文章编号:1000-5641(2014)05-0001-16

NoSQL 系统的容错机制:原理与系统示例

孔 超, 钱卫宁, 周傲英
(华东师范大学 数据科学与工程研究院,上海 200062)

摘要: NoSQL 数据管理系统因其具有良好的可扩展性和容错性,在以 Web 数据管理和分析处理为代表的新型大数据应用环境中得到了广泛使用.这些系统通过新型一致性模型和数据冗余等技术,实现了集群环境中的容错处理.本文在对集群环境数据管理系统的一致性保持和容错处理基本原理进行介绍的基础上,对 Bigtable、HBase、Dynamo、Cassandra,以及 PNUTS 五个典型的 NoSQL 系统的容错机制及其实现进行分析与对比,并讨论它们的设计原则和实现技术对于系统的可用性、性能、复杂负载的处理能力等方面的影响.最后,讨论现有 NoSQL 系统容错机制对于设计和实现支持关键任务的内存数据管理系统的借鉴意义.

关键词: NoSQL 系统; 容错; 一致性保持
中图分类号: TP311.131 **文献标识码**: A **DOI**:10.3969/j.issn.1000-5641.2014.05.001

Fault-tolerance in NoSQL systems: Principles and system analysis

KONG Chao, QIAN Wei-ning, ZHOU Ao-ying
(*Institute for Data Science and Engineering, East China Normal University, Shanghai* 200062, *China*)

Abstract: NoSQL data management systems have been widely used in Web data management and processing applications, for their high scalabilities and fault tolerence. The fault tolerence is achieved by using new consistency models and data replications in clustered systems. In this paper, the mechanism and implementation details of five representative NoSQL systems, i.e. Bigtable, HBase, Dynamo, Cassandra, and PNUSTS, were discussed and analyzed, after a general introduction to the principles of consistency preserving and fault tolerent processing. Furthermore, the impact of these technologies on system availability, performance and workload balance, was analyzed. Finally, their influence on the design of in-memory database management systems was discussed.

Key words: NoSQL systems; fault-tolerance; consistency preserve

0 引 言

传统的关系型数据管理系统已经不能满足高并发的读写、高可用性和高可扩展性的新

收稿日期:2014-06
基金项目:国家 973 课题(2010CB731402);国家自然科学基金(61170086)
第一作者:孔超,男,博士研究生,研究方向为 Web 数据管理. E-mail: 52121500012@student.ecnu.edu.cn.
第二作者:钱卫宁,男,博士生导师,研究方向为 Web 数据管理与挖掘. E-mail: wnqian@sei.ecnu.edu.cn.
通信作者:周傲英,男,博士生导师,研究方向为数据管理与信息系统. E-mail: ayzhou@sei.ecnu.edu.cn.

兴互联网应用的需求. NoSQL 系统在这种背景下应运而生. NoSQL 常被认为不同于传统关系型数据管理系统,是具有非关系型、分布式、开源、横向扩展等特性的新型数据管理系统[1]. 这些系统牺牲了一些已在传统关系型数据库中成为标准的功能,如数据一致性、标准查询语言以及参照完整性等,以换取高可扩展性、高可用性和高可靠性.

随着互联网技术的快速发展,海量数据的存储、管理和处理已经成为全球各大互联网公司不可回避的严峻问题. 以业务范围跨越 C2C(个人对个人)和 B2C(商家对个人)的淘宝网为例:截至 2013 年,淘宝网拥有近 5 亿的注册用户数,每天有超过 6 000 万的固定访客,每天的在线商品数已经超过了 8 亿件,平均每分钟售出 4.8 万件商品[2,3];2013 年 11 月 11 日零时,开场仅 1 分钟成交的订单数量达到 33.9 万笔,总成交金额达到 1.17 亿元,第二分钟,成交数字突破 3.7 亿元,到了零时 6 分 7 秒,成交额直接冲上 10 亿元,截至 11 日 24 时,"双 11"天猫及淘宝的总成交额破 300 亿元,达 350.19 亿元[4],这些海量交易信息的存储、分析和处理对淘宝网提出了巨大的挑战,类似的问题也出现在 Google、Facebook、Yahoo 等互联网应用上. 例如,2012 年 Facebook 在总部的一次会议中披露了一组 Facebook 每天要处理的数据:25 亿条分享的内容条数,27 亿个"赞"的数量,3 亿张上传的照片数,超过 500 TB 新产生的数据,每半小时通过 Hive 扫描 105 TB 的数据,单个 HDFS 集群中的磁盘容量超过 100 PB[5].

当今的互联网应用具有以下特性:

(1) 用户基数大,而且增长速度快;

(2) 数据类型多、总量大;

(3) 对数据操作较为单一,一致性要求较弱.

虽然互联应用中涉及的数据类型多样,但普通用户的数据处理操作较为单一,对数据的操作无非读或写,或者是增加、删除、修改和查询等. Henderson 在 2008 年 Web 2.0 expo 中的报告指出:通常在互联网应用中,用户更多的是读数据;据统计,读/写数据的比例大约为 80:20 或者 90:10[6]. 低延迟的用户响应、高吞吐量是首先要考虑的技术问题,以满足基本的用户需求;而对数据没有严格的强一致性要求,这有别于金融行业的数据操作.

互联网应用的这些特性对海量数据存储、管理和处理提出了巨大的挑战,例如:支持 PB 级甚至 EB 级数据的存储系统、具有良好的扩展性以满足不断增长的数据和用户需求,具有低延迟的用户响应和高吞吐量,具备良好的容错机制以保证互联网应用的高可用性和高可靠性等.

现有的 NoSQL 系统都有相应的机制来解决容错问题. 容错机制与 NoSQL 系统的高可用性、高可靠性息息相关,在采用大量非可靠硬件的集群环境中尤为重要. 良好的容错机制使得 NoSQL 系统在某些组件发生故障时,仍能继续为用户服务,满足基本的用户需要. Bigtable、HBase 等几种典型的 NoSQL 系统的容错机制越来越成熟,大大提高了 NoSQL 系统的可靠性和可用性.

随着硬件技术,特别是内存技术的发展,基于内存计算的数据管理系统,如 SAP HANA,因其所具有的高性能,已经引起了学术界和工业界的广泛关注[7]. 而在依赖于易失内存的数据管理系统中的容错处理,则是内存数据管理系统所无法回避的重要问题. NoSQL 系统中能够保持系统高可扩展性的容错机制和实现技术,为这一问题的研究提供了思路.

本文以下主要从三个部分展开论述. 第 1 部分对集群环境数据管理系统的一致性保持

和容错处理基本原理进行介绍；第 2 部分对 Bigtable、HBase、Dynamo、Cassandra 和 PNUTS 五个典型的 NoSQL 系统的容错机制及实现进行分析对比，并讨论它们的设计原则和实现技术对于系统的可用性、性能、复杂负载的处理能力等方面的影响；第 3 部分讨论现有 NoSQL 系统容错机制对于设计和实现支持关键任务的内存数据管理系统的借鉴意义.

1　基础理论

数据库领域的容错指系统从故障恢复的能力，是数据管理应用中必须考虑的关键问题.

NoSQL 存储系统的容错机制设计，需要考虑恢复和复制技术，还必须考虑它们对系统的性能和负载能力等方面的影响. 本节介绍构建高可用、高可靠的 NoSQL 系统的基础理论，包括 CAP 理论，以及分布式锁服务、恢复和复制等实现技术.

1.1　CAP 理论

2000 年，Brewer 提出了 CAP 理论，即一个分布式系统，最多只能同时满足：一致性(Consistency，用 C 表示)，可用性(Availability，用 A 表示)和网络分区容忍性(Tolerance to Network Partitions，下文简称分区容忍性，用 P 表示)，这三个需求中的两个[8]. 2002 年 Gilbert 和 Lynch 论证了该理论的正确性[9].

一般而言，分布式系统首先应该具备分区容忍性，以满足大规模数据中心的相关操作. 因此 CAP 理论意味着对于一个网络分区，在一致性和可用性二者之间的取舍[10]. 传统的关系数据库系统选择一致性，而互联网应用更倾向于可用性. Brewer 指出：在大多数情况下，分区故障不会经常出现，因此在设计系统的时候允许一致性和可用性并存[11]. 当分区故障发生时，应该有一套策略来检测出这些故障，并有合理的故障恢复方法. 当然，在系统设计时，不可能完全舍弃数据一致性，否则数据是不安全的和混乱错误的，以致再高的可用性和可靠性也没了意义. 牺牲一致性指允许系统弱化一致性要求，只要满足最终一致性(Eventual Consistency)即可，而不再要求关系型数据库中的强一致性(即时一致性). 最终一致性指：若对一个给定的数据项没有新的更新，那么最终对该数据项所有的访问都返回最后更新的值. 它被集群环境的数据管理系统广泛采用，也常被称为“乐观复制”(Optimistic Replication)[12].

根据 CAP 理论，C、A、P 三者不可兼得，必须有所取舍. 传统关系型数据库系统保证了强一致性(ACID 模型)和高可用性，但其扩展能力有限. 而 NoSQL 系统则通过牺牲强一致性，以最终一致性进行替代，来使得系统可以达到很高的可用性和扩展性[13].

1.1.1　一致性(Consistency)

分布式存储系统领域的一致性指：在相同的时间点，所有节点读到相同的数据[14]. 传统的关系型数据库系统很少存在一致性问题，数据的存取具有良好的事务性，不会出现读写的不一致；对于分布式存储系统，一个数据存多份副本，一致性要求用户对数据的修改操作要么在所有的副本中操作成功，要么全部失败. 若保证一致性，那么用户读写的数据则可以保证是最新的，不会出现两个客户端在不同的节点中读到不同的情况.

1.1.2　可用性(Availability)

可用性指：用户发送访问请求时，无论操作成功与否，都能得到及时反馈[14]. 系统可用不代表所有节点提供的数据是一致的. 实际应用中，往往对不同的应用设定一个最长响应时间，超过这个响应时间的服务被认为是不可用的.

1.1.3 网络分区容忍性(Tolerance to Network Partitions)

分区容忍性指:任意消息丢失或部分系统故障发生时,系统仍能良好地运行[14].一个存储系统只运行在一个节点上,要么系统整个崩溃,要么全部运行良好,一旦分布到了多个节点上,整个存储系统就存在分区的可能性.

1.2 分布式锁服务:以 Chubby 为例

分布式锁是分布式系统中控制同步访问共享资源的一种方式.如果不同的系统或同一个系统的不同主机之间共享了一个或一组资源,当访问这些资源时,需要互斥来防止彼此干扰,从而保证一致性,则需要使用分布式锁[15].

分布的一致性问题描述为:在一个分布式系统中,有一组进程,需要这些进程确定一个值.于是每个进程都给出了一个值,并且只能其中的一个值被选中作为最后确定的值以保证一致性,当这个值被选出来以后,要通知所有的进程.Naïve 的解决方案为:构建一个 master server,所有进程都向它提交一个值,master server 从中挑一个值,并通知其他进程.在分布式环境下,该方案不可行,可能会发生的问题有:master server 宕机怎么办? 由于网络传输的延迟,如何保证每个值到达 master server 的顺序等[16].

为解决以上问题,Chubby 被构建出来,它并不是一个协议或者算法,而是 Google 精心设计的一个分布式的锁服务[17].通过 Chubby,client 能够对资源进行"加锁"、"解锁".Chubby 通过文件实现这样的"锁"功能,创建文件就是进行"加锁"操作,创建文件成功的 server 则抢占到了"锁",用户通过打开、关闭和读取文件,获取共享锁或者独占锁,并且通过通信机制向用户发送更新信息[18].Chubby 的结构见图 1.

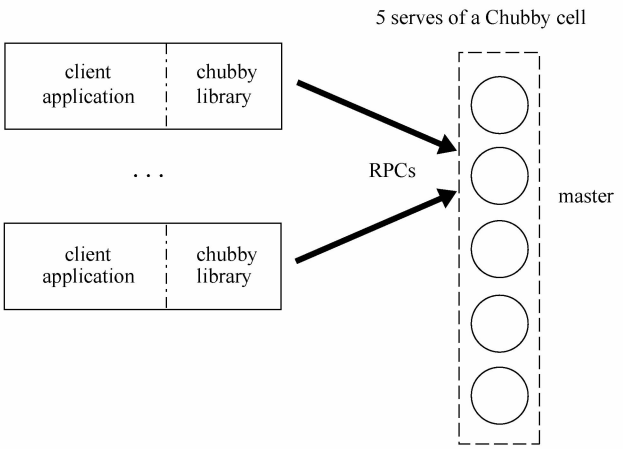

图 1 Chubby 结构[17]

Fig. 1 Structure of Chubby

如图 1 所示,Chubby 有两个主要组件:Chubby cell 和 Chubby library;两者通过 RPCs 通信. Chubby cell 由五个被称为 *replica* 的 server 组成;只要其中三个正常,它就能提供服务.这些 server 通过分布式一致性协议选举一个作为 master server;client application 和 server 间的通信都需要通过 Chubby library;client application 通过 Chubby library 的接口调用,在 Chubby cell 上创建文件获得相应锁功能.

文献[17]阐述了 Chubby 的工作流程:首先,Chubby cell 从五个 *replica* 按照分布式一

致性协议选举出一个 master，master 必须获得半数以上的选票，同时保证在该 master 的租约内不会再选举出新的 master，*replica* 通过 Paxos 协议[19]保持日志的一致性，采用 Quorums[20]做决策，使用多副本满足高可用性；其次，每个 *replica* 维护一个数据库的拷贝，但只有 master 能够执行读/写操作，master 通过一致性协议将更新传送到其他 *replica* 上. client 向 DNS 中 *replica* 列表发送 master 定位请求来找到 master，非 master 的 *replica* 返回 master 标识符响应请求，一旦 client 找到 master 就会将所有请求直接发送给 master. 读请求由 master 处理，只要 master 还在租约期内就是安全的. master 会通过一致性协议将写请求发送给其他 *replica*，当半数以上的 *replica* 收到请求，则认为操作成功；最后，一旦 master 意外停机，其他 *replica* 在 master 租约过期后选举其他的 *replica* 作为 master.

1.3　分布式容错中的复制技术和恢复技术

容错指：一个系统的程序在出现逻辑故障的情况下仍能被正确执行[21]. 分布式系统中容错的概念指：在系统发生故障时，以不降低系统性能为前提，用冗余资源完成故障恢复，使系统具备容忍故障的能力[22]. 本节介绍分布式容错中的复制技术和恢复技术.

1.3.1　复制

分布式存储系统中的数据一般存储多个副本以保证系统的高可用性和高可靠性，当存储某个副本的节点发生故障时，系统能自适应地将运行中的服务切换到其他副本，实现自动容错. 复制技术可以分为同步复制和异步复制两大类：同步复制能够保证主备副本的强副本一致性：当客户端访问一组被复制的副本时，每个副本就如同一个逻辑服务[23]，但当备副本发生故障时，可能会影响系统正常的写操作，降低系统的可用性；异步复制通常只能满足最终一致性：保证副本的最终状态相同，在异步系统中，可能存在一个或多个主节点来完成副本间的状态同步[24]，但当主副本出现故障时，可能会导致数据丢失，一致性得不到保障. 分布式存储系统中使用的复制协议主要有：基于主副本的复制协议(Primary-based protocol)和基于多个存储节点的复制协议(Replicated-write protocol)[25].

主备复制通常在分布式存储系统中保存多个副本，选举其中一个副本为主副本，其他的为备副本，数据写入主副本中，由主副本按照一定的操作顺序复制到备副本[26]. 同步复制和异步复制都是基于主副本的复制协议，两者的区别在于：异步复制协议中，主副本无需等待备副本的响应，只需本地操作成功便告知客户端更新成功. 同步复制的流程如下：第一，客户端向主副本发送写请求，记为 W1；第二，主副本通过操作日志(Commit Log)的方式将日志同步到备副本，记为 W2；第三，备副本重放(replay)日志，完成后通知主副本，记为 W3；第四，主副本进行本地修改，更新完毕后通知客户端操作成功，记为 W4. R1 表示客户端将读请求发送给一个副本，R2 表示将读取结果返回给客户端，具体见图 2.

主备副本之间的复制通过操作日志的方式来实现：第一，将客户端的写操作顺序写到磁盘中；第二，利用内存的随机读写特性，将其应用到内存中，有效组织数据；第三，宕机重启后，重放操作日志恢复内存状态. 为了减少执行长日志的代价，系统会定期创建 checkpoint 文件将内存状态 dump 到磁盘中. 若服务器出现故障，则只需恢复在 checkpoint 之后插入的日志条目.

1.3.2　恢复

构建具备健壮性的分布式存储系统前提是具备良好的容错性能，具备从故障种恢复的

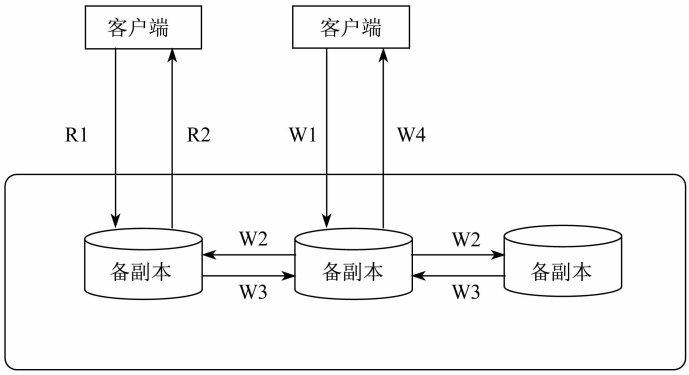

图 2 同步复制流程[27]

Fig. 2 The process of synchronous replication

能力. 在 NoSQL 系统中,这样的恢复能力有一个前提:拥有有效的故障检测(Failure Detection)手段,只有能有效、即时地检测到故障发生,才有制定恢复策略的可能. 故障检测是任何一个拥有容错性的分布式系统的基本功能,实际上所有的故障检测都基于心跳(Heartbeat)机制:被监控的组件定期发送心跳信息给监控进程,若给定时间内监控进程没有收到心跳信息,则认为该组件失效[28].

在 NoSQL 系统中,当 master server 检测到 slave server 发生故障时,便会将服务迁移到其他节点. 常见的分布式系统有两种结构:单层结构和双层结构,如图 3 所示.

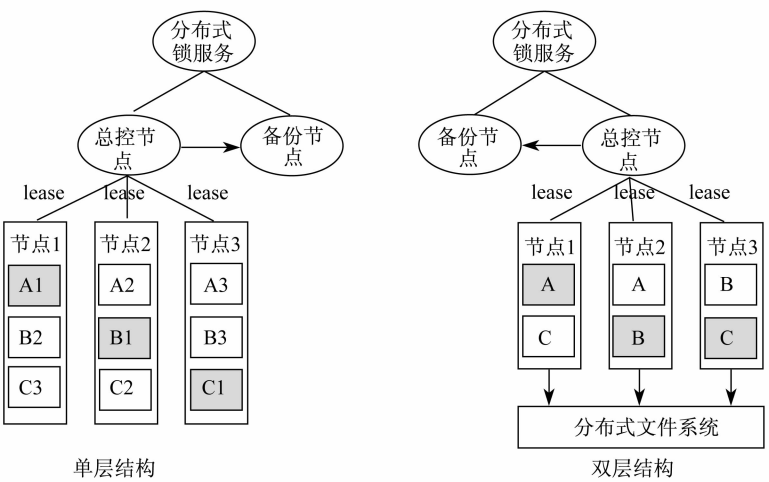

图 3 故障恢复[27]

Fig. 3 Failure recovery

单层结构的分布式存储系统有三个数据分片 A、B 和 C,分别存储了三个副本;其中 A1、B1、C1 为主副本,分别存储在节点 1、节点 2 和节点 3. 若节点 1 发生故障,主控节点检测到该故障,就会选择一个最新的副本,A2 或 A3 替换 A1 成为新的主副本提供写服务.

双层结构的分布式存储系统会将所有的数据持久化写入底层的分布式文件系统,每个数据分片同一时刻只有一个提供服务的节点. 若节点 1 发生故障,主控节点将选择节点 2 加

载 A 的服务. 由于 A 的所有数据都存储在共享的分布式文件系统中，节点 2 只需要从底层分布式文件系统中读取 A 的数据并加载到内存中.

2　系统示例

高可用性和高可靠性是互联网应用需求下海量数据存储考虑的首要问题，也是核心存储系统的基本特性. 以下通过分析对比 Bigtable、HBase、Dynamo、Cassandra 和 PNUTS 五个典型的 NoSQL 系统的容错机制：故障检测手段和故障恢复策略，探讨对系统的可用性、性能、一致性保持和复杂负载处理能力的影响.

2.1　Bigtable 和 HBase

Bigtable 是一个高性能、高可扩展性的分布式的结构化数据存储系统，其系统结构如图 4 所示. Google 的很多项目使用 Bigtable 存储数据，包括 Web 索引、Google Earth 和 Google Finance. 这些应用对 Bigtable 提出的要求差异非常大，无论是在数据量上（从 URL 到网页到卫星图像）还是在响应速度上（从后端的批量处理到实时数据服务）[29]. 作为集中式数据管理系统，Bigtable 采用传统的服务器集群架构，整个系统由一个主控服务器（master server）和多个片服务器（tablet server）构成，主控节点集中控制、管理、维护从节点的元信息. 集中式管理的优势在于：人为可控且维护方便，在处理数据同步时较简单；其劣势在于：系统存在单点故障的危险.

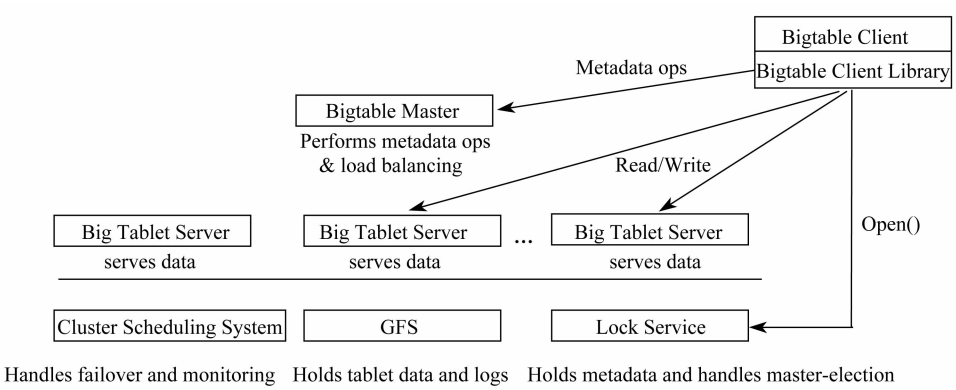

图 4　Bigtable 的系统结构[30]

Fig. 4　Architecture of Bigtable

HBase 是在 HDFS（Hadoop 分布式文件系统）上开发的基于列的分布式存储系统，具有高可靠性、高性能、可伸缩、支持实时读写等特性，其系统结构见图 5. 该项目由 Powerset 公司的 Chad Walters 和 Jim Kelleman 在 2006 年末发起的，根据 Chang 等人发表的论文“Bigtable：A Distributed Storage System for Structured Data”来设计的[31]. HBase 的 Client 使用远程过程调用（RPC）机制与 master 和 region server 通信，完成管理和读写等操作；HBase 中的 ZooKeeper 当作一个协调工具[32]. ZooKeeper 中存储-ROOT-目录表的地址信息和 master 的地址信息. region server 在 ZooKeeper 中注册，所以 master 可以跟踪每个 region server 的状态；HBase 不同于 Bigtable，允许启动多个 master，ZooKeeper 保证总有有效个 master 运行，因此弱化了 Bigtable 中单点故障的问题.

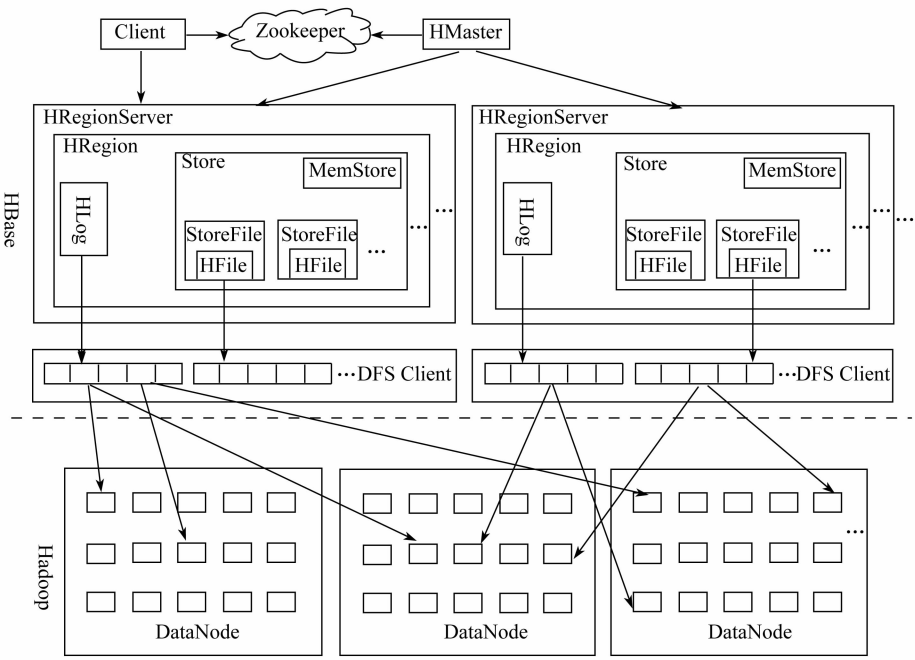

图 5 HBase 系统结构[33]

Fig. 5 Architecture of HBase

HBase 作为 Google 的 Bigtable 架构的一个开源实现,在容错机制和一致性保持的某些方面继承了 Bigtable 的特性;但二者之间也存在细致的差别. 下文主要从这两个方面进行阐述.

2.1.1 Bigtable 和 HBase 的容错机制

Bigtable 和 HBase 的故障检测手段都是基于 Heartbeat 机制,但具体的实现方式有区别. Bigtable 通过分布式锁服务 Chubby 确保任何时刻最多只有一个活动的 master 副本,检测 tablet server 和 master server 状态,以便进行故障恢复.

在 Bigtable 中,通过 Chubby 跟踪 tablet server 状态信息. 当启动一个 tablet server 时,系统会在特定的 Chubby 目录下,对一个唯一标识的文件获得排他锁(exclusive lock),master 会通过该目录检测 tablet server 状态,当诸如网络不稳定等因素导致 tablet server 和 Chubby 之间的会话中断,tablet server 失去排他锁,便停止对 tablet 的一切服务.

当 tablet server 出现故障时,tablet server 尝试重新获取唯一标识的文件的排他锁,若该文件不存在, tablet server 将永远不会提供服务,便自行终止进程. 一旦 tablet server 终止,它将尝试释放锁,此时 master 便对 tablet 重新合并、切分、分配到其他 tablet sever 上. 该过程中,master 为了随时检测 tablet server 的状态信息,会通过 Heartbeat 机制周期性地询问每个 tablet server 排他锁的持有状态,若 tablet server 报告失去排他锁,或者 master 和 tablet server 无法通信,master 便获取锁,一旦 tablet server 宕机或与 Chubby 通信出现故障,master 便确认 tablet server 无法提供服务,便删除该 tablet server 在 Chubby 目录下唯一标识的文件.

当 master 与 Chubby 之间的会话失效时,master 终止服务,但是不会改变 tablet server

当前对 tablet 的分配情况. 此时,master 会被系统重启:首先,master 在 Chubby 中获取一个唯一的 master lock,以防出现并发的 master 实例,确保只有一个 master;其次,master 扫描 Chubby 下 server 目录,获取现存的 server 信息;第三,master 与现有的每个 tablet server 进行通信,以确定哪些已被分配了 tablet;最后, master 扫描 METADATA 表,获取 tablet 信息. 在扫描时,一旦发现某个 tablet 尚未被分配,master 便把该 tablet 信息添加到"未分配"的 tablet 集合中,保证这些未被分配的 tablet 有被分配的机会.

Bigtable 的底层存储是基于 GFS(Google File System)的,其对于数据、日志的容错主要通过 GFS 多副本冗余来保证. GFS 存储的文件都被分割成一个个大小固定的 chunk. 在 chunk 创建之时,master 服务器会给每个 chunk 分配一个唯一、不变的 64 位标识,chunk 服务器把 chunk 以 Linux 文件的形式保存在本地硬盘上,每个 chunk 将副本写入多台 chunk 服务器中,master 节点管理所有从节点文件的元数据(METADATA):命名空间、访问控制信息、文件和 chunk 的映射信息以及当前 Chunk 位置信息等[34]. 当客户端进行读操作时,从 Namenode 中获取文件和 chunk 的映射信息,再从可用的 chunk 服务器中读数据.

HBase 基于 Heartbeat 机制进行故障检测, master 和 region server 组件的故障恢复策略如下:

(1) 当 master 出现故障时,ZooKeeper 会重新选择一个 master;当新的 master 被启用之前,数据的读取正常进行,但不能进行 region 分割和均衡负载等操作.

(2) 当 region server 出现故障时,它通过 Heartbeat 机制定期和 ZooKeeper 通信;若一段时间内未出现心跳,master 会将该 region server 上的 region 分配到其他 region server 上. 由图 5 可知,MemStore 中内存数据全部丢失;此时,region server 中的一个实现预写日志(Write Ahead Log)的类 HLog 保存了用户每次写入 MemStore 的数据. 具体恢复过程如下:master 通过 ZooKeeper 感知 region server 出现故障,master 首先处理该 region server 遗留的 HLog 文件,将不同 region 的日志文件拆分,放到相应 region 目录下;其次将失效的 region 重新分配到 region server 上,这些 region server 在加载被分配到的 region 的同时会重写历史 HLog 中的数据到 MemStore;最后,Flush 到 StoreFile,数据得以恢复[35].

HBase 0.90 版本以后开始使用基于操作日志(put/get/delete)的副本机制进行失效恢复:region server 将客户端的操作写入本地 HLog 中,每个 region server 将 HLog 放入对应 znode(ZooKeeper 维护的树形层次结构中的一个节点)上的副本队列,将 HLog 内容发送到集群中某个 region server 上,并将当前复制的偏移量保存在 ZooKeeper 上,整个过程采用异步复制机制,满足高可用性的需求[36].

在工程实践中,Bigtable 和 HBase 为避免某个节点的访问压力过载造成的节点失效,有专门的负载均衡策略[37]来解决这一问题. Bigtable 的 tablet 服务节点上的负载均衡依靠 master 通过心跳机制周期性地监控 tablet server 的负载情况:将增长到阈值的 tablet 切分后迁移到负载压力较轻的节点上,可以将用户的请求均衡分布到 tablet server 上;HBase 的数据在存储节点上的负载均衡由 HDFS 完成. 一个文件的数据保存在一个个 block 中,当增加一个 block 时,通过以下四种方式保证数据节点的均衡负载[38]:

(1) 将数据块的一个副本放在正在写这个数据块的节点上;

(2) 尽量将数据块的不同副本分布在不同的机架上,这样集群可在完全失去某一机架

的情况下还能存活;

(3) 一个副本通常被放置在和写文件的节点同一机架的某个节点上,这样可以减少跨越机架的网络 I/O;

(4) 尽量均匀地将 HDFS 数据分布在集群的数据节点中.

2.1.2 Bigtable 和 HBase 的一致性保持

Bigtable 和 HBase 采用多副本冗余的方式满足系统性能,但多副本冗余的的直接后果就是数据一致性问题,本节主要探讨 Bigtable 和 HBase 如何在不影响系统的可用性和性能的前提下考量一致性问题.

Bigtable 保证强一致性:某个时刻某个 tablet 只能为一台 tablet server 服务,即 master 将子表分配给某个 tablet server 时确保没有其他的 tablet server 正在使用这个 tablet[29]. 通过 Chubby 的互斥锁机制来实现:首先,启动 tablet server 时获取 Chubby 互斥锁,一旦 tablet server 出现故障,master 要等到 tablet sever 的互斥锁失效时才能把出现故障的 tablet server 上的 tablet 分配到其他 tablet server 上.

HBase 保证最终一致性(eventual consistency),通过 ZooKeeper 来实现:第一,客户端的更新请求以其发送顺序被提交;第二,更新操作要么成功,要么失败,一旦操作失败,则不会有客户端看到更新后的结果;第三,更新一旦成功,结果会持久保存,不会受到服务器故障的影响;第四,一个客户端无论连接哪一个服务器,看到的是同样的系统视图;第五,客户端看到的系统视图滞后是有限的,不会超过几十秒[31].

2.2 Dynamo 和 Cassandra

Dynamo 和 Cassandra 的数据管理方式是非集中式的. 系统中没有主从节点的区分,每个节点都和其他节点周期性地分享元数据,通过 Gossip 协议[39],节点之间的运行状态可以相互查询. 非集中式的数据管理方式避免了单点故障,但同时由于没有 master,实现 METADATA 的更新操作会比较复杂.

Dynamo 是 Amazon2007 年推出的基于 Key-value 的大规模高性能数据存储系统,面向服务的 Amazon 平台体系结构如图 6 所示[40].

Cassandra 是 Facebook 的采用 P2P 技术实现的去中心化的结构分布式存储系统,Cassandra 系统设计目标是:运行在廉价商业硬件上,高写入吞吐量处理,同时又不用以牺牲读取效率为代价[41].

2.2.1 Dynamo 和 Cassandra 的容错机制

Dynamo 采用多副本冗余的方式保证系统的高可用性,各节点之间通过 Gossip 机制相互感知,进行故障检测. Dynamo 的容错机制主要有以下两点:(1) 数据回传. 在 Dynamo 中,一份数据被写到编号 K, $K+1$, …, $K+N-1$ 的 N 台机器上,若机器 $K+i(0\leqslant i\leqslant N-1)$ 宕机,原本写入该机器的数据转移到机器 $K+N$ 上,若在给定的时间 T 内机器 $K+i$ 恢复,$K+N$ 通过 Gossip 机制感知到,将数据回传到 $K+i$ 上. (2) Merkle Tree[42] 同步. 当超过时间 T 机器 $K+i$ 还没有恢复服务的话,被认为是永久性异常,通过 Merkle Tree 机制从其他副本进行数据同步[27].

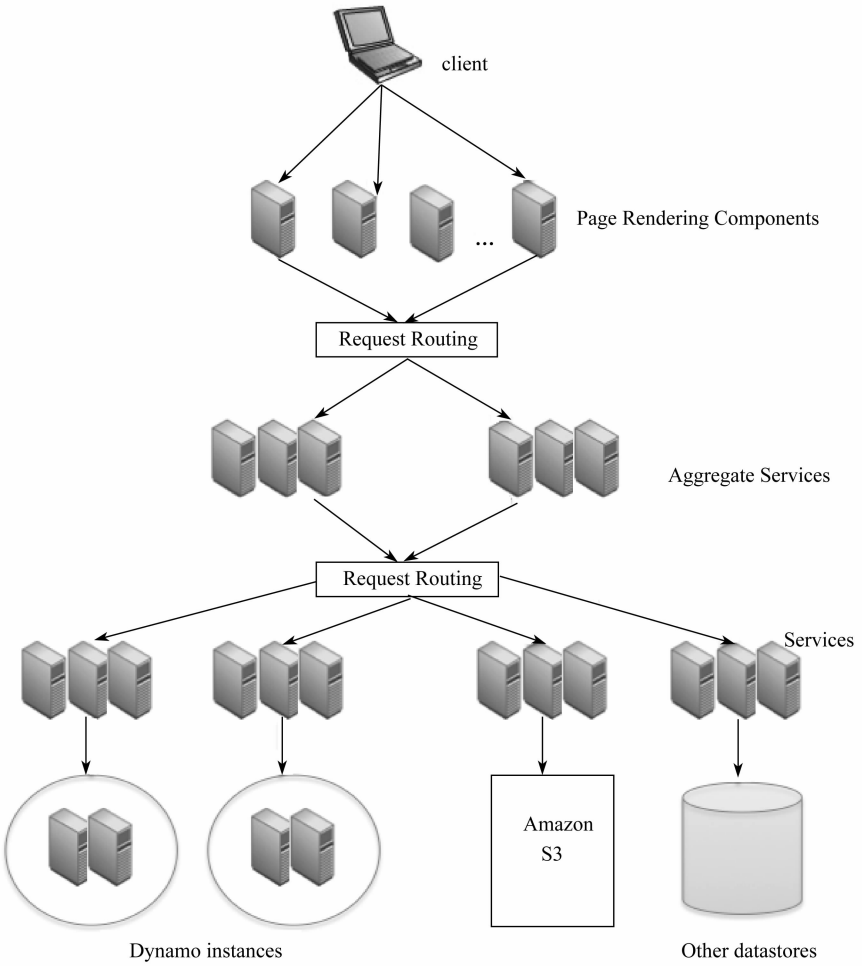

图 6　面向服务的 Amazon 平台[40]

Fig. 6　Service-Oriented Amazon Platform

Cassandra 同 Dynamo 一样通过 Gossip 跟踪各节点心跳信息，判断其异常与否. Cassandra 通过网络条件、服务器负载等复杂的检测机制判断是否宕机. 一旦检测到某一节点故障，原先对该节点的操作，会由其他节点替代，需要开启 Hinted Handoff 操作，若一个节点，宕机时间超过 1 h，hints 的数据将不会写入到其他节点. 因此在每一个节点上由 Node Tool 定期执行 Node Repair 确保数据的一致性. 在宕机节点恢复之后，也要执行 repair，帮助恢复数据[43].

Dynamo 采用虚拟节点技术改进了传统一致性 hash[44] 中由于节点的异构性带来的负载不均衡问题，将数据均匀地划分到各个节点上，保证系统的健壮性：每个节点根据性能差异分配多个 token，每个 token 对应一个虚拟节点，其处理能力基本相同. 存储数据时，按照 hash 值映射到的虚拟节点区域，最终存储在该虚拟节点对应的物理节点上. 假设 Dynamo 集群中原有三个节点，每个节点分配三个 token：node 1(1，4，7)，node 2(2，3，8)，node 3(0，5，9). 存数据时，先计算 key 的 hash 值，根据 hash 值将数据存放在对应 token 所在的节点上. 若此时增加一个节点 node 4，集群可能会将 node 1 和 node 3 的 token 1、token 5 迁移到

node 4,token 重新分配后的结果为:node 1(4,7),node 2(2,3,8),node 3(0,9),node 4(1,5),从而达到负载均衡的目标,见图 7.

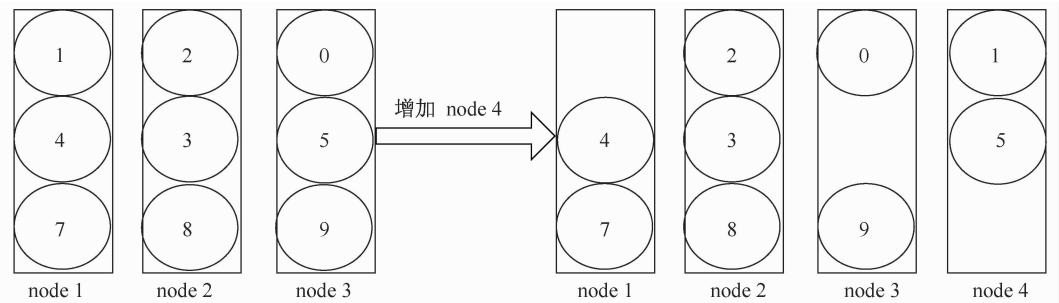

图 7 Dynamo 虚拟节点[27]

Fig. 7 Virtual nodes of Dynamo

改进后的一致性 hash,将负载较大的虚拟节点分配给性能较强的物理节点,将负载较小的虚拟节点分配给性能较弱的物理节点,最终达成动态负载均衡.

Cassandra 避免节点访问过载的容忍机制同 Dynamo 一样,来源于一致性 hash 算法,通过 Order-preserving hash function(保序 Hash)实现. Cassandra 的设计考虑到传统的一致性 hash 算法无视节点处理能力的不均衡和分配数据的不均匀的弊端,不同于 Dynamo 增加虚拟节点的做法,Cassandra 采用 Stoica 等人的的思想,分析 hash 环上各节点的负载信息,优先 routing 轻载节点减轻重载节点负担以均衡负载[45].

2.2.2 Dynamo 和 Cassandra 的一致性保持

在 Dynamo 中,最重要的是要保证写操作的高可用性(Always Writeable),Dynamo 牺牲部分的一致性,采用多副本冗余的方式来保证系统的高可用性. Dynamo 只保证最终一致性,即:若多个节点之间的更新顺序不一致,客户端可能读取不到预期的结果. Dynamo 中涉及三个重要参数 NWR,N 代表数据的备份数,W 代表成功写操作的最少节点数,R 代表成功读操作的最少节点数. Dynamo 中要求 $W+R>N$,以保证当不超过一台机器发生故障的时候,至少能读到一份有效的数据.

Cassandra 有一套自己的机制来保障最终一致性:(1) Read Repair. 在读数据时,系统会先读取数据副本,若发现不一致,则进行一致性修复. 根据读一致性等级不同,有不同的解决方案:当读一致性要求为 ONE 时,会立即返回用户最近的一份副本,后台执行 Read Repair,意味着可能第一次读不到最新的数据;当读一致性要求为 QUORUM 时,则在读取超过半数的一致性的副本后返回一份副本给客户端,剩余节点的一致性检查和修复则在后台执行;当读一致性为 ALL 时,则只有 Read Repair 完成后才能返回一份一致性的副本给客户端;(2) Anti-Entropy Node Repair. 通过 Node Tool 负责管理维护各个节点,由 Anti-Entropy 声称的 Merkle Tree 对比发现数据副本的不一致,通过 org. apache. cassandra. streaming 来进行一致性修复;(3) Hinted Handoff 作为实现最终一致性的优化措施,减少最终一致的时间窗口[46].

2.3 PNUTS

PNUTS 是由 Yahoo 公司推出的超大规模并发分布式数据库系统. PNUTS 提供哈希表和顺序表两种数据存储方式,可以对海量并发的更新和查询请求提供快速响应,它是一个

托管型、集中式管理的分布式系统，并且能够实现自动化的负载均衡和故障恢复，以减轻使用的复杂度[47]. PNUTS 的系统结构如图 8 所示：系统被划分为多个 region，每个 region 包含整套完整的组件，例如：tablet controller 维护活动节点的路由信息以判断节点失效与否；Yahoo! Message Broker(以下简称 YMB)是基于 topic 的订阅/发布系统，一旦数据的更新操作发布到 YMB 则被认为是提交了的. 在提交后的某个时间点，该更新操作被异步广播到不同的 region 和副本上. PNUTS 区别于其他 NoSQL 系统的特性是：没有采用传统的日志或归档数据实现复制保证高可用性，而是利用订阅/发布机制来实现.

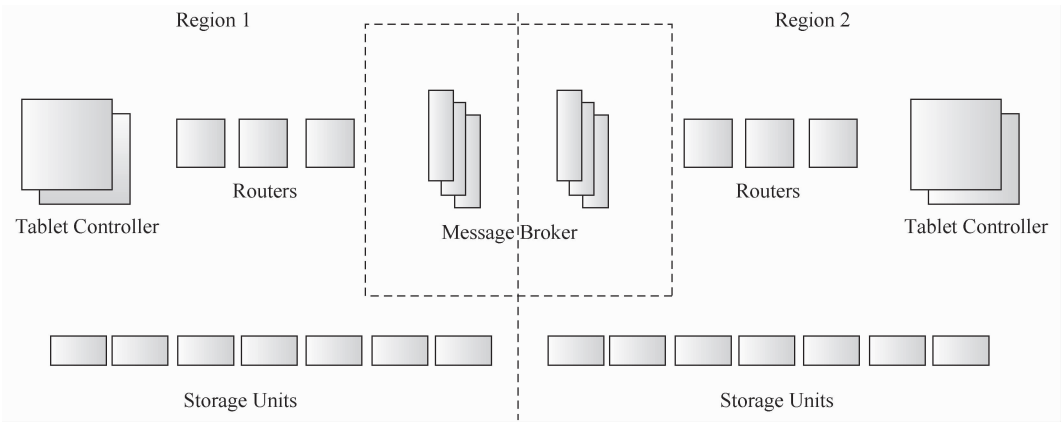

图 8　PNUTS 系统结构[47]

Fig. 8　Architecture of PNUTS

2.3.1　PNUTS 的容错机制

PNUTS 通过 YMB 防止在更新过程中的节点故障，从远程副本的拷贝实现故障恢复. YMB 保证在一个 Message Broker 机器宕机后，已发布的消息仍然会传递给 topic 订阅者，该功能通过将 message 记录在不同服务器上的多个磁盘上来实现. 所有的消息直到被系统确认已经写到数据库中后才会被 YMB 清洗.

PNUTS 中的故障恢复是指从其他的副本中复制已丢失的 tablet，其复制策略具体实现如下：首先，tablet controller 从 source tablet(一种特定的远程副本)申请一份拷贝；接着，发布检查点消息到 YMB，确保正在执行的更新操作都在 source tablet 上进行；最后，将 source tablet 复制到目标 region. 要实现这个恢复策略要求 tablet boundary 在副本间保持同步，并且所有 region 的 table 在同一时间分裂. 该策略的最大开销是将一个 tablet 从一个 region 迁移到另一个 region，为避免远程获取的开销，通常会创建 backup regions 就近维护一个 backup 副本.

PNUTS 拥有类似 Bigtable 均衡节点访问压力避免“热点”的容错机制，PNUTS 主要瓶颈是存储单元和 YMB 的磁盘访问量. 目前，用户还不能共享所有的组件，只能共享 routers 和 tablet controller，不同的用户会被分配到不同的存储单元和 YMB. PNUTS 的 tablet 代表的是数据表被水平切分成的一组组记录. tablet 分散在服务器上，每个服务器可能包含成百上千个 tablet. 在 PNUTS 中，一个 tablet 占用几百 M 或几个 G 的容量，包含几千条记录，tablet 被灵活地分配到不同的 server 上以均衡负载. 系统使用 n-bit hash 函数来得到 hash 值 H()，其中 $0 \leqslant H() \leqslant 2^n$. hash 空间$[0, 2^n]$被分裂为多个区间，每个区间对应一个 tablet.

要将一个 key 映射到一个 tablet 上,首先对这个 key 做 hash,得到该 key 的 H();然后搜索区间集;最后,采用二分查找定位到相应的闭合区间及对应的 tablet 和存储单元. PNUTS 也采用了顺序分裂的方式按照 key 或 H(key)来划分顺序表及其中的数据. 图 8 中 Routers 仅包含一个区间映射的缓存副本,这个映射为 tablet controller 所有,router 周期性地轮询 tablet controller 获取映射更新,当 tablet 空间到达阈值需要分裂时,tablet controller 便在存储单元间移动 tablet 以均衡负载或处理故障.

2.3.2 PNUTS 的一致性保持

PNUTS 的一致性模型介于即时一致性和最终一致性之间,被称为 *per-record timeline consistency*:一条记录所有的副本按照相同的顺序执行更新操作,如图 9 所示. timeline 上包括对主键的增、删、改操作. 对任意副本的读操作都会从这个 timeline 上返回一致的版本,并且副本总是沿着 timeline 向前移动. 该一致性模型实现过程如下:首先,指派一个副本作为 master,每条记录之间相互独立,互不干扰,对记录的所有操作转发给 master,被指派为 master 的副本是自适应不断变化的,这主要依据哪个副本接受大多数的写操作;其次,随着写操作的执行,记录的序列号递增,每条记录的序列号包括该记录的 generation(每一个新的 insert 操作)和 version(每一个 update 操作创建一个 version).

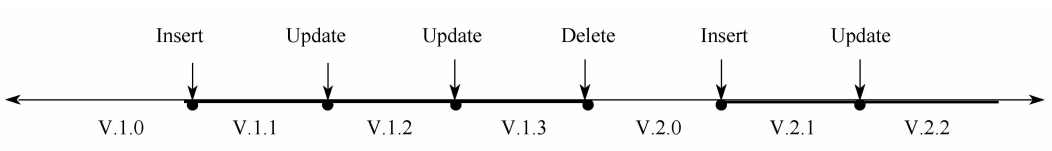

图 9 Per-record timeline 一致性[47]

Fig. 9 Per-record timeline consistency

3 讨 论

本文探讨了五种典型的 NoSQL 系统的容错机制及相关的一致性保持解决方案. 它们有各自的适用场景,因此不能简单地判断孰优孰劣,只有结合业务特性,才会产生高效的解决方案. 例如 Bigtable 和 Dynamo,后者的亮点是"无主"的架构,从而能避免单点故障,但并不代表 Dynamo 比 Bigtable 更优秀. Bigtable 的主控节点足够可靠,达到"4 个 9"的可靠性,足够满足通常的应用场景,因此可以采取简单的一个 master 主控节点的解决方案. Bigtable、HBase、Dynamo、Cassandra、PNUTS 的容错机制及一致性保持方案的异同点及自身主要的优缺点概括如表 1 所示.

面对海量数据的挑战,Bigtable、HBase、Dynamo、Cassandra、PNUTS 等优秀的 NoSQL 存储系统解决了高可靠性、可用性、和可扩展性的问题,这些面向互联网的应用满足横向扩展,面对用户量、访问量的急速增长的挑战,灵活提高负载的能力为学术界、工业界提供了新思路. 2011 年,"云计算"、"大数据"袭来,越来越多的企业趋之若鹜,大量企业级的应用也在发生革命. 越来越多的企业级应用要求更快的访问速度,内存计算和内存数据管理也成为热门的研究方向. 内存数据管理中列存储、多核多节点的分布式实现等本质与本文分析比较的五种典型 NoSQL 系统实现有诸多的契合点,本文从一定程度上为内存数据管理的研究提供经验指导.

表 1　NoSQL 系统的对比

Tab. 1　Comparison of NoSQL systems

	Bigtable	HBase	Dynamo	Cassandra	PNUTS
一致性	通过 Chubby 互斥锁机制保证强一致性	通过 ZooKeeper 保证最终一致性	*NRW* 保证最终一致性	*Read Repair*、*Anti-Entropy Node Repair*、*Hinted Handoff* 保证最终一致性	*per-record timeline* 一致性
容错机制	1) Chubby lock Service 对节点故障的恢复 2) 底层存储通过的 GFS 自动恢复	1) Heartbeat 机制检测故障 2) ZooKeeper 进行故障恢复	1) 通过 Gossip 感知心跳，检测故障 2) 数据回传 3) Merkel Tree 数据同步	1) 通过 Gossip 感知心跳，检测故障 2) Hinted Handoff 操作 3) 定期例行的在每一个节点上由 Node Tool 执行 Node Repair 进行数据恢复	1) 通过 YMB 以消息日志的方式防止节点故障 2) 远程副本拷贝实现故障恢复
优点	1) 数据模型简单，用户可以控制数据分布和格式 2) 良好的可扩展性 3) 只保证单挑记录的原子性，不支持事物，易于实现 4) 低成本、高可用性	1) 支持高并发读写操作 2) 自动切分数据以保障数据存储的水平扩展	1) 设计简单，代码复用率高 2) 可根据应用自由调整 NWR 策略 3) 不存在单点故障的危险	1) 根据应用场景的不同，可以灵活设置一致性和性能间的偏好 2) 不存在单点故障	1) 地理式分布 2) 高可用性：即使整个数据中心不可用都不会影响前段 web 访问
缺点	随机读取小型记录的性能比较差	1) 只能按照 Row key 查询 2) 存在单点故障的危险	1) 系统扩展性较差 2) 数据存储依赖哈希，无法直接执行 Mapreduce任务 3) 负载均衡策略需要预估机器规模，比较不可控	无法存储超大文件	只适合存储较小的记录，对流媒体等大型记录力不从心

[参　考　文　献]

[1] NoSQL matters. NoSQL DEFINITION[EB/OL]. [2014-06-08]. http://nosql-database.org.

[2] 百度百科. 淘宝网[EB/OL]. [2014-06-08]. http://baike.baidu.com/view/1590.htm?fromtitle=%E6%B7%98%E5%AE%9D&fromid=145661&type=syn

[3] 淘宝网品牌介绍[EB/OL]. [2014-06-08]. http://www.maigoo.com/maigoocms/special/services/170taobao.html.

[4] 张潇. “双 11”网购疯狂！天猫及淘宝成交额突破 350 亿[EB/OL]. [2014-06-08]. http://money.163.com/13/1112/16/9DGCT0BJ00253B0H.html.

[5] CONSTINE J. How Big Is Facebook's Data? 2.5 Billion Pieces Of Content And 500+ TerabytesIngested Every Day[EB/OL]. [2014-06-08]. http://techcrunch.com/2012/08/22/how-big-is-facebooks-data-2-5-billion-pieces-of-content-and-500-terabytes-ingested-every-day/.

[6] HENDERSON C. Flickr-Scalable Web Architectures: Common Patterns and Approaches[EB/OL]. [2014-06-08]. http://krisjordan.com/2008/09/16/cal-henderson-scalable-web-architectures-common-patterns-and-approaches.

[7] PLATTNER H, ZEIER A. In-memory Data Management: Technology and Applications[M]. Berlin: Springer, 2012.

[8] BREWER E A. Towards robust distributed systems (Invited Talk). ACM SIGACT-SIGOPS, 2000.

[9] GILBERT S, LYNCH N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services[C]. SIGACT News, 2002.

[10] BREWER E A. Pushing the cap: Strategies for consistency and availability[J]. IEEE Computer, 2012,42(2):23-29.

[11] AGRAWAL D, DAS S, ABBADI A E. Data Management in the Cloud Challenges and Opportunities[M]. California: Morgan & Claypool, 2013.

[12] VOGELS W. Eventually consistent[J]. ACM Queue, 2008,6(6):14-19.
[13] 何坤. 基于内存数据库的分布式数据库架构[J]. 程序员,2010(7):116-118.
[14] WIKIPEDIA. CAP theorem[EB/OL]. [2014-06-08]. http://en.wikipedia.org/wiki/CAP_theorem.
[15] COULOURIS G, DOLLIMORE J, KINDBERG T, et al. Distributed Systems: Concepts and Design[M]. 5th ed. [s.l.]:Addison-Wesley,2011.
[16] Chubby 总结[EB/OL]. [2014-06-08]. http://blog.csdn.net/ggxxkkll/article/details/7874465.
[17] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems[C]. OSDI, 2006.
[18] Google 利器之 Chubby[EB/OL]. [2014-06-08]. http://blog.csdn.net/historyasamirror/article/details/3870168.
[19] LAMPORT L. Paxos made simple[J]. ACM SIGACT News, 2001.
[20] GIFFORD D K. Weighted voting for replicated data[C]. SOSP, 1979.
[21] AVIZIENIS A. Design of fault-tolerant computers[C]. AFIPS, 1967.
[22] OZSU M T, VALDURIEZ P. Principles of Distributed Database Systems[M]. 3rd ed. [s.l.]:Springer,2011.
[23] GUERRAOUI R, SCHIPER A. Fault-Tolerance by Replication in Distributed System[EB/OL]. [2014-06-12]. http://link.springer.com/chapter/10.1007%2FBFb0013477#page-1.
[24] SAITO Y, SHAPIRO M. Optimistic Replication[C]. ACM Computing Surveys, 2005.
[25] BERNSTEIN P A, NEWCOMER E. Principles of Transaction Processing[M]. 2nd ed. Elsevier, 2009.
[26] 李磊. 分布式系统中容错机制性能优化技术研究[D]. 湖南:国防科技大学, 2007.
[27] 杨传辉. 大规模分布式存储系统原理解析与架构实践[M]. 北京:机械工业出版社, 2013.
[28] Distributed Algorithms in NoSQL Databases[EB/OL]. [2014-06-10]. http://vdisk.weibo.com/s/t3HPkX2GaIpf.
[29] CHANG F, DEAN J, GHEMAWAT S, et al. Bigtable: A Distributed Storage System for Structured Data[C]. OSDI, 2006.
[30] DEAN J. Building Large-Scale Internet Services[EB/OL]. [2014-06-19]. http://static.googleusercontent.com/media/research.google.com/zh-CN//people/jeff/SOCC2010-keynote-slides.pdf.
[31] HADOOP W T. The Definitive Guide[M]. O'Reilly, 2012.
[32] Apache ZooKeeper. What is ZooKeeper[EB/OL]. [2014-06-19]. http://zookeeper.apache.org/.
[33] Searchtb. HBase 技术介绍[EB/OL]. [2014-06-08]. http://www.searchtb.com/2011/01/understanding-hbase.html.
[34] GHEMAWAT S, Howard Gobioff, and Shun-Tak Leung. The Google File System[C]. SOSP, 2006.
[35] 康毅. HBase 大对象存储方案的设计与实现[D]. 南京大学, 2013.
[36] APACHE HBASE[EB/OL]. [2014-06-15]. http://hbase.apache.org /replication.html.
[37] 负载均衡技术大盘点[EB/OL]. [2014-06-10]. http://www.mmic.net.cn/news/865/9.htm.
[38] 陆嘉恒. Hadoop 实战[M]. 北京:机械工业出版社, 2011.
[39] SUBRAMANIYAN R. Gossip-based failure detection and consensus for terascale computing[EB/OL]. [2014-06-15]. http://etd.fcla.edu/UF/UFE0000799/subramaniyan_r.pdf.
[40] DECANDIA G, HASTORUN D, JAMPANI, et al. Dynamo: Amazon's highly available key-value store[C]. SOSP, 2007.
[41] LAKSHMAN A, MALIK P. Cassandra - A Decentralized Structure Storage System[C]. SIGMOD, 2008.
[42] MERKLE R. A digital signature based on a conventional encryption function[C]. Proceedings of CRYPTO, 1988.
[43] DATASTAX. Internode communications[EB/OL]. [2014-06-11]. http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architectureGossipAbout_c.html.
[44] KARGER D, LEHMAN E, LEIGHTONT, et al. Consistent hashing and random trees[C]//Procceding STOC'97 ACM Symposium, 1997:643-663.
[45] STOICA I, MORRIS R, KARGER D, et al. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications[C]//SIGCOMM'01. San Diego:ACM, 2001.
[46] DATASTAX. Configuring data consistency[EB/OL]. [2014-06-11]. http://link.springer.com/chapter/10.1007%2FBFb0013477#page-1.
[47] COOPER B F, RAMAKRISHNAN R, SRIVASTAVA U, et al. PNUTS: Yahoo!'s Hosted Data Serving Platform[C]. VLDB'08. Auckland: ACM, 2008.

(责任编辑 王善平)