|  |  |
| --- | --- |
|  | 学校代码： 10246 |
|  | 学 号： 14210240068 |
|  |  |



|  |
| --- |
| 硕 士 学 位 论 文 |

**（专业学位）**

|  |
| --- |
| **基于私有云的缓存数据一致性**  **维护技术与实现** |

|  |
| --- |
| **Design and Implementation of Cache Data Consistency Maintenance Technology Based on Private Cloud** |

院 系： 计算机科学技术学院

专业学位类别（领域）： 计算机技术

姓 名： 程沛

指 导 教 师： 顾宁 教授

完 成 日 期： 2017 年 3月 25 日

指导成员小组名单

顾宁 教授

张亮 教授

卢暾 副教授

丁向华 副教授

目录

[目录 I](#_Toc483558213)

[摘要 III](#_Toc483558214)

[Abstract V](#_Toc483558215)

[第一章 引言 1](#_Toc483558216)

[1.1 研究背景与意义 1](#_Toc483558217)

[1.2 国内外研究现状 2](#_Toc483558218)

[1.2.1 数据一致性维护的研究现状 2](#_Toc483558219)

[1.2.2 热点数据动态迁移问题的研究现状 5](#_Toc483558226)

[1.2.3 缓存存储数据结构和缓存替换算法的研究现状 6](#_Toc483558227)

[1.3 论文的研究内容 6](#_Toc483558228)

[1.4 论文的组织与结构 7](#_Toc483558229)

[第二章 基于私有云平台的系统设计 9](#_Toc483558230)

[2.1 云计算的服务模式与技术介绍 9](#_Toc483558233)

[2.1.1 云计算的部署方式和解决方案 9](#_Toc483558237)

[2.1.2 主流的云计算开源框架介绍 10](#_Toc483558238)

[2.2 基于私有云平台的系统架构设计 11](#_Toc483558250)

[2.2.1 基于私有云平台的整体架构设计方案 11](#_Toc483558255)

[2.2.2 单个站点的应用架构设计方案 12](#_Toc483558256)

[第三章 基于AST的缓存数据一致性维护技术 15](#_Toc483558257)

[3.1 AST算法的实现原理 15](#_Toc483558261)

[3.1.1 AST算法中的基本概念和定义 15](#_Toc483558266)

[3.1.2 地址空间回溯算法 16](#_Toc483558267)

[3.1.3 控制算法 17](#_Toc483558268)

[3.1.4 操作执行算法 18](#_Toc483558269)

[3.2 缓存结构的设计介绍 18](#_Toc483558275)

[3.3 基于AST的缓存数据一致性维护技术原理 21](#_Toc483558276)

[3.3.1 基于缓存的地址空间回溯算法 21](#_Toc483558283)

[3.3.2 基于缓存的控制算法 23](#_Toc483558284)

[3.3.3 基于缓存的操作执行算法 23](#_Toc483558285)

[第四章 基于AST的缓存数据一致性维护技术的实现 25](#_Toc483558286)

[4.1 基于AST的缓存操作生成和广播 25](#_Toc483558291)

[4.2 基于AST的缓存数据一致性维护技术的操作实现 28](#_Toc483558292)

[4.2.1 本地操作的执行实现 28](#_Toc483558299)

[4.2.2 远程操作的执行实现 32](#_Toc483558300)

[4.2.3 缓存系统使用场景模拟 35](#_Toc483558301)

[4.3 基于AST的缓存数据一致性维护技术的优化 39](#_Toc483558302)

[4.3.1 基于TTL的缓存数据更新 39](#_Toc483558310)

[4.3.2 缓存空间的优化 40](#_Toc483558311)

[第五章 实验与缓存数据一致性维护技术的应用 43](#_Toc483558312)

[5.1 实验 43](#_Toc483558314)

[5.1.1 原型系统介绍 43](#_Toc483558321)

[5.1.2 对比实验设计以及结果分析 43](#_Toc483558322)

[5.2 面向残疾人无障碍自理管理平台的缓存应用 48](#_Toc483558323)

[5.2.1 残疾人无障碍自理管理平台介绍 48](#_Toc483558331)

[5.2.2 残疾人无障碍自理管理平台下部分接口的实现 49](#_Toc483558332)

[5.2.3 残疾人无障碍自理管理平台存在的问题分析 53](#_Toc483558333)

[5.2.4 残疾人无障碍自理管理平台的架构优化与缓存应用 54](#_Toc483558334)

[第六章 总结与展望 59](#_Toc483558335)

[6.1 总结 59](#_Toc483558342)

[6.2 展望 59](#_Toc483558343)

[参考文献 61](#_Toc483558344)

[致谢 67](#_Toc483558345)

摘要

云作为存储资源，计算资源，服务资源等一系列抽象化资源的总称，在系统架构设计中应用越来越为广泛。云计算则是对云进行管理和使用的一种技术。根据云计算对数据的访问控制的特点可以分为公有云，私有云和混合云三种。其中私有云凭借其可定制化，部署灵活，对数据访问更加高效和安全的特点，在云计算服务提供解决方案中的应用越来越为广泛。为了保证私有云平台下的系统具有一定的负载能力，一般情况下设计多个应用服务器节点来构建一个稳定的，可扩展的，具有伸缩性的系统架构。此外，为了保证系统获取数据的高效性，常常会在应用服务器节点中添加缓存。在高吞吐量的负载下，单个节点会出现宕机的情况，为了保证私有云下数据服务的高可用性，常常将原定位到该宕机节点的请求重新定位到其他可用的应用服务器节点。由于缓存原本承担了大量的数据请求压力，而重定位的请求将会直接击穿缓存层，使得数据库的访问压力大大增加，这将导致整个私有云下的数据服务效率下降。针对上述问题，本文的研究工作主要包含以下三个部分:

一.提出了基于私有云的缓存架构设计。本文通过多个应用服务器节点和云控制器节点构建了基本的私有云架构。为了提高系统的负载能力和保证数据服务的高效性，在每个应用服务器节点中添加了缓存，各个缓存节点构成了私有云下的缓存层，为用户提供了高效的数据服务。

二.在基于私有云的缓存架构基础之上，提出了缓存数据一致性维护技术。该技术对AST(即地址空间转换技术)进行了一定的修改和创新，使其能够更好地应用在缓存中。通过维护私有云下每个缓存节点中存储的数据的一致性，使得重定向的请求能够直接从新定位的节点中获取数据，而不会出现大量请求击穿缓存层直接访问数据持久层的情况，解决了原本数据服务在架构动态伸缩的情况下无法保证高效的问题。

三.实现了具有缓存数据一致性维护技术的原型系统UserManage，通过四个不同维度的对比实验验证了缓存数据一致性维护技术的有效性；最后将缓存数据一致性维护技术应用在残疾人无障碍自理管理平台项目中，对该系统下的缓存进行了设计和实现。

**关键词**：私有云 缓存 AST 缓存数据一致性维护技术

**中图法分类号**：TP3

Abstract

Cloud as a general term of storage resources, computing resources, service resources of a series of abstract resources, is more and more widely used in the system architecture design. Cloud computing is a technology that manages and uses virtual cloud resources. According to the deployment mode and the data access control of cloud computing, it could be divided into three kinds, public cloud, private cloud and hybrid cloud. Among them, the private cloud has become more and more popular in the cloud computing solutions because of its customization, flexible in deployment and more secure and efficient in data access. In order to ensure the enough capacity of load under the private cloud platform, it often designs multiple application server nodes to build stable, scalable system architecture. In addition, cache is often used to improve the efficiency of data fetching. In a high-throughput load, a single node could be in a downtime situation. Consequently, to guarantee the high availability of data services, the request to the application server that was originally targeted to the downtime is relocated to other available application server nodes. As the cache assumes a large number of service requests, the relocated request will directly break down the cache layer, making the load of data persistence layer increase greatly, which leads to lower the entire private cloud data service efficiency. Based on the above problems, the research work of this paper mainly includes the following three parts:

First of all, it proposes a cache module architecture design based on the private cloud. It builds up the basic cloud computing architecture by multiple cloud computing nodes and cloud controller nodes. Besides, it adds cache to the cloud computing nodes for enhancing the capacity of load and ensuring the efficiency of data services. Thus, these cache nodes consist of the cache module based on the private cloud. Which provide the efficient data services.

Secondly, it proposes a cache consistency maintain technology based on the proposed private cloud architecture. This technology has made some innovations and changes to the address space transformation technology in order to be more suitably applied in the cache system. By maintaining the consistency of data in the cache node under the private cloud, the redirection request can obtain the data directly from the newly located cache node without the circumstance of a large number of requests

directly access the persistence layer, which solves the problem that data services are not efficient enough in the case of dynamic expansion and contraction of architecture.

Thirdly, it implements the prototype system UserManage with cache consistency technology. The comparison experiment of different dimensions verifies the validity of the consistency technology in application; and the design of the cache is according to the needs of Disabled Accessible Self-care Management Platform.

**Key words:** private cloud cache AST data consistency maintenance technology

**The classification method code**: TP3

# 引言

## 研究背景与意义

在如今海量数据存储和计算越来越为热门的信息时代，云计算[1][2][3][9][10][11][12][13]已经逐渐渗透到信息技术领域的方方面面，并对传统软件的开发部署和交付形式都产生了巨大的影响，它是由网格计算[4]，效用计算[5]，虚拟化技术[6]等多种技术进行融合与创新而产生的一种新技术[7]。根据云计算的访问模式和控制模式来分，可以分为私有云，公有云和混合云三类[8]。

公有云相当于建立了一个计算和存储数据的公共资源，用户向公有云请求资源的时候，公有云将会根据用户的实际资源需求划分给用户使用，类似于日常用水用电一样，用户只需要交付一定的费用，就可以使用国家水电站和国家电网提供的水电[64]；私有云，相对的，则是为用户专门定制服务，根据用户对于数据的安全性访问控制，业务的功能、性能等方面的需求，来专门设计为用户内部业务所需要的服务。因此公有云经常用来提供廉价的计算服务(比如海量数据的计算)，从而降低传统模式下的计算成本；私有云则经常用在企业或者政府组织内部的系统设计中，用来提供定制化的计算服务以及更为安全高效的数据访问。混合云兼具了公有云和私有云的特点，当企业中既需要部署私有云来进行内部业务地安全性管理，又需要利用公有云强大而又廉价的计算功能的时候，就会将混合云来作为云计算解决方案。

私有云[63][64][65][66][73]对用户开放的是服务接口，封闭的是内部的具体实现。对于用户来说，只要通过互联网连接上云，就可以使用私有云提供的服务；而私有云的内部架构则将会按照实际的项目需求背景来设计。总体来说，云计算平台在架构上的特点是具有高可用性，可伸缩性和稳健性等特点。云计算框架的可伸缩性体现在在系统动态运行过程中，能够根据用户的请求流量来动态分配资源，提供合理的资源数目。因此常常将会事先配置好可用的服务器节点，当系统启动的时候，云控制器将会事先调用默认的应用服务器数目来提供数据服务，当在运行过程中，发现用户流量过大将会从配置好的服务器节点列表中进行动态扩展；如果发现用户请求流量减小到一定程度，则会动态减小提供服务的节点数目，从而合理地利用私有云平台上的服务器资源。

在系统设计中，缓存作为应用层和数据持久层的中间层，常常被用来作为加快应用层的数据检索速度以及减轻数据库访问压力的中间件。当云控制器节点收到用户请求时，将会把用户请求分发给其下挂载的各个应用服务器节点，每个应用服务器节点都负责响应指定区域用户的请求。当系统在运行一段时间之后(假设没有服务器在此期间发生宕机的情况)，每个缓存节点中存储的都是数据库中的热点数据，并且由于缓存承担了一部分的访问压力，因此即使系统在访问并发量较大的情况下，数据库也能在较为平稳的访问压力下运行。但是在实际的系统运行中，缓存也会造成系统的性能下降[67][68][69]。当私有云下的服务器节点在运行过程中突然宕机，或者是由于用户流量减少(比如夜间用户访问量骤减)，控制器自动减少其下运行的服务器的情况下，控制器会把原来定位到该台宕机服务器上的请求重定位到其他服务器节点上，这导致在短时间内会出现大量数据请求击穿缓存层直接访问数据库的情况，从而降低私有云下数据服务的效率[74]。

基于以上问题，本文提出了基于AST(Address Space Transformation，地址空间转换技术)[13][76]的缓存数据一致性维护技术。通过研究AST中的算法思想，分析缓存中的数据结构和存储模式的特点，将AST技术进行扩充和创新，使其更好地应用在缓存中。缓存数据一致性维护技术通过维护私有云下每个缓存节点中数据的一致性，从而当服务器节点宕机或者由于用户流量减少而使得服务器节点动态减少的情况下，稳定了其他缓存节点中的缓存命中率，确保数据库服务器在可控的数据访问压力下运行。当私有云中每个缓存节点中的缓存数据一致的时候，整体上来看，相当于将所有的请求中的热点数据均匀地存储在了每个缓存节点上。当发生节点宕机时，用户仍旧可以享受到高效透明的数据服务；对于私有云内部来说，重新定位的请求就如同发送给原服务器一样，因此对于其他受影响的缓存节点来说，不会致使缓存中大部分数据失效，同时也不会对数据持久层在短时间内造成较大的访问压力。最后，本文将缓存数据一致性维护技术应用到残疾人无障碍自理管理平台中，并根据项目的实际需求将系统架构迁移到了私有云平台下，对其中的缓存部分进行了设计和实现。

## 国内外研究现状

### 1.2.1 数据一致性维护的研究现状

现在针对数据一致性维护有以下几个方面的研究方向，一是大型应用系统中常用的基于悲观复制的一致性算法，二是关于数据源和缓存节点之间的一致性维护的研究，三是基于乐观复制的以操作转换技术(OT)和地址空间转换技术(AST)为代表的一致性算法的研究。接下来分这三个部分来详细描述目前对这块的研究现状。

首先是基于悲观复制算法的一致性维护的研究。悲观复制算法中往往包含锁的概念，即副本在未被更新到最新的状态之前都会被锁住。著名的paxos[14][15]算法就是解决对数据有强一致性要求的问题的理论基石。Paxos算法在假设消息不会被篡改的情况下，使得分布式结构的所有参与节点中对于某个变量有一致的认知。在系统应用层面，最开始使用的也是锁机制[16]，例如在很多大型系统中使用的两段锁/三段锁提交协议[17][75]。两段锁/三段锁提交协议中是将数据的提交分成两个部分，这两个部分都有参与者和协调者两种角色，对应于提交数据的节点以及决定数据是否可以提交的节点。第一个部分准备好数据提交的准备工作，并锁定资源，第二部分协调者在收到每个参与者的确认可以提交的响应信息之后，就完成数据提交。这之中但凡有一个参与者的响应信息为拒绝，则协调者就会通知各个参与者回滚。三段锁在两段锁之上进行了改进，在第一阶段并不会把资源锁定，而是在收到所有参与者的同意信息之后才会锁定资源。这里可以看到两段锁/三段锁提交协议在如今要求快速响应和需求高并发的应用背景下，该策略的执行效率可能并不能够满足响应性能方面的需求，因为很多操作会在等待锁的情况下出现响应延迟，大大影响了用户体验。

另外，在数据一致性维护方面，一个很典型的应用场所就是持久层数据和缓存节点之间的数据一致性维护。缓存是持久层数据源的部分拷贝，在系统运行了一段时间之后，缓存中的数据将会出现过期的现象，严重的情况下还会影响系统内部的业务逻辑处理。在这种需求背景下，最为普遍的做法是使用TTL[18][19][20]机制，为缓存中的每个数据项设置存活时间，在存活时间之内该数据在缓存中都是有效的，超过存活时间时，则需要重新从数据库中获取数据，并进行缓存数据更新之后再返回给用户。例如，[21]提出了一种基于客户端的缓存一致性方案，并通过实现自适应性的TTL策略，来提供近乎强一致性的数据。每个缓存数据项的TTL时间是根据数据在数据源中的更新速率来设定的，过期的数据将会批量请求数据源更新，而未过期的但是访问速度很高的数据，则会根据一定的预测算法预先从服务器中获取。还有一些提供缓存一致性的架构设计，在[22][23]中，介绍了一种在移动自组织网络(Mobile ad hoc network)环境下的数据缓存系统，通过用户使用系统中的行为来生成包含缓存查询节点和查询数据集节点的缓存系统。在基于该结构的缓存系统之上，它提出了一种基于服务器的一致性控制策略，通过控制服务器端和缓存节点间的数据版本号，以及根据缓存中数据不同的更新速率，有选择性的更新缓存中的数据，从而使得数据的一致性的程度和系统的性能保持在一个最佳的平衡点。在这些论文中阐述的都是架构设计中如何平衡系统性能和缓存数据更新而做的动态更新策略。除此之外，[24][25]介绍了在不同应用场景下对不同数据的一致性要求，并进行了不同层次的定义，并通过选择距离用户附近的数据源来动态地为用户选择满足应用需求的一致性数据。还有提出过事务型数据如何在缓存节点以及数据库中实现其一致性的算法，用来解决事务型读操作在缓存中的一致性问题(写操作将直接在数据库中执行)，比如TxCache的缓存一致性模型[26]。TxCache模型通过入侵数据库代码来实现每个事务ID和数据库快照ID的绑定，从而为每个事务查询保存了对应的数据源视图。当事务第一次执行的时候，将会在数据库中相关的数据元组中生成标签，在数据放入到缓存中，生成数据开始有效的时间戳。在系统后续的运行过程中，如果数据源中的数据被修改，则将生成标签流，调用后台的线程逐一将标签流发送到各个缓存节点，每个缓存节点则会对存储在自己内部的数据进行识别和数据失效的时间戳。当一个请求到达系统的时候，就可以根据用户对于数据新鲜度的要求，以缓存中标记的有效时间戳和失效时间戳来选择返回对应的数据集。

除此之外，还有基于乐观复制的一致性算法。这类算法的主要代表就是操作转换技术(Operational Transformation，简称OT)[29]和地址空间转换技术(Address Space Transformation，简称AST)[76]。OT和AST都是讨论在组编辑系统下各个站点的副本在同一组操作序列下能够保持最终的一致性的算法。OT算法的主要思想是在协同编辑的时候，将操作进行转化，使得每个站点的最终结果一致，而AST算法，则是将当前站点的文本状态切换到该远程操作执行时的本文状态下执行该远程操作来维护最终的一致性。

OT有20多年的研究历史，最早是在[27]中首次介绍并定义了协同编辑系统，同时提出了一个解决并发控制的算法(论文中并没有显式提出OT)，该算法的创新之处是提出了无锁的机制来解决操作并发的问题。接着OT算法在[28][29][30][31]中得到发展。[28]中第一次提到了OT算法，定义了操作转换中所需要的基本属性，解决了协同编辑中的Undo问题。[29]提出了一致性模型和一个通用的操作转换算法，用来实现用户操作意图保留，一致性实现和因果关系保留。[30]是将OT算法应用到Jupiter系统中，用来优化并发控制和冲突解决的算法。[31] 整理了OT技术发展过程中出现的问题和解决的方案，以及面临的挑战，然后还提出了一个优化的OT控制算法。

相比较OT来说，AST算法的发展历史比较短，最早是在[13]论文中提出的， 通过地址空间回溯算法，控制算法和执行算法三个重要部分来实现最终结果的一致性。在[32]中将AST的思想应用到了分布式的镜像站点中，提出了基于xml格式数据的因果一致性维护和事务型操作一致性维护的解决方案。在[33]中将AST算法的思想应用到了移动app使用场景中提出部分复制的解决方案，使得在无网络的评论系统中，用户和系统也能够进行快速交互。当系统能够连上网络的时候，再将本地的评论内容和远程服务器进行同步。总体来说，AST技术在响应速度和执行效率上比OT更高。



### 1.2.2 热点数据动态迁移问题的研究现状

在私有云平台下，缓存为数据库充当着非常重要的减压作用。但是同样的，缓存的存在也会给系统的运行造成问题。例如在运行过程中，私有云下的服务器节点数量由于节点宕机或者控制器自动减少挂载的计算节点，使得之前存储在缓存中的数据都可能变成无效数据，这将给数据库在瞬间带来较大的访问压力，从而造成数据服务整体性能下降的问题，即热点数据迁移问题。

在这里热点数据迁移问题指的是：由于业务的发展和项目新增的需求，一些系统需要在上线过程中进行扩容，以便加强系统的负载能力。此时由于系统之前运行了一段时间，因此每个应用服务器节点上的缓存数据皆为稳定的热点数据。当添加新的服务器节点时，所有的用户请求将会通过控制器节点进行重新分配，因此短时间内，重新分配的请求由于无法定位到原服务器，缓存层失效，因此大量的请求将会直接负载到持久层，严重的情况下，如果访问压力大大超过持久层的承载能力，则会影响系统的整体运行。这种情况产生的原因就是热点数据在运行过程中发生了存储位置的迁移。

一致性哈希算法[70]通过添加服务器虚拟层的思想来减少缓存数据迁移带来的影响。一致性哈希算法首先构造一个一致性哈希环（环上有2^32个节点，每个节点的编号为0-2^32-1），并对系统中已有的服务器节点编号(0~N-1)进行哈希取余，分布到哈希环上。当用户请求数据到达的时候，控制器对请求数据根据路由算法进行计算，并按照顺时针的方向查找离该数值最近的服务器节点，从该服务器上获取数据返回给用户。当系统下的节点动态扩充的时候，新加入的节点也同样根据编号进行键值取余，放到哈希环对应的位置上去。通过这种方法，尽可能把缓存数据的影响范围缩小到附近的缓存服务器节点，而不会影响整个的缓存数据服务效率。由此可见，一致性哈希算法是将每个服务器节点虚拟化为一组节点，先找到虚拟节点之后，再找到对应的物理节点，从而减小由于架构变动而造成的对热点数据的存储影响。但是一致性哈希算法针对的是基于求余数哈希的路由算法的请求分发系统，对于类似残疾人无障碍自理管理平台系统根据地区用户固定请求分发的情况来说，是不符合实际需求的。

### 1.2.3 缓存存储数据结构和缓存替换算法的研究现状

缓存作为一种加快数据访问效率和减轻数据库访问压力的中间件，有多种数据存储结构。目前较为流行的开源框架memcached，redis等内存数据库在存储结构方面都选择了以键值对为主要形式的哈希表结构。

缓存替换算法方面。为了保证数据的高效存取速度，缓存通常是放在内存中的，这就使得缓存的容量大小是受限的。缓存替换算法是当缓存空间已满并且需要存储新的数据时，根据一定的策略从缓存中选择过时的数据进行剔除，从而及时更新缓存中存储的数据，提高缓存命中率。

缓存替换算法所解决的问题，是如何从缓存中选择删除过时的或者已经失效的数据，让尽可能多的热点数据放在缓存中。首先应用的替换算法是从操作系统中所延伸出来的经典算法，例如LRU(Least Recently Used)，通过维护一个最近使用数据历史队列来进行更新替换，LFU(Least Frequently Used)，替换缓存中使用频率最少的数据，Size，替换缓存中占用空间最大的数据，以及随机替换算法等等。这些经典算法各有优缺点，但是因为实现简单并且性能表现较好，因此也一直在系统中使用。

除此之外，还有一类替换算法，例如[60]，根据系统用户访问的日志来进行海量数据分析，提取出对缓存产生影响的因素，如数据大小，访问频率等，对不同的因素赋予不同的权重值，然后根据公式为每一个缓存中的数据计算出一个grade值，从而把grade值最小的那个数据对象给替换掉；例如设计出一个缓存替换策略[61]，由随机替换算法延伸出来的随机替换策略，首次替换的时候，先选取N个样本，然后根据一定的替换算法(可以是LRU，LFU等)来替换数据对象，并从这N个样本中保留M个，其余的删除。接着下一次需要进行替换的时候，从缓存中再次随机抽取N-M个数据样本，和之前保留的M个样本一起，根据同样的替换算法剔除数据，每次重复以上步骤。在该替换策略中，核心思想是根据每次的随机替换迭代，最终会在样本中保留的N个数据样本，为整个缓存数据中最为需要替换的前N个数据。

本文的缓存数据一致性维护技术中的缓存数据结构以哈希表为主，缓存替换算法选用的是LRU。

## 论文的研究内容

本文的工作重点是提出了基于AST的缓存数据一致性维护技术，并实现了该技术的原型系统。通过在原型系统上进行不同维度的对比试验，验证了该技术的有效性。最后根据残疾人无障碍自理管理平台项目的需求背景，将缓存数据一致性维护技术在在私有云平台下进行了设计和应用。本文的研究内容主要有以下四点：

1. 提出了基于私有云平台的系统架构设计。通过对云计算的部署方式以及云计算解决方案进行调研和了解之后，在私有云平台下对系统架构进行了设计。

2. 在该架构设计基础上，提出了基于AST的缓存数据一致性维护技术。该技术以哈希表为数据存储结构，包含了基于缓存的地址空间回溯算法，基于缓存的控制算法以及缓存数据操作执行算法这三个重要组成部分。

3. 实现了具有缓存数据一致性维护技术的原型系统，在该原系统上做了一系列的对比实验，验证了算法的有效性。

4. 提出了面向残疾人无障碍自理管理平台的缓存架构设计。根据残疾人无障碍自理管理平台的项目需求背景，基于对原有系统中存在的问题和架构设计的局限性的分析，将缓存数据一致性维护技术应用到该项目中，并对该系统下的缓存部分进行了设计和实现。

## 论文的组织与结构

本文总共为六章，每一章节的概要内容如下：

第一章介绍了私有云的研究背景，并引出了在架构动态伸缩而导致缓存数据迁移的情况下引发的问题，同时介绍了一致性数据维护，热点数据迁移问题等方面的研究现状，最后简述了本文的主要研究工作。

第二章介绍了基于私有云平台下的系统设计，包括云计算的服务模式和相关技术的介绍以及基于私有云的系统架构设计，是缓存数据一致性维护技术的系统架构讨论基础。

第三章介绍了基于AST的缓存数据一致性维护技术的原理，主要介绍了地址空间转换技术的算法原理，缓存的结构以及缓存数据一致性维护技术的算法原理。

第四章详细描述了基于AST的缓存数据一致性维护技术的具体实现过程，并用伪代码的形式来阐述了其中的算法逻辑流程，最后场景模拟了用户使用该系统时的内部具体实现。

第五章实现了具有缓存数据一致性维护技术的原型系统UserManage。在该原型系统上，通过不同维度的对比实验，说明了该算法对于系统性能的影响以及在节点失效时对于缓存节点中数据存储的影响。接着介绍了复旦大学和上海市残联合作的残疾人无障碍自理管理平台，分析了该平台的局限性和存在的问题，最后将缓存数据一致性维护技术应用到该平台下，针对残疾人系统中原本存在的问题和局限性，对缓存部分进行了设计。

第六章总结了前五章中的主要内容，在此基础上提出了未来工作中继续深入的研究方向。

# 基于私有云平台的系统设计



## 云计算的服务模式与技术介绍



### 云计算的部署方式和解决方案

云计算[34][35][36]根据对数据的安全性控制模式可以分为共有云，混合云和私有云三大类。公有云对用户的数据开放程度最大，因此通常把对数据保密性和安全性要求较低的系统应用部署在公有云上，例如国有的大型的基础设施建设：国家电网，国有电信等企业，常常将公有云作为系统的部署解决方案。就如同国家供居民用水用电一样，所有的基础设施都是国家统一规划建造的，居民只需要根据自己的需求缴费使用水电即可，而公有云，也是将计算资源按照一定的收费标准，提供给需要使用资源的企业或者个人。相对地，私有云则是专门应用于企业或者组织内部的，通过架设内部网络以及防火墙等网络安全手段来保证私有云内部数据的安全性和可靠性。除此之外，私有云还具有为用户的需求而专属定制化的特点，因此私有云中可以根据企业组织实际的业务场景需求来进行专属的设计，例如企业中的日常员工打卡管理，业务运转，系统交易数据统计，营业额统计等等，这些内部数据的信息化管理就比较适合部署在私有云平台，因为企业或者机构内部的数据需要有安全性保证，根据不同安全性级别的数据需要有不同层次的权限才能够查看和修改，并且，每个企业或者机构组织都有自己专门的运作方式，因此私有云的定制化特点就可以满足这些需求。因为私有云是专门为用户来定制的云计算服务的，对于数据的访问控制以及安全性上要求更为严格，因此成本上也比公有云更高。混合云从字面上可以理解为私有云和共有云的混合体，从应用上面，可以考虑需求方面的多样性。例如在有的需求中，需要在私有云平台下存储数据保密性和安全性要求较高的数据，同时也需要利用公有云强大而低廉的计算能力来进行海量计算，比方说公司的年度交易额，全年的业绩增长百分比，更深层次的还有基于数据挖掘，机器学习等海量数据分析计算等等，这些应用都将会需要公有云所可以提供的强大的计算资源，因此在这种情况下，常常在私有云中接入部分的公有云，协作来完成整体的功能需求。

云计算[37][38][39][40]除了以数据控制模式来进行的划分，还有根据所提供的云服务的不同层次来进行的划分：例如将服务器，网络设备，存储设备等硬件设备根据设备不同的负载能力进行整合与抽象向用户提供服务器资源的Iaas(基础设施即服务)；基于Iaas之上，以应用开发平台以及开发环境平台抽象为服务的Paas(平台即服务)；还有处于云计算解决方案中的最上层，将所有的软件功能进行了封装，通过互联网向用户提供服务的Saas(软件即服务)。

不管是以何种维度对云计算进行划分，云计算提供的都是一种计算模型，它要求在架构设计上的有一定的可伸缩性，高可用性，稳定性；在数据服务上，它需要对于资源进行抽象化，对外按需提供可自动化监控和管理的虚拟化的服务。

### 主流的云计算开源框架介绍

随着对云计算的认识和研究的不断深入，为了方便组织，企业，个人的云计算应用和开发，目前也有很多开源的云计算框架，在某种程度上提供了统一的设计模式。

目前主流的云计算开源项目有Openstack[41][42][43][44][45][46][47][71]， Eucalyptus[48][51] [52][53]，Cloud stack[54][55]等等，用户可以根据自己所拥有的硬件资源和运维能力来搭建不同大小的云计算服务平台。这些开源框架类似于一种专门来管理具有云特点的操作系统，它们对各种硬件设施资源进行了封装和抽象，并提供了各种管理数据存储，验证服务等等的解决方案，方便了人们的个人研究需要或者是企业组织的自行搭建等需求。下面依次来简要介绍一下主要的云计算平台框架以及各自的特点。

Openstack主要是基于Iaas层次的公有云解决方案(但是也可以用来搭建私有云)，它的内部功能组件结构是基于分布式的，主要包含存储服务组件 (Swift)，计算服务组件(Nova)，身份验证服务组件(keystone)和镜像服务组件(Glance)]五种。组件与组件之间是通过消息队列和提供的API来进行通讯的。结构设计上，单个组件的功能集中，组件和组件之间耦合性低，因此还可以根据实际的应用需求部署单个组件来实现特定的应用。但是Openstack安装配置细节复杂，要求对Openstack内部构造有较为深入的了解基础，目前一些开源社区也推出了帮助安装的组件例如fuel等来帮助初学者快速入门。

Eucalyptus是一个专门用来部署私有云的开源框架，比起Openstack，它更专注于私有云以及混合云领域(主要是私有云)。Eucalyptus是一个支持AWS[49]的云平台，因此对于原本支持AWS的云计算平台可以无缝地将应用和数据迁移到Eucalyptus中[50]，和一般的云计算框架一样，Eucalyptus也可以支持应用的测试环境或者是各种web服务。从组件上来看，Eucalyptus的组件功比Openstack更为简单，它包含各类控制器和负责存储的组件。除此之外，Eucalyptus的安装与配置比起Openstack来说要稍微简单。官网上还为初学者提供了fast(一键安装)版本。Eucalyptus的安装部署对于硬盘要求比较高，需要可用空间要在200G以上(fast版本是将eucalyptus的所有组件全部都安装在一台硬件设备上)；如果希望每个组件安装在不同的硬件中，例如希望搭建高可用的组件架构(多个云控制器，节点控制器)，则单个节点控制器组件也需要在100G以上的容量，因此Eucalyptus对于硬件有比较高的要求。由此也可以反映出Eucalyptus所解决的云计算需求是基于企业级的私有云管理平台。

Cloudstack和Openstack类似，它既可以用来提供公有云服务，也可以提供私有云或者混合云服务[56]。Cloudstack在架构设计上和openstack明显的不同之处就是它采用的是集中式的架构设计，所有的组件部署都可以在一台服务器上完成。但是集中式也会带来耦合性较高所引发的一系列问题。



## 基于私有云平台的系统架构设计



### 基于私有云平台的整体架构设计方案

2.1中介绍了云计算的服务模式与相关技术，在此背景下介绍基于私有云的系统架构设计，并且缓存数据一致性维护技术也是在该架构设计基础上进行讨论的。

在该架构下，各个服务器节点的设计是按功能分层的，包含五类服务器：反向代理服务器，用来将用户的请求转发给下面挂载的云控制器节点；云控制器节点，根据路由算法将请求分发到各个节点控制器；节点控制器，下面挂载了多个应用服务器；应用服务器，包含了缓存节点，处理对持久层的数据请求以及处理相应的业务逻辑；持久层则负责整个系统的数据管理。整个架构设计如图2-1所示。

在该私有云平台下，用户只需要使用手机，iPad或者是笔记本电脑等设备连接上网络就可以直接访问云平台提供的服务。反向代理服务器将所有的用户请求根据用户的就近原则分发给其下挂载的控制器节点，因此反向代理很容易成为整个系统的性能瓶颈，这里可以考虑使用master-slave模式或者架设集群模式来保证私有云服务的高可用，从而解决反向代理单点宕机的问题。云控制节点可以分布在不同的网段中，每个云控制器节点负责一块局域网下的节点控制器。节点控制器下面挂载的多个应用服务器充当了整个系统中的应用层，可以对每台应用服务器的运行状态进行监控，从而通过当前流量的大小来自动配置对应数目的应用服务器节点，比如对于一般的网站来说，夜间的用户流量会比白天的流量小，因此在晚上可以考虑运行较少的应用服务器来提供服务，从而节省资源；当某段运行时间内，当前请求量的流量突增，则可以动态添加闲置的应用服务器来进行分流减压。每个应用服务器中都对应了一个缓存节点，系统对外提供的数据大多数

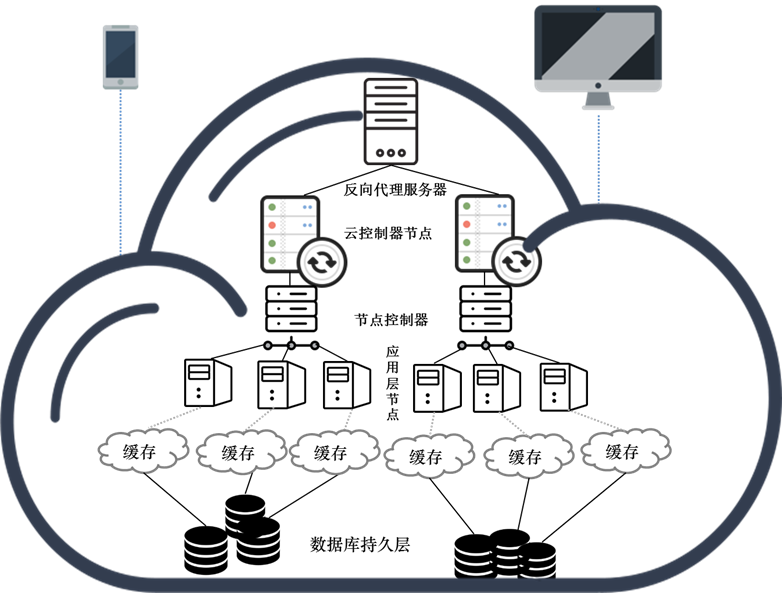


图 2 ‑1

都来自于缓存，因此缓存优化了系统的整体性能。在实际使用中，缓存层可以使用内存数据库例如redis，memcached，mangoDB等开源数据库来做缓存层，其性能和安全问题上将更有保障性。

### 单个站点的应用架构设计方案

对于单个站点来说，即单个应用服务器内的部署，在整体逻辑上可以简单地分为四个层次，分别为请求(action)层，业务逻辑(service)层，数据库访问(DAO)层和实体(model)层。整个层次划分是按照spring框架来做参照基础的。

model层是根据项目需求背景抽象出来的实体类，封装了相关的属性和方法，通常该实体类会和关系型数据库的每张表的每个属性一一对应。在有些开源框架中将实体类和数据库表建立了映射关系，便于实体类和持久层存储之间的相互装换。Model层是其他三个层的基础类，所有的业务处理和操作都是基于实体类。

action层用来截获用户的请求，获取请求参数，并从请求参数中解析出其中的属性值(有时还需要将解析出来的值封装成实体类)，并将解析出来的值传递给service层。service层以action层传递过来的实体对象为参数，调用所需要的DAO层的查询方法，将结果集进行一定的逻辑处理，并将获得的最终结果以实体类的方式返回给action层，action层需要将service层的返回结果重新封装成与前端规定格式的参数(例如json数据格式)进行返回。

DAO层负责和数据库直接进行交互，封装了对于数据库访问的所有的方法，除此之外，为了数据库资源的高效利用，还可以使用数据库连接池和数据库会话工厂等一系列设计模式的工具，从而提高应用服务器的效率。

除了这四个基本层次之外，还有缓存层。缓存层是放在逻辑应用层(service)和数据库访问层(DAO)之间。在数据库访问量大的时候是数据访问的缓冲层，起到对数据库减压的作用，在访问量小的时候，可以进行数据预热同步。这样使得数据库服务的资源可以得到重复的利用。单个应用服务器中的层次结构以及调用逻辑如图2-2所示。

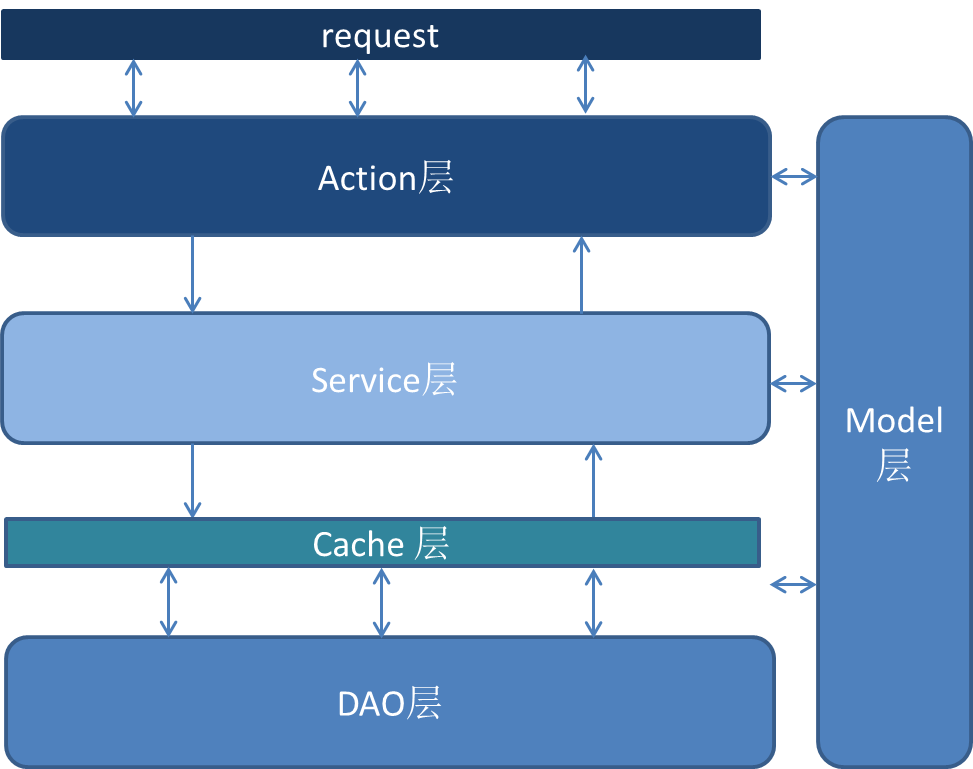


图 2 ‑2

# 基于AST的缓存数据一致性维护技术



## AST算法的实现原理



### AST算法中的基本概念和定义

AST(Address Space Transformation)全称地址空间转换技术[13][76][72]，是用来解决组编辑中多个用户在计算机终端同时编辑共享文档而造成的不一致的问题。该算法主要是用来解决不同操作之间的冲突，使得在不同的站点上，只要执行了相同的操作序列，就能够得到一致的副本结果。

AST的系统架构是基于所定义的1-N个站点，每个数字编号为每个站点的ID号。在协同编辑中，每个站点都会产生操作序列，在本站点执行之后发送到其他远程站点。为了保证操作产生结果的实时性和系统的高响应性，AST算法规定本地操作需要立即执行，而远程操作则需要判断操作的时间戳是否符合当前站点的执行条件才能够执行，这样做是为了维护操作间因果关系的一致性。

AST算法首先是定义了文档编辑的最小单元和基于文档编辑的两种操作。基于文档上的每个字符都是不可分割的假设，将字符的操作抽象为两类，一类是Insert操作，还有一类是Delete操作。Insert操作包含两个参数，一个是P，代表从文档左边开始起的第P个位置的字符，还有一个就是C，即插入的内容；Delete操作只包含一个参数P，代表的是删除从文档左边开始起的第P个位置的有效字符。其余任何的对文档进行的复杂的操作都可以是这两类基本操作的组合。

除此之外，AST算法还定义了每个操作之间的关系。

定义1.（操作间的关系）操作的关系主要包含两种，一种是并发操作，用符号”||”来表示，还有一类是因果关系的操作，用符号→来表示。

定义2. （因果操作关系的定义）因果关系的操作定义如下：假设有两个操作OP1和OP2，如果OP1→OP2，则包含以下三种情况：①在同一个站点上，OP1执行的时间戳产生在OP2执行时间戳之前 ②在不同的站点上，OP1执行的时间戳产生在OP2执行时间戳之前③存在一个操作OP，OP1→OP并且OP→OP2，则可以得出OP1→OP2。

定义 3.（并发操作的定义）如果两个操作不是因果关系的操作，则为并发操作。

因此操作之间的关系，以及不同操作之间的执行顺序是由它所附着的时间戳属性所决定的。时间戳需要能够统一全局的精准的时间，因此不能使用时钟时间，因为时钟时间存在一定的误差。因此在AST中引入的是逻辑时间，即向量时间戳[5]。时间戳每一维代表了每个站点，每个维度的值代表了当前站点上已经执行的操作个数，这样就可以为整个系统上每个站点的操作定义全局的，统一的时间。除此之外，为了实现每个站点上副本的最终一致性，还需要统一不同关系操作之间的执行先后顺序，如果是因果关系的操作，则可以按照向量来比较大小，至于并发操作，就需要定义一个统一的比较时间戳大小的函数。这里AST算法采用的是Torder排序，如果是并发操作，则首先对操作的每个维度的值求和，根据和的值来比较大小，如果两个和值相同，则按照站点号来进行比较，默认站点号越小的时间戳就越小。

介绍了以上基本概念之后，接下来的三小节中，将会重点介绍一下AST算法中重要的三个部分，分别是地址空间回溯算法(retrace)，控制算法(control)以及执行操作算法(range-scan)。

### 地址空间回溯算法

AST算法的主要思想，就是在保证协同编辑系统中能够对并发操作快速响应的情况下，保证副本结果的最终一致性。对于本地操作将会立即执行，而远程操作则需要进行地址空间回溯，将当前的站点的数据视图切换到该远程操作产生时的时间戳视图下，执行该操作，然后更新站点的时间戳，再次调用地址空间回溯算法进行回溯操作，从而把远程操作对该站点的数据影响结果包含进来。因此地址空间回溯算法至关重要，它决定了操作执行的上下文。地址空间回溯的算法过程是对文档中的每个字符节点进行顺序遍历，在给定时间戳下对每个字符进行有效性标记，这样就可以屏蔽一些无效的数据，从而将站点视图切换到远程操作产生的时间戳下。

地址空间回溯算法中的基本数据结构如图3-1所示。即文档中的每个字符为一个节点，节点下面都包含了一系列对该字符进行的操作，以及每个操作的所附着的时间戳。当一个远程操作到达本地站点的时候，首先判断操作是否符合当前的执行条件。如果符合，则需要将文档状态切换到远程操作执行的状态下，即标记每个字符节点的有效性属性。该算法在遍历文档节点的过程中，首先对每个字符节点设置为无效，接着判断远程操作的时间戳和当前字符节点的时间戳之间的大小，如果远程操作的时间戳大于或者等于当前字符节点下面的操作时间戳，并且操作为插入操作，即在该远程操作执行之前已经存在该字符节点，则设置为有效；如果远程操作的时间戳大于或者等于当前字符下面的操作时间戳，且操作为删除操作，即在该远程操作执行之前该字符已经被删除，则设置为无效，。在如图3-1所示的结构体中，b节点在时间戳(1,2,1)下面就是无效的，因此此时b节点已经由操作Delete被删除了。

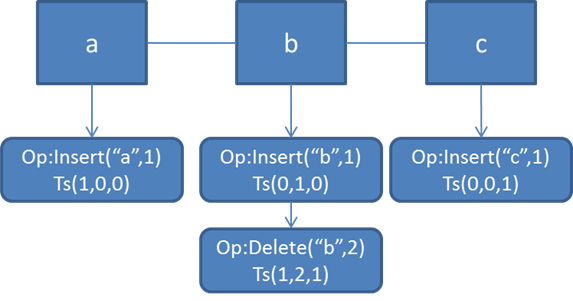


图 3 ‑1

### 控制算法

当远程操作到达本地站点，并且满足执行条件之后，则可以使用控制算法。执行条件是否满足是根据远程操作的时间戳与当前站点的时间戳来进行判断的。远程操作可以在本地操作的执行条件如下：

定义4（远程操作的执行条件）执行条件需要满足两个条件，第一，远程操作所在站点的所对应的维度的值要等于当前站点的时间戳所对应维度的值加一，第二，当前站点其他维度的值都要大于或者等于该远程操作的时间戳所对应维度的值[32]。

远程操作执行需要满足的两个条件主要目的是为了远程操作在本地站点中的因果一致性。在满足了上述的执行条件之后，将会进入控制算法。在控制算法中，首先要调用回溯算法，目的是将当前文档的操作上下文切换到远程操作产生的时候，执行该远程操作，接着将该操作添加到它所作用的字符上，然后更新站点时间戳信息，即对该远程站点所对应的维度的值自增1，最后再调用回溯算法，将远程操作所做的影响包含到当前站点状态中来。控制算法主要是描述了远程操作如何在本地站点中执行的算法流程。

### 操作执行算法

执行操作算法讨论的是在控制算法中，当地址空间回溯完成之后，需要执行远程操作。按照之前的操作类型的定义，如果是Delete操作，则在地址空间回溯完成之后，就可以知道删除的确切位置，并且根据参数找到对应位置的字符把它删除即可，而对于Insert操作，仅仅是地址空间回溯算法是不够的。因为在两个有效节点之间，可能会包含很多个无效的节点，因此需要知道在两个有效节点间具体插入到多个无效节点间的哪个位置。这就需要用到range-scan算法。

range-scan算法执行过程如下：遍历两个有效节点之间包含的无效节点，使用插入排序算法将insert操作附着到正确位置的字符节点下。插入排序算法用来比较节点间时间戳的大小就是依据之前提到过的Torder函数。即当插入操作的时间戳和当前的无效节点的时间戳是因果关系时，则按照一般的向量来进行比较大小，如果是并发操作，则按照Torder来进行比较大小，插入到对应的位置上去。



## 缓存结构的设计介绍

根据AST算法思想理论，这里首先需要定义一下缓存的数据结构以及存储模式。

定义5(缓存数据结构的定义)CacheMap<K,V>，K代表键值，V代表数据存储项。

定义6（缓存数据存取）应用层生成键值，在缓存哈希表中查找对应的数据项。如果缓存命中，则直接返回数据项；如果缓存不命中，则以键值在数据库中重新查询数据，然后把数据项和对应的键值存入缓存中，并将查询数据进行返回。

系统中缓存的作用主要是为了加快服务器查询数据库的效率从而减轻对应用层对持久层的访问压力，因此会把经常用到的数据查询语句存放在缓存中。为了提高缓存中数据的存取效率，缓存中的基础存储结构为哈希表，键值要求可以和查询数据结果集有一对一的定义，并且还可以利用键值来进行数据库查询，在本文中定义键值是包含查询数据库的方法，参数列表和参数值等信息的字符串。

除了缓存的存储数据结构，为了能够更好地将地址空间转换思想运用到缓存应用背景下，还需要定义以下几个基本实体类：Timestamp(时间戳类)，Operation(操作类)，CacheObject(缓存数据类)，这些实体类封装了AST算法在缓存模块中实现数据一致性的主要功能和基本属性。现在来主要介绍一下这三个类的作用和应用场景。

定义7（Timestamp 时间戳类），Timestamp包含两个基本属性：一. 所在的站点ID，二. 时间戳向量SV。

Timestamp类中站点ID用来判断一个操作是远程操作还是本地操作，时间戳向量在基于缓存的地址空间回溯操作中，判断远程操作是否因果就绪，以及更新本地站点的时间戳等用途。

定义8（Operation 操作类），操作类中基本包含三个属性：一.时间戳类，二.操作类型，三.操作内容。

Operation类中的时间戳类含义见定义7。对于本地站点产生的操作，在执行之后将会更新站点时间戳，然后站点时间戳将会附着到本地操作对象中，然后发送到其他站点。操作类型为两种，一种是PUT操作，还有一种是REMOVE的操作，在回溯操作中不仅要判断数据项下附着的操作时间戳，还要判断操作的类型来决定数据项在当前时间戳下是否有效。操作内容里面包含了该操作的执行内容。操作内容需要包含所调用的方法的全名(包含包名以及类名，这样才能够在其他站点执行的时候找到对应的方法)，参数类型列表，类型列表下对应的每一个参数值等信息。操作内容的执行利用了Java语言中的反射机制，可以在JVM运行的时候动态获得执行对象和执行对象下面的方法。这样当一个远程操作需要在本地站点进行执行的时候，可以从操作内容中解析出需要调用的方法并传入的对应的参数，从而获得对应的数据。本文所述的系统原型中，这些信息是由字符串来作为存储的，并且这些字符串在缓存中作为键值索引来映射CacheObject（即缓存对象类）。

定义9（CacheObject 缓存对象类），包含三种重要的属性，一.value值，二.所附着的操作列表，三.isEffect，该缓存对象是否有效

CacheObject（缓存对象类）是封装了存储在缓存哈希表中的对象。value值是由键值解析所得出的对象和方法名利用反射机制动态调用执行的数据库查询返回结果；操作列表是所有在该缓存对象上执行过的操作，这些操作列表是按照时间戳来进行排序的，算法伪代码如Function1所示。算法描述如下：如果操作列表为空，则直接把操作加入到队列中即可，对应于Function1的1-2 行；反之，则需要从操作队列的尾端开始遍历，获取每个操作o，就把该操作和输入操作op

Function1：addToOps(Operation op，CacheObject) ,insert the Operation op into Operation List of CacheObject

1：If CacheObject.operationList is empty

2： CacheObject.operationList.add(op);

3：Else

4：For (int i=CacheObject.OperationList.size()-1;i>=0;i--){

5： Operation o is from Operation List.get(i);

6： If(o||op && Torder(o,op)<=0)

7： Insert op at the Position of i+1 in CacheObject.OperationList;

8： Break;

9： End If

10： If(o→op &&o<=op)

11： Insert op at the Position of i+1 in CacheObject.OperationList;

12： Break;

13： End If

14： i--;

15：End For

16：End Else

进行比较，如果是并发关系，则采用Torder排序算法来进行排序，如果是因果关系的操作，则采用一般的向量比较法来进行排序，如果输入操作op比该操作o的时间戳要大，则直接在该操作o后面的位置进行插入并跳出循环体，否则，继续遍历直到遍历结束。执行逻辑对应于Function1的4-16。isEffect属性主要是在地址空间回溯算法的时候使用，retrace算法将会对缓存中的所有数据项在给定的时间戳下进行有效性标记。CacheObject（缓存对象类）主要是对AST算法中的实体类的封装以及针对该实体类的一些操作实现。接下来要讨论的是4.2中操作类型的生成和本地操作的产生，以及实现如何利用消息队列机制将已经执行的本地操作广播到其他站点。

## 基于AST的缓存数据一致性维护技术原理



### 基于缓存的地址空间回溯算法

不同于基本的AST算法[13]在每个站点的线性的文档副本上进行一致性维护，也不同于[32]在XML树形结构的镜像站点的一致性维护，缓存中的数据结构以哈希表为存储结构体，因此在数据查找和数据获取上效率很高，几乎为常数级别的复杂度。接下来的章节中，将会详细讨论基于哈希表的一致性维护算法的技术原理以及算法实现。

之前在3.2.1节中提到了在系统中基本的缓存的存储数据结构，在这节中，为了让哈希表结构支持AST算法，需要在每个缓存数据节点下添加对该数据项所作用过的操作列表，以此来支持地址空间回溯。缓存中的数据结构如图3-2所示。这里TS1，TS2等是操作产生的时间戳，PUT和REMOVE代表操作类型。

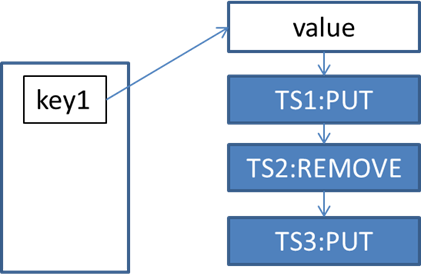


图 3 ‑2

和对文档字符节点的操作类似，在缓存中，对于数据的基本操作可以分为两种，一种为PUT操作，输入参数为key和value值，作用是将数据库中获得的数据放入到缓存中，类似3.1.1节中提到的Insert操作；还有一种为remove操作，输入参数为key值，将缓存中该键值的数据从缓存中移除，对应于3.11节中提到的Delete操作。其他任何的复杂操作可以由这两种基本操作分解得到。

和AST算法对线性结构的文档操作一样，对于缓存数据也需要进行有效性和无效性的标记，从而方便切换远程操作的执行上下文。基于哈希表数据结构的retrace算法如Function2所示。该算法会遍历缓存中的每一个数据项，将缓存空间切换到SV时间戳下的数据视图，从而使得远程操作能够在当前站点中执行。

Function2 Retrace (CacheMap,SV), retrace the address space of CacheMap with timestamp SV

1：for all the data item DIi in the cache

2：do

3：set DIi.mark is ineffective

4：For(var i = length-1;i>=0;i--)

5： var opDIi.operationList[i]

6： If(op.timestamp || SV) //concurrent operations

7： If(Torder(op.timestamp) <= Torder(SV) && op.Type is PUT )

8： DIi.mark←effective

9： break

10： End If

11： End If

10： Else //op→SV

11： If(op.timestamp<=SV && op.Type is PUT) //

12： DIi.mark effective

13： break

14： End If

15： End Else

15：End For

16：End interation

该算法执行过程如下，首先遍历缓存哈希表中的每个数据项，并标记为无效，如Function2所示1-3，然后获取该数据项下面附着的操作列表，操作列表是根据时间戳向量排序算法已经排序好的。因此只需从操作列表的最后一个操作对象开始进行从后往前遍历，如算法所示4-5。如果操作列表中的操作对象的时间戳和当前回溯空间的时间戳SV为并发操作，则用Torder来进行比较大小，如果操作对象的时间戳比SV要小，并且为操作类型为PUT，则设置为有效，跳出循环体，如算法所示7-9；如果是因果关系的操作，则直接比较两个向量时间戳的大小，并且如果操作类型为PUT，同样也设置为有效，并退出循环，如算法所示10-15行。

### 基于缓存的控制算法

缓存中的控制算法和AST控制算法类似，控制算法是针对远程操作的，因此在执行控制算法之前，执行的远程操作必须符合当前站点的执行条件。

基于缓存的控制算法的伪代码表示如Function3所示。首先获得远程操作O的时间戳SVo，然后调用地址空间回溯算法将缓存空间切换到SVo时间戳下的缓存数据视图中。执行算法O，如Function3所示1-2行，然后创建一个的操作节点，将该节点根据Torder时间戳比较算法利用插入排序插入到缓存数据项下的

Function3 Control(CacheMap，O), the operation O executes on the CacheMap of timestamp , SVS and its timestamp is SVO

1：Retrace(CacheMap，SVO) //retrace the view of the cacheMap to the timestamp of SVO

2：executeOp(CacheMap,O)

3：create a new operation node and attach the node to the cache item

4：SVSSVS+1 //update the timestamp of the current site

5：Retrace(CacheMap, SVS);

操作节点中。然后更新当前站点的时间戳，即远程操作所在站点的维度值加一，然后以更新的站点时间戳进行地址空间回溯，将已经执行过的远程操作对缓存数据的更新包含到当前的站点中，该过程如Function3 4-5行所示。

### 基于缓存的操作执行算法

当一个远程操作满足执行条件时，就可以在进行操作执行算法。远程操作的执行逻辑如Function4所示。执行操作的算法首先从操作O中获取操作内容，操作内容包含如何根据已有的信息快速地从缓存中获取数据，并且要求和缓存中的键值有一对一的映射关系，即可以根据操作内容直接得到键值。

第2行根据操作内容得到的判断在缓存中是否已经包含该键值，如果缓存中已经存在该键值，则直接从获取该键值对应的数据，否则，就创建一个新的数据对象。如果缓存中包含该键值，则从缓存中获取该存储对象，并判断该value值是否有效(在地址空间回溯算法中将会对缓存中的每个数据项设置有效性属性)，如果在当前时间戳下该数据是无效的，则需要执行操作O，即从数据库中重新获取一遍数据，如Function4的4-6行。如果缓存中不存在该键值，则同样也需要执行操作O，如Function4中的8-10行。最后把对应的键值以及获取到的数据执行结果对放入到缓存中去，对应Function4中的11行。

Function4 executeOp(CacheMap,O), the operation O will be executed on the CacheMap

1：get the operation contain as key

2：If CacheMap contains key

3： Then get value named as Value

4： If Value is not effect

5： Then

6： execute O and get the execution result

7：Else

8： create a new Value Object

9： execute O and get the execution result

10：End Else

11：CacheMap.put(key,Value);

缓存一致性算法和AST算法不一样的地方是，AST算法是基于线性文档的，因此需要通过地址空间回溯算法找到两个有效的字符的位置，再通过range-scan算法来确定两个有效字符之间（可能包含多个无效字符）的具体位置P（具体过程详见3.1.4）。但是缓存一致性算法的数据结构是哈希表，因此只需要通过键值就可以直接找到对应的数据，因此在复杂度上比原本的AST效率更高。

第三章主要介绍了AST算法实现的基本原理，缓存的数据结构和数据存储，以及缓存数据一致性维护技术在缓存中的实现原理，在接下来的第四章节中，将会着重描述缓存数据一致性维护技术在缓存中的具体实现过程。

# 基于AST的缓存数据一致性维护技术的实现



## 基于AST的缓存操作生成和广播

在这小节中，主要讲的是操作的产生和操作的发送实现。在3.3.1小节中，本文提到了在缓存中包含两种类型的操作，一个是PUT，还有一个是REMOVE操作，在这节主要描述本地操作产生和操作广播的实现以及接收远程操作的实现。为了保证AST的一致性维护的线程运行不会影响到缓存对用户端提供的数据服务请求性能应答，因此一致性维护和service层缓存请求应答是放在不同的线程异步进行的，这里用springAOP机制实现了这个思想。

spring AOP的机制，即原型为设计模式中的代理模式，它是将执行方法之前，之中或者之后的逻辑用动态的生成类包装起来，进行统一的处理。在缓存系统中，可以利用springAOP机制来统一对于DAO层的数据库查询方法来进行拦截，然后在方法中对缓存进行处理，如果缓存命中就直接返回数据，如果不命中则重新从数据库中获取数据。除此之外，为了保证DAO层数据的返回效率，同时产生的本地操作将生成没有时间戳的操作对象，放入到本地操作执行队列中，在守护线程中依次执行。

具体做法如下：由于缓存中存储的数据皆为查询数据，因此需要定义统一的数据库查询命名方法，这里为统一的前缀名，例如find\*，select\*，get\*等，这样springAOP的拦截器可以对这些方法进行统一的拦截，然后在代理生成的Around方法中，对传入参数进行判断，是远程操作还是本地操作调用调用DAO层方法，如果是本地操作，则生成本地PUT操作对象，并把PUT操作对象添加到本地执行队列中；如果是远程操作，则调用相应的执行函数进行执行。以上逻辑的实现的伪代码如Function5所示。

Function5 Around(ProceedingJoinPoint,CacheMap)

ProceedingJoinPoint is the SpringAOP object which provides the intercepted Method object

CacheMap is the reference of the cache.

This function will return the Object that from CacheMap if cache hit or database if cache miss.

/\*\*params is an array that generates from ProceedingJoinPoint \*\*/

1：If( params contains op of Operation class type)

/\*\*get contain of op and resolve it to generate key that can be searched in the CacheMap\*\*/

2： generateRemoteOperationKey;

3： executeRemotePut(params[0]);

4：Else

5： generateLocalOperationKey;

6： If (CacheMap.contains(key))

7： cacheObjectCacheMap.get(key);

8： If(!cacheObject.isEffect)

9： cacheObject.value ProceedingJoinPoint.proceed();

10： End If

11： Else

12： cacheObject.value ProceedingJoinPoint.proceed();

13：End Else

14： CacheMap.put(key,cacheObject);

15： op generateLocalPut();

16： LocalQueue.add(op);

该算法首先从springAOP封装好的ProceedingJoinPoint对象中获取拦截方法的参数列表，如果参数列表中包含操作对象，则说明是远程PUT操作，因为远程操作是由其它站点生成操作，因此当远程PUT操作在本地进行执行的时候就会调用DAO层的方法，从而再次被springAOP拦截器拦截，因此为了分辨是本地的service层调用DAO层方法还是远程操作调用的DAO层方法，这里把远程操作对象作为额外的参数进行传递来进行识别。但是因为参数中添加了操作参数，因此远程操作在缓存中查找对应的键值的时候，就会不能完全匹配，因此远程操作需要处理一下参数列表信息，将额外的操作参数移除，从而如果缓存命中的话，能够键值匹配。因此在Function5，2-3中，如果是远程操作，也要重新生成键值字符串。反之，如果参数列表中不包含操作对象，则说明是本地操作。

如果是本地操作的话，需要根据ProceedingJoinPoint对象来生成键值，生成键值所需要的信息包含操作调用的方法的全名(包括包名，类名)，以及方法的参数类型列表和参数值列表，这些数据都将首先生成json类型的数据，并通过java方法来转化成string类型生成键值。接着根据生成的键值来判断CacheMap中是否已经包含该键值的对象，即缓存是否命中，如果缓存不命中，则需要继续执行被拦截的方法，获取方法的返回结果集；如果缓存命中，则需要判断缓存对象是否被设置为无效，如果是无效，同样也需要继续执行被拦截的方法，从而获取最新的数据库中的数据，将缓存数据对象进行更新，并更新缓存。最后把生成的本地的操作对象添加到本地执行队列中。注意，这里生成的本地操作对象，会赋值操作的类型和操作的内容(即生成的键值)，但是操作中的时间戳的属性会设置为空，因为所有本地操作对本地站点时间戳的修改以及本地的操作发送都通过统一的本地操作队列进行顺序执行，以防止多线程并发对全局的站点时间戳的不安全修改以及远程操作的地址空间回溯相互干扰的问题。

接收远程操作对象以及本地操作发送到其他站点都需要依靠站点和站点之间的通讯来实现。这个实现包括每个站点对其他各个站点发送本地操作，以及本地站点监听其他站点实时发送过来的远程操作两个部分的通讯内容。同样，为了保证系统中各个模块的松耦合，站点与站点之间的通讯是通过消息队列插件来实现的，消息队列是在大型系统中，用于不同子系统或者不同子模块进行通讯的一个常用插件，文本使用的是activeMQ开源框架来实现消息队列，并且activeMQ和spring可以通过配置来进行整合。消息队列基本上可以分为两大类，一类是生产者/消费者模式，还有一类是订阅/发布模式，前者适合于点对点的消息发送，后者则是一对多的模式，在缓存模块与缓存模块的消息通讯中，本文选择的是订阅/发布模式。即每个站点即是订阅者又是发布者，因此在实现过程中需要定义两个基本的类，一个是监听器类，主要负责实时监听远程操作，并通过调用MessageConvert类来对序列化的消息转化操作类对象，并将操作类对象添加到远程操作执行队列中，另一个类是发送消息的类，即从已经执行完毕的本地操作队列中获取操作对象，发送到其他订阅同一个主题(topic)的缓存节点中。

将远程操作和本地操作放在队列中执行也是为了保证系统的执行效率，并且考虑到在实际系统运行环境为多线程，因此本地操作队列和远程操作队列都应该注意控制线程安全。

## 基于AST的缓存数据一致性维护技术的操作实现



### 本地操作的执行实现

本文以基于LRU替换算法的缓存来作为一种基本模型进行算法说明。在4.2节中提到本地操作在spring AOP中被拦截的方法中产生，并且操作对象只被赋值了操作的类型以及操作的内容等属性，而时间戳属性是为空的。因此在执行队列中，每当执行完一个本地操作，就需要对该操作对象赋值本地的时间戳值。

在本地操作执行中，因为所有的本地操作全部都放在了队列中执行，因此需要开启一个守护线程，在系统启动的时就开始监听本地操作队列是否为空，如果队列中包含操作对象，则开始获取队首中的操作对象，开始执行本地操作。这样就可以确保AST算法思想中，本地操作需要立即执行的思想，从而保证运行的高效性。这里的本地操作都为PUT类型的操作，而REMOVE类型的操作会在接下来提到如何执行。本地PUT操作的执行算法的相关伪代码如Function6，Function7，Function8所示。

Function6 executeLocalPut(op, CacheMap, SV), op is the local PUT operation that has been generated on the CacheMap before when in the Function7 and SV is the timestamp of the local site

/\* key is the operation contain that generate in Function7 \*/

1：If(CacheMap.isFull(SV) && !LRUQueue().contains(key))

2： If (executeLocalDel(op))

3： generateLocalPutOp(op);

4： End If

5：End If

6：Else

7： generateLocalPutOp(op);

8：End Else

Function6 的功能是判断什么时候产生本地PUT操作并且执行本地PUT操作。第一行判断缓存空间是否已满。缓存是否已满是判断在当前时间戳下，缓存中的有效数据项是否小于缓存的大小。因此isFull()函数实质上是调用retrace方法来获取当前的有效数据项，因为retrace方法会对当前时间戳下的缓存数据项进行有效性标记，因此可以得知当前缓存中有效性的数据个数。当有效性个数等于用户定义的缓存大小时，isFull函数将会返回true；否则，返回false。

在Function6第一行的判断中，除了缓存空间已满之外还需要考虑另一种情况，即：即将进入缓存的数据项是否有缓存命中的情况，如果缓存命中的话，则不需要产生本地的删除操作，而直接更新缓存中的数据即可。之前提到缓存的替换算法是LRU，因此为了支持缓存中LRU算法的实现，需要每次在缓存中执行一个PUT操作从而添加一个数据项的时候，就以该操作的内容属性生成一个节点，添加到LRUQueue队列中的末端，当缓存已满，需要从缓存中选择一个节点进行删除的时候，则只需要从LRUQueue队列中选取首端的节点，然后根据键值去找到缓存中的CacheObject对象，进行REMOVE操作即可。因此，在判断语句中，需要查看LRUQueue队列中，是否包含了该操作的内容，如果已经包含该操作的内容，则只需要执行操作op，并把该操作对应的在LRUQueue中的节点移到队列末端即可。

除此之外，LRU替换算法本来的规则，是根据数据项进入缓存的时钟先后顺序来进行排列的，因此传统的LRU算法是基于时钟时间的，但是AST算法中为了保证全局的时间一致性，采用的是逻辑时间，因此LRU算法在AST的缓存应用中，不能以时钟时间来进行排序和替换，而仍旧需要使用逻辑时间来进行队列的排序。而依据逻辑时间戳来进行先后排列准则还是基于Torder排序算法，即如果是因果关系的操作，则按照一般的向量比较法来进行比较，如果是并发操作，则按照Torder来进行比较大小。因此在LRUQueue中，每当在缓存中PUT一个数据项的时候，就需要在LRUQueue中按照时间戳排序算法插入缓存数据项对应的key值所生成的节点，这样，当进行LRU算法替换的时候，就可以从队列中得到队列首端的key值对象，然后从缓存中获取该key值所对应的缓存值，然后在缓存值中执行REMOVE操作。因此Function6的第一行中，要判断一下队列中是否已经包含这个键值，如果已经包含这个key值，则只需要将该key值的节点移到LRUQueue的末端即可，这里还可以看到，在LRUQueue队列中，其实维护的是缓存节点中的有效值的先后逻辑时间顺序。

Function7 executeLocalDel(op,CacheMap,SV) op is the operation that probably replace the cacheObject in the CacheMap if executeLocalDel returns true and SV is the timestamp of the local site

1：If(!LRUQueue.isEmpty)

2： cacheObjectCacheMap.get(LRUQueue.getFirstNode.val);

3： o cacheObject.getOperationList().getLast();

4： If(op.timestamp is NULL)

5： create a new Timestamp named SVtemp and initialize it as site timestamp

6： SVtemp  SVtemp+1

7： Else

8： SVtemp o.SV

9： If(op.SVop> SVtemp)

10： key LRUQueue.removeFirstNode

11： create a REMOVE node removeNode and initialize the basic attribute

12： SV SV+1

13： attach the SV to removeNode

14： attach the removeNode to cacheObject

15： CacheMap.put(key,cacheObject);

16： SendQueue.add(removeNode);

17： return true;

18： Else

19： return false;

20：End If

Function7是当缓存的有效节点个数等于缓存大小的时候，首先需要从缓存中选择一个数据项进行REMOVE操作。在AST中，REMOVE操作和缓存的remove操作有一点不一样的是，REMOVE操作不是将缓存中的对应键值的数据项移除，而是创新一个删除操作的节点，附着到该REMOVE操作作用的数据项下面。

而且，REMOVE操作不是必然会发生的，考虑这样一种情况，如图4-1所示，在LRUQueue队列中，假设缓存大小为2，并且在缓存中已经存在两个有效的数据节点，他们所对应的时间戳为 (0,1,0)，(0,0,1)，这个时候又来了一个操作，它所附着的时间戳为(1,0,0)，如果按照Torder来对三个节点进行排序的话，应该最终保留下来的节点为(0,1,0)和(0,0,1) ，因此这个最后进来的节点(1,0,0)如果是本地操作的话，是不需要进入缓存的，也不需要执行从而改变本地站点的时间戳(如果是远程操作，则会改变本地站点的时间戳值)。因此，需要每次在缓存已满的情况下，判断一下缓存中的有效节点中，时间戳最小的节点对应的值在缓存数据项下的操作中的最大时间戳是否大于即将要进入缓存的节点的时间戳。

如果缓存中有效数据中最小的时间戳的数据节点比当前要进入缓存的数据的时间戳要小，则首先需要把该数据节点从LRUQueue队列中移除(即队列中的第一个节点)，然后创建本地的REMOVE操作并赋值基本的属性值，例如数据项设置为无效，操作类型设置为REMOVE等，然后更新站点时间戳，把更新的站点时间戳赋值给该REMOVE操作，并将操作节点附着在对应的数据项下面，然后更新缓存信息；否则，则直接返回false。

值得注意的是，无论是远程PUT操作还是本地PUT操作都可能也只能产生本地REMOVE操作，且本地REMOVE操作和其他本地PUT操作一样，都需要放入到SendQueue队列中，通过消息队列组件，依次将每一个本地操作对象发送给其他站点。

如果缓存空间未满或者已经执行了本地REMOVE操作，则将调用Function8来执行本地PUT操作。首先将获得操作op中的操作内容，即键值。然后从缓存哈希表中获得对应的数据项CacheObject，将该数据对象设置为有效，然后更新本地站点时间戳。接着创建一个新的时间戳节点，将更新的本地站点时间戳赋值给它，并附着在缓存数据项CacheObject下面，重新更新缓存。最后将本地PUT操作的操作内容生成队列节点，添加到发送队列末端，由发送队列依次向其他站点发送本地操作。伪代码实现如Function8所示。

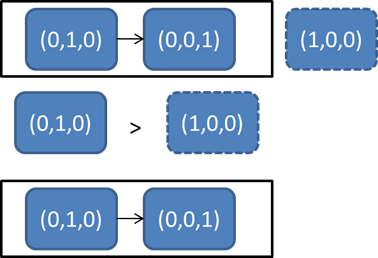


图 4‑1

Function8 generateLocalPutOp(op,CacheMap,SV) op is the local PUT operation that is casual ready to be executed in the CacheMap,and SV is the timestamp of local site

1：CacheObject cacheMap.get(op.getOperationContain);

2：CacheObject.setEffect true;

3：SV SV+1;

4：create a new Timestamp and attach it to the CacheObject

5：CacheMap.put(key,CacheObject)

6：SendQueue.add(op);

至此，本地PUT和REMOVE操作的执行逻辑就介绍到这里。在AST中，本地操作是立即执行的，因此在执行本地操作的时候，不需要进行地址空间回溯操作，因此可以直接设置缓存中数据项的有效性，并且在执行完毕之后，需要将本地操作加入到发送队列中，通过消息队列广播到其他站点。

### 远程操作的执行实现

在4.2节中提到过远程操作是通过对远程站点消息的监听，将收到的序列化消息通过MessageConvert类转化为操作对象，然后将生成的远程操作都将放在远程操作执行队列中进行执行。为了保证本地站点的立即执行思想，因此在每个远程操作执行的时候，都需要判断一下本地执行队列中是否有本地操作对象，如果有，则先执行完本地队列中的操作，然后再执行远程操作。另外，在执行远程操作之前，需要判断一下该远程操作是否满足执行条件。算法伪代码如Function9所示。即假设站点i产生一个操作OPi,时间戳为SVi发送到站点j，站点j的时间戳为SVj，则如果OPi在站点j中因果就绪，就需要满足两个条件：1. SVi[i] = SVj[i]+1，2. SVi[k] <= SVj[k](k>=1 && k<=站点数目，ki)[32]。

Function9 isCausalReady(op ,SV) op is the remote operation that is waiting to be executed on the CacheMap of timestamp SV, if op is satisfied with the execution condition then this function will return true, else return false

1：For(i = 0;i<SITE\_NUM;i++)

2： If (i == op.siteId -1)

3： If(SV+1 != op.SV )

4： return false;

5： End If

6： Else

7： If (SV < op.SV)

8： return false;

9： End If

10：End For

11：return true

当一个远程操作满足Function9中的true条件的时候，才可以执行远程操作。在执行远程操作之前，需要调用retrace操作，将缓存的地址空间切换到远程操作的执行视图下，当远程执行完毕，更新本地的时间戳信息之后，也需要调用retrace操作，将远程操作对缓存数据的作用效果添加到当前的站点中来。Function10描述的是执行远程REMOVE操作的过程。可以看到，Function10 在执行远程操作的时候，只需要把对应的REMOVE操作对象附着在目标数据项下，

Function10 executeRemoteDel(op,CacheMap,SV) op is the REMOVE remote operation that is casual ready in the CacheMap with its timestamp SV

1：retrace(op.SV);

2：key op.getOperationContain;

3：CacheObject CacheMap.get(key);

4：attatch the op to the CacheObject;

5：CacheMap.put(key,CacheObject);

6：SV SV+1

7：retrace(SV)

并更新缓存即可。同时还需要更新本地站点时间戳，然后再进行地址空间回溯操作。

如果远程操作的操作类型为PUT，则首先需要获取该操作的内容，解析出该远程操作需要调用的对象，方法名，以及方法参数列表和参数值列表，利用java语言的反射机制在系统运行过程中动态的调用函数。在调用函数的过程中，同样也会被之前提到的spring AOP中的拦截器拦截，即运行Function5。在Function5中，将识别为远程操作对象，然后调用Function11，来判断是否要执行远程PUT操作。

Function11 executeRemotePut(op, CacheMap, SV, ProceedingJoinPoint), op is the remote PUT operation that will be executed on the CacheMap of SV and ProceedingJoinPoint is the springAOP object.

1：retrace(op.SV)

2：If(CacheMap.isFull() && !LRUQueue.contains(op.getOperationContain))

3： If(executeLocalDel(op))

4： exeRemotePut(ProceedingJoinPoint ,op,CacheMap)

5： End If

6：Else

7： exeRemotePut(ProceedingJoinPoint ,op,CacheMap);

8：SV SV+1;

9：retrace(SV);

Function 11在执行之前需要将当前状态切换到远程操作op的执行视图下，然后判断缓存是否已满(即有效节点数据是否已经超出缓存大小)并且LRU队列中是否包含该操作的执行内容，如果满足If条件，则先执行本地删除操作，如果executeLocalDel函数返回true，则执行远程PUT操作；如果If条件不满足，也需要执行远程PUT操作。接着更新站点时间戳，并回溯缓存地址空间，将远程PUT操作对缓存的操作影响包含进来。这里值得注意的一点是，远程PUT操作无论是否在CacheMap中执行过，都需要在本地时间戳SV中进行相应的更新，这是和本地操作PUT操作不一样的地方。

Function12 exeRemotePut(ProceedingJoinPoint ,op,CacheMap) ProceedingJoinPoint is the springAOP object and op is the remote PUT operation that will be executed in the CacheMap

1：key op.getOperationContain;

2：If(CacheMap.contains(key))

3： cacheObject CacheMap.get(key);

4： If(!cacheObject.isEffect)

5： cacheObject .value ProceedingJoinPoint.proceed();

6： End If

7：Else

8： create a new cacheObject

9： cacheObject.value ProceedingJoinPoint.proceed();

10：End Else

11：attach op to the cacheObject

12：CacheMap.put(key,cacheObject)；

13：LRUQueue.add(key);

当缓存空间未满或者已经执行过本地删除操作，则将会调用Function12。该算法中首先从操作对象中获得操作内容Function12,1行，然后判断缓存中是否已经包含该键值，如果包含该键值但是数据项已经被标记为无效，则需要重新从数据库中获取一遍数据(这里用到了springAOP封装的ProceedingJoinPoint对象的proceed方法)，如果不包含该键值，则创建一个缓存数据对象cacheObject，调用ProceedingJoinPoint对象的proceed方法获取操作的返回结果，赋值给创建的cacheObject对象，如Function12,8-9行所示。接着将远程操作对象附着到缓存对象中，同时更新缓存。最后将新加入的操作节点的内容添加到LRUQueue队列的尾端。

### 缓存系统使用场景模拟

在4.1-4.3节的前两个小节中用伪代码描述了基于AST的缓存数据一致性维护技术的基本实现，接下来这一小节中，将会模拟用户使用系统的场景，来描述在系统应用环境中，该技术是如何实现最终的一致性的。

假设在当前私有云系统中，云控制器下面挂载了两台虚拟机，每台虚拟机都部署了相同的应用，共同对外提供一致的数据服务。假设缓存空间的大小为2，缓存内容初始化为空。现在假设有两个用户，张三和李四，他们都是相互独立的。现张三和李四对整个系统的请求的操作将会分别定位到两个缓存应用服务器中，张三对应站点1，李四定位站点2。

现在张三和李四同时访问系统。所有对系统的访问请求都是通过服务器的action层截获的，并且由action层到达service层至DAO层，所有DAO层中的数据库查询服务以find为前缀，这些数据库查询函数都会被springAOP框架中的拦截器拦截，并且在Around方法中集中处理缓存数据更新和生成本地操作。因为一个用户请求中可能包含多个数据库查询返回结果，即包含多个操作，因此在场景模拟中，以操作为基本单位来进行描述，所有的前端用户请求全部都分解为操作来进行情景展现。

假设张三在站点1和李四在站点2分别产生了如下的操作，如图4-2所示。每个操作的具体内容描述如图4-3所示。

最开始站点1和站点2都为空，当站点1执行完OP1和OP3的时候，站点1中的缓存内容为OP1和OP3产生的数据集合。接着来自站点2的操作OP2到达站点1，缓存空间已满，因此按照之前所述的算法，执行LRU替换操作。OP1所对应的缓存是在LRUQueue队列最前端的，因此OP1操作的时间戳将会和OP2的时间戳相比较，OP2比OP1时间戳要大，因此在OP2操作执行之前，将会在本地产生REMOVE操作OP4，然后再执行操作OP2，这个时候缓存中的有效数据项就为OP2和OP3所对应的操作内容。接着收到由站点2发送过来的远程操作OP5和OP6，OP5是远程REMOVE操作，因此只要直接把该操作附着到

对应的数据项下面即可，因为OP5也是作用于OP1的缓存内容，因此缓存中的有效数据项为OP2，OP3所对应的操作内容。接着远程操作OP6到达站点，操作O6作用的内容和OP3相同，说明此事缓存命中，因此只需要把OP6操作的内容(也是OP3的)在从LRUQueue队列中原本的节点取出，并添加在队列末端即可。此时，站点1中缓存的有效数据项为OP2和OP3的操作内容。站点1整个缓存过程如图4-4所示。

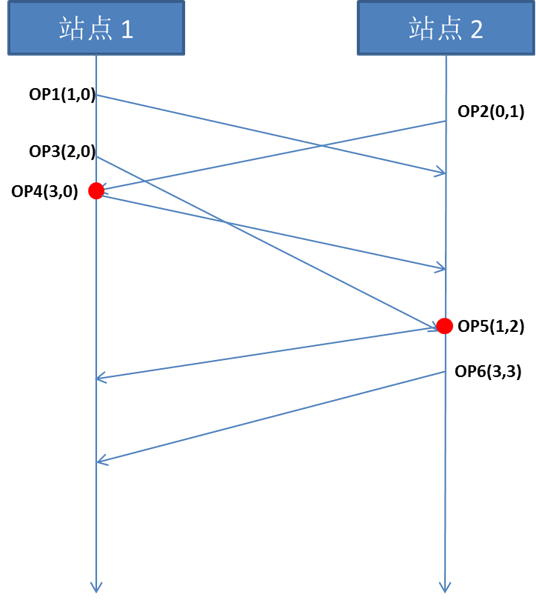


图 4 ‑2

接着再来看站点2的情况。先是执行本地操作OP2，接着执行远程操作OP1(OP1在站点2上因果就绪)，因此缓存中的数据项为OP1,OP2的操作内容。接着到达站点2的是OP4，因为OP4是一个远程操作，且它的时间戳为(3,0)，而本地站点的时间戳为(1,1)，因此不符合Function9的执行条件。因此OP4将会重新放入到远程操作队列中。这个时候，OP3到达了，OP3也是远程操作，它的时间戳为(2,0)，符合站点2的当前的执行条件。因此执行OP3。在执行OP3之前，缓存空间已满，这里调用LRU算法，从队列中选取LRUQueue队列的首端即OP1的操作内容，然后产生一个REMOVE操作，即OP5，然后发送到其他站点，然后再执行操作OP3(因为此时的OP3已经符合执行条件)，执行完OP3之后，此时缓存中的有效数据为OP2和OP3的操作内容。接着，站点2产生了OP6，OP6的操作缓存命中，因此只需要把对应的OP6操作内容在站点2中的LRUQueue队列中的节点移动到队列末端，然后附着操作即可。这样，在站点2中，缓存的有效内容也为OP2和OP3。整个站点2的操作内容如图4-5所示。

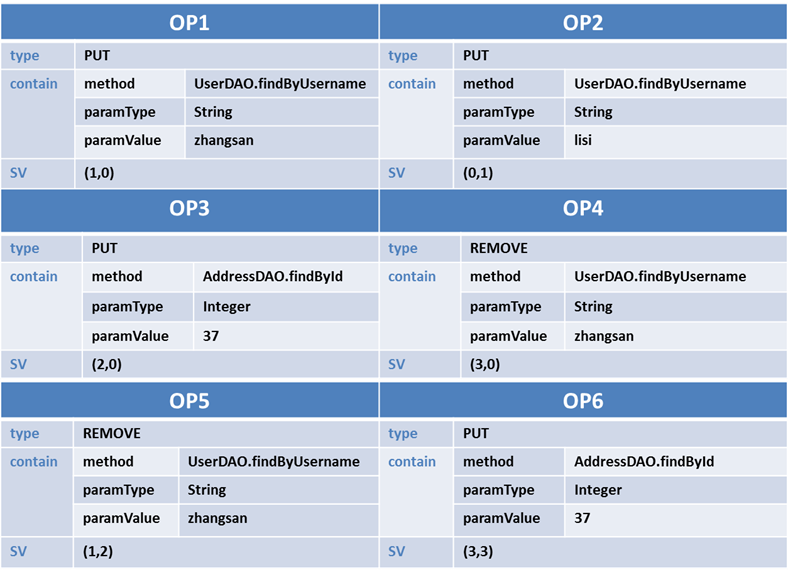


图 4‑3

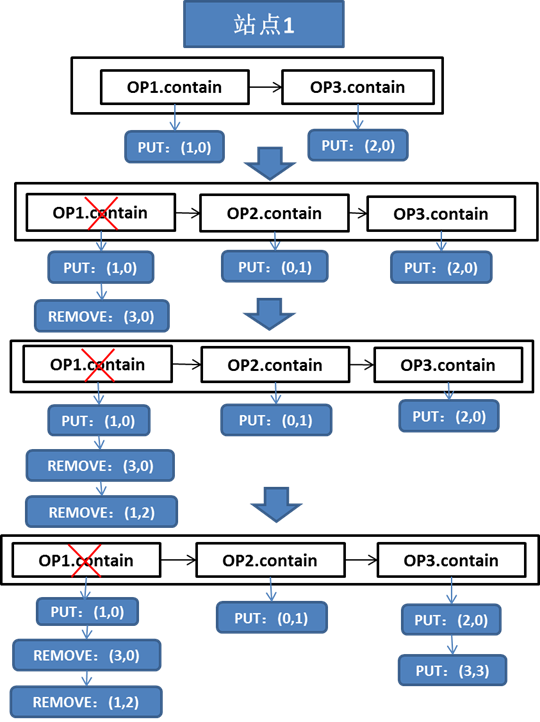


图 4‑4

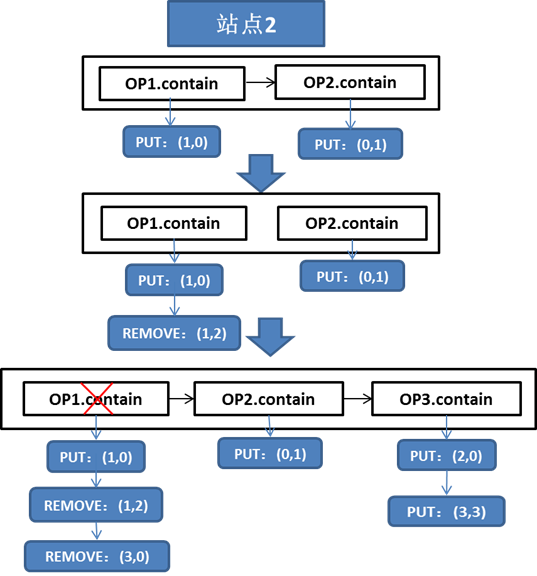


图 4‑5

## 基于AST的缓存数据一致性维护技术的优化



### 基于TTL的缓存数据更新

4.1-4.2小节主要描述了缓存数据一致性维护技术的基本实现。本节将介绍除了替换算法之外，基于TTL更新策略的缓存如何实现一致性。TTL和缓存替换算法不同的是，缓存替换算法常常是当缓存空间已满的情况下，发生数据更新和替换，但是这样常常会导致当前端用户获取数据的时候，数据很可能已经失效的问题；而TTL策略是通过对数据进行定时的更新和删除来保证缓存数据的实时性的。

TTL，即Time To Live，该值表明在缓存中的每个对象从进入缓存到数据失效的时间。当数据失效时，如果用户访问到了该失效数据，则需要重新获取数据放入缓存中。同时，该失效时间的长短和系统性能也有很大的关系。失效时间设置过短，将会导致系统频繁访问数据库，给数据库的造成额外的访问压力；设置时间过长，则会导致访问过期的数据，影响用户的操作。尽管在某种程度上来说，大型系统允许数据在一定程度上的过时，以便系统能够提供更快的响应速度。

在基于TTL的缓存数据一致性维护的时候，根据之前介绍的技术思想，每当一个数据项过期失效，则会产生一个REMOVE操作，发送到其他站点。这个时候就需要为在该站点由于TTL超时而产生的RMOVE操作设置全局唯一的时间戳，并发送到其他站点。例如：假设有A,B两个站点，A产生了一个PUT操作，生成了数据对象O，根据AST一致性算法，这个时候B站点也会和A同步，在缓存中也有数据对象O。这个时候，TTL设置时间过时，A站点中的数据对象过期，设置为无效，则将产生一个REMOVE操作，而站点B的数据对象O也会过期，并产生REMOVE操作，但是如果不规定一个唯一的全局REMOVE操作，则会造成站点B的REMOVE操作和站点A的REMOVE操作发生在不同的时间内，导致不一致性产生。因为操作的传输以及一致性维护是需要时间的，因此两个站点中的REMOVE操作虽然针对的是同一个数据项，但是产生的时间将会不同。为了解决这个问题，需要规定A站点的REMOVE操作为有效操作，而B站点的REMOVE操作则要自动消除掉，这样就指定了全局唯一的REMOVE操作。除此之外，数据在站点之间的传输时间要小于TTL失效时间，以上的算法才能成立。

### 缓存空间的优化

根据4.2节中所描述的操作执行的算法，缓存数据一致性维护技术在维护数据一致性的过程中存在的一个比较大的问题就是由于地址空间回溯而造成的缓存空间占用过大的问题。缓存为了加快数据获取速度，都是在内存中运行的，而内存空间的大小其实是受限制的。缓存一致性维护技术为了支持基于缓存的地址空间回溯算法，常常需要存储历史操作以及操作涉及的数据项，因此随着操作数量的递增，缓存所占的内存空间会越来越大，而有效的数据存储也会因此受到影响。因此，需要定时对缓存空间进行优化，清理已经不再使用的数据，以便既能够满足地址空间回溯操作，又能够使得缓存空间不会占用过大的内存空间。

在[32]的论文中也提到过如何处理多余的数据的算法。从缓存数据一致性维护技术的原理中可以得知，操作的历史记录都是为了地址空间回溯算法准备的。当一个操作OP在每个站点执行过之后，就不会存在其他的并发操作的在执行的过程中需要地址空间回溯，因此该操作以及操作涉及的数据项可以从缓存中删除。根据这个想法，可以对缓存空间进行定期的更新，主要算法思想如下：首先需要获取每个站点的当前时间戳，然后对于时间戳向量的每个维度，取该维度的最小值，这样当遍历完系统中时间戳下的所有维度的时候，便可以得到一个时间戳向量，该时间戳向量的每个维度都是各个站点在该维度的最小值。因此，对于每个站点，在小于该时间戳向量之前的已经执行操作，都可以在缓存数据项中消除其操作对象以及相关的数据，从而使得缓存的空间稳定在一个可控的大小之内。例如：假设一共有三个站点，每个站点的当前时间戳为(2,3,2),(2,4,1),(2,3,3)，每个维度的最小时间戳为(2,3,1)，因此，每个站点都可以删除在时间戳(2,3,1)之前的操作以及操作所影响的数据，因为这代表在该时间戳之前的操作已经在每个站点都执行过了，因此不会存在一个操作和这时间戳之前的操作有并发的关系，并且这些操作也不需要进行地址空间回溯。

# 实验与缓存数据一致性维护技术的应用



## 实验



### 原型系统介绍

在第三章的原理介绍和第四章的算法实现介绍的基础上，本文实现了一个具有缓存数据一致性维护技术的原型系统UserManage，里面主要包含三个实体类，User(用户类)，Address(地址类)，(Role)角色类，主要的功能模块有用户登录，用户注册，按照条件搜索用户，返回所有的权限等。系统的实现框架为Structs2+Spring3+Hibernate3，执行远程操作以及本地操作的后台程序采用spring框架封装好的quarz组件来实现后台程序的运行。系统中缓存的实现，本地操作队列(即本地产生的操作)，远程操作队列(即监听到的远程站点发送过来的操作)，本地发送队列(即已经执行过的本地操作需要发送到远程站点)等数据结构采用的皆为java.concurrent包下面的数据结构，以保证队列在高并发的情况下也能保证增删操作的线程安全。

### 对比实验设计以及结果分析

实验主要分为四个部分，针对四个不同的目的性验证：实验一. 验证基于AST的缓存数据一致性维护技术对系统性能的影响；实验二.验证在多个并发操作的情况下，缓存数据一致性维护技术的性能情况；实验三.验证缓存大小对于缓存数据一致性维护技术的影响。实验四，验证在节点失效情况下缓存数据一致性维护技术对缓存命中率的影响。

实验一.通过请求数目递增来验证基于AST的缓存数据一致性维护技术对于系统性能的影响。

实验环境：操作系统为Windows7的虚拟机(IP地址为10.131.200.188)作为服务器，并已经安装了Java JDK7.0并配置好Java和Tomcat6的环境变量，以及安装了Myeclipse开发环境，方便部署和服务器的启动。采用一台独立的虚拟机(IP地址为10.131.200.97)来模拟客户端，向服务器点发送请求。

实验过程：1. 向没有添加缓存的UserManage系统中依次发送 10-100个包含并发操作的请求，并计算出对于每个请求系统响应所需要的平均时间；2.向有缓存的并且包含一致性算法的UserManage系统中同样发送10-100个包含并发操作的请求，缓存大小设置为100，计算每个请求系统响应所需要的平均时间；3.向有缓存但是不包含一致性算法的UserManage系统中同样发送10-100个包含并发操作的请求，缓存大小设置为100，计算每个请求系统响应所需要的平均时间。

根据以上三种情况画出对比图如图5-1所示。图中折线图的横坐标为一次性发送的请求数目，纵坐标为每个请求所响应的时间。蓝色的折线代表无缓存的情况下系统的性能表现，红色的折线代表的是有缓存有一致性算法的情况下系统的性能表现，绿色的折线代表的是有缓存但是没有一致性算法的情况下系统的性能表现。

根据实验一中三条折线图的比较结果可以看到：一.三条折线随着单位时间内的并发请求数目的增加，每个请求的平均响应时间都在递减。无缓存的情况中，虽然系统没有添加缓存，但是由于UserManage采用了hibernate框架来管理数据库，hibernate自带的数据库缓存的机制会随着查询语句的命中缓存而提高查询效率，使得单位时间内的请求响应速度会随之上升。而其他两种情况下，不仅有hibernate自带的缓存查询机制，还有为应用逻辑层添加的缓存，因此响应的速度将会更加明显。随着请求数目的增加，测试数据集中包含的重复的请求数目会越来越高，因此更多的操作只需要直接从缓存中获取数据即可。在这种情况下，请求数目越多，反而会导致整体性能的提升，因为此时大多数的数据结果集都在内存中。二.在有缓存无一致性的情况下，系统运行的性能是最好的(对应绿色折线)，有缓存有一致性的情况(对应红色折线)稍逊一筹。这是因为一致性维护会消耗系统一定的性能，这其中包含地址空间回溯，远程操作的监听和执行等等额外的资源消耗。但是令人惊喜的是，有缓存有一致性的情况比无缓存的情况要表现的好，说明一致性算法尽管对性能有一定的损耗，但是影响并不大。由此可以看到，基于AST的缓存数据一致性维护技术的开销还是比较小的。

实验二.缓存数据一致性维护技术达到最终一致性所需要的时间。

实验环境：两台虚拟机A(IP地址为10.131.200.188)和B(IP地址为10.131.200.189)模拟两个缓存数据节点，分别安装了java jdk以及部署了java环境变量和tomcat环境变量，Myeclipse(主要用来方便服务器的部署)。将另一台虚拟机(IP地址为10.131.200.97)作为独立的客户端，安装部署了用户请求发送的程序。

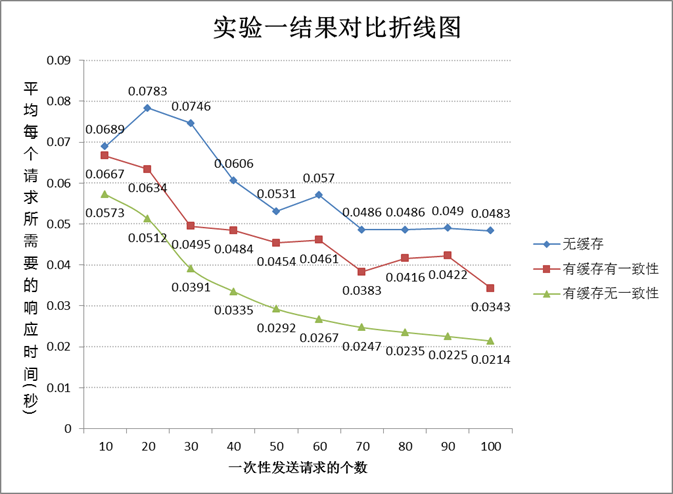


图 5-1

实验过程：两个缓存节点的缓存大小皆为100，分别随机向A和B节点总共发送10-60个请求(每个请求包含一定的重复请求，以及一个请求可能包含多种操作)，来统计实现缓存最终一致性所需要的时间。

以请求数目(操作个数)为横坐标，最终实现一致性时每个操作的平均同步时间为纵坐标，做出如下折线图，如图5-2所示。横坐标以请求数目为单位，每个请求数目刻度都列明了大概包含的操作个数，这个数据主要用来估算每个操作执行大致的平均时间，纵坐标为每个操作同步需要执行的平均时间。

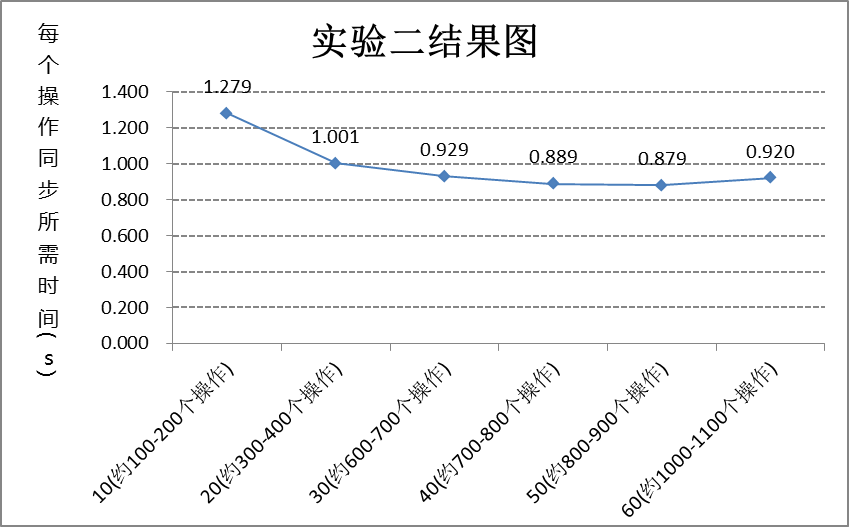


图 5-2

从实验二的结果图中可以看到一开始随着并发操作数目的增加，每个操作同步所需要的单位时间反而在缩短，这是因为随着操作数目的不断增加，缓存命中率上升，使得单位时间内的性能有所提升，但是随着并发操作数据的继续增加，这个时候一致性维护所需要消耗的系统代价将会减弱缓存对系统性能带来的提升，从而使得一致性维护所需要的单位时间再次增加。根据实验结果可以得出，平均每个操作的执行结果在每个缓存节点上达成一致的时间大概在一秒左右。

实验三.缓存大小对缓存数据一致性维护技术在性能上的影响。

实验环境：两台虚拟机模拟两个缓存数据节点，A(IP地址分别为10.131.200.188)和B(IP地址为10.131.200.189)。两台虚拟机分别安装了java jdk并部署了java环境变量，tomcat环境变量以及Myeclipse系统部署平台，用来方便服务器的部署和启动。另一台虚拟机(IP地址为10.131.200.97)模拟客户端。

实验过程：客户端随机将向两台服务器随机发送总共100个请求(请求和请求之间包含重复操作，同时也包含重复的请求，操作个数大约为2250，2150，2110，1960，1780，1660个操作)，之所以100个同样的请求数据集将会产生不一样的操作集合，是因为缓存大小的改变会导致REMOVE操作的数目改变，从而改变整体的操作个数。在实验三中，缓存大小是唯一的变量参数。

以缓存大小作为横坐标，每个操作的执行效果在每个站点上达成一致所需要的时间作为纵坐标，画出如图5-3所示的折线图。从图中可以看出，随着缓存大小的增加，在一定范围内(10到50之间)，系统的整体性能都在提升，因为随着缓存容量的增大，由于缓存已满而产生的本地REMOVE操作(比PUT操作多了LRU替换算法的步骤)数目将会减少，因此一致性算法的整体性能将会提升。但是随着缓存容量的增加，需要维护的缓存中的数据也在增加，每个缓存中数据项也在变得越来越为庞大(因为每个数据项下面附着的操作个数会随着系统的运行越来越多)，因此内存空间也会逐渐变满，使得CPU的执行效率下降，所以随着缓存空间变大，一致性算法的性能也会下降。

实验四. 验证缓存数据一致性维护技术在节点失效情况下对缓存命中率的影响。

实验环境：两台虚拟机分别模拟两个缓存节点A(IP地址为10.131.200.188)和B(IP地址为10.131.200.189)，每台虚拟机上都安装了Java jdk和MyEclipse，并部署了原型系统UserManage。两台虚拟机a(IP地址为10.131.241.41)和b(IP地址为10.131.200.97)模拟两个独立的客户端，同时向A和B两个节点发送请求。

实验过程：对比实验分为两组。第一组：A和B节点上部署的是具有一致性技术的原型系统，将100个请求平均分成两组，这两组请求数据交集为0。客户端a和b分别向A和B节点各发送50个请求，并每秒记录下A节点的缓存命中率变化。当A和B节点数据一致性之后，将之前发送给B节点的50个请求再次发送给A节点，同样也需要每秒记录A节点的缓存命中率变化；第二组：A和B节点上部署的是不具有一致性技术的原型系统，先将50个请求发送给A节点，当A节点的命中率稳定之后，再将之前发送给B节点的50个节点发送给A节点，记录下A节点在整个过程中的缓存命中率变化。

实验结果如图5-4所示，横坐标为时间轴，纵坐标为缓存命中率。从实验结果可以得出以下结论：1.一致性技术在某种程度上削弱了缓存命中率，但是削弱的程度并不是很大；2.当B节点宕机时，在没有一致性技术的系统中，A节点的命中率会急速下降，需要经过一段时间之后才能够回到原来的状态，而有一致性技术的系统曲线更加平缓，因此B节点的宕机对于A节点来说，虽然缓存命中率略有下降但是影响不是很大，并且经过比较短的时间就可以恢复到原来的状态。

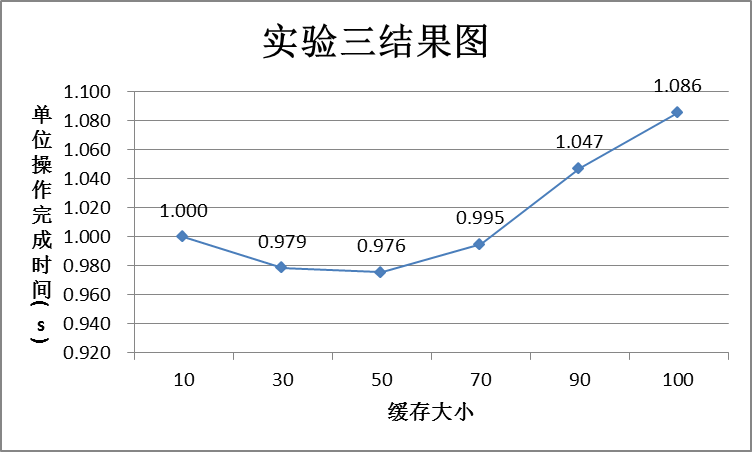


图 5-3

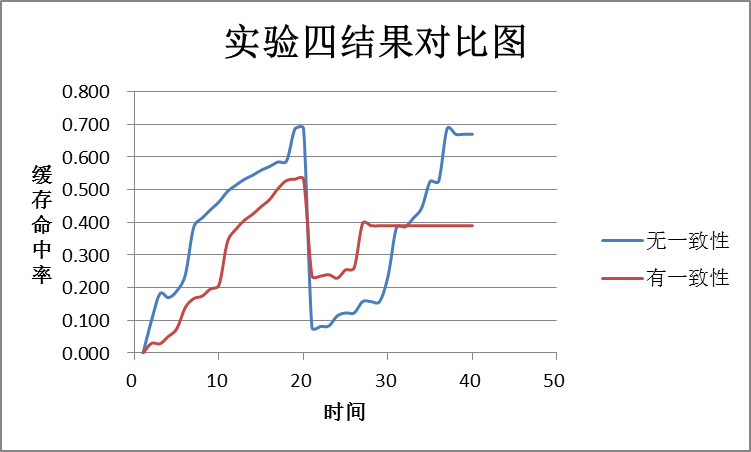


图 5-4

## 面向残疾人无障碍自理管理平台的缓存应用



### 残疾人无障碍自理管理平台介绍

残疾人无障碍自理管理平台是复旦大学和上海市残联共同合作完成的项目，该项目基于人本计算[62]的思想来改善残疾人的日常生活。该系统分为前后台两个部分，前台接口支持残疾人用户通过APP设备语音控制智能辅具，后台接口支持残联工作人员的日常信息化管理服务。

残疾人无障碍自理管理平台的架构设计图如图5-5所示。整体的架构设计是基于分布式的，残疾人用户或者残联工作人员通过移动端和网页访问残疾人自理管理平台。反向代理服务器在收到用户请求之后，根据用户的注册所在地区，将请求分发到指定负责响应的应用服务器节点上。数据源配置是以master-slave模式来实现持久层的高可用，读写数据库之间的同步采用的是MySQL的配置实现的。每台应用服务器默认连接读数据库，当需要写数据时，则进行数据源切换。在用户权限为区级以下的用户中，产生的相关数据存储在对应服务器下数据库中；对于市级权限以上的用户中，则将进行数据库全局切换，将全局数据存储在每个数据源中。系统的功能图如图5-6所示。前台功能是对残疾人控制智能家具的接口支持，包括按键学习，智能辅具功能按钮的管理，智能辅具下载和语音控制；后台功能则是对残联工作人员信息化管理残疾人用户以及辅具的接口支持，包括残疾人管理，智能辅具管理，残疾人数据统计等。

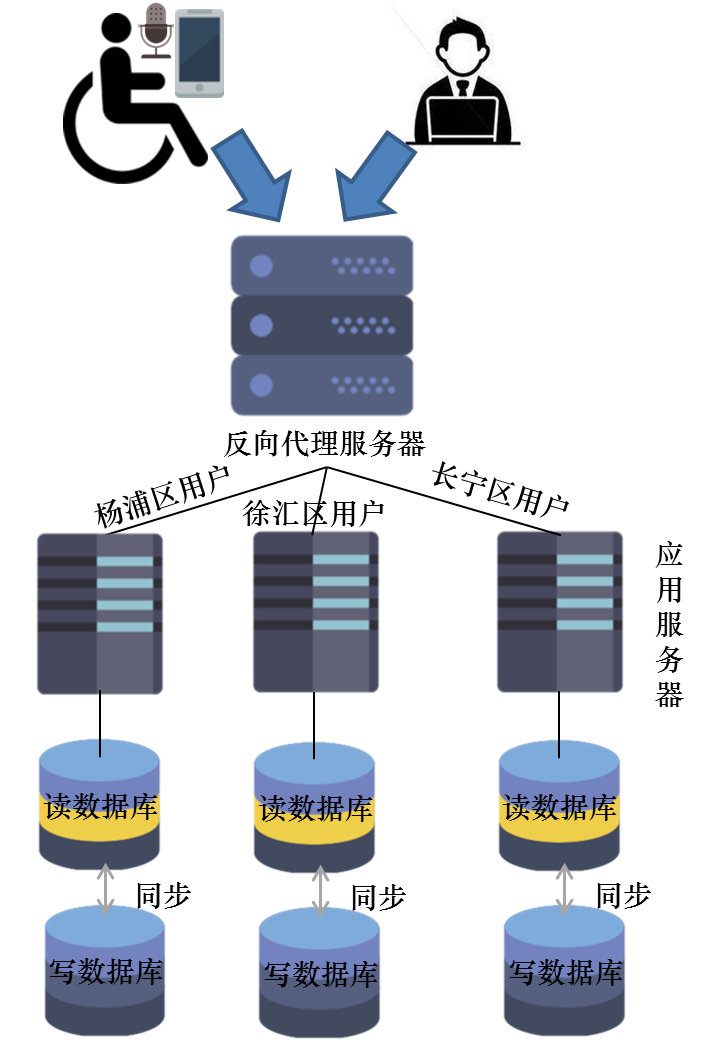


图 5‑5

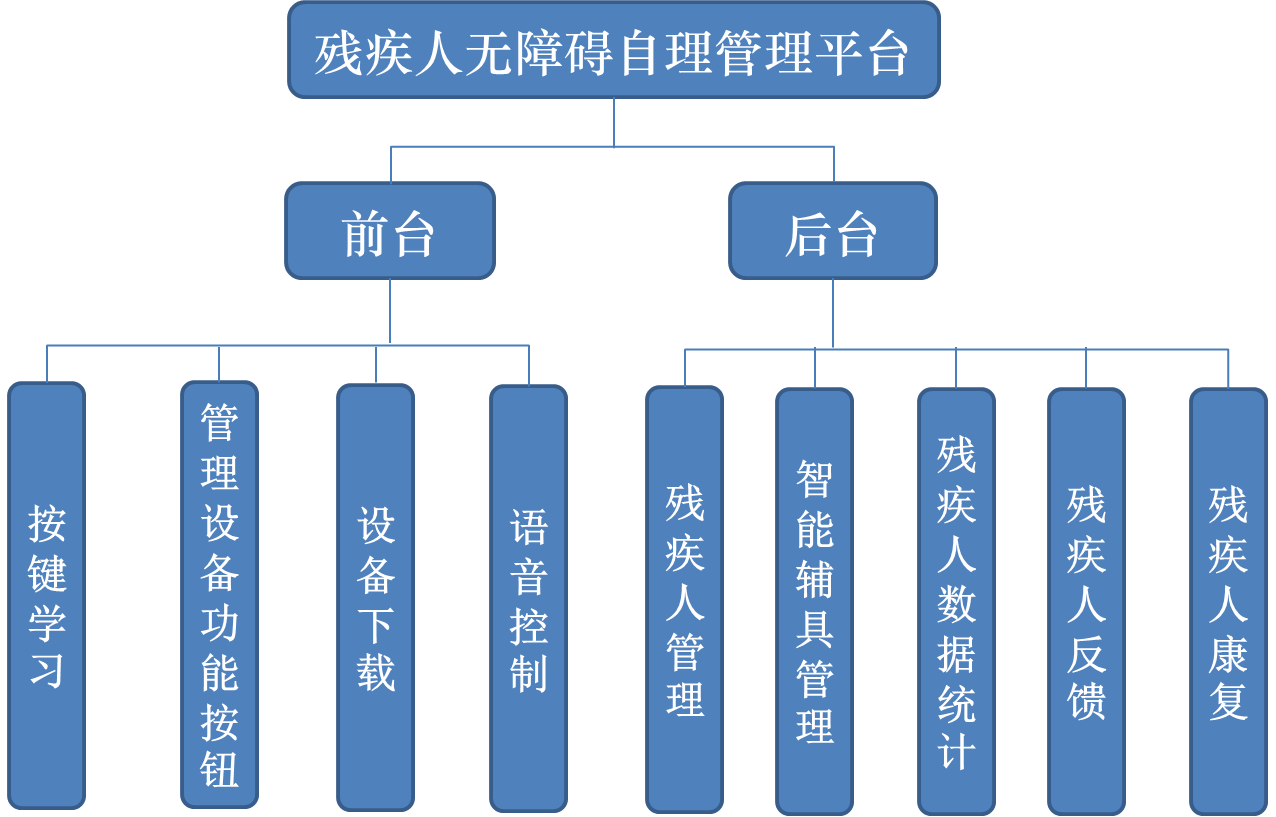


图 5‑6

### 残疾人无障碍自理管理平台下部分接口的实现

在5.2.1节中概括性地介绍了残疾人无障碍自理管理平台的基本架构和主要功能模块，本节将详细介绍残疾人数据统计模块中部分接口的实现过程。

残疾人数据统计模块是残疾人无障碍自理管理平台的所有模块中和数据库交互最为频繁的模块，该模块包含了对残疾人以及残疾人使用智能辅具的使用情况等各个方面的信息统计，目的是为了让上海市残联对残疾人使用辅具的情况有更为直观全面的了解。该模块中的各个接口的逻辑实现过程包含各种较为复杂的数据库查询操作，如嵌套查询，级联查询等。

统计内容包含按区域统计用户数量、智能辅具的数目，按照辅具种类统计用户数量，按照对辅具的控制方式统计用户操作的次数等。这些接口涉及了用户表，辅具表，中控器表，辅具类型表以及地址表之间的逻辑关联操作，每张表之间的主键和外键之间的关联示意图如图5-7所示。下面将详细地介绍其中三个接口的实现过程。

接口一：按照区域统计用户数目和辅具数目。该接口的功能是统计指定地区内的残疾人数目以及智能辅具的数目。该接口的输入参数有两个，一个参数是地址ID数组，数组中的每个数字代表了在地址表中的地址ID。当数组长度为1时代表统计的是该地址ID下的所有子地址(地址表是一个树状的结构，每个地址都包含其上一级地址)；还有一个参数是角色ID，代表的是当前数据请求方的权限级别，不同权限的管理员只能统计所在区域内的数据（残疾人无障碍自理管理平台系统中最大的权限级别是超级管理员，最小的权限级别是普通用户。在统计模块中，最小的用户权限是街道管理员，因为普通用户不具有统计权限）。接口的输出参数为地址数组中的每个地址ID下统计的辅具数目和用户数目。实现的伪代码如下所示。

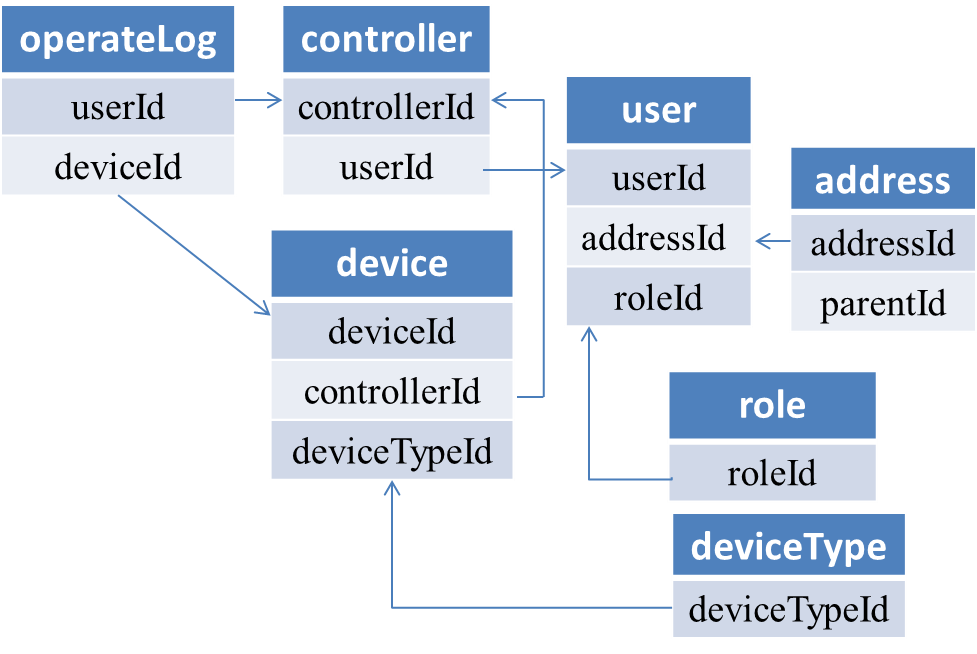


图 5‑7

StatisticUsersAndDevicesByArea (AreaIdArray,roleId), AreaIdArray is the array that stores the id of area and roleId presents the right of user

1：If roleId equals Street Administrator

2： addressList🡨getAddressList;

3：End If

4：Else

5： If ArrayIdArray.length = 1;

6： addressList 🡨getAddressChildrenList;

7： Else

8： addressList 🡨 getAddressList;

9：End Else

10：For address in addressList

11： userNum 🡨 statisticUserFromAddress;

12： userList 🡨 getUserListByAddress;

13： controllerList 🡨statisticControllerByUserList;

14： deviceNum🡨statisticDeviceByControllerList;

15：End For;

接口一首先解析请求参数，获取权限ID和地址ID数组，然后判断该权限ID是否属于街道管理员。如果是街道管理员，则不会出现需要返回下一级地址信息的情况，因此直接得到地址ID数据对应的地址信息列表即可。反之，需要判断地址ID数组的长度是否为1，如果为1，则需要返回下一级地址的统计信息。逻辑代码对应1-9，这一部分的作用是为了确定需要统计的区域范围。接着对地址列表进行遍历，对于每个地址对象，根据user表中的外键addressId来统计在该地址下的用户数目。而辅具数目的统计相对来说会复杂一点：首先从数据库中获取该地址下的所有用户，然后从controller表中查找所有包含这些用户ID的控制器列表，最后从控制器列表中统计对应的辅具数据，对应逻辑代码10-15。最后将统计的信息按照一定的格式进行返回。

接口二：按照辅具类型统计用户操作数目。该接口功能是在给定的地址ID下，对该地址下对应的所有用户的操作智能辅具的情况按照辅具的类型进行归类统计。接口的输入参数为需要统计的智能辅具操作数目的地址ID，输出参数是该地址下所有用户对每种智能辅具的操作数目。该接口的具体实现过程如下所示。

StatisticOperationByDeviceType(areaId),statistic the operation numbers of every device for the given area

1:deviceTypeSet🡨getAllDeviceTypes;

2:For dt in deviceTypeSet

3: initial dt.num 🡨 0

4:End For

5:operationList 🡨 getOperationByAreaId

6: For op in operationList

7： device 🡨 op.getTargetDevice;

8: deviceTypeSet.get(device.deviceType)++;

9： End For

10：End For

接口二首先从数据库中获取所有的辅具类型，将辅具类型存储在类型为哈希表的deviceTypeSet中，键为辅具类型，值为该辅具的操作次数，初始化为0，对应伪代码第1-4行；接着从数据库中按照地址ID查找该区域内所有用户的所有操作记录，并赋值给变量operationList，对应伪代码第5行；然后遍历operationList，从每个操作中获取操作的智能辅具，并根据智能辅具中包含的辅具类型属性从deviceTypeSet中查找对应的键值，并将相应的操作次数自增一，对应伪代码6-10行。

接口三：按照智能辅具的控制方式统计操作次数。该接口的功能是根据给定的区域ID按照控制方式来统计所有区域用户对智能辅具的操作，旨在了解不同类型的残疾人控制智能辅具的倾向性。该接口的输入参数是地址ID，输出参数以每种控制方式为归类的操作次数统计。对智能辅具的控制方式主要是三类：按键控制，语音控制和眼识控制。接口三的伪代码如下所示。

StatisticOperationByControll(areaId),statistic operation numbers of the user by control ways under the given areaId

1:operationList 🡨 getOperationByAreaId

2:For op in operationList;

3: If op.opMethod equals ‘眼识’

4： resultSet.get(“眼识”)++;

5: End If

6: If op.opMethod equals ‘语音’

7： resultSet.get(“语音”)++;

8： End If

9： If op.opMethod equals “按键”

10： resultSet.get(“按键”)++;

11： End If

12：End For

接口三首先根据请求参数地址ID从operation表中查找所有该地址下的所有用户的操作记录，然后对每条操作记录，判断操作中的控制方式属性，对相应的计量自增一，最后将统计结果按照一定的格式返回。

### 残疾人无障碍自理管理平台存在的问题分析

根据5.2.1节中介绍的残疾人无障碍自理管理平台的基本架构和主要功能以及5.2.2节中对残疾人统计模块中部分接口具体实现过程的介绍，以此来分析该系统中主要存在的问题。这些问题主要分为两大类：第一类是残疾人数据统计模块常常出现响应速度过慢的问题；第二类是应用服务器的高可用性问题。

统计模块响应过慢的问题。在大多数情况下，原系统中的数据库操作都是读操作，而大部分的读操作都集中在残疾人数据统计模块。数据统计模块中几乎每个接口的实现过程都涉及了多张数据库表的条件查询，包括模糊查询，外连接查询，子查询，分页查询等，具有比较复杂的逻辑关联度和繁琐的计算量。当用户数据规模增大、访问并发量增加时，该模块的响应速度将会大大下降。

服务器的高可用性问题。原本的架构设计是基于分布式的，三台应用服务器分担了来自各个地区用户的请求流量，使得整体的性能有了一定的提升。但是在架构设计时，并没有考虑到服务的高可用性的问题，因为每台服务器对于某个地区来说都是单点的，因此假如某台服务器宕机时，对该台应用服务器下负责响应的区域用户来说，系统服务将出现不可用的情况，将大大影响用户体验。

### 残疾人无障碍自理管理平台的架构优化与缓存应用

基于5.2.3节中对残疾人无障碍自理管理平台存在问题的分析，在本节中根据残疾人无障碍自理管理平台的项目需求背景，将该架构迁移到了私有云下，凭借私有云具有定制化，强大的数据存储能力和高可用性等优点，来改进原系统在架构设计中存在的问题。更进一步，将缓存数据一致性维护技术在该系统中进行了应用，旨在提高系统整体的数据服务效率和保证系统的稳定运行。

整体的系统架构设计图如图5-8所示。残疾人用户和残联工作人员的请求通过网络访问私有云，私有云中的反向代理服务器将请求根据用户注册的所在地区分发给不同的控制器，节点控制器再将请求分发给各个应用服务器节点。在应用层中，所有的查询残疾人的数据请求都将先访问缓存。由于残疾人系统中的应用层和缓存层在业务逻辑上是相互独立的，缓存访问的仅仅是数据库DAO层以及应用层提供的数据接口，因此缓存在残疾人无障碍自理管理平台上的应用实现采用的是松耦合的方式：只需要将封装好的缓存数据一致性维护技术中算法的功能模块以及操作广播的实现模块的java类分别打包整合到残疾人系统中，缓存数据一致性维护技术就可以在系统上运行。

在残疾人无障碍自理管理平台中，缓存和应用层以及数据库之间的操作对应的数据流向逻辑如图5-9所示。在缓存中，key存储的是对残疾人以及相关智能辅具操作的数据请求，value是key所对应的数据查询结果集。当应用层产生残疾人数据请求时，如果请求的残疾人数据在缓存中命中，则直接从缓存中获取查询结果集返回；反之，则从数据库中获取残疾人相关数据返回，同时，还需要将该查询结果放入缓存中，以便下次请求相同的残疾人数据时可以直接从缓存中读取。除此之外，为了维护每个缓存节点之间的数据一致性，还需要将每个请求操作附着在缓存中对应的数据项下面以实现地址空间回溯操作。当请求操作是本地操作时，根据缓存数据一致性维护技术，要把操作发送到其他缓存节点中，实现各个缓存节点中残疾人查询数据的一致性。

以5.2.2节中残疾人数据统计模块中的接口一为例来详细阐述数据统计接口和缓存接口之间的逻辑过程，如图5-10所示。接口一的功能是统计指定区域下的用户数目以及智能辅具数目，当残联工作人员将统计请求发送给应用层时，应用层将该请求分解为三个部分的数据请求操作：操作1为请求给定区域下的所有用户列表，操作2为用户(操作1的返回值)下已申请的中控器列表，操作3为中控器(操作2的返回值)下所有连接的智能辅具列表。每个操作在访问数据库之前都会被缓存层截获，并从缓存中以平均O(1)的复杂度查找是否包含该操作的数据结果集。如果缓存命中，则直接从缓存中获取数据返回给应用层，同时当该操作是本地操作时则需要将操作广播到其他残疾人应用服务器节点上；如果缓存失效，则操作绕过缓存，执行数据库查询语句，将查询结果集返回给应用层，同时在缓存中存储相应的统计数据。整个过程中，缓存相当于数据拦截器，将所有的查询数据库的操作截获并根据操作请求所带的参数生成唯一的key值来查找缓存中是否有操作请求的数据，如果有，则缓存直接返回，如果缓存不命中，则操作访问数据库，返回数据的同时还需要将数据存储在缓存中。因此以缓存的视图来看，它调用的仅仅是应用层和数据库层提供的接口，和应用的业务逻辑耦合性非常小。

在如图5-8所示的私有云架构下，当具有缓存数据一致性维护技术的残疾人无障碍自理管理平台中的统计模访问并发量增大时（例如有多个管理员同时统计当前的残疾人用户以及智能辅具的使用情况），缓存中存储的热点数据将会大大减少应用层数据获取的时间，从而加快了残疾人数据统计模块在高并发量下的平均响应速度。当某个应用服务器节点在系统运行过程中发生宕机时，节点控制器将会把原本定位在宕机服务器上的请求重新定位到其他可用的服务器节点上，由于缓存数据一致性维护技术维护了每个缓存节点中数据的一致性，因此重新定位的请求在一定程度上能够在新定位的缓存节点中找到数据，进行快速的请求响应，从而保障了残疾人无障碍自理管理平台下数据服务的高效性。

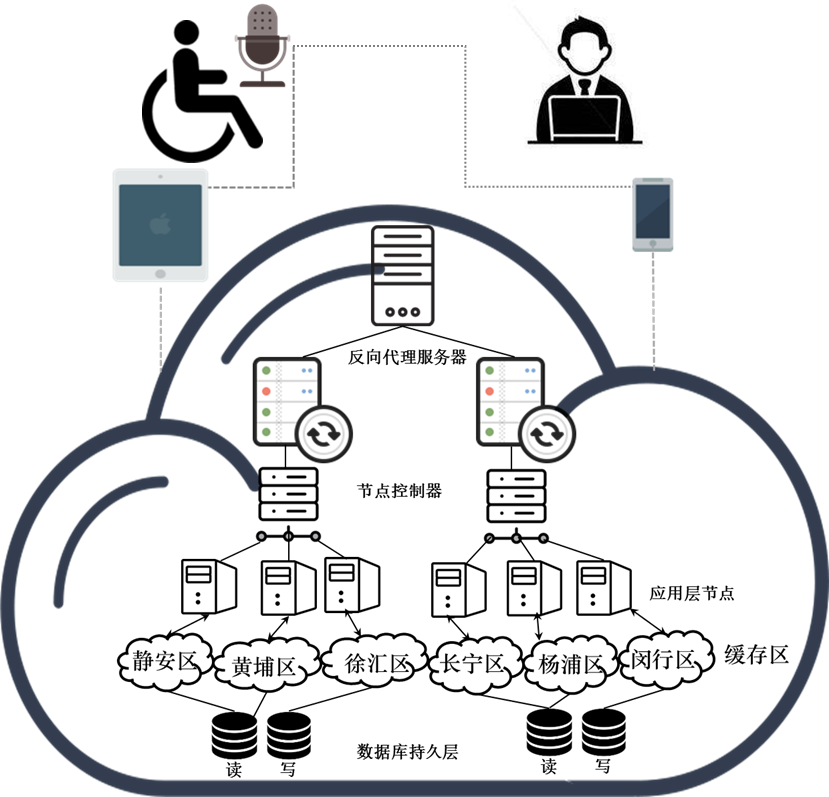


图 5‑8

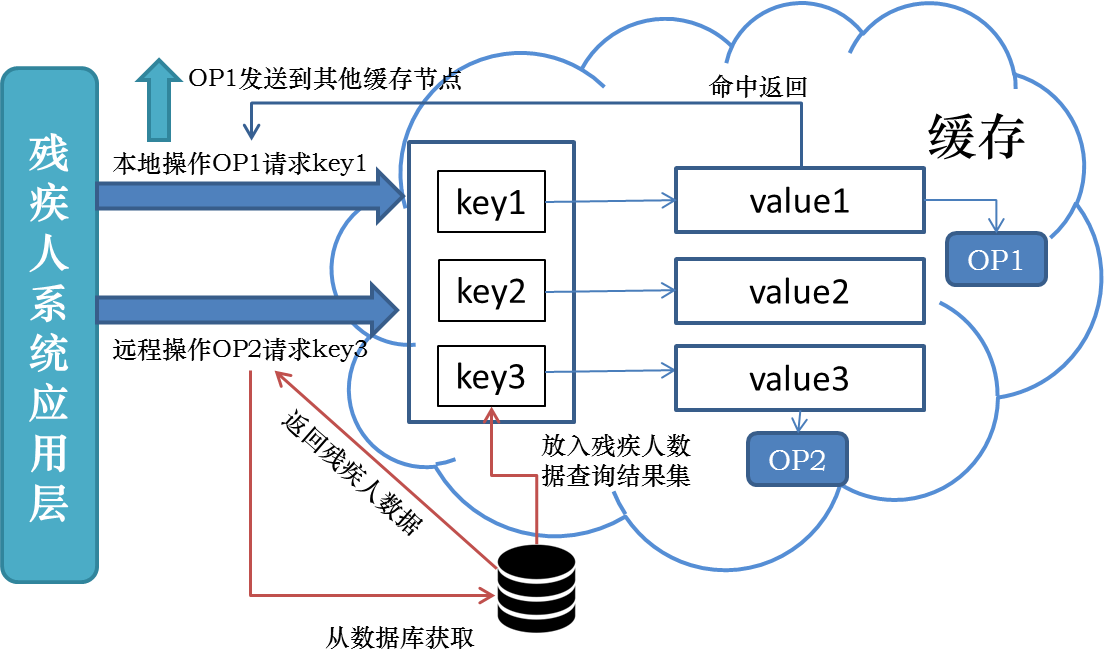


图 5-9

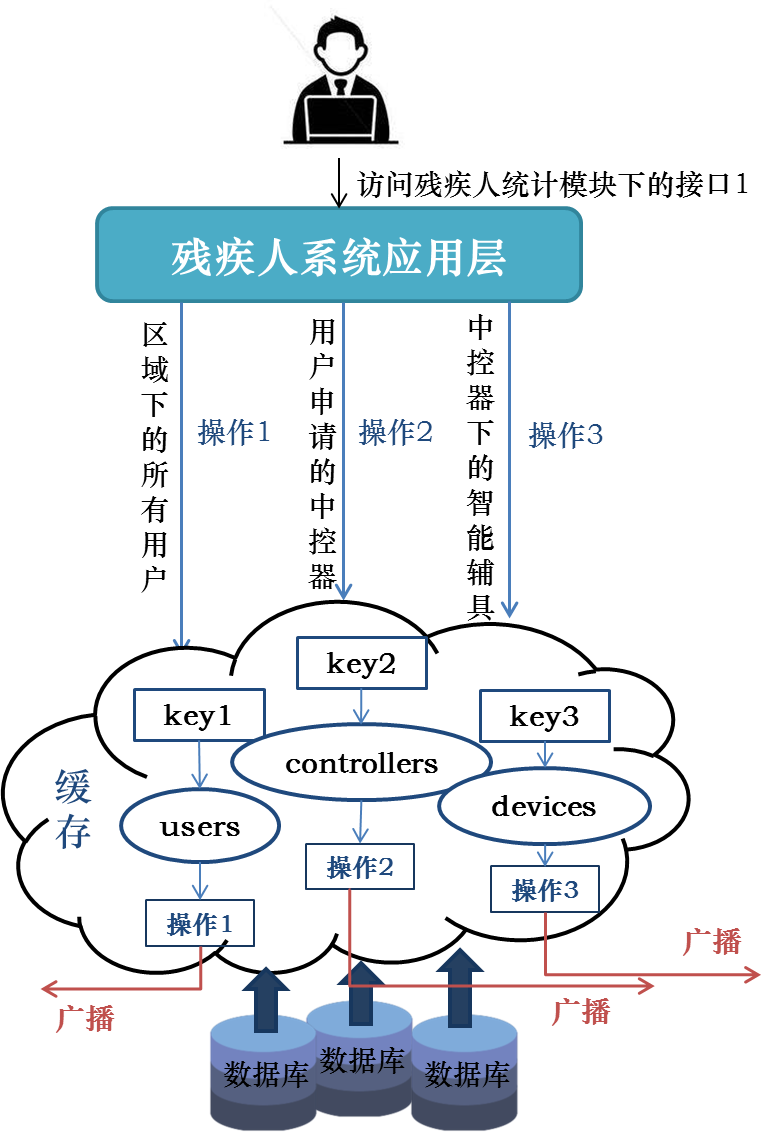


图 5-10

# 总结与展望



## 总结

本文在私有云平台下重点提出了基于AST[13]的缓存数据一致性维护技术，解决了由于缓存节点在动态伸缩情况下而产生的缓存命中率下降的问题，并实现了缓存数据一致性维护算法的原型系统，在该原型系统上进行了对比实验，最后将具有缓存数据一致性维护算法的缓存应用到基于私有云的残疾人无障碍自理管理平台的设计中。

首先本文介绍了私有云的特点和应用场景，提出了基于私有云平台的系统设计。私有云以其可定制化，数据控制更为安全等特点在个人，企业和政府组织的云计算解决方案中应用越来越为普遍。私有云中也常常部署控制器和多个应用服务器来组成具有高可用性，可伸缩性和稳定性的云计算架构。同时，为了保证私有云下的数据服务效率，常常在应用中部署缓存来提高平均请求响应速度和减轻数据库的访问压力。

接着，在私有云的系统架构设计基础之上，本文重点提出了基于AST的缓存数据一致性维护技术。该技术在AST的思想上，根据缓存的应用特点进行了一定的扩展和创新。该技术主要包含三个重要组成部分：基于缓存的地址空间回溯算法，基于缓存的控制算法和缓存中的操作执行算法，并以LRU替换算法的缓存为研究对象，探讨了缓存中的数据一致性维护策略。进一步，本文实现了具有缓存数据一致性维护技术的用户管理系统userManage，并在该原型系统上通过实验验证了技术的有效性。

最后，本文提出了面向残疾人无障碍自理管理平台项目的缓存架构设计。首先介绍残疾人无障碍自理管理平台，包括基本架构，主要功能模块特别是残疾人数据统计模块下的三个重要接口的具体实现过程；接着对该平台目前存在的问题进行了分析，最后将缓存数据一致性维护技术在该平台上进行了设计与实现。

## 展望

本文的主要研究工作是对缓存数据一致性维护技术的设计与实现，并将该技术应用在面向残疾人的无障碍自理管理平台的架构设计中，用来解决当热点数据由于系统架构伸缩所引发的数据服务效率低下的问题。除了维护每个缓存节点之间的数据一致性的相关问题，在系统设计中还存在其它一致性方面的问题有待研究和解决：

一. 事务型数据在缓存中的一致性维护问题。在系统的需求中，包含一类特定的业务逻辑，需要进行事务型的操作，对于写事务的操作，可以直接在数据库中执行（因为缓存是只读的，不支持数据的修改）；对于读事务的操作，考虑到大型系统下高并发的情况，通常会选择在缓存中执行，但是这就会引出一个需要解决的问题，即如何利用缓存数据一致性维护技术来保证读事务操作的一致性维护。尽管[32]中给出了基于XML数据结构的事务型操作的解决方案，但是由于缓存中的操作执行逻辑以及数据存储逻辑的不同，事务型操作还需要进行深入的研究。

二.缓存和持久层之间的同步问题。缓存是数据持久层的一部分，在系统运行过程中，缓存中的数据将会动态稳定地存储热点数据。尽管会有TTL策略或者缓存替换策略来提高数据的新鲜度，但是在高频写的情况下，比方说股票市场交易系统等数据随时随刻会更新的系统平台下，缓存数据和持久层中的数据将会产生不一致性的情况。因此，如何保持持久层数据和缓存数据的一致性同步也是一个值得深入研究下去的方向。

参考文献

[1] Armbrust M, Fox A, Griffith R, et al. A view of cloud computing[J]. International Journal of Computers & Technology, 2010, 53(4):50-58.

[2] Bhardwaj S, Jain L, Jain S. Cloud computing: A study of infrastructure as a service (IAAS)[J]. International Journal of Information Technology & Web Engineering, 2010, 2(1):60-63.

[3] Vaquero L M. EduCloud: PaaS versus IaaS Cloud Usage for an Advanced Computer Science Course[J]. IEEE Transactions on Education, 2011, 54(54):590-598.

[4] Berman F, Fox G, Hey A J G. Grid computing: making the global infrastructure a reality[M]. John Wiley and sons, 2003.

[5] Ross J W, Westerman G. Preparing for utility computing: The role of IT architecture and relationship management[J]. Ibm Systems Journal, 2004, 43(1):5-19.

[6] Uhlig R, Neiger G, Rodgers D, et al. Intel virtualization technology[J]. Computer, 2005, 38(5):48-56.

[7] 肖磊. 基于云计算的移动商务研究与实现[D]. 南昌航空大学, 2013.

[8] 吴绍忠, 李靖. 基于云计算架构的公安情报信息平台建设研究[J]. 中国人民公安大学学报:自然科学版, 2010, 16(3):39-41.

[9] Cusumano M. Cloud computing and SaaS as new computing platforms[J]. Communications of the Acm, 2010, 53(4):27-29.

[10] 杨志豪, 赵太银, 姚兴苗,等. 一种适应数据与计算密集型任务的私有云系统实现研究[J]. 计算机应用研究, 2011, 28(2):621-624.

[11] 岳冬利, 刘海涛, 孙傲冰. IaaS公有云平台调度模型研究[J]. 计算机工程与设计, 2011, 32(6):1889-1892.

[12] 王崇霞, 高美真, 刘倩,等. 混合云联合身份认证与密钥协商协议设计[J]. 电信科学, 2014, 30(4):95-99.

[13] Gu N, Yang J, Zhang Q. Consistency maintenance based on the mark & retrace technique in

groupware systems[C]// International ACM Siggroup Conference on Supporting Group Work,

Group 2005, Sanibel Island, Florida, Usa, November. 2005:264-273.

[14] Lamport L. Paxos Made Simple[J]. Acm Sigact News, 2001, 32(4).

[15] Lamport L. Time, clocks, and the ordering of events in a disributed system[M]// Trends in the study of morbidity and mortality /. WHO, 1965:558-565.

[16] Derler P, Eidson J C, Goose S, et al. Using ptides and synchronized clocks to design distributed systems with deterministic system wide timing[C]//2013 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication (ISPCS) Proceedings. IEEE, 2013: 41-46.

[17] Corbett J C, Dean J, Epstein M, et al. Spanner: Google’s globally distributed database[J]. ACM Transactions on Computer Systems (TOCS), 2013, 31(3): 8.

[18] Cao P, Liu C. Maintaining strong cache consistency in the World Wide Web[J]. IEEE Transactions on Computers, 1998, 47(4): 445-457.

[19] Jung J, Berger A W, Balakrishnan H. Modeling TTL-based Internet caches[C]//INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies. IEEE, 2003, 1: 417-426.

[20] Wessels D. Squid internet object cache[J]. 1998.

[21] Fawaz K, Artail H. DCIM: Distributed cache invalidation method for maintaining cache consistency in wireless mobile networks[J]. IEEE Transactions on Mobile Computing, 2013, 12(4): 680-693.

[22] Artail H, Safa H, Mershad K, et al. COACS: A cooperative and adaptive caching system for MANETs[J]. IEEE Transactions on Mobile Computing, 2008, 7(8): 961-977.

[23] Mershad K, Artail H. SSUM: smart server update mechanism for maintaining cache consistency in mobile environments[J]. IEEE Transactions on Mobile Computing, 2010, 9(6): 778-795.

[24] Huang Y, Cao J, Jin B, et al. Flexible cache consistency maintenance over wireless ad hoc networks[J]. IEEE Transactions on Parallel and Distributed Systems, 2010, 21(8): 1150-1161.

[25] Terry D B, Prabhakaran V, Kotla R, et al. Consistency-based service level agreements for cloud storage[C]//Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, 2013: 309-324.

[26] Ports D R K, Clements A T, Zhang I, et al. Transactional Consistency and Automatic Management in an Application Data Cache[C]//OSDI. 2010, 10: 1-15.

[27] Ellis C A, Gibbs S J. Concurrency control in groupware systems[C]//Acm Sigmod Record. ACM, 1989, 18(2): 399-407.

[28] Ressel M, Nitsche-Ruhland D, Gunzenhäuser R. An integrating, transformation-oriented approach to concurrency control and undo in group editors[C]//Proceedings of the 1996 ACM conference on Computer supported cooperative work. ACM, 1996: 288-297.

[29] Sun C, Jia X, Zhang Y, et al. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems[J]. ACM Transactions on Computer-Human Interaction (TOCHI), 1998, 5(1): 63-108.

[30] Nichols D A, Curtis P, Dixon M, et al. High-latency, low-bandwidth windowing in the Jupiter collaboration system[C]//Proceedings of the 8th annual ACM symposium on User interface and software technology. ACM, 1995: 111-120.

[31] Sun C, Ellis C. Operational transformation in real-time group editors: issues, algorithms, and achievements[C]//Proceedings of the 1998 ACM conference on Computer supported cooperative work. ACM, 1998: 59-68.

[32]Yang J, Wang H, Gu N, et al. Lock-free consistency control for web 2.0 applications[C]//Proceedings of the 17th international conference on World Wide Web. ACM, 2008: 725-734.

[33] Xia H, Lu T, Shao B, et al. A partial replication approach for anywhere anytime mobile commenting[C]//Proceedings of the 17th ACM conference on Computer supported cooperative work & social computing. ACM, 2014: 530-541.

[34] Ren K, Wang C, Wang Q. Security challenges for the public cloud[J]. IEEE Internet Computing, 2012, 16(1): 69.

[35] Zhang H, Jiang G, Yoshihira K, et al. Intelligent workload factoring for a hybrid cloud computing model[C]//2009 Congress on Services-I. IEEE, 2009: 701-708.

[36] Yang J, Chen Z. Cloud computing research and security issues[C]//Computational intelligence and software engineering (CiSE), 2010 international conference on. IEEE, 2010: 1-3.

[37] Bhardwaj S, Jain L, Jain S. Cloud computing: A study of infrastructure as a service (IAAS)[J]. International Journal of engineering and information Technology, 2010, 2(1): 60-63.

[38] Genez T A L, Bittencourt L F, Madeira E R M. Workflow scheduling for SaaS/PaaS cloud providers considering two SLA levels[C]//2012 IEEE Network Operations and Management Symposium. IEEE, 2012: 906-912.

[39] Paraiso F, Haderer N, Merle P, et al. A federated multi-cloud PaaS infrastructure[C]//Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on. IEEE, 2012: 392-399.

[40] Cusumano M. Cloud computing and SaaS as new computing platforms[J]. Communications of the ACM, 2010, 53(4): 27-29.

[41] Sefraoui O, Aissaoui M, Eleuldj M. OpenStack: toward an open-source solution for cloud computing[J]. International Journal of Computer Applications, 2012, 55(3).

[42] Pepple K. Deploying openstack[M]. " O'Reilly Media, Inc.", 2011.

[43] Duan Z Y, Cao Y Z. The Implementation of Cloud Storage System Based on OpenStack Swift[C]//Applied Mechanics and Materials. Trans Tech Publications, 2014, 644: 2981-2984.

[44] Rosado T, Bernardino J. An overview of openstack architecture[C]//Proceedings of the 18th International Database Engineering & Applications Symposium. ACM, 2014: 366-367.

[45] Teixeira J. Developing a cloud computing platform for big data: The openstack nova case[C]//Big Data (Big Data), 2014 IEEE International Conference On. IEEE, 2014: 67-69.

[46]Cui B, Xi T. Security analysis of openstack keystone[C]//Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2015 9th International Conference on. IEEE, 2015: 283-288.

[47] Pereira J, Prata P. Management of virtual machine images in heterogeneous clouds[J]. International Journal of Computational Science and Engineering, 2015, 10(1-2): 113-129.

[48] Nurmi D, Wolski R, Grzegorczyk C, et al. The eucalyptus open-source cloud-computing system[C]//Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on. IEEE, 2009: 124-131.

[49] Varia J. Best practices in architecting cloud applications in the AWS cloud[J]. Cloud Computing: Principles and Paradigms, 2011: 457-490.

[50] Varia J. Migrating your existing applications to the AWS cloud[J]. A Phase-driven Approach to Cloud Migration, 2010.

[51] Waqar A, Raza A, Abbas H. User privacy issues in Eucalyptus: a private cloud computing environment[C]//2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications. IEEE, 2011: 927-932.

[52] Dantas J, Matos R, Araujo J, et al. An availability model for eucalyptus platform: An analysis of warm-standy replication mechanism[C]//2012 IEEE International Conference on Systems, Man, and Cybernetics (SMC). IEEE, 2012: 1664-1669.

[53] Mahmood Z. Cloud computing for enterprise architectures: concepts, principles and approaches[M]//Cloud Computing for Enterprise Architectures. Springer London, 2011: 3-19.

[54] Kumar R, Jain K, Maharwal H, et al. Apache cloudstack: Open source infrastructure as a service cloud computing platform[J]. Proceedings of the International Journal of advancement in Engineering technology, Management and Applied Science, 2014: 111-116.

[55] Muñoz V M, Albor V F, Diaz R G, et al. The integration of cloudstack and occi/opennebula with dirac[C]//Journal of Physics: Conference Series. IOP Publishing, 2012, 396(3): 032075.

[56] Paradowski A, Liu L, Yuan B. Benchmarking the performance of openstack and cloudstack[C]//2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing. IEEE, 2014: 405-412.

[57] Kovári A, Dukan P. KVM & OpenVZ virtualization based IaaS open source cloud virtualization platforms: OpenNode, Proxmox VE[C]//2012 IEEE 10th Jubilee International Symposium on Intelligent Systems and Informatics. IEEE, 2012: 335-339.

[58] Li P. Centralized and decentralized lab approaches based on different virtualization models[J]. Journal of Computing Sciences in Colleges, 2010, 26(2): 263-269.

[59] Bian N, Chen H. A Least Grade Page Replacement Algorithm for Web Cache Optimization[J]. Knowledge Discovery & Data Mining. wkdd. first International Workshop on, 2008:469-472.

[60] Psounis K, Prabhakar B. Efficient randomized Web-cache replacement schemes using samples from past eviction times[J]. IEEE/ACM Transactions on Networking, 2002, 10(4):441-454.

[61] Psounis K, Prabhakar B. Efficient randomized Web-cache replacement schemes using samples from past eviction times[J]. IEEE/ACM Transactions on Networking, 2002, 10(4):441-454.

[62] 石为人, 周彬, 许磊. 普适计算: 人本计算[J]. 计算机应用, 2005, 25(07): 1479-1484.

[63] “Iaas，Paas，SaaS解释”， http://blog.chinaunix.net/uid-14710393-id-4721207.html （2014/12/26）

[64]“云计算的五大特征”http://wenku.baidu.com/link?url=OVDHWS3AZyMn01A7uqZeeGOm

HqKOM8otjzFPQm0Yix2bPEN1ZzuCyo7UoeFTHQ3Z25PiRZv-72m81X2WRW5tI6F-g8JYAX3Zk0iDdJH7jw（2014/03/25）

[65] Chow R, Golle P, Jakobsson M, et al. Controlling data in the cloud: outsourcing computation without outsourcing control[C]//Proceedings of the 2009 ACM workshop on Cloud computing security. ACM, 2009: 85-90

[66] Doelitzscher F, Sulistio A, Reich C, et al. Private cloud for collaboration and e-Learning services: from IaaS to SaaS[J]. Computing, 2011, 91(1): 23-42.

[67] 丘群业. 企业私有云计算基础架构研究与设计[D].华南理工大学,2012.

[68] 李智慧.大型网站技术架构[M].电子工业出版社，2013.16-119

[69] 陈康贤.大型分布式网站架构设计与实践[M] 电子工业出版社，2014.30-331

[70] Karger, David, Lehman, et al. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web[J]. Stoc, 2001(1):654-663.

[71] 杨绍光, 张云勇, 陈清金,等. 基于OpenStack的云计算laaS管理平台研究[J]. 互联网天地, 2013(3):11-15.

[72] 杨江明, 顾宁, 吴筱媛. 基于地址空间转换方法的Undo操作支持[J]. 通信学报, 2006, 27(3):48-56.

[73] 卢丽. 云计算技术的产业现状与挑战性问题[J]. 硅谷, 2011(16):28-28.

[74] 崔春焱. 基于中间件的企业集群系统结构设计[D]. 电子科技大学, 2006.

[75] 李妩可. 移动广播环境中实时嵌套事务并发控制机制关键技术研究[D]. 中南大学, 2009.

[76] 顾宁, 杨江明, 张琦炜. 协同组编辑中基于地址空间转换的一致性维护方法[J]. 计算机学报, 2007, 30(5):763-774.

致谢

研究生生活就这样到了尾声，时间真的过得好快，一转眼三年居然从指缝间悄悄溜走了。在复旦校园三年间的点点滴滴，仿佛还历历在目。犹记得参加实验室网络面试的时候；也记得第一次来复旦复试的时候；还记得暑假拖着行李箱，来复旦求学开始研究生生涯的时候，那么多珍贵的瞬间在脑海里一幕幕闪过，让我顿时感慨万分。

三年之前，我知道努力学习基础知识，三年之后，我知道如何掌握科研的方法；三年之前，我知道要不明白的地方要及时问老师问题，三年之后，我懂得如何在查询大量文献资料的基础上和同研究方向的同学讨论思考；三年之前，我对于专业知识的了解只是局限于肤浅的书本理论知识，三年之后，我从之前的基础之上看到了更为广阔的理论系统。三年的研究生生涯，既是阶段性的结束又是新的启程。

首先我想感谢顾宁老师。我第一次见到顾宁老师是在复旦复试结束之后，觉得顾老师是那么严肃又带着一点亲切友善。在我选修顾老师的CSCW课程的时候，通过论文汇报和提问等环节，让我对于AST，OT算法有了一定的了解。这使得我在之后的深入研究中，更为得心应手。后来在对我的毕业论文的指导中，顾老师的循循善诱以及研究思路的指引，给我提供了不少的灵感，帮助我在每次研究陷入桎梏的时候，能够及时给我建议和意见。同时，在生活中，顾老师也根据同学的科研情况以及工程情况，给了我们足够的生活补贴，使得我们能够完全地投入到研究学习中去。接下来，我想感谢卢暾老师，他在我毕业论文撰写期间，和我进行了不少的探讨，并且给了我不少的专业的意见和思路。也非常感谢丁向华老师，在人机交互课程上，她生动有趣的授课使得我对于在CSCW分支上的社会学部门有所了解。

接着我还想感谢实验室的同学们。我想感谢师兄师姐们，在刚进入实验室的前半段时间中，我就进入了实验室紧张而又有趣的工程研发中，由博士生和上一届的师兄带领，学到了不少工程系统上的实践知识，为之后的实验部署研发奠定了良好的基础。在我配置以及调试遇到问题的时候，师兄的耐心指导和经验性地教导，使得我对于系统框架以及系统架构有了更为深刻的了解。我想感谢实验室的张健豪同学，在我的研究工作中，他给了我很多中肯的意见。我还想感谢我的室友，程金花同学，以及其他的我在复旦中结交的朋友们，在三年的相处中，我们建立了深厚的友谊，在我难过伤心无助的时候，是你们给予了我温暖和关怀，在我高兴取得一定成绩的时候，是你们的鼓励和祝福给我的研究生生涯中更增一抹亮色！最后，再次感谢各位给予我的帮助和温暖，谢谢！

**复旦大学**

**学位论文独创性声明**

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。论文中除特别标注的内容外，不包含任何其他个人或机构已经发表或撰写过的研究成果。对本研究做出重要贡献的个人和集体，均已在论文中作了明确的声明并表示了谢意。本声明的法律结果由本人承担。

作者签名： 日期：

**复旦大学**

**学位论文使用授权声明**

本人完全了解复旦大学有关收藏和利用博士、硕士学位论文的规定，即：学校有权收藏、使用并向国家有关部门或机构送交论文的印刷本和电子版本；允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其它复制手段保存论文。涉密学位论文在解密后遵守此规定。

作者签名： 　 导师签名： 日期：