Redis篇书签入口

1. Redis SDS

Redis字符串底层实现

Redis的字符串采用SDS实现.

SDS定义

```
struct sbdhdr {

    //记录buf数组中已使用字节的数量,等于SDS中字符串的长度
    int len;

    //记录buf数组中未使用字节的数量
    int free;

    //字节数组,用于保存字符串
    char buf[];

}
```

- SDS遵循C字符串以'\0'结尾的惯例,字节数组会保存这个'\0',但不会计算在长度中,只是会为这个'\0'多分配空间.

SDS与C字符串的区别

获取字符串长度的复杂度

SDS会记录字符串长度信息,因此获取字符串长度的时间复杂度为O(1).C字符串不会记录长度信息,获取长度需要遍历一遍字符串,时间复杂度为O(n)

缓冲区溢出问题

C字符串对于内存是否足够不会进行判断,导致有可能在内存不足情况下对一个字符串进行拼接之后,溢出到其他内存空间中.而SDS会先判断空间是否足够,如果不足会进行扩容,保证缓冲区的安全.

内存重分配次数

C字符串每次拼接字符串时都会重新分配内存来容纳新的字符串,而截断字符串会将额外的内存空间释放.而SDS只有在无法容纳新的字符串时才会开辟新的内存空间,并且会进行预分配,对于多余的空间则交由程序来主动释放

表 2-1　C 字符串和 SDS 之间的区别

C 字符串	SDS
获取字符串长度的复杂度为 $O(N)$	获取字符串长度的复杂度为 $O(1)$
API 是不安全的，可能会造成缓冲区溢出	API 是安全的，不会造成缓冲区溢出
修改字符串长度 N 次必然需要执行 N 次内存重分配	修改字符串长度 N 次最多需要执行 N 次内存重分配
只能保存文本数据	可以保存文本或者二进制数据
可以使用所有 <string.h> 库中的函数	可以使用一部分 <string.h> 库中的函数

空间预分配

如果SDS不足以容纳拼接后的新的字符串时,那么将会分配一块内存,让其足以容纳拼接后的字符串,并且最再多分配一倍的空间或者1MB(取决于当前字符串大小),再加个空字符串的1字节空间.

惰性回收

对于SDS多余的空间则交由程序来主动释放

二进制安全

由于SDS是通过长度来判断字符串的结束位置,因此可以用来存储二进制数据,而C字符串遇到'\0'则会认为是一个字符串的结束.

兼容部分C字符串函数

参考资料

Redis设计与实现第2章

2.Redis字典

哈希键的底层实现

当一个哈希键包含的键值对比较多,又或者键值对是比较长的字符串时,会使用字典作为哈希键的底层实现.

Redis字典数据结构

字典

```
typedef struct dict {

    // 类型特定函数
    dictType *type;

    // 私有数据
    void *privdata;

    // 哈希表
    dictht ht[2];

    // rehash 索引
    // 当 rehash 不在进行时，值为 -1
    in trehashidx; /* rehashing not in progress if rehashidx == -1 */

} dict;
```

哈希表

```
typedef struct dictht {

    // 哈希表数组
    dictEntry **table;

    // 哈希表大小
    unsigned long size;

    // 哈希表大小掩码，用于计算索引值
    // 总是等于 size-1
    unsigned long sizemask;

    // 该哈希表已有节点的数量
    unsigned long used;

} dictht;
```

哈希表节点

```
typedef struct dictEntry {

    // 键
    void *key;

    // 值
    union{
        void *val;
        uint64_tu64;
        int64_ts64;
    } v;

    // 指向下个哈希表节点，形成链表
    struct dictEntry *next;
} dictEntry;
```

原理

1. 哈希表由节点数组构成,每个节点都是一个链表,而字典拥有 两个哈希表数组,其中一个哈希表用于重哈希
2. 当往字典里面添加键值对时,先根据哈希值%哈希大小获得节点应该存放的数组位置,如果那个位置已经有了节点,那么节点之间通过链表连接
3. 当哈希表中的节点数量过多或者过少,也就是负载因子过大或过小时,哈希表会进行rehash操作,相应的增大或减小哈希表的大小.负载因子的大小等于哈希表节点数量/哈希表大小,当哈希表的负载因子大于等于1时,会进行扩容操作,负载因子小于0.1时,会进行收缩操作.rehash需要用到另外一个哈希表,即将保存在当前哈希表的节点rehash到另外一个哈希表中,然后清空当前哈希表
4. rehash并不是一次性完成,而是渐进式的.在这个过程中,会同时使用两个哈希表,每次执行特定的操作时,会将节点重哈希到新的哈希表中,并使用rehashIndex记录目前正在处理的数组节点索引.并且后续对哈希表的新增操作只会在新的哈希表中进行,删除,更新和查找会在两个哈希表中进行,随着rehash的不停操作,最终原来的哈希表会变成空表,哈希完成之后,rehashIndex变为-1.

参考资料

Redis设计与实现第4章 字典

3 Redis分布式锁原理

分布式锁需要具备的三个基础属性

1. 安全属性:互斥.在任何时间,最多只有一个客户端可以持有锁
2. 活跃属性A:不能够产生死锁.最终总是可能获取到锁,即使持有锁的其它客户端宕机了

3. 活跃属性B:错误容忍.只要大多数的Redis节点处于正常运行状态,客户端可以获取和释放锁

单实例上的实现

```
SET resource_name my_random_value NX PX 30000
```

获取锁:使用一个key,和一个唯一标识来设置键,并且只有当key不存在的时候(NX)才能执行成功,这个唯一标识my_random_value必须在所有的客户端上都是唯一的值,以保证释放锁的客户端是获取锁的那个客户端.

释放锁:当释放锁时,使用Lua脚本加上这个唯一标识.告诉Redis只有当key存在并且值是这个唯一标识时,再释放锁.一个Lua脚本的例子如下:

```
if redis.call("get",KEYS[1]) == ARGV[1] then
    return redis.call("del",KEYS[1])
else
    return 0
end
```

如果没有使用唯一标识,那么有可能当一个客户端获取到锁,执行一段业务逻辑,但还没执行完锁已经过期了,此时锁被其他客户端获取到,那么因为没有使用唯一标识,一个客户端释放了别的客户端获取到的锁.

红锁算法

由于单实例可能存在单点故障,为了保证高可用性,需要使用多个独立的Redis实例来实现分布式锁.每个实例上获取分布式锁的模式和单实例的模式是一样的,只不过在这个基础上加了其他一些步骤.

假如有N个独立的Redis实例,获取锁的步骤如下:

1. 获取当前时间
2. 尝试使用同样的key和唯一标识在所有的N个实例上顺序的获取锁.当在每一个实例上获取锁时,使用一个相对于总体的锁释放时间来得很小的超时时间,例如锁的过期时间是10秒,那么超时时间可以是5-50毫秒,一旦获取锁超时,那么就赶紧去下一个实例获取锁.这么做是防止长时间在某个Redis实例上发生长时间的阻塞,如果一个实例不可用,我们应该尽可能的去下一个实例上获取锁.
3. 通过当前时间减去步骤1锁获取到的时间,客户端可以计算为了获取锁使用了多少时间,如果能在半数以上的Redis实例上获取到锁,并且获取锁所使用的时间小于锁的有效时间,那么就认为获取到了锁
4. 如果锁获取到了,那么锁的真实有效时间被认为是锁的过期时间减去获取锁所使用的时间
5. 如果客户端未能获取到锁,那么应该尝试对所有的实例进行锁的释放,即使在某个实例上并没有获取到锁

开启持久化配置

每个实例需要使用持久性配置fsync=always,否则某个实例宕机后恢复,那么有可能导致一个分布式锁被两个客户端获取到.

延长锁时间

如果在锁过期之前,客户端还未能处理完逻辑,那么可能需要考虑去在获取到Redis锁的实例上去延长锁的时间,具体方式可以通过Lua脚本来完成,如果key和value都是之前获取锁的值,那么就延长时间.如果没有在半数以上的实例上延长成功,那么就视为不成功,更具体的步骤类似于红锁算法中获取锁的步骤.

参考资料

- https://redis.io/docs/manual/patterns/distributed-locks/#analysis-of-redlock

4 为什么Redis使用16384个槽

为什么Redis使用16384个槽

- 正常的心跳包携带了一个节点的配置信息,其中也带有槽的分配信息,如果是16384个槽的话将使用2K的空间,而是用65535个槽将使用8K的空间.
- 由于设计权衡,Redis集群的大小通常也不可能超过1000个,槽的数量是合理的.

因此16834个槽是一个比较合理的数量让每一个集群都拥有足够的槽.

参考资料

https://github.com/redis/redis/issues/2576

5 Redis为什么这么快

Redis为什么这么快

- Redis将数据存放在内存中
- IO多路复用机制加上单线程处理事件
- 高效的底层数据结构

参考资料

- https://blog.bytebytego.com/p/why-is-redis-so-fast

6 Redis IO多路复用

Redis IO多路复用原理

1. Redis使用IO多路复用程序监听多个套接字
2. 一旦监听到可用的文件描述符之后,将把套接字放入一个队列中
3. IO多路复用程序以单线程的方式从队列中获取套接字,将其传送给文件事件分派器进行处理
4. 文件事件分派器分派给对应的事件处理器进行处理
5. 当处理完毕之后,继续执行3步骤

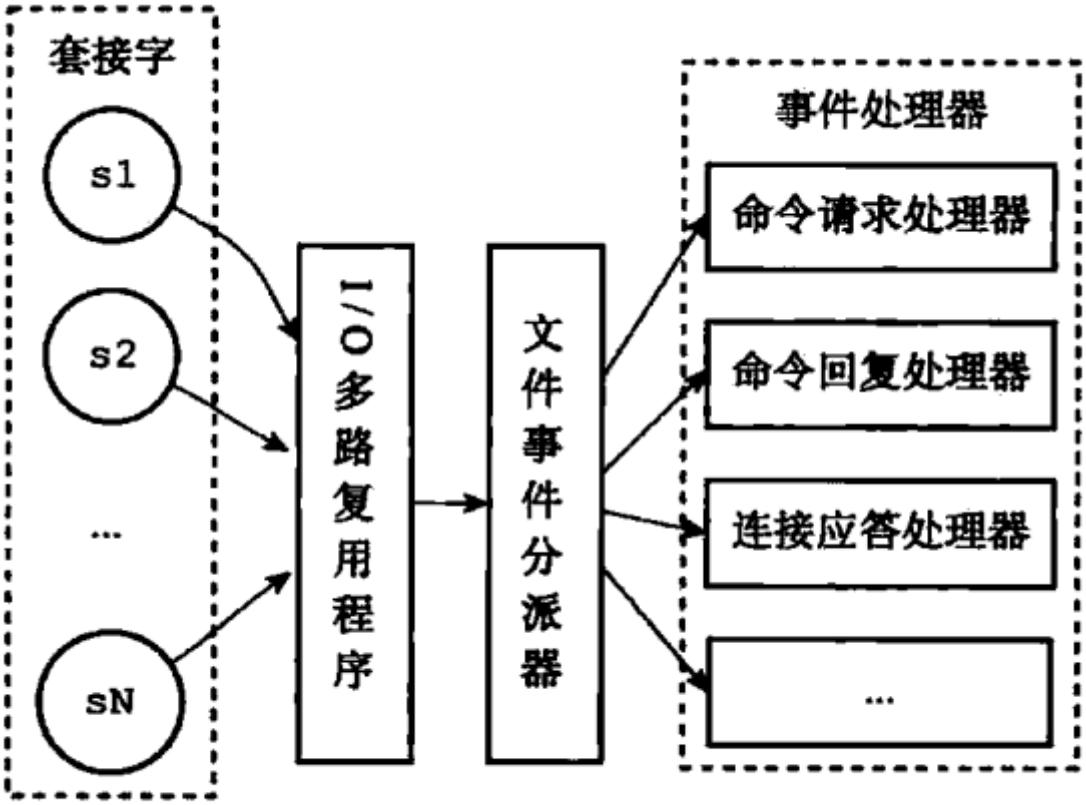

图 12-1　文件事件处理器的四个组成部分

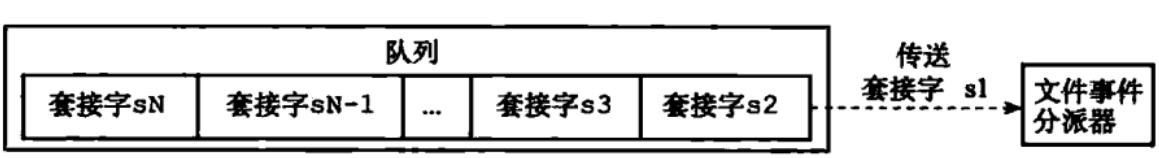

图 12-2　I/O 多路复用程序通过队列向文件事件分派器传送套接字

参考资料

- Redis设计与实现 12.1 文件事件

7 Redis和数据库的一致性

如何保证Redis和数据库的一致性

方案一:延时双删

读操作

对于读操作,如果数据存在于Redis中,那么直接从Redis中返回数据.如果不存在于Redis,那么从数据库中获取数据,然后存储到Redis中,向客户端返回数据.

写操作(insert,update,delete)

对于写操作,使用以下步骤

1. 删除Redis中的key
2. insert,update,delete数据库中的数据
3. 睡眠一会(例如500ms,需要根据具体应用进行评估)
4. 再次删除Redis中的key

第一次删除Redis中的key之后,在没有对数据库中的数据进行操作之前,有可能有其他请求直接访问了数据库的老数据,并且将其更新到Redis中,这个时候通过睡眠500ms,可以使得这些请求全部执行完成,最后再删除Redis中的key后,可以消除掉这些请求带来的脏数据的影响.

方案二:Write Behind变体-通过Canal订阅Binlog

可以通过Canal订阅MySql的binlog并且将数据的写操作同步到Redis.这种方式比较成熟可靠,目前是最好的方案.

参考资料

https://yunpengn.github.io/blog/2019/05/04/consistent-redis-sql/

8 Redis缓存驱逐策略

常见的缓存驱逐策略有哪些

- LRU(Least Recent Use).移除最近最少访问的元素.比如实现一个有界栈,元素每访问一次就将其放入栈头部,或者新元素也放入栈头部, 然后移除掉栈底元素.
- LFU(Least Frequent Use).移除最不经常使用的元素.比如给某个元素记录访问次数,如果哪个元素最不经常访问,则将其移除。
- Window TinyLFU (W-TinyLFU).考虑最经常使用以及最近使用两个维度来进行元素的驱逐，比如说使用两者的权重来考虑
- TTL.到达缓存时间后就被清除,这尽可能确保缓存中的数据不是脏数据

Redis具体有哪些缓存驱逐策略

- allkeys-lru.移除最近最少访问的元素,设置了过期时间和没有设置过期时间的元素都考虑
- volatile-lru.移除最少使用的元素,只考虑处理设置了过期时间的元素
- allkeys-lfu. 移除最不经常使用的元素,设置了过期时间和没有设置过期时间的元素都考虑
- volatile-lfu. 移除最不经常使用的元素,只考虑处理设置了过期时间的元素
- volatile-ttl.移除最快过期的元素
- 无驱逐策略.当内存不足时,执行写命令将提示错误信息

参考资料

https://redis.com/blog/cache-eviction-strategies/

9 Redis过期键删除策略

Redis过期键删除策略

Redis使用的是惰性删除和定期删除两种策略.

- 惰性删除.当redis收到获取键的命令时,Redis会调用expireIfNeeded函数判断键是否已经过期,如果过期,则删除键.
- 定期删除.Redis在定时任务函数中执行activeExpireCycle函数.定期删除的过程如下:

1. 顺序遍历所有的数据库,每次随机从里面获取N个带有过期时间的键判断是否过期,如果过期,则删除键.在检查键是否过期的时候,会判断是否此次的activeExpireCycle函数已经到达了运行时间限制,如果到达了,则结束此次定期删除函数的执行,此时会记录当前执行到哪个数据库.
2. 下一次执行定期删除函数的时候,会接着上次的数据库继续向下进行遍历.
3. 如果遍历了所有数据库一轮之后,函数会从头开始新一轮的遍历.

参考资料

- Redis设计与实现9.6 Redis的过期键删除策略

10 缓存穿透,缓存击穿,缓存雪崩

什么是缓存穿透

缓存穿透指的是对数据库不存在的key进行访问,此时key无法被缓存,那么大量的请求会直接请求到数据库,可能造成数据库崩溃.

缓存穿透的解决方案

- 针对于频繁访问的key,如果数据库中不存在key,那么可以在redis缓存一个空值,并设置一个合适的过期时间.(过期时间不要过长).
- 针对于不频繁访问的key或者访问的key比较多的情况,先使用布隆过滤器判断是否存在key,如果不存在,则直接返回.如果存在,再访问缓存或者数据库.

什么是缓存击穿

缓存击穿指的是缓存突然失效,导致大量对某个key的请求直接请求到数据库,造成数据库压力

缓存击穿的解决方案

- 这个问题通常发生在高并发环境下.此时需要使用锁来保证只有一个线程可以去访问数据库,然后由这个线程去更新缓存,等缓存被更新了且锁被释放之后,其它线程才可以读取最新缓存的数据.
- 可以使用另外的线程去异步的更新缓存中的热点key,这样热点key就不会过期

什么是缓存雪崩

缓存雪崩指的是大量缓存的数据同时过期或者缓存服务宕机,所有对key的搜索直接访问了数据库,造成数据库的高负载.

缓存雪崩的解决方案

- 使用集群去确保缓存服务的高可用
- 使用熔断器或者限流方案来保证系统还能处理请求
- 将key的过期时间交错开来

参考资料

https://www.pixelstech.net/article/1586522853-What-is-cache-penetration-cache-breakdown-and-cache-avalanche

11 持久化

什么是RDB

RDB是一个二进制文件,用来保存数据库中的数据.当Redis重启时,可以通过RDB文件来还原数据库状态.

RDB的实现原理

- 可以使用SAVE(阻塞服务器进程)或者BGSAVE(不阻塞服务器进程)命令生成RDB文件
- 可以配置自动间隔保存条件来让Redis执行BGSAVE来生成RDB文件,比如数据库经过多少次修改就生成一次RDB文件
- 服务器启动时会自动载入RDB文件恢复数据库状态

什么是AOF

AOF是一个文件,通过保存Redis所执行的写命令来记录数据库状态

AOF的实现原理

- 服务器在执行完一个写命令时,同时也将命令以协议格式追加到AOF缓冲区中
- Redis的事件循环中将AOF缓冲区的数据写入到AOF文件中,这里可以配置appendfsync来表示何时同步到AOF文件中,具体的配置有always,everysec,no
-
  - always:每次写到文件中,刷新缓冲区
  - everysecond:每隔1秒刷新缓冲区
  - no：依靠操作系统的时机刷新缓冲区
- Redis在启动中通过读取AOF文件中的命令就可以恢复数据库状态

参考资料

- Redis设计与实现10.1 RDB文件的创建和载入 10.2 自动间隔性保存
- Redis设计与实现11.1 AOF持久化的实现 11.2 AOF文件的载入和数据库还原

12 Redis Hot key Error

什么是Redis Hot key Error

数据库设计不合理或者流量激增，对hot key进行频繁访问而导致系统过载而产生的错误

如何解决Redis Hot key错误

- 增加内存,升级硬件
- 使用多级缓存
- 将数据分片到不同Redis实例
- 减少key的大小和数量
- 使用合适的Redis驱逐策略

参考资料

https://www.dragonflydb.io/error-solutions/redis-hotkey-problem

13 Redis复制原理

Redis复制的原理

一次完整同步的过程

1. 同步.同步操作使得从服务器的数据库状态与当前主服务器的数据库状态一致
2. 传播.主服务器会接着处理客户端的命令,此时需要将这些命令传播给从服务器,才能使得数据重新到达一致状态

完整重同步和部分重同步

完整重同步指的是从服务器完整的进行一次主服务器上所有数据的复制.部分重同步指的是只对从服务器缺少的数据进行获取.

完整重同步的实现

1. 从服务器发送SYNC命令给主服务器
2. 主服务器执行BGSAVE命令生成一个RDB文件,并使用一个缓冲区记录从生成RDB文件之后的所有写命令
3. 主服务器将RDB文件发送给从服务器,从服务器加载RDB文件,更新到主服务器执行BGSAVE时数据库状态
4. 主服务器将缓冲区中的写命令发送给从服务器,从服务器执行这些写命令,主从数据库达到一致状态

部分重同步的实现

新版Redis复制功能中,使用PSYNC来进行数据的同步,具有两种模式,一种是完整重同步,用于处理初次复制的情况,这种模式和SYNC模式完全一样,另外一种是部分重同步,用于处理断线之后复制的情况

- 部分重同步由三个部分来实现:
- - 复制偏移量.主服务器和从服务器都维护复制偏移量,主服务器传播数据给从服务器后,会将偏移量加上N,从服务器收到数据后,也会将偏移量的值加上N
  - 复制积压缓冲区.复制积压缓冲区由服务器维护的一个固定的FIFO队列实现.当主服务器传播命令时,除了将命令发送给从服务器,也会将命令写入队列中,复制积压缓冲区中的每个字节都会有一个偏移量值.当从服务器断线重新连接到主服务器时,从服务器将会把当前自己的偏移量值发给主服务器,如果偏移量存在于复制积压缓冲区中且从服务器复制的主服务器为本主服务器,那么将执行部分重同步,否则将执行完整重同步.
  - 服务器运行ID.当从服务器首次复制一个主服务器时,主服务器会向从服务器发送自己的服务器标识,从服务器会保存这个标识.当从服务器断线重新去复制主服务器时,会将这个服务器标识发送给主服务器,如果这个标识不是主服务器的标识,那么说明复制的主服务器改变了,只能执行完整重同步操作,否则主服务器根据偏移量是否在复制积压缓冲区来决定执行完整重同步操作或者是部分重同步操作

心跳检测

在命令传播阶段,从服务器会以每秒一次的频率,向主服务器发出REPLCONF ACK OFFSET命令,作用如下:

- 主服务器会记录从服务器上次发送ACK的时间,用lag来表示,用来判断主从服务器之间的连接是否出了问题
- 辅助实现mini-slave配置,如果某个从服务器的lag太大,那么主服务器可以拒绝客户端传输的写命令
- 从服务器在ACK中发送偏移量给主服务器,当主服务器发现从服务器的偏移量不等于自己的偏移量,这个时候可能是发生数据丢失了,那么将会从复制积压缓冲区中获取数据,重新发送给从服务器

参考资料

Redis设计与实现 第15章 复制

14 Redis哨兵

哨兵的作用

哨兵用于监视主服务器及主服务器下的从服务器的情况,当主服务器下线时,对从服务器进行选举,最终将一个从服务器升级为主服务器,来接替下线的主服务器的工作

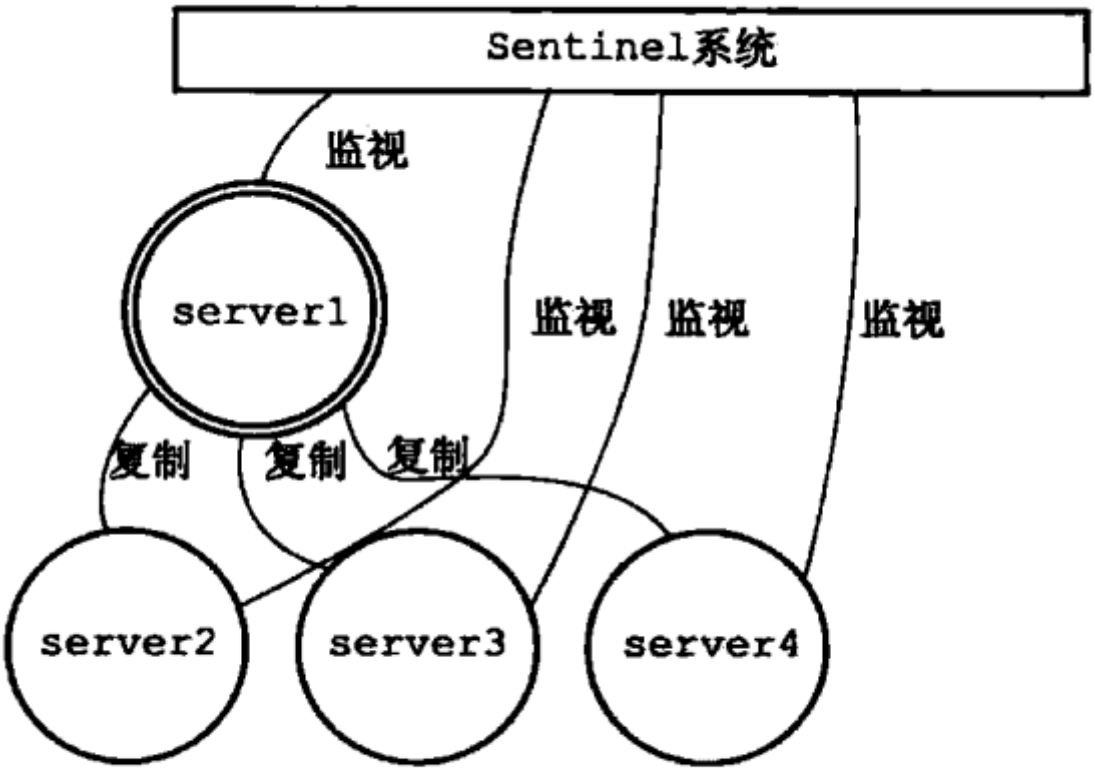

图 16-1　服务器与 Sentinel 系统

哨兵如何发现从服务器和其他哨兵

发现主服务器下的从服务器

哨兵只配置了主服务器的连接信息.哨兵首先会向主服务器创建一个命令连接.每隔10秒,哨兵会向主服务器发送INFO命令,用于获取主服务器的信息.在主服务器返回的信息中,有主服务器本身的信息以及主服务器下所有从服务器的信息,通过这些信息,哨兵就可以发现从服务器.哨兵会创建与从服务器之间的命令连接,通过这个命令连接,哨兵可以主动向从服务器发送INFO命令,进而获取从服务器的最新信息.

发现哨兵

除了向主服务器和从服务器创建命令连接之外,哨兵还会向主服务器和从服务器创建订阅连接.哨兵会通过命令连接定期向主服务器和从服务器的频道中发布一条消息.对于主服务器,哨兵发送自身的信息,加上这个主服务器的信息.对于从服务器,哨兵发送自身的信息加上这个从服务器所复制的主服务器的信息.,其他哨兵由于通过订阅连接订阅了同样的频道,那么通过这个频道的消息就能知道其他哨兵的信息以及所监视的主服务器的最新信息.当哨兵知道了其他哨兵的信息之后,就可以互相创建之间的命令连接,之后可以主动发送INFO命令来获取其它哨兵的信息.

主观下线

哨兵会每隔1s向主服务器,从服务器及哨兵发送PING命令,如果在down-after-milliseconds毫秒内,没有返回有效回复,那么哨兵会认为对应的服务器处于主观下线状态.由于不同哨兵配置的主观下线时间不同,那么可能存在一个哨兵认为某个服务器处于主观下线状态,但是另外一个哨兵并不认为该服务器处于主观下线状态.

客观下线

1. 当哨兵发现一个主服务器处于主观下线状态,那么哨兵会向其他哨兵发送命令SENTINEL is-master-down-by-addr <current_epoch> 征求它对这个主服务器下线状态的看法
2. 当哨兵收到其它哨兵询问某个服务器是否下线的看法时,哨兵会根据传来的ip和port找到对应的服务器,并确定其是否下线(认为其主观下线或者客观下线),然后向询问的哨兵返回响应,如果down_state为1,则表示认为服务器已下线,否则认为服务器未下线.
3. 当哨兵征求完其他哨兵对服务器下线状态的看法后,会统计认为服务器已下线的哨兵的数量,当这个数量大于当前哨兵所配置的客观下线所需数量,则哨兵认为该服务器处于客观下线状态

由于不同哨兵配置的客观下线所需数量不同,那么可能存在一个哨兵认为某个服务器为客观下线状态,而另外一个哨兵认为该服务器不处于客观下线状态

领头哨兵选举

1. 哨兵发现一个服务器处于客观下线状态后,会向其他哨兵发出SENTINEL is-master-down-by-addr <current_epoch> 来请求其他哨兵将自己作为它的局部领头哨兵.
2. 在一个配置纪元里,一个哨兵只可以将另外一个哨兵作为自己的局部领头哨兵一次,并且规则是先到先得.如果一个哨兵被超过半数的哨兵作为局部领头哨兵,那么该哨兵将成为此次选举的首领,负责对下线的主服务器执行故障转移操作

哨兵如何选出一个新的主服务器

1. 删除从服务器列表中不在线的从服务器
2. 删除最近5秒没有通信过的从服务器
3. 删除所有连接断开超过down-after-milliseconds*10毫秒的从服务器
4. 选出偏移量最接近主服务器的从服务器
5. 将从服务器升级为主服务器

参考资料

- Redis设计与实现 16章 哨兵

15 Redis集群

Redis集群组成

Redis集群由节点组成,节点之间通过握手形成集群,当集群中的节点A与另外一个节点B握手完成之后,将会通过Gossip协议将节点B的信息传播给集群中的其它节点,让其它节点与B进行握手,经过一段时间,B将与集群中的其他节点也握手完成.只有当16384个槽都得到分配,集群才能够处于上线状态

如何发现集群中的节点处于下线状态

1. 集群中的节点会定期向其他节点发送PING消息,如果在规定的时间内可以收到PONG消息,则认为处于上线状态,否则将认为其处于疑似下线状态
2. 节点会将认为某个节点疑似下线的信息通过消息广播给其他节点
3. 其他节点会更新疑似下线节点的下线报告

4. 如果一个集群里面,超过半数的主节点认为某个节点处于下线状态,那么会将这个节点置为下线状态
5. 一个节点被置为已下线状态之后,节点会通过消息的方式将这个信息广播到集群中
6. 执行故障转移流程

故障转移流程

1. 当从节点发现自己复制的主节点下线之后,将向集群里的其他主节点请求一次投票给自己的机会,采用先到先得的方式来获取投票
2. 如果某个从节点获得半数以上的主节点的支持,那么这个从节点将选举成主节点

参考资料