

redis

20-

问题

1.

redis单线程

为啥redis单线程模型也能效率这么高？

- 1) 纯内存操作
- 2) 核心是基于非阻塞的IO多路复用机制
- 3) 单线程反而避免了多线程的频繁上下文切换问题（百度）

redis数据类型

- (1)string
- (2)hash 存对象
- (3)list
- (4)set

直接基于set将系统里需要去重的数据扔进去，自动就给去重了，如果你需要对一些数据进行快速的全局去重，你当然也可以基于jvm内存里的HashSet进行去重，但是如果你的某个系统部署在多台机器上呢？

得基于redis进行全局的set去重

可以基于set玩儿交集、并集、差集的操作，比如交集吧，可以把两个人的粉丝列表整一个交集，看看俩人的共同好友是谁

- (5)sorted set
去重并排序

redis的过期策略

(1) 定期删除+惰性删除

定期删除: **redis**会定期随机抽取一部分设置了过期时间的数据, 检查是否过期, 过期则删除掉

惰性删除: 在查询时**redis**先会查该**key**对应的数据是否已过期, 过期的话就删除掉。

两者结合保证不会查询出已经过期的数据, 但可能因为大量过期时间没有被删除掉导致内存占满

(2) 淘汰机制

1) **noeviction**: 当内存不足以容纳新写入数据时, 新写入操作会报错, 这个一般没人用吧, 实在是太恶心了

2) **allkeys-lru**: 当内存不足以容纳新写入数据时, 在键空间中, 移除最近最少使用的**key** (这个是最常用的)

3) **allkeys-random**: 当内存不足以容纳新写入数据时, 在键空间中, 随机移除某个**key**, 这个一般没人用吧, 为啥要随机, 肯定是把最近最少使用的**key**给干掉啊

4) **volatile-lru**: 当内存不足以容纳新写入数据时, 在设置了过期时间的键空间中, 移除最近最少使用的**key** (这个一般不太合适)

5) **volatile-random**: 当内存不足以容纳新写入数据时, 在设置了过期时间的键空间中, 随机移除某个**key**

6) **volatile-ttl**: 当内存不足以容纳新写入数据时, 在设置了过期时间的键空间中, 有更早过期时间的**key**优先移除

手写一个LRU算法

```
public class LRUCache<K, V> extends LinkedHashMap<K, V> {

    private final int CACHE_SIZE;

    // 这里就是传递进来最多能缓存多少数据
    public LRUCache(int cacheSize) {
        super((int) Math.ceil(cacheSize / 0.75) + 1, 0.75f, true); // 这块就是设置
        // 一个hashmap的初始大小, 同时最后一个true指的是让linkedhashmap按照访问顺序来进行排序, 最近访问
        // 的放在头, 最老访问的就在尾
        CACHE_SIZE = cacheSize;
    }

    @Override
    protected boolean removeEldestEntry(Map.Entry eldest) {
        return size() > CACHE_SIZE; // 这个意思就是说当map中的数据量大于指定的缓存个数的时候, 就自动删除最老的数据
    }
}
```

主从架构实现了读写分离, **master**机器(可以进行读写操作)负责写入数据, 并同步到多台**salve**机器(只能读, 不能写)上, 读的操作都放在**salve**机器上, 当需要水平扩容时只需在增加**salve**机器

redis replication的核心机制

(1) **redis**采用异步方式复制数据到**slave**节点, 不过**redis 2.8**开始, **slave node**会周期性地确认自己每次复制的数据量

(2) 一个**master node**是可以配置多个**slave node**的

(3) **slave node**也可以连接其他的**slave node**

(4) **slave node**做复制的时候, 是不会**block master node**的正常工作的

(5) **slave node**在做复制的时候, 也不会**block**对自己的查询操作, 它会用旧的数据集来提供服务; 但是复制完成的时候, 需要删除旧数据集, 加载新数据集, 这个时候就会暂停对外服务了

(6) **slave node**主要用来进行横向扩容, 做读写分离, 扩容的**slave node**可以提高读的吞吐量

master持久化对于主从架构的安全保障的意义

如果采用了主从架构，那么建议必须开启master node的持久化！

不建议用slave node作为master node的数据热备，因为那样的话，如果你关掉master的持久化，可能在master宕机重启的时候数据是空的，然后可能一经过复制，slave node数据也丢了

master -> RDB和AOF都关闭了 -> 全部在内存中

master宕机，重启，是没有本地数据可以恢复的，然后就会直接认为自己IDB数据是空的

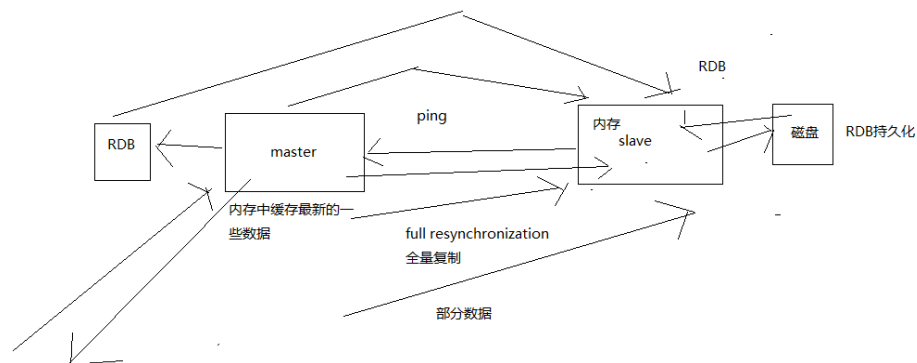
master就会将空的数据集同步到slave上去，所有slave的数据全部清空

100%的数据丢失

master节点，必须要使用持久化机制

第二个，master的各种备份方案，要不要做，万一说本地的所有文件丢失了；从备份中挑选一份rdb去恢复master；这样才能确保master启动的时候，是有数据的

即使采用了后续讲解的高可用机制，slave node可以自动接管master node，但是也可能sentinel还没有检测到master failure，master node就自动重启了，还是可能导致上面的所有slave node数据清空故障



1、主从架构的核心原理

当启动一个slave node的时候，它会发送一个PSYNC命令给master node

如果这是slave node重新连接master node，那么master node仅仅会复制给slave部分缺少的数据；否则如果是slave node第一次连接master node，那么会触发一次full resynchronization

开始full resynchronization的时候，master会启动一个后台线程，开始生成一份RDB快照文件，同时还会将从客户端收到的所有写命令缓存在内存中。RDB文件生成完毕之后，master会将这个RDB发送给slave，slave会先写入本地磁盘，然后再从本地磁盘加载到内存中。然后master会将内存中缓存的写命令发送给slave，slave也会同步这些数据。

slave node如果跟master node有网络故障，断开了连接，会自动重连。master如果发现有多slave node都来重新连接，仅仅会启动一个rdb save操作，用一份数据服务所有slave node。

2、主从复制的断点续传

从redis 2.8开始，就支持主从复制的断点续传，如果主从复制过程中，网络连接断掉了，那么可以接着上次复制的地方，继续复制下去，而不是从头开始复制一份

master node会在内存中常见一个backlog，master和slave都会保存一个replica offset还有一个master id，offset就是保存在backlog中的。如果master和slave网络连接断掉了，slave会让master从上次的replica offset开始继续复制

但是如果没有找到对应的offset，那么就会执行一次resynchronization

3、无磁盘化复制

在.conf文件中配置这两个属性

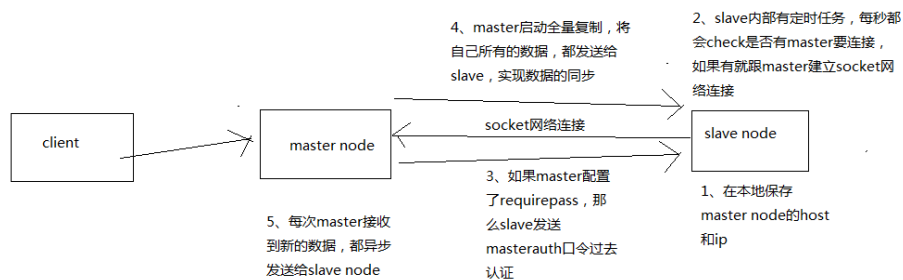
master在内存中直接创建rdb，然后发送给slave，不会在自己本地落地磁盘了

repl-diskless-sync

repl-diskless-sync-delay，等待一定时长再开始复制，因为要等更多slave重新连接过来

4、过期key处理

slave不会过期key，只会等待master过期key。如果master过期了一个key，或者通过LRU淘汰了一个key，那么会模拟一条del命令发送给slave。



1、复制的完整流程

(1) slave node 启动，仅仅保存 master node 的信息，包括 master node 的 host 和 ip，但是复制流程没开始

master host 和 ip 是从哪儿来的，redis.conf 里面的 slaveof 配置的

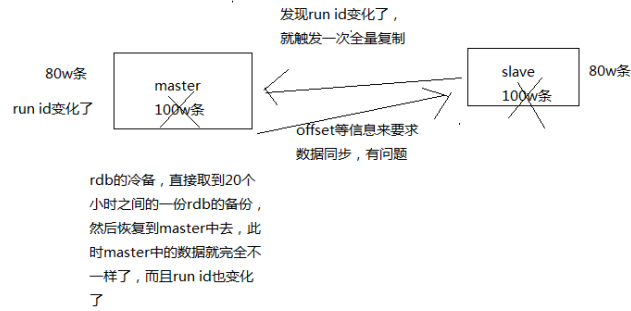
(2) slave node 内部有个定时任务，每秒检查是否有新的 master node 要连接和复制，如果发现，就跟 master node 建立 socket 网络连接

(3) slave node 发送 ping 命令给 master node

(4) 口令认证，如果 master 设置了 requirepass，那么 slave node 必须发送 masterauth 的口令过去进行认证

(5) master node 第一次执行全量复制，将所有数据发给 slave node

(6) master node 后续持续将写命令，异步复制给 slave node



2、数据同步相关的核心机制

指的就是第一次slave连接master的时候, 执行的全量复制, 那个过程里面你的一些细节的机制

(1) master和slave都会维护一个offset

master会在自身不断累加offset, slave也会在自身不断累加offset

slave每秒都会上报自己的offset给master, 同时master也会保存每个slave的offset

这个倒不是说特定就用在全量复制的, 主要是master和slave都要知道各自的数据的offset, 才能知道互相之间的数据不一致的情况

(2) backlog

master node有一个backlog, 默认是1MB大小

master node给slave node复制数据时, 也会将数据在backlog中同步写一份

backlog主要是用来做全量复制中断候的增量复制的

(3) master run id

(23-05视频 06: 56)

info server, 可以看到master run id(在redis的启动路径下)

如果根据host+ip定位master node, 是不靠谱的, 如果master node重启或者数据出现了变化, 那么slave node应该根据不同的run id区分, run id不同就做全量复制

如果需要不更改run id重启redis, 可以使用redis-cli debug reload命令

(4) psync

从节点使用psync从master node进行复制, psync runid offset

master node会根据自身的情况返回响应信息, 可能是FULLRESYNC runid offset触发全量复制, 可能是CONTINUE触发增量复制

3、全量复制

(1) master执行bgsave, 在本地生成一份rdb快照文件

(2) master node将rdb快照文件发送给slave node, 如果rdb复制时间超过60秒(repl-timeout), 那么slave node就会认为复制失败, 可以适当调大这个参数

(3) 对于千兆网卡的机器, 一般每秒传输100MB, 6G文件, 很可能超过60s

(4) master node在生成rdb时, 会将所有新的写命令缓存在内存中, 在slave node保存了rdb之后, 再将新的写命令复制给slave node

(5) client-output-buffer-limit slave 256MB 64MB 60, 如果在复制期间, 内存缓冲区持续消耗超过64MB, 或者一次性超过256MB, 那么停止复制, 复制失败

(6) slave node接收到rdb之后, 清空自己的旧数据, 然后重新加载rdb到自己的内存中, 同时基于旧的数据版本对外提供服务

(7) 如果slave node开启了AOF, 那么会立即执行BGREWRITEAOF, 重写AOF

rdb生成、rdb通过网络拷贝、slave旧数据的清理、slave aof rewrite, 很耗费时间

如果复制的数据量在4G~6G之间, 那么很可能全量复制时间消耗到1分半到2分钟

4、增量复制

(1) 如果全量复制过程中, master-slave网络连接断掉, 那么salve重新连接master时, 会触发增量复制

(2) master直接从自己的backlog中获取部分丢失的数据, 发送给slave node, 默认backlog就是1MB

(3) msater就是根据slave发送的psync中的offset来从backlog中获取数据的

5、heartbeat

主从节点互相都会发送heartbeat信息

master默认每隔10秒发送一次heartbeat, salve node每隔1秒发送一个heartbeat

6、异步复制

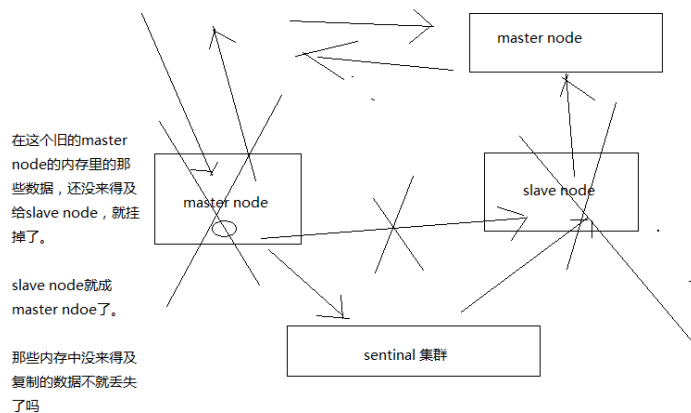
master每次接收到写命令之后, 现在内部写入数据, 然后异步发送给slave node

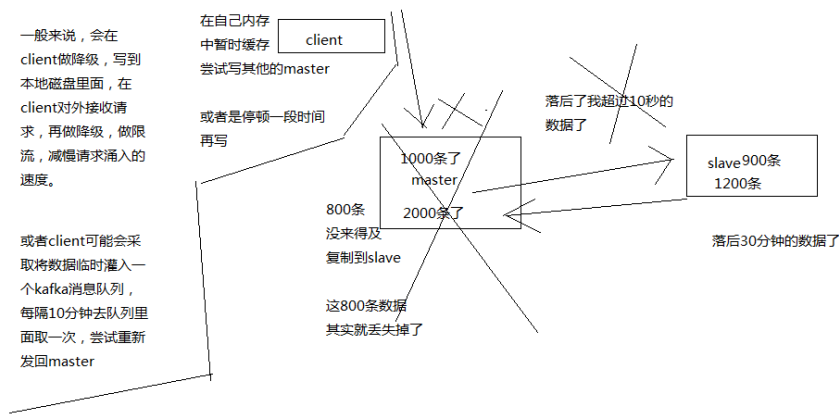
redis的高可用

当一个redis的master宕机, 会通过哨兵模式检测到, 然后将一个salve主从切换为新的master, 进行读写操作, 保证缓存的正常运行

哨兵模式

通过哨兵来监控master node的运行状况, 如果哨兵判断了master node是故障的, 就进行故障转移, 推选出新的master, 哨兵也是分布式的, 保证哨兵模式的高可用, 哨兵节点必须三个以上, 因为推选模式是半数以上的哨兵认为master是故障的才会进行故障转移





主从的异步复制导致的数据丢失的原因

因为master -> slave的复制是异步的，所以可能有部分数据还没复制到slave，master就宕机了，此时这部分数据就丢失了

（原来的master由于网络原因不能与slave进行数据异步复制了，但master还在正常的写入的操作，导致新写入的数据在slave上没有备份，此时，master发生宕机，这部分数据就会丢失）

降低数据丢失的措施

有了min-slaves-max-lag这个配置，就可以确保说，一旦slave复制数据和ack延时太长，就认为可能master宕机后损失的数据太多了，那么就拒绝写请求，这样可以把master宕机时由于部分数据未同步到slave导致的数据丢失降低的可控范围内

min-slaves-max-lag 10

要求至少有1个slave，数据复制和同步的延迟不能超过10秒

如果说一旦所有的slave，数据复制和同步的延迟都超过了10秒钟，那么这个时候，master就不会再接收任何请求了

脑裂导致的数据丢失

脑裂，也就是说，某个master所在机器突然脱离了正常的网络，跟其他slave机器不能连接，但是实际上master还运行着

此时哨兵可能就会认为master宕机了，然后开启选举，将其他slave切换成了master

这个时候，集群里就会有二个master，也就是所谓的脑裂

此时虽然某个slave被切换成了master，但是可能client还没来得及切换到新的master，还继续写向旧master的数据可能也丢失了

因此旧master再次恢复的时候，会被作为一个slave挂到新的master上去，自己的数据会清空，重新从新的master复制数据

2、解决异步复制和脑裂导致的数据丢失

min-slaves-to-write 1

min-slaves-max-lag 10

要求至少有1个slave，数据复制和同步的延迟不能超过10秒

如果说一旦所有的slave，数据复制和同步的延迟都超过了10秒钟，那么这个时候，master就不会再接收任何请求了

上面二个配置可以减少异步复制和脑裂导致的数据丢失

（1）减少异步复制的数据丢失

有了`min-slaves-max-lag`这个配置，就可以确保说，一旦`slave`复制数据和`ack`延时太长，就认为可能`master`宕机后损失的数据太多了，那么就拒绝写请求，这样可以把`master`宕机时由于部分数据未同步到`slave`导致的数据丢失降低的可控范围内

（2）减少脑裂的数据丢失

如果一个`master`出现了脑裂，跟其他`slave`丢了连接，那么上面两个配置可以确保说，如果不能继续给指定数量的`slave`发送数据，而且`slave`超过10秒没有给自己`ack`消息，那么就直接拒绝客户端的写请求

这样脑裂后的旧`master`就不会接受`client`的新数据，也就避免了数据丢失

上面的配置就确保了，如果跟任何一个`slave`丢了连接，在10秒后发现没有`slave`给自己`ack`，那么就拒绝新的写请求

因此在脑裂场景下，最多就丢失10秒的数据

哨兵模式的机制

1、`sdown`和`odown`转换机制

`sdown`和`odown`两种失败状态

`sdown`是主观宕机，就一个哨兵如果自己觉得一个`master`宕机了，那么就是主观宕机

`odown`是客观宕机，如果`quorum`数量的哨兵都觉得一个`master`宕机了，那么就是客观宕机

`sdown`达成的条件很简单，如果一个哨兵`ping`一个`master`，超过了`is-master-down-after-milliseconds`指定的毫秒数之后，就主观认为`master`宕机

`sdown`到`odown`转换的条件很简单，如果一个哨兵在指定时间内，收到了`quorum`指定数量的其他哨兵也认为那个`master`是`sdown`了，那么就认为是`odown`了，客观认为`master`宕机

2、哨兵集群的自动发现机制

哨兵互相之间的发现，是通过`redis`的`pub/sub`系统实现的，每个哨兵都会往`__sentinel__:hello`这个`channel`里发送一个消息，这时候所有其他哨兵都可以消费到这个消息，并感知到其他哨兵的存在

每隔两秒钟，每个哨兵都会往自己监控的某个`master+slaves`对应的`__sentinel__:hello channel`里发送一个消息，内容是自己的`host`、`ip`和`runid`还有对这个`master`的监控配置

每个哨兵也会去监听自己监控的每个`master+slaves`对应的`__sentinel__:hello channel`，然后去感知到同样在监听这个`master+slaves`的其他哨兵的存在

每个哨兵还会跟其他哨兵交换对`master`的监控配置，互相进行监控配置的同步

3、`slave`配置的自动纠正

哨兵会负责自动纠正`slave`的一些配置，比如`slave`如果要成为潜在的`master`候选人，哨兵会确保`slave`在复制现有`master`的数据；如果`slave`连接到了一个错误的`master`上，比如故障转移之后，那么哨兵会确保它们连接到正确的`master`上

4、slave->master选举算法

如果一个master被认为odown了，而且majority哨兵都允许了主备切换，那么某个哨兵就会执行主备切换操作，此时首先要选举一个slave来

会考虑slave的一些信息

- (1) 跟master断开连接的时长
- (2) slave优先级
- (3) 复制offset
- (4) run id

如果一个slave跟master断开连接已经超过了down-after-milliseconds的10倍，外加master宕机的时长，那么slave就被认为不适合选举为master

$(\text{down-after-milliseconds} * 10) + \text{milliseconds_since_master_is_in_SDOWN_state}$

接下来会对slave进行排序

- (1) 按照slave优先级进行排序，slave priority越低，优先级就越高
- (2) 如果slave priority相同，那么看replica offset，哪个slave复制了越多的数据，offset越靠后，优先级就越高
- (3) 如果上面两个条件都相同，那么选择一个run id比较小的那个slave

5、quorum和majority

每次一个哨兵要做主备切换，首先需要quorum数量的哨兵认为odown，然后选举出一个哨兵来做切换，这个哨兵还得得到majority哨兵的授权，才能正式执行切换

如果 $\text{quorum} < \text{majority}$ ，比如5个哨兵，majority就是3，quorum设置为2，那么就3个哨兵授权就可以执行切换

但是如果 $\text{quorum} \geq \text{majority}$ ，那么必须quorum数量的哨兵都授权，比如5个哨兵，quorum是5，那么必须5个哨兵都同意授权，才能执行切换

6、configuration epoch

哨兵会对一套redis master+slave进行监控，有相应的监控的配置

执行切换的那个哨兵，会从要切换到的新master (slave->master) 那里得到一个configuration epoch，这就是一个version号，每次切换的version号都必须是唯一的

如果第一个选举出的哨兵切换失败了，那么其他哨兵，会等待failover-timeout时间，然后接替继续执行切换，此时会重新获取一个新的configuration epoch，作为新的version号

7、configuraiton传播

哨兵完成切换之后，会在自己本地更新生成最新的master配置，然后同步给其他的哨兵，就是通过之前说的pub/sub消息机制

这里之前的version号就很重要了，因为各种消息都是通过一个channel去发布和监听的，所以一个哨兵完成一次新的切换之后，新的master配置是跟着新的version号的

其他的哨兵都是根据版本号的大小来更新自己的master配置的

