基于 Bully 算法的 Redis 集群选举方案优化

王 芬 顾乃杰 黄增士
(中国科学技术大学计算机科学与技术学院 合肥 230037)
(中国科学技术大学先进技术研究院 合肥 230037)

摘 要 随着互联网的迅速发展,用户从系统获取的信息越来越多,访问系统的频率也在迅速增加。当大量客户端访问系统时,请求的响应时间也会大幅增加,传统关系型数据库已经无法满足用户的需求,而内存数据库在保证系统稳定的前提下,改善了用户体验,并得到了越来越广泛的应用。作为 NoSQL 内存数据库,Redis 支持很多数据类型,适用于多种情况下的缓存与存储需求。文中主要介绍 Redis 集群,它是 Redis 的分布式实现,支持主从复制,也具有一定的容错性和线性可扩展性,当前使用 Redis 集群的网站有新浪微博、github 等。虽然 Redis 集群应用广泛,但目前它在节点下线后会出现恢复时间长的现象,这与现有 Redis 集群的选举算法有关,即与 Raft 算法的实现有关。分析了 Redis 集群的可靠性,并优化了集群的选举算法。测试结果显示,在单个主节点下线 50s 内,优化后的集群都能成功恢复,比社区版本的集群提高了 40%。

关键词 Redis 集群,选举算法,纪元,节点,下线,恢复
中图法分类号 TP392 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.10.031

Election Scheme Optimization of Redis Cluster Based on Bully Algorithm

WANG Fen GU Nai-jie HUANG Zeng-shi
(School of Computer Science and Technology, University of Science and Technology of China, Hefei 230037, China)
(Institute of Advanced Technology, University of Science and Technology of China, Hefei 230037, China)

Abstract With the rapid development of Internet, users obtain more and more information from the system, and the frequency of accessing to system also grows rapidly. While a large number of clients access to the system, the response time of the request greatly increases, the traditional relational database is unable to meet the demand of the user, but in-memory database guarantees the stability of system, improves the user experience, and obtains more and more application. As a kind of in-memory database of NoSQL, Redis supports many data types, and it is applicable to many cases of requirements in caching and storage. In this paper, we mainly introduced Redis cluster, which is a distributed implementation of the Redis, supports master-slave replication, has a certain degree of fault tolerance and linear scalability, and recently is used by Sina microblog, github and so on. Although it is widely used, current Redis cluster occasionally has the case that long recovery time is needed after node fails, which has something to do with election algorithm of current Redis cluster, that is the implementation of Raft algorithm. In this paper, we analyzed the reliability of Redis cluster, and optimized the election algorithm of cluster. The results in test show that the optimized cluster can successfully recover in 50 seconds while only one master node is offline, and it is 40% higher than that of the cluster of the community version.

Keywords Redis cluster, Election algorithm, Epoch, Node, Fail, Recovery

1 引言

随着互联网的不断发展,用户每天都会产生海量数据,而传统关系型数据库[1]很难快速处理这些海量数据,因此多种 NoSQL 数据库[2-3]得到了广泛应用,如 Memcache[4],Redis 等[5]。随着数据库的飞速发展,内存数据库[6]引起了更多的关注。内存数据库将所有数据放在内存中,并直接操作数据,有效地解决了 CPU 与磁盘 I/O 之间的冲突。内存数据库读写速度快,减少了用户请求的响应时间,改善了用户体验;如今内存容量不再受 32 位系统限制,而内存价格也不断降低,

到稿日期:2016-08-30 返修日期:2017-02-08 本文受安徽省自然科学基金(1408085MKL06),高等学校学科创新引智计划项目(B07033),华为创新研究计划资助。

王 芬(1992-),女,硕士生,主要研究方向为分布式系统和分布式算法,E-mail:wfgirl1@mail.ustc.edu.cn;顾乃杰(1961-),男,博士,教授,主要研究方向为多级互联网络、并行计算;黄增士(1989-),男,博士生,主要研究方向为一致性存储系统的优化与应用。

这使得内存数据库的广泛应用成为一种不可逆转的趋势。

作为 NoSQL 内存数据库，远程字典服务(REmote DIctionary Server，Redis)是一种高性能的键值对存储系统[7]，由 Salvatore Sanfilippo 于 2009 年开发，从 2013 年 5 月开始其开发工作由 Pivotal 赞助。

Redis 支持很多数据类型，如字符串、链表、集合、有序集合等，因此满足多种情况下的缓存与存储需求。Redis 周期性地把数据写入磁盘以支持备份[8]；同时也会将修改操作写入追加的记录文件中，并在此基础上实现主从同步[9]，即 Redis 集群支持主从模式的数据备份。本文主要介绍 Redis 集群，它是 Redis 的分布式实现，在一定的容错性和线性可扩展性的基础上支持主从模式的数据备份。

DB-Engines[10]发布了 2016 年 7 月份的全球数据库排名，在所有数据库排名中 Redis 排名第十，但在键值对数据库排名中 Redis 排名第一。近几年，Redis 的发展及就业趋势虽然不是最好，但其依然不容被忽视。在许多大型网站中，键值对存储扮演着很重要的角色，如当前使用 Redis 的网站 github，Digg 等[11]。虽然 Redis 集群应用广泛，但测试结果显示节点下线后会出现恢复缓慢的情况，恢复时间甚至超过 5min，这与 Redis 集群的现有选举算法有关，即与 Raft 算法的实现有关。集群不能及时恢复，意味着客户端无法及时得到相应的数据，因此优化选举算法是必要的。

本文第 2 节描述了 Redis 的基础知识和 Raft 算法，以及与本文优化工作相关的 Bully 算法；第 3 节分析了 Redis 集群的可靠性，并提出了选举算法的优化方案；第 4 节将社区版的 Redis 集群与优化后的 Redis 集群进行了对比；最后对全文进行总结并讨论未来的研究方向。

2 相关工作

2.1 Redis

上节已经简单介绍了 Redis，现在通过将 Redis 与 Memcached 进行对比[9]以进一步描述 Redis。

Memcached[4]将数据都保存在内存中，重启时数据就会丢失；而 Redis 根据需要将内存中的部分数据保存在磁盘中，以保证数据的持久化。Memcached 仅支持简单的键值对类型的数据；而 Redis 支持很多数据类型，应用范围更加广泛。因为 Redis 是单线程程序，Memcached 支持多线程，所以在多核服务器上，相比于 Memcached，Redis 在性能上稍逊色。Redis 的出现弥补了 Memcached 作为键值对存储数据库的不足。

2.2 Redis 集群

作为 Redis 的分布式实现，Redis 集群将节点分为主节点和从节点，一个主节点与其对应的从节点组成一个分片。Redis 集群将键映射到 16384(2^{14})个槽中，每个分片只负责部分的槽位，并且这些槽位都互不相交。只有主节点负责处理槽，当主节点下线后，从节点通过选举成为主节点，这样就保证了数据库在故障时能够及时恢复数据。

图 1 给出了 Redis 集群的架构图，其中圆代表主节点，由此可以看出，客户端与 Redis 集群的主节点是直接连接的，不需要中间 Proxy 层；且因为主节点彼此互联，客户端只需要连接其中任何一个在线主节点就相当于连接了整个 Redis 集群。若超过半数主节点与某个节点通信超时，则认为该节点下线。客户端可以通过主节点和从节点来对数据库进行访问，因此从节点分担了对数据库进行访问的带宽负担。

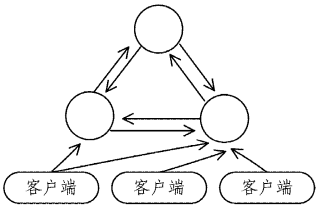

图 1 Redis 集群的架构图

2.3 Raft 算法及其在 Redis 中的实现

主节点下线后，Redis 使用 Raft 算法[12-13]选取主节点。本文只简单介绍与优化相关的 Raft 算法知识及 Raft 算法在 Redis 中的实现。

2.3.1 纪元

纪元是一个时间段，并且其值是单调递增的。Raft 的时间是由不同的纪元段拼接在一起的，如图 2 所示。

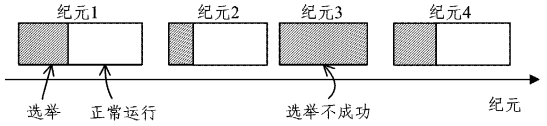

图 2 纪元时间轴

每个纪元开始于一个选举，若成功选举出一个主节点，则该主节点管理其他从节点直到纪元段结束，如纪元 1，2 和 4；若没能成功选出一个主节点，则选举失败，如纪元 3。

2.3.2 日志复制

我们已在 Redis 集群中提及主节点下线后的选举过程，即当主节点下线后，从节点通过选举成为主节点。下面介绍日志复制。

当选出主节点后，主节点就开始负责处理客户端的请求。如果请求包含增加日志的指令，则主节点首先将该指令追加到日志中，并形成一个新条目；然后将该新条目广播发送给从节点，并要求从节点都将该新的日志条目追加到从节点的日志中，追加成功后答复主节点；主节点在收到大多数(超过 $N/2$ 票，在本文中 N 表示集群总主节点数)从节点的答复后，应用该日志，并返回结果给客户端。日志组成如图 3 所示。

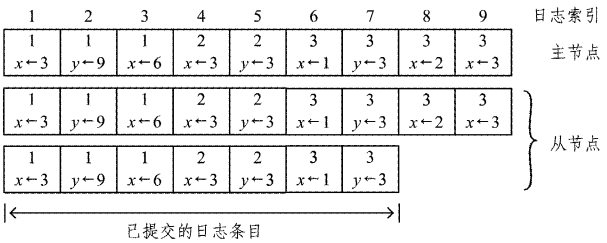

图 3 日志组成

2.3.3 Raft 算法在 Redis 中的实现

Redis 集群初始建立时纪元为 0，它是单调自增的。节点

保存配置纪元和当前纪元。当前纪元是实际所处的纪元,所有节点都保存当前纪元,配置纪元是上次投票所处的纪元,且只有主节点保存配置节点;配置纪元稳定存在,当前纪元则不断更新;当发现其他节点的当前纪元大于自身的当前纪元时,节点会更新自己的当前纪元;当投票给其他节点时,主节点会更新自己的配置纪元。本文中 m_i 的从节点为 s_{i1},在接下来介绍的选举过程中,m_1 是主节点,s_{11} 是其从节点,如图 4 所示。

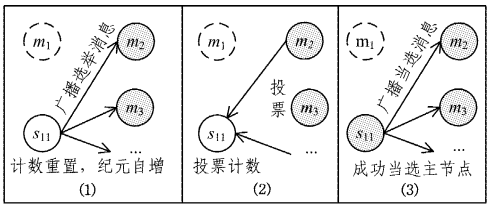

图 4　选举算法

(1)m_1 下线,s_{11} 发现正在复制的主节点 m_1 进入已下线状态时,计数重置,当前纪元自增 1;随后向集群广播选举消息,请求所有具有投票权的主节点向 s_{11} 投票。

(2)具有投票权的主节点收到 s_{11} 的投票,如果 s_{11} 的投票满足投票要求(接下来会详细讨论),那么主节点将给从节点投票,表示该主节点支持该从节点成为新的主节点,如 m_2 给 s_{11} 投票,而 m_3 并没有给 s_{11} 投票;参加选举的从节点 s_{11} 接收来自主节点的投票消息,并统计其获得的主节点投票数目。

(3)当票数达到 $N/2+1$ 时,从节点 s_{11} 当选为主节点;s_{11} 广播当选主节点的消息给其他主节点。

如果在一个纪元内,即在超时时间(60s)以内,集群中没有从节点能够收到足够多的支持票数,那么集群进入下一个纪元,即从(1)重新开始。

主节点收到从节点的投票请求后,检测投票请求是否满足投票要求,若满足,则投票给从节点。3 个投票要求为:

1)如果请求内的当前纪元小于主节点的当前纪元,那么主节点会拒绝投票;当主节点回复从节点投票请求时,主节点的当前纪元值会更新为请求内的当前纪元;

2)只有在集群判定从节点的主节点下线后,其他主节点才可以给从节点投票;

3)主节点保存上一次投票的纪元值,且上一次投票的纪元小于或等于当前纪元;如果请求内的当前纪元等于上一次投票的纪元,那么主节点会拒绝投票;当主节点回复从节点投票请求时,主节点上一次投票的纪元值会更新为请求内的当前纪元。

在某种情形下,这种选举算法会导致从节点出现多次选举才能成功的现象,而重选意味着超过超时时间(60s),从而使得恢复时间较长,因此在此处进行优化是必要的。

2.4　Bully 算法

本文中优化的选举算法使用类似 Bully 选举算法进行选举。Bully 选举算法[14] 的思想较简单:当任何在线进程发现协调者进程不响应请求时,发起一个选举,并向所有编号比它大的进程发送一个选举消息,如编号为 6 的进程向所有编号大于 6 的进程发送选举消息;任何在线进程收到选举消息后,发送响应给发起选举的进程,并接管选举工作;当一个进程在接管选举工作后无其他进程响应选举消息时,该进程成为新的协调者。Bully 选举算法的目的是在所有在线进程中选出编号最大的进程。

值得注意的是,当崩溃的协调者进程恢复后,它会发起一个选举,并且一定会获胜,从而再次成为协调者。Bully 算法很简单,但在选举过程中交换的信息数量较多,如今改进的 Bully 算法的存在使其在分布式系统中可以实现[15]。在下文介绍的 Redis 集群优化选举方案中,为了减少选举过程中交换的信息数量,利用节点的已知信息(日志偏移和 ID)直接在参与选举的节点中选出“等级”最高的节点,并投票给此节点,即基于 Bully 算法的特殊实现方式对 Redis 集群进行选举优化。

3　Redis 集群的可靠性分析及优化方案

3.1　可靠性分析

2.3.3 节提出了重选问题,本节介绍由于纪元号不一致导致的重选现象,并用一主一从的例子来说明纪元号冲突。

当 m_3,s_{11} 和 s_{21} 的当前纪元分别是 10,11,12 且 m_3 上一次投票的纪元是 10 时,m_3 收到 s_{11} 和 s_{21} 的投票请求顺序不同,会产生不同的结果。图 5 给出了 s_{11} 和 s_{21} 请求投票的不同顺序所产生的两种不同结果。主节点给从节点投票的要求在 2.3.3 节中已经详细描述,此处不再强调。

图 5 中实线箭头部分表示 m_3 先收到 s_{11} 的选举消息:

(1)m_3 收到 s_{11} 的选举消息,而未收到 s_{21} 的选举消息。

(2)因为 m_3 的当前纪元是 10,上一次投票的纪元是 10,它们都小于 s_{11} 的当前纪元 11,所以 m_3 的当前纪元和上一次投票的纪元都更新为 11,并给 s_{11} 投票;此时 m_3,s_{11} 和 s_{21} 的当前纪元分别是 11,11 和 12,m_3 上一次投票的纪元是 11。

(3)m_3 收到 s_{21} 的选举消息。

(4)因为 m_3 的当前纪元是 11,上一次投票的纪元是 11,它们都小于 s_{21} 的纪元 12,所以 m_3 的当前纪元和上一次投票的纪元都更新为 12,并给 s_{21} 投票;此时 m_3,s_{11} 和 s_{21} 的当前纪元分别是 12,11 和 12,m_3 上一次投票的纪元是 12。

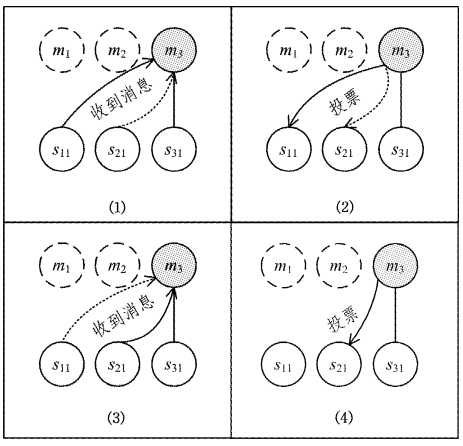

图 5　不同顺序下的请求结果

图 5 中虚线箭头部分表示 m_3 先收到 s_{21} 的选举消息:

(1)m_3 收到 s_{21} 的选举消息,而未收到 s_{11} 的选举消息。

(2)因为 m_3 的当前纪元是 10,上一次投票的纪元是 10,它们都小于 s_{21} 的纪元 12,所以 m_3 的当前纪元和上一次投票的纪元都更新为 12,并给 s_{21} 投票;此时 m_3,s_{11} 和 s_{21} 的当前纪元分别是 12,11 和 12,m_3 上一次投票的纪元是 12。

(3)m_3 收到 s_{11} 的选举消息。

(4)因为 m_3 的当前纪元和上一次投票的纪元是 12,它们都大于 s_{11} 的纪元 11,所以 m_3 不给 s_{11} 投票;此时 m_3,s_{11} 和 s_{21} 的当前纪元分别是 12,11 和 12,m_3 上一次投票的纪元是 12。

综上所述,在图 5 的虚线部分,m_3 不给 s_{11} 投票。如果多数主节点的投票状况与之类似,那么从节点 s_{11} 在当前纪元内就得不到充足选票,从而不得不进入下一轮选举。而重新选举需要重置参数、等待延迟等操作,从而便大幅度地降低了集群恢复的效率。

基于 2.3.3 节提到的主节点给从节点投票的要求,可以总结出:一个主节点在同一纪元内只能给一个从节点投票,并且拒绝之前纪元的投票请求。在逻辑上,不同分片间的选举操作是相互独立的,而在社区版的投票方式中不同分片之间的选举是相互影响的,并且导致从节点需经过多次选举才能当选主节点的结果。然而,在优化方案中,选举局限在一个分片之内,使得不同分片内的选举操作在实际情况下是相互独立的,不会因纪元冲突而出现多次选举的情况。

3.2 优化方案

从节点发现主节点下线后,会先发送主节点下线的消息给分片内的从节点,再发起选举消息,如果得到其他所有从节点(不包含分片内下线的从节点)的投票,则当选为主节点。从节点收到其他分片内从节点的选举消息,如果发现自己的“等级”低,就给该从节点投票。

一主两从的选举过程如图 6 所示,其步骤如下:

(1)m_1 是主节点,s_{11},s_{12} 是从节点。

(2)在 m_1 下线后,除 m_1 外的节点发送 ping 消息给 m_1,若得不到 m_1 的 pong,则标记 m_1 节点为 pfail,并在通过 ping 和其他节点通信时,告诉其他节点“我认为 m_1 是 pfail”,但是不会影响其他节点对 m_1 的看法(其中 pfail 表示疑似下线,fail 表示下线)。

(3)集群中任何一个节点 m 发现主节点 m_1 的 pfail 计数超过服务器总数的一半时就会把 m_1 标记为 fail,然后广播 fail 消息给集群中的其他节点。m_1 的从节点 s_{11} 和 s_{12} 也会收到主节点 fail 的消息,或者 m_1 的从节点 s_{11} 和 s_{12} 中某个节点发现 m_1 下线(pfail 计数超过一半以上)。

(4)当 s_{11} 或 s_{12} 发现主节点 m_1 fail 时,就会向其他片内节点广播 m_1 的 fail 消息,这些节点可能不是同时广播 m_1 的 fail 消息,且先后关系也并不会影响最终的选举结果,因此从节点可能从分片外的节点得知 m_1 的 fail 消息,也可能从分片内某个节点广播 m_1 的 fail 消息而得知。

(5)s_{11} 或 s_{12} 广播 m_1 的 fail 消息后就会发起选举。如(4)所述,这些节点可能不同时广播选举消息,且先后关系也并不会影响最终的选举结果。

(6)一个从节点收到投票信息后,只给比其“等级”高的从节点投票。在收到 s_{12} 发起的投票信息后,s_{11} 发现其“等级”低于 s_{12},于是给 s_{12} 投票。s_{12} 收到投票后统计投票数(V)。

(7)当投票数(V)+分片内从节点下线的数量(F)+1=n(本文中 n 表示分片内从节点数目)时,即投票数(V)+1=分片所有在线从节点的数目($n-F$)时,从节点当选为主节点。此时 $V=1$,$F=0$,$n=2$,显然 $V+F+1=n$,因此 s_{12} 当选为主节点。

(8)s_{12} 将它自己当选为主节点的消息通知 s_{11},s_{11} 得到此消息后,中止选举。

(9)s_{11} 为从节点,s_{12} 为主节点,s_{12} 可以向 s_{11} 发出指令,比如日志复制。

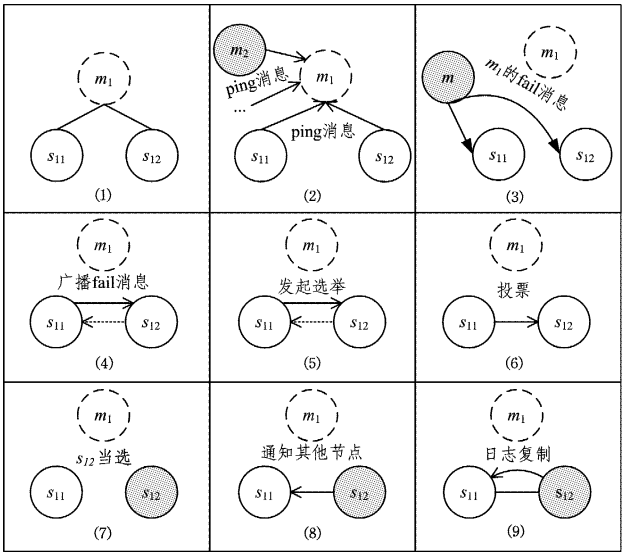

图 6 一主两从的选举过程

从节点的“等级”由两个指标决定:日志偏移和节点 ID。

上文已经介绍了日志复制,而在片内选举并不需要考虑纪元的比较,因此只需要关注日志偏移,如图 3 所示。由此可得,从节点的日志偏移越大,其成为主节点后丢失的数据越少。图 3 中若主节点下线,日志偏移为 9 的从节点成为主节点后,数据不丢失。

每个节点都有一个唯一的长度为 40bit 的字符串作为 ID。节点将它的 ID 保存到配置文件中,只要该配置文件不被删除,节点就会一直使用此 ID。一个节点可以改变它的 IP 和端口号,但不能改变 ID。因为 ID 是独一无二的,当节点的配置文件不被删除,ID 就一直被保存,并且比较 ID 大小的操作简单快速,所以使用 ID 来作为比较基准。

综上所述,首先比较日志偏移,日志偏移越大,等级越高;如果日志偏移相等,ID 越大(字符串比较),等级越高。

3.3 优化方案的优缺点

主节点下线后,从节点在发起选举后可能下线,这会影响分片内其他从节点的选举。下面对这种情况进行讨论,其中前 5 步与图 6 的(1)-(5)相同,此处不再介绍,其后面 3 个步骤如图 7 所示。

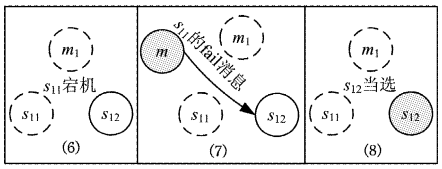

图 7　一主两从的部分选举过程

(6)s_{11}收到 m_1 的 fail 消息时就向其他分片内的从节点发送选举消息,不久后 s_{11}下线;在 s_{11}下线后,s_{12}并未及时发现,同时 s_{12}收到 m_1 的 fail 消息,发送选举消息,并等待其他从节点的投票,即 s_{11}的投票。此时 $V=0,F=0$(因为此时 s_{12}并不知道 s_{11}下线),$n=2$,显然 $V+F+1<n$,因此此时 s_{12}不能当选为主节点。

(7)s_{11}下线,除 m_1,s_{11}外的节点发送 ping 消息给 s_{11},若得不到 s_{11}的 pong,则标记 s_{11}节点为 pfail;集群中任何一个节点 m 发现从节点 s_{11}的 pfail 计数超过服务器总数的一半时就会把 s_{11}标记为 fail,然后广播 fail 消息给集群中的其他节点。从节点 s_{12}也会收到从节点 s_{11} fail 的消息;或者从节点 s_{12}发现 s_{11}下线(pfail 计数超过一半以上)。

(8)s_{12}得到的票数 $V=0$,片内下线的节点数 $F=1$,分片内的从节点数 $n=2$,$V+F+1=n$ 成立,s_{12}当选为主节点,此时该主节点没有从节点。

由图 7 可得,分片内某从节点下线后其他从节点只能在收到这些下线消息后才可能成为主节点,这使得恢复时间变长。假如一个从节点下线的概率为 0.1,则所有节点在整个选举过程中都不下线的概率为 0.9^n;当从节点越多,选举过程都不下线的概率越小,则下线的可能性越大,而选举过程中下线会增加恢复时间,如图 7 所示。因此可以得出结论:优化的选举算法并不适合分片内的从节点数目较多的情况。

然而,若主节点只有一个从节点,主节点下线后,从节点在收到主节点下线的消息时可以立即成为主节点(此时 $V=0,F=0,n=1;V+F+1=n$)。当主节点有两个从节点且出现如图 7 所示的情况时,从节点需要额外等待大约 15s 后才能成功当选为主节点。在从节点较少的情况下,优化算法的优势很明显,尤其是一主一从的情况。同时,因为优化的选举算法只在分片内进行选举,并未综合全局来选取主节点,所以集群的规模只对判断主节点和从节点是否下线有影响(图 6 中步骤(2)－(3),及图 7 中步骤(6)－(7)),而对选举算法的其他步骤并无影响。

4　测试

测试和优化所用的 Redis 的版本是 Redis-3.0.7,这是目前的最新版本。测试集群建立在 6 台服务器上,服务器的相关信息如图 8 所示。

Linux系统版本	CentOS7.1.1503
服务器品牌	Dell Inc. PowerEdge R720
CPU型号	Intel® Xeon® CPU E5-2690 v2*2
CPU主频	3.00GHz
内存	125GB
硬盘	300GB

图 8　服务器的相关信息

测试集群分布情况如下:每个主节点有两个从节点;主节点平均分布在两台服务器上,从节点平均分布在 4 台服务器上;当主节点分布在某一台服务器上时,其从节点一定分别分布在特定的服务器上。图 9 的集群规模是 72 个主节点,其他测试集群的分布情况类似。

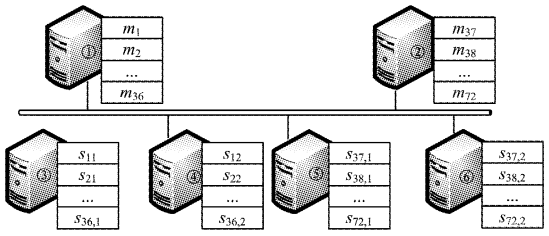

图 9　主节点和从节点的分布情况

在图 9 中,服务器①和②上放置 72 个主节点,其他服务器放置从节点。若主节点 m_2 放在服务器②上,那么从节点 s_{21},s_{22}就会分开放在服务器⑤和⑥上,其他主从节点类似。这样放置的原因在于:主节点和从节点存放在不同服务器上,可以在下文的性能测试中减少从节点对主节点的干扰;一个分片内的主节点和从节点不在同一台服务器上,当一台服务器出现问题时,其他服务器可以使集群恢复成功,这样降低了一个分片内所有节点都下线的概率。

将社区版的 Redis 集群记为原始版本,优化后的 Redis 集群记为优化版本。首先在固定集群规模的情况下将社区版本和优化版本的恢复时间进行对比;然后通过测试不同集群规模,将优化版本与原始版本在性能上进行比较,以检测优化版本在性能上是否下降。

图 10 中的集群规模为 72 个主节点,在此集群规模下手动下线 28 个主节点(约为 $72*\frac{2}{5}$),并统计集群恢复正常所需的时间。将以上实验进行 120 次测试,共得到 120 个恢复时间。图 10 为恢复时间的 CDF 曲线,CDF 曲线的定义为 $F(x)=P(X\leqslant x)$。x 为端点的横坐标,表示恢复时间;$F(x)$为端点的纵坐标,表示 CDF 值。图 10 中,当原始版本的 CDF 曲线上 $x_1=60$ 时,$F(x_1)=0.6$,代表 120 个恢复时间中恢复时间小于 60s 的数目与 120 之比为 0.6。在 120 次测试中,原始版本的恢复时间小于 60s 的次数为 $120*0.6=72$;优化版本的恢复时间小于 60s 的次数为 $120*1=120$;且优化版本的恢复时间在 30s 内,当恢复时间为 30s 时,CDF 值为 1;而原始版本的恢复时间即使为 140s 也不能保证 120 次下线能够成功恢复,在恢复时间为 140s 时,比率小于 1,原始版本最长恢复时间为 392s(并未在图中表示)。显然,优化版本能快速且稳定地恢复,大幅提升了恢复效率。

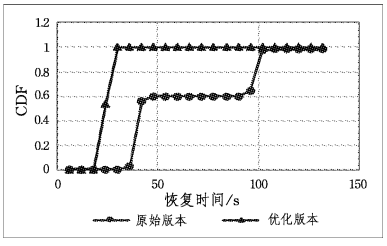

图 10　恢复时间的 CDF 曲线

图 11 与图 12 中测试集群的规模为 4,8,12,…,72,对每个集群规模测试 10 次,去除最大值和最小值后取平均值,优化版本的平均值与对应原始版本的平均值的比值为平均值之比。图 11 给出了 get 操作延时的平均值之比与 set 操作延时的平均值之比;平均值大于 1 说明优化版本在延时上的性能降低,否则性能提高。

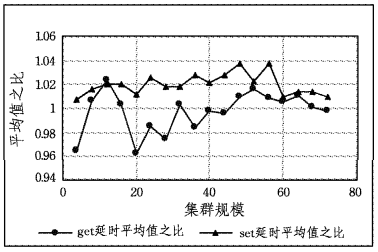

图 11 操作延时平均值之比

图 12 给出了 get 的 ops 平均值之比与 set 的 ops 平均值之比(ops 表示每秒操作次数);平均值大于 1 说明优化版本在 ops 上的性能得到提升,否则性能降低。

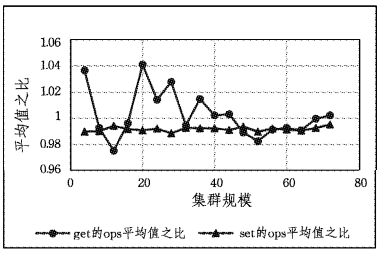

图 12 ops 平均值之比

图 11 显示 get 操作延时的平均值之比在 1 上下波动,波动范围为 0.96～1.03,因此在 get 操作延时上,优化版本与原始版本的性能大体一致;set 操作延时的平均值之比在 1 之上波动,波动范围为 1.00～1.04,因此在 set 操作延时上,优化版本性能有所下降。

图 12 显示 get 的 ops 平均值之比在 1 上下波动,波动范围为 0.98～1.04,因此在 get 的 ops 上优化版本与原始版本的性能大体一致;set 的 ops 平均值之比在 1 之下波动,波动范围为 0.98～1.00,因此在 set 的 ops 上优化版本性能有所下降。

结束语 Redis 是一种高性能的 NoSQL 内存数据库,支持的数据类型较多,同时支持数据持久化,适用范围较广。从 3.1 节可以了解到 Redis 集群在选举算法上还有所欠缺,因此本文优化了选举算法。针对选举算法,本文的优化结果为:1)保证在少数主节点下线的情况下,集群一定能够成功恢复;2)将节点同一纪元内投票的限制范围从整个集群改到单个分片内,在分片中从节点较少的情况下,集群恢复时间短且稳定;例如在分片内只有一个从节点的情况下,主节点下线,直接将从节点提升为主节点,集群成功恢复;通过测试一主两从的集群,结果显示片内优化选举方案使集群下线恢复的效率大幅提升。虽然本文优化了选举算法,但是在优化算法过程中可能需要等待集群判定片内从节点是否下线。因为在优化的选举算法中只能在收到其他所有从节点的下线消息后选举才能成功,所以片内选举并不适合节点下线概率较大的集群。本文使用社区版的方式来判断节点下线,而在集群大部分节点下线的情况下集群是不可用的,因此本文的优化算法在大部分节点下线的情况下无法完成主从切换。在未来,可以优化节点下线判断机制,增强节点下线判断的可靠性和稳定性,从而突显本文中优化算法的优势,使集群恢复时间更加稳定。

参 考 文 献

[1] CODD E F. Relational database:a practical foundation for productivity[J]. Communications of the ACM,1982,25(2):109-117.

[2] SHASHANK T. Professional NoSQL[OL]. http://is.muni.cz/publication/1062829.

[3] HAN J,HAIHONG E,LE G,et al. Survey on NoSQL database[C]// 2011 6th International Conference on Pervasive Computing and Applications (ICPCA). IEEE,2011:363-366.

[4] Bradley Joseph Fitzpatrick. memcached[OL]. http://memcached.org.

[5] HAN L,LI X. The Implementation of the Redis protocol based on NOSQL database storage[OL]. http://www.paper.edu.cn/html/releasepaper/2011/08/519. (in Chinese)
韩利,李昕. 基于 MySQL 数据库存储的 Redis 协议实现[OL]. http://www.paper.edu.cn/html/releasepaper/2011/08/519.

[6] GARCIA-MOLINA H,SALEM K. Main memory database systems:An overview[J]. IEEE Transactions on Knowledge and Data Engineering,1992,4(6):509-516.

[7] Salvatore Sanfilippo. Redis[OL]. http://redis.io.

[8] CATTELL R. Scalable SQL and NoSQL data stores[J]. ACM SIGMOD Record,2011,39(4):12-27.

[9] WANG X Y. Memcached and Redis's application in cache[J]. Wireless Internet Technology,2012(9):8-9. (in Chinese)
王心妍. Memcached 和 Redis 在高速缓存方面的应用[J]. 无线互联科技,2012(9):8-9.

[10] DB-Engines ranking[OL]. http://db-engines.com/en.

[11] BEREZECKI M,FRACHTENBERG E,PALECZNY M,et al. Many-core key-value store[C]//2011 International Green Computing Conference and Workshops (IGCC). IEEE,2011:1-8.

[12] ONGARO D,OUSTERHOUT J. In search of an understandable consensus algorithm[C]// 2014 USENIX Annual Technical Conference (USENIX ATC 14). 2014:305-319.

[13] AGRAWAL P. Fault tolerance in multiprocessor systems without dedicated redundancy[J]. IEEE transactions on computers,1988,37(3):358-362.

[14] GARCIA-MOLINA H. Election s in a distributed computing system[J]. IEEE Transactions on Computers,1982,100(1):48-59.

[15] KORDAFSHARI M S,GHOLIPOUR M,MOSAKHANI M,etal. Modified bully election algorithm in distributed systems[J]. WSEAS Transactions on Information Science and Applications,2005,2(8):1189-1194.