct zskiplist {
2     // 头尾节点指针
```



```

3     struct zskiplistNode *header, *tail;
4     // 节点数量
5     unsigned long length;
6     // 最大的索引层级
7     int level;
8 } zskiplist;

```

可以看到SkipList主要属性是header和tail，也就是头尾指针，因此它是支持双向遍历的。

跳表中节点的结构体如下：

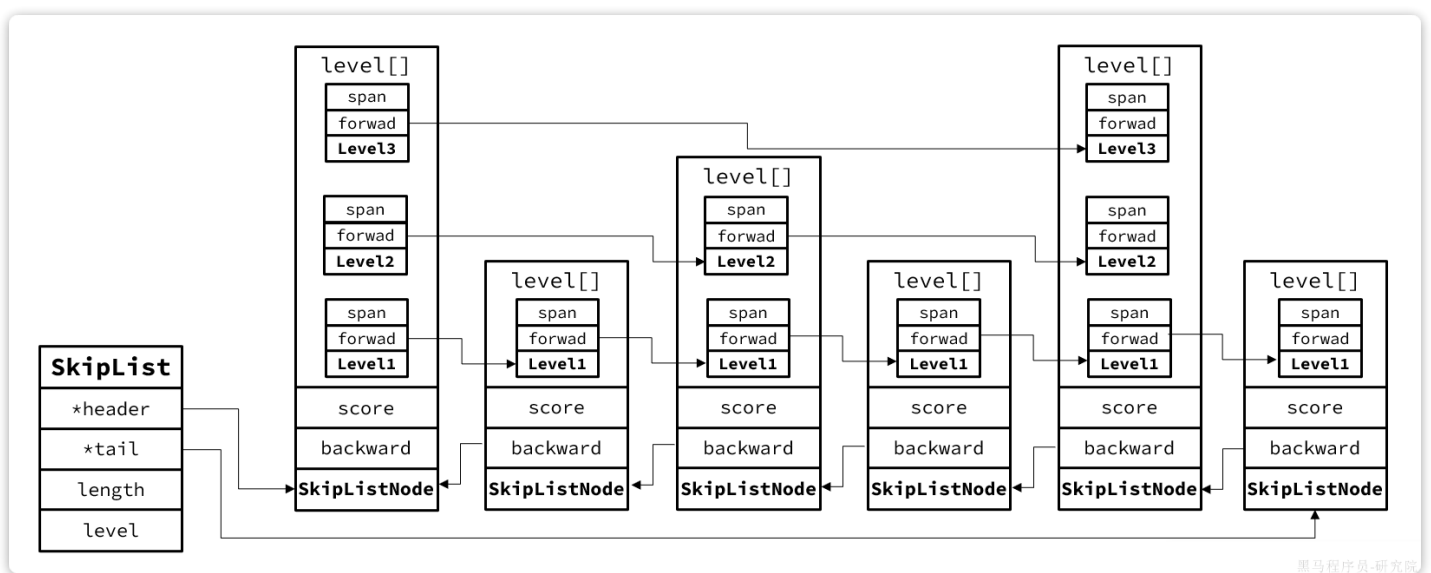
```

1 typedef struct zskiplistNode {
2     sds ele; // 节点存储的字符串
3     double score; // 节点分数，排序、查找用
4     struct zskiplistNode *backward; // 前一个节点指针
5     struct zskiplistLevel {
6         struct zskiplistNode *forward; // 下一个节点指针
7         unsigned long span; // 索引跨度
8     } level[]; // 多级索引数组
9 } zskiplistNode;

```

每个节点中都包含ele和score两个属性，其中score是得分，也就是节点排序的依据。ele则是节点存储的字符串数据指针。

其内存结构如下：



4.3.SortedSet

✓ **面试题：**Redis的 `SortedSet` 底层的数据结构是怎样的？

答：SortedSet是有序集合，底层的存储的每个数据都包含element和score两个值。score是得分，element则是字符串值。SortedSet会根据每个element的score值排序，形成有序集合。

它支持的操作很多，比如：

- 根据element查询score值
- 按照score值升序或降序查询element

要实现根据element查询对应的score值，就必须实现element与score之间的键值映射。SortedSet底层是基于**HashTable**来实现的。

要实现对score值排序，并且查询效率还高，就需要有一种高效的有序数据结构，SortedSet是基于**跳表**实现的。

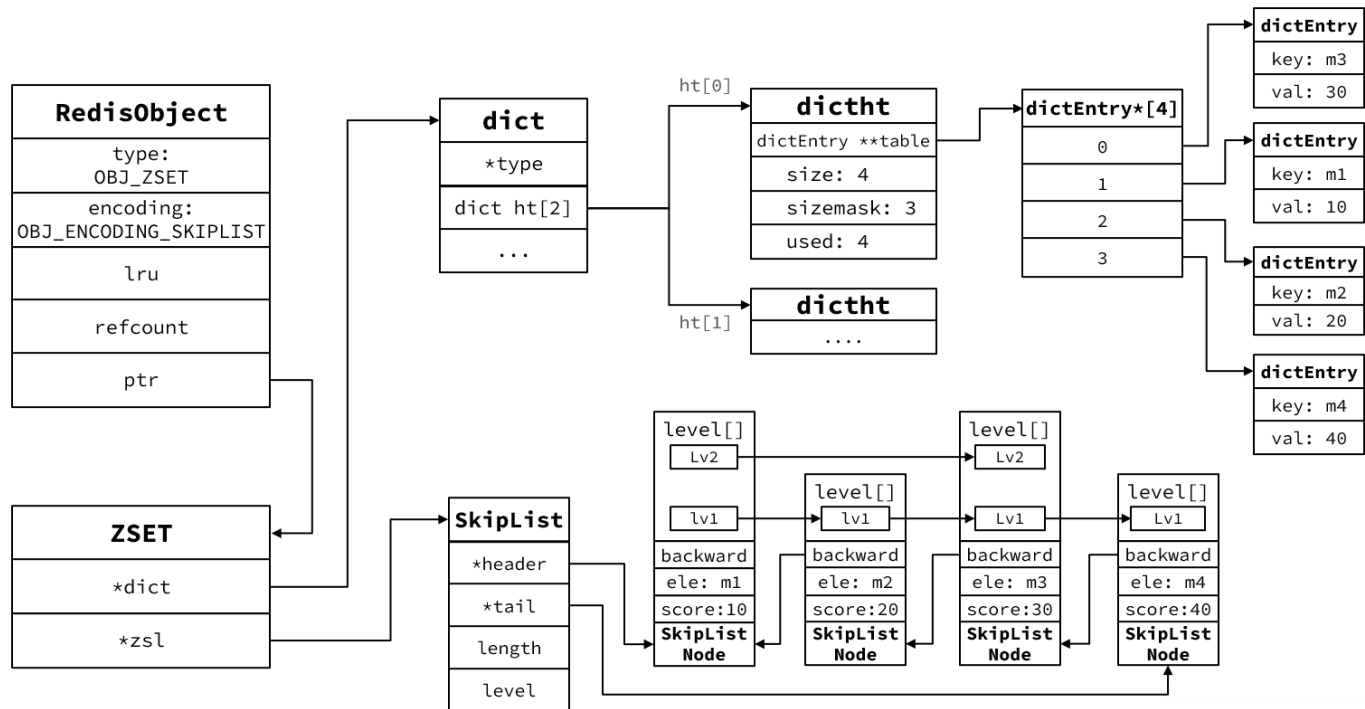
加分项：因为SortedSet底层需要用到两种数据结构，对内存占用比较高。因此Redis底层会对SortedSet中的元素大小做判断。如果**元素大小小于128且每个元素都小于64字节**，SortedSet底层会采用**ZipList**，也就是**压缩列表**来代替**HashTable**和**SkipList**

不过，`ZipList` 存在连锁更新问题，因此而在Redis7.0版本以后，`ZipList` 又被替换为**Listpack**（紧凑列表）。

Redis源码中 `zset` ，也就是 `SortedSet` 的结构体如下：

```
1 typedef struct zset {  
2     dict *dict; // dict, 底层就是HashTable  
3     zskiplist *zsl; // 跳表  
4 } zset;
```

其内存结构如图：



黑马程序员-研究院

5.Redis内存回收

Redis之所以性能强，最主要的原因就是基于内存存储。然而单节点的Redis其内存大小不宜过大，会影响持久化或主从同步性能。

我们可以通过修改redis.conf文件，添加下面的配置来配置Redis的最大内存：

```
1 maxmemory 1gb
```

当内存达到上限，就无法存储更多数据了。因此，Redis内部会有两套内存回收的策略：

- 内存过期策略
- 内存淘汰策略

5.1.内存过期处理

存入Redis中的数据可以配置过期时间，到期后再次访问会发现这些数据都不存在了，也就是被过期清理了。

5.1.1.过期命令

Redis中通过 `expire` 命令可以给KEY设置 `TTL`（过期时间），例如：

```
1 # 写入一条数据
2 set num 123
3 # 设置20秒过期时间
4 expire num 20
```

不过set命令本身也可以支持过期时间的设置：

```
1 # 写入一条数据并设置20s过期时间
2 set num EX 20
```

当过期时间到了以后，再去查询数据，会发现数据已经不存在。

5.1.2.过期策略

那么问题来了：

- Redis如何判断一个KEY是否过期呢？
- Redis又是何时删除过期KEY的呢？

Redis不管有多少种数据类型，本质是一个 `KEY-VALUE` 的键值型数据库，而这种键值映射底层正式基于HashTable来实现的，在Redis中叫做Dict.

来看下RedisDB的底层源码：

```
1 typedef struct redisDb {
2     dict dict;                /* The keyspace for this DB , 也就是存放KEY和VALUE
   的哈希表*/
3     dict *expires;            /* 同样是哈希表，但保存的是设置了TTL的KEY，及其到期时
   间*/
4     dict *blocking_keys;      /* Keys with clients waiting for data (BLPOP)*/
5     dict *ready_keys;         /* Blocked keys that received a PUSH */
6     dict *watched_keys;       /* WATCHED keys for MULTI/EXEC CAS /
7     int id;                   /* Database ID, 0 ~ 15 /
8     long long avg_ttl;        /* Average TTL, just for stats /
9     unsigned long expires_cursor; /* Cursor of the active expire cycle. */
```

```
10     list *defrag_later;           /* List of key names to attempt to defrag one
    by one, gradually. */
11 } redisDb;
```

现在回答第一个问题：

✅ **面试题：**Redis如何判断KEY是否过期呢？

答：在Redis中会有两个Dict，也就是HashTable，其中一个记录KEY-VALUE键值对，另一个记录KEY和过期时间。要判断一个KEY是否过期，只需要到记录过期时间的Dict中根据KEY查询即可。

Redis是何时删除过期KEY的呢？

Redis并不会在KEY过期时立刻删除KEY，因为要实现这样的效果就必须给每一个过期的KEY设置时钟，并监控这些KEY的过期状态。无论对CPU还是内存都会带来极大的负担。

Redis的过期KEY删除策略有两种：

- 惰性删除
- 周期删除

惰性删除，顾名思义就是过期后不会立刻删除。那在什么时候删除呢？

Redis会在每次访问KEY的时候判断当前KEY有没有设置过期时间，如果有，过期时间是否已经到期。对应的源码如下：

```
1 // db.c
2 // 寻找要执行写操作的key
3 robj *lookupKeyWriteWithFlags(redisDb *db, robj *key, int flags) {
4     // 检查key是否过期，如果过期则删除
5     expireIfNeeded(db, key);
6     return lookupKey(db, key, flags);
7 }
8
9 // 寻找要执行读操作的key
10 robj *lookupKeyReadWithFlags(redisDb *db, robj *key, int flags) {
11     robj *val;
12     // 检查key是否过期，如果过期则删除
13     if (expireIfNeeded(db, key) == 1) {
14         // 略 ...
```

```
15     }
16     val = lookupKey(db,key,flags);
17     if (val == NULL)
18         goto keymiss;
19     server.stat_keyspace_hits++;
20     return val;
21 }
```

周期删除：顾名思义是通过一个定时任务，周期性的抽样部分过期的key，然后执行删除。

执行周期有两种：

- **SLOW模式：**Redis会设置一个定时任务 `serverCron()`，按照 `server.hz` 的频率来执行过期key清理
- **FAST模式：**Redis的每个事件循环前执行过期key清理（事件循环就是NIO事件处理的循环）。

SLOW模式规则：

- ① 执行频率受 `server.hz` 影响，默认为10，即每秒执行10次，每个执行周期100ms。
- ② 执行清理耗时不超过一次执行周期的25%，即25ms。
- ③ 逐个遍历db，逐个遍历db中的bucket，抽取20个key判断是否过期
- ④ 如果没达到时间上限（25ms）并且过期key比例大于10%，再进行一次抽样，否则结束

FAST模式规则（过期key比例小于10%不执行）：

- ① 执行频率受 `beforeSleep()` 调用频率影响，但两次FAST模式间隔不低于2ms
- ② 执行清理耗时不超过1ms
- ③ 逐个遍历db，逐个遍历db中的bucket，抽取20个key判断是否过期
- ④ 如果没达到时间上限（1ms）并且过期key比例大于10%，再进行一次抽样，否则结束

5.2.内存淘汰策略

对于某些特别依赖于Redis的项目而言，仅仅依靠过期KEY清理是不够的，内存可能很快就达到上限。因此Redis允许设置内存告警阈值，当内存使用达到阈值时就会主动挑选部分KEY删除以释放更多内存。这叫做**内存淘汰**机制。

5.2.1.内存淘汰时机

那么问题来了，当内存达到阈值时执行内存淘汰，但问题是Redis什么时候会执去判断内存是否达到预警呢？

Redis每次执行任何命令时，都会判断内存是否达到阈值：

```
1 // server.c中处理命令的部分源码
2 int processCommand(client *c) {
3     // ... 略
4     if (server.maxmemory && !server.lua_timedout) {
5         // 调用performEvictions()方法尝试进行内存淘汰
6         int out_of_memory = (performEvictions() == EVICT_FAIL);
7         // ... 略
8         if (out_of_memory && reject_cmd_on_oom) {
9             // 如果内存依然不足，直接拒绝命令
10            rejectCommand(c, shared.oomerr);
11            return C_OK;
12        }
13    }
14 }
```

5.2.2.淘汰策略

好了，知道什么时候尝试淘汰了，那具体Redis是如何判断该淘汰哪些 `Key` 的呢？

Redis支持8种不同的内存淘汰策略：

- `noeviction`：不淘汰任何key，但是内存满时不允许写入新数据，默认就是这种策略。
- `volatile-ttl`：对设置了TTL的key，比较key的剩余TTL值，TTL越小越先被淘汰
- `allkeys-random`：对全体key，随机进行淘汰。也就是直接从db->dict中随机挑选
- `volatile-random`：对设置了TTL的key，随机进行淘汰。也就是从db->expires中随机挑选。
- `allkeys-lru`：对全体key，基于LRU算法进行淘汰
- `volatile-lru`：对设置了TTL的key，基于LRU算法进行淘汰
- `allkeys-lfu`：对全体key，基于LFU算法进行淘汰
- `volatile-lfu`：对设置了TTL的key，基于LFI算法进行淘汰

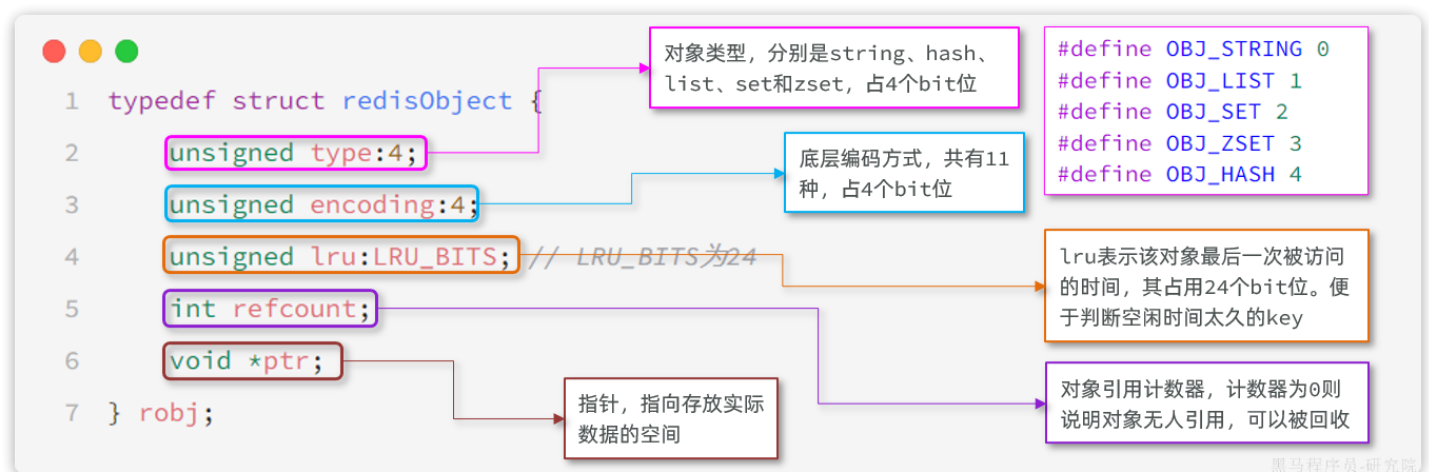
比较容易混淆的有两个算法：

- **LRU** (Least Recently Used)，最近最久未使用。用当前时间减去最后一次访问时间，这个值越大则淘汰优先级越高。
- **LFU** (Least Frequently Used)，最少频率使用。会统计每个key的访问频率，值越小淘汰优先级越高。

Redis怎么知道某个KEY的 最近一次访问时间 或者是 访问频率 呢？

还记不记得之前讲过的RedisObject的结构？

回忆一下：



其中的 **lru** 就是记录最近一次访问时间和访问频率的。当然，你选择 **LRU** 和 **LFU** 时的记录方式不同：

- **LRU**：以秒为单位记录最近一次访问时间，长度24bit
- **LFU**：高16位以分钟为单位记录最近一次访问时间，低8位记录逻辑访问次数

时间就不说了，那么逻辑访问次数又是怎么回事呢？8位无符号数字最大才255，访问次数超过255怎么办？

这就要聊起Redis的**逻辑访问次数**算法了，LFU的访问次数之所以叫做**逻辑访问次数**，是因为并不是每次key被访问都计数，而是通过运算：

- ① 生成 $[0, 1)$ 之间的随机数 **R**
- ② 计算 $1 / (\text{旧次数} * \text{lfu_log_factor} + 1)$ ，记录为 **P**，**lfu_log_factor** 默认为10
- ③ 如果 $R < P$ ，则计数器 **+1**，且最大不超过255

- ④ 访问次数会随时间衰减，距离上一次访问时间每隔 `lfu_decay_time` 分钟(默认1)，计数器 -1

显然LFU的基于访问频率的统计更符合我们的淘汰目标，因此**官方推荐使用LFU算法**。

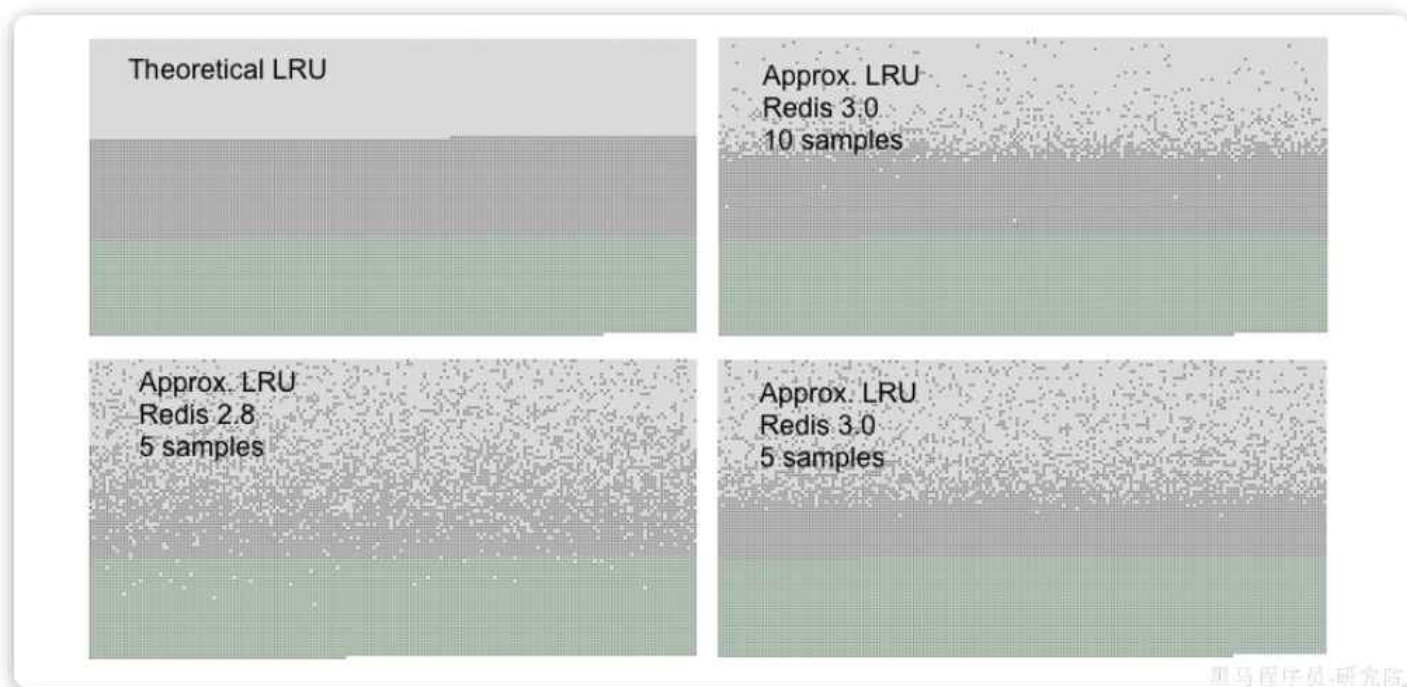
算法我们弄明白了，不过这里大家要注意一下：Redis中的 `KEY` 可能有数百万甚至更多，每个KEY都有自己访问时间或者逻辑访问次数。我们要找出时间最早的或者访问次数最小的，难道要把Redis中**所有数据排序**？

要知道Redis的内存淘汰是在每次执行命令时处理的。如果每次执行命令都先对全量数据做内存排序，那命令的执行时长肯定会非常长，这是不现实的。

所以Redis采取的是**抽样法**，即每次抽样一定数量（`maxmemory_smples`）的key，然后基于内存策略做排序，找出淘汰优先级最高的，删除这个key。这就导致Redis的算法并不是真正的**LRU**，而是一种基于抽样的**近似LRU算法**。

不过，在Redis3.0以后改进了这个算法，引入了一个淘汰候选池，抽样的key要与候选池中的key比较淘汰优先级，优先级更高的才会被放入候选池。然后在候选池中找出优先级最高的淘汰掉，这就使算法的结果更接近与真正的LRU算法了。特别是在抽样值较高的情况下（例如10），可以达到与真正的LRU接近的效果。

这也是官方给出的真正LRU与近似LRU的结果对比：



你可以在图表中看到三种颜色的点形成三个不同的带，每个点就是一个加入的 `KEY` 。

- 浅灰色带是被驱逐的对象

- 灰色带是没有被驱逐的对象
- 绿色带是被添加的对象

5.3.总结

✓ 面试题：Redis如何判断KEY是否过期呢？

答：在Redis中会有两个Dict，也就是HashTable，其中一个记录KEY-VALUE键值对，另一个记录KEY和过期时间。要判断一个KEY是否过期，只需要到记录过期时间的Dict中根据KEY查询即可。

✓ 面试题：Redis何时删除过期KEY？如何删除？

答：Redis的过期KEY处理有两种策略，分别是惰性删除和周期删除。

惰性删除是指在每次用户访问某个KEY时，判断KEY的过期时间：如果过期则删除；如果未过期则忽略。

周期删除有两种模式：

- **SLOW**模式：通过一个定时任务，定期的抽样部分带有TTL的KEY，判断其是否过期。默认情况下定时任务的执行频率是每秒10次，但每次执行不能超过25毫秒。如果执行抽样后发现时间还有剩余，并且过期KEY的比例较高，则会多次抽样。
- **FAST**模式：在Redis每次处理NIO事件之前，都会抽样部分带有TTL的KEY，判断是否过期，因此执行频率较高。但是每次执行时长不能超过1ms，如果时间充足并且过期KEY比例过高，也会多次抽样

✓ 面试题：当Redis内存不足时会怎么做？

答：这取决于配置的内存淘汰策略，Redis支持很多种内存淘汰策略，例如LRU、LFU、Random. 但默认的策略是直接拒绝新的写入请求。而如果设置了其它策略，则会在每次执行命令后判断占用内存是否达到阈值。如果达到阈值则会基于配置的淘汰策略尝试进行内存淘汰，直到占用内存小于阈值为止。

✅ 面试题：那你能聊聊LRU和LFU吗？

答：LRU 是最近最久未使用。Redis的Key都是RedisObject，当启用LRU算法后，Redis会在Key的头信息中使用24个bit记录每个key的最近一次使用的时间 `lru`。每次需要内存淘汰时，就会抽样一部分KEY，找出其中空闲时间最长的，也就是 `now - lru` 结果最大的，然后将其删除。如果内存依然不足，就重复这个过程。

由于采用了抽样来计算，这种算法只能说是一种近似LRU算法。因此在Redis4.0以后又引入了 LFU 算法，这种算法是统计最近最少使用，也就是按key的访问频率来统计。当启用LFU算法后，Redis会在key的头信息中使用24bit记录最近一次使用时间和逻辑访问频率。其中高16位是以分钟为单位的最近访问时间，后8位是逻辑访问次数。与LFU类似，每次需要内存淘汰时，就会抽样一部分KEY，找出其中逻辑访问次数最小的，将其淘汰。

✅ 面试题：逻辑访问次数是如何计算的？

答：由于记录访问次数的只有 `8bit`，即便是无符号数，最大值只有255，不可能记录真实的访问次数。因此Redis统计的其实是逻辑访问次数。这其中有一个计算公式，会根据当前的访问次数做计算，结果要么是次数 `+1`，要么是次数不变。但随着当前访问次数越大，`+1` 的概率也会越低，并且最大值不超过255。

除此以外，逻辑访问次数还有一个衰减周期，默认为1分钟，即每隔1分钟逻辑访问次数会 `-1`。这样逻辑访问次数就能基本反映出一个 `key` 的访问热度了。

6.缓存问题

Redis经常被用作缓存，而缓存在使用的过程中存在很多问题需要解决。例如：

- 缓存的数据一致性问题
- 缓存击穿
- 缓存穿透
- 缓存雪崩

6.1.缓存一致性

我们先看下目前企业用的最多的缓存模型。缓存的通用模型有三种：

- **Cache Aside**：有缓存调用者自己维护数据库与缓存的一致性。即：
 - 查询时：命中则直接返回，未命中则查询数据库并写入缓存
 - 更新时：更新数据库并删除缓存，查询时自然会更新缓存
- **Read/Write Through**：数据库自己维护一份缓存，底层实现对调用者透明。底层实现：
 - 查询时：命中则直接返回，未命中则查询数据库并写入缓存
 - 更新时：判断缓存是否存在，不存在直接更新数据库。存在则更新缓存，同步更新数据库
- **Write Behind Cahing**：读写操作都直接操作缓存，由线程异步的将缓存数据同步到数据库

目前企业中使用最多的就是 **Cache Aside** 模式，因为实现起来非常简单。但缺点也很明显，就是无法保证数据库与缓存的强一致性。为什么呢？我们一起来分析一下。

Cache Aside 的写操作是要在更新数据库的同时删除缓存，那为什么不选择更新数据库的同时更新缓存，而是删除呢？

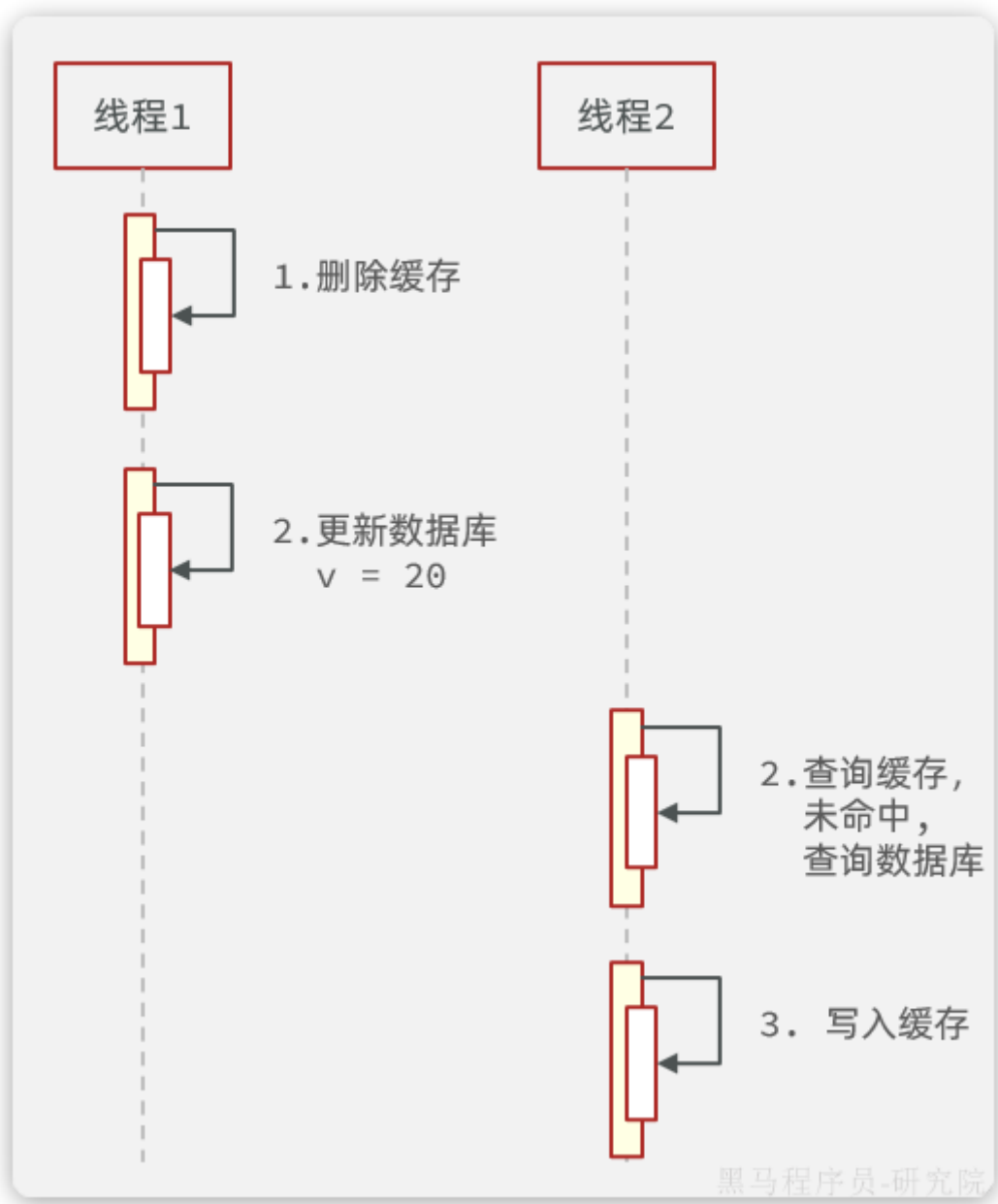
原因很简单，假如一段时间内无人查询，但是有多次更新，那这些更新都属于无效更新。采用删除方案也就是延迟更新，什么时候有人查询了，什么时候更新。

那到底是先更新数据库再删除缓存，还是先删除缓存再更新数据库呢？

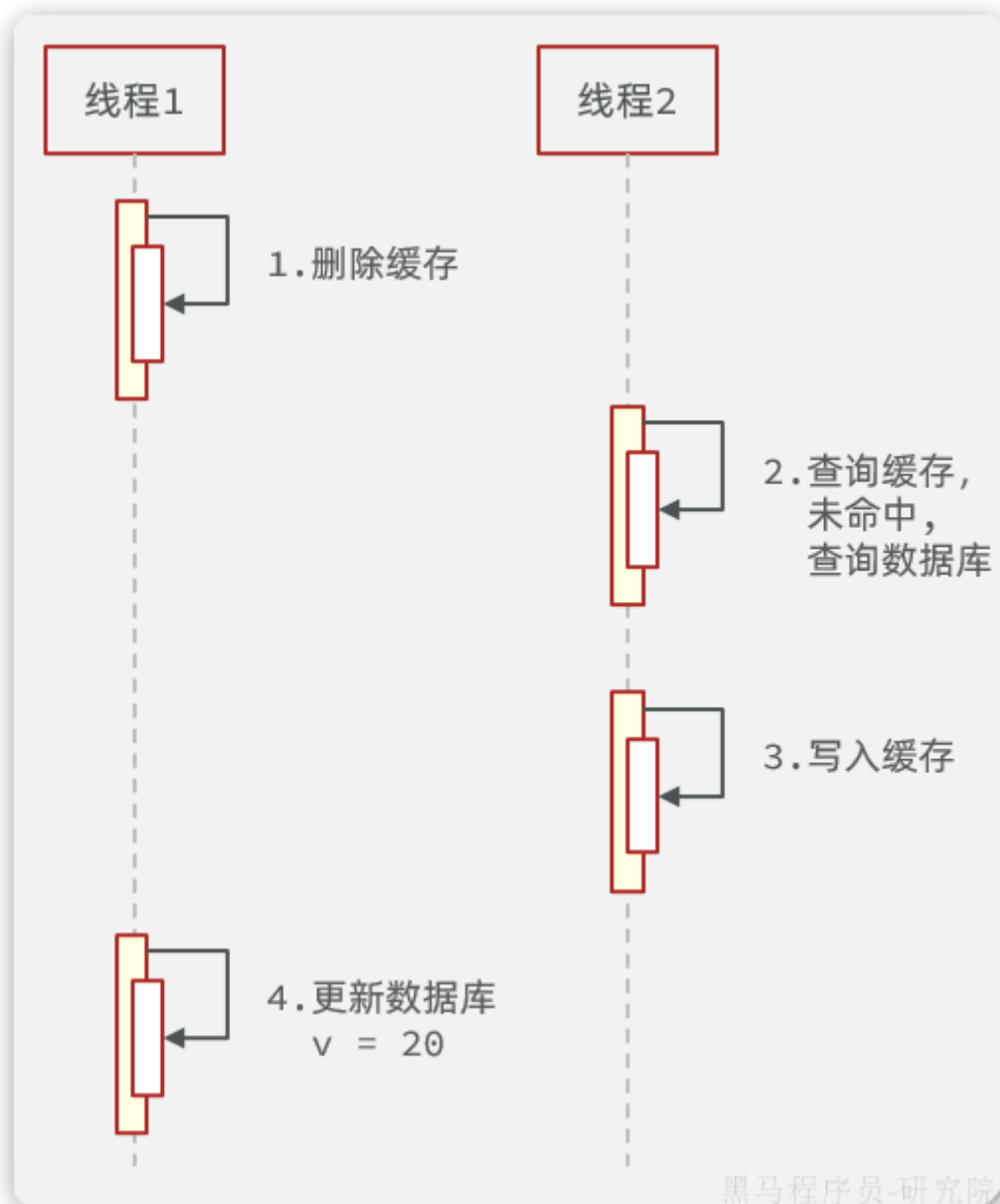
现在假设有两个线程，一个来更新数据，一个来查询数据。我们分别分析两种策略的表现。

我们先分析策略1，先更新数据库再删除缓存：

正常情况



异常情况



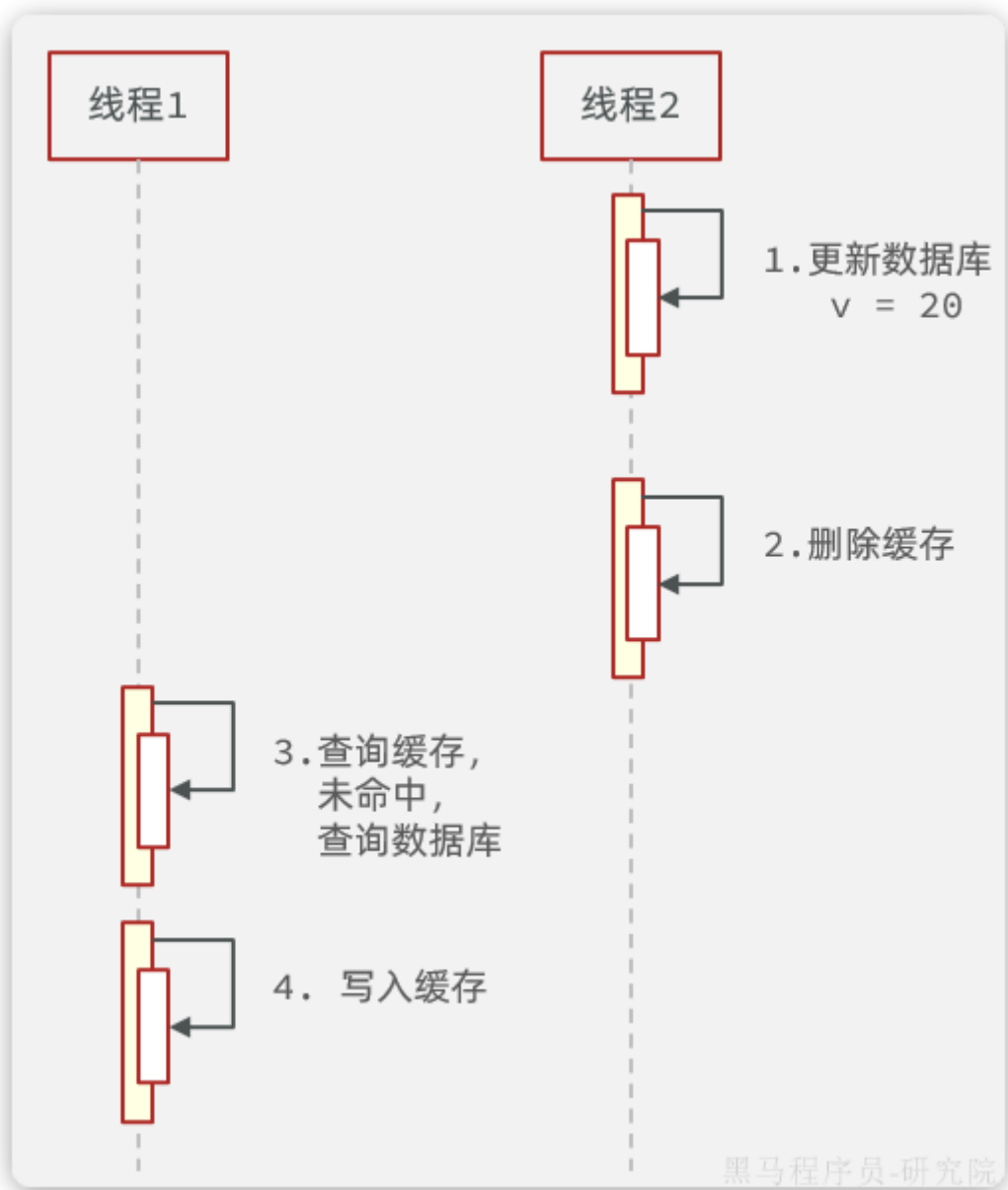
异常情况说明：

- 线程1删除缓存后，还没来得及更新数据库，
- 此时线程2来查询，发现缓存未命中，于是查询数据库，写入缓存。由于此时数据库尚未更新，查询的是旧数据。也就是说刚才的删除白删了，缓存又变成旧数据了。
- 然后线程1更新数据库，此时数据库是新数据，缓存是旧数据

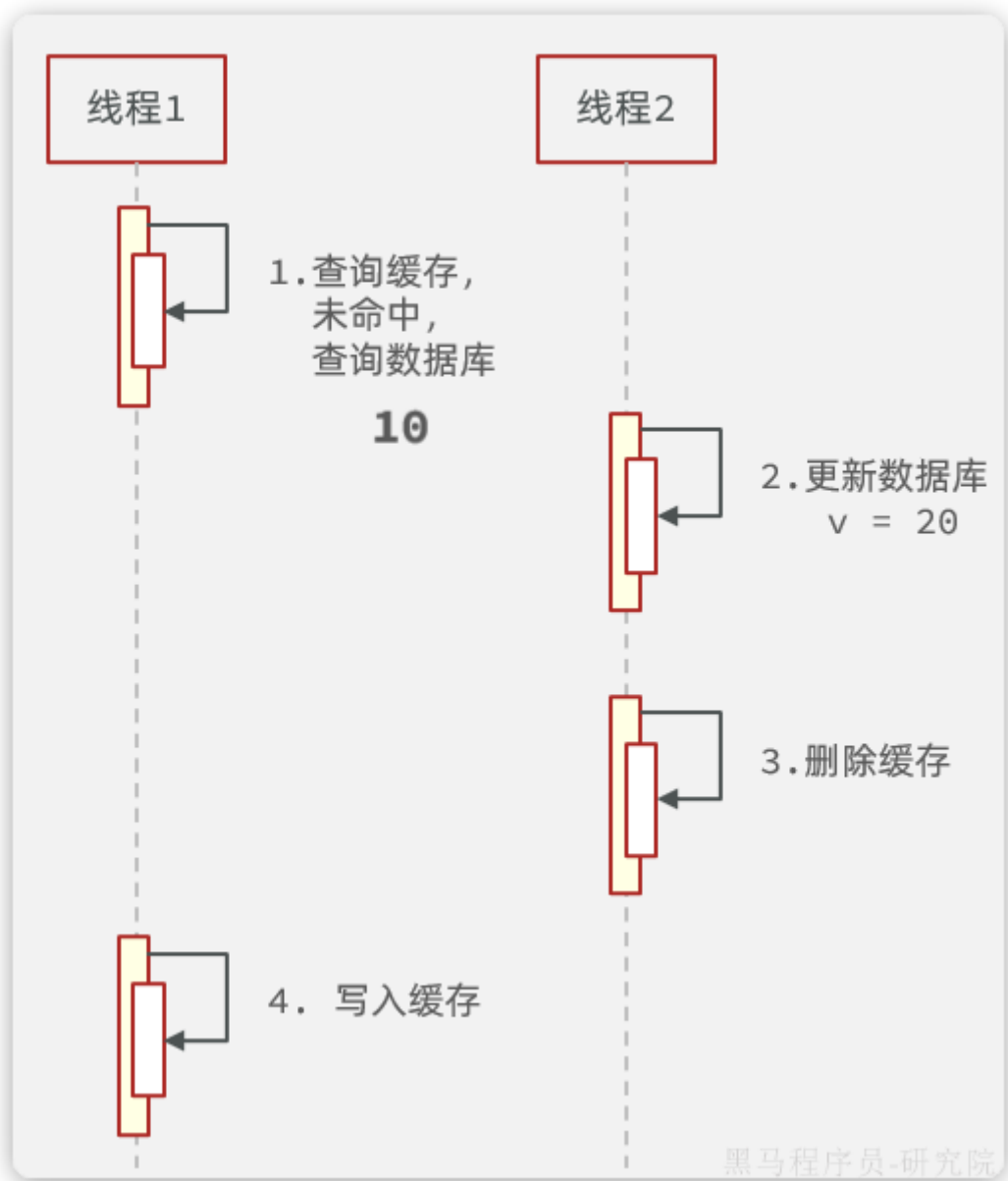
由于更新数据库的操作本身比较耗时，在期间有线程来查询数据库并更新缓存的概率非常高。因此不推荐这种方案。

再来看策略2，先更新数据库再删除缓存：

正常情况



异常情况



异常情况说明：

- 线程1查询缓存未命中，于是去查询数据库，查询到旧数据
- 线程1将数据写入缓存之前，线程2来了，更新数据库，删除缓存
- 线程1执行写入缓存的操作，写入旧数据

可以发现，异常状态发生的概率极为苛刻，线程1必须是查询数据库已经完成，但是缓存尚未写入之前。线程2要完成更新数据库同时删除缓存的两个操作。要知道线程1执行写缓存的速度在毫秒之间，速度非常快，在这么短的时间要完成数据库和缓存的操作，概率非常之低。

✅ **综上**，添加缓存的目的是为了提高系统性能，而你要付出的代价就是缓存与数据库的强一致性。如果你要求数据库与缓存的强一致，那就需要加锁避免并行读写。但这就降低了性能，与缓存的目标背道而驰。

因此不管任何缓存同步方案最终的目的都是尽可能保证最终一致性，降低发生不一致的概率。我们采用先更新数据库再删除缓存的方案，已经将这种概率降到足够低，目的已经达到了。

同时我们还要给缓存加上过期时间，一旦发生缓存不一致，当缓存过期后会重新加载，数据最终还是能保证一致。这就可以作为一个兜底方案。

6.2.缓存穿透

什么是缓存穿透呢？

我们知道，当请求查询缓存未命中时，需要查询数据库以加载缓存。但是大家思考一下这样的场景：

如果我访问一个数据库中也不存在的数据。会出现什么现象？

由于数据库中不存在该数据，那么缓存中肯定也不存在。因此不管请求该数据多少次，缓存永远不可能建立，请求永远会直达数据库。

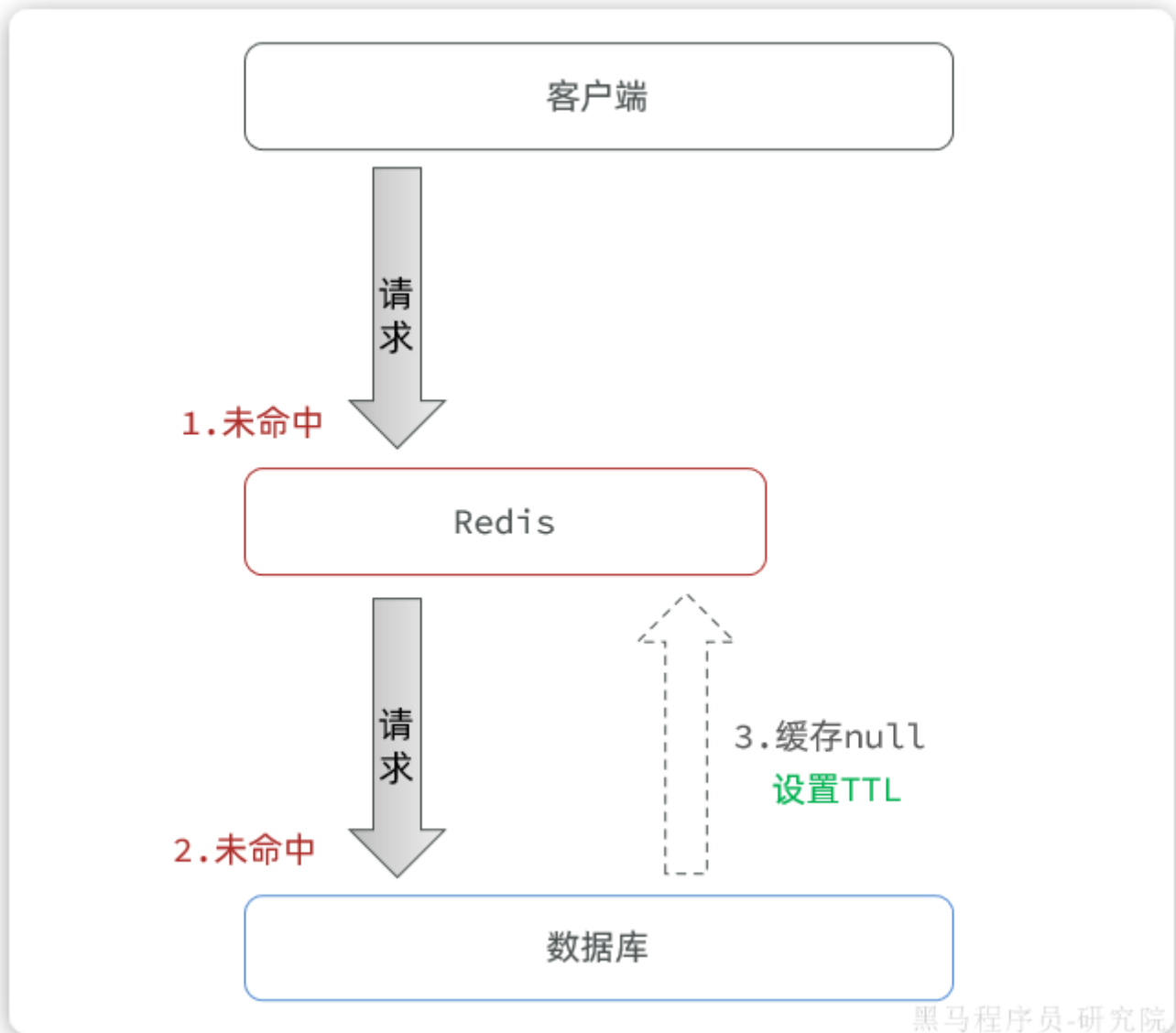
假如有不怀好意的人，开启很多线程频繁的访问一个数据库中也不存在的数据。由于缓存不可能生效，那么所有的请求都访问数据库，可能会导致数据库因过高的压力而宕机。

解决这个问题有两种思路：

- 缓存空值
- 布隆过滤器

6.2.1.缓存空值

简单来说，就是当我们发现请求的数据即不存在与缓存，也不存在与数据库时，将空值缓存到Redis，避免频繁查询数据库。实现思路如下：



优点：

- 实现简单，维护方便

缺点：

- 额外的内存消耗

6.2.2.布隆过滤器

布隆过滤是一种数据统计的算法，用于检索一个元素是否存在一个集合中。

一般我们判断集合中是否存在元素，都会先把元素保存到类似于树、哈希表等数据结构中，然后利用这些结构查询效率高的特点来快速匹配判断。但是随着元素数量越来越多，这种模式对内存的占用也越来越大，检索的速度也会越来越慢。而布隆过滤的内存占用小，查询效率却很高。

布隆过滤首先需要是一个很长的bit数组，默认数组中每一位都是0。

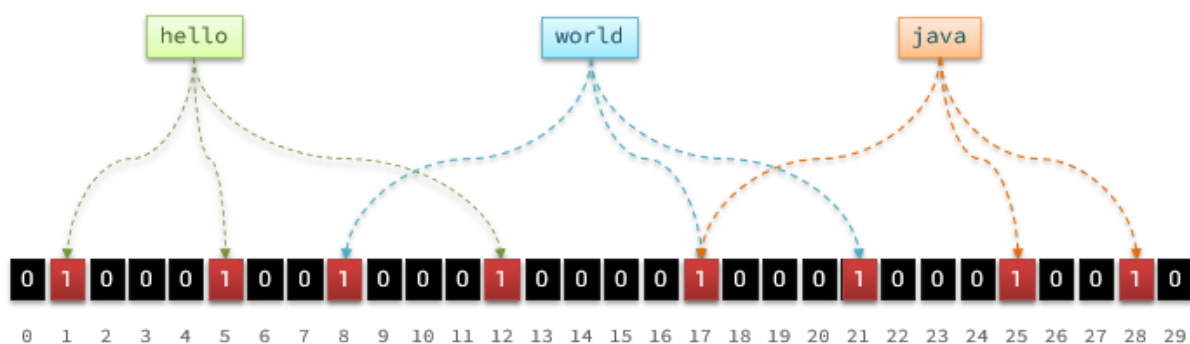


黑马程序员-研究院

然后还需要 K 个 hash 函数，将元素基于这些 hash 函数做运算的结果映射到 bit 数组的不同位置，并将这些位置置为 1，例如现在 $k=3$ ：

- `hello` 经过运算得到 3 个角标：1、5、12
- `world` 经过运算得到 3 个角标：8、17、21
- `java` 经过运算得到 3 个角标：17、25、28

则需要将每个元素对应角标位置置为 1：



黑马程序员-研究院

此时，我们要判断元素是否存在，只需要再次基于 K 个 hash 函数做运算，得到 K 个角标，判断每个角标的位置是不是 1：

- 只要全是 1，就证明元素存在
- 任意位置为 0，就证明元素一定不存在

假如某个元素本身并不存在，也没添加到布隆过滤器过。但是由于存在 hash 碰撞的可能性，这就会出现这个元素计算出的角标已经被其它元素置为 1 的情况。那么这个元素也会被误判为已经存在。

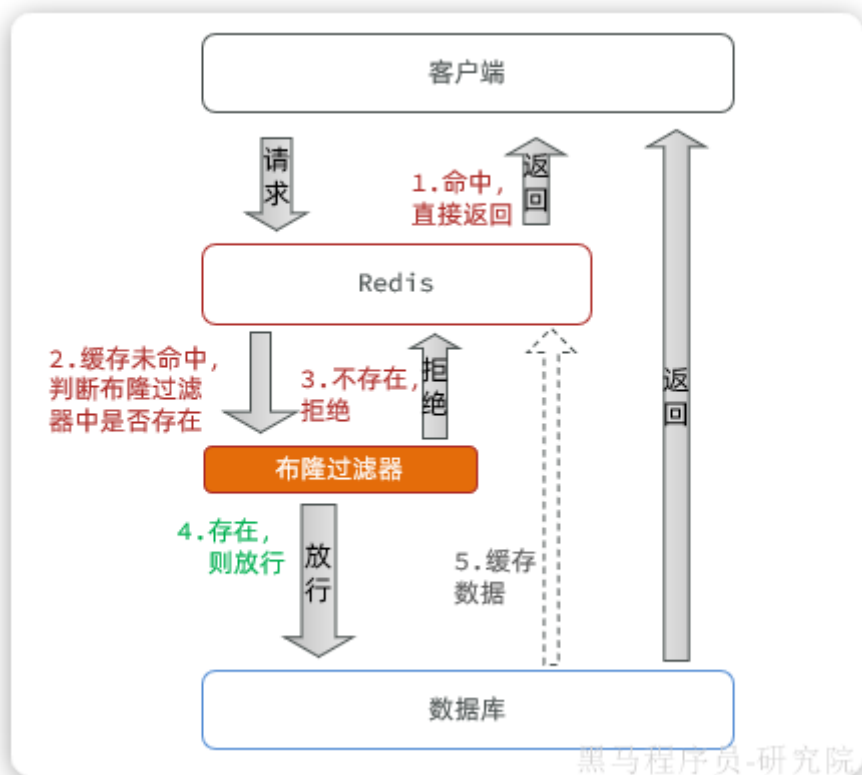
因此，布隆过滤器的判断存在误差：

- 当布隆过滤器认为元素不存在时，它**肯定不存在**
- 当布隆过滤器认为元素存在时，它**可能存在，也可能不存在**

当 bit 数组越大、Hash 函数 K 越复杂， K 越大时，这个误判的概率也就越低。由于采用 bit 数组来标示数据，即便 4,294,967,296 个 bit 位，也只占 512mb 的空间

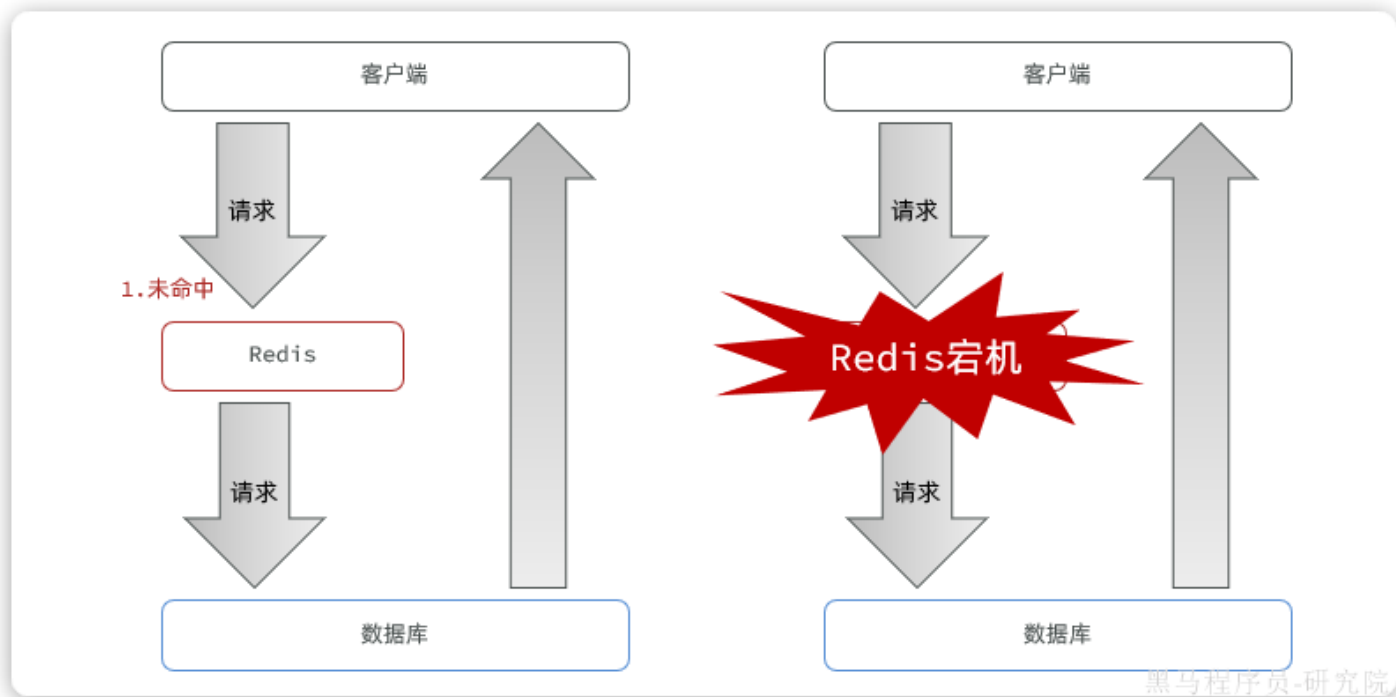
我们可以把数据库中的数据利用布隆过滤器标记出来，当用户请求缓存未命中时，先基于布隆过滤器判断。如果不存在则直接拒绝请求，存在则去查询数据库。尽管布隆过滤存在误差，但一般都在0.01%左右，可以大大减少数据库压力。

使用布隆过滤后的流程如下：



6.3.缓存雪崩

缓存雪崩是指在同一时段大量的缓存key同时失效或者Redis服务宕机，导致大量请求到达数据库，带来巨大压力。



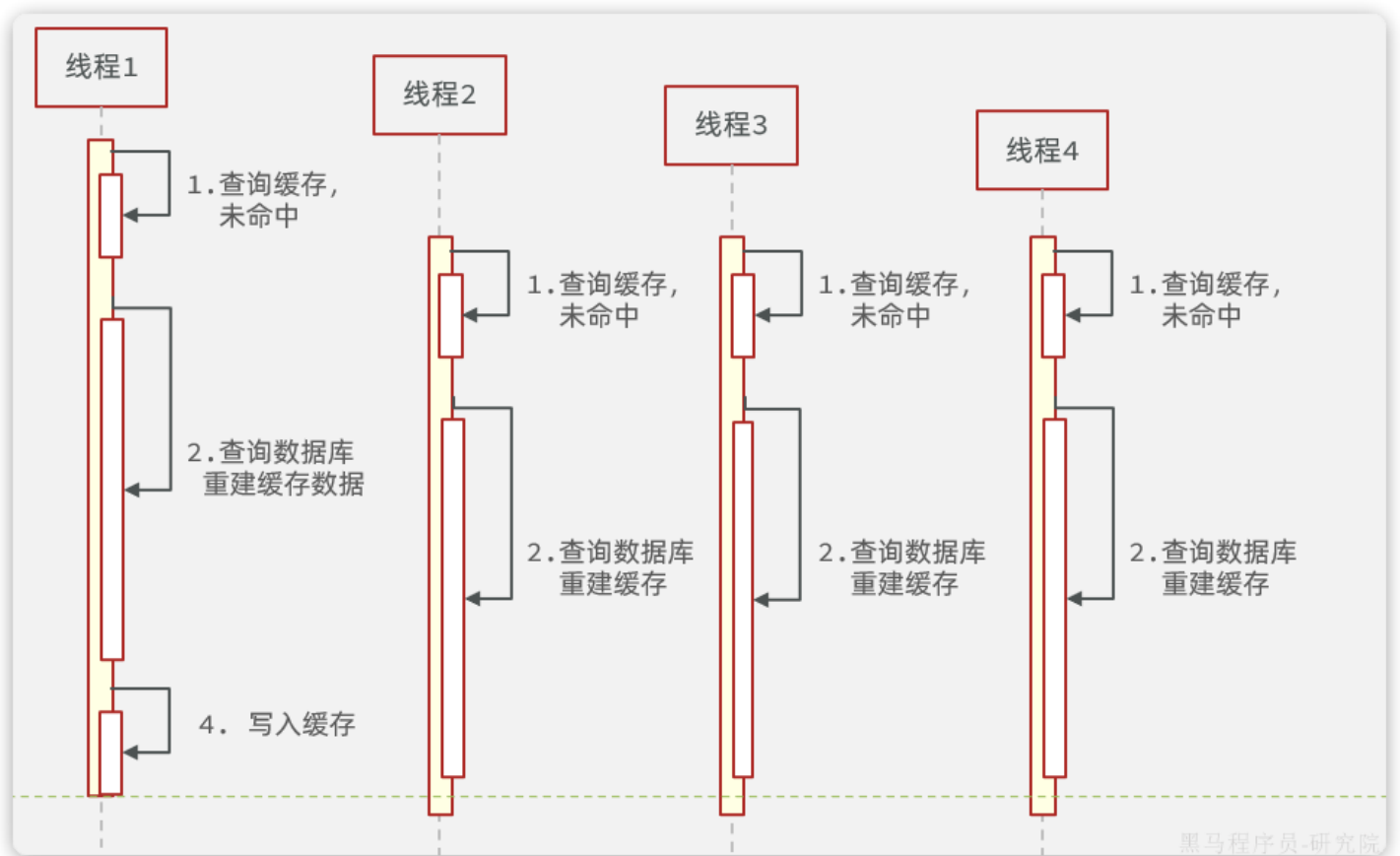
常见的解决方案有：

- 给不同的Key的TTL添加随机值，这样KEY的过期时间不同，不会大量KEY同时过期
- 利用Redis集群提高服务的可用性，避免缓存服务宕机
- 给缓存业务添加降级限流策略
- 给业务添加多级缓存，比如先查询本地缓存，本地缓存未命中再查询Redis，Redis未命中再查询数据库。即便Redis宕机，也还有本地缓存可以抗压力

6.4.缓存击穿

缓存击穿问题也叫**热点Key**问题，就是一个被高并发访问并且缓存重建业务较复杂的key突然失效了，无数的请求访问会在瞬间给数据库带来巨大的冲击。

由于我们采用的是 `Cache Aside` 模式，当缓存失效时需要下次查询时才会更新缓存。当某个key缓存失效时，如果这个key是热点key，并发访问量比较高。就会在一瞬间涌入大量请求，都发现缓存未命中，于是都会去查询数据库，尝试重建缓存。可能一瞬间就把数据库压垮了。



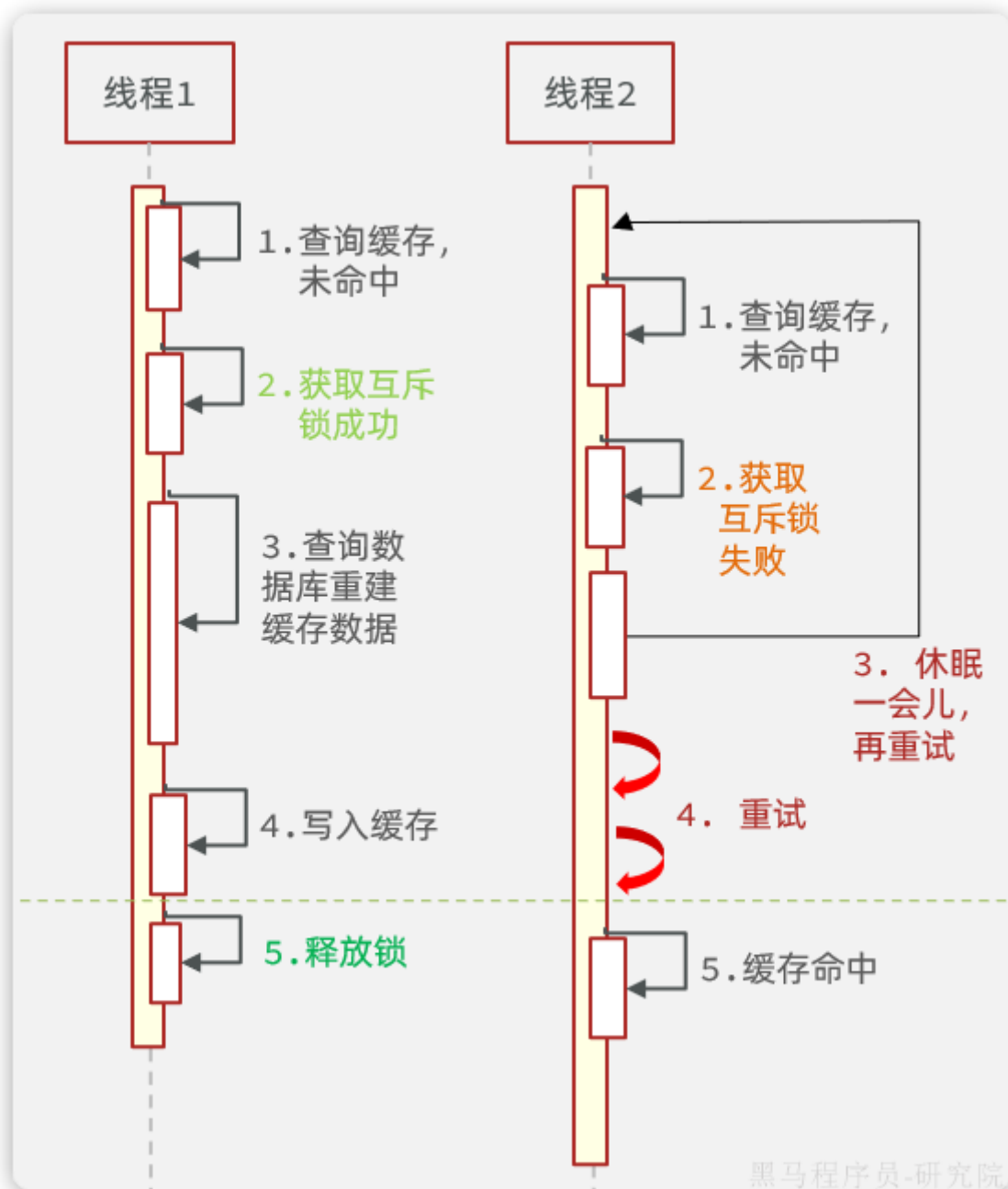
如上图所示：

- 线程1发现缓存未命中，准备查询数据库，重建缓存，但是因为数据比较复杂，导致查询数据库耗时较久
- 在这个过程中，一下次来了3个新的线程，就都会发现缓存未命中，都去查询数据库
- 数据库压力激增

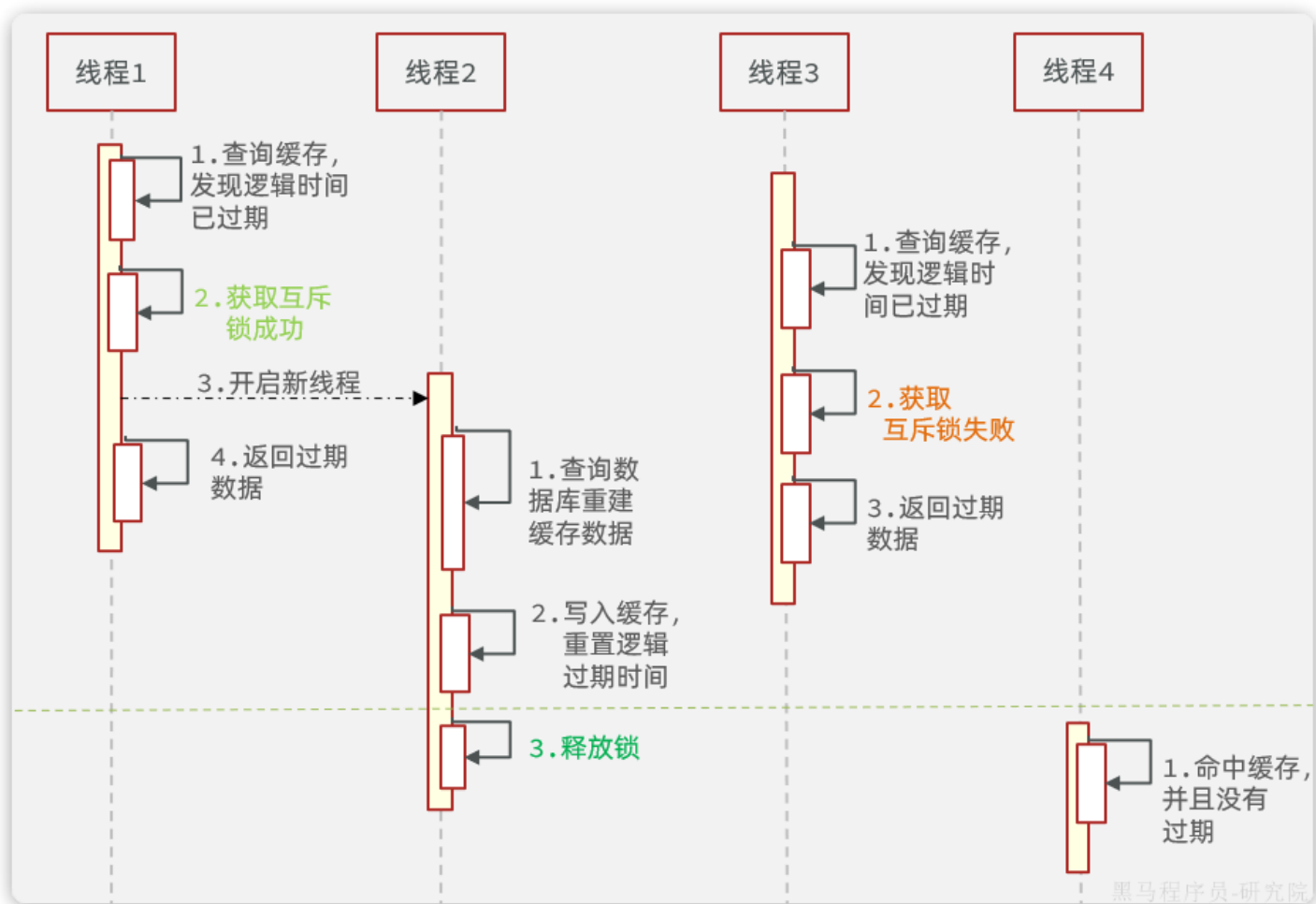
常见的解决方案有两种：

- 互斥锁：给重建缓存逻辑加锁，避免多线程同时指向
- 逻辑过期：热点key不要设置过期时间，在活动结束后手动删除。

基于互斥锁的方案如图：



逻辑过期的思路如图：



6.5.面试总结

✅ 面试题：如何保证缓存的双写一致性？

答：缓存的双写一致性很难保证强一致，只能尽可能降低不一致的概率，确保最终一致。我们项目中采用的是 `Cache Aside` 模式。简单来说，就是在更新数据库之后删除缓存；在查询时先查询缓存，如果未命中则查询数据库并写入缓存。同时我们会给缓存设置过期时间作为兜底方案，如果真的出现了不一致的情况，也可以通过缓存过期来保证最终一致。

追问：为什么不采用延迟双删机制？

答：延迟双删的第一次删除并没有实际意义，第二次采用延迟删除主要是解决数据库主从同步的延迟问题，我认为这是数据库主从的一致性问题，与缓存同步无关。既然主节点数据已经更新，Redis的缓存理应更新。而且延迟双删会增加缓存业务复杂度，也没能完全避免缓存一致性问题，投入回报比太低。

✅ 面试题：如何解决缓存穿透问题？

答：缓存穿透也可以说是穿透攻击，具体来说是因为请求访问到了数据库不存在的值，这样缓存无法命中，必然访问数据库。如果高并发的访问这样的接口，会给数据库带来巨大压力。

我们项目中都是基于布隆过滤器来解决缓存穿透问题的，当缓存未命中时基于布隆过滤器判断数据是否存在。如果不存在则不去访问数据库。

当然，也可以使用缓存空值的方式解决，不过这种方案比较浪费内存。

✅ 面试题：如何解决缓存雪崩问题？

答：缓存雪崩的常见原因有两个，第一是因为大量key同时过期。针对这个问题我们可以给缓存key设置不同的TTL值，避免key同时过期。

第二个原因是Redis宕机导致缓存不可用。针对这个问题我们可以利用集群提高Redis的可用性。也可以添加多级缓存，当Redis宕机时还有本地缓存可用。

✅ 面试题：如何解决缓存击穿问题？

答：缓存击穿往往是由热点Key引起的，当热点Key过期时，大量请求涌入同时查询，发现缓存未命中都会去访问数据库，导致数据库压力激增。解决这个问题的主要思路就是避免多线程并发去重建缓存，因此方案有两种。

第一种是基于互斥锁，当发现缓存未命中时需要先获取互斥锁，再重建缓存，缓存重建完成释放锁。这样就可以保证缓存重建同一时刻只会有一个线程执行。不过这种做法会导致缓存重建时性能下降严重。

第二种是基于逻辑过期，也就是不给热点Key设置过期时间，而是给数据添加一个过期时间的字段。这样热点Key就不会过期，缓存中永远有数据。

查询到数据时基于其中的过期时间判断key是否过期，如果过期开启独立新线程异步的重建缓存，而查询请求先返回旧数据即可。当然，这个过程也要加互斥锁，但由于重建缓存是异步的，而且获取锁失败也无需等待，而是返回旧数据，这样性能几乎不受影响。

需要注意的是，无论是采用哪种方式，在获取互斥锁后一定要再次判断缓存是否命中，做 dubbo check. 因为当你获取锁成功时，可能是在你之前有其它线程已经重建缓存了。