

第1章 绪 论

很少有一种技术能够像“云计算”这样，在短短的两三年间就产生巨大的影响力。Google、Amazon、IBM 和微软等 IT 巨头们以前所未有的速度和规模推动云计算技术和产品的普及，一些学术活动迅速将云计算提上议事日程。一年前支持者和反对者还在喋喋不休地争论，而如今业界已对云计算高度认同。那么，云计算到底是什么？发展现状如何？它的实现机制是什么？它与网格计算是什么关系？本章将分析这些问题，目的是帮助读者对云计算形成一个初步认识。

1.1 云计算的概念

云计算（Cloud Computing）是在 2007 年第 3 季度才诞生的新名词，但仅仅过了半年多，其受到关注的程度就超过了网格计算（Grid Computing），而且关注度至今一直居高不下，如图 1-1 所示。

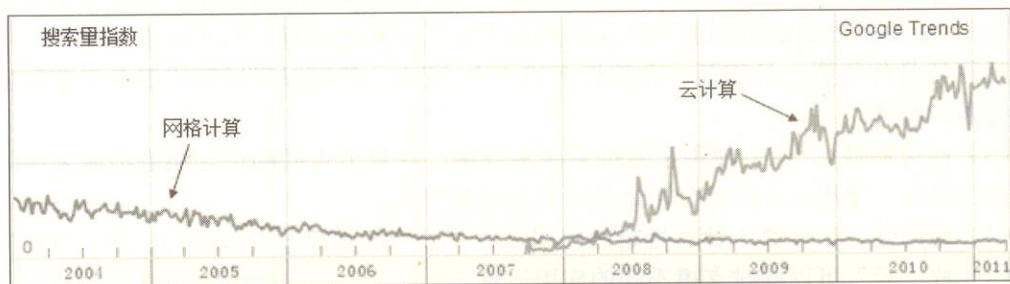


图 1-1 云计算和网格计算在 Google 中的搜索趋势

然而，对于到底什么是云计算，至少可以找到 100 种解释，目前还没有公认的定义。本书给出一种定义，供读者参考。

云计算是一种商业计算模型，它将计算任务分布在大量计算机构成的资源池上，使用户能够按需获取计算力、存储空间和信息服务^{[1][2]}。

这种资源池称为“云”。“云”是一些可以自我维护 and 管理的虚拟计算资源，通常是一些大型服务器集群，包括计算服务器、存储服务器和宽带资源等。云计算将计算资源集中起来，并通过专门软件实现自动管理，无需人为参与。用户可以动态申请部分资源，支持各种应用程序的运转，无需为烦琐的细节而烦恼，能够更加专注于自己的业务，有利于提高效率、降低成本和技术创新。云计算的核心理念是资源池，这与早在 2002 年就提出的网格计算池（Computing Pool）的概念非常相似^{[3][4]}。网格计算池将计算和存储资源虚拟成为一个可以任意组合分配的集合，池的规模可以动态扩展，分配给用户的处理能力可以动

态回收重用。这种模式能够大大提高资源的利用率,提升平台的服务质量。

之所以称为“云”,是因为它在某些方面具有现实中云的特征:云一般都较大;云的规模可以动态伸缩,它的边界是模糊的;云在空中飘忽不定,无法也无需确定它的具体位置,但它确实存在于某处。之所以称为“云”,还因为云计算的鼻祖之一 Amazon 公司将大家曾经称为网格计算的东西,取了一个新名称“弹性计算云”(Elastic Computing Cloud),并取得了商业上的成功。

有人将这种模式比喻为从单台发电机供电模式转向了电厂集中供电的模式。它意味着计算能力也可以作为一种商品进行流通,就像煤气、水和电一样,取用方便,费用低廉。最大的不同在于,它是通过互联网进行传输的。

云计算是并行计算(Parallel Computing)、分布式计算(Distributed Computing)和网格计算(Grid Computing)的发展,或者说是这些计算科学概念的商业实现。云计算是虚拟化(Virtualization)、效用计算(Utility Computing)、将基础设施作为服务 IaaS (Infrastructure as a Service)、将平台作为服务 PaaS (Platform as a Service) 和将软件作为服务 SaaS (Software as a Service) 等概念混合演进并跃升的结果。

从研究现状上看,云计算具有以下特点。

(1) 超大规模。“云”具有相当的规模,Google 云计算已经拥有 100 多万台服务器,Amazon、IBM、微软和 Yahoo 等公司的“云”均拥有几十万台服务器。“云”能赋予用户前所未有的计算能力。

(2) 虚拟化。云计算支持用户在任意位置、使用各种终端获取服务。所请求的资源来自“云”,而不是固定的有形的实体。应用在“云”中某处运行,但实际上用户无需了解应用运行的具体位置,只需要一台笔记本或一个 PDA,就可以通过网络服务来获取各种能力超强的服务。

(3) 高可靠性。“云”使用了数据多副本容错、计算节点同构可互换等措施来保障服务的高可靠性,使用云计算比使用本地计算机更加可靠。

(4) 通用性。云计算不针对特定的应用,在“云”的支撑下可以构造出千变万化的应用,同一片“云”可以同时支撑不同的应用运行。

(5) 高可伸缩性。“云”的规模可以动态伸缩,满足应用和用户规模增长的需要。

(6) 按需服务。“云”是一个庞大的资源池,用户按需购买,像自来水、电和煤气那样计费。

(7) 极其廉价。“云”的特殊容错措施使得可以采用极其廉价的节点来构成云;“云”的自动化管理使数据中心管理成本大幅降低;“云”的公用性和通用性使资源的利用率大幅提升;“云”设施可以建在电力资源丰富的地区,从而大幅降低能源成本。因此“云”具有前所未有的性能价格比。Google 中国区前总裁李开复称,Google 每年投入约 16 亿美元构建云计算数据中心,所获得的能力相当于使用传统技术投入 640 亿美元,节省了 40 倍的成本。因此,用户可以充分享受“云”的低成本优势,需要时,花费几百美元、一天时间就能完成以前需要数万美元、数月时间才能完成的数据处理任务。

云计算按照服务类型大致可以分为三类:将基础设施作为服务 IaaS、将平台作为服务 PaaS 和将软件作为服务 SaaS,如图 1-2 所示。

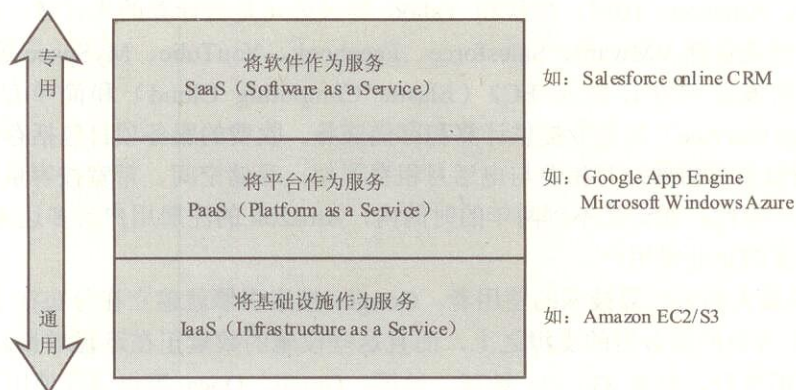


图 1-2 云计算的服务类型

IaaS 将硬件设备等基础资源封装成服务供用户使用，如 Amazon 云计算 AWS (Amazon Web Services) 的弹性计算云 EC2 和简单存储服务 S3。在 IaaS 环境中，用户相当于在使用裸机和磁盘，既可以让它运行 Windows，也可以让它运行 Linux，因而几乎可以做任何想做的事情，但用户必须考虑如何才能让多台机器协同工作起来。AWS 提供了在节点之间互通消息的接口简单队列服务 SQS (Simple Queue Service)。IaaS 最大的优势在于它允许用户动态申请或释放节点，按使用量计费。运行 IaaS 的服务器规模达到几十万台之多，用户因而可以认为能够申请的资源几乎是无限的。同时，IaaS 是由公众共享的，因而具有更高的资源使用效率。

PaaS 对资源的抽象层次更进一步，它提供用户应用程序的运行环境，典型的如 Google App Engine。微软的云计算操作系统 Microsoft Windows Azure 也可大致归入这一类。PaaS 自身负责资源的动态扩展和容错管理，用户应用程序不必过多考虑节点间的配合问题。但与此同时，用户的自主权降低，必须使用特定的编程环境并遵照特定的编程模型。这有点像在高性能集群计算机里进行 MPI 编程，只适用于解决某些特定的计算问题。例如，Google App Engine 只允许使用 Python 和 Java 语言、基于称为 Django 的 Web 应用框架、调用 Google App Engine SDK 来开发在线应用服务。

SaaS 的针对性更强，它将某些特定应用软件功能封装成服务，如 Salesforce 公司提供的在线客户关系管理 CRM (Client Relationship Management) 服务。SaaS 既不像 PaaS 一样提供计算或存储资源类型的服务，也不像 IaaS 一样提供运行用户自定义应用程序的环境，它只提供某些专门用途的服务供应用调用。

需要指出的是，随着云计算的深化发展，不同云计算解决方案之间相互渗透融合，同一种产品往往横跨两种以上类型。例如，Amazon Web Services 是以 IaaS 发展的，但新提供的弹性 MapReduce 服务模仿了 Google 的 MapReduce，简单数据库服务 SimpleDB 模仿了 Google 的 Bigtable，这两者属于 PaaS 的范畴，而它新提供的电子商务服务 FPS 和 DevPay 以及网站访问统计服务 Alexa Web 服务，则属于 SaaS 的范畴。

1.2 云计算发展现状

由于云计算是多种技术混合演进的结果，其成熟度较高，又有大公司推动，发展极为

迅速。Google、Amazon、IBM、微软和 Yahoo 等大公司云计算的先行者。云计算领域的众多成功公司还包括 VMware、Salesforce、Facebook、YouTube、MySpace 等。

Amazon 研发了弹性计算云 EC2 (Elastic Computing Cloud) 和简单存储服务 S3 (Simple Storage Service) 为企业提供计算和存储服务。收费的服务项目包括存储空间、带宽、CPU 资源以及月租费。月租费与电话月租费类似，存储空间、带宽按容量收费，CPU 根据运算量时长收费。在诞生不到两年的时间内，Amazon 的注册用户就多达 44 万人，其中包括为数众多的企业级用户。

Google 是最大的云计算技术的使用者。Google 搜索引擎就建立在分布在 200 多个站点、超过 100 万台的服务器的支撑之上，而且这些设施的数量正在迅猛增长。Google 的一系列成功应用平台，包括 Google 地球、地图、Gmail、Docs 等也同样使用了这些基础设施。采用 Google Docs 之类的应用，用户数据会保存在互联网上的某个位置，可以通过任何一个与互联网相连的终端十分便利地访问和共享这些数据。目前，Google 已经允许第三方在 Google 的云计算中通过 Google App Engine 运行大型并行应用程序。Google 值得称颂的是它不保守，它早已以发表学术论文的形式公开其云计算三大法宝：GFS、MapReduce 和 Bigtable，并在美国、中国等高校开设如何进行云计算编程的课程，2010 年 4 月，Google 公开了其云计算平台监控系统 Dapper 的实现技术，紧接着在 2011 年 1 月，Google 又公开了 Megastore 分布式存储技术。相应的，模仿者应运而生，Hadoop 是其中最受关注的开源项目。

IBM 在 2007 年 11 月推出了“改变游戏规则”的“蓝云”计算平台，为客户带来即买即用的云计算平台。它包括一系列自我管理和自我修复的虚拟化云计算软件，使来自全球的应用可以访问分布式的大型服务器池，使得数据中心在类似于互联网的环境下运行计算。IBM 正在与 17 个欧洲组织合作开展名为 RESERVOIR 的云计算项目，以“无障碍的资源和服务虚拟化”为口号，欧盟提供了 1.7 亿欧元作为部分资金。IBM 已在全球范围内建立了 13 个云计算中心，并且已帮助数个客户成功部署了云计算中心。

微软紧跟云计算步伐，于 2008 年 10 月推出了 Windows Azure 操作系统。Azure（译为“蓝天”）是继 Windows 取代 DOS 之后，微软的又一次颠覆性转型——通过在互联网架构上打造新云计算平台，让 Windows 真正由 PC 延伸到“蓝天”上。Azure 的底层是微软全球基础服务系统，由遍布全球的第四代数据中心构成。目前，微软已经配置了 220 个集装箱式数据中心，包括 44 万台服务器。微软在 2010 年 10 月的 PDC 大会上，公布了 Windows Azure 云计算平台的未来蓝图，跳出单纯的基础架构作服务的框架，将 Windows Azure 定位为平台作服务：一套全面的开发工具、服务和管理系统。它可以让开发者们致力于开发可用和可扩展的应用程序。微软将为 Windows Azure 用户推出许多新的功能，不但能更简单地将现有的应用程序转移到云中，而且可以加强云托管应用程序的可用服务，充分体现微软的“云”+“端”战略。

在我国，云计算发展也非常迅猛。2008 年，IBM 先后在无锡和北京建立了两个云计算中心；世纪互联推出了 CloudEx 产品线，提供互联网主机服务、在线存储虚拟化服务等；中国移动研究院已经建立起 1024 个 CPU 的云计算试验中心，并于 2010 年 5 月发布了“Big Cloud”1.0；解放军理工大学研制了云存储系统 MassCloud，并以它支撑基于 3G 的大规模视频监控应用和数字地球系统；Alibaba 集团也成立了专注于云计算领域研究和

研发的阿里云公司，启动大淘宝战略，研制了淘宝的分布式文件系统（TFS）；中国电信与 EMC 公司合作推出面向家庭和个人用户的运营商级的云信息服务——“E 云”，并在第二届中国云计算大会的展台上展示了其云终端产品，将其命名为“E 云手机”。

作为云计算技术的一个分支，云安全技术通过大量客户端的参与和大量服务器端的统计分析来识别病毒和木马，取得了巨大成功。瑞星、趋势、卡巴斯基、McAfee、Symantec、江民、Panda、金山、360 安全卫士等均推出了云安全解决方案。值得一提的是，云安全的核心思想，与早在 2003 年就提出的反垃圾邮件网格非常接近^[5]。2008 年 11 月 25 日，中国电子学会专门成立了云计算专家委员会。2009 年 5 月 22 日，中国电子学会隆重举办首届中国云计算大会，1200 多人与会，盛况空前。2009 年 11 月 2 日，中国互联网大会专门召开了“2009 云计算产业峰会”。2009 年 12 月，中国电子学会举办了中国首届云计算学术会议。2010 年 5 月，中国电子学会举办了第二届中国云计算大会。2011 年 5 月，中国电子学会将举办第三届中国云计算大会。

1.3 云计算实现机制

由于云计算分为 IaaS、PaaS 和 SaaS 三种类型，不同的厂家又提供了不同的解决方案，目前还没有一个统一的技术体系结构，对读者了解云计算的原理构成了障碍。为此，本书综合不同厂家的方案，构造了一个供参考的云计算体系结构。这个体系结构如图 1-3 所示，它概括了不同解决方案的主要特征，每一种方案或许只实现了其中部分功能，或许也还有部分相对次要功能尚未概括进来。

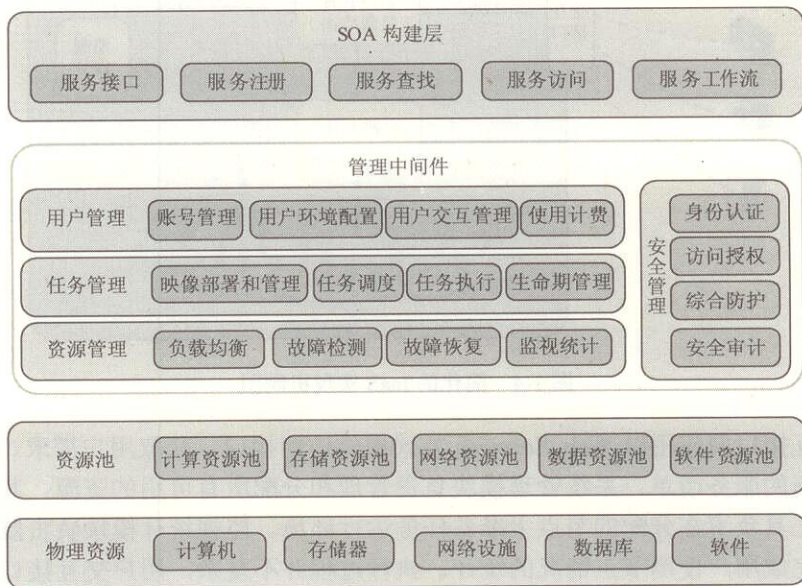


图 1-3 云计算技术体系结构

云计算技术体系结构分为四层：物理资源层、资源池层、管理中间件层和 SOA（Service-Oriented Architecture，面向服务的体系结构）构建层。物理资源层包括计算机、存储器、网络设施、数据库和软件等。资源池层是将大量相同类型的资源构成同构或接近同构的资源池，如计算资源池、数据资源池等。构建资源池更多的是物理资源的集成和管

理工作，例如研究在一个标准集装箱的空间如何装下 2000 个服务器、解决散热和故障节点替换的问题并降低能耗。管理中间件层负责对云计算的资源进行管理，并对众多应用任务进行调度，使资源能够高效、安全地为应用提供服务。SOA 构建层将云计算能力封装成标准的 Web Services 服务，并纳入到 SOA 体系进行管理和使用，包括服务接口、服务注册、服务查找、服务访问和服务工作流等。管理中间件层和资源池层是云计算技术的最关键部分，SOA 构建层的功能更多依靠外部设施提供。

云计算的管理中间件层负责资源管理、任务管理、用户管理和安全管理等工作。资源管理负责均衡地使用云资源节点，检测节点的故障并试图恢复或屏蔽之，并对资源的使用情况进行监视统计；任务管理负责执行用户或应用提交的任务，包括完成用户任务映象（Image）的部署和管理、任务调度、任务执行、任务生命期管理等；用户管理是实现云计算商业模式的一个必不可少的环节，包括提供用户交互接口、管理和识别用户身份、创建用户程序的执行环境、对用户的使用进行计费等等；安全管理保障云计算设施的整体安全，包括身份认证、访问授权、综合防护和安全审计等。

基于上述体系结构，本书以 IaaS 云计算为例，简述云计算的实现机制，如图 1-4 所示。

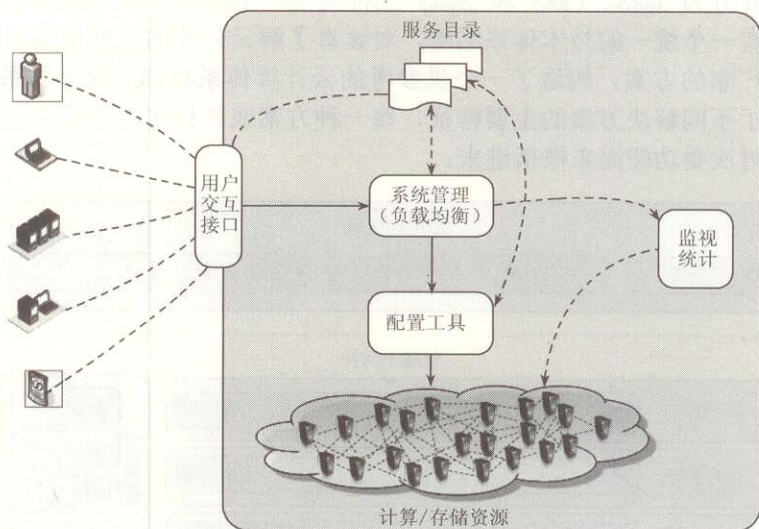


图 1-4 简化的 IaaS 实现机制图

用户交互接口向应用以 Web Services 方式提供访问接口，获取用户需求。服务目录是用户可以访问的服务清单。系统管理模块负责管理和分配所有可用的资源，其核心是负载均衡。配置工具负责在分配的节点上准备任务运行环境。监视统计模块负责监视节点的运行状态，并完成用户使用节点情况的统计。执行过程并不复杂，用户交互接口允许用户从目录中选取并调用一个服务，该请求传递给系统管理模块后，它将为用户分配恰当的资源，然后调用配置工具为用户准备运行环境。

1.4 网络计算与云计算

网络（Grid）是 20 世纪 90 年代中期发展起来的下一代互联网核心技术。网络技术的

开创者 Ian Foster 将之定义为“在动态、多机构参与的虚拟组织中协同共享资源和求解问题”^[6]。网格是在网络基础之上,基于 SOA,使用互操作、按需集成等技术手段,将分散在不同地理位置的资源虚拟成为一个有机整体,实现计算、存储、数据、软件和设备等资源的共享,从而大幅提高资源的利用率,使用户获得前所未有的计算和信息能力。

国际网格界致力于网格中间件、网格平台和网格应用建设。就网格中间件而言,国外著名的网格中间件有 Globus Toolkit、UNICORE、Condor、gLite 等,其中 Globus Toolkit 得到了广泛采纳。就网格平台而言,国际知名的网格平台有 TeraGrid、EGEE、CoreGRID、D-Grid、ApGrid、Grid3、GIG 等。美国 TeraGrid 是由美国国家科学基金会计划资助构建的超大规模开放的科学研究环境。TeraGrid 集成了高性能计算机、数据资源、工具和高端实验设施。目前 TeraGrid 已经集成了超过每秒 750 万亿次计算能力、30PB 数据,拥有超过 100 个面向多种领域的网格应用环境。欧盟 e-Science 促成网格 EGEE (Enabling Grids for E-sciencE),是另一个超大型、面向多种领域的网格计算基础设施。目前已有 120 多个机构参与,包括分布在 48 个国家的 250 个网格站点、68000 个 CPU、20PB 数据资源,拥有 8000 个用户,每天平均处理 30000 个作业,峰值超过 150000 个作业。就网格应用而言,知名的网格应用系统数以百计,应用领域包括大气科学、林学、海洋科学、环境科学、生物信息学、医学、物理学、天体物理、地球科学、天文学、工程学、社会行为学等。

我国在十五期间有 863 支持的中国国家网格 (CNGrid, 863-10 主题) 和中国空间信息网格 (SIG, 863-13 主题)、教育部支持的中国教育科研网格 (ChinaGrid)、上海市支持的上海网格 (ShanghaiGrid) 等。中国国家网格拥有包括香港地区在内的 10 个节点,聚合计算能力为每秒 18 万亿次,目前拥有 408 个用户和 360 个应用。中国教育科研网格 ChinaGrid 连接了 20 所高校的计算设施,运算能力达每秒 3 万亿次以上,开发并实现了生物信息、流体力学等五个科学研究领域的网格典型应用。十一五期间,国家对网格支持的力度更大,通过 973 和 863、自然科学基金等途径对网格技术进行了大力支持。973 计划有“语义网格的基础理论、模型与方法研究”等,863 计划有“高效能计算机及网格服务环境”、“网格地理信息系统软件及其重大应用”等,国家自然科学基金重大研究计划有“网络计算应用支撑中间件”等项目。

就像云计算可以分为 IaaS、PaaS 和 SaaS 三种类型一样,网格计算也可以分为三种类型:计算网格、信息网格和知识网格^[6]。计算网格的目标是提供集成各种计算资源的、虚拟化的计算基础设施。信息网格的目标是提供一体化的智能信息处理平台,集成各种信息系统和信息资源,消除信息孤岛,使得用户能按需获取集成后的精确信息,即服务点播 (Service on Demand) 和一步到位的服务 (One Click is Enough)。知识网格^[8]研究一体化的智能知识处理和理解平台,使得用户能方便地发布、处理和获取知识。

需要说明的是,目前大家对网格的认识存在一种误解,认为只有使用 Globus Toolkit 等知名网格中间件的应用才是网格。我们认为,只要是遵照网格理念,将一定范围内分布的异构资源集成为有机整体,提供资源共享和协同工作服务的平台,均可以认为是网格。这是因为,由于网格技术非常复杂,必然有一个从不规范到规范化的过程,应该承认差异存在的客观性。虽然网格界从一开始就致力于构造能够实现全面互操作的环境,但由于网格处于信息技术前沿、许多领域尚未定型、已发布的个别规范过于复杂造成易用性差等原因,现有网格系统多针对具体应用采用适用的、个性化的框架设计和实现技术等,造成网格系统之间互操作困难,这也是开放网格论坛 OGF (Open Grid Forum) 提出建立不同网

格系统互通机制计划 GIN（Grid Interoperation Now）的原因。从另一个角度看，虽然建立全球统一的网格平台还有很长的路要走，但并不妨碍网格技术在各种具体的应用系统中发挥重要的作用。

网格计算与云计算的比较如表 1-1 所示。

表 1-1 网格计算与云计算的比较

	网 格 计 算	云 计 算
目标	共享高性能计算力和数据资源，实现资源共享和协同工作	提供通用的计算平台和存储空间，提供各种软件服务
资源来源	不同机构	同一机构
资源类型	异构资源	同构资源
资源节点	高性能计算机	服务器/PC
虚拟化视图	虚拟组织	虚拟机
计算类型	紧耦合问题为主	松耦合问题
应用类型	科学计算为主	数据处理为主
用户类型	科学界	商业社会
付费方式	免费（政府出资）	按量计费
标准化	有统一国际标准 OGSA/WSRF	尚无标准，但已经有了开放云计算联盟 OCC

网格计算在概念上争论多年，在体系结构上有三次大的改变，在标准规范上花费了大量的人力，所设定的目标又非常远大——要在跨平台、跨组织、跨信任域的极其复杂的异构环境中共享资源和协同解决问题，所要共享的资源也是五花八门——从高性能计算机、数据库、设备到软件，甚至知识。云计算暂时不管概念、不管标准，Google 云计算与 Amazon 云计算的差别非常大，云计算只是对它们以前所做事情新的共同的时髦叫法，所共享的存储和计算资源暂时仅限于某个企业内部，省去了许多跨组织协调的问题。以 Google 为代表的云计算在内部管理运作方式上的简洁一如其界面，能省的功能都省略，Google 文件系统（GFS）甚至不允许修改已经存在的文件，只允许在文件后追加数据，大大降低了实现难度，而且借助其无与伦比的规模效应释放了前所未有的能量。

网格计算与云计算的关系，就像是 OSI 与 TCP/IP 之间的关系：国际标准化组织（ISO）制定的 OSI（开放系统互联）网络标准，考虑得非常周到，也异常复杂，在多年之前就考虑到了会话层和表示层的问题。虽然很有远见，但过于理想，实现的难度和代价非常大。当 OSI 的一个简化版——TCP/IP 诞生之后，将七层协议简化为四层，内容也大大精简，因而迅速取得了成功。在 TCP/IP 一统天下之后多年，语义网等问题才被提上议事日程，开始为 TCP/IP 补课，增加其会话和表示的能力。因此，可以说 OSI 是学院派，TCP/IP 是现实派；OSI 是 TCP/IP 的基础，TCP/IP 又推动了 OSI 的发展。两者不是“成者为王、败者为寇”，而是滚动发展。

没有网格计算打下的基础，云计算也不会这么快到来。云计算是网格计算的一种简化实用版，通常意义的网格是指以前实现的以科学研究为主的网格，非常重视标准规范，也非常复杂，但缺乏成功的商业模式。云计算是网格计算的一种简化形态，云计算的成功也是网格的成功。网格不仅要集成异构资源，还要解决许多非技术的协调问题，也不像云计算有成功的商业模式推动，所以实现起来要比云计算难度大很多。但对于许多高端科学或军事应用而言，云计算是无法满足需求的，必须依靠网格来解决。

目前，许多人声称网格计算失败了，云计算取而代之了，这其实是一种错觉。网格计

算已经有十多年历史,不如刚兴起时那样引人注目是正常的。事实上,有些政府主导、范围较窄、用途特定的网格,已经取得了决定性的胜利。代表性的有美国的 TeraGrid 和欧洲的 EGEE 等,这些网格每天都有几十万个作业在上面执行。未来的科学研究主战场,将建立在网格计算之上。在军事领域,美军的全球信息网格 GIG 已经囊括超过 700 万台计算机,规模超过现有的所有云计算数据中心计算机总和。

相信不久的将来,建立在云计算之上的“商业 2.0”与建立在网格计算之上的“科学 2.0”都将取得成功。

1.5 云计算的发展环境

1.5.1 云计算与 3G

3G (3rd-Generation) 是第三代移动通信技术的简称。3G 是指支持高速数据传输的蜂窝移动通信技术,是将无线通信与互联网相结合的新一代通信技术。目前国际电信联盟确定了三个 3G 标准制式: CDMA2000、WCDMA 和 TD-SCDMA。在我国,中国电信、中国联通和中国移动分别运营这三种不同制式的 3G 网络。3G 的代表性特征是具有高速数据传输能力,能够提供 2Mbps 以上的带宽。因此,3G 可以支持语音、图像、音乐、视频、网页、电话会议等多种移动多媒体业务。

3G 与云计算是互相依存、互相促进的关系。一方面,3G 将为云计算带来数以亿计的宽带移动用户。到 2009 年 7 月,全球移动用户已达 44 亿,普及率达 65%。3G 用户已超过 5 亿,并以惊人的速度增长。2009 年是中国 3G 元年,当年用户数就超过了 1 千万。这些用户的终端是手机、PDA、笔记本、上网本等,计算能力和存储空间有限,却有很强的联网能力,对云计算有着天然的需求,将实实在在地支持云计算取得商业成功;另一方面,云计算能够给 3G 用户提供更好的用户体验。云计算有强大的计算能力、接近无限的存储空间,并支撑各种各样的软件和信息服务,能够为 3G 用户提供前所未有的服务体验。

1.5.2 云计算与物联网

物联网(The Internet of Things)即“物物相连的互联网”。物联网通过大量分散的射频识别(RFID)、传感器、GPS、激光扫描器等小型设备,将感知的信息,通过互联网传输到指定的处理设施上进行智能化处理,完成识别、定位、跟踪、监控和管理等工作。笼统地看,物联网属于传感网的范畴。其实,传感器的应用历史悠久而且相当普及。那为什么还提物联网的概念呢?物联网是传感网的一个高级阶段,它通过大量信息感知节点采集信息,通过互联网传输和交换信息,通过强大的计算设施处理信息,然后再对实体世界发出反馈或控制信息。

物联网根据其实际用途可以归结为三种基本应用模式:对象的智能标签、环境监控和对象跟踪、对象的智能控制。物联网基于云计算平台和智能网络,可以依据传感器网络所获取的数据进行决策,改变对象的行为进行控制和反馈。

云计算服务物联网的驱动力有以下几个方面。

(1) 需求驱动:海量信息的处理,在目前技术下的高成本压力;云计算充分利用并合理使用资源,降低运营成本。

(2) 技术驱动: IT 与 CT (Computed Tomography, 电子计算机 X 射线断层扫描技术) 技术融合, 推动 IT 架构的升级; 云计算的标准逐渐快速发展。

(3) 政策驱动: 政府的低碳经济与节能减排的政策要求; 政府高度关注物联网、云计算等基础设施自助发展战略。

物联网具有全面感知、可靠传递和智能处理三个特征, 其中智能处理需要对海量的信息进行分析 and 处理, 对物体实施智能化的控制, 这就需要信息技术的支持。云计算的超大规模、虚拟化、多用户、高可靠性、高扩展性等特点正是物联网规模化、智能化发展所需的技术。

云计算架构在互联网之上, 而物联网将主要依赖互联网来实现有效延伸, 云计算模式可以支撑具有业务一致性的物联网集约运营。因此, 很多研究提出了构建基于云计算的物联网运营平台, 该平台主要包括云基础设施、云平台、云应用和云管理。依托公众通信网络, 以数据中心为核心, 通过多接入终端实现泛在接入, 面向服务的端到端体系架构。基于云计算模式, 实现资源共享与产业协作, 提高效率, 降低成本, 提升服务。

有观点认为云计算是物联网“后端”支撑关键。所谓物联网的“后端”, 是指基于互联网计算的涌现智能及对物理世界的反馈和控制。“后端”是实现物联网智能化管理目标 and 价值追求的关键所在。云计算协同信息处理与计算平台对信息处理与决策。实时感应、高度并发、自主协同和涌现效应等特征对物联网“后端”提出了新的挑战, 需要有针对性地研究物联网特定的应用集成问题、体系结构以及标准规范, 特别是大量高并发时间驱动的应用自动关联和智能协作问题。在互联网计算领域, 将软件的实现与运维和用法相关部分 (服务) 相剥离, 并纳入到互联网级基设中, 这是大势所趋^[9]。而互联网级基设也是云计算、网格计算的本质所在。

物联网与云计算是交互辉映的关系。一方面, 物联网的发展也离不开云计算的支撑。从量上看, 物联网将使用数量惊人的传感器 (如数以亿万计的 RFID、智能尘埃和视频监控等), 采集到的数据量惊人。这些数据需要通过无线传感网、宽带互联网向某些存储和处理设施汇聚, 而使用云计算来承载这些任务具有非常显著的性价比优势; 从质上看, 使用云计算设施对这些数据进行处理、分析、挖掘, 可以更加迅速、准确、智能地对物理世界进行管理和控制, 使人类可以更加及时、精细地管理物质世界, 从而达到“智慧”的状态, 大幅提高资源利用率和社会生产力水平。可以看出, 云计算凭借其强大的处理能力、存储能力和极高的性能价格比, 很自然就会成为物联网的后台支撑平台; 另一方面, 物联网将成为云计算最大的用户, 将为云计算取得更大商业成功奠定基石。

1.5.3 云计算与移动互联网

互联网和移动通信网是当今最具影响力的两个全球性网络, 移动互联网恰恰融合了两者的发展优势。被称作破坏性创新的云计算, 在宽带移动互联网上将成为一种绕不开的趨勢。市场调研公司认为, 云计算将成为移动世界中一股爆破理论, 最终会成为移动应用的主导运行方式^[10]。掌握了云计算核心技术的企业无疑在移动互联网时代可以获得更强的主动性。

移动互联网和云计算是相辅相成的。通过云计算技术, 软硬件获得空前的集约化应用, 人们完全可以通过手持一个终端就能实现传统 PC 能达到的功能。二者在软硬件设施成本上的极大节约为中小企业带来了福音, 为人们带来舒适和便捷。

云计算和移动互联网似乎天生就是绝配。手机拥有便携性和通信能力等众多天然优势,而计算能力、存储能力弱,虽然各厂商推出的手机正逐渐向智能化演进,但受限于体积和便携性的要求,短时间内手机的处理能力难以和计算机相比。

从这点出发,云计算的特点更能在移动互联网上充分体现,将应用的“计算”与存储从终端转移到服务器的云端,从而弱化了对移动终端设备的处理需求。例如,在各种数据业务快速推陈出新中,手机很难及时支持这些新业务的要求,成为新业务的发展瓶颈。在云计算下,只要配备功能强大的浏览器,就能应用各种新业务。在后台,云计算的存储量和计算能力也解决了手机存储量有限和丢失信息的问题。同时,实现了手机移动与固定计算、笔记本电脑计算的协同。对于追求个性化的移动互联网市场来说,云计算的力量十分关键。

移动互联网时代来临,对用户来讲,最好的体验是淡化有线和无线的概念。在这样的理念下,云计算有望突破各种终端,包括手机、计算机、电视和视听设备等在存储及运算能力上的限制,显示的内容、应用都能保持一致性和同步性。各大IT厂商都在利用云计算制定如IaaS、PaaS和SaaS策略,希望通过利用互联网的力量,以软件为基准,将无缝的服务提供给移动终端用户。

云计算正从互联网逐渐过渡到移动互联网。目前社交网站越来越火爆,国外的Facebook及国内的人人网、开心网等都是其典型的代表。社交网站运用云计算思维,实现了网站上各种信息的同步更新。沿着这个思路的移动云计算已经出现,如摩托罗拉推出的手机解决方案。

如今,随着一些典型的互联网云计算应用,互联网的“云”与“端”之间已经形成了平滑对接,而在移动互联网上,“云”与“端”之间还需要“管”来沟通它们之间的鸿沟。浏览器或许将成为重要的“管”道角色。

云计算对于云和端两侧都具有传统模式不可比拟的优势。在云的一侧,为内部开发者和业务使用者提供更多的服务,提升基础设施的使用效率和资源部署的灵活性;在端的一侧,能够迅速部署应用和服务,按需调整业务使用量。从目前云计算的成功案例中可以看出云计算极大地提高了互联网信息技术的性能,具有巨大的计算和成本优势。

1.5.4 云计算与三网融合

所谓的三网融合,是指广播电视网、电信网与互联网的融合,其中互联网是核心。据国务院三网融合领导小组专家组组长、中国工程院副院长邬贺铨估算,三网融合启动的相关产业市场规模达6880亿元人民币。其中电信宽带升级、广电双向网络改造、机顶盒产业发展以及基于音频视频内容的信息服务系统建设的有效投资额为2490亿元,可激发和释放社会的信息服务与终端消费额近4390亿元。

三网融合被纳入“十二五”计划,并明确写入《国务院关于加快培育和发展战略性新兴产业的决定》。业内权威专家认为,三网融合的政策持续加码,将推动电信与广电业务相互进入、广电网络整合、网络运营商角色再定位等一系列革命性变化同步加速。仅中国电信一家运营商,其两年内用于宽带升级的投资将达到近300亿元。

云计算使计算能力从分散终端向网络综合服务转变,使商业模式从网络设备基础设施向服务转变,从连接计算机资源向连接个人和设备转变^[1]。云计算的基础仍然是宽带,其服务手段和服务对象都需要宽带。社会的各种生活、娱乐和就业都对宽带发展提出了高要

求，各国也加大对宽带建设的投入，各厂商也都在加强对宽带技术的研发。

业内专家认为，随着三网融合政策的出台以及下一代广电网络的上马，云计算不但会为现有广电和电信产业带来新商机，还会大大拓展相关产业链，使更多企业受益，为云计算提供切实的应用机会。三网融合和下一代广电网项目是要为用户提供多样、便捷的服务。由于云计算可以大大降低数据储存、计算和分发成本，一些以前无法实现的应用，现在都有可能变为现实。云计算完成计算任务，加上物联网等终端应用和 3G 的数据信息传输，将三网整合形成一个系统的信息采样、接受和处理的整体。

三网融合和下一代广电网络的最终目标是构建全数据、全融合的国家骨干网络，借助云计算技术，下一代广电网络还会和传统行业相融合，实现诸如远程教育、网络医疗会诊、股票信息、交通查询、精确广告投放等更多应用。有了云计算技术，一些从事传统行业的企业也能搭上三网融合和下一代广电网的快车。例如，传统的 GPS 厂商只是生产商，而借助云计算技术，他们可以成为服务性企业，通过增值业务获取更多收入。中国电子学会计算机委员会专家刘鹏教授认为，云计算在三网融合及下一代广电网中的应用，涉及数据存储、数据计算、数据再处理、软件开发、数据传输、网络协同等多个方面，因此需要大量不同类型的企业参与其中。

1.6 云计算压倒性的成本优势

为什么云计算拥有划时代的优势？主要原因在于它的技术特征和规模效应所带来的压倒性的性能价格比优势。

全球企业的 IT 开销，分为三部分：硬件开销、能耗和管理成本。根据 IDC 在 2007 年做过的一个调查和预测（如图 1-5 所示），从 1996 年到 2010 年，全球企业 IT 开销中的硬件开销是基本持平的。但能耗和管理的成本上升非常迅速，以至于到 2010 年管理成本占了 IT 开销的大部分，而能耗开销越来越接近硬件开销了。

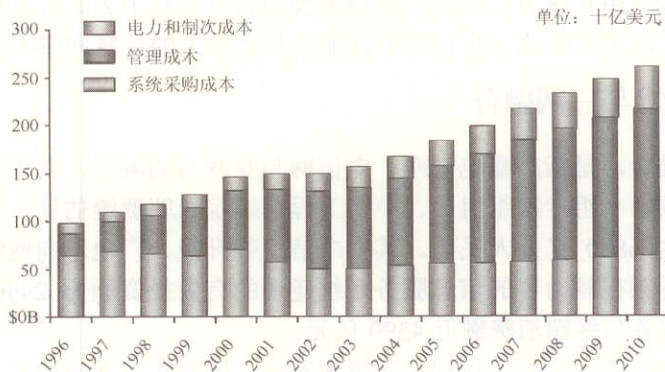


图 1-5 全球企业 IT 开销发展趋势

如果使用云计算的话，系统建设和管理成本有很大的区别，如表 1-2 所示。根据 James Hamilton 的数据^{[1][2]}，一个拥有 5 万个服务器的特大型数据中心与拥有 1000 个服务器中型数据中心相比，特大型数据中心的网络和存储成本只相当于中型数据中心的 1/5 到 1/7，而每个管理员能够管理的服务器数量则扩大到 7 倍之多。因而，对于规模通常达到

几十万乃至上百万台计算机的 Amazon 和 Google 云计算而言, 其网络、存储和管理成本较之中型数据中心至少可以降低 5~7 倍。

表 1-2 中型数据中心和特大型数据中心的成本比较

技 术	中型数据中心成本	特大型数据中心成本	比率
网络	\$95 每 Mb/秒/月	\$13 每 Mb/秒/月	7.1
存储	\$2.20 每 GB/月	\$0.40 每 GB/月	5.7
管理	每个管理员约管理 140 个服务器	每个管理员管理 1000 个服务器以上	7.1

电力和制冷成本也会有明显的差别。虽然我国的电价是全国统一的, 但实际上不同地区的电力成本是不一样的。例如^[2], 美国爱达荷州的水电资源丰富, 电价很便宜。而夏威夷州是岛屿, 本地没电力资源, 电力价格就比较贵。二者最多相差 7 倍, 如表 1-3 所示。

表 1-3 美国不同地区电力价格的差异

每千瓦时的价格	地 点	可能的定价原因
3.6 美分	爱达荷州	水力发电, 没有长途输送
10.0 美分	加州	加州不允许煤电, 电力需在电网上长途输送
18.0 美分	夏威夷州	发电的能源需要海运到岛上

主要由于电价有如此显著的差异, Google 的数据中心一般选择在人烟稀少、气候寒冷、水电资源丰富的地区, 这些地点的电价、散热成本、场地成本、人力成本等都远远低于人烟稠密的大都市。剩下的挑战是要专门铺设光纤到这些数据中心。不过, 由于光纤密集波分复用技术 (DWDM) 的应用, 单根光纤的传输容量已超过 10Tbit/s, 在地上开挖一条小沟埋设的光纤所能传输的信息容量几乎是无限的, 远比将电力用高压输电线路引入城市要容易得多, 而且没有衰减。拿 Google 的话来说, “传输光子比传输电子要容易得多”。这些数据中心采用了高度自动化的云计算软件来管理, 需要的人员很少, 而为了技术保密而拒绝外人进入参观, 让人有一种神秘的感觉, 故被人戏称为“信息时代的核电站”, 如图 1-6 所示。



图 1-6 被称为“信息时代的核电站”的 Google 数据中心

再者，云计算与传统互联网数据中心（IDC）相比，资源的利用率也有很大不同。IDC 一般采用服务器托管和虚拟主机等方式对网站提供服务。每个租用 IDC 的网站所获得的网络带宽、处理能力和存储空间都是固定的。然而，绝大多数网站的访问流量都不是均衡的。例如，有的时间性很强，白天访问的人少，到了晚上 7、8 点钟就会流量暴涨；有的季节性很强，平时访问人不多，但是到圣诞节前访问量就很大；有的一直默默无闻，但是由于某些突发事件（如迈克尔·杰克逊突然去世），使得访问量暴增而陷入瘫痪。网站拥有者为了应对这些突发流量，会按照峰值要求来配置服务器和网络资源，造成资源的平均利用率只有 10%~15%，如图 1-7 所示。而云计算平台提供的是有弹性的服务，它根据每个租用者的需要在一个超大的资源池中动态分配和释放资源，而不需要为每个租用者预留峰值资源。而且云计算平台的规模极大，其租用者数量非常多，支撑的应用种类也是五花八门，比较容易平稳整体负载，因而云计算资源利用率可以达到 80%左右，这又是传统模式的 5~7 倍。

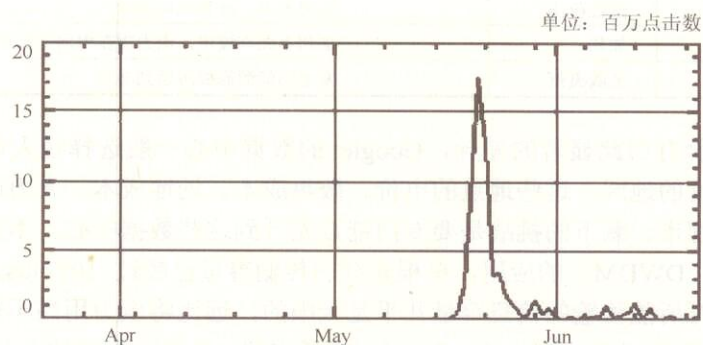


图 1-7 某典型网站的流量数据

综上所述，由于云计算有更低的硬件和网络成本，更低管理成本和电力成本，以及更高的资源利用率，两个乘起来就能够将成本节省 30 倍以上，如图 1-8 所示。这是个惊人的数字！这是云计算成为划时代技术的根本原因。

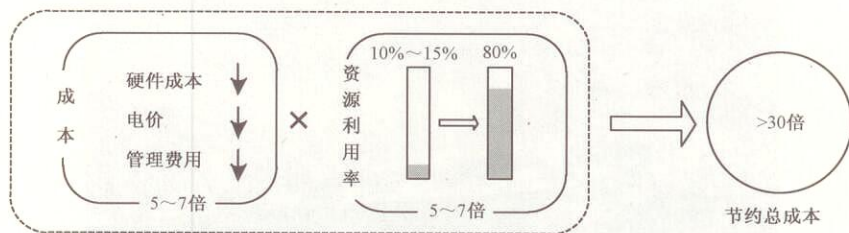


图 1-8 云计算较之传统方式的性价比优势

从前面可以知道，云计算能够大幅节省成本，规模是极其重要的因素。那么，如果企业要建设自己的私有云，规模不大，也无法享受到电价优惠，是否就没有成本优势了呢？仍然会有数倍的优势。一方面硬件采购成本还是会节省好几倍，这是因为云计算技术的容错能力很强，使得我们可以使用低端硬件代替高端硬件。另一方面，对云计算设施的管理是高度自动化的，极少需要人工干预，可以大大减少管理人员的数量。中国移

动研究院建立了 1024 个节点的 Big Cloud 云计算设施,并用它进行海量数据挖掘,大大节省了成本。

对云计算用户而言,云计算的优势也是无与伦比的。他们不用开发软件,不用安装硬件,用低得多的使用成本,就可以快速部署应用系统,而且可以动态伸缩系统的规模,可以更容易地共享数据。租用公共云的企业不再需要自建数据中心,只需申请账号并按量付费,这一点对于中小企业和刚起步的创业公司尤为重要。目前,云计算的应用领域涵盖应用托管、存储备份、内容推送、电子商务、高性能计算、媒体服务、搜索引擎、Web 托管等多个领域,代表性的云计算应用企业包括 Abaca、BeInSync、AF83、Giveness、纽约时报、华盛顿邮报、GigaVox、SmugMug、Alexa、Digitaria 等。纽约时报使用 Amazon 云计算服务在不到 24 个小时的时间里处理了 1100 万篇文章,累计花费仅 240 美元。如果用自己的服务器,需要数月时间和多得多的费用。

习题

1. 云计算有哪些特点?
2. 云计算按照服务类型可以分为哪几类?
3. 云计算技术体系结构可以分为哪几层?
4. 云计算与网格计算的异同?
5. 云计算与物联网、三网融合的发展关系?

参考文献

- [1] Michael Armbrust, Armando Fox, and Rean Griffith, et al. Above the Clouds: A Berkeley View of Cloud Computing, mimeo, UC Berkeley, RAD Laboratory, 2009
- [2] Ian Foster, Carl Kesselman, and Steve Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. International Journal of High Performance Computing Applications, 15(3), 2001
- [3] 刘鹏. 提出一种实用的网格实现方式——网格计算池模型, 2002
<http://www.chinagrid.net/show.aspx?id=1672&cid=57>
- [4] Peng Liu, Yao Shi, San-li Li, Computing Pool—a Simplified and Practical Computational Grid Model, the Second International Workshop on Grid and Cooperative Computing (GCC 2003), Shanghai, Dec 7-10, 2003, published in Lecture Notes in Computer Science (LNCS), Vol. 3032, Heidelberg: Springer-Verlag, 2004
- [5] Peng Liu, Yao Shi, Francis C. M. Lau, Cho-Li Wang, San-Li Li, Grid Demo Proposal: AntiSpamGrid, IEEE International Conference on Cluster Computing, Hong Kong, Dec 1-4, 2003, selected as one of the excellent Grid research projects for the GridDemo session
- [6] 李国杰. 信息服务网格——第三代 Internet. 计算机世界, 2001 年第 40 期

- [7] Foster, I., C. Kesselman, and S. Tuecke, The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 2001. 15(3): p. 200-222
- [8] H. Zhuge, The Knowledge Grid, World Scientific Publishing Co., Singapore, 2004
- [9] 韩燕波, 赵卓峰等. 物联网与云计算. 中国计算机学会通讯, 2010 年 2 月
- [10] 杨扬. 移动互联网混搭云计算. 北京: 人民邮电报, 2010 年 4 月 7 日
- [11] 杜百川. 下一代网络服务核心—云计算, IBTC2010 论坛

第2章 Google 云计算原理与应用

Google 拥有全球最强大的搜索引擎。除了搜索业务，Google 还有 Google Maps、Google Earth、Gmail、YouTube 等其他业务。这些应用的共性在于数据量巨大，且要面向全球用户提供实时服务，因此 Google 必须解决海量数据存储和快速处理问题。Google 研发出了简单而又高效的技术，让多达百万台的廉价计算机协同工作，共同完成这些任务，这些技术在诞生几年后才被命名为 Google 云计算技术。Google 云计算技术包括：Google 文件系统 GFS、分布式计算编程模型 MapReduce、分布式锁服务 Chubby、分布式结构化数据表 Bigtable、分布式存储系统 Megastore 以及分布式监控系统 Dapper 等。其中，GFS 提供了海量数据的存储和访问的能力，MapReduce 使得海量信息的并行处理变得简单易行，Chubby 保证了分布式环境下并发操作的同步问题，Bigtable 使得海量数据的管理和组织十分方便，构建在 Bigtable 之上的 Megastore 则实现了关系型数据库和 NoSQL 之间的巧妙融合，Dapper 能够全方位的监控整个 Google 云计算平台的运行状况。本章详细介绍这六种核心技术和 Google 应用程序引擎。

2.1 Google 文件系统 GFS

Google 文件系统（Google File System，GFS）是一个大型的分布式文件系统。它为 Google 云计算提供海量存储，并且与 Chubby、MapReduce 及 Bigtable 等技术结合十分紧密，处于所有核心技术的底层。GFS 不是一个开源的系统，我们仅能从 Google 公布的技术文档来获得相关知识。文献[1]是 Google 公布的关于 GFS 的最为详尽的技术文档，它从 GFS 产生的背景、特点、系统框架、性能测试等方面进行了详细的阐述。

当前主流分布式文件系统有 RedHat 的 GFS^[3]（Global File System）、IBM 的 GPFS^[4]、Sun 的 Lustre^[5]等。这些系统通常用于高性能计算或大型数据中心，对硬件设施条件要求较高。以 Lustre 文件系统为例，它只对元数据管理器 MDS 提供容错解决方案，而对于具体的数据存储节点 OST 来说，则依赖其自身来解决容错的问题。例如，Lustre 推荐 OST 节点采用 RAID 技术或 SAN 存储区域网来容错，但由于 Lustre 自身不能提供数据存储的容错，一旦 OST 发生故障就无法恢复，因此对 OST 的稳定性就提出了相当高的要求，从而大大增加了存储的成本，而且成本会随着规模的扩大线性增长。

正如李开复所说的那样，创新固然重要，但有用的创新更重要。创新的价值，取决于一项创新在新颖、有用和可行性这三个方面的综合表现。Google GFS 的新颖之处在于它采用廉价的商用机器构建分布式文件系统，同时将 GFS 的设计与 Google 应用的特点紧密结合，简化实现，使之可行，最终达到创意新颖、有用、可行的完美组合。GFS 将容错的任务交给文件系统完成，利用软件的方法解决系统可靠性问题，使存储的成本成倍下降。GFS 将服务器故障视为正常现象，并采用多种方法，从多个角度，使用不同的容错措施，确保数据存储的安全、保证提供不间断的数据存储服务。

2.1.1 系统架构

GFS 的系统架构如图 2-1^[1]所示。GFS 将整个系统的节点分为三类角色：Client（客户端）、Master（主服务器）和 Chunk Server（数据块服务器）。Client 是 GFS 提供给应用程序的访问接口，它是一组专用接口，不遵守 POSIX 规范，以库文件的形式提供。应用程序直接调用这些库函数，并与该库链接在一起。Master 是 GFS 的管理节点，在逻辑上只有一个，它保存系统的元数据，负责整个文件系统的管理，是 GFS 文件系统上的“大脑”。Chunk Server 负责具体的存储工作。数据以文件的形式存储在 Chunk Server 上，Chunk Server 的个数可以有多个，它的数目直接决定了 GFS 的规模。GFS 将文件按照固定大小进行分块，默认是 64MB，每一块称为一个 Chunk（数据块），每个 Chunk 都有一个对应的索引号（Index）。

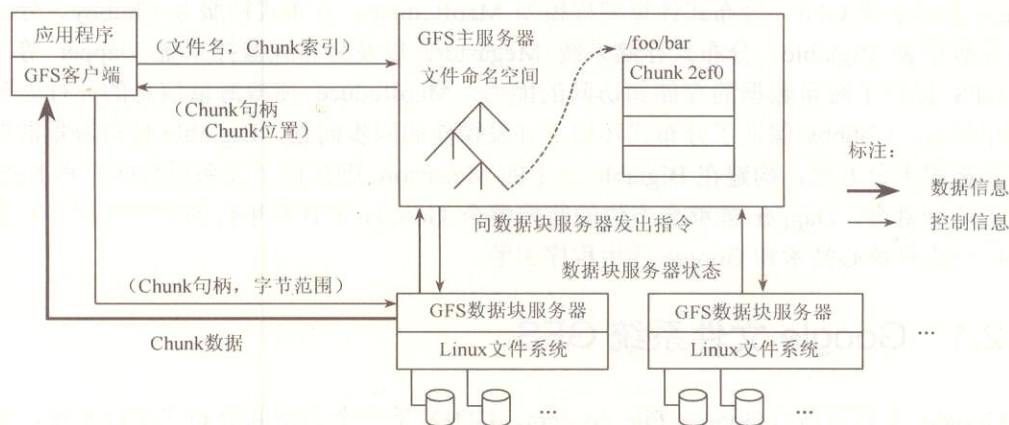


图 2-1 GFS 体系结构

客户端在访问 GFS 时，首先访问 Master 节点，获取与之进行交互的 Chunk Server 信息，然后直接访问这些 Chunk Server，完成数据存取工作。GFS 的这种设计方法实现了控制流和数据流的分离。Client 与 Master 之间只有控制流，而无数据流，极大地降低了 Master 的负载。Client 与 Chunk Server 之间直接传输数据流，同时由于文件被分成多个 Chunk 进行分布式存储，Client 可以同时访问多个 Chunk Server，从而使得整个系统的 I/O 高度并行，系统整体性能得到提高。

针对多种应用的特点，Google 从多个方面简化设计的 GFS，在一定规模下达到了成本、可靠性和性能的最佳平衡。具体来说，它具有以下几个特点。

1. 采用中心服务器模式

GFS 采用中心服务器模式管理整个文件系统，简化了设计，降低了实现难度。Master 管理分布式文件系统的所有元数据。文件被划分为 Chunk 进行存储，对于 Master 来说，每个 Chunk Server 只是一个存储空间。Client 发起的所有操作都需要先通过 Master 才能执行。这样做有许多好处，增加新的 Chunk Server 是一件十分容易的事情，Chunk Server 只需要注册到 Master 上即可，Chunk Server 之间无任何关系。如果采用完全对等的、无中心的模式，那么如何将 Chunk Server 的更新信息通知到每一个 Chunk Server，会

是设计的一个难点，而这也将在一定程度上影响系统的扩展性。Master 维护了一个统一的命名空间，同时掌握整个系统内 Chunk Server 的情况，据此可以实现整个系统范围内数据存储的负载均衡。由于只有一个中心服务器，元数据的一致性自然解决。当然，中心服务器模式也带来一些固有的缺点，比如极易成为整个系统的瓶颈等。GFS 采用多种机制来避免 Master 成为系统性能和可靠性上的瓶颈，如尽量控制元数据的规模、对 Master 进行远程备份、控制信息和数据分流等。

2. 不缓存数据

缓存 (Cache) 机制是提升文件系统性能的一个重要手段，通用文件系统为了提高性能，一般需要实现复杂的缓存机制。GFS 文件系统根据应用的特点，没有实现缓存，这是从必要性和可行性两方面考虑的。从必要性上讲，客户端大部分是流式顺序读写，并不存在大量的重复读写，缓存这部分数据对提高系统整体性能的作用不大；对于 Chunk Server，由于 GFS 的数据在 Chunk Server 上以文件的形式存储，如果对某块数据读取频繁，本地的文件系统自然会将其缓存。从可行性上讲，如何维护缓存与实际数据之间的一致性是一个极其复杂的问题，在 GFS 中各个 Chunk Server 的稳定性都无法确保，加之网络等多种不确定因素，一致性问题尤为复杂。此外由于读取的数据量巨大，以当前的内存容量无法完全缓存。对于存储在 Master 中的元数据，GFS 采取了缓存策略，因为一方面 Master 需要频繁操作元数据，把元数据直接保存在内存中，提高了操作的效率。另一方面，采用相应的压缩机制降低元数据占用空间的大小，提高内存的利用率。

3. 在用户态下实现

文件系统是操作系统的重要组成部分，通常位于操作系统的底层（内核态）。在内核态实现文件系统，可以更好地和操作系统本身结合，向上提供兼容的 POSIX 接口。然而，GFS 却选择在用户态下实现，主要基于以下考虑。

(1) 在用户态下实现，直接利用操作系统提供的 POSIX 编程接口就可以存取数据，无需了解操作系统的内部实现机制和接口，降低了实现的难度，提高了通用性。

(2) POSIX 接口提供的功能更为丰富，在实现过程中可以利用更多的特性，而不像内核编程那样受限。

(3) 用户态下有多种调试工具，而在内核态中调试相对比较困难。

(4) 用户态下，Master 和 Chunk Server 都以进程的方式运行，单个进程不会影响到整个操作系统，从而可以对其进行充分优化。在内核态下，如果不能很好地掌握其特性，效率不但不会高，甚至还会影响到整个系统运行的稳定性。

(5) 用户态下，GFS 和操作系统运行在不同的空间，两者耦合性降低，方便 GFS 自身和内核的单独升级。

4. 只提供专用接口

通常的分布式文件系统一般都会提供一组与 POSIX 规范兼容的接口，使应用程序可以通过操作系统的统一接口透明地访问文件系统，而不需要重新编译程序。GFS 在设计之初，是完全面向 Google 的应用的，采用了专用的文件系统访问接口。接口以库文件的形式提供，应用程序与库文件一起编译，Google 应用程序在代码中通过调用这些库文件的 API，完成对 GFS 文件系统的访问。采用专用接口有以下好处。

（1）降低了实现的难度。通常与 POSIX 兼容的接口需要在操作系统内核一级实现，而 GFS 是在应用层实现的。

（2）采用专用接口可以根据应用的特点对应用提供一些特殊支持，如支持多个文件并发追加的接口等。

（3）专用接口直接和 Client、Master、Chunk Server 交互，减少了操作系统之间上下文的切换，降低了复杂度，提高了效率。

2.1.2 容错机制

1. Master 容错

具体来说，Master 上保存了 GFS 文件系统的三种元数据。

（1）命名空间（Name Space），也就是整个文件系统的目录结构。

（2）Chunk 与文件名的映射表。

（3）Chunk 副本的位置信息，每一个 Chunk 默认有三个副本。

首先就单个 Master 来说，对于前两种元数据，GFS 通过操作日志来提供容错功能。第三种元数据信息则直接保存在各个 Chunk Server 上，当 Master 启动或 Chunk Server 向 Master 注册时自动生成。因此当 Master 发生故障时，在磁盘数据保存完好的情况下，可以迅速恢复以上元数据。为了防止 Master 彻底死机的情况，GFS 还提供了 Master 远程的实时备份，这样在当前的 GFS Master 出现故障无法工作的时候，另外一台 GFS Master 可以迅速接替其工作。

2. Chunk Server 容错

GFS 采用副本的方式实现 Chunk Server 的容错。每一个 Chunk 有多个存储副本（默认为三个），分布存储在不同的 Chunk Server 上。副本的分布策略需要考虑多种因素，如网络的拓扑、机架的分布、磁盘的利用率等。对于每一个 Chunk，必须将所有的副本全部写入成功，才视为成功写入。之后，如果相关的副本出现丢失或不可恢复等情况，Master 自动将该副本复制到其他 Chunk Server，从而确保副本保持一定的个数。尽管一份数据需要存储三份，好像磁盘空间的利用率不高，但综合比较多种因素，加之磁盘的成本不断下降，采用副本无疑是最简单、最可靠、最有效，而且实现的难度也最小的一种方法。

GFS 中的每一个文件被划分成多个 Chunk，Chunk 的默认大小是 64MB，这是因为 Google 应用中处理的文件都比较大，以 64MB 为单位进行划分，是一个较为合理的选择。Chunk Server 存储的是 Chunk 的副本，副本以文件的形式进行存储。每一个 Chunk 以 Block 为单位进行划分，大小为 64KB，每一个 Block 对应一个 32bit 的校验和。当读取一个 Chunk 副本时，Chunk Server 会将读取的数据和校验和进行比较，如果不匹配，就会返回错误，使 Client 选择其他 Chunk Server 上的副本。

2.1.3 系统管理技术

GFS 是一个分布式文件系统，包含从硬件到软件的整套解决方案。除了上面提到的 GFS 的一些关键技术外，还有相应的系统管理技术来支持整个 GFS 的应用，这些技术可能不一定为 GFS 独有。

1. 大规模集群安装技术

安装 GFS 的集群中通常有非常多的节点, 文献[1]中最大的集群超过 1000 个节点, 而现在的 Google 数据中心动辄有万台以上的机器在运行。因此迅速地安装、部署一个 GFS 的系统, 以及迅速地进行节点的系统升级等, 都需要相应的技术支撑。

2. 故障检测技术

GFS 是构建在不可靠的廉价计算机之上的文件系统, 由于节点数目众多, 故障发生十分频繁, 如何在最短的时间内发现并确定发生故障的 Chunk Server, 需要相关的集群监控技术。

3. 节点动态加入技术

当有新的 Chunk Server 加入时, 如果需要事先安装好系统, 那么系统扩展将是一件十分烦琐的事情。如果能够做到只需将裸机加入, 就会自动获取系统并安装运行, 那么将会大大减少 GFS 维护的工作量。

4. 节能技术

有关数据表明, 服务器的耗电成本大于当初的购买成本, 因此 Google 采用了多种机制来降低服务器的能耗, 例如对服务器主板进行修改, 采用蓄电池代替昂贵的 UPS (不间断电源系统), 提高能量的利用率。Rich Miller 在一篇关于数据中心的博客文章中表示, 这个设计让 Google 的 UPS 利用率达到 99.9%, 而一般数据中心只能达到 92%~95%。

2.2 分布式数据处理 MapReduce

MapReduce 是 Google 提出的一个软件架构, 是一种处理海量数据的并行编程模式, 用于大规模数据集 (通常大于 1TB) 的并行运算。“Map (映射)”、“Reduce (化简)”的概念和主要思想, 都是从函数式编程语言和矢量编程语言借鉴来的^[5]。正是由于 MapReduce 有函数式和矢量编程语言的共性, 使得这种编程模式特别适合于非结构化和结构化的海量数据的搜索、挖掘、分析与机器智能学习等。

2.2.1 产生背景

MapReduce 这种并行编程模式思想最早是在 1995 年提出的, 文献[6]首次提出了“map”和“fold”的概念, 和 Google 现在所使用的“Map”和“Reduce”思想相吻合。

与传统的分布式程序设计相比, MapReduce 封装了并行处理、容错处理、本地化计算、负载均衡等细节, 还提供了一个简单而强大的接口。通过这个接口, 可以把大尺度的计算自动地并发和分布执行, 使编程变得非常容易。另外, MapReduce 也具有较好的通用性, 大量不同的问题都可以简单地通过 MapReduce 来解决。

MapReduce 把对数据集的大规模操作, 分发给一个主节点管理下的各分节点共同完成, 通过这种方式实现任务的可靠执行与容错机制。在每个时间周期, 主节点都会对分节点的工作状态进行标记, 一旦分节点状态标记为死亡状态, 则这个节点的所有任务都将分配给其他分节点重新执行。

据相关统计, 每使用一次 Google 搜索引擎, Google 的后台服务器就要进行 1011 次运

算。这么庞大的运算量，如果没有好的负载均衡机制，有些服务器的利用率会很低，有些则会负荷太重，有些甚至可能死机，这些都会影响系统对用户的服务质量。而使用 MapReduce 这种编程模式，就保持了服务器之间的均衡，提高了整体效率。

2.2.2 编程模型

MapReduce 的运行模型如图 2-2 所示。图中有 M 个 Map 操作和 R 个 Reduce 操作。

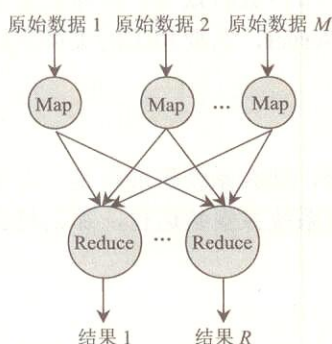


图 2-2 MapReduce 的运行模型

简单地说，一个 Map 函数就是对一部分原始数据进行指定的操作。每个 Map 操作都针对不同的原始数据，因此 Map 与 Map 之间是互相独立的，这使得它们可以充分并行化。一个 Reduce 操作就是对每个 Map 所产生的一部分中间结果进行合并操作，每个 Reduce 所处理的 Map 中间结果是互不交叉的，所有 Reduce 产生的最终结果经过简单连接就形成了完整的结果集，因此 Reduce 也可以在并行环境下执行。

在编程的时候，开发者需要编写两个主要函数：

Map: $(in_key, in_value) \rightarrow \{(key_j, value_j) | j = 1 \cdots k\}$

Reduce: $(key, [value_1, \cdots, value_m]) \rightarrow (key, final_value)$

Map 和 Reduce 的输入参数和输出结果根据应用的不同而有所不同。Map 的输入参数是 in_key 和 in_value ，它指明了 Map 需要处理的原始数据是哪些。Map 的输出结果是一组 $\langle key, value \rangle$ 对，这是经过 Map 操作后所产生的中间结果。在进行 Reduce 操作之前，系统已经将所有 Map 产生的中间结果进行了归类处理，使得相同 key 对应的一系列 value 能够集结在一起提供给一个 Reduce 进行归并处理，也就是说，Reduce 的输入参数是 $(key, [value_1, \cdots, value_m])$ 。Reduce 的工作是需要对这些对应相同 key 的 value 值进行归并处理，最终形成 $(key, final_value)$ 的结果。这样，一个 Reduce 处理了一个 key，所有 Reduce 的结果并在一起就是最终结果。

例如，假设我们想用 MapReduce 来计算一个大型文本文件中各个单词出现的次数，Map 的输入参数指明了需要处理哪部分数据，以“ \langle 在文本中的起始位置，需要处理的数据长度 \rangle ”表示，经过 Map 处理，形成一批中间结果“ \langle 单词，出现次数 \rangle ”。而 Reduce 函数处理中间结果，将相同单词出现的次数进行累加，得到每个单词总的出现次数。

2.2.3 实现机制

MapReduce 操作的执行流程图^[7]如图 2-3 所示。

用户程序调用 MapReduce 函数后，会引起下面的操作过程（图中的数字标示和下面的数字标示相同）：

(1) MapReduce 函数首先把输入文件分成 M 块，每块大概 16M~64MB（可以通过参数决定），接着在集群的机器上执行分派处理程序。

(2) 这些分派的执行程序中有有一个程序比较特别，它是主控程序 Master。剩下的执行程序都是作为 Master 分派工作的 Worker（工作机）。总共有 M 个 Map 任务和 R 个 Reduce 任务需要分派，Master 选择空闲的 Worker 来分配这些 Map 或 Reduce 任务。

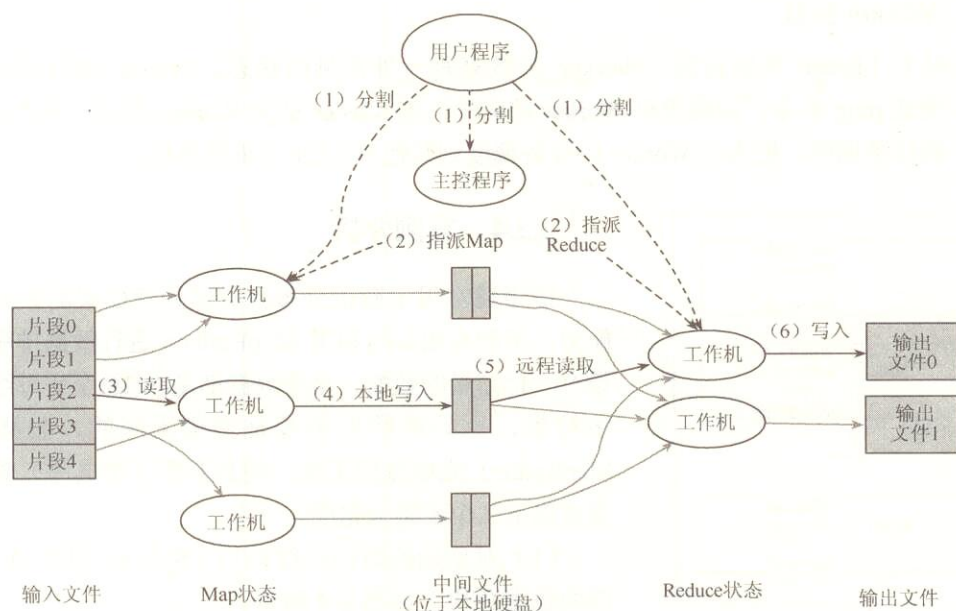


图 2-3 MapReduce 执行流程图

(3) 一个被分配了 Map 任务的 Worker 读取并处理相关的输入块。它处理输入的数据，并且将分析出的 $\langle \text{key}, \text{value} \rangle$ 对传递给用户定义的 Map 函数。Map 函数产生的中间结果 $\langle \text{key}, \text{value} \rangle$ 对暂时缓冲到内存。

(4) 这些缓冲到内存的中间结果将被定时写到本地硬盘，这些数据通过分区函数分成 R 个区。中间结果在本地硬盘的位置信息将被发送回 Master，然后 Master 负责把这些位置信息传送给 Reduce Worker。

(5) 当 Master 通知执行 Reduce 的 Worker 关于中间 $\langle \text{key}, \text{value} \rangle$ 对的位置时，它调用远程过程，从 Map Worker 的本地硬盘上读取缓冲的中间数据。当 Reduce Worker 读到所有的中间数据，它就使用中间 key 进行排序，这样可使相同 key 的值都在一起。因为有许多不同 key 的 Map 都对应相同的 Reduce 任务，所以，排序是必需的。如果中间结果集过于庞大，那么就需要使用外排序。

(6) Reduce Worker 根据每一个唯一中间 key 来遍历所有的排序后的中间数据，并且把 key 和相关的中间结果值集合传递给用户定义的 Reduce 函数。Reduce 函数的结果写到一个最终的输出文件。

(7) 当所有的 Map 任务和 Reduce 任务都完成的时候，Master 激活用户程序。此时 MapReduce 返回用户程序的调用点。

由于 MapReduce 在成百上千台机器上处理海量数据，所以容错机制是不可或缺的。总的说来，MapReduce 通过重新执行失效的地方来实现容错。

1. Master 失效

Master 会周期性地设置检查点 (checkpoint)，并导出 Master 的数据。一旦某个任务失效，系统就从最近的一个检查点恢复并重新执行。由于只有一个 Master 在运行，如果 Master 失效了，则只能终止整个 MapReduce 程序的运行并重新开始。

2. Worker 失效

相对于 Master 失效而言, Worker 失效算是一种常见的状态。Master 会周期性地给 Worker 发送 ping 命令, 如果没有 Worker 的应答, 则 Master 认为 Worker 失效, 终止对这个 Worker 的任务调度, 把失效 Worker 的任务调度到其他 Worker 上重新执行。

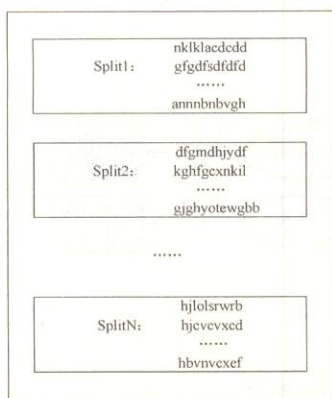


图 2-4 数据分块

2.2.4 案例分析

排序通常用于衡量分布式数据处理框架的数据处理能力, 下面介绍如何利用 MapReduce 进行数据排序。假设有一批海量的数据, 每个数据都是由 26 个字母组成的字符串, 原始的数据集合是完全无序的, 怎样通过 MapReduce 完成排序工作, 使其有序 (字典序) 呢? 可通过以下三个步骤来完成。

(1) 对原始的数据进行分割 (Split), 得到 N 个不同的数据分块, 如图 2-4 所示。

(2) 对每一个数据分块都启动一个 Map 进行处理。采用桶排序的方法, 每个 Map 中按照首字母将字符串分到 26 个不同的桶中, 图 2-5 是 Map 的过程及其得到的中间结果。

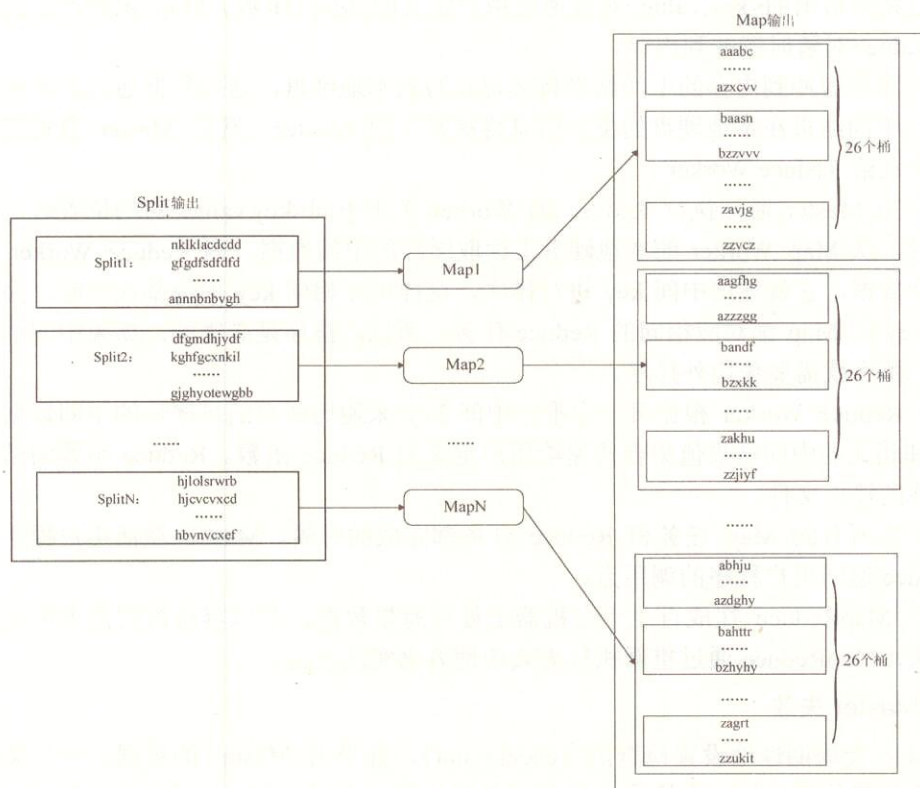


图 2-5 Map 过程及其得到的中间结果

(3) 对于 Map 之后得到的中间结果, 启动 26 个 Reduce。按照首字母将 Map 中不同桶中的字符串集合放置到相应的 Reduce 中进行处理。具体来说就是首字母为 a 的字符串全部放在 Reduce1 中处理, 首字母为 b 的字符串全部放在 Reduce2, 以此类推。每个 Reduce 对于其中的字符串进行排序, 结果直接输出。由于 Map 过程中已经做到了首字母有序, Reduce 输出的结果就是最终的排序结果。这一过程如图 2-6 所示。

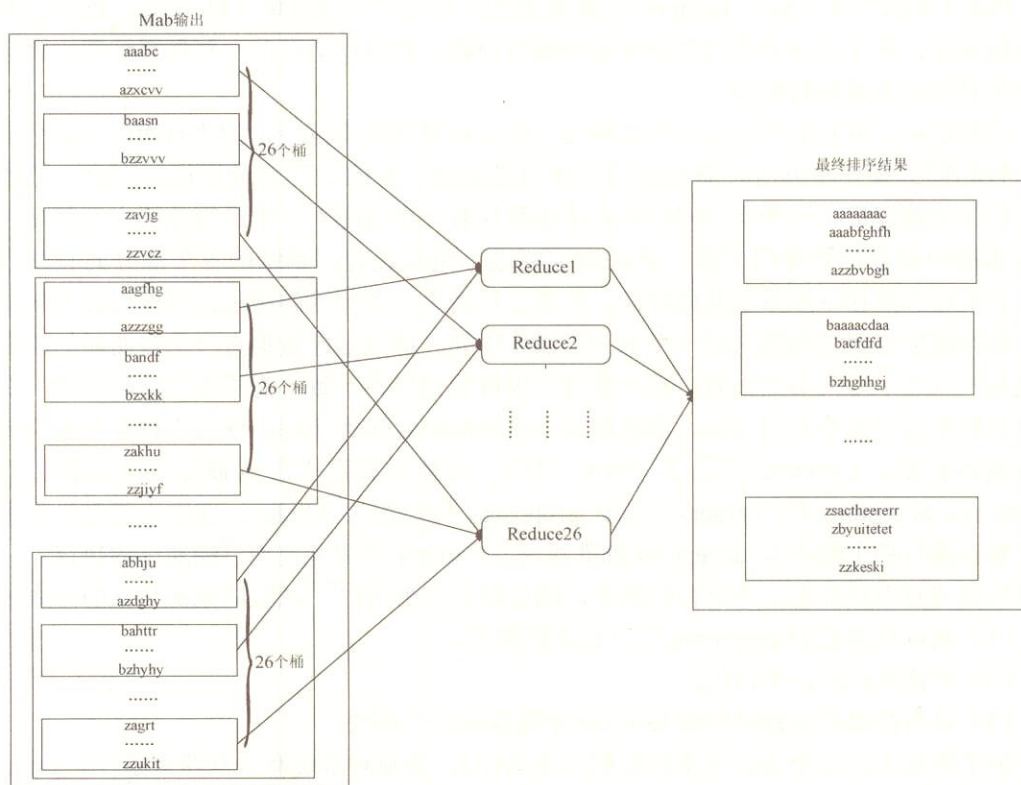


图 2-6 Reduce 过程

从上述过程中可以看出, 由于能够实现处理过程的完全并行化, 因此利用 MapReduce 处理海量数据是非常适合的。

2.3 分布式锁服务 Chubby

Chubby 是 Google 设计的提供粗粒度锁服务的一个文件系统, 它基于松耦合分布式系统, 解决了分布的一致性问题。通过使用 Chubby 的锁服务, 用户可以确保数据操作过程中的一致性。不过值得注意的是, 这种锁只是一种建议性的锁 (Advisory Lock) 而不是强制性的锁 (Mandatory Lock), 这种选择使系统具有更大的灵活性。

GFS 使用 Chubby 选取一个 GFS 主服务器, Bigtable 使用 Chubby 指定一个主服务器并发现、控制与其相关的子表服务器。除了最常用的锁服务之外, Chubby 还可以作为一个稳定的存储系统存储包括元数据在内的小数据。同时 Google 内部还使用 Chubby 进行名

字服务 (Name Server)。本节首先简要介绍 Paxos 算法, 因为 Chubby 内部一致性问题的实现用到了 Paxos 算法; 然后围绕 Chubby 系统的设计和实现展开讲解。

2.3.1 Paxos 算法

Paxos 算法^[14]是 Leslie Lamport 最先提出的一种基于消息传递 (Messages Passing) 的一致性算法, 用于解决分布式系统中的一致性问题。在目前所有的一致性算法中, 该算法最常用且被认为是最有效的。

简单地说, 分布式系统的一致性问题, 就是如何保证系统中初始状态相同的各个节点在执行相同的操作序列时, 看到的指令序列是完全一致的, 并且最终得到完全一致的结果。怎样才能保证在一个操作序列中每个步骤仅有一个值呢? 一个最简单的方案就是在分布式系统中设置一个专门节点, 在每次需要进行操作之前, 系统的各个部分向它发出请求, 告诉该节点接下来系统要做什么。该节点接受第一个到达的请求内容作为接下来的操作, 这样就能够保证系统只有一个唯一的操作序列。但是这样做也有一个很明显的缺陷, 那就是一旦这个专门节点失效, 整个系统就很可能出现不一致。为了避免这种情况, 在系统中必然要设置多个专门节点, 由这些节点来共同决定操作序列。针对这种多节点决定操作系列的情况, Lamport 提出了 Paxos 算法。在他的算法中节点被分成了三种类型: proposers、acceptors 和 learners。其中 proposers 提出决议 (Value, 实际上就是告诉系统接下来该执行哪个指令), acceptors 批准决议, learners 获取并使用已经通过的决议。一个节点可以兼有多重类型。在这种情况下, 满足以下三个条件^[15]就可以保证数据的一致性。

- (1) 决议只有在被 proposers 提出后才能批准。
- (2) 每次只批准一个决议。
- (3) 只有决议确定被批准后 learners 才能获取这个决议。

为了满足上述三个条件 (主要是第二个条件), 必须对系统有一些约束条件。Lamport 通过约束条件的不断加强, 最后得到了一个可以实际运用到算法中的完整约束条件。那么, 如何得到这个完整的约束条件呢? 在决议的过程中, proposers 将决议发送给 acceptors, acceptors 对决议进行批准, 批准后的决议才能成为正式的决议。决议的批准采用少数服从多数原则, 即大多数 acceptors 接受的决议将成为最终的正式决议。从集合论的观点来看, 两组 “多数派” (Majority) 至少有一个公共的 acceptor。如果每个 acceptor 只能接受一个决议, 则条件 (2) 就能够得到保证, 因此不难得到第一个约束条件^[15]:

P1: 每个 acceptor 只接受它得到的第一个决议。

P1 表明一个 acceptor 可以收到多个决议, 为了区分, 对每个决议进行编号, 后到的决议编号大于先到的决议编号。约束条件 P1 不是很完备, 假设系统中一半的 acceptors 接受了决议 1, 剩下的一半接受了决议 2。此时仅靠约束 P1 是根本无法得到一个 “多数派”, 从而无法得到一个正式的决议。进一步加强约束得到:

P2: 一旦某个决议得到通过, 之后通过的决议必须和该决议保持一致。

P1 和 P2 能够保证条件 (2)。对 P2 稍作加强得到:

P2a: 一旦某个决议 v 得到通过, 之后任何 acceptor 再批准的决议必须是 v 。

表面上看起来已经不存在什么问题了, 但实际上 P2a 和 P1 是有矛盾的。考虑下面这

种情况：假设在系统得到决议 v 的过程中一个 proposer 和一个 acceptor 因为出现问题并没有参与到决议的表决中。在得到决议 v 之后出现问题 proposer 和 acceptor 恢复过来，此时这个 proposer 提出一个决议 w (w 不等于 v) 给这个 acceptor。如果按照 P1，这个 acceptor 应该接受这个决议 w ，但是按照 P2a，则不应该接受这个决议。所以还需进一步加强约束条件：

P2b：一旦某个决议 v 得到通过，之后任何 proposer 再提出的决议必须是 v 。

满足 P1 和 P2b 就能够保证条件 (2)，而且彼此之间不存在矛盾。但是 P2b 很难通过一种技术手段来实现它，因此提出了一个蕴涵 P2b 的约束 P2c：

P2c：如果一个编号为 n 的提案具有值 v ，那么存在一个“多数派”，要么它们中没有谁批准过编号小于 n 的任何提案，要么它们进行的最近一次批准具有值 v 。

为了保证决议的唯一性，acceptors 也要满足一个约束条件：当且仅当 acceptors 没有收到编号大于 n 的请求时，acceptors 才批准编号为 n 的提案。

在这些约束条件的基础上，可以将一个决议的通过分成如下两个阶段^[15]。

(1) 准备阶段：proposers 选择一个提案并将它的编号设为 n ，然后将它发送给 acceptors 中的一个“多数派”。Acceptors 收到后，如果提案的编号大于它已经回复的所有消息，则 acceptors 将自己上次的批准回复给 proposers，并不再批准小于 n 的提案。

(2) 批准阶段：当 proposers 接收到 acceptors 中的这个“多数派”的回复后，就向回复请求的 acceptors 发送 accept 请求，在符合 acceptors 一方的约束条件下，acceptors 收到 accept 请求后即批准这个请求。

为了减少决议发布过程中的消息量，acceptors 将这个通过的决议发送给 learners 的一个子集，然后由这个子集中的 learners 去通知所有其他的 learners。一般情况下，以上的算法过程就可以成功地解决一致性问题，但是也有特殊情况。根据算法一个编号更大的提案会终止之前的提案过程，如果两个 proposer 在这种情况下都转而提出一个编号更大的提案，那么就可能陷入活锁。此时需要选举出一个 president，仅允许 president 提出提案。

以上简要地介绍了 Paxos 算法的核心内容，关于更多的实现细节读者可以参考 Lamport 关于 Paxos 算法实现的文章。

2.3.2 Chubby 系统设计

通常情况下 Google 的一个数据中心仅运行一个 Chubby 单元^[13] (Chubby cell，下面会有详细讲解)，这个单元需要支持包括 GFS、Bigtable 在内的众多 Google 服务，因此，在设计 Chubby 时候，必须充分考虑系统需要实现的目标以及可能出现的各种问题。

Chubby 的设计目标主要有以下几点。

(1) 高可用性和高可靠性。这是系统设计的首要目标，在保证这一目标的基础上再考虑系统的吞吐量和存储能力。

(2) 高扩展性。将数据存储在价格较为低廉的 RAM，支持大规模用户访问文件。

(3) 支持粗粒度的建议性锁服务。提供这种服务的根本目的是提高系统的性能。

(4) 服务信息的直接存储。可以直接存储包括元数据、系统参数在内的有关服务信息，而不需要再维护另一个服务。

(5) 支持通报机制。客户可以及时地了解到事件的发生。

(6) 支持缓存机制。通过一致性缓存将常用信息保存在客户端, 避免了频繁地访问主服务器。

Google 没有直接实现一个包含了 Paxos 算法的函数库, 而是在 Paxos 算法的基础上设计了一个全新的锁服务 Chubby。Chubby 中涉及的一致性问题的都由 Paxos 解决, 除此之外 Chubby 中还添加了一些新的功能特性。这种设计主要是考虑到以下几个问题^[13]。

(1) 通常情况下开发者在开发的初期很少考虑系统的一致性问题, 但是随着开发的不断进行, 这种问题会变得越来越严重。单独的锁服务可以保证原有系统的架构不会发生改变, 而使用函数库的话很可能需要对系统的架构做出大幅度的改动。

(2) 系统中很多事件的发生是需要告知其他用户和服务器的, 使用一个基于文件系统的锁服务可以将这些变动写入文件中。这样其他需要了解这些变动的用户和服务器直接访问这些文件即可, 避免了因大量的系统组件之间的事件通信带来的系统性能下降。

(3) 基于锁的开发接口容易被开发者接受。虽然在分布式系统中锁的使用会有很大的不同, 但是和一致性算法相比, 锁显然被更多的开发者所熟知。

Paxos 算法的实现过程中需要一个“多数派”就某个值达成一致, 进而才能得到一个分布式一致性状态。这个过程本质上就是分布式系统中常见的 quorum 机制 (quorum 原意是法定人数, 简单说来就是根据少数服从多数的选举原则产生一个决议)。为了保证系统的高可用性, 需要若干台机器, 但是使用单独的锁服务的话一台机器也能保证这种高可用性。也就是说, Chubby 在自身服务的实现时利用若干台机器实现了高可用性, 而外部用户利用 Chubby 则只需一台机器就可以保证高可用性。

正是考虑到以上几个问题, Google 设计了 Chubby, 而不是单独地维护一个函数库 (实际上, Google 有这样一个独立于 Chubby 的函数库, 不过一般情况下并不会使用)。在设计的过程中有一些细节问题也值得我们关注, 比如在 Chubby 系统中采用了建议性的锁而没有采用强制性的锁。两者的根本区别在于用户访问某个被锁定的文件时, 建议性的锁不会阻止访问, 而强制性的锁则会阻止访问, 实际上这是为了方便系统组件之间的信息交互。另外, Chubby 还采用了粗粒度 (Coarse-Grained) 锁服务而没有采用细粒度 (Fine-Grained) 锁服务, 两者的差异在于持有锁的时间。细粒度的锁持有时间很短, 常常只有几秒甚至更少, 而粗粒度的锁持有的时间可长达几天, 选择粗粒度的锁可以减少频繁换锁带来的系统开销。

如图 2-7^[13]所示是 Chubby 的基本架构。很明显, Chubby 被划分成两个部分: 客户端和服务端, 客户端和服务端之间通过远程过程调用 (RPC) 来连接。在客户这一端每个客户应用程序都有一个 Chubby 程序库 (Chubby Library), 客户端的所有应用都是通过调用这个库中的相关函数来完成的。服务器一端称为 Chubby 单元, 一般是由五个称为副本 (Replica) 的服务器组成的, 这五个副本在配置上完全一致, 并且在系统刚开始时处于对等地位。

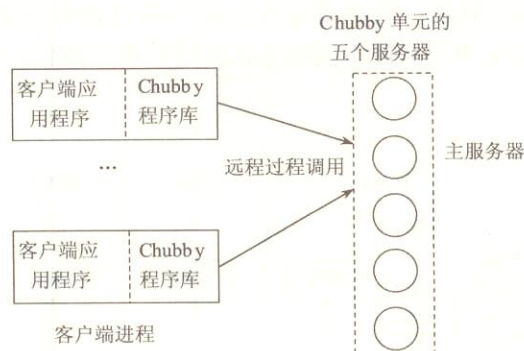


图 2-7 Chubby 的基本架构

2.3.3 Chubby 中的 Paxos

一致性问题是在 Chubby 需要解决的一个关键性问题，那么 Paxos 算法在 Chubby 中究竟是怎样起作用的呢？

为了了解 Paxos 算法作用，需要将单个副本的结构剖析来看，单个 Chubby 副本结构如图 2-8^[16] 所示。

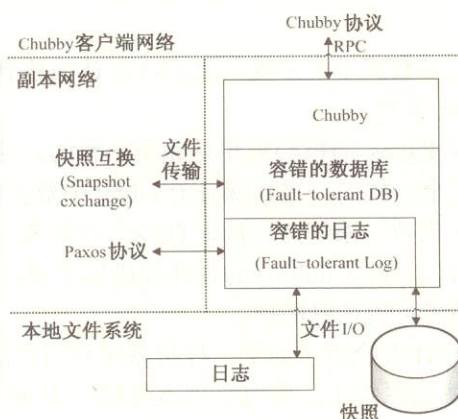


图 2-8 单个 Chubby 副本结构

从图中可以看出，单个副本主要由以下三个层次组成。

(1) 最底层是一个容错的日志，该日志对于数据库的正确性提供了重要的支持。不同副本上日志的一致性正是通过 Paxos 算法来保证的。副本之间通过特定的 Paxos 协议进行通信，同时本地文件中还保存有一份同 Chubby 中相同的日志数据。

(2) 最底层之上是一个容错的数据库，这个数据库主要包括一个快照（Snapshot）和一个记录数据库操作的重播日志（Replay-log），每一次的数据库操作最终都将提交至日志中。和容错的日志类似的是，本地文件中也保存着一份数据库数据副本。

(3) Chubby 构建在这个容错的数据库之上，Chubby 利用这个数据库存储所有的数据。Chubby 的客户端通过特定的 Chubby 协议和单个的 Chubby 副本进行通信。

由于副本之间的一致性问题，客户端每次向容错的日志中提交新的值（value）时，Chubby 就会自动调用 Paxos 构架保证不同副本之间数据的一致性。图 2-9^[16]就显示了这个过程。

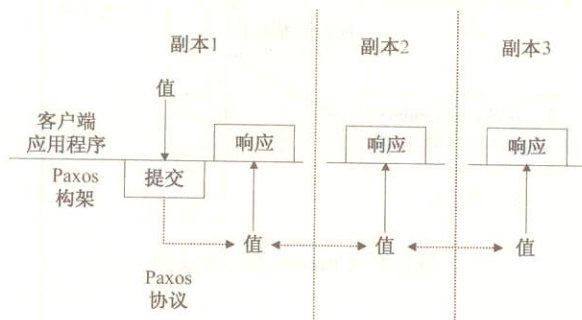


图 2-9 容错日志的 API

结合图 2-9 来看，在 Chubby 中 Paxos 算法的实际作用为如下三个过程。

- (1) 选择一个副本成为协调者（Coordinator）。
- (2) 协调者从客户提交的值中选择一个，然后通过一种被称为 **accept** 的消息广播给所有的副本，其他的副本收到广播之后，可以选择接受或者拒绝这个值，并将决定结果反馈给协调者。
- (3) 一旦协调者收到大多数副本的接受信息后，就认为达到了一致性，接着协调者向相关的副本发送一个 **commit** 消息。

上述三个过程实际上跟 Paxos 的核心思想是完全一致的，这些过程保证提交到不同副本上容错日志中的数据是完全一致的，进而保证 Chubby 中数据的一致性。

由于单个的协调者可能失效，系统允许同时有多个协调者，但多个协调者可能会导致多个协调者提交了不同的值。对此 Chubby 的设计者借鉴了 Paxos 中的两种解决机制：给协调者指派序号或限制协调者可以选择的值。

针对前者，Chubby 的设计者给出了如下一种指派序号的方法。

- (1) 在一个有 n 个副本的系统中，为每个副本分配一个 $id\ i_r$ ，其中 $0 \leq i_r \leq n-1$ 。则副本的序号 $s = k \times n + i_r$ ，其中 k 的初始值为 0。

- (2) 某个副本想成为协调者之后，它根据规则生成一个比它以前的序号更大的序号（实际上就是提高 k 的值），并将这个序号通过 **propose** 消息广播给其他所有的副本。

- (3) 如果接受到广播的副本发现该序号比它以前见过的序号都大，则向发出广播的副本返回一个 **promise** 消息，并且承诺不再接受旧的协调者发送的消息。如果大多数副本都返回了 **promise** 消息，则新的协调者就产生了。

对于后一种解决方法，Paxos 强制新的协调者必须选择和前任相同的值。

为了提高系统的效率，Chubby 做了一个重要的优化，那就是在选择某一个副本作为协调者之后就长期不变，此时协调者就被称为主服务器（Master）。产生一个主服务器避免了同时有多个协调者而带来的一些问题。

在 Chubby 中，客户端的数据请求都是由主服务器来完成，Chubby 保证在一定的时间内有且仅有一个主服务器，这个时间就称为主服务器租约期（Master Lease）。如果某个服务

器被连续推举为主服务器的话, 这个租约期就会不断地被更新。租约期内所有的客户端请求都由主服务器处理。客户端如果需要确定主服务器的位置, 可以向 DNS 发送一个主服务器定位请求, 非主服务器的副本将对该请求做出回应, 通过这种方式客户端能够快速、准确地对主服务器做出定位。客户端和服务端之间的通信过程将在 2.3.5 节详细介绍。

需要注意的是, Chubby 对于 Paxos 论文中未提及的一些技术细节进行了补充, 所以 Chubby 的实现是基于 Paxos, 但其技术手段更加的丰富, 更具有实践性。但这也导致了最终实现的 Chubby 不是一个完全经过理论上验证的系统。

2.3.4 Chubby 文件系统

Chubby 系统本质上就是一个分布式的、存储大量小文件的文件系统, 它所有的操作都是在文件的基础上完成的。例如在 Chubby 最常用的锁服务中, 每一个文件就代表了一个锁, 用户通过打开、关闭和读取文件, 获取共享 (Shared) 锁或独占 (Exclusive) 锁。选举主服务器的过程中, 符合条件的服务器都同时申请打开某个文件并请求锁住该文件。成功获得锁的服务器自动成为主服务器并将其地址写入这个文件夹, 以便其他服务器和用户获取主服务器的地址信息。

Chubby 的文件系统^[13]和 UNIX 类似。例如在文件名 “/ls/foo/wombat/pouch” 中, ls 代表 lock service, 这是所有 Chubby 文件系统的共有前缀; foo 是某个单元的名称; /wombat/pouch 则是 foo 这个单元上的文件目录或者文件名。由于 Chubby 自身的特殊服务要求, Google 对 Chubby 做了一些与 UNIX 不同的改变。例如 Chubby 不支持内部文件的移动; 不记录文件的最后访问时间; 另外在 Chubby 中并没有符号连接 (Symbolic Link, 又叫软连接, 类似于 Windows 系统中的快捷方式) 和硬连接 (Hard Link, 类似于别名) 的概念。在具体实现时, 文件系统由许多节点组成, 分为永久型和临时型, 每个节点就是一个文件或目录。节点中保存着包括 ACL (Access Control List, 访问控制列表, 将在 2.3.6 节讲解) 在内的多种系统元数据。为了用户能够及时了解元数据的变动, 系统规定每个节点的元数据都应当包含以下四种单调递增的 64 位编号^[13]。

- (1) 实例号 (Instance Number): 新节点实例号必定大于旧节点的实例号。
- (2) 内容生成号 (Content Generation Number): 文件内容修改时该号增加。
- (3) 锁生成号 (Lock Generation Number): 锁被用户持有时该号增加。
- (4) ACL 生成号 (ACL Generation Number): ACL 名被覆写时该号增加。

用户在打开某个节点的同时会获取一个类似于 UNIX 中文件描述符 (File Descriptor) 的句柄^[13] (Handles), 这个句柄由以下三个部分组成。

- (1) 校验数位 (Check Digit): 防止其他用户创建或猜测这个句柄。
- (2) 序号 (Sequence Number): 用来确定句柄是由当前还是以前的主服务器创建的。
- (3) 模式信息 (Mode Information): 用于新的主服务器重新创建一个旧的句柄。

在实际的执行中, 为了避免所有的通信都使用序号带来的系统开销增长, Chubby 引入了 sequencer 的概念。sequencer 实际上就是一个序号, 只能由锁的持有者在获取锁时向系统发出请求来获得。这样一来 Chubby 系统中只有涉及锁的操作才需要序号, 其他一概不用。在文件操作中, 用户可以将句柄看做一个指向文件系统的指针。这个指针支持一系列的操作, 常用的句柄操作函数及其作用如表 2-1 所示。

表 2-1 常用的句柄函数及其作用

函数名称	作用
Open()	打开某个文件或者目录来创建句柄
Close()	关闭打开的句柄，后续的任何操作都将中止
Poison()	中止当前未完成及后续的操作，但不关闭句柄
GetContentsAndStat()	返回文件内容及元数据
GetStat()	只返回文件元数据
ReadDir()	返回子目录名称及其元数据
SetContents()	向文件中写入内容
SetACL()	设置 ACL 名称
Delete()	如果该节点没有子节点的话则执行删除操作
Acquire()	获取锁
Release()	释放锁
GetSequencer()	返回一个 sequencer
SetSequencer()	将 sequencer 和某个句柄进行关联
CheckSequencer()	检查某个 sequencer 是否有效

2.3.5 通信协议

客户端和主服务器之间的通信是通过 KeepAlive 握手协议来维持的，这一通信过程的简单示意图如图 2-10^[13]所示。

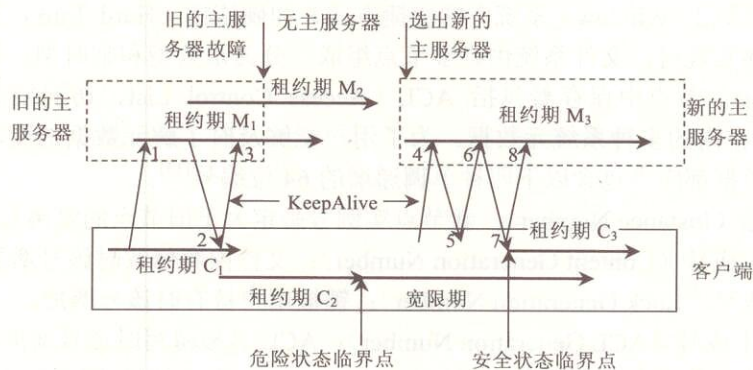


图 2-10 Chubby 客户端与服务器的通信过程

图 2-10 中，从左到右的水平方向表示时间在增加，斜向上的箭头表示一次 KeepAlive 请求，斜向下的箭头则是主服务器的一次回应。M₁、M₂、M₃ 表示不同的主服务器租约期。C₁、C₂、C₃ 则是客户端对主服务器租约期时长做出的一个估计。KeepAlive 是周期发送的一种信息，它主要有两方面的功能：延迟租约的有效期和携带事件信息告诉用户更新。主要的事件包括文件内容被修改、子节点的增加、删除和修改、主服务器出错、句柄失效等。正常情况下，通过 KeepAlive 握手协议租约期会得到延长，事件也会及时地通知给用户。但是由于系统有一定的失效概率，引入故障处理措施是很有必要的。通常情况下系统可能会出现两种故障：客户端租约期过期和主服务器故障，对于这两种情况系统有着

不同的应对方式。

1. 客户端租约过期

刚开始时, 客户端向主服务器发出一个 KeepAlive 请求 (见图 2-10 中的 1), 如果有需要通知的事件时则主服务器会立刻做出回应, 否则主服务器并不立刻对这个请求做出回应, 而是等到客户端的租约期 C_1 快结束的时候才做出回应 (见图 2-10 中的 2), 并更新主服务器租约期为 M_2 。客户端在接到这个回应后认为该主服务器仍处于活跃状态, 于是将租约期更新为 C_2 并立刻发出新的 KeepAlive 请求 (见图 2-10 中的 3)。同样的, 主服务器可能不是立刻回应而是等待 C_2 接近结束, 但是在这个过程中主服务器出现故障停止使用。在等待了一段时间后 C_2 到期, 由于并没有收到主服务器的回应, 系统向客户端发出一个危险 (Jeopardy) 事件, 客户端清空并暂时停用自己的缓存, 从而进入一个称为宽限期 (Grace Period) 的危险状态。这个宽限期默认是 45 秒。在宽限期内, 客户端不会立刻断开其与服务器端的联系, 而是不断地做探询。图 2-10 中新的主服务器很快被重新选出, 当它接到客户端的第一个 KeepAlive 请求 (见图 2-10 中的 4) 时会拒绝 (见图 2-10 中的 5), 因为这个请求的纪元号 (Epoch Number) 错误。不同主服务器的纪元号不相同, 客户端的每次请求都需要这个号来保证处理的请求是针对当前的主服务器。客户端在主服务器拒绝之后会使用新的纪元号来发送 KeepAlive 请求 (见图 2-10 中的 6)。新的主服务器接受这个请求并立刻做出回应 (见图 2-10 中的 7)。如果客户端接收到这个回应的的时间仍处于宽限期内, 系统会恢复到安全状态, 租约期更新为 C_3 。如果在宽限期未接到主服务器的相关回应, 客户端终止当前的会话。

2. 主服务器出错

在客户端和主服务器端进行通信时可能会遇到主服务器故障, 图 2-10 就出现了这种情况。正常情况下旧的主服务器出现故障后系统会很快地选举出新的主服务器, 新选举的主服务器在完全运行前需要经历以下九个步骤^[13]。

(1) 产生一个新的纪元号以便今后客户端通信时使用, 这能保证当前的主服务器不必处理针对旧的主服务器的请求。

(2) 只处理主服务器位置相关的信息, 不处理会话相关的信息。

(3) 构建处理会话和锁所需的内部数据结构。

(4) 允许客户端发送 KeepAlive 请求, 不处理其他会话相关的信息。

(5) 向每个会话发送一个故障事件, 促使所有的客户端清空缓存。

(6) 等待直到所有的会话都收到故障事件或会话终止。

(7) 开始允许执行所有的操作。

(8) 如果客户端使用了旧的句柄则需要为其重新构建新的句柄。

(9) 一定时间段后 (1 分钟), 删除没有被打开过的临时文件夹。

如果这一过程在宽限期内顺利完成, 则用户不会感觉到任何故障的发生, 也就是说新旧主服务器的替换对于用户来说是透明的, 用户感觉到的仅仅是一个延迟。使用宽限期的好处正是如此。

在系统实现时, Chubby 还使用了一致性客户端缓存 (Consistent Client-Side Caching) 技术, 这样做的目的是减少通信压力, 降低通信频率。在客户端保存一个和单元上数据一致的本地缓存, 需要时客户可以直接从缓存中取出数据而不用再和主服务器通信。当某个

文件数据或者元数据需要修改时，主服务器首先将这个修改阻塞；然后通过查询主服务器自身维护的一个缓存表，向对修改的数据进行了缓存的所有客户端发送一个无效标志（Invalidation）；客户端收到这个无效标志后会返回一个确认（Acknowledge），主服务器在收到所有的确认后才解除阻塞并完成这次修改。这个过程的执行效率非常高，仅仅需要发送一次无效标志即可，因为对于没有返回确认的节点，主服务器直接认为其是未缓存的。

2.3.6 正确性与性能

1. 一致性

前面提到过每个 Chubby 单元是由五个副本组成的，这五个副本中需要选举产生一个主服务器，这种选举本质上就是一个一致性问题。在实际的执行过程中，Chubby 使用 Paxos 算法来解决这个问题。

主服务器产生后客户端的所有读写操作都是由主服务器来完成的。读操作很简单，客户直接从主服务器上读取所需数据即可，但是写操作就会涉及数据一致性的问题。为了保证客户的写操作能够同步到所有的服务器上，系统再次利用了 Paxos 算法。因此，可以看出 Paxos 算法在分布式一致性问题中的作用是巨大的。

2. 安全性

Chubby 采用的是 ACL 形式的安全保障措施。系统中有三种 ACL 名^[13]，分别是写 ACL 名（Write ACL Name）、读 ACL 名（Read ACL Name）和变更 ACL 名（Change ACL Name）。只要不被覆写，子节点都是直接继承父节点的 ACL 名。ACL 同样被保存在文件中，它是节点元数据的一部分，用户在进行相关操作时首先需要通过 ACL 来获取相应的授权。图 2-11 是一个用户成功写文件所需经历的过程。

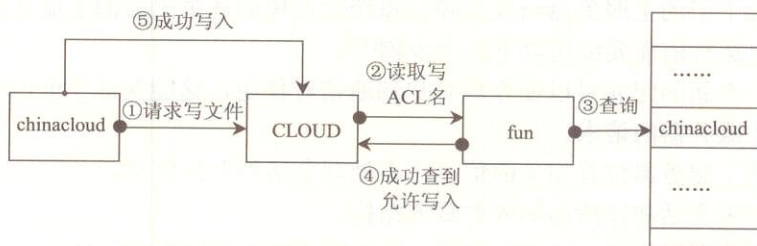


图 2-11 Chubby 的 ACL 机制

用户 chinacloud 提出向文件 CLOUD 中写入内容的请求。CLOUD 首先读取自身的写 ACL 名 fun，接着在 fun 中查到了 chinacloud 这一行记录，于是返回信息允许 chinacloud 对文件进行写操作，此时 chinacloud 才被允许向 CLOUD 写入内容。其他的操作和写操作类似。

3. 性能优化

为了满足系统的高可扩展性，Chubby 目前已经采取了一些措施^[13]。比如提高主服务器默认的租约期、使用协议转换服务将 Chubby 协议转换成较简单的协议、客户端一致性缓存等。除此之外，Google 的工程师们还考虑使用代理（Proxy）和分区（Partition）技术，虽然目前这两种技术并没有实际使用，但是在设计时还是被包含进系统，不排除将来

使用的可能。代理可以减少主服务器处理 KeepAlive 以及读请求带来的服务器负载，但是它并不能减少写操作带来的通信量。Google 自己的数据统计表明，在所有的请求中，写请求仅占极少的一部分，几乎可以忽略不计。使用分区技术的话可以将一个单元的命名空间（Name Space）划分成 N 份。除了少量的跨分区通信外，大部分的分区都可以独自地处理服务请求。通过分区可以减少各个分区上的读写通信量，但不能减少 KeepAlive 请求的通信量。因此，如果需要的话，将代理和分区技术结合起来使用才可以明显提高系统同时处理的服务请求量。

2.4 分布式结构化数据表 Bigtable

Bigtable 是 Google 开发的基于 GFS 和 Chubby 的分布式存储系统。Google 的很多数据，包括 Web 索引、卫星图像数据等在内的海量结构化和半结构化数据，都存储在 Bigtable 中。从实现上看，Bigtable 并没有什么全新的技术，但是如何选择合适的技术并将这些技术高效、巧妙地结合在一起恰恰是最大的难点。Bigtable 在很多方面和数据库类似，但它并不是真正意义上的数据库。通过本节的学习，读者将会对 Bigtable 的数据模型、系统架构、实现以及它使用的一些数据库技术有一个全面的认识。

2.4.1 设计动机与目标

Google 设计 Bigtable 的动机主要有如下三个方面。

(1) 需要存储的数据种类繁多。Google 目前向公众开放的服务很多，需要处理的数据类型也非常多。包括 URL、网页内容、用户的个性化设置在内的数据都是 Google 需要经常处理的。

(2) 海量的服务请求。Google 运行着目前世界上最繁忙的系统，它每时每刻处理的客户服务请求数量是普通的系统根本无法承受的。

(3) 商用数据库无法满足 Google 的需求。一方面现有商用数据库的设计着眼点在于其通用性，根本无法满足 Google 的苛刻服务要求，而且在数量庞大的服务器上根本无法成功部署普通的商用数据库。另一方面对于底层系统的完全掌控会给后期的系统维护、升级带来极大的便利。

在仔细考察了 Google 的日常需求后，Bigtable 开发团队确定 Bigtable 设计应达到如下几个基本目标。

(1) 广泛的适用性。Bigtable 是为了满足一系列 Google 产品而并非特定产品的存储要求。

(2) 很强的可扩展性。根据需要随时可以加入或撤销服务器。

(3) 高可用性。对于客户来说，有时候即使短暂的服务中断也是不能忍受的。Bigtable 设计的重要目标之一就是确保几乎所有的情况下系统都可用。

(4) 简单性。底层系统的简单性既可以减少系统出错的概率，也为上层应用的开发带来便利。

在目标确定之后，Google 希望巧妙地结合各种数据库技术，扬长避短。最终实现的系统也确实达到了原定的目标。下面详细讲解 Bigtable。

2.4.2 数据模型

Bigtable 是一个分布式多维映射表，表中的数据通过一个行关键字（Row Key）、一个列关键字（Column Key）以及一个时间戳（Time Stamp）进行索引。Bigtable 对存储在其中的数据不做任何解析，一律看做字符串，具体数据结构的实现需要用户自行处理。

Bigtable 的存储逻辑可以表示为：

$(\text{row: string, column: string, time: int64}) \rightarrow \text{string}$

Bigtable 数据的存储格式如图 2-12 所示^[8]。

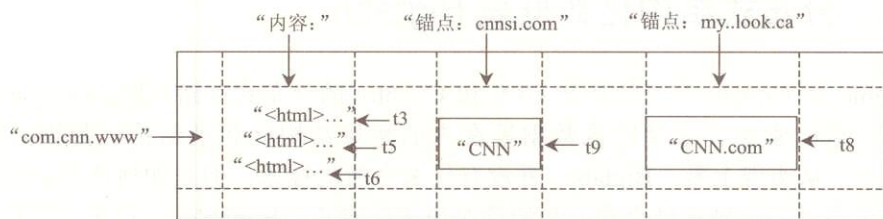


图 2-12 Bigtable 数据模型

1. 行

Bigtable 的行关键字可以是任意的字符串，但是大小不能够超过 64KB。Bigtable 和传统的关系型数据库有很大不同，它不支持一般意义上的事务，但能保证对于行的读写操作具有原子性（Atomic）。表中数据都是根据行关键字进行排序的，排序使用的是词典序。图 2-12 是 Bigtable 数据模型的一个典型实例，其中 com.cnn.www 就是一个行关键字。不直接存储网页地址而将其倒排是 Bigtable 的一个巧妙设计。这样做至少会带来以下两个好处。

(1) 同一地址域的网页会被存储在表中的连续位置，有利于用户查找和分析。

(2) 倒排便于数据压缩，可以大幅提高压缩率。

由于规模问题，单个的大表不利于数据的处理，因此 Bigtable 将一个表分成了很多子表（Tablet），每个子表包含多个行。子表是 Bigtable 中数据划分和负载均衡的基本单位。有关子表的内容在 2.4.5 节详细讲解。

2. 列

Bigtable 并不是简单地存储所有的列关键字，而是将其组织成所谓的列族（Column Family），每个族中的数据都属于同一个类型，并且同族的数据会被压缩在一起保存。引入了列族的概念之后，列关键字就采用下述的语法规则来定义：

族名：限定词（family: qualifier）

族名必须有意义，限定词则可以任意选定。在图 2-12 中，内容（Contents）、锚点（Anchor，就是 HTML 中的链接）都是不同的族。而 cnnsi.com 和 my.look.ca 则是锚点族中不同的限定词。通过这种方式组织的数据结构清晰明了，含义也很清楚。族同时也是 Bigtable 中访问控制（Access Control）的基本单元，也就是说访问权限的设置是在族这一级别上进行的。

3. 时间戳

Google 的很多服务比如网页检索和用户的个性化设置等都需要保存不同时间的数据, 这些不同的数据版本必须通过时间戳来区分。图 2-12 中内容列的 t3、t5 和 t6 表明其中保存了在 t3、t5 和 t6 这三个时间获取的网页。Bigtable 中的时间戳是 64 位整型数, 具体的赋值方式可以采取系统默认的方式, 也可以用户自行定义。

为了简化不同版本的数据管理, Bigtable 目前提供了两种设置: 一种是保留最近的 N 个不同版本, 图 2-12 中数据模型采取的就是这种方法, 它保存最新的三个版本数据。另一种就是保留限定时间内的所有不同版本, 比如可以保存最近 10 天的所有不同版本数据。失效的版本将会由 Bigtable 的垃圾回收机制自动处理。

2.4.3 系统架构

Bigtable 是在 Google 的另外三个云计算组件基础之上构建的, 其基本架构如图 2-13 所示^[11]。

图中 WorkQueue 是一个分布式的任务调度器, 它主要被用来处理分布式系统队列分组和任务调度, 关于其实现 Google 并没有公开。在前面已经讲过, GFS^[9]是 Google 的分布式文件系统, 在 Bigtable 中 GFS 主要用来存储子表数据以及一些日志文件。Bigtable 还需要一个锁服务的支持, Bigtable 选用了 Google 自己开发的分布式锁服务 Chubby。在 Bigtable 中 Chubby 主要有以下几个作用^[10]。

- (1) 选取并保证同一时间内只有一个主服务器 (Master Server)。
- (2) 获取子表的位置信息。
- (3) 保存 Bigtable 的模式信息及访问控制列表。

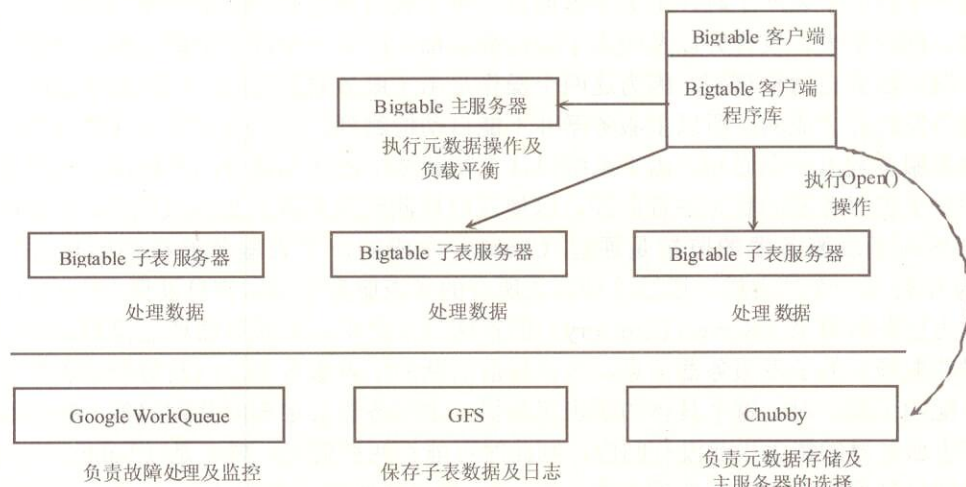


图 2-13 Bigtable 基本架构

另外在 Bigtable 的实际执行过程中, Google 的 MapReduce 和 Sawzall 也被用来改善其性能, 不过需要注意的是这两个组件并不是实现 Bigtable 所必需的。

Bigtable 主要由三个部分组成: 客户端程序库 (Client Library)、一个主服务器 (Master Server) 和多个子表服务器 (Tablet Server), 这三个部分在图 2-13 中都有相应的

表示。从图 2-13 可以看出，客户访问 Bigtable 服务时，首先要利用其库函数执行 Open() 操作来打开一个锁（实际上就是获取了文件目录），锁打开以后客户端就可以和子表服务器进行通信了。和许多具有单个主节点的分布式系统一样，客户端主要与子表服务器通信，几乎不和主服务器进行通信，这使得主服务器的负载大大降低。主服务主要进行一些元数据的操作以及子表服务器之间的负载调度问题，实际的数据是存储在子表服务器上的。

2.4.4 主服务器

主服务器的主要作用如图 2-14 所示。

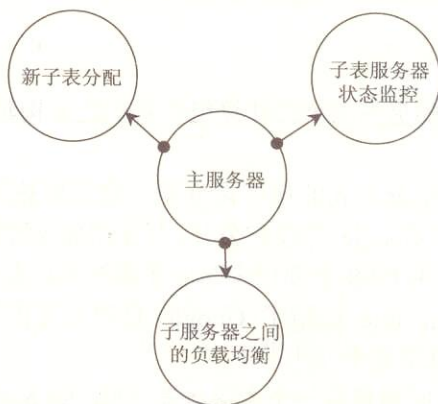


图 2-14 主服务器的主要作用

当一个新的子表产生时，主服务器通过一个加载命令将其分配给一个空间足够的子表服务器。创建新表、表合并以及较大子表的分裂都会产生一个或多个新子表。对于前面两种，主服务器会自动检测到，因为这两个操作是由主服务器发起的，而较大子表的分裂是由子服务发起并完成的，所以主服务器并不能自动检测到，因此在分割完成之后子服务器需要向主服务发出一个通知。由于系统设计之初就要求能达到良好的扩展性，所以主服务器必须对子表服务器的状态进行监控，以便及时检测到服务器的加入或撤销。Bigtable 中主服务器对子表服务器的监控是通过 Chubby 完成的，子表服务器在初始化时都会从 Chubby 中得到一个独占锁。通过这种方式所有的子表服务器基本信息被保存在 Chubby 中一个称为服务器目录（Server Directory）的特殊目录之中。主服务器通过检测这个目录可以随时获取最新的子表服务器信息，包括目前活跃的子表服务器，以及每个子表服务器上现已分配的子表。对于每个具体的子表服务器，主服务器会定期向其询问独占锁的状态。如果子表服务器的锁丢失或没有回应，则此时可能有两种情况，要么是 Chubby 出现了问题（虽然这种概率很小，但的确存在，Google 自己也做过相关测试），要么是子表服务器自身出现了问题。对此主服务器首先自己尝试获取这个独占锁，如果失败说明 Chubby 服务出现问题，需等待 Chubby 服务的恢复。如果成功则说明 Chubby 服务良好而子表服务器本身出现了问题。这种情况下主服务器会中止这个子表服务器并将其上的子表全部移至其他子表服务器。当在状态监测时发现某个子表服务器上负载过重时，主服务器会自动对其进行负载均衡操作。

基于系统出现故障是一种常态的设计理念（Google 几乎所有的产品都是基于这个设计理念），每个主服务器被设定了一个会话时间的限制。当某个主服务器到时退出后，管理系统就会指定一个新的主服务器，这个主服务器的启动需要经历以下四个步骤^[8]。

- （1）从 Chubby 中获取一个独占锁，确保同一时间只有一个主服务器。
- （2）扫描服务器目录，发现目前活跃的子表服务器。
- （3）与所有的活跃子表服务器取得联系以便了解所有子表的分配情况。
- （4）通过扫描元数据表（Metadata Table），发现未分配的子表并将其分配到合适的子表服务器。如果元数据表未分配，则首先需要将根子表（Root Tablet）加入未分配的子表中。由于根子表保存了其他所有元数据子表的信息，确保了扫描能够发现所有未分配的子表。

在成功完成以上四个步骤后主服务器就可以正常运行了。

2.4.5 子表服务器

Bigtable 中实际的数据都是以子表的形式保存在子表服务器上的，客户一般也只和子表服务器进行通信，所以子表以及子表服务器是我们重点讲解的概念。子表服务器上的操作主要涉及子表的定位、分配以及子表数据的最终存储问题。其中子表分配在前面已经有了详细介绍，这里略过不讲。在讲解其他问题之前我们首先介绍一下 SSTable 的概念以及子表的基本结构。

1. SSTable 及子表基本结构

SSTable 是 Google 为 Bigtable 设计的内部数据存储格式。所有的 SSTable 文件都存储在 GFS 上，用户可以通过键来查询相应的值，图 2-15 是 SSTable 格式的基本示意图。

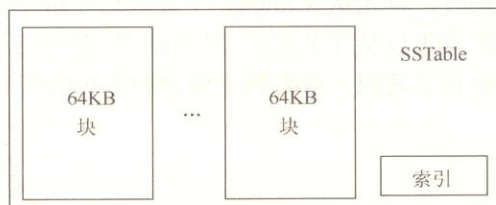


图 2-15 SSTable 格式的基本示意图

SSTable 中的数据被划分成一个个的块（Block），每个块的大小是可以设置的，一般来说设置为 64KB。在 SSTable 的结尾有一个索引（Index），这个索引保存了 SSTable 中块的位置信息，在 SSTable 打开时这个索引会被加载进内存，这样用户在查找某个块时首先在内存中查找块的位置信息，然后在硬盘上直接找到这个块，这种查找方法速度非常快。由于每个 SSTable 一般都不是很大，用户还可以选择将其整体加载进内存，这样查找起来会更快。

从概念上讲子表是表中一系列行的集合，它在系统中的实际组成如图 2-16 所示。

每个子表都是由多个 SSTable 以及日志（Log）文件构成。有一点需要注意，那就是不同子表的 SSTable 可以共享，也就是说某些 SSTable 会参与多个子表的构成，而由子表构成的表则不存在子表重叠的现象。Bigtable 中的日志文件是一种共享日志，也就是说系统并不是对子表服务器上每个子表都单独地建立一个日志文件，每个子表服务器上仅保存

一个日志文件，某个子表日志只是这个共享日志的一个片段。这样会节省大量的空间，但在恢复时却有一定的难度，因为不同的子表可能会被分配到不同的子表服务器上，一般情况下每个子表服务器都需要读取整个共享日志来获取其对应的子表日志。Google 为了避免这种情况出现，对日志做了一些改进。Bigtable 规定将日志的内容按照键值进行排序，这样不同的子表服务器都可以连续读取日志文件了。一般来说每个子表的大小在 100MB 到 200MB 之间。每个子表服务器上保存的子表数量可以从几十到上千不等，通常情况下是 100 个左右。

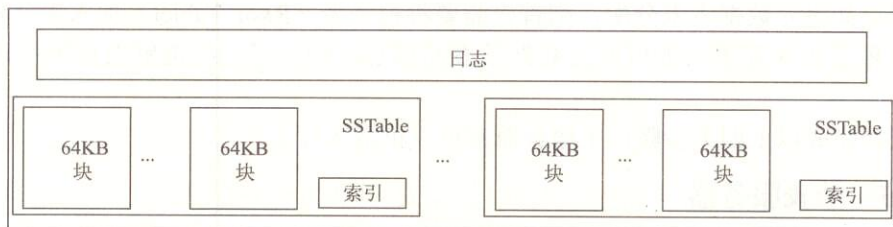


图 2-16 子表实际组成

2. 子表地址

子表地址的查询是经常碰到的操作。在 Bigtable 系统的内部采用的是一种类似 B+树的三层查询体系。子表地址结构如图 2-17 所示^[8]。

所有的子表地址都被记录在元数据表中，元数据表也是由一个个的元数据子表（Metadata tablet）组成的。根子表是元数据表中一个比较特殊的子表，它既是元数据表的第一条记录，也包含了其他元数据子表的地址，同时 Chubby 中的一个文件也存储了这个根子表的信息。这样在查询时，首先从 Chubby 中提取这个根子表的地址，进而读取所需的元数据子表的位置，最后就可以从元数据子表中找到待查询的子表。除了这些子表的元数据之外，元数据表中还保存了其他一些有利于调试和分析的信息，比如事件日志等。

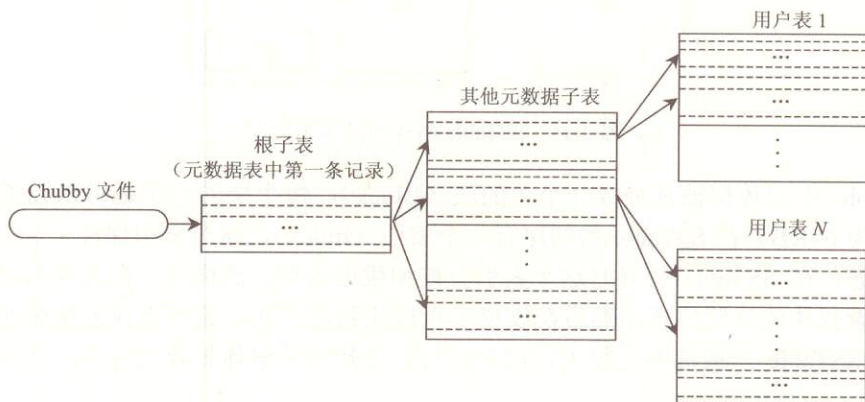


图 2-17 子表地址结构

为了减少访问开销，提高客户访问效率，Bigtable 使用了缓存（Cache）和预取（Prefetch）技术，这两种技术手段在体系结构设计中是很常用的。子表的地址信息被缓存在客户端，客户在寻址时直接根据缓存信息进行查找。一旦出现缓存为空或缓存信息过时

的情况，客户端就需要按照图 2-17 所示方式进行网络的来回通信（Network Round-trips）进行寻址，在缓存为空的情况下需要三个网络来回通信。如果缓存的信息是过时的，则需要六个网络来回通信。其中三个用来确定信息是过时的，另外三个获取新的地址。预取则是在每次访问元数据表时不仅仅读取所需的子表元数据，而是读取多个子表的元数据，这样下次需要时就不用再次访问元数据表。

3. 子表数据存储及读/写操作

在数据的存储方面 Bigtable 做出了一个非常重要的选择，那就是将数据存储划分成两块。较新的数据存储在内存中一个称为内存表（Memtable）的有序缓冲里，较早的数据则以 SSTable 格式保存在 GFS 中。这种技术在数据库中不是很常用，但 Google 还是做出了这种选择，实际运行的效果也证明 Google 的选择虽然大胆却是正确的。

从图 2-18^[8]中可以看出读和写操作有很大的差异性。做写操作（Write Op）时，首先查询 Chubby 中保存的访问控制列表确定用户具有相应的写权限，通过认证之后写入的数据首先被保存在提交日志（Commit Log）中。提交日志中以重做记录（Redo Record）的形式保存着最近的一系列数据更改，这些重做记录在子表进行恢复时可以向系统提供已完成的更改信息。数据成功提交之后就被写入内存表中。在做读操作（Read Op）时，首先还是要通过认证，之后读操作就要结合内存表和 SSTable 文件来进行，因为内存表和 SSTable 中都保存了数据。

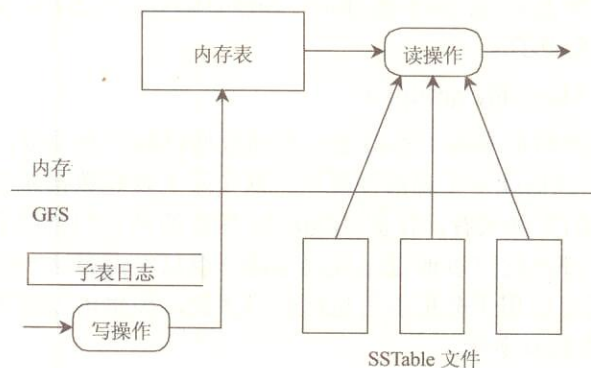


图 2-18 Bigtable 数据存储及读/写操作

在数据存储中还有一个重要问题，就是数据压缩的问题。内存表的空间毕竟是很有限的，当其容量达到一个阈值时，旧的内存表就会被停止使用并压缩成 SSTable 格式的文件。在 Bigtable 中有三种形式的数据压缩，分别是次压缩（Minor Compaction）、合并压缩（Merging Compaction）和主压缩（Major Compaction）。三者之间的关系如图 2-19 所示。

每一次旧的内存表停止使用时都会进行一个次压缩操作，这会产生一个 SSTable。但如果系统中只有这种压缩的话，SSTable 的数量就会无限制地增加下去。由于读操作要使用 SSTable，数量过多的 SSTable 显然会影响读的速度。而在 Bigtable 中，读操作实际上比写操作更重要，因此 Bigtable 会定期地执行一次合并压缩的操作，将一些已有的 SSTable 和现有的内存表一并进行一次压缩。主压缩其实是合并压缩的一种，只不过它将所有的 SSTable 一次性压缩成一个大的 SSTable 文件。主压缩也是定期执行的，执行一次主压缩之后可以保证将所有的被压缩数据彻底删除，如此一来，既回收了空间又能保证敏

感数据的安全性（因为这些敏感数据被彻底删除了）。

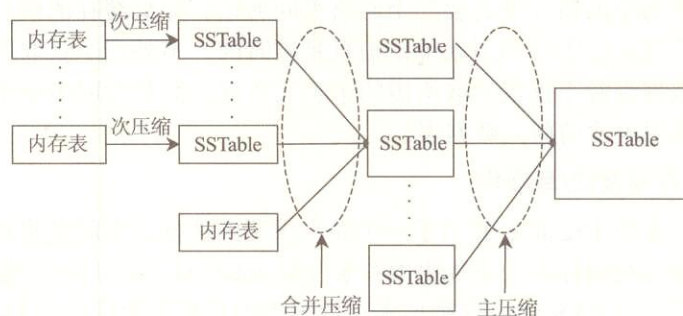


图 2-19 三种形式压缩之间的关系

2.4.6 性能优化

上述各种操作已经可以实现 Bigtable 的所有功能了，但是这些基本的功能很多时候并不是很符合用户的使用习惯，或者执行的效率较低。有些功能 Bigtable 自身已经进行了优化，包括使用缓存、共享式的提交日志以及利用系统的不变性。除此之外，Bigtable 还允许用户个人在基本操作基础上对系统进行一些优化。这一部分主要向读者介绍用户可以使用的几个重要优化措施。实际上这些技术手段都是一些已有的数据库方法，只不过 Google 将它具体地应用于 Bigtable 之中了。

1. 局部性群组（Locality groups）

Bigtable 允许用户将原本并不存储在一起的数据以列族为单位，根据需要组织在一个单独的 SSTable 中，以构成一个局部性群组。这实际上就是数据库中垂直分区技术的一个应用。结合图 2-13 的实例来看，在被 Bigtable 保存的网页列关键字中，有的用户可能只对网页内容感兴趣，那么它可以通过设置局部性群组只看内容这一列。有的则会对诸如网页语言、网站排名等可以用于分析的信息比较感兴趣，他也可以将这些列设置到一个群组中。局部性群组如图 2-20 所示。

通过设置局部性群组用户可以只看自己感兴趣的内容，对某个用户来说的大量无用信息无需读取。对于一些较小的且会被经常读取的局部性群组，用户可以直接将其 SSTable 文件直接加载进内存，这可以明显地改善读取效率。

2. 压缩

压缩可以有效地节省空间，Bigtable 中的压缩被应用于很多场合。首先压缩可以被用在构成局部性群组的 SSTable 中，可以选择是否对个人的局部性群组的 SSTable 进行压缩。Bigtable 中这种压缩是对每个局部性群组独立进行的，虽然这样会浪费一些空间，但是在需要读时解压速度非常快。通常情况下，用户可以采用两步压缩的方式^[8]：第一步利用 Bentley & McIlroy 方式（BMDiff）在大的扫描窗口将常见的长串进行压缩；第二步采取 Zippy 技术进行快速压缩，它在一个 16KB 大小的扫描窗口内寻找重复数据，这个过程非常快。压缩技术还可以提高子表的恢复速度，当某个子表服务器停止使用后，需要将上面所有的子表移至另一个子表服务器来恢复服务。在转移之前要进行两次压缩，第一次压缩减少了提交日志中的未压缩状态，从而减少了恢复时间。在文件正式转移之前还要进行

一次压缩，这次压缩主要是将第一次压缩后遗留的未压缩空间进行压缩。完成这两步之后压缩的文件就会被转移至另一个子表服务器。

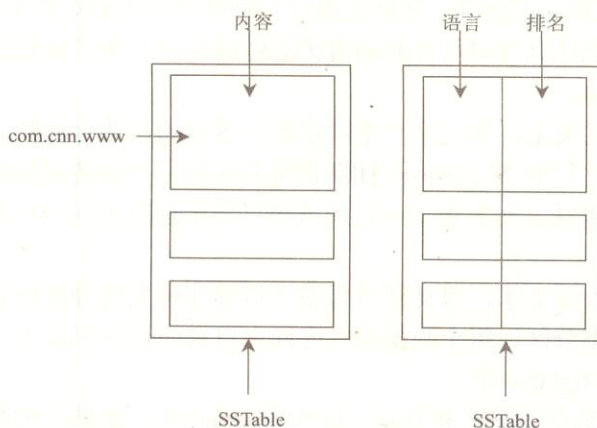


图 2-20 局部性群组

3. 布隆过滤器 (Bloom Filter)

Bigtable 向用户提供了一种称为布隆过滤器^[12]的数学工具。布隆过滤器是巴顿·布隆在 1970 年提出的，实际上它是一个很长的二进制向量和一系列随机映射函数，在读操作中确定子表的位置时非常有用。布隆过滤器的速度快，省空间。而且它有一个最大的好处是它绝不会将一个存在的子表判定为不存在。不过布隆过滤器也有一个缺点，那就是在某些情况下它会将不存在的子表判断为存在。不过这种情况出现的概率非常小，跟它带来的巨大好处相比这个缺点是可以忍受的。

目前包括 Google Analytics、Google Earth、个性化搜索、Orkut 和 RRS 阅读器在内的几十个项目都使用了 Bigtable。这些应用对 Bigtable 的要求以及使用的集群机器数量都是各不相同的，但是从实际运行来看，Bigtable 完全可以满足这些不同需求的应用，而这一切都得益于其优良的构架以及恰当的技术选择。与此同时 Google 还在不断地对 Bigtable 进行一系列的改进，通过技术改良和新特性的加入提高系统运行效率及稳定性。

2.5 分布式存储系统 Megastore

互联网的迅速发展带来了新的数据应用场景，和传统的数据存储有别的是，互联网上的应用对于数据的可用性和系统的扩展性具有很高的要求。一般的互联网应用都要求能够做到 7 天×24 小时的不间断服务，达不到的话则会带来较差的用户体验。热门的应用往往会在短时间内经历急剧的用户数量增长，这就要求系统具有良好的可扩展性。在互联网的应用中，为了达到好的可扩展性，常常会采用 NoSQL 存储方式。但是从应用程序的构建方面来看，传统的关系型数据库又有着 NoSQL 所不具备的优势。Google 设计和构建了用于互联网中交互式服务的分布式存储系统 Megastore，该系统成功的将关系型数据库和 NoSQL 的特点与优势进行了融合。本节将向大家介绍该系统，着重突出 Megastore 设计与构建过程中的核心思想和技术。

2.5.1 设计目标及方案选择

Megastore 的设计目标很明确，那就是设计一种介于传统的关系型数据库和 NoSQL 之间的存储技术，尽可能达到高可用性和高可扩展性的统一。为了达到这一目标，设计团队采用了如下的两种方法：

(1) 针对可用性的要求，实现了一个同步的、容错的、适合远距离传输的复制机制。在方案的选择和实现过程中 Megastore 团队研究和比较了一些传统的远距离复制技术，最终确定了引入 Paxos 算法并对其做出一定的改进以满足远距离同步复制的要求。具体的实现将在 2.5.5 节介绍。

(2) 针对可扩展性的要求，设计团队借鉴了数据库中数据分区的思想，将整个大的数据分割成很多小的数据分区，每个数据分区连同它自身的日志存放在 NoSQL 数据库中，具体来说就是存放在 Bigtable 中。

图 2-21^[17]显示了数据的分区和复制。在 Megastore 中，这些小的数据分区被称为实体组集 (Entity Groups)。每个实体组集包含若干的实体组 (Entity Group，相当于分区中表的概念)，而一个实体组中又包含很多的实体 (Entity，相当于表中记录的概念)。从图中还可以看出单个实体组支持 ACID 语义，以上这些都体现了关系型数据库的特征。

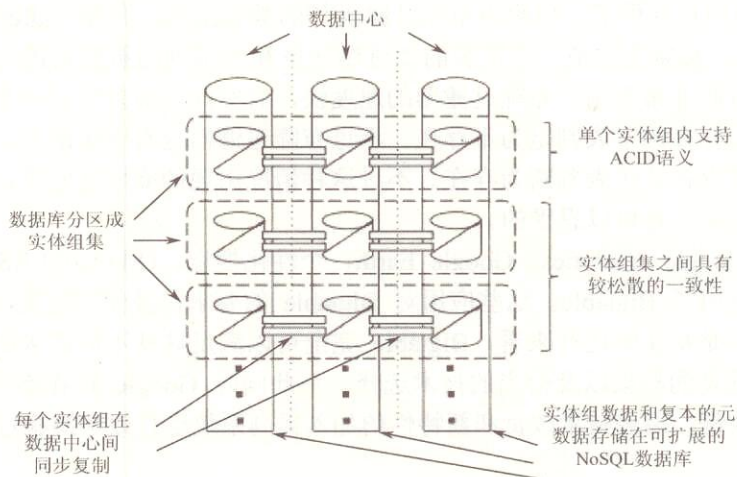


图 2-21 数据分区和复制

实体组集之间只具有比较松散的一致性。每个实体组都通过复制技术在数据中心中保存若干数据副本，这些实体组及其副本都存储在 NoSQL 数据库 (Bigtable) 中。

2.5.2 Megastore 数据模型

传统的关系型数据库是通过连接 (Join) 来满足用户的需求的，但是就 Megastore 而言，这种数据模型是不合适的，主要有以下三个原因。

(1) 对于高负载的交互式应用来说，可预期的性能提升要比使用一种代价高昂的查询语言所带来的好处多。

(2) Megastore 所面对的应用是读远多于写的，因此好的选择是将读操作所需要做的

工作尽可能地转移到写操作上。

(3) 在 Bigtable 这样的键/值存储系统中存储和查询级联数据 (Hierarchical Data) 是很方便的。

基于上述三点考虑, Google 团队设计了一种能够提供细粒度控制的数据模型和模式语言。Megastore 中关系型数据库的特征就集中体现在这种数据模型。同关系型数据库一样, Megastore 的数据模型是在模式 (schema) 中定义的且是强类型的 (strongly typed)。每个模式都由一系列的表 (tables) 构成, 表又包含有一系列的实体 (entities), 每个实体中又包含一系列的属性 (properties)。属性是命名的且具有类型, 这些类型包括字符型 (strings)、数字类型 (numbers) 或者 Google 的 Protocol Buffers。这些属性可以被设置成必须的 (required)、可选的 (optional) 或者可重复的 (repeated, 即允许单个属性上有多个值)。图 2-22^[17]是 Megastore 中一个照片共享服务的数据模型实例。

```
CREATE SCHEMA PhotoApp;

CREATE TABLE User {
  required int64 user_id;
  required string name;
} PRIMARY KEY(user_id), ENTITY GROUP ROOT;

CREATE TABLE Photo {
  required int64 user_id;
  required int32 photo_id;
  required int64 time;
  required string full_url;
  optional string thumbnail_url;
  repeated string tag;
} PRIMARY KEY(user_id, photo_id),
  IN TABLE User,
  ENTITY GROUP KEY(user_id) REFERENCES User;

CREATE LOCAL INDEX PhotosByTime
  ON Photo(user_id, time);

CREATE GLOBAL INDEX PhotosByTag
  ON Photo(tag) STORING (thumbnail_url);
```

图 2-22 照片共享服务的数据模型实例

从图中可以很容易地发现, 这种模式定义的方式和关系型数据库中的定义方法非常的类似。在 Megastore 中, 所有的表要么是实体组根表 (Entity Group Root Table), 要么是子表 (Child Table)。所有的子表必须有一个参照根表的外键, 这个外键是通过 ENTITY GROUP KEY 来声明的。图 2-22^[17]中表 Photo 就是一个子表, 因为它声明了一个外键, User 则是一个根表。一个 Megastore 实例中可以有若干个不同的根表, 表示不同类型的实体组集。

图 2-22 中的实例还可以看到三种不同的属性设置, 既有必须的 (如 user_id), 也有可选的 (如 thumbnail_url)。值得注意的是 Photo 中的可重复类型的 tag 属性, 这也就意味着一个 Photo 中允许同时出现多个 tag 属性。

Megastore 数据模型中另一个非常重要的概念——索引（Index）也在图 2-22 中得到体现。Megastore 将索引分成了两大类：局部索引（Local Index）和全局索引（Global Index）。局部索引定义在单个实体组中，它的作用域仅限于单个实体组。全局索引则可以横跨多个实体组集进行数据读取操作。图 2-22 中 PhotosByTime 就是一个局部索引，而 PhotosByTag 则是一个全局索引。除了这两大类的索引外，Megastore 还提供了一些额外的索引特性，主要包括以下几个。

（1）STORING 子句（STORING Clause）：通过在索引中增加 STORING 子句，应用程序可以存储一些额外的属性，这样在读取数据时可以更快地从基本表中得到所需内容。PhotosByTag 这样一个索引中就对 thumbnail_url 使用了 STORING 子句。

（2）可重复的索引（Repeated Indexes）：Megastore 提供了对可重复属性建立索引的能力，这种可重复的索引对于子表来说常常是很有效的。

（3）内联索引（Inline Indexes）：任何一个有外键的表都能够创建一个内联索引。内联索引能够有效的从子实体中提取出信息片段并将这些片段存储在父实体中，以此加快读取速度。

最后简单地了解在这种数据模型下数据是如何存储在 Bigtable 中的。Megastore 中的实体组都存储在 Bigtable 中，表 2-2^[17]列出了上面照片共享服务实例的数据在 Bigtable 中的存储情况。

表 2-2 Bigtable 中数据存储情况

行键（Row Key）	User.name	Photo.time	Photo.tag	Photo._url
101	John			
101,500		12:30:01	Dinner, Paris	...
101,502		12:15:22	Betty, Paris	...
102	Mary			

从表中不难看出，Bigtable 的列名实际上是表名和属性名结合在一起得到的。不同表中的实体可以存储在同一个 Bigtable 行中。

2.5.3 Megastore 中的事务及并发控制

每个实体组实际上就像一个小的数据库，在实体组内部提供了完整的序列化 ACID 语义（Serializable ACID Semantics）支持。

Megastore 提供了三种方式的读，分别是 current、snapshot 和 inconsistent。其中 current 读和 snapshot 读总是在单个实体组中完成的。在开始某次 current 读之前，需要确保所有已提交的写操作已经全部生效，然后应用程序再从最后一个成功提交的事务时间戳位置读取数据。对于 snapshot 读，系统取出已知的最后一个完整提交的事务的时间戳，接着从这个位置读数据。和 current 读不同的是，snapshot 读的时候可能还有部分事务提交了但未生效。inconsistent 读忽略日志的状态直接读取最新的值。这对于那些要求低延迟并能容忍数据过期或不完整的读操作是非常有用的。

Megastore 事务中的写操作采用了预写式日志（Write-ahead Log），也就是说只有当所有的操作都在日志中记录下后写操作才会对数据执行修改。一个写事务总是开始于一个

current 读以便确认下一个可用的日志位置。提交操作将数据变更聚集到日志，接着分配一个比之前任意一个都高的时间戳，然后使用 Paxos 将数据变更加入到日志中。这个协议使用了乐观并发 (Optimistic Concurrency)：尽管可能有多个写操作同时试图写同一个日志位置，但只会有 1 个成功。所有失败的写都会观察到成功的写操作，然后中止并重试它们的操作。

一个完整的事务周期要经过如下几个阶段。

- (1) 读：获取最后一次提交的事务的时间戳和日志位置。
- (2) 应用逻辑：从 Bigtable 读取并且聚集数据到日志入口。
- (3) 提交：使用 Paxos 达到一致，将这个入口追加到日志。
- (4) 生效：将数据更新到 Bigtable 中的实体和索引。
- (5) 清除：清理不再需要的数据。

Megastore 中事务间的消息传递是通过队列 (Queue) 实现的，图 2-23^[17]显示了这一过程。

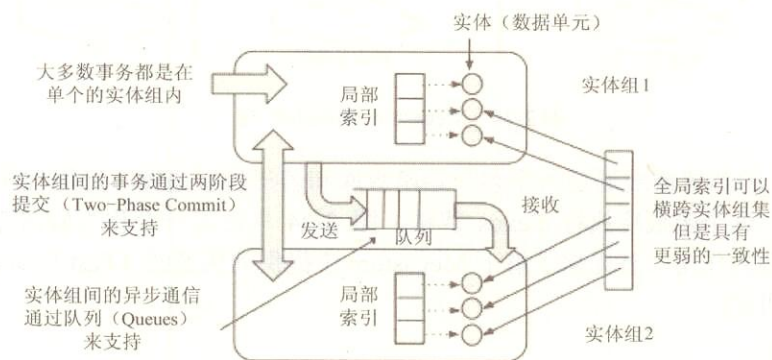


图 2-23 Megastore 中的事务机制

Megastore 中的消息能够横跨实体组，在一个事务中分批执行多个更新或者延缓作业 (Defer Work)。在单个实体组上执行的事务除了更新它自己的实体外，还能够发送或收到多个信息。每个消息都有一个发送和接收的实体组；如果这两个实体组是不同的，那么传输将会是异步的。虽然这种消息队列机制在关系型数据库中已经有了很长的应用历史，Megastore 实现的这种消息机制的最大特点在于其规模：声明一个队列后可以在其他所有的实体组上创建一个收件箱。

除了队列机制之外，Megastore 还支持两阶段提交 (Two-phase Commit)。但是这会产生比较高的延迟并且增加了竞争的风险，一般情况下不鼓励使用。

2.5.4 Megastore 基本架构

图 2-24^[17]是 Megastore 的基本架构，最底层的数据是存储在 Bigtable 中的。不同类型的副本存储不同的数据。在 Megastore 中共有三种副本，分别是完整副本 (Full Replica)、见证者副本 (Witness Replica) 和只读副本 (Read-only Replica)。图 2-24 中出现了两种副本，分别是完整副本 A 和 B，以及见证者副本 C。对于完整副本，Bigtable 中存储完整的日志和数据。见证者副本的作用是在 Paxos 算法执行过程中无法产生一个决议时参与投

票，因此对于这种副本，**Bigtable** 只存储其日志而不存储具体数据。最后一种只读副本和见证者副本恰恰相反，它们无法参与投票。它们的作用只是读取到最近过去某一个时间点的一致性数据。如果读操作能够容忍这些过期数据，只读副本能够在不加刷写延迟的情况下将数据在较大的地理空间上进行传输。

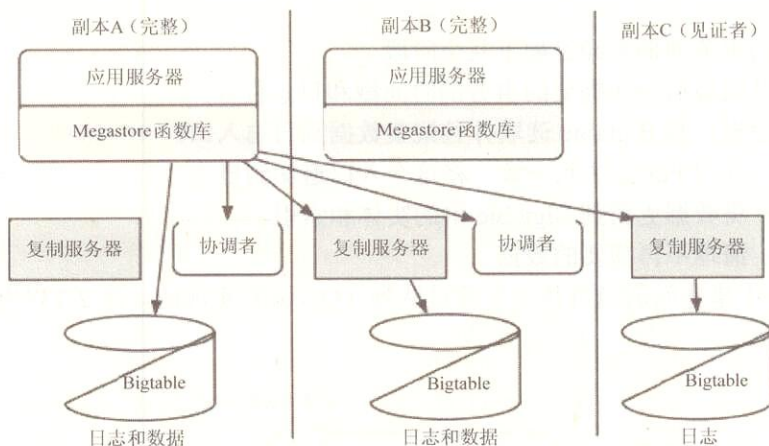


图 2-24 Megastore 的基本架构

Megastore 的部署需要通过一个客户端函数库和若干的服务器。应用程序连接到这个客户端函数库，这个函数库执行 Paxos 算法。图 2-24 中还有一个称为协调者的服务，要想理解这个服务的作用，首先来了解下 Megastore 中提供的快速读（Fast Reads）和快速写（Fast Writes）机制。

1. 快速读

如果读操作不需要副本之间进行通信即可完成，那么读取的效率必然相对较高。由于写操作基本上能在所有的副本上成功，一旦成功认为该副本上的数据都是相同的且是最新的，就能利用本地读取（Local Reads）实现快速读，能够带来更好的用户体验及更低的延迟。确保快速读成功的关键是保证选择的副本上数据是最新的。为了达到这一目标，设计团队引入了协调者的概念。协调者是一个服务，该服务分布在每个副本的数据中心里面。它的主要作用就是跟踪一个实体组集合，集合中的实体组需要具备的条件就是它们的副本已经观察到了所有的 Paxos 写。只要出现在这个集合中的实体组，它们的副本就都能够进行本地读取，也就是说能够实现快速读。协调者的状态是由写算法来保证，关于这点将在 2.5.5 中再次介绍。

2. 快速写

为了达到快速的单次交互的写操作，Megastore 采用了一种在主/从式系统中常用的优化方法。如果一次写成功，那么下一次写的时候就跳过准备过程，直接进入接受阶段。因为一次成功的写意味着也准确地获知了下一个日志的位置，所以不再需要准备阶段。Megastore 没有使用专门的主服务器，而是使用 leaders。系统在每一个日志位置都运行一个 Paxos 算法实例。leader 主要是来裁决哪个写入的值可以获取 0 号提议。第一个将值提交给 leader 的可以获得一个向所有副本请求接收这个值作为 0 号提议最终值的机会。其他

的值就需要重新使用 Paxos 算法。

由于写入者在提交值给其他副本之前必须要和 leader 通信，为了尽可能地减少延迟，Megastore 做了一个简单的优化，即在提交值最多的位置附近选择一个副本作为 leader。

客户端、网络及 Bigtable 的故障都会导致一个写操作处于不确定的状态。图 2-24 中的复制服务器会定期扫描未完成的写入并且通过 Paxos 算法提议没有操作的值（No-op Values）来让写入完成。

2.5.5 核心技术——复制

复制可以说是 Megastore 最核心的技术，如何实现一个高效、实时的复制方案对于整个系统的性能起着决定性的作用。通过复制保证所有最新的数据都保存有一定数量副本，能够很好地提高系统的可用性。

1. 复制的日志

每个副本都存有记录所有更新的数据。即使是它正从一个之前的故障中恢复数据，副本也要保证其能够参与到写操作中的 Paxos 算法，因此 Megastore 允许副本不按顺序接受日志，这些日志将独立的存储在 Bigtable 中。图 2-25^[17]是 Megastore 中预写式日志的一个典型应用场景。

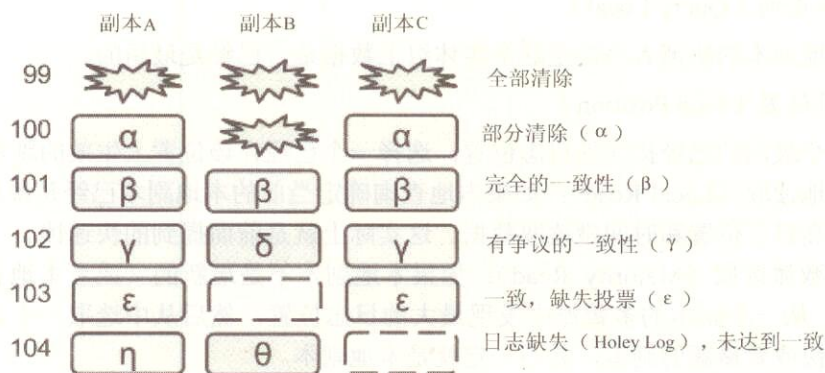


图 2-25 预写式日志

当日志有不完整的前缀时我们就称一个日志副本有“缺失”（Holes）。在图 2-25 中 0~99 的日志位置已经被全部清除，100 的日志位置被部分清除，因为每个副本都会被通知到其他副本已经不再需要这个日志。101 的日志位置被全部副本接受。102 的日志位置被 γ 获得，这是一种有争议的一致性。103 的日志位置被副本 A 和 C 接受，副本 B 则留下了一个“缺失”。104 的日志位置则未达到一致性，因为副本 A 和副本 B 存在争议。

2. 数据读取

在一次 Current 读之前，要保证至少有一个副本上的数据是最新的，也就是说所有之前提交到日志中的更新必须复制到该副本上并确保在该副本上生效。这个过程称之为追赶（Catchup）。

图 2-26^[17]是一次数据读取过程，总的来看，该过程要经过如下几个步骤。

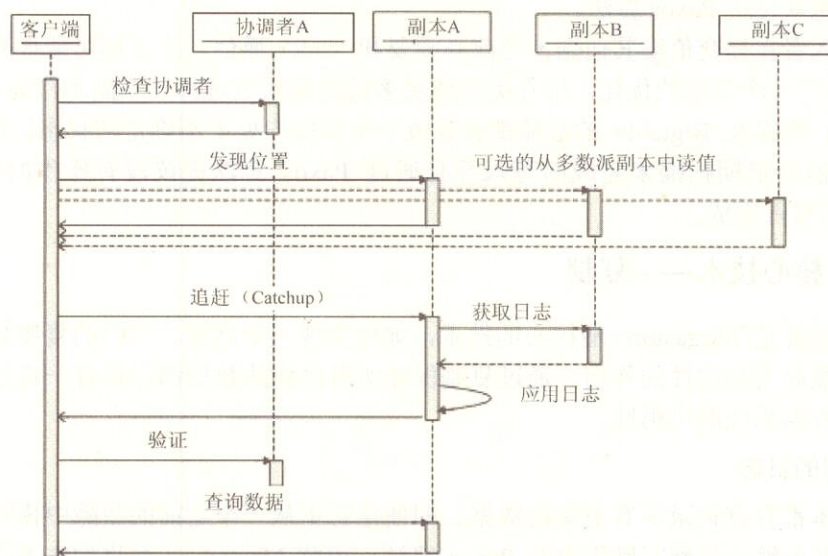


图 2-26 数据读取

1) 本地查询 (Query Local)

查询本地副本的协调者来决定这个实体组上数据是否已经是最新的。

2) 发现位置 (Find Position)

确定一个最高的已经提交的日志位置，选择一个已经在该位置上生效的副本。

(1) 本地读取 (Local Read): 如果本地查询确定当前的本地副本已经是最新的，则从副本中的最高日志位置和时间戳读取数据。这实际上就是前面提到的快速读。

(2) 多数派读取 (Majority Read): 如果本地副本不是最新的 (或者本地查询或本地读取超时)，从一个副本的多数派中发现最大的日志位置，然后从中选取一个读取。选择一个响应最快或者最新的副本，并不一定就是本地副本。

3) 追赶

一旦某个副本被选中，就采取如下方式使其追赶至已知的最大日志位置处。

(1) 对于所选副本中所有不知道共识值 (Consensus Value) 的日志位置，从其他的副本中读取值。对于任意的没有任何可用的已提交的值的日志位置，将会利用 Paxos 算法发起一次无操作的写。Paxos 将会促使绝大多数副本达成一个共识值——可能是无操作的写也可能是以前的一次写操作。

(2) 接下来就所有未生效的日志位置生效成上面达成的共识值，以此来达到一种分布式一致状态。

4) 验证 (Validate)

如果本地副本被选中切数据不是最新，发送一个验证消息到协调者断定 (entity group, replica) 对 ((entity group, replica) pair) 能够反馈所有提交的写操作。无需等待回应，如果请求失败，下一个读操作会重试。

5) 查询数据 (Query Data)

在所选的副本中利用日志位置的时间戳读取数据。如果所选的副本不可用了, 重新选中一个替代副本, 执行追赶操作, 然后从中读取数据。单个的较大查询结果可能是从多个副本中汇聚而来。

需要指出的是, 本地查询和本地读取是并行执行的。

3. 数据写入

执行完一次完整的读操作之后, 下一个可用的日志位置、最后一次写操作的时间戳, 以及下一次的 leader 副本都知道了。在提交时刻所有的更新都被打包 (Packaged) 和提议 (Proposed), 同时还包含一个时间戳、下一次 leader 提名及下一个日志位置的共识值。如果该值赢得了分布式共识, 它将应用到所有的副本中。否则整个事务将中止且从读操作重新开始。

在 2.5.4 节中介绍快速读时曾经提到协调者的状态是由写算法来保证的。这实际上描述了这样的一个过程: 如果一次写操作不是被所有的副本所接受, 必须要将这些未接受写操作的副本中相关的实体组从协调者中移去, 这个过程称为失效 (Invalidation)。失效的过程可以保证协调者所看到的副本上数据都是接受了写操作的最新数据。在一次写操作被提交并准备生效之前, 所有的副本必须选择接受或者在协调者中将有关的实体组进行失效。

图 2-27^[17]是数据写入的完整过程, 具体包括如下几个步骤。

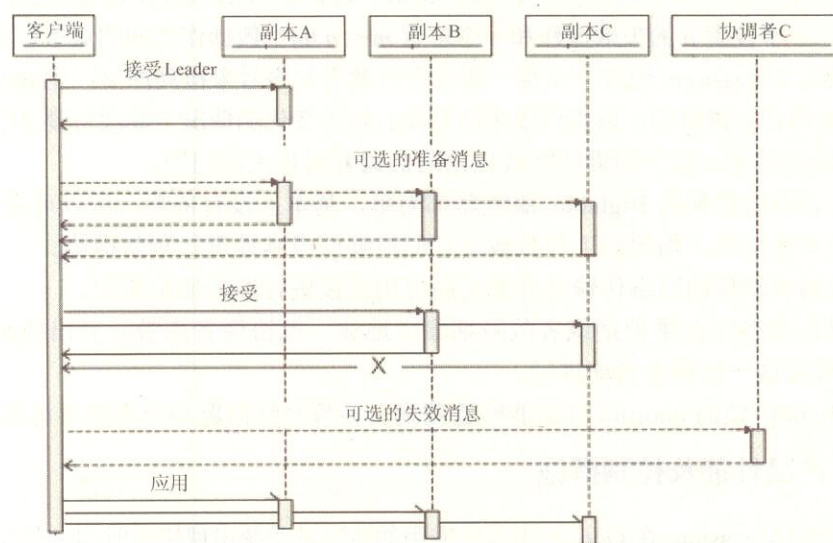


图 2-27 数据写入

(1) 接受 leader: 请求 leader 接受值作为 0 号提议。这实际上就是前面介绍的快速写方法。如果成功, 跳至步骤 (3)。

(2) 准备: 在所有的副本上使用一个比其当前所见的日志位置更高的提议号进行 Paxos 准备阶段。将值替换成拥有最高提议号的那个值。

(3) 接受: 请求剩余的副本接受该值, 如果大多数副本拒绝这个值, 返回步骤 (2)。

(4) 失效: 将不接受值的副本上的协调者进行失效操作。

(5) 生效: 将值的更新在尽可能多的副本上生效。如果选择的值和原来提议的有冲突, 返回一个冲突错误。

4. 协调者的可用性

从上面的介绍中可以发现协调者在系统中是比较重要的, 协调者的进程运行在每个数据中心。每次的写操作中都要涉及协调者, 因此协调者的故障将会导致系统的不可用。虽然在实践中由协调者导致的系统不可用的情况很少出现, 但是网络和主机故障还是有可能导致协调者出现暂时的不可用。

Megastore 使用了 Chubby 锁服务, 协调者在启动的时候从数据中心获取指定的 Chubby 锁。为了处理请求, 一个协调者必须持有其多数锁。一旦因为出现问题导致它丢失了大部分锁, 协调者就会恢复到一个默认保守状态——认为所有它所能看见的实体组都是失效的。

写入者通过测试一个协调者是否丢失了锁从而让其在协调者不可用的过程中得到保护。写入者知道在恢复之前协调者会认为自己是失效的。当一个协调者突然不可用时, 这个算法需要面对一个短暂 (几十秒) 的写停顿风险——所有的写入者必须等待协调者的 Chubby 锁过期。

除了可用性问题, 对于协调者的读写协议必须满足一系列的竞争条件。失效的信息总是安全的, 但是生效的信息必须谨慎处理。在协调者中较早的写操作生效和较晚的写操作失效之间的竞争通过带有日志位置而被保护起来。较高位置的失效操作总是胜过较低位置的生效操作。一个位置 n 的失效操作和一个位置 $m < n$ 的生效操作之间的竞争常常和一个冲突联系在一起。Megastore 通过一个唯一的代表协调者的序号来检测冲突: 生效操作只允许在最近一次对协调者进行的读取操作以来序号没有发生变化的情况下修改协调者的状态。

在实际的应用中, 以下因素能够减轻使用协调者所带来的问题。

- (1) 协调者比任何的 Bigtable 服务器都简单, 基本上没有依赖, 所以可用性更高。
- (2) 协调者简单、均匀的工作负载让它们能够低成本地进行预防措施。
- (3) 协调者轻量的网络传输允许使用高可用连接进行服务质量监控。
- (4) 操作者能够在维护期或者故障期集中地让一批协调者失效。当出现某些系统默认的监控信号时这一过程会自动进行。
- (5) Chubby 锁的 quorum 机制能够监测到大多数网络问题和节点的不可用。

2.5.6 产品性能及控制措施

本节将介绍 Megastore 在 Google 中实际应用的情况及系统出现错误时的一些控制措施。

Megastore 在 Google 中已经部署和使用了若干年, 有超过 100 个产品使用 Megastore 作为其存储系统。图 2-28^[17]显示了这些产品可用性的分布情况, 从图中可以看出, 绝大多数产品具有极高的可用性 (>99.999%)。这表明 Megastore 系统的设计是非常成功的, 基本达到了预期目标。

图 2-29^[17]是产品延迟情况的分布, 根据数据中心的距离和写入数据的大小, 应用程序的平均读取延迟在万分之一毫秒之内, 平均写入延迟在 100 至 400 毫秒之间。

当某个完整副本忽然变的不可用或失去连接时, 为了避免 Megastore 的性能下降, 可采取以下三种应对方法。

(1) 通过重新选择路由使客户端绕开出现问题的副本，这是最重要的一种错误处理机制。

(2) 将出现问题副本上的协调者禁用，确保问题的影响降至最小。

(3) 禁用整个副本，这是最严厉的一种手段，但是这种方法比较少使用。

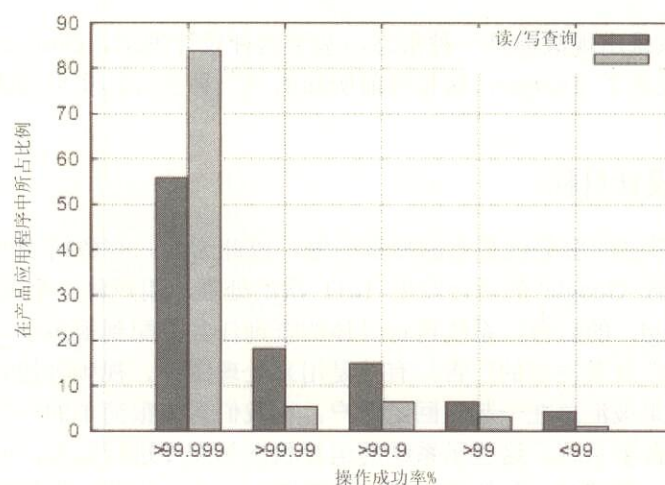


图 2-28 可用性分布情况

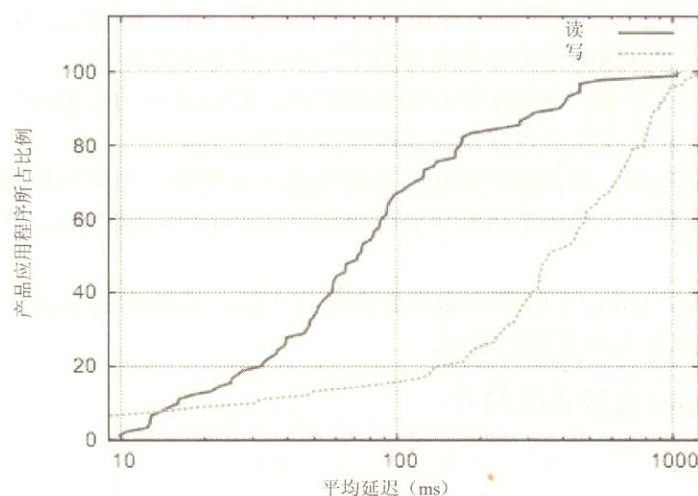


图 2-29 平均延迟的分布

一般来说，出现错误之后，上述三种方法可能会结合起来使用，而不是仅仅使用某种方法。

本节主要介绍了 Megastore 的设计思想以及核心的技术手段，其中很多内容对于设计 NoSQL 存储系统是有借鉴意义的。但需要跟读者指出的是：Megastore 已经是 Google 相对过时的存储技术。Google 目前正在使用的存储系统是 Spanner 架构，Spanner 的设计目标是能够控制一百万到一千万台服务器，Spanner 最强大之处在于能够在 50 毫秒之内为数

据传递提供通道——即使这两个数据中心分布于地球的两端。相信在不久之后 Google 也会公开 Spanner 的实现论文。

2.6 大规模分布式系统的监控基础架构 Dapper

Google 认为系统出现故障是一种常态, 基于这种设计理念, Google 的工程师们结合 Google 的实际开发出了 Dapper。这是目前所知的第一种公开其实现的大规模分布式系统的监控基础架构。

2.6.1 基本设计目标

Google 使用最多的服务就是它的搜索引擎, 以此为例, 有资料表明, 用户的平均每一次前台搜索会导致 Google 的后台发生 1011 次的处理。用户将一个关键字通过 Google 的输入框传到 Google 的后台, 系统再将具体的查询任务分配到很多子系统中, 这些子系统有些是用来处理涉及关键字的广告, 有些是用来处理图像、视频等搜索的, 最后所有这些子系统的搜索结果被汇总在一起返回给用户。在我们看来很简单的一次搜索实际上涉及了众多 Google 后台子系统, 这些子系统的运行状态都需要进行监控, 而且随着时间的推移 Google 的服务越来越多, 新的子系统也在不断被加入, 因此在设计时需要考虑到的第一个问题就是设计出的监控系统应当能够对尽可能多的 Google 服务进行监控, 即广泛可部署性 (Ubiquitous Deployment)。另一方面, Google 的服务是全天候的, 如果不能对 Google 的后台同样进行全天候的监控很可能会错过某些无法再现的关键性故障, 因此需要进行不间断的监控。这两个基本要求导致了如下三个基本设计目标。

(1) 低开销: 这个是广泛可部署性的必然要求。监控系统的开销越低, 对于原系统的影响就越小, 系统的开发人员也就越愿意接受这个监控系统。

(2) 对应用层透明: 监控系统对程序员应当是不可见的。如果监控系统的使用需要程序开发人员对其底层的一些细节进行调整才能正常工作的话, 这个监控系统肯定不是一个完善的监控系统。

(3) 可扩展性: Google 的服务增长速度是惊人的, 设计出的系统至少在未来几年里要能够满足 Google 服务和集群的需求。

2.6.2 Dapper 监控系统简介

1. 基本概念

对系统行为进行监控的过程非常的复杂, 特别是在分布式系统中。为了理解这种复杂性, 首先来看如图 2-30^[18]所示的一个过程。

在图中, 用户发出一个请求 X, 它期待得到系统对它做出的应答 X。但是接收到该请求的前端 A 发现该请求的处理需要涉及服务器 B 和服务器 C, 因此 A 又向 B 和 C 发出两个 RPC (远程过程调用)。B 收到后立刻做出响应, 但是 C 在接到后发现它还需要调用服务器 D 和 E 才能完成请求 X, 因此 C 对 D 和 E 分别发出了 RPC, D 和 E 接到后分别做出了应答, 收到 D 和 E 的应答之后 C 才向 A 做出响应, 在接收到 B 和 C 的应答之后 A 才对用户请求 X 做出一个应答 X。在监控系统中记录下所有这些消息不难, 如何将这些消

息记录同特定的请求（本例中的 X）关联起来才是分布式监控系统设计中需要解决的关键性问题之一。

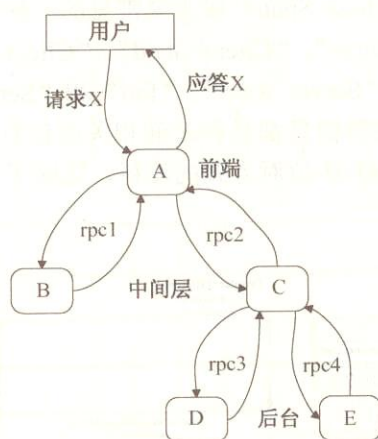


图 2-30 典型分布式系统的请求及应答过程

一般来说，有两种方案可供选择：黑盒（Black Box）方案及基于注释的监控（Annotation-based Monitoring）方案。二者比较而言，黑盒方案比较轻便，但是在消息关系判断的过程中，黑盒方案主要是利用一些统计学的知识来进行推断，有时不是很准确。基于注释的方案利用应用程序或中间件给每条记录赋予一个全局性的标示符，借此将相关消息串联起来。考虑到实际的需求，Google 的工程师最终选择了基于注释的方案，为了尽可能消除监控系统的应用程序对被监控系统的性能产生的不良影响，Google 的工程师设计并实现了一套轻量级的核心功能库，这将在后面进行介绍。

Dapper 监控系统中有三个基本概念：监控树（Trace Tree）、区间（Span）和注释（Annotation）。如图 2-31^[18]所示是一个典型的监控树，从中可以看到所谓的监控树实际上就是一个同特定事件相关的所有消息，只不过这些消息是按照一定的规律以树的形式组织起来。树中的每一个节点称为一个区间，区间实际上就是一条记录，所有这些记录联系在一起就构成了对某个事件的完整监控。从图 2-31 不难看出，每个区间包括如下的内容：区间名（Span Name）、区间 id（Span id）、父 id（Parent id）和监控 id（Trace id）。区间名主要是为了方便人们记忆和理解，因此要求这个区间名是人们可以读懂的。区间 id 是为了在一棵监控树中区分不同的区间。父 id 是区间中非常重要的一个内容，正是通过父 id 才能够对树中不同区间的关系进行重建，没有父 id 的区间称为根区间（Root Span）。图 2-31 中的 Frontend Request 就是一个根区间。在图中还能看出，区间的长度实际上包括了区间的开始及结束时间信息。

监控 id 在图 2-31 中并没有列出，一棵监控树中所有区间的监控 id 是相同的，这个监控 id 是随机分配的，且在整个 Dapper 监控系统中是唯一的。正如区间 id 是用来在某个监控树中区分不同的区间一样，监控 id 是用来在整个 Dapper 监控系统中区分不同的监控。注释主要用来辅助推断区间关系，也可以包含一些自定义的内容。图 2-32^[18]展示了图 2-31 中 Helper.Call 区间的更详细信息。

在图 2-32 中可以清楚地看到这个区间的区间名是“Helper.Call”，监控 id 是 100，区

间 id 是 5，父 id 是 3。一个区间既可以只有一台主机的信息，也可以包含来源于多个主机的信息；事实上，每个 RPC 区间都包含来自客户端（Client）和服务端（Server）的注释，这使得双主机区间（Two-host Span）成为最常见的一种。图 2-32 中的区间就包含了来自客户端的注释信息：“<Start>”、“Client Send”、“Client Recv”和“<End>”，也包含了来自服务器端的注释信息：“Server Recv”、“foo”和“Server Send”。除了“foo”是用户自定义的注释外，其他的注释信息都是和时间相关的信息。Dapper 不但支持用户进行简单的文本方式的注释，还支持键-值对方式的注释，这赋予了开发者更多的自由。

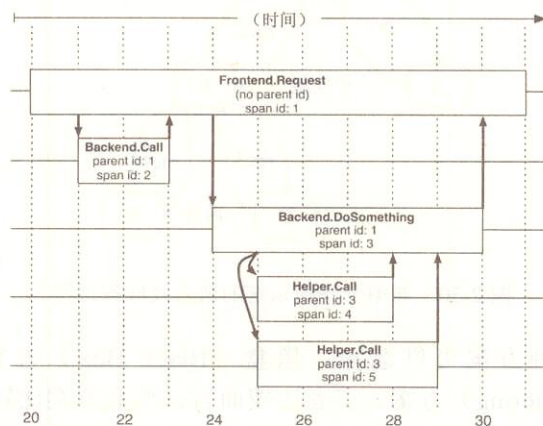


图 2-31 监控树

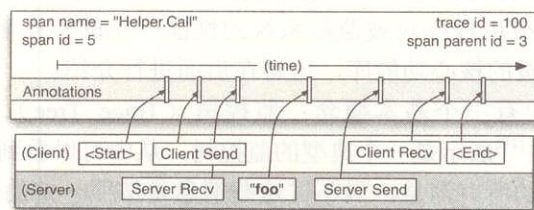


图 2-32 区间 Helper.Call 的详细信息

2. 监控信息的汇总

Dapper 对几乎所有的 Google 后台服务器进行监控。海量的消息记录必须通过一定的方式汇集在一起才能产生有效的监控信息。在实际中，Dapper 监控信息的汇总需要经过三个步骤，如图 2-33^[18]所示。

- (1) 将区间的数据被写入到本地的日志文件。
- (2) 利用 Dapper 守护进程（Dapper daemon）和 Dapper 收集器（Dapper Collectors）将所有机器上的本地日志文件汇集在一起。
- (3) 将汇集后的数据写入到 Bigtable 存储库中。

从图中也很容易地看出，（1）和（2）是一个读的过程，而（3）是一个写的过程。选择 Bigtable 主要是因为区间的数目非常多，而且各个区间的长度变化很大，Bigtable 对于这种很松散的表结构能够很好地进行支持。写入数据后的 Bigtable 中，单独的一行表示一个记录，而一列则相当于一个区间。这些监控数据的汇总是单独进行的，而不是伴随系统

对用户的应答一起返回的。如此选择主要有如下的两个原因：首先，一个内置的汇总方案（监控数据随 RPC 应答头返回）会影响网络动态。一般来说，RPC 应答数据规模比较小，通常不超过 10kb。而区间数据往往非常的庞大，如果将二者放在一起传输，会使这些 RPC 应答数据相对“矮化”进而影响后期的分析。另一方面，内置的汇总方案需要保证所有的 RPC 都是完全嵌套的，但有许多的中间件系统在其所有的后台返回最终结果之前就对调用者返回结果，这样有些监控信息就无法被收集。基于这两个考虑，Google 选择将监控数据和应答信息分开传输。

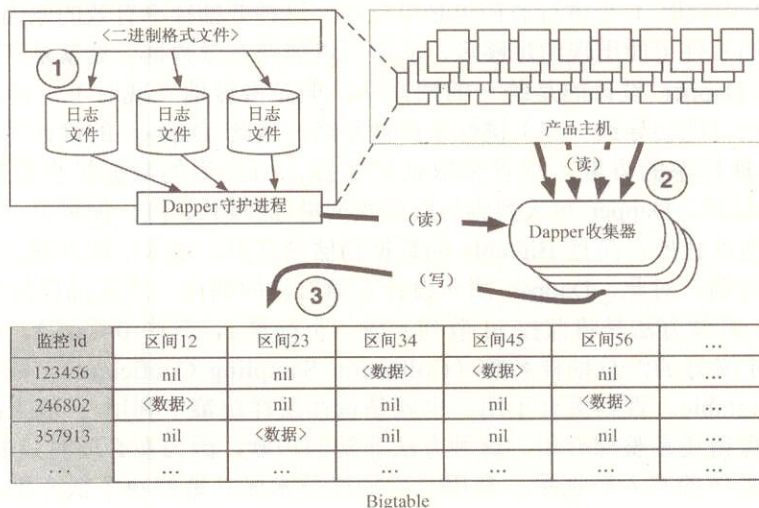


图 2-33 监控信息的汇总

安全问题是所有系统都必须考虑的问题，为了防止未授权用户对于 RPC 信息的访问，信息汇总过程中 Dapper 只存储 RPC 方法的名称却不存储任何 RPC 负载数据，取而代之的是，应用层注释提供了一种方便的选择机制（Opt-in Mechanism）：应用程序开发者可以将任何对后期分析有益的数据和区间关联起来。

2.6.3 关键性技术

前面提到了 Dapper 的三个基本设计目标，在这三个目标中，实现难度最大的是对应用层透明。为了达到既定设计目标，Google 不断进行创新，最终采用了一些关键性技术解决了存在的问题。这些关键性技术概括起来主要包括以下两个方面。

1. 轻量级的核心功能库

这主要是为了实现对应用层透明，设计人员通过将 Dapper 的核心监控实现限制在一个由通用线程（Ubiquitous Threading）、控制流（Control Flow）和 RPC 代码库（RPC Library Code）组成的小规模库基础上实现了这个目标。其中最关键的代码基础是基本 RPC、线程和控制流函数库的实现，主要功能是实现区间创建、抽样和在本地磁盘上记录日志。用 C++ 的话 Dapper 核心功能的实现不超过 1000 行代码，而用 Java 则不到 800 行。键/值对方式注释功能的实现需要额外增加 500 行代码。将复杂的功能实现限制在一个轻量级的核心功能库中保证了 Dapper 的监控过程基本对应用层透明。

2. 二次抽样技术

监控开销的大小直接决定 Dapper 的成败,为了尽可能地减小开销,进而将 Dapper 广泛部署在 Google 中,设计人员设计了一种非常巧妙的二次抽样方案。二次抽样顾名思义包括两次抽样过程。Google 每天需要处理的请求量惊人,如果对所有的请求都进行监控的话所产生的监控数据将会十分的庞大,也不利于数据分析,因此 Dapper 对这些请求进行了抽样,只有被抽中的请求才会被监控。在实践中,Dapper 的设计人员发现了一个非常有意思的现象,那就是当抽样率低至 $1/1024$ 时也能够产生足够多的有效监控数据,即在 1024 个请求中抽取 1 个进行监控也是可行的,这种低抽样率有效的原因在于巨大的事件数量使关注的事件可能出现的足够多,从而可以捕获有效数据。这就是 Dapper 的第一次抽样。最初 Dapper 设计的是统一的抽样率,但是慢慢地发现对于一些流量较低的服务,低抽样率很可能会导致一些关键性事件被忽略,因此 Dapper 的设计团队正在设计一种具有适应性抽样率的方案。尽管采取抽样监控,所产生的数据量也是惊人的。根据 Dapper 团队的统计,Dapper 每天得到的监控数据量已经超过 1T,如果将这些数据全部写入 Bigtable 中效率较低,而且 Bigtable 的数据存储量有限,必须定期处理,较少的数据能够保存更长的时间。对此,Dapper 团队设计了第二次的抽样。这次抽样发生在数据写入 Bigtable 之前,具体方法是将监控 id 散列)成一个标量 z ,其中 $0 \leq z \leq 1$ 。如果某个区间的 z 小于事先定义好的汇总抽样系数(Collection Sampling Coefficient),则保留这个区间并将它写入 Bigtable。否则丢弃不用。也就是说在采样决策中利用 z 值来决定某个监控树是整棵予以保留还是整棵弃用。这种方法非常的巧妙,因为在必要时只需改动 z 值就可以改变整个系统的写入数据量。利用二次抽样技术成功地解决了低开销及广泛可部署性的问题。

上面的两种技术手段解决了主要设计问题,这使得 Dapper 在 Google 内部得到了广泛的应用。Dapper 守护进程已成为 Google 镜像的一部分,因此 Google 所有的服务器上都有运行 Dapper。

2.6.4 常用 Dapper 工具

1. Dapper 存储 API

Dapper 的“存储 API”简称为 DAPI,提供了对分散在区域 Dapper 存储库(DEPOTS)的监控记录的直接访问。一般来说,有以下三种方式可以对这些记录进行访问。

(1) 通过监控 id 访问 (Access by Trace id): 利用全局唯一的监控 id 直接访问所需的监控数据。

(2) 块访问 (Bulk Access): DAPI 可以借助 MapReduce 来提供对数以十亿计的 Dapper 监控数据的并行访问。用户覆写一个将 Dapper 监控作为其唯一参数的虚函数 (Virtual Function),在每次获取用户定义的时间窗口内的监控数据时架构都将引用该函数。

(3) 索引访问 (Indexed Access): Dapper 存储库支持单索引 (Single Index),因为监控 id 的分配是伪随机的,这是快速访问同特定服务或主机相关监控的最好方式。

根据不完整的统计,目前大约有三个基于 DAPI 的持久应用程序,八个额外的基于 DAPI 的按需分析工具及大约 15~20 个使用 DAPI 框架构建的一次性分析工具。

2. Dapper 用户界面

大部分的用户在使用 Dapper 时都是通过基于 Web 的交互式用户界面,图 2-34~

图 2-38 显示其一般性的使用流程。

(1) 首先用户需要选择监控对象，包括监控的起止时间、区分监控模式的信息（图 2-34^[18]中是区间名）及一个衡量开销的标准（图 2-34 中是服务延迟）。

Job Selection

Start Date: 05/06/2008
 Start Hour: 09
 End Date: 05/06/2008
 End Hour: 10
 Cluster: clusterABC
 User: user123
 Job: jobXYZ

Node Information

☐ User
☒ RPC or Span Name
☐ Job
☐ Cluster

Cost Metric

☒ Latency
☐ Parent Latency
☐ Request Size
☐ Response Size
☐ Recursive Size
☐ Recursive Queue Time

图 2-34 监控对象选择

(2) 如图 2-35^[18]所示，一个大的性能表给出了所有同指定监控对象有关的分布式执行模式的简要情况。用户可以按其意愿对这些执行模式进行排序并选择某一个查看更多的细节。

Id	Calls	Total (ms)	Global 90%ile Contribution (count)	Local 90%ile (ms)	Absolute Histogram (ms)	Scaled Histogram (ms)	View
All	40,990,720 (100.00%)	139,773,132.8 (100.00%)	4,098,118 (100.00%)	8.91			View
E	3,450,880 (8.42%)	39,437,312.0 (28.22%)	1,918,437 (46.81%)	19.17			View
R	1,658,880 (4.05%)	55,939,686.4 (40.02%)	1,658,880 (40.48%)	47.21			View

图 2-35 监控对象相关的执行模型

(3) 图 2-36^[18]是某个选中的分布式执行模式，该执行模式以图形化描述呈现给用户。

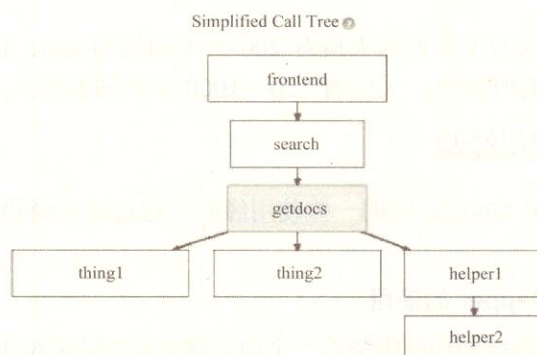


图 2-36 特定的执行模式

(4) 根据最初选择的开销度量标准, Dapper 会以频度直方图的形式将步骤 (3) 中选择的执行模式的开销分布展示出来, 如图 2-37^[18]所示, 同时呈现给用户的还有一系列特殊的监控样例信息, 这些信息落在直方图的不同部分。用户可以进一步的选择这些监控样例。

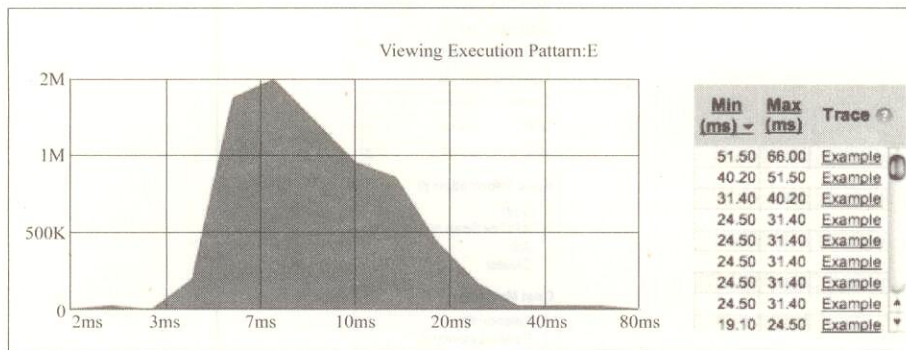


图 2-37 执行模式开销的频度直方图

(5) 在用户选择了某个监控样例后, 就会进入所谓的监控审查视图 (Trace Inspection View)。图 2-38^[18]是部分的监控审查视图, 在这个视图中, 最顶端是一条全局的时间线 (Global Time Line)。每一行是一个监控树, 选择 “+” 或 “-” 能够展开或折叠监控树。每个监控树用嵌套的彩色长方形表示的。每个 RPC 区间又被进一步的分成花在服务器处理上的时间和花在网络通信上的时间。用户注释并未在图中显示出来, 但是它们可以按照每个区间被选择包含在全局时间线上。

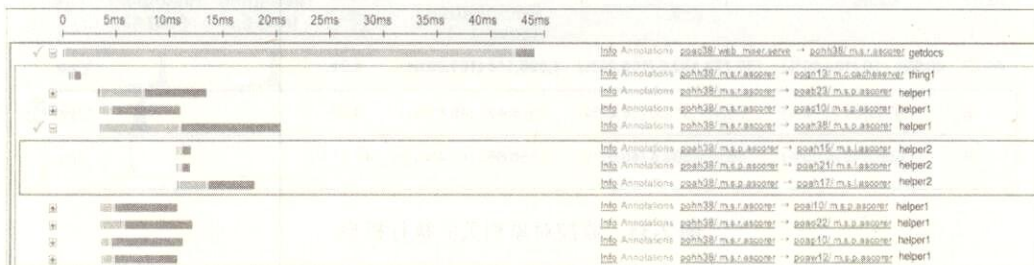


图 2-38 监控审查视图

根据统计, 一个普通的工作日内大概有 200 个不同的 Google 工程师在使用 Dapper 用户界面。因此, 在一周的时间里, 大约有 750~1000 个不同的用户。

2.6.5 Dapper 使用经验

本节介绍 Dapper 在 Google 中的一些使用经验, 通过这些经验可以看出在哪些场景中 Dapper 是最适用的。

1. 新服务部署中 Dapper 的使用

Google 的 AdWords 系统的构建围绕着一个由关键字命中准则和相关的文字广告组成的大型数据库。在这个系统进行重新开发时, 开发团队从原型系统直到最终版本的发布过程中,

反复的使用了 Dapper。开发团队利用 Dapper 对系统的延迟情况进行一系列的跟踪,进而发现问题,最终证明 Dapper 对于 AdWords 系统的开发起到了至关重要的作用。

2. 定位长尾延迟 (Addressing Long Tail Latency)

Google 最重要的产品就是搜索引擎,由于规模庞大,对其进行调试是非常复杂的。当用户请求的延迟过长,即延迟时间处于延迟分布的长尾时,即使最有经验的工程师对这种端到端性能表现不好的根本原因也常常判断错误。通过图 2-39^[18]不难发现,端到端性能和关键路径上的网络延迟有着极大的关系,因此发现关键路径上的网络延迟常常就能够发现端到端性能表现不佳的原因。利用 Dapper 恰恰能够比较准确的发现关键路径。

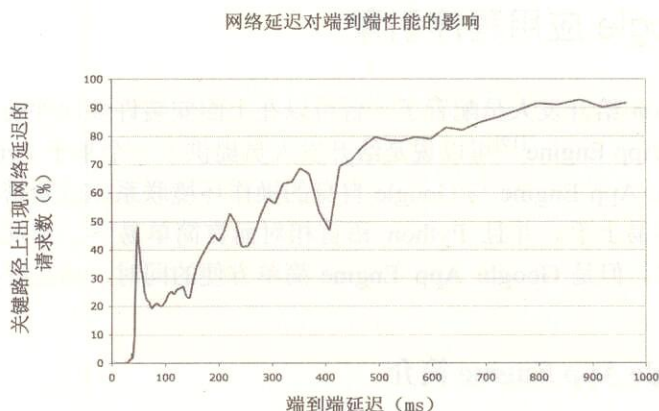


图 2-39 关键路径网络延迟对于端到端性能表现的影响

3. 推断服务间的依存关系 (Inferring Service Dependencies)

Google 的后台服务之间经常需要互相的调用,当出现问题时需要确定该时刻哪些服务是相互依存的,因为这样有利于发现导致问题的真正原因。Google 的“服务依存关系”项目使用监控注释和 DPAI 的 MapReduce 接口实现了服务依存关系确定的自动化。

4. 确定不同服务的网络使用情况

在 Dapper 出现之前,Google 的网管人员在网络出现故障时几乎没有工具能够确定到底是哪个部分的网络出现的故障。而现在 Google 利用 Dapper 平台构建了一个连续不断更新的控制台,用来显示内部集群网络通信中最活跃的应用层终端。这样在出现问题时可以最快的定位占用网络资源最多的几个服务。

5. 分层的共享式存储系统

Google 中的许多存储系统都是由多个相对独立且具有复杂层次的分布式基础架构组成。例如,Google App Engine 是构建在一个可扩展的实体存储系统之上的。而该实体存储系统则是构建在底层的 Bigtable 之上,展现出一些 RDBMS (关系型数据库管理系统) 的功能。而 Bigtable 又依次用到了 Chubby 和 GFS。在这样的层次式系统中决定端用户的资源消耗模式并不总是那么简单。例如,由 Bigtable 的单元引起的 GFS 高流量可能主要由一个用户或几个用户产生,但是在 GFS 的层次上这两种不同的使用模式是没法分开的。更进一步,在没有 Dapper 之类的工具的情况下对于这种共享式服务资源的争用也同样难以调试。

6. 利用 Dapper 进行“火拼”(Firefighting with Dapper)

这里所谓的“火拼”是指处于危险状态的分布式系统的代表性活动。正在“火拼”中的 Dapper 用户需要访问最新的数据却没有时间来编写新的 DAPI 代码或者等待周期性的报告,此时可以通过和 Dapper 守护进程的直接通信,将所需的最新数据汇总在一起。

Dapper 在 Google 内部取得了巨大的成功,虽然这种成功在一定程度上得益于 Google 内部系统的同构性,但是 Dapper 团队的创新性设计才是系统取得成功的根本性因素。Google 的后台系统可以说是目前全球最大的一个云平台,读者借鉴 Dapper 的设计思想一定能够为不同规模的云平台设计出合适的监控系统。

2.7 Google 应用程序引擎

如果说 Amazon 给开发人员配置了一台可以在上面安装许多软件的虚拟机的话(参见第3章),Google App Engine^[19]可以说是给开发人员提供了一个基于 Python 语言的 Django 框架。由于 Google App Engine 与 Google 自身的操作环境联系比较紧密,涉及底层的操作很少,用户比较容易上手。并且 Python 语言相对而言简单易学,开发人员可以很容易地开发出自己的程序。但是 Google App Engine 简单方便的同时,却在提供的解决方案上有着自己的局限性。

2.7.1 Google App Engine 简介

Google 公司发展迅速,不断推出自己的新产品,比如 Google 搜索、Google Maps、Google Earth、Google AdSense、Google Reader 等。在推出自己产品的同时,Google 倾力打造了一个平台,来集成自己的服务并供开发者使用,这就是 Google App Engine 平台。

简单地讲,Google App Engine 是一个由 Python 应用服务器群、Bigtable 数据库及 GFS 数据储存服务组成的平台,它能为开发者提供一体化的、可自动升级的在线应用服务。

从云计算平台的分类来看,Amazon 提供的是 IaaS 平台,而 Google 提供的 Google App Engine 是一个 PaaS 平台,用户可以在上面开发应用软件,并在 Google 的基础设施上运行此软件。其定位是易于实施和扩展,无需服务器维护。

Google App Engine 可以让开发人员在 Google 的基础架构上运行网络应用程序。在 Google App Engine 之上易构建和维护应用程序,并且应用程序可根据访问量和数据存储需要的增长轻松进行扩展。使用 Google App Engine,开发人员将不再需要维护服务器,只需上传应用程序,它便可立即为用户提供服务。

在 Google App Engine 中,用户可以使用 appspot.com 域上的免费域名为应用程序提供服务,也可以使用 Google 企业应用套件从自己的域为它提供服务。开发人员可以与全世界的人共享自己的应用程序,也可以限制为只有自己组织内的成员可以访问。

除此之外,还可以免费使用 Google App Engine。注册一个免费账户即可开发和发布应用程序,而且不需要承担任何费用和责任。免费账户可以使用多达 500MB 的持久存储空间,以及可支持每月约 500 万页面浏览量的超大 CPU 和带宽。

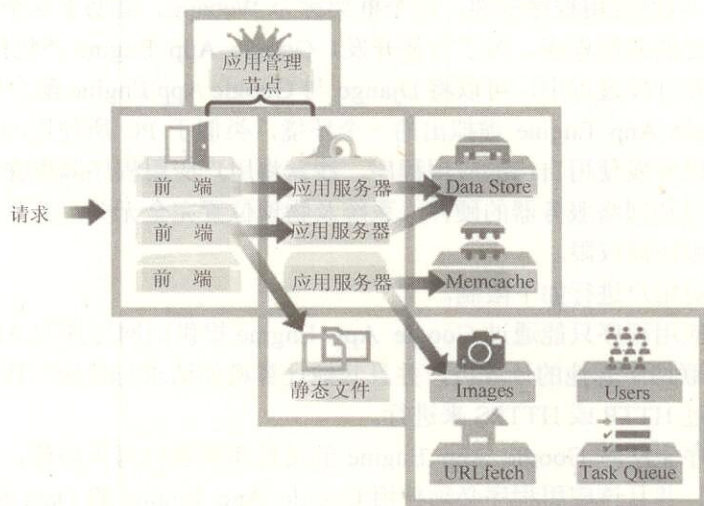
Google App Engine 作为一个开发平台,有其自身的特点。

Google App Engine 的整体架构如图 2-40^[20]所示。Google App Engine 的架构可以分成

四部分：前端和静态文件负责将请求转发给应用服务器并进行负载均衡和静态文件的传输；应用服务器则能同时运行多个应用的运行时（Runtime）；服务器群提供了一些服务，主要有 Memcache、Images、URLfetch、Email 和 Data Store 等；Google App Engine 还有一个应用管理节点，主要负责应用的启停和计费。

关于 Google App Engine 的一些基本概念，比如应用程序环境、沙盒、Python 运行时环境、数据库、Google 账户、App Engine 服务、开发流程、配额和限制等，总体而言，每个开发程序都将涉及这些概念。每个开发程序有自身的应用程序环境（这个环境由 Google App Engine 提供），该环境对应用程序提供了一些基本的支持，使应用程序可以在 Google App Engine 上正常运行。除此之外，Google App Engine 为每个应用程序提供了一个安全运行环境（即沙盒），该沙盒可以保证每个应用程序能够安全地隔离运行。现阶段，Google App Engine 支持 Java 和 Python 语言，通过 Google App Engine 的 Java 运行时环境，可以使用标准 Java 技术构建应用程序。开发程序时还可能要使用到 Python 运行时环境，该环境包括 Python 运行库等模块，并且 Google App Engine 还提供了一个由 Python 语言编写的网络应用程序框架 webapp。Google App Engine 上开发的应用程序使用的是 Data Store 数据库，该数据库不同于日常使用的 Oracle、SQL Server 等数据库，它是一个分布式存储数据库，可以随着应用程序访问量的增加而增加。使用 Google App Engine 开发应用程序必须拥有一个 Google 账户，有了该账户之后才可以在 Google App Engine 上运行开发的程序。为了简化开发流程，Google App Engine 提供了一些服务，这些服务统称为 App Engine 服务，使用 Google App Engine 开发应用程序必须遵守一定的开发流程。Google App Engine 为每个 Google 账户用户提供了一些免费的空间与流量支持，但是免费的空间和流量有一定的配额和限制。

通过对这些概念的了解，可深入理解 Google App Engine。



2.7.2 应用程序环境

Google App Engine 有着自身的应用程序环境，这个应用程序环境包括以下特性。

- (1) 动态网络服务功能。能够完全支持常用的网络技术。
- (2) 具有持久存储的空间。在这个空间里面平台可以支持一些基本操作, 如查询、分类和事务的操作。
- (3) 具有自主平衡网络和系统的负载、自动进行扩展的功能。
- (4) 可以对用户的身份进行验证, 并且支持使用 Google 账户发送邮件。
- (5) 有一个功能完整的本地开发环境, 可以在自身的计算机上模拟 Google App Engine 环境。
- (6) 支持在指定时间或定期触发事件的计划任务。

基于这样的环境支持, Google App Engine 可以在负载很重和数据量极大的情况下轻松构建安全运行的应用程序。

最开始 Google App Engine 只支持 Python 开发语言, 现阶段开始支持 Java 语言。本书案例中, Google App Engine 应用程序使用 Python 编程语言实现。该运行时环境包括完整的 Python 语言和绝大多数的 Python 标准库。在 Python 运行时环境中使用的是 Python 2.5.2 版本。这里先详细介绍一下 Python 运行时环境。

Python 运行时环境包括 Python 标准库, 开发人员可以调用库中的方法来实现程序功能, 但是不能使用沙盒限制的库方法。这些受限制的库方法包括尝试打开套接字、向文件进行写入操作等。为了便于编程, Google App Engine 设计人员将一些模块进行了禁用, 被禁用的这些模块的主要功能是不受运行时环境的标准库支持的, 因而, 开发者在导入这些模块的代码时程序将给出错误提示。

在 Python 运行时环境中, 应用程序只能以 Python 语言编写, 扩展代码中若有 C 语言, 则应用程序将不受系统支持。Python 环境为开发平台中的数据库、Google 账户、网址抓取和电子邮件服务等提供了丰富的 Python API。此外, Google App Engine 还提供了简单的 Python 网络应用程序框架, 这个框架称为 Webapp。借助于这个框架, 开发人员可以轻松构建自己的应用程序。为了方便开发, Google App Engine 还包括了 Django 网络应用程序框架, 在开发过程中, 可以将 Django 与 Google App Engine 配合使用。

沙盒是 Google App Engine 虚拟出的一个环境, 类似于 PC 所使用的虚拟机。在这个环境中, 用户可以开发使用自己的应用程序, 沙盒将用户应用程序隔离在自身的安全可靠的环境中, 该环境和网络服务器的硬件、系统及物理位置完全无关, 并且沙盒仅提供对基础操作系统的有限访问权限。

沙盒还可以对用户进行如下限制。

- (1) 用户的应用程序只能通过 Google App Engine 提供的网址抓取 API 和电子邮件服务 API 来访问互联网中其他的计算机, 并且其他计算机如请求与该应用程序相连接, 只能在标准接口上通过 HTTP 或 HTTPS 来进行。
- (2) 应用程序无法对 Google App Engine 的文件系统进行写入操作, 只能读取应用程序代码上的文件, 并且该应用程序必须使用 Google App Engine 的 Data Store 数据库来存储应用程序运行期间持续存在的数据。
- (3) 应用程序只有在响应网络请求时才运行, 并且这个响应时间必须极短, 在几秒之内必须完成。与此同时, 请求处理的程序不能在自己的响应发送后产生子进程或执行代码。

简言之, 沙盒给开发人员提供了一个虚拟的环境, 这个环境使应用程序与其他开发者开发使用的程序相隔离, 从而保证每个使用者可以安全地开发自己的应用程序。

开发人员开发程序必须使用 Google App Engine SDK, 即 Google App Engine 软件开发套件。可以先下载这个套件到自己的本地计算机上, 然后进行开发和运行。使用 SDK 时, 可以在本地计算机上模拟包括所有 Google App Engine 服务的网络服务器应用程序, 该 SDK 包括 Google App Engine 中的所有 API 和库。该网络服务器还可以模拟沙盒环境, 这些沙盒环境用来检查是否存在禁用的模块被导入的情况, 以及对不允许访问的系统资源的尝试访问等情况的发生。

Google App Engine SDK 完全使用 Python 实现, 这个开发套件可以在装有 Python 2.5 的任何平台上面运行, 包括 Windows、Mac OS X 和 Linux 等, 开发人员可以在 Python 网站上获得适合自己系统的 Python。

该开发套件还包括将应用程序上传到 Google App Engine 之上的工具。用户创建自己应用程序的代码、静态文件和配置文件之后, 就可以运行这个工具将数据上传到平台上面。在上传过程中, 该工具还将提示开发者输入 Google 账户和电子邮件地址及密码等信息。

系统中有一个管理控制台, 这个管理控制台有一个网络接口, 用于管理在 Google App Engine 上运行的应用程序。开发人员可以使用管理控制台来创建应用程序、配置域名、更改应用程序当前的版本、检查访问权限和错误日志以及浏览应用程序数据库等。

2.7.3 Google App Engine 服务

Google App Engine 提供了多种服务。这些服务可以帮助开发人员在管理应用程序的同时执行常规操作, 可以通过以下 API 来使用 Google App Engine 提供的服务。

1. 图像操作 API

开发的应用程序可以使用 Google App Engine 提供的图像操作 API 对图像进行操作, 使用该 API 可以对 JPEG 和 PNG 格式的图像进行缩放、裁剪、旋转和翻转等操作。

1) Image 类

Image 类来自于 `google.appengine.api.images` 模块, 该类可以用来封装图像信息及转换该图像, 转换时可以使用 `execute_transforms()` 方法; 可以使用 `class Image(image_data)` 来构造函数, 参数 `image_data` 表示字节字符串 (str) 格式的图像数据; 可以采用 PNG、JPEG、TIFF 或 ICO 等格式对图像数据进行编码。

Image 类中主要有如下实例方法。

(1) `resize(width=0, height=0)`: 该方法用来缩放图像, 可以将图像缩小或放大到参数指定的宽度或者高度。参数 `width` 和 `height` 都是以像素数量来表示, 并且必须是 int 型或 long 型。

(2) `crop(left_x, top_y, right_x, bottom_y)`: 该方法可以将图像裁剪到指定边界框的大小, 并且裁剪后以相同的格式返回转换的图像。参数 `left_x` 表示边界框的左边界, `top_y` 表示边界框的上边界, `right_x` 表示边界框的右边界, `bottom_y` 表示边界框的下边界。以上四个参数均采用指定为 float 类型值的从 0.0 到 1.0 的图像宽度的比例 (其中 float 值包括了 0.0 和 1.0)。

(3) `rotate(image_data, degrees, output_encoding=images.PNG)`: 该方法是用来旋转图像。参数 `degrees` 表示图像旋转的量, 采用的形式是度数, 且这个度数必须是 90° 的倍数, 数据格式必须为 int 型或 long 型, 使用该函数对图像进行旋转是沿顺时针方向执行。

`image_data` 是指要旋转的图像, 是 JPEG、GIF、BMP、TIFF 或者 ICO 等格式的字节字符串 (str)。`output_encoding` 指转换的图像所需的格式, 可以是 `images.PNG` 或 `images.JPEG` 格式, 默认的格式是 `images.PNG` 格式。

(4) `horizontal_flip(image_data, output_encoding=images.PNG)`: 该函数表示对图像进行水平翻转。参数 `image_data` 表示要翻转的图像是 JPEG、PNG、TIFF 或 ICO 格式的字节字符串 (str)。`output_encoding` 参数表示要转换的图像所需要的格式, 可以是 `images.PNG` 或是 `images.JPEG`, 默认的格式是 `images.PNG` 格式。

(5) `vertical_flip(image_data, output_encoding=images.PNG)`: 该函数表示垂直地翻转图像, 并且转换后的图像与以前的格式一样。

2) exception 类

`google.appengine.api.images` 包为用户主要提供了以下 exception 类。

(1) `exception Error()`: 这是该包中所有异常的基类。

(2) `exception TransformationError()`: 表示尝试转换图像时发生错误。

(3) `exception BadRequestError()`: 表示转换参数无效。

2. 邮件 API

Google App Engine 为开发的应用程序提供了电子邮件服务。邮件 API 为用户提供了两种方式来发送电子邮件, 分别是 `mail.send_mail()` 函数和 `EmailMessage` 类。发送电子邮件时可以发送附件, 为了安全考虑, 用户发送的附件必须是所允许的文件类型。

1) 允许的附件类型

允许作为电子邮件附件的 MIME 类型以及相对应的文件扩展名主要有: 图像格式包括 BMP、GIF、JPEG、JPG、JPE、PNG、TIFF、TIF、WBMP; 文本格式包括 CSS、CSV、HTM、HTML、TEXT、TXT、ASC、DIFF、POT; 应用程序格式包括 PDF、RSS。

2) EmailMessage 类

邮件 API 中的 `EmailMessage` 类由 `google.appengine.api.mail` 包提供。`EmailMessage` 实例代表那些要使用 Google App Engine 邮件服务来进行发送的电子邮件, 电子邮件中有一组字段, 这组字段可以使用构造函数来进行初始化。

(1) 构造函数。在构造函数 `class EmailMessage(**kw)` 中, 邮件的字段可以使用传递到构造函数的关键字参数进行初始化, 并且字段还可以在构造之后对实例的属性进行设置, 也可以通过 `initialize()` 方法来设置。

(2) 实例方法。`check_initialized()` 方法用来检查 `EmailMessage` 类是否已经进行了正确的初始化, 以便对邮件进行发送。若邮件成功发送, 则该方法不会返回错误, 否则会抛出与其找到的第一个问题对应的错误。

`initialize(**kw)` 方法只是对 `EmailMessage` 是否进行了正确的初始化进行判断。如果是则返回 `True`, 与 `check_initialized()` 一样执行同样的操作, 区别只是不抛出错误。

`send()` 方法用来发送电子邮件。

(3) 函数。`google.appengine.api.mail` 包为邮件 API 主要提供了以下函数。

(a) `is_email_valid(email_address)`: 如果参数 `email_address` 是有效的电子邮件地址,

则函数返回 True。该函数会执行与 `check_email_valid` 相同的检查，但是不会抛出异常。

(b) `send_mail(sender, to, subject, body, **kw)`: 创建并且发送一封电子邮件。`sender`、`to`、`subject` 和 `body` 参数是邮件必填的字段。其他的字段也可以指定为关键字参数。

(4) 异常。`google.appengine.api.mail` 包为邮件 API 主要提供了以下 `exception` 类。

(a) `exception Error()`: 该包中所有异常的基类。

(b) `exception BadRequestError()`: 邮件服务以无效为理由拒绝 `EmailMessage`。

(c) `exception InvalidEmailError()`: 表示该电子邮件的地址无效。电子邮件地址字段仅接受有效的电子邮件地址，例如 `sender` 或 `to`。

3. Memcache API

高性能的网络应用程序一般在运行之前需要使用分布式内存数据缓存 (Memcache)，或用分布式内存数据缓存来代替某些任务的稳定持久存储，Google App Engine 为用户提供了这样一个高性能的内存键值缓存，可以使用应用程序的实例来访问这个缓存。Memcache 适合存储永久性功能和事务性功能的数据，例如，可以将临时数据或数据库数据复制到缓存以进行高速的访问。

Memcache API 提供了一个基于类的接口，以便和其他 Memcache API 相兼容。这里 `Client` 类由 `google.appengine.api.memcache` 包提供。

1) 构造函数

`class Client()` 产生与 Memcache 服务通信的客户端。

2) 实例方法

构造的 `Client` 实例主要有以下几种方法。

(1) `set(key, value, time=0, min_compress_len=0)`: 该方法用来设置键的值，与先前缓存中的内容无关。其中参数 `key` 表示要设置的键，`key` 可以是字符串或 (哈希值，字符串) 格式的元组；参数 `value` 表示要设置的值；参数 `time` 是指可选的过期时间，可以是相对当前时间的秒数 (最多 1 个月)，也可以是绝对 Unix 时间戳的时间；`min_compress_len` 是为了兼容性而忽略的选项。

(2) `get(key)`: 该方法用来在 Memcache 中查找一个键。参数 `key` 指明要在 Memcache 中查找的键，`key` 可以是字符串或 (哈希值，字符串) 格式的元组。如果在 Memcache 中找到键，则返回值为该键的值，否则返回 `None`。

(3) `delete(key, seconds=0)`: 该方法用来从 Memcache 删除键。参数 `key` 是指要删除的键，可以是字符串或 (哈希值，字符串) 格式的元组，参数 `seconds` 指定删除的项目对 [添加] 操作 [锁定] 的可选秒数，值可以是当前时间开始的增量，也可以是绝对 Unix 时间戳时间，默认情况下值为 0。

(4) `add(key, value, time=0, min_compress_len=0)`: 该方法用来设置值，但是只在项目没有处于 Memcache 时设置。参数 `key` 指明要设置的键，它可以是字符串或 (哈希值，字符串) 格式的元组；参数 `value` 是指要设置的值；参数 `time` 指明可选的过期时间，可以是相对当前时间的秒数，也可以是绝对 Unix 时间戳时间；参数 `min_compress_len` 是为了兼容性而忽略的选项。

(5) `replace(key, value, time=0, min_compress_len=0)`: 该方法用来替换键的值。参数

key 指要设置的键，key 可以是字符串或（哈希值，字符串）格式的元组；参数 value 指明要设置的值；参数 time 是指可选的过期时间，可以是相对当前时间的秒数，也可以是绝对 Unix 时间戳时间；参数 min_compress_len 是为了兼容性而忽略的选项。

(6) incr(key, delta=1): 该方法可以自动增加键的值。在内部，值是无符号 64bit 整数，同时 Memcache 不会检查 64bit 溢出，如果值过大则会换行。这里的键必须已存在于缓存中才能增加值。初始化计数器时可以使用 set() 进行初始值的设置。参数 key 是指要增加的键，key 可以是字符串或（哈希值，字符串）格式的元组；参数 delta 值作为键的增加量的非负整数值（int 型或 long 型），默认值为 1。

(7) decr(key, delta=1): 该方法可以自动减少键的值。内部而言，值是无符号的 64bit 数，并且 Memcache 不检查 64bit 溢出，若值过大则会换行。初始化计数器时可以使用 set() 进行初始值设置。参数 key 指要减少的键，key 可以是字符串或（哈希值，字符串）格式的元组；参数 delta 是键的减少量的非负整数值（int 型或 long 型），默认值为 1。

(8) flush_all(): 该方法用来删除 Memcache 中的所有内容。若成功则返回 True，若是 RPC 或服务错误，则返回 False。

(9) get_stats(): 该方法指获取该应用程序的 Memcache 统计信息。函数的返回值是将统计信息名称映射到相关值的参照表。

4. 用户 API

Google App Engine 的功能和账号是集成的，因此应用程序可以让用户使用他们自身的 Google 账户登录。

1) User 对象

用户 API 主要是通过 User 类来实现其功能的，每个 User 类的对象代表着一个用户。User 对象是唯一的且可比较，若两个对象相同，则这两个对象代表着同一个用户。开发的应用程序可通过调用 users.get_current_user() 函数来访问当前用户的 User 对象，也可以利用电子邮件地址来构造 User 对象。

2) 登录网址

用户 API 提供了函数来构建到 Google 账户的网址，这样 Google 账户允许用户登录或退出，并重新定向到用户的应用程序。登录或退出目标网址可以使用 users.create_login_url() 和 users.create_logout_url()。

3) User 类

User 类的一个对象代表具有 Google 账户的一个用户，User 类是由 google.appengine.api.users 模块提供的。

(1) 构造函数。class User(email=None) 这个函数代表具有 Google 账户的用户。函数中参数 email 表示用户的电子邮件地址，默认为当前用户。若系统没有指定电子邮件地址，并且当前用户没有登录，那么系统将抛出 UserNotFoundError 错误。

系统在创建 User 对象时，不检查这个电子邮件地址是否有效。若该 User 对象的邮件地址不是有效的，则该 User 对象仍然可能存储在数据库中，但是不会与真正的用户相匹配。

(2) 实例方法。User 实例主要提供以下方法。

(a) `nickname()`: 用来返回用户的“昵称”。

(b) `email()`: 用于返回用户的电子邮件地址。

(3) 函数。 `google.appengine.api.users` 包主要提供以下函数。

(a) `create_login_url(dest_url)`: 用于返回一个网址。当用户访问这个网址时, 它将提示用户使用自己的 Google 账户登录, 并将用户重新定向到指定的 `dest_url` 网址。其中 `dest_url` 可以是完整的网址, 也可以是相对于应用程序的域的路径。

(b) `create_logout_url(dest_url)`: 用来返回一个网址。当用户访问这个网址时会注销这个用户, 然后将用户重新定位到指定的 `dest_url` 网址。其中参数 `dest_url` 可以是完整的网址, 或者是相对于应用程序的域的路径。

(c) `get_current_user()`: 若用户已登录, 则该函数返回当前用户的 `User` 对象; 若用户未登录, 返回 `None`。

(4) 异常。 `google.appengine.api.users` 包主要提供以下 `exception` 类。

(a) `exception Error()`: 这个包中所有异常的基类。

(b) `exception UserNotFoundError()`: 若用户没有提供电子邮件地址, 且当前用户未登录, 则系统将由 `User` 构造函数抛出异常。

(c) `exception RedirectTooLongError()`: 表示 `create_login_url()` 或 `create_logout_url()` 函数的重定向网址的长度超过了所允许的最大长度。

5. 数据库 API

Google App Engine 提供了一个强大的分布式数据存储服务。该服务包含查询引擎、事物功能等功能, 并且该数据库规模可以随着访问量的上升而扩大。Google App Engine 数据库和传统的关系数据库不同, 该数据库中的数据对象有一个类和一组属性。数据库中的查询可以检索按照属性值过滤的实体, 也可以检索按照分类的指定种类的实体, 其中属性值可以是任何一种受系统数据库支持的属性值类型。

Google App Engine 的数据库使用了简单的 API 来为用户提供查询引擎和事务存储服务, 并且这些服务都运行在 Google 的可扩展结构上。在 Google App Engine 中, Python 接口包含了数据建模 API 和类似于 SQL 的一种查询语言 (称为 GQL)。通过这些 API 和 GQL 查询语言, 可以极大地方便用户开发可扩展数据库的应用程序。

Google App Engine 的数据库 API 拥有一个用于定义数据模型的机制。这里 `Model` 用来描述实体的类型 (包括其属性的类型和配置)。数据库 API 提供两种查询接口: 查询对象接口和 GQL 查询语言。查询的结果以 `Model` 类的实例形式来返回实体, 并且这些 `Model` 类可以被修改并且放到数据库中。

1) Model 类

`Model` 类是数据模型定义的超类, 由 `google.appengine.ext.db` 包提供。

`Model` 类的构造函数定义如下。

`class Model(parent=None, key_name=None, **kw)`。其中参数 `parent` 用来作为新实体的父实体的 `Mode` 实例或 `key` 实例; 参数 `key_name` 是指新实体的名称, 并且 `key_name` 的值不得以数字开头, 也不能采用 `__*` 的形式, 存储为 `Unicode` 字符串, `str` 值转换为 `ASCII` 文本; 参数 `**kw` 表示实例的属性的初始值, 作为关键字参数。

(1) 类方法。Model 类主要提供以下类方法。

(a) Model.get(keys): 用来获取指定 key 对象的 Model 实例, 键值必须代表 Model 类的实例。若程序提供的键类型不符合, 则系统抛出 KindError。参数 keys 是指 Key 对象或 Key 对象的列表, 还可以是 Key 对象的字符串版本, 或字符串列表。

(b) Model.all(): 返回代表与该 Model 对应类型的所有实体的 Query 对象。在执行 Query 对象上的方法之前, 可以对查询进行过滤和排序。

(c) Model.gql(query_string, *args, **kwds): 用来对该 Model 的实例执行 GQL 查询。其中参数 query_string 指明 GQL 查询中 SELECT * FROM model 后的部分; 参数*args 用于位置参数绑定, 类似于 GqlQuery 构造函数; 参数**kwds 表示关键字参数绑定, 类似于 GqlQuery 构造函数。

(2) 实例方法。Model 实例主要有以下方法。

(a) key(): 返回该 Model 实例的数据库 Key。在 put()入数据库之前, Model 实例没有键。在实例拥有键之前调用 key()会抛出 NotSavedError 错误。

(b) put(): 将 Model 实例存储在数据库中。如果 Model 实例是新创建的并且之前从未存储过, 则该方法会在数据库中创建新的数据实体, 否则, 该方法会用当前属性值更新数据实体。该方法会返回存储的实体的 Key。

(c) delete(): 用来从数据库中删除 Model 实例。如果实例从未被 put()到数据库, 则删除不会起任何作用。

2) Property 类

Property 类也是一个超类, 用来对数据模型的属性进行定义。它可以定义属性值的类型、值的验证方式以及在数据库中的存储方式等, Property 由 google.appengine.ext.db 包提供。

(1) 类构造函数。Property 基类的构造函数定义如下。

class Property(verbose_name=None, name=None, default=None, required=False, validator=None, choices=None)。这是 Model 属性定义的超类。其中参数 verbose_name 表示用户友好的属性名称, 属性构造函数的第一个参数必须始终是这个参数。参数 name 表示的是属性的存储名称, 默认情况下, 该名称表示属性的属性名称。参数 default 是指属性的默认值, 若属性值从未被指定或值是 None, 则该属性值被视为默认值。参数 required 若是 True, 则属性值不能为 None, Model 实例必须要利用构造函数来初始化所有必需的属性, 这样创建实例时候才不会缺少值。参数 validator 表示分配属性值的时候应该调用以便用来验证值的函数, 函数使用该属性值为唯一的参数。参数 choices 表示可接受的属性值的列表, 如果设置了该参数, 则不能给属性分配该列表以外的其他值。

(2) 类属性。Property 类下面的子类可以定义下面的属性。

data_type 属性用来接受作为 Python 自有值的 Python 数据类型或类。

(3) 实例方法。Property 类实例主要具有以下方法。

(a) default_value(): 返回属性的默认值。其中基础的实施方案使用的是传递到构造函数 default 参数的值。

(b) validate(value): 表示属性的完整验证程序。若 value 值有效, 则函数返回该值。程序的基础实施方案将会检查以下内容: value 值是否为 None; 若已经根据选择的内容

对属性进行了设置,那么该值是否是一个有效的选择 (choices 参数);若这个值存在,那么该值是否已经通过自定义的程序的验证 (alidator 参数)。

(c) empty(value): 若这个属性类型的 value 使用的是空值,那么该应用程序将返回 True。

3) Query 类

Query 类是一个数据库查询的接口,程序可以使用对象和方法来准备这个查询。

Query 类由 google.appengine.ext.db 包提供。

(1) 构造函数。Query 类的构造函数的定义如下。

class Query(model_class)。函数主要表示使用对象和方法来准备数据查询的接口,由构造函数返回的 Query 实例表示的是对该类型的所有实体的查询。函数中参数 model_class 代表了查询的数据库实体类型的 Model (或者是 Expando) 类。

(2) 实例方法。Query 类主要有以下几种实例方法。

(a) filter(property_operator, value): 对属性的条件进行过滤,并加到该查询中,因而该查询只会返回满足所有条件的属性的实体。参数 property_operator 包含了属性名称和比较运算符的字符串,并且支持下列的比较运算符: <、<=、=、>=、>; 参数 value 用来代表比较过程中所用的置于表达式右侧的值。

(b) order(property): 用来给结果添加排序,并且结果将根据首先添加的顺序进行排列。参数 property 表示的是一个字符串,是要为其排序的属性的名称,若要将排列顺序改为降序,可以在名称前加上一个连字符 (-),若不加表示进行升序排列。

(c) ancestor(ancestor): 对祖先条件进行过滤,并且将它加入到查询,该查询只会返回以这个祖先条件为过滤器的那些实体。参数 ancestor 代表的是该祖先的 Model 实例或 Key 实例。

(d) get(): 执行查询,然后返回第一个结果。若这个查询没有返回任何结果,则会返回 None。

(e) fetch(limit, offset=0): 执行查询,然后返回结果。参数 limit 是必须有的一个参数,表示要返回的结果的数量。若满足条件的结果数量不够,则返回的结果或许会少于 limit 个。参数 offset 表示要跳过的结果的数量;返回值是一个 Model 实例列表,也可能是一个空的列表。

(f) count(limit): 返回这个查询所抓取的结果的数量。参数 limit 表示的是要计数的结果的最大数量。

4) GqlQuery 类

GqlQuery 类是一种使用了 Google App Engine 查询语言 (即 GQL) 的数据库查询接口。GqlQuery 类由 google.appengine.ext.db 包提供。

(1) 构造函数。GqlQuery 类的构造函数定义如下。

class GqlQuery(query_string, *args, **kwds), 函数使用的是 GQL 的 Query 对象。参数 query_string 表示的是以 SELECT * FROM model-name 开头的完整 GQL 语句;参数 *args 表示位置参数绑定;参数 **kwds 表示关键字参数绑定。

(2) 实例方法。GqlQuery 实例主要有以下方法。

(a) bind(*args, **kwds): 重新绑定参数进行查询。新的查询将会在重新绑定参数之后第一次访问结果时来执行。参数 *args 表示新位置参数绑定;参数 **kwds 表示新关键字参数绑定。

(b) `get()`: 执行查询, 并且将返回第一个结果。若查询之后没有返回结果就返回 `None`。

(c) `fetch(limit, offset=0)`: 执行查询, 然后返回结果。参数 `limit` 表达的是程序将要返回的结果的数量, 是必需的参数。当结果数是未知的时候, 可以迭代地使用 `GqlQuery` 对象而不是使用 `fetch()` 方法来从查询结果中获取每个结果。参数 `offset` 是指要跳过的结果的数量, 返回的值是一个 `Model` 实例列表, 也可能是一个空的列表。

(d) `count(limit)`: 返回该查询抓取的结果的数量。`count()` 比那些通过常量系数来进行检索的速度要快一些。参数 `limit` 表示的是要计数的结果的最大值。

5) Key 类

`Key` 类的实例代表的是数据库实体唯一键, `Key` 类由 `google.appengine.ext.db` 包提供。

(1) 构造函数。`class Key(encoded=None)`。函数表示的是数据库对象的唯一键。用户可以通过将 `Key` 对象传递到 `str()` (或调用对象的 `__str__()` 方法), 也可以把键编码成字符串。参数 `encoded` 表示的是 `Key` 实例的 `str` 形式。

(2) 类方法。`Key` 类提供以下类方法。

`Key.from_path(*args, **kwargs)` 方法表示从一个或者多个实体键的祖先路径来构建新的 `Key` 对象。这里的路径代表的是实体中父子之间关系的层次结构, 每一个实体都是由实体的类型以及其数字 ID 或键名来代表。参数 `*args` 是从根实体到主题的路径; 参数 `**kwargs` 是关键字参数。

(3) 实例方法。`Key` 实例主要有以下方法。

(a) `app()`: 返回存储数据实体的应用程序的名称。

(b) `kind()`: 以字符串形式返回数据实体的类型。

(c) `id()`: 以整数形式返回数据实体的数字 ID, 若实体没有数字 ID, 则函数返回 `None`。

(d) `name()`: 返回数据实体的名称, 若实体没有名称则返回 `None`。

(4) 函数。`google.appengine.ext.db` 包主要提供以下函数。

(a) `get(keys)`: 用于获取任何 `Model` 的指定键的实体。其中参数 `keys` 表示的是 `Key` 对象或 `Key` 对象的列表。

(b) `put(models)`: 将一个或多个 `Model` 实例放置到数据库中。参数 `models` 表示的是要存储的 `Model` 实例或 `Model` 实例的列表。

(c) `delete(models)`: 从数据库中删除一个或多个 `Model` 实例。参数 `models` 表示要删除的 `Model` 实例、实体的 `Key`, 或 `Model` 实例列表, 也可以是实体的 `Key` 列表。

(d) `run_in_transaction(function, *args, **kwargs)`: 用于在一个事务中运行包含数据库更新的函数。若代码在事务处理过程之中抛出异常, 则事务中进行的所有数据库更新都将回滚。参数 `function` 指的是要在数据库事务中运行的函数; 参数 `*args` 是指传递到函数的位置参数; 参数 `**kwargs` 是指传递到函数的关键字参数。

2.7.4 Google App Engine 编程实践

Google App Engine 是 Google 推出的一项 Web 主机服务, 可以让用户在 Google 的基础架构上运行网络应用程序。相比其他的 Web 主机服务, Google App Engine 有下列独到之处。

- (1) 将 Web 应用部署到 Google 的基础设施之上。
- (2) 提供数据存储服务。
- (3) 集成了 Gmail、Google User 认证、URL Fetch、Memcache 和图片操作 (PIL) 等多种 API。
- (4) Google App Engine 提供存储空间为 500M, 每月 500 万页面访问的免费服务, 超出部分需要支付相应的费用。

Google App Engine 最初只支持 Python, 在 2009 年 4 月向 Java 开发人员开放了其云计算平台, 但出于对平台安全的保护, 限制了一部分 API 的使用, 本文用 Java 在 Google App Engine 上实现了个人空间的日志管理功能。

1. 搭建开发平台

- (1) 下载 JDK1.6 安装并配置好环境变量。
- (2) 下载 eclipse3.6 安装包, 解压。
- (3) 下载 Google Plugin for Eclipse3.6, 解压后将插件包中 feature 下的文件复制到 eclipse 的 feature 目录下, 将插件包中 plugin 下的文件复制到 eclipse 的 plugin 目录下 (也可以通过建立 link 文件来安装插件)
- (4) 下载 appengine-java-sdk, Google Plugin for Eclipse3.6 内部集成了 sdk, 如果下载的版本里面没有, 则需要下载 sdk 并解压, 在 eclipse 中依次点击 Window>Preference>Google>App Engine>Add, 添加 sdk 根目录, 如图 2-41 所示。appengine java sdk 是对 Google App Engine 的本地模拟, 内嵌服务器软件, 通过它可以在本地对用户的应用进行测试。

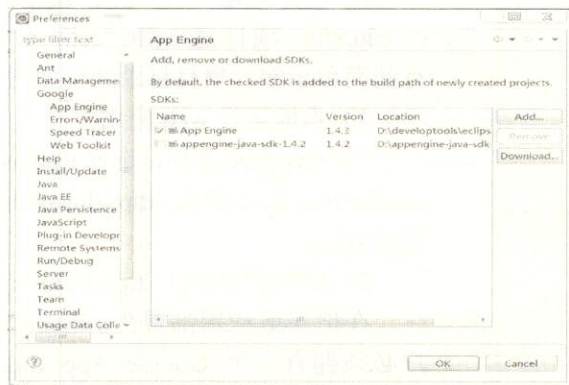


图 2-41 添加 appengine java sdk 环境

安装完成之后重启 eclipse, 会看到工具栏上多了四个图标, 如图 2-42 所示, 表示用户已经安装成功, 可以进行 Google App Engine 应用开发了。



图 2-42 工具栏 Google App Engine 图标

2. 创建工程并测试

创建一个新的 Web Application, 单击 file→New→project, 在对话框中选择 google 下面的 “google web application”。填写 web 应用的名称和包的名称。如果不使用 GWT, 则不选 “Use Google Web Toolkit”, 勾选 “Use Google App Engine”, 之后单击完成按钮, 如图 2-43 所示。

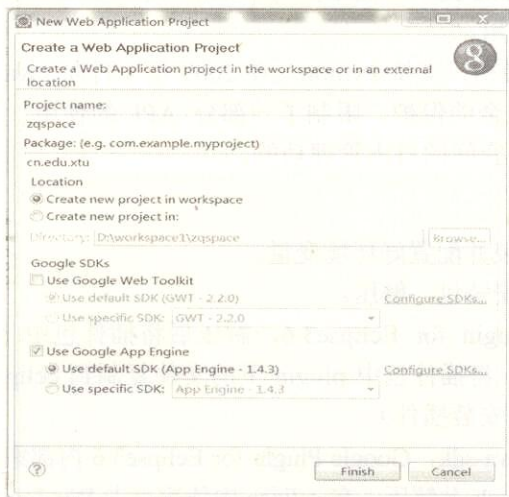


图 2-43 新建工程

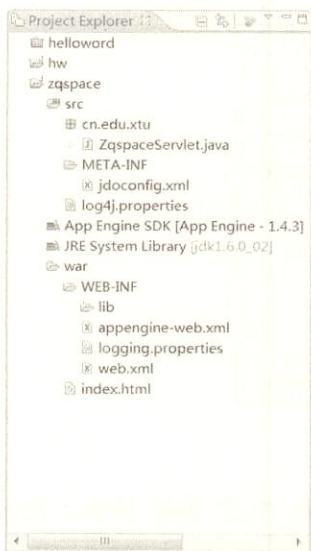


图 2-44 项目目录结构

在这里新建一个名为 zqspace 的项目, 包名为 cn.edu.xtu, 项目建立后, eclipse 自动生成了一个可运行的简单示例, 项目目录结构如图 2-44 所示。

1) 本地运行 web 应用

右击 zqspace 工程, 选择 Run as...>Web Application, 然后再浏览器中输入 <http://localhost:8888/zqspace> 查看运行效果。

2) 部署到 Google App Engine

在把 web 应用部署在 Google App Engine 以前, 用户必须拥有一个 Google App Engine 账号。可以登录 <http://appengine.google.com> 按照步骤创建账号。

获得账号登录后, 当第一次创建 Application 时必须通过短消息验证。选择其他国家和地区, 输入用户手机号码 (手机号码前需要加+86), 之后在下一步的验证中输入手机收到的验证码, 填写一个唯一的 application ID 和一个 title 创建 application, 这里创建的 id 为 dengpenggac, title 为 zqspace。

有了 Google App Engine 账号和 Application 后, 在 eclipse 中右击工程, google >app engine settings, 输入 application id 和版本号, 也可修改 appengine-web.xml 中的 application 标签。

右击工程 zqspace> Google > Deploy to app engine.输入 Google App Engine 账号名和密

码, 单击“deploy”按钮。如图 2-45 所示, 部署成功后在浏览器中输入 <http://dengpenggae.appspot.com> 进行查看, 如果浏览器中出现 hello word 字样说明应用部署成功。

3. 开发个人空间 zqspace

在 Google App Engine 上进行开发与传统的 Java web 开发在某些方面是不同的, 特别需要注意 Google App Engine 对 Java 功能上的限制, 这些在官方的文档上有相应的说明, 同时在 Google App Engine 上可以直接调用 Google 的 API, 实现丰富的应用。

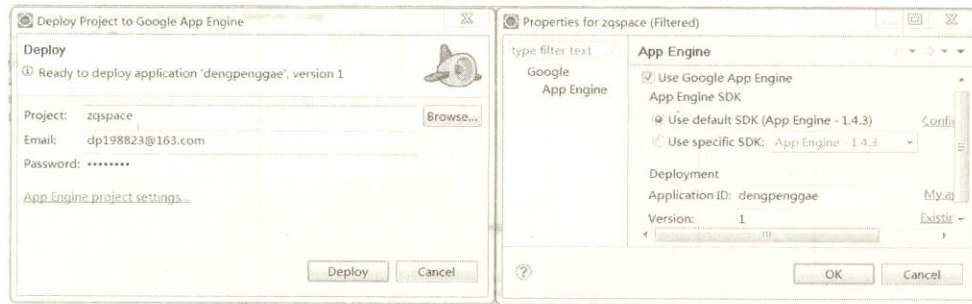


图 2-45 部署应用

1) 需求分析

系统用例图如图 2-46 所示。

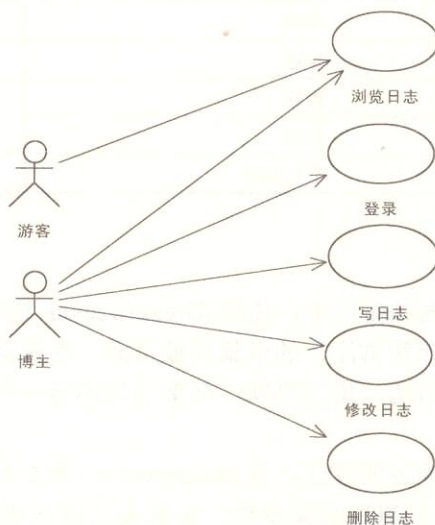


图 2-46 系统用例

针对写日志用例的序列图如图 2-47 所示, 修改日志与写日志处理流程相似。

2) 数据模型设计

在个人空间的日志模块中涉及用户和日志这两个数据类型, 可以通过 Google 的用户 API 实现, 在这里只需设计日志 (Article) 这一数据模型, 如表 2-3 所示, app engine 的数据持久化在 Bigtable 中的, Bigtable 不是关系型数据库, 但是可以通过 JDO 的封装将数据持久化在 Bigtable 中。

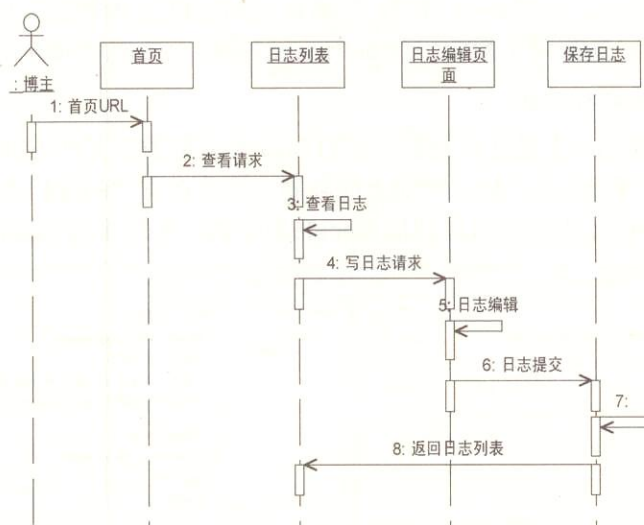


图 2-47 写日志时序图

表 2-3 日志（Article）的数据模型

字段名	描述	类型
id	主键（自增）	Long
author	作者	User
title	标题	String
content	内容	String
tag	标签	String
date	更新时间	Date

3) 页面设计

在日志模块内涉及的页面共有三个，使用 Dreamweaver 进行设计，首页 index.jsp 是基于“列弹性，右侧栏、标题和脚注”的框架模板页面，在首页上主要包括的组件有：近期日志列表，登录框，时钟日历，最近更新，框架的顶部是一幅图片和菜单栏。首页预览如图 2-48 所示。

ArticleList.jsp 显示日志列表的页面，在 Dreamweaver 中是基于“列弹性，居中，标题和脚注”框架的页面，上部是图片和菜单栏，中部是二行二列的表格，左侧是日志管理栏，右侧是日志列表，下部是版权声明。日志列表页面预览如图 2-49 所示。

writeArticle.jsp 是写日志的界面，基于“列弹性，居中，标题和脚注”框架的页面，中部是一个填写日志的表单，包括标题，内容，标签等，编辑器采用 CKeditor。写日志页面预览如图 2-50 所示。



图 2-48 首页



图 2-49 日志列表页面

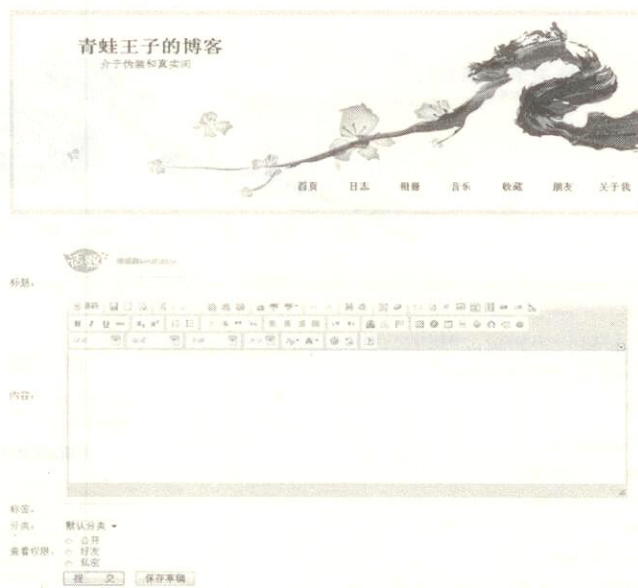


图 2-50 写日志页面

4) 实体类编码

在 cn.edu.xtu 包下新建一个 entity 包，在 entity 包下新建 Article 类，使用 JDO 实现持久化，Article 的关键代码如下。

```
package cn.edu.xtu.entity;
import java.util.Date;
import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.IdentityType;
import javax.jdo.annotations.PersistenceCapable;
import javax.jdo.annotations.Persistent;
import javax.jdo.annotations.PrimaryKey;
import com.google.appengine.api.users.User;
@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class Article {
    @PrimaryKey    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Long id;
    @Persistent
    private User author;
    @Persistent
    private String title;
    @Persistent
    private String content;
    @Persistent
    private String tag;
```



```
public class CH {
    private static Cache cache;
    static {
        try {cache= CacheManager.getInstance().getCacheFactory().
            createCache(Collections.emptyMap());
        } catch (CacheException e) {}
    }
    public static Cache getCache(){
        return cache;
    }
    private CH(){
    }
}
```

登录信息提交到 userCheck 这个 servlet 中，在此 servlet 中对身份进行了验证，并把登录用户的用户名保存在 Cache 中，userCheck 的关键代码如下。

```
public class userCheck extends HttpServlet {
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        //获取用户名和密码
        String username=req.getParameter("username");
        String password=req.getParameter("password");
        //登录成功标志
        String flag2;
        //登录失败标志
        String flag3;
        //验证用户名和密码,UP.username 中保存了用户名，UP.pw 保存了密码，为静态常量
        if((username.equals(UP.username))&&(password.equals(UP.pw)))
        {   flag2="欢迎"+username+"登录";
            //将用户名，标志变量保存在 Cache 中
            CH.getCache().put("username",username);
            CH.getCache().put("flag", flag2);
        } else {
            flag3="登录失败";
            CH.getCache().put("flag", flag3);
        }
        resp.sendRedirect("index.jsp");
    }
}
```

注销操作就是把 Cache 用户名信息清空并跳转到首页，代码如下。

```
CH.getCache().clear();
resp.sendRedirect("index.jsp");
```


所有的业务逻辑的处理都放在 spaceService 这个服务类中。在 spaceService 中实现了对日志的增、删、改和查等操作，spaceService 中一些常用方法的关键代码如下。

```
public class spaceService {
    //获取 JDO 的 PersistenceManager, 静态的
    private static PersistenceManager pm=PMF.get().getPersistenceManager();
    //获取 google 账号的 nickName
    public static String getUser(){
        UserService US=UserServiceFactory.getUserService();
        User user=US.getCurrentUser();
        if (user!=null)
            return user.getNickname();
        else
            return "";
    }
    .....
    //获取所有保存在 Bigtable 中的日志对象
    public List<Article> articleListAll() {
        String query = "select from " + Article.class.getName();
        return (List<Article>) pm.newQuery(query).execute();
    }
    //获取日志的数目
    public int articleCount(){
        String query="select from "+Article.class.getName();
        List<Article> L2=(List<Article>)pm.newQuery(query).execute();
        int count=(L2==null ?0:L2.size());
        return count;
    }
    //获取 spaceService 对象
    public static spaceService newService(){
        return (new spaceService());
    }
}
```

在 spaceService 中反复用到了 PersistentManager 对象 pm，由于 pm 的实例化需要比较大的开销，所以 pm 也采用单件模式，即一个会话中只有一个 PersistentManager 对象，在 PMF 中获取 PersistentManagerFactory 对象，在 spaceService 中获取静态的 PersistenceManager，PMF 的代码如下。

```
public class PMF {
    private static final PersistenceManagerFactory
    pmfInstance=JDOHelper.getPersistenceManagerFactory("transactions-optional");
    private PMF(){}
}
```



```
public static PersistenceManagerFactory get(){
    return pmfInstance;
}
}
```

6) 读取日志

采用 JDO 的接口从底层的数据库中读取数据，在这里为了实现分页显示，必须为查找操作设定范围，在 `spaceService` 中实现读取操作的代码如下。

```
//获取指定范围的日志对象，在日志分页中会用到
public List<Article> articleList(long start,long end){
    Query qry=pm.newQuery(Article.class);
    qry.setRange(start, end);
    //按照更新日期排序
    qry.setOrdering("date descending");
    return(List<Article>) qry.execute();
}
```

在页面 ArticleList.jsp 中显示日志的标题, 更新日期, 内容, 和分页链接, 代码如下:

[illegible]


```
for(int i=0;i<pagecount;i++){
%><a href="ArticleList.jsp?start=<%=i%>"><%=i+1%>&nbsp;  </a><%=i%>
```

7) 删除日志

页面上每条日志记录下面会有一个删除链接,删除操作交由 articleEdit.jsp 处理并且提交删除日志的 id 参数,另外因为是和修改操作一起处理,所以附带一个 cmd 标志。删除操作首先是按主键查找到相应的 Article 对象,再删除,分别是由 spaceService 的 findArticleByKey 和 deleteArticle 方法来执行,代码如下。

```
//用主键查找日志
public Article findArticleByKey(long key){
String query = "select from " +Article.class.getName()+" where id==" +key;
Query qry=pm.newQuery(query);
List<Article> l1 =(List<Article>)qry.execute();
Article a1=l1.get(0);
qry.closeAll();
return a1;
}

//删除日志
public void deleteArticle(Article ar){
// 删除对象中的数据并提交到数据库
pm.currentTransaction().begin();
pm.deletePersistent(ar);
pm.currentTransaction().commit();
}
```

执行删除后返回日志列表页面, articleEdit.jsp 的关键代码如下:

```
<%//获取参数
String cmd=request.getParameter("cmd");
if(cmd!=null&& !cmd.isEmpty()){
String key=request.getParameter("key");
Long key1=Long.parseLong(request.getParameter("key"));
if("modify".equalsIgnoreCase(cmd)){
response.sendRedirect("writeArticle.jsp");
}

//查找相应的对象
Article ar=spaceService.newService().findArticleByKey(key1);
//修改
if("modify".equalsIgnoreCase(cmd)){
response.sendRedirect("writeArticle.jsp");
}

//删除
if("delete".equalsIgnoreCase(cmd)){
```



```
spaceService.newService().deleteArticle(ar);
response.sendRedirect("ArticleList.jsp");
}
}%>
```

8) 写日志

写日志页面主要包含一个表单，表单中的编辑器采用 CKeditor，表单提交到 saveArticle 这个 servlet 里处理，saveArticle 将调用 spaceService 的 saveArticle 方法来持久化对象，代码如下。

```
//保存日志
public void saveArticle(Article ar){
    try{
        pm.makePersistent(ar);
    }
    catch(Exception ex){
        ex.printStackTrace();
    }
}

saveArticle 的关键代码如下
public void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws IOException {
    //获取参数
    String cmd=req.getParameter("cmd");
    if(cmd!=null&& !cmd.isEmpty()){
        String title=req.getParameter("title");
        String content=req.getParameter("content");
        String tag=req.getParameter("tag");
        User user=UserServiceFactory.getUserService().getCurrentUser();
        Date date=new Date();
        //实例化对象
        Article article=new Article(user,title,content,tag,date);
        //持久化对象
        spaceService.newService().saveArticle(article);
    }
    resp.sendRedirect("ArticleList.jsp");
}
```

9) 修改日志

修改日志操作与保存日志操作区别不大，这里不再赘述。

10) 权限管理

在日志管理模块只有登录用户才有权限做修改、删除和写日志操作，游客只能浏览日

志, 所以在这里采用过滤器来实现权限管理, 限制游客执行需要管理员权限的操作, 过滤器 userFilter 的关键代码如下。

```
String username=(String)CH.getCache().get("username");
    if(username!=null)
    {
        if (username.equals(UP.username)){
            chain.doFilter(request, response);
        }
    }
    }else{
        CH.getCache().put("flag", "你没有权限访问!");
        response2.sendRedirect("index.jsp");
    }
}
```

在 war/WEB-INF/web.xml 中的配置如下

```
<filter>
    <filter-name>userFilter</filter-name>
    <filter-class>cn.edu.xtu.userFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>userFilter</filter-name>
    <url-pattern>/articleEdit.jsp</url-pattern>
</filter-mapping>
```

4. 项目部署

全部编码完成后, 先在本地测试, 本地测试通过后, 就可以将项目发布到 Google App Engine 上了, 发布方法如前所述, 本项目部署在 dengpenggae 应用上, 版本号设置为 2, 部署后通过 <http://dengpenggae.appspot.com> 访问 (国内有时无法直接访问), 发布后效果如图 2-51 所示。源代码的工程目录可在 <http://bbs.chinacloud.cn> 的教材板块下载。



图 2-51 发布后预览图

习题

1. Google 云计算技术包括哪些内容?
2. 当前主流分布式文件系统有哪些? 各有什么优缺点?
3. GFS 采用了哪些容错措施来确保整个系统的可靠性?
4. MapReduce 与传统的分布式程序设计相比有何优点?
5. Chubby 的设计目标是什么? Paxos 算法在 Chubby 中起什么作用?
6. 阐述 Bigtable 的数据模型和系统架构。
7. 分布式存储系统 Megastore 的核心技术是什么?
8. 大规模分布式系统的监控基础架构 Dapper 关键技术是什么?
9. Google App Engine 提供了哪些服务?
10. Google App Engine 的沙盒对开发人员进行哪些限制?

参考文献

- [1] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung. The Google File System, Proceedings of 19th ACM Symposium on Operating Systems Principles, 2003, 20-43
- [2] Sun "Lustre Networking: High-Performance Features and Flexible Support for a Wide Array of Networks" https://www.sun.com/offers/details/lustre_networking.xml
- [3] Soltis, Steven R; Erickson, Grant M; Preslan, Kenneth W (1997), "The Global File System: A File System for Shared Disk Storage", IEEE Transactions on Parallel and Distributed Systems
- [4] Schmuck, Frank; Roger Haskin (January 2002). "GPFS: A Shared-Disk File System for Large Computing Clusters". Proceedings of the FAST'02 Conference on File and Storage Technologies. Monterey, California, USA
- [5] Wikipedia. <http://zh.wikipedia.org/wiki/MapReduce>
- [6] John Darlington, Yi-ke Guo, Hing Wing To. Structured parallel programming: theory meets practice. Computing tomorrow: future research directions in computer science book contents Pages: 49-65
- [7] Jeffrey Dean, Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters
- [8] Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE. Bigtable: A distributed storage system for structured data. In: Proc. of the 7th USENIX Symp. on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2006. 205-218
- [9] Ghemawat S, Gobioff H, Leung ST. The Google file system. In: Proc. of the 19th ACM Symp. on Operating Systems Principles. New York: ACM Press, 2003. 29-43
- [10] Burrows M. The chubby lock service for loosely-coupled distributed systems. In: Proc. of the 7th USENIX Symp. on Operating Systems Design and Implementation. Berkeley:

- USENIX Association, 2006. 335-350
- [11] 陈康, 郑伟民. 云计算: 系统实例与研究现状. 软件学报, 2009, 20(5): 1337-1348
 - [12] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. CACM 13, 7 (1970), 422-426
 - [13] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In Proc. of the 7th OSDI, Nov, 2006
 - [14] LAMPORT, L. Paxos made simple. ACM SIGACT News 32, 4 (2001), 18-25
 - [15] Paxos 算法. 维基百科. <http://zh.wikipedia.org/zh-cn/Paxos%E7%AE%97%E6%B3%95>
 - [16] T.D.Chandra, R.Griesemer, and J.Redstone. Paxos made live: an engineering perspective. In PODC, 2007
 - [17] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for Interactive services. In Proc. CIDR, 2011
 - [18] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspán, Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Google Technical Report, 2010
 - [19] <http://code.google.com/intl/zh-CN/appengine/docs/>
 - [20] <http://q.sohu.com/forum/5/topic/21388686>

第3章 Amazon 云计算 AWS

Amazon（亚马逊）依靠电子商务逐步发展起来，凭借其在电子商务领域积累的大量基础性设施、先进的分布式计算技术和巨大的用户群体，Amazon 很早就进入了云计算领域，并在云计算、云存储等方面一直处于领先地位。在传统的云计算服务基础上，Amazon 不断进行技术创新，开发出了一系列新颖、实用的云计算服务。目前 Amazon 的云计算服务主要包括：弹性计算云 EC2^[13]、简单存储服务 S3^[15]、简单数据库服务 Simple DB^[16]、简单队列服务 SQS^[17]、弹性 MapReduce 服务^[19]、内容推送服务 CloudFront^[21]、电子商务服务 DevPay^[24]和 FPS^[25]等。这些服务涉及云计算的方方面面，用户完全可以根据自己的需要选取一个或多个 Amazon 云计算服务。所有的这些服务都是按需获取资源，具有极强的可扩展性和灵活性。

本章详细介绍 Amazon 平台基础存储架构 Dynamo^[1]及 Amazon 的主要云计算服务，重点剖析这些云计算服务背后涉及的重要技术、服务的基本架构和核心的概念。除了一些常用的 API 之外，本章将不介绍各个 Amazon 云计算服务的具体使用方法，感兴趣的读者可以参考 Amazon 的相关技术文档。

3.1 Amazon 平台基础存储架构：Dynamo

在 Web 服务刚兴起时，各种平台大多采用的是关系型数据库。由于大量的 Web 数据是半结构化数据，随着数据量的急剧增加，传统的关系型数据库已经无法满足这种存储要求，为此不少服务商都设计开发了自己的存储系统。Amazon 的 Dynamo 就是其中非常具有代表性的一种存储架构，作为状态管理组件被用于 Amazon 很多系统中。2007 年，Amazon 将其以论文形式发表，很快 Dynamo 就被应用于其他云存储架构，如 Twitter 和 Facebook 的 Cassandra^[39]架构和 NoSQL 数据库等。本节详细介绍具有高可用特性的 Dynamo 存储架构。

3.1.1 Dynamo 在 Amazon 服务平台的地位

Amazon 作为目前世界上最主要的电子商务提供商之一，它的系统每天要接受全球数以百万计的服务请求，高效的平台架构是保证其系统稳定性的根本。图 3-1^[1]是面向服务的 Amazon 平台基本架构。

从图中可以看出整个 Amazon 平台的架构是完全的分布式、去中心化的，在 Amazon 的平台中处于底层位置的存储架构 Dynamo 也是如此。Amazon 平台中有很多服务对存储的需求只是读取、写入，即满足简单的键/值（key/value）式存储即可，例如：常用的购物车、信息会话管理和推荐商品列表等，如果采取传统的关系数据库方式，则效率低下。针对这种需求，Dynamo 应运而生，虽然 Dynamo 目前并不直接向公众提供服务，但是大量的用户服务数据被存储在 Dynamo 中。可以说它为 Amazon 的电子商务平台及其云计算服务提供了最基础的支持。

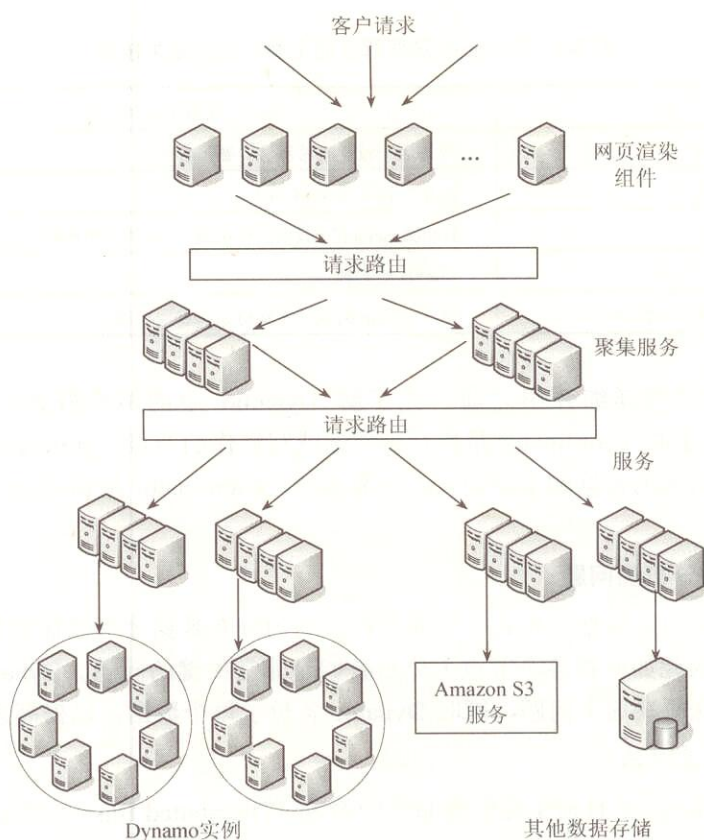


图 3-1 面向服务的 Amazon 平台架构

Dynamo 以很简单的键/值方式存储数据，不支持复杂的查询。但是这并不影响客户的使用，因为通常情况下用户只需要根据键读取值就足够了。Dynamo 中存储的是数据值的原始形式，也就是以位（bit）的形式存储，不解析数据的具体内容。Dynamo 不识别任何数据结构，这使得它几乎可以处理所有的数据类型。

目前，关于 Amazon 云服务的技术细节尚未公开，Dynamo 作为底层存储架构是如何支持这些服务的尚不清楚，但是，可以通过现有资料来分析其与各个云服务的关系。我们认为，从功能角度，Dynamo 存储使用各种云服务的用户数据；从实现角度，和存储有关的云服务与 Dynamo 关系密切，其中 S3 就是构建在 Dynamo 之上，SimpleDB 也极有可能使用或借鉴了 Dynamo 的技术。

3.1.2 Dynamo 架构的主要技术

相比传统的集中式存储系统，Dynamo 在设计之初就被定位为一个高可靠、高可用且具有良好容错性的系统。实践表明 Dynamo 是一种非常成功的分布式存储架构。表 3-1^[1]列出了 Dynamo 设计时面临的主要问题及最终采取的解决办法。

表 3-1 Dynamo 需要解决的主要问题及相关技术

问 题	采取的相关技术
数据均衡分布	改进的一致性哈希算法，数据备份
数据冲突处理	向量时钟（Vector Clock）
临时故障处理	Hinted handoff（数据回传机制），参数（W,R,N）可调的弱 quorum 机制
永久故障后的恢复	Merkle 哈希树
成员资格以及错误检测	基于 gossip 的成员资格协议和错误检测

在对表中内容作详细介绍之前，先了解 Dynamo 中的两个概念：coordinator^[1]和 preference list^[1]。其中 coordinator 是执行一次读或写操作的节点，preference list 是存储与某个特定键值相对应的数据的节点列表。一般来说，coordinator 是 preference list 上的第一个节点。

1. 数据均衡分布的问题

Dynamo 采取的是数据分布式存储的架构，均匀分布数据才可以保证负载平衡和系统良好的扩展性，因此如何在各个节点上分布数据是非常关键的问题。Dynamo 使用改进后的一致性哈希算法解决这个问题，同时 Dynamo 备份了每个数据，以提高系统的可用性。

1) 一致性哈希算法

一致性哈希算法^[2]是目前主流的分布式哈希表（Distributed Hash Table, DHT）协议之一，由麻省理工学院于 1997 年提出。一致性哈希算法通过修正简单哈希算法，解决了网络中的热点（Hot Pot）问题，使得 DHT 可以真正的应用于 P2P 环境中。

一致性哈希算法满足以下四个要求。

（1）平衡性：哈希算法运算后的结果能充分分散到整个缓冲空间，提高了缓冲空间的利用率。

（2）单调性：一致性哈希算法保证当加入新的缓冲区域时，旧的缓冲空间会被映射到新的空间中，而不会出现已分配的空间被映射到旧的缓冲集合的其他区域的情况。这主要是避免在新的节点加入时频繁改动原有映射关系所带来的巨大开销。

（3）分散性：在分布式环境中，不同的终端可能只看见整个缓冲区的某个部分，这样可能会导致对于同一数据，不同的终端将其映射到不同的缓冲区，这显然降低了数据存储的效率。一致性哈希算法就要求尽量避免和降低分散性。

（4）负载：既然不同的终端可以将相同的内容映射到不同的缓冲区，那么同一缓冲区也可能被不同的终端映射成不同的内容。一致性哈希算法可以有效避免这种情况的出现。

一致性哈希算法一般分两步进行。首先求出设备节点的哈希值，将设备配置到环上的一个点（环上的每个点代表一个哈希值）；接着计算数据的哈希值，按顺时针方向将其映射到环上距其最近的节点，如图 3-2 所示。添加新节点时，按照上述规则，调整相关数据到新的节点上，如图 3-3 所示。删除节点和添加节点过程相反。

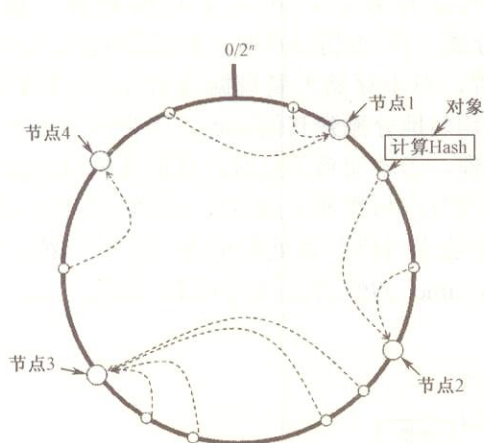


图 3-2 一致性哈希算法

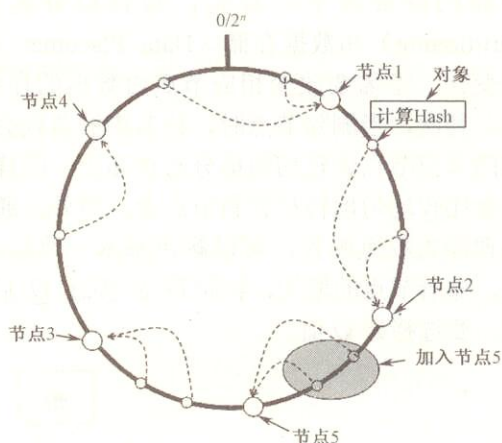


图 3-3 添加节点示意图

2) Dynamo 采用的改进算法

Dynamo 采用一致性哈希算法的主要原因是每个节点只需处理落在它和它的前驱节点之间的数据，这样增/删节点时系统振荡较小。一致性哈希函数是一种随机性的函数，在节点数量较少的情况下很可能造成环上节点分布的不均匀，导致负载不均匀；在选择节点位置时，并没有考虑环上不同节点的性能差异。为了解决这些问题，Amazon 在 Dynamo 中引入了虚拟节点^[1]的概念。每个虚拟节点都属于某一个实际的物理节点，一个物理节点根据其性能的差异可能拥有一个或多个虚拟节点。每个虚拟节点能力基本相当，并随机分布在哈希空间中。存储时，数据按照哈希值落到某个虚拟节点负责的区域，然后被存储在该虚拟节点所对应的物理节点中，如图 3-4 所示。

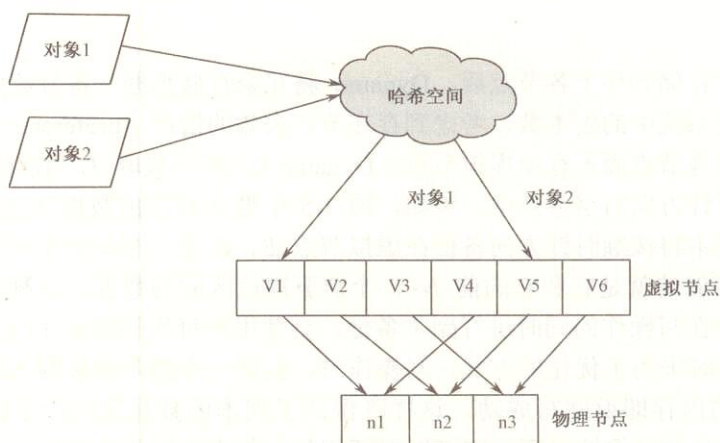


图 3-4 Dynamo 虚拟节点和物理节点

假设将一个值域为 2^n 的 Hash 环划分成 Q 个等份，每个等份称做一个数据分区 (Partition)^[1]，有 S 个虚拟节点，并且满足 $Q \gg S$ ，则每个虚拟节点对应的分区数 $V=Q/S$ ，将两个虚拟节点之间的所有分区称为一个键值区间，如图 3-5 所示。这种数据分区和等份

存储的好处在于：首先，数据以分区为单元进行存储，实现了数据划分（Data Partitioning）和数据存储（Data Placement）的分离。在增/删节点时，只是引起键值区间的变化，不需要改变相应节点内数据的存储位置，对于存储大量数据的系统是非常有效的。其次，当删除节点时，该节点的负载会比较均匀地分摊到其他节点，因为该节点所对应的虚拟节点比较均匀地分布在环上；同样的道理，当添加新节点时，其他虚拟节点的负载会比较均匀地转移到新节点上。另外，通过虚拟节点和物理节点多对一的配置可以实现处理能力权重配置，可以解决基本一致性哈希算法未考虑节点处理能力差异的问题。注意，随着节点的增加，特别是 S 接近 Q 后，Dynamo 的性能会急剧下降，因此在设计之初，要选择好 Q 值。

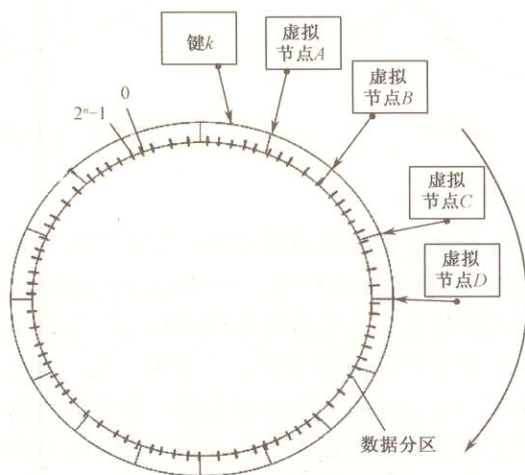


图 3-5 Dynamo 节点划分方式示意图

3) 数据备份

当数据被均匀存储到环上各节点后，Dynamo 将冗余存储数据（备份数据）。设 N 代表每份数据保存在系统中的副本数，考虑到存在节点失效的情况，preference list 大于 N 并且只存储实际的物理节点而不存储虚拟节点。Dynamo 中 N 一般取 3，节点的数据副本备份在环上它的顺时针方向后继节点中。例如，图 3-5 中键 k 对应的数据存储在虚拟节点 A 中，而它的数据副本将按顺时针方向备份在虚拟节点 B 、 C 上。根据这个规定，上面提到的键值区间实际存储的就是它和它的前 $N-1$ 个前驱键值区间的数据。这种设计提高了系统的可用性。由于在写操作的同时进行副本备份，会使用户每次的写操作时延变长，因此 Dynamo 在写操作时采用了优化的方式，写操作时，保证一个副本必须写入硬盘，其他副本只要写入节点的内存即返回写成功。这样既保证了副本的数量又减少了时延。实际上 Amazon 提供了更高的可靠性保证，它可以保证相邻的节点分别位于不同地区区域，即使某个数据中心由于自然灾害或断电的原因整体瘫痪，仍可以保证在世界上其他数据中心中保存有数据的备份。这里就有一个非常重要的问题，如何进行节点分布，保证相邻节点位于不同的数据中心，有兴趣的读者可以自己思考一下。

2. 数据冲突问题

分布式系统架构通常需要考虑三个因素：可靠性（Reliability）、可用性（Availability）

和一致性 (Consistency)。三者不能同时达到，最多只可以实现其中的两个。Dynamo 系统选择牺牲一致性来换取系统的可靠性和可用性，这也是由 Amazon 业务特点决定的。Dynamo 要保证完美的用户体验，就必须保证数据总是可写的，但是这样就可能出现数据冲突，如何解决数据冲突呢？Dynamo 采用了最终一致性模型 (Eventual Consistency)，这种模型和强一致性模型 (Strong Consistency) 的不同点在于：它并不在意数据更新过程中的一致性问题，只要最终的所有数据副本能够保证一致性即可。简单地讲就是“只求结果，不看过程”。由于最终一致性模型不保证过程中数据的一致性，在某些情况下（比如说某个存储节点出现故障）不同的数据副本可能会出现不同的版本，如何确保所有的副本最终都会被正确更新是一个很棘手的问题。数据副本可能会以不同的顺序看到更新结果，而不同顺序的更新很可能造成数据的不一致。为此 Dynamo 利用技术手段推断各个更新的实际发生次序，这种技术就是向量时钟^[3]，其原理图如图 3-6 所示。

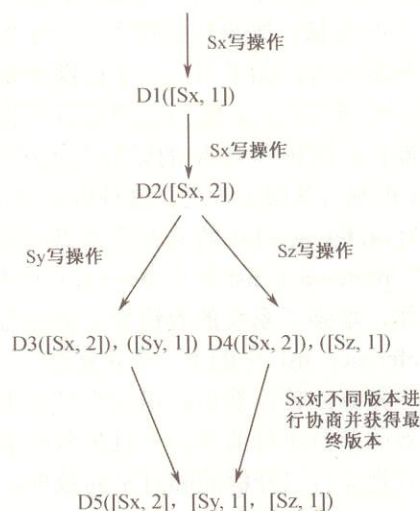


图 3-6 向量时钟原理图

Dynamo 中的向量时钟用一个 (nodes, counter) 对表示。其中 nodes 表示节点，counter 是一个计数器，初始为 0，节点每发生一次事件就将计数器加 1。首先，Sx 对某个对象进行一次写操作，产生一个对象版本 D1([Sx,1])，接着 Sx 再次操作，由于 Sx 是第二次进行操作，所以 counter 值更新为 2，产生第二个版本 D2([Sx,2])；之后，Sy 和 Sz 同时对该对象进行写操作，Sy 将自身的信息加入向量时钟产生了新的版本 D3([Sx,2], [Sy,1])，Sz 同样产生了新的版本信息 D4([Sx,2], [Sz,1])，这时系统中就有了两个版本的对象，但是系统不会自行选择，它会将这两个版本同时保存，等待客户端解决冲突。最后 Sx 再次对对象进行操作，这时它会同时获得两个数据版本，用户根据版本的信息，重新计算获得一个新的对象记做 D5([Sx,2], [Sy,1], [Sz,1])，并将新的对象保存到系统中。需要注意的是，向量时钟的数量是有限制的，当超过限制时需根据时间戳 (Timestamp) 删除最开始的一个。这种解决一致性的方式对 Amazon 的购物网站来说非常有用。例如购物车模型，用户可以通过它查询曾经浏览的各种商品，而不是仅仅查询最终购买的商品。

3. 容错机制

出于成本等方面的考虑，Dynamo 中很多服务器采用的是普通 PC 主机（无显示器、键盘）。普通 PC 硬盘的性能和专业的服务器硬盘的性能相比相距甚远，出错在所难免，因此容错机制在 Dynamo 中非常重要。机器故障包括临时故障和永久性故障，下面从这两方面分别介绍 Dynamo 容错机制。

1) 临时故障处理机制

Dynamo 临时故障处理机制主要体现在两方面，首先，在数据读写中采用了一种称为弱 quorum (Sloppy quorum) 的机制^[1]，涉及三个参数 W 、 R 、 N ，其中 W 代表一次成功的写操作至少需要写入的副本数， R 代表一次成功读操作需由服务器返回给用户的最小副本数， N 是每个数据存储的副本数。Dynamo 要求 $R+W>N$ ，满足这个要求，保证用户读取数据时，始终可以获得一个最新的数据版本。用户可以自行配置 R 和 W ，调节可用性和容错性之间的平衡。如果将 R 、 W 配低，则系统的可用性好于容错性，反之提高系统的容错性会降低可用性。从用户的角度来看，如果用户对于读操作要求较高，可以将 R 设置为 1， W 设置成 N ，保证只要有一个节点可用，就可以进行读操作，反之则可以保证在只有一个节点情况下也可以进行写操作。这种固定 N 的设置方式和传统的 quorum 机制类似，但是 Dynamo 采用的弱 quorum 机制对其做了改进。这种机制规定系统对每个数据的存储量仍为 N ，但此时的 N 并不限定为 preference list 的前 N 个节点，而是将 N 限制为前 N 个正常节点。巧妙之处在于，一方面当 preference list 中前 N 个节点中有出现临时故障的，则会由后面的正常节点接替其工作，增强了系统的容错性。另一方面，一旦某个节点出现问题，则将这个节点值传送给 preference list 中的下一个正常节点，并在这个数据副本的元数据中记录失效的节点位置，便于数据回传；然后，由这个节点上一个临时空间进行存储和处理数据，同时该节点还对失效的节点进行监测，一旦失效的节点重新可用，则将自己所保存的最新数据回传给它，然后删除自己开辟的临时空间数据。简单地说这是一种带有监听的数据回传机制 (Hinted Handoff)^[1]。图 3-7 是 Dynamo 临时故障处理机制示意图。

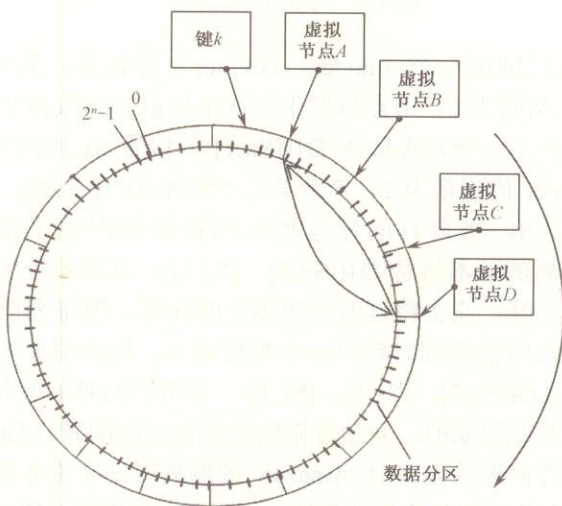


图 3-7 Dynamo 临时故障处理机制

2) 永久性故障处理机制

节点出现永久性故障时, Dynamo 必须检查和保持数据的同步, Dynamo 采用一种称为反熵协议^[1] (Anti-entropy Protocol) 的手段来保证数据的同步。为了减少数据同步检测中需要传输的数据量, 加快检测速度, Dynamo 使用了 Merkle 哈希树技术^[1], 每个虚拟节点保存三颗 Merkle 树, 即每个键值区间建立一个 Merkle 树。Dynamo 中 Merkle 哈希树的叶子节点是存储每个数据分区内所有数据对应的哈希值, 父节点是其所有子节点的哈希值。图 3-8 是两棵不同的 Merkle 哈希树 A 和 B。

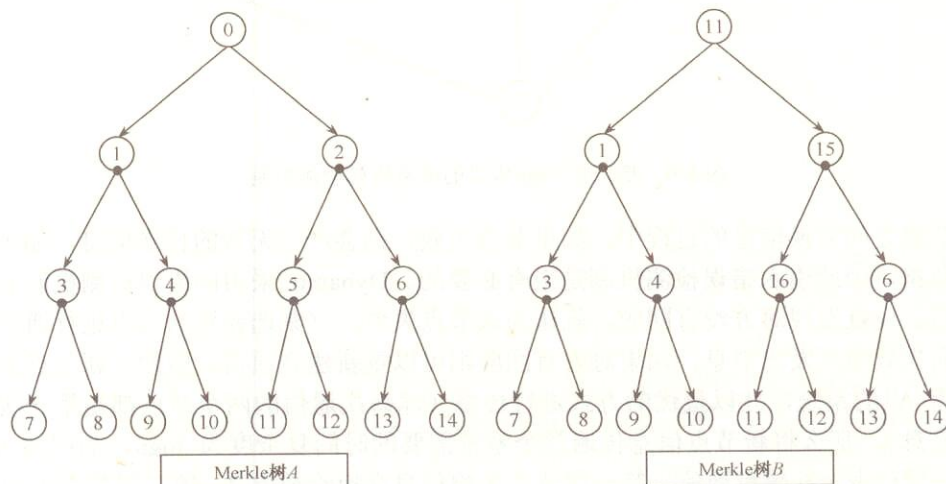


图 3-8 Merkle 哈希树

系统比较两棵同一键值区的 Merkle 哈希树时, 首先查看根节点, 如果相同则说明数据一致, 不需要进行数据同步, 否则需要继续比较, 直到哈希值不同的叶子节点, 快速定位差异。例如, 图 3-8 中 A 和 B 的根节点不同, 说明需要进行数据同步。紧接着比较 A 和 B 的子节点, 发现右子树的根节点 $2 \neq 15$, 继续比较右子树根节点的子节点, 按同样的步骤一直进行下去, 发现需要同步的数据位置。Merkle 树最大特点是只要比较某个子树就可以完成数据同步检测和定位, 进而进行同步, 大大减少了同步过程中所需传输数据量, 提高了系统效率。

4. 成员资格及错误检测

Dynamo 中的每个节点就是 Dynamo 的一个成员, Amazon 为了使系统间数据的转发更加迅速 (减少数据传送时延、增加响应速度), 规定每个成员节点都要保存其他节点的路由信息。由于机器或人为的因素, 系统中成员的加入或撤离时常发生, 为了保证每个节点保存的都是 Dynamo 中最新的成员信息, 所有节点每隔固定时间 (1 秒) 利用一种类似于 Gossip (闲聊) 协议^[1] 的方式从其他节点中任意选择一个与之通信的节点。如果连接成功, 双方交换各自保存的信息 (包括存储数据情况、路由信息)。为了避免新加入的节点之间不能及时发现对方的存在, Dynamo 中设置了一些种子节点 (Seed Node)。种子节点和所有的节点都有联系, 当新节点加入时, 它扮演一个中介的角色, 使新加入节点之间互相感知。这种基于 Gossip 协议的成员资格检测机制如图 3-9 所示。

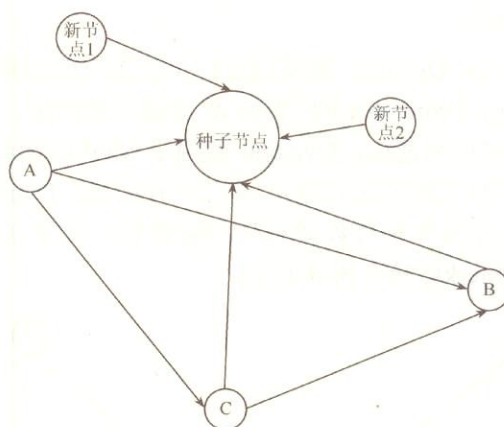


图 3-9 基于 Gossip 协议的成员资格检测机制

在节点之间交换信息的过程中，如果节点失效，则会产生无效的传送信息，加重系统的传输负担，因此引入错误检测机制是很有必要的。Dynamo 采用的错误检测机制非常简单、实用。一旦发现对方没有回应，就认为该节点失效，立刻选择别的节点进行通信。同时定期向失效节点发出消息，如果对方有回应则可以重新建立通信。假如一新节点加入节点总数为 N 的系统，并以最优的方式进行传播（即每次通信的两个节点都是第一次交换新节点信息），那么将新节点信息传遍整个系统需要的时间复杂度为 $\log n$ ，如图 3-10 所示。每一层代表一次随机通信，第一层节点 1 将信息交换给节点 2；第二层节点 1 和 2 同时开始随机选择其他节点交换信息，比如节点 1 向节点 3 发送信息，节点 2 向节点 4 发送信息；依此类推直到全部 N 个节点全部传遍，整个过程形成一个倒的二叉树，树高为 $\log n$ 。很明显当 N 很大时，时间复杂度会变得很大，所以 Dynamo 的节点数不能太多。根据 Amazon 的实际经验，当节点数在数千时，Dynamo 的效率是非常高的，但当节点数增加到数万后，效率就会急剧下降。如何解决这个问题呢，Amazon 给出了分层 Dynamo 结构，有兴趣的读者可以进一步阅读参考文献[1]和[40]。

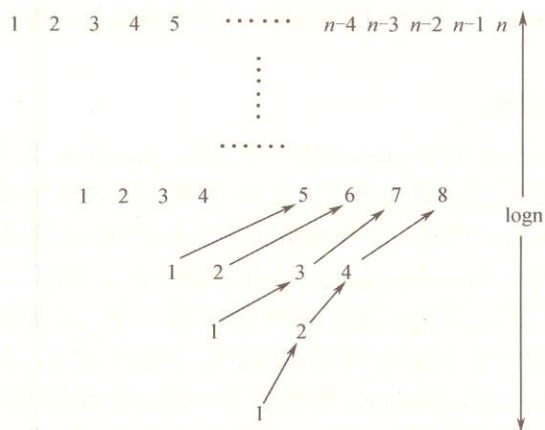


图 3-10 基于 Gossip 协议的最优传遍路径

3.2 弹性计算云 EC2

Amazon 弹性计算云服务 (Elastic Compute Cloud, EC2) 是 Amazon 云计算环境的基本平台。本节主要介绍 EC2 体系的基本架构, 侧重介绍其中涉及的一些基本概念, 最后简单介绍服务中经常使用的一些命令。

3.2.1 EC2 的主要特性

EC2 向用户提供了下面列举的一些非常有价值的特性^{[8][10]}。

(1) 灵活性: EC2 允许用户对运行的实例类型、数量自行配置, 还可以选择实例运行的地理位置, 可以根据用户的需求随时改变实例的使用数量。

(2) 低成本: EC2 使得企业不必为暂时的业务增长而购买额外的服务器等设备。EC2 的服务都是按小时来收费的, 而且价格非常合理。

(3) 安全性: EC2 向用户提供了一整套安全措施, 包括基于密钥对机制的 SSH 方式访问、可配置的防火墙机制等, 同时允许用户对它的应用程序进行监控。

(4) 易用性: 用户可以根据 Amazon 提供的模块自由构建自己的应用程序, 同时 EC2 还会对用户的服务请求自动进行负载平衡。

(5) 容错性: 利用系统提供的诸如弹性 IP 地址之类的机制, 在故障发生时 EC2 能最大程度地保证用户服务仍能维持在稳定的水平。

3.2.2 EC2 基本架构及主要概念

EC2 的基本架构如图 3-11 所示。

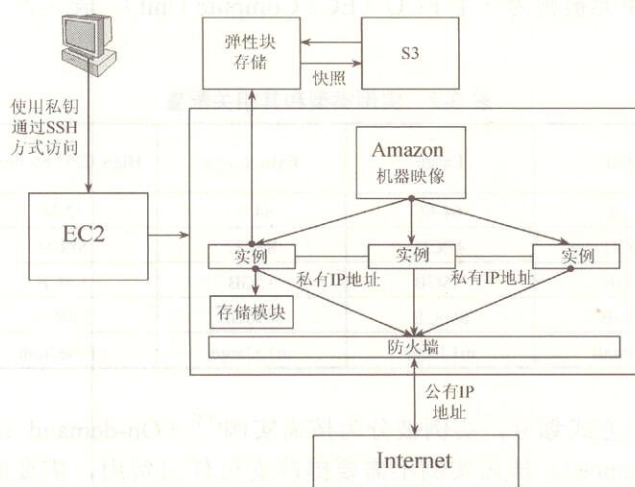


图 3-11 EC2 的基本架构

1. Amazon 机器映像 (AMI)

AMI^[13] (Amazon Machine Image) 是 Amazon 机器映像, 它是一个可以将用户的应用

程序、配置等一起打包的加密机器映像。AMI 是用户云计算平台运行的基础，所以，用户使用 EC2 服务的第一步就是要创建一个自己的 AMI，这和使用 PC 首先需要操作系统的道理相同。AMI 存储在 S3（在 3.3 节中介绍）中，目前 Amazon 提供的 AMI 有以下四种类型。

- （1）公共 AMI：由 Amazon 提供，可免费使用。
- （2）私有 AMI：用户本人和其授权的用户可以进入。
- （3）付费 AMI：向开发者付费购买的 AMI。
- （4）共享 AMI：开发者之间相互共享的一些 AMI。

初次使用 EC2 时，用户可以以 Amazon 提供的 AMI 为基础创建自己的服务器平台，也可以用 EC2 社区提供的脚本来创建新的 AMI（对用户的要求比较高），一般来说，使用 Amazon 提供的 AMI 即可。选好 AMI 后，将 AMI 打包（压缩），加密并分割上传，最后再使用相关的命令将 AMI 恢复即可。

2. 实例（Instance）

用户创建好 AMI 后，实际运行的系统称为一个实例，实例和我们平时用的主机很像。EC2 服务的计算能力是由实例提供的。按照 Amazon 目前的规定，每个用户最多可以拥有 20 个实例。每个实例自身携带一个存储模块（Instance Store），临时存放用户数据。当用户实例重启时，它其中的内容还会存在，但如果出现故障或实例被终止，存储在其中的数据将全部消失。因此，Amazon 建议把重要数据保存在 EBS 中（下文介绍这种方法）。按照计算能力划分，实例被分成标准型^[13]和高 CPU 型^[13]。标准型实例的 CPU 和内存是按一定比例配置的，对于大多数的应用来说已经足够了。如果用户对于计算能力的要求比较高，可以选择高 CPU 型的实例，这种实例的 CPU 资源比内存资源要高。为了屏蔽底层硬件的差异，准确地度量用户实际使用的计算资源，EC2 定义了所谓的 CPU 计算单元。一个 EC2 计算单元被称为一个 ECU（EC2 Compute Unit）。表 3-2^[4]是目前可选的实例类型和其相关配置。

表 3-2 实例类型和其相关配置

资 源	Small	Large	Extra Large	High-CPU Medium	High-CPU Extra Large
平台	32 位	64 位	64 位	32 位	64 位
CPU	1ECU	4ECU	8ECU	5ECU	20ECU
内存	1.7GB	7.5GB	15GB	1.7GB	7GB
存储容量	160GB	850GB	1690GB	350GB	1690GB
实例类型名	m1.small	m1.large	m1.xlarge	c1.medium	c1.xlarge

如果按照使用的方式划分，实例被分为按需实例^[13]（On-demand Instance）和预留实例^[13]（Reserved Instance）。按需实例不需要提前支付任何费用，需要时直接使用，用完后按小时付费即可。如果需要使用预留实例的话，则必须首先对所需实例做出预定，并为这些实例预先支付一定的费用。需要注意的是，实际使用的实例必须和预留的实例是同一类型且在同一个可用区域，只有这样才可以享受到相应的优惠。可用区域的概念将在后面介绍。

3.2.3 EC2 的关键技术

1. 弹性块存储 (EBS)

对于需要长期保存或比较重要的数据，需要用弹性块存储 (Elastic Block Store, EBS)^[13]。和 S3 不同，它是专门为 EC2 设计的，可以更好地和 EC2 配合使用。EBS 允许用户创建卷 (Volume)，卷的功能和平常使用的移动硬盘非常类似。Amazon 限制每个 EBS 最多创建 20 个卷，每一个卷可以作为一个设备挂载 (Mounted as a Device) 在任何实例上。挂载以后就可以像使用 EC2 的一个固有模块一样来使用它，这点和 S3 是完全不同的。快照 (Snapshot) 是 EBS 提供的一个非常实用的功能，可以捕捉当前卷的状态，然后数据就可以被存储在 S3 中。对于习惯使用 S3 的用户来说这是一个很方便的功能，快照的另一个功能是用来作为创建一个新卷的起始点。

2. 地理区域和可用区域

区域^[13] (Zone) 是 EC2 中独有的概念。Amazon 将区域分为两种：地理区域 (Region Zone) 和可用区域 (Availability Zone)。其中地理区域是按照实际的地理位置划分的，目前 Amazon 在全世界有五个地理区域：美国东 (北弗吉尼亚州)、美国西 (北加州)、欧盟 (爱尔兰) 和亚太地区 (新加坡、东京)。而可用区域的划分则是根据是否有独立的供电系统和冷却系统等，这样某个可用区域的供电或冷却系统错误就不会影响到其他可用区域，一般情况下人们把一个数据中心看做一个可用区域。

从图 3-12 可以很明显地看出两者关系。EC2 系统中包含多个地理区域，而每个地理区域中又包含多个可用区域。为了确保系统的稳定性，用户最好将自己的多个实例分布在不同的可用区域和地理区域中。这样在某个区域出现问题时可以用别的实例代替，最大程度地保证了用户利益。

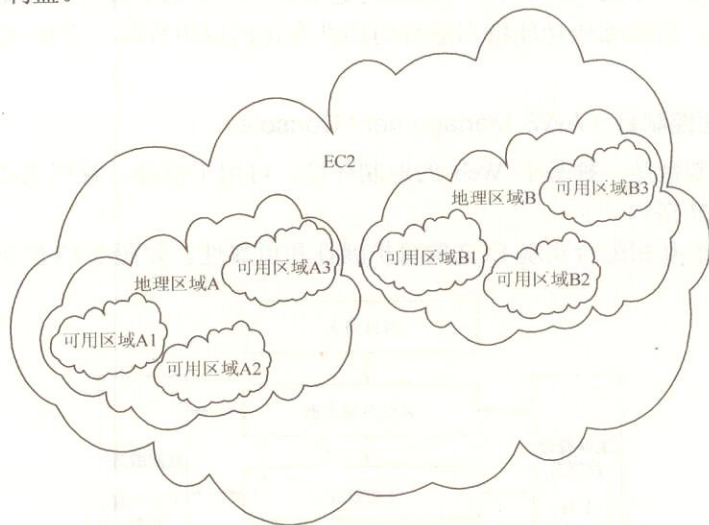


图 3-12 EC2 中区域间关系

3. EC2 的通信机制

在 EC2 服务中，系统各模块之间及系统和外界之间的信息交互是通过 IP 地址进行

的。EC2 中的 IP 地址包括三大类：公共 IP 地址^[13]（Public IP Address）、私有 IP 地址^[13]（Private IP Address）及弹性 IP 地址^[13]（Elastic IP Address）。EC2 的实例一旦被创建就会动态地分配两个 IP 地址，即公共 IP 地址和私有 IP 地址。公共 IP 地址和私有 IP 地址之间通过网络地址转换（Network Address Translation, NAT）技术实现相互之间的转换。公共 IP 地址和特定的实例相对应，在某个实例终结或被弹性 IP 地址替代之前，公共 IP 地址会一直存在，实例通过这个公共 IP 地址和外界进行通信。私有 IP 地址也和某个特定的实例相对应，它由动态主机配置协议（DHCP）分配产生。私有 IP 用于实例之间的通信流程如图 3-11 所示。

4. 弹性负载均衡（Elastic Load Balancing）

弹性负载均衡功能允许 EC2 实例自动分发应用流量，从而保证工作负载不会超过现有能力，并且在一定程度上支持容错。弹性负载均衡功能可以识别出应用实例的状态，当一个应用运行不佳时，它会自动将流量路由到状态较好的实例资源上，直到前者恢复正常才会重新分配流量到其实例上。

5. 监控服务（CloudWatch）

Amazon CloudWatch 是一个 Web 服务，提供了 AWS 资源的可视化检测功能，包括 EC2 实例状态、资源利用率、需求状况、CPU 利用率、磁盘读取、写入和网络流量等指标。使用 CloudWatch 时，用户只需选择 EC2 实例，设定监视时间，CloudWatch 就可以自动收集和存储检测数据。之后，用户可以通过 AWS 服务管理控制台或命令行工具来维护和这些检测数据。

6. 自动缩放（AutoScaling）

自动缩放可以按照用户自定义的条件，自动调整 EC2 的计算能力。在需求高峰期时，该功能可以确保 EC2 实例的处理能力无缝增大；在需求下降时，自动缩小 EC2 实例规模以降低成本。自动缩放功能特别适合周期性变化的应用程序，它由 CloudWatch 自动启动。

7. 服务管理控制台（AWS Management Console）

服务管理控制台是一种基于 Web 的控制环境，可用于启动、管理 EC2 实例和提供各种管理工具和 API 接口。

各个技术通过互相配合实现 EC2 的可扩展性和可靠性。如图 3-13 所示。

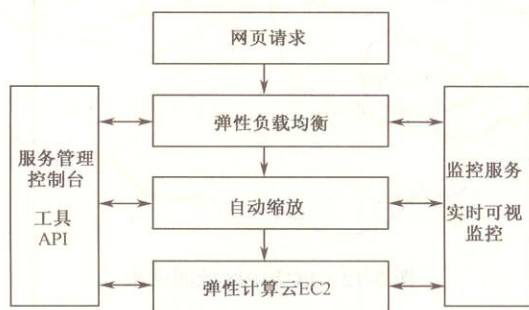


图 3-13 各关键技术的配合工作图

3.3.4 EC2 安全及容错机制

对网络传输中的数据进行控制的一个非常有效的办法是配置防火墙，但是传统的防火墙的规则是建立在 IP 地址、子网范围等基础之上的。EC2 的特点之一就是允许用户随时更新实例状态，用户可以随时加入或删除实例，实例状态的动态变化方便了用户，但是却给防火墙的配置带来了麻烦，为此 EC2 引入了安全组^[13]（Security Group）的概念。安全组其实就是一组规则，用户利用这些规则来决定哪些网络流量会被实例接受，其他则全部拒绝。一个用户目前最多可以创建 100 个安全组。当用户的实例被创建时，如果没有指定安全组，则系统自动将该实例分配给一个默认组（Default Group）。默认组只接受组内成员的消息，拒绝其他任何消息。当一个组的规则改变后，改变的规则自动适用于组中所有的成员。

用户在访问 EC2 时需要使用 SSH（Secure Shell）密钥对^[13]（Key Pair）来登录服务。SSH 是目前对网络上传输的数据进行加密的一种很可靠的协议，当用户创建一个密钥对时，密钥对的名称（Key Pair Name）和公钥（Public Key）会被储存在 EC2 中。在用户创建新的实例时，EC2 会将它保存的信息复制一份放在实例的元数据（Metadata）中，然后用户使用自己保存的私钥（Private Key）就可以安全地登录 EC2 并使用相关服务了。这一过程如图 3-14^[13]所示。

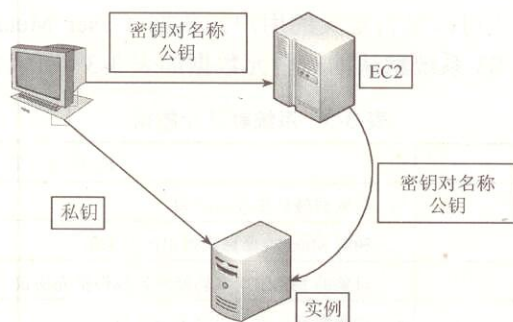


图 3-14 用户使用密钥对登录服务

在 EC2 的容错机制中，使用弹性 IP 地址是非常有效的一种方法。在创建实例时，系统会分配一个公共 IP 和一个私有 IP 给实例。用户是通过 Internet 利用公共 IP 地址访问实例的，但每次启动实例时这个公共 IP 地址都会发生变化。而 DNS 解析器中的 IP 地址和 DNS 名称的映射关系的更新大概需要 24 小时，为了解决这个问题，EC2 引入了弹性 IP 地址的概念。弹性 IP 地址和用户账号绑定而不是和某个特定的实例绑定，这给系统的容错带来极大的方便，每个账号默认绑定 5 个弹性 IP 地址。当系统正在使用的实例出现故障时，用户只需要将弹性 IP 地址通过网络地址转换技术转换为新实例所对应的私有 IP 地址，这样就将弹性 IP 地址与新的实例关联起来，访问服务时不会感觉到任何差异。这也是前面为什么建议在不同的区域建立实例的原因，当某一区域出现问题时可以直接用其他区域的实例来代替。因为所有区域的实例都出现故障的概率几乎为零，所以通过弹性 IP 地址改变映射关系总可以保证有实例可用。

3.3 简单存储服务 S3

S3 (Simple Storage Services) 是 Amazon 推出的简单存储服务, 用户通过 Amazon 提供的服务接口可以将任意类型的文件临时或永久地存储在 S3 服务器上, S3 的总体设计目标是可靠、易用及很低的使用成本^[9]。

3.3.1 基本概念和操作

S3 系统是构架在 Dynamo 之上的, 它采取的并不是传统的关系数据库存储方式。这么做主要有两个原因: 一方面是为了使文件操作尽量简单、高效; 另一方面对于一个普通的用户来说最常用的操作是存储和读取数据, 传统的关系数据库最擅长的查询在此无用武之地, 使用关系数据库只会增加系统的复杂性。S3 存储系统中涉及三个基本概念: 对象 (Object)、键 (Key) 和桶 (Bucket)。

1. 对象

对象^[15]是 S3 的基本存储单元, 主要由两部分组成: 数据和元数据。数据可以是任意类型; 元数据是用来描述数据的数据, 它一般和具体数据相关联, 并不单独存在。S3 中元数据存储的是对象数据内容的附加描述信息, 这些信息可以是系统默认定义的系统元数据 (System Metadata), 也可以是自定义的用户元数据 (User Metadata), 其中用户元数据的大小不得超过 2048B。S3 系统默认的一些元数据如表 3-3^[5]所示。

表 3-3 系统默认元数据

元数据名称	名称含义
last-modified	对象被最后修改的时间
ETag	利用 MD5 哈希算法得出的对象值
Content-Type	对象的 MIME (多功能网际邮件扩充协议) 类型, 默认为二进制/八位组
Content-Length	对象数据长度, 以字节为单位

元数据是通过一对键-值 (Name-Value) 集合来定义的。S3 中元数据的处理由用户自己完成, 系统并不干预。Amazon 对于对象存储的内容没有限制, 但每个对象最大容量目前被限制在 5GB, 且在使用 UTF-8 编码时对象名称不能超过 1024B。重命名操作在对象中无效, 对象数据的实际存储方式对于用户来说是不透明的, 一旦用户对象被创建并添加数据, 就无法对数据的某一子部分直接进行修改, 间接的修改办法是重新创建对象并向其中添加新的数据。

2. 键

键^[15]是对象的唯一标示符。如同每个人都有一个身份证号一样, 每个对象必须指定一个键, 否则该对象无意义。

3. 桶

顾名思义, 桶^[15]是一个用来存储对象的容器。桶的作用类似于我们的文件夹, 对象是存储在桶中的。Amazon 目前对于每个用户限制最多创建 100 个桶, 但是并不限制每个桶

中对象的数量。桶不可以被嵌套，也就是桶中不能创建桶。桶的名称必须在整个 Amazon 的 S3 服务器中是全局唯一的，这是因为 S3 中文件可以被共享，如果桶名不是全局唯一则会出现类似 IP 冲突的情况，所以桶在命名前最好使用相关命令来查看该名是否已被使用。具体的命名规则^[15]如下。

(1) 可以包含小写字母、数字、句号 (.)、下划线 (_)、破折号 (-)。

(2) 必须以字母或数字开头。

(3) 名称长度为 3~255 个字符。

(4) 不能使用类似 IP 地址的格式 (例如, 210.45.212.1)。

为了符合域名解析器 (DNS) 的要求, 建议使用以下规则。

(1) 名称中不要包含下画线。

(2) 名称长度为 3~63 个字符。

(3) 名称不要使用破折号结尾。

(4) 两个句号不能在一起使用。

(5) 名称中破折号和句号不能一起使用 (例如, chinacloud.com 和 chinacloud.com 都是不合法的名称)。

建议读者使用符合 DNS 要求的命名规则, 这样一旦需要使用 CloudFront 等其他 Amazon 的云计算服务时就不需要担心名称问题了。当桶和键都已经确定后, 对象也就被唯一确定了。用户可以对磁盘上文件的查询, 当要找的文件所在的文件夹及文件名都确定后, 文件自然就被找到了。但必须指出的是 S3 默认并不是普通 PC 上的那种分层式树状文件存储结构, 开发者可以根据需要自己定义。S3 中除了以上三个基本部分外, 每个对象还有一个访问控制模块, 具体内容将在 S3 安全措施部分进行介绍。S3 的基本结构如图 3-15 所示。

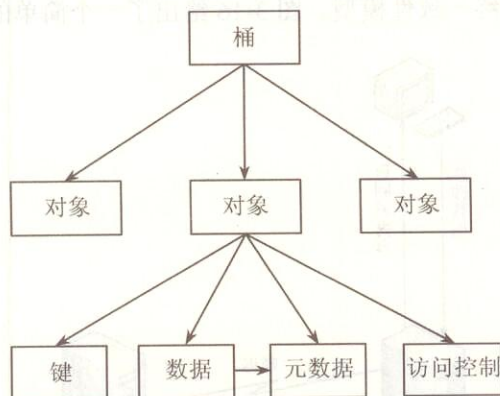


图 3-15 S3 的基本结构图

4. 基本操作

根据 Amazon 提供的技术文档, 目前 S3 支持的主要操作包括: Get、Put、List、Delete 和 Head。表 3-4^[5]列出了五种操作的主要内容。

表 3-4 S3 主要操作

操作目标	Get	Put	List	Delete	Head
桶	获取桶中对象	创建或更新桶	列出桶中所有键	删除桶	无
对象	获取对象数据和元数据	创建或更新对象	无	删除对象	获取对象元数据

3.3.2 数据一致性模型

为了保证用户数据信息的安全性，Amazon 在 S3 系统中采用了冗余存储的方式，也就是对于每个用户数据都产生多个副本，并将这些副本保存在不同的服务器上。这样做的好处是在某些服务器出现故障时用户仍然可以对其数据进行操作。但是这也有不可避免的弊端，用户在操作时可能会出现如下几种情况^[15]。

(1) 一个进程写入一个新的对象并立即尝试读取它，但在该改变被传送到 S3 的多个服务器前，服务器对该操作可能返回“键不存在”。

(2) 一个进程写入一个新的对象并立即尝试列出桶中已有的对象，但在该改变被传送到 S3 的多个服务器前，该对象很可能不会出现在列表中。

(3) 一个进程用新数据替换现有的对象并立即尝试读取它，但在该改变被传送到 S3 的多个服务器前，S3 可能会返回以前的数据。

(4) 一个进程删除现有的对象并立即尝试读取它，但在该改变被传送到 S3 的多个服务器前，S3 可能会返回被删除的数据。

(5) 一个进程删除现有的对象并立即尝试列出桶中的所有对象，但在该改变被传送到 S3 的多个服务器前，S3 可能会列出被删除的对象。

出现这些现象是因为 S3 为了保证用户数据的一致性而采取的一种折中手段，即在数据被充分传播到所有的存放节点之前返回给用户的仍是原数据，这其实也就是前面提到的 Dynamo 架构中采用的最终一致性模型。图 3-16 给出了一个简单的示意图。

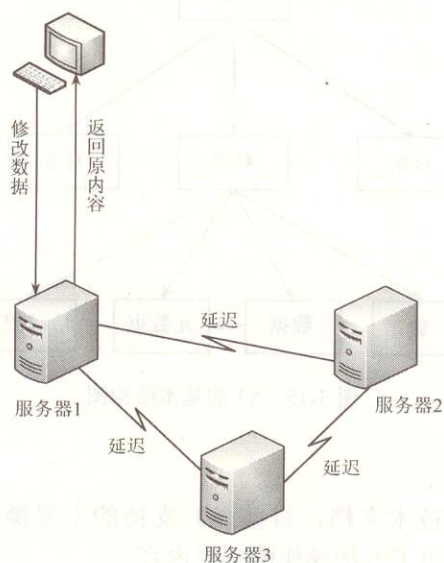


图 3-16 S3 数据一致性模型示意图

和前面提到的 Dynamo 架构不同,当用户对同一桶的同一对象先后进行两次不同的操作时,后一次操作值会直接覆盖前一次操作。比如,对于名为 chinacloud 的桶中的对象 A 先设定一个值为 100,后来又设定其值为 150,则 A 的最后值为 150。这种方法看似简单却很有效。

3.3.3 S3 安全措施

对于用户尤其是商业用户来说,系统的易用性是其考虑的一方面,但最终决定其是否使用 S3 服务的通常是 S3 的安全程度。S3 向用户提供包括身份认证^[15](Authentication)和访问控制列表^[15](ACL)的双重安全机制。

1. 身份认证

我们生活的世界是一个现实的世界,每个人都有其独特的身份,几乎不可能被假冒。但网络是一个完全虚拟的世界,随着网络犯罪的增加,如何在虚拟环境下对用户的身份做出准确、快速的识别已经成为一个重要的研究课题。

S3 中使用基于 HMAC-SHA1 的数字签名方式来确定用户身份。HMAC-SHA1 是一种安全的基于加密 Hash 函数和共享密钥的消息认证协议,它可以有效地防止数据在传输过程中被截获和篡改,维护了数据的完整性、可靠性和安全性。HMAC-SHA1 消息认证机制的成功在于一个加密的 Hash 函数、一个加密的随机密钥和一个安全的密钥交换机制。在新用户注册时,Amazon 会给每个用户分配一个 Access Key ID 和一个 Secret Access Key。Access Key ID 是一个 20 位的由字母和数字组成的串,Secret Access Key 是一个 40 位的字符串。Access Key ID 用来确定服务请求的发送者,而 Secret Access Key 则参与数字签名过程,用来证明用户是发送服务请求的账户的合法拥有者。S3 数字签名具体实现过程如图 3-17 所示。

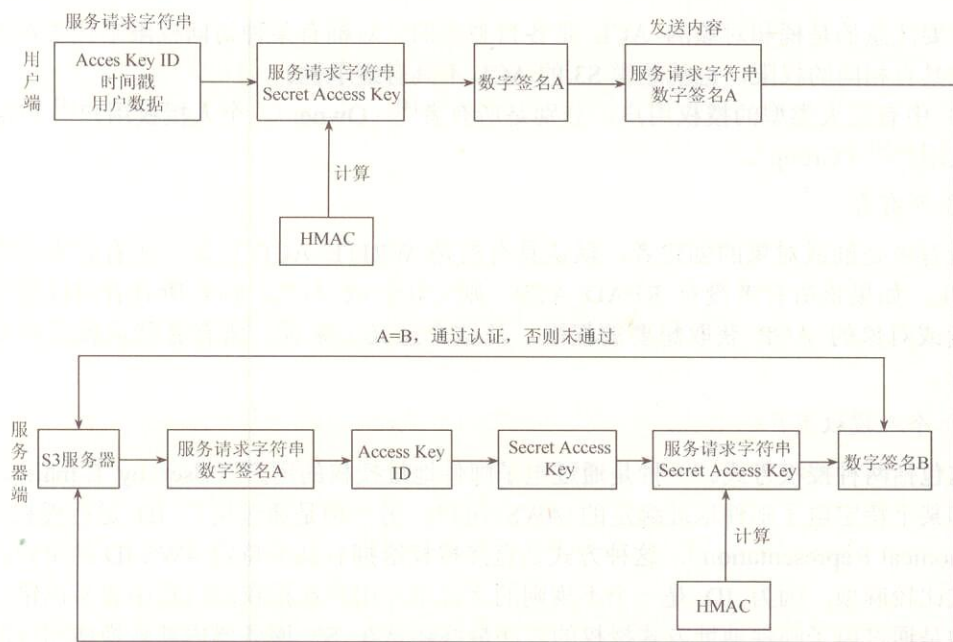


图 3-17 S3 数字签名具体实现过程

S3 用户首先发出服务请求，系统会自动生成一个服务请求字符串。HMAC 函数的主要功能是计算用户的服务请求字符串和 Secret Access Key 生成的数字签名，并将这个签名和服务请求字符串一起传给 S3 服务器。当服务器接收到信息后会从中分离出用户的 Access Key ID，通过查询 S3 数据库得到用户的 Secret Access Key。利用和上面相同的过程生成一个数字签名，然后和用户发送的数字签名做比对，相同则通过验证，反之拒绝。

2. 访问控制列表

访问控制列表（Access Control List, ACL）是 S3 提供的可供用户自行定义的访问控制策略列表。很多时候用户希望将自己的文件和别人共享但又不想未经授权的用户进入，此时可以根据需要设置合适的访问控制列表。S3 的访问控制策略（Access Control Policy, ACP）提供如表 3-5 所示的五种访问权限。

表 3-5 S3 的访问控制策略

权 限	允许操作目标	具体权限内容
READ	桶	列出已有桶
	对象	读取数据及元数据
WRITE	桶	创建、覆写、删除桶中对象
READ_ACP	桶	读取桶的 ACL
	对象	读取对象中的 ACL
WRITE_ACP	桶	覆写桶的 ACP
	对象	覆写对象的 ACP
FULL_CONTROL	桶	允许进行以上所有操作，是 S3 提供的最高权限
	对象	

需要注意的是桶和对象的 ACL 是各自独立的，对桶有某种访问权限不代表对桶中的对象也具有相同的权限，也就是说 S3 的 ACL 不具有继承性。

S3 中有三大类型的授权用户，分别是所有者^[4]（Owner）、个人授权用户^[4]（User）、组授权用户^[4]（Group）。

1) 所有者

所有者是桶或对象的创建者，默认具有的是 WRITE_ACP 权限。所有者本身也要服从 ACL，如果该所有者没有 READ_ACP，则无法读取 ACL。但是所有者可以通过覆写相应桶或对象的 ACP 获取想要的权限，从这个意义上来说，所有者默认就是最高权限拥有者。

2) 个人授权用户

这包括两种授权方式。一种是通过电子邮件地址授权的用户（User by E-mail），即授权给和某个特定电子邮件地址绑定的 AWS 用户；另一种是通过用户 ID 进行授权（User by Canonical Representation），这种方式是直接授权给拥有某个特定 AWS ID 的用户。后一种方式比较麻烦，因为 ID 是一个不规则的字符串，用户在授权的过程中容易出错。值得注意的是通过电子邮件地址方式授权的方法最终还是在 S3 服务器内部转换成相应的用户 ID 进行授权。

3) 组授权用户

同样包括两种方式。一种是 AWS 用户组 (AWS User Group)，它将授权分发给所有 AWS 账户拥有者；另一种是所有用户组 (All User Group)，这是一种有着很大潜在危险的授权方式，因为它允许匿名访问，所以不建议使用这种方式。

3.4 简单队列服务 SQS

要想构建一个灵活且可扩展的系统，低耦合度是很有必要的。因为只有系统各个组件之间的关联度尽可能低，才可以根据系统需要随时从系统中增加或者删除某些组件。但松散的耦合度也带来了组件之间的通信问题，如何实现安全、高效地通信是设计一个低耦合度的分布式系统所必须考虑的问题。简单队列服务 (Simple Queue Service, SQS) 是 Amazon 为了解决其云计算平台之间不同组件的通信而专门设计开发的。

3.4.1 SQS 基本模型

SQS 基本模型非常简单，如图 3-18 所示。

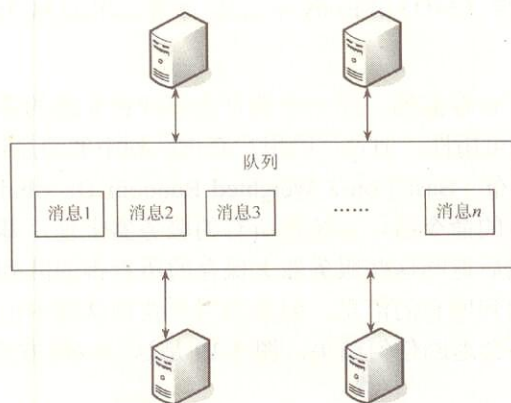


图 3-18 SQS 基本模型

从图中不难看出 SQS 由三个基本部分组成：系统组件^[17] (Component)、队列^[17] (Queue) 和消息^[17] (Message)。系统组件是 SQS 的服务对象，而 SQS 则是组件之间沟通的桥梁。组件在这里有双重角色，它既可以是消息的发送者，也可以是消息的接收者。组件、队列和消息可以形象地比喻为储户、银行和储户账户中的资金。储户随时可以向银行中自己的账户存钱；同时，储户还可以接受别人给他的汇款或给别人汇款；当有需要时，用户可以从银行中取出自己账户中的钱；不需要时，账户中的资金会很安全地保存在银行中。SQS 也是如此，组件既发送消息也接收消息，不接收时消息会被安全地存放在队列中。

3.4.2 两个重要概念

消息和队列是 SQS 中最重要的两个概念。消息是发送者创建的具有一定格式的文本数据，接收对象可以是一个或多个组件。消息的大小是有限制的，目前 Amazon 规定每条消息不得超过 8KB，但是消息的数量并未做限制。队列是存放消息的容器，类似于 S3 中的桶，队列的数目也是任意的，创建队列时用户必须给其指定一个在 SQS 账户内唯一的

名称。当需要定位某个队列时采用 URL 的方式进行访问，URL 是系统自动给创建的队列分配的。队列在发送消息时尽最大努力保证“先进先出”；并非绝对地保证先进的数据一定会最先被投递给指定的接收者，这是它和普通的队列最大不同之处。不过 SQS 允许用户在消息中添加有关的序列数据，对于数据发送顺序要求比较高的用户可以在发送消息之前向其中加入相关信息。和队列相比，消息涉及的内容更多，需要考虑的问题更复杂。下面就消息的内容进行分析。

3.4.3 消息

1. 消息的格式

消息由以下四个部分组成^[11]。

- (1) 消息 ID (Message ID)：由系统返回给用户，用来标识队列中的不同消息。
- (2) 接收句柄 (Receipt Handle)：当从队列中接受消息时就会从消息那里得到一个接收句柄，这个句柄可以用来对消息进行删除等操作。
- (3) 消息体 (Body)：消息的正文部分，需要注意的是消息存放的是文本数据并且不能是 URL 编码方式。
- (4) 消息体 MD5 摘要 (MD5 of Body)：消息体字符串的 MD5 校验和。

2. 消息取样

队列中的消息是被冗余存储的，同一个消息会存放在系统的多个服务器上。这样做的目的是为了保证系统的高可用性，但是这给用户查询队列中的消息带来了一定的麻烦。SQS 使用的是基于加权随机分布 (Based on a Weighted Random Distribution) 的消息取样^[17]，当用户发出查询队列中消息的命令后，系统在所有的服务器上使用基于加权随机分布算法随机地选出部分服务器，然后返回这些服务器上保存的所查询的队列消息副本。虽然用户只要一直查询下去总会查询到所有的消息，但是当用户查询队列中消息数目相对较少的话，在某次查询时系统可能不会返回任何结果。图 3-19 是这种取样方式的示意图。

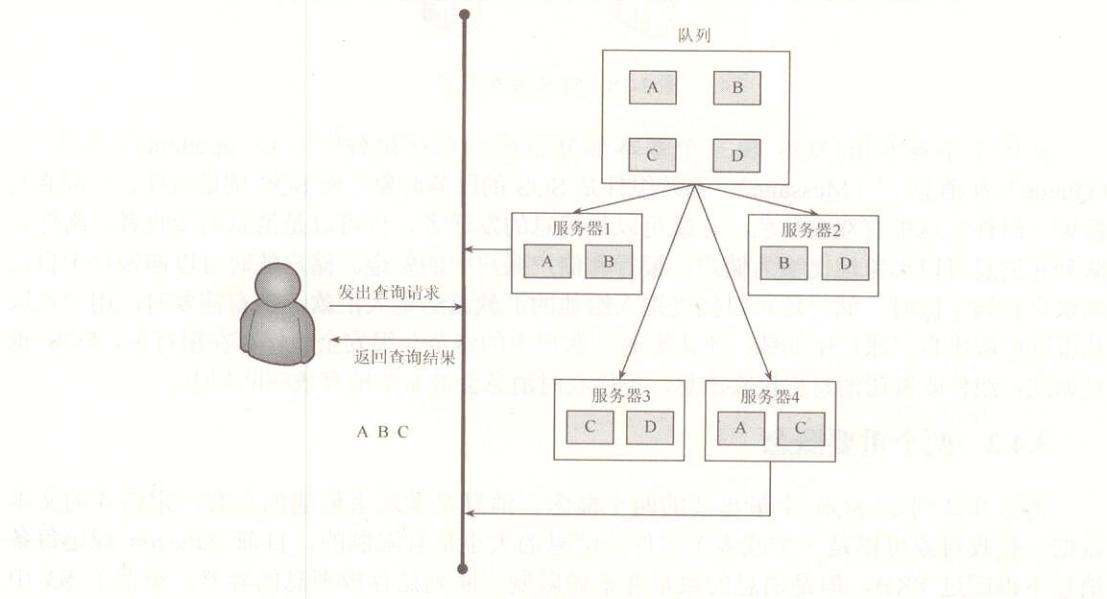


图 3-19 基于加权随机分布的消息取样

可以看出,在图 3-19 中就出现了返回结果不准确的现象,这是因为消息数较少且服务器的选择具有随机性。

3. 消息的可见性超时值及生命周期

在 SQS 中,消息是否被接受是由用户自己确认的。当用户执行删除操作后,系统就会认为用户已经准确地接收到消息,将队列中的消息彻底删除。如果用户未接收到数据或接收到数据并没有执行删除操作, SQS 将在队列中保留该消息。为了保证其他组件不会看见用户的消息, SQS 会将该消息阻塞,也就相当于给消息加了一把锁。但是这把锁并不会一直锁住消息,因为系统保留消息的目的是为了给用户重传数据。为此 SQS 引入了一个可见性超时值^{[1][17]} (Visibility Timeout)。可见性表明该消息可以被所有的组件查看,可见性超时值相当于一个计时器,在设定好的时间内,发给用户的消息对于其他所有的组件是不可见的。如果在计时器到时之前用户一直未执行删除操作,则 SQS 会将该消息的状态变成可见并给用户重传这个消息。可见性超时值可以由用户自行设置,用户可以根据自己操作的需要改变这个值,经验表明太长或太短的超时值都是不合适的。除了在计时器开始计时前改变设置,在计时器计时的过程中还可以对计时器进行两种操作:扩展(Extend)和终止(Terminate)。扩展操作就是将计时器按照新设定的值重新计时,终止就是将当前的计时过程终止,直接将消息由不可见变为可见。这两个操作的设置都只是临时性设置,不会被系统保存。消息从产生并发送至队列一直到其从队列中被删除的全过程称为消息的生命周期(Life Cycle)。如果消息在队列中存放的时间超过 4 天, SQS 也会自动将其删除。图 3-20 是消息的可见性超时值和生命周期的示意图。

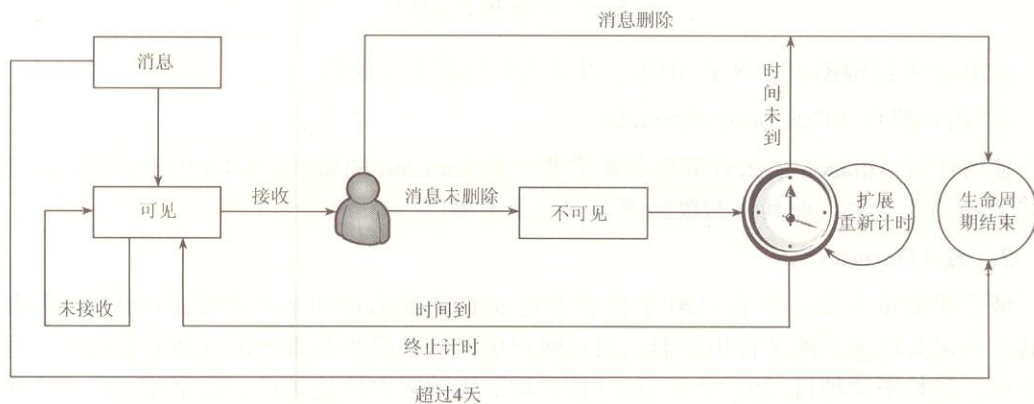


图 3-20 消息可见性超时值及生命周期

3.4.4 身份认证

SQS 中采用的是和 S3 中一致的数字签名方式,具体的内容可以参见 3.3 节,这里不再展开。

3.5 简单数据库服务 Simple DB

与 S3 不同,简单数据库服务 Simple DB (SDB) 主要用于存储结构化的数据,并为

这些数据提供查找、删除等基本的数据库功能。对于 SDB 的具体实现,有人称 SDB 是利用 Erlang (一种面向并发的编程语言) 实现的,有人称其是构建在 Dynamo 之上的,说法不一。Amazon 公司没有公开其实现,具体的技术细节也就无从得知。本节主要探讨 SDB 中的主要概念,与传统的关系型数据库的比较;也简单阐述 SDB 中存在的问题、解决办法及它和 Amazon 其他云计算服务如何有效地结合使用。

3.5.1 重要概念

SDB 基本结构图如图 3-21 所示。

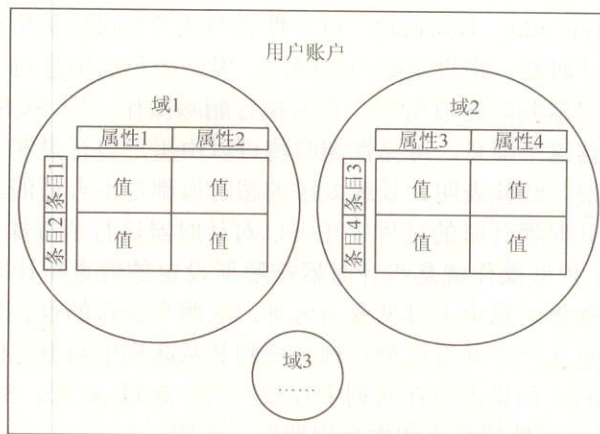


图 3-21 SDB 基本结构图

SDB 基本结构图中包含了 SDB 中以下几个最重要的概念。

1. 用户账户 (Customer Account)

使用任何 Amazon 的云计算服务都需要一个 Amazon 的账户, SDB 也不例外。用户账户^[16]就相当于全集,而具体的数据库则相当于子集。

2. 域 (Domain)

域^[16]是数据容器,由于 SDB 的数据都是以 UTF-8 编码的字符串形式存储的,因此域是具有一定关联关系的字符串容器。用户账户中的域名必须是唯一的且域名长度至少为 3 个字符,最长不能超过 255 个字符。SDB 对于域的限制比较宽松,根据规定,一个账户可以最多拥有 100 个域,而每个域的大小则可以达到 10GB。创建域的目的一般来说是将不同特征的数据分开,有些数据本身就具有可分性,我们可以直接将其划分在不同的域,而像 Web 数据这类本身不好划分的数据 Amazon 建议利用哈希函数将其散列到不同的域中。不过在哈希函数的选择时一定要慎重,因为哈希函数一个很大的问题就是散列地址冲突,在数据集较大时频繁地处理冲突问题会使效率大大降低。SDB 的数据库操作都是以域为基本单位的,即所有的查询都只可以在一个域内进行,域间操作是不允许的。这是一个比较麻烦的限制,从目前来看,Amazon 并没有要解除这个限制的迹象,因此只能由客户端自行解决这个问题。在非必要的情况下最好不要创建新的域,一方面是跨域操作的限制,另一方面是由于建域消耗的时间和相关资源较多。

3. 条目 (Item)

条目^[16]就是一个实际的对象，在 SDB 中，用一系列的属性来描述这个对象，也就是说条目是属性的集合，而且条目名必须是域内唯一的。SDB 不需要事先定义相关的模式 (Schema)，这是 SDB 和传统的数据库最大的不同之一。正因为这种差异使得 SDB 在操作上具有极大的灵活性，用户可以随时创建、删除以及修改条目的内容。条目和条目之间可以属于一类，也可以是完全不相关的两类，这在关系数据库一般也是不允许出现的。

4. 属性 (Attribute)

属性^[16]是条目的特征，它是一个抽象的概念，每个属性都是对条目某一个方面特性的概括性描述。每个条目可以有多个属性，SDB 将域内的属性总数限制为 10 亿。属性的操作也是异常自由的，当某个条目有新的属性时只需要简单地将这个新属性添加进去即可，不用考虑这个属性和其他的条目是否相关。

5. 值 (Value)

每个条目的某个属性的具体内容就是值^[16]，每个条目的所有值可以使用户对于该条目有一个具体、形象的了解。SDB 的值相对于关系数据库而言有一个很实用的改进，那就是允许多值属性。关系数据库中一个字段是不允许有多个值的，但在现实生活中这种情况却是经常发生。比如有两个计算机键盘，它们除了颜色是黑色和白色的区别之外其他参数完全一致，在 SDB 中我们就可以很方便地在颜色属性这一栏中填入黑色、白色而不用担心出错。需要特别注意的是，每个属性值的大小不能超过 1KB，这个限制使得 SDB 存储的数据范围极其有限。出现这种情况应该是 Amazon 刻意为之，Amazon 希望用户将相对大的数据存储在 S3 中，在 SDB 中只保存指向某个特定文件位置的指针。

SDB 和关系型数据库有很多相同之处，但也有很大的不同，用户在决定使用 SDB 时绝不能简单地将 SDB 和关系型数据库做类比。其实严格地说，如果将传统的关系数据库看成一张张表，SDB 更像我们平时常接触到的文件夹的树状结构而不是表结构。图 3-22 显示了 SDB 的树状组织方式。

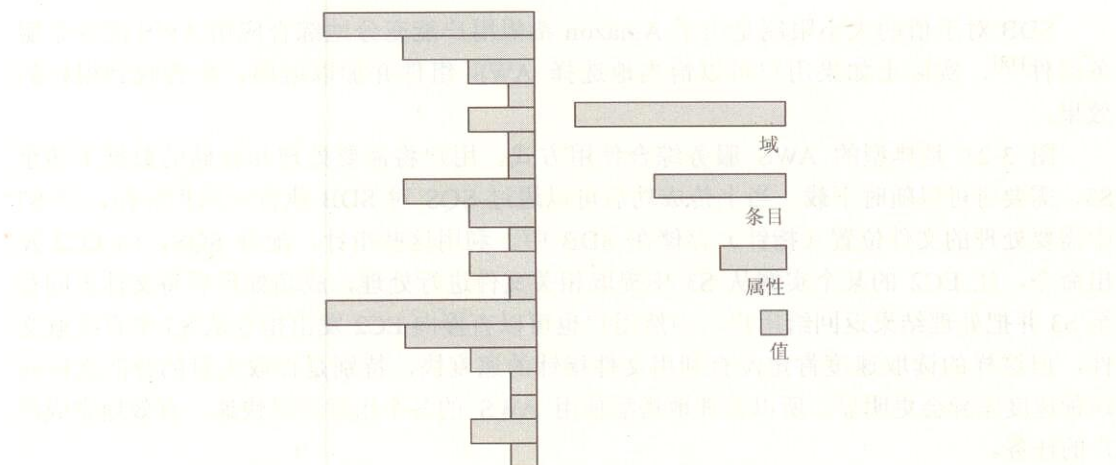


图 3-22 SDB 树状组织方式

除了这种结构上的根本性差异外, SDB 为了系统的高可用性采取了最终一致性数据模型。SDB 还对每次操作设定了一个超时值 (目前是 5 秒), 当操作超过这个时间时, 系统就会向用户返回一个错误。同时 SDB 也对关系数据库做了一些有益的改进, 比如当用户创建数据库后, SDB 会自动对用户添加的数据进行索引, 这样在查询某些数据时速度会大大加快。

3.5.2 存在的问题及解决办法

SDB 出现的一系列问题是由于它的“简单”特性造成的, SDB 主要有以下两个方面的问题。一方面是支持的操作类型不够, 像连接、对结果的排序这样的重要操作目前 SDB 都不支持, 这些工作用户必须自己通过程序来完成, 当然也不排除 Amazon 会在日后向 SDB 中添加这样的功能。另一方面的问题是由它的简单存储方式造成的, 所有的数据在 SDB 中都以字符串形式存储, 因此在做查询操作时采取的是词典顺序 (Lexicographical Order), 有些时候直接采用这种比较方法会出现一些意想不到的问题。对于这类问题, Amazon 也提供了以下一些解决方式^[16]。

(1) 整数补零 (Zero Padding)。一般情况下对数字 20 和 100 的比较很简单, 20 显然排在 100 之前, 但是用字符串方式存储并按词典顺序比较则结果相反, 因为 1 排在 2 之前。对于这种情况, Amazon 建议在整数之前补零。也就是用 00020 和 00100 进行比较, 显然这可以得到和按数字顺序比较一样的结果。

(2) 对负整数集添加正向偏移量 (Negative Numbers Offsets)。这种方法就是对有负数存在的数据集中的每个数加上一个较大的整数, 使负数全部变成整数, 相当于对所有数做正偏移, 如此就能保证比较结果的准确性。

(3) 采用 ISO 8601 格式对日期进行转换 (Convert Dates to Strings Following ISO 8601 Format)。ISO 8601 格式的具体要求大家可以查阅相关标准。

3.5.3 Simple DB 和其他 AWS 的结合使用

SDB 对于值的大小限制是由于 Amazon 希望用户能充分地综合应用 AWS 的各个服务组件^[12], 实际上如果用户可以恰当地选择 AWS 组件并加以运用, 将会收到很好的效果。

图 3-23 是典型的 AWS 服务综合使用方式。用户将需要处理和存储的数据上传至 S3, 需要时可以随时下载。当上传成功后可以通过 SQS 对 SDB 执行一系列操作, 将 S3 中需要处理的文件位置 (指针) 存储在 SDB 中。利用这些指针, 配合 SQS, 向 EC2 发出命令, 让 EC2 的某个实例从 S3 中提取相关文件进行处理, 成功处理后将文件再回存至 S3 并把处理结果返回给用户。当然用户也可以直接向 EC2 发出指令从 S3 中直接取文件, 但这样的读取速度肯定没有利用文件指针的速度快, 特别是在取大量的分散文件时这种速度差异会更明显。所以合理地搭配使用 AWS 的各个组件可以快速、有效地完成用户的任务。

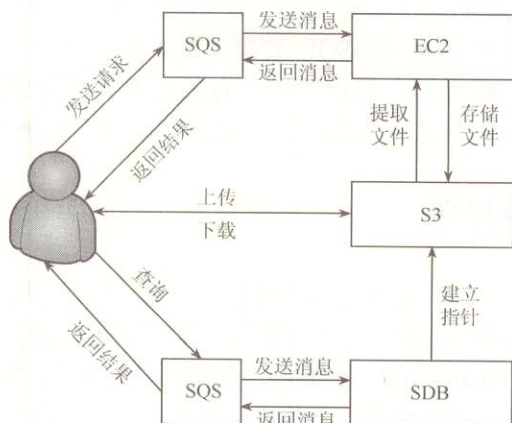


图 3-23 AWS 服务的综合使用方式

3.6 关系数据库服务 RDS

关系数据库主要是以 SQL 为搜索引擎，也称做 SQL 关系数据库；与之相对的是 NoSQL 非关系数据库，它以键值数据库（Key-Value Store DB）^[41]为代表，3.5 节介绍的 Simple DB 就属于键值数据库。随着网络的不断发展，特别是超大规模和高并发的社交网络的出现，传统的关系型数据库已经显得有些力不从心，暴露了很多难以克服的问题，这时非关系数据库应运而生。但是非关系数据库在处理 ACID 类问题时存在着一些先天性的不足，于是人们又开始尝试对传统的关系数据库进行修改，提高它的可扩展性。这节将要介绍的 Amazon RDS（Relational Database Service）就是这样一种技术。

3.6.1 SQL 和 NoSQL 数据库的对比

下面对 SQL 和 NoSQL 数据库进行如下几个方面的对比。

（1）数据模型：SQL 数据库对数据有严格的约束，包括数据之间的关系和数据的完整性。比如 SQL 数据库中某个属性的数据类型是确定的（如整型、字符串等），数据的范围是确定的（如 0~1023 等）。在 NoSQL 数据库中这些都没有，在 Key-Value 存储中，Key 和 Value 可以是任意的数据类型。

（2）数据处理：传统的 SQL 数据库满足 CAP 原则（一致性（Consistency）、可用性（Availability）、分区容忍性（Partition Tolerance））的 C 和 A，所以在 P 方面很弱，进而导致传统数据库在可扩展性方面，面临很多问题。NoSQL 数据库满足 CAP 原则的 A 和 P，所以在 C 比较弱，进而使得非关系数据库无法满足 ACID 要求。

（3）接口层的区别：SQL 数据库都是以 SQL 语言对数据进行访问的，一方面 SQL 语言提供了强大的查询功能，另一方面，目前所有的 SQL 数据库都支持 SQL 语言，移植性很高。NoSQL 数据库对数据的操作都是通过一些 API 实现的，支持的查询功能很简单，并且不同的数据库有不同的 API，移植性较差。

(4) 优势和劣势：SQL 数据库具有高的一致性，在 ACID 方面能力非常强，移植性很高，但在可扩展性方面能力较弱。NoSQL 数据库最大的优点是非常高的可扩展性，可以通过增加服务器的数量不断提高存储规模，具有很强的并发处理能力，但缺乏数据一致性保证。另外由于分布在多个服务器上，所以跨表、跨服务器查询很困难。

3.6.2 RDS 数据库原理

Amazon RDS^[42]是一种云中的 MySQL 数据库系统，它采用集群方式将 MySQL 数据库移植到云中，在一定的范围内解决了关系数据库的可扩展性问题。

MySQL 集群采用了 Share-Nothing 架构，如图 3-24 所示。每台数据库服务器都是完全独立的计算机系统，通过网络相连，不共享任何资源。这是一个具有较高可扩展性的架构，当数据库处理能力不足时，可以通过增加服务器数量来提高处理能力，同时多个服务器也增加了数据库并发访问的能力。

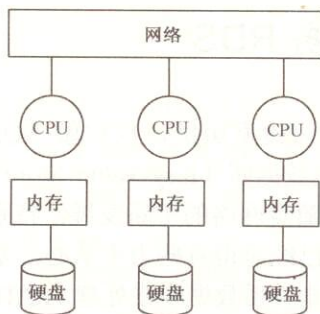


图 3-24 Share-Nothing 架构

集群 MySQL 通过表单划分（Sharding）的方式将一张大表划分为若干个小表，分别存储在不同的数据库服务器上，这样就从逻辑上保证了数据库的可扩展性。但是表单的划分没有固定的方式，主要根据业务的需要进行针对性的划分，这就对数据库的管理人员提出了非常高的要求，如果划分得不科学，则查询经常会跨表单和服务器，性能就会严重下降。

集群 MySQL 通过主从备份和读副本技术提高可靠性和数据处理能力，如图 3-25 所示。Master A 为主数据库，Master B 为从数据库，组成主从备份。如果 Master B 检测到 Master A 瘫痪，则立刻接替 Master A 的位置，成为主服务器，并会重新创建一台从服务器。在数据库升级时，先对从数据库进行升级，然后将从数据库转变为主数据库，再对新的从数据库进行升级，这样就可以实现数据库的实时升级，保证业务的连续性；为了提高数据库的并发处理能力，集群 MySQL 设置了若干个读副本（Slave），顾名思义，读副本中的数据只能读，不能写，写操作只能由主数据库来完成。

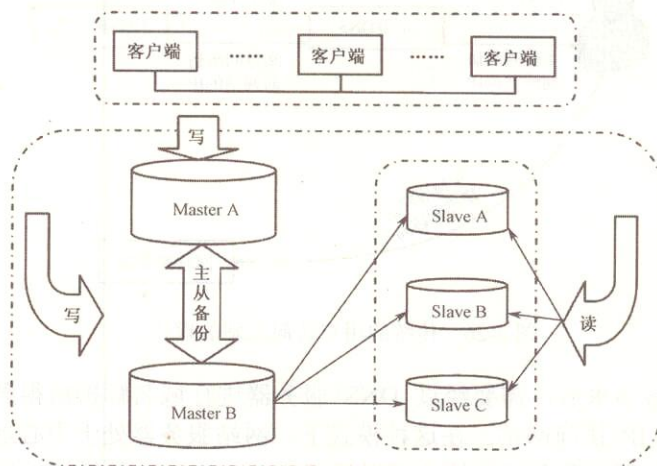


图 3-25 集群 MySQL

3.6.3 RDS 的使用

从用户和开发者的角度来看，RDS 和一个远程 MySQL 数据库没什么两样。Amazon 将 RDS 中的 MySQL 服务器实例称做 DB Instance，通过基于 Web 的 API 进行创建和管理，其余的操作可以通过标准的 MySQL 通信协议完成。创建 DB Instance 时需要指定一些属性来确定数据库实例的行为和能力，例如 Class 属性决定了所创建的 DB Instance 可用的内存和处理能力。Amazon 以 ECU（Elastic Compute Unit）作为其计算能力单位（1 个 ECU 差不多相当于 1 个 1.0GHz 2007 Xeon 处理器），用户可以选择创建拥有 1.7GB 内存和 1 ECU 的小型 DB Instance 或者是拥有 68GB 内存和 26 ECU 的超级大型（Quadruple Extra Large）DB Instance。创建 DB Instance 时还需要定义可用的存储，存储范围为 5GB 到 1024GB，RDS 数据库中表最大可以达到 1TB。

可以通过两种工具对 RDS 进行操作：命令行工具和兼容的 MySQL 客户端程序。命令行工具是 Amazon 提供的 Java 应用套装，负责处理 DB Instance 的管理，比如创建、参数调整、删除等，可以从 Amazon 网站下载。MySQL 客户端是可以与 MySQL 服务器进行通信的应用程序，比如 MySQL Administrator 客户端。

3.7 内容推送服务 CloudFront

CloudFront 实际上就是一个基于 Amazon 云计算平台实现的内容分发网络（Content Delivery Network，CDN）。通过 Amazon 部署在世界各地的边缘节点，用户可以快速、高效地对由 CloudFront 提供服务的网站进行访问。

3.7.1 内容推送网络 CDN

传统的网络服务模式中，用户和内容提供商位于服务的两端，网络服务提供商将两者联系起来。在这种情况下，网络服务提供商仅仅起“桥梁”作用。图 3-26 是传统的用户访问网站的模式。

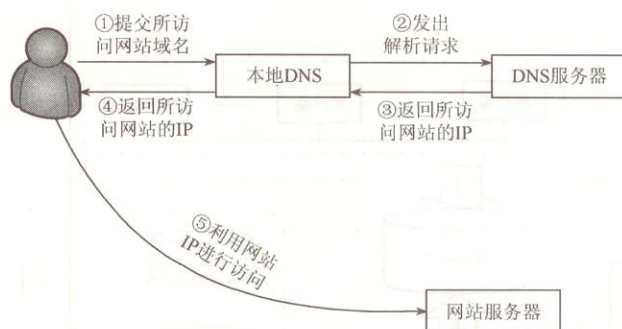


图 3-26 传统的用户访问网站的模式

用户在发出服务请求后，需要经过 DNS 服务器进行域名解析后得到所访问网站的真实 IP，然后利用该 IP 访问网站。在这种模式下，网站服务器处于中心地位，世界各地的访问者都必须直接和网站服务器连接才可以访问相关内容，这种模式的缺点不言而喻。第一，网站服务器可以容纳的访问量是有限的，通常情况下访问量一般不会超过或仅仅略微超过这个值；但是一旦发生突发事件，例如该网站首先发布了某个重大消息或遭受到 DDOS（分布式拒绝服务攻击），网站的流量会在短时间内急剧上升，带来的必然结果就是访问速度下降，更有甚者直接导致网站瘫痪。第二，这种模式并没有考虑访问者的地域问题，假设该网站服务器架设在美国，很显然的是中国和美国的访问速度有很大区别，甚至有时候在中国的用户根本无法访问。第三，使用不同网络服务提供商服务的用户之间的互访速度也会受到限制，国内电信和网通之间资源互访速度相对较慢就是一个典型例子。

为了解决这个弊端，人们发明了很多技术，从目前来看，CDN 是其中非常具有代表性的一种。CDN 通过将网站内容发布到靠近用户的边缘节点，使不同地域的用户在访问相同网页时可以就近获取。这样既可以减轻源服务器的负担，也可以减少整个网络中流量分布不均的情况，进而改善整个网络性能。所谓的边缘节点是 CDN 服务提供商经过精心挑选的距离用户非常近的服务器节点，仅“一跳”（Single Hop）之遥。用户在访问时就无需再经过多个路由器，访问时间大大缩减。CDN 是通过在现有的网络上增加一层网络架构来实现的。图 3-27 是使用了 CDN 后用户访问网站的基本流程图。

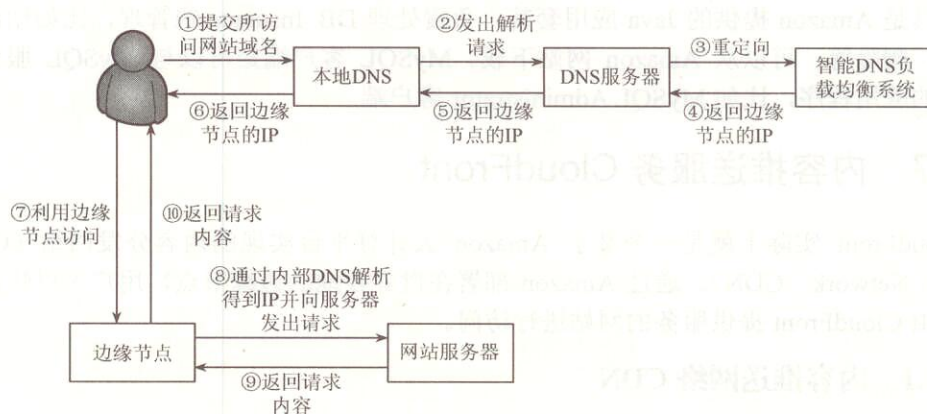


图 3-27 加入 CDN 后用户访问流程

从图中我们可以看出, DNS 在对域名进行解析时不再向用户返回网站服务器的 IP, 而是返回了由智能 CDN 负载均衡系统选定的某个边缘节点的 IP。用户利用这个 IP 来访问边缘节点, 然后该节点通过其内部 DNS 解析出真实的网站 IP 并发出请求来获取用户所需的页面, 请求成功后向用户显示该页面并加以保存以便用户再次访问时可以直接读取。这种访问模式的好处主要有以下几点。

(1) 将网站的服务流量以比较均匀的方式分散到边缘节点中, 减轻了网站源服务器的负担。

(2) 由于边缘节点与访问者的地理位置较近, 访问速度快。

(3) 智能 DNS 负载均衡系统和各个边缘节点之间始终保持着通信联系, 可以确保分配给用户的边缘节点始终可用且在允许的流量范围之内。

CDN 的实现需要多种网络技术的支持, 主要包括以下几种。

(1) 负载均衡技术: 负载均衡就是将流量均匀的分发到可以完成相同功能的若干个服务器上, 在减轻服务器压力的同时也避免了单一网络通道的流量拥堵。

(2) 分布式存储: 在使用 CDN 服务之后, 网站的内容不再是单一的被保存在源服务器上, 多个边缘节点中都可能保存相应的副本。如何对网页内容进行分发以及如何保证边缘节点内容的时效性都是需要考虑的问题。

(3) 缓存技术: 缓存技术通过将内容存储在本地或者网络服务提供商的服务器上来改善用户的响应时间。

目前国内一些大的门户网站像新浪、网易等都已经采用了 CDN, 用户无论在何地访问这些网站时可能都感觉不到网络拥堵的情况。但对一些经济实力有限的中小企业来说, 资金的限制使他们无法大规模地使用普通的 CDN 服务, CloudFront 的推出无疑给这些企业带来了便利。下面就简单介绍 CloudFront 这种云端的 CDN。

3.7.2 云内容推送 CloudFront

CloudFront 正是通过 Amazon 设在全球的边缘节点来实现 CDN 的, 但是较普通的 CDN 而言, 它的优势无疑是巨大的。首先, CloudFront 的收费方式和 Amazon 的其他云计算收费方式一样是按用户实际使用的服务来收费, 这尤其适合那些资金缺乏的中小企业。其次, CloudFront 的使用非常简单, 只要配合 S3 再加上几个简单的设置就可以完成 CDN 的部署。下面先介绍 CloudFront 中的几个基本概念。

1. 对象 (Object)

对象^[21]就是希望利用 CloudFront 进行分发的任意一个文件, 但该文件首先须满足两个条件: 一个是必须存储在 S3 中; 另一个是它必须被设置为公开可读 (Publicly Readable)。一般来说, 通过 CloudFront 分发网页中的静态内容比较合适。

2. 源服务器 (Origin Server)

源服务器^[21]就是存储需要分发文件的位置, 对 CloudFront 服务而言就是 S3 中的桶。

3. 分发 (Distribution)

分发^[21]的作用是在 CloudFront 服务和源服务器之间建立一条通道, 告诉 CloudFront 需要对这个源服务器上的文件使用 CloudFront 服务, 所以要使用 CloudFront 必须要创建

新的分发并将分发和指定的源服务器做关联，这种关联关系一旦建立就不可更改。每个用户最多可以创建 100 个分发，在创建时可以设置分发的状态为启用（Enabled）或禁用（Disabled），这两种状态之间可以任意切换。

4. 别名指向（CNAME）

用户在使用 CloudFront 服务之后，系统会自动给用户分配一个域名，以便用户使用这个新域名对源服务器中的文件进行引用而不是使用 S3 中原来使用的引用方式。CNAME^[21]实际上就是这个系统分配给用户域名的一个别名。举个简单的例子：假设用户需要对 S3 中的文件 pic.jpg 进行分发，系统分配给你的域名是 http://abc.cloudfront.net，那么如果不使用别名指向，用户看到的该文件的链接就会是 http://abc.cloudfront.net/pic.jpg。这显然不能满足很多用户的需求，他们希望所有的链接看起来都是由自己的网站发布的，这时用户就可以使用别名指向。假设想使用的别名指向是 http://chinacloud.cn，那么用户访问时所见到的最终链接地址就是 http://chinacloud.cn/pic.jpg。这项功能是相当实用的，多数用户都会选择使用它。

5. 边缘节点位置（Edge Location）

边缘节点位置^[21]就是实际的边缘节点服务器位置，目前 Amazon 在全球共有 14 个边缘节点，分布在美国、欧洲和亚洲。

6. 有效期（Expiration）

有效期^[21]就是文件副本在边缘节点上的存放时间，默认的是 24 小时，用户可以对这个值进行设置，但最少不能低于 24 小时，当时间到了之后边缘节点就会自动删除文件副本。

图 3-28 是 CloudFront 的基本架构，它和上面的 CDN 结构很类似。

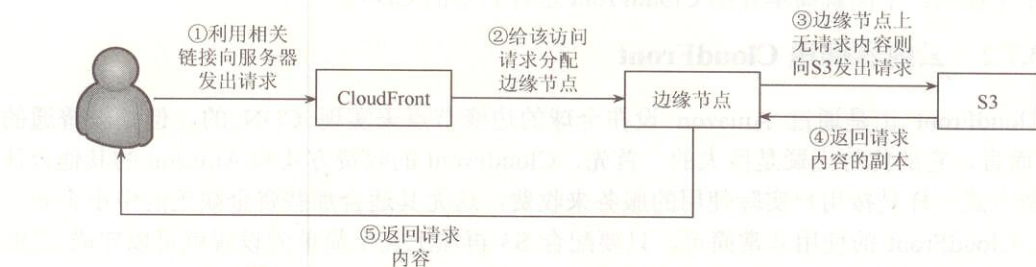


图 3-28 CloudFront 基本架构

从图中可以看出，CloudFront 在此处就相当于 CDN 中的智能 DNS 负载均衡系统，用户实际是和 CloudFront 进行服务交互而不是直接和 S3 中的原始文件进行交互。这样既保证了 CloudFront 和 S3 的相对独立性，又使访问效率得以提高。

CloudFront 服务的安全措施也至关重要。除了所有 AWS 共有的安全措施之外，CloudFront 还向用户提供了访问日志^[21]，用户可以自行决定是否启用这项功能，访问日志会记录所有通过 CloudFront 服务访问用户分发的文件的行为。访问日志本身并不会阻止某些非法用户的访问，它的作用主要是用来对于访问者行为进行分析以便发现某些漏洞，进而采取更严密措施保证系统安全。CloudFront 与其他一些 AWS 不同的是，它只接受安全的 HTTPS 方式而不接受 HTTP 方式进行访问，这又进一步提高了安全性。

3.8 其他 Amazon 云计算服务

3.8.1 快速应用部署 Elastic Beanstalk 和服务模板 CloudFormation

为了更好地、更方便地使用各种云服务, Amazon 提供了快速应用部署 Elastic Beanstalk 和服务模板 CloudFormation 两种服务。

AWS Elastic Beanstalk^[48]是一种简化在 AWS 上部署和管理应用程序的服务。用户只需上传自己的程序,系统会自动地进行需求分配、负载均衡、自动缩放、监督检测等一些具体部署细节。在使用 AWS Elastic Beanstalk 的同时,用户可以随时对其使用的资源和程序进行访问。而传统的程序容器或以平台为服务的解决方案,在减少编程工作量的同时也大大减弱了开发人员的灵活性和对资源的控制能力。开发者只能使用供应商提供的接口来控制资源。目前 AWS Elastic Beanstalk 仅针对 Java 开发者提供支持。

Elastic Beanstalk 虚拟机是一种运行 Apache Web Server、Tomcat 和 the Enterprise Edition of the Java platform 的 AMI 虚拟机,具有以下特点。

(1) Elastic Beanstalk 构筑于 AWS 之上,因此它具有负载均衡、云监控、自动缩放等特性。

(2) 通过 Elastic Beanstalk,用户可以采用多种方式对其程序进行控制和参数设置,也可以通过登录 EC2 实例来处理程序出现的问题,或者采用 Elastic Beanstalk AMI 提供的默认处理方式。

(3) Elastic Beanstalk 目前版本只支持 Java,但以后将会支持多种编程语言。

(4) Elastic Beanstalk 为每个应用运行多个 EC2 实例,提高程序的可靠性。

(5) 利用 Elastic Beanstalk 部署的用户程序可以调用部署在其他 EC2 实例上的程序,并能保证时延。

AWS CloudFormation^[46]的功能是为开发者和系统管理员提供一个简化的、可视的 AWS 资源调用方式。开发者可以直接利用 CloudFormation 提供的模板或自己创建的模板方便地建立自己的服务,这些模板包含了 AWS 资源及相关的参数的设置、应用程序的调用方式。用户不需要了解 AWS 的资源及相互依赖关系, CloudFormation 都可以自动地处理完成。

3.8.2 云中的 DNS 服务 Router 53

传统的 DNS 服务器都存在这样一个问题:域名对应的 IP 地址变更,有时传播得非常缓慢。但是用户要求好的用户体验,企业也有业务联系性的内在要求,为了很好地解决这个问题,Amazon 推出了云中的 DNS 服务:Route 53 服务。

Router 53^[45]是用来管理 DNS、处理 DNS 请求的全新 AWS。该服务运行在 Amazon 的云中,提供了 DNS 授权服务器的功能,可以通过 RESTAPI 进行访问,这个 API 允许用户创建管理区 (Zone),并在区中保存 DNS 记录。创建管理区的时候,Router 53 同时分配四个域名服务器来处理域名的请求,这些域名都是与用户创建的管理区关联的。Route 53 可以为运行在 Amazon 云中的域名、互联网中的域名,或者是两种相混合的域名提供服务。

为了提供高可用、低延迟的 DNS 服务,Amazon 在全球分布了多台服务器,其中有九

台分布在美国各州，四台分布在欧洲，三台分布在亚洲。Route53 会把 DNS 请求路由到最近的服务器，以便快速地响应用户请求。

3.8.3 虚拟私有云 VPC

Amazon 虚拟私有云（Virtual Private Cloud, VPC）^[44]是一个安全的、可靠的、可以无缝连接企业现有的基础设施和 Amazon 云平台的技术，如图 3-29 所示。VPC 将企业现有网络和 AWS 计算资源连接成一个虚拟专用网络资源，提供强大的网络功能，如安全检测、防火墙和入侵检测等。对于一些小规模或初创的企业来说，维护和管理自己的 IT 基础设施往往会分散企业注意力，减弱企业竞争力和提供服务的能力。通过 Amazon VPC，企业可以很容易地获得需要的基础资源，有效地控制成本、节省时间。

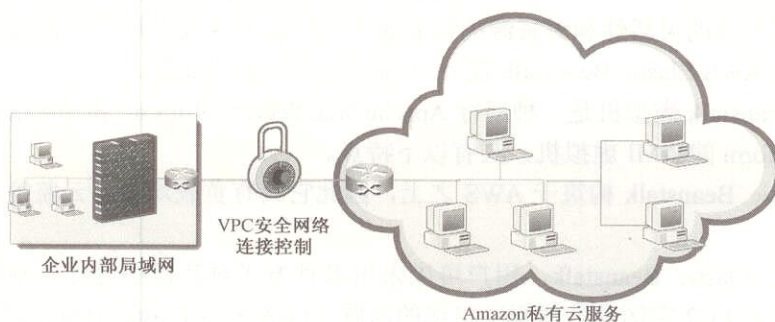


图 3-29 VPC 示意图

3.8.4 简单通知服务 SNS 和简单邮件服务 SES

Amazon 简单通知服务（Simple Notification Service, SNS）^[47]是一种 Web 服务，提供方便的信息发布平台，具有高的可扩展性和成本优势。应用程序可以通过 SNS 发布消息，快速地传送给用户或其他应用程序的开发员。无需其他中间件和管理程序的辅助，用户可以直接通过 SNS 来创建高可靠性的、事件驱动的工作流程和信息应用，使得通过网络进行大规模的计算和程序开发变得更容易。SNS 潜在的用途包括监控应用、工作流系统、事件敏感的信息更新、移动应用等。例如，运行在 EC2 上应用程序发布更新版本，可以通过发布事件 SNS 信息，传递给其他应用程序或终端用户。此外，用户可以用 SNS 作为 Amazon SQS 的传输协议，将 SQS 消息传递到消息队列，保证消息的传递和持久性。在未来，Amazon 的 SNS 将整合到如 Amazon S3 和 Simple DB 的其他 AWS 服务中。

Amazon 简单邮件服务（Simple Email Service, SES）是一个简单的高扩展性和具有成本优势的电子邮件发送服务。SES 消除了建立企业内部电子邮件系统的复杂性，节省了设计、开发、安装和运行第三方电子邮件系统的费用。通过简单的 API 调用，企业就可以获得高品质电子邮件系统，将高效率、低成本的优势转移到用户身上。同时 SES 采用了内容过滤技术，有效地阻止垃圾邮件。

3.8.5 弹性 MapReduce 服务

在 Amazon 推出弹性 MapReduce 服务之前，已经有人成功地通过在 EC2 上部署

Hadoop 实现了 MapReduce 的功能^[6]，现在 Amazon 将这项服务整合到 AWS 之中，为需要进行海量数据处理的用户提供了极大便利。用户可以忽略服务器及软件部署的细节问题，而将主要精力集中在对数据的处理和研究之中。MapReduce 及 Hadoop 的相关内容在书中其他部分有详细介绍，这里就不再重复，本节重点介绍 Amazon 的弹性 MapReduce 实现方式。

Amazon 的弹性 MapReduce 是通过 EC2 和 S3 来实现的，其基本架构如图 3-30^[19]所示。

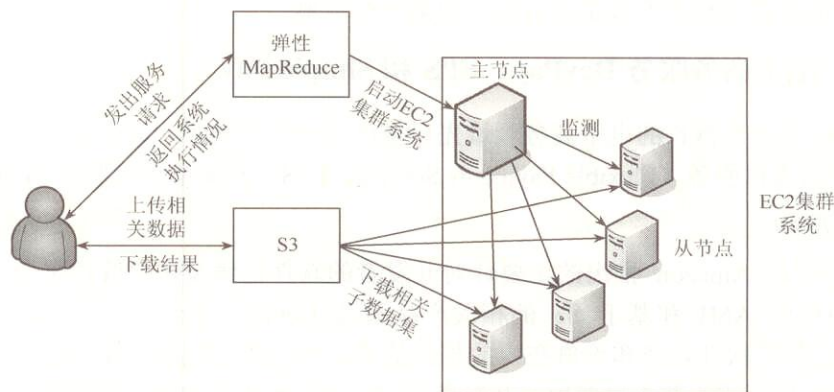


图 3-30 弹性 MapReduce 架构

用户在使用弹性 MapReduce 时，首先要将相关数据上传至 S3，在 Amazon 弹性 MapReduce 中，S3 作为原始数据和处理结果的存储系统。需要上传的相关数据中既包括用户待处理的数据，也包括一个 Mapper^[20]和一个 Reducer^[20]执行代码。Mapper 和 Reducer 分别实现了 MapReduce 中的 Map（映射）和 Reduce（化简）功能。这两个功能实现的语言并没有限制，用户可以根据自己的习惯选择。在弹性 MapReduce 内部也有一些 Amazon 提供给用户的默认 Mapper 和 Reducer。相关数据上传成功后用户就可以向系统发出一个服务请求，系统接收到请求后就会启动一个由一定数量的 EC2 实例组成的集群系统，集群中的实例数量和实例类型用户可以自行设置。为了使集群的效率达到最高，用户在使用前最好做相关测试以确定需要的实例数量和类型。EC2 集群系统采用主/从（Master/Slave）模式，即系统中有一个主节点和若干数量的从节点，主/从节点上都运行着 Hadoop。主节点上的 Hadoop 在主节点接受启动集群的服务请求后，将 S3 中的待处理数据划分成若干个子数据集；从节点从 S3 中下载相关子数据集，这包括划分好的待处理数据的子集、一个 Mapper 以及一个 Reducer；接下来，每个从节点都独自处理分发到的子数据集。整个运行过程在主节点的监测之下，每个从节点需要向主节点发送运行状态元数据（Status Metadata）。处理完的结果将再次被汇总至 S3，此时弹性 MapReduce 服务会通知用户数据处理完毕，用户直接从 S3 上下载最终结果即可。

从上面的处理过程可以看出，弹性 MapReduce 的运行过程非常简单，用户根本不需要考虑计算中涉及的服务器部署、维护及软件环境的配置。除了这些基本的设置不需要用户处理之外，Amazon 在可靠性、数据安全等方面也采取了和其他云计算服务类似的措施。例如为了保证高可靠性，子数据集不是被分发到一个从节点而是被分发到多个从节点，这样保证单个从节点的失败不会影响最后结果。Amazon 允许用户在上传数据前对

数据进行加密并通过安全的 HTTPS 协议上传数据。弹性 MapReduce 中的实例被划分成两个安全组：一个是主节点安全组，另一个是从节点安全组。Amazon 提供了诸如此类的一系列完善的用户安全服务。在弹性 MapReduce 中，有一个概念需要特别提请注意：任务流^[19] (Job Flow)。任务流实际上是由一系列前后相关的处理过程组成的，可以与线性链表的结构类比，除了第一个节点和最后一个节点，每个节点既是前一个节点的后继也是后一个节点的前驱。同样的道理，任务流中除了第一个任务和最后一个任务外，其他的任务既是作为上一个任务的输出也是作为下一个任务的输入。Amazon 的弹性 MapReduce 将数据的实际计算过程都看成是任务流中的某一个步骤。

3.8.6 电子商务服务 DevPay、FPS 和 Simple Pay

Amazon 在其最擅长的电子商务领域先后推出了一系列服务，其中比较有代表性的是 DevPay、灵活支付服务 (Flexible Payments Service, FPS) 和简单支付服务 Simple Pay。

1. DevPay

DevPay^[24]是 Amazon 推出的主要针对开发者的软件销售及账户管理平台。开发者将自己开发的付费 AMI 和基于 S3 的相关产品通过 DevPay 平台进行发布，用户则通过 DevPay 浏览包括软件功能和价格在内的相关信息，一旦觉得该软件比较适合自己则可以通过 DevPay 进行购买并支付费用。开发者通过 DevPay 提供的账户管理功能对自己的账户及产品进行管理，可以进行诸如查看使用产品的用户情况、修改产品价格等操作。

为了便于理解，将 DevPay 和淘宝做简单的类比。这种类比严格意义上来讲不是完全正确，但这并不影响对基本概念的理解，而且这种类比更加形象。如图 3-31 所示，DevPay 和淘宝交易平台类似，不同之处在于淘宝出售的大多是实体商品，而 DevPay 售卖的都是软件。Amazon Payments^[24]和支付宝的功能完全相同，属于第三方支付平台，DevPay 中的所有的交易都通过 Payments 完成。

开发者和用户都可以从 DevPay 中受益，用户可以利用开发者开发的软件更加方便地使用包括 EC2、S3 在内的 Amazon 云计算服务。开发者则可以在 Amazon 的巨大用户群体中推广自己的产品，除此之外还能利用 Amazon 先进的支付手段来降低开发难度，并能有效地保证资金安全。图 3-32 是 DevPay 服务的基本架构图。



图 3-31 DevPay 和淘宝的简单类比关系图

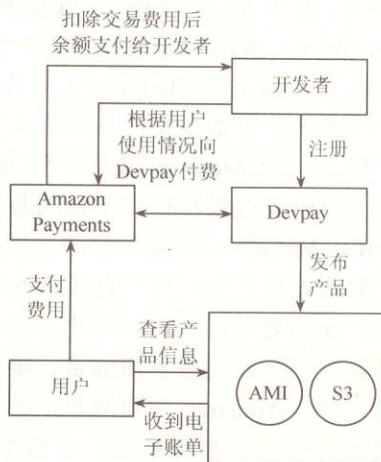


图 3-32 DevPay 服务的基本架构图

开发者首先在 DevPay 上将自己希望发布的产品进行注册。DevPay 允许开发者发布的产品目前只有两类：一类是付费 AMI^[24]；另一类是基于 S3 服务开发的产品^[24]。用户在产品成功发布后就可通过有关页面看到产品的信息。产品成功购买后用户就会收到 Amazon 发出的电子账单，用户可以立即或稍后通过 Amazon Payments 进行支付，Amazon Payments 扣除交易费用后余额会支付给开发者。在每月的固定时间开发者还需要向 Amazon 支付使用 DevPay 的费用，关于收费情况将在稍后介绍。需要注意的是用户会收到系统产生的一个激活码，用户在使用产品之前需要利用这个激活码来激活产品。

DevPay 的计费方式是开发者和用户都很关心的。在 DevPay 服务中，计费包括两部分：开发者向用户收取的费用和 DevPay 向开发者收取的费用。整个计费系统如图 3-33 所示。

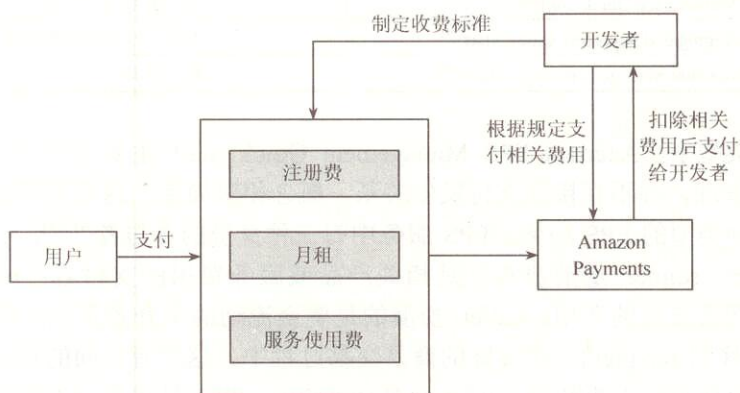


图 3-33 DevPay 计费系统

根据规定，开发者可以向用户收取的费用包括三种^[24]：一次性注册费（One-time Sign-up Charge）、月租（Monthly Charge）、服务使用费（Usage-based Charges）。开发者可以根据自身的情况从这三种费用中选取若干种对用户收取费用，这种收费方式和移动电话的收费方式很像。在入网时，有时需要支付一定入网费用，有时则不用。电信运营商按月收取一定数量月租，但也有些套餐是不需要月租的。在月末结算时，用户要对实际使用的服务支付费用，用户在选择某些包月服务时也可能不用支付这笔费用。开发者可以根据需要通过 DevPay 随时改变自己产品的定价策略。在正常情况下，DevPay 服务向开发者收取的费用包括两部分，所谓的正常情况是指用户按时足额缴纳了使用费用，此时 DevPay 所收取费用的计算方式^[24]是：

$$(\text{用户支付的费用} - \text{AWS 服务使用费}) \times 0.03 + 0.3 \times \text{使用该产品的用户数}$$

AWS 服务使用费是用户使用开发者产品过程中引发的相关 AWS 服务费用，比如使用基于 S3 的产品产生的每月的存储费、数据上传及下载的费用。用户在使用开发者的产品后就不需要再支付这些费用，因为这些费用将由开发者支付，DevPay 在开发者额外获取的费用^[24]（Value-Add）收取百分之三的服务费。另外对于每个用户，Devpay 将向开发者征收 0.3 美金的交易费，用户数是按实际使用的用户数目来计算的。如果用户不支付或只支付了一部分费用，DevPay 有另外的计费方式，限于篇幅这里不再介绍。

2. FPS

灵活支付服务 FPS 并不是 Amazon 推出的第一项支付服务，但有别于其他的支付服务，FPS 的特色体现在它的灵活性上。FPS 允许用户根据需求和实际情况对支付服务进行各种个性化的设置，使其和用户的电子商务平台更加契合。为了实现这种灵活性，FPS 将网上交易中可能出现的交易类型进行了细化，并在此基础上将 FPS 服务划分成五种类型，这五种类型及其适用的范围如表 3-6^[25]所示。

表 3-6 FPS 服务类型

FPS 服务类型	适合的交易类型
Amazon FPS Basic Quick Start ^[26]	一次性的交易
Amazon FPS Advanced Quick Start ^[27]	买卖双方多次或重复交易
Amazon FPS Marketplace Quick Start ^[28]	有中介参与的三方交易
Amazon FPS Aggregated Payments Quick Start ^[29]	将数个小额交易集成单个交易
Amazon FPS Account Management Quick Start ^[30]	账户管理

在这五种类型中，Account FPS Management Quick Start 主要是便于用户对于自己的 FPS 账户进行管理，并不直接和支付发生关系，概念相对简单，这里不展开介绍，这里主要介绍其他四种类型的 FPS 服务。FPS 服务中有三种身份的参与者^{[4][25]}，分别是 sender、recipient、caller。sender 是消费者，是相关产品或服务费用的支付者；recipient 是销售者，它接受消费者支付的费用；caller 扮演的是资金流动的中介者角色，它的作用是将资金从 sender 转移到 recipient。在实际的商品交易过程中，这三者之间的身份界限有时并不是那么清晰，但是用户不能既是 sender 又是 recipient，也就是说资金不能从用户流向其自身。和 DevPay 一样，FPS 服务中的资金流动也是通过 Amazon Payments 实现的，所以使用 FPS 的用户也需要拥有一个 Amazon Payments 账户。

顾客在使用了 FPS 服务的网站上购买产品或服务的基本流程如图 3-34 所示。

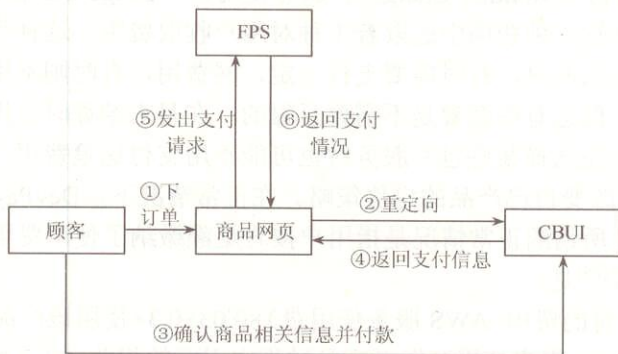


图 3-34 顾客购买基本流程

整个购买流程由包括顾客在内的四个部分组成，其中和顾客直接发生关系的有两个部分，分别是商品网页和 CBUI^[25]。商品网页很好理解，顾客在该网页上选购产品。CBUI 是 Co-Branded User Interface 的简写，也就是联合品牌标志用户界面。在 CBUI 上会有商家及 Amazon Payments 双重品牌标志，这样做的目的是保持购物过程中用户体验的一致

性。如果不使用 CBUI，用户在付款时忽然跳转到一个完全没有该商家标志的支付网页可能会产生一种不信任感。当用户在 CBUI 对所购买的商品做出确认并付款后系统向用户返回一个事先设定好的表示交易成功的界面，同时系统会向商品页面返回支付信息。支付信息中有一个称为 Payment Token^[25]的 ID，这个 ID 非常重要，因为它包含了用户购买产品数量、产品类型等交易信息，通过这个 ID 可以区分 FPS 服务类型。Payment Token 有以下几种^[25]。

(1) Single-use: 一次性交易所需的 Token。

(2) Recurring-use: 每隔固定的间隔时间就对购买进行确认所需的 Token。例如，用户在網上订阅了一份周报，那么就需要使用 Recurring-use Token 每隔一周就对付款做出确认。

(3) Multi-use: 可以在多次交易中使用的 Token。Recurring-use Token 是 Multi-use Token 的一种。

(4) Prepaid: 使用预付款方式进行交易所需的 Token。用户首先预付一定的款项，下次交易时产生的费用直接从预付款中扣除直到预付款为零。这类似于在超市购买消费卡。

(5) Postpaid: 使用赊账方式进行交易所需的 Token。消费者在购买商品时不是直接付款，而是采取了赊账的方式进行消费，等到了赊账的限额或到了买卖双方商定的额度时消费者一次付清所有费用。这和使用信用卡透支消费类似。

(6) Editing: 对已存在的 Token 修改时所需。

不同类型的 FPS 服务中会返回不同的 Payment Token，这就是几种 FPS 服务的最主要区别。图 3-35 显示了不同的 FPS 服务可能返回的 Payment Token。

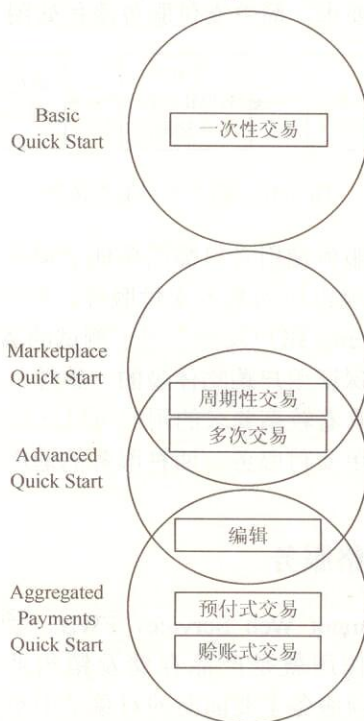


图 3-35 不同 FPS 服务返回的 Payment Token

在收到 Payment Token 后，商品网页会向 FPS 服务发出支付请求，成功之后顾客的付款就转移到销售者的账户上。

FPS 还向开发者提供了一个沙盒^[25]（Sandbox）用来做测试，在正式使用 FPS 之前利用沙盒进行测试是非常有必要的，而且不会产生任何费用。

3. Simple Pay

简单支付服务（Simple Pay）^[23]是一种允许顾客使用其 Amazon 账户进行支付的服务，商家只需要在相应的 Web 支付页面放置合适的按钮就可以使用户利用其 Amazon 账户对商品进行支付。目前简单支付服务有五种常用的支付按钮^[23]，按钮类型及其功能如表 3-7 所示。

表 3-7 简单支付服务常用按钮

按钮类型	功能
Standard Button	普通的一次性购物
Marketplace Button	作为交易的中介者
Basic Donation Button	允许在美的通过美国国税局认证的非营利性机构募集捐款
Marketplace-Enabled Donation Button	允许第三方机构代表非营利性组织来募集捐款
Subscription Button	通过该按钮可以收取类似订阅费的重复性费用，还可以利用该按钮对用户提供免费试用服务或进行产品介绍

简单支付服务的功能和 FPS 服务类似，但和 FPS 相比，它的最大优势就是简单。FPS 服务允许开发者自行定制其支付页面，可以实现各种复杂的支付方式，但高度的灵活性带来的必然是实现上的复杂性。FPS 服务需要用户具有一定的编程经验，而简单支付服务对用户的编程技术几乎没有什么要求，简单支付服务流程如图 3-36 所示。

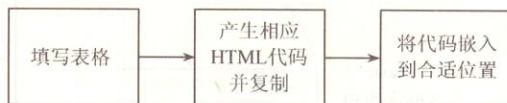


图 3-36 简单支付服务流程

从图中可以看出简单支付服务使用者只需简单地了解一些网页开发技术即可，使用者不需要编写任何代码就可以实现常用的基本支付服务。和 FPS 类似的是，使用简单支付服务也需要一个 Amazon Payments 账户及一个用于测试的 Amazon 沙盒账户。简单支付服务同样提供 FPS 中的 CBUI 来保证用户购物体验的一致性。

总的来讲，对于支付服务有着较高要求的用户可以选择 FPS，但只是简单地完成一些日常支付服务的则推荐使用简单支付服务。两种服务的适用范围不同，用户根据需要自行选择。

3.8.7 Amazon 执行网络服务

Amazon 执行网络（Fulfillment Web Service, FWS）^[32]是一个非常有用的代理订单执行网络服务，简单来说它的作用就是产品存储及销售业务的托管，也可直接理解为 Amazon 替用户销售产品。该项服务主要面向的对象是中小企业，这些企业受限于自身厂房和配送渠道，它们可以将自己的产品全权委托给 Amazon，由 Amazon 帮助其完成产品

存储及销售过程。FWS 服务分成两个部分：Inbound 服务和 Outbound 服务。Inbound 对应着用户将自己的产品运送到 Amazon 的存储中心的过程。当这些产品顺利到达后，用户就可以使用相关的 Inbound API 来管理自己的产品，发布产品相关的信息并跟踪产品的存储和销售状况。Outbound 则对应着顾客购买产品后的一系列流程。如果用户是在 Amazon 上售卖自己的产品则不用考虑 Outbound，因为 Amazon 会自动替用户完成这些功能，否则用户就需要利用 Outbound API 自行处理 Outbound 流程。FWS 流程图如图 3-37 所示。

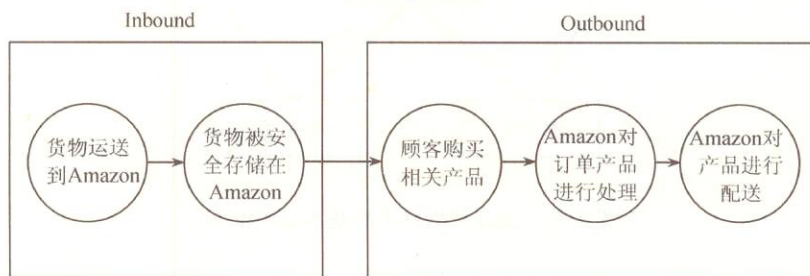


图 3-37 FWS 流程图

3.8.8 土耳其机器人

Amazon 的土耳其机器人^[31]是一个很特殊的服务，它由 Wolfgang von Kempelen 发明^[7]，名称来自于 18 世纪一种名为“The Turkwhich”的自动象棋机设备。众所周知，计算机擅长的是有着固定流程的程式化计算，而对于像写作、翻译等具有高度灵活性且无固定规律可循的任务则显得无能为力。土耳其机器人的推出就是为了解决这个问题。和 EC2 等服务聚集大量的计算机不同的是，土耳其机器人聚集的是人这种特殊的“计算工具”，所以将土耳其机器人称为“人计算”似乎更为恰当。土耳其机器人中涉及的概念主要有以下几个。

(1) Requester: 任务的发布者，可以是个人也可以是某个组织。

(2) HIT: 是 Human Intelligence Task 的简写。HIT 就是 Requester 发布的任务，HIT 有一个时间限制，在该时间内接受该任务是有效的，否则无效。同时 HIT 还规定了接受任务者完成任务的时间。

(3) Worker: 任务的接受者，对于同一个 HIT 每个 Worker 只能完成一次。

(4) Assignment: 可以用来监督 HIT 的完成情况，对于每个 Worker 都会创建一个 assignment。

(5) Reward: Worker 成功完成 HIT 后需要支付给其的奖励。

土耳其机器人的基本工作流程如图 3-38 所示，从图中可以发现，土耳其机器人的原理和国内的百度知道、天涯问答等服务有些类似。首先是任务的发布，在任务发布后，全球各地的土耳其机器人服务的使用者就会在网页上看到该任务，如果觉得各方面都合适他们会接受这个任务。任务发布者可以从中挑选适当的人来完成任务（土耳其机器人允许任务发布者事先对任务接受者的专业素质进行测试），在完成的过程中可以对其完成情况进行监督。任务完成后，只要检验合格任务发布者就需要向完成者支付事先约定好的报酬。

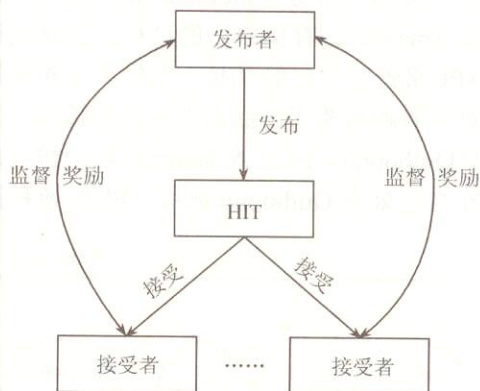


图 3-38 土耳其机器人的基本工作流程

比如用户有一篇很长的稿件需要翻译，用户可以约定好报酬和截止时间后将该任务发布到土耳其机器人的网页上。如果报酬合适则会有人接受任务，用户可以将该稿件分成好几个部分，然后指定几个人各自翻译其中的某一部分，最后只需将其合并到一起即可。Amazon 土耳其机器人将人从烦琐的工作中解放出来，使一件工作集合全球人的智慧来完成，效率得以提高，工作结果得到改进。

目前该服务的使用者还远不如 Amazon 的其他 AWS 用户，但是这项服务的想法非常独特，效果也很显著，相信在不久的将来会有更多的人使用。

3.8.9 Alexa Web 服务

Alexa^[33]公司是一家专注于世界网站排名的公司，它于 1999 年被 Amazon 收购，从而成为其全资子公司。Alexa 通过其发布的 Alexa Toolbar 来对网民的浏览习惯进行监测，安装有 Alexa Toolbar 的用户在浏览每个网页时都会自动向 Alexa 发回一串代码，对这段代码进行分析之后 Alexa 就会得到用户的访问信息，通过统计一定时间内全球网站的访问记录后对各大网站进行排名。由于 Alexa 所统计的浏览记录只是针对那些安装了 Alexa Toolbar 的用户，而 Alexa Toolbar 的用户在网民中所占的比例较小并且存在作弊的可能性，这种排名的准确性一直受到大家的置疑。但是由于目前没有更好的权威排名，Alexa 仍被用来作为衡量网站排名的标准。

其实从概率的角度来看，Alexa 所做的工作就是一个数据抽样并进行统计的过程。如果 Alexa Toolbar 的安装概率一定，这个排名结果应该还是比较准确的，Alexa 发布的排名包括综合排名和分类排名。综合排名就是该网站在全球所有的网站中的绝对排名，分类排名则是该网站在其所属分类中的排名，分类的方式有按主题分类和按语言分类。

Alexa 计算排名的方式^[7]是每个网站的访问率。访问率越高，排名越靠前。在旧的排名算法中，网站的排名主要取决于访问用户数和页面浏览数。访问用户数=（访问人数/全部 Alexa 用户数）×100%。页面浏览数是指用户访问了某个网站的页面数量。除此之外，还有其他因素影响网站排名，但这些因素所起的作用并不是很大。最近 Alexa 对其排名算法做了改进，除了访问用户数和页面浏览数之外，新算法引入了更多的排名因素以期获得

更加准确的排名。与旧的排名侧重于网站的流量相比，新的排名更加看重网站的影响力。有些流量相对较小但是在某一领域很有影响力的网站在新算法下的排名要远高于旧算法下的排名。

3.9 AWS 应用实例

3.9.1 在线照片存储共享网站 SmugMug

数亿张照片，几十万付费用户，维持这样规模的公司需要多少人呢？不同的读者可能会有不同的答案，但你绝对想不到 SmugMug^[34]给出的答案：50 人。

在公司发展的初期 SmugMug 和传统的公司一样建立自己的数据中心并通过不断添置新的 IT 设备以适应业务量的增长，但是很快就发现业务量的增长速度大大超过设备添置速度。作为一家未完全盈利的新兴公司，SmugMug 显然难以长期承受巨额的基础设施开销。最后公司选择使用 Amazon 的 S3 服务。结合公司的实际情况，SmugMug 将网站上最热门的部分照片仍旧存储在公司自己的服务器中，剩下的绝大部分照片则转移到 S3 服务器中，由 Amazon 来提供照片的安全存储。这样既能保证基础设施不会成为公司发展的瓶颈，又能节省大量成本。照片转移的过程仅仅花费一周的时间。

完成数据迁移后，由于不需再考虑基础设施问题，SmugMug 将公司的主要精力集中在提高服务质量上。目前 SmugMug 向用户提供了以下三种照片访问方式^[35]。

- (1) SmugMug 以代理的身份处理用户访问请求。
- (2) SmugMug 对用户访问请求进行重定向。
- (3) 利用有关 API 直接对存储在 S3 中的数据进行访问。

在这三种访问方式中，以第一种方式访问的用户超过 99%。也就是说几乎所有的用户都选择这种访问方式，这也正是 SmugMug 所期待的结果，因为它希望 S3 对于普通用户来说是透明的。SmugMug 公司还引入了 EC2 服务，使客户可以利用 EC2 来完成图片的在线编辑和处理。

将基础设施部分外包给 Amazon 后，SmugMug 基本架构如图 3-39 所示。几乎所有的用户都是采用直接访问 SmugMug 的方式处理照片，实际的图片处理过程对于用户是透明的。SmugMug 的系统后台则如虚线框所示。主要包括三个部分^[37]：队列服务、Amazon AWS 和控制器，目前使用的 AWS 包括 EC2 和 S3，而队列服务和控制器则由 SmugMug 提供。SmugMug 并没有采用 SQS，而是建立了自己的队列服务，控制器每隔固定的时间就会自动决定增加还是减少 EC2 实例。整个 SmugMug 的系统具有高度的智能型，绝大部分操作都会自动完成，这也是为什么 SmugMug 仅用几十人就可以完成如此巨大的工作量的原因。

总而言之，Amazon 通过提供云计算服务实现了冗余基础设施的高利用率，SmugMug 则以合理的投入解决了公司急速发展和基础设施之间的矛盾，双方达到了一个双赢的局面。

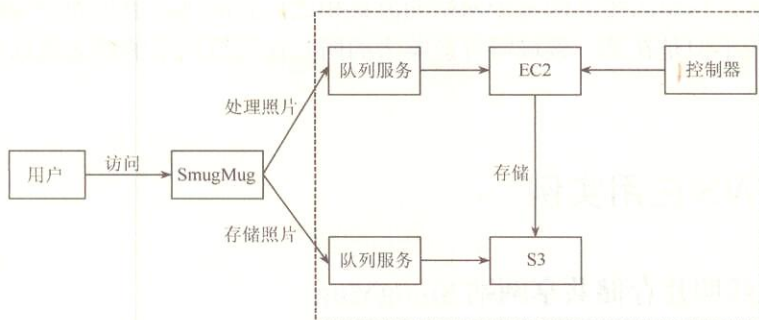


图 3-39 SmugMug 基本架构

3.9.2 在线视频制作网站 Animoto

云计算的新颖之处在于它几乎可以提供无限的廉价存储和计算能力。纽约一家名为 Animoto^[36]的创业企业已证明云计算的强大能力（此案例引自和讯网维维编译《纽约时报》2008 年 5 月 25 日报道）。Animoto 允许用户上传图片和音乐，自动生成基于网络的视频演讲稿，并且能够与好友分享，该网站目前向注册用户提供免费服务。

Brad Jefferson 最初创办 Animoto 公司时选择了一家 Web 托管服务提供商来完成公司所需的数据处理和存储信息。2008 年年初，网站每天用户数约为 5000 人，这种情况下通过 Web 托管服务完全可以满足要求。但在 4 月中旬，由于 Facebook 用户开始使用 Animoto 服务，该网站在三天内的用户数大幅上升至 75 万人。Animoto 联合创始人 Stevie Clifton 表示，为了满足用户需求的上升，该公司需要将服务器能力提高 100 倍，但是该网站既没有资金，也没有能力建立规模如此巨大的计算能力。因此，该网站与云计算服务公司 RightScale 合作，设计能够在 Amazon 的网云中使用的应用程序。通过这一举措，该网站大大提高了计算能力，而费用只有每服务器每小时 10 美分。

图 3-40^[38]是 Animoto 基本架构。和 SmugMug 类似的是，用户不能直接访问 Animoto 使用的包括 S3 在内的一系列 Amazon 服务。用户的访问方式跟未使用 AWS 之前完全一致，用户的所有操作通过 Animoto 转到 AWS 中。这样的方式也增加了创业企业的灵活性。当需求下降时，Animoto 只需减少所使用的服务器数量就可以降低服务器支出。

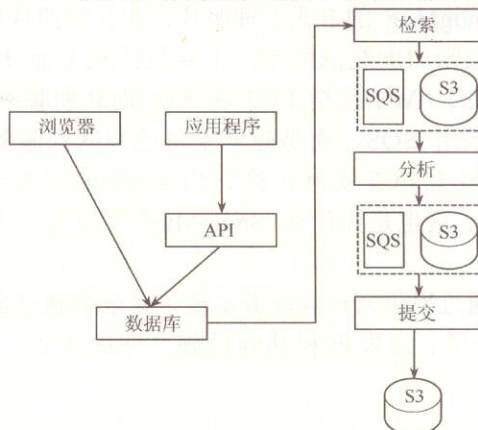


图 3-40 Animoto 基本架构

3.10 小结

Amazon 既不是操作系统开发商，也不是软件开发商，更不是 IT 设备制造商。那么 Amazon 要保持其云计算领域的优势，只能通过两个途径：① 采用开放式的架构；② 提供更为丰富的云服务，满足用户的需求。只有这样，Amazon 才可以持续保持先发优势，因此我们可以看到 Amazon 是推出云服务速度最快、也是最全的公司。对于学习，想要真正了解 Amazon 云计算，Dynamo 架构是学习和掌握的重点。Dynamo 是一个开放式的架构，开发者可以在其节点中使用其他的存储引擎，例如 S3、SimpleDB、MySQL 等其他存储系统。如果认为 Amazon 的 Dynamo 是 1.0 版本的话，那么 Cassandra 则可以认为是 Dynamo 的 2.0 版本，Cassandra 是 Dynamo 和 Bigtable 的一种结合，对 Dynamo 感兴趣的可以继续学习 Cassandra。

习题

1. 在 Hash 环中添加一个新节点 5 时，节点 3 的键值区间发生改变，如图 3-3 中所示。那么添加原先各节点保存的副本是否也要改变？如果改变，应该如何变化？
2. Merkle 哈希树的创建需要耗费较多的时间，如果频繁地进行重建就会对系统造成很大的负担，所以要尽量少地做重建工作。请简单设计一个 Merkle 树重建方案（假设 Merkle 树的叶子节点表示的是数据分区的 Hash 值）。
3. 私有 IP、公有 IP 和弹性 IP 的区别在哪里？
4. 地理区域和可用区域有哪些区别？
5. 简单存储服务 S3 与传统的文件系统有哪些区别？
6. 简单阐述 SQS 在 Amazon 云计算中的作用。
7. 如何理解传统数据库在可扩展性方面的能力较弱？
8. NoSQL 数据库是如何解决可扩展性问题的？
9. 简单阐述 Share-Nothing 架构的特点。

参考文献

- [1] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Peter Voss, Werner Vogels, Swaminathan Sivasubramanian. Dynamo: Amazon's Highly Available Key-value Store. SOSP'07, October 14-17, 2007
- [2] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In Proceedings of the Twenty-Ninth Annual ACM Symposium on theory of Computing. STOC '97. ACM Press, New York, NY, 654-663
- [3] Lamport, L. Time, clocks, and the ordering of events in a distributed system. ACM Communications, 21(7), pp. 558-565, 1978
- [4] James Murty. Programming Amazon Web Services: S3, EC2, SQS, FPS, and SimpleDB. O'Reilly, 2008

- [5] Scott Patten. The S3 Cookbook: Get cooking with Amazon's Simple Storage Service. Sopo, 2009
- [6] Jinesh Varia. Cloud Architectures, 2008. <http://jineshvaria.s3.amazonaws.com/public/cloud-architectures-varia.pdf>
- [7] Francis Shanahan. Amazon.com Mashup. Wiley, 2007
- [8] Prabhakar Chaganti. Cloud computing with Amazon Web Services, Part 1: Introduction, 2008. <http://www.ibm.com/developerworks/library/ar-cloudaws1/>
- [9] Prabhakar Chaganti. Cloud computing with Amazon Web Services, Part 2: Storage in the cloud with Amazon Simple Storage Service (S3), 2008. <http://www.ibm.com/developerworks/library/ar-cloudaws2/>
- [10] Prabhakar Chaganti. Cloud computing with Amazon Web Services, Part 3: Servers on demand with EC2, 2008. <http://www.ibm.com/developerworks/library/ar-cloudaws3/>
- [11] Prabhakar Chaganti. Cloud computing with Amazon Web Services, Part 4: Reliable messaging with SQS, 2008. <http://www.ibm.com/developerworks/library/ar-cloudaws4/>
- [12] Prabhakar Chaganti. Cloud computing with Amazon Web Services, Part 5: Dataset processing in the cloud with SimpleDB, 2009. <http://www.ibm.com/developerworks/library/ar-cloudaws5/>
- [13] Amazon. Amazon Elastic Compute Cloud Developer Guide, 2009. <http://docs.amazonweb-services.com/AWSEC2/latest/DeveloperGuide/>
- [14] Frank Bitzer. Management Framework for Amazon EC2, 2009. http://elib.uni-stuttgart.de/opus/volltexte/2009/3917/pdf/DIP_2841.pdf
- [15] Amazon. Amazon S3 Developer Guide, 2009. <http://docs.amazonwebservices.com/AmazonS3/latest/>
- [16] Amazon. Amazon SimpleDB Developer Guide, 2009. <http://docs.amazonwebservices.com/AmazonSimpleDB/latest/DeveloperGuide/>
- [17] Amazon. Amazon Simple Queue Service Developer Guide, 2009. <http://docs.amazon-webservices.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/>
- [18] Amazon. Getting Started with Amazon EC2 and Amazon SQS: Building Scalable, Reliable Amazon EC2 Applications with Amazon SQS, 2008. http://sqs-public-images.s3.amazonaws.com/Building_Scalable_EC2_applications_with_SQS2.pdf
- [19] Amazon. Amazon Elastic MapReduce Developer Guide, 2009. <http://docs.amazonweb-services.com/ElasticMapReduce/latest/DeveloperGuide/>
- [20] Amazon. Introduction to Amazon Elastic MapReduce, 2006. <http://awsmedia.s3.amazonaws.com/pdf/introduction-to-amazon-elastic-MapReduce.pdf>
- [21] Amazon. Amazon CloudFront Developer Guide, 2009. <http://docs.amazonwebservices.com/AmazonCloudFront/latest/DeveloperGuide/>
- [22] Amazon. Amazon Simple Pay Getting Started Guide, 2009. <http://docs.amazonwebservices.com/AmazonSimplePay/latest/ASPGettingStartedGuide/>
- [23] Amazon. Amazon Simple Pay Advanced User Guide, 2009. <http://docs.amazonwebservices.com/AmazonSimplePay/latest/ASPAAdvancedUserGuide/>

- [24] Amazon. Amazon DevPay Developer Guide, 2009.
<http://docs.amazonwebservices.com/AmazonDevPay/latest/DevPayDeveloperGuide/>
- [25] Amazon. Amazon Flexible Payments Service Getting Started Guide, 2009.
<http://docs.amazonwebservices.com/AmazonFPS/latest/FPSGettingStartedGuide/>
- [26] Amazon. Amazon FPS Basic Quick Start Developer Guide, 2009.
<http://docs.amazon-webservices.com/AmazonFPS/latest/FPSBasicGuide/>
- [27] Amazon. Amazon FPS Advanced Quick Start Developer Guide, 2009.
<http://docs.amazon-webservices.com/AmazonFPS/latest/FPSAdvancedGuide/>
- [28] Amazon. Amazon FPS Marketplace Quick Start Developer Guide, 2009.
<http://docs.amazon-webservices.com/AmazonFPS/latest/FPSMarketplaceGuide/>
- [29] Amazon. Amazon FPS Aggregated Payments Quick Start Developer Guide, 2009.
<http://docs.amazonwebservices.com/AmazonFPS/latest/FPSAggregatedGuide/>
- [30] Amazon. Amazon FPS Account Management Quick Start Developer Guide, 2009.
<http://docs.amazonwebservices.com/AmazonFPS/latest/FPSAccountManagementGuide/>
- [31] Amazon. Amazon Mechanical Turk Getting Started Guide, 2009.
<http://docs.amazon-webservices.com/AWSMechTurk/latest/AWSMechanicalTurkGettingStartedGuide/>
- [32] Amazon. Amazon Fulfillment Web Service Developer Guide, 2009.
<http://docs.amazon-webservices.com/AWSFWS/latest/DeveloperGuide/>
- [33] Amazon. Alexa Web Information Service Developer Guide, 2005.
<http://docs.amazon-webservices.com/AlexaWebInfoService/2005-07-11/>
- [34] <http://www.smugmug.com/>
- [35] <http://www.royans.net/arch/2007/09/19/scaling-smugmug-from-startup-to-profitability/>
- [36] <http://animoto.com/>
- [37] Don MacAskill. SkyNet Lives! (aka EC2 @ SmugMug), 2008.
<http://blogs.smugmug.com/don/2008/06/03/skynet-lives-aka-ec2-smugmug/>
- [38] Thorsten von Eicken. One Year of Scaling Rails on Amazon EC2, 2008. <http://assets.en.oreilly.com/1/event/6/One%20Year%20of%20Scaling%20Rails%20on%20Amazon%20EC2%20Presentation.pdf>
- [39] <http://cassandra.apache.org/>
- [40] Ramasubramanian, V., and Sirer, E. G. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation, San Francisco, CA, March 29 - 31, 2004.
- [41] Michael A. Olson, Keith Bostic, Margo Seltzer. Sleepycat Software, Inc. Berkeley DB.
- [42] Amazon. Amazon Relational Database Service (Amazon RDS).
<http://aws.amazon.com/rds/>
- [43] Amazon. Amazon Elastic Compute Cloud (Amazon EC2).
<http://aws.amazon.com/ec2/>
- [44] Amazon. Amazon Virtual Private Cloud (Amazon VPC).
<http://aws.amazon.com/vpc/>

- [45] Amazon. Amazon Route 53. <http://aws.amazon.com/route53/>
- [46] Amazon. Amazon CloudFront. <http://aws.amazon.com/cloudfront/>
- [47] Amazon. Amazon Simple Notification Service (Amazon SNS).
<http://aws.amazon.com/sns/>
- [48] Amazon. AWS Elastic Beanstalk.
<http://aws.amazon.com/elasticbeanstalk/>
- [49] Amazon. Amazon Simple Email Service (Amazon SES).
<http://aws.amazon.com/ses/>

第4章 微软云计算 Windows Azure

微软的商业模式建立在个人计算机（PC）时代，在网络时代软件免费商业模式的推动下，微软也推出了自己的云计算平台。微软 PDC2008 年度大会上，微软公司首席软件架构师 Ray Ozzie 隆重宣布了微软云计算战略及微软云计算服务平台——Windows Azure Service Platform，当时只允许运行在 .NET 框架下构建的应用程序。微软 PDC2010 大会公布了 Windows Azure 平台的最新版本——Windows Azure Platform，允许用户使用非微软编程语言和框架开发自己的应用程序，不但支持传统的微软编程语言和开发平台如 C# 和 .NET 平台，还支持 PHP、Python、Java 等多种非微软编程语言和架构。本章着重介绍微软的云计算操作系统、云数据库和其他两个组件，以及如何利用微软开发环境 Visual Studio 2010 开发和部署云计算应用。

4.1 微软云计算平台

传统的企业和用户在开发和部署自己的应用程序时，主要有两种方式：一种是购买和维护自己基础设施——如服务器、各种桌面软件等，这需要耗费大量的资金和维护精力；另一种是租用服务器或租用虚拟主机，这种方式大大降低了在人力和资金上的投入，但是对后台服务器的控制权也随之降低，有时会受到其他应用程序的影响。微软的云计算技术有效结合了上述两种方式的优点。云计算平台提供了可以通过互联网访问的基础设施，包括处理器、存储设施、服务等，用户也可以将他们的应用程序和数据部署在微软云计算平台上。另外，在开发运行在本地的应用程序时，用户也可以在云中存储数据或依赖其他的云计算基础设施服务。由于云计算平台依赖于微软强大的分布式集群，所以能够提供巨大的计算能力和存储能力，并具有很好的稳定性和可靠性。同时云计算平台采用量入为出的方式，用户只需按照他们动态使用的计算和存储资源来付费。所谓动态是指用户可以根据需要利用云提供商提供的巨大的数据中心和服务，轻易地扩展自己的应用程序，这个费用相比建设和维护峰值负载规模的庞大的服务器群更低，这样可以为应用程序开发商大大节约成本。

微软的云计算服务平台 Windows Azure 属于 PaaS 模式，一般面向的是软件开发商。Windows Azure 平台包括一个云计算操作系统和一系列为开发者提供的服务。当前版本的 Windows Azure 平台包括 4 个组成部分，如图 4-1 所示。

(1) Windows Azure。位于云计算平台最底层，是微软云计算技术的核心。它作为微软云计算操作系统，提供了一个在微软数据中心服务器上运行应用程序和存储数据的 Windows 环境。

(2) SQL Azure。它是云中的关系数据库，为云中基于 SQL Server 的关系型数据提供服务。

(3) Windows Azure AppFabric。为在云中或本地系统中的应用提供基于云的基础架构

服务。部署和管理云基础架构的工作均由 AppFabric 完成, 开发者只需要关心应用逻辑。

(4) Windows Azure Marketplace。为购买云计算环境下的数据和应用提供在线服务。

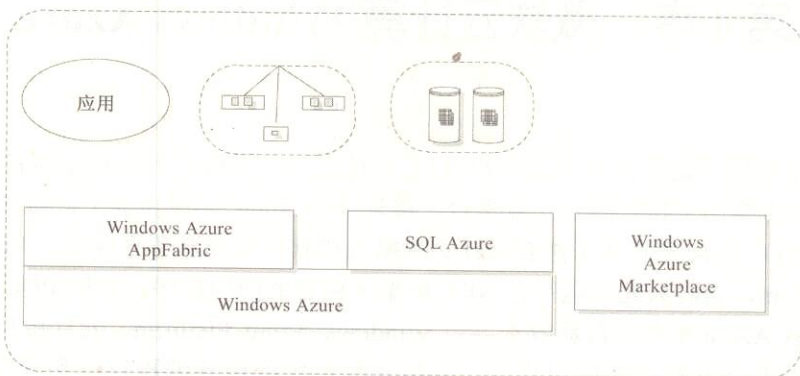


图 4-1 Windows Azure 平台体系架构

上述 4 个部分均运行在微软的 6 个数据中心。6 个数据中心分别部署在北美 (2 个)、欧洲 (2 个) 和亚洲 (2 个)。开发者能够通过云平台指定某个数据中心来运行应用程序和存储数据, 以确保这些应用程序和数据与用户在地理位置上更靠近。

虽然微软已连续发布了几个版本, 但是很多特性和服务还在不断完善和改进阶段, 相信在不久的将来微软会推出功能更加完善和强大的版本。

4.2 微软云操作系统 Windows Azure

4.2.1 Windows Azure 概述

如 Windows XP 一样, Windows Azure 是微软云计算战略的核心——云计算操作系统。不同于微软以前的战略, 即向用户提供软件, 用户在自己的机器上安装和运行这些软件, Windows Azure 是一个服务平台, 用户利用该平台, 通过互联网访问微软数据中心运行 Windows 应用程序和存储应用程序数据, 这些应用程序可以向用户提供服务。Windows Azure 提供了托管的、可扩展的、按需应用的计算和存储资源, 同时还提供了云平台管理和动态分配资源的控制手段。Windows Azure 最新版本包含 5 个部分, 如图 4-2 所示。

(1) 计算服务。计算服务为在 Azure 平台中运行的应用提供支持, 尽管 Windows Azure 编程模型与本地 Windows Server 模型不一样, 但是这些应用通常被认为是在一个 Windows Server 环境下运行的。这些应用可以在 .NET Framework 中使用 C#、Visual Basic 语言创建, 或在非 .NET 平台下使用 C++、Java 和其他语言创建。可以使用 Visual Studio 或其他开发工具, 也可以自由使用 ASP.NET、WCF (Windows Communication Foundation) 和 PHP 等技术。

(2) 存储服务。Windows Azure 存储服务主要用来存储二进制和结构化的数据, 允许存储大型二进制对象 (Binary Large Objects, Blobs), 同时提供消息队列 (Queue), 用于 Windows Azure 应用组件间的通信, 还提供一种表形式 (Table) 存储结构化数据。Windows Azure 应用和本地应用都能够通过 REST 协议访问 Windows Azure 存储服务。

(3) Fabric 控制器。Fabric 控制器主要用来部署、管理和监控应用。Fabric 控制器的作用主要是将单个 Windows Azure 数据中心的机器整合成一个整体。Windows Azure 计算和存储服务建立在这个整合的资源池上。

(4) 内容分发网络 CDN (Content Delivery Network)。CDN 的主要作用是通过维持世界各地数据缓存副本, 提高全球用户访问 Windows Azure 存储中的二进制数据的速度。

(5) Windows Azure Connect。在本地计算机和 Windows Azure 之间创建 IP 级连接, 使本地应用和 Azure 平台相连。

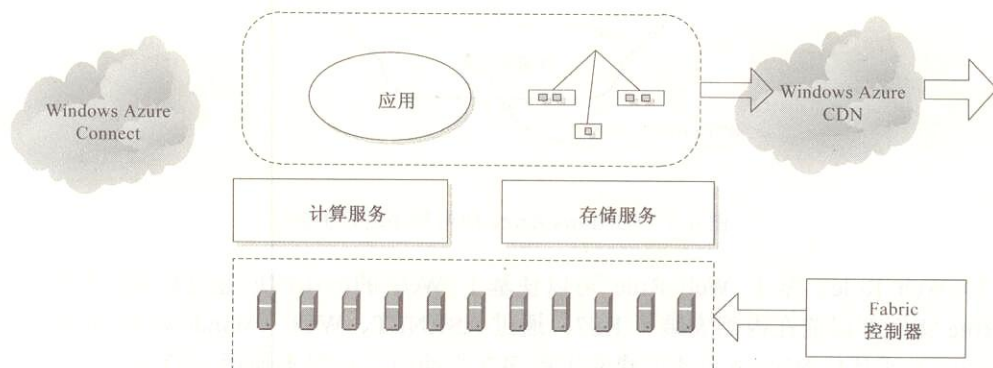


图 4-2 Windows Azure 体系架构

在后面的内容中将对上述 Windows Azure 的 5 个组成部分做具体的介绍。

4.2.2 Windows Azure 计算服务

Windows Azure 计算服务可以支持运行有大量并行用户的大型应用程序。Windows Azure 中, 每个虚拟机运行一个 64bit 的 Windows Server 2008, 这些虚拟机由微软数据中心负责维护和管理, 每个实例都运行在自己的虚拟机上。用户只关心如何构建和配置自己的应用程序, 比如决定运行实例的数量、实例运行代码区域等。用户运行自己的应用程序时, 只需通过 Web 浏览器访问 Windows Azure 入口, 使用 Window Live ID 登录 Windows Azure, 然后创建自己的运行应用程序账户或自己的存储账户, 一旦用户创建了宿主账户, 就可以加载自己的应用程序到 Windows Azure 上, 并指定应用程序要运行的实例数目。这时, Windows Azure 将自动地创建虚拟机并运行用户的应用程序。

不同于 Amazon 云计算 (用户自己提供机器的虚拟映像 (Image) 到虚拟机), Windows Azure 能够自动虚拟出虚拟机, 用户不用考虑如何维护 Windows 操作系统的备份问题, 只要专注于如何创建应用程序即可。目前, Windows Azure 服务平台的 CTP 版提供了一整套的开发工具和组件允许创建 .NET 4.0 应用程序。与传统的 .NET 应用程序不同的是, Windows Azure 应用程序包括 Web Role 实例、Worker Role 实例和 VM Role 实例, 使用这三种实例的 Windows Azure 应用程序运行机制如图 4-3 所示。

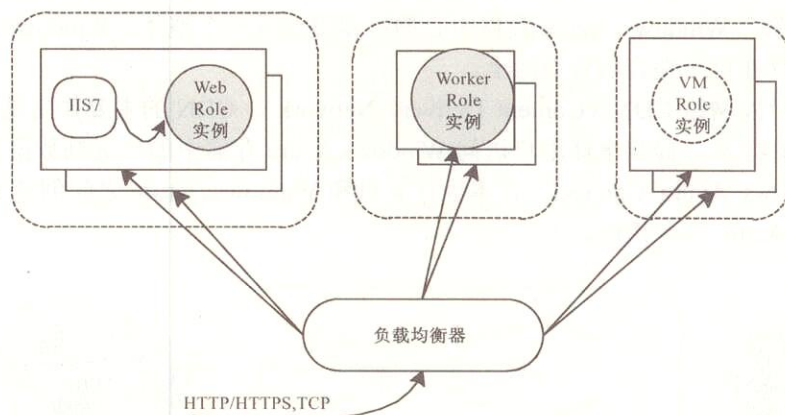


图 4-3 Windows Azure 应用程序运行机制

(1) Web Role。基于 Web Role 可以使基于 Web 的应用的创建过程变得简单。每个 Web Role 实例都提前在内部安装了 IIS7，通过 ASP.NET、WCF（Windows Communication Foundation）或其他 Web 技术使创建应用程序变得简单。如果不使用 .NET Framework，而通过本机代码创建应用，开发者可以安装或运行非微软的技术，如 PHP 和 Java。

(2) Worker Role。Worker Role 设计用来运行各种各样的基于 Windows 的代码。Web Role 和 Worker Role 的最大不同在于：Worker Roles 内部没有安装 IIS，所以 IIS 并没有托管 Worker Roles 运行的代码。比如，Worker Role 可以运行一个模拟、进行视频处理等。应用通过 Web Role 与用户相互作用，然后利用 Worker Role 进行任务处理。

(3) VM Role。VM Role 运行系统提供的 Windows Server 2008 R2 镜像。此外，将本地的 Windows Server 应用移动到 Windows Azure 中时，VM Role 将会起作用。

可以使用 Windows Azure 门户将应用提交到 Windows Azure 中，提交应用的同时，需要同时提交配置信息，告知平台每个 Role 需要运行实例的数量。Windows Azure Fabric 控制器再为每个实例创建一个虚拟机，在虚拟机中运行相应的 Role。

Windows Azure 支持 HTTP、HTTPS 和 TCP 协议，用户可以通过这些协议向 Windows Azure 发起请求。这些请求在分发给各个实例之前均会被负载均衡，同时负载均衡器不允许用户与各个 Role 实例之间保持联系，因此来自同一个用户的多种请求可能会被负载均衡器分发给不同的 Role 实例。

创建 Windows Azure 应用时，可以任意结合使用 Web Role、Worker Role 和 VM Role 实例。当应用的负载增加时，可以使用 Windows Azure 门户为应用中的 Role 请求更多的实例。如果负载减少，可以减少运行实例的数量。Windows Azure 也提供了一个 API 接口，通过程序改变运行实例的数量，不需要人工干预，但是平台本身不能根据应用的负载自动地调整应用规模。

4.2.3 Windows Azure 存储服务

Windows Azure 存储服务依靠微软数据中心，允许用户在云端存储应用程序数据。应用程序可以存储任何数量的数据，并且可以存储任意长的时间，用户可以在任何时间、任

何地方访问自己的数据。Windows Azure 存储服务目前提供了 3 种主要的数据存储结构, 即 Blob 类型、Table 类型和 Queue 类型, 如图 4-4 所示。

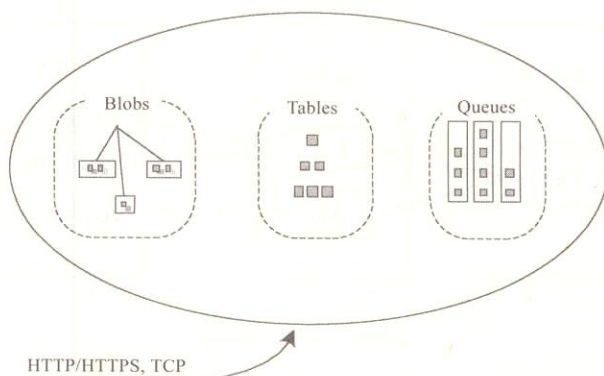


图 4-4 Windows Azure 存储服务

Blob 数据类型存储二进制数据, 可以存储大型的无结构数据, 容量巨大, 能够满足海量数据存储需求。Table 数据类型能够提供更加结构化的数据存储, 但是它不同于关系数据库管理系统中的二维关系表, 查询语言也不是大家熟悉的关系查询语言 SQL。Queue 类型的作用和微软消息队列 (MSMQ) 相近, 用来支持在 Windows Azure 应用程序组件之间进行通信。

1. Windows Azure Blob

Windows Azure Blob 用来存储大型数据对象, 用于构建重量级、可扩展的应用程序。Blob 分层分块管理数据, 这样管理数据有很多好处, 比如可以对数据建立索引, 根据一定的算法可以方便地对所要查找的数据定位, 当数据传送失败或产生错误时, 可以用最近的一个块来进行重传, 而不必传送整个 Blob。Blob 还提供了对数据信息进行描述的元数据机制, 可以更好地对数据进行管理。

1) Blob 数据模型

Blob 中的所有数据资源以 URI (统一资源标示符) 的方式标记, 它具有层次结构的命名空间, URI 形式如下:

```
http://<account>.blob.core.windows.net/<container>/<blobname>
```

对所有存储资源的访问必须通过一个存储账户, Blob 对象也不例外, 最高层为存储账户 (Storage Account), 被指定为 URI 的第一部分。一个存储账户能够拥有一个或多个容器 (Container), 每个容器拥有一个或多个 Blob, 每个 Blob 数据容量可以高达 50GB (每个 Blob 还可以分成若干 Block), 每个 Blob 还可以包含数据的元数据信息。Blob 的结构如图 4-5 所示。

对于数据的安全性, 系统提供了访问控制策略, Windows Azure Blob 对容器设置了共享策略, 目前系统提供了 Public READ 和 Private 两种策略修饰符, 前者表明该容器的内容不需授权就可以被任何应用程序访问, 后者表明只有账户的所有者可以通过授权访问。对容器中的数据, 可以建立相应的元数据, 元数据以 <name,value> 对的形式表示。

每个容器最多有 8KB 的元数据说明。Blob 包含于容器中, 同一份容器中每个 Blob 具有唯一的用字符串标识的名字。和容器一样, 它也具有相同形式的元数据。

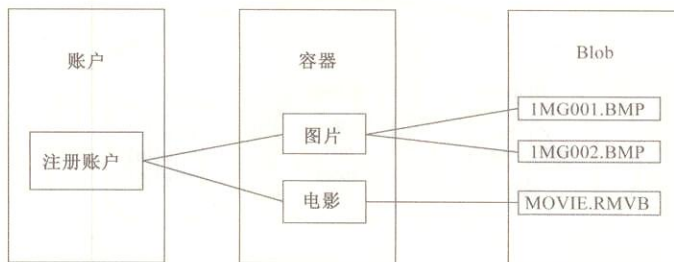


图 4-5 Blob 的结构

Blob 有以下两种形式。

(1) Block Blob。每个 Block Blob 存储容量可达 200GB。为了方便存储对象的转移, 每个 Blob 都被划分为多个 Blocks。在传输的过程中如果发生错误, 那么只需要将最近传送的 Block 重新传输, 一旦所有 Blob Block 都上传完成, 整个 Blob 能够立即被提交。

(2) Page Blob。每个 Page Blob 存储容量可以达到 1TB。Page Blob 可以被划分为很多页, 每页大小为 512 字节, 而且应用可以任意地自由读写 Blob 中的单个页。

2) Blob REST Interface

如果应用程序要访问 Blob, 可以利用系统提供的标准 HTTP REST PUT/GET/DELECT 接口, 使用这些接口可以对 Blob 执行如下 5 种操作。

- (1) PUT Blob: 插入一个新的 Blob 或替换给定的 Blob 对象。
- (2) GET Blob: 获取整个 Blob, 或使用标准 HTTP Range GET 操作获取 Blob 的指定部分。
- (3) DELETE Blob: 删除一个 Blob。
- (4) COPY Blob: 复制 Blob。主要作用是在源 Blob 和目标 Blob 之间复制一个 Blob。复制内容包括 Blob 元数据, 属性和 Block 列表。
- (5) GET Block List: 获取 Block 列表。主要作用是检索上传 Blob 的 Block 列表。每个 Blob 均有两类 Block 列表: 一类是 Committed Block List, 指已上传的作为 PutBlockList 一部分的 Block 列表; 另外一类是 Uncommitted Block List, 指当前正在提交或尚未提交的 Blocks 列表。

使用 PUT 操作请求, 一次最多可以上传 64MB 的 Blob 到云端服务器。如果用户上传大于 64MB 的 Blob, Windows Azure Blob 提供了 Block 接口来解决这一问题。该方法的主要思想是分割重组, 主要分为以下几步。

首先将需要上传的 Blob 分割成连续的若干个 Block, 例如, 一个 4GB 的电影可以划分成 1000 个 Block, 每个 4MB, 每个 Block 具有一个唯一的 ID, 如“Block 1”、“Block 2”等, 如图 4-6 所示。每个 Block 属于特定的 Blob, 因此, 同一个 Blob 的 Block 具有不同的 ID, 不同的 Blob 的 Block 可以具有相同的 ID, 每个 Block 的最大长度为 4MB, 且 Block 的大小是任意的, 同一个 Blob 的 Block 的大小不必相等。

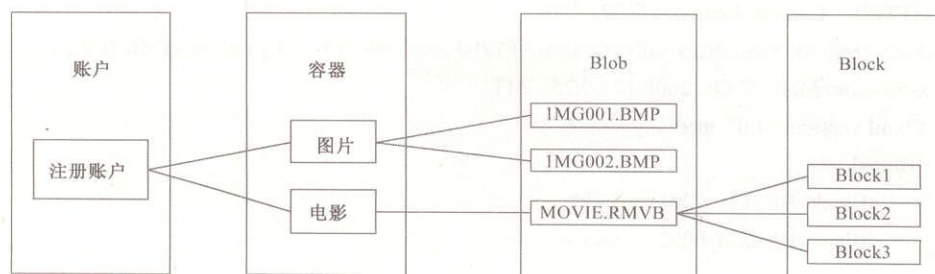


图 4-6 Blob 数据分割

Blob 数据被分割成更小的 Block 后，可以通过接口将每个 Block 上传到云端，上传时需要指明是哪个 Blob 的哪个 Block。当所有的 Block 都上传到 Windows Azure Storage 中后，提交上传的 Block 的列表信息及其所属的 Blob，系统使用这些信息将这些 Block 重组可读的 Blob，以便客户端可以使用下面形式的 URI 完整地获取该 Blob。

```
http://<account>.blob.core.windows.net/<container>/<blobname>
```

在上面例子中，当电影的所有 Block 和列表信息上传完成后，就可使用下面的 URI 来读取该电影。

```
http://Lj.blob.core.windows.net/movies /MOVIE.RMVB
```

3) Block 操作

Windows Azure Blob 提供了一系列对 Block 的操作，下面将主要介绍 PUT Block、PUT Block List 和 GET Block List。

PUT Block 操作用于上传一个 Block 到云端，应用程序可以使用一个 REST 请求执行 PUT Block 操作。下面代码展示了一个 REST PUT Block 操作。

```
PUT http://Lj.blob.core.windows.net/movies/MOVIE.RMVB
?comp=block &blockid=BlockId_0001 &timeout=60
HTTP/1.1 Content-Length: 4194304
Content-MD5: HUXZLQLMuI/KZ5KDcJPcOA==
Authorization: SharedKey MyAccount: F5a+dUDvef+PfMb4T8Rc2jHcwFK58KecSZY+l2naIao=
x-ms-date: Mon, 27 Oct 2008 17:00:25 GMT
..... Block Data Contents .....
```

上述代码用到了 PUT 这个 HTTP 动词，“?comp=block”操作表明它是一个 PUT Block 的操作，blockid=BlockId_0001 指明了 blockid。Content-MD5 设定了该 Block 数据的校验和，用于防止数据传输过程中出现错误，确保数据的完整性。Content-Length 和 Authorization 指明该 Block 的数据大小和授权码。这是一个典型的 PUT Block 操作。

上传了所有的 Block 后，只完成了其中的一步，还要将这些 Block 的列表信息（一般是 BlockID）上传到云端来说明这些 Block 的所属信息，这些列表信息具有次序性，按照这个列表的顺序可以组成分割前的可读的 Blob 版本。下面的代码说明了如何上传 Block 的列表信息。

```
PUT http://Lj.blob.core.windows.net/movies/MOVIE.RMVB
?comp=blocklist &timeout=120
```

```

HTTP/1.1 Content-Length: 161221
Authorization: SharedKey sally: QrmowAF72IsFEs0GaNcTRU143JpkfIgRTcOdKZaYxw=
x-ms-date: Mon, 27 Oct 2008 17:00:25 GMT
<?xml version="1.0" encoding="utf-8"?>
<BlockList>
  <Block>BlockId_0001</Block>
  <Block>BlockId_0002</Block>
  .....
</BlockList>

```

同上，“?comp=blocklist”指明了所上传的为 Block 的列表信息，而 PUT 命令的 HTTP 请求指明了 blocklist 上的 Block 属于该账户下的 moives 容器中的 Blob，这个 Blob 就是 MOVIE.RMVB。

上传完毕后，我们就可以从云端获取整个或部分 Blob，下面的代码用于获取给定 Blob 的全部内容。

```

GET http://Lj.blob.core.windows.net/movies/MOV1.avi
HTTP/1.1
Authorization: SharedKey sally: RGIlHMTzKMi4y/nedSk5Vn74IU6/fRMwiPsL+uYSDjY=
X-ms-date: Mon, 27 Oct 2008 17:00:25 GMT

```

如果获取数据的一部分，只需要在 GET HTTP 请求命令中使用关键词 Range 说明要获取哪部分数据，如 Range: bytes=1024000-2048000。

使用分割重组的方法向云中心上传大型数据对象具有很多优点。如果上传过程中出现错误时，只重传发生错误的 Block，而不必重传这个数据对象；并行上传这些 Block 以节省时间；上传可以是无序的，只要 blocklist 中这些 Block 的顺序是正确的，就可以保证 Blob 正确地重组。

每个 Blob 的 Block 均具有唯一性，当向同一个 Blob 上传了相同 ID 的 Block 时，系统会用后者覆盖前者。另外一种情况是，上传了一个 Block，但是在 Blocklist 中并没有说明其所属信息，由于这些数据占据大量的空间，系统将会收集这些数据，然后在规定时间内（目前设为一周）后销毁这些数据。

2. Windows Azure Table

Blob 适于存储某些无结构数据，不适于存储结构性很强的数据。传统的关系数据库是可扩展的，运行于大型机上的 DBMS 可以处理日益见长的用户信息。但是当需要支持大量的并发用户时，存储只能向外扩展，而不是按比例增加。为了能够实现上述性能，同时为了使存储机制变得简单，Windows Azure 存储提供了 Table 类型。

Table 结构如图 4-7 所示，包含数据的基本单元是具有层次结构的实体（Entity），每个实体具有若干属性（Property），一个 Table 没有固定的模式，它的属性具有各种类型，如 Int、String、Bool 或 Datetime 等。实体大小为 1MB，而且总是将实体作为单元来进行访问。当读取一个实体时，能够返回其所有的属性；当写入一个实体时，能够替代其所有的属性。在单个的 Table 中，可以按照原子的方式来更新一组实体，使得这些实体的更新要么都成功、要么都失败。

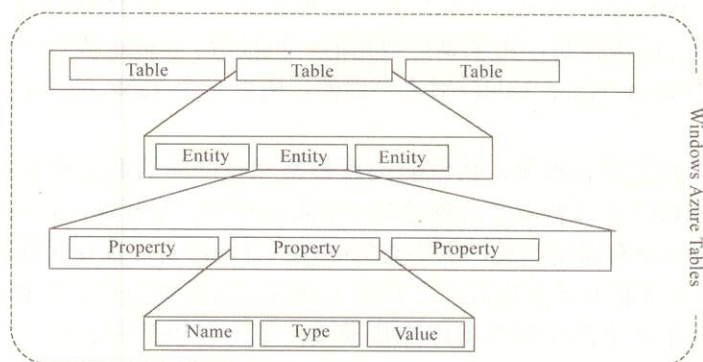


图 4-7 Table 的结构

不同于关系数据库中的二维表，不能通过 ADO.NET 访问 Table 结构，也不支持传统的 SQL 查询。一个应用程序使用 LINQ 语法书写的语句来访问表里面的数据。像 Blob 一样，一个 Table 可以包括成千上万个实体，Windows Azure 存储机制能够把 Table 分成多个部分存储在多个服务器，以提高数据访问的性能。同时微软提供了图形化的界面来创建数据库。而且，Windows Azure tables 有不同的存储风格——每个实体的属性可以有不同的类型，这些类型可以随时改变。

1) Table 数据模型

每个应用程序必须使用一个有效的账户来访问 Windows Azure 存储资源，用户通过 Windows Azure 的 Web 接口创建账户，Windows Azure 将会返回给用户一个 256 位的密码，用来授权访问存储系统。Table 的宿主名字是 <accountName>.table.core.windows.net。一旦创建了一个存储账户，就可以在该账户下创建一个或多个 Table。一个 Table 包括多个实体，一个实体最多可以包括 255 个属性。属性相当于关系表中的列，它描述一个实体的单个值，Windows Azure 为它的值提供了各种类型，如表 4-1 所示。

表 4-1 实体类型及其说明

属性类型	说明
Binary	最多可达 64KB 的字节流
Bool	布尔值
DateTime	64bit 的 UTC 时间值，值的范围是 1/1/1600 到 12/31/9999
Double	64bit 的浮点值
GUID	128bit 的全局唯一的标识符
Int	32bit 整数
Int64	64bit 整数
String	用 UTF 编码的字符串，值可达 64KB

每个 Table 由 PartitionKey 和 RowKey 两个属性一起唯一地标识一个 Table 中的实体，由于一个 Table 包含的实体很多，数据规模很大，在必要的情况下系统可以将一个 Table 分成多个分区 (Partition) 存储在不同的节点上，PartitionKey 就是用来表示每个部分的。而 RowKey 用来在每个分区中表示每个实体，系统规定 PartitionKey 和 RowKey 必须是

String 类型。在 Windows Azure 存储中,系统还为每个实体维护了一个用时间戳 (Timestamp) 表示的版本信息,用来维持数据的一致性等。Table 没有一个固定的模式,所有的属性都是以 <name,typed value> 对来存储的,所以一个 Table 的两个实体可能有完全不同的属性。

与传统数据库的关系表相比,传统的关系数据库有严谨的数据库设计模式和查询模式,往往需要在 DBMS 上耗费很多的资源对关系表进行维护和管理,Windows Azure 的这种表结构的存储机制没有固定模式,也不支持 SQL 查询语言,仅仅是简单的层次关系,在数据的管理上简单便捷且不耗资源,有利于大规模应用程序的开发中节省资源。如果应用程序中不需要使用严谨的传统数据库操作模式,完全可以使用这种方式存储数据;反之,可以利用 4.3 节介绍的 SQL 服务。

2) 分割 Table

前面在解释 PartitionKey 和 RowKey 的时候提到可以将表进行分割,通过分发 Table 的实体将 Table 扩展到数以千计的存储节点上,如何进行数据的颗粒化取决于应用的特征。比如为中国移动设计一个 Azure 版本的应用,就表存储设计来说,将整个中国移动的客户放在一个表里肯定是不现实的,必须考虑分区,PartitionKey 可以是地域属性,比如按省来分区,RowKey 是手机号码 (也可以将 PartitionKey 设定成移动业务的品牌,RowKey 是手机号码)。即使这样数据量还很大,如果将一个省的客户数据放到一个表里,实体变成了广东省移动客户,这样 PartitionKey 可以代表省下面的城市,比如广州、深圳……,RowKey 是手机号码。如果划分得更细致,可以将广州市的用户放到一个表里,实体变成了广州市移动客户,PartitionKey 变成市下面的区,比如白云区、天河区,RowKey 可以换成业务类型。甚至还可以根据具体应用进一步地细分。

在分发这些实体时,理想的情况下希望相关的一些实体保存在一个存储节点上组成一个分区,这就要选择合适的 PartitionKey 和 RowKey。程序运行时,系统将监视每个分区的使用模式,自动地在所有的存储节点上对分区进行负载均衡,使系统和应用程序实时扩展以满足 Table 的访问流量。也就是当频繁访问某些分区时,系统自动地将这些分区扩展到其他服务器上,减少访问拥塞。

在对 Table 中的实体进行划分时,分到相同分区的实体将被存储在一起,这样方便在分区中对实体查询、高效地缓存及执行优化。

选择合适的 PartitionKey 对应用程序的可扩展性非常重要。在有效查询和可扩展性之间存在一个矛盾,一个 Table 中的实体越集中对于实体的查询效率越高,而一个 Table 具有越多的实体,越不利于应用程序在服务期间进行扩展。理想的情况是最常用的或潜在必要的查询将 PartitionKey 作为自己表达式的一部分,这样将会加大查询的有效性,因为这样查询将只需要在单一的分区内进行查找,返回需要的实体。相反,如果 PartitionKey 不包含在查询表达式里面,查询将在 Table 中的所有分区中进行查询,这将影响效率。所以在确定 PartitionKey 时,一定要选取重要的属性作为 PartitionKey,精心设计,保证查询和扩展效率的统一。

3) 对 Table 编程

在 .NET 应用程序中使用 Table,开发者可以使用 ADO.NET 数据服务,Table 支持创建、获取、删除 Table (或实体)、对实体进行更新,但不能对 Table 进行更新。这些操

作可以通过 ADO.NET 数据服务的 API 来完成, 表 4-2 列出了这些 API。

表 4-2 ADO.NET API

操 作	ADO.NET 数据服务	HTTP Verb	资源	描 述
查询	LINQ Query	GET	Table	返回该存储账户下所有满足过滤条件的 Table 列表
			实体	返回指定 Table 中的所有的实体或满足过滤条件的实体的子集
更新所有的实体	UpdateObject & SaveChanges(SaveChangesOptions.ReplaceOnUpdate)	PUT	实体	更新一个实体的属性的值, 一个 PUT 操作可以替代整个的实体, 可以用来移除属性
更新部分实体	UpdateObject & SaveChanges()	MERGE	实体	更新一个实体的属性值
创建新的实体	AddObject & SaveChanges()	POST	Table	在指定的存储账户下创建一个新的 Table
			实体	在指定的 Table 中插入一个新的实体
删除实体	DeleteObject & SaveChanges()	DELETE	Table	在指定的存储账户下删除一个 Table
			实体	从指定的 Table 中删除一个实体

对于 Table 的操作还包括标页码 (Pagination) 和同时更新时产生的冲突处理等。

下面用一个例子说明如何利用 ADO.NET 数据服务的 API 编程。假如在某个存储账户下创建一个名为“NEWS”的 Table, 并在其上做相应的操作。操作步骤如下:

- (1) 定义表 NEWS 的模式;
- (2) 创建表 NEWS;
- (3) 向 NEWS 中插入一条新闻;
- (4) 获取表中新闻的列表;
- (5) 更新表中的一条新闻;
- (6) 从表中删除新闻。

在创建 Table 之前, 首先定义 Table 的模式, 即定义该表中包含哪些实体, 可以用一个 C# 类定义该模式, 代码段如下。

```
[DataServiceKey("PartitionKey", "RowKey")]
Public class NEWS
{
    //Category
    Public string PartitionKey {get; set;}
    //PostedDate
    //用户定义的属性
    Public string Title {get; set;}
    Public int ReadingNum {get; set;}
    Public string ReadingNumAsString {get;}

    Protected string Id {get;set;}
}
```

在上面的代码段中, 用 DataServiceKey 声明了表 NEWS 的主属性 (Key Property), 这意味着表 NEWS 可以按照 Category 划分成分区, 而应用程序可以按照新闻的发布日期有效地查询到每个种类的新闻。除了主属性, 上述代码还定义了额外的属性, 定义了 get

和 set 方法的公共属性可以存储在 Windows Azure Table 中，如 Title 和 ReadingNum，而 ReadingNumAsString 和 Id 则不能存储在 Windows Azure Table 中。

定义 Table 的模式后，就可以创建 Table 了。用户在申请存储账户时，系统会自动给该账户创建一个称为“Tables”的主表，该账户创建的所有其他的表必须把自己的表名注册到这个主表下。下面的代码显示如何创建一个表 NEWS。

```
[DataServiceKey("NEWS")]
Public class TableStorageTable
{
    Public string NEWS {get; set;}
    //声明存储服务账户的 URI
    String ServiceUri= "http://<Myaccount>.table.core.windows.net/"
    DataServiceContext context = new DataServiceContext(serviceUri);
    //创建表
    TableStorageTable table = new TableStorageTable("NEWS");
    //向主表"Tables"添加新创建的表 NEWS
    context.AddObject("Tables", table);

    //调用服务器保存修改
    DataServiceResponse response = context.SaveChanges();
}
```

上述代码中 ServiceUri 是 Table 服务的 URI，DataServiceContext 是 ADO.NET 数据及服务的一个主要的类，用来描述一个服务运行时上下文，它提供了使用 LINQ 或 RESTful URIs 方式向表中插入、更新、删除和查询实体的 API。

向表 NEWS 中插入一条新闻，如下面代码所示。

```
//首先创建一条新闻
NEWS news1=new NEWS
{
    PartitionKey="military", //新闻种类
    RowKey=DateTime.UtcNow.ToString(), //新闻发布时间

    Title="微软云计算平台 Windows Azure", //标题
    ReadingNum=10
}
ServiceUri=new Uri("http://<Myaccount>.table.core.windows.net")
Var Context=new DataServiceContext(ServiceUri);
Context.AddObject("News" news1);
DataServiceResponse response = context.SaveChanges();
```

应用程序可以对 Table 中的实体进行查询，对实体进行查询可以用 LINQ，下面的代码显示了如何获取被阅读了 10 次的所有新闻。

```
Var ServiceUri=new Uri("http://<Myaccount>.table.core.windows.net");
DataServiceContext context=new DataServiceContext(ServiceUri);
//构造 LINQ 查询语句，使用 DataServiceContext 查询被阅读了 10 次的新闻
Var allnews =
    From news in context.CreateQuery<NEWS>()
```



```
Where news.ReadingNum==10
Select news;
```

上述查询结果将以 XML 的形式返回要查询的实体, 应用程序可以使用返回结果。还可以更新一个表中的实体, 要更新表中的实体首先得使用 LINQ 查询获取将要更新的实体, 然后对将要更新的实体属性重新赋值, 同时更新并保存修改。代码如下所示。

```
NEWS news=
    (From news in context.CreateQuery<news>("NEWS")
     where news.PartitionKey == "military"
       && news.RowKey == "sep-1"
     select news).FirstOrDefault();
//给实体的属性重新赋值
news.title = "微软 AZURE 平台";
//将更新重新复制, 保存修改
context.UpdateObject(blog);
DataServiceResponse response = context.SaveChanges();
```

不但可以修改实体的部分属性, 还可以修改整个实体。删除表的某个实体和更新一个实体相类似, 首先获取要删去的对象, 然后使用方法 `context.DeleteObject()` 删除该对象。

上面的代码主要用到了 ADO.NET 数据服务 API, 其中最主要用到了 `DataServiceContext` 等重要的类和方法。除了使用客户端类库外, 还可以使用 REST API 进行编程。

在更新表中的实体对象时, 还必须考虑并行更新, 即多个用户同时更新某个对象时如何进行并发控制, 保证数据的一致性和准确性。更新一个实体需要两步: ①从服务器中获取要更新的实体对象, ②在本地更新该对象, 并将其发送回服务并保存更新。上述两个步骤不是原子操作, 不能保证执行的事务性。Windows Azure Table 提供了乐观并发控制 (Optimistic Concurrency) 机制解决多个进程并发更新一个实体对象的并发控制问题。Windows Azure Table 为每个实体维护了一个版本号, 当创建新的或更新一个实体时, 就给它分配一个新的版本号。当应用程序获取要更新的实体时, 服务器端将把该版本号作为 HTTP ETag 发送给客户端。当应用程序在本地更新了实体并向服务器发送保存时, 客户端将这个 ETag 发送到服务器端, 和服务器端的 ETag 进行匹配。如果匹配成功, 则服务器接收更新, 否则拒绝更新并向客户端返回“匹配前提条件错误”的信息, 更新后该实体就会被分配一个新的 ETag。当客户端应用程序获取该错误后, 它将重新获取该对象并重新执行上述操作。例如两个客户端可以执行相同的代码更新同一个实体的某个属性, 它们的执行逻辑和上面所述的相同。

```
//设置 MergeOption 为允许更新 PreserveChanges, 默认为追加 AppendOnly 不允许重写
context.MergeOption = MergeOption.PreserveChanges;
NEWS news =
    (from news in context.CreateQuery<news>("NEWS")
     where news.PartitionKey == "military"
       && news.RowKey == "sep-1"
     select news).FirstOrDefault();
news.title = "hello azure";
try
{
```



```

context.UpdateObject(news);
DataServiceResponse response = context.SaveChanges();
{
    catch (DataServiceRequestException e)
    {
        OperationResponse response = e.Response.First();
        if (response.StatusCode == (int)HttpStatusCode.PreconditionFailed)
        {
            // 重新查询该对象获取最新的 ETag,重新执行更新操作
        }
    }
}

```

在某些时候, 应用程序还可以执行某些强制的更新。执行强制更新时, 只需将获取的将要更新的实体的 ETag 设置为 “*”, 因为用它可以匹配任何的 ETag。

对于一个查询, 可能返回成千上万条结果, 为了方便查看和处理, Windows Azure Table 提供了两种机制, 一种是使用 LINQ 提供的 Take(N)方法获取前 N 条结果, 另一种是使用延续符标记下一个结果集的起始位置。

Windows Azure Table 支持返回结果的前 N 条结果。在 .NET 下, 可以使用 LINQ 的 Take(N)方法获取前 N 条结果, 代码如下所示。

```

serviceUri = new Uri("http://<account>.table.core.windows.net");
DataServiceContext svc = new DataServiceContext(serviceUri);
var allnews = context.CreateQuery<news>("NEWS");

foreach (Blog blog in allBlogs.Take(100))
{
    // do something with each blog
}

```

第二种方法使用称为 NextPartitionKey 和 NextRowKey 的延续码来获取相应的查询结果, 客户端在 HTTP 查询中嵌入相应的选项, 该查询设定相应 NextPartitionKey 和 NextRowKey 值来获取每个部分中的数据。它具有如下格式。

```

http://<serviceUri>/news?<originalQuery>&NextPartitonKey=<someValue>&NextRowKey=<someOtherValue>

```

客户端可以对所有的延续码执行这些操作, 取得相应的结果。

4) Windows Azure Table 一致性模型

为了保证数据的有效性、一致性和正确性, Windows Azure Table 为我们提供了一致性数据模型。一致性模型包括单表一致性和跨表一致性。在单个表中, 系统使用类似关系数据库的 ACID 事务特性保证所有的插入、更新、删除的事物性和排他性, 并以此来保证单个实体数据的准确性。在表的单一部分中, 系统为查询提供了事务性的快照隔离, 一个查询从开始到返回结果在一个一致的视图执行。快照保证了查询不会读出脏的数据, 因为事务不会看见其他并发执行的事务还没有提交的更改, 而只能看见查询开始之前提交的更改, 并且快照还允许在更新过程中对于数据的并行读操作。快照隔离机制只适合于单个部分的查询, 而不支持在表的多个部分或不同的延续查询。应用程序还必须保证跨表的数据一致性, 例如某个表的某些属性参照了另外表的属性, 当一个表的实体删除时, 为了保持参照完整性, 必须相应地删除其所参照的表中的实体。为了做到跨表的数据一致性, 应用

程序必须在 Windows Azure Queue 中保持一个事务，该事务保证对被参照表的更新会相应地反映到参照表中。

3. Windows Azure Queue

Table 和 Blob 主要用来存储应用程序数据，而 Queue 可以用来在应用程序各个部分如 Web Role 实例和 Worker Role 实例间进行通信。Worker Role 实例不能直接接收来自外部的访问请求，只能接收来自 Web Role 实例的信息作为自己的输入，然后将任务的执行结果返回给 Web Role 实例，在这个过程中，Queue 用来存储它们之间的交互数据。

1) Queue 通信机制

Queue 通信机制如图 4-8 所示，图中详细阐述了利用 Queue 在 Web Role 和 Worker Role 之间进行通信的详细过程，主要包括 5 个步骤。

(1) 接受任务。一般情况下，Web Role 中有多个应用实例在运行，每个实例都接受来自用户的任务。

(2) 消息入队。为了将这个任务提交给 Worker Role 实例，需要将一些消息写入对队列。

(3) 消息出队。Worker Role 实例从消息队列中读取信息，写入队列的消息通常会达到 8KB，包含一个指向 Blob 或者 Table 中实体的 URL。

(4) 任务执行。由 Worker Role 实例执行请求信息。

(5) 消息删除。Worker Role 实例执行完请求信息的任务后，队列中的消息将被删除。从队列中读取消息时，并没有将消息从队列中删除，这使得消息对于其他访问者而言有一段时间是不可见的（大约 30s）。

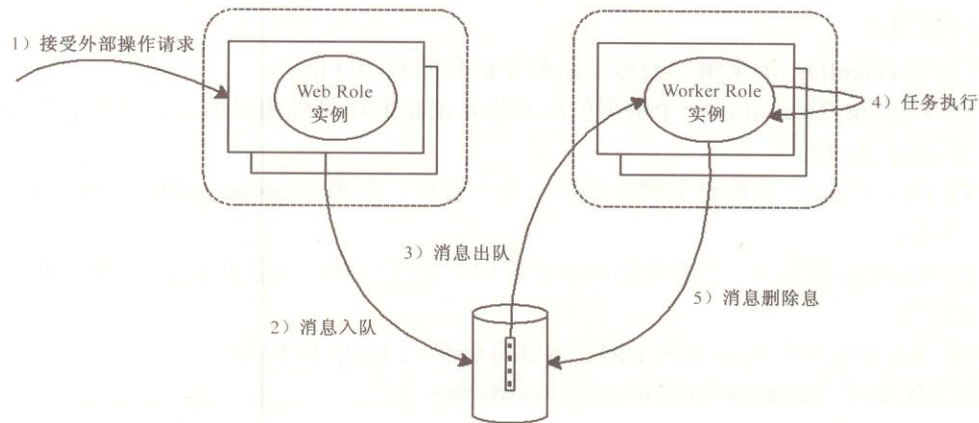


图 4-8 Queue 通信机制

2) 用 Queue 构建云端应用程序的优点

(1) 由于应用程序各个部分之间松散耦合，应用程序可以根据业务量对各个部分进行扩展。Queue 的作业数量直接反映了后台处理节点的负载大小。若 Queue 长度不断增长，则表明后台服务器处理业务逻辑能力的不足，应用程序将增加一定数量的后台节点来加快处理。相反如果 Queue 中的作业数量在某个时间段内很少，应用程序则会减少后台处理节点，以释放更多的资源供其他应用程序使用。这样通过监视 Queue 的长度调整后台处理节

点的数量可以有效地根据业务量扩展应用程序。同样,应用程序还可以根据用户的访问量实时地增加和减少处理用户请求的前端节点。另外,应用程序还可以为具有不同优先级或权重的作业设置不同的队列,并为资源池分配不同的资源,处理这些不同的作业,以便有效地使用可用资源来满足业务量的需求。

(2) 使用 Queue 可以更加灵活地构建应用程序。只要应用程序的前端和后台之间都能够理解 Queue 的消息,就可以使用不同的技术和编程语言实现应用程序的不同部分。另外,应用程序组件的改变对于其他部分是透明的。例如,可以使用完全不同的技术和语言重写一个组件,这不影响该组件无缝的和其他组件协同工作。

(3) Windows Azure Queue 提供了缓存机制来处理突发流量及应用程序组件失效。在某些时候,对于应用程序的访问量可能突然大幅度增加,而后台处理节点不能及时地处理这些突发的请求,Queue 将暂时缓存这些请求,而不是丢掉这些作业。同样,当一个后台处理服务器或程序运行失败或发生错误时,Queue 将缓存未执行的作业,直到后台服务可以处理这些作业。这增加了系统的可用性。

3) Windows Azure Queue 数据模型

同 Table 和 Blob 一样,Queue 也具有层次结构的数据模型。一旦申请了一个存储账户,可以在该账户下创建多个 Queue。一个 Queue 可以包括数目不限的消息(Message)。在 Queue 中,每个消息可以最多存储一周,超过一周被当做垃圾收集处理。在 Queue 中可以为消息创建形如<name,value>对的元数据,每个 Queue 最多可以有 8KB 的元数据,而每个消息大小也可以达 8KB。如果要存储超过 8KB 的数据,可以将这些数据存在 Table 或 Blob 中,而将其名字存储在消息中。系统还为 Queue 服务定义了如下 4 个参数。

(1) MessageID: 用来唯一的标识队列中消息的 GUID 值。

(2) VisibilityTimeout: 一个整型值用来指定消息的可见性超时值。该值最大值为两个小时,默认为 30 秒。

(3) PopReceipt: 获取消息时生成的一个字符串,用来和 MessageID 一起从 Queue 删除一个消息。

(4) MessageTTL: 一个整型值用来指明消息的生存时间(以秒表示),最大值和默认值均为 7 天。

一个指定的队列具有如下的 URI,各部分解释同 Table 和 Blob。

```
http://<account>.queue.core.windows.net/<QueueName>。
```

4) 存储队列 REST 接口

应用程序通过 HTTP REST 接口^[7]来访问 Windows Azure Queue,这些接口支持 HTTP 和 HTTPS。

Queue 具有层次结构数据模型,系统提供了对不同级别对象操作的命令。存储账户级别的 HTTP/REST 命令只有一个 List Queues 接口,该命令可以列出指定存储账户的所有 Queue。

Queue 级别的 HTTP/REST 操作命令包括四个: Create Queue、Delete Queue、Set Queue Metadata 和 Set Metadata,分别用来创建一个队列、删除一个队列、设置和更新指定队列的元数据及获取指定队列的元数据。

消息级别的 HTTP/REST 操作包括五个：PutMessage、GetMessages、DeleteMessage、PeekMessage、ClearQueue。

PutMessage 操作用来向指定的队列末尾添加一个新的消息，它包括三个参数：QueueName、Message、MessageTTL。MessageTTL 指定该消息在队列中生存时间。

GetMessages 用于获取指定队列中最前端的 N 个消息，该接口有三个参数：QueueName、NumOfMessages N 和 VisibilityTimeout T ，VisibilityTimeout T 设定了时间 T ，在该时间内，这 N 个消息将不为外界的其他操作可见。这 N 个消息的返回次序是随机的，并且每个消息可能返回多次。

DeleteMessage 用于从队列中删除消息。它包括 QueueName、MessageID、PopReceipt 三个参数，该命令删除与给定 PopReceipt 相关的消息。如果一个消息在指定的 VisibilityTimeout 时间内没有删除，它将重新出现在队列中。

PeekMessage 操作从指定队列的前端获取前 N 个消息，参数为 QueueName 和 NumOfMessages N ，不同于 GetMessages，它并不设定这些消息为不可见。该操作还为每个返回的消息分配一个 ID。

ClearQueue 操作删除指定队列的所有消息。

Queue 和 Message 级别的操作分别基于如下 URI。

`http://<account>.queue.core.windows.net/<QueueName>`

`http://<account>.queue.core.windows.net/<QueueName>/messages`

下面代码演示了向队列中添加一个消息。

```
PUT http://Lj.queue.core.windows.net/queue1/messages
? messageTtl=3600
HTTP/1.1 Content-Length: 3900
Content-MD5: HUXZLQLMuI/KZ5KDcJPcOA==
Authorization: SharedKey Myaccount: G6a+dJDvef+PfMb4T8Rc2jHcwfK58KecSZY+l2naIao=
x-ms-date: Mon, 27 Sep 2009 17:00:25 GMT
..... Message Data Contents .....
```

关键词 PUT HTTP 表明该操作向队列中添加消息，MessageTTL 指明了消息在队列中的可见间隔时间。注意 Windows Azure 的消息机制是，当从队列中读取一个消息后，此消息并不马上从队列中删除，而是变成不可见的状态，这个间隔时间是可以由 MessageTTL 设置的，默认是 30 秒，最长可达到 2 小时，在这段时间内，无法从队列中读到此条消息，只有当这条消息完全执行完毕后，该消息才会彻底从队列中删除。这样的好处是即使一条消息没有成功执行，也没有关系，过了不可见时间，它就可以再现在队列中，直到执行成功为止。其他参数与 Table 和 Blob 代码实例相同，这里不再赘述。其他的操作如获取消息等代码和上述代码相似，这里不再举例。

上面论述了 Windows Azure Storage 的三种数据存储方式，分别是 Blob、Table 和 Queue，它们为构建云计算应用程序提供了灵活和功能强大的存储服务。微软对 Windows Azure 平台提供的计算和存储资源独立收费，这意味着创建在本地运行的应用程序时也可以访问 Windows Azure 的存储服务。这使得即使 Windows Azure 应用程序不再运行时，它的数据仍然能够被其他的应用程序进行访问，提高了数据的可用性。

无论数据的存储方式是 Blob 类型、Table 类型还是 Queue 类型，保存的所有信息都被复制了 3 次。由于复制的副本不是特别重要，所以复制的副本允许容错。但是 Windows Azure

具备较强的一致性，应用程序写入数据后，会立即读取这些数据以确保能得到它所写入的数据。Windows Azure 在同一地区的另外一个数据中心中也会保存所有数据的备份副本。如果数据中心保存的主要副本不可用或被损毁，这个备份将是可访问的。

4.2.4 Windows Azure Connect

虽然云计算发展迅速，但是用户在本地的应用程序和数据可能还会继续使用，如何将本地环境和 Windows Azure 环境连接起来显得尤为重要。

Windows Azure Connect 被设计用来实现上述需求的功能。Connect 在 Windows Azure 应用和本地运行的机器之间建立一个基于 IPsec 协议的连接，使两者更容易结合起来使用，如图 4-9 所示。

图 4-9 中，当本地计算机需要连接到 Windows Azure 应用时，需要在本地计算机上安装一个终端代理。由于该技术依赖于 IPv6，所以终端代理仅对 Windows Server 2008、Windows Server 2008R2、Windows Vista 和 Windows 7 适用。Windows Azure 应用实例需要配置以方便和 Windows Azure Connect 工作。一旦这些工作完成，终端代理使用 IPsec 连接应用中的一个具体的 Role。

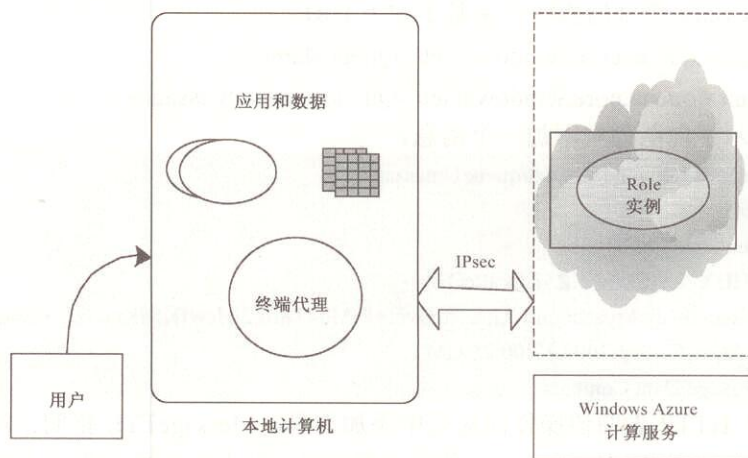


图 4-9 IPsec 连接

需要注意的是，Connect 不是一个成熟的 VPN (Virtual Private Network)，只是一个简单的解决方案。创建 Connect 并不需要与网络管理员进行约定，所有 IPsec 协议的配置工作均由 Connect 完成。一旦 Connect 创建完成之后，Windows Azure 应用中的 Roles 将会和本地的机器一样显示在同一个 IP 网络中，并允许以下两个事件。

(1) Windows Azure 应用能够直接访问本地的数据库。如某个组织机构需要将现有的由 ASP.NET 创建的 Windows Server 应用移动到 Windows Azure Web Role 中去。如果这个应用使用的数据库需要保留在本地机器上，那么 Windows Azure Connect 的连接能够使运行在 Windows Azure 上的应用正常访问本地数据库，甚至使用的连接字符串都不需要改变。

(2) Windows Azure 应用能够区域连接到本地环境。本地用户能够以单一登录的方式登录到云应用中，应用也能够使用现有活动目录账户和组织进行访问控制。

4.2.5 Windows Azure CDN

4.2.2 节介绍的 Blob 存储类型可以存储来自不同地方的访问信息。现在要将一个视频应用提供给全球的 Flash、Silverlight 或 HTML5 用户，可以使用 Blob 进行存储。

为了提高访问性能，Windows Azure 提供了一个内容分发网络 CDN (Content Delivery Network)。这个 CDN 存储了距离用户较近的站点的 Blobs 副本，如图 4-10 所示。需要注意的是，Blob 所存放容器都能够被标记为 Private 或 Public READ。对于“Private”容器中的 Blobs，所有存储账户的读写请求都必须标记。而对于 Public READ 型 Blob，允许任何应用读数据。Windows Azure CDN 只对存储在“Public READ”Blob 上的容器起作用。

用户第一次访问 Blob 时，CDN 存储了 Blob 的副本，存放的地点与用户在地理位置上比较靠近。当这个 Blob 被第二次访问时，它的内容将来自于缓存，而不是来自于离它位置较远的原始数据。

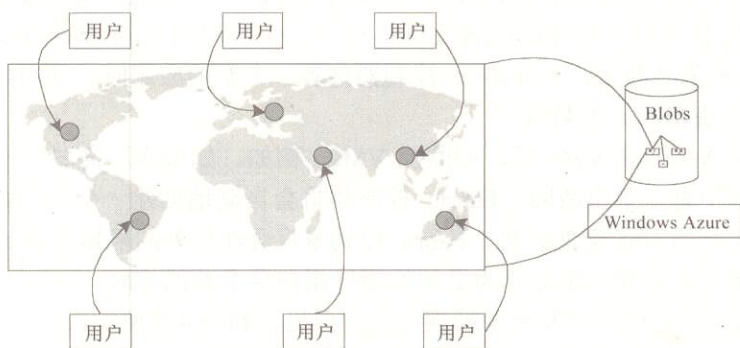


图 4-10 Windows Azure CDN 信息访问

例如，Windows Azure 提供一天体育事件的视频，第一个用户访问视频时，用户不会从 CDN 中获益，因为 Blob 还没有缓存一个离用户较近点的位置，而同一地里的其他的用户将会从 CDN 中获得更好的性能，同时缓存副本可以使视频装载得更快。

4.2.6 Fabric 控制器

Windows Azure 的所有应用和存储的数据都是基于微软数据中心的。在数据中心中，Windows Azure 的机器集合和运行在这些机器上的软件均由 Fabric 控制器控制，如图 4-11 所示。

Fabric 控制器是一个分布式应用，拥有计算机、交换机、负载均衡器等各种资源。在 Role 实例中需要安装 Fabric 代理，每台机器的 Fabric 控制器均可以与 Fabric 代理进行通信。Fabric 控制器同样也知道运行在其上的每个 Windows Azure 应用，但是数据管理和复制的详细过程对于 Fabric 控制器而言是不可知的，这是因为 Fabric 控制器将 Windows Azure 存储作为另一个应用。

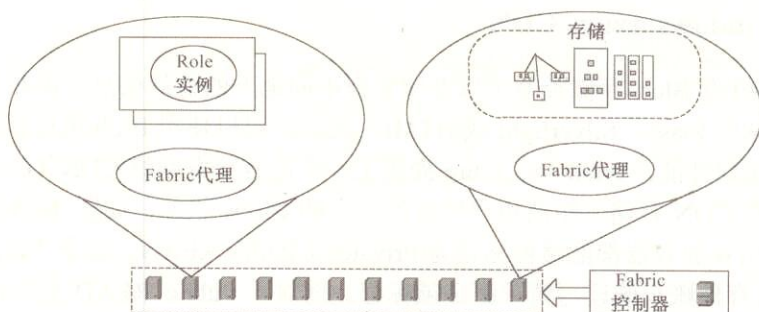


图 4-11 Fabric 控制器

Fabric 控制器作用很广，它可以控制所有运行的应用。Fabric 控制器通常依赖 Windows Azure 应用上传的配置信息决定新应用运行的位置，选择物理服务器来最优化硬件使用。这个基于 XML 描述的配置文件提供了一个应用需要的 Web Role 实例数量、Worker Role 实例的数量等。当 Fabric 控制器部署一个新的应用时，使用配置文件决定需要创建的 VMs（虚拟机）的数量。

Fabric 控制器在创建 VMs 后，还监控 VMs。例如，如果应用需要 5 个 Web Role 实例，运行的过程中有一个出故障，Fabric 控制器将会自动地创建一个新的实例。类似的，如果一个正在运行的 VM 突然宕机，Fabric 控制器将会在另外的机器上开始一个新的 Role 实例，同时重新设置负载均衡器作为必须的指针指向这个新的 VM。

Windows Azure 提供给开发者 5 种规格的虚拟机，如表 4-3 所示。

表 4-3 VM 规格说明

虚拟机规格	配置情况	存储容量
Extra-small	单核、1.0GHz CPU、768MB 内存	200GB 实例存储容量
Small	单核、1.6GHz CPU、1.75GB 内存	225GB 实例存储容量
Medium	双核、1.6GHz CPU、3.5GB 内存	490GB 实例存储容量
Large	四核、1.6GHz CPU、7GB 内存	1000GB 实例存储容量
Extra-large	八核、1.6GHz CPU、14GB 内存	2048GB 实例存储容量

其中，每个 Extra-small 实例均与其他的 Extra-small 实例共享一个处理器内核；对于其他规格的虚拟机，每个实例都有一个或多个专有的内核。这意味着应用性能是可以预计的，可执行的实例的长度是有限制的。比如在计算 π 时，Worker Role 实例可以将 π 的值精确计算到一百万位。

对于 Web Role 和 Worker Role 而言，Fabric 控制器能够管理他们每个实例中的操作系统，包括更新操作系统补丁和其他操作系统软件。这使得开发者只关心开发应用的过程，而不需要管理平台本身。对于每个运行的 Role 而言，Fabric 控制器总是假设至少有两个实例运行，这样关掉其中的一个来更新软件不会导致整个应用关闭。需要注意的是，在任何 Windows Azure Role 上只运行一个实例不是一个好的选择。

4.2.7 Windows Azure 应用场景

为了更好地了解 Windows Azure, 这里先介绍一些 Windows Azure 的应用场景。这些场景包括创建可伸缩的 Web 应用、创建并行计算应用、创建后台处理的 Web 应用及本地或托管应用的云存储应用。

1. 创建可伸缩的 Web 应用

在传统环境下, 用户通常使用本地或托管机构内部的数据中心运行供 Internet 访问的 Web 应用。与传统的 Web 技术相比, Windows Azure 为可伸缩的应用和数据提供了内在的支撑, 可以处理更加大的负载。当用户使用 Windows Azure 发布 Web 应用后, Web 应用可以处理大量并发用户的请求。

有些应用的负载变化十分显著, 例如在线售票系统、视频网站等。在传统的数据中心运行这类应用需要保留大量的机器来应对访问的高峰时间段, 而在一般情况下这些机器是空闲的。如果在 Windows Azure 上创建这类应用, 组织机构可以根据需要扩充和缩减实例的数量。用户可以使用 Web Roles 和 Tables 在 Windows Azure 上创建一个可伸缩的 Web 应用, 如图 4-12 所示。

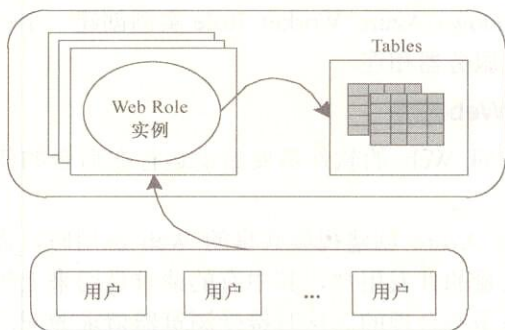


图 4-12 可伸缩的 Web 应用

上述实例中, 客户端是浏览器, 用户可以使用 ASP.NET 或其他 Web 技术实现应用逻辑, 也可以使用 WCF 创建一个可伸缩的 Web 应用。在这两种情况下, 均需要指定 Web Role 运行的实例数量, Windows Azure Fabric 控制器创建相应数量的虚拟机。正如 4.2.6 节中所描述的, Fabric 控制器也可以监控这些实例以确保请求的数量总是可用的。对于数据存储而言, 应用使用 Windows Azure Tables 进行存储。Windows Azure Tables 提供了一个可扩展的存储, 能够处理大量的数据。

2. 创建并行计算应用

对于银行金融建模、电影特技效果渲染、医药公司的新药开发等这些应用, 往往需要使用并行计算。在传统的情况下, 采用一个大的集群满足这个偶尔的并行计算需求的代价很大。Windows Azure 可以为用户提供一种按需使用的计算集群, 在这种情况下, 利用 Windows Azure 创建并行计算应用显得十分有优势。

用户选择使用 Worker Roles 创建并行计算应用, 并行计算应用通常需要使用很大的数据集, 这些数据集通常存放在 Windows Azure Blobs 中, 如图 4-13 所示。

图 4-13 中, 许多的 Worker Role 实例同时运行并行的作业。Windows Azure 对实例的运行时间施加影响, 每个实例都能够运行任意数量的作业。为了能够与应用相互作用, 用户依赖于一个单独的 Web Role 实例。通过这个接口, 用户能够决定应该需要运行多少个 Worker 实例, 同时用户还可以决定开始和停止的实例、查看访问结果等。Web Role 实例和 Worker Role 实例之间的通信依赖于 Windows Azure Queues。

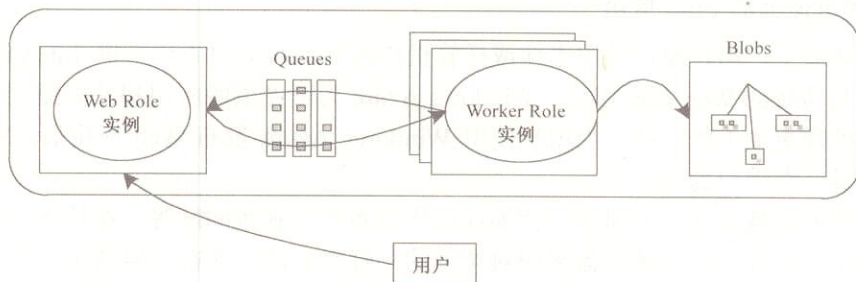


图 4-13 创建并行计算应用

云中提供了大量可用的处理能力, 这很有可能改变高性能计算。比如微软 Windows HPC 服务器允许使用 Windows Azure Worker Role 实例创建一个计算集群, 从而取代本地物理服务器或与本地物理服务器相连。

3. 创建后台处理的 Web 应用

在很多情况下, 可访问 Web 的软件需要启动运行在后台的任务, 把应用的请求和响应部分独立开来。

例如, 利用 Windows Azure 创建视频共享的 Web 应用时, 需要接受来自浏览器的请求, 这些请求一般来自大量的并发用户, 其中有的请求是用来上传视频, 上传视频的过程中, 让用户处于等待状态是不合理的。这时接受浏览器请求的应用应该能够执行一个后台任务完成上传视频的操作。这里结合使用 Web Roles 和 Worker Roles 描述上述应用的创建过程, 如图 4-14 所示。

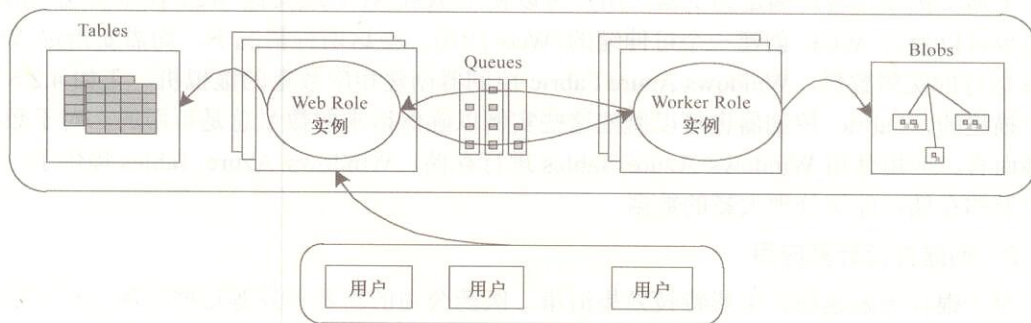


图 4-14 创建后台处理应用

与可伸缩的 Web 应用一样, 创建后台处理的 Web 应用需要使用多个 Web Role 实例来处理用户的请求。为了支撑大量的并发用户, 它使用 Tables 存储基本信息。在应用执行的

过程中, Queue 将任务传递给 Worker Role 实例, 然后利用 Worker Role 实例进行后台处理, 视频存放在 Blob 中。

4. 创建本地或者托管应用的云存储

当本地应用或者托管应用需要存储大量数据时, Windows Azure 为这些应用提供云存储服务。

(1) 本地应用云存储。一些公司需要存储所有过去的电子邮件, 此时所考虑的问题是如何节约成本, 同时确保这些邮件是可以被访问的。

(2) 托管应用云存储。托管主机上的网站需要一个可伸缩的、全球范围内都能够访问的存储位置来保存大量的文本、图片、视频和用户的配置文件信息。

Windows Azure 存储服务可以为上述两种应用提供云存储服务, 如图 4-15 所示。

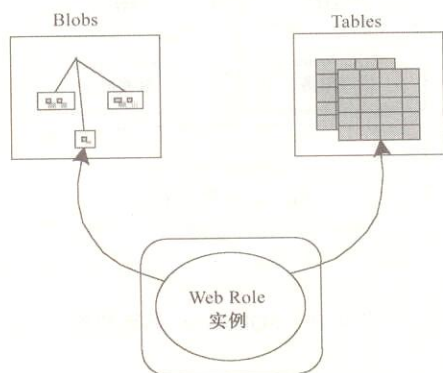


图 4-15 创建本地或托管应用的云存储

本地或者托管应用能够直接访问 Windows Azure 存储, 访问速度比访问本地存储慢, 但可以使存储变得更便宜、更具有可伸缩性、更可靠。

4.3 微软云关系数据库 SQL Azure

4.3.1 SQL Azure 概述

SQL Azure 是微软的云中关系型数据库, 是基于 SQL Server 技术构建的, 主要为用户提供数据应用。SQL Azure 数据库简化了多数据库的供应和部署, 开发人员无需安装、设置数据库软件, 也不需要进行数据库补丁升级或数据库管理。同时, SQL Azure 还为用户提供了内置的高可用性和容错能力。

SQL Azure 提供了关系型数据库存储服务, 包含三个部分, 如图 4-16 所示。

1) SQL Azure 数据库

SQL Azure 数据库提供了一个云端的 DBMS, 这使得本地应用和云应用可以在微软数据中心的服务器上存储数据。和其他的云技术一样, 用户按需付费, 最主要的费用是操作费用, 而不是磁盘和 DBMS 软件投入的费用。

2) SQL Azure 报表服务

SQL Azure 报表服务 (SQL Azure Reporting) 是 SQL Server Reporting Service (SSRS) 的云化版本。主要是用 SQL Azure 数据库提供报表服务, 允许在云数据中创建标准的 SSRS 报表。

3) SQL Azure 数据同步

SQL Azure 数据同步 (SQL Azure Data Syn) 允许同步 SQL Azure 数据库和本地 SQL Server 数据库中的数据, 也能够在不同的微软数据中心之间同步不同的 SQL Azure 数据库。

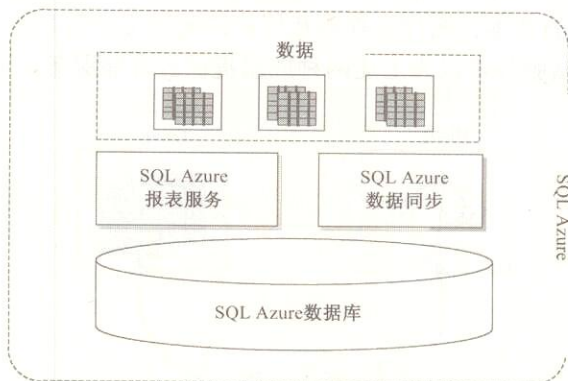


图 4-16 SQL Azure 体系架构

4.3.2 SQL Azure 关键技术

在本地数据库中, DBMS 起到了十分重要的作用。对于用户而言, 管理功能在云中实现起来也非常困难, 微软通过 SQL Azure 数据库在云中实现了这一功能。在 SQL Azure 中, 数据库规模的扩展也由 SQL Azure 数据库完成。SQL Azure 除了提供 SQL Azure 数据库服务外, 还提供报表服务和数据同步服务。

1. SQL Azure 数据库

SQL Azure 数据库是 SQL Azure 的一种云服务, 提供了核心的 SQL Server 数据库功能, 基本架构如图 4-17 所示。SQL Azure 数据库支持 TDS 和 Transact-SQL (T-SQL), 用户可以使用现有技术在 T-SQL 上进行开发, 还可以使用与现有的本地数据库软件相对应的关系型数据模型。SQL Azure 数据库提供的是一个基于云的数据库管理系统, 能够整合现有工具集并对应用户的本地软件。

图 4-17 中, 在创建一个部署在 Windows Azure 的应用中, 用户使用了 SQL Azure 数据库, 这个应用可以运行在企业数据中心或移动设备上。上述应用通常使用 TDS (Tabular Data Stream, 表型数据流) 或 Odata 协议来访问本地的 SQL Server 数据库, 因而 SQL Azure 数据库应用能够使用任何现有的 SQL Server 客户端, 这些客户端包括 Entity Framework、ADO.NET、ODBC 和 PHP 等。SQL Azure 和 SQL Server 看起来并无太大的差别, 也可以使用 SQL Server 中的大量工具, 比如 SQL Server Management Studio、SQL Server Integration Services 和大量数据副本备份的 BCP。

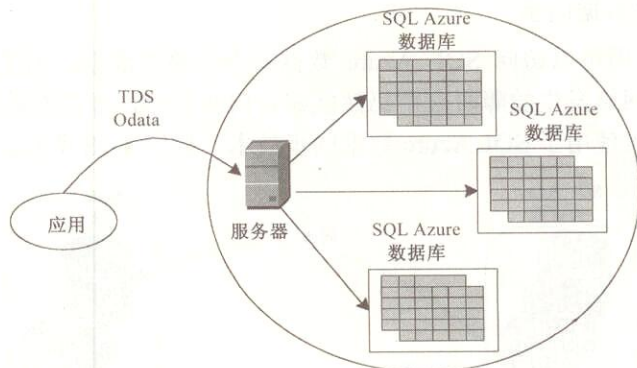


图 4-17 SQL Azure 数据库

每个 SQL Azure 账户都拥有一个或多个逻辑服务器，这些逻辑服务器可以组织账户数据和账单，但这些服务器并不是真正意义上的 SQL Server 实例。每台服务器都拥有多个 SQL Azure 数据库，每个 SQL Azure 数据库均可以达到 50GB 的大小。用户可以自由地使用 SQL Azure 数据库，能够在某个 SQL Azure 数据库中存放另一个数据库的快照以实现整个数据库的备份。

SQL Azure 与 SQL Server 有一些差别。SQL Azure 省略了 SQL Server 中的一些技术点，比如 SQL CLR (Current Language Runtime, 公共语言运行时)、全文本搜索技术等。用户没有底层管理功能，所有管理功能都由微软实现。这样用户不能直接关闭自身运行的系统，也不能管理运行应用的硬件设施。但是，相比于 SQL Server 所提供的单个实例而言，SQL Azure 运行环境比较稳定，应用获取的服务也比较健壮。出于可靠性的考虑，SQL Azure 数据库与 Windows Azure 存储服务一样，存储的所有数据均备份了 3 份。

2. SQL Azure 报表服务

用户使用 SQL Azure 数据库存储数据时，通常需要 SQL Azure 数据库支持报表功能。在 SQL Azure 中，SQL Azure 报表服务实现了这一个功能，它是基于 SQL Server 报表服务 (SSRS, SQL Server Reporting Services) 实现的。

现在 SQL Azure Reporting 主要有两个使用场景。第一，SQL Azure 报表创建的报表可以发布到某一个门户上，云端用户可以访问这个门户的报表，也可以通过 URL 地址直接访问报表；第二，ISV (Independent Software Vendor, 独立的软件开发商) 能够嵌入发布到 SQL Azure 报表门户的报表，这些门户来自于不同的应用，包括 Windows Azure 应用。ISV 可以使用 Visual Studio 标准的 ReportViewer 控制，这与将本地报表嵌入到应用中没有任何差别。

SQL Azure 报表服务与存储在 SQL Azure 数据库中的数据相互作用。SQL Azure 使用的报表可以通过 Business Intelligence Developer Studio 创建。SQL Azure Reporting 与 SSRS 的报表格式是相同的，都使用微软定义的 RDL (Report Definition Language)。

需要注意的是，SQL Azure Reporting 并没有实现本地情况下 SSRS 提供的所有的功能。比如，当前的 SQL Azure Reporting 并不支持调度和订阅功能，这使得报表每隔一定的时间将会运行和分发一次。

3. SQL Azure 数据同步

Internet 上的应用可以访问 SQL Azure 数据库中存储的数据。为了提高存储数据的访问性能，同时确保网络发生故障时应用仍然能够访问数据库，需要在本地拥有 SQL Azure 的数据库副本，微软使用了 SQL Azure 数据同步技术，如图 4-18 所示。

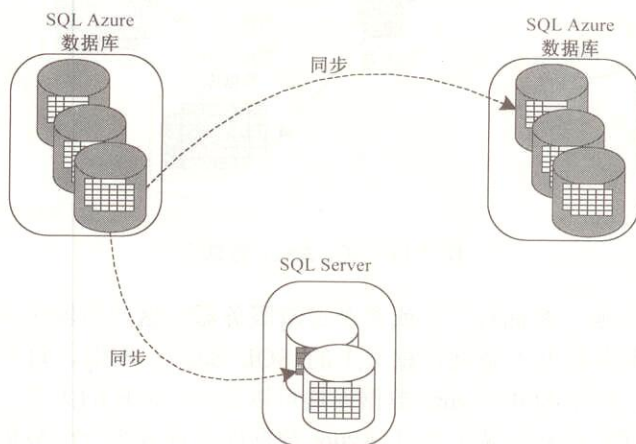


图 4-18 SQL Azure 数据同步

该技术主要包括一下两个方面。

(1) SQL Azure 数据库与 SQL Server 数据库之间的数据同步。用户选择这类同步的原因有很多，除了前面提到的网络故障等因素外，数据调度也需要数据副本在某一区域范围内进行，同时需要防止某些操作失误所带来的数据丢失。这时用户可以通过 SQL Azure 数据库与 SQL Server 数据库的信息同步在本地数据库保存副本。

(2) SQL Azure 数据库之间的同步。某些 ISVs（独立的软件开发商）或全球化的企业需要创建一个应用，为了满足高性能的需求，应用的创建者也许会选择在三个不同的 Windows Azure 数据中心（比如，北美数据中心、欧洲数据中心、亚洲数据中心）运行这个应用。如果这个应用将数据存放在 SQL Azure 数据中，需要使用 SQL Azure 数据同步服务保持三个数据中心之间的信息同步。

SQL Azure 数据同步服务使用“轮辐式（hub-and-spoke）”模型，所有的变化将会首先被复制到 SQL Azure 数据库“hub”上，然后再传送到其他“spoke”上。这些“spoke”成员可以是一个 SQL Azure 数据库，也可以是本地 SQL Server 数据库。上述的同步过程可以同步整个数据库，也可以只同步有更新的数据库表格。

4.3.3 SQL Azure 应用场景

SQL Azure 可以使用在很多地方。下面简要地介绍 SQL Azure 的 4 种应用场景。

1. Web 应用

对于大部分 Web 站点而言，用户输入和电子商务交易的数据都需要使用数据库进行存储。在传统的状况下，数据驱动的 Web 站点通常会在数据中心中放置一个数据库服务器作为 Web 服务器。

SQL Azure 提供了高可用并具有容错性能的数据库服务。在这种情况下, Web 开发者能够选择存放数据的地点。对于部门应用场景而言, 可以将 Web 应用托管在自己的服务器上, 也可以托管在第三方的服务器上, 然后再通过 Internet 访问 SQL Azure 中的数据。为了防止潜在的性能问题、减少应用的复杂度, 可以考虑将 Web 站点托管在 Windows Azure 平台上, 数据库可以互相定位。

2. 部门级应用

在一些大型的组织中, 要求数据库服务器具有容错的功能保证服务不中断。这些组织通常采用 RAID 技术和集群服务器。SQL Azure 可以重新组织不同种类的数据库。当使用 SQL Azure 为小型部门应用提供服务时, 用户能够获得数据库的管理能力, 还可以进行容错设计。

当将本地的客户端服务器应用迁移到 SQL Azure 中时, 服务提供商可以选择将客户端应用留在本地环境中, 仅将数据移动到云中。当使用这种方式设计时, 服务提供商必须考虑 Internet 连接的一些潜在问题, 这必然会使得客户端的代码变得十分复杂。解决上述问题最好的办法就是将控制逻辑移动到 Windows Azure, 这样数据访问代码和数据本身都存放在同一个数据中心中。在 Windows Azure 中, 服务提供商可以为它所面向的浏览器用户提供一些基于 Web 的用户接口, 用户也可以使用 ADO.NET 数据服务为桌面用户接口创建服务, 这些创建的用户接口具有 SOAP、REST 或 JSON 接口的特性。

3. 数据集应用

在一个数据集应用场景中, 通常希望远程用户和移动用户能够通过使用同一个数据集而集合起来。比如有一家保险公司, 用户数量不确定, 均分布在北美。在整个保险销售团队中保持顾客和保险价格数据同步是非常重要的。保险公司一般有两个要求: ① 保持每个销售人员便携式计算机能够获悉最新的价格信息; ② 保证整个系统拥有每个销售人员手中的最新用户信息。

生产数据和顾客数据都被存放在数据中心的中央 SQL Server 数据上, 销售员工使用自己的便携电脑运行应用并在 SQL Server Express 中存放数据。销售人员访问本地数据中心时需要通过数据中心外部的防火墙, 而出于访问安全的考虑, 这些防火墙并不能随意打开。在上述情况下, 可以选择提供了安全和同步考虑的 SQL Azure。使用 SQL Azure 时有三个任务, 如图 4-19 所示。

- (1) 在 SQL Azure 中创建一个数据库用来存储产品数据和顾客数据;
- (2) 在数据中心中创建一个 Sync Framework 提供者。Sync Framework 提供者可以保持数据中心和 SQL Azure 中产品和用户数据同步。
- (3) 为销售人员创建一个二级的 Sync Framework 提供者。这个二级 Sync Framework 提供者可以保持销售人员和 SQL Azure 数据集上产品和用户数据的同步。

4. “软件+服务”应用

ISVs 通常都具有较好的软件开发能力, 他们拥有开发基础架构的能力。因而, ISVs 可以使用 SQL Azure 提供“软件+服务”解决方案, 这些供应商称为 S2 (Software and Services) 供应商。而 Windows Azure 提供了一个理想环境用来托管软件服务, 这使得 ISVs 不需要考虑如何维持托管环境的基础架构。

金融、政府机关、医疗和房地产等行业通常需要存储大量的历史数据，S2 供应商可以提供比较好的支撑。

S2 供应商通常结合使用 SQL Azure 和 Windows Azure，他们会注册 Windows Azure 账户用来上传不同格式的文件，比如 E-mail 等。同时 S2 供应商还会使用 SQL Azure 账户存储结构化的数据。

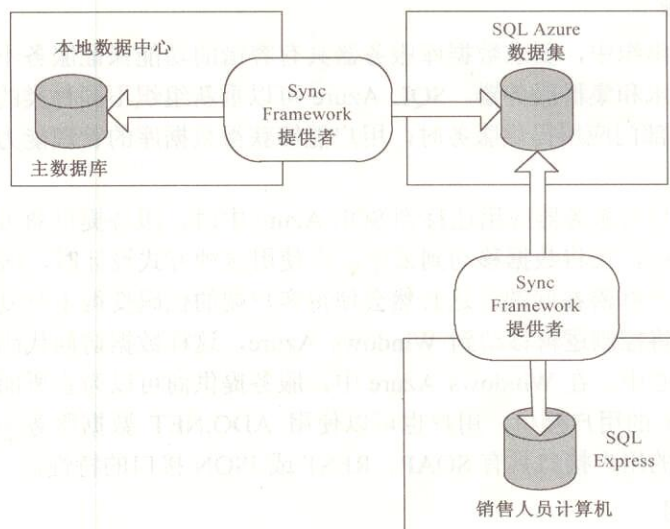


图 4-19 保险行业数据集应用

4.3.4 SQL Azure 和 SQL Server 对比

SQL Azure 是云中的关系数据库，和本地的 SQL Server 数据库有很多相似的地方。比如 SQL Azure 提供了一个表格数据流（Tabular Data Stream, TDS）接口供基于 Transact-SQL 的数据库进行访问，这和 SQL Server 中的实例访问数据库情况是相似的。SQL Azure 和 SQL Server 之间也有一些不同之处。在 SQL Azure 中，由于物理管理工作是由微软进行的，所以在管理、服务提供、Transact-SQL 支持和编程方式等方面，与 SQL Server 有所不同。

1. 物理管理和逻辑管理

SQL Azure 在管理上突出强调了物理管理，DBA（Database Administrator，数据库管理员）在管理 SQL Azure 数据应用方面仍然发挥着很积极的作用。DBAs 管理模式创建、统计、索引优化、查询优化，同时还进行安全管理（包括登陆安全、用户安全和创建角色的安全等）。

SQL Azure DBA 和 SQL Server DBA 在物理管理方面存在很大的差异。SQL Azure 能够自动复制所有存储的数据以提供高可用性，同时 SQL Azure 还可以管理负载均衡、故障转移等功能。

用户不能管理 SQL Azure 的物理资源。比如用户不能指定数据库索引所在的物理硬盘或者文件组，物理资源是由微软自行管理。同样，由于无法访问计算机文件系统，SQL

Azure 不能使用 SQL Server 备份机制，所有的数据都是自动复制备份的。

2. 服务提供

在部署本地 SQL Server 时，需要准备和配置所需要的硬件和软件，这些工作一般由 DBA 或 IT 部门完成。而使用 SQL Azure 时，这些任务均由 SQL Azure 服务程序来执行。

当用户在 Windows Azure 平台上创建了一个账户后，用户便可以使用 SQL Azure 数据库，同时还可以访问所有提供的服务，比如 Windows Azure、.NET 服务和 SQL Azure 等服务。通过这些服务可以创建和管理用户的订阅。

每个 SQL Azure 订阅都会绑定到微软数据中心的某个 SQL Azure 服务器上。在 SQL Azure 服务器上通常定义了一组数据库。为了提供负载均衡和高可用性，SQL Azure 服务器上的数据库通常会在数据中心其他物理机上进行备份。

3. Transact-SQL 支持

大多数 SQL Server Transact-SQL 语句都有一些参数，用户通过这些参数可以指定文件组或物理文件的路径。由于这些参数依赖于物理配置，在 SQL Azure 中由微软进行物理资源的管理，因而这些类型的参数并不适用于 SQL Azure。

4. 特征和类型

SQL Azure 不支持 SQL Server 的所有特征和数据类型。在现今版本的 SQL Azure 中，不支持分析、复制、报表和服务代理等服务。

SQL Azure 提供物理管理，会锁住任何试图操作物理资源的命令语句，比如 Resource Governor、文件组管理和一些物理服务器 DDL 语句等。另外还有一些操作是不允许的，比如设置服务器选项和 SQL 追踪标签、使用 SQL Server 分析器或使用“数据库引擎优化顾问”。

4.4 Windows Azure AppFabric

4.4.1 AppFabric 概述

Windows Azure AppFabric 为本地应用和云中应用提供了分布式的基础架构服务，使用户本地应用与云应用之间进行安全联接和信息传递，让在云应用和现有应用或服务之间的联接及跨语言、跨平台、跨不同标准协议的互操作变得更加容易，并且与云提供商或系统平台无关。AppFabric 目前主要提供互联网服务总线（Service Bus）、访问控制（Access Control）服务和高速缓存服务，如图 4-20 所示。

Windows Azure AppFabric 的所有部件都是在 Windows Azure 的基础上创建的（尽管 AppFabric 并没有为所有的 Windows Azure 应用提供服务），其部件描述如下。

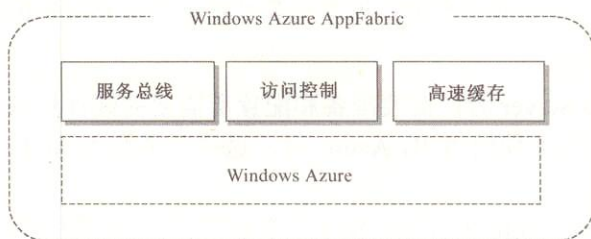


图 4-20 Windows Azure AppFabric 体系架构

1. 服务总线

服务总线的目标是通过云中应用公开的终端使公开应用服务变得简单，这个终端是可以被其他应用（无论是本地应用还是云应用）访问的。每个公开的终端都被分配了一个 URI，用户可以通过这个 URI 来定位和访问服务。服务总线同样能够处理网络地址转换所带来的挑战，并且可以在没有打开新的公开应用端口的情况下通过防火墙。

2. 访问控制

用户可以通过很多种方法获得一个数字身份认证，包括 Active Directory、Windows Live ID、Google Account、Facebook 等。如果一个应用希望注册带有其中的一种数字身份认证，那么这个应用的创建者为了支撑这个身份认证将面临很多严峻的挑战。AppFabric 访问控制服务简化了这一工作，同时也定义了一定的规则来控制用户的访问。

3. 高速缓存

在很多情况下，应用需要重复访问存取同一个数据。为了提升这类应用的访问速率，可以缓存这些经常被访问的信息，从而减少应用查询数据库的次数。高速缓存服务实现了上述功能，提高了应用的访问效率。

4.4.2 AppFabric 关键技术

Windows Azure AppFabric 为应用提供了各种各样的基础架构，用户可以从这些基础架构上获益，AppFabric 的关键技术就是服务总线、访问控制和高速缓存这三个部件。

1. 服务总线

运行在组织内部的应用提供了 WCF 创建的 Web 服务，此服务可以连接到组织外部的软件上。软件通常运行在 Windows Azure 这类云平台或其他组织内部。

在具体实现服务连接的过程中会出现很多问题。比如其他组织内部的客户端连接到 Web 服务时，需要知道如何定位到服务的终端。其他组织的软件请求需要确定如何通过服务端的服务。网络地址转换（Network Address Translation, NAT）是十分普遍的，应用对外通常不会有一个固定的 IP 地址。那么，在没有使用 NAT 的情况下，请求需要确定如何通过防火墙。

AppFabric 中，服务总线（Service Bus）解决了这些问题，如图 4-21 所示。一个 WCF 服务可以通过服务总线注册终端，然后由客户端发现和使用这些终端访问服务。

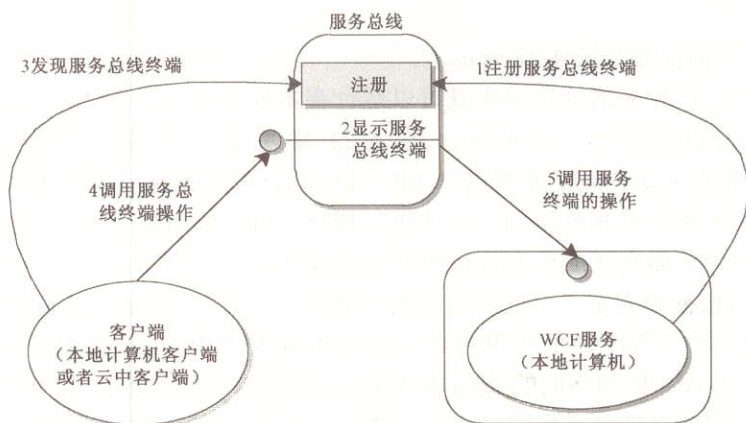


图 4-21 AppFabric 服务总线

开始时，WCF 服务注册一个或多个服务总线的终端（图 4-21 中步骤 1）。对于每个注册的终端，服务总线都会显示其通信终端（步骤 2）。服务总线分配一个 URI 口令给组织，组织通过这个 URI 可以自由创建命名层次。这样，终端就被分配了具体的 URIs。

在提供了终端 URI 的情况下，客户端可通过服务总线注册（步骤 3）发现终端。这个请求使用了 Atom Publishing Protocol，并且返回了一个 AtomPub 服务参考文档到代表应用的终端服务总线上。在上述工作完成后，客户端可以调用通过上述显示终端的服务操作（步骤 4）。对于每个服务总线接受请求，调用 WCF 服务显示的终端通信操作（步骤 5）。

用户服务需要使用 AppFabric 服务总线的开放 TCP 连接显示终端，并保持这个连接一直处于开放的状态，这就解决了两个问题：一是解决了 NAT 问题，服务总线上的开放连接可以路由到应用程序；二是由于连接是在防火墙内初始化的，所以通过连接将消息传回应用时防火墙不会阻止该消息。

服务总线也提高了安全性。由于客户端只可以看见服务总线提供的一个 IP 地址，看不到内部的 IP 地址。服务总线充当了一个外部 DMZ（Demilitarized Zone，隔离区）的角色，起到了间接阻止攻击的作用。

通过服务总线展示其服务的应用一般是使用 WCF 实现，客户端可以由 WCF 或其他技术创建，比如 Java 等。这些客户端创建完成之后，他们可以通过 TCP、HTTP 或者 HTTPS 发送请求。应用同样可以自由地使用自己的安全机制，比如加密、屏蔽通信攻击等等。

服务总线提供了以下一些有用的特征。

（1）支持消息缓冲。消息缓冲是通过一个简单的队列来实现的。客户端可以放置一个多达 256MB 大小的消息到消息缓冲池中去，而不需要客户端直接响应服务。存储消息持久存放在磁盘上，服务可以从磁盘上读取这些被放置的消息。为了防止故障的发生，存放的消息通常需要进行备份，与 Windows Azure 平台上消息备份方式相同。

（2）多个 WCF 服务监听同一个 URI。服务总线通过监听服务随机传播客户端请求，为 WCF 服务提供负载均衡和容错能力。

2. 访问控制

AppFabric 访问控制如图 4-22 所示。

一个依赖于访问控制的应用通常既可以运行在本地平台上，也可以运行在云平台上。首先用户打算通过浏览器访问应用（图 4-22 中步骤 1）。如果应用接受 IdP 令牌（Token），那么将重新定位浏览器到这个 IdP（Identity Providers）。用户使用 IdP 来进行授权，比如通过输入用户名和密码的方式来进行授权，IdP 返回的令牌包含声明信息（步骤 2）。接下来用户浏览器发送 IdP Token 到访问控制中去（步骤 3），访问控制验证接受得到 IdP Token，然后根据事先定义好的应用规则来创建一个新的 Token（步骤 4）。访问控制包含了一个规则引擎，允许每个应用管理员定义不同的 IdPs Token 转换到 AC（Access Control）Token 方式。比如不同的 IdPs 有不同的定义用户名的方式，访问控制规则可以将这些不同格式的用户名转换成相同格式的用户名字符串。然后，访问控制将 AC Token 返回到浏览器（步骤 5），再由浏览器将这个新的 Token 发送给应用（步骤 6）。一旦应用获得了 AC Token，可以验证这个 Token 并使用其中所包含的声明（步骤 7）。

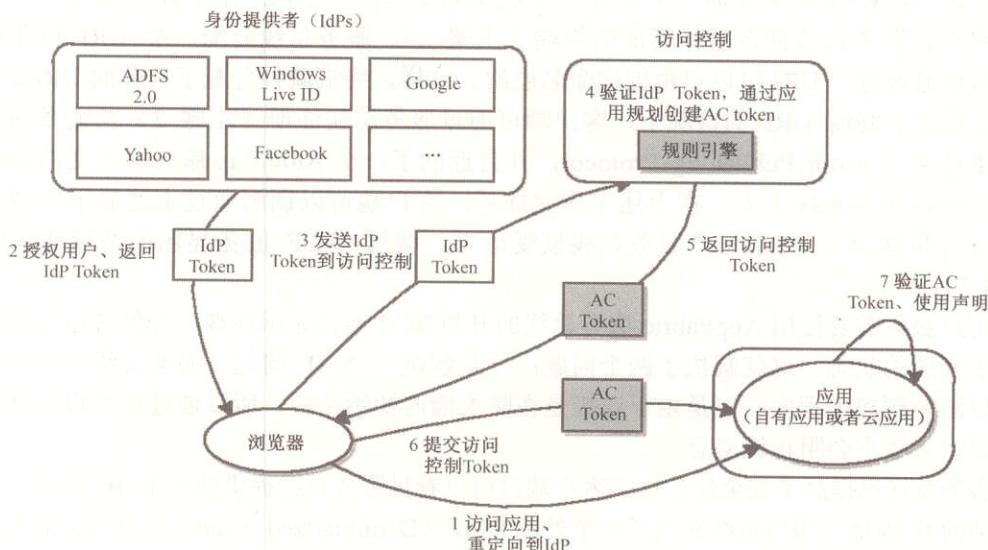


图 4-22 AppFabric 访问控制

应用接受来自多个 IdPs 发出的身份和常见声明的 Token，而不是处理包含不同声明的各种 Tokens，这样就不需要配置应用来使得不同的 IdPs 可信，这些信任关系可以由访问控制维持。

图 4-22 中，访问控制是为一些 IdPs（包括 AD FS 2.0、Windows Live ID、Google、Yahoo、Facebook 等）提供支撑服务的，它同样可以对支持 OpenID 的 IdP 有效。浏览器和其他客户端可以通过 OAuth 2 或 WS-Trust 请求 AC Tokens，AC Tokens 通常有不同的格式，包括 SAML 1.1、SAML 2.0 和 SWT（Simple Web Token）。为了创建应用，Windows 开发者使用 WIF（Windows Identity Foundation）接受 AC Tokens。

在每个分布式应用中，身份都是非常重要的。用户创建的安全应用都是来自于不同提供者的身份，访问控制的目的是为了使创建过程变得简单。通过将访问控制这个服务放到

云中，微软可以保证任何平台上的应用都可以使用它。

3. 高速缓存

AppFabric 高速缓存服务为 Windows Azure 应用提供了一个分布式缓存，同时为访问高速缓存提供了一个库，如图 4-23 所示。高速缓存服务保存每个应用角色实例近期访问数据条款副本的缓存。如果应用需求的数据条款不在本地的高速缓存中，高速缓存库将会自动地连接高速缓存服务提供的共享高速缓存。高速缓存可以通过一些 Windows Azure 实例进行传播，每个实例都保存了不同的缓存数据。然而，使用高速缓存过程中出现的分集对于应用是不可见的。应用只需要请求数据条款，如果高速缓存中没有这个条款，则让高速缓存找到这个请求的条款，最后返回实例中包含所有缓存数据条款。

在 Windows Azure 中，AppFabric 高速缓存并不是缓存最近的访问信息，通常通过 Caching API 在高速缓存中插入一个明确的数据条款。在不修改代码的情况下，为了方便存储正在会话的对象数据，可以通过高速缓存服务配置 Windows Azure 上的 ASP.NET 应用来加速访问。

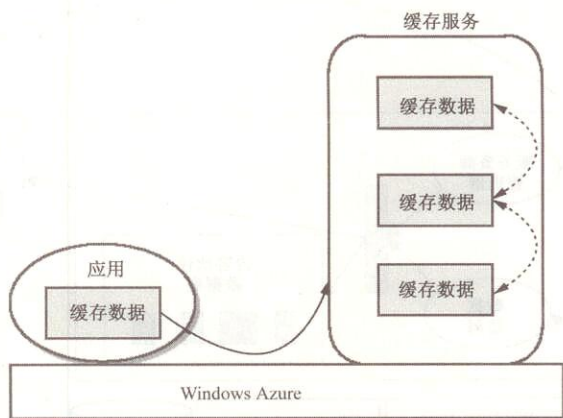


图 4-23 AppFabric 高速缓存

本地环境可使用 Windows Server AppFabric 提供高速缓存服务，与 Windows Azure AppFabric 有许多相似之处。两者之间最大的区别在于：Windows Azure AppFabric 是一种服务，它不需要配置服务器和管理高速缓存，而且是面向多租户的，每个应用都可以获得实例。由于应用对其自身的实例进行了授权访问，所以某个应用高速缓存服务器中的数据对于其他应用而言是无法访问的。

4.5 Windows Azure Marketplace

在本地计算机上，不是所有的应用都是定制的，用户通常也会购买很多应用。许多组织除了购买应用，有时候也会购买数据集。随着云计算越来越受到关注，微软提供了 Windows Azure Marketplace 方便顾客寻找、购买云应用和数据集。

目前 Windows Azure Market 由以下两个部分组成。

(1) DataMarket。内容提供者通过 DataMarket 可以提供交易的数据集。顾客可以浏览

这些数据集，并购买他们感兴趣的数据集。无论是定制的应用还是现有的应用（比如 Microsoft Excel）都可以通过 REST 请求或 OData 门户访问这些数据。

(2) AppMarket。云应用创建者通过 AppMarket 可以将应用展现给潜在的用户。目前 AppMarket 尚未实现，微软只是将其列入了研究计划。

现今社会中，购买应用已经变得十分普遍，而购买数据却没有那么广泛。很多公司均出售各种各样数据，包括人口统计、金融、版权信息等。DataMarket 可以查找内容提供者存储的所有种类的数据，同时检查这些数据是否满足购买者的需求。图 4-24 详细说明了这一过程。

应用和用户都可以通过 DataMarket 访问信息。DataMarket 中存在一个服务资源管理器，是一个 Windows Azure 应用，用户通过这个资源管理器可以查看所有可用的数据集，然后购买需要的数据。应用可以通过 REST 或者 OData 请求访问数据，数据集通常使用 Windows Azure 存储服务或者 SQL Azure 数据库进行存储的。当然，数据集也可以存放在外部内容提供者处。

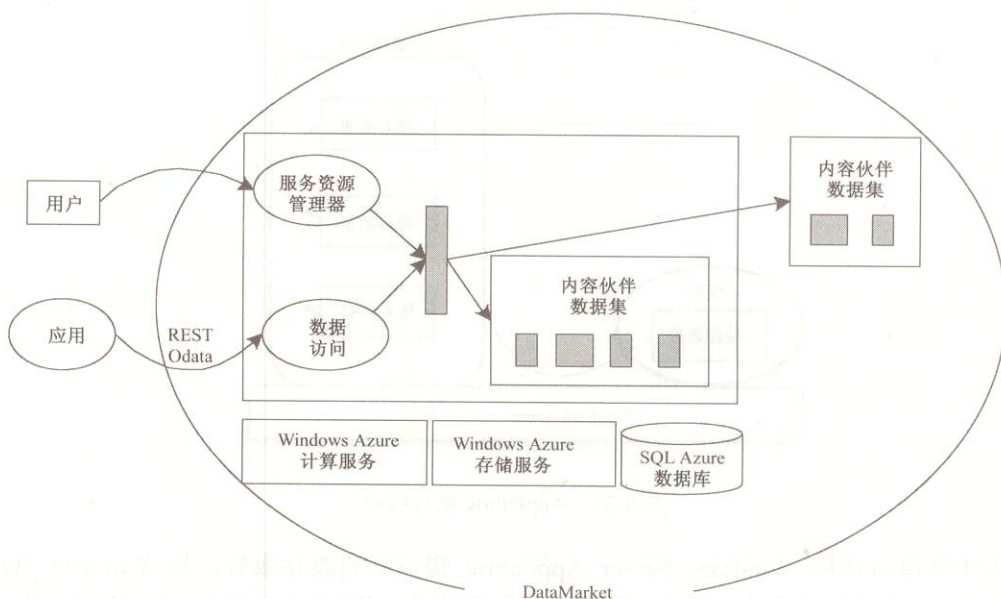


图 4-24 DataMarket 信息访问

4.6 微软云计算编程实践

4.6.1 利用 Visual Studio2010 开发简单的云应用程序

1. 实验环境搭建

本书中所编写的 Windows Azure 实验均是基于 Windows 7 操作系统。首先需要安装 Visual Studio 2010 或 Visual Web Developer 2010 Express，本书中安装的是 Visual Studio 2010。

2. 创建云服务

安装完成后, 启动 Visual Studio 2010, 在“开始页面”上选择“新建项目”, 出现一个“新建项目”的对话框, 如图 4-25 所示。

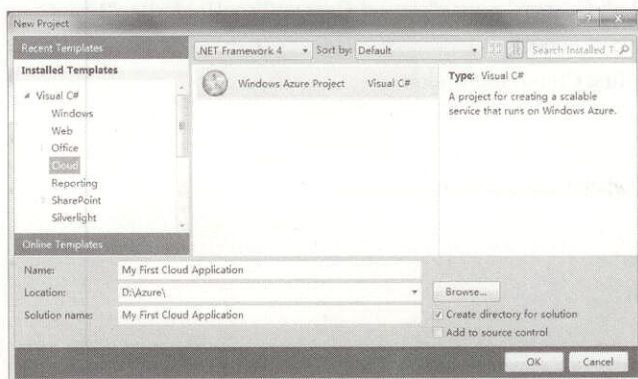


图 4-25 新建项目对话框

在对话框中选择“Cloud”, 然后将项目名称修改为“My First Cloud Application”, 单击“OK”按钮。然后会出现一个界面, 提示需要安装 Windows Azure Tools, 单击“Download Windows Azure Tools”后进入下载界面, 下载 Microsoft Web Platform Installer 后安装, 如图 4-26 所示。



图 4-26 安装 Windows Azure Tools 提示界面

安装完成后重新启动 VS2010, 会出现一个新建 Windows Azure 项目的对话框, 在其中选择 ASP.NET Web Role, 然后重命名为“My Cloud App”, 单击“OK”按钮, 如图 4-27 所示。

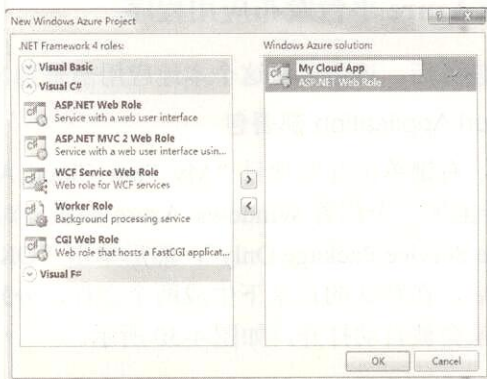


图 4-27 新建 Windows Azure 项目对话框

3. 写本地代码并运行

在 VS 2010 创建的云应用程序中，可以对其进行修改。在 VS 2010 解决方案浏览器中找到 Default.aspx，在其中添加如下代码：

```
<asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">
    <h2 style="color: #FF0000">
        This is my first Cloud Application
    </h2>
    <p>
        I feel well when I use this cloud service.
    </p>
    <p align="center">
        Writer: Aiken
    </p>
    <p align="center">
        E-mail: brucexu1988@163.com
    </p>
</asp:Content>
```

代码编写完成后，从调试菜单中选择“开始调试”，选择默认启动页，然后在浏览器中会出现调试结果，如图 4-28 所示。



图 4-28 本地应用运行结果

4.6.2 向 Windows Azure 平台发布应用程序

完成本地应用的编写任务后，还需要将这个本地应用部署到 Windows Azure 平台。

1. 创建 My First Cloud Application 部署包

在解决方案浏览器中，右键单击开发项目“My First Cloud Application”，从菜单中选择“发布”，单击确定后会弹出一个部署 Windows Azure 云服务对话框，如图 4-29 所示。在对话框中，选择“Create Service Package Only”，然后单击“OK”按钮。

当服务包创建完成之后，在默认的目录下生成两个文件，分别是服务包文件和配置文件，这两个文件所在的目录会被自动打开，如图 4-30 所示。

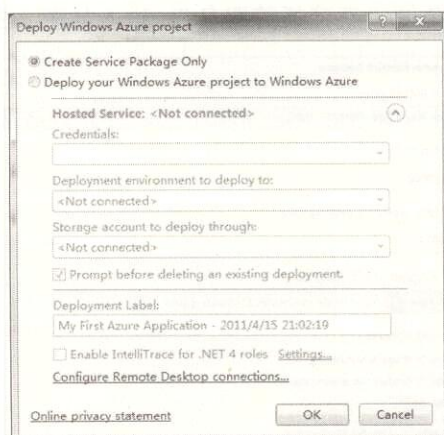


图 4-29 部署 Windows Azure 云服务对话框



图 4-30 部署包目录

2. 向 Windows Azure 部署项目

当用户在 Windows Azure 上创建项目时, 需要访问 <http://windows.azure.com>, 这时出现一个登录的界面, 输入 Windows Live ID, 然后注册 Azure 账号。注册完成后, 重新登陆上述网址, 进入到 Windows Azure 平台的主界面, 如图 4-31 所示。

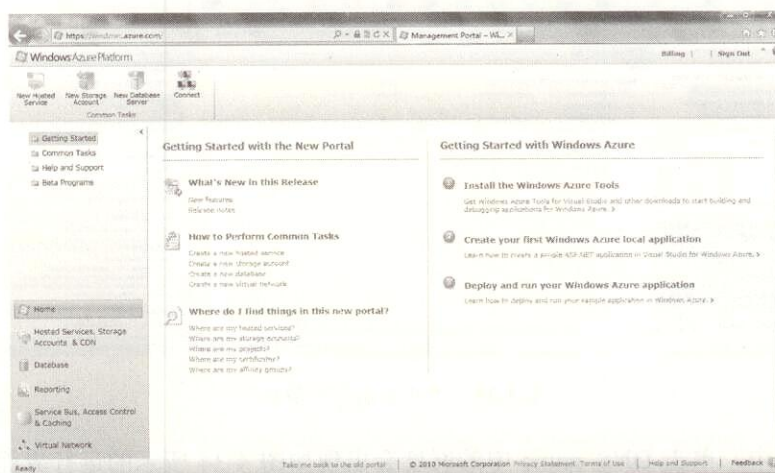


图 4-31 Windows Azure 平台主界面

进入界面后，选择左上角的“新建托管服务”，进入如图 4-32 所示的界面。

图 4-32 服务属性

在服务属性页面中，在服务名称文本框中填写“FirstCloudApp”，在 URL 文本框中填写自主命名的 URL，选择“Deploy to stage environment”，并在部署名文本框中输入“FirstCloud”，定位到本地 VS2010 发布的部署包中的两个文件，单击“OK”。上传完成后，主界面下“托管服务”的界面如图 4-33 所示。

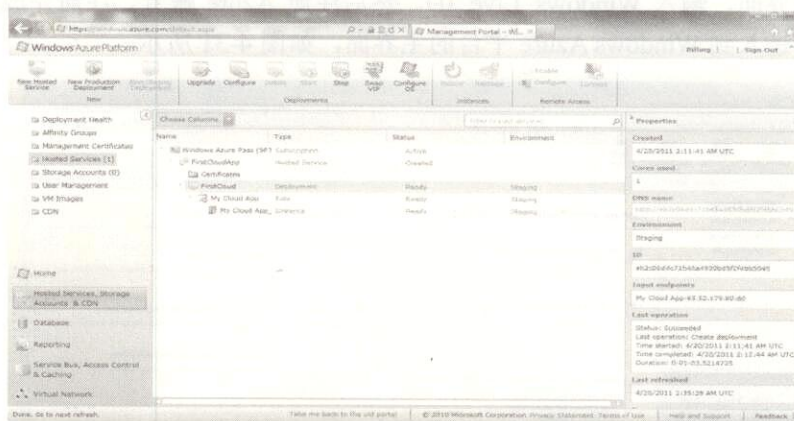


图 4-33 “托管服务”界面

在图 4-33 中，单击 DNS name 中的网址会出现如图 4-34 所示的运行界面，这便是在 Windows Azure 平台上应用的运行结果。

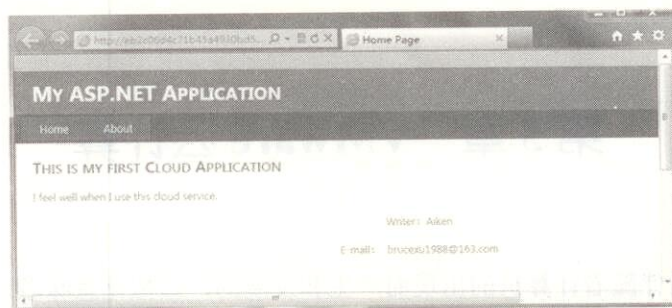


图 4-34 Windows Azure 中应用的运行结果

习题

1. 微软云计算平台包含几个部分？每部分的作用是什么？
2. Windows Azure 存储服务提供了几种类型的存储方式？阐述每种存储方式主要存储对象。
3. 阐述 Web Role 实例和 Worker Role 实例之间的通信机制。
4. SQL Azure 数据同步技术主要有几种？分别如何实现的？
5. 阐述 SQL Azure 和 SQL Server 的相同点和不同点。
6. AppFabric 高速缓存技术是如何实现的？
7. 利用 Visual Studio2010 开发一个简单的应用程序，并将其部署到 Windows Azure 平台上。

参考文献

- [1] Introducing the Windows Azure Platform, Final PDC10
- [2] Introducing Windows Azure, Final PDC10
- [3] MS800_SQL Azure Database Whitepaper_r01
- [4] Similarities and Differences (SQL Azure vs. SQL Server)
- [5] Windows Azure Blob - May 2009
- [6] Windows Azure Queue - Dec 2008
- [7] Windows Azure Table - May 2009.
- [8] Windows-Azure-AppFabric-PDC10-Overview
- [9] <http://blogs.msdn.com/b/azchina/archive/2010/02/18/webrole.aspx>
(参考文献[1]~[8]来自微软官方网站: <http://www.microsoft.com/windowsazure/whitepaperdownload/>)

第5章 VMware 云计算

虚拟化技术是伴随着计算机的出现而产生和发展的，虚拟化意味着对计算机资源的抽象。在云计算概念提出以后，虚拟化技术可以用来对数据中心的各种资源进行虚拟化和管管理，在物理服务器上虚拟出多个操作系统和应用程序，以便更好地利用计算资源。因此虚拟化技术已经成为构建云计算环境的一项关键技术。作为 X86 体系结构虚拟化技术的代表，VMware 公司¹基于已有的虚拟化技术优势，面向云计算推出了一系列解决方案和新的技术，尤其是其推出的面向服务器的虚拟机产品 vSphere，号称云计算的首款操作系统。本章重点讲解 VMware 公司的云战略，以及在构建企业私有云过程中所采用的关键技术。

5.1 VMware 云计算战略

VMware 公司基于已有的虚拟化技术和优势，提供了云基础架构及管理、云应用平台和终端用户计算等多个层次上的解决方案，主要支持企业级组织机构利用服务器虚拟化技术，实现从目前的数据中心向云计算环境转变^[1]。本节根据 VMware 云计算解决方案的三层架构简要介绍 VMware 面向云计算的产品和技术，如图 5-1 所示。

5.1.1 VMware 云战略三层架构

1. 云基础架构及管理层 (IaaS)

云基础架构及管理层由数据中心与云基础架构、安全产品、基础架构和运营管理三大部分组成。数据中心和基础架构是 VMware 云计算解决方案的基石。在这一层 VMware 的主要思路是通过虚拟化技术将数据中心转变为云计算基础架构，然后通过 VMware 虚拟化提供自助部署和调配的功能，企业可以创建私有云，将 IT 基础架构作为服务来交付使用。面向 IaaS 层的主要产品包括 VMware vSphere 系列和 VMware Server 系列，后者为免费版本，性能上不如 vSphere。

2. 云应用平台层 (PaaS)

在 PaaS 层，VMware 通过收购 SpringSource 来构建基于云的应用开发平台，用于满足用户在云计算模式与环境开发相应的应用。SpringSource 框架能通过动态、一致的基础架构满足各种企业和 Web 应用的需要，以及简化新应用程序开发的开发者工具和功能。因此，VMware 的云应用平台以 SpringSource 应用和 VMware vSphere 为基础，采用高级消息队列协议 AMQP，具有无缝扩展的弹性数据管理技术和跨物理/虚拟环境可见性的性能

¹ 2004 年，VMware 被 EMC 公司收购，成为 EMC 公司的一个子公司。

监控和应用管理机制，并能实现私有云和公有云之间的迁移。

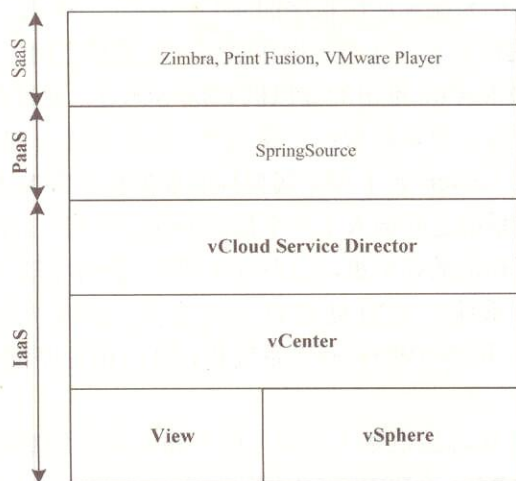


图 5-1 VMware 云计算三层架构

3. 终端用户计算解决方案：桌面虚拟化产品（SaaS）

在 SaaS 层，VMware 的定位还不是特别明确，主要是基于桌面和应用程序虚拟化，提供了 VMware ThinApp、VMware Workstation、VMware Fusion、Zimbra、VMware Player、VMware 移动虚拟平台（MVP）及 VMware ACE 等产品。

VMware 的云计算解决方案的重点在于对数据中心等基础架构的虚拟化，因此，在 IaaS 层上 VMware 的工作较多，所以本节重点介绍 VMware vSphere 结构、vCloud Service Director 和 VMware View。

5.1.2 VMware vSphere 架构

VMware 在原来的 VMware Infrastructure 3（以下简称 VI3）基础上推出的 VMware vSphere 被称为业界首款云计算操作系统。VMware vSphere 主要包括两部分：一是虚拟化管理器 VMM 部分，VMware ESX 4；二是用于整合和管理 VMM 的 VMware vCenter^[3]。其架构如图 5-2 所示。

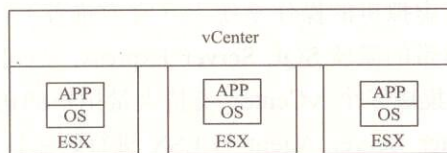


图 5-2 vSphere 架构

1. ESX Server

虚拟化从结构上可以分为寄居架构和裸金属架构（Bare Metal）。寄居架构指的是在操作系统的层面之上进行虚拟机实现，VMware 开发的 VMware Workstation 系列就属于寄居架构。裸金属架构是在计算机硬件上直接进行虚拟化，是架设在计算机硬件和操作系统之

间的虚拟化。通过裸金属架构的虚拟化, 计算机硬件直接被切割成若干个虚拟机, 然后在这些虚拟机上再进行各自的系统和应用程序的安装, 这样一来, 虚拟机的底层是虚拟出来的 CPU、内存等计算机硬件资源, 而不是操作系统, 虚拟机之间完全独立。

vSphere 的底层就是 VMware 推出的虚拟机 ESX Server。通过 ESX 虚拟化数据中心服务器, 将数据中心转换为云计算基础架构, 满足 IT 组织利用内部和外部资源、低成本地提供云服务的能力。ESX Server 属于裸金属架构的虚拟机。ESX Server 直接安装在服务器硬件上, 在硬件和操作系统之间插入了一个稳固的虚拟化层。ESX Server 将一个物理服务器划分为多个安全、可移植的虚拟机, 这些虚拟机在同一物理服务器上运行。每个虚拟机都呈现为一个完整的系统 (具有处理器、内存、网络、存储器和 BIOS), 因此 Windows、Linux、Solaris 和 NetWare 操作系统和软件应用程序都可以在虚拟机中运行, 无需进行任何修改。

ESX Server 是向 IT 环境提供基于虚拟化的分布式服务的基础。ESX 最新的版本是 VMware ESX 4, 和之前 VMware ESX 3.5/3 相比, 在功能和特性上有很多更新和扩展, 其中最大的区别在于 VMware ESX 4 只支持 64bit 运行模式, 只能安装在支持 64 位计算的 X86 物理服务器上。除了 ESX, VMware 还推出了精简版的 ESXi, ESXi 与 ESX 的最大区别在于 ESXi 去除了 Service Console。

VMware ESX 4 主要功能体现在如下 3 个方面。

(1) 基础架构服务: 即虚拟化管理器 (VMM) 功能, 是整个产品的基础。通过在物理机之上的虚拟层可以抽象处理器、内存和 I/O 等资源来运行多个虚拟机。虚拟机能支持高达 8 个 vCPU 和 256GB 内存; 还支持热添加功能, 可以热添加虚拟 CPU、内存和网络设备, 满足应用程序无缝扩展的功能。

(2) 增强型的基础架构服务: 在基础架构服务以外, ESX 4 还提供了能增强网络和存储 I/O 性能的 VMDirectPath、能减少存储空间使用的 VSorage Thin Provisioning 和 Linked Clone 等增强的功能。

(3) 应用程序服务: ESX 4 提供了 vCenter Agent, 用于向 vCenter 上传本机的管理和性能信息, 根据 vCenter 的指示协助 vMotion。

2. 云管理平台 vCenter

vCenter 作为管理节点控制和整合属于其域的 vSphere 主机, 既可以安装在物理机的操作系统上, 也可以安装在虚拟机的操作系统上 (官方推荐)。从实现方式上看, 它是基于 Java 技术的, 后台连接自带的微软 SQL Server Express, 也可以使用 Oracle 的数据库, 并可以使用其 “链接模式” 集成多个 vCenter 支持大量用户的访问。在通信方面, 它通过 vSphere 主机内部自带 vCenter Server Agent 与 ESX 进行联系, 并提供 API 供外部程序和 vCenter 客户端调用。在扩展性方面, 它支持很多第三方的插件。

vCenter 包括以下 6 项基本功能。

(1) 资源和虚拟机的清单管理。该功能可以列出和管理 vCenter 管理域内所有的资源 (如存储、网络、CPU 和内存等) 和虚拟机。

(2) 任务调度。支持定时任务或者及时任务 (如 vMotion), 满足各个任务之间不出现抢占资源或者冲突的要求。

- (3) 日志管理。用于记录任务和事件的日志。
- (4) 警告和事件管理。使用户可以及时获知系统出现的新情况。
- (5) 虚拟机部署。通过部署向导,上传 vApp 和虚拟磁盘等,部署虚拟机。
- (6) 主机和虚拟机的设置。用户可以修改一些主机和虚拟机的主要配置,而且还能对那些非常底层的特性进行设置,比如是否开启硬件辅助虚拟化。

vCenter 还有以下 7 个方面的高级功能。

(1) 动态迁移。vSphere 提供了 vMotion 和 Storage vMotion 技术,分别满足虚拟机和虚拟磁盘的热迁移。

(2) 资源优化。VMware 的分布式资源调度 (Distributed Resource Scheduler, DRS) 技术,通过将虚拟机从资源紧张的主机迁移到资源剩余的主机等方式来实现资源优化,使得每个虚拟机都能找到合适的位置。

(3) 安全方面。VMware 推出了两大虚拟机安全技术,一是推出 VMsafe API,对虚拟机进行安全扫描检测病毒和恶意软件;二是 VMware Shield Zones,主要起到防火墙的作用,可监视、记录和阻止 vSphere 主机内部或集群中主机之间和虚拟机之间流量。

(4) 容错。VMware Fault Tolerance 是 VMware 提供的虚拟机容灾技术。

(5) 高可用性。VMware HighAvailability 技术通过心跳机制来检测虚拟机的运行状态,并通过在其他主机上重启无响应的虚拟机的方式来保障系统的可用性。

(6) 备份。VMware 采用了加固备份技术 (VMware Consolidated Backup, VMCB),在没有安装 Agent 时对多个虚拟机进行集中备份。

(7) 应用部署。VMware vApp 基于开放式虚拟化格式 (Open Virtualization Format, OVF) 协议,将应用程序转化为自描述和自管理型实体,以方便部署和降低管理开支。

5.1.3 底层架构服务 vCloud Service Director

vSphere 的主要目的是将底层物理资源进行虚拟和管理,但仅安装了 vSphere 的数据中心并不能称之为云平台。VMware 通过 vCloud Service Director,在 vSphere 架构上利用一系列虚拟技术,提供连接企业虚拟环境与私有云的接口和自动化工具,通过运行 vCloud Express 与外部服务商无缝地连接,向外提供云 IaaS 服务。VMware vCloud Director 使 IT 部门能够通过基于 Web 的门户向用户开放虚拟数据中心,并定义和开放能部署在虚拟数据中心的 IT 服务目录^[4]。

vCloud Service Director 早期被称为 Redwood 项目,目前该产品的资料 VMware 公司向外部公布的不多。vCloud Service Director 的架构如图 5-3 所示,它具有数据库与管理资源池的服务总线通信的功能。另外,利用基于 VMware vCloud Director 提供云服务的 VMware 服务提供商体系,可以将数据中心容量扩展到安全、兼容的公共云中,并像管理企业的私有云一样方便地管理它。利用基于策略的用户控制技术和 VMware vShield 安全技术,可以保持多租户环境的安全性和可控性。以虚拟数据中心的形式向内部组织高效地提供资源,提高整合率并简化管理,降低成本。以渐进方式实现云计算,利用现有投资和开放标准,以保证云之间的互操作性和应用程序的可移植性。

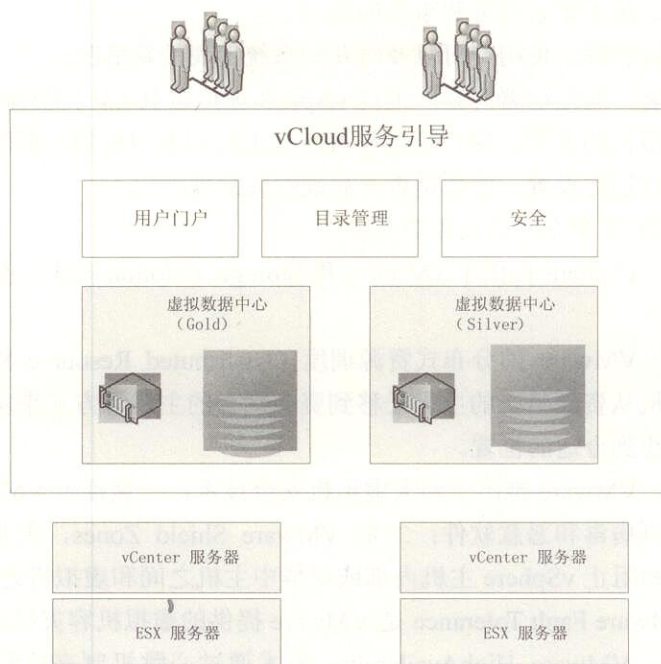


图 5-3 vCloud Service Director 的架构

5.1.4 虚拟桌面产品 VMware View

VMware View 是 VMware 桌面虚拟化产品，通过 VMware View 能够在一台普通的物理服务器上虚拟出很多台虚拟桌面（Virtual Desktop）供远端的用户使用。

VMware View 的主要部件^[11]包括：

(1) View Connection Server: View 连接服务器，View 客户端通过它连接 View 代理，将接收到的远程桌面用户请求重定向到相应的虚拟桌面、物理桌面或终端服务器。

(2) View Manager Security Server: View 安全连接服务器，是可选组件。

(3) View Administrator Interface: View 管理接口程序，用于配置 View Connection Server、部署和管理虚拟桌面、控制用户身份验证。

(4) View 代理: View 代理程序，安装在虚拟桌面依托的虚拟机、物理机或终端服务器上，安装后提供服务，可由 View Manager Server 管理。该代理具备多种功能，如打印、远程 USB 运行和单点登录。因为在 VMware vSphere Server 提供的虚拟机不包括声卡、USB 接口支持等，必须安装该软件，才可以将 VMware vSphere Server 提供的虚拟机连接到 View Client 计算机的相应设备上并显示、应用在客户端。

(5) View Client: View 客户端程序，安装在需要使用“虚拟桌面”的计算机上，通过它可以与 View Connection Server 通信，从而允许用户连接到虚拟桌面。

(6) View Client with Offline Desktop: 也是 View 客户端程序，但该软件支持 View 脱机桌面，可以让用户“下载”vSphere Server 中的虚拟机到“本地”运行。

(7) View Composer: 安装在 vCenter Server 上的软件服务，可以通过 View Manager 使用“克隆链接”的虚拟机，这是 View 4 提供的新功能，在以前的 View 3 版本中，每个虚拟

桌面只能使用一个独立的虚拟机，而添加该组件后，可以让虚拟桌面使用“克隆链接”的虚拟机，这不仅提高了部署虚拟桌面的速度，也减少了 vSphere Server 的空间占用。

5.2 vSphere 中的云管理平台 vCenter

vCenter 是 ESX 的分布管理工具，图 5-4 显示了 vCenter 的各个组件之间的关系，本节介绍其中几个关键部分。

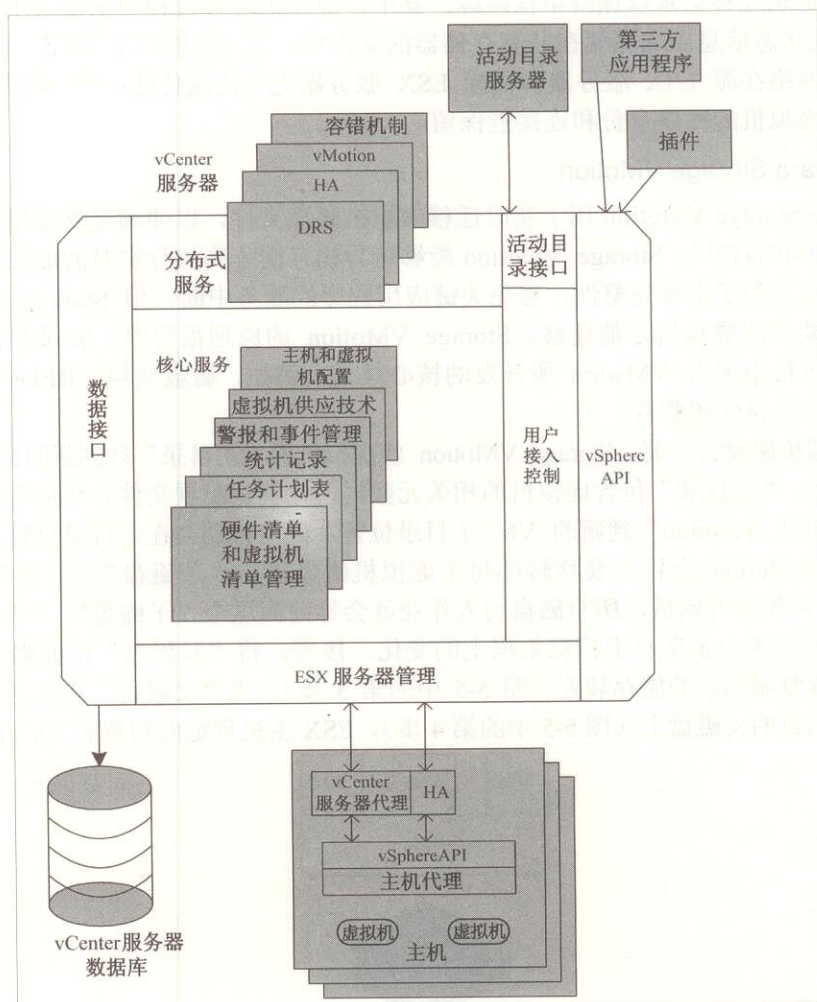


图 5-4 vCenter 组件结构

5.2.1 虚拟机迁移工具

1. VMotion

虚拟机的迁移是指把源主机上的操作系统和应用程序移动到目的主机，并且能够在目的主机上正常运行^[6]。VMotion 是 VMware 用于在数据中心的服务器之间进行虚拟机迁移的技术。通过将服务器、存储和网络设备完全虚拟化，利用 VMotion 能够将正在运行的整个

虚拟机实时从一台服务器移到另一台服务器上。虚拟机的全部状态由存储在共享存储器上的一组文件进行封装,而 VMware 的 VMFS 群集文件系统允许源和目标 ESX 同时访问这些虚拟机文件。然后,虚拟机的活动内存和精确的执行状态可通过高速网络迅速传输。由于网络也被 ESX 虚拟化,因此,虚拟机保留其网络标识和连接,从而确保实现无缝迁移。

VMotion 可以在不停机、不中断业务的情况下自动维护硬件,并行地将多个任意操作系统的虚拟机从运行不正常的服务器中迁出,实时提供迁移向导,以确定虚拟机迁移的最佳目的地,无需管理员在场即可跨 ESX 所支持的所有类型的硬件和存储器进行虚拟机迁移,并详细记录迁移记录以保持审核跟踪。其中,虚拟机迁移过程中主要采用三项技术:

- ① 将虚拟机状态信息压缩存储在共享存储器的文件中;
- ② 将虚拟机的动态内存和执行状态通过高速网络在源 ESX 服务器和目标 ESX 服务器之间快速传输;
- ③ 虚拟化网络以确在迁移后虚拟机的网络身份和连接能保留。

2. VMware Storage VMotion

VMware Storage VMotion 用于实时迁移虚拟机磁盘文件,以便满足对虚拟机磁盘文件的升级、维护和备份^[7]。Storage VMotion 能够跨异构存储阵列执行实时的虚拟机磁盘文件迁移,同时考虑到了事务完整性,避免关键应用程序的服务中断,但 Storage VMotion 迁移虚拟机时要关闭虚拟机才能迁移。Storage VMotion 的原理很简单,就是存储之间的转移。在操作过程中采用 VMware 所开发的核心技术,例如,磁盘快照、REDO 记录、父/子磁盘关系,以及快照整合。

移动虚拟机磁盘文件前,Storage VMotion 将虚拟机的“主目录”移到新的位置(图 5-5 中的第 1 步)。“主目录”包含虚拟机的相关元数据,也就是配置文件、交换文件、日志文件。它会“自动 Vmotion”到新的 VM 主目录位置。磁盘移动会在主目录移转后进行。首先,Storage VMotion 会针对要移转的每个虚拟机磁盘建立“子磁盘”(图 5-5 中的第 2 步)。一旦移转作业开始后,所有磁盘写入作业就会导向到这个“子磁盘”。“子磁盘”相当于缓冲磁盘,用来记录所有虚拟机数据上的变化。接着,将“父磁盘”或原始虚拟磁盘从旧的储存装置复制到新的储存装置(图 5-5 中的第 3 步)。当“父磁盘”传输完毕,最后将子磁盘整合到目的父磁盘上(图 5-5 中的第 4 步),ESX 主机重定向到新的父磁盘位置。

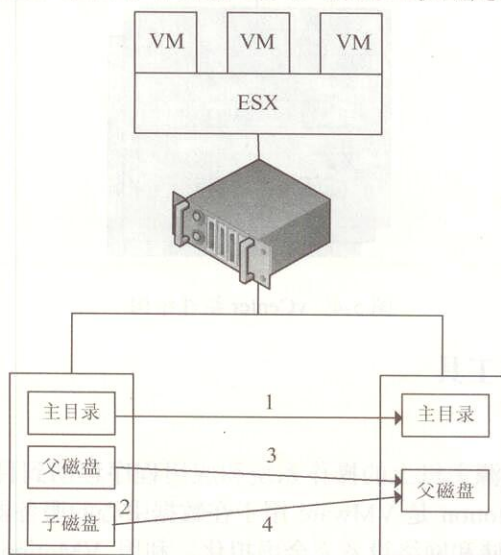


图 5-5 Storage VMotion

5.2.2 虚拟机数据备份恢复工具

1. VMware Consolidated Backup (VCB)^[14]

VCB 是一个备份代理，本身没有备份功能，需要第三方备份软件来配合。VCB 对待备份的虚拟机创建快照后，基于此快照配合第三方软件进行文件级（仅 Windows）或镜像级备份，VCB 将虚拟机的数据集中映射到 VCB 备份服务器上供第三方备份软件使用，备份完成后删除快照。使用 VCB 时，对于 Windows 虚拟机可看到虚拟机里的文件都出现在 VCB 备份服务器上，对于其他操作系统虚拟机，其镜像出现在 VCB 备份服务器上，可以使用熟悉的传统方式备份这些文件。VCB 卸载了虚拟机中的备份代理，降低了备份对虚拟机的影响，使用传统备份软件平滑的过渡到对虚拟机的备份。

2. VMware Data Recovery^[15]

Data Recovery 是 vSphere 新提供的数据库备份功能，与 VCB 共存，不替代 VCB，是一种基于磁盘的数据备份方式，不支持以磁带为目标的备份。VDR 由 VCB 插件、运行在 ESX 主机上的虚拟机及备份存储这三个部件组成。通过插件以向导的方式进行配置和调度备份任务。

1) 备份过程

首先选择要保护的虚拟机、备份作业计划、数据保留策略以及目标磁盘，之后，作业被分派给 VMware Data Recovery 虚拟工具以启动受保护虚拟机的快照，然后开始备份。VMware Data Recovery 确保在创建快照的时间点捕捉到虚拟机的完整状态，将快照直接装载到 VMware Data Recovery 虚拟工具中，避免备份流量占用 LAN 资源。装载快照之后，VMware Data Recovery 即开始将多个数据块以串流方式传输到目标存储设备。在此过程中，VMware Data Recovery 将消除数据块的传输流中的重复数据，确保备份数据写入至目标磁盘之前消除冗余的数据。写入全部数据后，VMware Data Recovery 将卸载快照，并使虚拟磁盘退出快照模式。

2) 恢复过程

在整个虚拟机的恢复过程中，VMware Data Recovery 将显示每个虚拟机的时间还原点。一旦选定了所需的还原点，VMware Data Recovery 通过检索特定的数据块，将虚拟机还原到目标主机或资源池。当覆盖现有的虚拟机或虚拟磁盘时，VMware Data Recovery 只高效传输已发生更改的数据块。VMware Data Recovery 也支持文件级的还原，恢复过程与上述过程类似，只不过此恢复过程是从虚拟机控制台内启动的。

5.2.3 虚拟机安全工具

VMware vShield^{[8][9]}是 VMware 开发的用于云环境安全的产品。对于各种规模的组织机构而言，VMware vShield 提供了敏捷、动态、经济高效的安全性，确保平滑地实现云部署，并获得云计算的实际好处。VMware vShield 为第三方解决方案提供了一个可编程的框架，以便将其整合和扩展至监控和管理服务中。

VMware vSphere 与 VMware vShield 产品系列相结合，能够确保云中应用和数据的安全性，能够应对云计算中与应用和数据安全性相关的诸多挑战。这些解决方案将确保应用和数据能被恰当地划分至信任区域，以满足法规遵从的需求，也可以满足将数据保持在特

定权限范围内的要求。

VMware vShield Product Family 包括 VMware vShield App、VMware vShield Edge、VMware vShield Endpoint。

5.2.4 可靠性组件 FT 和 HA

1. VMware Fault Tolerance

VMware 的 Fault Tolerance^[5]（简称 FT）是一种针对关键应用的双机热备份机制^[2]。FT 的基本原理是利用虚拟化原理，让两个完全一样的 VM 在内存之间相互备份，做到 CPU 命令级和内存比特级的完全克隆。FT 使用 vLockstep 技术，该技术的核心是取得一个 VM 上的命令，到另一个 VM 上运行。FT 使两台物理机上的 ESX 各有一个 VM 的副本。这两个副本分别称作主副本和次副本，主次副本完全一样，不管主副本做什么操作，都会立即在 CPU 和内存级通知次副本。

VMware Fault Tolerance 机制延长了数据中心的正常运行时间，消除了传统硬件或软件集群解决方案的成本和复杂性，但目前只支持单 CPU 的虚拟机。

2. VMware High Availability

HA^[16] 支持集群内主机发生故障时的虚拟机自动迁移到其他主机恢复运行。传统的集群解决方案致力于在发生主机故障或虚拟机故障时，在最短的应用程序停机时间内实现即时恢复，需要给每台计算机（或虚拟机）配备一个镜像虚拟机（可能在另一台主机上），使用群集软件将计算机（或虚拟机及其主机）设置为互相镜像，由主虚拟机向镜像发送心跳信号，一旦发生故障，镜像将立即接管。

利用 VMware HA，可以将一组 ESX Server 主机合并为一个具有共享资源池的集群。VMware HA 监控集群中的所有主机，在每台主机上的代理程序不断向集群中的其他主机发出“心跳信号”，“心跳信号”终止时（主机发生故障），VMware HA 立即响应，并在另一台主机上重启故障虚拟机。

图 5-6 中填充为灰色的“ESX 服务器 1”为发生故障的服务器，当 HA 检测到该机器故障以后，通过共享状态，在另两台机器上分别启动“ESX 服务器 1”原来的虚拟机。

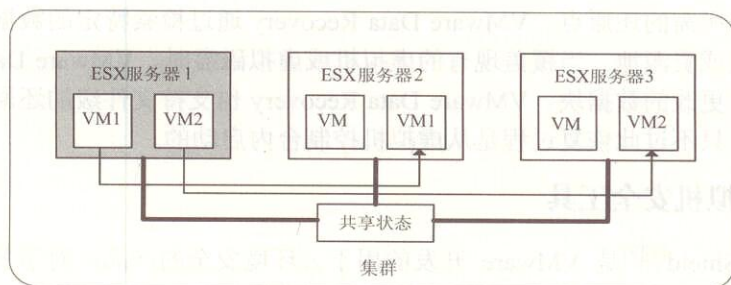


图 5-6 VMware HA

5.3 云架构服务提供平台 vCloud Service Director

vCloud Service Director 主要是通过整合多个基于 vCenter Server 的资源池来实现一个基本完备的 IaaS 云。vCloud Service Director 的主要功能包括以下几个方面^[4]。

5.3.1 创建虚拟数据中心和组织

不管是私有云还是公有云，都会面对各种类型的客户和场景，所以 vCloud Director 在设计上支持资源隔离和多租户机制。vCloud Director 引入了两个非常核心的概念：一是用于对资源进行隔离的虚拟数据中心（VDC）；二是用于支持多租户机制的组织。

VDC 是包含用于云计算的计算和存储等资源的集合。在使用上，管理员首先在 Director 上添加一些 vCenter Server，然后将 vCenter Server 管理的计算资源公布出来，把这些资源组合成一个巨大的资源池，之后管理员可以创建一个 VDC，并按照自己的思路或某些规则将资源池中部分或全部计算和存储资源添加到这个新建的 VDC 中。

组织是管理员通过规则将多个用户组合在一起。比如，属于人事部门的人员都归类到人事部门这个组织。每个组织都有独占的虚拟资源和目录、独立的 LDAP 认证系统和特定的规则管理。通过组织的特性能够让多个单位分享同一套基础设施，而且 Director 会为每个组织生成不同的 URL 来让管理登录。在每个组织内部，管理员可以创建其下属的用户和小组，还可以为每个组织设定相应的租约、额度和限制等参数。组织中的用户可以通过三种方式进行认证：一是使用 Director 本地数据库；二是使用与 Director 相匹配的 Active Directory 或 LDAP 服务器；三是使用这个组织特定的 Active Directory 或 LDAP 服务器。

VDC 和组织之间的关系如图 5-7 所示。首先，VDC 按照规模大小分为两个类别，供应商级和组织级。在使用的时候，管理员先创建多个供应商级 VDC，比如图 5-7 中的 Gold VDC 和 Silver VDC 等。之后，管理员在供应商 VDC 的基础上为组织创建新的组织级 VDC，如图 5-7 中的 Org 1 Gold VDC。注意，一个组织级 VDC 能够和创建其供应商级 VDC 一样大，并且一个组织可以拥有多个组织级 VDC。

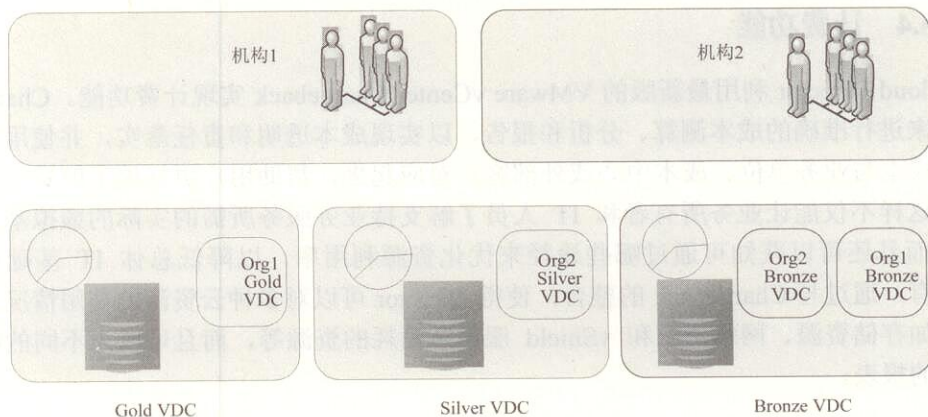


图 5-7 VDC 和组织的关系

供应商级 VDC 可以通过三种方式在其上创建组织级 VDC：一是按需使用，只有当用户在组织级 VDC 上部署一个虚拟机，才会消耗相关供应商级 VDC 的资源；二是预留池机制，在组织级 VDC 创建的时候，供应商级 VDC 分配一定的资源，通过由组织来控制诸如共享值和保留值等高级资源管理配置；三是分配池机制，这个机制和前面预留池机制相同的是，供应商级 VDC 会为组织级 VDC 分配一定的资源，但是类似共享值和保留值等高级资源管理配置则由负责供应商级 VDC 的管理员设置。

5.3.2 网络的设计

在网络方面, vCloud Director 主要有两大类机制: 一是外部网络机制; 二是网络池机制。外部网络机制主要给部署的虚拟机提供链接该虚拟机所属组织之外网络 (包括属于其他组织的网络或互联网) 的能力, 在实现上面, 一个外部网络就是一个用于传输对外虚拟机流量的端口组, 这个端口组通过使用一个 VLAN 标签实现网络隔离。在使用方面, 管理员会首先创建一个外部网络, 填写的参数有网络的子网掩码、默认的网关、首选和备选的 DNS 地址、DNS 前缀和静态 IP 地址池, 之后将这个外部网络和相关的虚拟机联系起来即可。

网络池是一系列隔离的第 2 层物理网段, 是用来创建组织和虚拟机网络的基石, 主要用于组织内部虚拟机之间的通信, 并且它也确保网络能够在云中自动地被使用和部署。在使用方面, 每当用户部署一个虚拟机, 都会消耗其对应网络池的一个 IP 地址。在实现方面, 网络池主要由三种技术支持: 一是基于 VLAN 的; 二是依赖 Director 自己的网络隔离技术 VCDNI (VMware vCloud Director Network Isolation technology); 三是使用 Portgroup 的。关于网络的问题, 在 5.4 节做进一步的说明。

5.3.3 目录管理

在 vCloud Director 中, 目录主要用于存储各种资源的容器。一个目录隶属于一个组织, 并主要由这个组织的管理员负责创建, 并且可根据需要来设置这个目录的共享设置。主要存储的东西包括两大类: 一是 vApp, 它是基于 OVF 格式的虚拟器件, 通过部署 vApp 快速搭建一个包含多个虚拟机的应用; 二是一些诸如 ISO 格式和 floppy 格式的镜像和介质, 可用于在虚拟机上安装操作系统或者传递数据给虚拟机。

5.3.4 计费功能

vCloud Director 利用最新版的 VMware vCenter Chargeback 实现计费功能。Chargeback 主要用来进行准确的成本测算、分析和报告, 以实现成本透明和责任落实, 并使用户能够将 IT 成本与业务单位、成本中心或外部客户对应起来, 帮助用户更好地了解资源成本是多少, 这样不仅能让业务所有者和 IT 人员了解支持业务服务所需的实际的虚拟基础架构成本, 而且还可以获知可通过哪些途径来优化资源利用率, 以降低总体 IT 基础架构开支。还有, 通过与 Chargeback 的整合, 使得 Director 可以对多种云资源的使用情况进行计费, 比如存储资源、网络资源和 vShield 服务所消耗的资源等, 而且可以为不同的组织生成不同的报表。

5.4 VMware 的网络和存储虚拟化

除了对数据中心的服务器通过 ESX 进行虚拟化之外, vSphere 产品系列还包括网络和存储资源的虚拟化^[3]。

5.4.1 网络虚拟化

VMware 的网络虚拟化技术主要是通过 VMware vSphere 中的 vNetwork 网络元素实现, 其虚拟网络架构如图 5-8 所示。通过这些元素, 部署在数据中心物理主机上的虚拟机

可以像物理环境一样进行网络互联。vNetwork 的组件主要包括虚拟网络接口卡 vNIC、vNetwork 标准交换机 vSwitch 和 vNetwork 分布式交换机 dvSwitch。

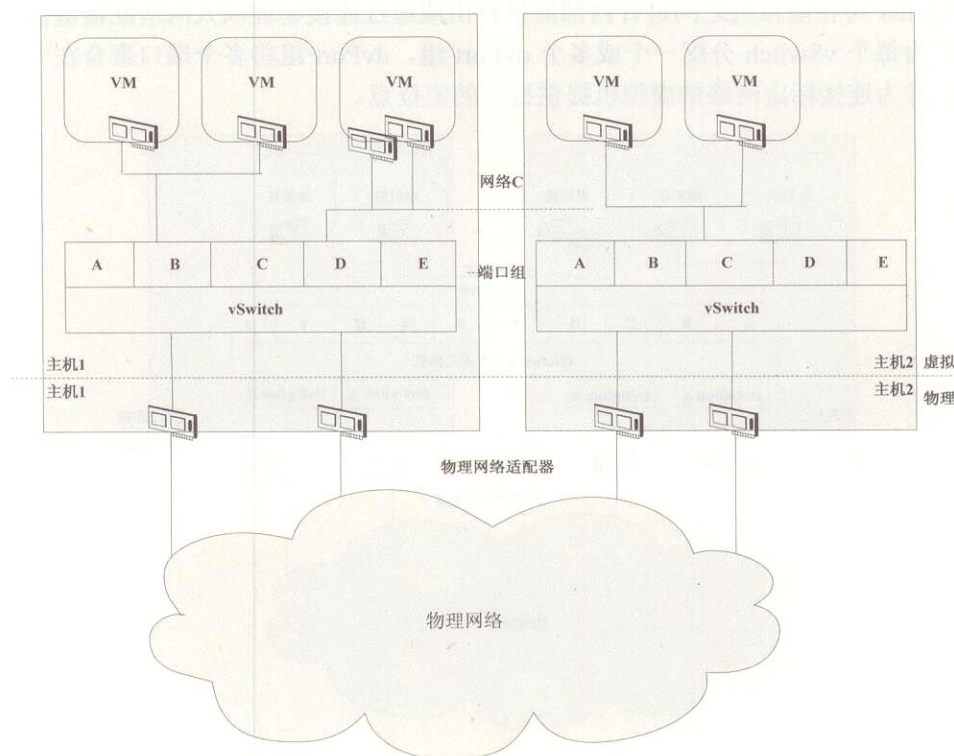


图 5-8 VMware 虚拟网络架构

1) 虚拟网络接口卡

每个虚拟机都可以配置一个或者多个虚拟网络接口卡 vNIC。安装在虚拟机上的客户操作系统和应用程序利用通用的设备驱动程序与 vNIC 进行通信。从虚拟机的角度来看，客户操作系统中的通信过程就像与真实的物理设备通信一样。而在虚拟机的外部，vNIC 拥有独立的 MAC 地址以及一个或多个 IP 地址，且遵守标准的以太网协议。

2) 虚拟交换机 vSwitch

虚拟交换机用来满足不同的虚拟机和管理界面进行互联。虚拟交换机的工作原理与以太网中的第 2 层物理交换机一样。每台服务器都有自己的虚拟交换机。虚拟交换机的一端是与虚拟机相连的端口组，另一端是与虚拟机所在服务器上的物理以太网适配器相连的上行链路。虚拟机通过与虚拟交换机上行链路相连的物理以太网适配器与外部环境连接。虚拟交换机可将其上行链路连接到多个物理以太网适配器以启用网卡绑定。通过网卡绑定，两个或多个物理适配器可用于分摊流量负载，或在出现物理适配器硬件故障或网络故障时提供被动故障切换。

3) 分布式交换机

vNetwork 分布式交换机（dvSwitch）是 vSphere 的新功能，如图 5-9 所示。dvSwitch 将原来分布在一台 ESX 主机上的交换机进行集成，成为一个单一的管理界面，在所有关

联主机之间作为单个虚拟交换机使用。这使得虚拟机可在跨多个主机进行迁移时确保其网络配置保持一致。与 vSwitch 一样，每个 dvSwitch 都是一种可供虚拟机使用的网络集线器。dvSwitch 可在虚拟机之间进行内部流量路由或通过连接物理以太网适配器链接外部网络，可以为每个 vSwitch 分配一个或多个 dvPort 组。dvPort 组将多个端口聚合在一个通用配置下，并为连接标定网络的虚拟机提供稳定的定位点。

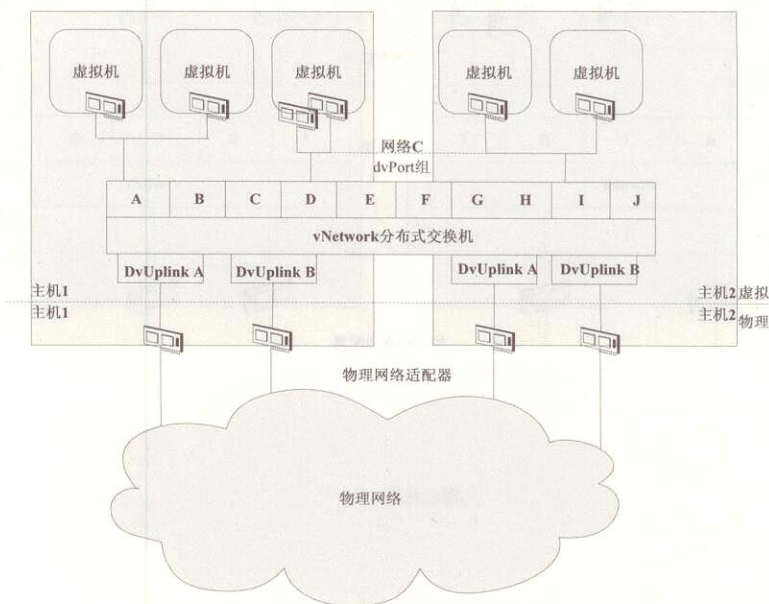


图 5-9 分布式交换机

4) 端口组

端口组是虚拟环境特有的概念。端口组是一种策略设置机制，这些策略用于管理与端口组相连的网络。一个 vSwitch 可以有多个端口组。虚拟机不是将其 vNIC 连接到 vSwitch 上的特定端口，而是连接到端口组。与同一端口组相连的所有虚拟机均属于虚拟环境内的同一网络，即使它们属于不同的物理服务器也是如此。可将端口组配置为执行策略，以提供增强的网络安全、网络分段、更佳的性能、高可用性及流量管理。

5) VLAN

VLAN 支持将虚拟网络与物理网络 VLAN 集成。专用 VLAN 可以在专用网络中使用 VLAN ID，而不必担心 VLAN ID 在较大型的网络中会出现重复。流量调整定义平均带宽、峰值带宽和流量突发大小的 QOS 策略，设置策略以改进流量管理。网卡绑定为个别端口组或网络设置网卡绑定策略，以分摊流量负载或在出现硬件故障时提供故障切换。

5.4.2 存储虚拟化

数据存储对虚拟化技术是非常重要的，vSphere 支持多种不同的本地存储和网络存储，包括 SCSI、SAS 和 SATA 磁盘及基于网络的 iSCSI、NFS 协议的存储设备和光纤通道（FC）数据存储。前面讲到的 VMotion、Storage VMotion 及 FT、HA 都用到了 VMware 的虚拟化共享存储的技术^[10]。

vSphere 提出的高性能集群文件系统，叫做虚拟机文件系统（Virtual Machine File System, VMFS），允许来自多个不同主机服务器的并发访问，即允许多个物理主机同时读写同一存储器。VMFS 的功能主要包括以下 3 点。

（1）磁盘锁定技术。磁盘锁定技术是指锁定已启动的虚拟机的磁盘，以避免多台服务器同时启动同一虚拟机。如果物理主机出现故障，系统则释放该物理主机上每个虚拟机的磁盘锁定，以便这些虚拟机能够在其他物理主机上重新启动。

（2）故障一致性和恢复机制。故障一致性和恢复机制可以用于快速识别故障的根本原因，帮助虚拟机、物理主机和存储子系统从故障中恢复。该机制中包括了分布式日志、故障一致的虚拟机 I/O 路径和计算机状况快照等。

（3）裸机映射（RDM）。RDM 使得虚拟机能够直接访问物理存储子系统（iSCSI 或光纤通道）上的 LUN（Logical Unit Number）。RDM 可以用于支持虚拟机中运行的 SAN 快照或其他分层应用程序，及 Microsoft 群集服务。

VMware vSphere 存储架构由各种抽象层组成，这些抽象层隐藏并管理物理存储子系统之间的复杂性和差异^[3]，如图 5-10 所示。

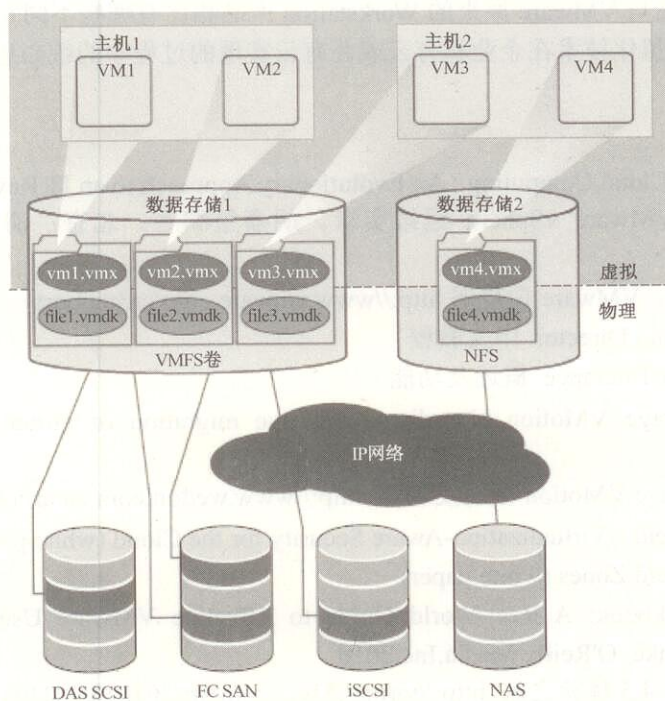


图 5-10 VMware 虚拟存储架构

对于每个虚拟机内的应用程序和客户机操作系统，存储子系统显示为与一个或多个虚拟 SCSI 磁盘相连的虚拟 SCSI 控制器。虚拟机只能发现并访问这些类型的 SCSI 控制器，包括 BusLogic 并行、LSI 逻辑并行、LSI 逻辑 SAS 和 VMware 准虚拟。虚拟 SCSI 磁盘通过数据中心的数据存储元素置备。数据存储就像一个存储设备，为多个物理主机上的虚拟机提供存储空间。数据存储抽象概念是一种模型，可将存储空间分配到虚拟机，使客户机不必使用复杂的基础物理存储技术。客户机虚拟机不对光纤通道 SAN、iSCSI SAN、直接

连接存储器和 NAS 公开。

每个虚拟机被作为一组文件存储在数据存储的目录中。这类文件可以作为普通文件在客户磁盘上进行操作,包括复制、移动、备份等。在无需关闭虚拟机的情况下,可向虚拟机添加新虚拟磁盘。此时,系统将在 VMFS 中创建虚拟磁盘文件(.vmdk 文件),从而为添加的虚拟磁盘或与虚拟机关联的现有虚拟磁盘文件提供新存储。每个数据存储都是存储设备上的物理 VMFS 卷。NAS 数据存储是带有 VMFS 特征的 NFS 卷,数据存储可以跨多个物理存储子系统。单个 VMFS 卷可包含物理主机上本地 SCSI 磁盘阵列、光纤通道 SAN 磁盘场或 iSCSI SAN 磁盘场中的一个或多个 LUN。添加到任何物理存储子系统的新 LUN 可被检测到,并可供所有的现有数据存储或新数据存储使用。先前创建的存储器容量可以扩展,此时不必关闭物理主机或存储子系统。如果 VMFS 卷内的任何 LUN 出现故障或不可用,则只有那些与该 LUN 关联的虚拟机才受影响。

习题

1. VMware 云产品包括哪些最主要的产品? 各个产品在云架构和实现中的作用是什么?
2. vSphere 产品与 VMware 原来的 Workstation 产品相比有哪些不同?
3. VMware 虚拟化技术在企业私有云或公有云实现的过程中的核心思想是什么?

参考文献

- [1] VMware and Cloud Computing : An Evolutionary Approach to an IT Revolution
- [2] 虚拟智慧: VMware vSphere 运维实录。胡嘉玺编著。北京:清华大学出版社, 2011.2
- [3] vSphere 简介。VMware 白皮书 <http://www.vmware.com/cn/support/>
- [4] VMware vCloud Director 中文教程
- [5] VMware Fault Tolerance 概述及功能
- [6] VMware Storage VMotion Non-disruptive, live migration of virtual machine storage (white paper)
- [7] VMware Storage VMotion 概述及功能 <http://www.wedoit.com.cn/article.php?id=57>
- [8] VMware vShield --Virtualization-Aware Security for the Cloud (white paper)
- [9] VMware vShield Zones (white paper)
- [10] VMware Cookbook: A Real-World Guide to Effective VMware Use. Ryan Troy and Matthew Helmke. O'Reilly Media, Inc. 2010
- [11] VMware View 4.5 体验之旅. http://virtual.51cto.com/art/201012/241131_1.htm
- [12] VMware VMotion. www.vmware.com/files/pdf/VMware-VMotion-DS-EN.pdf
- [13] VMware Storage VMotion. www.vmware.com/files/pdf/VMware-Storage-VMotion-DS-EN.pdf
- [14] Understanding VMware Consolidated Backup. www.vmware.com/pdf/vi3_consolidated_backup.pdf
- [15] VMware Data Recovery. www.vmware.com/pdf/vdr_10_admin.pdf
- [16] VMware High Availability. www.vmware.com/pdf/ha_datasheet.pdf
- [17] 深入介绍 VMware vCloud Director. http://labs.chinamobile.com/mblog/390324_64570

第6章 Hadoop: Google 云计算的开源实现

自从云计算的概念被提出,不断地有 IT 厂商推出自己的云计算平台。Amazon 的 AWS、微软的 Azure 和 IBM 的蓝云等都是云计算的典型代表,但它们都是商业性平台,对于想要继续研究和发展云计算技术的个人和科研团体来说,无法获得更多的了解,Hadoop 的出现给研究者带来了希望。本章将重点介绍 Hadoop 的 HDFS、MapReduce 和 HBase,以及 Hadoop 的具体应用。

6.1 Hadoop 简介

Hadoop^[1]是 Apache 开源组织的一个分布式计算框架,可以在大量廉价的硬件设备组成的集群上运行应用程序,为应用程序提供了一组稳定可靠的接口,旨在构建一个具有高可靠性和良好扩展性的分布式系统。随着云计算的逐渐流行,这一项目被越来越多的个人和企业所运用。Hadoop 的核心是 HDFS、MapReduce 和 HBase,它们分别是 Google 云计算最核心技术 GFS、MapReduce 和 Bigtable 的开源实现(表 6-1)。

表 6-1 Hadoop 云计算系统与 Google 云计算系统

Hadoop 云计算系统	Google 云计算系统
Hadoop HDFS	Google GFS
Hadoop MapReduce	Google MapReduce
Hadoop HBase	Google Bigtable
Hadoop ZooKeeper	Google Chubby
Hadoop Pig	Google Sawzall

Hadoop 源于另外两个开源项目 Lucene 和 Nutch,它们是一脉相承的关系。Lucene 是一个用 Java 开发的开源高性能全文检索工具包,可以很方便地嵌入到各种实际应用中,实现搜索/索引功能;Nutch 是第一个开源的 Web 搜索引擎,它在 Lucene 的基础上增加了网络爬虫、Web 相关的一些功能及一些解析各类文档格式的插件等,还包含一个分布式文件系统用于存储数据。从 Nutch 0.8.0 开始,将其中实现分布式文件系统和 MapReduce 算法的代码独立出来,形成了一个新的开源项目,这就是 Hadoop^[2]。

Hadoop 主要由以下几个子项目组成。

(1) Hadoop Common: 即原来的 Hadoop Core。这是整个 Hadoop 项目的核心,其他的 Hadoop 子项目都是在 Hadoop Common 的基础上发展的。

(2) Avro: Hadoop 的 RPC(远程过程调用)方案。

(3) Chukwa: 一个用来管理大型分布式系统的数据采集系统。

(4) HBase: 支持结构化数据存储的分布式数据库,是 Bigtable 的开源实现。

(5) HDFS (Hadoop Distributed File System): 提供高吞吐量的分布式文件系统, 是 GFS 的开源实现。

(6) Hive: 提供数据摘要和查询功能的数据仓库。

(7) MapReduce: 大型数据的分布式处理模型, 是 Google 的 MapReduce 的开源实现。

(8) Pig: 是在 MapReduce 上构建的一种高级的数据流语言, 它是 Sawzall 的开源实现。Sawzall 是一种建立在 MapReduce 基础上的领域语言, 它的程序控制结构 (如 if、while 等) 与 C 语言无异, 但它的领域语言语义使它完成相同功能的代码比 MapReduce 的 C++ 代码简洁得多。

(9) ZooKeeper: 用于解决分布式系统中一致性问题, 是 Chubby 的开源实现。

在这些子项目中, Pig 最初是由 Yahoo 的网格部门开发的, 后来捐献给了 Apache 基金会。Avro 和 Chukwa 刚加入不久, 目前还不是很成熟。从实现的功能来看, Hadoop 几乎就是 Google 的一个“翻版”, 几乎每个子项目都是 Google 某项技术的开源实现。

除了是开源的之外, Hadoop 还有很多优点。

(1) 可扩展。不论是存储的可扩展还是计算的可扩展都是 Hadoop 的设计根本。

(2) 经济。Hadoop 可以运行在廉价的 PC 上。

(3) 可靠。HDFS 的备份恢复机制及 MapReduce 的任务监控机制保证了分布式处理的可靠性。

(4) 高效。分布式文件系统的高效数据交互实现及 MapReduce 结合 Local Data 处理的模式, 为高效处理海量的信息做了基础准备。

目前此项目正在进行中, 虽然现在还没有到达 1.0 版本, 和 Google 系统还有很大差距, 但是前景非常好, 值得我们关注。

6.2 Hadoop 分布式文件系统 HDFS

Hadoop 分布式文件系统 HDFS^[3]可以部署在廉价硬件之上, 能够高容错、可靠地存储海量数据 (可以达到 TB 甚至 PB 级)。它可以和 MapReduce 编程模型很好地结合, 能够为应用程序提供高吞吐量的数据访问, 适用于大数据集应用程序。

6.2.1 设计前提与目标

HDFS 的设计前提与目标^[3]如下。

(1) 硬件错误是常态而不是异常。HDFS 被设计为运行在普通硬件上, 所以硬件故障是很正常的。HDFS 可能由成百上千的服务器构成, 每个服务器上存储着文件系统的部分数据, 而 HDFS 的每个组件随时都有可能出现故障。因此, 错误检测并快速自动恢复是 HDFS 的最核心设计目标。

(2) 流式数据访问。运行在 HDFS 上的应用主要是以流式读为主, 做批量处理; 更注重数据访问的高吞吐量。

(3) 超大规模数据集。HDFS 的一般企业级的文件大小可能都在 TB 级甚至 PB 级, 支持大文件存储, 而且提供整体上高的数据传输带宽, 一个单一的 HDFS 实例应该能支撑数以千万计的文件, 并且能在一个集群里扩展到数百个节点。

(4) 简单一致性模型。HDFS 的应用程序一般对文件实行一次性写、多次读的访问模

式。文件一旦创建、写入和关闭之后就不需要再更改了。这样就简化了数据一致性问题，高吞吐量的数据访问才成为可能。

(5) 移动计算比移动数据更简单。对于大文件来说，移动数据比移动计算的代价要高。操作海量数据时效果越加明显，这样可以提高系统的吞吐量和减少网络的拥塞。

(6) 异构软硬件平台间的可移植性。这种特性便于 HDFS 作为大规模数据应用平台的推广。

6.2.2 体系结构

HDFS 是一个主从结构的体系，HDFS 集群有一个 NameNode 和很多个 DataNode 组成。NameNode 管理文件系统的元数据，DataNode 存储实际的数据。客户端联系 NameNode 以获取文件的元数据，而真正的文件 I/O 操作是直接和 DataNode 进行交互的。

NameNode 就是主控制服务器，负责维护文件系统的命名空间（Namespace）并协调客户端对文件的访问，记录命名空间内的任何改动或命名空间本身的属性改动。DataNode 负责它们所在的物理节点上的存储管理，HDFS 开放文件系统的命名空间以便让用户以文件的形式存储数据。HDFS 的数据都是“一次写入、多次读取”，典型的块大小是 64MB，HDFS 的文件通常是按照 64MB 被切分成不同的数据块（Block），每个数据块尽可能地分散存储于不同的 DataNode 中。NameNode 执行文件系统的命名空间操作，比如打开、关闭、重命名文件或目录，还决定数据块到 DataNode 的映射。DataNode 负责处理客户的读写请求，依照 NameNode 的命令，执行数据块的创建、复制、删除等工作。图 6-1 是 HDFS 的结构示意图。例如客户端要访问一个文件，首先，客户端从 NameNode 获得组成文件的数据块的位置列表，也就是知道数据块被存储在哪些 DataNode 上；然后客户端直接从 DataNode 上读取文件数据。NameNode 不参与文件的传输。

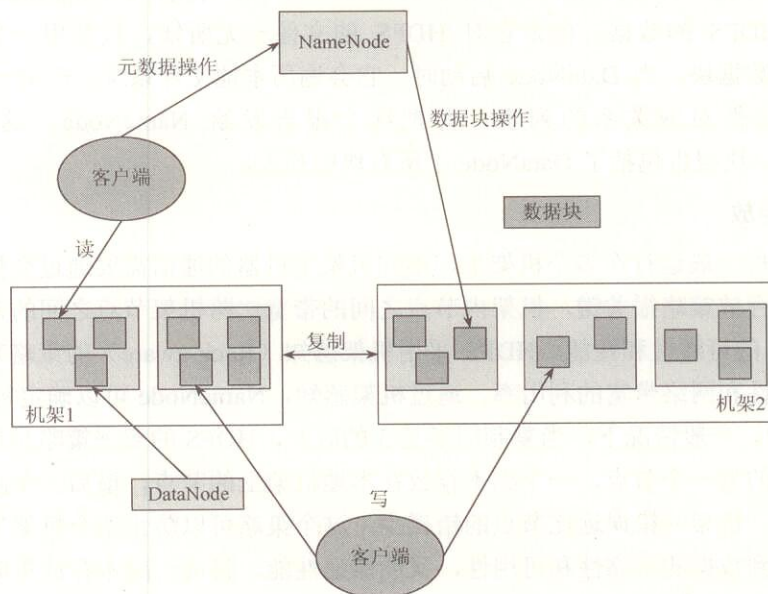


图 6-1 HDFS 的结构示意图

HDFS 典型的部署是在一个专门的机器上运行 NameNode，集群中的其他机器各运行一个 DataNode；也可以在运行 NameNode 的机器上同时运行 DataNode，或者一台机器上运行多个 DataNode。这种一个集群只有一个 NameNode 的设计大大简化了系统架构。

NameNode 使用事务日志（EditLog）记录 HDFS 元数据的变化，使用映像文件（FsImage）存储文件系统的命名空间，包含文件的映射、文件的属性信息等。事务日志和映像文件都存储在 NameNode 的本地文件系统。

NameNode 启动时，从磁盘中读取映像文件和事务日志，把事务日志的事务都应用到内存中的映像文件上，然后将新的元数据刷新到本地磁盘的新的映像文件中，这样可以截去旧的事务日志，这个过程称为检查点（Checkpoint）。HDFS 还有 Secondary NameNode 节点，它辅助 NameNode 处理映像文件和事务日志。NameNode 启动的时候合并映像文件和事务日志，而 Secondary NameNode 会有周期地从 NameNode 上复制映像文件和事务日志到临时目录，合并生成新的映像文件后再重新上传到 NameNode，NameNode 更新映像文件并清理事务日志，使得事务日志的大小始终控制在可配置的限度下。

6.2.3 保障可靠性的措施

HDFS 的主要设计目标之一就是在故障情况下也能保证数据存储的可靠性。HDFS 具备了较为完善的冗余备份和故障恢复机制^[3]，可以实现在集群中可靠地存储海量文件。

1. 冗余备份

HDFS 将每个文件存储成一系列数据块（Block），默认块大小为 64MB（可配置）。为了容错，文件的所有数据块都会有副本（副本数量即复制因子，可配置）。HDFS 的文件都是一次性写入的，并且严格限制为任何时候都只有一个写用户。DataNode 使用本地文件系统存储 HDFS 的数据，但是它对 HDFS 的文件一无所知，只是用一个个文件存储 HDFS 的每个数据块。当 DataNode 启动时，它会遍历本地文件系统，产生一份 HDFS 数据块和本地文件对应关系的列表，并把这个报告发给 NameNode，这就是块报告（Blockreport）。块报告包括了 DataNode 上所有块的列表。

2. 副本存放

HDFS 集群一般运行在多个机架上，不同机架上机器的通信需要通过交换机。通常情况下，副本的存放策略很关键，机架内节点之间的带宽比跨机架节点之间的带宽要大，它能影响 HDFS 的可靠性和性能。HDFS 采用机架感知（Rack-aware）的策略来改进数据的可靠性、可用性和网络带宽的利用率。通过机架感知，NameNode 可以确定每个 DataNode 所属的机架 ID。一般情况下，当复制因子是 3 的时候，HDFS 的部署策略是将一个副本放在同一机架上的另一个节点，一个副本存放在本地机架上的节点，最后一个副本放在不同机架上的节点。机架的错误远比节点的错误少，这个策略可以防止整个机架失效时数据丢失，不会影响到数据的可靠性和可用性，又能保证性能。目前，副本存放策略还正在开发中。图 6-2 体现了复制因子为 3 的情况下，各数据块的分布情况。

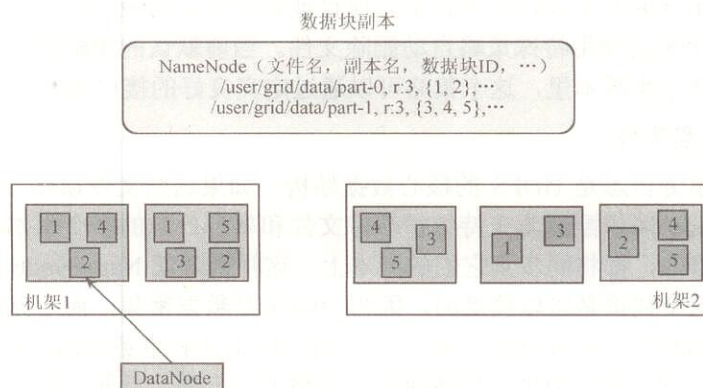


图 6-2 复制因子为 3 时数据块分布情况

3. 心跳检测

NameNode 周期性地从集群中的每个 DataNode 接受心跳包和块报告, 收到心跳包说明该 DataNode 工作正常。NameNode 会标记最近没有心跳的 DataNode 为死机, 不会发给它们任何新的 I/O 请求。任何存储在死机的 DataNode 的数据将不再有效, DataNode 的死机会造成一些数据块的副本数下降并低于指定值。NameNode 会不断检测这些需要复制的数据块, 并在需要的时候重新复制。重新复制的引发可能有多种原因, 比如 DataNode 不可用、数据副本的损坏、DataNode 上的磁盘错误或复制因子增大等。

4. 安全模式

系统启动时, NameNode 会进入一个安全模式。此时不会出现数据块的写操作。NameNode 会收到各个 DataNode 拥有的数据块列表对的数据块报告, 因此 NameNode 获得所有的数据块信息。数据块达到最小副本数时, 该数据块就被认为是安全的。在一定比例(可配置)的数据块被 NameNode 检测确认是安全之后, 再等待若干时间, NameNode 自动退出安全模式状态。当检测到副本数不足的数据块时, 该块会被复制到其他数据节点, 以达到最小副本数。

5. 数据完整性检测

多种原因会造成从 DataNode 获取的数据块有可能是损坏的。HDFS 客户端软件实现了对 HDFS 文件内容的校验和(Checksum)检查, 在 HDFS 文件创建时, 计算每个数据块的校验和, 并将校验和作为一个单独的隐藏文件保存在命名空间下。当客户端获取文件后, 它会检查从 DataNode 获得的数据块对应的校验和是否和隐藏文件中的相同, 如果不同, 客户端就会认为数据块有损坏, 将从其他 DataNode 获取该数据块的副本。

6. 空间回收

文件被用户或应用程序删除时, 并不是立即就从 HDFS 中移走, 而是先把它移动到 /trash 目录里。只要还在这个目录里, 文件就可以被迅速恢复。文件在这个目录里的时间是可以配置的, 超过了这个时间, 系统就会把它从命名空间中删除。文件的删除操作会引起相应数据块的释放, 但是从用户执行删除操作到从系统中看到剩余空间的增加可能会有一个时间延迟。只要文件还在 /trash 目录里, 用户可以取消删除操作。当用户想取消时,

可以浏览这个目录并取回文件，这个目录只保存被删除文件的最后副本。这个目录还有一个特性，就是 HDFS 会使用特殊策略自动删除文件。当前默认的策略是：文件超过 6 个小时后自动删除，在未来版本里，这个策略可以通过定义良好的接口来配置。

7. 元数据磁盘失效

映像文件和事务日志是 HDFS 的核心数据结构。如果这些文件损坏，将会导致 HDFS 不可用。NameNode 可以配置为支持维护映像文件和事务日志的多个副本，任何对映像文件或事务日志的修改，都将同步到它们的副本上。这样会降低 NameNode 处理命名空间事务的速度，然而这个代价是可以接受的，因为 HDFS 是数据密集，而非元数据密集的。当 NameNode 重新启动时，总是选择最新的一致映像文件和事务日志。在 HDFS 集群中 NameNode 是单点存在的，如果它出现故障，必须手动干预。目前，还不支持自动重启或切换到另外的 NameNode。

8. 快照

快照支持存储某个时间的数据复制，当 HDFS 数据损坏时，可以回滚到过去一个已知正确的时间点。HDFS 目前还不支持快照功能。

6.2.4 提升性能的措施

HDFS 被设计为支持非常大的文件，它的应用都是处理大数据集的。高效的数据存储也是 Hadoop 的优点之一。

1. 副本选择

HDFS 会尽量使用离程序最近的副本来满足用户请求，这样可以减少总带宽消耗和读延时。如果在读取程序的同一个机架有一个副本，那么就使用这个副本；如果 HDFS 机群跨了多个数据中心，那么读取程序将优先考虑本地数据中心的副本。

2. 负载均衡

HDFS 的架构支持数据均衡策略。如果某个 DataNode 的剩余磁盘空间下降到一定程度，按照均衡策略，系统会自动把数据从这个 DataNode 移动到其他节点。当出现对某个文件有很高的需求时，系统可能会启动一个计划创建该文件的新副本，并重新平衡集群中的其他数据。目前这些均衡策略还没有实现。

3. 客户端缓存

客户端创建文件的请求不是立即到达 NameNode，HDFS 客户端先把数据缓存到本地的一个临时文件，程序的写操作透明地重定向到这个临时文件。当这个临时文件累积的数据超过一个块的大小（64MB）时，客户端才会联系 NameNode。NameNode 在文件系统中插入文件名，给它分配一个数据块，告诉客户端 DataNode 的 ID 和目标数据块 ID，这样客户端就把数据从本地的缓存刷新到指定的数据块中。当文件关闭后，临时文件中剩余的未刷新数据也会被传输到 DataNode 中，然后客户端告诉 NameNode 文件已关闭，此时 NameNode 才将文件创建操作写入日志进行存储。如果 NameNode 在文件关闭之前死机，那么文件将会丢失。如果不采用客户端缓存，网络速度和拥塞都会对输出产生很大的影响。

4. 流水线复制

当客户端要写数据到 HDFS 的文件中时，就像前面介绍的那样，数据一开始会写入本

地临时文件。假设该文件的复制因子是 3, 当本地临时文件累积到一个数据块的大小时, 客户端会从 NameNode 获取一个副本存放的 DataNode 列表, 列表中的 DataNode 都将保存那个数据块的一个副本。客户端首先向第一个 DataNode 传输数据, 第一个 DataNode 一小块一小块 (4KB) 地接收数据, 写入到本地库的同时, 把接受到的数据传输给列表中的第二个 DataNode; 第二个 DataNode 以同样的方式边收边传, 把数据传输给第三个 DataNode; 第三个 DataNode 把数据写入本地库。DataNode 从前一个节点接收数据的同时, 即时把数据传给后面的节点, 这就是流水线复制。

6.2.5 访问接口

访问 HDFS 可以通过很多种方式。可以通过 Java API 调用, 也可以使用 C 语言封装的 API; HDFS 还提供了浏览器访问的方式; 目前通过 WebDAV 协议访问的工作正在开发中。

1. Hadoop API

Hadoop API^[4]包括以下几种主要的包 (Package)。

- (1) org.apache.hadoop.conf: 定义了系统参数的配置文件处理 API。
- (2) org.apache.hadoop.dfs: Hadoop 分布式文件系统 (HDFS) 模块的实现。
- (3) org.apache.hadoop.fs: 定义了抽象的文件系统 API。
- (4) org.apache.hadoop.io: 定义了通用的 I/O API, 用于针对网络、数据库、文件等数据对象做读/写操作。
- (5) org.apache.hadoop.ipc: 用于网络服务端和客户端的工具, 封装了网络异步 I/O 的基础模块。
- (6) org.apache.hadoop.mapred: Hadoop 分布式计算系统 (MapReduce) 模块的实现, 包括任务的分发调度等。
- (7) org.apache.hadoop.metrics: 定义了用于性能统计信息的 API, 主要用于 mapred 和 dfs 模块。
- (8) org.apache.hadoop.record: 定义了针对记录的 I/O API 类及一个记录描述语言翻译器, 用于简化将记录序列化成语言中性的格式 (Language-neutral Manner)。
- (9) org.apache.hadoop.tools: 定义了一些通用的工具。
- (10) org.apache.hadoop.util: 定义了一些公用的 API。

在 org.apache.hadoop.fs 众多类中, 最重要的是 FileSystem 抽象类。为了便于方便编程使用和提供一定的文件访问安全性, HDFS 将上层客户端需要的操作封装在 FileSystem 类中, 通过这个类提供给上层文件操作的抽象。它定义了文件系统中的一些基本操作, 如 create、rename、delete、mkdirs 等, 还定义了一些分布式文件系统具有的操作, 如 copyFromLocalFile, copyToLocalFile 等。其中, LocalFileSystem 和 DistributedFileSystem 继承于此类, 分别实现了本地文件系统和分布式文件系统。

2. 浏览器接口

典型的 HDFS 安装会配置一个 Web 服务器开放自己的命名空间, 其 TCP 端口是可配的, 这样用户就可以通过 Web 浏览器浏览 HDFS 的命名空间并查看集群当前的基本状态

和信息。在默认配置下 <http://namenode-name:50070> 这个页面列出了集群里的所有 DataNode 和集群的基本状态。

6.3 分布式数据处理 MapReduce

Hadoop 实现了 Google 的 MapReduce 编程模型。MapReduce 是一种分布式计算模型，也是 Hadoop 的核心。它是开源的，任何人都可以使用这个框架进行并行编程。基于这个模型，分布式并程序的编写变得非常简单。

6.3.1 逻辑模型

MapReduce 把运行在大规模集群上的并行计算过程抽象为两个函数：Map 和 Reduce，也就是映射和化简。简单说，MapReduce 就是“任务的分解与结果的汇总”。Map 把任务分解成为多个任务，Reduce 把分解后多任务处理的结果汇总起来，得到最终结果。

适合用 MapReduce 处理的任务有一个基本要求：待处理的数据集可以分解成许多小的数据集，而且每一个小数据集都可以完全并行地进行处理。

图 6-3 介绍了用 MapReduce 处理大数据集的过程。一个 MapReduce 操作分为两个阶段：映射阶段和化简阶段。

在映射阶段，MapReduce 框架将用户输入的数据分割为 M 个片断，对应 M 个 Map 任务。每一个 Map 操作的输入是数据片断中的键值对 $\langle K1, V1 \rangle$ 集合，Map 操作调用用户定义的 Map 函数，输出一个中间态的键值对 $\langle K2, V2 \rangle$ 集合。接着，按照中间态的 $K2$ 将输出的数据集进行排序，并生成一个新的 $\langle K2, \text{list}(V2) \rangle$ 元组，这样可以使得对应同一个键的所有值的数据都在一起。然后，按照 $K2$ 的范围将这些元组分割为 R 个片断，对应 Reduce 任务的数目。

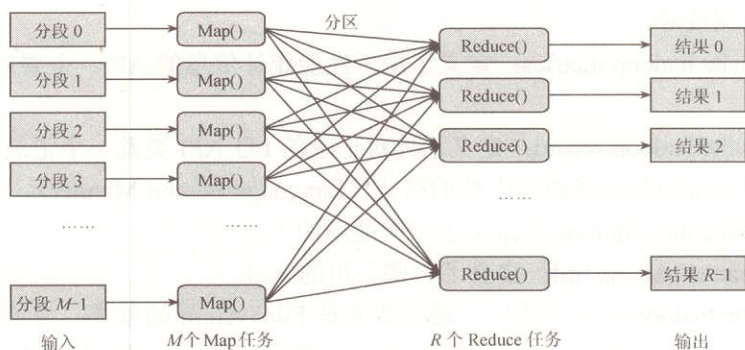


图 6-3 MapReduce 处理大数据集的过程

在化简阶段，每一个 Reduce 操作的输入是一个 $\langle K2, \text{list}(V2) \rangle$ 片断，Reduce 操作调用用户定义的 Reduce 函数，生成用户需要的键值对 $\langle K3, V3 \rangle$ 进行输出。

6.3.2 实现机制

1. 分布式并行计算

MapReduce 框架是由 JobTracker 和 TaskTracker 这两类服务调度的^{[5][6]}。JobTracker 是

主控服务, 只有一个, 负责调度和管理 TaskTracker, 把 Map 任务和 Reduce 任务分配给空闲的 TaskTracker, 让这些任务并行运行, 并负责监控任务的运行情况。TaskTracker 是从服务, 可以有多个, 负责执行任务。如果某个 TaskTracker 出故障了, JobTracker 会把其负责的任务分配给其他空闲的 TaskTracker 重新运行。

2. 本地计算

通常, MapReduce 框架和分布式文件系统是运行在一组相同的节点上的, 也就是说, 计算节点和存储节点通常在一起。这种配置允许框架在那些存储数据的节点上高效地调度任务, 这可以使整个集群的网络带宽被非常高效地利用。

3. 任务粒度

对于小数据集, 一般小于或等于 HDFS 中数据块的大小, 这使得一个小数据集位于一台计算机上, 有利于计算的数据本地性。一个小数据集启动一个 Map 任务, M 个 Map 任务可以在 N 台计算机上并行运行, 用户可以指定 Reduce 任务的数量。

4. Combine (连接)

Combine 将 Map 任务输出的中间结果集中有相同 key 值的多个 $\langle \text{key}, \text{value} \rangle$ 组合成一个 $\langle \text{key}, \text{list}(\text{value}) \rangle$ 对。Combine 在执行完 Map 函数后紧接着执行, 很多情况下可以直接使用 Reduce 函数, Combine 能减少中间结果的数量, 从而减少数据传输中的网络流量。

5. Partition (分区)

Combine 之后, 把产生的中间结果按 key 的范围划分成 R 份 (R 是预先定义的 Reduce 任务的个数)。划分时通常使用 Hash 函数, 如 $\text{hash}(\text{key}) \bmod R$, 这样可以保证某一范围内的 key, 一定是由一个 Reduce 任务来处理, 可以简化 Reduce 的过程。

6. 读取中间结果

Map 任务的中间结果在做完 Combine 和 Partition 之后, 以文件形式存于本地磁盘。中间结果文件的位置会通知主控 JobTracker, JobTracker 再通知 Reduce 任务到哪一个 DataNode 上去取中间结果。注意所有的 Map 任务产生中间结果均按其 key 用同一个 Hash 函数划分成了 R 份, R 个 Reduce 任务各自负责一段 key 区间。每个 Reduce 需要向多个 Map 任务节点取得落在其负责的 key 区间内的中间结果, 然后执行 Reduce 函数, 形成一个最终的结果文件。

7. 任务管道

在某些情况下 Reduce 任务的输出结果并非所需要的最终结果, 这时可以将这些输出结果作为另一个计算任务的输入开始另一个 MapReduce 计算任务。

6.4 分布式结构化数据表 HBase

HBase 数据库是基于 Hadoop 的项目, 是对 Google 的 Bigtable 的开源实现。它与 Google 的 Bigtable 相似, 但也存在许多的不同之处。

6.4.1 逻辑模型

HBase 的数据存放逻辑模型与 Bigtable 类似，HBase 中的表与 Bigtable 类似。用户在表格里存储一系列的数据行，每行包含一个可排序的行关键字、一个可选的时间戳及一些可能有数据的列（稀疏）。数据行有三种基本类型的定义：行关键字（Row Key）、时间戳（Time Stamp）和列（Column）。行关键字是数据行在表中的唯一标识，时间戳是每次数据操作对应关联的时间戳，列定义为：<family>:<label>（<列族>:<标签>），通过这两部分可以唯一地指定一个数据的存储列。对列族的定义和修改需要管理员权限，而标签可以在任何时候添加。HBase 在磁盘上按照列族储存数据，一个列族里的所有项有相同的读/写方式。HBase 的更新操作有时间戳，对每个数据单元，只存储指定个数的最新版本。客户端可以查询某个时间后的最新数据，或者一次得到数据单元的所有版本。

表 6-2 是有关 www.cnn.com 网站的数据存放逻辑视图。表中仅有一行数据，行的唯一标识为 com.cnn.www，它采用了倒排的方式；对这行数据的每一次逻辑修改都有一个时间戳关联对应；共有四个列定义：<contents:>、<anchor:cnnsi.com>、<anchor:my.look.ca>、<mime:>。每一行就相当于传统数据库中的一个表，行关键字是表名，这个表根据列的不同划分，每次操作都会有时间戳关联到具体操作的行^[7]。

表 6-2 数据存放逻辑视图

行 关 键 字	时 间 戳	列"contents:"	列"anchor:"		列"mime:"
"com.cnn.www"	t9		"anchor:cnnsi.com"	"CNN"	
	t8		"anchor:my.look.ca"	"CNN.com"	
	t6	"<html>..."			"text/html"
	t5	"<html>..."			
	t3	"<html>..."			

6.4.2 物理模型

HBase 是按照列存储的稀疏行/列矩阵。物理模型实际上就是把概念模型中的一个行进行分割，并按照列族存储。如表 6-3 所示是在物理上的存储方式。

表 6-3 在物理上的存储方式

行 关 键 字	时 间 戳	列"contents:"	
"com.cnn.www"	t5	"<html>..."	
	t4	"<html>..."	
	t3	"<html>..."	
行关键字	时 间 戳	列 "anchor:"	
"com.cnn.www"	t8	"anchor:cnnsi.com"	"CNN"
	t7	"anchor:look.ca"	"CNN.com"
行关键字	时 间 戳	列 "mime:"	
"com.cnn.www"	t6	"text/html"	

在表 6-3 中，空的单元格不存储。因此查询时间戳为 t7 的“contents:”将返回空值，

查询时间戳为 `t8`, “`anchor:`” 值为 “`look.ca`” 的项也返回空值。若未指明时间戳, 则返回指定列的最新数据值, 并且最新的值在表格里也是最先找到的, 因为它们是按照时间排序的。所以, 若查询 “`contents:`” 而不指明时间戳, 将返回 `t5` 时刻的数据; 查询 “`anchor:`” 的 “`look.ca`” 而不指明时间戳, 将返回 `t7` 时刻的数据^[8]。

6.4.3 子表服务器

在物理上, 表格分为多个子表 (HRegion), 每个子表存储在适当的地方。物理上所有数据都存储在 HDFS 上, 由一些子表服务器来提供数据服务, 一般一台计算机只运行一个子表服务器程序。某一时刻一个子表服务器只管理一个子表。

当客户端进行更新操作时, 首先连接相关的子表服务器, 之后向子表提交变更。提交的数据被添加到子表的 HMemcache 和子表服务器的 HLog。作为缓存服务 HMemcache 在内存中存储最近的更新。HLog 是磁盘上的日志文件, 记录所有的更新操作。客户端的 `commit()` 方法调用直到更新操作写入 HLog 文件后才返回。

在提供服务时, 子表首先查询缓存 HMemcache。若没有, 再查找磁盘上的 HStore。子表中的每个列族都对应着一个 HStore, 一个 HStore 又包括了若干个磁盘上的 HStoreFile 文件。每个 HStoreFile 的结构都类似 B 树, 可以快速地查找。

HRegion.flushcache() 定期被调用, 把 HMemcache 中的内容写到磁盘上 HStore 文件里, 这样给每个 HStore 都增加了一个新的 HStoreFile。之后清空 HMemcache 缓存, 再在 HLog 文件里加入一个特殊的标记, 表示刷新了 HMemcache。

在启动时, 每个子表检查最后的 flushcache() 方法调用之后是否还有写操作在 HLog 文件里未应用。如果没有, 则子表里的全部数据就是磁盘上 HStore 文件内的数据; 如果有, 则子表就把 HLog 文件里的更新操作重新应用一遍, 写入到 HMemcache 里, 再调用 flushcache()。最后, 子表会删除 HLog 文件并开始数据服务。

所以, 调用 flushcache() 方法越少, 工作量就越少, 而 HMemcache 就要占用更多的内存空间, 启动时 HLog 文件也需要更多的时间来恢复数据。而调用 flushcache() 越频繁, HMemcache 占用内存就越少, HLog 文件恢复数据时也就越快, 但是也需要考虑 flushcache() 的资源消耗。

方法 flushcache() 的调用会给每个 HStore 增加一个 HStoreFile。要从一个 HStore 里读数据, 可能需要访问它的所有 HStoreFile, 这是很耗时的, 因此需要定时把多个 HStoreFile 合并成一个 HStoreFile, 这是通过调用 HStore.compact() 方法来实现的。

两个子表都要处于 “下线” 状态时, 调用 HRegion.closeAndMerge() 可以把两个子表合并成一个。当一个子表大到超过某个指定值时, 子表服务器就需要调用 HRegion.closeAndSplit(), 将它分割为两个新的子表。新子表被上报给主服务器, 主服务器来决定哪个子表服务器接管哪个子表。分割过程很快, 这是由于新子表只维护了到旧子表的 HStoreFile 的引用, 一个引用 HStoreFile 的前半部分, 另一个引用后半部分。当引用建立完毕, 旧子表被标记为 “下线” 并继续保存, 直到新子表的紧缩操作将对旧子表的引用全部清除掉时, 旧子表才被删除。

6.4.4 主服务器

HBase 只使用了一个核心来管理所有子表服务器: 主服务器。每个子表服务器都只与

唯一的主服务器联系,主服务器告诉每个子表服务器应该装载哪些子表并进行服务。

主服务器维护子表服务器在任何时刻的活跃标记。当一个新的子表服务器向主服务器注册时,主服务器让新的子表服务器装载若干个子表,也可以不装载。如果主服务器和子表服务器间的连接超时,那么子表服务器将“杀死”自己,之后以一个空白状态重启。主服务器假定子表服务器已“死”,并将其上的子表标记为“未分配”,同时尝试把它们分配给其他子表服务器。

与 Google 的 Bigtable 不同的是^[9],Bigtable 使用分布式锁服务 Chubby 保证了子表服务器访问子表操作的原子性。子表服务器即使和主服务器的连接断掉了,还可以继续服务。它们都依赖于一个核心的网络结构(HMaster 或 Chubby),只要核心还在运行,整个系统就能运行,而 HBase 不具备这样的 Chubby。

每个子表都由它所属的表格名字、首关键字和 region Id 来标识。例如,表名是 hbaserepository,首关键字是 w-nk5YNZ8TBb2uWFIRJo7V==,region Id 是 689060145591-4043。它的唯一标识符就是:

hbaserepository, w-nk5YNZ8TBb2uWFIRJo7V==,6890601455914043

6.4.5 元数据表

子表的元数据存储在另一个子表里,子表的唯一标识符可以作为子表的行标签,映射子表标识符到物理子表服务器位置的表格称为元数据表。

元数据表可能会增长,并且可以分裂为多个子表。为了定位元数据表的各个部分,所有元数据子表的元数据被保存在根子表(ROOT Table)里。启动时,主服务器立即扫描唯一根子表(其名字是硬编码的),这可能需要等待根子表分配到某个子表服务器上。一旦根子表可用,主服务器扫描它得到所有的元数据子表位置,然后主服务器扫描元数据子表。同样的,主服务器可能要等待所有的元数据子表都被分配到子表服务器上。最后,当主服务器扫描完元数据子表,它就知道了所有子表的位置,然后就可以把这些子表分配到子表服务器上去。

6.5 Hadoop 安装

6.5.1 在 Linux 系统中安装 Hadoop

1. 安装环境配置

普通用户更习惯使用 Windows 操作系统。Linux 虚拟机可以在 Windows 操作系统上虚拟 Linux 环境,虽然它并不是真正意义上的操作系统,但是实际效果没有区别。对于习惯了 Windows 操作系统的用户来说,这是一个不错的选择。

1) 安装 Linux 虚拟机

首先,安装软件 VMware Workstation v7.1.3。按照提示一步步完成 Linux 虚拟机的安装,方法与 Linux 操作系统的安装类似。软件安装完成后,可以打开 Linux 虚拟机,进入虚拟 Linux 环境。需要注意的是,虚拟机工具必须安装。

通过 VMware Workstation v7.1.3 软件的虚拟机设置选项,可以在 Windows 主机和

Linux 虚拟机之间建立共享文件夹, 如图 6-4 所示。

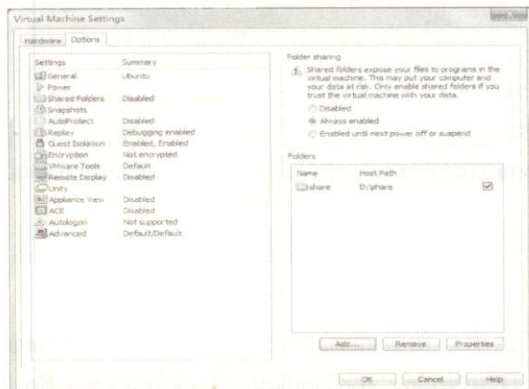


图 6-4 虚拟机设置

添加共享文件夹之后, 可以在 Linux 虚拟机的/mnt/hgfs 目录下看到对应 Windows 主机的共享文件夹。

2) 安装 SSH

Hadoop 运行过程中需要管理远端 Hadoop 守护进程, 如果在安装 Linux 虚拟机时没有安装 SSH Server, 可以使用下面的命令安装:

```
$ sudo apt-get install ssh
```

3) 安装 Java

实验使用 jdk-6u24-linux-i586.bin, 软件放在共享文件夹下。把 Java 安装到目录/usr/java/jdk1.6.0_24, 使用如下命令:

```
$ cd /usr/java/  
$ /mnt/hgfs/share/jdk-6u24-linux-i586.bin
```

4) 解压 Hadoop 安装包

实验使用 hadoop-0.20.2.tar.gz, 软件放在共享文件夹下。把 Hadoop 解压到 Linux usr 目录, 命令如下:

```
$ tar -zxvf /mnt/hgfs/share/hadoop-0.20.2.tar.gz
```

5) 编辑 conf/hadoop-env.sh 文件

把 JAVA_HOME 设置为 Java 安装的根路径, 即通过如下命令修改 export JAVA_HOME=/usr/java/jdk1.6.0_24:

```
$ vi conf/hadoop-env.sh
```

2. 安装步骤

Hadoop 集群支持三种运行模式: 单机模式、伪分布式模式和完全分布式模式。

1) 单机模式

默认情况下, Hadoop 被配置成一个以非分布式模式运行的独立 Java 进程, 适合开始时做调试工作。

下面是 WordCount 实例的运行, 统计一批文本文件中各单词出现的次数, 输出写入到

指定的 output 目录。

```
$ mkdir input
$ cd input
$ echo "hello world" >test1.txt
$ echo "hello hadoop" >test2.txt
$ bin/hadoop jar hadoop-mapred-examples-0.20.2.jar wordcount input output
查看执行结果:
```

```
$ cat output/*
```

2) 伪分布式模式

Hadoop 可以在单节点上以伪分布式模式运行, 用不同的 Java 进程模拟分布式运行中各类节点 (NameNode、DataNode、JobTracker、TaskTracker、Secondary NameNode)。

(1) Hadoop 配置。

Hadoop0.20.2 版本以前的配置文件是 conf/hadoop-default.xml, 但 hadoop0.20.2 及其以后版本中, 该配置文件拆分为 core-site.xml、hdfs-site.xml 和 mapred-site.xml。其中 core-site.xml 和 hdfs-site.xml 是站在 HDFS 角度上的配置文件; core-site.xml 和 mapred-site.xml 是站在 MapReduce 角度上的配置文件。

实验使用如图 6-5、6-6、6-7 所示的配置文件。

core-site.xml 文档内容:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
<name>fs.default.name</name>
<value>hdfs://localhost:9000</value>
</property>
</configuration>
```

图 6-5 实验用 core-site.xml 配置文档

hdfs-site.xml 文档内容:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
<name>dfs.replication</name>
<value>1</value>
</property>
</configuration>
```

图 6-6 实验用 hdfs-site.xml 配置文档

mapred-site.xml 文档内容:

```
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
<name>mapred.job.tracker</name>
<value>localhost:9001</value>
</property>
</configuration>
```

图 6-7 实验用 mapred-site.xml 配置文档

(2) 免密码 SSH 设置。

生成密钥对, 执行如下命令:

```
$ ssh-keygen -t rsa
```

然后一直按【Enter】键, 就会按照默认的选项将生成的密钥对保存在.ssh/id_rsa 文件中, 如图 6-8 所示。

```
grid@ubuntu:~$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/grid/.ssh/id_rsa):
Created directory '/home/grid/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/grid/.ssh/id_rsa.
Your public key has been saved in /home/grid/.ssh/id_rsa.pub.
The key fingerprint is:
9a:72:a3:77:58:84:77:82:b9:b5:6f:bc:19:c7:47:08 grid@ubuntu
The key's randomart image is:
[ RSA 2048 ]
+-----+
+ . E . +
+ . . . +
+ S . . +
+ . . . +
+ . . . +
+ . . . +
+ . . . +
+ . . . +
+ . . . +
+ . . . +
+-----+
grid@ubuntu:~$
```

图 6-8 将密钥对保存在.ssh/id_rsa 文件中

进入.ssh 目录, 执行如下命令:

```
$ cp id_rsa.pub authorized_keys
```

此后执行\$ ssh localhost, 可以实现用 SSH 连接并且不需要输入密码。

(3) Hadoop 运行。

(a) 格式化分布式文件系统。使用如下命令进行格式化:

```
$ bin/hadoop namenode -format
```

(b) 启动 Hadoop 守护进程。启动守护进程的命令如下:

```
$ bin/start-all.sh
```

成功执行后将会在本机上启动 NameNode、DataNode、JobTracker、TaskTracker 和 Secondary NameNode 5 个新的 Java 进程。

访问 <http://localhost:50070> 可以查看 NameNode 以及整个分布式文件系统的状态, 浏览分布式文件系统中的文件以及日志 (log) 等; 访问 <http://localhost:50030> 可以查看 JobTracker 的运行状态; 访问 <http://localhost:50060> 可以查看 TaskTracker 的运行状态。

(c) 运行 WordCount 实例。将本地文件系统中 input 目录复制到 HDFS 的根目录下, 重命名为 in, 运行 Hadoop 自带的 WordCount 实例:

```
$ bin/hadoop dfs -copyFromLocal input in
```

```
$ bin/hadoop jar hadoop-mapred-examples-0.20.2.jar wordcount in out
```

out 为数据处理完成后的输出目录, 默认为在 Hadoop 根目录下, 运行之前必须清空或者删除 out 目录, 否则会报错。

任务执行完, 使用下面的命令查看 Hadoop 分布式文件系统上数据处理的结果:

```
$ bin/hadoop dfs -cat out/*
```

也可以把输出文件从 Hadoop 分布式文件系统复制到本地文件系统查看:

```
$ bin/hadoop dfs -get out output
```

```
$ cat output/*
```

(d) 停止 Hadoop 守护进程。最后需要停止 Hadoop 守护进程, 命令如下:

```
$ bin/stop-all.sh
```

3) 完全分布式模式

对于 Hadoop, 不同的系统会有不同的节点划分方式。在 HDFS 看来, 节点分为 NameNode 和 DataNode, 其中 NameNode 只有一个, DataNode 可以有多个; 在 MapReduce 看来, 节点又分为 JobTracker 和 TaskTracker, 其中 JobTracker 只有一个, TaskTracker 可以有多个。NameNode 和 JobTracker 可以部署在不同的机器上, 也可以部署在同一机器上。部署 NameNode 和/或 JobTracker 的机器是 master (主服务器), 其余的都是 slaves (从服务器)。用户甚至可以将 NameNode、DataNode、JobTracker 和 TaskTracker 全部部署在一台机器上。详细的配置方法介绍如下。

(1) 配置 NameNode 和 DataNode。

配置成功的关键在于确保各机器的主机名和 IP 地址之间能正确解析。修改每台机器的 /etc/hosts, 如果该台机器作 NameNode 用, 则需要在文件中添加集群中所有机器的 IP 地址及其对应的主机名; 如果该台机器仅作 DataNode 用, 则只需要在文件中添加本机和 NameNode 的 IP 地址及其对应的主机名。

假设把 ubuntuamenode 作为 NameNode, 配置该节点的/etc/hosts, 如图 6-9 所示。

```
grid@ubuntuamenode:/$ cat /etc/hosts
127.0.0.1 localhost
192.168.122.136 ubuntuamenode
192.168.122.140 ubuntuata1
192.168.122.141 ubuntuata2
```

图 6-9 配置 ubuntuamenode 的/etc/hosts

把 ubuntuata1, ubuntuata2 作为 DataNode, 分别配置它们的/etc/hosts, 如图 6-10 所示。

grid@ubuntuata1:~\$ cat /etc/hosts	grid@ubuntuata2:~\$ cat /etc/hosts
127.0.0.1 localhost	127.0.0.1 localhost
192.168.122.140 ubuntuata1	192.168.122.141 ubuntuata2
192.168.122.136 ubuntuamenode	192.168.122.136 ubuntuamenode

图 6-10 ubuntuata1 和 ubuntuata2 的 hosts 配置

(2) 在所有的机器上建立相同的用户 grid。

这一步通过以下命令实现:

```
$ useradd -m grid
$ passwd grid
```

成功建立 grid 用户后, 输入的密码就是该用户的密码。

(3) SSH 配置。

该配置主要是为了实现在机器之间执行指令时不需要输入密码。在所有机器上建立.ssh 目录, 执行:

```
$ mkdir .ssh
```

在 ubuntuamenode 上生成密钥对, 执行:

```
$ ssh-keygen -t rsa
```

然后一直按【Enter】键, 就会按照默认的选项将生成的密钥对保存在.ssh/id_rsa 文件中。接着执行如下命令:

```
$ cd ~/.ssh
```



```
$cp id_rsa.pub authorized_keys
$scp authorized_keys ubuntu1:/home/grid/.ssh
$scp authorized_keys ubuntu2:/home/grid/.ssh
```

最后进入所有机器的.ssh目录, 改变authorized_keys文件的许可权限:

```
$chmod 644 authorized_keys
```

这时从ubuntunamenode向其他机器发起SSH连接, 只有在第一次登录时需要输入密码, 以后则不需要。

(4) 在所有机器上配置Hadoop。

首先在ubuntunamenode上配置, 执行如下的解压缩命令:

```
$tar -zxvf /mnt/hgfs/share/hadoop-020.2.tar.gz
```

(a) 编辑core-site.xml、hdfs-site.xml和mapred-site.xml, 如图6-5、图6-6、图6-7所示。

(b) 编辑conf/masters, 修改为master的主机名, 每个主机名一行, 此外即为ubuntunamenode。

(c) 编辑conf/slaves, 加入所有slaves的主机名, 即ubuntu1和ubuntu2。

(d) 把Hadoop安装文件复制到其他机器上:

```
$scp -r hadoop-0.20.2 ubuntu1:/home/grid
```

```
$scp -r hadoop-0.20.2 ubuntu2:/home/grid
```

(e) 编辑所有机器的conf/hadoop-env.sh文件, 将JAVA_HOME变量设置为各自JAVA安装的根目录, 不同机器可以使用不同的JAVA版本。

现在Hadoop已经在集群上部署完毕。如果要新加入或删除节点, 仅需修改NameNode的master和slaves。

(5) Hadoop运行。

格式化分布式文件系统。启动守护进程的命令如下:

```
$bin/hadoop namenode -format
```

执行后的结果如图6-11所示。

```
grid@ubuntunamenode:~/hadoop-0.20.2$ bin/hadoop namenode -format
11/04/17 05:59:54 INFO namenode.NameNode: STARTUP_MSG:
STARTUP_MSG: Starting NameNode
STARTUP_MSG: host = ubuntunamenode/192.168.122.136
STARTUP_MSG: args = [-format]
STARTUP_MSG: version = 0.20.2
STARTUP_MSG: build = https://svn.apache.org/repos/asf/hadoop/common/branches/h
ranch-0.20 -r 911707: compiled by 'chrisdo' on Fri Feb 19 08:07:34 UTC 2010
11/04/17 05:59:55 INFO namenode.FSNamesystem: fsOwner=grid.grid,admin
11/04/17 05:59:55 INFO namenode.FSNamesystem: supergroup=supergroup
11/04/17 05:59:55 INFO namenode.FSNamesystem: isPermissionEnabled=true
11/04/17 05:59:56 INFO common.Storage: Image file of size 94 saved in 0 seconds.
11/04/17 05:59:56 INFO common.Storage: Storage directory /home/grid/hadoopnps/df
Ename has been successfully formatted.
11/04/17 05:59:56 INFO namenode.NameNode: SHUTDOWN_MSG:
SHUTDOWN_MSG: Shutting down NameNode at ubuntunamenode/192.168.122.136
grid@ubuntunamenode:~/hadoop-0.20.2$
```

图 6-11 格式化分布式文件系统

启动Hadoop守护进程。在ubuntunamenode上启动NameNode、JobTracker和Secondary NameNode, 在ubuntu1和ubuntu2上启动DataNode和TaskTracker, 并用jps命令检测启动情况如下:

```
$bin/start-all.sh
```

```
$usr/java/jdk1.6.0_24/bin/jps
```

结果如图6-12所示。

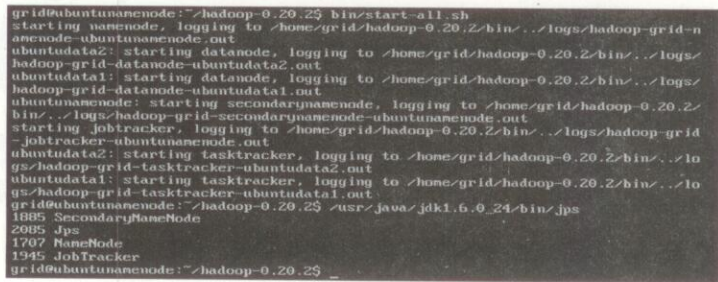


图 6-12 启动 Hadoop 守护进程

用户也可以根据自己的需要来执行如下命令。

- (a) start-all.sh: 启动所有的 Hadoop 守护进程，包括 NameNode、DataNode、JobTracker 和 Tasktrack。
- (b) stop-all.sh: 停止所有的 Hadoop 守护进程。
- (c) start-mapred.sh: 启动 Map/Reduce 守护进程，包括 JobTracker 和 Tasktrack。
- (d) stop-mapred.sh: 停止 Map/Reduce 守护进程。
- (e) start-dfs.sh: 启动 Hadoop DFS 守护进程，包括 NameNode 和 DataNode。
- (f) stop-dfs.sh: 停止 DFS 守护进程。

修改 C:\WINDOWS\system32\drivers\etc\hosts 文件，加入三台虚拟机的 IP 地址及其对应的主机名，即：

127.0.0.1	localhost
192.168.122.136	ubuntunamenode
192.168.122.140	ubuntudata1
192.168.122.141	ubuntudata2

访问 <http://ubuntunamenode:50070> 可以查看 NameNode 以及整个分布式文件系统的状态，浏览分布式文件系统中的文件以及日志等，如图 6-13 所示。

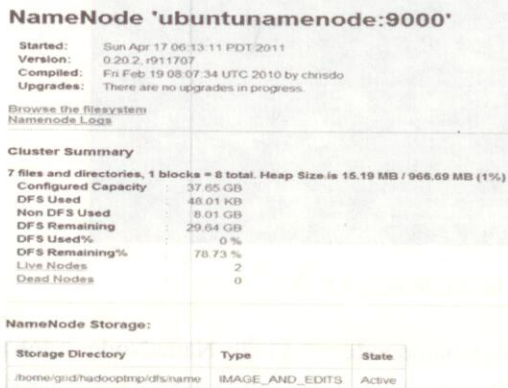


图 6-13 NameNode 运行状态

访问 <http://ubuntunamenode:50030> 可以查看 JobTracker 的运行状态，如图 6-14 所示。

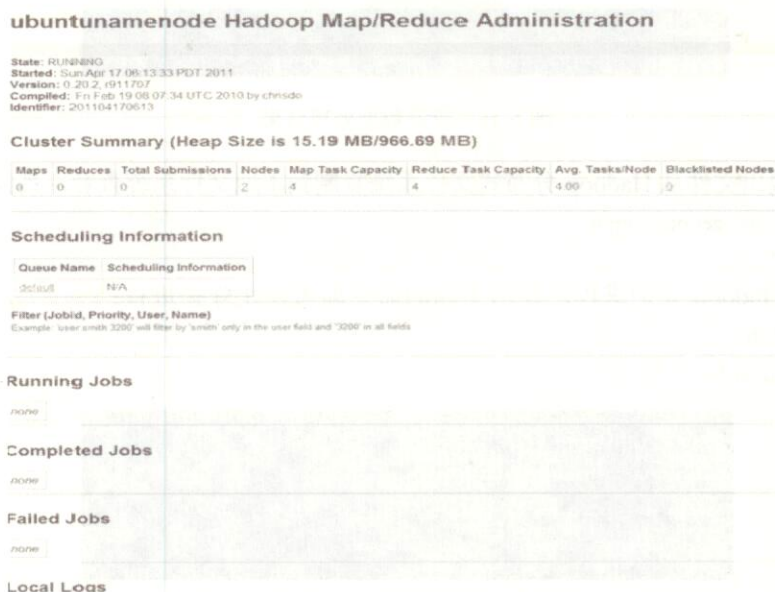


图 6-14 JobTracker 运行状态

(g) 运行 WordCount 实例。将本地文件系统上 input 目录复制到 HDFS 的根目录下，重命名为 in。out 为数据处理完成后的输出目录，默认存放在分布式文件系统用户的根目录下。执行以下的命令：

```
$ bin/hadoop dfs -put input in
$ bin/hadoop jar hadoop-0.20.2-examples.jar wordcount in out
```

运行结果如图 6-15 所示。

任务执行完毕，查看 Hadoop 分布式文件系统上数据处理的结果，执行：

```
$ bin/hadoop dfs -cat out/*
```

结果如图 6-16 所示。

```
grid@ubuntunamenode:~/hadoop-0.20.2$ bin/hadoop jar hadoop-0.20.2-examples.jar wordcount in out
11/04/17 06:27:09 INFO input.FileInputFormat: Total input paths to process : 2
11/04/17 06:27:10 INFO mapred.JobClient: Running job: job_201104170613_0001
11/04/17 06:27:11 INFO mapred.JobClient: map 0% reduce 0%
11/04/17 06:27:23 INFO mapred.JobClient: map 50% reduce 0%
11/04/17 06:27:32 INFO mapred.JobClient: map 50% reduce 16%
11/04/17 06:27:33 INFO mapred.JobClient: map 100% reduce 16%
11/04/17 06:27:47 INFO mapred.JobClient: map 100% reduce 100%
11/04/17 06:27:49 INFO mapred.JobClient: Job complete: job_201104170613_0001
11/04/17 06:27:49 INFO mapred.JobClient: Counters: 10
11/04/17 06:27:49 INFO mapred.JobClient: Job Counters
11/04/17 06:27:49 INFO mapred.JobClient: Launched reduce tasks=1
11/04/17 06:27:49 INFO mapred.JobClient: Rack-local map tasks=1
11/04/17 06:27:49 INFO mapred.JobClient: Launched map tasks=2
11/04/17 06:27:49 INFO mapred.JobClient: Data-local map tasks=1
11/04/17 06:27:49 INFO mapred.JobClient: FileSystemCounters
11/04/17 06:27:49 INFO mapred.JobClient: FILE_BYTES_READ=55
11/04/17 06:27:49 INFO mapred.JobClient: HDFS_BYTES_READ=25
11/04/17 06:27:49 INFO mapred.JobClient: FILE_BYTES_WRITTEN=186
11/04/17 06:27:49 INFO mapred.JobClient: HDFS_BYTES_WRITTEN=25
11/04/17 06:27:49 INFO mapred.JobClient: Map-Reduce Framework
11/04/17 06:27:50 INFO mapred.JobClient: Reduce input groups=3
11/04/17 06:27:50 INFO mapred.JobClient: Combine output records=4
11/04/17 06:27:50 INFO mapred.JobClient: Map input records=2
11/04/17 06:27:50 INFO mapred.JobClient: Reduce shuffle bytes=61
11/04/17 06:27:50 INFO mapred.JobClient: Reduce output records=3
11/04/17 06:27:50 INFO mapred.JobClient: Spilled Records=8
11/04/17 06:27:50 INFO mapred.JobClient: Map output bytes=41
11/04/17 06:27:50 INFO mapred.JobClient: Combine input records=4
11/04/17 06:27:50 INFO mapred.JobClient: Map output records=4
11/04/17 06:27:50 INFO mapred.JobClient: Reduce input records=4
grid@ubuntunamenode:~/hadoop-0.20.2$
```

图 6-15 运行 WordCount 实例

```
grid@ubuntunamenode:~/hadoop-0.20.2$ bin/hadoop dfs -cat out/*
hadoop 1
hello 2
world 1
```

图 6-16 查看数据处理结果

也可以把输出文件从 Hadoop 分布式文件系统复制到本地文件系统查看, 具体命令如下:

```
$ bin/hadoop dfs -get out output
```

```
$ cat output/*
```

(h) 停止 Hadoop 守护进程。使用下面的命令即可停止守护进程:

```
$ bin/stop-all.sh
```

运行结果如图 6-17 所示。

```
grid@ubuntunamenode:~/hadoop-0.20.2$ bin/stop-all.sh
stopping jobtracker
ubuntudata1: stopping tasktracker
ubuntudata2: stopping tasktracker
stopping namenode
ubuntudata1: stopping datanode
ubuntudata2: stopping datanode
ubuntunamenode: stopping secondarynamenode
grid@ubuntunamenode:~/hadoop-0.20.2$ _
```

图 6-17 停止 Hadoop 守护进程运行结果

6.5.2 在 Windows 系统中安装 Hadoop

1. 安装环境配置

1) 下载安装 Cygwin

在 Windows 系统中安装 Hadoop, 需要安装 Cygwin。Cygwin 是一个在 Windows 平台上运行的 Unix 模拟环境, 提供 shell 支持。下载 Cygwin 安装包时要选中 Net category 里的 openssh, 如图 6-18 所示。

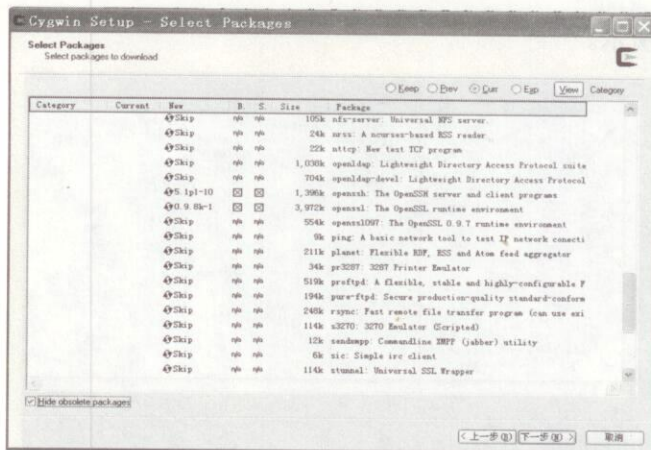


图 6-18 下载 Cygwin 安装包

编辑系统变量, 新建系统变量 CYGWIN, 变量值为 ntsec tty, 如图 6-19 所示。编辑系统变量里的 Path 变量, 加入 C:\cygwin\bin, 如图 6-20 所示。

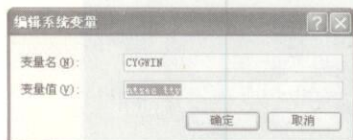


图 6-19 编辑系统变量 (1)

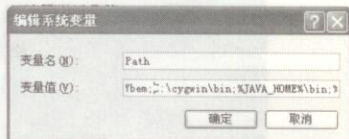


图 6-20 编辑系统变量 (2)

2) 安装 Java

实验使用 jdk-6u13-windows-i586-p.exe, 默认的安装目录为 C:\Program Files\Java\jdk1.6.0_13。

新建系统变量 JAVA_HOME, 如图 6-21 所示。

编辑系统变量里的 Path 变量, 加入 %JAVA_HOME%\bin;%JAVA_HOME%\jre\bin, 如图 6-22 所示。

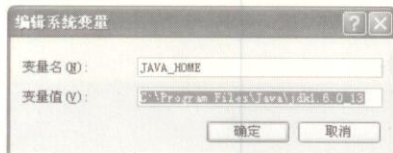


图 6-21 编辑系统变量 (3)

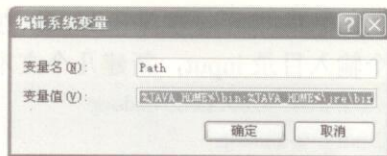


图 6-22 编辑系统变量 (4)

新建系统变量 CLASSPATH, 值为 .;%JAVA_HOME%\lib;%JAVA_HOME%\lib\tools.jar, 如图 6-23 所示。

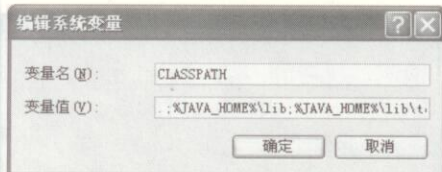


图 6-23 编辑系统变量 (5)

2. 安装步骤

与 Linux 系统一样, 在 Windows 系统上运行 Hadoop 集群也支持三种运行模式: 单机模式、伪分布式模式和完全分布式模式。

1) 单机模式

启动 Cygwin, 解压缩 Hadoop 安装包。通过 cygdrive (位于 Cygwin 根目录中) 可以直接映射到 Windows 下的各个逻辑磁盘分区。例如 Hadoop 安装包放在目录 D:\下, 则解压缩命令为 \$ tar -zxvf /cygdrive/d/hadoop-0.20.2.tar.gz, 解压后可使用 ls 命令查看, 如图 6-24 所示。

默认的解压缩目录为用户的根目录, 即 C:\cygwin\home\Administrator。

编辑 conf/hadoop-env.sh 文件, 将 JAVA_HOME 变量设置为 JAVA 安装的根目录。例如 JAVA 安装在目录 C:\Program Files\Java\jdk1.6.0_13, 如果路径中没有空格, 直接配置即可。但是因为本例中安装路径中存在空格, 所以需要把 Program Files 缩写成 Progra~1, 如图 6-25 所示。

```
Administrator@QJ8UETBZCQVE7FU /cygdrive/d/hadoop-0.20.2
$ ls
CHANGES.txt    conf            ivy
LICENSE.txt     contrib        ivy.xml
NOTICE.txt      docs           java.exe.stackdump
README.txt      hadoop-0.20.2-ant.jar  lib
bin             hadoop-0.20.2-core.jar  librecordio
build           hadoop-0.20.2-examples.jar  logs
build.xml       hadoop-0.20.2-test.jar    src
c++             hadoop-0.20.2-tools.jar   webapps
```

图 6-24 用户解压缩后的 hadoop 目录

```
# The java implementation to use. Required.
export JAVA_HOME=/cygdrive/c/Program Files/Java/jdk1.6.0_13
```

图 6-25 编辑 conf/hadoop-env.sh 文件

配置完之后即可运行 WordCount 实例。在 C:\cygwin\home\user\hadoop-0.20.2 目录下创建一个输入目录 input，新建几个文本文件，例如：

```
file1.txt: hello world hello hadoop
file2.txt: bye hadoop
```

然后运行实例，并将结果输出到指定的 output 目录：

```
$ bin/hadoop jar hadoop-0.20.2-examples.jar wordcount input output
```

需要注意的是执行之前 output 目录必须为空，或者不存在这个目录，否则会出错。

执行结果如图 6-26 所示。

```
$ cat output/*
bye      1
hadoop   2
hello    2
world    1
```

图 6-26 Word Count 实例运行结果

2) 伪分布式模式

(1) 编辑文件 conf/core-site.xml 和 mapred-site.xml。

编辑过程如图 6-27 和图 6-28 所示。

```
<property>
  <name>fs.default.name</name>
  <value>hdfs://localhost:8181</value>
  <description>The name of the default file system. A URI whose
  scheme and authority determine the FileSystem implementation. The
  uri's scheme determines the config property (fs.SCHEME.impl) naming
  the FileSystem implementation class. The uri's authority is used to
  determine the host, port, etc. for a filesystem.</description>
</property>
```

图 6-27 编辑文件 conf/core-site.xml


```
<property>
  <name>mapred.job.tracker</name>
  <value>localhost:9191</value>
  <description>The host and port that the MapReduce job tracker runs
    at. If "local", then jobs are run in-process as a single map
    and reduce task.
  </description>
</property>
```

图 6-28 编辑文件 conf/mapred-site.xml

(2) 安装配置 SSH。

启动 cygwin, 执行命令: \$ ssh-host-config, 如图 6-29 所示。

```
$ ssh-host-config
*** Info: Generating /etc/ssh_host_key
*** Info: Generating /etc/ssh_host_rsa_key
*** Info: Generating /etc/ssh_host_dsa_key
*** Info: Creating default /etc/ssh_config file
*** Info: Creating default /etc/sshd_config file
*** Info: Privilege separation is set to yes by default since OpenSSH 3.3.
*** Info: However, this requires a non-privileged account called 'sshd'.
*** Info: For more info on privilege separation read /usr/share/doc/openssh/REME.privsep.
*** Query: Should privilege separation be used? <yes/no>
```

图 6-29 安装配置 SSH (1)

当询问 “Should privilege separation be used?” 时, 输入 no, 如图 6-30 所示。

```
*** Info: Updating /etc/sshd_config file
*** Info: Added ssh to C:\WINDOWS\system32\drivers\services

*** Warning: The following functions require administrator privileges!
*** Query: Do you want to install sshd as a service?
*** Query: <Say "no" if it is already installed as a service> <yes/no>
```

图 6-30 安装配置 SSH (2)

当询问 “Do you want to install sshd as a service?” 时, 选择 yes, 把 sshd 作为一项服务安装, 如图 6-31 所示。

```
*** Info: Note that the CYGWIN variable must contain at least "ntsec"
*** Info: for sshd to be able to change user context without password.
*** Query: Enter the value of CYGWIN for the daemon: [ntsec]
```

图 6-31 安装配置 SSH (3)

当提示 “Enter the value of CYGWIN for the daemon: [ntsec]” 时, 选择 ntsec, 如图 6-32 所示。

```
*** Info: The sshd service has been installed under the LocalSystem
*** Info: account (also known as SYSTEM). To start the service now, call
*** Info: 'net start sshd' or 'cygrunsrv -S sshd'. Otherwise, it
*** Info: will start automatically after the next reboot.
*** Info: Host configuration finished. Have fun!
```

图 6-32 安装配置 SSH (4)

则提示 sshd 服务已经在本地系统安装好。

输入命令 \$ net start sshd, 启动 SSH, 如图 6-33 所示。



图 6-33 启动 SSH (1)

也可以通过服务启动 CYGWIN sshd，如图 6-34 所示。

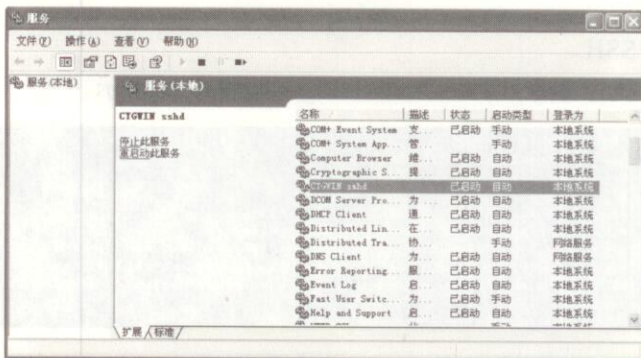


图 6-34 启动 SSH (2)

执行\$ ssh-keygen 来生成密钥对，然后一直按【Enter】键，就会按照默认选项将生成的密钥对保存在.ssh 目录下。将 RSA 公钥加入到公钥授权文件 authorized_keys 中，命令如下：

```
$ cd .ssh
$ cat id_rsa.pub >> authorized_keys
```

最后执行\$ ssh localhost，就可以实现无需密码的 SSH 连接。

(3) Hadoop 运行。

(a) 格式化分布式文件系统。使用如下的命令来格式化：

```
$ bin/hadoop NameNode -format
```

(b) 启动 Hadoop 守护进程。下面的命令用来启动守护进程：

```
$ bin/start-all.sh
```

(c) 运行 WordCount 实例。在本地文件系统上建立 input 目录，放入若干个文本文件，复制到 HDFS 的根目录下，重命名为 in，并运行：

```
$ bin/hadoop dfs -put input in
$ bin/hadoop jar hadoop-0.20.2-examples.jar wordcount in out
```

out 为数据处理完成后的输出目录，默认在 Hadoop 根目录下。同样的，在运行之前必须清空或者删除 out 目录，否则会报错。

任务执行完，用以下命令查看 Hadoop 分布式文件系统上数据处理的结果：

```
$ bin/hadoop dfs -cat out/*
```

也可以把输出文件从 Hadoop 分布式文件系统复制到本地文件系统查看：

```
$ bin/hadoop dfs -get out output
$ cat output/*
```


(d) 停止 Hadoop 守护进程。最后需要停止所有的 Hadoop 守护进程, 命令如下:

```
$ bin/stop-all.sh
```

3) 完全分布式模式

在完全分布式模式下运行 Hadoop 需要经历以下的步骤。

(1) 修改相应文件。

修改所有机器的 C:\WINDOWS\system32\drivers\etc\hosts 文件, 加入各机器 IP 地址及其对应的主机名, 即:

```
127.0.0.1                localhost
192.168.11.254           centos254
192.168.11.253           centos253
192.168.11.252           centos252
```

(2) 在所有机器上建立相同的账户 stony。

(3) 在所有机器上进行 SSH 配置。

执行 `$ ssh-keygen` 来生成密钥对。然后一直按【Enter】键, 就会按照默认选项生成密钥对, 并保存在 `~/.ssh/id_rsa` 文件中。执行下面的命令将 RSA 公钥加入到公钥授权文件 `authorized_keys` 中:

```
$ cd ~/.ssh
$ cat id_rsa.pub >> authorized_keys
```

接着在 centos254 上执行如下代码:

```
$ cd ~/.ssh
$ scp authorized_keys stony@centos253:/home/stony/.ssh
$ scp authorized_keys stony@centos252:/home/stony/.ssh
```

分别进入 centos253、centos252 和 centos251 的 `~/.ssh` 目录, 改变 `authorized_keys` 文件的许可权限, 命令如下:

```
$ chmod 644 authorized_keys
```

这样从 centos254 到其他机器的 SSH 连接不需要输入密码就可以实现。

(4) 在所有机器上配置 Hadoop。

首先在 centos254 上配置, 执行如下的解压缩命令和修改目录命令:

```
$ tar -zxvf /cygdrive/d/hadoop-0.20.2.tar.gz
$ mv hadoop-0.20.2 hadoop
```

编辑 `conf/core-site.xml`, 如图 6-35 所示。

```
<property>
  <name>fs.default.name</name>
  <value>hdfs://centos254:8181</value>
  <description>The name of the default file system. A URI whose
  scheme and authority determine the FileSystem implementation. The
  uri's scheme determines the config property (fs.SCHEME.impl) naming
  the FileSystem implementation class. The uri's authority is used to
  determine the host, port, etc. for a filesystem.</description>
</property>
```

图 6-35 core-site.xml

编辑 `conf/mapred-site.xml`, 如图 6-36 所示。

```
<property>
  <name>mapred.job.tracker</name>
  <value>centos254:9111</value>
  <description>The host and port that the MapReduce job tracker runs
    at. If "local", then jobs are run in-process as a single map
    and reduce task.
  </description>
</property>
```

图 6-36 mapred-site.xml

编辑 conf/hdfs-site.xml, 如图 6-37 所示。

```
<property>
  <name>dfs.replication</name>
  <value>3</value>
  <description>Default block replication.
    The actual number of replications can be specified when the
    file is created. The default is used if replication is not specified in the
    create call.
  </description>
</property>
```

图 6-37 hdfs-site.xml

编辑 conf/master, 修改为 master 的主机名, 每个 ip 一行, 在这里就是 192.168.11.254。编辑 conf/slaves, 加入所有 slaves 的主机名, 即 192.168.11.253 和 192.168.11.252。把 hadoop 复制到其他机器, 命令如下:

```
$scp ~/hadoop stony@centos253:/home/stony/
$scp ~/hadoop stony@centos252:/home/stony/
```

编辑所有机器的 conf/hadoop-env.sh 文件, 将 JAVA_HOME 变量设置为 JAVA 安装的根目录, 不同机器可以使用不同的 JAVA 版本, 但需要 jdk1.5 版本以上。

(5) Hadoop 运行。

格式化分布式文件系统。在 centos254 上执行如下的命令:

```
$bin/hadoop namenode -format
```

启动 Hadoop 守护进程。下面的命令用来启动守护进程:

```
$ bin/start-all.sh
```

同样的, 访问 <http://centos254:50070> 可以查看 NameNode 以及整个分布式文件系统的状态, 浏览分布式文件系统中的文件以及日志等; 访问 <http://centos254:50060> 可以查看 TaskTracker 的运行状态。

运行 WordCount 实例。将本地文件系统上 input 目录复制到 HDFS 的根目录下, 重命名为 in, 执行:

```
$ bin/hadoop dfs -put input in
$ bin/hadoop jar hadoop-0.20.2-examples.jar wordcount in out
```

out 为数据处理完成后的输出目录, 在分布式文件系统用户的根目录下。

任务执行完, 用下列命令查看 Hadoop 分布式文件系统上数据处理的结果:

```
$ bin/hadoop dfs -cat out/*
```

也可以把输出文件从 Hadoop 分布式文件系统复制到本地文件系统查看:

```
$ bin/hadoop dfs -get out output
```



```
$ cat output/*
```

停止 Hadoop 守护进程。最后要停止所有的 Hadoop 守护进程:

```
$ bin/stop-all.sh
```

4) 注意事项

虽然 Hadoop 支持 Windows 平台,但其官方网站声明,Hadoop 的分布式操作尚未在 Windows 平台上严格测试,建议只作为开发平台。

格式化 Hadoop 文件系统是启动 Hadoop 的第一步,不过一旦安装 Hadoop 后,不要格式化运行中的文件系统,否则所有的数据会被清除。如果需要执行格式化,每次格式化(format)前,清空 tmp 目录下的所有文件。因为 Hadoop 格式化时会重新创建 NameNodeID,而 tmp 中还包含上次格式化留下的信息,格式化虽然清空了 NameNode 的数据,但是保留了 DataNode 的数据,这样会导致启动失败。正确的步骤如下:

- (1) 用 bin/stop-all.sh 停止所有的守护进程。
- (2) 删除 \$HADOOP_HOME/tmp 这个文件夹。
- (3) 重新格式化 NameNode。
- (4) 重新启动守护进程。

在 {HADOOP_HOME}/logs 目录下,NameNode、DataNode、Secondary NameNode、JobTracker、TaskTracker 各有一个对应的日志文件,每一次运行的计算任务也有对应的日志文件。当出现故障时,分析这些日志文件有助于找到故障原因。例如可通过 <http://192.168.11.254:50070/logs/> 进行浏览日志信息,如图 6-38 所示。

Directory: /logs/

hadoop-stony-jobtracker-centos254.log	1463 bytes	2011-4-16 16:00:41
hadoop-stony-jobtracker-centos254.out	0 bytes	2011-4-16 16:00:40
hadoop-stony-namenode-centos254.log	4800 bytes	2011-4-16 16:00:55
hadoop-stony-namenode-centos254.out	0 bytes	2011-4-16 16:00:35
hadoop-stony-secondarynamenode-centos254.log	1296 bytes	2011-4-16 16:00:55
hadoop-stony-secondarynamenode-centos254.out	0 bytes	2011-4-16 16:00:39
history/	4096 bytes	2011-4-16 16:00:41

图 6-38 日志信息

6.6 HDFS 使用

6.6.1 HDFS 常用命令

HDFS 是 Hadoop 的重要组成部分,也可以作为独立的分布式文件系统使用。HDFS 集群由一个 NameNode 和多个 DataNode 组成:NameNode 负责管理文件系统的元数据;DataNode 负责存储实际的数据。不少于三台 PC 组成的 Hadoop 集群。其中 censtos254 为 namenode,其 hosts 文件配置如图 6-39 所示。

```
[root@centos254 ~]# strings /etc/hosts
# Do not remove the following line, or various programs
# that require network functionality will fail.
127.0.0.1          localhost.localdomain localhost
::1               localhost6.localdomain6 localhost6
192.168.11.254    centos254
192.168.11.253    centos253
192.168.11.252    centos252
```

图 6-39 hosts 文件配置

1. HDFS 管理

DFSAdmin 命令支持一些和 HDFS 管理相关的操作。例如 \$ bin/hadoop dfsadmin -report 报告 HDFS 的基本统计信息, 如图 6-40 所示。

命令 \$ bin/hadoop dfsadmin -help 能列出当前支持的所有命令。

2. 安全模式

NameNode 在启动时会自动进入安全模式。安全模式是 NameNode 的一种状态, 在这个阶段, 文件系统不允许有任何修改。安全模式的目的是在系统启动时检查各个 DataNode 上数据块的有效性, 同时根据策略对数据块进行必要的复制或删除, 当数据块最小百分比数满足配置的最小副本数条件时, 会自动退出安全模式。

实践过程中, 可以通过查看 namenode 的日志如图 6-41 所示。

```
[stony@centos254 bin]$ ./hadoop dfsadmin -report
Configured Capacity: 20317372416 (18.92 GB)
Present Capacity: 4131975168 (3.85 GB)
DFS Remaining: 4131819520 (3.85 GB)
DFS Used: 155648 (152 KB)
DFS Used%: 0%
Under replicated blocks: 1
Blocks with corrupt replicas: 0
Missing blocks: 0

-----
Datanodes available: 2 (2 total, 0 dead)

Name: 192.168.11.253:50110
Decommission Status : Normal
Configured Capacity: 10158686208 (9.46 GB)
DFS Used: 77824 (76 KB)
Non DFS Used: 8018829312 (7.47 GB)
DFS Remaining: 2139779072 (1.99 GB)
DFS Used%: 0%
DFS Remaining%: 21.06%
Last contact: Sat Apr 16 14:50:00 CST 2011

Name: 192.168.11.252:59999
Decommission Status : Normal
Configured Capacity: 10158686208 (9.46 GB)
DFS Used: 77824 (76 KB)
Non DFS Used: 8166567936 (7.61 GB)
DFS Remaining: 1992040448 (1.86 GB)
DFS Used%: 0%
DFS Remaining%: 19.61%
Last contact: Sat Apr 16 14:50:02 CST 2011
```

图 6-40 HDFS 基本统计信息


```

the ratio of reported blocks 0.0000 has not reached the threshold 0.9990. Safe mode will be turned off automatically.
at org.apache.hadoop.hdfs.server.namenode.FSNamesystem.deleteInternal(FSNamesystem.java:1700)
at org.apache.hadoop.hdfs.server.namenode.FSNamesystem.delete(FSNamesystem.java:1680)
at org.apache.hadoop.hdfs.server.namenode.NameNode.delete(NameNode.java:517)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:597)
at org.apache.hadoop.ipc.RPC$Server.call(RPC.java:508)
at org.apache.hadoop.ipc.Server$Handler$1.run(Server.java:959)
at org.apache.hadoop.ipc.Server$Handler$1.run(Server.java:955)
at java.security.AccessController.doPrivileged(Native Method)
at javax.security.auth.Subject.doAs(Subject.java:396)
at org.apache.hadoop.ipc.Server$Handler.run(Server.java:953)

```

图 6-41 安全模式提示

Name node is in safe mode, 说明系统正处于安全模式, 这时只需要等待即可, 也可以通过下面的命令关闭安全模式:

```
$ bin/hadoop dfsadmin -safemode leave
```

结果如图 6-42 所示。

```

[stony@centos254 hadoop]$ bin/hadoop dfsadmin -safemode leave
Safe mode is OFF

```

图 6-42 关闭安全模式

在必要情况下, 可以通过以下命令把 HDFS 置于安全模式:

```
$ bin/hadoop dfsadmin -safemode enter
```

结果如图 6-43 所示。

```

[stony@centos254 hadoop]$ bin/hadoop dfsadmin -safemode enter
Safe mode is ON

```

图 6-43 设置 HDFS 安全模式

3. 添加节点

可扩展性是 HDFS 的一个重要特性, 向 HDFS 集群中添加节点是很容易实现的。添加一个新的 DataNode 节点, 首先在新加节点上安装好 Hadoop, 要和 NameNode 使用相同的配置 (可以直接从 NameNode 复制), 修改 \$HADOOP_HOME/conf/master 文件, 加入 NameNode 主机名。然后在 NameNode 节点上修改 \$HADOOP_HOME/conf/slaves 文件, 加入新加节点主机名, 再建立到新加节点无密码的 SSH 连接, 运行启动命令:

```
$ bin/start-all.sh
```

通过地址 <http://192.168.11.254:50070> 可以看到新的 DataNode 节点 centos251 添加成功, 如图 6-44 所示。

Live Datanodes : 3

Node	Last Contact	Admin State	Configured Capacity (GB)	Used (GB)	Non DFS Used (GB)	Remaining (GB)	Used (%)	Used (%)	Remaining (%)	Blocks
centos251 50166	3	In Service	9.46	0	7.6	1.86	0		19.62	2
centos252 59999	3	In Service	9.46	0	7.61	1.86	0		19.61	2
centos253 50110	0	In Service	9.46	0	7.47	1.99	0		21.06	2

图 6-44 添加 DataNode 节点 centos251

实验中复制因子为 3, 部分数据块未达到最小副本数。等待一段时间之后, 数据块自动进行了必要的复制。数据分布情况如图 6-45 所示。

Live Datanodes : 3

Node	Last Contact	Admin State	Configured Capacity (GB)	Used (GB)	Non DFS Used (GB)	Remaining (GB)	Used (%)	Used (%)	Remaining (%)	Blocks
centos251.50166	3	In Service	9.46	0	7.6	1.86	0		19.62	3
centos252.59999	3	In Service	9.46	0	7.61	1.86	0		19.61	3
centos253.50110	3	In Service	9.46	0	7.47	1.99	0		21.06	3

图 6-45 数据分布情况

4. 节点故障

HDFS 运行在普通的硬件上，出现故障很正常，这就对 HDFS 的容错能力有很高的要求，故障情况下分布式文件系统仍能正常使用显得尤为重要。

DataNode 会定期向 NameNode 发送心跳信号，当由于某些原因（如网络故障）导致 DataNode 发出的心跳信号没有被 NameNode 正常收到，NameNode 就会认为该 DataNode 上的数据无效，它会检测出哪些数据块的副本数小于设定值，自动复制新的副本并分发到其他 DataNode 节点。所以一定数目的 DataNode 节点出现故障，不会影响 HDFS 的正常使用。

实验中关掉任意一个 DataNode 节点，不会影响 HDFS 上文件的正常访问。

5. 负载均衡

HDFS 的数据在各个 DataNode 中的分布可能很不均匀，尤其是在 DataNode 节点出现故障或新增 DataNode 节点时。新增数据块时 NameNode 对 DataNode 节点的选择策略也有可能导致数据块分布的不均匀。用户可以使用命令重新平衡 DataNode 上的数据块的分布：

```
$ bin/start-balancer.sh
```

结果如图 6-46 所示。

```
[stony@centos254 bin]$ ./start-balancer.sh
starting balancer, logging to /home/stony/hadoop/bin/../logs/hadoop-stony-balancer-centos254.out
Time Stamp      Iteration#  Bytes Already Moved  Bytes Left To Move  Bytes Being Moved
The cluster is balanced. Exiting...
Balancing took 467.0 milliseconds
```

图 6-46 HDFS 数据块负载均衡

命令执行前，DataNode 节点上数据的分布情况如图 6-47 所示。

Live Datanodes : 3

Node	Last Contact	Admin State	Configured Capacity (GB)	Used (GB)	Non DFS Used (GB)	Remaining (GB)	Used (%)	Used (%)	Remaining (%)	Blocks
centos251.50166	3	In Service	9.46	0	7.6	1.86	0		19.62	2
centos252.59999	3	In Service	9.46	0	7.61	1.86	0		19.61	2
centos253.50110	0	In Service	9.46	0	7.47	1.99	0		21.06	2

图 6-47 负载均衡前的数据分布情况

负载均衡完毕后，DataNode 节点上数据的分布情况如图 6-48 所示。

Live Datanodes : 3

Node	Last Contact	Admin State	Configured Capacity (GB)	Used (GB)	Non DFS Used (GB)	Remaining (GB)	Used (%)	Used (%)	Remaining (%)	Blocks
centos251.50166	3	In Service	9.46	0.2	7.61	1.66	2.08		17.52	13
centos252.59999	2	In Service	9.46	0.2	7.61	1.66	2.08		17.52	10
centos253.50110	2	In Service	9.46	0.2	7.47	1.8	2.08		18.98	13

图 6-48 负载均衡后的数据分布情况

6.6.2 HDFS 基准测试

TestDFSIO 用来测试 HDFS 的 I/O 性能, 通过 MapReduce 并行完成作业, 一个 Map 负责一个读写任务, 由 Reduce 统计结果测试命令:

```
bin/hadoop jar $HADOOP_HOME/hadoop-0.20.2-test.jar TestDFSIO -write -nrFile 20 -filesize 200
```

基准测试信息如图 6-49 所示。

```
[stony@centos254 ~]$ bin/hadoop jar hadoop-0.20.2-test.jar TestDFSIO -write -nrfile 20 -filesize 200
TestDFSIO.0.0.4
11/04/16 17:23:41 INFO mapred.FileInputFormat: nrFiles = 1
11/04/16 17:23:41 INFO mapred.FileInputFormat: fileSize (MB) = 200
11/04/16 17:23:41 INFO mapred.FileInputFormat: bufferSize = 1000000
11/04/16 17:23:41 INFO mapred.FileInputFormat: creating control file: 200 mega bytes, 1 files
11/04/16 17:23:42 INFO mapred.FileInputFormat: created control files for: 1 files
11/04/16 17:26:22 INFO mapred.FileInputFormat: ----- TestDFSIO ----- : write
11/04/16 17:26:22 INFO mapred.FileInputFormat: Date & time: Sat Apr 16 17:26:22 CST 2011
11/04/16 17:26:22 INFO mapred.FileInputFormat: Number of files: 1
11/04/16 17:26:22 INFO mapred.FileInputFormat: Total MBytes processed: 200
11/04/16 17:26:22 INFO mapred.FileInputFormat: Throughput mb/sec: 1.5823160359818667
11/04/16 17:26:22 INFO mapred.FileInputFormat: Average IO rate mb/sec: 1.5823160409927368
11/04/16 17:26:22 INFO mapred.FileInputFormat: IO rate std deviation: 2.1124071110718262E-4
11/04/16 17:26:22 INFO mapred.FileInputFormat: Test exec time sec: 159.792
```

图 6-49 基准测试信息

如果 eclipse 已经安装了 hadoop-0.20.2-eclipse-plugin.jar 插件包, 那么可以在 eclipse 下查看结果, 如图 6-50 所示。

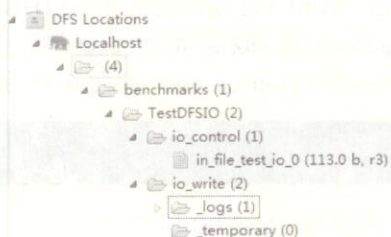


图 6-50 eclipse 下的 dfs

运行后, 结果被写入控制台并记录到本地文件 TestDFSIO_results.log, 文件默认写在 benchmarks/TestDFSIO 目录下 (可以通过设置 test.build.data 系统属性来改变)。在对 HDFS 进行基准测试后, 要通过参数 -clean 从 HDFS 上删除所有生成的文件。

```
% bin/hadoop jar $HADOOP_HOME/hadoop-0.20.2-test.jar TestDFSIO -clean
```

6.7 HBase 安装使用

6.7.1 HBase 的安装配置

HBase 的安装配置步骤如下。

(1) 安装 HBase。

首先在 Hadoop master, 即 ubuntu1 上安装, 把 hbase-0.19.2. tar.gz 解压缩, 目录为用户根目录, 使用如下命令:

```
$ tar -zxvf hbase-0.19.2.tar.gz
```

(2) 编辑 {HBASE_HOME}/conf/hbase-env.sh, 把 JAVA_HOME 变量设置为 JAVA 安装的根目录, 把 HBASE_CLASSPATH 设置为 HADOOP_CONF_DIR, 如图 6-51 所示。

```
# Set environment variables here.

# The java implementation to use. Java 1.6 required.
export JAVA_HOME=/usr/java/jdk1.6.0_12

# Extra Java CLASSPATH elements. Optional.
export HBASE_CLASSPATH=/home/grid/hadoop-0.19.1/conf_
```

图 6-51 配置 HBase (1)

(3) 编辑`{HBASE_HOME}/conf/hbase-site.xml`, 如图 6-52 所示。其中, `hbase.rootdir` 的端口设置要与 Hadoop 保持一致。

```
<property>
  <name>hbase.master</name>
  <value>ubuntu1:60000</value>
</property>

<property>
  <name>hbase.rootdir</name>
  <value>hdfs://ubuntu1:9000/hbase</value>
</property>
```

图 6-52 配置 HBase (2)

(4) 复制`${HADOOP_HOME}/conf/hadoop-site.xml`

用命令复制`${HADOOP_HOME}/conf/hadoop-site.xml` 到`{HBASE_HOME}/conf` 目录:

```
$ cp ~/hadoop-0.19.1/conf/hadoop-site.xml ~/hbase-0.19.2/conf
```

(5) 编辑`{HBASE_HOME}/conf/regionservers`, 如图 6-53 所示。

```
ubuntu1
ubuntu2
ubuntu3
```

图 6-53 配置 HBase (3)

(6) 把 HBase 复制到其他机器上, 使用的命令如下:

```
$ scp -r ~/hbase-0.19.2/ ubuntu2: /home/grid/
```

```
$ scp -r ~/hbase-0.19.2/ ubuntu3: /home/grid/
```

(7) 复制`{HBASE_HOME}/hbase-0.19.2.jar`。将`{HBASE_HOME}/hbase-0.19.2.jar` 复制到所有节点的`${HADOOP_HOME}/lib` 目录, 使用的命令如下:

```
$ cp ~/hbase-0.19.2/hbase-0.19.2.jar ~/hadoop-0.19.1/lib/
```

如果其他机器使用不同版本的 JDK, 编辑`{HBASE_HOME}/conf/hbase-env.sh` 文件, 将 `JAVA_HOME` 变量改为正确配置。

6.7.2 HBase 的执行

执行 HBase 的步骤如下。

(1) 启动 Hadoop。命令如下:

```
$ cd ~/hadoop-0.19.1
```

```
$ bin/start-all.sh
```

(2) 启动 Hbase。使用下面的命令启动 Hbase:

```
$ cd ~/hbase-0.19.2/
```

```
$ bin/start-hbase.sh
```

结果如图 6-54 所示。


```

grid@ubuntu1:~$ cd hbase-0.19.2/
grid@ubuntu1:~/hbase-0.19.2$ bin/start-hbase.sh
Starting master, logging to /home/grid/hbase-0.19.2/bin/../logs/hbase-grid-maste
r-ubuntu1.out
ubuntu2: starting regionserver, logging to /home/grid/hbase-0.19.2/bin/../logs/h
base-grid-regionserver-ubuntu2.out
ubuntu3: starting regionserver, logging to /home/grid/hbase-0.19.2/bin/../logs/h
base-grid-regionserver-ubuntu3.out
ubuntu1: starting regionserver, logging to /home/grid/hbase-0.19.2/bin/../logs/h
base-grid-regionserver-ubuntu1.out

```

图 6-54 启动 HBase

这时通过地址 <http://ubuntu1:50070> 可以看到, 在 HDFS 上自动生成的了/hbase 目录, 如图 6-55 所示。

Goto:

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
-ROOT-	dir				2009-06-23 21:27	rw-r--r--	grid	supergroup
META_	dir				2009-06-23 21:27	rw-r--r--	grid	supergroup
hbase.version	file	0 KB	3	64 MB	2009-06-23 21:26	rw-r--r--	grid	supergroup
log_192.168.1.11_1245763600476_60020	dir				2009-06-23 21:27	rw-r--r--	grid	supergroup
log_192.168.1.12_1245763552476_60020	dir				2009-06-23 21:27	rw-r--r--	grid	supergroup
log_192.168.1.13_1245763594106_60020	dir				2009-06-23 21:27	rw-r--r--	grid	supergroup

图 6-55 HBase 运行状态

(3) 进入 shell 模式进行 HBase 数据操作。执行以下命令进入 shell 模式:

```
$ bin/hbase shell
```

结果如图 6-56 所示。

```

grid@ubuntu1:~/hbase-0.19.2$ bin/hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Version: 0.19.2, r771918, Tue May 5 16:08:54 PDT 2009
hbase(main):001:0> _

```

图 6-56 HBase 数据操作

输入 help 会显示有关的帮助信息。可以对 Hbase 进行简单测试, 如图 6-57 所示。

```

hbase(main):001:0> create 't1','f1','f2','f3'
0 row(s) in 7.4565 seconds
hbase(main):002:0> list
t1
1 row(s) in 0.3128 seconds

```

图 6-57 HBase 的测试

(4) 停止 HBase。停用 HBase 的命令如下:

```
$ bin/stop-hbase.sh
```

(5) 停止 Hadoop。最后需要用下面的命令停止 Hadoop:

```
$ cd ~/hadoop-0.19.1
```

```
$ bin/stop-all.sh
```

6.7.3 Hbase 编程实例

启动 Eclipse, 新建 Map/Reduce Project, 命名为 Proj, 在此工程中新建类 test, 加入代码, 如图 6-58 所示。

```

static HBaseConfiguration conf = new HBaseConfiguration();
private static String tablename;
public static void main(String args[]) throws Exception {
    // 添加路径，连接到指定的HDFS及Hbase
    conf.addResource(new Path("conf/hadoop-site.xml"));
    conf.addResource(new Path("conf/hbase-default.xml"));
    conf.addResource(new Path("conf/hbase-site.xml"));
    tablename = "stu";
    test.createTable(tablename);
    // test.deleteTable(tablename);
}
//*****创建表 *****
public static void createTable(String tablename) throws IOException {
    HBaseAdmin admin = new HBaseAdmin(conf);
    HTableDescriptor tableDesc = new HTableDescriptor(tablename);
    tableDesc.addFamily(new HColumnDescriptor("ID:"));
    tableDesc.addFamily(new HColumnDescriptor("NAME:"));
    tableDesc.addFamily(new HColumnDescriptor("AGE:"));
    admin.createTable(tableDesc);
    System.out.println("Table "+tablename+" create!!!");
}
//*****删除表*****
public static void deleteTable(String tablename) throws IOException {
    HBaseAdmin admin = new HBaseAdmin(conf);
    if (admin.tableExists(tablename)) {
        admin.disableTable(tablename);
        admin.deleteTable(tablename);
        System.out.println("Table "+tablename+" delete!!!");
    }else{
        System.out.println("Table "+tablename+" not exist!!");
    }
}
}

```

图 6-58 HBase 表的创建和删除程序代码

在 Run Configurations 选项中选择工作路径为 HBASE_HOME，如图 6-59 所示。

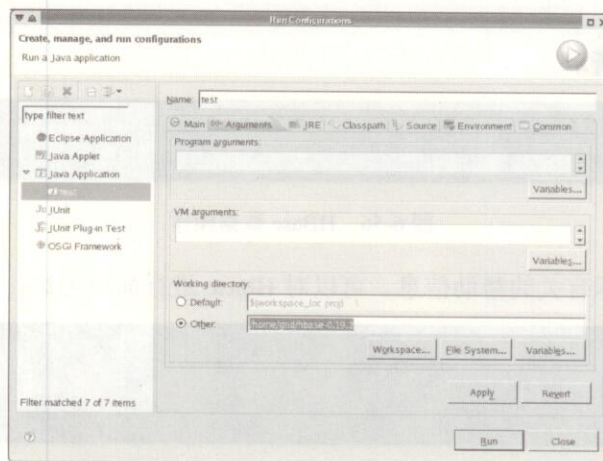


图 6-59 工作路径选择

运行后，可以从控制台看到运行结果，如图 6-60 所示。

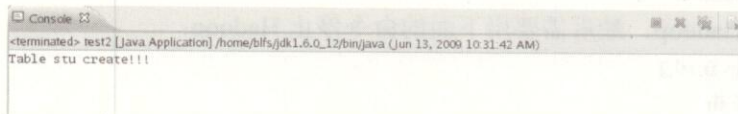


图 6-60 Hbase 表的创建实例运行结果（1）

通过地址 <http://ubuntul:60010> 可以看到 HBase 运行的相关信息，表“stu”添加成功，如图 6-61 所示。

Master Attributes

Attribute Name	Value	Description
HBase Version	0.19.2, r771918	HBase version and svn revision
HBase Compiled	Tue May 5 16:08:54 PDT 2009, stack	When HBase version was compiled and by whom
Hadoop Version	0.19.1, r745977	Hadoop version and svn revision
Hadoop Compiled	Fri Feb 20 00:16:34 UTC 2009, ndaley	When Hadoop version was compiled and by whom
HBase Root Directory	hdfs://ubuntu1:9000/hbase	Location of HBase home directory
Load average	2.0	Average load across all region servers. Naive computation.
Regions On FS	5	The Number of regions on FileSystem. Rough count

Catalog Tables

Table	Description
<u>-ROOT-</u>	The -ROOT- table holds references to all META regions
<u>META</u>	The META table holds references to all User Table regions

图 6-61 HBase 表的创建实例运行结果 (2)

6.8 MapReduce 编程

6.8.1 矩阵相乘算法设计

本节使用一个矩阵相乘的例子简单说明利用 Hadoop 的 MapReduce 编程框架的程序设计过程。

1. MapReduce 程序设计过程

程序设计过程大致包括以下 5 个步骤。

1) <key,value>对

<key,value>对是 MapReduce 编程框架中基本的数据单元,其中 key 实现了 WritableComparable 接口,value 实现了 Writable 接口,这使得框架可以对其序列化并可以对 key 执行排序。

2) 数据输入

InputFormat、InputSplit、RecordReader 是数据输入的主要编程接口。InputFormat 主要实现的功能是将输入数据分切成多个块,每个块都是 InputSplit 类型;而 RecordReader 负责将每个 InputSplit 块分解成多个<key₁,value₁>对传送给 Map 任务。

3) Mapper 阶段

此阶段设计的编程接口主要有 Mapper、Reducer、Partitioner。实现 Mapper 接口主要是实现其 Map 方法,Map 主要用来处理输入<key₁,value₁>对并产生输出<key₂,value₂>对。在 Map 处理过<key₁,value₁>对之后,可以实现一个 Combiner 类对 Map 的输出进行初步的规约操作,此类实现了 Reducer 接口。而 Partitioner 主要是根据 Map 的输出<key₂,value₂>对的值,将其分发给不同 Reduce 任务。

4) Reducer 阶段

此阶段需要实现 Reducer 接口, 主要是实现 Reduce 方法, 框架将 Map 输出的中间结果根据相同的 key_2 组合成 $\langle key_2, list(value_2) \rangle$ 对作为 Reduce 方法的输入数据并对其进行处理, 同时产生输出数据 $\langle key_3, value_3 \rangle$ 对。

5) 数据输出

数据输出阶段主要实现两个编程接口, 其中 FileOutputFormat 接口用来将数据输出到文件, RecordWriter 接口负责输出一个 $\langle key, value \rangle$ 对。

2. 矩阵相乘

一般来说, 矩阵相乘就是左矩阵乘右矩阵结果为积矩阵, 左矩阵的列数与右矩阵的行数相等, 设左矩阵为 $a \times b$ 的矩阵, 右矩阵为 $b \times c$ 的矩阵, 左矩阵的行与右矩阵的列对应元素乘积之和为积矩阵中的元素值。本例中的矩阵相乘也是这种传统算法, 左矩阵的一行和右矩阵的一列组成一个 InputSplit, 其存储 b 个 $\langle key, value \rangle$ 对, key 存储积矩阵元素位置, value 为生成一个积矩阵元素的 b 个数据对中的一个; Map 方法计算一个 $\langle key, value \rangle$ 对的 value 中数据对的积; 而 Reduce 方法计算 key 值相同的所有积的和。本例中的矩阵为整数矩阵。

6.8.2 编程实现

1. 程序中的类

- (1) matrix 类用于存储矩阵。
- (2) IntPair 类实现 WritableComparable 接口用于存储整数对的。
- (3) matrixInputSplit 类继承了 InputSplit 接口, 每个 matrixInputSplit 包括 b 个 $\langle key, value \rangle$ 对, 用来生成一个积矩阵元素。key 和 value 都为 IntPair 类型, key 存储的是积矩阵元素的位置, value 为计算生成一个积矩阵元素的 b 个数据对中的一个。
- (4) 继承 InputFormat 的 matrixInputFormat 类, 用来数据输入。
- (5) matrixRecordReader 类继承了 RecordReader 接口, MapReduce 框架调用此类生成 $\langle key, value \rangle$ 对赋给 map 方法
- (6) 主类 matrixMulti, 其内置类 MatrixMapper 继承了 Mapper 重写覆盖了 Map 方法, 类似地, FirstPartitioner、MatrixReducer 也是如此。在 main 函数中, 需要设置一系列的类, 详细内容参考源码。
- (7) MultipleOutputFormat 类用于向文件输出结果。
- (8) LineRecordWriter 类被 MultipleOutputFormat 中的方法调用, 向文件输出一个结果 $\langle key, value \rangle$ 对。

2. 部分代码片段

matrixInputFormat:

```
public class matrixInputFormat extends InputFormat<IntPair, IntPair> {
    public matrix[] m=new matrix[2]; //新建两个 matrix 实例, m[0]为左矩阵, m[1] 为右矩阵
    public List<InputSplit> getSplits(JobContext context) throws IOException, InterruptedException {
        //从文件里读取矩阵填充 m[0]、 m[1], 文件在 HDFS 中
    }
}
```



```

int NumOffFiles = readFile(context);

for(int n=0;n<row;n++){// row 为 m[0]的行数
for(int m=0;m<col;m++){// col 为 m[1]的列数
    // 以 m[0]的第 n 行与 m[1]的第 m 列为参数实例化一个 matrixInputSplit
    matrixInputSplit split = new matrixInputSplit(n,this.m[0],m,this.m[1]);
    splits.add(split);
}
}
return splits;
}

```

matrixMulti:

```

public class matrixMulti {

    public static class MatrixMapper extends Mapper<IntPair, IntPair, IntPair, IntWritable> {
        public void map(IntPair key, IntPair value, Context context) throws IOException,
            InterruptedException {
            int left=value.getLeft();
            int right=value.getRight();
            intWritable result=new IntWritable(left*right);
            context.write(key, result);
        }
    }

    public static class FirstPartitioner extends Partitioner<IntPair,IntWritable>{
    public int getPartition(IntPair key, IntWritable value, int numPartitions) {
        //按 key 的左值既行号分配<k,v>对到对应的 Reduce 任务, numPartitions 为
        //Reduce 任务的个数
        int abs=Math.abs(key.getLeft()) % numPartitions;
        return abs;
    }
    }

    public static class MatrixReducer extends Reducer<IntPair, IntWritable, IntPair, IntWritable> {
        private IntWritable result = new IntWritable();
        public void reduce(IntPair key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values){
                int v=val.get();
                sum += v; }//对 key 值相同的 val 求和
            result.set(sum);
            context.write(key, result);
        }
    }
}

```

3. 程序的运行过程

(1) 程序从文件中读出数据到内存，生成 matrix 实例，通过组合左矩阵的行与右矩阵的列生成 $a \times c$ 个 matrixInputSplit。

(2) 一个 Mapper 任务对一个 matrixInputSplit 中的每个 $\langle \text{key}_1, \text{value}_1 \rangle$ 对调用一次 Map 方法对 value_1 中的两个整数相乘。输入的 $\langle \text{key}_1, \text{value}_1 \rangle$ 对中 key_1 和 value_1 的类型均为 IntPair，其输出为 $\langle \text{key}_1, \text{value}_2 \rangle$ 对， key_1 不变， value_2 为 IntWritable 类型，值为 value_1 中的两个整数的乘积。

(3) MapReduce 框架调用 FirstPartitioner 类的 getPartition 方法将 Map 的输出 $\langle \text{key}_1, \text{value}_2 \rangle$ 对分配给指定的 Reducer 任务（任务个数可以在配置文件中设置）。

(4) Reducer 任务对 key_1 值相同的所有 value_2 求和，得出积矩阵中的元素 k 的值。其输入为 $\langle \text{key}_1, \text{list}(\text{value}_2) \rangle$ 对，输出为 $\langle \text{key}_1, \text{value}_3 \rangle$ 对， key_1 不变， value_3 为 IntWritable 类型，值为 key_1 值相同的所有 value_2 的和。

(5) MapReduce 框架实例化一个 MultipleOutputFormat 类，将结果输出到文件。

4. 程序执行过程

程序需要两个参数：输入目录和输出目录，如图 6-62 首行的 input、output。

```
[w1826214@avatarnode0 hadoop-0.20.2]$ bin/hadoop jar matrix2.jar matrixMulti input output
matrix
[0][0]:1 [0][1]:2 [0][2]:0
[1][0]:2 [1][1]:1 [1][2]:1
matrix
[0][0]:3 [0][1]:5
[1][0]:4 [1][1]:2
[2][0]:1 [2][1]:1
11/04/12 12:57:45 INFO mapred.JobClient: Running job: job_201104121011_0016
11/04/12 12:57:46 INFO mapred.JobClient: map 0% reduce 0%
11/04/12 12:57:54 INFO mapred.JobClient: map 50% reduce 0%
11/04/12 12:57:57 INFO mapred.JobClient: map 100% reduce 0%
11/04/12 12:58:06 INFO mapred.JobClient: map 100% reduce 100%
11/04/12 12:58:08 INFO mapred.JobClient: Job complete: job_201104121011_0016
11/04/12 12:58:08 INFO mapred.JobClient: Counters: 16
11/04/12 12:58:08 INFO mapred.JobClient:   Job Counters
11/04/12 12:58:08 INFO mapred.JobClient:     Launched reduce tasks=1
11/04/12 12:58:08 INFO mapred.JobClient:     Rack-local map tasks=4
11/04/12 12:58:08 INFO mapred.JobClient:     Launched map tasks=4
11/04/12 12:58:08 INFO mapred.JobClient:   FileSystemCounters
11/04/12 12:58:08 INFO mapred.JobClient:     FILE_BYTES_READ=62
11/04/12 12:58:08 INFO mapred.JobClient:     FILE_BYTES_WRITTEN=270
11/04/12 12:58:08 INFO mapred.JobClient:     HDFS_BYTES_WRITTEN=13
11/04/12 12:58:08 INFO mapred.JobClient:   Map-Reduce Framework
11/04/12 12:58:08 INFO mapred.JobClient:     Reduce input groups=4
11/04/12 12:58:08 INFO mapred.JobClient:     Combine output records=4
11/04/12 12:58:08 INFO mapred.JobClient:     Map input records=12
11/04/12 12:58:08 INFO mapred.JobClient:     Reduce shuffle bytes=80
11/04/12 12:58:08 INFO mapred.JobClient:     Reduce output records=4
11/04/12 12:58:08 INFO mapred.JobClient:     Spilled Records=8
11/04/12 12:58:08 INFO mapred.JobClient:     Map output bytes=144
11/04/12 12:58:08 INFO mapred.JobClient:     Combine input records=12
11/04/12 12:58:08 INFO mapred.JobClient:     Map output records=12
11/04/12 12:58:08 INFO mapred.JobClient:     Reduce input records=4
[w1826214@avatarnode0 hadoop-0.20.2]$ bin/hadoop dfs -cat /user/w1826214/output/result
11 9
11 13
```

图 6-62 操作界面

习题

1. 分析比较 Hadoop 的优缺点。
2. Hadoop 里有哪些机制？解决了哪些问题？

3. HDFS 有哪些保障可靠性的措施?
4. MapReduce 模型适合 (不适合) 哪些环境?
5. 阐述 Hbase 的特点, 以及和其他 NoSQL 数据库的异同。

参考文献

- [1] <http://hadoop.apache.org/>
- [2] <http://www.ibm.com/developerworks/cn/opensource/os-cn-hadoop1/index.html>
- [3] http://cn.hadoop.org/doc/hdfs_design.html
- [4] <http://hadoop.apache.org/core/docs/r0.18.2/api/index.html>
- [5] Dean, Jeffrey & Ghemawat, Sanjay (2004). "MapReduce: Simplified Data Processing on Large Clusters". Retrieved Apr. 6, 2005
- [6] <http://www.ibm.com/developerworks/cn/opensource/os-cn-hadoop1>
- [7] <http://wiki.apache.org/hadoop/HBase/HBaseArchitecture>
- [8] http://tech.ccidnet.com/art/5833/20090318/1713499_1.html
- [9] <http://blog.csdn.net/daidodo/archive/2008/02/24/2116761.aspx>

第7章 Eucalyptus: Amazon 云计算的开源实现

Eucalyptus 是 Amazon EC2 的一个开源实现,它与 EC2 的商业服务接口兼容。Eucalyptus 是一个面向研究社区的软件框架,它不同于其他的 IaaS 云计算系统,能够在已有的常用资源上进行部署,Eucalyptus 采用模块化的设计,它的组件可以进行替换和升级,为研究人员提供了一个进行云计算研究的很好的平台。目前 Eucalyptus 系统已经提供下载,并且可以在集群和各种个人计算环境中进行安装使用。随着研究的深入,Eucalyptus 已经引起越来越多的关注。本章将重点介绍 Eucalyptus 的体系结构、主要构件和访问接口,以及 Eucalyptus 的安装与使用。

7.1 Eucalyptus 简介

Eucalyptus^[1,4]是加州大学圣巴巴拉分校建立的开源项目。Eucalyptus 直译为“桉树”,实际上,是语句“Elastic Utility Computing Architecture for Linking Your Programs to Useful Systems (将程序连接到有用系统的弹性效能计算体系结构)”的缩写。Eucalyptus 全局掌控各种基于物理设施的虚拟设备,实现对整个集群的计算能力的动态配置。

Eucalyptus 已经从单一支持 EC2 逐步扩展到支持包括 S3 在内的多种客户端接口,基本架构如图 7-1 所示^[7]。其中,云控制器(Cloud Controller)是用户使用 Eucalyptus 云服务的接入点;集群控制器(Cluster Controller)负责监控集群内的信息(包括节点虚拟机的执行情况、网络通信等);节点控制器(Node Controller)控制虚拟机的运行状态。云控制器和集群之间可以通过互联网连接,集群内节点之间通过内部网络通信。

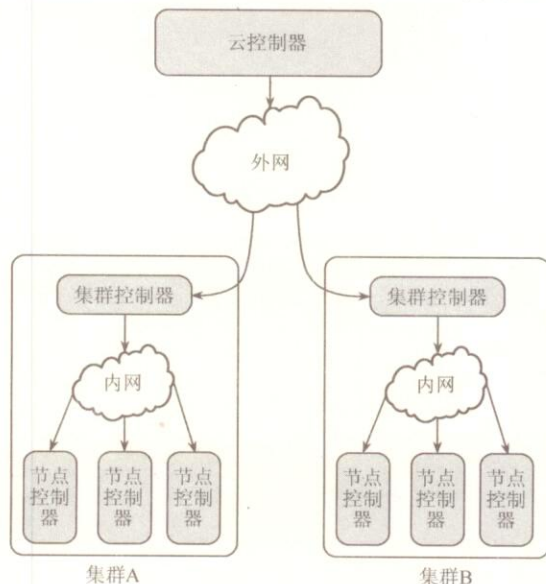


图 7-1 Eucalyptus 基本架构图

7.2 Eucalyptus 技术实现

7.2.1 体系结构

可扩展性和非侵入性是 Eucalyptus 的两个主要设计目标。Eucalyptus 采用简单的组织结构和模块化的设计和开源的 Web 服务技术。Eucalyptus 组件由若干个 Web 服务组成（由 WSDL 文档描述接口），且使用 WS-Security 策略支持安全通信。Eucalyptus 依赖符合行业标准的软件包如 Axis2、Apache 和 Rampart 等。只要使用 Eucalyptus 的节点通过 Xen 支持虚拟化执行和部署 Web 服务，就可在不修改基本基础设施的情况下安装和执行 Eucalyptus。

学术研究组织可以访问多种资源，如小的集群、工作站池和各种服务器及台式机。由于 IP 地址匮乏及对完全通过互联网访问资源引起的安全方面的担忧，系统管理员通常将集群部署在一个私有的不可路由的网络上，该网络由一个头节点负责在计算池和互联网之间的路由。虽然这种配置通过使用最少的公共可路由的 IP 地址提供安全保障，但这意味着大部分机器可以和外部机器连接的同时，外部机器却不能直接和集群内部的机器进行通信。比如，有两个小的 Linux 集群，一个小的服务器池及一个工作站集合，每个集群有一个具有可公开访问的 IP 地址的前端机器，而其节点之间、节点与集群头节点之间通过专用网络相连。服务器和工作站具有公开的 IP 地址，但是这些工作站都位于防火墙的后面，不能够从外部连接它们。显然，这种情形下，安装一个完全互联的系统是不可能的，因为许多机器只能向外部主机发起连接，或者完全与外界网络隔离。此外，两个集群中的节点由于位于不同专用网络，或许还有重叠的私有 IP 地址。为了在单一的云计算系统中使用所有的资源，Eucalyptus 采用了分层的拓扑结构，如图 7-2 所示^[4]。其中，CLC 代表云控制器（Cloud Controller），CC 代表集群控制器（Cluster Controller），NC 代表节点控制器（Node Controller）。

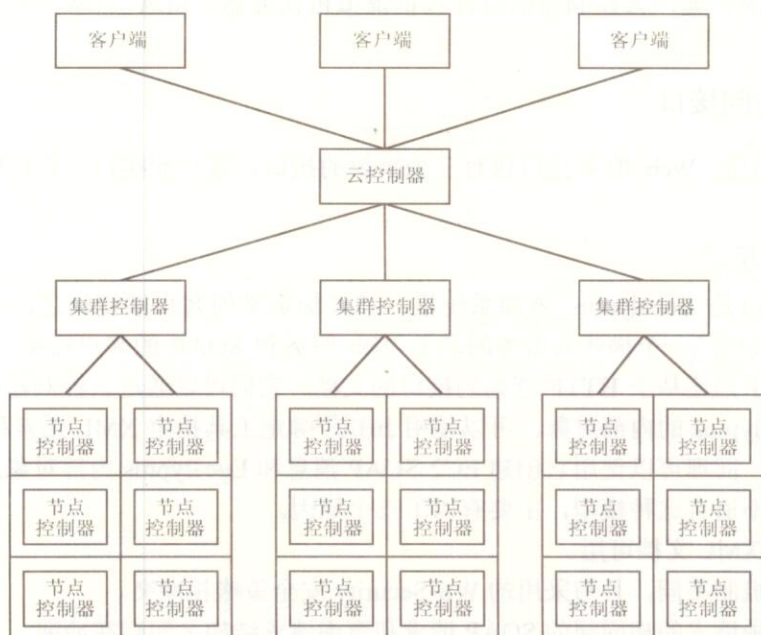


图 7-2 Eucalyptus 的分层拓扑结构

这些分层的组件能够容易地在常见的网络分层结构上进行安装。

7.2.2 主要构件

Eucalyptus 的主要构件包括节点控制器、集群控制器和云控制器。

1. 节点控制器

节点控制器负责管理一个物理节点。节点控制器是运行在虚拟机寄宿的物理资源上的一个组件，负责启动、检查、关闭和清除虚拟机实例等工作。可以安装多个节点控制器，但一台机器上只需运行一个节点控制器，因为一个节点控制器可以管理该节点上运行的多个虚拟机实例。

2. 集群控制器

集群控制器运行在集群的头节点或服务器上，可以访问私有或公共网络。一个集群控制器可以管理多个节点控制器，集群控制器负责从所属的节点控制器收集节点的状态信息，根据这些节点的资源状态信息分配虚拟机实例，并管理公共和私有实例网络的配置。

3. 云控制器

每个安装系统只有一个云控制器。云控制器相当于系统的中枢神经，是用户的可见入口点和做出全局决定的组件，负责处理用户发起的请求或系统管理员发出的管理请求，做出高层的虚拟机实例调度决定，处理服务等级协议和维护系统及用户相关的元数据。

云控制器由一组服务组成，这些服务用于处理用户请求、验证和维护系统、用户元数据（虚拟机映像和 SSH 密钥对等），并可管理和监视虚拟机实例的运行。这些服务由企业服务总线 ESB 来配置、管理和发布。

Eucalyptus 的设计强调透明和简单，以便促进 Eucalyptus 的实验和扩展。为了达到这一粒度级别的扩展，云控制器的组件包括虚拟机调度器、SLA 引擎、用户接口和管理接口等。

7.2.3 访问接口

云控制器中的 Web 服务接口包有三个重要的接口：客户端接口、管理接口和实例控制接口。

1. 客户端接口

客户端接口是 Eucalyptus 内部系统接口和外部定义的客户端接口之间的转换器。例如，Amazon 提供了一个描述其服务的基于 Web 服务和 SOAP 的客户端接口的 WSDL 文档，和一个用于描述基于 HTTP 查询的接口的文档，它们可以通过云控制器的用户接口服务转换成 Eucalyptus 的内部对象。可以使用 JiBX^[3] 绑定工具指定 XML 元素和 Java 对象实例之间的映射，同理可以使用它创建 EC2 SOAP 消息和 Eucalyptus 内部对象之间的映射。但查询接口却不适合这种模型，主要有以下三个原因。

- (1) 没有 XML 文档可用。
- (2) 认证机制不同，且与采用的 WS-Security 安全策略相冲突。
- (3) 在同种请求的相同域的 SOAP 请求和查询请求结构之间存在冲突。

但是由于 EC2 的查询接口是 SOAP 接口的严格子集，Eucalyptus 开发了一个简单的绑

定框架,在有关注释的引导下将 HTTP 参数名字映射到对象字段,然后依靠目标对象的注释来澄清和修改映射的不一致性,最后 JiBX 使用命名空间为 EC2 SOAP 接口的边界对象分组。结果包括以下两个方面。

(1) JiBX 验证该对象,它实际上是一个合法的 SOAP 接口请求,也是一个合法的 EC2 客户端请求。

(2) 分组后的 XML 文档可以当做 SOAP 的一部分来进行进一步的处理。

2. 管理接口

除了支持主要的任务如启动和停止一个实例外,云计算基础设施还应该支持基本的管理任务,如添加和移除用户及管理磁盘映像等。Eucalyptus 通过一个基于 Web 的接口进行管理操作,它由云控制器通过一个基于 Web 的接口或通过命令行的方式实现,管理接口只有系统管理人员可以看见,且管理接口具有唯一性。可以管理员手动或以在线申请验证的方式添加用户。在线验证方式往往要求用户提供邮件地址,并单击系统发送到邮件中的验证链接获取验证,因为 Eucalyptus 将用户身份和其邮件地址进行了绑定。用户添加成功后,管理员可以暂时或永久性地移除该用户,且管理员可以管理、终止正在运行的实例。

通过管理入口,还可以将磁盘映像添加到系统中。一个映像包括与 Xen 兼容的客户操作系统内核,一个根文件系统映像及一个可选的 RAM 硬盘映像,添加一个映像的过程包括加载这三个映像文件并给它们命名。管理员可以暂时或永久性地删除一个映像,另外管理员还可以通过控制器配置文件添加和删除集群中的节点。

3. 实例控制接口

云控制器提供虚拟机控制服务(VmControl Service)管理虚拟机实例元数据的创建。虚拟机控制器不间断地维护一个基本资源状态的简单本地描述,如一个集群控制器潜在的能够创建的实例的个数。当发起一个实例创建请求时,虚拟机控制器将和云控制器的其他服务进行协作,将用户的请求分解成映像、密钥对、网络和安全组等,并根据相应的元数据和资源应用配置策略预先生成一个解决方案,然后将消息散播至涉及的集群控制器,集群控制器将调度这些请求到其所辖节点控制器,最后由节点控制器创建虚拟机实例来运行用户作业和应用程序。

7.2.4 服务等级协议

服务等级协议(Service-Level Agreement, SLA)是作为消息处理服务的扩展来实现的,消息处理服务可以检查、修改、丢弃消息及虚拟机控制器(VmControl)保存的状态。虚拟机控制器决定要访问的资源并执行系统级或用户指定的服务等级协议,虚拟机控制器依靠一个本地状态模型做出这些决定,该模型通过集群控制器获取其实例的可用性、配置、虚拟网络和注册映像的状态信息,虚拟机控制器依靠这些信息及其更新事件来做出全局服务决定。Eucalyptus 实现了一个可扩展的 SLA 模式,它耦合了状态模型和事件处理以支持对 SLA 的进一步定量研究。

虚拟机控制器依靠本地模型做出决策。为了保持模型更新,每一个云控制器会以被动轮询的方式获取其实例的状态,包括实例的可用性、分配情况、虚拟网络及注册的映像等。通过轮询获取的信息被当成判断的基准,用户请求被提交给事务,当资源满足该事务时就提交该事务进行处理。然而,当由于网络问题引起信息丢失或资源状态改变时,该模

型可能变得不一致,从而导致系统同一个不能满足的用户达成服务协议。然而,由于轮询是半同步的,丢失信息能够被确定,且模型处于无效的时间点也可以被检测到。最终,模型在给定时间段的哪个时间点上无效可以被计算出来,从而可以避免模型不一致时导致的问题。

Eucalyptus 已经实现了一个简单却强大的初步的 SLA, 它可以使用户控制其实例的高层网络拓扑结构。Eucalyptus 使用 Amazon EC2 提出的“区域”(Zone)概念指代由计算和存储资源组成的“池”(Pools)或“集群”(Clusters)。区域是由多个节点控制器和单个集群控制器从逻辑上构成的机器集合。Eucalyptus 允许用户为一个实例的执行指定一个区域配置,该配置提供不同的管理和网络性能参数,根据该配置,一个实例集可以在一个集群或跨集群运行,以获得所要的性能。

考虑到用户得到的资源数量及其拓扑结构的相对动态性,Eucalyptus 将区域的概念进行扩展以支持不同的服务协议。目前 Eucalyptus 所提供的区域允许用户在执行作业时,可以具有多种选择,包括根据服务协议获取指定的集群、选择空闲的集群、指定单个及多个集群来为自己服务。

7.2.5 虚拟组网

虚拟机实例的互联问题是构建云计算基础设施最重要的工作之一。不同于由物理机器组成的物理网络,虚拟机实例组成的网络是一个虚拟化的网络,通过虚拟化处理可以使网络简单和易配置。

某个域内的虚拟机实例彼此之间应有网络连接,且它们中至少有一个虚拟机实例和外部公共网络相连,以便于为其所有者提供访问入口及与其他域实例进行交互。由于用户对自己所监管的虚拟机具有超级用户权限,可以访问基本的网络接口,因此具有获取系统 IP 和 MAC 地址的能力,并可对系统网络造成干扰。此外,如果两个实例运行在同一台物理机器上,虚拟机用户可以影响和窥探另外一个虚拟机的网络包,这将导致安全问题。因此在有不同用户共享的云计算平台上,协作完成单一任务的虚拟机之间应该可以通信,而属于不同用户的虚拟机之间应该是通信隔离的。

为了解决上述问题,每个虚拟机提供两个虚拟网络接口,一个作为公共接口,一个称为私有接口。公共接口的作用是和用户所管辖虚拟机的外部进行通信,或在由服务协议定义的可用区域的实例之间进行通信。在一个有可用的 IP 地址的环境中,这些地址可以在虚拟机实例启动时分配给它们以允许实例进行通信。而在具有支持外部通信路由器的私有网络中,虚拟机实例进行网络地址转换后,其公共网络接口可以分配一个有效的私有地址,通过网络地址转换的路由器访问外部网络。实例的私有接口只能在跨域的虚拟机之间进行通信,以解决不同虚拟机实例之间的通信问题。

集群控制器负责创建和销毁实例虚拟网络接口,节点控制器通过配置以下三种方式建立公共接口网络。

(1) 使虚拟机的公共接口直接连接到与物理机器网络相连的以太网网桥软件上,管理员可以像处理常规的 DHCP 请求那样处理虚拟机网络的 DHCP 请求。

(2) 允许管理员定义一个动态的 IP 地址池和一个网络,集群控制器通过一个接口连接到该网络,运行在集群控制器上的 DHCP 服务器负责在实例启动时将地址动态地分配给这些实例。

(3) 定义静态的 MAC/IP 地址对, 每个实例启动时系统给其分配一个空闲的 MAC/IP 对, 实例终结后释放该 MAC/IP 对。

实例的私有接口通过桥接器连接到一个被称为虚拟分布式以太网 (Virtual Distributed Ethernet, VDE)^[5] 的全虚拟以太网系统, VDE 是一个以太网协议的线程级实现。VDE 网络通过通用的 TUN/TAP 接口连接到真正的以太网上, TUN/TAP 提供了从 Linux 内核到用户空间的以太网包通信服务。

一个 VDE^[5] 网络由 VDE 交换机和它们之间的电缆连接构成, VDE 交换机位于节点控制器和集群控制器上, VDE 交换机采用生成树协议 (Spanning Tree Protocol), 它防止环路的同时允许冗余存在于网络之中。如果没有防火墙, VDE 网络是完全连接的, 即 VDE 交换机和另外的每一个 VDE 交换机都直接相连; 有防火墙的情况下, VDE 至少和系统中的一个 VDE 交换机相连。

为了保证系统的安全, 需要对实例实行网络流量隔绝。这些实例可以是运行在相同主机上, 也可以运行在同一物理网络中的不同机器上。系统要求任意两个实例之间不能够相互检查和修改彼此的网络通信, 具体方法是用虚拟局域网 (VLAN) 给属于特定用户的实例集打上网络标签, 以此来隔绝和转发不同的网络流量。

7.3 Eucalyptus 安装与使用

准备两台 PC, 一台作为 Front-end 节点, 一台作为 node 节点。Front-end 节点可以用普通的低配机器, node 节点推荐使用高配机器 (支持 VT-d 技术), 即可开始安装。

7.3.1 在 Linux 系统中安装 Eucalyptus

1. CentOS-5 + XEN 环境安装 Eucalyptus

1) 安装配置 CentOS-5 操作系统

由于 Eucalyptus 对系统配置要求较高, 需要在物理机上安装 CentOS 系统, 不要在 VMware 等虚拟机中安装 CentOS 系统来搭建 Eucalyptus 平台。

先在两台物理机上安装 CentOS-5 系统。在安装最后阶段, 需要配置防火墙和 SELinux, 此两项都选择禁止, 最后重新启动系统配置 CentOS 的更新源, 为了提高在线安装软件的速度, 如果是教育网用户可以选择上海交通大学、清华大学或中科院的更新源, 如果是电信或网通用户可以使用 163 的更新源。

2) 安装 Eucalyptus 系统^[6]

(1) 在 Front-end 节点安装下述软件。

(a) 安装 Network Time Protocol, 用来同步 Front-end 节点和 node 节点的时钟。

```
yum install -y ntp
ntpdate pool.ntp.org
```

(b) 安装 java、ant、dhcp、bridge、perl 和 httpd 等服务。

```
yum install -y java-1.6.0-openjdk ant ant-nodeps dhcp bridge-utils
yum install -y perl-Convert-ASN1.noarch scsi-target-utils httpd
```

(c) 安装 Eucalyptus 系统前端软件。

在 /etc/yum.repos.d/ 目录下新建 eucalyptus.repo, 内容如下:

```
[euca]
name=Eucalyptus
baseurl=http://www.eucalyptussoftware.com/downloads/repo/eucalyptus/$VERSION/yum/centos/
enabled=1
```

设置安装 Eucalyptus 的版本号。

```
export VERSION=2.0.2
export ARCH=x86_64 或 i386
```

安装 Eucalyptus 的 Cloud Controller, Cluster Controller, Walrus, Storage Controller。

```
yum install eucalyptus-cloud.$ARCH eucalyptus-cc.$ARCH eucalyptus-walrus.$ARCH eucalyptus-sc.$ARCH --nogpgcheck
```

(2) 在 node 节点安装下述软件。

(a) 安装 Network Time Protocol, 用来同步 Front-end 节点和 node 节点的时钟。

```
yum install -y ntp
ntpdate pool.ntp.org
```

(b) 安装 java、ant、bridge 和 perl 等服务。

```
yum install -y java-1.6.0-openjdk ant ant-nodeps bridge-utils perl-Convert-ASN1.noarch scsi-target-utils
```

(c) 安装 xen 和 xen 的 linux 内核。

```
yum install -y xen xen-kernel
```

修改 xen 的配置。

```
sed --in-place 's/#(xend-http-server no)/(xend-http-server yes)/' /etc/xen/xend-config.sxp
sed --in-place 's/#(xend-address localhost)/(xend-address localhost)/' /etc/xen/xend-config.sxp
/etc/init.d/xend restart
```

修改/boot/grub/menu.lst 中的 default=1, 保存, 重启计算机。重启后, 开启被 xen 修改后的 CentOS 内核。

用命令 “uname -r” 查看允许的是否是 xen 的内核。

用命令 “export VERSION=2.0.2” 引入安装 Eucalyptus 的版本号。

在/etc/yum.repos.d/目录下新建 Eucalyptus.repo, 内容如下:

```
[euca]
name=Eucalyptus
baseurl=http://www.eucalyptussoftware.com/downloads/repo/eucalyptus/$VERSION/yum/centos/
enabled=1
```

安装 Eucalyptus 的 Node Controller(nc)。

```
yum install eucalyptus-nc.$ARCH --nogpgcheck
```

(3) 注册 Eucalyptus 组件。

```
euca_conf --register-walrus walrus 所在节点 ip
euca_conf --register-cluster mycluster cluster 节点 IP //IP 地址前的是自定义的 CLC 的名字
euca_conf --register-sc mycluster storage 节点 IP //注册 storage control
euca_conf --register-nodes NC 节点的 IP //注册 NC 节点
```

可以通过以下命令来验证注册。

```
euca_conf --list-walruses
euca_conf --list-clusters
euca_conf --list-scs
euca_conf --list-nodes
```

2. Ubuntu + KVM 环境安装 Eucalyptus

1) 下载 Ubuntu10.04 服务版

Ubuntu10.04 服务版自带 Eucalyptus1.6.2 版本, 是目前最稳定的版本。在 Ubuntu 官方下载 ubuntu10.04 服务版 32 位和 64 位。

下载网址: <http://www.ubuntu.com/business/get-ubuntu/download>, 还可以在该网址下载

ubuntu10.10 服务版, 自带 Eucalyptus2.0 版本。

2) 安装 Ubuntu10.04 服务版^[6]

(1) 安装前端。

将下载的 ubuntu10.04 32 位服务版安装包刻录成光盘。将光盘放入所要安装机器的光驱, 开始正式安装。当出现安装界面后, 会让你选择所安装方式, 此时选择 “Install Ubuntu Enterprise Cloud” 菜单项, 然后根据自己的需要选择不同的语言、网络配置、主机名称等, 直到出现如图 7-3 所示界面。

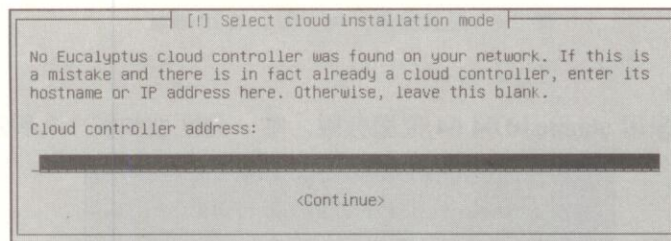


图 7-3 Eucalyptus 私有云安装模式的选择

如果已经安装 Cloud controller, 输入所对应的 IP, 否则选择 Continue。出现如图 7-4 所示的界面。

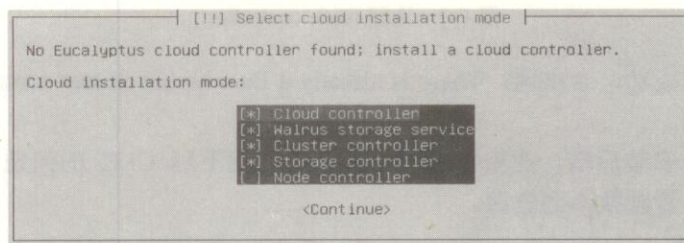


图 7-4 前端组件选择

选中 Cloud controller、Walrus storage service、Cluster controller、Storage controller 后, 单击 Continue。分区、安装系统、更新软件包后, 出现如图 7-5 所示的界面。

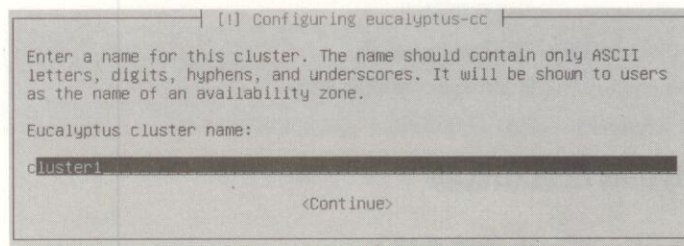


图 7-5 配置 Eucalyptus Cluster controller

输入集群名称, 例如 cluster1。单击 Continue。进入如图 7-6 所示界面。

如果所在环境下, 采用 DHCP 自动获取 IP, 那么需要置空, 否则根据实验环境, 填入 IP 段。

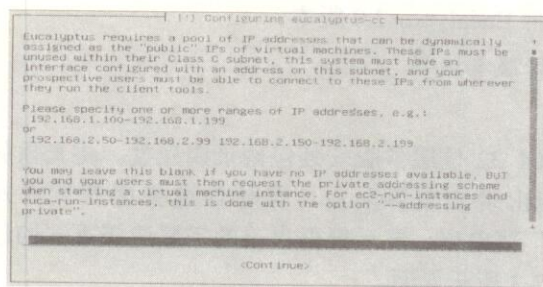


图 7-6 配置集群环境下的公共 IP 地址

(2) 安装后端。

后端的安装，采用 ubuntu10.04 64 位服务器版，唯一的区别如图 7-7 所示。

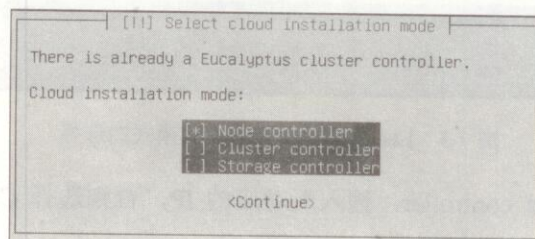


图 7-7 选择 Node controller

如果前端安装成功，会出现：There is already a Eucalyptus cluster controller。此时根据默认直接 Continue。

注意：如果先安装后端，在安装过程中就会出现找不到 CLC 的问题，并且后端不能在虚拟机上安装，否则找不到资源。

(3) 注册 node。

上述五大组件安装完成后，除了 node 以外，采用自动注册的方式，用户不需再注册。在前端机器上，用下列命令注册 node 至 Cluster controller。

```
sudo euca_conf --discover-nodes//该命令将找到目前环境中的所有没有注册的 node。
```

使用下列命令可以查看目前注册的情况：

```
sudo euca_conf --list-clusters //查看集群信息
```

```
sudo euca_conf --list-walrus //查看 walrus 信息
```

```
sudo euca_conf --list-scs //查看 Storage controller 信息
```

```
sudo euca_conf --list-nodes //查看节点信息
```

7.3.2 Eucalyptus 配置和管理

1. 登入 Eucalyptus 的 web 管理系统

在客户端，用浏览器访问 <https://cloud-controller-ip-address:8443/>，出现登录界面，输入用户名和密码。系统默认登录用户名和密码同为 admin。

登录系统后，主界面包含：证书、镜像、在线安装镜像、用户管理、配置等，如图 7-8 所示。

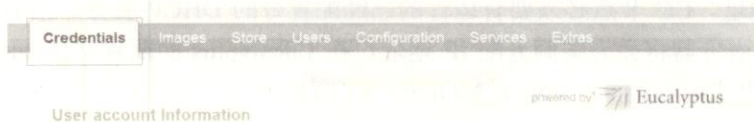


图 7-8 Eucalyptus 主界面

Configuration 菜单包括：云端配置、DNS 配置、walrus 配置、集群配置、存储配置、VM 类型配置。用户可以根据实验条件具体配置。

在 Store 菜单中，单击 serach，可以找到 ubuntu 官方提供的镜像模板。例如 ubuntu9.10 等，用户可以在线安装该模板，也可以在 Extras 下载其他操作系统模板。

2. 安装配置客户端

(1) 下载证书，将其放在 ~/.euca 目录下，然后解压。

```
mkdir -p ~/.euca
cd ~/.euca
unzip euca2-admin-x509.zip
chmod 0700 ~/.euca
chmod 0600 ~/.euca/*
. eucarc
```

(2) 在线安装 euca2ools 用户工具。

```
sudo apt-get install euca2ools
```

CentOS 下安装 euca2ools 工具。设置 euca2ools 版本为 1.3.1。

```
export VERSION=1.3.1
export ARCH=x86_64 或 i386
```

新建并编辑/etc/yum.repos.d/euca2ools.repo 文件。

```
[euca2ools]
name=Euca2ools
baseurl=http://www.eucalyptussoftware.com/downloads/repo/euca2ools/$VERSION/yum/centos/
enabled=1
```

在线安装 Eucalyptus 的 euca2ools 工具。

```
yum install euca2ools.$ARCH --nogpgcheck
```

(3) 申请 key，以便管理和登入某个虚拟机。

euca-add-keypair mykey > mykey.private//把 key 导入到 mykey.private 文件中。

```
chmod 0600 mykey.private //改变 mykey 的读写权限
```

(4) 验证系统是否安装、配置正确。

运行 euca-describe-availability-zones verbose，如果所有资源全为 0，说明集群与 Node 之间注册错误，请重新注册。

(5) 网络配置。

Eucalyptus 系统为用户提供 4 种网络模式：SYSTEM，STATIC，MANAGED-NOVLAN 和 MANAGED。选择适当的模式即可。

3. 各种网络配置模式介绍

Eucalyptus 系统提供的 4 种网络模式的功能如下。

1) SYSTEM 模式

SYSTEM 模式是四种网络配置模式中最简单的一种，建议第一次安装 Eucalyptus 平台的用户，选择配置为此模式。SYSTEM 模式特点：不具有 IP 地址管理功能，不支持为实例

分配外部 IP 地址, CC 节点必须要有运行的已经配置好的 DHCP 服务。当启动实例时, CC 节点的 DHCP 服务器将会为实例分配 IP 地址。在 Eucalyptus 的配置文件 `eucalyptus.conf` 中需要修改网络模式内容如下: `VNET_MODE="SYSTEM"`。

2) STATIC 模式

STATIC 模式是一种实现 IP 地址与 MAC 地址对应的网络模式。STATIC 模式具有 IP 地址管理功能, 支持为实例分配外部 IP 地址, CC 节点需要安装并运行未配置好的 DHCP 服务, Eucalyptus 的 CC 节点会调用 DHCP 服务, 为实例分配 IP 地址和与其对应的 MAC 地址。在 Eucalyptus 的配置文件 `eucalyptus.conf` 中需要修改网络模式内容如下:

```
VNET_MODE="STATIC"
#VNET_SUBNET="192.168.1.0"
#VNET_NETMASK="255.255.255.0"
#VNET_BROADCAST="192.168.1.255"
#VNET_ROUTER="192.168.1.1"
#VNET_DNS="192.168.1.1"
#VNET_MACMAP="AA:DD:11:CE:FF:ED=192.168.1.2AA:DD:11:CE:FF:EE=192.168.1.3"
```

3) MANAGED-NOVLAN 和 MANAGED 模式

这两种模式非常相似, 具有 IP 地址管理功能, 支持为实例分配外部 IP 地址, 并进一步支持弹性 IP 地址分配 (弹性 IP 地址分配是指可以动态的给实例分配并收回外部 IP 地址的功能)。这两种模式的不同之处是, MANAGED 模式支持 VLAN 的划分, 实现实例之间的隔离, 增加了安全性。与 STATIC 模式一样, CC 节点需要运行未配置好的 DHCP 服务来实现 Eucalyptus 的 IP 地址管理和分配。在 Eucalyptus 的配置文件 `eucalyptus.conf` 中需要修改网络模式内容示例如下:

```
#VNET_MODE="MANAGED"//或者 MANAGED-NOVLAN
#VNET_SUBNET="192.168.0.0"
#VNET_NETMASK="255.255.0.0"
#VNET_DNS="your-dns-server-ip"
#VNET_ADDRSPERNET="32"
#VNET_PUBLICIPS="your-free-public-ip-1 your-free-public-ip-2 ..."
#VNET_LOCALIP="your-public-interface's-ip"
#VNET_CLOUDIP="your-cloud-controller's-ip"
```

7.3.3 Eucalyptus 常用命令的示例和说明

1. 系统查询

- (1) 查看可用的资源域: `euca-describe-availability-zones verbose`。
- (2) 描述密钥: `euca-describe-keypairs`。
- (3) 列出 Eucalyptus 中的实例信息: `euca-describe-instances`。
- (4) 列出 Eucalyptus 中可用的镜像: `euca-describe-images`。
- (5) 列出 Eucalyptus 中的 volume : `euca-describe-volumes`。

2. 创建密钥

(1) `euca-add-keypair mykey >mykey.private`。

(2) `chmod 0600 mykey.private`。

3. 启动一个实例

`euca-run-instances -k key -n 1 -t m1.small emi-xxxxxxx`。

-k 参数是在安装 euca2ools 时申请并导入的 `keypair.private`。

-n 参数是需要启动实例的个数。

-t 的参数是实例的类型: `m1.small`, `c1.medium`, `m1.large`, `m1.xlarge`, `c1.xlarge`。

`emi-xxxxxxx` 是实例的镜像。

4. 终止或重启一个实例

(1) `euca-terminate-instances i-xxxxxxx`。

(2) `euca-reboot-instances i-xxxxxxx`。

`i-xxxxxxx` 为实例号。

5. 创建和挂载 volume

(1) `euca-create-volume -S size -Z zone`。

size 表示创建 volume 的大小。

zone 代表创建 volume 所在的 cluster。

(2) `euca-attach-volume -i instance -d device volume_id`。

Instance 代表要挂载的实例 ID。

device 是实例中的设备号, 如 `sdb1` 等。

volume_id 是要挂载的 volume 的 ID 号。

6. 上传 Eucalyptus 镜像

(1) 打包 Eucalyptus 镜像文件。

`euca-bundle-image -i 文件 --kernel true`。

打包内核文件: 参数为 `--kernel true`。

打包引导文件: 参数为 `--ramdisk true`。

打包系统镜像: 参数为 `--kernel $EKI --ramdisk $ERI`, 其中 EKI 和 ERI 是代表要绑定的内核镜像 ID 和引导镜像 ID。

(2) 上传打包好的 Eucalyptus 镜像文件。

`euca-upload-bundle -b bucket -m manifest`。

bucket 是在服务器存储镜像的文件夹名, 需要自己定义。

manifest 是 `euca-bundle-image` 生成的 `xxxxxx.manifest.xml` 文件。

(3) 注册已上传镜像。

`euca-registerbucket-file/xxxxxx.manifest.xml`。

参数为 `euca-upload-bundle` 命令行中最后生成的 `manifest.xml` 文件路径。

习题

1. 阐述 Eucalyptus 的基本架构及主要构件的功能。
2. Eucalyptus 系统的工作流程是什么?
3. 在 Linux 系统中安装 Eucalyptus 需要注意哪些问题?
4. Eucalyptus 系统提供哪几种网络配置模式?
5. Eucalyptus 的常用命令分哪几类? 分别说明其功能。

参考文献

- [1] <http://www.eucalyptus.com/>
- [2] libvirt. Enomaly Elastic Computing Platform (ECP). [http://wiki.libvirt.org/page/Enomaly_Elastic_Computing_Platform_\(ECP\)#Enomaly_Elastic_Computing_Platform_.28ECP.29](http://wiki.libvirt.org/page/Enomaly_Elastic_Computing_Platform_(ECP)#Enomaly_Elastic_Computing_Platform_.28ECP.29)
- [3] JiBX home page. <http://jibx.sourceforge.net/>
- [4] Daniel Nurmi, Rich Wolski, et al, Eucalyptus : A Technical Report on an Elastic Utility Computing Archietcture, http://open.eucalyptus.com/documents/nurmi_et_al-eucalyptus_tech_report-august_2008.pdf
- [5] Virtual bistributed ethernet (VDE) home page – <http://vde.sourceforge.net/>
- [6] Eucalyptus Administrator Guide Version 1.6.pdf: <http://bbs.chinacloud.cn/showtopic-3428.aspx>
- [7] D.Nurmi, R.Wolski, C.Grzegorzcyk, G.Obertelli, S.Soman, L.Youseff and D.Zagorodnov. The eucalyptus open-source cloud-computing system. In Cloud Computing and Applications 2008(CCA08), 2008

第8章 其他开源云计算系统

自 Hadoop、Eucalyptus 这两种典型的开源云计算系统问世后, 利用这些开源项目提供的各种工具, 研究者可以在实验室用很低的成本, 在由普通机器构成的集群系统中模拟出近似商业云的环境, 这也掀起了开源云计算系统研究的热潮, 本章将介绍一些其他新兴的开源云计算系统, 带领读者进入多姿多彩的开源云计算世界。

8.1 简介

本节对一些开源云计算系统作概念性介绍, 在后续的三节中, 重点介绍 Cassandra、Hive 和 VoltDB 这三个新兴的开源云计算系统。

8.1.1 Cassandra

Cassandra 是一套高度可扩展、最终一致、分布式的结构化键值存储系统^[1]。它结合了 Dynamo 的分布技术和 Google 的 Bigtable 数据模型, 更好地满足了海量数据存储的需求, 解决了应用与关系数据库模型之间存在的非依赖关系。同时, Cassandra 变更垂直扩展为水平扩展, 相比其他典型的键值数据存储模型, Cassandra 提供了更为丰富的功能。

Cassandra^[2]最初由 Avinash Lakshman (Amazon's Dynamo 的作者之一) 和 Prashant Malik (Facebook 工程师) 在 Facebook 设计开发, 2008 年 Facebook 把它贡献给了开源社区。从某种程度上说, 可以把 Cassandra 看成是 Dynamo 的升级版本 2.0, 或者是 Dynamo 与 BigTable 的结合。Cassandra 的设计目标如下:

- (1) 高可用性。
- (2) 最终一致性。
- (3) 动态可伸缩。
- (4) 可以动态调整一致性/持久性与延时。
- (5) 节点管理要保持低开销。
- (6) 最小化管理开销。

考虑到 Cassandra 的设计目标, 在一致性、可用性和分区容忍度 (CAP 理论) 的折中问题上, Cassandra 选择了 AP (即可用性和分区容忍度)。针对基本的一致性哈希分布不均匀且不能根据节点能力强弱分配的缺点, Dynamo 让每个点管理环中的多个位置, 而 Cassandra 让负载轻的节点在环上移动来实现负载均衡。在应用中, Cassandra 表现出了以下几个突出特点^[3]。

(1) 模式灵活。使用 Cassandra 就像文档存储, 可以在系统运行时随意地添加或删除字段, 而不必提前去解决记录中的字段。特别是在大型部署上将极大地提升效率。

(2) 真正的可扩展性。Cassandra 是纯粹意义上的水平扩展。为给集群添加更多容量, 可以动态添加节点而不必重启任何进程、改变应用查询或手动迁移任何数据。

(3) 多数据中心识别。通过调整节点的布局避免某一个数据中心出现故障, 一个备用的数据中心将至少有每条记录的完全复制。

(4) 范围查询。可以设置键的范围来进行键值查询。

(5) 列表数据结构。在混合模式可以将超级列添加到五维, 这使得每个用户的索引将变得非常方便。

(6) 分布式写操作。可以在任何地方, 任何时间集中读或写任何数据, 并且不会有任何单点失败。

当前, Cassandra 系统正在得到迅猛发展, 社交网站巨头 Facebook 采用 Cassandra 存储 Inbox, Twitter、Webex 和 Digg 也做了大量向 Cassandra 的迁移工作。8.2 节将从数据模型、存储机制等方面详细介绍 Cassandra。

8.1.2 Hive

Hive^[4]起源于 Facebook, 是一个基于 Hadoop 的数据仓库工具, 同时也是 Hadoop 的一个主要子项目。Hive 提供了一系列的工具, 可以用来进行数据的提取、转换和加载 (ETL), 同时可以实现对 Hadoop 中大规模数据的存储、查询和分析。Hive 定义了一种简单的类似 SQL 语言——HiveQL。HiveQL 使熟悉 SQL 的用户可以很方便地在 Hadoop 中查询数据。同时 Hive 还有很强的灵活性, 没有将用户限制在一个框架中, 主要表现在当 Hive 内建的 Mapper 和 Reducer 不能满足用户的需求时, 用户可以通过 Map/Reduce 将自己开发的 Mapper 和 Reducer 加入到 Hive, 以满足用户特殊的需求。

Hive 没有定义所谓的 Hive 格式的数据, 可以在 Thift 上很好地工作, 控制分隔符, 甚至可以自己定义数据格式。

作为 Hadoop 的主要子项目, Hive 秉承开源的精神, 在不断地发展中, 不断有新的特性加入其中。现在已经增加和将要增加的一些新特性如下^[5]:

- (1) 增加了用于收集分区和列的水平统计数值的命令;
- (2) 支持在 Partition 级别去更改 Bucket 的数量;
- (3) 在 Hive 中实现检索;
- (4) 为 Hive 增加并发模型;
- (5) 支持在两个或两个以上列中的差别选择;
- (6) 利用 bloom 过滤器提高连接的效果;
- (7) 建立 Hive 的授权结构和认证结构;
- (8) 在 Hive 中使用位图检索。

Hive 正在不断发展, 8.3 节将从整体构架、数据模型、使用语言等几个方面对 Hive 做一个简要介绍。

8.1.3 VoltDB

VoltDB 是 Mike Stonebraker (Postgres 和 Ingres 的联合创始人) 领导团队开发的下一代开源数据库管理系统。在 VoltDB 中, 所有事务被实现为 Java 存储过程。VoltDB 大幅降低了服务器资源开销, 单节点每秒数据处理远远高于其他数据库管理系统。它可以在现有的廉价服务器集群上实现每秒数百万次数据处理。不同于 NoSQL 的 key-value 储存, VoltDB 能使用 SQL 存取, 并支持传统数据库的 ACID 模型^[6]。

在 VoltDB 内部,采用并行单线程从而保证了事务的一致性和高效率,由于减少了锁的管理、资源管理等开销, VoltDB 具有极高的处理效率和速度。VoltDB 开发人员的测试表明^[7]:与一个优化过的传统数据库管理系统相比, VoltDB 可以达到后者 45 倍的事务处理速度。VoltDB 还具有以下一些传统数据库不能同时具有的优点:

- (1) 可以达到几乎线性的扩展;
- (2) 满足 ACID 特性;
- (3) 提供相比传统数据库好很多的性能;
- (4) 使用 SQL 作为数据库接口。

VoltDB 支持多节点并行事务处理,理论上不存在节点上限,目前 VoltDB 开发人员最大的测试集群可以达到 20 个节点。同时, VoltDB 还存在不少限制,主要包括以下几种^[8]:

- (1) 不支持动态修改 Schema;
- (2) 增加节点需要停止服务;
- (3) 不支持 xDBC;
- (4) Adhoc 查询性能不优化。

8.4 节将对 VoltDB 的架构和主要技术做简要的分析,使读者能够对 VoltDB 的特点和思想有个初步的了解。

8.1.4 Enomaly ECP

Enomaly ECP^[9]的全称是 Enomaly 弹性计算平台 (Enomaly's Elastic Computing Platform),之前的名称为 Enomalism。它是一个可编程的虚拟云架构,企业可以利用 ECP 将自己的数据中心和云计算服务商的设备连接起来,简化不同数据中心间虚拟机的转移及各种云应用发布的过程。

Enomaly ECP 是一个开放源代码项目,提供了一个功能类似于 EC2 的云计算框架。Enomaly ECP 基于 Linux,同时支持 Xen 和 KVM (Kernel Virtual Machine)。与其他解决方案不同的是, Enomaly ECP 提供了一个基于 TurboGears Web 应用程序框架和 Python 的软件栈。Enomaly ECP 架构如图 8-1 所示^[10]。



图 8-1 Enomaly ECP 架构

Enomaly ECP 具有以下几个特性。

- (1) 自动供应: ECP 提供自动化的云供应引擎, 用于配置、管理和部署成组的虚拟机。
- (2) 灵活性: ECP 能够直接将资源提供给用户的应用程序, 并且可以对获得授权使用应用程序者进行限制。
- (3) 可扩展性: ECP 的混合云模型无缝连接使用 ECP 的内部云和外部云提供商。
- (4) 整合现有基础设施: 通过 ECP 提供的丰富的 API, 用户可以直接管理和配置当前系统。

目前 Enomaly ECP 有两个版本: 一个是仍旧免费的社区版 (Community Edition), 另一个是提供全方位技术支持的服务提供商版 (Service Provider Edition)。

8.1.5 Nimbus

Nimbus^[11]是基于网格中间件 Globus 的作品, 从最早的 Virtual Workspace 演化而来, 提供与 EC2 类似的功能和接口。Nimbus 是一个开源的工具集, 它可以把集群部署到 IaaS 云中。Nimbus 通过一整套的工具来提供 IaaS 形式的云计算解决方案。Nimbus 属于科学云 (Science Clouds) 的一部分, 该项目创建的最初目的是为了搭建一个科学试验用的云计算平台, 但现在其应用已经超出了这个范围, 开始涉及其他领域。Nimbus 具有如下几个特点。

- (1) 具有两套 Web 服务接口——Amazon EC2 WSDL 和符合网格社区 WSRF 规范的接口。
- (2) 可以执行基于 Xen 管理程序。
- (3) 可以使用如 PBS 或 SGE 调度器去调度虚拟机。
- (4) 定义了一个可扩展架构, 用户可以根据项目的需求进行定制。

Nimbus 的架构图如图 8-2 所示^[12]。Nimbus 的架构比较复杂, 涉及的新概念较多。这里只对其中几个重要的概念进行简单解释。

- (1) 标准客户端 (Reference Client): 以命令行的方式访问服务, 全面支持 WSRF 前台的各种特性。
- (2) WSRF: Web Services Resource Framework, 即 Web 服务资源框架。
- (3) RM API: RM 是 Resource Management 的简写, 也就是资源管理。
- (4) 工作区 (Workspace): 实际上就是一个计算节点。

不过并不是所有的平台都必须使用图中的全部组件, 图中的各个组件之间可以采取不同的组合方式选择使用。

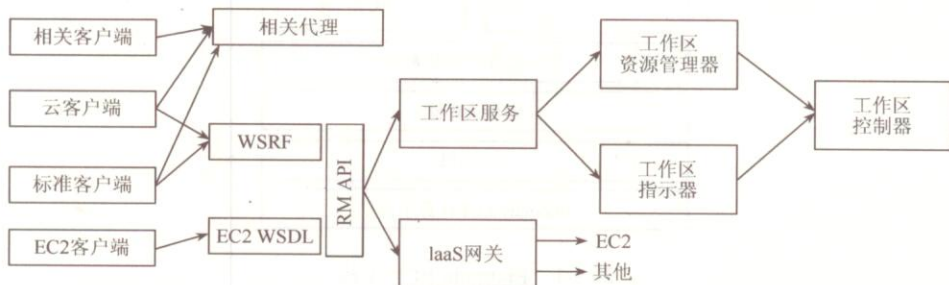


图 8-2 Nimbus 的架构图

8.1.6 Sector and Sphere

Yunhong Gu 等设计实现了 Sector and Sphere 云计算平台^[13]。Sector and Sphere 是用 C++ 编写的, 包括 Sector 和 Sphere 两个部分。Sector 是部署在广域网上的分布式存储系统, 它为了使系统有高可靠性和可用性而采用了自动的文件副本冗余方式, 已经用于 Sloan 数字巡天系统。Sphere 是建立在 Sector 之上的计算服务, 它为用户编写分布式密集型数据应用提供了简单的编程接口。

Sector 采用主/从服务器模式, 其架构如图 8-3 所示^[13]。安全服务器维护用户的账户、密码、文件访问信息和授权的从节点的 IP 地址; 主服务器维护存储在系统中的文件的元数据, 控制所有从节点的运行, 同时和安全服务器进行通信来验证从节点、客户服务器和用户; 从节点用来存储数据, 并对 Sector 客户端的请求进行处理。

Sphere 是以 Sector 为基础构建的计算云, 提供大规模数据的分布式处理。Sphere 的基本数据处理模型如图 8-4 所示^[13]。

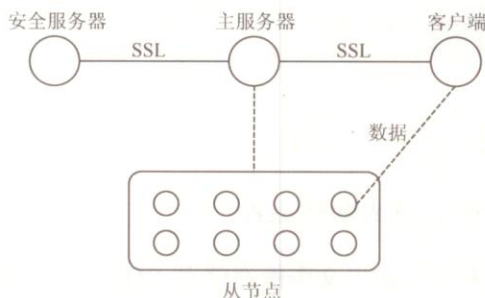


图 8-3 Sector 架构

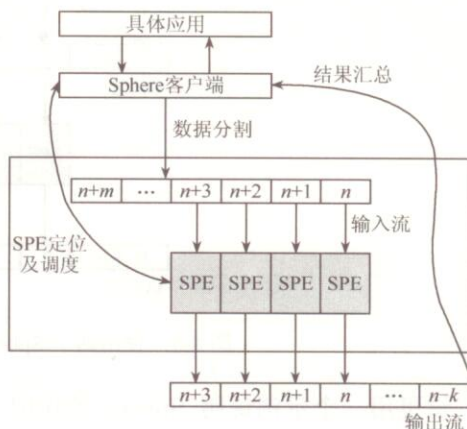


图 8-4 Sphere 的基本数据处理模型

针对不同的应用会有不同的数据, Sphere 统一地将它们以数据流的形式输入。为了便于大规模地并行计算, 首先需要对数据进行分割, 分割后的数据交给 SPE 执行。SPE 是 Sphere 处理引擎 (Sphere Processing Engine), 是 Sphere 的基本运算单元。除了进行数据处理外 SPE 还能起到负载均衡的作用, 因为一般情况下数据量远大于 SPE 数量, 当前负载较重的 SPE 能继续处理的数据就较少, 反之则较多, 如此就实现了系统的负载均衡。SPE 处理后的结果既可以作为最终结果以输出流形式输出, 也可以作为下一个处理过程的输入。Sphere 客户端为编写分布式应用程序的开发者提供了一系列的 API 包, 开发者可以使用这些 API 包来初始化输入流、加载处理函数库、启动 Sphere 进程和读取处理结果。具体流程如下:

- (1) 当主服务器接收到 Sphere 数据处理的客户端请求, 主服务器向客户端发送一个可用的从节点列表。
- (2) 客户端选择一些或者所有从节点, 让 SPE 在其上运行。
- (3) 客户端与 SPE 建立 UDT 连接。
- (4) 流处理函数被发送给每个 SPE, 并储存在从节点上。

(5) SPE 打开动态库并获得各种处理函数。

图 8-5^[13]展示了使用两个 Sphere 进程执行分布式排序的过程。第一阶段采用哈希函数扫描全部的数据流,把每个元素放置到相应的桶中。第二个阶段使用 SPE 对每个桶排序。在图中,首先将需要排序的数据进行散列(Hash)处理,将他们比较均匀地分布到有序的位置上,最好能使每个位置所包含的数据量大致相同。比如要排序的是数字,并且这些数字都在 500 之内,那么我们就可以将大于 300 的全部散列到位置 A,150~300 的散列到位置 B,剩下的散列到位置 C。这一步完成后再次利用 SPE 对各个位置进行排序,具体的排序方法可以自行选择。经过这两个过程后即可完成排序,因为此时将 A、B、C 中的数据依次输出就是一个有序数列。

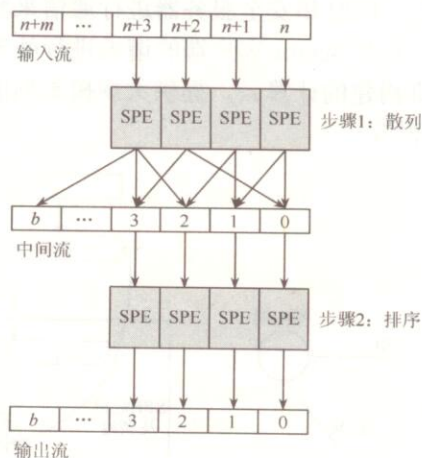


图 8-5 使用两个 Sphere 过程执行分布式排序的过程

这里用一个示例说明 Sphere 的作用。假设有十亿张天文图像存储在 SDSS 中,要从中找出褐矮星图片,用户需要写一个 findBrownDwarf 的函数从大量图片中找出需要的图片,在这个函数中,把所有图片作为输入,褐矮星图片作为输出。findBrownDwarf(input,output)标准的内部函数如下:

```
for each file F in(SDSS slices)
  for each image I in F
    findBrownDwarf(I,...);
```

使用 Sphere 客户端的 API,伪代码如下:

```
SphereStream sdss;
sdss.init(/*list of SDSS slices*/);
SphereProcess myproc;
myproc.run(sdss,"findBrownDwarf");
myproc.read(result);
```

其中, sdss 是存储 Sector 文件元数据的流数据结构。

通过使用 Sector 和 Sphere 能够将数据传输速度提升至 Hadoop 的两倍,速度提升的原因之一就是采用 UDT 协议,这一协议主要是针对极高速网络和大型数据集设计的。

8.1.7 abiquo

abiquo^[14]公司推出了一套比较完整的云计算解决方案，它可以帮助用户在各种复杂环境下高效地构建公有、私有或混合云。这套方案主要包括三个部分：abiCloud、abiNtense 和 abiData。三个部分可以单独使用，也可搭配起来使用。

abiCloud 是 abiquo 公司的最重要产品，是一款开源云管理软件，可以创建管理资源并且可以按需扩展。该工具能够以快速、简单和可扩展的方式创建和管理大型、复杂的 IT 基础设施（包括虚拟服务器、网络、应用和存储设备等）。abiCloud 较之同类其他产品的一个主要区别在于其强大的 Web 界面管理，用户可以通过拖曳一个虚拟机来部署一个新的服务。它允许通过 VirtualBox 部署实例，还支持 VMware、KVM 和 Xen 等不同虚拟机。

abiCloud 目前主要有三个版本：社区版（Community Version）、企业版（Enterprise Version）和 ISP 版（ISP Version，ISP 表示互联网服务提供商）。社区版向公众免费提供，企业版则在社区版基础上添加了一些高级特性，而 ISP 版通过扩展企业版的内容来允许 ISP 出售其云计算服务。

abiCloud 的基本构架如图 8-6 所示。

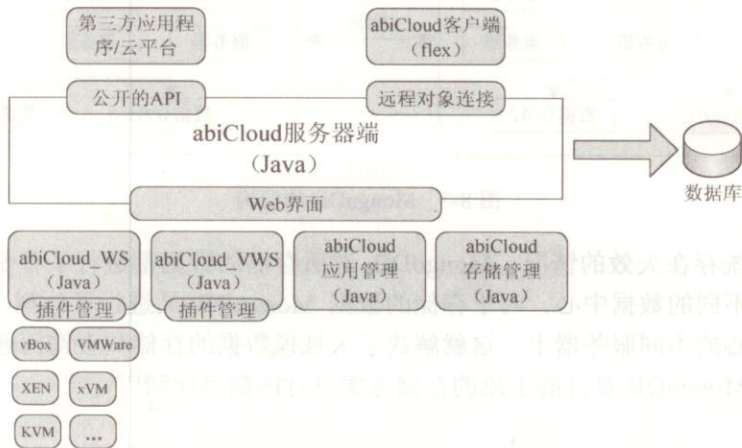


图 8-6 abiCloud 的基本构架

abiCloud 的基本构架清晰明了，其中，abiCloud_WS 是平台的虚拟工厂，主要负责管理各种虚拟化技术。abiCloud_VMS (abiCloud Virtual Monitor System) 用来监控虚拟化设备的运行状态。从图 8-6 中也可以看出，abiCloud 除了客户端采用了 flex 技术外，其他部分几乎都是由 Java 语言来实现的。和 abiCloud 相比，其他两个产品使用的并不是很多。abiNtense 通过使用基于网格的架构，有效地减少了大规模高性能计算的执行时间。abiData 由 Hadoop Common、HBase 和 Pig 开发而来，它是一个信息管理系统，可以用来搭建分析大量数据的应用，是一种低成本的云存储解决方案。

8.1.8 MongoDB

MongoDB^[16]是由 10gen^[17]公司支持的一项开源计划。10gen 云平台可用于创建私有

云，是类似于 Google App Engine 的一个软件栈，提供与 App Engine 类似的功能，但有一些不同之处，通过 10gen 可以使用 Python、JavaScript 和 Ruby 编程语言开发应用程序。该平台还使用沙盒概念隔离应用程序，并且使用自己的应用服务器在 Linux 上构建可靠的环境。

MongoDB 的目标是构建一个基于分布式文件存储系统的数据库，由 C++ 语言编写。MongoDB 易于部署、管理和使用，主要设计目标是高性能、可扩展和适当的功能。MongoDB 主要有以下几个特性。

- (1) 易存储对象类型的数据。
- (2) 高性能，特别适合“高容量、值较低”的数据类型。
- (3) 支持动态查询。
- (4) 支持复制和故障恢复。
- (5) 自动处理碎片以支持云计算层次上的扩展性。
- (6) 使用高效的二进制数据存储方式，可以存储包括视频在内的大型数据。

MongoDB 的架构如图 8-7 所示^[17]。

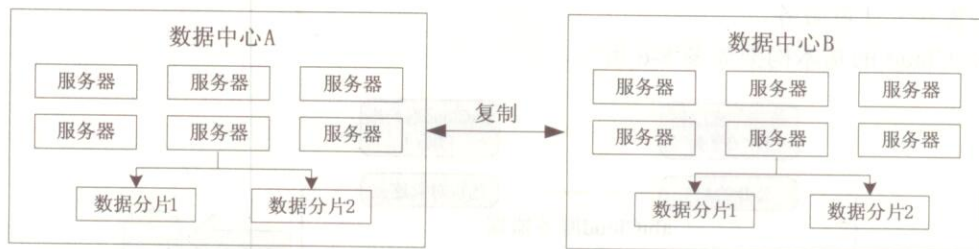


图 8-7 MongoDB 的架构

考虑到系统存在失效的情况，MongoDB 对所存储的数据都进行了备份，将不同的数据副本存储在不同的数据中心。对于存储的数据 MongoDB 又进行了分割，将不同的分片存放在数据中心的服务器上，这就解决了大规模数据的存储和查询问题。

图 8-8 是 MongoDB 对目前主流的存储方案进行的简单比较^[16]。

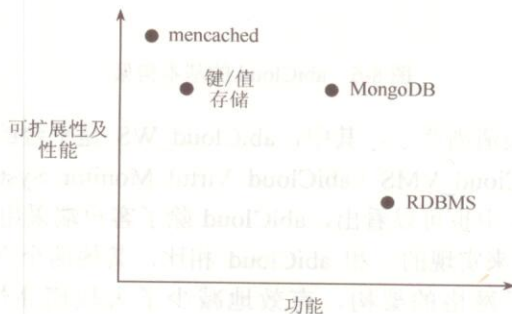


图 8-8 主流存储方案对比

从图中看出，MongoDB 的最大优势在于它的均衡性，MongoDB 可以在功能以及扩展性方面找到一个绝佳的平衡点。其他存储方式要么功能强大但扩展性和性能较差（RDBMS），要么可扩展很好但功能有限（memcached、键/值对方式存储）。

8.2 Cassandra

Cassandra 是一个高可靠的大规模分布式键值存储系统。Facebook 已经将 Cassandra 系统投入到电子邮件、搜索引擎等方面的实际应用中, 该项目 2009 年成为 Apache 的孵化项目。

8.2.1 体系结构

Cassandra 的体系结构充分吸收了 Dynamo 架构的 Consistent Hashing、Hinted Handoff 等技术以及 Bigtable 的 Storage layer、data mode 等内容。Cassandra 的体系结构^[3]共分为三层, 分别为核心层、中间层和顶层, 如图 8-9 所示。

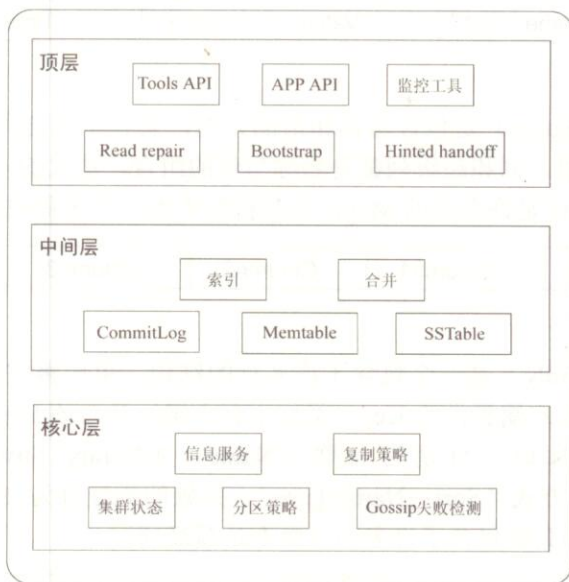


图 8-9 Cassandra 的体系结构

(1) 核心层。主要提供一些底层服务包括用户的信息服务、基于 DHT 的分区策略、复制策略及 Gossip 失败检测策略等。

(2) 中间层。主要融合了 Bigtable 存储系统的架构技术, 同样采用了 CommitLog 日志记录, Memtable/SStable 存储模型, 以及合并压缩 SStable 所使用的压缩技术。

(3) 顶层。主要提供 API, 监控工具, 以及一些针对一致性问题的策略。顶层的 API 主要分为应用 API (App API) 和工具 API (Tools API), 应用 API 主要包括 Cassandra API、Admin API 等; 工具 API 主要包括 Search API、System Loading API 等。顶层的 Bootstrap 指启动新节点的引导文件。针对系统应用的最终一致性问题, Cassandra 通过提示移交 (Hinted Handoff) 技术来解决节点 Down 掉以后的复制问题, 这也是实现最终一致性的优化措施。采取读修复技术 (Read Repair) 来解决读取一致性问题, 它是指在客户端读取某一个列的时候, 执行客户请求的存储节点会负责检查该列的各个备份是否一致, 如果不一致则修复。

8.2.2 数据模型

Cassandra 提供的数据模型支持对数据的布局 and 格式动态的控制。Cassandra 中的表是一个按照主键索引的分布式多维图，它的值是一个高度结构化的对象。在 Cassandra 中，列的结构近似于 Bigtable 中的列结构。下面详细介绍 Cassandra 的数据模型^[3]。

1) 列

在 Cassandra 中，列（Column）是最小的数据单元，它是一个三元组，包含名称（Name）、值（Value）和时间戳（Timestamp）。将一个列用 JSON 的形式可以表示为 { Name: “ID”, Value: “92938493”, Timestamp: 2450016708 }。时间戳的作用是用来解决数据冲突，为了简单起见，可以忽略它，就把列想象成一个名称/值即可。在 Cassandra 中列总是有序的，通过其 Column-Name 进行排序。

Column	Name: “ID”	Value: “92938493”	Timestamp: 2450016708
--------	------------	-------------------	-----------------------

2) 超级列

超级列（SuperColumn）是包含名称和值的元组，它并不包含列中的时间戳。可以将超级列想象成列的数组。列和超级列都是名称与值的组合。最大的不同在于列的值是一个“String”，而超级列的值是许多列的 Map，即超级列中的一个名称映射出了多个列。

SuperColumn	Column1	Column2	Column3	ColumnN
-------------	---------	---------	---------	---------

3) 列族

列族（ColumnFamily）是一个包含了许多行的结构，可以将它想象成 RDBMS 中的表。每一行都包含有客户端提供的 Key（类似于行主键）及与该 Key 相关联的许多列。列族由三个元素作为标记，分别为名称（Name: Arbitrary String）、类型（Type: Simple/Super）和排序方式（Sort: Name/Time）。列族的类型分为 Standard 类型和 Super 类型。Standard 类型的列族包含了许多列（而不是超级列）。

ColumnFamily	Column1	Column2	Column3	ColumnN
--------------	---------	---------	---------	---------

Super 类型的列族包含了一系列的超级列，但是并不能像超级列那样包含一系列 Standard 列族。

ColumnFamily	SuperColumn 1	SuperColumn 2	SuperColumn N
--------------	---------------	---------------	---------------

4) 行

行（Row）以 key 为表示，一个 key 对应的数据可以分布在多个列族中。通常我们都只会存放在一个列族中。

Row	ColumnFamily 1	ColumnFamily 2	ColumnFamily N
-----	----------------	----------------	----------------

5) 键值空间

键值空间（Keyspace）是数据的最外层，键值空间是 Cassandra 哈希表的第一维，是列族的容器。所有的列族都属于某一个键值空间，一般说来，一个程序应用只会有一个键

值空间，相当于关系数据库中的表空间的概念。

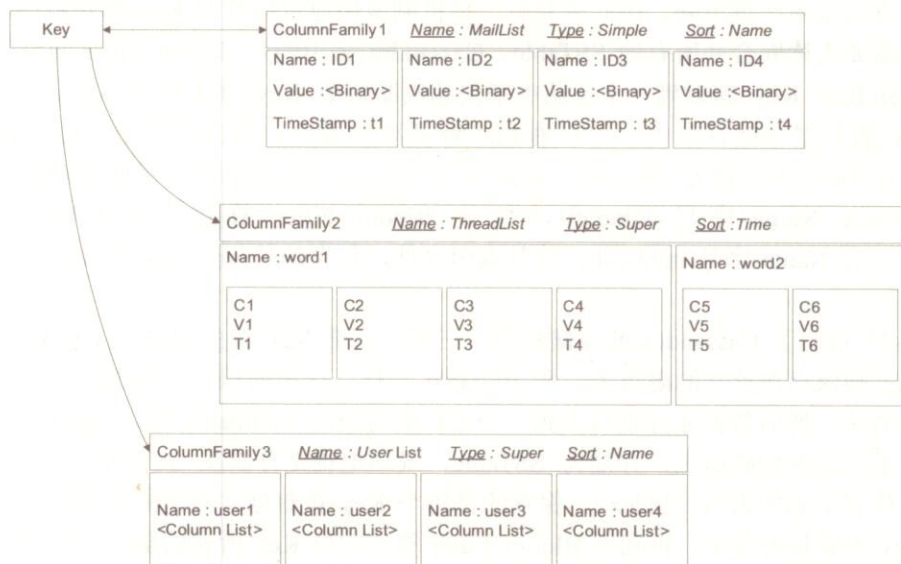


图 8-10 Cassandra 的数据模型

在每一个放置了列族的节点上，具有相同的 CF-Name（列族 ColumnFamily 简称为 CF）的列族会被保存在一起，称为一个 ColumnFamilyStore（CFS）。在一个 CFS 中，各个列族是按照 Row-Key 排序的。Cassandra 的数据模型中采用了两级索引方式来进行数据定位。第一级索引所用的 Key 为（Row-Key, CF-Name），即用一个 Row-Key 和 CF-Name 可以定位一个列族，图 8-10 所示的 Cassandra 的数据模型中通过（Key, MailList）就可以定位到 ColumnFamily1。第二级索引所用的 Key 为 Column-Name，即通过一个 Column-Name 可以在一个列族中定位一个列。总结来说，在每个节点本地看来，一个 CFS 相当于数据库的一个表，列族相当于表中的一行，列相当于一行中的一个域。

8.2.3 存储机制

Cassandra 的存储机制充分借鉴了 Bigtable 的设计框架，采用 Memtable/SSTable 的存储方式，而且对于 Cassandra 来说只有顺序写，没有随机写^[3]。

与关系数据库一样，Cassandra 在写数据之前，也需要先记录再提交日志 Commitlog，然后数据才会写入到列族对应的 Memtable 中，并且 Memtable 中的内容则按照 Key 排序好。Memtable 满足一定条件后批量刷新到磁盘上，存储为 SSTable，SSTable 一旦完成写入操作，就不可变更而只能读取。下一次 Memtable 需要刷新到一个新的 SSTable 文件中。采用这种机制就相当于缓存写回机制（Write-back Cache），优势在于将随机 IO 写变成顺序 IO 写，降低了大量的写操作对于存储系统所带来的压力。

为了避免大量 SSTable 带来的性能影响，Cassandra 也提供一种类似于 Bigtable 中定期将多个 SSTable 合并压缩成一个新的 SSTable 的机制，因为每个 SSTable 中的 Key 都是已经排序好的，因此 Cassandra 只需要做一次合并排序就可以完成该任务。同时，由于 SSTable 数据不可更新，可能导致同一个列族中的数据存储在多个 SSTable 中，这时查询

数据时, 需要去合并读取列族所有的 SSTable 和 Memtable, 当列族中的数量很大的时候, 可能导致查询效率严重下降。因此需要有一种机制能快速定位查询 Key 落在哪些 SSTable 中, 而不需要去读取合并所有的 SSTable。Cassandra 采用的是 Bloom Filter 算法^[3], 通过多个哈希函数将 Key 映射到一个位图中来快速判断这个 Key 属于哪个 SSTable。因此在 Cassandra 的数据存储目录中, 可以看到三种类型的文件, 格式类似于: ColumnFamily Name-序号-Data.db, 这是 SSTable 数据文件, 按照 Key 排序后存储键/值字符串。ColumnFamily Name-序号-Filter.db, 这是 Bloom Filter 算法生产的映射文件。ColumnFamily Name-序号-index.db, 这是索引文件, 保存的是每个 Key 在数据文件中的偏移位置。

图 8-11 说明了 Cassandra 的存储机制, 三个列族的 Key 值先记录在 Commitlog 中, Commitlog 则保存在独立的磁盘上。和 Bigtable 一样, 这里的日志内容也同样需要按照键值进行序列化, 然后数据才分别写入到三个 CF 所对应的 Memtable 中。Memtable 满足一定条件后批量刷新到磁盘上, 存储为 SSTable, 确切地说是存储在 SSTable 的块上并设置了保存块位置信息的索引 (Index), 保存的是每个 Key 在数据文件中的偏移位置。当查找时将 Index 加载到内存中, 再利用 Bloom Filter 算法定位 Key 所属的块, 如此就可以快速进行查找。

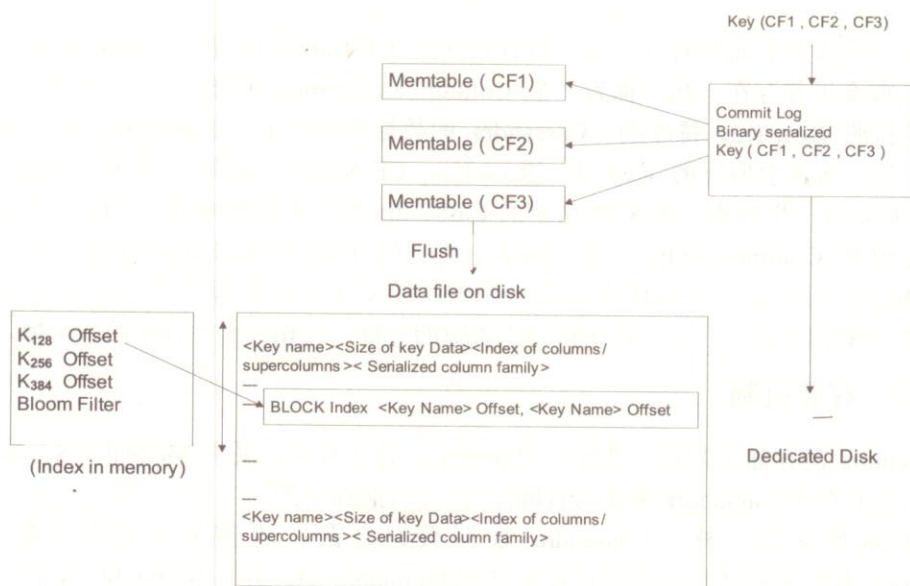


图 8-11 Cassandra 的存储机制

8.2.4 读/写删过程

Cassandra 设计的初衷是运行在集群环境下, 为了确保单点故障不会导致整个集群不可用, Cassandra 采取了节点数据复制策略。同时 Cassandra 系统要求快速、高效地处理大量的写操作, 同时不会牺牲读操作的效率。下面具体介绍 Cassandra 完成写、读以及删除的过程。

1. Cassandra 写入过程

- (1) 客户端向 Cassandra 集群中单一随机节点发出写请求;
 - (2) 此节点将作为代理节点, 并根据复制放置策略 (Replication Placement Strategy) 将写请求发送到 N 个不同节点;
 - (3) 这 N 个节点以 “RowMutation” 消息的形式接收到此写请求, 节点会执行以下两个操作, 一是消息追加到 CommitLog 中以满足事务性目的; 二是将数据写入到 Memtable, 当 Memtable 结构数据满的时候需要刷新到 SSTable。同时每个给定列族的一组临时的 SSTable 会被合并到一个大的 SSTable, 临时的 SSTable 会被当做垃圾回收掉;
 - (4) 代理节点必须等待这 N 个不同节点中的某些节点写响应的返回, 才能将写操作成功的消息告诉客户端 (根据写一致性水平来确定需要等待写成功响应的节点个数)。
- Cassandra 的写一致性水平 (假设副本个数: n) 分为以下三种情况:

- (1) ONE: 确保写入到至少一个节点中的 Commitlog 和 Memtable;
- (2) QUORUM: 确保至少写入到 $n/2+1$ 个节点上;
- (3) ALL: 确保写入到 n 个节点上。

为了确保单点故障不会导致整个集群不可用, Cassandra 采取了节点数据复制策略^[3]。Cassandra 为客户端提供了很多可供选择的复制策略, 比如 “架构不可知” (RackUnaware), “架构可知” (RackAware) (同一数据中心内) 和 “数据中心可知” (DatacenterAware)。应用程序可以根据需要选择数据复制策略。Cassandra 的写具有以下几个特性^[18]:

- (1) 关键路径上无任何锁;
- (2) 表现出类似于写入式缓存 (Write through cache), 快速高效;
- (3) 顺序磁盘访问;
- (4) 只有 Append 操作, 没有额外的读开销;
- (5) 即使出现节点故障时也都总是可写;
- (6) 只保证基于列族的原子性。

2. Cassandra 读取过程

- (1) 客户端发送一个读请求到 Cassandra 集群中的单一随机节点 (即存储代理节点 Storage Proxy);
 - (2) 该节点根据复制放置策略将读请求发送到 N 个不同节点;
 - (3) 收到读请求的节点都要合并读取 SSTable 和 Memtable, 由于在内存中进行操作, 数据量也相对较小, 因此从 Memtable 中读取数据相对简单而且循环查找很快。当扫描 SSTable 时, Cassandra 使用一个更低级别的列索引与布隆过滤器 (Bloom Filter) 来查找数据块。通过 Bf 确定待查找 Key 所在的 SSTable, 以及 Idx 确定 Key 在 SSTable 中的偏移位置。
 - (4) 代理节点必须等待这 N 个不同节点中的某些节点读响应的返回, 才能将读操作成功的消息告诉客户端 (根据读一致性水平来确定需要等待读成功响应的节点个数)。
- Cassandra 的读一致性水平 (假设副本个数为 n) 分为以下三种情况^[3]。
- (a) ONE: 返回第一个响应的节点上面的数据, 但不保证数据是最新的, 通过读修复和一致性检查可保证后续的调用能够读取最新的数据。
 - (b) QUORUM: 查询 n 个节点, 返回至少 $n/2+1$ 个节点上的最新数据。
 - (c) ALL: 查询 n 个节点, 返回 n 个节点中的最新数据, 一个节点失效将导致读失败。

3. Cassandra 删除过程

分布式数据库在删除方面存在以下的问题：如果客户端执行一个删除操作，并且有一个副本还没有收到这个删除操作，这个时候这个副本依然是可用的。此外，该节点还认为那些已经执行删除操作的节点丢失了一个更新操作，它还要去修复这些节点，也就是说一个删除操作不可能一次性将数据立即删除掉。

在 Cassandra 系统中，它不会直接去删除数据，Cassandra 使用了一个删除标记称为墓碑^[19] (Tombstone)。这个墓碑可以被传播到那些丢失了初始删除请求的节点，让每一个节点跟踪本地墓碑值的年龄，同时 Cassandra 定义了一个常量 GCGraceSeconds，默认值为 10 天。一旦墓碑值的年龄超过这个常量，它将在进行合并压缩的时候被收集。这意味着如果有一个节点宕机时间比 GCGraceSeconds 更长的话，就把它作为一个失败的节点。对于 GCGraceSeconds 的初始设置，若进行了 Anti Entropy (逆熵) 配置，可以减少 GCGraceSeconds 的初始值。当然，如果只运行单一 Cassandra 节点，可以把它减少到零。

Cassandra 真正删除数据的过程是：当客户端从 Cassandra 中读取数据的时候，节点在返回数据之前都会主动检查是否该数据被设置了删除标记，并且该删除标志的添加时长已经大于 GCGraceSeconds，则要先删除该节点的数据再返回。

8.3 Hive

Hive 搭建在 Hadoop 之上，这使用户编写 Map/Reduce 程序更加方便，同时让用户可以像使用关系数据库那样使用 HDFS。作为 Hadoop 的一个主要子项目，Hive 在不断的不断发展之中，鉴于篇幅有限，本节将从整体构架、数据模型、使用语言等几个方面对 Hive 做简要介绍。

8.3.1 整体构架

Hive 的整体架构和主要组件，以及 Hive 组件和 Hadoop 之间的关系如图 8-12 所示。从图中可以看出，Hive 包括以下几个组成部分^[20]。

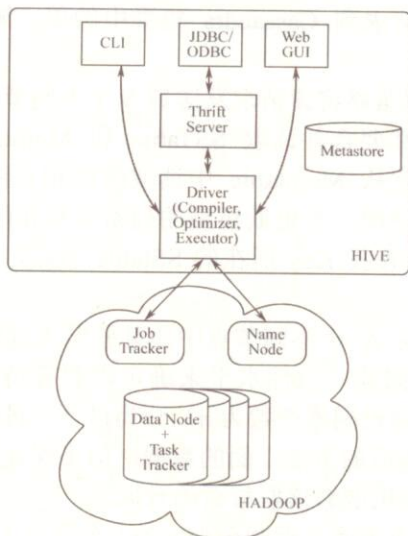


图 8-12 Hive 的整体架构和主要组件

(1) 用户界面 (UI): 用户通过用户界面提交查询及其他操作。当前用户界面包括一个命令行界面 (CLI), 以及一个正在开发的基于 Web 的图形用户界面 (WGUI), WGUI 可以使用户摆脱繁琐的命令行界面, 通过 Web 来访问 Hive。用户界面还包括一个客户端模式 (Client), 当使用客户端模式时需要指出 Hive Server 所在的节点, 并且在这个节点启用 Hive Server。

(2) 驱动器 (Driver): 驱动器用来接收用户的查询请求, 这个组件实现了会话控制 (Session Handles) 的概念, 同时运行和获取模仿 JDBC/ODBC 接口的 API。

(3) 编译器 (Compiler)。编译器用来解析查询。通过在不同查询块和查询表达上的语义分析, 最终生成一个基于表和分区元数据的执行计划。

(4) 元数据存储 (Metastore): 这个组件存储数据仓库中所有不同表和分区的信息, 包括列和列类型信息。

(5) 执行引擎 (Execution Engine): 执行引擎执行由编译器制订的计划。这个计划是关于阶段的有向无环图。执行引擎管理着不同阶段之间的联系, 当需要执行这些阶段时, 执行引擎会将他们放到合适的组件上执行。

图 8-13 展示了一个典型的查询流程^[20]。必须说明的是, 当读写存储数据的 HDFS 文件时需要使用序列化器和反序列化器。首先, 用户通过用户界面调用连接到驱动器的执行接口; 随后在步骤 2 中, 驱动器会为此查询建立一个会话控制, 同时将查询送到编译器以便生成一个执行计划。在步骤 3 和步骤 4 中, 收到查询后编译器从元数据存储取得必要的元数据。这些元数据对查询中的表达式进行类型检测及根据查询谓词进行分区修剪。当取得必要的元数据后, 由编译器生成执行计划。这个计划是一个关于阶段的有向无环图 (DAG), 其中每一个阶段都是一个 Map/Reduce 任务、元数据操作或是在 HDFS 上的操作。对于 Map/Reduce 阶段, 这个计划包含了多个 Map 操作树 (operator tree) (其在 Mapper 上执行) 和一个 Reduce 操作树 (在 Reducer 上执行)。在步骤 6.1~6.3 中, 针对不同的阶段, 执行引擎将其送到相应的组件执行。在 Mapper/Reducer 上的每一个任务中, HDFS 文件中的数据通过与表或与中间输出结果相关的反序列化器读取出来, 这些数据都通过了相关的操作树。一旦生成了输出数据, 输出的结果将通过序列化器写入到一个临时的 HDFS 文件中 (这个过程发生在 Mapper 中, 以防止运算不需要进行 Reduce), 临时文件为随后的 Map/Reduce 阶段提供了数据。对于 DML 操作, 最后的临时文件将会移动到表的存储位置。这样做是为了防止“脏数据”被读取 (在 HDFS 中, 文件的重命名是一个原子操作)。由步骤 7~9 可以看出, 对于查询, 临时文件的内容是由执行引擎直接从 HDFS 读出来, 这作为驱动器请求的一部分。

8.3.2 数据模型

Hive 中的数据模型主要有以下 3 种^[20]: 表 (Table)、分区 (Partitions)、桶 (Buckets)。

(1) 表 (Table)。Hive 中的表和关系数据库中的表的概念是相似的。表能被过滤、投影、连接和合并。除此之外, 表中的所有数据被存储在 HDFS 的一个目录中。同时 Hive 也支持外表 (External Table) 的概念。对于外表, 只要为创建表的 DDL 提供一个合适的存储位置, 表就可以创建在已经存在的文件或目录之中。在表中, 数据是以典型的列的形式组织的, 这和关系数据库中的概念相似。

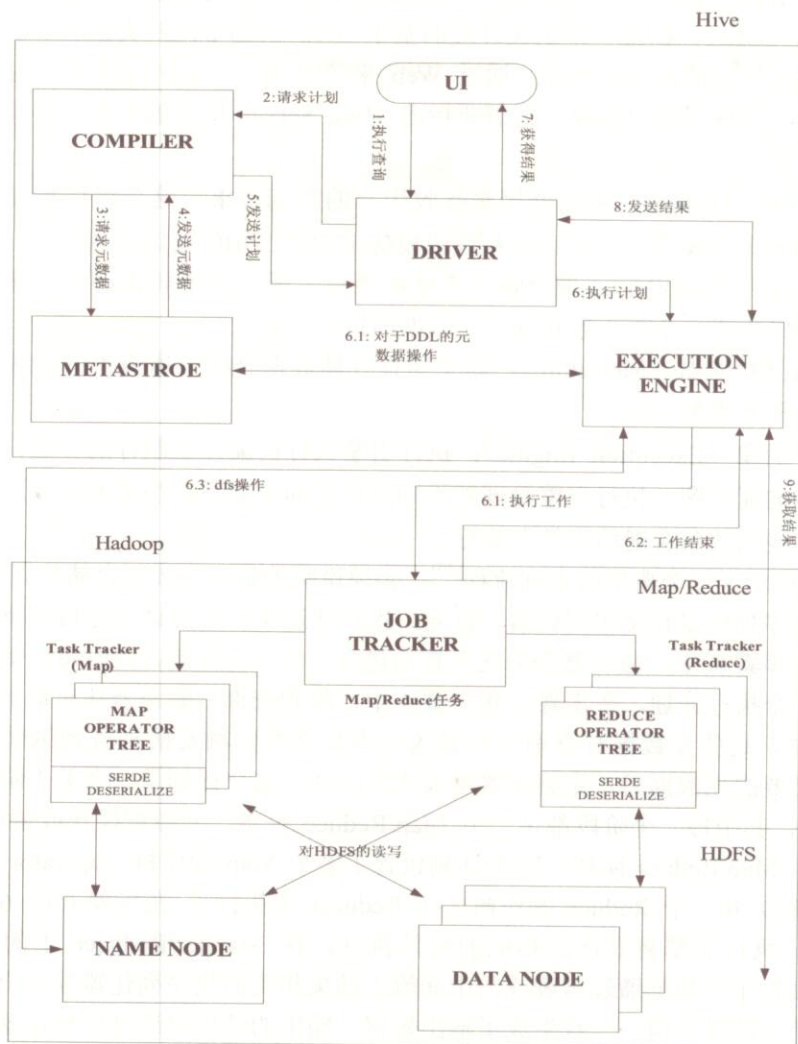


图 8-13 查询流程图

(2) 分区 (Partition)。每一个表可以有一个或几个分区 (Partition) 键，键值决定了数据是怎样存储的。例如有一个表 Table1，在 Table1 中有以日期分割的列 Date1。Table1 在 HDFS 中是分布存储在几个文件中的。这些文件中的数据是独立的日期数据，而文件存储在 <table location>/date1=<date> 目录中。Partition 准许系统修剪数据，以便根据查询谓词进行数据检测。

(3) 桶 (Bucket)。根据表中列的 Hash 值，每一个分区 (Partition) 中的数据都可以被分割成几部分存储到几个 Bucket 中。每一个 Bucket 以文件的形式存储在目录中。数据存储到 Bucket 中使系统能有效地评估依据一个数据样本的查询。

表和外表有以下区别：表的创建过程和数据加载过程可以在同一个语句中完成。当加载数据时，实际产生的数据会被存储到数据仓库的目录中，之后对于数据的所有操作都会在目录中进行。当删除表时，表中的数据和元数据是一同被删除的。而外表只有一个过

程，即表的创建和加载是同时完成的。外表中，真正的数据是存储在（CREATE EXTERNAL TABLE...LOCATION）的 LOCATION 之后指定的 HDFS 路径中，而不是在数据仓库的目录之中。当删除外表时，并不删除实际的数据，只删除相应的元数据。

除了基本的列数据类型（整型、单精度浮点型、字符串、日期和布尔型）以外，Hive 也支持数组和图。此外，用户可以在任何的简单数据类型、集合或其他用户定义的数据类型之上构建自己的数据类型。类型系统与 *serde*（序列化/反序列化）及目标检测接口有着密切的联系。用户通过使用目标接口可以创建自己的数据类型和自己的 *serdes*，用户可以使用自己的 *serdes* 对 HDFS 中的数据进行序列化或反序列化。当需要 Hive 支持其他的数据格式和更丰富的数据类型时，这两个接口提供了必要的钩子（Hook）去扩展 Hive 的能力。

8.3.3 HQL 语言

Hive 使用 HIVE QL 语言，HQL 类似于 SQL 语言，通过 HQL 语言，可以提供给使用者与传统 RDBMS（关系型数据模型系统）一样的表格查询特性和分布式存储计算特性。

1. DDL: Data Definition Language

创建表的示例如下：

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] table_name
    [(col_name data_type [COMMENT col_comment], ...)]
    [COMMENT table_comment]
    [PARTITIONED BY (col_name data_type
    [COMMENT col_comment], ...)]
    [CLUSTERED BY (col_name, col_name, ...)
    [SORTED BY (col_name [ASC|DESC], ...)]
    INTO num_buckets BUCKETS]
    [ROW FORMAT row_format]
    [STORED AS file_format]
    [LOCATION hdfs_path]
```

其中，[ROW FORMAT DELIMITED]关键字是用来设置创建的表在加载数据时，支持的列分隔符；[STORED AS file_format]关键字是用来设置加载数据的数据类型。Hive 本身支持的文件格式只有：Text File，Sequence File。如果文件数据是纯文本，可以使用 [STORED AS TEXTFILE]。如果数据需要压缩，使用 [STORED AS SEQUENCE]。通常情况，只要不需要保存序列化的对象，默认采用 [STORED AS TEXTFILE]。

而且，Hive 支持的字段类型并不多，可以简单地理解为数字类型和字符串类型，详细的类型有：TINYINT；SMALLINT；INT；BIGINT；BOOLEAN；FLOAT；BOUBLE；STRING。

Hive 总体来说可以分为三种不同类型的表。

（1）普通表。每个普通表，就对应一个表名对应的文件。创建一张普通的 Hive 表，HQL 如下：

```
CREATE TABLE test_1(id INT, name STRING, city STRING) SORTED BY TEXTFILE ROW FORMAT
DELIMITED'\t'
```

(2) 外部表。EXTERNAL 关键字可以让用户创建一个外部表,在建表的同时指定一个指向实际数据的路径 (LOCATION)。Hive 创建内部表时,会将数据移动到数据仓库指向的路径。创建外部表时,仅仅记录数据所在的路径,而不对数据的位置做任何改变。在删除表时,内部表的元数据和数据会一起被删除,而外部表只删除元数据,不删除数据。具体 HQL 如下:

```
CREATE EXTERNAL TABLE test_1(id INT, name STRING, city STRING) SORTED BY TEXTFILE
ROW FORMAT DELIMITED '\t' LOCATION 'hdfs://.../...'
```

(3) 分区表。分区表实际是一个文件夹,表名即文件夹名。每个分区,实际是表名这个文件夹下面的不同文件。有分区的表在创建时使用 PARTITIONED BY 语句。当一个表拥有一个或多个分区时,可以让每一个分区单独存在一个子目录下。而且,表和分区都可以对某个列进行 CLUSTERED BY 操作,将若干个列放入一个桶中。同时也可以利用 SORT BY 对数据进行排序,这样可以使特定的应用提高性能。具体 HQL 如下:

```
CREATE TABLE test_1(id INT, name STRING, city STRING) PARTITIONED BY (pt STRING) SORTED
BY TEXTFILE ROW FORMAT DELIMITED '\t'
```

2. DML: Data Manipulation Language

其中,最基本的是加载数据 (Load Data)。

Hive 不支持一条一条的用 insert 语句进行插入操作,也不支持 update 的操作。数据是以 load 的方式,加载到建立好的表中。数据一旦导入,则不可修改。要么 drop 掉整个表,要么建立新的表,导入新的数据。基本的加载数据例子如下:

```
LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO TABLE tablename [PARTITION
(partcol1=val1, partcol2=val2 ...)]
```

Hive 在 load 数据这块,可以分为以下 4 种方式。

(1) Load data 到指定的表。直接将 file 加载到指定的表中,其中,表可以是普通表或者分区表。具体 HQL 如下:

```
LOAD DATA LOCAL INPATH '/home/admin/test/test.txt' OVERWRITE INTO TABLE test_1
```

其中,关键字 [OVERWRITE] 意思是覆盖原表里的数据,不写则不会覆盖;关键字 [LOCAL] 是指你加载文件的来源为本地文件,不写则为 hdfs 的文件。

(2) Load 到指定表的分区。直接将 file 加载到指定表的指定分区,其中表本身必须是分区表,如果是普通表,数据实际不会被导入。在 load 数据时, Hive 支持文件夹的方式,将文件夹内的所有文件,都 load 到指定表中。Hdfs 会将文件系统内的某文件夹路径内的文件,分散到不同的实际物理地址中。这样,在数据量很大的时候, Hive 支持读取多个文件载入,而不需要限定在唯一的文件中。具体 HQL 如下:

```
LOAD DATA LOCAL INPATH '/home/admin/test/test.txt' OVERWRITE INTO TABLE test_1
PARTITION (pt='xxxx')
```

(3) Insert+Select 方式。这个是完全不同于文件操作的数据导入方式。

标准的语法为:

```
INSERT OVERWRITE TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)]
select_statement1 FROM from_statement
```

多重插入为:

```
INSERT OVERWRITE TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)]
```



```
select_statement1[INSERT OVERWRITE TABLE tablename2 [PARTITION ...] select_statement2] ...
```

动态分区插入:

```
INSERT OVERWRITE TABLE tablename PARTITION (partcol1[=val1], partcol2[=val2] ...)
select_statement FROM from_statement
```

(4) Alter 表, 对分区操作。在对表结构进行修改的时候, 可以增加一个新的分区, 在增加新分区的同时, 将数据直接 load 到新的分区当中。具体 HQL 如下:

```
ALTER TABLE table_name ADD partition_spec [ LOCATION 'location1' ]partition_spec [ LOCATION
'location2' ] ...
```

3. Query (查询)

基本的有 Query - Join (连接)。Hive 只支持等值连接 (Equality Joins)、外连接 (Outer Joins) 和 Left Semi Joins。Hive 不支持所有非等值的连接, 因为非等值连接非常难转化到 Map/Reduce 任务。Join 的实现如图 8-14 所示。

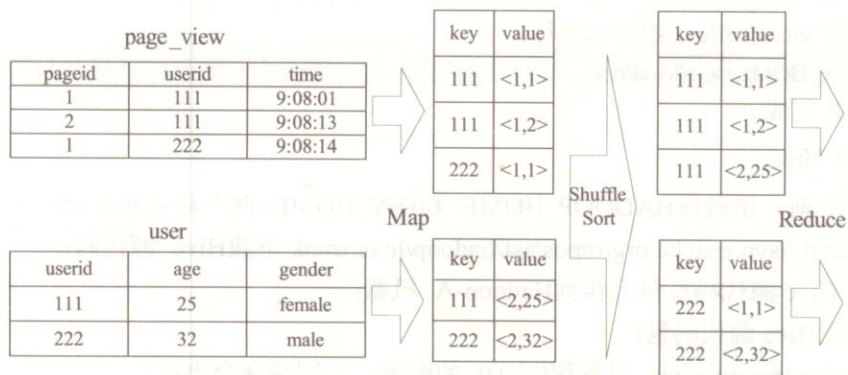


图 8-14 Join 的实现

另外 Hive 支持多于两个表的连接。其中多表 Join 时, 每次 Map/Reduce 任务的逻辑是这样的: Reducer 会缓存 Join 序列中除了最后一个表的所有表的记录, 再通过最后一个表将结果序列化到文件系统。这一实现有助于在 Reduce 端减少内存的使用量。实践中, 应该把最大的那个表写在最后 (否则会因为缓存浪费大量内存)。

- (1) Join LEFT/RIGHT OUTER: 输出左边/右边的每一行对应的结果。
- (2) Left Semi Join: 用于实现 a.key in select key from table b (即 in/exist 功能)。
- (3) 多个表的 Join key 是同一个时, Join 会被转化为单个 Map/Reduce 任务: Reduce 端会缓存 a 表和 b 表的记录, 然后每次取得一个 c 表的记录就计算一次 Join 结果。
- (4) 不同 Join key 时, 会被转化为多个 Map/Reduce 任务: 第一次缓存 a 表, 用 b 表序列化; 第二次缓存第一次 Map/Reduce 任务的结果, 然后用 c 表序列化。

8.3.4 环境搭建

Hadoop 0.20.0 中并没有集成二进制的 Hive, 所以需要通过源代码编译搭建环境。

1. 安装 ant

- (1) 从<http://ant.apache.org/>下载ant 二进制安装包，选择1.7.1 版本；
- (2) 将包apache-ant-1.7.1-bin.zip 上传到/usr/local 目录；
- (3) 在/usr/local 目录将 apache-ant-1.7.1-bin.zip 解压：unzip apache-ant-1.7.1-bin.zip；
- (4) 在/usr/local 目录为ant 建一个软连接：ln -s apache-ant-1.7.1 ant；
- (5) 修改/etc/profile，增加如下行： export PATH=/usr/local/ant/bin:\$PATH。

2. 安装 ivy

- (1) 从<http://www.apache.org/dist/ant/ivy/>下载ivy 二进制安装包，选择2.1.0-rc2 版本；
- (2) 将包apache-ivy-2.1.0-rc2-bin.tar.gz 上传到/usr/local 目录；
- (3) 在/usr/local 目录将 apache-ivy-2.1.0-rc2-bin.tar.gz 解压：

```
tar xzf apache-ivy-2.1.0-rc2-bin.tar.gz;
```

- (4) 在/usr/local 目录为ivy 建一个软连接：ln -s apache-ivy-2.1.0-rc2 ivy；
- (5) 修改/etc/profile，增加如下行：

```
export IVY_HOME=/usr/local/ivy
```

至此 ivy 安装完成。

3. 编译 Hive

编译Hive 前，请确保HADOOP_HOME 和IVY_HOME 两个环境变量已经生效。

- (1) 从<http://svn.apache.org/repos/asf/hadoop/hive/trunk> 下载Hive 源代码；
- (2) 将 Hive 源代码打包上传到Hadoop-A 机器；
- (3) 解压Hive 源代码包；
- (4) 修改shims/ivy.xml：只保留0.20.0 的配置，否则编译会出错；
- (5) 运行ant 开始编译：

```
ant -Dtarget.dir=/usr/local/hadoop/hive -Dhadoop.version=0.20.0 package
```

编译后，Hive 会被安装到/usr/local/hadoop/hive 目录下；

- (6) 添加Hive 环境变量，在/etc/profile 文件中增加如下两行：

```
export HIVE_HOME=/usr/local/hadoop/hiveexport PATH=$HIVE_HOME/bin:$PATH
```

8.4 VoltDB

VoltDB 不仅支持传统数据库的 ACID 模型，提供 SQL 接口，而且具有极高的处理速度和近乎线性的扩展能力，在运算效率上具有很大的优势，给用户提供了一个新的选择。

8.4.1 整体架构

VoltDB 数据库采用标准的“客户端-服务器”模型，数据库运行在一个或多个服务器上，调用程序则作为客户端运行。这些服务器组成一个集群来管理数据和内部的分配工作。VoltDB 的目标是实现最好的，最具扩展性的 OLTP 应用。为了实现这一目标，在实际环境中通常让 VoltDB 数据库服务端运行在一个节点集群上，而一个或多个的客户端应

用则运行在不同的计算机上,如图 8-15 所示。

在一个节点上启动和停止 VoltDB 数据库服务端是一件很简单的操作。然而,随着集群中节点数目的增加,手动管理这些数据库服务端就会变得十分乏味和低效。为此,VoltDB 提供了一个管理控制台去简化这个事情。

VoltDB 企业管理器是一个管理工具,使用它能够简化 VoltDB 数据库的管理和有关操作。VoltDB 企业管理器由服务端软件和一个基于 Web 的管理控制台组成,数据库管理员可以通过这个管理控制台来发出命令和评估集群的状态。企业管理器和控制台接口在整个系统中的地位如图 8-16 所示。

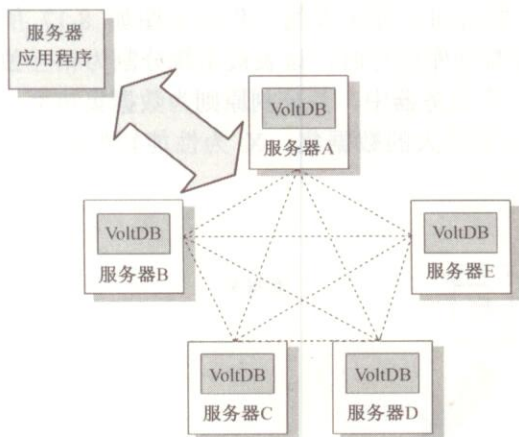


图 8-15 VoltDB 架构模型^[20]

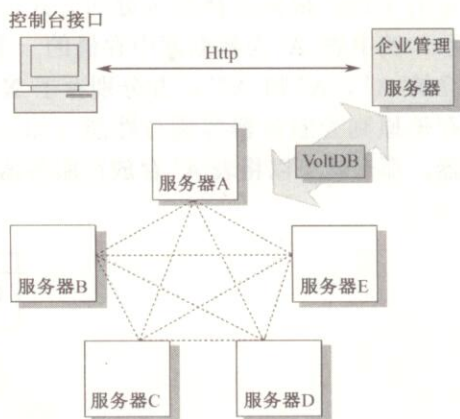


图 8-16 企业管理器的位置^[20]

当运行 VoltDB 企业管理器时,它会创建自己的专用 Web 服务器,管理员就可以从任何一个浏览器访问这个 Web 服务器。通过 Web 接口控制台,管理员可以随时随地地管理 VoltDB 集群。此外,有了 VoltDB 企业管理器,管理员就不需要在每个节点预安装 VoltDB,管理控制台会向集群中的节点分发 VoltDB 软件和数据库目录。

控制台还为以下功能提供了一个统一的接口:

- (1) 创建和收集快照。
- (2) 提供数据库卷和性能的实时统计。
- (3) 比较和更新数据库的目录。
- (4) 向一个高可用性的集群中删除和加入节点。

除了一个基于 Web 的图形界面,企业管理器还包含一个 Rest 编程接口。通过这个接口,数据库管理员可以使用脚本进一步定制上述功能和简化数据库的管理任务。

8.4.2 自动数据分片技术

VoltDB 作为一款新兴的数据库产品,有着其自身特有的技术特点,但由于篇幅所限,本节仅对 VoltDB 的自动数据分片技术做一定的分析。

1. 自动数据分片技术

自动数据分片技术不同于传统数据库,它实现了将整体数据自动分摊到多个存储设备

上, 这样每个存储设备的数据量相对就会小很多, 以满足系统的高性能需求。VoltDB 在运行时, 会通过分析和预编译存储过程中的数据访问逻辑, 使其中与之相关集群的每个节点都可以自主处理和分发数据。这样, 群集的每个节点都包含一个独特的“数据片”, 并提供数据处理能力。

VoltDB 引入了“分区表”和“表复制”的概念, 并利用“串行单线程处理”的方法, 以解决自动数据分片工作^[20]。

2. 分区表

分区表是 VoltDB 提出的一个新的概念, 在数据库的存储访问过程中, 它基于一个给定的主键, 根据选择分区键的方式, 以匹配数据访问。分区表的工作示意图如 8-17 所示, 其中表 A 为数据库中存储的一个表单, 在数据库运行时, 该表被系统分割为相互独立的 A', A'' 和 A''', 并分别存于 X、Y、Z 三个服务器中, 其分割原则为数据集热度, 存储原则为服务器容量与性能 (如: A' 为访问量较大的数据集, X 为性能较好的服务器, 那么就可以将表 A' 存放在服务器 X 中)。

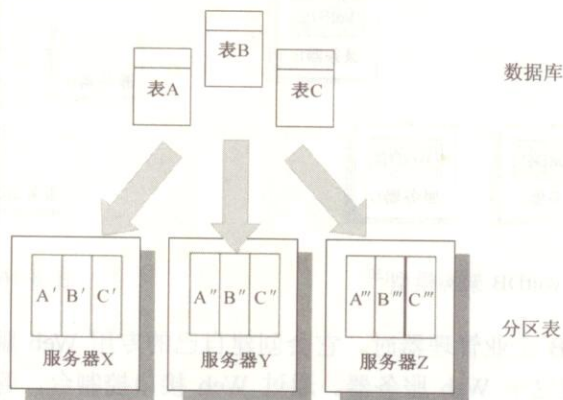


图 8-17 VoltDB 分区表示意图

3. 表复制

表复制是在分区表的基础上提出的, 其目的是进一步优化系统性能。VoltDB 允许将数据库中的某些表复制到群集的所有分区上, 表复制主要针对的是一些只读的小表, 用以和关联的大表创建连接, 而不需频繁地在各节点间进行检索与通信。表复制的工作示意图如 8-18 所示, 其中表 D 为数据库中与表 A、B、C 都关联的小表, VoltDB 会将其复制到每个服务器上, 并与服务器上被分割的大表创建联接, 表 D 在服务器上的作用为索引, 基本不会被修改, 仅用作查询功能。这样做的好处是显而易见的, 表 D 占用的空间不大, 但却能做到索引的作用, 虽然每个服务器都浪费了一点存储空间用以保存重复的表 D, 但这样做却大大节约了数据库运行状态下各节点间的通信量, 每个节点都可以根据自身的索引表处理数据, 而不需要访问别的服务器中的相关表, 这使整个数据库系统的查询能力大大提升。

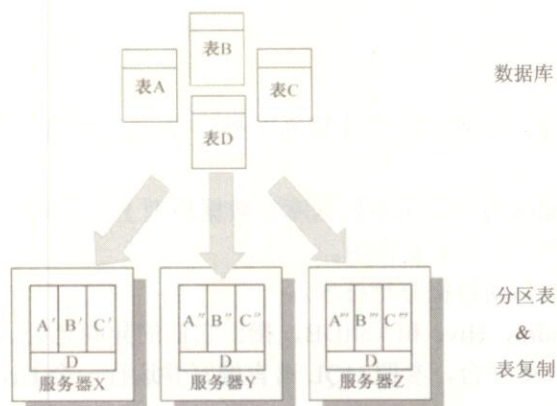


图 8-18 VoltDB 表复制示意图

4. 串行单线程处理

在 VoltDB 数据库运行状态下，系统通过“分区表”和“表复制”将数据库中的所有表单都散存于各服务器之上，并使用“串行单线程处理”方式，即：访问队列依次进入，各节点分布式，序列化运算，以达到性能最优的指标，其工作示意图如 8-19 所示。访问请求按队列的形式进入系统，并自动分配给每个服务器，服务器分别处理其内部对数据的查询、添加、删除、或更新操作，采用分布式与序列化的处理方法，使 VoltDB 数据库的访问效率进一步提升。

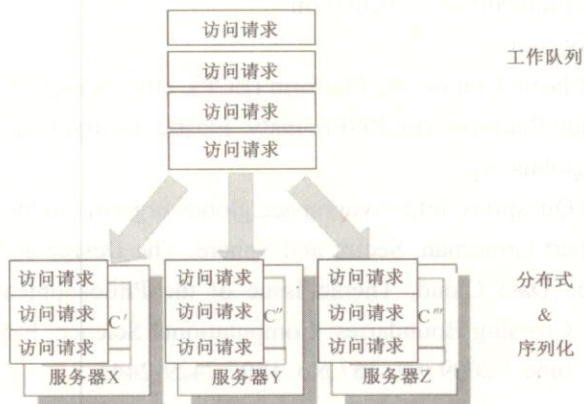


图 8-19 VoltDB 串行单线程处理示意图

以上对 VoltDB 的架构和自动数据分片技术进行了简要分析，当然 VoltDB 还包含很多新型技术，自动数据分片技术仅为关键技术之一，还有自动快照技术（用以在微秒级别完成一个事务，且事务内部不需要有 I/O 操作）和异步事务提交技术（用以解决对信息访问和提交的控制，增强系统的稳定性）等^[22]，限于篇幅，本书不展开介绍，有兴趣的读者可以进一步学习 VoltDB 的文献资料。

习题

1. 除了本章介绍的几种开源云计算系统外, 请再列举出几个其他的开源云计算系统。
2. 试比较 Cassandra 与传统关系数据库在数据模型上的异同。
3. 简述 Hive 各个组件的主要作用。
4. 简述 VoltDB 的自动数据分片技术。
5. 试比较 Cassandra、Hive 和 VoltDB, 指出它们的异同之处。
6. 试自行搭建 Hive 平台, 参照 HQL 语言编写并运行一个 Hive 实例。

参考文献

- [1] <http://cassandra.apache.org/>
- [2] <http://wiki.apache.org/cassandra>
- [3] <http://perspectives.mvdirona.com/2009/02/07/FacebookCassandraArchitectureAndDesign.aspx>
- [4] <http://wiki.apache.org/hadoop/Hive>
- [5] <http://wiki.apache.org/hadoop/Hive/Roadmap>
- [6] <http://www.im286.com/thread-6788602-1-1.html>
- [7] <http://voltdb.com/voltdb-launches-next-generation-open-source-oltp-dbms>
- [8] <http://community.voltdb.com/documentation>
- [9] <http://www.enomaly.com/>
- [10] libvirt. Enomaly Elastic Computing Platform (ECP). [http://wiki.libvirt.org/page/Enomaly_Elastic_Computing_Platform_\(ECP\)#Enomaly_Elastic_Computing_Platform_.28ECP.29](http://wiki.libvirt.org/page/Enomaly_Elastic_Computing_Platform_(ECP)#Enomaly_Elastic_Computing_Platform_.28ECP.29)
- [11] <http://workspace.globus.org/>
- [12] Frequently Asked Questions. <http://workspace.globus.org/vm/faq.html>
- [13] Yunhong Gu, Robert Grossman, Sector and Sphere: The Design and Implementation of a High Performance Data Cloud, Theme Issue of the Philosophical Transactions of the Royal Society A: Crossing Boundaries: Computational Science, E-Science and Global E-Infrastructure, 28 June 2009 vol. 367 No. 1897 2429-2445
- [14] <http://www.abiquo.com/>
- [15] abicloud technical overview, 2009.
<http://www.slideshare.net/abiquodocs/abicloud-technical-overview>
- [16] <http://www.mongodb.org/display/DOCS/Home>
- [17] <http://www.10gen.com/>
- [18] <http://www.slideshare.net/jbellis/cassandra-open-source-bigtable-dynamo>
- [19] <http://www.odbms.org/download/cassandra.pdf>
- [20] <http://wiki.apache.org/hadoop/Hive/Design>
- [21] <http://binotes.net/>
- [22] <http://dba.sky-mobi.com/?p=774>

第9章 云计算仿真器 CloudSim

CloudSim^[1,2]是澳大利亚墨尔本大学 Rajkumar Buyya 教授领导团队开发的云计算仿真器，它的首要目标是在云基础设施（软件、硬件、服务）上，对不同应用和服务模型的调度和分配策略的性能进行量化和比较，达到控制使用云计算资源的目的。基于云计算仿真器，用户能够反复测试自己的服务，在部署服务之前调节性能瓶颈，既节约了大量资金，也给用户的开发工作带来了极大的便利。

9.1 CloudSim 简介

为基于互联网的应用服务提供可靠、安全、容错、可持续、可扩展的基础设施，是云计算的主要任务。由于不同的应用可能存在不同的组成、配置和部署需求，云端基础设施（包括硬件、软件和服务）上的应用及服务模型的负载、能源性能（能耗和散热）和系统规模都在不断地发生变化，因此，如何量化这些应用和服务模型的性能（调度和分配策略）成为一个极富挑战性的问题。为了简化问题，墨尔本大学的研究小组提出了云计算仿真器 CloudSim。CloudSim 是一个通用、可扩展的新型仿真框架，支持无缝建模和模拟，并能进行云计算基础设施和管理服务的实验。

这个仿真框架有如下几个特性：

- (1) 支持在单个物理计算节点上进行大规模云计算基础设施的仿真和实例化。
- (2) 提供一个独立的平台，供数据中心、服务代理、调度和分配策略进行建模。
- (3) 提供虚拟化引擎，可在一个数据中心节点创建和管理多个独立、协同的虚拟化服务。
- (4) 可以在共享空间和共享时间的处理核心分配策略之间灵活地切换虚拟化服务。

CloudSim 方便用户在组成、配置和部署软件前评估和模拟软件，减少云计算环境下，访问基础设施产生的资金耗费。基于仿真的方法使用户可在一个可控的环境内免费地反复测试他们的服务，在部署之前调节性能瓶颈。

9.2 CloudSim 体系结构

CloudSim 采用分层的体系结构，CloudSim 的架构及其架构组件如图 9-1 所示。

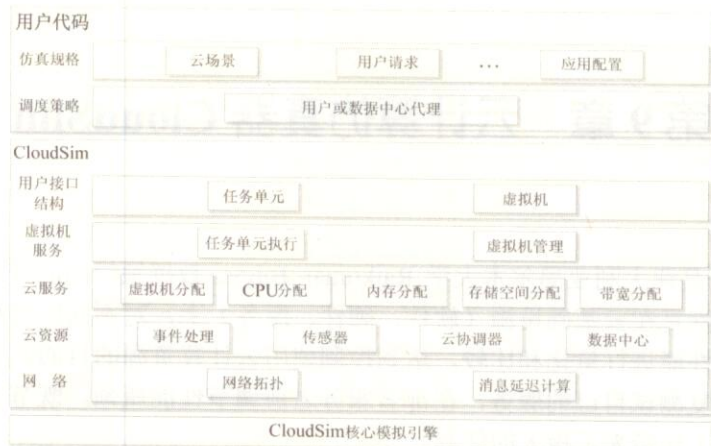


图 9-1 分层的 CloudSim 体系结构

早期的 CloudSim 使用 SimJava 作为离散事件模拟引擎，支持的核心功能有事件的排队和处理、云系统实体（在 CloudSim 环境下，实体是组件的一个实例，CloudSim 组件可以是一个类，或是由多个类组成的 CloudSim 模型，如服务、主机、数据中心、代理和虚拟机）的创建、组件之间的通信及模拟时钟的管理等。在 2.0 以上的版本中，为了支持一些 SimJava 不支持的高级操作，已经将 SimJava 从 CloudSim 的架构中移除。

云计算依然是一个新兴的分布式计算模型，由于缺乏相应的标准，因此，在未来几年无论是学术界还是企业界都会根据执行环境对核心算法、策略和应用标准进行研究。通过扩展 CloudSim 提供的基本功能，研究人员能够基于特定的环境和配置进行测试工作，实现对关键部分的最好的开发实践。

9.2.1 CloudSim 核心模拟引擎

GridSim 原本是 CloudSim 的一个组成部分，但是 GridSim 将 SimJava 库作为事件处理和实体间消息传递的框架，而 SimJava 在创建可伸缩仿真环境时暴露出如下一些不足：

- (1) 不支持在运行时通过编程方式重置仿真。
- (2) 不支持在运行时创建新的实体。
- (3) SimJava 的多线程机制导致性能开销与系统规模成正比，线程之间过多的上下文切换导致性能严重下降。
- (4) 多线程使系统调试变得更加复杂。

为了克服这些限制并满足更为复杂的仿真场景，墨尔本大学的研究小组开发了一个全新的离散事件管理框架。图 9-2 (a) 为相应的类图，下面介绍一些相关的类。

1) CloudSim

这是主类，负责管理事件队列和控制仿真事件的顺序执行。这些事件按照它们的时间参数构成有序队列。在每一步调度的仿真事件会从未来事件队列（Future Event Queue）中被删除，并被转移到延时事件队列（Deferred Event Queue）中。之后，每个实体调用事件处理方法，从延时事件队列中选择事件并执行相应的操作。这样灵活的管理方式，具有以下优势。

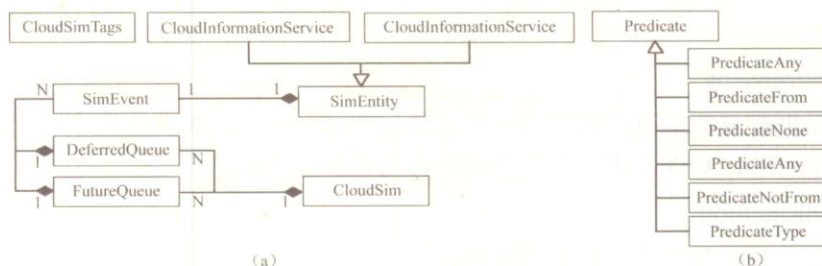


图 9-2 CloudSim 核心模拟引擎类图

- (1) 支持实体失活操作。
- (2) 支持不同状态实体的上下文切换，暂停或继续仿真流程。
- (3) 支持运行中创建新实体。
- (4) 支持运行中终止或重启仿真流程。

2) DeferredQueue

实现 CloudSim 使用的延时事件队列。

3) FutureQueue

实现 CloudSim 使用的未来事件队列。

4) CloudInformationService (IS)

CIS 是提供资源注册、索引和发现能力的实体。CIS 支持两个基本操作：`publish()`允许实体使用 CIS 进行注册；`search()`允许类似于 CloudCoordinator 和 Brokers 的实体发现其他实体的状态和位置，该实体也会在仿真结束时通知其他实体。

5) SimEntity

该抽象类代表一个仿真实体，该实体既能向其他实体发送消息，也能处理接收到的消息。所有的实体必须扩展该类并重写其中的三个核心方法：`startEntity()`、`processEvent()`和 `shutdownEntity()`，它们分别定义了实体初始化、事件处理和实体销毁的行为。SimEntity 类提供调度新事件和向其他实体发送消息的能力，其中消息传递的网络延时是由 BRUTE 模型计算出来的。实体一旦建立就会使用 CIS 自动注册。

6) CloudSimTags

该类包含多个静态的时间或命令标签，CloudSim 实体在接收和发送事件时使用这些标签决定要采取的操作类型。

7) SimEvent

该实体给出了在两个或多个实体间传递仿真事件的过程。SimEvent 存储了关于事件的信息，包括事件的类型、初始化时间、事件发生的时间、结束时间、事件转发到目标实体的时间、资源标识、目标实体、事件标签及需要传输到目标实体的数据。

8) CloudSimShutdown

该实体用于结束所有终端用户和代理实体，然后向 CIS 发送仿真结束信号。

9) Predicate

该类是个抽象类且必须被扩展，用于从延时队列中选择事件。图 9-2 (b) 给出了一些

标准的扩展。

10) PredicateAny

该类表示匹配延时队列中的任何一个事件。在 CloudSim 的类中有个可以公开访问的实例 CloudSim.SIM_ANY, 因此不需要为该类创建新的实例。

11) PredicateFrom

该类表示选择被特定实体放弃的事件。

12) PredicateNone

表示不匹配延时队列中的任何事件。在 CloudSim 中有个可以公开访问的静态实体 CloudSim.SIM_NONE, 因此用户不需要为该类创建任何新的实体。

13) PredicateNotFrom

选择已经被特定对象发送的事件。

14) PredicateType

根据特定标签选择事件。

15) PredicateNotType

选择不满足特定标签的事件。

9.2.2 CloudSim 层

CloudSim 仿真层为云数据中心环境的建模和仿真提供支持, 包括虚拟机、内存、存储器和带宽的专用管理接口。该层主要负责处理一些基本问题, 如主机到虚拟机的调度、管理应用程序的执行、监控动态变化的系统状态。对于想对不同虚拟机调度 (将主机分配给虚拟机) 策略的有效性进行研究的云提供商来说, 他们可以通过这一层来实现自己的策略, 以编程的方式扩展其核心的虚拟机调度功能。这一层的虚拟机调度有一个很明显的区别, 即一个云端主机可以同时分配给多台正在执行应用的虚拟机, 且这些应用满足 SaaS 提供商定义的服务质量等级。这一层也为云应用开发人员提供了接口, 只需扩展相应的功能, 就可以实现复杂的工作负载分析和应用性能研究。

CloudSim 又可以细化为 5 层。

1. 网络层

为了连接仿真的云计算实体 (主机、存储器、终端用户), 全面的网络拓扑建模是非常重要的。又因为消息延时直接影响用户对整个服务的满意度, 决定了一个云提供商的服务质量, 因此云系统仿真框架提供一个模拟真实网络拓扑及模型的工具至关重要。CloudSim 中云实体 (数据中心、主机、SaaS 提供商和终端用户) 的内部网络建立在网络抽象概念之上。在这个模型下, 不会为模拟的网络实体提供真实可用的组件, 如路由器和交换机, 而是通过延时矩阵中存储的信息来模拟一个消息从一个 CloudSim 实体 (如主机) 到另一个实体 (如云代理) 过程中产生的网络延时, 如图 9-3 所示。图 9-3 为 5 个 CloudSim 实体的延时矩阵, 在任意时刻, CloudSim 环境为所有的当前活动实体维护 $m \times n$ 大小的矩阵。矩阵中的元素 e_{ij} 代表一条消息通过网络从实体 i 传输到实体 j 产生的延时。

0	40	120	80	200
40	0	60	100	100
120	60	0	90	40
80	100	90	0	70
200	100	40	70	0

图 9-3 延时矩阵

CloudSim 是基于事件的仿真，不同的系统模型、实体通过发送事件消息进行通信。CloudSim 的事件管理引擎利用实体交互的网络延时信息来表示消息在实体间发送的延时，延时单位依据仿真时间的单位，如毫秒。

这意味着当仿真时间达到 $t+d$ 时，事件管理引擎就会将事件从实体 i 转发到实体 j ，其中 t 表示消息最初被发送时的仿真时间， d 表示实体 i 到 j 的网络延时。图 9-4 给出了这种交互的消息传递图。用这种模拟网络延时的方法，在仿真环境中为实用的网络架构建模，提供了一种既真实又简单的方式，并且比使用复杂的网络组件（如路由器和交换机等）建模更简单更清晰。



图 9-4 交互的消息传递图

2. 云资源层

与云相关的核心硬件基础设施均由该层数据中心组件来模拟。数据中心实体由一系列主机组成，主机负责管理虚拟机在其生命周期内的一系列操作。每个主机都代表云中的一个物理计算节点，它会被预先配置一些参数，如处理器能力（用 MIPS 表示）、内存、存储器及为虚拟机分配处理核的策略等，而且主机组件实现的接口支持单核和多核节点的建模与仿真。

为了整合多朵云，需要对云协调器（CloudCoordinator）实体进行建模。该实体不仅负责和其他数据中心及终端用户的通信，还负责监控和管理数据中心实体的内部状态。在监控过程中收到的信息将会活跃于整个仿真过程中，并被作为云交互时进行调度决策的依据。注意，没有一个云提供商提供类似于云协调器的功能，如果一个非仿真云系统的开发人员想要整合多朵云上的服务，必须开发一个自己的云协调器组件。通过该组件管理和整合云数据中心，实现与外部实体的通信，协调独立于数据中心的对象。

在模拟一次云整合时，有两个基本方面需要解决：通信和监控。通信由数据中心通过标准的基于事件的消息处理来解决，数据中心监控则由云协调器解决。CloudSim 的每一个数据中心为了让自己成为联合云的一部分，都需要实例化云协调器。云协调器基于数据中心的状态，对交互云的负载进行调整，其中影响调整过程的事件集合通过传感器（Sensor）实体实现。为了启用数据中心主机的在线监控，会将跟踪主机状态的传感器和云协调器关联起来。在监控的每个步骤，云协调器都会查询传感器。如果云协调器的负载

达到了预先配置的阈值,那么它就会和联合云中的其他协调器通信,尝试减轻其负载。

3. 云服务层

虚拟机分配是主机创建虚拟机实例的一个过程。在云数据中心,将特定应用的虚拟机分配给主机是由虚拟机分配控制器 (VmAllocationPolicy) 完成的。该组件为研究和开发人员提供了一些自定义方法,帮助他们实现基于优化目标的新策略。默认情况下, WmAllocationPolicy 实现了一个相对直接的策略,即按照先来先服务的策略将虚拟机分配给主机,这种调度的基本依据是硬件需求,如处理核的数量、内存和存储器等。在 CloudSim 中,要模拟和建模其他的调度策略是非常容易的。

给虚拟机分配处理内核的过程则是由主机完成的,需要考虑给每个虚拟机分配多少处理核及给定的虚拟机对于处理核的利用率有多高。可能采用的分配策略有:给特定的虚拟机分配特定的 CPU 内核 (空间共享策略)、在虚拟机之间动态分配内核 (时间共享策略) 以及给虚拟机按需分配内核等。

考虑下面这种情况,一个云主机只有一个处理核,而在这个主机上同时产生了两个实例化虚拟机的需求。尽管虚拟机上下文 (通常指主存和辅存空间) 实际上是相互隔离的,但是它们仍然会共享处理器核和系统总线。因此,每个虚拟机的可用硬件资源被主机的最大处理能力及可用系统带宽限制。在虚拟机的调度过程中,要防止已创建的虚拟机对处理能力的需求超过了主机的能力。为了在不同环境下模拟不同的调度策略, CloudSim 支持两种层次的虚拟机调度:主机层和虚拟机层。在主机层指定每个处理核可以分配给虚拟机的处理能力;在虚拟机层,虚拟机为在其内运行的单个应用服务 (任务单元) 分配一个固定的可用处理器能力。

在上述的每一层, CloudSim 都实现了基于时间共享和空间共享的调度策略。为了清楚地解释这些策略之间的区别及它们对应用服务性能的影响,可参见图 9-5 所示的一个简单的虚拟机调度场景。

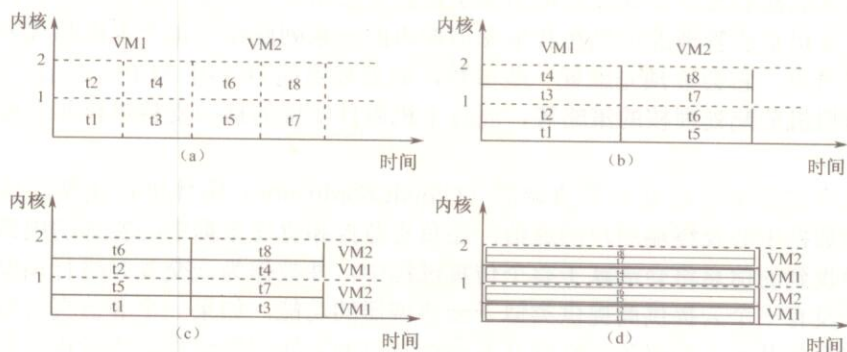


图 9-5 任务单元采用不同任务调度策略的影响

图 9-5 中,一台拥有两个 CPU 内核的主机将要运行两个虚拟机,每个虚拟机需要两个内核并要运行 4 个任务单元。更具体来说, VM1 上将运行任务 t1、t2、t3、t4,而 VM2 上将运行任务 t5、t6、t7、t8。

图 9-5 (a) 中虚拟机和任务单元均采用空间共享策略。由于采用空间共享模式,且每个虚拟机需要两个内核,所以在特定时间段内只能运行一个虚拟机。因此, VM2 只能在

VM1 执行完任务单元才会被分配内核。VM1 中的任务调度也是一样的, 由于每个任务单元只需要一个内核, 所以 t_1 和 t_2 可以同时执行, t_3 、 t_4 则在执行队列中等待 t_1 、 t_2 完成后再执行。

图 9-5 (b) 虚拟机采用空间共享策略, 任务单元采用时间共享策略。因此, 在虚拟机的生命周期内, 所有分配给虚拟机的任务单元在其生命周期内动态地切换上下文环境。

图 9-5 (c) 虚拟机采用时间共享策略, 任务单元使用空间共享策略。这种情况下, 每个虚拟机都会收到内核分配的时间片, 然后这些时间片以空间共享的方式分配给任务单元。由于任务单元基于空间共享策略, 这就意味着对于一台虚拟机, 在任何一个时间段内, 内核只会执行一个任务。

图 9-5 (d) 虚拟机和任务单元均采用时间共享策略。所有虚拟机共享处理器能力, 且每个虚拟机同时将共享的能力分给其任务单元。这种情况下, 任务单元不存在排队延时。

4. 虚拟机服务层

在这一层提供了对虚拟机生命周期的管理, 如将主机分配给虚拟机、虚拟机创建、虚拟机销毁以及虚拟机的迁移等, 以及对任务单元的操作。

5. 用户接口结构层

该层提供了任务单元和虚拟机实体的创建接口。

9.2.3 用户代码层

CloudSim 的最高层是用户代码层, 该层提供了一些基本的实体, 如主机 (机器的数量、特征等)、应用 (任务数和需求)、虚拟机, 还有用户数量和应用类型, 以及代理调度策略等。通过扩展这一层提供的基本实体, 云应用开发人员能够进行以下活动。

- (1) 生成工作负载分配请求和应用配置请求。
- (2) 模拟云可用性场景, 并基于自定义的配置进行稳健性测试。
- (3) 为云及联合云实现了自定义的应用调度技术。

9.3 CloudSim 技术实现

CloudSim 云模拟器的类设计图如图 9-6 所示, 本节详细介绍 CloudSim 的基础类, 这些类都是构建模拟器的基础。

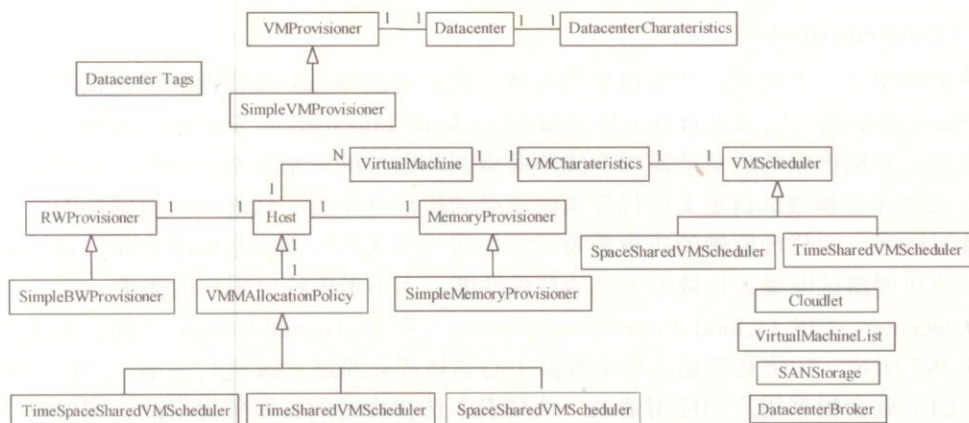


图 9-6 CloudSim 云模拟器的类设计图

主要类的功能描述如下。

1) BwProvisioner

这个抽象类用于模拟虚拟机的带宽分配策略。云系统开发和研究人員可以通过扩展这个类反映其应用需求的变化,实现自己的策略(基于优先级或服务质量)。BwProvisionerSimple 允许虚拟机保留尽可能多的带宽,并受主机总可用带宽的限制。

2) CloudCoordinator

这个抽象类整合了云数据中心,负责周期性地监控数据中心资源的内部状态和执行动态负载均衡的决策。这个组件的具体实现包括专门的传感器和负载均衡过程中需要遵循的策略。updateDatacenter()方法通过查询传感器实现监控数据中心资源。SetDatacenter()抽象方法实现了服务/资源的发现机制,这个方法可以被扩展实现自定义的协议及发现机制(多播、广播和点对点)。此外,还能扩展该组件模拟如 Amazon EC2 Load-Balancer 的云服务。对于想要在多个云环境下部署应用服务的开发人员,可以扩展这个类来实现自己的云间调度策略。

3) Cloudlet

这个类模拟了云应用服务(如内容分发、社区网络和业务工作流等)。每一个应用服务都会拥有一个预分配的指令长度和其生命周期内所需的数据传输开销。通过扩展该类,能够为应用服务的其他度量标准(如性能、组成元素)提供建模,如面向数据库应用的事务处理。

4) CloudletScheduler

该抽象类扩展实现了多种策略,用于决定虚拟机内的应用服务如何共享处理器能力。如上所述,这里支持两种调度策略:空间共享(CloudletSchedulerSpaceShared)和时间共享(CloudletSchedulerTimeShared)策略。

5) Datacenter

该类模拟了云提供商提供的核心基础设施级服务(硬件)。它封装了一系列的主机,且这些主机都支持同构和异构的资源(内存、内核、容量和存储)配置。此外,每个数据中心组件都会实例化一个通用的应用调度组件,该组件实现了一系列的策略用来为主机和虚拟机分配带宽、内存和存储设备。

6) DatacenterBroker

该类模拟了一个代理,负责根据服务质量需求协调 SaaS 提供商和云提供商。该代理代表 SaaS 提供商,它通过查询云信息服务(Cloud Information Service)找到合适的云服务提供者,并根据服务质量的需求在线协商资源和服务的分配策略。研究人员和系统开发人员如果要评估和测试自定义的代理策略就必须扩展这个类。代理和云协调器的区别是,前者针对顾客,即代理所做的决策是为了增加用户相关的性能度量标准;而后者针对数据中心,即协调器试图最大化数据中心的整体性能,而不考虑特定用户的需求。

Datacenter、CIS(Cloud Information Service)和 DatacenterBroker 之间信息交互的过程如图 9-7 所示。在仿真初期,每个数据中心实体都会通过 CIS 进行注册,当用户请求到达时,CIS 就会根据用户的应用请求,从列表中选择合适的云服务提供商。图中对交互的描述依赖于实际的情况,比如从 DatacenterBroker 到 Datacenter 的消息可能只是对下一个

执行动作的一次确认。

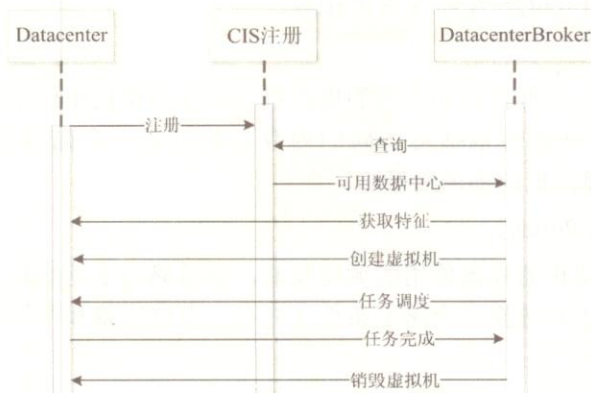


图 9-7 CloudSim 仿真数据流

7) DatacenterCharacteristics

该类包含了数据中心资源的配置信息。

8) Host

该类模拟了如计算机、存储服务器等物理资源。它封装了一些重要信息，如内存/存储器的容量、处理器内核列表及类型（多核机器）、虚拟机之间共享处理能力的分配策略、为虚拟机分配内存和带宽的策略等。

9) NetworkTopology

该类包含模拟网络行为（延时）的信息。它里面保存了网络拓扑信息，该信息由 BRUTE 拓扑生成器生成。

10) RamProvisioner

这个抽象类代表为虚拟机分配主存的策略。只有当 RamProvisioner 组件证实主机有足够的空闲主存，虚拟机在其上的执行和部署操作才是可行的。RamProvisionerSimple 对虚拟机请求的主存大小不强加任何限制，但是，如果请求超过了可用的主存容量，该请求就直接被拒绝。

11) SanStorage

该类模拟了云数据中心的存储区域网，主要用于存储大量数据，类似于 Amazon S3、Azure blob storage 等。SanStorage 实现了一个简单的接口，该接口能够用来模拟存储和获取任意量的数据，但同时受限于网络带宽的可用性。在任务单元执行过程中访问 SAN 中的文件会增加额外的延时，因为数据文件在数据中心内部网络传输时会发生延时。

12) Sensor

该接口的实现必须通过实例化一个能够被云协调器使用的传感器组件，用于监控特定的性能参数（能量消耗、资源利用）。该接口定义了如下方法：

- （1）为性能参数设置最小值和最大值。
- （2）周期性地更新测量值。
- （3）该类能够用于模拟由主流云提供商提供的真实服务，如 Amazon CloudWatch 和

Microsoft Azure FabricController 等。一个数据中心可以实例化一个或多个传感器, 每一个传感器负责监控数据中心的一个特定性能参数。

13) Vm

该类模拟了由主机组件托管和管理的虚拟机。每个虚拟机组件都能够访问存有虚拟机相关属性的组件, 这些属性包括可访问的内存、处理器、存储容量和扩展自抽象组件 CloudletScheduler 的虚拟机内部调度策略。

14) VmAllocationPolicy

该抽象类代表虚拟机监视器使用的调度策略, 该策略用于将虚拟机分配给主机。该类的主要功能是在数据中心选择一个满足条件 (内存、存储容量和可用性) 的可用主机, 提供给需要部署的虚拟机。

15) VmSheduler

该抽象类由一个主机组件实现, 模拟为虚拟机分配处理核所用的策略 (空间共享和时间共享)。该类的方法能很容易重写, 以此来调整特定的处理器共享策略。

9.4 CloudSim 的使用方法

CloudSim 提供基于数据中心的虚拟化技术、虚拟化云的建模和仿真功能, 支持云计算的资源管理和调度模拟。

9.4.1 环境配置

1) JDK 安装和配置

从 <http://www.oracle.com/technetwork/java/javase/downloads/index.html> 下载 JDK 最新版本并安装, CloudSim 需要运行在 jdk1.6 以上版本。以 jdk1.6.0_24 为例, 默认的安装目录为 C:\Program Files\Java\jdk1.6.0_24。设置环境变量: 新建系统变量 JAVA_HOME, 变量值设为 JDK 安装目录, 即 C:\Program Files\Java\jdk1.6.0_24; 在 Path 中加入路径 %JAVA_HOME%\bin; 在 ClassPath 中加入路径 %JAVA_HOME%\lib\dt.jar; %JAVA_HOME%\lib\tools.jar。

2) CloudSim 安装和配置

从 <http://www.cloudbus.org/cloudsim/> 下载 CloudSim, 本书以 CloudSim2.1.1 为例。将其解压到磁盘, 例如 C:\cloudsim-2.1.1。设置环境变量: 在 ClassPath 中加入路径 C:\cloudsim-2.1.1\jars\cloudsim-2.1.1.jar; C:\cloudsim-2.1.1\jars\cloudsim-examples-2.1.1.jar。

9.4.2 运行样例程序

1) 样例描述

C:\cloudsim-2.1.1\examples 目录下提供了一些 CloudSim 样例程序, 每个样例模拟的环境如下:

(1) CloudSimExample1.java: 创建一个一台主机、一个任务的数据中心。

(2) CloudSimExample2.java: 创建一个一台主机、两个任务的数据中心。两个任务具有一样的处理能力和执行时间。

(3) CloudSimExample3.java: 创建一个两台主机、两个任务的数据中心。两个任务对处理能力的需求不同,同时根据申请虚拟机的性能不同,所需执行时间也不相同。

(4) CloudSimExample4.java: 创建两个数据中心,每个数据中心一台主机,并在其上运行两个云任务。

(5) CloudSimExample5.java: 创建两个数据中心,每个数据中心一台主机,并在其上运行两个用户的云任务。

(6) CloudSimExample6.java: 创建可扩展的仿真环境。

(7) CloudSimExample7.java: 演示如何停止仿真。

(8) CloudSimExample8.java: 演示如何在运行时添加实体。

(9) network: 包含网络仿真的例子。

(10) power: 包含演示 CloudSim power-aware 特点的例子。

2) 运行步骤

需要安装 Windows 2000/XP/Vista 操作系统环境、JDK 及 Eclipse 集成开发环境。JAVA 版本要达到 1.6 或更高,CloudSim 和旧版本的 JAVA 不兼容,如果安装非 Sun 公司的 JAVA 版本,比如 gcj 或 J++,也可能不兼容。Eclipse 集成开发环境的版本要和 JDK 相匹配。本书使用 jdk 1.6.0_24 和 Eclipse 3.6.2。

由于已经配置了相应的环境变量,可以在 Windows 控制台直接运行样例程序。执行的命令及结果如图 9-8 所示。

```

C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>java org.cloudhuc.cloudsim.examples.CloudSimExample1
Starting CloudSimExample1...
Initialising...
Starting CloudSim version 2.0
Datacenter_0 is starting...
Broker is starting...
Entities started.
0.0: Broker: Cloud Resource List received with 1 resource(s)
0.0: Broker: Trying to Create VM #0 in Datacenter_0
0.0: Broker: VM #0 has been created in Datacenter #2, Host #0
0.0: Broker: Sending cloudlet 0 to VM #0
400.0: Broker: Cloudlet 0 received
400.0: Broker: All Cloudlets executed. Finishing...
400.0: Broker: Destroying VM #0
Broker is shutting down...
Simulation: No more future events
CloudInformationService: Notify all CloudSim entities for shutting down.
Datacenter_0 is shutting down...
Broker is shutting down...
Simulation completed.
Simulation completed.

===== OUTPUT =====
Cloudlet ID   STATUS   Data center ID   VM ID   Time   Start Time   Finish Time
0            SUCCESS      2              0      400      0           400
*****PowerDatacenter: Datacenter_0*****
User id      Debt
3            35.6
=====
CloudSimExample1 finished!

C:\Documents and Settings\Administrator>

```

图 9-8 样例 3 控制台输出

为了方便查看和修改代码,通常选择在 Eclipse 中执行,整个操作步骤如下。

(1) 首先启动 Eclipse 主程序,在 Eclipse 主界面上选择 File→New→Project 命令,打开“New Project”窗口(如图 9-9 所示),新建一个工程。

- (2) 选择“Java Project”，单击“Next”，创建一个 Java 工程，进入图 9-10 所示界面。
- (3) 填写 Java 工程的名称，取消选择复选框“Use default location”，浏览 CloudSim 源代码所在的目录，并选定该目录，如图 9-10 所示。
- (4) 单击“Next”按钮，显示 Java 工程的配置界面，该界面的选项卡包括源代码、工程和库等信息。CloudSim 源代码包括 examples、sources、jars、docs 等目录，如图 9-11 所示。
- (5) 单击“Finish”按钮完成创建 Java 工程的工作。

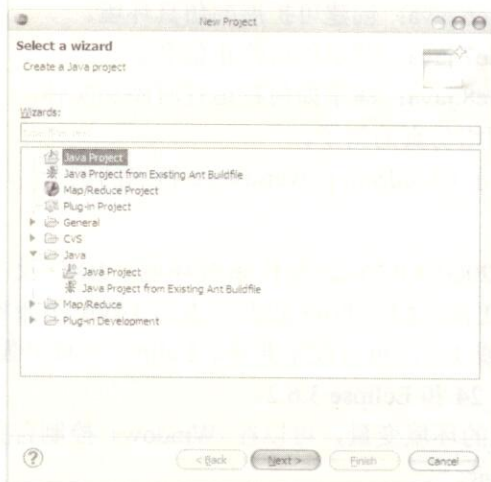


图 9-9 选择创建工程的类型

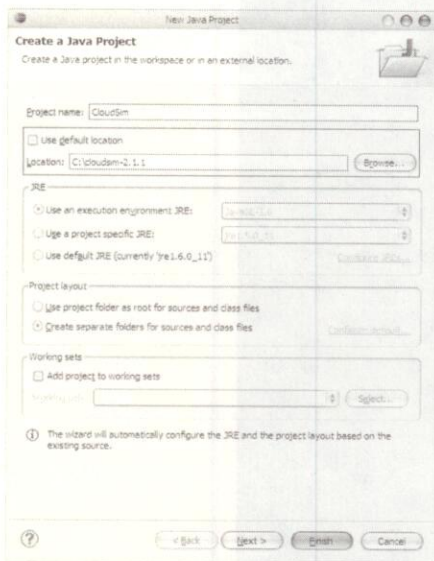


图 9-10 创建一个 Java 工程

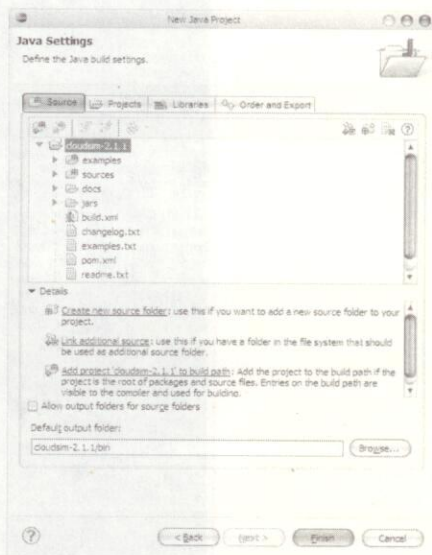


图 9-11 Java 工程的配置

在 Eclipse 的主界面上，选中一个实例的源代码，这里选择 CloudSimExample3，然后单击运行，如图 9-12 所示。

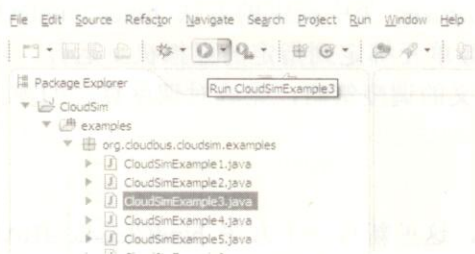


图 9-12 选择一个 CloudSim 样例并运行

程序的运行结果如图 9-13 所示。

```

Problems  @ Javadoc  Declaration  Console
<terminated> CloudSimExample3 [Java Application] E:\Java\jdk1.6.0_11\bin\javaw.exe (2011-4-15 上午09:55:01)
Starting CloudSimExample3...
Initialising...
Starting CloudSim version 2.0
Datacenter_0 is starting...
Broker is starting...
Entities started.
0.0: Broker: Cloud Resource List received with 1 resource(s)
0.0: Broker: Trying to Create VM #0 in Datacenter_0
0.0: Broker: Trying to Create VM #1 in Datacenter_0
0.0: Broker: VM #0 has been created in Datacenter #2, Host #0
0.0: Broker: VM #1 has been created in Datacenter #2, Host #1
0.0: Broker: Sending cloudlet 0 to VM #0
0.0: Broker: Sending cloudlet 1 to VM #1
80.0: Broker: Cloudlet 1 received
160.0: Broker: Cloudlet 0 received
160.0: Broker: All Cloudlets executed. Finishing...
160.0: Broker: Destroying VM #0
160.0: Broker: Destroying VM #1
Broker is shutting down...
Simulation: No more future events
CloudInformationService: Notify all CloudSim entities for shutting down.
Datacenter_0 is shutting down...
Broker is shutting down...
Simulation completed.
Simulation completed.

===== OUTPUT =====
Cloudlet ID   STATUS      Data center ID   VM ID   Time   Start Time   Finish Time
1            SUCCESS      2                1       80      0            80
0            SUCCESS      2                0      160      0           160
*****PowerDatacenter: Datacenter_0*****
User id      Debt
3            224.8
*****
CloudSimExample3 finished!

```

图 9-13 样例 3 程序运行结果

9.5 CloudSim 的扩展

CloudSim 是开源的，可以运行在 Windows 和 Linux 操作系统上，为用户提供了一系列可扩展的实体和方法，通过扩展这些接口实现用户自己的调度或分配策略，进行相关的性能测试。下面将通过一个简单的示例演示如何扩展 CloudSim，由于篇幅限制，本书仅以任务调度策略为例。源代码可在 <http://bbs.chinacloud.cn> 的教材板块下载。

9.5.1 调度策略的扩展

CloudSim 提供了很好的云计算调度算法仿真平台，用户可以根据自身的要求调用适

当的 API。如 DatacenterBroker 类中提供的方法 bindCloudletToVm (int cloudletId, int vmId)，实现了将一个任务单元绑定到指定的虚拟机上运行。除此之外，用户还可以对该类进行扩展，实现自定义的调度策略，完成对调度算法的模拟，以及进行相关测试和实验。

1. 顺序分配策略

作为一个简单的示例，这里新写一个方法 bindCloudletsToVmsSimple()，用于把一组任务顺序分配给一组虚拟机，当所有的虚拟机都运行有任务后，再从第一个虚拟机开始重头分配任务。该方法尽量保证每个虚拟机运行相同数量的任务以平摊负载，而不考虑任务的需求及虚拟机之间的差别。打开 org.cloudbus.cloudsim 包下的 DatacenterBroker 的源文件，如图 9-14 所示。

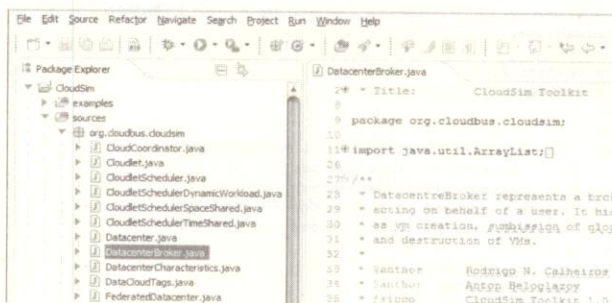


图 9-14 打开 DatacenterBroker 源文件

在该类的源文件中添加方法 bindCloudletsToVmsSimple()，实现代码如下：

```
public void bindCloudletsToVmsSimple()
{
    int vmNum=vmList.size();
    int cloudletNum=cloudletList.size();
    int idx=0;
    for(int i=0;i<cloudletNum;i++)
    { //将任务绑定到指定 id 的虚拟机
        cloudletList.get(i).setVmId(vmList.get(idx).getId());
        idx=(idx+1)%vmNum; //循环遍历虚拟机
    }
}
```

2. 贪心策略

实际上，任务之间和虚拟机之间的配置（参数）都不可能完全一样。顺序分配策略实现简单，但是忽略了它们之间的差异因素，如任务的指令长度（MI）和虚拟机的执行速度（MIPS）等。这里为 DatacenterBroker 类再写一个新方法 bindCloudletsToVmsTimeAwareed()，该方法采用贪心策略，希望让所有任务的完成时间接近最短，并只考虑 MI 和 MIPS 两个参数的区别。

通过分析 CloudSim 自带的样例程序，一个任务所需的执行时间等于任务的指令长度除以运行该任务的虚拟机的执行速度。读者也可以扩展相应的类，实现带宽、数据传输等对任务执行时间的影响。为了便于理解，不改变 CloudSim 当前的计算方式，即任务的执

行时间只与 MI 和 MIPS 有关。在这个前提下, 可以得出以下结论。

(1) 如果一个虚拟机上同时运行多个任务, 不论使用空间共享还是时间共享, 这些任务的总完成时间是一定的, 因为任务的总指令长度和虚拟机的执行速度是一定的。

(2) 如果一个任务在某个虚拟机上的执行时间最短, 那么它在其他虚拟机上的执行时间也是最短的。

(3) 如果一个虚拟机的执行速度最快, 那么它不论执行哪个任务都比其他的虚拟机要快。

定义一个矩阵 $time[i][j]$, 表示任务 i 在虚拟机 j 上所需的执行时间, 显然 $time[i][j] = MI[i]/MIPS[j]$ 。在初始化矩阵 $time$ 前, 首先将任务按 MI 的大小降序排序, 将虚拟机按 MIPS 的大小升序排列, 注意重新排序后矩阵 $time$ 的行号和任务 id 不再一一对应, 列号和虚拟机 id 的对应关系, 也相应改变。初始化后, 矩阵 $time$ 的每一行、每一列的元素值都是降序排列的, 然后再对 $time$ 做贪心。选用的贪心策略是: 从矩阵中行号为 0 的任务开始, 每次都尝试分配给最后一列对应的虚拟机, 如果该选择相对于其他选择是最优的, 就完成分配, 否则将任务分配给使当前结果最优的虚拟机。同时, 如果有多种分配方法都能使当前结果最优, 则将任务分配给运行任务最少的虚拟机, 实现一种简单的负载均衡。这种方式反映了越复杂的任务越需要更快的虚拟机来处理, 以解决复杂任务造成的瓶颈, 降低所有任务的总执行时间。实现代码如下。

//根据贪心算法将一组任务分配给一组虚拟机, 使总执行时间接近最短

```
public void bindCloudletsToVmsTimeAwarred(){
```

```
    int cloudletNum=cloudletList.size();
```

```
    int vmNum=vmList.size();
```

```
    double[][] time=new double[cloudletNum][vmNum];
```

```
    //重新排列任务和虚拟机, 需要导入包 java.util.Collections
```

```
    Collections.sort(cloudletList,new CloudletComparator());
```

```
    Collections.sort(vmList,new VmComparator());
```

```
    //初始化矩阵 time
```

```
    for(int i=0;i<cloudletNum;i++){
```

```
        for(int j=0;j<vmNum;j++){
```

```
            time[i][j]=
```

```
                (double)cloudletList.get(i).getCloudletLength()/vmList.get(j).getMips();
```

```
        }
```

```
    }
```

```
    double[] vmLoad=new double[vmNum]; //某个虚拟机上任务的总执行时间
```

```
    int[] vmTasks=new int[vmNum]; //某个虚拟机上运行的任务数
```

```
    double minLoad=0; //记录当前任务分配方式的最优值
```

```
    int idx=0; //记录当前任务最优分配方式对应的虚拟机列号
```

```
    //将行号为 0 的任务直接分配给列号最大的虚拟机
```

```
    vmLoad[vmNum-1]=time[0][vmNum-1];
```

```

vmTasks[vmNum-1]=1;
cloudletList.get(0).setVmId(vmList.get(vmNum-1).getId());

for(int i=1;i<cloudletNum;i++){
    minLoad=vmLoad[vmNum-1]+time[i][vmNum-1];
    idx=vmNum-1;
    for(int j=vmNum-2;j>=0;j--){
        //如果当前虚拟机还未分配任务，则比较完当前任务
        //分配给该虚拟机是否最优，即可以退出循环
        if(vmLoad[j]==0){
            if(minLoad>=time[i][j])idx=j;
            break;
        }

        if(minLoad>vmLoad[j]+time[i][j]){
            minLoad=vmLoad[j]+time[i][j];
            idx=j;
        }
        //实现简单的负载均衡
        else if(minLoad==vmLoad[j]+time[i][j]&&vmTasks[j]<vmTasks[idx])idx=j;
    }
    vmLoad[idx]+=time[i][idx];
    vmTasks[idx]++;
    cloudletList.get(i).setVmId(vmList.get(idx).getId());
}
}

//根据指令长度降序排列任务，需要导入包 java.util.Comparator
private class CloudletComparator implements Comparator<Cloudlet>{
    public int compare(Cloudlet cl1,Cloudlet cl2){
        return (int)(cl2.getCloudletLength()-cl1.getCloudletLength());
    }
}

//根据执行速度升序排列虚拟机
private class VmComparator implements Comparator<Vm>{
    public int compare(Vm vm1,Vm vm2){
        return (int)(vm1.getMips()-vm2.getMips());
    }
}

```

9.5.2 仿真核心代码

用户可以根据自己的需求，对主机相关参数配置（机器数量及特点）、云计算应用（任务、数量和需求）、VM、用户和应用类型的数量及代理调度策略等方面进行仿真测试。

1. 仿真步骤

下面将对 CloudSim 仿真每一步的作用进行详细介绍。

(1) 初始化 CloudSim 包。

每次进行仿真实验时，必须先进行初始化工作。这个过程主要是在其他实体创建前对 CloudSim 的参数进行初始化，包括用户数量、日期和跟踪标志。

(2) 创建数据中心。

在 CloudSim 仿真平台中，数据中心在 VM 的生命周期内负责管理 VM 的一组主机。一个数据中心由一个或多个主机组成，一个主机是由一个或多个 PE 或 CPU 组成。通过调用 API 函数，可以轻易地完成创建数据中心的工作。下面是创建数据中心的实现步骤。

(a) 创建主机列表。

(b) 创建 PE 列表。

(c) 创建 PE 并将其添加到上一步创建的 PE 列表中，可对其 ID 和 MIPS 进行设置。

(d) 创建主机，并将其添加到主机列表中，主机的配置参数有 ID、内存、带宽、存储、PE 及虚拟机分配策略（时间或空间共享）。

(e) 创建数据中心特征对象，用来存储数据中心的属性，包含体系结构、操作系统、机器列表、分配策略（时间、空间共享）、时区以及各项费用（内存、外存、带宽和处理资源费用）。

(f) 最后，创建一个数据中心对象，它的主要参数有名称、特征对象、虚拟机分配策略、用于数据仿真的存储列表以及调度间隔。

(3) 创建数据中心代理。

数据中心代理负责在云计算中根据用户的 QoS 要求协调用户及服务供应商和部署服务任务。数据中心代理函数，隐藏了虚拟机的管理，如创建、任务提交、虚拟机的销毁等，用于实现一定的数据中心代理策略，按照特定的规则提交虚拟机和云计算请求，结果返回数据中心代理数据类型。用户可以对其进行扩展，实现自己的任务调度算法。

(4) 创建虚拟机。

对虚拟机的参数进行设置，主要包括 ID、用户 ID、MIPS、CPU 数量、内存、带宽、外存、虚拟机监控器、调度策略，并提交给任务代理。

(5) 创建云任务。

创建指定参数的云任务，设定任务的用户 ID，并提交给任务代理。在这一步可以设置需要创建的云任务数量以及任务长度等信息。

(6) 在这一步调用自定义的任务调度策略，分配任务到虚拟机。

(7) 启动仿真。

(8) 在仿真结束后统计结果。

2. 详细实现代码

下面通过注释的方式讲解贪心策略的仿真核心代码，在 org.cloudbus.cloudsim.examples 包中新建类 ExtendedExample2，实现代码如下。

```
package org.cloudbus.cloudsim.examples;
```

```
import java.text.DecimalFormat;
import java.util.*;
import java.lang.Math;
import org.cloudbus.cloudsim.*;
import org.cloudbus.cloudsim.core.CloudSim;
import org.cloudbus.cloudsim.provisioners.*;

//测试调度策略的仿真核心代码
public class ExtendedExample2 {
    private static List<Cloudlet> cloudletList;    //任务列表
    private static int cloudletNum=10;           //任务总数

    private static List<Vm> vmList;               //虚拟机列表
    private static int vmNum=5;                  //虚拟机总数

    public static void main(String[] args) {
        Log.println("Starting ExtendedExample2...");
        try {
            int num_user = 1;
            Calendar calendar = Calendar.getInstance();
            boolean trace_flag = false;

            // 第一步：初始化 CloudSim 包
            CloudSim.init(num_user, calendar, trace_flag);

            // 第二步：创建数据中心
            Datacenter datacenter0 = createDatacenter("Datacenter_0");

            // 第三步：创建数据中心代理
            DatacenterBroker broker = createBroker();
            int brokerId = broker.getId();

            //设置虚拟机参数
            int vmid = 0;
            int[] mipss=new int[]{278, 289, 132, 209, 286};
            long size = 10000;
            int ram = 2048;
            long bw = 1000;
            int pesNumber = 1;
            String vmm = "Xen";

            // 第四步：创建虚拟机
            vmList = new ArrayList<Vm>();
            for(int i=0;i<vmNum;i++){
                vmList.add(new Vm(vmid, brokerId, mipss[i], pesNumber,
                    ram, bw, size, vmm, new CloudletSchedulerSpaceShared()));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



```

        vmid++;
    }
    //提交虚拟机列表
    broker.submitVmList(vmList);

    //任务参数
    int id = 0;
    long[] lengths = new long[]{19365, 49809, 30218, 44157, 16754,
        18336, 20045, 31493, 30727, 31017};
    long fileSize = 300;
    long outputSize = 300;
    UtilizationModel utilizationModel = new UtilizationModelFull();

    // 第五步: 创建云任务
    cloudletList = new ArrayList<Cloudlet>();
    for(int i=0;i<cloudletNum;i++){
        Cloudlet cloudlet=new Cloudlet(id, lengths[i], pesNumber, fileSize,
            outputSize, utilizationModel, utilizationModel, utilizationModel);
        cloudlet.setUserId(brokerId);
        cloudletList.add(cloudlet);
        id++;
    }
    //提交任务列表
    broker.submitCloudletList(cloudletList);

    // 第六步: 绑定任务到虚拟机
    broker.bindCloudletsToVmsTimeAwared();
    //broker.bindCloudletsToVmsSimple();

    // 第七步: 启动仿真
    CloudSim.startSimulation();

    // 第八步: 统计结果并输出结果
    List<Cloudlet> newList = broker.getCloudletReceivedList();
    CloudSim.stopSimulation();
    printCloudletList(newList);
    datacenter0.printDebts();
    Log.println("ExtendedExample2 finished!");
}
catch (Exception e) {
    e.printStackTrace();
    Log.println("Unwanted errors happen");
}
}

```

// 下面是创建数据中心的步骤

```
private static Datacenter createDatacenter(String name){
    // 1. 创建主机列表
    List<Host> hostList = new ArrayList<Host>();
    // PE 及主机参数
    int mips = 1000;
    int hostId=0;
    int ram = 2048;
    long storage = 1000000;
    int bw = 10000;
    for(int i=0;i<vmNum;i++){
        // 2. 创建 PE 列表
        List<Pe> peList = new ArrayList<Pe>();

        // 3. 创建 PE 并加入列表
        peList.add(new Pe(0, new PeProvisionerSimple(mips)));

        // 4. 创建主机并加入列表
        hostList.add(
            new Host(
                hostId,
                new RamProvisionerSimple(ram),
                new BwProvisionerSimple(bw),
                storage,
                peList,
                new VmSchedulerTimeShared(peList)
            )
        );
        hostId++;
    }

    // 数据中心特征参数
    String arch = "x86";
    String os = "Linux";
    String vmm = "Xen";
    double time_zone = 10.0;
    double cost = 3.0;
    double costPerMem = 0.05;
    double costPerStorage = 0.001;
    double costPerBw = 0.0;
    LinkedList<Storage> storageList = new LinkedList<Storage>();

    // 5. 创建数据中心特征对象
    DatacenterCharacteristics characteristics = new DatacenterCharacteristics(
        arch, os, vmm, hostList, time_zone, cost, costPerMem, costPerStorage, costPerBw);

    // 6. 创建数据中心对象
```



```

Datacenter datacenter = null;
try {
    datacenter = new Datacenter(name, characteristics,
        new VmAllocationPolicySimple(hostList, storageList, 0);
} catch (Exception e) {
    e.printStackTrace();
}
return datacenter;
}
//创建数据中心代理
private static DatacenterBroker createBroker(){

    DatacenterBroker broker = null;
    try {
        broker = new DatacenterBroker("Broker");
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
    return broker;
}

//输出统计信息
private static void printCloudletList(List<Cloudlet> list) {
    int size = list.size();
    Cloudlet cloudlet;
    String indent = "    ";
    Log.println();
    Log.println("===== OUTPUT =====");
    Log.println("Cloudlet ID" + indent + "STATUS" + indent +
        "Data center ID" + indent + "VM ID" + indent +
        "Time" + indent + "Start Time" + indent + "Finish Time");
    DecimalFormat dft = new DecimalFormat("###.##");
    for (int i = 0; i < size; i++) {
        cloudlet = list.get(i);
        Log.print(indent + cloudlet.getCloudletId() + indent + indent);
        if (cloudlet.getCloudletStatus() == Cloudlet.SUCCESS){
            Log.print("SUCCESS");
            Log.println(indent + indent + cloudlet.getResourceId() +
                indent + indent + indent + cloudlet.getVmId() +
                indent + indent + dft.format(cloudlet.getActualCPUTime()) +
                indent + indent + dft.format(cloudlet.getExecStartTime()) +
                indent + indent + dft.format(cloudlet.getFinishTime()));
        }
    }
}
}

```

3. 运行结果分析

由于虚拟机对任务分配使用了空间共享策略,所以运行在同一个虚拟机上的任务,必须按顺序完成。图 9-15 为基于贪心策略的仿真结果,图中显示任务 7、8 被分配到了虚拟机 0 上运行,从任务开始执行的时间来看,任务 8 确实是在任务 7 完成后执行的。图中还显示了所有任务的最终分配结果及运行情况,用户可以据此验证自己的调度策略是否符合要求。图 9-16 为基于顺序分配策略的仿真结果,该方法的总执行时间为 467.76,而贪心策略只需 283.46,节省了约 39%的时间。

```

===== OUTPUT =====
Cloudlet ID   STATUS   Data center ID   VM ID   Time   Start Time   Finish Time
7            SUCCESS    2                0       113.28    0           113.28
9            SUCCESS    2                3       198.51    0           198.51
3            SUCCESS    2                4       154.5     0           154.5
1            SUCCESS    2                1       172.45    0           172.45
8            SUCCESS    2                0       110.66    113.28      223.94
6            SUCCESS    2                4       70.22     154.5       224.71
2            SUCCESS    2                2       229.12    0           229.12
5            SUCCESS    2                3       87.93     198.51      236.44
0            SUCCESS    2                1       67.2      172.45      239.64
4            SUCCESS    2                4       58.75     224.71      283.46
****PowerDatacenter: Datacenter_0****
User id      Debt
3            562
*****
ExtendedExample2 finished!

```

图 9-15 贪心策略的仿真结果

```

===== OUTPUT =====
Cloudlet ID   STATUS   Data center ID   VM ID   Time   Start Time   Finish Time
4            SUCCESS    2                4       58.58     0           58.58
0            SUCCESS    2                0       69.76     0           69.76
5            SUCCESS    2                0       66.06     69.76       135.82
9            SUCCESS    2                4       108.56    58.58       167.14
1            SUCCESS    2                1       172.45    0           172.45
3            SUCCESS    2                3       211.38    0           211.38
2            SUCCESS    2                2       229.02    0           229.02
6            SUCCESS    2                1       69.49     172.45      241.94
8            SUCCESS    2                3       147.16    211.38      358.54
7            SUCCESS    2                2       238.73    229.02      467.76
****PowerDatacenter: Datacenter_0****
User id      Debt
3            562
*****
ExtendedExample2 finished!

```

图 9-16 顺序分配策略的仿真结果

9.5.3 平台重编译

实现自定义的调度算法后,用户就可以重新编译并打包 CloudSim,来测试或发布自己的新平台。CloudSim 平台重编译主要通过 Ant 工具完成。

从 <http://ant.apache.org/> 下载 Ant 工具,本书使用的版本为 1.8.2。将其解压到目录 C:\apache-ant-1.8.2。设置环境变量,在 Path 中加入 C:\apache-ant-1.7.1\bin。将命令行切换到扩展的 CloudSim 路径 (build.xml 所在目录),在命令行下输入命令 C:\cloudsim-2.1.1>ant,批量编译 CloudSim 源文件,生成的文件会按照 build.xml 的设置存储到指定位置,编译成功后自动打包生成 cloudsim-new.jar 并存放在 C:\cloudsim-2.1.1\jars。扩展的 CloudSim 平台生成后,在环境变量 ClassPath 中增加路径: C:\CloudSim\jars\cloudsim-new.jar。然后根据前面介绍的步骤,即可在新的平台下编写自己的仿真实验程序。

习题

1. 总结 CloudSim 的功能及其应用场景。
2. 与 SimJava 相比, 简要说明 CloudSim 核心模拟引擎新增了哪些功能。
3. CloudSim 层被细分为五层, 试总结每一层提供的功能。
4. 详细阅读 CloudSim 技术实现一节, 总结有哪些类提供了自定义调度接口, 并对该类的功能进行概述。
5. 任选 3 个 CloudSim 样例程序运行, 并对输出结果进行分析。
6. 尝试扩展 CloudSim, 实现一个自定义的调度算法, 并在扩展平台下对算法性能进行测试和分析。

参考文献

- [1] Rodrigo N. Calheiros , Rajiv Ranjan , César A. F. De Rose , and Rajkumar Buyya. CloudSim: A Novel Framework for Modeling and Simulation of Cloud Computing Infrastructures and Services. <http://www.cloudbus.org/reports/CloudSim-ICPP2009.pdf>
- [2] Rodrigo N. Calheiros , Rajiv Ranjan , César A. F. De Rose , and Rajkumar Buyya. CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. <http://www.buyya.com/papers/CloudSim2010.pdf>

第10章 云计算研究热点

与网格计算相反,云计算最先由企业界发起,然后才引起学术界的重视。云计算的概念是在2007年第3季度由Amazon和IBM等公司率先推动,2008年10月,网格之父Ian Foster举办了首届云计算会议(Cloud Computing and Applications Workshop, CCA'08)。随后,相关知名国际会议如Cluster、CCGrid、GCC、Gloud Expo、Cloudcom、Cloud Slam等都将云计算作为主题之一,2010年11月,第二届云计算国际学术会议CloudCom2010在美国举行。通过对迄今为止的云计算相关学术论文进行统计分析,反映出学术界对云计算的研究主要集中在以下六个方面:①云计算体系结构的归纳;②云计算关键技术,特别是云设施的管理技术,包括虚拟机、云监测、能耗管理、数据管理和资源调度等;③云计算编程模型,即如何针对某类应用特点提出效率更高的编程方式;④云计算支撑平台,即研究更好的云计算平台;⑤云计算应用,包括科学应用和商业应用;⑥云安全,即研究将云计算思想用于信息网络安全防护。学术界对云计算的研究方兴未艾,预计将在很多领域很快深入下去,例如云和云之间的互操作、跨云虚拟化、云计算与网格计算的结合等领域。

10.1 云计算体系结构研究

10.1.1 Youseff 划分方法

Youseff 等^[45]依据可组合性对云系统进行了分类,重点介绍了云的层次栈,阐述了各层次内涵、作用、架构,并深入分析了它们之间的相互依存的关系,有利于加强对云系统分类、联系和内部依存关系的理解。所谓可组合性,是指在云层次栈中,如果一个服务可以由下面的层次组合而成,则其可组合性较强。例如,云应用可以通过云软件环境开发而得到,则说明云应用是可以从云软件环境组合而来,云应用层也就在云软件环境层之上了。此方法是从SOA中借鉴而来,在此指重组云服务能力。

Youseff 等认为云计算可以看做一种新的计算机范式,它允许用户暂时利用网络计算设施,这些计算设施被云提供者包装成服务提供给用户。当前云计算的发展,其基础要素来源于很多其他计算领域和系统工程领域。除了集群和网格计算,虚拟化也是其前身,这些技术促成了云计算的诞生。还有一些其他技术对今天的云计算技术的形成产生了间接的影响,包括P2P、SOA和自治计算。

把云计算系统作为一个可组合的服务,使研究者能够定义不同云实体间更为健壮的操作模型。此外,它能够强化对不同云系统间内部相互依存关系的认识,提供了更多不同服务的合作机会,加强了给予分析技术的服务质量保证。

Youseff 等提出的云层次栈,将云计算的三种服务归属于云系统的不同软件架构层,如图10-1所示。

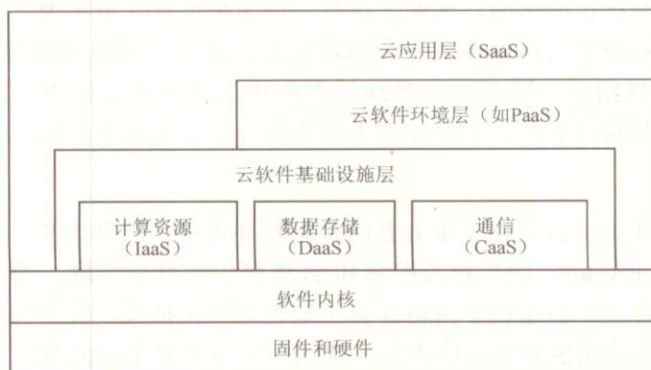


图 10-1 云层次栈

1. 云应用层

该层是云中对用户可视化最好的一层。用户通过该层提供的 Web 门户访问服务，通常这些服务是收费的。该商业模型已得到很多用户的青睐，因为它减轻了软件维护的压力和操作支持的耗费。此外，它将计算工作从用户终端导出到数据中心，即云应用部署的地方，大大降低了对用户终端硬件设施的要求。

对云应用的提供者而言，这个模型大大简化了他们更新和测试代码的工作，因为云应用是部署在供应商的计算基础设施上（而不是用户桌面），所以开发者只需回滚较小的片段并能够在不打扰用户的情况下更新程序。

该层次栈表明，云应用可以在云软件环境或基础设施组件上开发。此外，云应用可以由其他云系统提供的云服务组成。例如，一个工资单应用可以用另一个会计 SaaS 来计算每个员工的自付税额，而无需在软件中实现该服务。

但是，尽管有以上诸多好处，部署过程中存在的一些问题还是阻碍了其推广进程，尤其是其安全性和可行性这两个重要问题，目前 SLA 避免了这两个问题。另外，如何处理突然中断是 SaaS 用户和提供商必须考虑的问题，尤其是可能发生的网络中断和系统故障。

2. 云软件环境层

云层次栈的第二层是云软件环境层，也称为软件平台层，该层用户是云应用的开发者，他们实现应用并在云上部署服务。云软件环境提供商提供给开发者一组 API，该层所提供的服务通常被归为 PaaS，该策略的典型应用案例是 Google App Engine^[5]和 Salesforce Apex Language^[9]。

3. 云软件基础设施层

该层为高层提供功能性资源，可以组成新的云软件环境或应用。所提供的云服务可以分为三类：计算资源、数据存储和通信。

(1) 计算资源。在这一层，虚拟机是给云用户提供计算资源的普遍方式。

(2) 数据存储。允许用户在远程硬盘上存储数据并可以在任何地方任何时候访问。这项服务通常被称为 DaaS，它有效地扩大了云的用户群。数据存储系统通常需要满足维护用户数据的诸多要求，包括高可用性、可靠性、性能、备份和数据一致性等，由于这些要

求本质上相互之间是存在冲突的，因此，没有一个系统可以完全实现它们。

(3) 通信。云系统中对网络通信服务质量保证的需求，使得通信成为云系统中一个至关重要的云基础设施组件。通常，云系统需要提供这样的通信能力：面向服务的、结构的、可调度、可预测和可靠的。把通信作为服务的概念应运而生，即 CaaS。

4. 软件内核

该层次负责管理组成云的物理服务器的基础软件管理。软件内核在这一层可以看做由操作系统内核、hypervisor、虚拟机监控器和/或集群中间件来实现。通常，网格计算部署和运行在该层的集群上，但是由于网格计算中缺乏虚拟化抽象，任务与真实的硬件基础设施有着紧密的联系。应用的移植、检查和负载均衡一直是复杂的问题。

5. 固件和硬件

最底层是形成云的基础骨架的硬件和交换机等。HaaS 为用户提供硬件的操作、管理和更新服务。

文献[45]还依据所提出的体系结构对当前的云计算系统进行了分类，如表 10-1 所示。

表 10-1 云计算系统云层次栈中的分类

云 平 台	商业云系统例子
云应用层	Google Apps 与 Salesforce 客户关系管理 (CRM) 系统
云软件环境层	Google App Engine 和 Salesforce Apex 系统
云软件基础设施层	计算资源: Amzon EC2
	存储: Amzon S3, EMC 存储管理服务
	通信: 微软连接服务框架 (CSF)
软件内核	网格和集群计算系统 (如 Globus 和 Condor)
固件和硬件	IBM-Morgan Stanley 的计算分包和 IBM 的 Kittyhawk 项目

10.1.2 Lenk 划分方法

当前大量的云计算技术 (从开源的 Hadoop MapReduce 到像 Amzon、Google 等商业应用等) 使研究者们感到困惑，这些技术之间的关系是怎样的？该如何从技术角度、软件架构角度及商业前景角度对这些云技术和服务进行比较理解？Lenk^[46]解答了这些问题。他提出了一个通用的云计算栈，将云技术和服务分为不同的层次，并通过例子对每个层次进行阐述，说明该模型是如何从总体上涵盖现有的云计算技术的，进一步阐述了该栈在云计算环境建模中的应用。

云计算栈如图 10-2 所示，该栈将不同的技术和工具依据提供给用户的最高层接口进行分类，用户只需关注各个技术的最主要的用途。该体系结构栈为各个技术的联合、互换提供了指导。当然，在当今云计算快速发展的时候想要做一个完全的分析分类几乎是不可能的，因此，这种对当前云计算技术的分类并不全面。下面对文献[46]提出的云计算栈中的各个层次进行简要介绍。

1. 基础设施即服务层 (IaaS)

该层划分为资源集和基础设施服务两部分。最底层离硬件最近的部分即资源集层，分为两大类：物理资源集 (PRS) 和虚拟资源集 (VRS)。PRS 层的实现依赖于硬件，应用

案例有 Emulab^{[3][47]}、iLO^{[4][48]}等。VRS 层的应用实例有 Amazon EC2^[49]、Eucalyptus^[50]、Tycoon^[51]、Nimbus^[52]和 OpenNebula^[53]。这样分类在于以下两个原因。

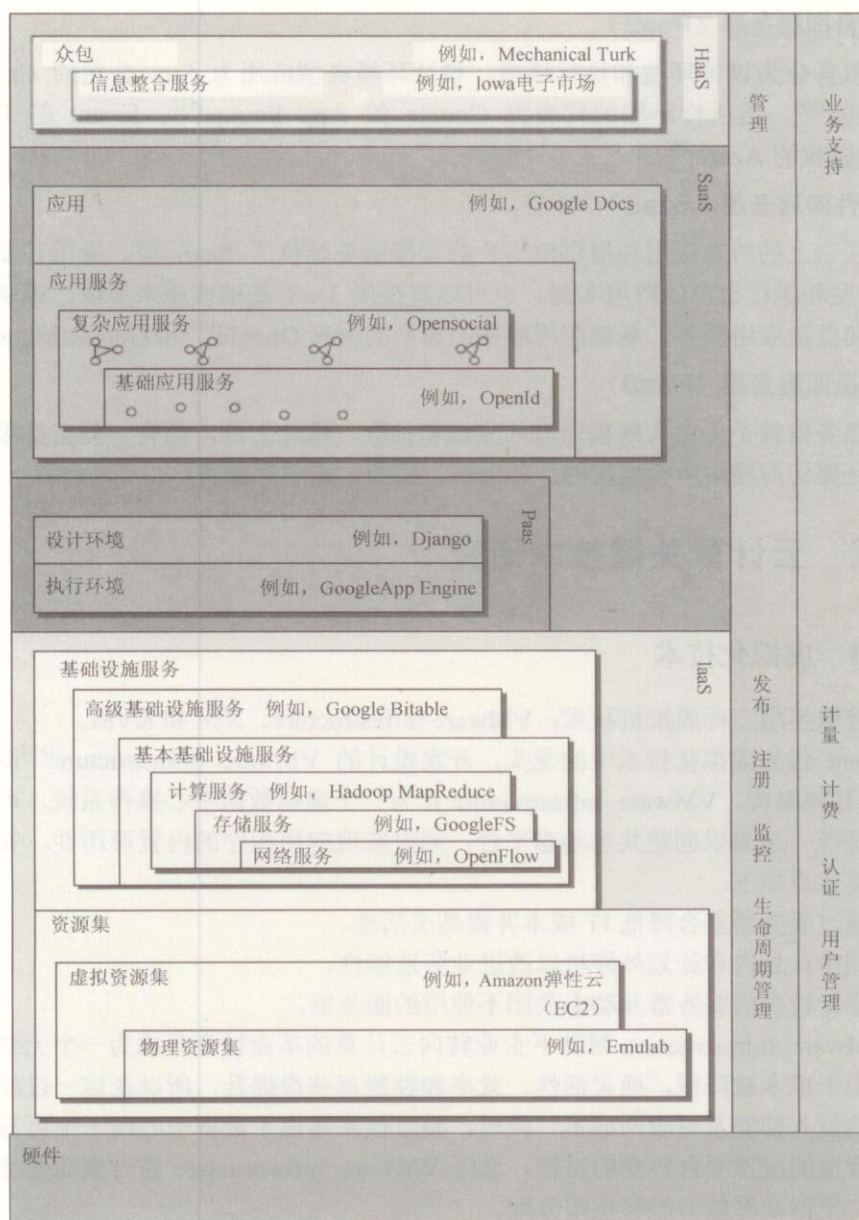


图 10-2 云计算栈

(1) 可以对物理硬件进行像虚拟资源一样的自动管理。

(2) 不同类型的资源如存储资源、网络资源、计算节点资源需要以不同的方式进行虚拟化, 但仍要有一个通用的 PRS 接口。

基础设施服务被分为基本的基础设施服务 (BIS) 和高级的基础设施服务 (HIS)。BIS 又分为计算服务、存储服务和网络服务, 这一类的实例有 MapReduce^[54] (计算服

务)、GoogleFS^[55] (存储服务) 和 OpenFlow^[56] (网络服务)。HIS 的典型实例有 Amazon's Dynamo^[57] 和 Google 的 Bigtable^[58], 它们是建立在 BIS 工具上的。

2. 平台即服务层 (PaaS)

该层服务分为设计环境和执行环境。设计环境典型应用为 Sun 公司的 Caroline^[59] 和 Django 框架^[60], 而执行环境的代表为 Google 的 App Engine^[61]、Joyent 的 Reasonably Smart^[62] 和微软的 Azure^[63]。

3. 软件即服务层 (SaaS)

运行在云上的所有应用和提供给用户的直接服务都位于 SaaS 层。应用开发者可以用 PaaS 层开发和运行自己的应用实例, 也可以直接用 IaaS 基础设施来实现。该层分为基础应用服务和复杂应用服务, 基础应用服务的典型实例有 OpenId^[64] 和 Google Maps^[65] 服务。

4. 人员即服务层 (HaaS)

一些服务依赖于人对大规模信息的集成和抽取, 除此之外, 还有一些支持服务。云中有些服务是要访问栈中所有层次的, 如调试、监控、账单等服务。

10.2 云计算关键技术研究

10.2.1 虚拟化技术

目前普遍使用三种虚拟机技术: VMware Infrastructure、Xen 和 KVM。

VMware 作为虚拟化技术中的龙头, 开发设计的 VMware Infrastructure^[18] 能创建自我优化的 IT 基础架构。VMware Infrastructure 作为一个虚拟数据中心操作系统, 可以将离散的硬件资源统一起来以创建共享动态平台, 同时实现应用程序的内置可用性、安全性和可扩展性, 其优点如下:

- (1) 通过服务器整合降低 IT 成本并提高灵活性。
- (2) 减少计划内和计划外停机以改进业务连续性。
- (3) 运行较少的服务器并动态关闭不使用的服务器。

在 VMware Infrastructure 帮助下企业转向云计算的革命性变革成为一个无缝地渐进发展过程, 由于成本被降低, 而灵活性、效率和性能都获得提升, 所以在这一过渡过程中每个阶段内的投入都能及时收回成本。同时, 整合技术缩短了数据中心向一个更易于管理、更高成本效益的虚拟平台转变的过程。通过 VMware Infrastructure 整合资源并且使基础架构自动化, 使得计算能力能够按需分配。

Xen^[19] 是由 XenSource 所管理的一个开源 GPL 项目。Xen 是 openSUSE 和 Novell 主要支持的虚拟化技术, 它能够创建更多的虚拟机, 每一个虚拟机都是运行在同一个操作系统上的实例。

虚拟机服务器上的虚拟机操作系统以两种模式运行: 全虚拟化和半虚拟化。全虚拟化是一种完全模拟所有硬件设备的虚拟化模式; 而半虚拟化是一种选择性的模拟硬件设备的虚拟化模式。

虚拟机监控系统 (VMM) 在服务器硬件和 SUSE Linux 系统内核之间运行。计算机启

动时首先加载 VMM, 然后以特权模式启动虚拟机服务器。特权模式指的是虚拟机服务器可以创建并控制虚拟机, 而且还可对计算机硬件进行直接存取。用户需要设置虚拟机服务器与本地设备驱动程序来匹配实际的计算机设备。

Xen 开源社区开发和维护着涉及软件层的 VMM 和系统管理软件 (Hypervisor), 同时它也提供功能性的虚拟机服务器。

KVM 是指基于 Linux 内核 (Kernel-based) 的虚拟机 (Virtual Machine), 是以色列的一个开源组织提出的一种新的虚拟机实现方案, 也称为内核虚拟机。

虚拟化技术通过将工作量灵活地分配给不同的物理机实现资源的共享。但这样一来, 内存中就会存在空闲。文献[70]中认为, 在信息处理高峰期, 虚拟机承担一定的工作量, 而客户端操作系统的内存 (包括未分配工作量的空闲虚拟机的内存, 及分配给特定虚拟机器却未被客户端操作系统充分利用的内存) 可能会存在“空闲”。即使客户端操作系统需要分配更多的内存, 也无法使用其他客户端操作系统中的空闲内存。在这种情形下, 客户端操作系统会因物理内存不足转而去利用交换设备。由于本地作为交换设备的硬盘驱动器处理效率远远低于物理内存的效率, 因此系统性能将会明显下降。研究证明, 很多技术能够通过将某台机器作为远程交换设备来使用远程内存, 以提高集群内存的使用效率, 但都缺乏远程内存的动态分配机制。为了提高系统性能及内存的有效利用率, 文中提出的 VSMM 虚拟化交换管理机制, 能够在云环境中实现交换设备的虚拟化, 以及内存灵活、动态的交换管理。

10.2.2 数据存储技术

Daniel 等^[5]探讨了在云中配置数据库管理系统的优势与局限性。

云计算环境有以下 3 个突出的特点。

- (1) 在工作量可并行计算的提前下, 计算能力是弹性的。
- (2) 数据存储在不信任的主机上。云计算这个名词给人的印象好像是计算存储资源的位置不确定, 但实际上数据是处于特定国家符合当地法律规范的主机中的, 所以用户对存储在云中的数据只有有限的控制, 数据还是处于一个相对来说不安全的环境中。
- (3) 数据通常是进行远程复制。数据的可用性是云存储提供商很关心的问题, 如果数据不可用将会影响提供商的信誉。提供商采用副本容错的方式保证数据的可用性。

从云计算环境的三个特点分析得出, 只有两种数据管理应用程序可能适合部署到云计算中: 一种是和事务处理相关的数据管理系统; 另一种是和分析相关的数据管理系统。

基于分析的云数据库管理系统应该具有以下性质和特点。

- (1) 效率。使用云计算的费用与请求的网络带宽、存储空间的能力和计算力是成正比的。
- (2) 容忍错误。基于分析的数据库管理系统不能因为某个节点失效而重新执行查询请求, 而是允许在节点失效的情况下同样保证查询的正确性。
- (3) 能够在异构的环境中运行。云计算中的计算机性能差别很大, 所以必须采取相应的机制, 防止某些任务在性能较差的计算机上执行时, 由于速度过慢对整个任务的完成造成较大影响。
- (4) 能够操作加密的数据。很多敏感信息都是经过加密后上传到云中, 为了防止非授权的访问, 云数据库管理系统要有处理加密数据的能力。

(5) 能够与商业化的智能产品进行交互。

为了保证信息的安全性,多数厂商对数据采用了冗余存储的方式。针对用户的数据产生多个副本,并将这些副本保存在不同地理位置的服务器和数据中心上,因此,如何保证这些数据的一致性至关重要。现有的数据一致性模型中,不同服务器之间的相互依赖,导致系统性能与吞吐量的降低,文献[71]中提出了一种基于树的一致性模型,通过对数据服务器副本进行部分一致性与完全一致性状态的标记处理,降低了服务器不同副本之间的相互依赖。基于服务器主副本到其余副本最可靠的路径进行一致性树的创建,提高了系统性能与吞吐量。

为了保证数据的一致性,数据服务器副本之间要始终保持通信,及时了解数据的相应变化。在进行事物处理之前,必须确认所有副本中都已经保存了最近更新的数据。任一服务器节点若对更新数据没有反应,则所有的副本节点均必须等待,直到全部确认了更新信息。这一过程很容易导致事务处理的失败,从而使得整体性能下降。在基于树的数据一致性方法中,为了保证数据的一致性且不会造成性能的下降,在云数据库系统中设立了两种节点,一种是控制器 (Controler),另外一种是数据服务器副本 (Database Replication)。系统中可能有两个或两个以上的控制器,其任务是建立一致性树。当用户访问数据服务器时,由控制器决定所要选择的数据库;当有故障发生时,由控制器重建树。数据服务器副本用于存储数据,完成各项事务处理及其他一些数据库操作。所有的副本之间是有联系的,其中委派一个主副本建立与用户的联系,并负责与其他副本保持通信以保证数据的及时更新。控制器与服务器副本之间的通信情况如图 10-3 所示。

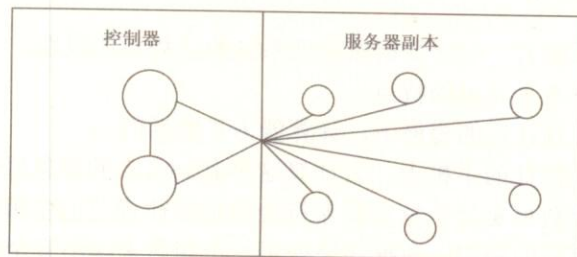


图 10-3 控制器与服务器副本之间的通信

一致性树的建立过程为:①建立加权图 (Weighted Connection Graph) $G(V, E)$, 将每一服务器副本作为一个节点并由这些副本组成加权图的顶点集 V , 服务器副本之间的直接联系作为边, 并组成加权图的边集 E , 网络路径的可靠性作为边的权重。②选择根节点, 由控制器选择可靠性最高的服务器副本作为将建立的一致性树的根节点, 并由该副本负责与用户的联系。③在指定了根节点后, 将树的根节点作为单一源点, 利用单源点最短路径 Dijkstra 算法, 找出根节点到各副本节点的最可靠的路径, 并建立一致性树。连通图与创建的一致性树如图 10-4 所示。

当需要更新数据时,每一数据服务器副本节点负责其子树节点的数据更新,副本中均包含两个标志位:部分一致性标志位与完全一致性标志位。当根节点需要更新其数据时,通知其所有的子节点所要更新的全部信息,子节点确认收到信息并给予回馈。当所有的子节点均确认信息后,根节点服务器开始更新数据,并将该次操作序列号作为部分一致性标志位存储起来。若其子节点为中间节点,则采用同样的方法,对其子节点进行数据更新。

对叶子节点更新其数据副本，需要建立一个空子节点链表，并向该子节点链表发送需要更新数据的信息，同样将操作序列号作为部分一致性标志位和完全一致性标志位存储起来，然后通知它的父节点存储全局一致性标志位。如果其父节点为中间节点，则继续往上，并存储完全一致性标志位，直到根节点将更新操作序列号作为其完全一致性标志位。

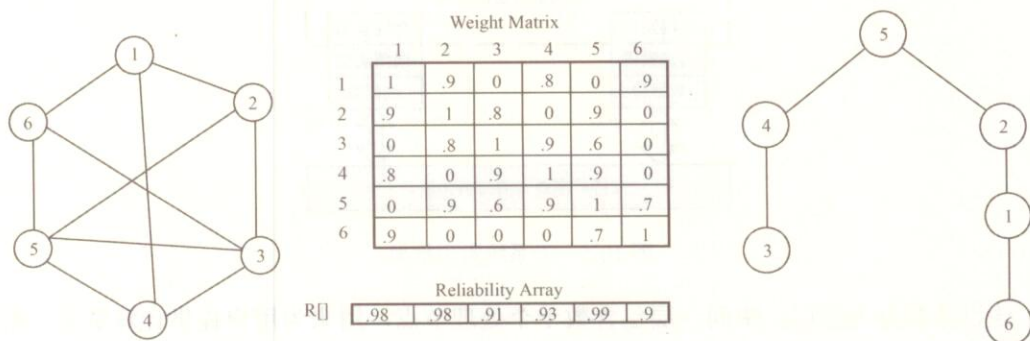


图 10-4 连通图与一致性树

操作中遇到的故障可能有：主副本服务器故障，即根节点发生故障，或者根节点之外的某一副本服务器发生故障。由于控制器始终与主副本服务器（根节点）保持通信，当主副本服务器节点发生故障时，控制器便与其子节点通信并查询其部分标志位与完全标志位。如果一致，控制器将选择所有服务器副本中可靠性最高的节点作为新的根节点；若不一致，通过查询含有相同的部分一致性序列号的节点即可找到最新更新的服务器副本，并在这些最近更新的服务器副本中查找可靠性最高的节点，并将其作为新的根节点。最后对这些正常工作的服务器组成的连通图进行重新配置并更新一致性树。如果根节点之外的某一服务器副本的子节点对于更新数据没有反映，控制器与其通信也联系不上，则说明该服务器节点发生故障，则将该节点从树中删除，如果是通信路径发生问题，则转通过另外的途径通信，并更新连通图及一致性树。

这种基于树的一致性模型减轻了副本之间的相互依赖，当某一副本需要更新数据时仅需要告知其子节点获得确认信息后即可进行更新，甚至在不可靠的网络中也能够降低事务处理失败的风险。

10.2.3 资源管理技术

云系统的出现使得软件供应商对大规模分布式系统的开发变得简单。云系统为开发商和用户提供了简单通用的接口，使开发商能够将注意力更多地集中在软件本身，而无需考虑底层架构。当前的云系统试图通过提高并行度来提升性能，文献[6]提出了一个新的解决方案——一个基于结构化覆盖的云系统索引框架。该框架可以减少云内部数据传输量，并便于数据库后台应用的开发。云系统依据用户的资源获取请求，动态分配计算资源。

Sai Wu 等人^[8]研究并提出了一个通用的云系统索引框架，如图 10-5 所示。在该框架中，处理节点以结构化覆盖网络的形式组织在一起，每个节点建立本体索引以加速数据访问。一个全局索引通过在覆盖网络中选择和发布一个本地索引分配来建立。全局索引是分布在整个网络中的，并且每个节点负责保持一个全局索引的子集。考虑到存储代价和其他维护开销，一个基于开销模型的自适应索引方法用于调整全局索引。两个运行在 Amazon

的 EC2 上的实验证明了该方法的可行性和有效性。

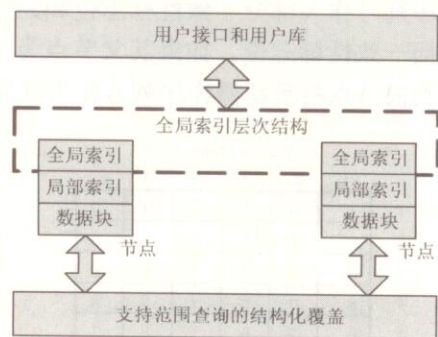


图 10-5 云系统索引框架

该设计共分为三层。中间一层包含数千个处理节点，用来为用户提供计算资源。用户数据被划分为一些数据块，这些数据块依据 DFS 协议分布在各个不同节点上，每个节点都为其上部署的数据创建一个本地索引。除了本地索引，每个节点还将其存储资源共享出一部分，以维护全局索引。所谓全局索引，是一个索引入口集，从本地索引中选取出来并散布在集群中。

为了给用户提供友好的接口，考虑应用结构化的覆盖网络来组织节点并管理全局索引。在最底层，处理节点的连接组织是松耦合的。每当有一个新节点连入到云系统中，其他节点都会运行覆盖网络的连接协议。新节点将会以一些节点作为路由邻居，并告知其加入信息。这个过程和 P2P 网络的结构比较相似，但又和 P2P 网络不同。在云系统里，服务由服务提供者进行管理，加入系统中的节点提供计算资源。在连接网络中，节点必须保持在线，除非出现硬件故障。相反，P2P 网络中，对等节点是完全自治的和不稳定的。一个对等节点接入到 P2P 网络中都有其自身的任务，并在结束任务时离开网络。但是，在该系统中，覆盖网络只有路由功能。

在最上层为用户应用提供了一个数据访问接口，该接口是基于全局索引的，用户可以根据不同的访问需求选择不同的数据访问方法。基于索引的访问更适合在线查询，同时，扫描是分析大型数据集的可行方法。

虚拟私有云 (Virtual Private Clouds, VPC) 对于公有云就相当于虚拟私有网络 (VPN) 对于公共网络。对数据进行“加密”后再通过网络进行传输，实现了 VPN。但是对于云计算来说，仅仅依赖加密解密和身份认证技术并不能在公共的“云”中虚拟一片私有“云”或 VPC。因为云服务中的处理器只对以明文形式存在的代码和数据进行计算，加密的内容只能存在于在网络上或磁盘中。内存中的内容不能是密文。所以，真正的 VPC 需要对云服务提供者的内存和 CPU 的寄存器作一种非加密方式的保护，使得当租客的代码和数据，在云服务提供者的内存和 CPU 的寄存器中，以明文形式被处理时，仍然能得到私密性及完整性的保护，避免被其他租客或服务提供者窃取。为了实现企业用户对于 VPC 的需求，基于虚拟机实现的虚拟机间的隔离技术得到了普遍的关注，但却不能避免诸如通信堵塞等影响系统性能的问题，因此对带宽需求较高。文献[72]中提出一种网络资源管理 (Network Resource Management, NRM) 系统，引入一个不断变化的基于 CHAMELEON 的软件模块及一个带有虚拟节点的多结点网络拓扑结构。这种基于软件架构的资源管理系统

NRM 能够通过接入相应的库来管理不同种网络设备。设计的 CHAMELEON 软件模块使得网络资源管理系统 NRM 能够支持网络基础设施的扩展，并在实验中运用 NRM 控制六种不同的网络设备不做任何修改。

大部分传统的 NRM 仅能控制一种特定的网络设备，如图 10-6 (a) 所示，当增加新的网络设备时，必须扩展其相应的 NRM。此外，NRM 之间还需要一致性机制进行彼此通信以便提供端到端的可靠网络，但是管理代价较高。虚拟私有云中包含各种不同的虚拟机，当有新的网络设备被添加到 WAN 或者数据中心网络时，通过一个 NRM 来处理多种不同的网络设备以实现弹性可变的、保证带宽的虚拟私有云至关重要。图 10-6 (b) 为持续的 NRM (Sustainable NRM)，通过导入对应的控制库实现不同种类的网络设备的管理。当需要添加新的网络设备时，利用基于 CHAMELEON 的软件模块在 NRM 中上传一个新的控制库到库管理服务器 (Repository Server)，NRM 不需要做任何改变即可管理新加设备。

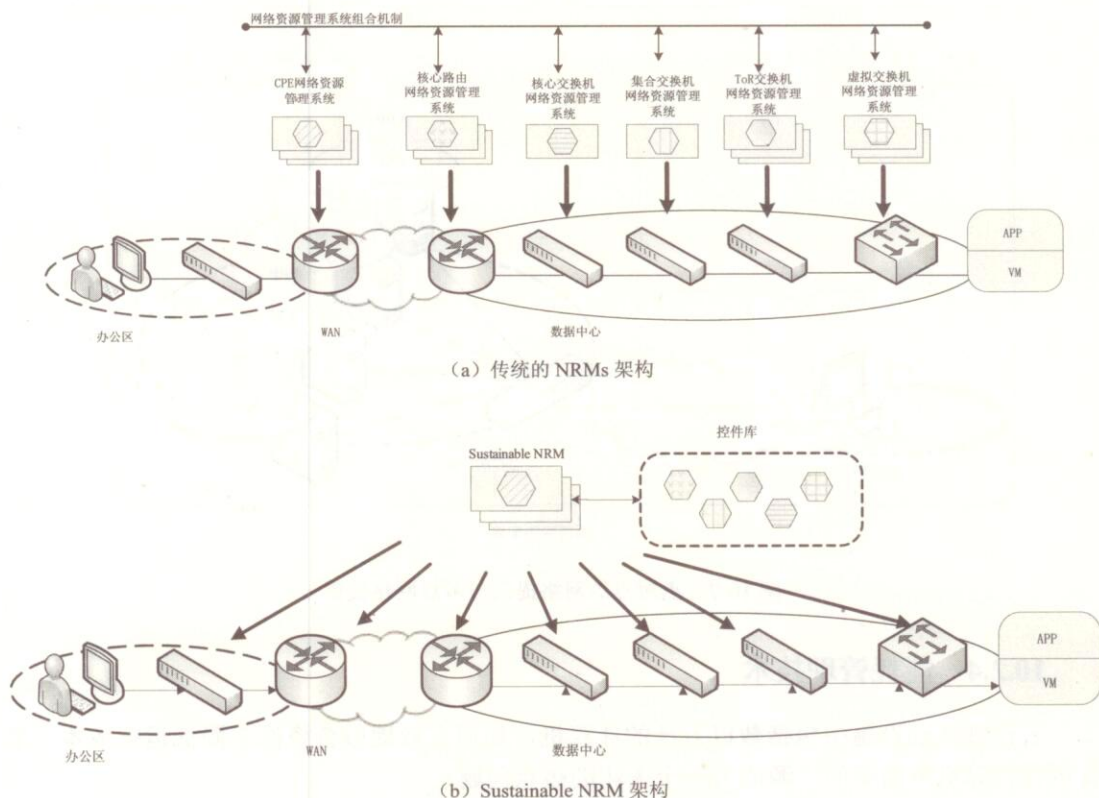


图 10-6 传统 NRM 架构与 Sustainable NRM 架构

传统的虚拟私有云不能保证网络吞吐量，在虚拟机之间采取一种提供点对点的网络的措施，如图 10-7 (a) 所示，这种完全网络结构需要虚拟机之间的完全连通，且这种带宽的分配不可扩展。如图中总体物理网络带宽可达到 1Gbps，而对于两个虚拟机之间的平均带宽却仅分配了 250Mbps，则虚拟机之间的带宽就限制在 250Mbps 之下。当需要有新的虚拟机加入时，分配的带宽需要重新计算和重新分配，这种方法较为低效且不够灵活。持

续的 NRM 提出的策略如图 10-7 (b) 所示。类似于星型的拓扑结构, 虚拟网络节点作为云网络的中心节点, 指派虚拟机与虚拟网络节点作为两终端节点, 当需要添加新的虚拟机时, 只需在虚拟机与虚拟网络节点之间开辟新的网络路径即可。

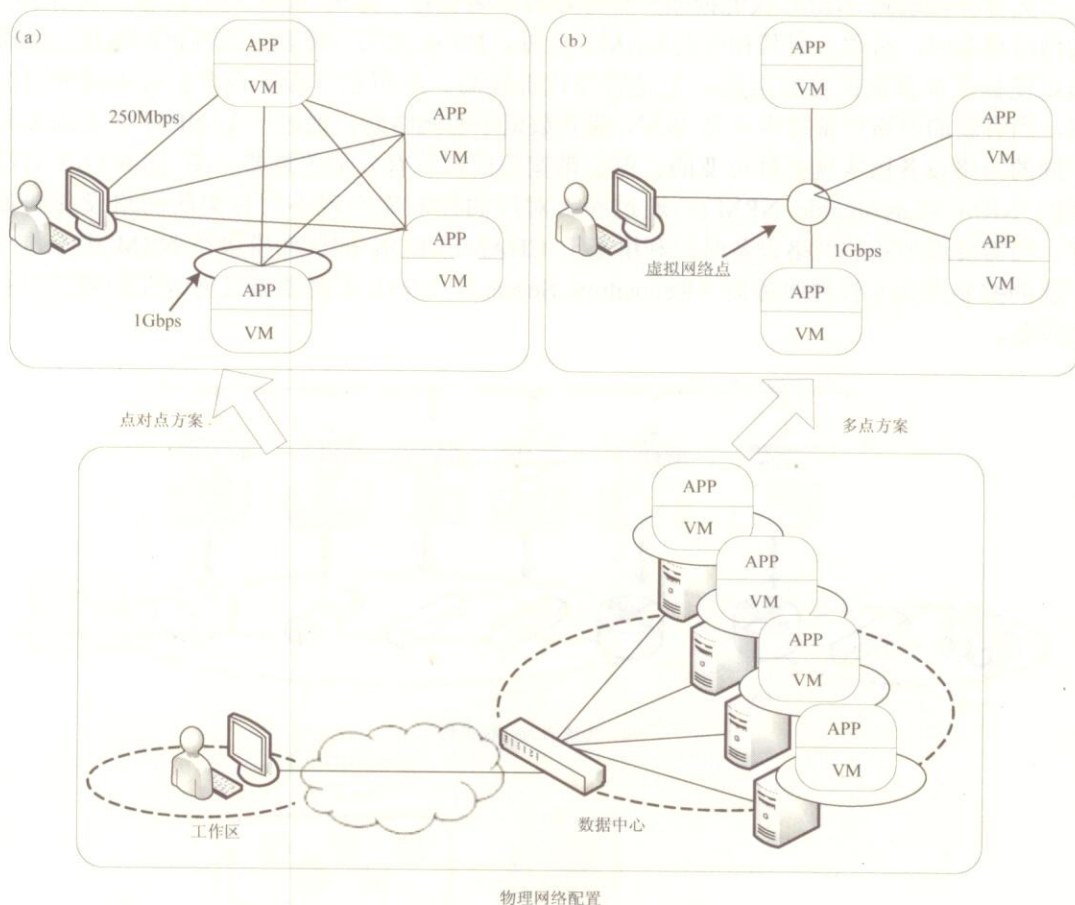


图 10-7 点对点的网络提供与多点网络提供

10.2.4 能耗管理技术

云计算基础设施中包括数以万计的计算机, 如何有效地整合资源、降低运行成本、节省运行计算机所需要的能源成为一个关注的热点问题。

Shekhar Srikantaiah 等^[4]研究了云计算中能源消耗、资源利用率及整合后的工作性能之间的内在关系, 对云平台中能源优化问题做出了实践和探索。

图 10-8 为一个研究资源利用率、计算机工作性能和能源消耗的实验步骤。云中包括 4 台服务器, 它们控制来自客户端的 k 个应用程序服务, 每个服务器都连接一个测定能量的功率计和一个监控资源利用率的跟踪器。



图 10-8 实验步骤

经测试发现,计算机性能受磁盘利用率的影响大于受 CPU 利用率的影响,当 CPU 利用率一定时,计算机性能随磁盘利用率的增高而线性降低,计算机性能变化曲线如图 10-9 所示。

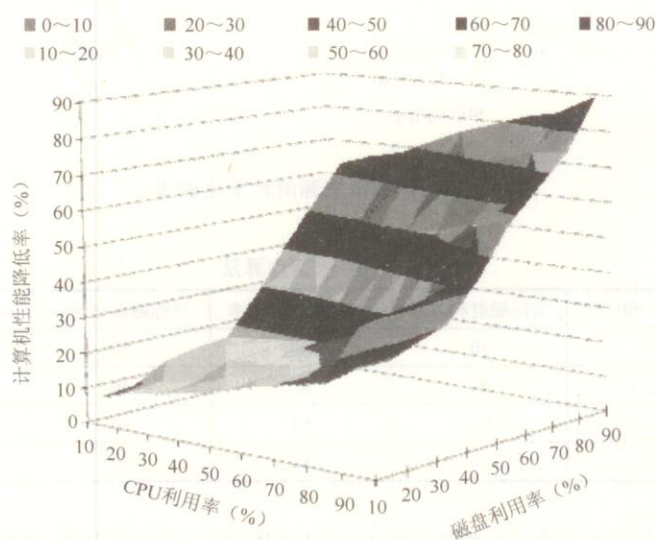


图 10-9 计算机性能变化曲线

计算机能源消耗受 CPU 利用率的影响大于受磁盘利用率的影响,同时能源的消耗在磁盘利用率为 50%, CPU 利用率为 70% 时取得最小值。计算机能源消耗变化曲线如图 10-10 所示。

降低能源消耗的资源整合算法如表 10-2 所示。假如服务器 A 的 CPU 资源利用率是 30%, 磁盘利用率是 30%, 表示为 [30,30], 服务器 B 为 [40,10], 两台服务器能源消耗最低的资源利用标准是 [80,50], 此时一个新的作业请求需要 [10,10] 的资源。该算法首先计算欧几里得距离 δ , 服务器 A 初始的距离为 $[30,30] - [80,50] = 53.8$, B 的初始距离为 $[40,10] - [80,50] = 56.6$, 如果新的作业请求分配给 A, 则 A 的距离变为 $[40,40] - [80,50] = 41.2$; 如分配给 B, 则 B 的距离变为 $[50,20] - [80,50] = 42.4$ 。经过比较把作业分配给 A 后使得服务器 A 和 B 的总的距离 $\Sigma\delta$ 更大, 所以选择此方案。

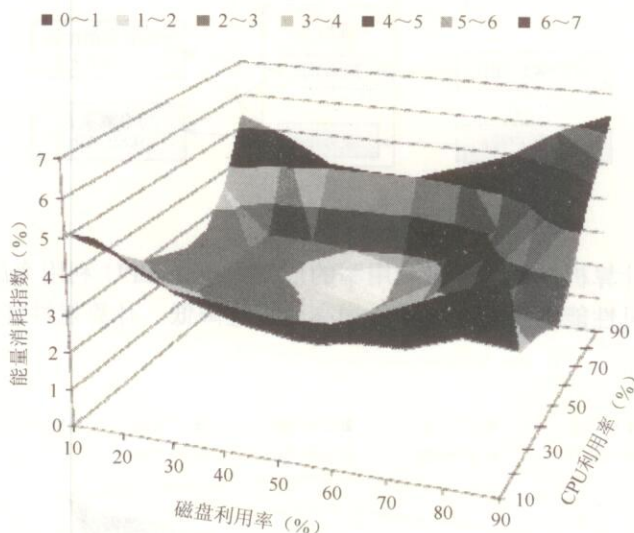


图 10-10 计算机能源消耗变化曲线

表 10-2 资源整合算法

	实际 CPU 利用率	实际磁盘利用率	目标 CPU 利用率	目标磁盘利用率	欧几里得距离 δ	$\Sigma\delta$
A (初始)	30	30	80	50	53.8	97.8
A (分配后)	40	40	80	50	41.2	
B (初始)	40	10	80	50	56.6	96.2
B (分配后)	50	20	80	50	42.4	

在有关能耗的研究中, 反弹效应 (Rebound Effect) 被认为是最简单直接的表现。即当某一产品设计得更为高效节能时, 其价格便越便宜且会有更多的人去使用它, 因此就会产生更多的能耗, 使得产品本身的节能与使用过程中产生的能耗相互抵消。同理, 在 IT 基础设施提供的服务中也是如此。技术的不断革新使得能源更能被充分利用, 这样可以节省能源并降低成本。为了估测我们将来所需的最大带宽水平, 首先需要分析消费者的潜在的网络行为并且分析这些行为对于网络服务的需求。当消费者在网上浏览网页、使用流媒体技术、视频下载时, 有个相关的概念叫做 “Data Intensity”, 即在单位时间内为了支持某种网络服务需要下载的平均数据量, 因此文献[73]中统计了不同网络行为的整体数据下载量需求, 分析了世界范围内的数据下载需求及由其导致的能量需求, 并提出了降低能耗与碳排放量的建议。

假设西方国家每人的媒体消费均通过网络, 但平均消费水平未超过目前的水平, 并且全球中产阶级均能达到西方国家的消费水平, 则预计全球平均每人每天对数据量的需求是 3200MB, 总计全球每年达到 2570EB, 统计出目前对带宽的能耗需求 4Wh/MB。按照目前的能耗水平, 进行这些网络活动所需要的电量将达到 1175GW。

为了满足更多的网络服务需求, 降低能耗, 减少数字媒体下载量, 可采用如下几种措施。

- (1) 减少数字垃圾。所谓数字垃圾是指用户下载到客户端的一些没有使用的数据。例

如,在有些多媒体信息极为丰富的页面上,数据量过兆,包括很多视频,用户难以拖动屏幕滚动条,浏览页面后面的内容,这导致了资源的浪费。

(2) 进行策略性的界面设计。进行具有说服性的外观设计,鼓励用户尽量少使用、少下载某些数据。例如,可以减少页面上的多媒体内容,减少媒体呈现方式的变化,把有关视频的一些链接放在页面的下方,使用户可以选择性观看等等。

(3) 提高使用意识。提供一些能够反映出能量消耗或温室气体排放量的直观展示,提高用户的使用与节能意识。同时改变一些付费的方式(如付费模型更能体现能量的消耗量),可以根据个人的低碳意识从而计算付费的多少。

(4) 避开使用高峰期。减少在网络使用高峰期下载数据的工作,既可以为用户提供优质的服务,还能够降低总体能耗,是一个不错的选择。

10.2.5 云监测技术

为了更好地体现云计算强大的处理海量数据的能力,检测和分析云计算系统、虚拟机的行为变得尤为重要和关键。在这方面研究人员做了大量的工作,目前已经出现了一些典型的研究案例。

1. 大规模监测系统 Chukwa

Jerome Boulon 等^[2]设计实现了 Chukwa,它是建立在 Hadoop 上的数据收集系统,用以监测和分析大规模分布式系统。同时它还包括一个可扩展的功能强大的工具集,用于显示监测和分析的结果,这样有利于更好地使用该数据收集系统。

Chukwa 有广泛的适用范围,它适合 Hadoop 用户、运营商、管理员和开发者使用。Hadoop 用户能运行 Chukwa 监测其工作的进程,掌握设备的使用状况,同时可以获得相关的日志信息;运营商可以使用 Chukwa 监测设备的性能、发现故障;管理员可以使用 Chukwa 分析设备的使用情况,并对未来的需求做出预测和判断;Hadoop 开发者可以使用 Chukwa 了解 Hadoop 的运行性能,用以指导设计与开发。

为了满足其应用的需求,Chukwa 需要弹性的自动可控的数据源和高性能大规模的存储系统,同时需要一个可分析大量数据的体系结构。Chukwa 体系结构如图 10-11 所示。

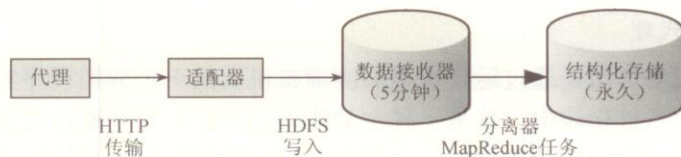


图 10-11 Chukwa 体系结构

在 Chukwa 系统中自动可控制的数据源称为适配器。适配器用于收集日志信息、应用程序数据和系统信息等。为了使适配器便于写数据,Chukwa 中设计了代理,代理其实是运行在监控节点中的一个进程,为适配器提供多种服务。代理有如下三个功能。

- (1) 通过外部命令对适配器进行启动或是停止。
- (2) 通过 HTTP 向收集器提交数据。
- (3) 可对适配器的状态做定期的检查。

Chukwa 系统中的存储子系统采用的是 HDFS,而 HDFS 缺点是并行的写速率不高,

不支持小文件的读/写。所以在 Chukwa 中对其进行了改进提高，设计实现了收集器。Chukwa 中不允许适配器直接往 HDFS 中写数据，而是通过网络把数据发送给收集器，收集器接收来自不同主机的数据，当接收一定数量的数据后，把这些数据写入到一个单独的文件中，然后存储到 HDFS 中去。

为了便于分析收集来的数据，Jerome Boulon 等建立了可扩展的、易配置的网页界面程序——Hadoop 基础服务中心，对数据进行分析 and 显示。

经过测试，高性能分布式监控系统 Chukwa 能很容易地建立在现有的分布式数据收集框架之上。

2. 虚拟机内部监测方法

云计算提供商为云用户提供大量计算资源，云用户在使用这些资源时，需要将自己的操作系统镜像上传到云设施计算机中，这样云计算环境允许用户在云供应商的硬件上执行任意代码。因此云用户面临很多安全的挑战，恶意用户可以利用供应商的硬件发动攻击，这种攻击能破坏供应商的信誉，同时影响服务其他客户的能力。这就必须要求云供应商利用虚拟机内部监测方法来检测用户恶意行为，从而保证虚拟机的正常工作，确保云用户不会受到恶意攻击。

Lionel Litty 等^[3]从监控虚拟机事件的范围和干预具体事件的能力、对被监控虚拟机影响的程度、健壮性三个方面对基于主机代理、陷阱和检查、检测点和回滚、体系结构监控四种内部监测方法进行了对比分析，结果如表 10-3 所示。

表 10-3 内部监测技术能力测试

内部监测方法	能 力	影 响 程 度	健 壮 性
主机代理	较好	较差	好
主机代理 w/driver	好	差	较差
陷阱与检查	好	较好	差
检测点与回滚	好	较好	差
体系结构监控	较差	较好	好

从表中可以看出四种方法各有优劣，适用于不同的应用场景。

1) 基于主机代理

基于主机代理的方法是通过运行在被监测虚拟机上的一个应用程序实现的，它可能位于用户空间或一个内核模块中。这是一种主动注入的方法，对被监控的虚拟机有很大的影响。这种方法的健壮性和监控事件的能力是由代理的设计和操作系统的特点决定的，不同的操作系统不可能为系统的监控提供完全兼容的 API 包，如果 API 没有提供足够的监控事件的能力，代理可以通过增加内核代码来弥补这种缺陷。因此，这种方法的健壮性和监控事件的能力相对较好。

2) 陷阱和检查

这种方法不是一种注入方式，与基于主机代理的方法相比，此方法防止对代码的篡改，避免对虚拟机执行的冲突，所以这种方法对被监控的虚拟机的影响较小。

3) 检测点与回滚

这种内部监测方法是对第二种方法的一个扩展和深入，当虚拟机监控器监测到有异常

行为时,会将系统回滚到以前设置的检测点的位置。此方法对被监控的虚拟机的影响较小,监控虚拟机事件的范围及干预具体事件的能力较强,但在健壮性方面有待加强。

4) 体系结构监控

这种方法的原理是监控那些有明确定义不易改变的接口。此方法有很强的健壮性,因为接口很稳定,攻击者很难实时地对其进行改变。这种方法没有在操作系统镜像中插入陷阱,它只是被动地监控硬件事件,所以这种方法在监控虚拟机事件的范围、干预具体事件的能力方面比不上前述三种监测方法。但是比较而言体系结构监控方法对云监控来说已经足够。

通过对几种方法的比较分析,可以给出云计算提供商如何选择内部监测方法的判断原则。就健壮性和对被监控虚拟机影响的程度上看,体系结构监控方法有很好的性质,通过设计体系结构监控方法,能使其监控操作系统更多的方面。如果用户信任安装在虚拟机上的监控代理,也可采取第一种方法。有时将几种方法同时使用,会取得更好的效果。

10.3 编程模型研究

10.3.1 All-Pairs 编程模型

虽然现代并行分布式计算系统可以提供强大的计算能力,但非专业用户并不能有效地利用,一个庞大的任务可能因为某个初学者的偶然操作而意外地导致滥用资源共享并造成性能的大幅下降。为了解决这个问题,应该提供给用户一个高度抽象的产品系统,以此来帮助用户进行大数据量的计算。

Christopher Moretti 和 Jared Bulosan 等^[67]提供一个抽象的实例——All-Pairs,它可以适用于数据密集型的科学应用。一个优化的 All-Pairs 抽象系统比底层系统更容易操作,而且和传统的方式相比,All-Pairs 拥有更好的性能。

许多科学家需要使用分布式计算系统解决复杂的科学问题,但是许多人不是分布式领域的专家,因而他们不能很好地解决数据的处理问题以及不能有效地利用资源。一个小的操作失误都可能导致很糟糕的结果,比如滥用工作队列和计算软件等,针对此类问题提供一个有效的抽象可以有效地避免分布式计算的缺陷。这个抽象为用户提供一个接口,以便从数据和计算方面来定义和描述问题,并且隐藏了具体的实现过程。

All-Pairs 问题可以简单地定义为:

编程模型 All-Pairs

输入参数: 集合 A, 集合 B, 函数 F

功能: 使用函数 F 将集合 A 中的所有元素与集合 B 中的所有元素进行比较

返回值: 矩阵 M, $M[i,j] = F(A[i], B[j])$

科学与工程的许多领域都存在着类似于 All-Pairs 的问题,例如,生物识别和数据挖掘技术。解决 All-Pairs 问题充满挑战,想要得到有效率的执行结果必须克服以下一些困难。

(1) 调度延迟。在一个传统的小批量系统中分发一个业务的成本的代价高得惊人(更不用说大尺度的网格层面)。

(2) 失效概率。不能让一个单独的业务运行时间太长。任何计算机系统都存在着硬件发生故障的可能性,但是共享计算机也有可能出现一个业务进程被更高优先级的业务抢占

的情况,这样通常将导致这个业务回滚到初始状态。

(3) 计算节点的数量。人们很容易地认为计算节点越多越好,事实上这个并不正确。在任何分布式系统中,每增加一个节点意味着用于交换的额外开销增加。

(4) 数据分发。在选择适当数量的服务器之后,我们必须明确每个计算节点应该分配多少数据。一个传统的集群有一个中央文件服务器,这样可以更好地访问数据。

(5) 隐藏的资源限制。分布式系统充满了意想不到的资源限制,这些限制会让粗心的用户陷入困境。

(6) 失败。在任何一种分布式系统,失败不仅常见,而且很难界定。

为了避免出现上述问题,共享计算系统的用户应该接受一个抽象规范,并且在现有资源的范围内选择一个引擎,特别是必须将数据传送到运行的引擎处。

All-Pairs 引擎原型运行在小批量系统上,并可利用本地存储连接到每个 CPU。这个模型包括以下 4 个步骤。

(1) 建立模型系统。在传统系统中,很难预测工作的性能,因为这取决于多种因素,如每个工作的详细的 I/O 行为、网络的行为等。如果使用一个抽象预先对本地存储进行有效地分配,而不是在系统运行期间通过网络来分配,那么就能很好地减小以上不利因素带来的误差。

(2) 分发数据。对于大数据量的并行运算而言,找一个计算节点只是需要考虑的一个方面。大数据集通常不会在每个计算节点复制数据,所以必须预先在这些节点放置数据。

(3) 调度批处理作业。将输入数据转发到选择好的节点后,All-Pairs 引擎则为每个分组作业构造批量提交脚本,并指示它们运行在存有数据的节点上。虽然批处理系统在这个阶段占有主动权,但是更高级别的抽象可以提供更多的性能、函数等。

(4) 收集结果和清理。随着批处理系统完成作业,抽象引擎必须收集得到的结果并以适当的结构组装起来。通常是一个单独的文件清单,以便得到一个规则的所有结果。这又提供了一个机会来增加容错:如果任何函数的输出与模板所提供的不匹配,那么这个作业将被重新提交直至得到正确的结果给用户。

为了使用抽象来对数据密集型计算进行评估,作者建立了一个 All-Pairs 引擎原型,并且在生物识别和数据挖掘中使用该模型采用了积极存储的策略,即预先将数据存储在每个计算节点的本地文件系统中。通过引用本地副本数据,避免了长期为每个接入网络分配 I/O 资源的负载。在共享计算系统中评价一个大工作量的性能需要克服一些挑战,除了资源的异构性,还有时间差的问题。在共享计算环境中,交互式用户可以优先使用 CPU,大量的批处理用户将竞争余下的 CPU 使用权。通过对比积极存储和按需进行存储两种方式的效果,结果表明,对于 4010×4010 标准生物识别的工作量,积极存储速度更快、效果更好。

10.3.2 GridBatch 编程模型

企业为了提高竞争力需要收集和分析海量的数据,这种高要求使企业面临两方面的挑战:一是数据量增长的速度超过了单处理器的计算力增长的速度;二是随着云计算的使用,企业需要调整其已有的体系架构。在这种情况下,Huan Liu 和 Dan Orban 等^[66]提出了 GridBatch 系统,该系统解决云基础设施环境下大规模数据批处理问题。GridBatch 是一个

编程模型,其中包含相关的并行编程库。该模型使用户能够完全控制数据的分发和计算力的分配,这样应用程序将获得很高的性能。

在 GridBatch 中有表和索引表两种基本数据类型,表中包含一系列的记录,这些记录相互独立,所有的记录遵照相同的格式。每一个记录都包含若干的域,域相当于表中的列。索引表类似于表,只是每个记录有一个相关的索引。GridBatch 系统包含两个相关的软件组件,分别是 DFS(分布式文件系统)和作业调度器。

DFS 负责文件的管理,同时把它们存储到系统中的节点内。一个大的文件通常分解成许多小块,每个小块被存储到单独的节点上。系统中的节点有两类,分别是名字节点和数据节点。名字节点保存文件系统的命名空间,维护一个从 DFS 文件到块列表的映射,从块列表可以知道,文件块被存储在哪个数据节点上及该数据节点的具体位置。数据节点保存文件的数据块,当需要对数据块进行读写操作时,DFS 客户端首先和名字节点进行通信,获得文件块被存储的位置列表,然后与数据节点通信,完成数据的读/写操作。

分布式文件系统是 Google 文件系统的一个扩展,它除了支持 GFS 中的文件类型外,还支持新的文件类型。DFS 主要存储两种类型的文件:固定块大小的文件和固定块数目的文件。固定块大小的文件和 GFS 中的文件类型相同,每个块的大小是 64MB。固定块数目的文件是指每个文件被分成含有固定块数目的文件,而每个块的大小是任意的,每个固定块数目的文件都含有一个相关联的分割函数。之所以在数据分析应用中提出固定块数目的文件类型,是因为大的分析应用中数据都是结构化的,新的类型便于数据的合理、高效地处理。

DFS 类似于 GFS,在系统中对数据块做了备份。当有节点失效时,系统中的数据不会丢失,系统拥有良好的数据恢复和容错能力。

作业调度系统中包含一个主节点和许多从节点,主节点负责把作业分解成许多小的任务,并将任务分发给从节点,同时对从节点进行监控,而从节点负责对小任务进行执行。

GridBatch 系统将 MapReduce 分解成更多更细的控制器,GridBatch 包含以下几种控制器:Map 控制器、分发控制器、递归控制器、合并控制器、笛卡儿控制器和 Neighbor 控制器。Map 控制器对表中的所有记录应用一个用户定义的函数;分发控制器把表或索引表转化成另一种具有不同索引的表;合并控制器把索引域匹配的索引表进行合并;笛卡儿控制器应用用户定义的函数把表 X 和表 Y 中的每条记录进行匹配;Neighbor 控制器将相邻的记录进行分组,同时调用用户定义的函数分析其序列。

GridBatch 是基于 Hadoop 的一个实现,在 Amazon 的 EC2 云平台上对 MapReduce、GridBatch 和单节点三种情况下对数据复制的性能进行的测试表明,GridBatch 较 MapReduce 和单节点有显著的提升。通过对 GridBatch 系统中数据吞吐量的性能进行测试,发现随着节点数量增加,吞吐量基本呈线性增长。测试和对比分析表明,GridBatch 在云基础设施环境上有很好的性能。

10.3.3 其他编程模型

Yahoo 公司扩展了 Map-Reduce 框架,在 Map-Reduce 步骤之后加入一个 Merge 的步骤,从而形成一个新的 Map-Reduce-Merge 框架^[74]。Map-Reduce-Merge 框架的数据和控制流如图 10-12 所示。其中,协调器(Coordinator)管理 mapper 和 reducer 操作的集合在任务结束之后,协调器还要发起一系列的 merger 操作,从挑选出来的 reducer 操作中读出输出结果并按照用户定义的逻辑进行合并。

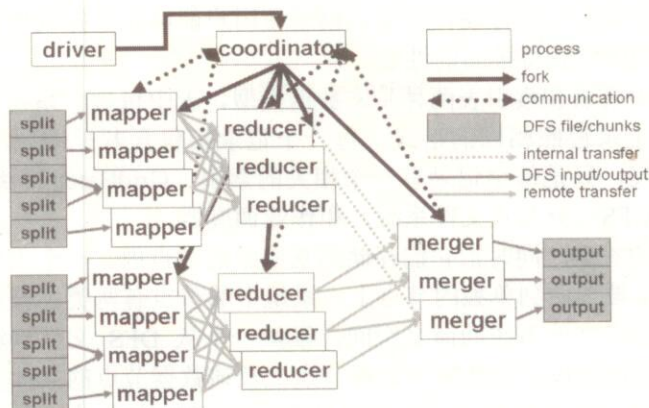


图 10-12 Map-Reduce-Merge 框架的数据和控制流

Map-Reduce-Merge 框架是 Map-Reduce 框架的扩展，在增加 Merge 操作的同时定义了额外的用户自定义的操作。这些操作包括：分区选择，即选择何种数据进行合并；处理器，用来处理数据；Merger 操作，定义合并的逻辑；可配置的迭代器，即如何在合并过程中逐步操作每个列表。Map-Reduce-Merge 框架实现了关系代数的操作，例如映射、聚合、选择、笛卡儿积、并、交、差等。这样应用程序开发人员就可以自主提供 Merge 函数来做两个数据集的任何关系代数操作。

Stanford 大学的研究人员将 MapReduce 的思想应用到多核处理器上。主要工作是在多核处理器的基础上构建了一套 MapReduce 的编程框架。Phoenix^[75]将 MapReduce 应用到内存共享系统上，它包括一个编程 API 和高效的运行库系统，通过使用线程来创建 Map 或 Reduce 任务，无需复制大量数据便可通过使用内存共享缓冲区进行任务交互。对于线程创建、动态任务调度、数据分区及跨处理节点的容错，Phoenix 运行时（Phoenix Runtime）都能进行动态管理，运行时调度任务动态通过访问可用的处理器来达到负载均衡和任务吞吐量的最大化。在重复或重新分配任务并正确合并输出其他计算时，运行时会自动恢复瞬间或永久故障，同时 Phoenix 还为程序员提供类似顾客数据分区的专业应用。

10.4 支撑平台研究

随着云计算的不断发展与成熟，研究机构和科研人员在云平台设计与实现方面做了大量的研究实践工作，满足不同应用需求的云计算平台不断涌现并投入使用，目前已经有很多典型的研究案例。

10.4.1 Cumulus: 数据中心科学云

Lizhe Wang 等^[9]通过采用新的云技术整合现有的网络基础设施来为数据中心建立一个科学云。Cumulus 是一个不断发展壮大的云计算项目，主要为科学计算应用程序提供虚拟机、虚拟应用程序和虚拟计算平台。

Cumulus 的体系结构如图 10-13 所示。Cumulus 前端节点作为 Cumulus 的访问点，接受用户虚拟机操作的请求。OpenNEbula 作为本地虚拟机管理系统，用于管理分布式的服

务器。OpenNebula 通过 SSH 与 Globus 虚拟工作空间服务、后台程序及主机上的 Xen 系统程序进行通信。操作系统 Farm 是产生和存储 Xen 虚拟机镜像和虚拟应用程序的服务。该系统中把操作系统 Farm 作为一种虚拟机模板管理的工具。

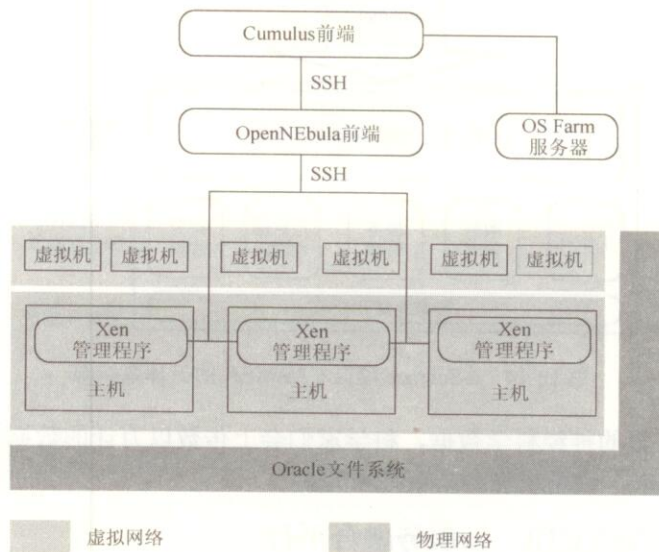


图 10-13 Cumulus 的体系结构

Cumulus 前端节点是 Globus 虚拟工作空间服务的再实现。Globus 虚拟工作空间服务包括两种软件：一种是工作空间服务前端节点，其作用是接收虚拟机的请求，同时把请求分发给各种后台服务器；另一种是工作空间控制代理，安装在后台服务器上，通过与 Xen 服务器通信来配置工作空间。为了使 Globus 虚拟工作空间服务适应 Cumulus，系统做了如下改进：①删除了控制代理，Globus 虚拟工作空间服务直接与安装在后台服务器上的虚拟机管理程序进行通信；②扩展了 Globus 前端服务，使其能与各种虚拟机管理程序协同工作；③支持新的网络解决方案，即转发模式，用户不需要输入网络配置信息，后台服务器为虚拟机分配 IP 地址，然后把信息反馈给用户。

系统在 OS Farm 服务上建立虚拟机镜像库，OS Farm 为云用户提供了两个界面：用户能输入虚拟机镜像构建参数的 Web 界面和能通过 wget 访问的 HTTP 服务。

Cumulus 为用户提供了两种访问方式：一种是 Globus 虚拟工作空间服务客户端，另一种是通过网格计算工作台或网格门户。

10.4.2 CARMEN: e-Science 云计算

全球超过 10 万的神学家正在全力攻克大脑如何工作这一难题，这是一项重大挑战，可能给生物学、医学和计算机科学带来革命性的变革。研究大脑如何工作就必须弄清楚大脑是怎样编码、传输和分析数据。Paul Watson 等^[13]概述了 e-Science 项目 CARMEN 的云体系结构，该项目供神经学家共享、整合、分析数据。CARMEN 的云体系结构如图 10-14 所示，总共分成四层。第一层是用户，通过 Web 浏览器和富客户端访问 CARMEN 系统；第二层是领域内的特定服务；第三层是 e-Science 核心云服务，包括工作流、数据管理、服务管理、元数据管理和安全组件；第四层是核心云服务，包括基本的存储和处理。

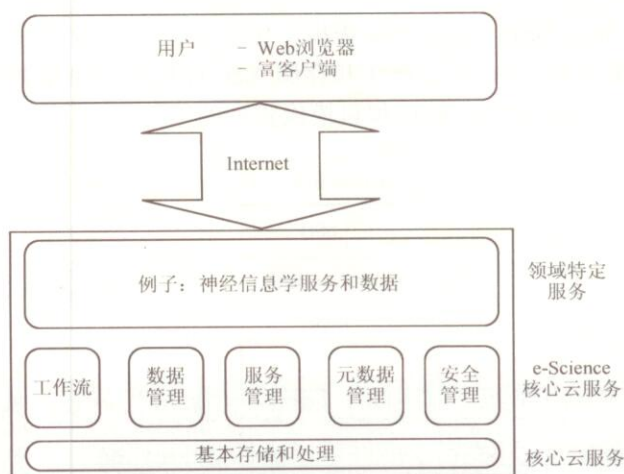


图 10-14 e-Science 项目 CARMEN 的云体系结构

为了分析 TB 级的神经科学数据，科学家们会上传数以万计的数据集和服务，通过总结科学家的工作需求，该系统将会不断地被完善。

10.4.3 RESERVOIR：云服务融合平台

IBM 与 SAP、Sun Microsystems 及其他若干欧洲科研机构联合开展一项名为 RESERVOIR 的云计算计划，欧盟投资经费为 1700 万欧元。该项目旨在建立一种“基于服务的网络经济模式”，具体内容是通过虚拟化技术、分布式虚拟管理和企业服务管理技术，实现跨越地域和平台的 IT 应用和服务，以支持基于服务的在线经济，对资源与服务进行透明式配置与管理。

RESERVOIR 研究了当前无法实现的商业服务对系统性能的需求。RESERVOIR 在公开标准基础上建立一个具有可扩展性、灵活性、可靠性的框架^[14]，以提供云计算服务，其体系结构如图 10-15 所示。

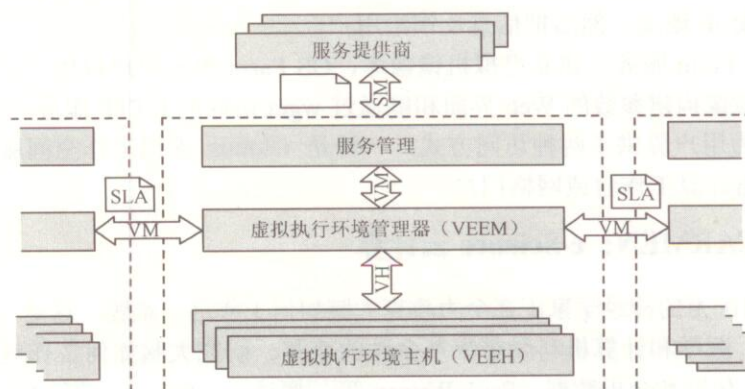


图 10-15 RESERVOIR 体系结构

服务管理器是最高级别的抽象层，它从服务供应商那里获得服务清单后，与其谈判定价，最后处理账单，其主要工作是根据服务清单部署和配置虚拟执行环境 (VEE)。虚拟

执行环境管理器 (VEEM) 是下一个抽象层, 与上层的服务管理器、下层的虚拟执行环境主机 (VEEH) 和其他站点的 VEEM 进行通信, VEEM 负责优化配置虚拟执行环境到 VEEH。VEEH 是最低级别的抽象层, 负责对虚拟执行环境进行基本控制和监测。

10.4.4 TPlatform: Hadoop 的变种

Peng Bo 等^[12]设计实现了 TPlatform 云计算平台, 体系结构如图 10-16 所示。该体系结构分为三层, 分别是 PC 集群层、云计算平台基础设施层和数据处理应用层, 其中 PC 集群层为大规模数据处理提供了硬件和存储设备; 基础设施层包括自行设计的 TFS 文件系统、改进的 MapReduce 编程模型和类似 Google 的分布式数据存储机制 Bigtable; 应用层为用户提供了可开发应用程序的服务。

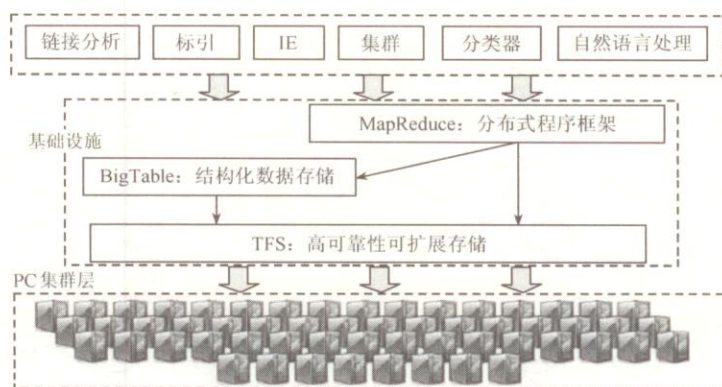


图 10-16 TPlatform 体系结构

该平台最大的特点是把大小不变的 GFS 固定文件块变为大小可变的 TFS 文件块, 这样的改进使得数据的分发和读取有更好的性能, 同时把 MapReduce 模型中的 Reduce 阶段中无控制的数据传送变为可调度的传送, 这种新的机制将避免由于网络拥塞导致的系统性能下降。

10.4.5 P2P 环境的 MapReduce

Fabrizio Marozzo 等^[11]提出了 P2P-MapReduce 的体系结构, 并概述了 P2P 框架下的实现过程。

P2P-MapReduce 的体系结构如图 10-17 所示, 它包含三个基本角色, 分别是用户 (User)、主节点 (Master) 和从节点 (Slave)。主节点和从节点形成了两个逻辑上的 P2P 网络 M-net 和 S-net。

图 10-18 为主节点失效时的作业提交和节点失效管理流程, 从中可以看出 P2P-MapReduce 的工作过程。

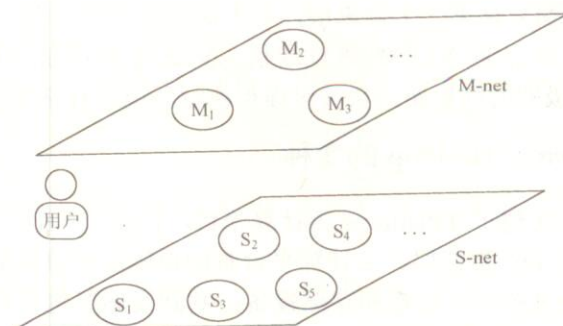


图 10-17 P2P-MapReduce 的体系结构

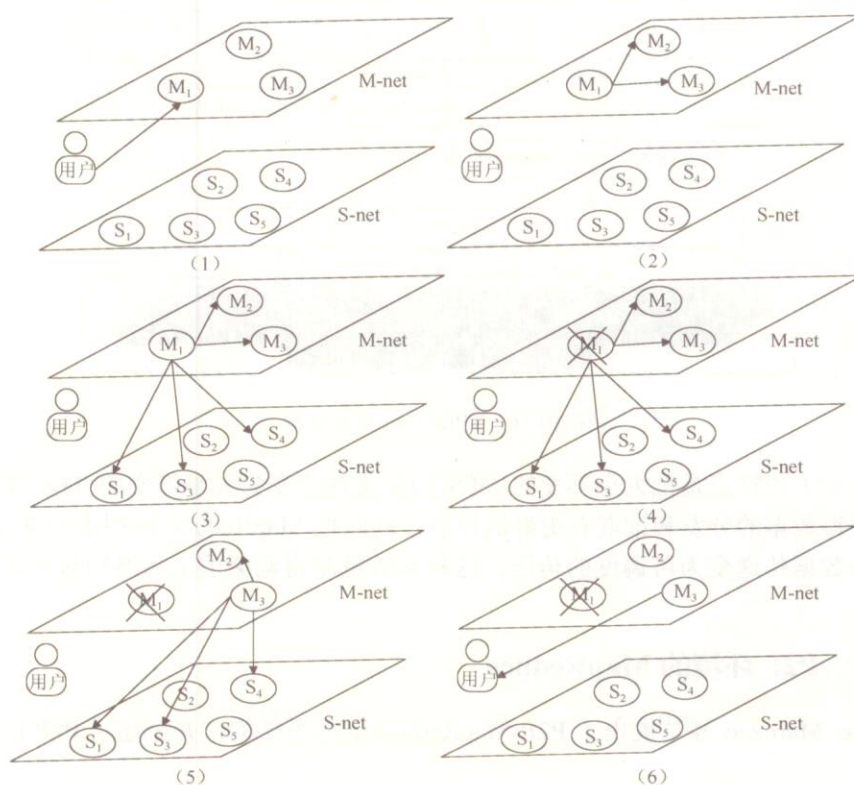


图 10-18 作业提交和节点失效管理流程

(1) 用户查询 M-net 中的可用节点，根据节点的繁忙程度对其进行排序，并从中选择最空闲的节点 M_1 作为主节点， M_2 和 M_3 作为备份节点，用户将 MapReduce 作业和备份节点的名字提交给 M_1 。

(2) M_1 通知 M_2 和 M_3 整个作业的状态，并告知它们是当前作业的备份节点，同时 M_2 和 M_3 定期检测 M_1 是否仍正常工作。

(3) M_1 查询 S-net 得到可用的从节点列表，选择全部或者部分从节点执行 Map 和 Reduce 任务。

(4) 当主节点 M_1 失效时, M_2 和 M_3 将检测到情况的发生, 并通过分布式过程选出新的主节点。

(5) 假设 M_3 成为新的主节点, 则只有 M_2 是备份节点, MapReduce 作业重新开始运行。

(6) 当作业执行完成后, M_3 把结果返回给用户。

10.4.6 Yahoo 云计算平台

Yahoo 构建了一系列可扩展、高度可用的数据存储和处理服务, 并将其部署在云模型中, 这使得应用开发和维护更加便利。Brian F. Cooper 等^[15]分析了 Yahoo 云计算的应用需求, 概述了其核心组件和体系结构。Yahoo 云平台有如下应用需求。

- (1) 在相同的硬件和软件基础设施上支持各种不同类型的应用程序;
- (2) 需要根据用户的需求动态的提供计算资源;
- (3) 需要很高的安全保证;
- (4) 云中的系统必须易于操作;
- (5) 提供简单易用的 API 接口;
- (6) 采用资源调度和负载均衡技术, 快速高效地为用户提供服务。

图 10-19 显示了 Yahoo 云计算平台的体系结构和核心服务组件。



图 10-19 Yahoo 云计算平台的体系结构和核心服务组件

Yahoo 计算平台包括三层服务, 分别是基于内容的边缘服务、消息服务和核心服务。边缘服务包括基于内容的边缘缓存服务和边缘路由服务; 消息服务层主要是把不同的服务有机地串联结合起来; 核心服务又包括三个系统, 分别是批处理系统、业务存储系统和配置系统。批处理系统主要管理大规模的并行作业; 业务存储系统管理应用程序数据的存储和查询, 将结构化的数据存储在 Sherpa 中, 而非结构化的数据则存储在 MObStor 中; 配置系统主要负责服务器虚拟机的管理和配置。

10.4.7 微软的 Dryad 框架

Dryad 系统^[76]主要用来构建支持有向无环图 (Directed Acycline Graph, DAG) 类型数据流的并行程序, 根据程序的要求进行任务调度, 自动在各个节点上完成任务其系统结构如图 10-20 所示。在 Dryad 平台上, 每个任务或并行计算过程都可以表示为一个有向无环图, 图中的每个节点表示一个将要执行的程序, 节点间构成的边表示数据通道中数据的传

输方式,其可能是文件、TCP Pipe、共享内存等。

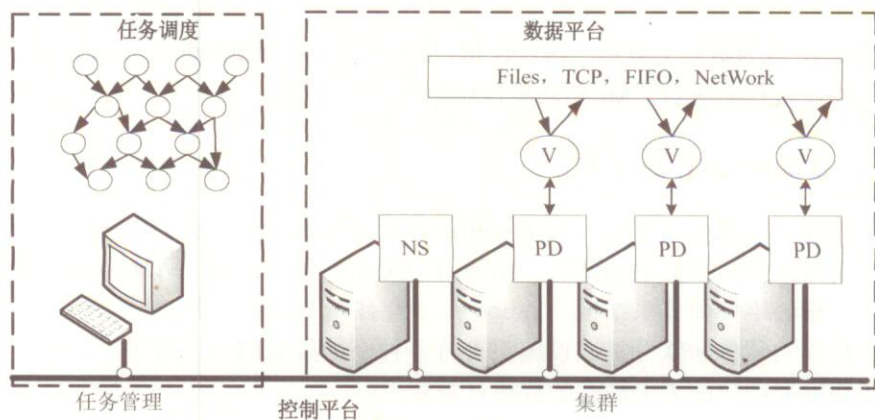


图 10-20 Dryad 系统结构

当用户使用 Dryad 平台时,首先需要在任务管理(JM)节点上建立自己的任务。每个任务由处理过程及在这些处理过程上的数据传递组成。任务管理器获取无环图之后,便会在程序的输入通道等待,当有可用机器时便开始对它进行调度。JM 从命名服务器(NS)处获得一个可用的计算机列表,并通过维护进程(PD)来调度这个程序。

Dryad 的执行过程可以看做二维管道流的处理过程。其中,每个节点可以执行多个程序,这样可以同时处理大规模数据。每个节点进程(Vertexes Processes)上都有一个处理程序在运行,并且进程相互之间通过数据管道(Channels)的方式传送数据。二维的 Dryad 管道模型定义了一系列的操作,包括建立新的节点、在节点之间加入边、合并两个图或是对任务的输入和输出进行处理等。这些操作可能会对有向无环图的创建和修改产生影响。

10.4.8 Neptune 框架

KaiShen 等^[77]研究的 Neptune 是一个基础结构中间件,通过提供标准系统的构建来保证可扩展性和有效性,同时减少服务结构的灵活性。它为存在的访问和复制服务模块提供了灵活的接口。它容纳各种各样的底层存储机制,并且保持本地透明服务的映射,增强了故障模块的管理和复制一致性。此外,它还可通过失效控制来保证多级别复制一致性模块的高效使用。

Neptune 具有信息独立和用户独立的优点。它通过一个包含 RPC-like 访问方式的服务访问接口来压缩应用级别的网络服务。通过服务访问一个模块,可以在数据分区上实现独占。简单 Neptune 服务集群的结构如图 10-21 所示。

图中,阴影部分为 Neptune 客户端模块,服务集群通过网络服务和 WAP 网关将讨论组和相册服务交付给大规模浏览器和无线客户端。通过用户报告,将持久数据分成 20 个部分。当相册服务依靠一个内部相片存储服务时,讨论组服务单独交付。这样一来,每个图片相册节点需要控制一个 Neptune 客户端模块,并以此来定位和访问服务节点中的图片服务。为把内部服务展现给外部客户端,Neptune 客户端模块也代表每个网关节点。凭借松耦合连接和结构,可以在短暂失效和业务引进时,对 Neptune 服务基础结构进行控制。

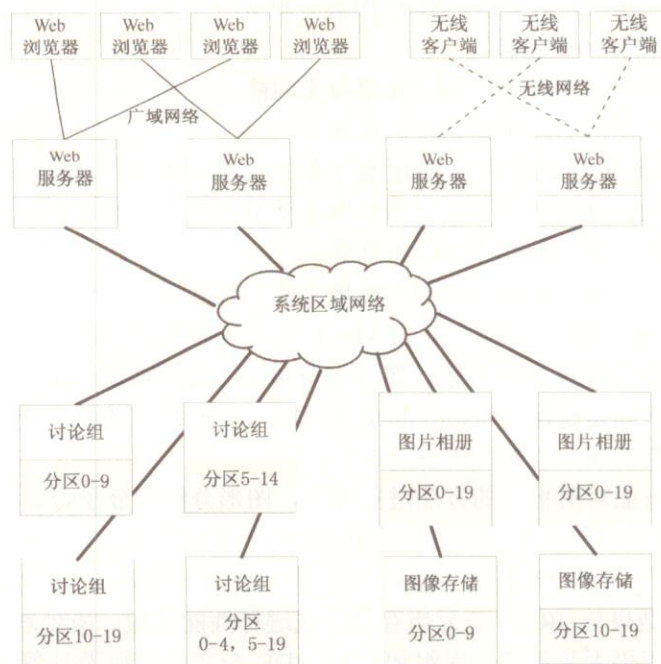


图 10-21 简单 Neptune 服务集群的结构

10.5 应用研究

10.5.1 语义分析应用

对大规模的 RDF 三元组进行语义推理和查询的速度之慢已深受广大研究者关注，文献[23]针对该问题提出应用 Google 的 MapReduce 框架来实现大规模分布式查询和推理，这是云计算的又一典型应用，该方法要求将 RDF 图分解成能够被计算节点处理的小单位。RDF 分子法^[24]提供一个处于 RDF 图和三元组之间的中间化的粒度方法，由于 RDF 分子的自身缺陷，使得其性能受到限制，该文献提出了将 RDF 进行扩展的方法，包括层次扩展和顺序扩展，以克服其缺陷，并给出了该方法在 MapReduce 中的一些实现细节。

文献[23]提出基于 Hadoop，分布式 RDF 分子存储以下内容。

- (1) 将 RDF 图分解成可以由集群节点进行分发、索引的小单元。
- (2) 用 SPARQL 作为查询语言来查询集群中的每个节点。
- (3) 从各个节点处收集并合并生成查询结果。

RDF 分子能够完成对 RDF 图的无损分解和合并，保持其语义性的同时适合 MapReduce 架构的分布式处理过程。因此，RDF 分子具有如下功能。

- (1) 提供一种方法使得 RDF 图可以由计算节点进行分解。
- (2) 能够从许多节点聚合所返回的查询结果。
- (3) 提供一个最小数据集以支持同步视图修改。
- (4) 提供基于上下文的空节点区分方法。

在原始的 RDF 分子的定义中, RDF 图被分解成分子集, 每个分子由三元组集组成。但是这个设计存在如下一些缺陷。

- (1) 缺乏消除带有两个空节点的三元组歧义的能力 (主体和客体)。
- (2) 缺乏对原始图的三元组结构表示形式。

文献[23]针对上述的缺陷对原始 RDF 分子的定义进行扩展, 通过增加层次和排序来弥补这些缺陷。层次的、嵌套的分子使得带有两个空节点的三元组可以通过分子的上下文进行区分。更重要的是, 分子的合并将更加有效。

针对大规模数据集进行有效的查询和推理是极具挑战性的工作, 尤其是面对动态用户时。语义 Web 技术如 RDF、OWL 和 SPARQL 由于其提供共享的、明确的和可扩展的解决方案, 是完成数据集成任务的首选, 像 MapReduce 这样的分布式算法已经表明其处理大规模数据的经济性和实用性。MapReduce 和语义 Web 技术的结合提供了一个针对大规模异构数据集成、查询和推理的完美解决办法。

该文献从三个方面对所提出的方法进行评估, 图形分解和分子合并、MapReduce 的性能和 SPARQL 查询反馈。该方法在图形分解和分子合并过程中, 与 Jena 进行对比, 当规则链数达到 100 时, 其速度高于 Jena, 可以获得更高的效率。对 RDF 分子存储运行 SPARQL 查询, 将结果与 JRDF 三元组存储方式进行性能比较, 所处理的 RDF 三元组数量有轻微增长。随着更大数量的三元组加载到 RDF 分子存储以及计算集群的增长, 可以预计传统 RDF 三元组的存储性能也将有相对的提高。

10.5.2 生物学应用

当前的生物信息科学应用要求具有海量数据管理和实现大规模计算的能力。为满足这些要求, 需要简单易行的实现并行计算的方法。MapReduce 是一种通用的并行技术, Hadoop 提供了具体的开源实现。由于生物信息科学中大量的数据在执行过程中不需要频繁修改, 所以 MapReduce 的单写多读技术 (Write Once Read Many, WORM) 非常适合应用在生物信息科学的计算问题中, 不论是大规模数据集的流计算还是小规模数据集的多路计算。

Massimo 等^[29]用 Hadoop 实现了两个算法来研究解决生物信息计算的问题: 一个是基本的局部相似性基本查询工具 (Basic Local Alignment Search Tool, BLAST); 另一个是基因集合增强分析 (Gene Set Enrichment Analysis, GSEA)。

BLAST^[30]是一个用来比对生物序列的一级结构的算法, 它以一个或几个蛋白质或核酸序列为检测序列, 搜索蛋白质或核酸序列数据库, 寻找与检测序列中一个或多个片段具有较高相似性的一组序列, 这种算法已经在生物信息领域得到了广泛的应用, 如 Soap-HT-BLAST^[32]、mpiBLAST^[33]、GridBLAST^[34]、W.NDBLAST^[35]、Squid^[36]、ScalaBlast^[37]。Massimo 等^[29]指出这些应用都不具备通用性, 并且他们的安装和维护比较复杂, 因此, 将 BLAST 移植到 Hadoop 上, 只需为 BLAST 创建一个可执行的映射 (Mapper), 将队列集转换成一行一列的形式。映射函数只需要从标准输入中一次读入一个队列, 与查询队列进行比较, 然后输出结果队列即可。

GSEA^[31]是 DNA 微点阵数据分析工具, 用来检测在显性环境中, 整个基因序列之间的相互关系, 通过使用 Hadoop, 用 Python 重写 GSEA 算法。在 MapReduce 应用执行之前, 需要经历两个阶段的预处理过程: 生成类标识向量的 N 随机序列, 并将它们放入文

件中建立索引。预处理完成后, Mapper 从标准输入和文件组中读取标识向量序列, 计算每个基因序列的富集度, 并输出数据流, Reducer 读取每个基因序列的观察和置换数据来计算 p 值。

10.5.3 数据库应用

随着云计算的兴起, 用户可以在云平台上部署的软件越来越多, 这其中就包括数据库应用。如今比较流行的云计算服务类型是将基础设施作为服务 (IaaS), 最著名的商业 IaaS 是 Amazon EC2, 将数据库应用部署到这种服务平台上存在诸多挑战^[41]。

1. 部署过程

当创建一个数据库应用副本的虚拟机时, 需要赋给新的虚拟机和其上的数据库系统一个特殊的认证, 同时需要将数据库实例化。然后, 必须确保数据库应用请求指向正确的端口并且不会被防火墙拦截。除此之外, 虚拟机必须能够识别所有需要与它连接的用户的身分认证, 且必须和其运行的云平台之间保持相对独立性。

2. 调整过程

虚拟化技术使云服务提供者能够在任何物理设施上运行用户的虚拟机, 虚拟机和物理设备之间的映射对性能有很大的影响。所以有一些问题必须要考虑: 第一个是需要决定在每个物理设施上运行虚拟机的数量, 重要的是在最小化物理设施数量和用户能够接受的性能之间找到平衡; 第二个问题是这些虚拟机的资源需求, 如通过算法避免将多个 I/O 密集型虚拟机映射到同一个物理设施, 以此来最小化虚拟机之间的 I/O 交互, 这是一种适合数据库系统的模式, 但是不一定适合其他的应用。

另外, 在每个物理设施上的虚拟机之间如何分配资源也是需要考虑的, 虽然大多数虚拟机监控器提供了工具和 API 接口控制物理资源的分配, 为了得到最好的性能, 考虑应用的特性是有必要的。

除此之外, 提高表达不同的服务水平对象的能力有利于优化云计算环境中数据库应用的性能, 不同的工作负载可以具有不同的服务水平对象, 如此可以使用最少的云资源达到合适的数据库应用性能。

具有不同服务水平对象的工作负载的动态性是否存在更加简洁和实用的工作负载的表达式, 是否能够在没有 SQL 表达式的知识指导, 这些情况下决定调整决策, 当工作负载的性质变化时是否能够准确检测, 这些都是很重要的问题。

解决这些问题需要如下一些工具和技术。

(1) 运用模型。一个精确、高效的运用模型可以预测不同的调整策略的效果。模型主要有两类: 白盒模型 (基于数据库系统的内部知识) 和黑盒模型 (基于数据库性能的外部观察的统计模型)。也有研究探讨将两种模型结合起来, 将数据库系统的内部模型作为起点, 然后通过数据库性能的外部观察进行调整^[42]。

(2) 优化和控制算法。在云计算环境中解决性能调整问题需要发展综合优化和自动控制算法, 可以是假定固定工作负载的静态算法, 也可以是适合工作负载变化的动态算法。

(3) 系统管理工具。除了模型和算法, 需要部署和调整数据库应用的系统管理工具, 这些工具能够得到虚拟机的性能特征和数据库系统的性能特征。

(4) 综合调整和暗示传递 (Co-tuning and Hint Passing)。除了调整虚拟机的参数, 数

数据库系统的参数调整也是至关重要的,因此需要对两类参数进行综合调整 (Co-tuning)。另外一种协调虚拟机和数据库系统的方法是传递暗示 (Hint Passing),数据库能够很容易获得暗示中包含的信息并且在虚拟层进行调整。

10.5.4 地理信息应用

Qichang Chen 等^[43]提出了高性能工作流系统 MRGIS,这是一个高效率执行 GIS 应用程序的基于 MapReduce 集群的并行和分布式计算平台。MRGIS 系统由设计接口、任务调度器和运行时间支持系统组成,如图 10-22 所示。设计接口包括两个组件,分别是基于 GUI 的工作流设计器和 Python 中基于 API 的编程库。当给定 GIS 工作流后,首先调度器分析任务间的数据相互关系,然后把它们分解给 MapReduce 集群进行执行。

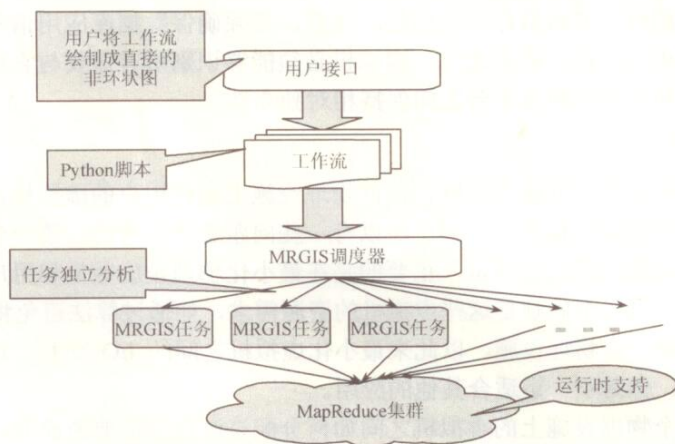


图 10-22 MRGIS 的体系结构

为了测试 MRGIS 系统的性能,作者设计了两组实验。第一组实验是 GIS 工作流中典型的代数运算,测试输入是单独的数据集,数据采用 Arc/ASC 格式。第二组实验的测试输入数据集从远程主机上获得,工作流执行 18 个任务。测试结果如图 10-23 所示。

第一组实验中采用 MRGIS 系统执行的速度比采用一台计算机执行的速度快很多。数据从 3 块增加到 11 块的过程中,系统性能连续提高。但是当数据增加到 15 块时,性能却有所下降,这是因为需要执行任务的数量超过了集群中空闲机器的数量,任务需要排队等待。

同样,第二组实验中采用 MRGIS 系统执行的速度比采用一台计算机执行的速度快,但是随着数据块的增加,系统性能不像第一组实验中那样提高明显,这是因为一方面等待执行的任务超过了空闲机器的数量,另一方面第二组实验中的工作量明显比第一组实验中的工作量大。

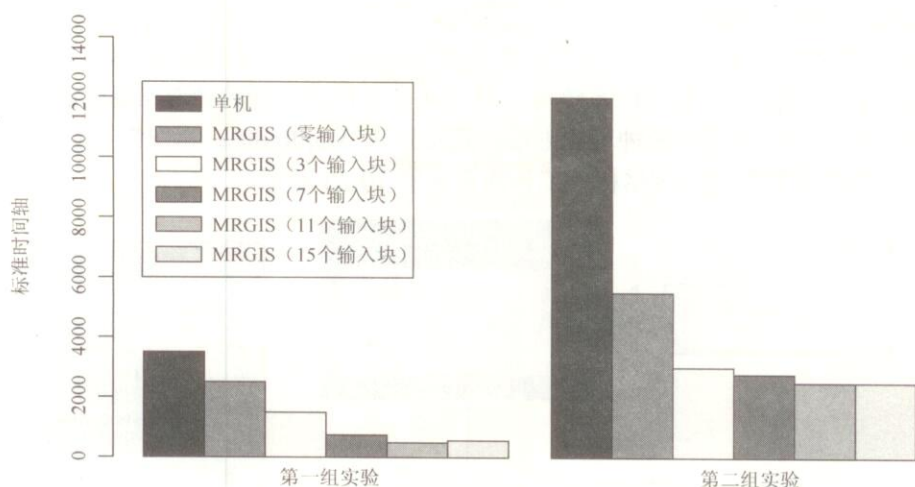


图 10-23 不同数据块情况下 MRGIS 系统的性能比较

10.5.5 商业应用

随着企业中半结构化和非结构化数据的大量增长，急需研究开发有效的商业应用。传统的企业架构或分析都面向结构化数据，没有通过对半结构化数据进行大规模计算来分析数据的设计。在 IBM 阿尔马登研究中心，研究者开发了一个企业文本分析平台，应用 Hadoop MapReduce 框架来支持分类工作。该平台包含两个核心组件：T 系统和 Jaql。T 系统是基于规则的高性能信息抽取系统，Jaql 是半结构化数据的转换语言。

该平台的体系结构如图 10-24 所示。除了 T 系统和 Jaql 外还包含其他一些组件。

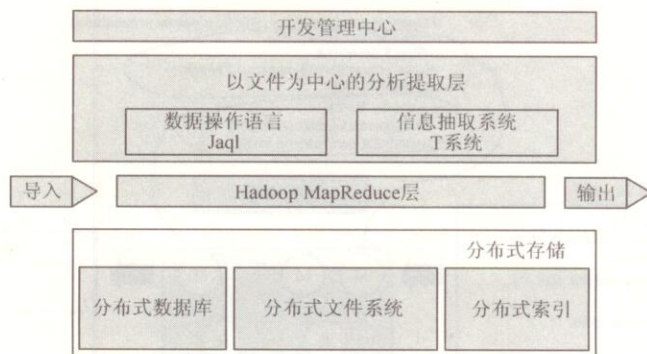


图 10-24 IBM 企业文本分析平台体系结构

在进行分析之前，文本就已经被获取并写入到分布式存储服务中了。在文献[44]中，研究者详细介绍了两个核心组件的工作原理和设计方案，并对设计中分布式运行时间、分布式存储、用户交互中心等方面进行了分析，指出实现这些组件存在的挑战。平台的两个核心部分分别介绍如下。

1) 用 T 系统进行信息抽取

T 系统是一个基于规则的信息抽取系统，由 IBM 阿尔马登研究中心于 2006 年开发。

T 系统用来从非结构化文本中抽取结构化信息, 并发现其中的一些关系。T 系统的核心是声明规则语言 AQL, 用来构建抽取器, 构建好的抽取器再由 T 系统编译器进行编译, 得到一个基于代数的执行引擎。T 系统可以通过从多文件中并行的抽取信息来适应大规模文本集。系统的当前版本支持两种不同方法, 直接嵌入 MapReduce 和作为 Jaql 查询的一部分进行并行执行。T 系统内部结构及工作原理如图 10-25 所示。

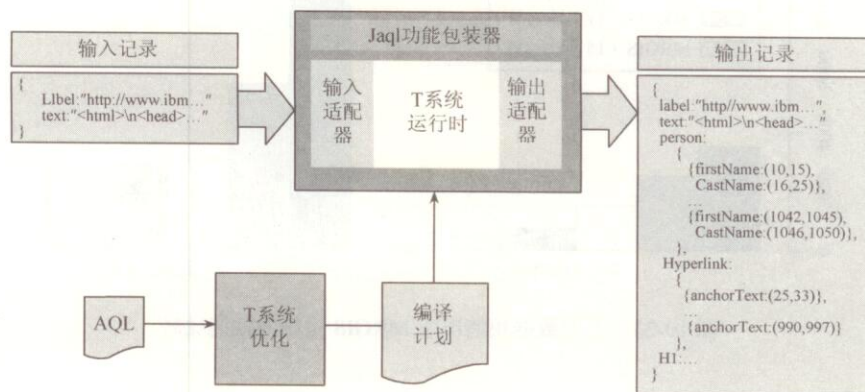


图 10-25 T 系统内部结构及工件原理

2) 用 Jaql 进行数据处理

为了方便信息的提取、分析、转换和数据导出, 需要一种轻量级描述语言, 该语言要支持半结构化数据并具有易扩展性。出于这个目的, Kevin Beyer 等^[44]设计了一种通用数据流语言 Jaql。图 10-26 展示了一个 Jaql 查询的实例, 用于计算文档中每个人被提及的次数。用 T 系统检测人名, 再用 Jaql 来计算统计次数并生成输出。

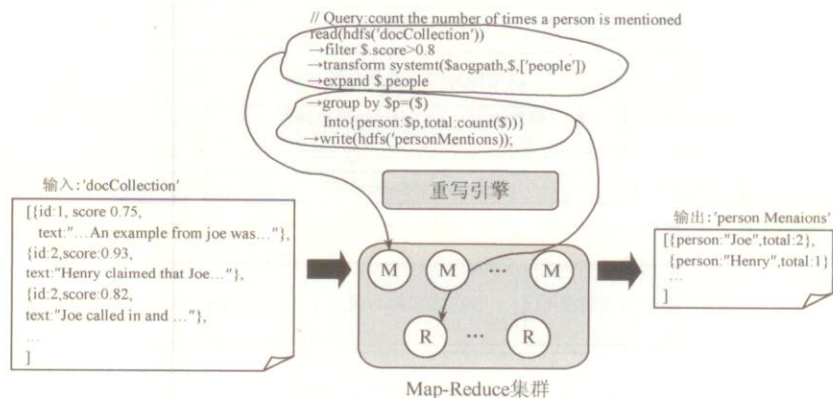


图 10-26 Jaql 查询实例

10.5.6 医学应用

Kathleen Ericson 等^[79]提出了运用云计算技术 Granules 分析脑电图的思想。作者采用 Map-Reduce 框架进行开发, 运用云运行时 Granules 分类脑电图数据流并对其进行实时处理, 以达到在分布式环境中训练神经网络, 并通过筛选多用户的脑电图信号来推断

他们预期行为的目的。脑机接口 (BCIs) 根据大脑产生的脑电图信号与计算机进行交互, 并且允许用户发起操作, 如键盘输入或是控制轮椅的移动等。BCI 软件是用 R 语言进行开发的, R 语言是一种为快速计算矩阵连乘而设计的解释性语言, 也是智能神经网络发展的有效语言。

Granules 是云计算的一个轻量级运行时, 被设计用来管理云中的大量计算, 如处理感应器产生的数据。Granules 支持云计算中两种主导的模式: MapReduce 和数据流图。Granules 单个计算根据输入数据集或外部触发器结果中数据的可用性来改变状态。当处理完成之后, 计算就进入休眠状态, 然后等待其他的输入数据集上的数据。在 Granules 中, 计算指定了调度策略, 而调度策略又管理计算的生命周期。计算从以下三个指标来指定调度策略: 计数、数据驱动以及周期性。

运用 Granules 分析脑电图信号有两方面优势:

(1) Granules 能够在单个机器上对多用户数据流进行交替并做伴随处理, 减少了成本。在目前 BCI 的实现中, 每个用户都有一个专有的处理单元。

(2) Granules 在一系列可用机器上管理计算, 甚至是一个中型的集群都能支持相当多的用户。所以来自大量用户的脑电图数据能够训练神经网络, 进而提高相关算法的准确性。

为了验证在分布式环境中运用 Granules 来分类脑电图数据流的适用性, 作者将 Granules 的处理方法与 Snowfall (基于 SNOW (Simple Network of Workstations) 的 R 包) 处理方法进行了对比, 并且设计了实验去评估训练神经网络和分类脑电图信号计划所带来的成本。

实验结果如表 10-4 所示, 表明在脑电图数据流分类上, Granules 相对于 Snowfall 有更好的分类效果, 并能够对脑电图数据流进行实时处理。

表 10-4 分类时间对比

方法	数据流时间 (ms)	平均时间 (ms)	最小时间 (ms)	最大时间 (ms)	标准时间差 (ms)
Snowfall	5s	8884.60	8797.745	9.69.47	85.82
Granules	5s	141.69	136.42	266.63	12.75
Snowfall	1s	5825.71	5815.38	5857.98	10.09
Granules	1s	93.16	47.51	492.68	32.13
Snowfall	250ms	2831.32	2830.38	2849.83	2.03
Granules	250ms	87.25	48.57	92.67	4.49

10.5.7 社会智能应用

Dexter H.Hu 等^[80]提出了 BetterLife 2.0 框架的设计, 其体系结构如图 10-27 所示, 主要包括三个部分: 云层、基于案例的推理机、应用程序接口。云层由 Hadoop 分布式文件系统 (HDFS) 集群组成。Hadoop 数据节点共同存储由案例和社交网络信息代表的应用数据, 包括关系拓扑和成对的社会紧密度 (Social Closeness) 信息。基于案例的推理机是根据典型的 CBR 框架 jCOLIBRI2 延伸而来, 有到云层中的数据连接器, 并且在案例之间进行相似度测量以获得最相似的两个案例。此外, 它还可以用 HBase 服务将新案例存回云层。应用程序接口用一个主节点来处理用户的请求查询, 然后将查询任务分发给集群中其他的服务器 (Map), 再由这些服务器接收计算结果 (Reduce)。在客户端有两种类型的端

口。一种是网络接口,它由社交网络扩展而来,为用户创建文件并产生和记录社会活动,编辑用户数据等,所有的这些信息都将存储到云层中。另一种是移动电话上的移动应用程序,供用户上传或是修改用户环境,查询和获取服务器建议。

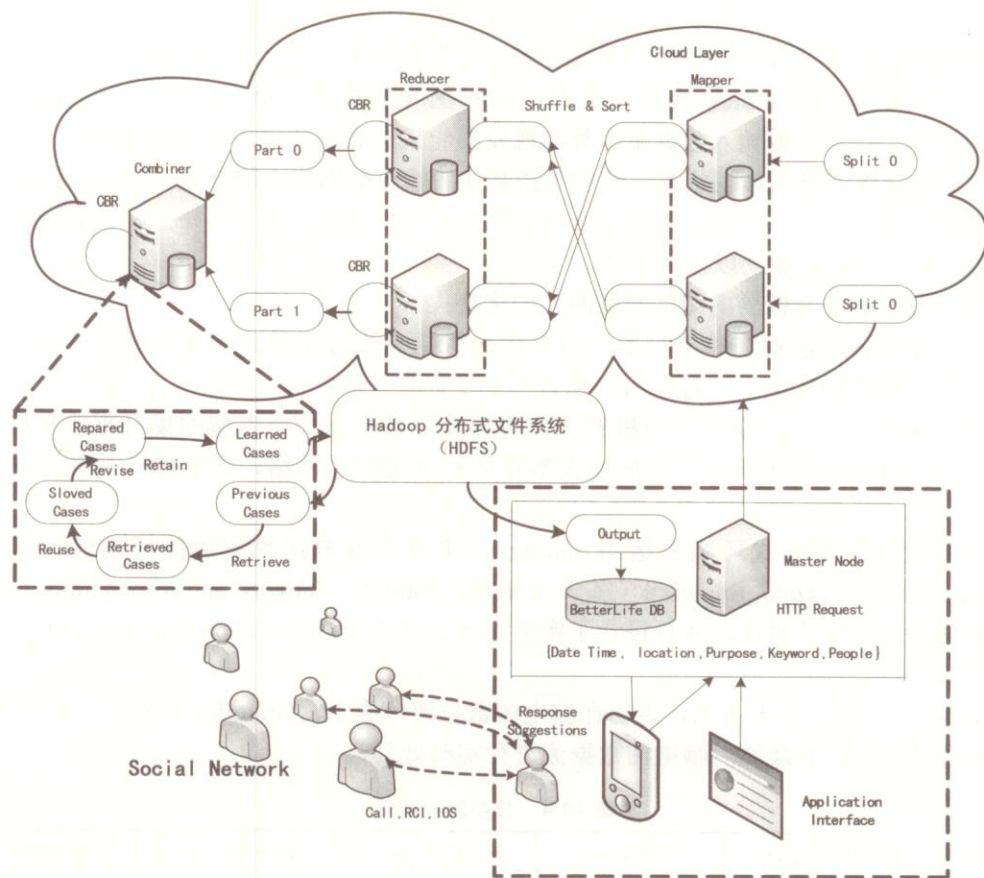


图 10-27 BetterLife 2.0 的体系结构

在 HDFS 上完成 CBR 推理过程的工作流如图 10-27 所示。Map 函数可以分为两个阶段,即检索阶段和过滤阶段。在检索阶段,每个 Mapper 操作从本地读取数据,且由于查询针对所有数据,所以此阶段不需要搜索数据。HDFS 自带有数据复制功能,当数据节点发生故障时会有另外一个具有备份数据的节点重新执行 Map 函数。此阶段结束后将会获得一系列数据,然后进入过滤阶段。在过滤阶段,这些数据被解释执行,一些不匹配的产品号或过期的数据都被过滤掉,剩下的数据会被计算相似度或是附加到社会紧密度因素 (Social Closeness Factor) 中。之后这些数据被写入中间体并输出到 Reducer 中。在 Reducer 操作中, CBR 对每条案例进行相似度测量。这里定义了四种相似度函数:位置相似度、时间戳相似度、社会紧密度相似度、价格相似度。

基于 BetterLife 2.0 框架开发了基于位置、价格对比的应用程序。通过此应用程序评估带有 MapReduce 和社会紧密信息的云中 CBR 技术的可用性、性能和准确性。该应用程序使用户利用移动电话找到最佳购买位置 (最优价格且来自可靠数据源的信息),用户只要

用手机拍下产品条形码,手机自动将用户 ID 号、条形码信息及检测到的 GPS 位置发送给服务器,服务器就会在可靠数据中进行分析,为用户找到最佳的购买位置,并将结果信息反馈给手机用户。该应用解决了顾客在商店位置和产品价格差异之间选择的问题。

10.6 云安全研究

云安全指将用户与一个安全平台通过互联网紧密相连,组成一个庞大的对病毒、垃圾邮件、木马、恶意软件等内容的监测网络,每个用户都为云安全贡献一份力量,同时分享其他用户的安全成果。

随着越来越多的企业投入云计算的研究中,未来通过互联网,任何内容都能够随时随地的在任何设备上找到,云计算的安全已经成为十分重要的挑战性问题。如何快速高效地收集用户的安全威胁、如何提高对新型病毒的攻击行为的分析准确度、如何保证登录用户的权限及用户的隐私信息不被泄露等,这些都成为未来云安全的重点与挑战。目前已经有些典型的研究案例。

10.6.1 Anti-Spam Grid: 反垃圾邮件网格

本书作者刘鹏早在 2003 年提出的反垃圾邮件网格 (Anti-Spam Grid) 技术^{[68][69]}是云安全技术的前身。

随着 Internet 的迅速普及,电子邮件成为人们进行信息交流的一种重要手段,然而,垃圾邮件 (Spam) 的泛滥给互联网带来了严重问题。所谓垃圾邮件通常是指不请自来的邮件。据统计,在 2002 年年初,垃圾邮件占整个邮件发送量的 16%,2003 年年初变成 42%,2004 年年初变成 60%。与大多数计算机中安装有防病毒软件不同,迄今为止尚无有效的手段可以自动清除垃圾邮件,只能依靠手工清除,因此会消耗大量人力。Radicati 集团认为,2007 年全球因为垃圾邮件造成的损失多达 1130 亿美元。

垃圾邮件泛滥而无法用技术手段很好地自动过滤,是因为所依赖的人工智能方法不是成熟技术。垃圾邮件的最大特征是它会将相同的内容发送给数以百万计的接收者,因而可以基于此提出一种全新的过滤垃圾邮件的方法。它基于网格技术,通过分布式统计和分布式贝叶斯学习,利用分布互联网里的千百万台主机协同工作来构建一道拦截垃圾邮件的“天网”。该方法可以大大提高垃圾邮件的识别率,同时避免将合法邮件误判为垃圾邮件,这就有可能通过技术手段有效地解决垃圾邮件问题。

具体来讲,是要建立一个分布式统计和学习平台,以大规模用户的协同计算来过滤垃圾邮件。首先,用户安装客户端,为收到的每一封邮件计算出一个唯一的“指纹”,通过比对“指纹”可以统计相似邮件的副本数,当副本数达到一定数量,就可以判定邮件是垃圾邮件;其次,由于互联网上多台计算机比一台计算机掌握的信息更多,因而可以采用分布式贝叶斯学习算法,在成百上千的客户端机器上实现协同学习过程,收集、分析并共享最新的信息。反垃圾邮件网格体现了真正的网格思想,每个加入系统的用户既是服务的对象,也是完成分布式统计功能的一个信息节点,随着系统规模的不断扩大,系统过滤垃圾邮件的准确性也会随之提高。用大规模统计方法过滤垃圾邮件的做法比用人工智能的方法更成熟,不容易出现误判的情况,实用性很强。

选择网格技术作为反垃圾邮件系统的基础主要基于以下考虑。

(1) 垃圾邮件是基于全网发送的, 故需要建立一个全局性的基础结构来收集垃圾邮件的信息。

(2) 中央控制系统可能会存在瓶颈问题, 而分布式服务网络具有更大的优势。

(3) 电子邮件系统是一个最具动态性的系统, 所有的服务器、客户端及电子邮件都在不断保持更新, 需要一个能很好地适应不断更新的灵活的平台。

图 10-28 显示了反垃圾邮件网络的架构。系统主要包括客户端、服务器、调度器 (Dispatcher), 其中客户端的主要任务是进行邮件的数字签名计算、贝叶斯学习及进行签名和贝叶斯学习结果上报; 服务器端的主要任务是对邮件数字签名及贝叶斯学习成果进行相互传播, 选择一台服务器进行统计工作并把统计结果反馈给客户端; 调度器的主要任务是根据客户端请求动态分配服务器。

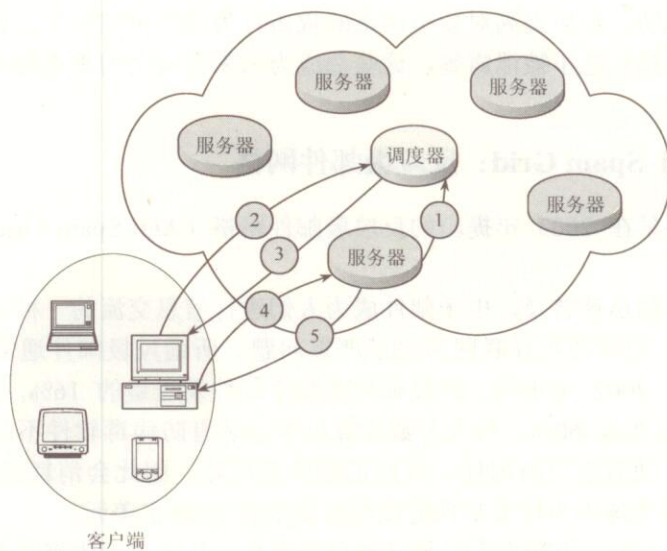


图 10-28 反垃圾邮件网络的架构

反垃圾邮件网络的主要工作步骤如下。

- (1) 反垃圾邮件服务器向某个调度器发布自己的服务。
- (2) 客户端向调度器提出过滤申请。
- (3) 调度器根据负载均衡或其他策略选择一台服务器为其服务。
- (4) 客户端向服务器报告邮件的签名和贝叶斯学习成果。
- (5) 服务器反馈此签名邮件的统计信息及其他客户端最近更新的贝叶斯学习成果。
- (6) 客户端据此过滤垃圾邮件。
- (7) 服务器之间共享数据。

反垃圾邮件网络具有较好的扩展性, 主要表现在无论是服务器还是调度器均可以动态地加入; 当有新服务器加入时, 调度器可以及时地让它分流一部分客户端请求。

2003 年 12 月, 反垃圾邮件网络被 IEEE Cluster 2003 国际会议选出赴中国香港做现场演示, 受到与会专家的热烈好评。2004 年 4 月, 国内最大规模的网格会议 Grid Computing World China 2004 上也进行了专题报告和现场演示。反垃圾邮件网络的思想和技术受到一些大型邮件服务提供商的重视和借鉴, 为近年来垃圾邮件的比重迅速下降作出了贡献。反

垃圾邮件网格的思想可以用在过滤垃圾邮件上,当然也可以用于发现病毒和木马——这就为目前的云安全技术作了很好的铺垫。

10.6.2 CloudAV: 终端恶意软件检测

当前,杀毒软件普及率很高,已经成为装机必备软件,然而病毒感染率却在逐年上升,这与传统杀毒模式有很大关系。传统的反病毒模式存在诸多局限性,主要体现在以下两个方面。

(1) 被动防御、滞后杀毒。现有反病毒软件病毒库即使升级到最新版本也已经不足以保护用户免受病毒感染。现有的反病毒软件通过从病毒体中提取病毒特征值构成病毒特征库,并对计算机中的文件或程序等目标逐一进行特征值比对,以判断计算机是否被病毒感染。只有发现并捕获到新病毒后,杀毒软件厂商才有可能从病毒体中提取其特征值。这种特征值扫描技术决定了杀毒软件的滞后性,使用户不能对网络新病毒及时防御,传统的杀毒软件扫描方法已经跟不上呈爆炸式增长的恶意软件发展速度,就会导致病毒乘虚而入。有研究数据显示,传统反病毒软件对最近出现的恶意软件的检测能力明显降低,一般情况下,防御引擎对出现一年后的恶意软件的防御效果最好^[20]。

(2) 反病毒软件本身存在漏洞。反病毒软件的复杂性也导致了其漏洞的增加,病毒制造者完全可以利用这些漏洞或者薄弱环节进行攻击。

针对传统杀毒软件的局限性,Jon 等^[20]提出了终端恶意软件检测模型,相对于传统的反病毒软件该模型有如下两个关键的变化。

(1) 云端检测。将反病毒功能包装成网络服务,由终端或单一的服务器端移植到网络云上,实现云端检测。在这个过程中,每个终端客户都运行一个轻量级的程序来检测新的文件,并将它们发送到网络服务进行分析,然后根据网络服务提供的报告,决定是否接受文件或进行隔离处理。这样做的好处是:用户再也不需要不断地更新本地签名数据库,简化的终端软件减少了其本身存在漏洞的可能性,并且轻量级的终端程序能够安装在各种移动设备或者是计算能力有限的设备上,随着 Wi-Fi、3G 网络的普及,这些设备也容易受到恶意软件的攻击。

(2) N-VERSION 保护技术。恶意软件的检测应该有多种多样的检测引擎并行执行,所以引入了 N-VERSION 的思想。由各种各样的检测引擎来提供分析结果,可以有效扩大检测范围,防御极度复杂并不断进化的恶意软件。现在已经有一些在线的恶意软件分析服务使用了这种思想,但是需要手动上传病毒样本,缺少自动实时的防御能力。

该模型包括三个主要的组件,如图 10-29 所示。

1) 客户端软件

客户端软件的作用主要是识别新的文件并且将他们发送到网络上进行分析,主要有两种实现形式。一种是运行在桌面、便携式计算机、移动设备等终端的轻量级的客户代理,用来识别新的文件并将它们发送到网络上;每个文件都将生成唯一的标识码,通过与之之前分析过的文件比较,决定是否将文件发送到网络云中进行分析。另外一种实现形式是利用网络传感器或网络监听器,它们能够使用深度分组检查技术,在文件没有到达终端用户时,就进行检测和分析。

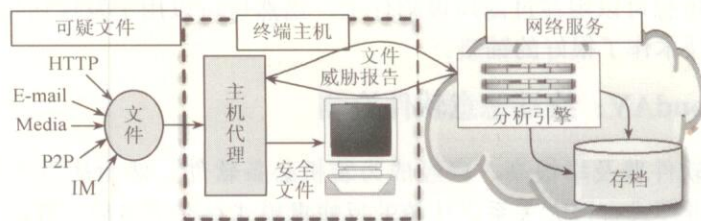


图 10-29 终端恶意软件检测模型

2) 网络服务组件

网络服务组件的作用是接收客户传来的文件，并识别恶意的内容。它与传统方式的不同之处在于，它集合了多个检测引擎，对文件进行并行分析，并通过综合给出分析结果。分析报告中包含如下几个内容。

(1) 操作指令。一组指导客户代理操作的说明，如文件应该存储、打开、执行或是隔离。

(2) 由不同的检测引擎分配的恶意软件分类标识。

(3) 行为分析。包括文件和密码的修改、网络行为或者其他状态的信息改变。分析报告将存储在用户代理的本地缓存中或服务器的远程共享缓存中，再出现相同的文件时，用户不需要再发送到网上进行分析，直接在本地就能进行检测分析。

3) 存档服务组件

存储服务用来存储分析结果信息并提供操作管理接口。Jon 等^[20]通过试验，证明 CloudAV 的 N-VERSION 保护技术对于新出现的恶意攻击的防御检测范围要比传统的单个反病毒引擎高 35%，而且能检测出 98% 的恶意软件，传统的单个反病毒引擎仅能检测出 83%。各项数据表明该模型具有如下优点。

(1) 更高效的检测恶意软件。多个反病毒引擎的结合，可以综合各个引擎的检测能力，使检测更加全面。

(2) 增强了辨别能力。建立一个大型丰富的数据库，存放用户存取文件的记录并加以辨别和干扰分析。

(3) 回溯性检测能力。新的威胁出现时，云反病毒服务能够根据历史记录发现以前相似或相同的病毒活动，甚至可以自动地隔离受感染的主机。

(4) 更强大的部署和管理能力。将检测功能移植到网络上，构建一个更大的平台，并提供集中管理签名和控制文件存取的策略。

10.6.3 AMSDS：恶意软件签名自动检测

恶意软件经常会危害计算机，并利用软件的漏洞盗取用户私人数据，所以通过反病毒扫描引擎产生数字签名来检测攻击是一个重要的防范途径。然而，现在的恶意软件能够利用代码迷惑技术，较容易地绕过反病毒扫描，使得扫描引擎根本无法检测到恶意文件的内容；并且如今的恶意软件签名技术通常需要繁重的手工处理，往往生成签名相对滞后。

为了有效地处理大规模的恶意软件变种，反病毒的功能被移植到了云端处理，Wei Yan^[21]等提出的恶意软件签名自动检测系统 (AMSDS) 是支持云端反病毒的新型的、轻量

级的桌面引擎，能够自动地产生一个轻量级的签名数据库，它比传统的签名数据库小得多。在反病毒云模型中，用户不需要安装大量的病毒签名文件，只需要一个轻量级的“云签名”集，只有遇到 AMSDS 无法检测的可疑文件时，客户端才向云服务器发送请求。

AMSDS 的工作分为以下两个部分。

(1) 去噪 (De-noise): 解析轻便型可执行格式 (Portable Executable Format) 样本，并列出的内部结构，为了加快签名产生的速度，AMSDS 只保留黑客可能插入恶意代码的部分。去噪过程能将原始文件缩小将近一半。

(2) 破坏恶意软件的 PE 格式，使恶意的样本无法执行。

AMSDS 在静态和动态的环境中都能产生恶意软件签名。静态环境中，AMSDS 的智能转换器解析恶意软件静态信息，然后用机器学习技术寻找多离散二进制序列，为内匹配产生静态的组合签名。在动态环境中，AMSDS 能够动态地产生行为签名，如今，沙盒和仿真器经常用来捕获恶意软件的动态行径。

由于恶意软件使用代码迷惑技术，不论是静态还是动态产生的“噪声”都很难分析，所以如今的恶意软件签名技术通常是通过繁重的手工劳动来完成的。据此，引入本体技术，实现基于本体的动态签名生成系统，不同的本体可以应用于不同的检测目的中，这种方法灵活，并在动态签名生成过程中有很好的效果。

10.6.4 CloudSEC: 协作安全服务体系结构

大量网络入侵都是由于在分布式平台上传播恶意软件造成的。“协作安全”能够抵抗来自恶意软件的分布式入侵。Jia Xu 等^[22]提出的 CloudSEC 可用于在云中组成协作安全服务，如相关入侵分析，反垃圾邮件、防 DDOS、自动化的恶意软件检测和控制等。作为一个动态的端对端覆盖的层次结构，CloudSEC 是由三种类型自上而下的体系结构组件建模而成。依靠这种结构，数据分配和任务调度覆盖可同时采取一种松散的耦合方式。这样不仅可以高效地从异构网络安全设施中获取数据资源，而且可以利用分布式集合计算资源来处理数据密集型任务集合。与此同时，CloudSEC 还赋予网络安全基本构架的动态适应能力，以及对一个组织范围的协作能力。初步评估结果表明，CloudSEC 不仅为处理分布式入侵的相关服务提供了高扩展性和可靠性，还在数据共享和任务调度方面取得了显著的成效。

CloudSEC 的目标是为各自的用户提供协同安全服务，且提供一个分布式的、可扩展的而且可靠的平台。CloudSEC 的体系结构如图 10-30 所示。

CloudSEC 系统由三个部分组成：管理组 (Administration Group)、协作组 (Collaboration Groups)、外围实体 (Peripheral Entities)。管理组和协作组的所有节点间的通信都进行了加密操作。

1) 管理组

管理组是 CloudSEC 体系结构的内核组件，每个任务协调员在一个动态管理域中都包含一组自主安全代理。任务协调员负责作出安全决策，管理协作任务以及分析跨多个管理域的分布式结果。它们的可扩展性和可用性是 CloudSEC 应变能力的基础。覆盖网络具有高弹性，且能够在大规模网络故障的情况下高效地传输信息。

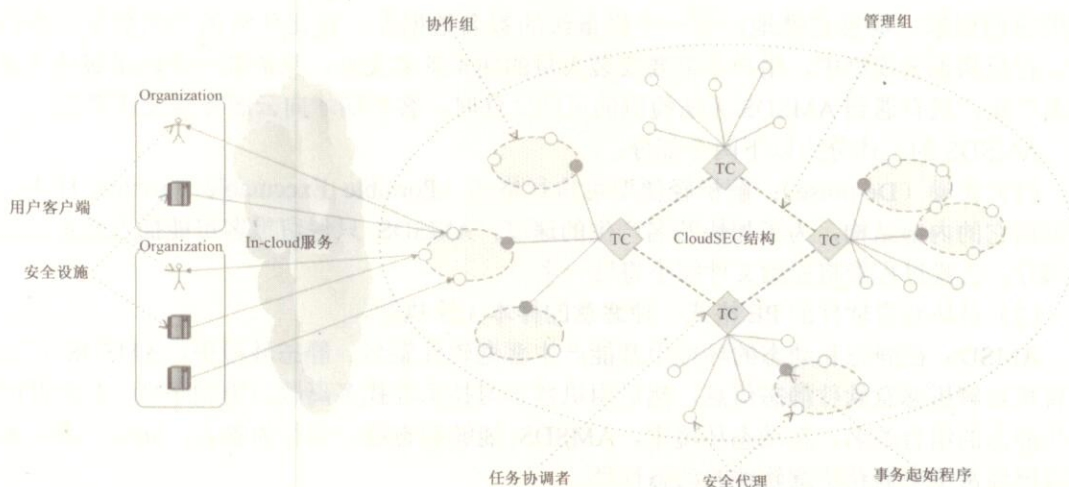


图 10-30 CloudSEC 的体系结构

2) 协作组

协作组是 CloudSEC “所求即所需”特征的重要组成部分，它们由一组动态集群的安全代理组成。每个代理通过提供一组普遍接入点来指向异构安全设施和用户客户端覆盖的 IP 空间，以确保代理能自发的提供和销毁数据资源。同时，还需在协作组上建立数据分布和任务调度覆盖层。凭借高效的分享数据的存储能力以及跨组织拓扑的计算能力来确保大规模的协作安全服务可以按需运行。第一个加入协作组的安全代理叫做任务发起者（Task Initiator），它是协作组和它的任务协调者之间的直接通信者。它从任务协调者处获得了合作任务和指令，并传递任务状况和最终结果。通过这种方式来对管理组和协作组保留分层的属性，这样每个安全代理都可通过任务协作者的转发机制来高效路由 CloudSEC 体系结构中信息的潜力。

3) 外围实体

外围实体用来汇总所有服务提供者和使用者的抽象功能。这些节点未实现 CloudSEC 协议，且对体系结构层没有访问权限。但这些节点可以安装大量的异构安全设施和客户软件，并且通过推拉机制来实现提供和销毁数据资源。CloudSEC 以插件的形式来支持扩展。

管理组和协作组采用数据的分布式覆盖，促进了 XML 数据在 P2P 网络构架中的存储和检索。因此 CloudSEC 可以保持局部和全局所需网络的安全活动视图。此外，用分布式方法分解和重组任意 XML 碎片可以使 CloudSEC 把一个资源紧密型数据集任务分解为一组小的子任务，并且把它们分散到一个协作组中去完成。

构成一个 in-cloud 安全服务（如相关入侵分析）常需要大量的计算资源，这些资源极可能超出单个安全代理的能力。因此，在协作组部署一个任务调度覆盖层动态地联合大量的安全代理，以便提供理想的具有按需分配计算能力的单系统。

在为数据密集型任务提供高并行计算方面，CloudSEC 协作服务组和网格计算有些相似。但协作组有一些专业属性，包括高自治与高扩展性、按需为远程用户提供安全服务以及动态配置服务。另外，传统网格调度拓扑主要专注于利用有效系统的细节和更新信息确定一个最佳的计算安排，这不可避免的带来了过多的管理开支。因此，CloudSEC 用一个轻量级、分散化、自适应的方法来组织计算资源。

习题

1. 常见的云系统的分类有哪些?
2. 云的支撑平台有哪些, 它们之间有哪些共同点和不同点?
3. 云计算的关键技术主要有哪些?
4. 云计算还可以在哪些领域有所应用?
5. 云安全研究还有哪些, 请查阅相关文献, 至少列举两点。

参考文献

- [1] S.Pearson.Taking Account of Privacy when Designing Cloud Computing Services. http://www.chinacloud.cn/upload/2009-04/temp_09043009508106.pdf
- [2] J.Boulon,A.Konwinski,R.Qi,A.Rabkin,E.Yang,and M.Yang. Chukwa,a large-scale monitoring system.In Proc.CCA,2008
- [3] L.Litty,D.Lie.Computer Meteorology: Monitoring Compute Clouds www.eecg.toronto.edu/~lie/papers/litty-hotos2009-web.pdf
- [4] S.Srikantaiah,A.Kansal,F.Zhao. Energy Aware Consolidation for Cloud Computing http://www.chinacloud.cn/upload/2009-04/temp_09043010395080.pdf
- [5] Daniel J. Abadi. Data Management in the Cloud: Limitations and Opportunities. Data Engineering.2009,3,32(1)
- [6] S.Wu,K.L.Wu. An Indexing Framework for Efficient Retrieval on the Cloud. Data Engineering.2009,3,32(1)
- [7] S.Fisher.Service Computing:The AppExchange Platform.SCC, 2006
- [8] S.Wu, K.L.Wu.An Indexing Framework for Efficient Retrieval on the Cloud. <http://sites.computer.org/debull/A09mar/wusai.pdf>
- [9] L.Z.Wang,J.Tao,M.Kunze,D.Rattu.The Cumulus Project: Build a Scientific Cloud for a Data Center. www.cca08.org/papers/Paper29-Lizhe-Wang.pdf
- [10] Y.Gu,R. Grossman. Sector and Sphere: The Design and Implementation of a High Performance Data Cloud. Philosophical Transactions A Special Issue associated with the 2008 UK e-Science All Hands Meeting
- [11] F.Marozzo,D.Talia,P.Trunfio. Adapting MapReduce for Dynamic Environments Using a Peer-to-Peer Model. www.cca08.org/papers/Poster7-Domenico-Talia.pdf
- [12] B.Peng,B.Cui and Xiaoming Li. Implementation Issues of A Cloud Computing Platform. Data Engineering.2009,3,32(1)
- [13] P.Watson,P.Lord,F.Gibson,P.Periorellis,and G.Pitsilis. Cloud Computing for e-Science with CARMEN. In 2nd Iberian Grid Infrastructure Conference Proceedings, pages 3-14, 2008
- [14] B.Rochwerger,D.Breitgand,E.Levy,A.Galis,and K.Nagin. The reservoir model and architecture for open federated cloud computing. IBM Systems Journal, October 2008

- [15] Brian F.Cooper,E.Baldeschwieler,R.Fonseca.Building a Cloud for Yahoo!.Data Engineering. 2009,3,32(1)
- [16] <http://www.mongodb.org/display/DOCS/Home>
- [17] <http://www.abiquo.com/>
- [18] VMware Infrastructure: <http://www.vmware.com/cn/products/vi/>
- [19] Xen. http://cn.opensuse.org/Xen_Virtual_Machine_Overview
- [20] Oberheide J,Cooke E,Jahanian F (2008) Clouday: N-version antivirus in the network cloud. In:Proceedings of the 17th USENIX security symposium (Security'08),San Jose, 28 July-1 August 2008
- [21] W.Yan, E.Wu. Toward Automatic Discovery of Malware Signature for Anti-virus Cloud Computing.www.springerlink.com/index/n3xw6286w1052w52.pdf
- [22] Jia Xu, Jia Yan, Liang He, Purui Su, Dengguo Feng. State Key Laboratory of Information Security, Institute of Software, Chinese Academy of Sciences. CloudSEC: A Cloud Architecture for Composing Collaborative Security Services. CloudCom.2010.
- [23] A Newman, YF Li, J Hunter. Scalable Semantics-the Silver Lining of Cloud.IEEE Fourth International Conference on eScience, 2008
- [24] Ding, L., et al. Tracking RDF Graph Provenance using RDF Molecules.2005,UMBC
- [25] K.Keahey, R. Figueiredo, J. Fortes, T. Freeman, and M.Tsugawa. Science clouds:Early experiences in cloud computing for scientific applications. In Cloud Computing and Applications 2008 (CCA08), 2008
- [26] The Nimbus Cloud: <http://workspace.globus.org/clouds/nimbus.html>
- [27] The Nimbus Toolkit: <http://workspace.globus.org/>
- [28] Amazon Elastic Compute Cloud (Amazon EC2): <http://www.amazon.com/ec2>
- [29] Massimo Gaggero,Simone Leo,Simone Manca. Parallelizing bioinformatics applications with MapReduce. www.cca08.org/papers/Poster10-Simone-Leo.pdf
- [30] BLAST, <http://www.ncbi.nlm.nih.gov/BLAST>
- [31] A.Subramanian,P.Tamayo,V.K.Mootha,S.Mukherjee,B.L.Ebert,M.A.Gillette, A.Paulovich, S.L.Pomeroy,T.R.Golub,E.S.Lander,and J.P.Mesirov,"Gene set enrichment analysis:A knowledge-based approach for interpreting genome-wide expression profiles," PNAS, vol.102,no.43,pp.15 545-15 550,2005
- [32] J.Wang and Q.Mu,"Soap-HT-BLAST: high throughput BLAST based on Web services," Bioinformatics,vol.19,no.14,pp.1863-1864,2003
- [33] F.Schmuck and R.Haskin,"The design,implementation, and evaluation of mpiblast," in Proc.ClusterWorld,2003
- [34] A.Krishnan,"Gridblast:a globus-based high-throughput implementation of blast in a grid computing framework: Researcharticles, "Concurr.Comput. :Pract. Exper.,vol.17,no.13, pp.1607-1623,2005
- [35] S.E.Dowd,J.Zaragoza,J.R. Rodriguez,M.J.Oliver, and P.R.Payton,"Windows .net network distributed basic local alignment search toolkit,"BMC Bioinformatics,vol.6,no.93,2005
- [36] P.C.Carvalho, R.V.Gloria, A.B.de Miranda, and W.M.Degrave,"Squid-a simple bioinformatics

- grid,"BMC Bioinformatics,vol.6,no.197,2005
- [37] C.Oehmen and J.Nieplocha,"Scalablast: A scalable implementation of blast for high-performance data-intensive bioinformatics analysis,"IEEE Transactions on Parallel and Distributed Systems,vol.17,no.8,pp.740-749,2006
 - [38] H.Trease,D.Fraser,R.Farber and S.Elbert. Using Transaction Based Parallel Computing to Solve Image Processing and Computational Physics Problem. www.cca08.org/papers/Poster31-Harold-Trease.pdf
 - [39] Apache Hadoop: <http://hadoop.apache.org/>
 - [40] PNNL's MeDICI:<http://dicomputing.pnl.gov/capabilities/softwarearchitecture>
 - [41] A.Aboulmaga,K.Salem,A.A.Soror. Deploying Database Appliances in the Cloud. Data Engineering.2009,3,32(1)
 - [42] A.Ganapathi, H.Kuno, U.Dayal, J.Wiener, A.Fox, M.Jordan, and D.Patterson.Predicting multiple metrics for queries: Better decisions enabled by machine learning. In Proc. IEEE Int.Conf. on Data Engineering (ICDE), 2009
 - [43] Q.C.Chen,L.Q.Wang. MRGIS:A MapReduce-Enabled High PerformanceWorkflow System for GIS. www.chinacloud.cn/show.aspx?id=1949&cid=28
 - [44] K.Beyer,V.Ercegovic,R.Krishnamurthy. Towards a Scalable Enterprise Content Analytics Platform.<http://sites.computer.org/debull/A09mar/sandeep.pdf>
 - [45] L.Youseff,M.Butrico, D.D.Silva. Toward a Unified Ontology of Cloud Computing. www.collab-ogce.org/gce08/images/7/76/LamiaYouseff.pdf
 - [46] A.Lenk,M.Klems,J.Nimis,S.Tai. What's Inside the Cloud? An Architectural Map of the Cloud Landscape.
<http://www2.computer.org/portal/web/csdl/doi/10.1109/CLOUD.2009.5071519>
 - [47] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale Virtualization in the Emulab Network Testbed. In Proceedings of the 2008 USENIX Annual Technical Conference, 2008
 - [48] Hewlett-Packard. HP Integrated Lights-Out 2 User Guide. Technical report, HP, 2009
 - [49] Amazon Web Services. Amazon webservicess homepage.
<http://aws.amazon.com>, Seen: 2008-12-05, 2008
 - [50] D. Nurmi, R. Wolski, C. Grzegorzczak, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. Eucalyptus open-source cloud-computing system. In CCA08: Cloud Computing and Its Applications, 2008
 - [51] K. Lai, L. Rasmusson, E. Adar, S. Sorkin, L. Zhang, and B. A. Huberman. Tycoon: an Implementation of a Distributed Market-Based Resource Allocation System. Multiagent and Grid Systems, 1(3):169-182, Aug. 2005
 - [52] K. Keahey and T. Freeman. Contextualization: Providing one-click virtual clusters. In eScience 2008, 2008
 - [53] B. Sotomayor, R. Montero, I. M. Llorente, and I. Foster. Capacity Leasing in Cloud Systems using the OpenNebula Engine. In CCA08: Cloud Computing and its Applications, 2008

- [54] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In Symposium on Operating System Design and Implementation, 2004
- [55] S.Ghemawat and H.Gobioff and S.T.Leung. The Google File System. In ACM Symposium on Operating Systems Principles, 2003
- [56] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in college networks, 2008
- [57] G.DeCandia and D.Hastorun and M.Jampani and G.Kakulapati and A.Lakshman and A. Pilchin and S.Sivasubramanian and P.Vosshall and W.Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In ACM Symposium on Operating Systems Principles, 2007
- [58] F.Chang and J.Dean and S.Ghemawat and W.C. Hsieh and D. A. Wallach and M.Burrows and T.Chandra and A. Fikes and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In Symposium on Operating System Design and Implementation, 2006
- [59] S. Microsystems. Project caroline.
<http://research.sun.com/projects/caroline>, Seen: 2009-01-23, 2009
- [60] Django. Django web framework. <http://www.djangoproject.com>, Seen: 2009-01-23, 2009
- [61] Google Inc. Google apps engine. <http://www.google.com/apps>, Seen: 2008-12-05, 2008
- [62] Joyent. Reasonably smart. <http://www.joyent.com>, Seen: 2009-01-26, 2009
- [63] Microsoft. Azure services platform.
<http://www.microsoft.com/azure>, Seen: 2009-01-23, 2009
- [64] OpenId Foundation. Openid homepage. <http://www.openid.net>, Seen: 2008-12-05, 2008.
- [65] Google Inc. Google maps api. <http://code.google.com/apis/maps>, Seen: 2009-01-23, 2009.
- [66] Huan Liu, Dan Orban. GridBatch: Cloud Computing for Large-Scale Data-Intensive Batch Applications. In Eighth IEEE International Symposium on Cluster Computing and the Grid. 2008.
- [67] Christopher Moretti, Jared Bulosan, Douglas Thain, and Patrick J. Flynn. All-Pairs: An Abstraction for Data-Intensive Cloud Computing.
- [68] Peng Liu, Yao Shi, Francis C. M. Lau, Cho-Li Wang, San-Li Li, Grid Demo Proposal: AntiSpamGrid, IEEE International Conference on Cluster Computing, Hong Kong, Dec 1-4, 2003. <http://www.chinagrid.net/dvnews/show.aspx?id=290&cid=5>
- [69] 刘鹏, 赵伟. 反垃圾邮件网格清扫网络, 软件世界, 2006 年 13 期
http://media.ccidnet.com/art/3017/20060718/640971_1.html
- [70] T Okuda, E Kawai, S Yamaguchi. A Mechanism of Flexible Memory Exchange in Cloud Computing Environments. In CloudCom2010
- [71] M.A. Islam and S.V. Vrbsky, Tree-Based Consistency Approach for Cloud Databases. In Proceedings of CloudCom'2010. pp. 401-404
- [72] T Miyamoto, M Hayashi, K Nishimura. Sustainable Network Resource Management System for Virtual Private Clouds
- [73] Chris Preist, Paul Shabajee. Energy Use in the Media Cloud Behaviour Change, or Technofix?
- [74] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, D.Stott Parker. Map-Reduce-Merge:

- Simplified Relational Data Processing on Large Clusters.2007
- [75] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, Christos Kozyrakis
Computer System Laboratory Stanford University. Evaluating MapReduce for Multi-core
and Multiprocessor Systems.2007
- [76] <http://www.mcplive.cn/index.php/article/index/id/9326/page/3>
- [77] KaiShen, TaoYang, LingkunChu, JoAnneL.Holliday, DouglasA.Kuschner, Huican Zhu.
Department of Computer Science, University of California at Santa Barbara, CA93106.
Neptune: Scalable Replication Management and Programming Support for Cluster-based
Network Services.2001
- [78] Peter Sempolinski and Douglas Thain, University of Notre Dame.A Comparison and
Critique of Eucalyptus,OpenNebula and Nimbus. CloudCom.2010
- [79] Kathleen Ericson, Shrideep Pallickara, and Charles W.Anderson. Department of Computer
Science Colorado State University Fort Collins, US. Analyzing Electroencephalograms
Using Cloud Computing Techniques.CloudCom.2010
- [80] Dexter H.Hu, Yinfeng Wang, Cho-LiWang, Department of Computer Science, BetterLife
2.0: Large-Scale Social Intelligence Reasoning on Cloud. CloudCom.2010

第 11 章 总结与展望

本章横向比较 Google、Amazon、微软和 VMware 的商业云计算解决方案，Hadoop、Eucalyptus、Nimbus、Sector and Sphere 等开源云计算系统，以及大云、阿里云等国内主要的云计算平台，方便读者更好地掌握本书的主体内容。另外，云计算在互联网和信息社会发展史中处于什么位置？它将朝什么方向发展？这些读者关心的问题也将在本章加以阐述。

11.1 主流商业云计算解决方案比较

云计算时代已经到来，从 Google App Engine^[1]到 Amazon 的 AWS^[2]，从微软的 Azure^[3]到 VMware 的 vCloud^[4]，为了在云计算时代继续保持自己的领先优势，IT 业的巨头们纷纷推出自己的云计算解决方案。这些方案的着眼点和应用场景不尽相同，技术实现上各有千秋。解决方案的多种多样反映了云计算蓬勃发展的势头，也使得用户需要面临解决方案选择的问题。不论是个人用户还是企业用户，了解各种云计算解决方案的异同点，都将有助于选择最合适自己的方案。

本节将从应用场景、使用流程、体系结构、实现技术和核心服务五个方面比较 Google、Amazon、微软和 VMware 这四家公司的云计算解决方案。四家公司解决方案的具体细节参见前面的相关章节。

11.1.1 应用场景

Google、Amazon、微软和 VMware 这四家公司在不同时间陆续推出各自的云计算方案，在应用领域和赢利模式上，Amazon 和 Google 处于领跑者地位，微软和 VMware 紧随其后。

Google 在 2007 年率先提出了“云计算”的概念，开发了电子邮件、在线文档等一系列 SaaS 类型的云计算产品，并根据产品开发中积累的技术，打造了 Google App Engine 平台来对外提供 PaaS 类型的服务。开发人员可以在 Google App Engine 平台上开发应用程序并对外服务。该平台采用了 Google GFS、Bigtable 等关键技术，具有很高的可靠性和稳定性。但由于该平台和 Google 自身产品的开发需要结合得过于紧密，所以在使用中限制较多，例如只支持 Python 和 Java 语言、基于 Django 架构的 Web 应用等。目前 Google App Engine 的使用者大都是个人用户，使用的内容也主要是开发一些比较实用的小规模程序，比如搭建 CDN、使用 iPhone 访问 GAE 等。不过考虑到 Google 在全球的服务器数量已经超过 200 万台，海量的服务器将会产生巨大的聚集效应，再加上 Google 强大的科研创新实力，相信 Google 很快就会推出专门面向企业的“杀手级”服务。

Amazon 在“云计算”概念出现之前就开始了提供弹性的计算、存储等服务，其云计算解决方案在技术上最为全面深入。整套方案被统一命名为 Amazon Web Service

(AWS)。Amazon 的 AWS 包括了云计算服务的所有类型 (IaaS, PaaS 和 SaaS)。个人和企业可以通过 EC2 和 S3 来构建 SmugMug、Animoto 等典型应用, 通过 SimpleDB 来处理日常简单的数据库业务, 利用弹性 MapReduce 进行大规模数据处理, 利用 CloudFront 分发网页内容, 利用 FPS 提供安全的网上支付服务等。可见, Amazon 提供的云计算服务是目前所有的商业解决方案中覆盖领域最全面、应用范围最广泛的, 并且 Amazon 还在根据市场需求不断推出更多的新型服务。

微软在 2008 年发布了其云计算战略及云计算服务平台 Windows Azure Platform, 其后连续发布了几个版本, 很多特性和服务都在不断地完善和改进中。微软的 Azure 平台中包含了 IaaS 和 PaaS 类型的云计算服务, 主要面向软件开发商。其中, Windows Azure 云操作系统是整个云计算方案的核心, 包含了多种计算和存储服务; 在此基础上 AppFabric 和 SQL Azure 分别提供了云的基础架构服务和数据库服务。此外, 微软还提供了 Azure Marketplace 用于在线购买基于云计算机的数据与应用。与其他方案不同的是, Azure 中考虑了本地环境在云计算方案中的作用, Azure 上的程序在离线状态下仍可在本地环境中运行。

VMware 在云计算解决方案中充分利用了自身在虚拟化技术上的领先优势, 在 2008 年与 EMC、思科等公司联合推出了 vCloud 计划, 在 2009 年又推出了首款云操作系统 vSphere。目前, VMware 通过自主研发和收购合作, 提供了包括云基础架构及管理、云应用平台和终端用户计算在内的所有类型的一系列云计算产品和解决方案, 其中以 IaaS 类型服务为主, 用以支持企业级组织机构从现有的数据中心向云计算环境进行转变。

表 11-1 展示了 Google、Amazon、微软和 VMware 云计算解决方案总体的异同点。

表 11-1 主流商业云计算解决方案比较

	Google	Amazon	微软	VMware
提供的服务类型	PaaS, SaaS	IaaS, PaaS, SaaS	IaaS, PaaS, SaaS	IaaS, PaaS, SaaS
服务间的关联度	所有服务被捆绑在一起, 耦合度高	可以任意选择服务组合, 耦合度低	可以任意选择服务组合, 耦合度低	可以任意选择服务组合, 耦合度低
虚拟化技术	未使用	Xen	Hyper-V	ESX Server
运行环境	Google 提供的环境, 位于云端	Amazon 平台, 位于云端	位于云端或本地	位于云端
支持的编程语言	Python, Java	多种	多种	多种
使用限制	最多	最少	较少	较少
实现功能	最少	最多	较多	较多
计费方式	有免费部分和收费项目	按实际使用量付费	按实际使用量付费	按实际使用量付费
可扩展性	自动扩充所需资源并进行负载均衡	需要手动或通过编程自动的增加所需的虚拟机数量	需要手动或通过编程自动地增加所需的虚拟机数量	需要手动或通过编程自动地增加所需的虚拟机数量
不同应用间的隔离	通过沙盒来实现	通过将不同的应用运行在不同的虚拟机上来实现	通过将不同的应用运行在不同的虚拟机上来实现	通过将不同的应用运行在不同的虚拟机上来实现

从表中可以看出, 四家公司的云计算解决方案各有特色, 除了各个公司提供的云计算软件服务外, 用户可以根据自身需要选择合适的平台来构建自己的云计算服务。例如, 如果用户对定制性要求不高、希望简化操作时, 可以选择 Google App Engine; 如果需要直

接定制底层硬件配置，可以选择 Amazon 的 AWS 或 VMware 的 vCloud；如果希望利用已有的 IT 环境或需要离线操作，则可以选择微软的 Azure。

11.1.2 使用流程

个人或企业用户在使用各种云计算解决方案时都要遵从一定的使用流程。Google、Amazon、微软和 VMware 的云计算方案从整体上来看基本的流程是一致的，但是具体的细节有所不同。

1. Google App Engine 的使用流程

- (1) 注册 Google 账户，填写注册信息，登录。
- (2) 创建 Google App Engine 应用，通过手机号码完成验证，填写应用的详细信息（注意应用的标示符无法更改）。
- (3) 下载 App Engine SDK。
- (4) 使用 Python 或 Java 语言在本地开发应用程序，并完成本地调试。
- (5) 将程序上传到 Google App Engine 后运行。

2. Amazon AWS 的使用流程

- (1) 注册亚马逊账户，填写注册信息，登录。
- (2) 根据需要选择需要的服务进行注册，填写相关信息，完成服务配置（对于 IaaS 类型服务需要选定所需的资源数，对于其他类型服务需要对设置参数）。
- (3) 上传应用程序或待处理数据，有时需要按要求上传附加程序。
- (4) 运行服务，直至获取结果。
- (5) 停止使用，根据实际使用量支付相关费用。

3. 微软 Azure 的使用流程

- (1) 在 Azure 页面上输入 Live ID，注册 Azure 帐号，填写注册信息，登录。
- (2) 在项目列表中选择“Windows Azure”，然后在新建服务向导中选择“托管服务”。
- (3) 在本地新建“cloud”类型项目，编写应用程序并完成调试。
- (4) 创建应用程序服务包，将服务包上传到 Windows Azure 上，设定 URL 地址，选择“部署”，选择“运行”。
- (5) 停止使用，根据实际使用量支付相关费用。

4. VMware vCloud 的使用流程

- (1) 加入 VMware 技术联盟计划，填写基本信息，获取账号和信息支持，登录。
- (2) 选择编程语言（支持 Java、C、C++）编写在不同操作系统（包括 Linux、Windows、Solaris）上运行的软件应用程序，并可根据 vCloud API 来利用基于 VMware 的云计算基础架构。
- (3) 在 VMware 认证服务提供商列表中选择合适的服务提供商，或选择使用企业自身的支持 vCloud 的云计算环境。
- (4) 在虚拟机、虚拟设备和 vApp 三种模式中选择一种，将应用程序部署到云平台中运行。

(5) 停止使用, 如果使用了服务提供商的服务, 根据实际使用量支付相关费用。

11.1.3 体系结构

Google、Amazon、微软和 VMware 的云计算解决方案所提供服务的差别与其云计算系统体系结构的差异密切相关。

Google App Engine 的结构是主/从式的, 共分为四个部分。前端用于负载均衡、静态文件转发和请求转发; 应用服务器用于运行程序; 应用管理节点用于复杂应用启停和计费; 服务群用于提供多种类型的服务。这四个部分被集成为一个整体, 对外提供服务。

Amazon AWS 的架构是完全分布式、去中心化的。不同用户请求在经过一系列请求路由后被分发到各自的目的地。服务之间的低耦合度可以保证各个服务之间的运行互不影响, 且整个系统不存在弱点。

微软 Azure 的架构由“云”和“端”两部分组成, “云”和“端”能够无缝地运行应用程序和提供服务是 Azure 的特点。Azure 的核心是云计算操作系统 Windows Azure, 用以提供多种计算和存储服务。在此之上, AppFabric 为本地应用和云中应用提供了分布式的基础架构服务, SQL Azure 提供类似 SQL Server 的数据库服务、报表服务、数据同步服务等。

VMware 平台的云计算服务中以云基础架构和管理为主, 其中主要包括云操作系统 vSphere 和底层架构服务 vCloud Service Director 两大部分。vSphere 中包含了底层的虚拟机 ESX Server 和 vCenter, ESX Server 负责对数据中心虚拟化, vCenter 负责整合和管理 ESX Server。在 vSphere 架构之上, vCloud Service Director 利用一系列虚拟技术提供连接企业虚拟环境与私有云的接口和自动化工具, 通过运行 vCloud Express 与外部服务商无缝地连接, 向外提供云 IaaS 服务。

四家公司云计算体系结构的主要相同之处有如下两点。

- (1) 整个云计算平台对外提供统一的 Web 接口。
- (2) 后台实现的细节对用户透明。

主要的区别也有两点。

(1) Amazon、微软和 VMware 的云计算服务都是由多种服务组成, 需要为不同的服务提供不同的入口。Google 的云计算服务实现相对简单, 没有实现多个服务的单独入口。

(2) 微软的云计算不仅支持云端应用程序, 还支持本地的应用程序, 这是微软云计算和其他三种方案的最大不同之一。这也反映了微软在云计算中的“云+端”策略。

11.1.4 实现技术

云计算至今还没有公认的统一定义, 技术实现上也是千差万别。各个公司都以自己原先的技术优势为基础, 来构建各自的云计算系统。下面将四家公司的核心技术进行总结。

1. Google App Engine 的实现技术

Google 云计算系统中所采用的技术来源于搜索等核心产品, 具有很强的创新性。Google App Engine 实现中所涉及的关键技术被 Google 以论文的形式陆续公开。总体来讲, 可以分为 GFS、MapReduce、Bigtable 和 Chubby 四个相互独立却又紧密联系的组成部分。这四部分都是由 Google 独立开发的全新系统。其中, GFS 是为 Google 应用程序量

身定做的分布式文件系统,数据分块存储在块服务器上并自动备份;MapReduce 是一种并行数据处理的编程规范,通过自定义的 Map 函数和 Reduce 函数可以实现大规模数据的快速并行处理;Bigtable 是一个采取了多种容错措施的分布式数据库,具有很高的可用性;Chubby 是一个分布式的锁服务,用于保证系统服务的一致性。

2. Amazon AWS 的实现技术

Amazon 在技术上进行了一系列的创新。最具代表性的是基础存储架构 Dynamo,它是一个完全分布式的存储架构,采用了改进的一致性哈希算法、向量时钟、Merkle 树等技术,在负载均衡、系统扩容等方面有着天然的优势。在此基础上,Amazon 设计了 EC2、S3、SimpDB 等计算、存储、数据库服务,并积极地引入已有的先进技术,如在 EC2 上使用 Hadoop 的 MapReduce 来构建弹性 MapReduce 服务等。

3. 微软 Azure 的实现技术

微软 Azure 以微软在个人计算机操作系统和应用软件上多年的技术积累为基础,通过在虚拟机上运行 Windows Server 2008、基于 SQL Server 实现 SQL Azure 等方式构建云计算系统。这些已有的技术具有很好的成熟性和广泛的使用性,经过了大量用户长期使用的检验,符合多数用户的使用习惯。通过整合和扩展已有技术,微软 Azure 可以保证用户在使用体验上的无缝过渡,也使得开发者可以使用习惯的编程语言和框架在相对熟悉的平台上进行软件开发。

4. VMware vCloud 的实现技术

VMware 充分利用在虚拟化技术上的优势,对云计算中涉及的计算、存储、网络等方面进行了虚拟化,提供以 IaaS 类型为主的云计算服务。在底层,VMware 开发了云操作系统 vSphere,实现了对数据中心服务器的虚拟化和对虚拟机的管理。在 vSphere 之上,VMware 又开发了 vCloud Service Director,利用一系列虚拟技术提供连接企业虚拟环境与私有云的接口和自动化工具。VMware 还提供了桌面虚拟化产品 VMware View,通过在一台普通的物理服务器上虚拟出很多台虚拟桌面来供远端的用户使用,以简化 IT 管理和节省开支。除此以外,VMware 还通过收购合作等方式,借助已有的基础来推出 PaaS 和 SaaS 类型的云计算服务。

11.1.5 核心业务

四种主流的商业云计算解决方案中均涉及计算服务、存储服务和数据库服务这三个核心业务。下面对不同方案在这三个核心业务上分别进行比较,找出其中的异同点。

计算服务是所有的云计算解决方案最核心的业务之一,同时也是用户最常用的服务。Google 提供基于 MapReduce 的数据处理,整个过程对用户而言是透明的。Amazon 的 EC2 给予用户配置硬件参数的权利,使得用户可以根据实际的需求动态地改变配置,从而提高效率和节省资源。微软的 Azure 允许用户在处理数据之前设置部分参数,但相对于 EC2 其灵活性要差很多。VMware 的 vCloud 中提供了 DRS 和 DPM 技术,可以通过迁移和关闭虚拟机来实现资源优化。表 11-2 是这四种计算服务的比较。

表 11-2 商业云计算方案的计算服务比较

	Google MapReduce	Amazon EC2	微软 Azure 计算服务	VMware vCloud 计算服务
服务类型	PaaS	IaaS	PaaS	IaaS
虚拟机的使用	未使用	用户可以根据需要设置运行虚拟机的硬件配置	系统自动分配	vCenter 自动进行资源优化
运行环境	Google 自身提供的环境, 用户无法自行调配	用户自行提供运行程序所需的 AMI	系统自动为用户生成的装有 Windows Server 2008 的虚拟机	用户在虚拟机、虚拟设备和 vApp 三种模式中选择一种
易用性	最好	稍差	较好	较好
灵活性	稍差	最好	较好	较好
适用的应用程序	适合可以并行处理的应用程序	任意程序	任意可在 Windows Server 2008 上运行的程序	任意程序

稳定、高效的存储系统既是系统正常运行的重要保证, 也可以单独作为一项服务提供给用户。四种方案之中, Amazon 的 S3 和微软的 Blob 存储比较的类似, Google 的 GFS 则完全不同, VMware 目前仅向虚拟机提供存储服务。表 11-3 是四种存储服务的简单对比。

表 11-3 商业云计算方案的存储服务比较

	Google GFS	Amazon S3	微软 Blob	VMware 存储
系统结构	文件分块存储	桶、对象两级模式	容器、Blob 两级模式	目录、文件两级模式
可扩展性	可通过增加数据块服务器数量扩展存储容量	可通过增加桶中对象数量扩展存储容量	可通过增加容器中 Blob 数量扩展存储容量	自动迁移虚拟机以获取更大存储容量, 及自动回收未使用存储容量
数据交互方式	用户和数据块服务器进行数据交互	用户可以从获得授权的对象中取得数据	用户可以从获得授权的 Blob 中取得数据	仅提供给虚拟机使用
存储限制	无特殊限制	桶的数量和对象大小有限制, 但对象的数量无限制	Blob 大小有限制, 但是容器和 Blob 数量未限制	数据存储可跨越多个物理存储子系统
容量扩展方式	自动扩容	手动或编程实现自动扩容	手动或编程实现自动扩容	自动迁移虚拟机以扩容
容错技术	针对主、从服务器有各自的容错技术	数据监听回传 Merkle 哈希树 数据冗余存储	仅重传出错的 Block, 数据冗余存储	为运行中虚拟机创建与同步的 Shadow 虚拟机 多个虚拟机的集中备份

四家公司都提供了“云”环境下的数据库存储服务。Google App Engine 的 Datastore 构建在 Bigtable 上, 但自身及其内部没有实现直接访问 Bigtable 的机制, 可以看做是 Bigtable 上的一个简单接口。Amazon 的 SimpleDB 采用的是“键/值”存储方式, 功能比较简单, 实现的查询功能也不太全面。SimpleDB 和 Datastore 使用的都是“实体-属性-值”(Entity-Attribute-Value) 的 EAV 数据模型。微软的 SQL Azure 是云环境下的关系数据库, 并支持报表、数据同步等服务。VMware 在最近的 CloudFoundary 中采用了 10gen 开发的开源云数据库 MongoDB, 可以实现均衡性较好的分布式数据库存储。表 11-4 是四种数据库之间的比较。

表 11-4 商业云计算方案的数据库服务比较

	Google Datastore	Amazon SimpleDB	微软 SQL Azure	VMware MongoDB
系统结构	实体组、实体、属性、值四级模式	域、条目、属性、值四级模式	Authority、容器、实体三级模式	集合、文档、域、值四级模式
主要存储的数据类型	结构化和半结构化数据	结构化数据	结构化数据	结构化和半结构化数据
所用的查询语言	GQL	支持有限的 SQL 语句	SQL	BSO
数据更新时间	有延迟，但不是常态	有延迟	没有延迟	有延迟
实现的功能	较多	最少	最多	较多
其他数据库服务	无	运行在 EC2 上的 Oracle、SQL Server 等	无	运行在 vCloud 上的 Oracle、SQL Server 等

从上述比较中不难发现，四种商业云计算解决方案在应用场景、使用流程、体系结构、实现技术、核心业务等方面都存在较大的差异。但不同方案之间没有绝对的优劣之分，仅有适用场合的区别，用户可在确定自身的需求后进行选择。

11.2 主流开源云计算系统比较

开源云计算系统为个人和科研团体研究云计算技术提供了平台，也为企业根据自身需要研发相应的云计算系统提供了基础。利用开源云计算系统，可以在低成本机器构成的集群系统上模拟出近似商业云计算的环境。

随着云计算研究的不断发展，开源云计算系统也层出不穷。其中，有对成熟商业云计算系统的模仿实现，例如模仿 Google 云计算系统的 Hadoop、模仿 Amazon EC2 和 S3 的 Eucalyptus；也有根据特定需要设计和开发的云计算系统，例如面向科学计算的 Nimbus、密集型数据应用的 Sector and Sphere 等；还有针对特定服务的云计算系统，例如面向存储的 Cassandra、VoltDB、MongoDB 等。

为了帮助用户更好地选择符合需要的开源计算系统，本节从开发目的、体系结构、实现技术和核心服务四个方面，对 Hadoop^[5]、Eucalyptus^[6]、Nimbus^[7]、Sector and Sphere^[8] 这四种同时包含了计算和存储服务的主流开源云计算系统进行比较分析。关于这四个开源云计算系统的具体细节参见前面的相关章节。

11.2.1 开发目的

Hadoop 旨在提供与 Google 云计算平台类似的开源系统，由开源组织 Apache 孵化。对于 Google 云计算平台中包含的 GFS、MapReduce、Bigtable 等组件，Hadoop 中分别有 HDFS、MapReduce、HBase 等开源实现的组件与之对应。此外，Hadoop 还包含了若干个独立的子系统，例如分布式数据仓库 Hive、分布式数据采集系统 Chukwa、远程过程调用方案 Avro。由于 Hadoop 具有良好的性能和丰富的功能，其改进版本目前已经在中国移动、淘宝等公司得到了应用。

Eucalyptus 提供了 Amazon EC2 的开源实现，并实现了类似于 Amazon S3 的存储系统 Walrus。此外，与 Amazon 不同的是，Eucalyptus 采用了模块化设计，其组件可以替换和

升级，允许研究者对云计算系统的安全性、可扩展性、资源调度、接口实现等进行测试，为开展云计算研究提供了很好的平台。目前，Eucalyptus 除免费的开源版本外，还拥有了功能更完善的商业版本。

Nimbus 旨在提供用于科学试验的开源云计算系统，是科学云的一部分，由网格中间件 Globus 开发和更新。Nimbus 的结构相对复杂，但具有很强的可定制性，可以根据不同目标来选择需要的组件进行组合使用。随着 Nimbus 的发展，Nimbus 的应用逐步超出了科学计算范围，开始涉及其他领域。

Sector and Sphere 旨在提供面向数据密集型应用的开源云计算系统，由 Yunhong Gu 等人设计和实现。Sector 是部署在广域网上的分布式存储系统，Sphere 是建立在 Sector 之上的计算服务。相对于 Hadoop 等平台，Sector and Sphere 能够提供更加高速和安全的数据传输，更加适应于极高速网络和大型数据集。

表 11-5 主流开源云计算系统比较

	Hadoop	Eucalyptus	Nimbus	Sector and Sphere
参照的商业方案	Google	Amazon	无	无
提供的服务类型	PaaS	IaaS	IaaS	PaaS
服务间的关联度	所有服务被捆绑在一起，耦合度高	可以选择服务组合，耦合度低	可以选择组件来实现不同的服务，耦合度低	所有服务被绑在一起，耦合度高
支持的编程语言	Java	多种	多种	C++
使用限制	较多	最少	较少	较多
支持的功能	最多	较多	较多	较少
可定制性	较弱	较弱	较强	较弱
可扩展性	自动扩充所需资源并进行负载均衡	需要手动或通过编程自动的增加所需的虚拟机数量	需要手动或通过编程自动地增加所需的虚拟机数量	自动扩充所需资源并进行负载均衡
特色	实现了 Google 云计算系统的关键功能，得到了广泛应用	支持多种 AWS 客户端接口，可替换和升级的模块化设计	可选择组件并进行组合，适用于科学计算	更高速和安全的数据传输，支持密集型数据应用

11.2.2 体系结构

Hadoop 采用与 Google 云计算平台类似的体系结构，主要由 Hadoop Common、HDFS、MapReduce、HBase、Zookeeper 等组件构成。其中，Hadoop Common 是整个 Hadoop 项目的核心，其他子项目都是在其基础上发展起来的；HDFS 是支持高吞吐量的分布式文件系统；MapReduce 是大规模数据的分布式处理模型；HBase 是构建在 HDFS 上的、支持结构化数据存储的分布式数据库；ZooKeeper 用于解决分布式系统中的一致性问题。此外，Hadoop 还包含了若干个相对独立的子项目，例如用于管理大型分布式系统的数据采集系统 Chukwa、提供数据摘要和查询功能的数据仓库 Hive 等。

Eucalyptus 具有简单的分层拓扑结构和模块化的设计，其中使用了开源的 Web 服务技术，每个组件都由若干个 Web 服务组成。Eucalyptus 的主要组件包括节点控制器、集群控制器和云控制器。其中，节点控制器用于管理单个物理节点，负责启动、检查、关闭和清除虚拟机实例等工作；集群控制器运行在集群的头节点或服务器上，用于管理一个或多个

节点控制器,负责从其所属的节点控制器收集状态信息、调度虚拟机实例到各个节点控制器和管理公共和私有实例网络配置;云控制器用于为用户提供可见入口点和做出全局决定,负责处理用户请求和管理请求、进行高层虚拟机实例调度决定和处理服务等级协议和维护元数据。

Nimbus 中包含了一整套的开源工具,在提供的工具集中,工作区服务是整个平台的核心模块,此外还包括工作区控制器、工作区资源管理器、工作区指示器、Web 服务资源框架、资源管理 API、标准客户端、云客户端等。

Sector and Sphere 主要包含了负责分布式存储的 Sector 和架构在 Sector 之上负责分布式计算 Sphere 两部分。其中, Sector 包含了主服务器、安全服务器和从节点; Sphere 中包含了 Sphere 处理引擎、Sphere 客户端等。

11.2.3 实现技术

Hadoop 在功能上尽可能地模仿 Google 云计算平台,实现分布式文件存储系统 HDFS、计算系统 MapReduce、分布式数据库 HBase 等。但对于 Google 云计算平台的部分功能, Hadoop 在实现上依然存在着差距。例如, Hadoop 中使用 ZooKeeper 代替 Google 云计算系统中的 Chubby,但前者在功能上存在一定的不足。

Euclyptus 实现了类似 Amazon EC2 和 S3 的组件。在系统中,每个虚拟机运行的磁盘镜像存储在计算节点上,这可以促进虚拟机的分散。Euclyptus 的结构中对用户空间和管理员空间有很明显的界线:超级用户的访问需要通过物理机管理员进行,而用户仅仅允许通过 Web 接口或者其他前端工具访问系统。

Nimbus 总体架构较为复杂,但提供了很强的可定制性,用户可以根据自身目标进行定制。在定制中, Nimbus 对底层虚拟机的细节做了较多的保护,只是为用户提供了很多自定义项和包括镜像存储在内的一些常用组件。此外, Nimbus 提供了两套 Web 服务接口,可以支持 Amazon EC2 WSDL 和符合网格社区 WSRF 规范的接口。

Sector and Sphere 分为 Sector 和 Sphere 两部分。Sector 中的主服务器负责维护文件的元数据、控制所有从节点的运行、与安全服务器通信进行验证。安全服务器负责维护用户信息、文件访问信息和从节点的 IP 地址。从节点用于存储数据。Sphere 中对不同应用采用统一的数据流形式输入,并在对数据流分割的基础上实现负载平衡和大规模并行计算。

11.2.4 核心服务

在计算服务、存储服务、数据库服务三大核心服务上, Hadoop 所提供的服务与 Google 云计算平台十分类似,分别由 MapReduce、HDFS、HBase 组件承担; Eucalyptus 提供了与 Amazon EC2 和 S3 类似的计算和存储服务,用户可以在计算环境中运行数据库软件来实现数据库服务; Nimbus 中提供了计算和存储服务,用户可以在计算环境中运行数据库软件来实现数据库服务; Sector and Sphere 提供了与 Hadoop 相似、更适用于极高速网络 and 大规模数据集的计算和存储服务,但没有提供数据库服务。表 11-6 展示了四个主流开源云计算系统在核心服务方面的比较。

表 11-6 开源云计算系统的核心服务比较

	Hadoop	Eucalyptus	Nimbus	Sector and Sphere
计算服务	基于 MapReduce 的计算任务	支持用户程序的计算环境	支持用户程序的计算环境	提供分布式计算的 Sphere
存储服务	提供分块存储的 HDFS	提供分布式存储的 Walrus	提供分布式存储的 Cumulus	提供分布式存储的 Sector
数据库服务	提供布式数据库的 HBase	用户运行的数据库软件	用户运行的数据库软件	无

11.3 国内代表性云计算平台比较

在国外云计算快速发展的同时，国内的企业也正在加快部署自己的云计算平台。国内的企业或是通过与国外厂商进行合作，或是在 Hadoop 等开源技术的基础上自主研发云计算产品。其中，中国移动和阿里巴巴作为国内云计算技术研究的先行者，已经开始提供实质性的云服务。阿里巴巴专门成立了从事云计算业务的新公司“阿里云”，而中国移动也从 2007 年开始启动了“大云”计划，下面将对两者云计算的发展做详细的介绍。

11.3.1 中国移动“大云”

中国移动的“大云”计划^[9]，是以中国移动研究院为主体，为打造中国移动云计算基础设施而实施的关键技术研究及原型系统开发计划。开展“大云”计划有两个目的：一是为满足中国移动 IT 支撑系统高性能、低成本、高扩展性、高可靠性的 IT 计算和存储的需要；二是为满足中国移动提供移动互联网业务和服务的需求。

中国移动基于“大云”计划推出了三朵“云”：

- (1) 支撑云。主要是构建私有云并应用于内部支撑系统、网管支撑系统等。
- (2) 业务云。主要支撑 GPRS、无线网、彩信等业务。
- (3) 公众服务云。主要提供各层的云服务，包括 IDC 上的推广服务。

2007 年，利用闲置的 15 台 PC 服务器，基于开源 Hadoop 系统搭建了海量数据处理试验平台，并成功运行搜索引擎软件。2008 年年底，中国移动进一步建设了 256 台 PC 服务器、1000 个 CPU Core、256TB 存储组成的“大云”试验平台，并成功运行数据挖掘工具等应用。在结合现网数据挖掘、用户行为分析等需求方面进行了应用试点，在提高效率、降低成本、节能减排等方面都取得了显著效果。2009 年 8 月，移动发布了供中国移动研究院内部研究测试的“大云”0.5 版本。2009 年年底，“大云”试验平台进一步扩容，达到 1000 台服务器、5000 个 CPU Core、3000TB 的存储规模。在 2010 年 5 月，发布了“大云”1.0 版本并对外发布了五个系统，目前 1.0 版本也多用于内部，对外的盘古搜索和双业务资源池也都采用了“大云”系统。“大云”的技术架构，总体分四个层面^[13]，如图 11-1 所示。

1) 虚拟化资源层

虚拟化资源层是硬件层面的东西，基于开源的 Linux 系统，使用开源 Xen、KVM 提供计算资源的虚拟化。通过对计算资源、网络资源和存储资源进行集中管理和调度，并与用户自服务流程进行管理整合，提供类似 Amazon EC2 的服务即弹性计算系统(BC-EC)。同时还提供方便的用户界面进行资源申请，并可快速进行系统和应用的自动化部署。

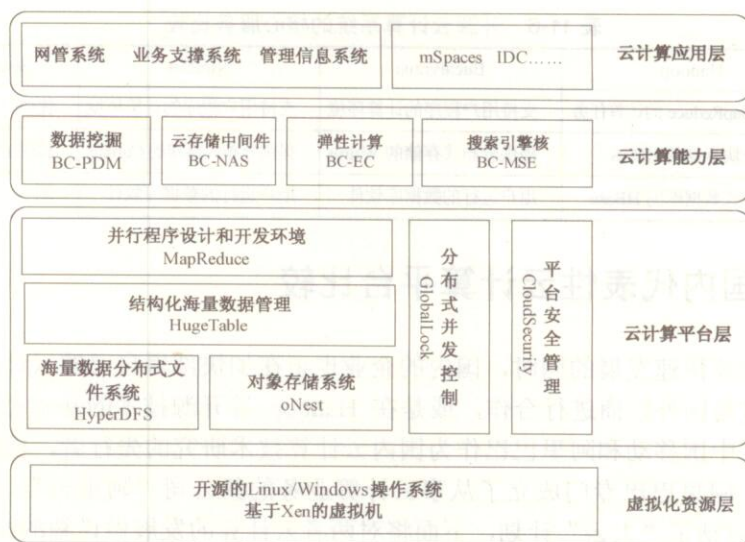


图 11-1 大云的技术架构

2) 平台层

平台层主要提供“大云”的可编程接口和技术支持，包含以下几个功能模块。

(1) 分布式文件系统 HyperDFS：为保证高可用、高可靠和经济性，采用分布式冗余存储的方式来存储数据，以高可靠软件来弥补硬件的不可靠。

(2) 分布式数据库 HugeTable：采用列存储的数据管理模式，保证海量数据存储和分析性能。

(3) 并行计算系统 MapReduce：采用 MapReduce 并行编程模式，将任务自动分成多个子任务，通过 Map 和 Reduce 两步实现任务在大规模计算节点中的调度与分配，保证后台复杂的并行执行和任务调度向用户和编程人员透明。

3) 能力层

能力层主要是面向应用开发的一些中间件和工具级，这些系统是“大云”里主要的研发的产品线。目前包括以下几种产品。

(1) 并行数据挖掘工具 BC-PDM：基于“大云”的并行数据挖掘工具库，提供基于 SaaS 的数据挖掘服务，支持高性能低成本的商务智能应用开发。

(2) 云存储中间件 BC-NAS：提供海量对象存储能力。

(3) 弹性计算 BC-EC：使用开源 Xen、KVM 提供计算资源的虚拟化，通过对计算资源、网络资源和存储资源进行集中管理和调度，并与用户自服务流程进行管理整合，提供弹性计算服务。

(4) 搜索引擎核 BC-MSE：提供基本的搜索引擎能力。

4) 应用层

主要是“大云”上的支撑应用，可以支撑网管、管理信息系统、互联网应用等。

在云计算的发展战略上，中国移动提出了“构建云计算信息服务平台，为个人用户和企业用户提供强大的移动互联网信息服务”的口号。在未来的“大云”发展中，将基于

“大云”平台整合包含网络资源、用户资源、业务能力资源等各种资源，并能提供强大的个性化移动互联网信息服务。

11.3.2 阿里巴巴“阿里云”

2009 年 9 月，阿里巴巴集团在十周年庆典上宣布成立新的子公司“阿里云”，该公司专注于云计算领域的研究。依托云计算的架构做一个可扩展、高可靠、低成本的基础设施服务，支撑包括电子商务在内的互联网应用的发展，从而降低进入电子商务生态圈的门槛、成本，并提高效率。所以阿里巴巴的云计算也被称为电子商务云。

“阿里云”的定位是云计算的全服务提供商。针对云计算不同层次，“阿里云”都进行了充分的部署，开发了自己的技术。

在云计算的底层，“阿里云”开发了自己的云计算平台^[10]。其中包括了分布式操作系统“飞天”、分布式文件系统“盘古”、分布式存储系统“有巢”、命名服务“女娲”、任务调度系统“伏羲”、消息中间件“夸父”和虚拟机资源集群管理系统“后羿”。虽然这些产品的技术细节外界还无法得知，但是可以看出“阿里云”有着相当深厚的技术储备。

在 PaaS 层，“阿里云”做了许多的尝试。首先，“阿里云”开发了一种类似 Google App Engine 的 XEngine 平台^[10]。Google App Engine 实现了对 Python/Java 的应用的托管，但是国内广大的中小 ISV（独立软件开发商）更多是使用 PHP/.NET。针对国内使用者这一特点，“阿里云”开发了可以托管 PHP 应用程序的平台——XEngine。XEngine 是一个分布式的服务器体系，其选择空闲应用容器，部署应用并开始提供服务。应用可以通过 XEngine 提供的 Library 调用 Apsara 提供的各种服务。XEngine 最终要实现与 Apsara 集成，提供更多的 API。但是现在没有看到任何 XEngine 商用的迹象，同时关于 XEngine 的资料也很匮乏，也许 XEngine 只是“阿里云”的实验性开发。

PHPWind 在 2008 年 5 月被阿里巴巴集团收购，前期隶属于支付宝，2010 年 4 月正式进入阿里云计算有限公司。PHPWind 提供了统一的建站工具以及各种接口，方便中小网站在 PHPWind 这个平台上建立自己的网站和相应的应用，并享受到淘宝、支付宝等的数据和服务^[11]。在 PHPWind8.3 版本中加入了“云统计”功能，这个功能和 Google 的 Google Analytics 类似。站长可以通过“云统计”利用淘宝的数据中心进行用户的行为分析，同时这也意味着站长向阿里云中心开放了自己的用户数据。阿里巴巴通过 PHPWind 平台，把用户的信息和行为整合在了一起，PHPWind 将会是阿里巴巴发展云计算的一个重要的平台。

“阿里云”只涉及基础研究，不涉及具体的软件产品的开发。“阿里云”会为阿里巴巴集团内其他公司提供技术支持，和其他的技术团队一起开发在线服务。现在“阿里云”已经为淘宝提供部分图片服务，为淘宝提供数据分析的服务，同时还为口碑网提供地图服务。

11.3.3 “大云”与“阿里云”的比较

由于中国移动和阿里巴巴两个企业专注的领域不同，两者的云计算发展的策略、目标及涉及的技术存在一些差异。

两者云计算发展的相同点在于：① 都致力于做云计算的全服务提供商，他们的产品和服务都涉及从 IaaS 到 SaaS 的三个层面；② 在技术上，两者在开发云计算平台上都采

用了开源的技术如 Hadoop、Xen 等。

两者的不同点在于：① 发展目标不同。中国移动作为运营商，其发展云计算的主要目标是为了满足中国移动提供移动互联网业务和服务的需求。而“阿里云”则是为阿里巴巴集团提供云计算支持，使阿里巴巴可以依托云计算建立一个全新的电子商务圈，“阿里云”更注重电子商务云计算。② 针对的客户不同。中国移动的“大云”主要为集团内客户以及个人客户提供服务。“阿里云”在专注于集团内部客户的同时，通过 PHPWind 平台等针对中小网站和企业提供云服务。③ 采用的技术存在差异。虽然两者都在开源的平台上研发云计算技术，但是两者都对开源系统进行了不同程度的修改和优化。详细的对比如表 11-7 所示。

表 11-7 “大云”与“阿里云”的比较

	“大云”	“阿里云”
提供的服务类型	IaaS、PaaS、SaaS	IaaS、PaaS、SaaS
发展目标	满足互联网业务的需求	注重电子商务云计算，打造全新电子商务圈
目标客户	集团内客户及个人客户	集团内客户及中小型企业
服务间的关联度	可以任意选择服务功能，耦合度较低	可以实现服务组合，耦合度较低
虚拟化技术	Xen	Xen、KVM
运行环境	自身提供的云平台	自身提供的云平台
使用限制	较少	较少
主要功能	并行数据挖掘工具（BC-PDM），搜索引擎核（BC-MSE），云存储系统（BC-NAS）	提供存储、计算以及数据挖掘、分析等服务

11.4 云计算的历史坐标与发展方向

云计算的兴起无疑是互联网发展史上最激动人心的事件之一。那么，云计算在互联网和信息社会的发展历程中究竟处于什么位置？云计算的发展方向是什么？本节将对这些问题进行分析。

11.4.1 互联网发展的阶段划分

回顾互联网的发展历史，展望未来，可以将互联网的发展进程大致划分为三个阶段。

1. 第一代互联网

1969 年，为了能在核战争爆发时保障通信畅通，美国国防部启动了具有抗核打击能力的计算机网络 ARPANET 研究项目。ARPANET 建立在分组交换技术之上，首批连接了美国四所大学。

从 20 世纪 70 年代末开始，个人计算机兴起，各式各样的计算机网络应运而生，如 MILNET、USENET、BITNET、CSNET 等。由此产生了实现不同网络之间互联的需求，导致了 TCP/IP 协议的诞生。1982 年，ARPANET 开始使用 IP 协议。

1986 年，美国国家科学基金会（NSF）资助建成了基于 TCP/IP 的主干网 NSFNET，连接了主要的科研机构，第一代互联网由此诞生。

这个阶段只有极其少量的计算机实现了相互连通，可以进行数据通信，所能够支持的

应用非常少,发挥的作用不大。然而,第一代互联网的意义却十分深远,它使人类社会初现信息社会的雏形,我们将其称之为“信息社会 0.1”。

2. 第二代互联网

1989 年, Tim Berners-Lee 提出万维网(WWW)的设想。他发明了超文本,使用超级链接将不同服务器上的网页互相链接起来,从而使人们很容易访问相互关联的信息。他将这项发明无偿提供给全世界使用。WWW 的出现推动互联网用户数呈指数增长。从 1995 年到 2002 年,互联网用户数平均每半年翻一番。

2003 年后, WWW 从单纯通过浏览器浏览 HTML 网页的 Web 1.0 模式演化到方便大量用户共同参与互联网内容编织的 Web 2.0 阶段。新的应用应运而生,如博客(Blog)、社会关系网络(SNS)、维基百科(Wiki)、内容聚合(RSS)、混搭编程(Mashup)等。

在 20 年的时间里, WWW 给全球信息交流和传播带来了革命性的变化,改变了商业运作模式,改变了人们的生活方式,改变了知识的获取和形成模式,缔造了许多优秀的 IT 公司,如 Google、eBay、Amazon、Facebook、腾讯、阿里巴巴等。这个阶段,我们将其称之为“信息社会 1.0”。

3. 第三代互联网

第三代互联网的呼声开始于上世纪 90 年代末。当时以网格技术、Web Services、IPv6 等为代表的新技术不断涌现,让人们看到了将网上所有信息资源融为一体的希望。然而,十多年的发展历程证明,人们对网络资源融合的预期过于乐观了。网络资源融合除了存在技术上的障碍外(例如互操作技术标准体系、信息安全等),还受到许多非技术因素的影响(例如政策因素、商业模式、利益冲突等)。

但是,网络资源高度融合的一天迟早会到来。云计算技术从 2007 年突然兴起,且迅速形成盈利模式,正式掀开了第三代互联网的面纱。看似突然,实际上是偶然中的必然——云计算只是众多下一代互联网技术中率先突围的一个。

我们认为,第三代互联网将实现信息节点之间的大协作,实现信息系统之间的互操作,实现信息平台一体化,从而构成紧密星球(Compact Planet)。第三代互联网的基础是无处不在的宽带网络, Internet 2 和 NGN 将融为一体, Wifi、3G、LTE 等的普及将导致终端泛化、全民上网。笔记本、上网本、手机、PDA、摄像头、传感器、RFID、电视、冰箱、汽车等,都将与互联网相连,从而使得几乎每一个物体、每一个人都成为互联网的一部分。第三代互联网的信息将以富媒体(Rich Media)的形式存在。它是由各种方式产生的多种媒体的有机集成,支持非特定人员的动态参与和协作,并可自适应地呈现在各种终端上和各种应用系统中。人类的所有信息将朝着被有序管理的方向迈进。第三代互联网的时代,我们称之为“信息社会 2.0”。

4. 三代互联网的比较

在第一代互联网时期,网络缺乏与传统信息传播业竞争的實力,传统行业(如电信、电视、新闻、出版、广告等)占主导地位。在第二代互联网时期,网络提高了传统行业效率,开始与传统信息传播业分庭抗礼。数字出版第一次超过了传统出版,《读者文摘》破产就是一个典型的代表。几乎所有人类行为都将与网络关联,90%以上的物理能力和行为都将在网络中体现,人们在网络上开展业务运营、视频监控、电子商务、金融服务、通信

联络等。

前两代互联网技术革命都是由美国引领的，第三代互联网也诞生在美国。究其原因，其科技创新模式和风险资本运作模式是一个重要因素。然而，我们相信，第三代互联网最终将由中国来引领。主要原因是：中国有无与匹敌的市场规模，有无与伦比的决策和执行效率（如收放自如的市场调控能力等）。目前，中国的科技创新体系已经初步形成，自主研发能力提升迅速，正依次在信息化程度越来越高的行业取得优势：从制造业（钢铁、轮船、集装箱、重型机械等），到精密制造业（汽车、高速列车、飞机、电子产品等），再到信息、服务业（电子商务、软件等）。根据世界知识产权组织（WIPO）公布的 2008 年国际专利申请状况，华为公司申请数名列榜首。根据世界著名的消费市场研究机构 Euromonitor 在 2009 年 12 月发布的调查结果，海尔冰箱在全球市场占有率已经位列第一。根据 Dell'Oro 在 2009 年 5 月公布的统计数据，华为、中兴通讯在全球移动通讯设备市场的排名已经分列全球第三和第五。我国的电子商务交易 2010 年达到 4.5 万亿元，阿里巴巴已经在 B2B 领域保持国际领先。近期我们已经看到中国高速铁路技术已经开始引领世界，我们将看到中国在更多高技术领域领先，包括移动宽带、云计算、IPv6、物联网、大飞机等领域。

表 11-8 呈现了对三代互联网横向对比的结果。

表 11-8 三代互联网的比较

	第一代互联网	第二代互联网	第三代互联网
社会形态	信息社会 0.1	信息社会 1.0	信息社会 2.0
历史时期	1970 年代，主机时代 1980 年代，PC 时代	1990 年代，Web 1.0 时代 2000 年代，Web 2.0 时代	2010 年代，云计算时代 2020 年代，云格时代
具体时段	1969—1989（20 年） 1969: ARPANET 诞生	1989—2007（18 年） 1989: WWW 诞生	2007—2023（16 年） 2007: 云计算诞生
主要特征	实现计算机与计算机的通信 连通	实现网页与网页的连通	实现信息平台的一体化
典型技术	分组交换传输技术（TCP/IP）	WWW、宽带网、Web 2.0	云计算、IPv6、移动宽带网、 Web Services、网格计算、物联网、 云格（Gloud）
媒体类型	文本	多媒体（MultiMedia）	富媒体（RichMedia）
典型应用	电子邮件、FTP、资料检索系统	搜索引擎、新闻、电子商务、论坛、聊天、视频、文件共享	计算资源租用、在线 CRM、在线 Office、GIG、一体化服务
典型特征	手工操作	半自动操作	信息随手可得
网络的地位	网络无力与传统信息传播业竞争 传统行业（包括电信、电视、新闻、出版、广告等）占主导地位	网络提高了传统行业效率 与传统信息传播业分庭抗礼	网络占绝对统治地位。 2009 年数字出版产值第一次超过了传统出版，美国标杆传统期刊《读者文摘》破产
潮流引领者	美国引领 • 军方需求推动	美国引领 • 科技创新模式 • 风险资本运作模式	中国引领 • 无与匹敌的市场规模优势 • 无与伦比的决策和执行效率

11.4.2 云格（Gloud）——云计算的未来

通过本书前面章节的学习，我们发现，云计算的规模可以动态扩展，处理能力超强，

存储空间海量,高度可靠,资源利用率很高,通用性很强而且成本极低。这些特性决定了云计算正在以前所未有的速度迅速扩张,使得传统IT企业纷纷转型^[12]。目前,硅谷已经涌现出上百家新型云计算创新企业,颇有20世纪90年代互联网刚刚兴起时的势头。

云计算无疑是迄今最为成功的商业计算模型,但它并不是完美无缺的,它的一些缺陷却是网格技术所擅长的。具体说来体现在以下五个方面:

(1)从平台统一角度看,目前云计算还没有统一的标准,不同厂商的解决方案风格迥异、互不兼容,未来一定会朝着形成统一平台的方向发展;而网格技术生来就是为了解决跨平台、跨系统、跨地域的异构资源动态集成与共享的,而且国际网格界已经形成了统一的标准体系和成功应用。网格技术能够帮助完成在云计算平台之间的互操作,从而实现云计算设施的一体化,使得未来的云计算不再是以厂商为单位提供服务,而是以构成的一个统一的虚拟平台提供服务。因而,可以预见,云和云之间的协同共享离不开网格的支持。

(2)从计算角度看,云计算管理的是由廉价PC和服务器构成的计算资源池,主要针对的是松耦合型的数据处理应用,对于不容易分解成众多相互独立子任务的紧耦合型计算任务来说,采用云计算模式处理数据的效率会很低(因为节点之间存在频繁的通信);网格技术能够将分布在不同机构的高性能计算机集成在一起,处理云计算不擅长的紧耦合型应用(如数值天气预报、汽车模拟碰撞试验、高楼受力分析等)。如果云计算与网格技术能够一体化,则可以充分发挥各自特点。

(3)从数据角度看,云计算主要管理和分析商业数据;网格技术已经集成了极其海量的科学数据,如物种基因数据、天文观测数据、地球遥感数据、气象数据、海洋数据、药物数据、人口统计数据等。如果将云计算与网格技术集成在一起,则可以大大扩大云计算的应用范围。目前亚马逊在不断征集供公众共享使用的数据集,包括人类基因数据、化学数据、经济数据、交通数据等,这充分说明可以利用云计算处理这些数据集,同时也反映出这种征集数据的方法过于原始。

(4)从资源集成角度看,使用云计算,就必须将各种数据、系统、应用集中到云计算数据中心,如果改变很多现有信息系统的运行模式,把他们迁移到云计算平台上,将面临难度和成本的双重挑战。特别是,有些系统的数据源距离数据中心可能较远,且数据源的数据是不断更新的(物联网就具有此种特性),若要求随时随刻将这些数据传送到云计算中心,则会消耗大量的网络带宽。因此,会有大量的应用系统处于分散运转状态,而不会集中到云计算平台上去;而网格技术可以在现有资源上实现集成,达到“物理分散、逻辑集中”的效果,巧妙地解决这方面的问题。

(5)从信息安全角度看,许多用户担心将自己宝贵的数据托管到云计算中心,就相当于丧失了对数据的绝对控制权,存在被第三方窥看、非法利用或丢失的可能,从而不敢采用云计算技术;而在网格环境中,数据可以仍然保存在原来的数据中心,仍然由其所有者管控,对外界提供数据访问服务,只是一种“可以用,但不能全部拿走”的模式,不会丧失数据的所有权,但数据资源的使用范围扩大了、利用率提高了。由于数据源头分别由不同所有者控制,所有者可以决定是否共享数据,以及在什么范围共享,避免了敏感数据的扩散,比把所有数据都放进云计算数据中心进行共享更安全。

因此,云计算与网格技术之间是互补关系,而不是取代关系。网格技术主要解决分布在不同机构的各种信息资源的共享问题,而云计算主要解决计算力和存储空间的集中共享使用问题。可以预见,云计算与网格技术终将融为一体,这就是云计算的明天,作者给它

取了个名字，叫云格（Gcloud），即 $Gcloud=Grid+Cloud$ 。

云计算与网格技术结合的基准将是面向服务的体系结构 SOA（Service Oriented Architecture）。SOA 最早由 Gartner 公司于 1996 年正式提出。SOA 架构模型的本质是业务建模——将一切信息资源封装成服务，以服务形式来解决业务间的互操作问题。目前，无论是网格技术，还是云计算，基本上都符合 Web Services 规范。Web Services 是 SOA 的实现机制之一。Web Services 是由 URI（Uniform Resource Identifier）标识的软件应用。该应用的接口和绑定可通过基于 XML 的语言进行定义、描述和发现。同时，该应用可通过基于互联网的 XML 消息协议与其他软件应用直接交互。Web Services 是简单的、标准的、跨平台的且与厂商无关的，可以大幅度降低构架耦合度，提供服务层次的集成。

在 SOA 框架下，无论是网格服务，还是云计算服务，或其他的 Web Services 服务，都能够非常容易地共存、共用、互操作。在这样的环境中，用户不必分清楚哪些部分是传统的 Web Services，哪些是云计算，哪些是网格技术，只要关心自己需要哪些服务（数据处理服务、高性能计算服务，企业管理服务、电子商务服务，等等），如何获取服务即可。如图 11-2 所示。

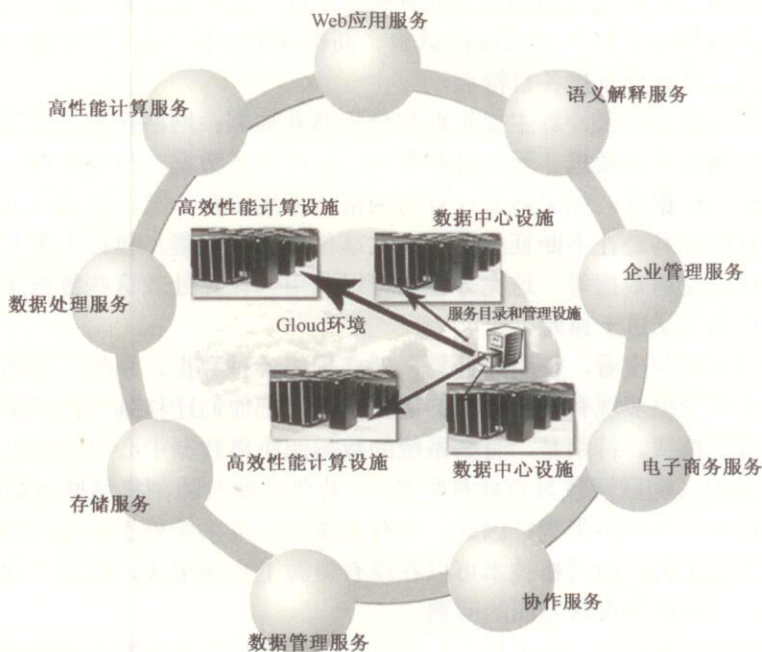


图 11-2 云计算与网格服务的融合远景

因此，我们有理由相信，未来的云格时代会更加美好！

习题

1. 查阅资料，列举其他商业云计算解决方案的应用场景。
2. 用图形方式描述 Google、Amazon、微软和 VMware 云计算平台的体系结构。
3. Google、Amazon、微软和 VMware 云计算平台在使用流程上有何异同？

4. 以“体系结构”为比较点, 在表 11-5 的基础上完善 11.2 节中关于主流开源云计算平台的对比。
5. 查阅资料, 简单描述一个适于“云格”技术思路解决问题的领域或场景。

参考文献

- [1] Google. Google App Engine.. <http://code.google.com/appengine/>
- [2] Amazon. Amazon Web Service.. <http://aws.amazon.com/>
- [3] Microsoft. Introducing the Windows Azure Platform (Final PDC10).
<http://go.microsoft.com/?linkid=9752185>
- [4] VMware. VMware vCloud. <http://www.vmware.com/products/vcloud/>
- [5] T. White. Hadoop: The definitive guide. 2nd Edition. O'Reilly Media, 2010
- [6] Eucalyptus. Eucalyptus open-source cloud computing infrastructure - An overview.
<http://www.eucalyptus.com/whitepapers>
- [7] K. Keahey. Cloud computing with Nimbus. Chicago: Fermilab, 2009
- [8] Y. Gu, and R.L. Grossman. Sector and Sphere: The design and implementation of a high-performance data cloud. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 2009, 367(1897):2429-2445
- [9] 中国移动通信. 中国移动大云. <http://labs.chinamobile.com/focus/bigcloud/>
- [10] 吴峥涛. XEngine 介绍.
<http://ecug.googlecode.com/svn-history/r371/trunk/cn-erlounge/iv/wuzhengtao/XEngine.pdf>
- [11] 至顶网. PHPwind 是阿里集团向用户延展业务的重要平台.
http://soft.zdnet.com.cn/software_zone/2010/1220/1969965.shtml
- [12] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the cloud: A Berkeley view of cloud computing. <http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf>
- [13] <http://cloud.it168.com/a2010/0521/889/000000889396.shtml>