第3章

分布式通信与协同

在大规模分布式系统中，为了高效地处理大量任务以及存储大量数据，通常需要涉及多个处理节点，需要在多个节点之间通信以及协同处理。高效的节点之间的通信以及节点之间的可靠协同技术是保证分布式系统正常运行的关键。

3.1 数据编码传输

3.1.1 数据编码概述

在分布式系统中需要处理大量的网络数据，为了加快网络数据的传输速度，通常需要对传输数据进行编码压缩，当然数据编码压缩传输技术也在其他电子信息领域中大量使用，由于数字化的多媒体信息尤其是数字视频、音频信号的数据量特别庞大，如果不对其进行有效的压缩难以得到实际的应用，因此数据编码压缩技术已成为当今数字通信、广播、存储和多媒体娱乐中的一项关键的共性技术。

数据压缩是以尽可能少的数码来表示信源所发出的信号，减少容纳给定的消息集合或数据采样集合的信号空间。这里讲的信号空间就是被压缩的对象，是指某信号集合所占的时域、空域和频域。信号空间的这几种形式是相互关联的，存储空间的减少意味着信号传输效率的提高，所占用带宽的节省。只要采取某种方法来减少某个信号空间就能够压缩数据。

一般来说，数据压缩主要是通过数据压缩编码来实现的。要想使编码有效，必须

建立相应的系统模型。在给定的模型下通过数据编码来消除冗余，大致有以下 3 种情况。

(1) 信源符号之间存在相关性。如果消除了这些相关性，就意味着数据压缩。例如，位图图像像素与像素之间的相关性，动态视频帧与帧之间的相关性。去掉这些相关性通常采用预测编码、变换编码等方法。

(2) 信源符号之间存在分布不等概性。根据不同符号出现的不同概率分别进行编码，概率大的符号用较短的码长编码，概率小的符号用较长的码长编码，最终使信源的平均码长达到最短。通常采用统计编码的方法。

(3) 利用信息内容本身的特点(如自相似性)。用模型的方法对需传输的信息进行参数估测，充分利用人类的视觉、听觉等特性，同时考虑信息内容的特性，确定并遴选出其中的部分内容(而不是全部内容)进行编码，从而实现数据压缩。通常采用模型基编码的方法。

目前比较认同的、常用的数据压缩的编码方法大致分为两大类。

(1) 冗余压缩法或无损压缩法。冗余压缩法或无损压缩法又称为无失真压缩法或熵编码法。这类压缩方法只是去掉数据中的冗余部分，并没有损失熵，而这些冗余数据是可以重新插入到原数据中的。也就是说，去掉冗余不会减少信息量，而且仍可原样恢复数据。因此，这类压缩方法是可逆的。

(2) 熵压缩法或有损压缩法。这类压缩法由于压缩了熵，也就损失了信息量，而损失的信息是不能恢复的。因此，在用门限值采样量化时，如果只存储门限内的数据，那么原来超过这个预置门限的数据将丢失。这种压缩方法虽然可压缩大量的信号空间，但那些丢失的实际样值不可能恢复，是不可逆的。也就是说，在用熵压缩法时数据压缩要以一定的信息损失为代价，而数据的恢复只能是近似的，应根据条件和要求在允许的范围内进行压缩。

3.1.2 LZSS 算法

LZSS 算法属于字典算法，是把文本中出现频率较高的字符组合做成一个对应的字典列表，并用特殊代码来表示这个字符。图 3-1 为字典算法原理图示。

LZSS 算法的字典模型使用自适应方式，基本的思路是搜索目前待压缩串是否在以前出现过，如果出现过，则利用前次出现的位置和长度来代替现在的待压缩串，输出该字符串的出现位置及长度；否则，输出新的字符串，从而起到压缩的目的。但是在实际使用过程中，由于被压缩的文件往往较大，一般使用“滑动窗口压缩”方式，也就是说将一个虚拟的、可以跟随压缩进程滑动的窗口作为术语字典。LZSS 算法最大的好处是压缩算法

的细节处理不同，只对压缩率和压缩时间有影响，不会影响到解压程序。LZSS 算法最大的问题是速度，每次都需要向前搜索到原文开头，对于较长的原文需要的时间是不可忍受的，这也是 LZSS 算法较大的一个缺点。

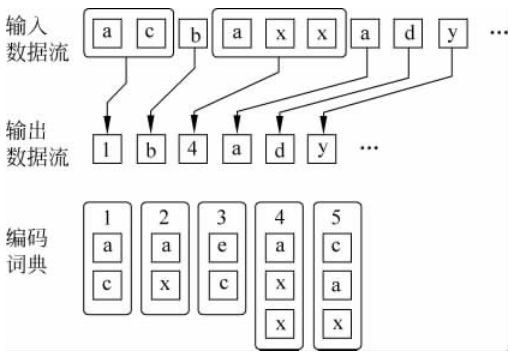

图 3-1　字典算法原理

3.1.3　Snappy 压缩库

Snappy 是在 Google 公司内部生产环境中被许多项目使用的压缩/解压缩的链接库，使用该库的软件包括 BigTable、MapReduce 和 RPC 等，Google 公司于 2011 年开源了该压缩/解压缩库。在 Intel 酷睿 i7 处理器上，在单核 64 位模式下，Snappy 的压缩速度大概可以达到 250MB/s 或者更快，解压缩可以达到大约 500MB/s 甚至更快。如此高的压缩速度是通过降低压缩率来实现的，因此其输出要比其他库大 20%～100%。Snappy 对于纯文本的压缩率为 1.5～1.7，对于 HTML 是 2～4，当然，对于 JPEG、PNG 和其他已经压缩过的数据的压缩率为 1.0。

Snappy 压缩库采用 C++实现，同时提供了多种其他语言的接口，包括 C、C＃、Go、Haskell 等。Snappy 是面向字节编码的 LZ77 类型压缩器。Snappy 采用的编码单元是字节(byte)，而不是比特(bit)，采用该压缩库压缩后的数据形成一个字节流的格式，格式如下：前面几个字节表示总体为压缩的数据流长度，采用小端方式(little-endian)存储，同时兼顾可变长度编码，每个字节的后面 7 位存储具体的数据，最高位用于表示下一个字节是否为同一个整数；剩下的字节用 4 种元素类型中的一种进行编码，元素类型在元素数据中的第一个字节，该字节的最后 2 位表示类型。

- 00。文本数据，属于未压缩数据，类型字节的高 6 位用于存储每个元素的数据内容长度。当数据内容超过 60 个字节时，采用额外的可变长编码方式存储数据。
- 01。数据长度用 3 位存储，偏移量用 11 位存储。紧接着类型字节后的第一个字节也用于存储偏移量。

- 10。类型字节中剩下的高 6 位用于存储数据长度，在类型字节后的两个字节用于存储数据的偏移量。
- 11。类型字节中剩下的高 6 位用于存储数据长度，数据偏移量存储在类型字节后的 4 个字节，偏移量采用小端方式存储数据。

3.2 分布式通信系统

分布式通信研究分布式系统中不同子系统或进程之间的信息交换机制。我们从各种大数据系统中归纳出 3 种最常见的通信机制：远程过程调用、消息队列和多播通信。其中，远程过程调用的重点是网络中位于不同机器上进程之间的交互；消息队列的重点是子系统之间的消息可靠传递；多播通信是实现信息的高效多播传递。这三者都是黏合子系统的有效工具，同时，它们对于减少大数据系统中构件之间的耦合、增强各自的独立演进有很大的帮助作用。

3.2.1 远程过程调用

远程过程调用(Remote Procedure Call，RPC)是一个计算机通信协议，通过该协议运行于一台计算机上的程序可以调用另一台计算机的子程序，而程序员无须额外地为这个交互编程。

通用的 RPC 框架都支持以下特性：接口描述语言、高性能、数据版本支持以及二进制数据格式。

Thrift 是由 Facebook 公司开发的远程服务调用框架，它采用接口描述语言定义并创建服务，支持可扩展的跨语言服务开发，所包含的代码生成引擎可以在多种语言中，如 C++、Java、Python、PHP、Ruby、Erlang、Perl、Haskell、C＃、Cocoa、Smalltalk 等，创建高效的、无缝的服务。其传输数据采用二进制格式，相对于 XML 和 JSON 体积更小，对于高并发、大数据量和多语言的环境更有优势。

Thrift 包含了一个完整的堆栈结构，用于构建客户端和服务器端。服务器包含用于绑定协议和传输层的基础架构，它提供阻塞、非阻塞、单线程和多线程的模式运行在服务器上，可以配合服务器/容器一起运行，可以和现有的服务器/容器无缝结合。

其使用流程大致如下。

首先使用 IDL 定义消息体以及 RPC 函数调用接口。使用 IDL 可以在调用方和被调用方解耦，比如调用方可以使用 C++，被调用方可以使用 Java，这样给整个系统带来了极

大的灵活性。

然后使用工具根据 IDL 定义文件生成指定编程语言的代码。

最后即可在应用程序中连接使用上一步生成的代码。对于 RPC 来说，调用方和被调用方同时引入后即可实现透明的网络访问。

3.2.2　消息队列

消息队列也是设计大规模分布式系统时经常采用的中间件产品。分布式系统构件之间通过传递消息可以解除相互之间的功能耦合，这样减轻了子系统之间的依赖，使得各个子系统或者构建可以独立演进、维护或重用。消息队列是在消息传递过程中保存消息的容器或中间件，其主要目的是提供消息路由并保障消息可靠传递。

下面通过 Linkedin 开源的分布式消息系统 Kafka 介绍消息队列系统的整体设计思路。

Kafka 采用 Pub-Sub 机制，具有极高的消息吞吐量、较强的可扩展性和高可用性，消息传递延迟低，能够对消息队列进行持久化保存，且支持消息传递的“至少送达一次”语义。

一个典型的 Kafka 集群中包含若干 producer、若干 broker、若干 consumer group，以及一个 ZooKeeper 集群。Kafka 通过 ZooKeeper 管理集群配置，选举 leader，以及在 consumer group 发生变化时进行 rebalance。producer 使用 push 模式将消息发布到 broker，consumer 使用 pull 模式从 broker 订阅并消费消息。

作为一个消息系统，Kafka 遵循了传统的方式，选择由 producer 向 broker push 消息并由 consumer 向 broker pull 消息。push 模式很难适应消费速率不同的 consumer，因为消息发送速率是由 broker 决定的。push 模式的目标是尽可能以最快的速度传递消息，但是这样很容易造成 consumer 来不及处理消息，典型的表现就是拒绝服务以及网络阻塞。pull 模式可以根据 consumer 的消费能力以适当的速率消费信息。

3.2.3　应用层多播通信

分布式系统中的一个重要的研究内容是如何将数据通知到网络中的多个接收方，这一般被称为多播通信。与网络协议层的多播通信不同，这里介绍的是应用层多播通信。Gossip 协议就是常见的应用层多播通信协议，与其他多播协议相比，其在信息传递的健壮性和传播效率方面有较好的折中效果，使其在大数据领域中得以广泛使用。

Gossip 协议也被称为“感染协议”(Epidemic Protocol),用来尽快地将本地更新数据通知到网络中的所有其他节点。其具体更新模型又可以分为 3 种:全通知模型、反熵模型和散步谣言模型。

在全通知模型中,当某个节点有更新消息时立即通知所有其他节点;其他节点在接收到通知后判断接收到的消息是否比本地消息要新,如果是,则更新本地数据,否则,不采取任何行为。反熵模型是最常用的“Gossip 协议”,之所以称之为“反熵”,是因为“熵”是用来衡量系统混乱无序程度的指标,熵越大说明系统越无序。系统中更新的信息经过一定轮数的传播后,集群内的所有节点都会获得全局最新信息,所以系统变得越来越有序,这就是“反熵”的含义。

在反熵模型中,节点 P 随机选择集群中的另一个节点 Q,然后与 Q 交换更新信息;如果 Q 信息有更新,则类似 P 一样传播给任意其他节点(此时 P 也可以再传播给其他节点),这样经过一定轮数的信息交换,更新的信息就会快速传播到整个网络节点。

散步谣言模型与反熵模型相比增加了传播停止判断。即如果节点 P 更新了数据,则随机选择节点 Q 交换信息;如果节点 Q 已经从其他节点处得知了该更新,那么节点 P 降低其主动通知其他节点的概率,直到一定程度后,节点 P 停止通知行为。散布谣言模型能够快速传播变化,但不能保证所有节点都能最终获得更新。

3.2.4 阿里云夸父 RPC 系统

在分布式系统中,不同计算机之间只能通过消息交换的方式进行通信。显式的消息通信必须通过 Socket 接口编程,而 RPC 可以隐藏显式的消息交换,使得程序员可以像调用本地函数一样来调用远程的服务。

夸父(Kuafu)是飞天平台内核中负责网络通信的模块,它提供了一个 RPC 的接口,简化编写基于网络的分布式应用。夸父的设计目标是提供高可用(7×24 小时)、大吞吐量(Gigabyte)、高效率、易用(简明 API、多种协议和编程接口)的 RPC 服务。

RPC 客户端(RPC Client)通过 URI 指定请求需要发送的 RPC 服务端(RPC Server)的地址,目前夸父支持两种协议形式。

- TCP。例如 tcp://fooserver01:9000。
- Nuwa。例如 nuwa://nuwa01/FooServer。

与用流(Stream)传输的 TCP 通信相比,夸父通信是以消息(Message)为单位的,支持多种类型的消息对象,包括标准字符串 std::string 和基于 std::map 实现的若干 string 键-值对。

夸父 RPC 同时支持异步(asynchronous)和同步(synchronous)的远程过程调用形式。

(1) 异步调用。RPC 函数调用时不等接收到结果就会立即返回,用户必须通过显式调用接收函数取得请求结果。

(2) 同步调用。RPC 函数调用时会等待,直到接收到结果才返回。在实现中,同步调用是通过封装异步调用来实现的。

在夸父的实现中,客户端程序通过 UNIX Domain Socket 与本机上的一个夸父代理(Kuafu Proxy)连接,不同计算机之间的夸父代理会建立一个 TCP 连接。这样做的好处是可以更高效地使用网络带宽,系统可以支持上千台计算机之间的互联需求。此外,夸父利用女娲来实现负载均衡;对大块数据的传输作了优化;与 TCP 类似,夸父代理之间还实现了发送端和接收端的流控(Flow Control)机制。

3.2.5　Hadoop IPC 的应用

这里以 Hadoop 中的 RPC 框架 Hadoop IPC 为基础讲述 RPC 框架在大数据系统中的应用。Hadoop 系统包括 Hadoop Common、Hadoop Distributed File System、Hadoop MapReduce 几个重要的组成部分,其中,Hadoop Common 用于提供整个 Hadoop 公共服务,包括 Hadoop IPC。在 Hadoop 系统中,Hadoop IPC 为 HDFS、MapReduce 提供了高效的 RPC 通信机制,在 HDFS 中,DFSClient 模块需要与 NameNode 模块通信、DFSClient 模块需要与 DataNode 模块通信、MapReduce 客户端需要与 JobTracker 通信,Hadoop IPC 为这些模块之间的通信提供了一种便利的方式。

目前实现的 Hadoop IPC 具有采用 TCP 方式连接、支持超时、缓存等特征。Hadoop IPC 采用的是经典的 C/S 结构。

Hadoop IPC 的 Server 端相对比较复杂,包括 Listener、Reader、Handler 和 Responder 等多种类型的线程,Listener 用于侦听来自 IPC Client 端的连接,同时也负责管理与 Client 端之间的连接,包括 Client 端超时需要删除连接;Reader 线程用于读取来自 Client 端的数据,Handler 线程用于处理来自 Client 端的请求,执行具体的操作;Responder 线程用于返回处理结果给 Client 端。一般配置是一个 Listener、多个 Reader、多个 Handler 和一个 Responder。Hadoop IPC 的组成如图 3-2 所示。

执行 HDFS 读文件操作,首先 DFSClient 利用 Hadoop IPC 框架发起一次 RPC 请求给 NameNode,获取 DataBlock 信息。

在执行 HDFS 数据恢复操作的时候,DFSClient 需要执行 recoverBlock RPC 操作,发送该请求到 DataNode 节点上。

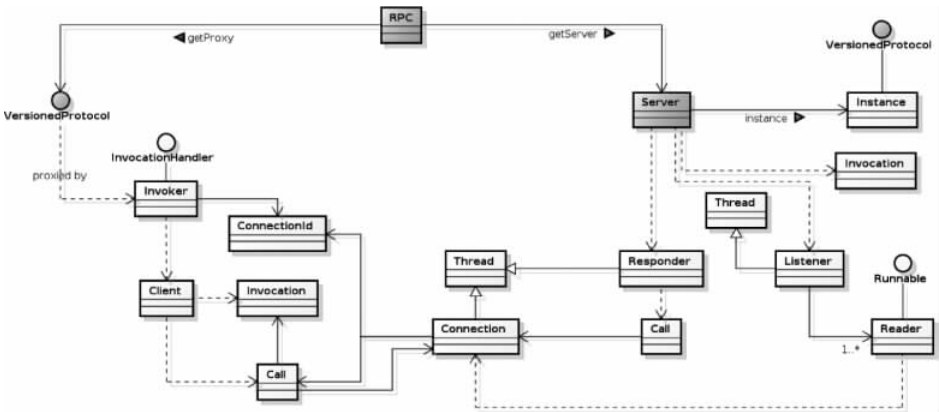

图 3-2 Hadoop IPC 的组成

3.3 分布式协同系统

当前的大规模分布式系统涉及大量的机器，这些机器之间需要进行大量的网络通信以及各个节点之间的消息通信协同。为了减少分布式系统中这些工作的重复开发，解耦出分布式协同系统，有效地提高了分布式计算、分布式存储等系统的开发速度。

3.3.1 Chubby 锁服务

Chubby 是 Google 公司研发的针对分布式系统协调管理的粗粒度服务，一个 Chubby 实例大约可以负责 1 万台 4 核 CPU 机器之间对资源的协同管理。这种服务的主要功能是让众多客户端程序进行相互之间的同步，并对系统环境或资源达成一致的认知。

Chubby 的理论基础是 Paxos(一致性协议)，Paxos 是在完全分布式环境下不同客户端能够通过交互通信并投票对于某个决定达成一致的算法。Chubby 以此为基础，但是也进行了改造，Paxos 是完全分布的，没有中心管理节点，需要通过多轮通信和投票来达成最终的一致，所以效率低；Chubby 出于对系统效率的考虑，增加了一些中心管理策略，在达到同一目标的情况下改善了系统效率。

Chubby 的设计目标基于以下几点：高可用性、高可靠性、支持粗粒度的建议性锁服务、支持小规模文件直接存储。这些当然是用高性能与存储能力折中而来的。

图 3-3 是 Google 论文中描述的 Chubby 的整体架构，可以容易地看出 Chubby 共有 5 台服务器，其中一个是主服务器，客户端与服务器之间使用 RPC 交互。那么，其他服务器是干什么的？它们纯粹是作为主服务器不可用后的替代品。而 ZooKeeper 的多余服

务器均是提供就近服务的，也就是服务器会根据地理位置与网络情况来选择对哪些客户端给予服务。

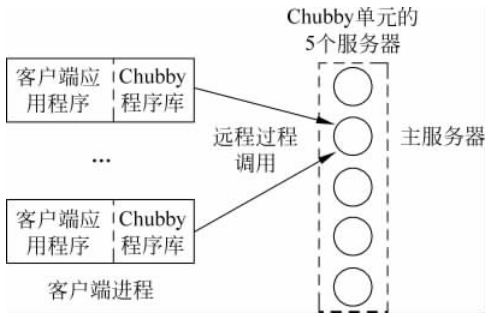

图 3-3　Chubby 整体架构

Chubby 单元中的主服务器由所有服务器选举推出，但是并非从始至终一直都由其担任这一角色，它是有“任期”的，即 Master Lease，一般长达几秒。如果无故障发生，一般系统尽量将“租约”交给原先的主服务器，否则可以通过重新选举得到一个新的全局管理服务器，这样就实现了主服务器的自动切换。

客户端通过嵌入的库程序，利用 RPC 通信和服务器进行交互，对 Chubby 的读/写请求都由主服务器负责。主服务器遇到数据更新请求后会更改在内存中维护的管理数据，通过改造的 Paxos 协议通知其他备份服务器对相应的数据进行更新操作，并保证在多副本环境下的数据一致性；当多数备份服务器确认更新完成后，主服务器可以认为本次更新操作正确完成。其他所有备份服务器只是同步管理数据到本地，保持数据和主服务器完全一致。通信协议如图 3-4 所示。

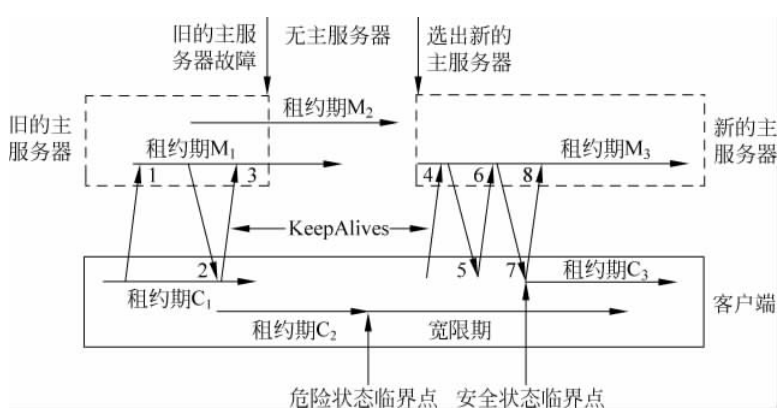

图 3-4　Client 与 Chubby 的通信

KeepAlive 是周期性发送的一种消息，它有两方面的功能：延长租约有效期，携带事件信息告诉客户端更新。事件包括文件内容的修改、子节点的增删改、Master 出错等。

在正常情况下，租约会由 KeepAlive 一直不断延长。如果 C_1 在未用完租约期时发现还需使用，便发送锁请求给 Master，Master 给它 Lease-M_1；C_2 在过了租约期后，发送锁请求给 Master，可是未收到 Master 的回答。其实此刻 Master 已经挂了，于是 Chubby 进入宽限期，在这期间 Chubby 要选举出新的 Master。Google 论文里对于这段时期有一个更形象的名字——Grace Period。在选举出 Master 后，新的主服务器下令前主服务器发的 Lease 失效，大家必须申请一份新的。然后 C_2 获得了 Lease-M_2。C_3 又恢复到正常情况。在图 3-4 中 4、5、6、7、8 是通过 Paxos 算法选举 Master 的颤抖期。在此期间最有可能产生问题，Amazon 的分布式服务就曾因此宕机，导致很长时间服务不可用。

3.3.2 ZooKeeper

ZooKeeper 是 Yahoo! 公司开发的一套开源高吞吐分布式协同系统，目前已经在各种 NoSQL 数据库及诸多开源软件中获得广泛使用。分布式应用中的各节点可以通过 ZooKeeper 这个第三方来确保双方的同步，比如一个节点是发送，另一个节点是接收，但发送节点需要确认接收节点成功收到这个消息，因而就可以通过与一个可靠的第三方交互来获取接收节点的消息接收状态。

ZooKeeper 也是由多台同构服务器构成的一个集群，共用信息存储在集群系统中。共用信息采用树形结构来存储，用户可以将其看作一个文件系统，只是这些文件是一直存放在内存中的，文件存储容量受到内存的限制。

既然 ZooKeeper 可以被看作一个文件系统，那么它就具有文件系统相应的功能，只是在 ZooKeeper 和文件系统中功能的叫法不同。ZooKeeper 提供创建节点、删除节点、创建子节点、获取节点内容等功能。

ZooKeeper 服务由若干台服务器构成，每台服务器内存中维护相同的树形数据结构。其中的一台通过 ZAB 原子广播协议选举作为主服务器，其他的作为从服务器。客户端可以通过 TCP 协议连接任意一台服务器，如果是读操作请求，任意一个服务器都可以直接响应请求；如果是写数据操作请求，则只能由主服务器来协调更新操作。Chubby 在这一点上与 ZooKeeper 不同，所有的读/写操作都由主服务器完成，从服务器只是用于提高整个协调系统的可用性。

在带来高吞吐量的同时，ZooKeeper 的这种做法也带来了潜在的问题：客户端可能读到过期的数据。因为即使主服务器已经更新了某个内存数据，但是 ZAB 协议还未能将其广播到从服务器。为了解决这一问题，在 ZooKeeper 的接口 API 函数中提供了 Sync 操作，应用可以根据需要在读数据前调用该操作，其含义是接收到 Sync 命令的从服务器从主服务器同步状态信息，保证两者完全一致。

3.3.3 阿里云女娲协同系统

女娲(Nuwa)系统为飞天提供高可用的协调服务(Coordination Service),是构建各类分布式应用的核心服务,它的作用是采用类似文件系统的树形命名空间来让分布式进程互相协同工作。例如,当集群变更导致特定的服务被迫改变物理运行位置时,如服务器或者网络故障、配置调整或者扩容时,借助女娲系统可以使其他程序快速定位到该服务新的接入点,从而保证了整个平台的高可靠性和高可用性。

女娲系统基于类 Paxos 协议,由多个女娲 Server 以类似文件系统的树形结构存储数据,提供高可用、高并发用户请求的处理能力。

女娲系统的目录表示一个包含文件的集合。与 UNIX 中的文件路径一样,女娲中的路径是以"/"分割的,根目录(Root entry)的名字是"/",所有目录的名字都是以"/"结尾的。与 UNIX 文件路径的不同之处在于:女娲系统中的所有文件或目录都必须使用从根目录开始的绝对路径。由于女娲系统的设计目的是提供协调服务,而不是存储大量数据,所以每个文件的内容(Value)的大小被限制在 1MB 以内。在女娲系统中,每个文件或目录都保存有创建者的信息。一旦某个路径被用户创建,其他用户就可以访问和修改这个路径的值(即文件内容或目录包含的文件名)。

女娲系统支持 Publish/Subscribe 模式,其中一个发布者、多个订阅者(One Publisher/Many Subscriber)的模式提供了基本的订阅功能;另外,还可用通过多个发布者、多个订阅者(Many Publisher/Many Subscriber)的模式提供分布式选举(Distributed Election)和分布式锁的功能。

另外一个使用女娲系统来实现负载均衡的例子:提供某一服务的多个节点,在服务启动的时候在女娲系统的同一目录下创建文件,例如 server1 创建文件 nuwa://cluster/myservice/server1,server2 在同一目录下创建 nuwa://cluster/myservice/server2。当客户端使用远程过程调用时首先列举女娲系统服务中 nuwa://cluster/myservice 目录下的文件,这样就可以获得 server1 和 server2,客户端随后可以从中选择一个节点发出自己的请求,从而实现负载均衡。

3.3.4 ZooKeeper 在 HDFS 高可用方案中的使用

HDFS 由 3 个模块构成,分别包括 Client、NameNode 和 DataNode。NameNode 负责管理所有的 DataNode 节点,保存 block 和 DataNode 之间的对应信息,Client 读取文件和写入文件都需要 NameNode 节点的参与,因此 NameNode 发挥着至关重要的作用。在

当前设计中，NameNode 是单节点方式，存在单点故障问题，即 NameNode 节点宕机之后 HDFS 无法再对外提供数据存储服务，需要设计一种 HDFS NameNode 节点的高可用方法。总体来讲，维护 HDFS 高可用基于以下两个目的：

（1）在出现 NameNode 节点故障时 HDFS 仍然可以对外提供数据的读取和写入服务。

（2）HDFS 会出现版本的更新迭代，以保证 HDFS 在更新过程中仍然可以对外提供服务。

HDFS 为了实现上述目的，采用的方式是再提供一个额外的 NameNode 节点，以此达到 HDFS 的高可用目的。在使用过程中部署两个 NameNode 节点，一个 NameNode 节点为 Active 节点，另一个 NameNode 节点是 Standby 节点。在正常情况下，Active 的 NameNode 节点服务正常的请求，一旦出现 Active NameNode 节点故障，则 Standby NameNode 节点切换变成 Active 节点，然后这个新的 Active NameNode 继续提供 NameNode 的功能，使 HDFS 可以继续正常工作。但是为了保证上述过程正常运行，需要解决以下问题：

（1）Standby 如何知道 Active 节点出现故障无法正常服务，需要探测系统何时出现故障。

（2）当出现 Active NameNode 节点故障时，多个 Standby NameNode 节点如何选择一个新的 Active NameNode 节点。

一种解决上述问题的 HDFS 高可用方法是采用 ZK Failover Controller 的方法，具体结构如图 3-5 所示。

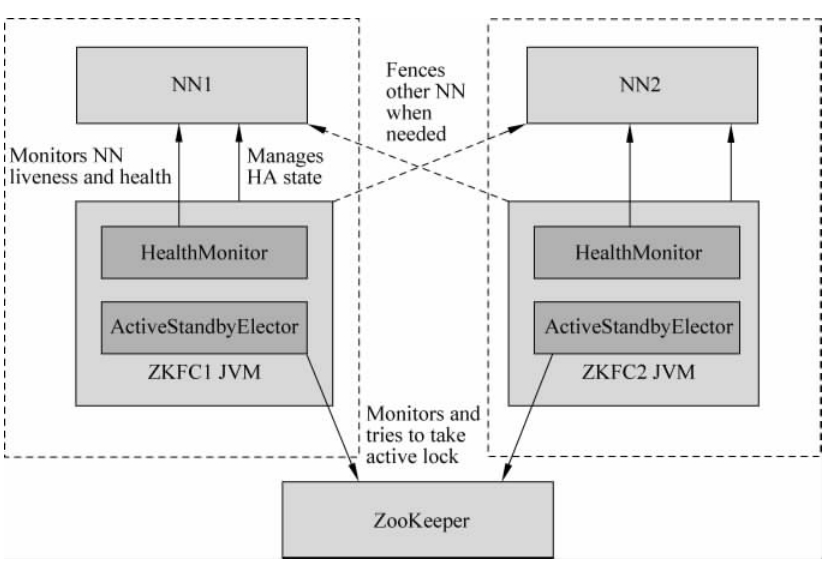

图 3-5　基于 ZooKeeper 的 HDFS 高可用方法

采用 ZK(ZooKeeper)设计 HDFS 高可用方案基于以下几点：

(1) ZooKeeper 提供了小规模的任意数据信息的强一致性。

(2) 可以在 ZooKeeper 集群中创建一个临时 znode 节点，当创建该 znode 节点的 Client 失效时，该临时 znode 节点会自动删除。

(3) 能够监控 ZooKeeper 集群中的一个 znode 节点的状态发生改变，并被异步通知。

上述设计的基于 ZK 的 HDFS 高可用方法由 ZKFC、HealthMonitor、ActiveStandbyElector 几个主要部分组成。

(1) HealthMonitor 是一个线程，用于监控本地 NameNode 的状态信息，维持一个状态信息的视图，监控采用 RPC 方式。当状态信息发生改变时，通过 callback 接口方式发送消息给 ZKFC。

(2) ActiveStandbyElector 主要用于和 ZooKeeper 进行协调，ZKFC 与它通信主要由两个函数调用，分别是 joinElection 和 quitElection。

(3) ZKFailoverController 订阅来自 ActiveStandbyElector 和 HealthMonitor 的消息，同时管理 NameNode 的状态。

整体运行过程如下：启动的时候初始化 HealthMonitor 去监控本地 NameNode 节点，同时用 ZooKeeper 信息来初始化 ActiveStandbyElector，不立即把该 NameNode 节点加入选举。同时，随着 ActiveStandbyElector 和 HealthMonitor 状态的改变，ZKFC 做出对应的响应。

3.4　习题

1. 简述数据编码传输的好处。
2. 简要介绍 Snappy 压缩库，包括功能和数据格式。
3. 简要介绍 Chubby 的工作原理。
4. 简述 ZooKeeper 在 HDFS 高可用方案中发挥作用的理由。