## **摘要**

自云存储这个概念诞生以来，短短10年内不同厂家纷纷推出了自己的云存储系统产品，这些系统形态各异，接口形式多样。接口的不统一带来了服务供应商绑定和数据绑定问题，影响到产业的长远发展。

本次课题任务参考了SNIA网络存储工业协会提出的CDMI（Cloud Data Management Interface）云数据管理接口标准，结合实际使用中遇到的问题做出一些设计上的修正，最终通过Rails框架和MongoDB数据库的存储支持实现了一个基于CDMI标准的云存储系统接口，可以实现类似文件系统中文件夹、文件、引用的创建、修改、读取、删除操作，解决了云存储系统中数据的基本管理问题，给后续开发者提供了一个基石。

关键词：云存储，管理接口，对象存储，键-值存储

## **Abstract**

Many companies have released their own cloud storage products since the first appearance of cloud storage, but those products have many differences between them, there is a wide diversity of their application interface(API). It bring the serivce vendor binding and data binding issues, which may affect the long-term development of cloud storage, because those APIs do not comply with a same standard.

This project design and realize a cloud storage data management interface which base on SNIA's Cloud Data Management Interface(CDMI) using Rails and MonogoDB, and this interface has been modified to make it more useful. Create, read, update, delete object is supported by this interface, it is similar to file system. This project can be use to manage data in the cloud storage system, and secondary development is welcome.

**Keywords** : Cloud Storge, Management Interface, Object-Based Storage, Key-Value, Web

目录

[摘要 I](#_Toc327202957)

[Abstract II](#_Toc327202958)

[1 绪言 1](#_Toc327202959)

[1.1 课题背景 1](#_Toc327202960)

[1.2 课题研究的目的和意义 2](#_Toc327202961)

[1.3 国内外概况 2](#_Toc327202962)

[2 接口设计方案的研究 5](#_Toc327202963)

[2.1 CDMI的特点及相关规格和概念 5](#_Toc327202964)

[2.2 接口实现的关键技术原理 9](#_Toc327202965)

[3 接口的设计 13](#_Toc327202966)

[3.1 对象的设计 13](#_Toc327202967)

[3.2 数据模型的设计 21](#_Toc327202968)

[3.3 URI路径部分的设计 25](#_Toc327202969)

[4 接口的实现 28](#_Toc327202970)

[4.1 接口开发环境 28](#_Toc327202971)

[4.2 在数据库中存储树形结构的实现 30](#_Toc327202972)

[4.3 对象ID的实现 30](#_Toc327202973)

[4.4 Container类型包含子对象的实现 30](#_Toc327202974)

[4.5 四个主要处理模块 31](#_Toc327202975)

[4.6 实际运行 36](#_Toc327202976)

[4.7 遗留问题 41](#_Toc327202977)

[5 总结与展望 42](#_Toc327202978)

[致谢 43](#_Toc327202979)

[参考文献 44](#_Toc327202980)

## **绪言**

### 课题背景

进入网络时代以来，个人信息迅速膨胀，越来越多的用户需要更长时间的保留那些具有价值的数据，而这些数据通常是分散的，存在于各种终端设备、手机、电子邮件、数码相机以及日常访问的网站上。那么如何才能更好的保存与访问这些信息就成为了一个问题。在访问方面，人们首先想到的方案是将数据进行复制，确保每一个终端设备上都有原信息的一个副本，这样就能在不同的终端上访问到相同的个人信息；保存方面，通常采取在异地保存一个原信息的副本的方式，以利于数据得到长久而安全的保存。这两种方式已经在各个行业中得到了广泛的应用，但他们存在以下缺点：

1. 投入成本和维护成本双高：在建设备份系统之初，个人和企业需要购买大量的硬件和软件设备。这在备份之初就无形提高了建设备份系统的门槛。然而在用户的使用过程中通常备份设备处于闲置状态，这也增加了备份的维护成本。
2. 访问区域受限：用户只能通过专门的网络和设备对备份数据进行访问，无法实现随时随地的数据存取。

云存储的出现能很好的解决以上问题。云存储是基于网络的一种线上存储方式，也被称为数据存储即服务（Data Storage as a Service，DaaS）。不同于本地存储，云存储中客户的数据被保存于云存储服务提供商（Cloud Storage Service Provider，以下简称CSP）的虚拟存储空间中，用户并不需要在自己的工作室里安装实体的储存装置。与传统存储方式相比，云存储具有以下优点：

1. 可扩展性和较低的成本：将大部分数据迁移到云存储上去后，所有的升级维护任务都是由CSP来完成，节约了企业存储系统管理员上的成本压力。还有云存储服务强大的可扩展性，当企业用户发展壮大后，如果发现自己先前的存储空间不足，那么云存储服务则可以很方便的在原有基础上扩展服务空间，满足需求。
2. 随时随地的访问数据：云存储通过HTTP协议，将存储数据通过URI进行定址。用户只要连接网络就可以在任何地点访问处于云端的数据。
3. 数据的高可靠性：CSP通常使用在地理上分布的数据中心保存数据及其冗余副本，在发生区域性灾难时增加了保险系数。

云存储产品的逐步兴起在解决这些问题的同时也带来了新的问题。这些云存储系统形态各异，接口形式多样。接口的不统一主要带来以下问题：

1. 导致用户面临服务提供商绑定问题。用户在不同的CSP之间迁移数据时遇到多重障碍。数据模型、服务功能和操作接口的差异使用户在选择云存储服务时存在一定风险。
2. 影响到产业的长远发展。受接口不一的影响，围绕数据管理服务接口的特定语言编程包和第三方管理软件的开发需要做多种适配，造成行业发展力量的过度分散，并最终会影响到整体市场的增长动力。

标准化协会SNIA（Storage Networking Industry Association）下属的Cloud Storage Technical Working Group工作组致力于云存储的标准化工作。SNIA的云存储模型包括了块、文件、对象和表等层面，考虑到现有标准和产业发展现状，SNIA选择对象作为突破口，并于2010年4月推出了云数据管理接口（Cloud Data Management Interface，CDMI）规范[1]。

### 课题研究的目的和意义

本课题将深入研究学习CDMI标准，并最终实现一个云存储管理接口，目的在于构建一个通用的访问云存储和管理云存储数据的接口，以及给后续开发者提供一个指导方向，使得后续开发者能够根据这个课题结合CDMI标准进行云存储平台的搭建与完善，让用户数据在不同CSP之间实现无缝迁移成为可能。

### 国内外概况

CDMI的编制由传统存储厂商主导，因此CDMI在设计之初就考虑到和传统存储产品和访问协议的兼容，导致整个CDMI体系比较复杂，目前，只有美国一家名为Mezeo的厂商声称其推出的一款存储设备“MezeoCloud”采用了CDMI标准。

在产业界，现时比较著名的云存储服务是Amazon的SimpleDB和Microsoft的Windows Azure Table Service。两者都提供了REST风格的接口，其中SimpleDB还提供了SOAP接口,见文献[2,3,4,5]。SimpleDB和Azure的管理接口所提供的操作都是最基础的功能，支持的操作如表1.1所示。

表 1.1 SimpleDB和Azure的管理接口所提供的操作

| 代号 | 功能 | SimpleDB的操作 | Azure的操作 |
| --- | --- | --- | --- |
| a | 列出账户下所有Domain/Table | ListDomains | Query Tables |
| b | 新建一个Domain/Table | CreateDomain | Create Table |
| c | 删除一个Domain/Table | DeleteDomain | Delete Table |
| d | 查询数据 | GetAttributes  Select | Query Entities |
| e | 插入数据 | PutAttributes  BatchPutAttributes | Insert Entity |
| f | 更新数据 | PutAttributes | Update Entity |
| g | 删除数据 | DeleteAttributes  BatchDeleteAttributes | Delete Entity |
| h | 查看元数据 | DomainMetadata | N/A |

b和c是对表的操作，SimpleDB和Azure差别不大。d、e、f和g是对数据的操作，SimpleDB是Key-Value对级别的（即一条数据中的一个键-值对），Azure则是条目级别的（即一条数据）。SimpleDB中GetAttributes 提供基于ID的查询，Select提供SQL-like语句的查询；Azure则使用Query Entities提供了上述两种功能，通过请求参数的不同来区别。SimpleDB提供了对多条数据批量插入的操作BatchPutAttributes。两者都提供条件插入，即在满足一定条件的情况下才执行数据插入。因为数据存储在分布式环境中，同一份数据会有多份拷贝放在不同地方，条件插入的主要功能是保证数据一致性。Azure提供了两种更新数据的操作Update Entity和Merge Entity，区别是前者用一条新的数据完全替换之前的数据，后者仅替换请求中所提供的属性，其他属性保持原来的不变；SimpleDB中通过参数的不同来区别是否删除旧的数据。SimpleDB提供了对多条数据批量删除的操作BatchDeleteAttributes。SimpleDB提供了对七项元数据的查询，Windows Azure则没有提供元数据的查询。SimpleDB的不同API方法使用请求参数Action来区分；Windows Azure 的不同API方法使用URL和请求方式（GET、POST等）来区分。

SimpleDB的操作参数作为HTTP请求参数写在URL中。Azure的操作参数大部分写在请求头部，只有少部分作为HTTP请求参数写在URL中。SimpleDB的所有REST请求都可以用GET方式，但是如果URL长度超过了GET所允许的最大长度，可以使用POST方式，将请求参数放在请求主体部分中发送。Azure的REST请求方式有GET、POST、PUT、DELETE、MERGE等，数据的CRUD 操作分别用POST、GET、PUT、DELETE实现。MERGE不是标准的HTTP请求方式，用作Update操作，作为PUT的补充。例如，Update Entity和Merge Entity都用作更新某一条数据，二者的请求参数、请求格式等都一模一样，唯一不同的就是请求方式：Update Entity使用PUT方式，用一条新的数据完全替换之前的数据；Merge Entity使用MERGE方式，仅替换请求中所提供的属性，其他属性保持原来的不变。

研究方面，中国电子技术标准化研究所的王洁萍、李海波，东北大学软件学院的宋杰、中国移动研究院的杜宇建联合发表了《云数据存储和管理标准化研究》[6]，文章分析了云数据存储和管理面临的挑战，指出云数据存储和管理接口标准化问题，提出云数据存储和管理接口标准体系架构，并对标准体系架构中两类云数据存储接口和三类云数据管理接口分别进行阐述，提出未来的标准化工作建议。还有东北大学软件学院和中国电子技术标准化研究所的韩峰、闫振兴、王洁萍、李海波、宋杰联合发表了《基于Key-Value的云数据管理应用接口标准化研究》[7]，文章分析了基于Key-Value的云数据管理应用接口的标准化问题和面临的挑战，对现有基于Key-Value的云数据管理服务进行了调研，总结出包括数据模型和REST API等方面的异同，并提出未来的标准化工作建议。

可以看出，通用云接口的研究与普及还处在一个探索阶段，更多的是在调研与比较现有产品。产业界的接口规范也是百家争鸣，不同公司的接口功能或多或少都会有不同，接口的不统一导致用户面临服务提供商绑定问题，影响到产业的长远发展。因此，对现有的云数据管理服务接口进行调研，制定统一的接口规范，方便用户选择适合自身需求的云数据管理服务，并支持数据在不同云存储系统间的无缝迁移。统一接口标准的制定将有助于产业在整个发展过程中遵循开放性和标准化的规则实现良性发展。

## **接口设计方案的研究**

本章节先介绍了CDMI所定义的几个关键概念和云存储所涉及到的几个关键技术，然后以CDMI所定义的接口为模型探讨了几个关键技术的原理。

### CDMI的特点及相关规格和概念

Cloud Data Management Interface（CDMI）是一个功能性管理接口，它提供了访问云存储和管理云存储数据的方式，用户可以通过这个接口对云中的数据进行创建、检索、更新、删除操作。通过这个接口用户也可以得知当前访问的云存储系统部署了哪些功能，哪些操作是被允许的。

CDMI所定义的管理接口中提供了5种基本对象，以这5种对象作为基础支持数据存储管理和访问操作，这5中对象是：域对象、容器对象、数据对象、队列对象和能力对象。其中，域对象、容器对象、数据对象和队列对象分别提供对域、容器、数据和队列的CRUD（Create、Read、Update、Delete）操作接口。容器对象可互相嵌套，形成像类似文件系统的树形结构。域对象、容器对象和数据对象之间存在一定的层次关系：域对象下包含容器对象，容器对象下包含数据对象。队列对象是容器对象的一类特殊的子对象，以先进先出的方式提供对数据的访问。能力对象用来描述云数据存储和管理服务提供商能够提供的能力，用户能通过访问能力对象来获知可以进行哪些操作。

CDMI使用支持Put，Get，Post和Delete四种HTTP方法的REST风格接口。

**RESTful Web服务：**

REST是表征状态转移（Representational State Transfer）的简写，是是Roy Fielding博士在2000年他的博士论文中提出来的一种软件架构风格。需要注意的是，REST是设计风格而不是标准，见文献[15,16,17]。CDMI是REST风格的，也可以说它是一个RESTful Web服务，这意味着：

1. CDMI中的每个对象都应该是可标识的，都应该拥有一个明显的ID——在Web中，代表ID的统一概念是：URI。URI构成了一个全局命名空间，使用URI标识对象意味着它们获得了一个唯一、全局的ID。通过URI来指定和访问对象，例如：http:// cloud.example.com /container1,就是一个URI。
2. 对对象的操作包括获取、创建、修改和删除，这些操作对应Http协议提供的Get、Post、Put和Delete方法。其中Get方法具有幂等性（指对同一个URI发送多个相同请求都产生相同的结果），如果你发送了一个Get请求没有得到结果，你可能不知道原因是请求未能到达目的地，还是响应在反馈的途中丢失了。幂等性保证了你可以简单地再发送一次请求来获得结果，无论发送多少次Get请求都不会对云存储系统造成任何影响。与此相对，Post方法就不具有幂等性，因为Post方法通常表示“创建一个新对象”，因而每次发送Post请求都会在云存储系统中创建一个新对象，每次得到的结果都不相同，因此不具有幂等性。
3. 客户端通过操作CDMI对象的表现形式来对对象进行操作。如果客户程序知道如何处理一种特定的数据格式，那就可以与所有提供这种表述格式的资源交互。利用HTTP内容协商，客户程序可以请求一种特定格式的表述，例如：

PUT /MyContainer/MyDataObject.txt HTTP/1.1

Host: cloud.example.com

Accept: application/vnd.org.snia.cdmi.dataobject+json

Content-Type: application/vnd.org.snia.cdmi.dataobject+json

X-CDMI-Specification-Version: 1.0

{

"mimetype" : "text/plain",

"metadata" : {

},

"value" : "Hello CDMI World!"

}

若操作成功执行，请求返回的结果会是SNIA所定义的专有的JSON格式表述的数据对象信息。对象的表现形式在CDMI中规定是SNIA专有的JSON格式，当然在实际使用过程中也可以输出为任何其他的格式，比如YAML和XML。

1. CDMI要求状态要么被放入对象状态中，要么保存在客户端上。或者换句话说，服务器端不能保持除了单次请求之外的，任何与其通信的客户端的通信状态。这样做的最直接的理由就是可伸缩性——如果服务器需要保持客户端状态，那么大量的客户端交互会严重影响服务器的内存可用空间。

**CDMI的对象类型及关系：**

CDMI定义的对象共有5种：Container、DataObject、Domain、Queue、Capability

对象间路径层次关系如图2.1所示。

Root

https://<offering>

Container A

https://<offering>/containerA

DataObject1

https://<offering>/containerA/databoject1

DataObject2

https://<offering>/containerA/databoject2

Container B

https://<offering>/containerB

Capability

https://<offering>/cdmi\_capabilities

Queue

https://<offering>/containerB/queue1

图 2.1 CDMI对象路径层次关系图

根目录下可以包含Container对象，Container对象下可以包含Dataobject对象、Queue对象、Container对象，可以实现Container对象的多层嵌套。Capability对象包含于根目录下的cdmi\_capabilities目录中。Domain对象包含于根目录下的cdmi\_domains目录中。

CDMI所定义对象的类型及对象行为及功能描述如表2.1所述。

表 2.1 CDMI对象及其行为与功能

| 对象类型 | 描述 |
| --- | --- |
| Container | Container对象可以包含子对象，但其本身不能带有用户需要存储的数据值，这和文件系统中的文件夹类似。Container对象可以通过协议连同它包含的子对象一起被输出。 |
| Dataobjects | Dataobjects本身带有用户需要存储的数据值，是CDMI中基本的存储组件，与文件系统中的文件类似，其不能包含子对象 |
| Queue | Queue本身带有用户需要存储的数据值，其不能有包含子对象。Queue是Container的一个特殊子对象，用来在存储和检索时提供先进先出的访问次序。Queue提供了一个简单的可靠的机制，实现了多个写入端向Queue对象写入数据而从一个读出端读出。 |
| Domain | Domain是一种特殊的Container，其可以包含子对象但其本身不能存储用户的数据值。Domain对象间组成的层次结构体现了所有权的管理关系。 |
| Capability | Capability是一种特殊的Container，可以包含子对象但其本身不能存储用户的数据值。Capability对象可以让CDMI客户端检索，让其了解当前的CDMI提供商部署了CDMI哪些功能子集。 |

**对象的引用：**

对象的引用是当前云存储命名空间的一个指向另一个URI的URI。这个引用的概念就像操作系统中的快捷方式，一个引用一旦创建后，云存储系统不保证被引用的URI在以后的使用中总是有效的。

对象的引用在Container中与其他子对象一起被显示。除了Create和Delete操作外，对引用做的其他任何操作都会返回一个302 Found HTTP重定向提示，HTTP头部的Location项中包含了重定向的目标URI，这个URI是在引用创建时就定义好了的。一旦引用被创建后，引用的目标URI就不能被修改了。

为了使操作得以完成，当CDMI的客户端收到一个302 Found重定向提示后，客户端应该重新向HTTP头部Location项中的URI发出请求。

**对象ID：**

每个存储于CDMI系统中的对象都会拥有一个全局上独一无二的ID标志着这个对象，对象的ID在对象被创建时得到。CDMI的对象ID是一个字符串但有其生成和确保其唯一性的规则。每个CDMI提供商都可以产生不与其他提供商相冲突的ID。

每个CDMI系统都应该提供访问存储对象的功能，既可以通过路径URI访问，也可以通过对象ID直接访问。例如：

* http://cloud.example.com/dataobject
* http://cloud.example.com/cdmi\_objectid/AAAAFAAo7EF

第一个例子是以URI访问，第二个例子是以对象ID访问。

**NoSQL与Key-Value：**

NoSQL是Not Only SQL的简写，是对不同于传统的关系型数据库的非关系型数据库管理系统的统称，见文献[18,19,20]。关系型数据库中的表都是存储一些格式化的数据结构，每个元组字段的组成都一样，即使不是每个元组都需要所有的字段，但数据库会为每个元组分配所有的字段，这样的结构可以便于表与表之间进行连接等操作，但从另一个角度来说它也是关系型数据库性能瓶颈的一个因素。而非关系型数据库以键值对存储，它的结构不固定，每一个元组可以有不一样的字段，每个元组可以根据需要增加一些自己的键值对，这样就不会局限于固定的结构，可以减少一些时间和空间的开销。

Key-Value数据库是NoSQL 数据库中使用最为广泛的一类。Key-Value数据库中包含有表，与关系数据库表的概念类似，是包含许多条目的集合。表中的条目是表中的一条数据，相当于关系数据库表中的一行。条目中包含的数据都是以Key-Value键值对的形式出现，每个条目包含了若干键值对，与关系数据模型不同，每个条目包含的键的名称和个数可以不一样。

### 接口实现的关键技术原理

CDMI将对象看作是可以通过URI来访问的独一无二的项目，因此每个对象都应该拥有一个全局唯一且独一无二的URI与ID，例如，Container1的URI是http://cloud.example.com/container1, ID是6sa7f82c，那么这个URI会与这个云存储系统中其他所有对象的URI都不同，ID也会和其余所有对象的ID都不同。CDMI的对象间呈现一个树形层次结构，如何在数据库中存储这个树形结构成为了一个需要解决的问题。

#### 在数据库中存储树形结构

CDMI中通过Container对象和Domain对象的嵌套，形成了一个树形结构，每个对象都可以用从根节点经过层层Container对象和Domain对象到所寻址对象所在的叶子节点的一条路径唯一标志。这条路径其实就是访问所寻址对象的一个URI，这个URI与上文提到的按ID访问对象的URI不同，通常反映了对象所处位置的层次结构。比如云存储系统中的一个URI是http://cloud.example.com/container1/container2/note.txt，访问的对象是note.txt，可以通过这个URI得知根路径下有一个Container类型的名为container1的对象，container1下包含有一个Container类型的名为container2的对象，container2下包含着将要访问的名为note.txt的数据对象。这其实与文件系统中的文件路径类似，因为文件系统中也是一条路径唯一确定一个文件，一个文件只能由一条路径标志。在文件系统中，从根目录到文件所在目录过程中可以嵌套多层文件夹，这文件夹就相当于CDMI中的Container，而文件就相当于Dataobject。于是http://cloud.example.com/container1/container2/note.txt这个URI相当于Linux系统中的一条文件路径“/container1/container2/note.txt”。

因此在云存储系统中，可以把每个对象的反映层次结构的URI字符串作为标志这个对象的Key，存储在Key-Value数据库中，想要找到这个对象只要在数据库中检索这个Key就可以了。按照这个思路，上文中的note.txt对象在数据库中的Key将是“/container1/container2/note.txt”，这里的Key也就相当于关系型数据库中的主键，是全局唯一的。通过检索这个Key找到数据表中的一行，代表着一个对象，一行通常有多个列，表示这个对象具有多个属性，列中的值是这个对象的各个属性值。

采用这样的思路后，数据库中的数据大致上会是这个样子，每一行对应一个CDMI对象，KeyURI相当于关系型数据库中的主键，如表2.2所示。

表 2.2数据库表项设计雏形

| objectID | KeyURI | ParentPath | ItemType | Value |
| --- | --- | --- | --- | --- |
| AAAAFAAo7EE | / | NULL | Container |  |
| AAAAFAAo7EF | /note.txt | / | Dataobject | "Hello World" |
| AAAAFAAo7EG | /containerA | / | Container |  |
| AAAAFAAo7EH | /containerA/containerB | /containerA | Container |  |
| AAAAFAAo7EI | /cdmi\_capabilities | / | Capability |  |
| AAAAFAAo7EJ | /cdmi\_capabilities/domain | /cdmi\_capabilities | Capability |  |
| AAAAFAAo7EK | /cdmi\_domains/MyDomain | /cdmi\_domains | Domain |  |
| AAAAFAAo7EL | /containerA/containerB/q | /containerA/containerB | Queue |  |

注：其中根目录的KeyURI是“/”，它没有父目录。根目录下可以包含子对象，如：/note.txt，因此根目录是一个Container类型。

#### 通过对象ID访问对象

若用户访问http://cloud.example.com/cdmi\_objectid/AAAAFAAo7EF，通过URI第一层的“cdmi\_objectid”字串可以得知这个URI将通过对象ID访问对象，因此按对象ID访问方式合法的URI应该有且只有两层，第一层是固定的“cdmi\_objectid”字串，第2层是对象在数据库中的ID。若URI第一层是“cdmi\_objectid”，但URI却没有2层，例如下述都不是按对象ID访问的合法URI：

* http://cloud.example.com/cdmi\_objectid/AAAAFAAo7EF/45526
* http://cloud.example.com/cdmi\_objectid
* http://cloud.example.com/cdmi\_objectid/

通过URI合法性检测后，截取出URI的ID部分，这里是AAAAFAAo7EF，然后在数据库中搜索有无ID是“AAAAFAAo7EF”的对象，这样就实现按对象ID访问的功能。

另外，“/cdmi\_objectid”是一个特殊路径，只起到判断当前URI是否是ID访问方式的作用，不能把cdmi\_objectid当做根目录下的一个Container类型的子对象。数据库中不存在KeyURI = /cdmi\_objectid的对象，也不存在KeyURI = /cdmi\_objectid/AAAAFAAo7EF之类的对象，也不允许用户在根目录下创建名为cdmi\_objectid的任何类型的对象。

#### Container间互相嵌套和包含子对象

通过判断当前所请求对象的上一级对象是否是Container类型完成。比如：一个HTTP请求发来，URI是“http://cloud.example.com/containerA/new.jpg”，HTTP方法是PUT，要求创建一个KeyURI为“/containerA/new.jpg”的数据对象，这时应根据所请求的URI判断这个KeyURI的上一级的对象是不是Container类型。这里的上一级对象是“/containerA”，通过检索数据库，发现其是一个Container类型的对象，可以拥有子对象，这里的子对象可以是Container、Dataobject、Queue类型的。于是，在数据库中加入一个KeyURI为“/containerA/new.jpg”的数据对象，并在containerA子对象中加入new.jpg一项。如果所请求URI的上一级对象不存在或上一级对象不是Container类型，例如通过以下两个URI请求创建new.jpg数据对象则会报错。

* http://cloud.example.com/containerF/new.jpg
* http://cloud.example.com/note.txt/new.jpg

#### 对云存储系统中的各个对象进行CRUD操作

将HTTP方法（GET、PUT、POST、DELETE）与CRUD的4种操作做一个映射，通过请求中的HTTP方法确定将要进行哪种操作，见文献[16]。对应关系如表2.3所示。

表 2.3 HTTP方法与CRUD操作的对应关系

|  |  |
| --- | --- |
| HTTP方法 | CRUD操作 |
| GET | Read |
| PUT | Create或Update |
| POST | Create |
| DELETE | Delete |

PUT操作最终确定为Create还是Update，要依所请求的对象是否已经存在以及HTTP请求头部的相关参数而定。

## **接口的设计**

本章节包括这个云数据管理接口设计的3个主要部分：基本对象在概念上的设计、实际编码中数据模型的设计、URI的设计。其中数据模型中与基本对象同名的模型是这个对象的一个编码实现，在程序中表现为一个类。其余的数据模型起到一个辅助作用。

### 对象的设计

本课题设计的云存储系统管理接口包含3种基本对象，Container、Dataobject、Reference。其中，Container与Dataobject对象的表现行为与操作方法和CDMI文档中所描述的一致，但功能字段部分有所删减，只是CDMI所描述的全部字段的一个子集。Reference是这个接口新定义的一个对象，专门用来表示对象的引用，所有的引用都是一个Reference类型的对象。

#### Dataobject对象

Dataobject对象是这个云存储系统中基本的存储组件，与文件系统中的文件类似。每个Dataobject对象在被创建时都预先设定好了一组字段，包括objectURI，objectID，parentURI，Metadata，Mimetype，Value等，有些字段的值是在对象创建时就由系统自动填充好的，如：objectID和parentURI，而另一些则需要由用户填充，如：Value。

Dataobject对象可以通过以下两种形式的URI访问：

* http://cloud.example.com/dataobject
* http://cloud.example.com/cdmi\_objectid/AAAAFAAo7EF

第一种通过对象路径层次定址，第二种通过对象ID直接定址。每个Dataobject对象都拥有一个全局唯一的对象标识符（ID），这个ID在对象的整个生存周期里都维持不变。

通过在Read操作的URI之后加一个“?”字符，然后加上想要读取的Dataobject字段可以只读取个别字段，而不用读取整个对象的所有字段。例如，以下URI将只读取并返回所要读取的Dataobject对象的Value字段值：

* http://cloud.example.com/dataobject?value

当要在一个请求中读取多个单独字段时，各个字段名间用“;”分开，以下例子将只返回Value和Metadata的字段值：

* http://cloud.example.com/dataobject?value;metadata

当读取的个别字段中包含有Value时，可以规定读取Value字段值的范围，以下例子将读取Value字段值的前1001个字节：

* http://cloud.example.com/dataobject?value:0-1000

当读取一个不存在的Dataobject对象时将返回一个错误。如果对象存在，而用户读取的是Value字段值的某个范围，但是这个范围内还没有写入任何数据时，将返回0值。例如Value字段值只有300个字节的长度，而用户读取的是500-800字节，此时将返回0。

Dataobject对象中的metadata字段可以包含任意由用户定义的元数据。

valuerange字段的功能与CDMI中的不同，用于保存在数据库中的对象的value字段值的实际字节范围，非CDMI中的所描述的返回给用户的HTTP响应中value字段值的字节范围。

一个Dataobject对象在数据库中存储有objectName、objectURI、objectID、parentURI、capabilitiesURI、mimetype、metadata、valuerange、value这9个字段的值，除了valuerange字段之外，具体功能和CDMI中的同名字段相同。objectName为新增字段，用于保存对象名。除了删除操作外，其余3种操作的HTTP响应主体中都默认会包含全部字段及其字段值。

##### 创建一个Dataobject对象

在URI所指定的位置创建一个新的Dataobject对象：

PUT <root URI>/<ContainerName>/<DataObjectName>

* <root URI>是云存储系统的根路径
* <ContainerName>是0个或多个已经存在的处于中间层次的Container对象的名称
* <DataObjectName>是将要创建的Dataobject对象的名称

一旦创建成功后，新的Dataobject对象将会获得一个ID，之后也可以通过<root URI>/cdmi\_objectid/<objectID>这种按ID访问的URI形式访问这个对象。

具体的参数见文献[1]中的Dataobject部分。但也有些地方和CDMI不同：

1. HTTP请求头部中如果出现“X-CDMI-NoClobber”字段，并且值设置为"true"，则能保证不会覆盖写一个已经存在的对象，PUT请求会被确定为一个Create操作。如果所请求URI所标志的对象已存在，则会返回一个304错误。
2. HTTP请求主体中只保留mimetype、metadata、reference、value这4个可用字段，具体功能与使用方法和CDMI中的同名字段相同。
3. HTTP响应主体中只保留objectName、objectURI、objectID、parentURI、capabilitiesURI、mimetype、metadata、valuerange、value这9个字段。

##### 读取一个Dataobject对象

读取请求中指定的URI所对应的对象。

GET <root URI>/<ContainerName>/<DataObjectName>

GET <root URI>/<ContainerName>/<DataObjectName>?<fieldname>;<fieldname>;...

GET <root URI>/<ContainerName>/<DataObjectName>?value:<range>;...

GET <root URI>/<ContainerName>/<DataObjectName>?metadata:<prefix>;...

* <root URI>是云存储系统的根路径
* <ContainerName>是0个或多个已经存在的处于中间层次的Container对象的名称
* <DataObjectName>是将要读取的Dataobject对象的名称
* <fieldname>是读取的字段名，对应的字段值将会出现在HTTP响应主体中
* <range>指定读取Value字段值的范围
* <prefix>是用于前缀匹配字符串，将返回Metadata字段值中以这个字符串开头的项

对象也可以通过ID访问，URI形式为<root URI>/cdmi\_objectid/<objectID>

具体的参数见文献[1]中的Dataobject部分。

如果在GET请求中只要求读取个别字段的字段值，那在响应主体中只返回这些字段的字段值。要是要求读取的字段名在对象中不存在，则返回一个400 Bad Request错误。

##### 修改一个Dataobject对象

修改一个URI指定的已存在的对象。

PUT <root URI>/<ContainerName>/<DataObjectName>

PUT <root URI>/<ContainerName>/<DataObjectName>?<fieldname>;<fieldname>;...

PUT <root URI>/<ContainerName>/<DataObjectName>?value:<range>;...

* <root URI>是云存储系统的根路径
* <ContainerName>是0个或多个已经存在的处于中间层次的Container对象的名称
* <DataObjectName>是将要修改的Dataobject对象的名称
* <fieldname>是指定要修改的个别字段名，对应的字段值将会被请求主体中同名字段的值替换。若指定修改某一个字段，但请求主体中却没有出现同名字段，则对象中该字段的值将会被清空。<fieldname>可以是metadata，mimetype，value中的任何一个。若没有指定修改任何一个<fieldname>，对象将要被修改的字段依请求主体中出现的字段而定，但只能修改metadata，mimetype，value这3个字段，主体中出现的其他字段将被忽略。
* <range>指定修改Value字段值的范围。

同样可以通过以ID访问的URI形式修改对象，<root URI>/cdmi\_objectid/<objectID>。修改操作不会改变对象的ID。

具体的参数见文献[1]中的Dataobject部分，但也有和CDMI不同的地方：

1. HTTP请求头部中如果出现“X-CDMI-MustExist”字段，并且值设置为"true"，那么PUT可以确定为一个Update操作，若URI所指定的对象不存在则会返回一个404错误。
2. HTTP请求主体中只保留mimetype、metadata、value这3个可用字段，具体功能与使用方法和CDMI中的同名字段相同。

##### 删除一个Dataobject对象

DELETE <root URI>/<ContainerName>/<DataObjectName>

* <root URI>是云存储系统的根路径
* <ContainerName>是0个或多个已经存在的处于中间层次的Container对象的名称
* <DataObjectName>是将要删除的Dataobject对象的名称

也可以通过ID直接删除对象，<root URI>/cdmi\_objectid/<objectID>

具体的参数见文献[1]中的Dataobject部分。

#### Container对象

Container对象是这个云存储系统中基本的起到分组功能的组件，与文件系统中的文件夹类似。每个Container对象可以拥有0个或多个子对象，Container对象自身也设定了一组字段，如：Metadata。

Container对象可以通过以下两种形式的URI访问：

* http://cloud.example.com/container
* http://cloud.example.com/cdmi\_objectid/AAAAFAAo7EE

第一种通过对象路径层次定址，第二种通过对象ID直接定址。每个Container对象都拥有一个全局唯一的对象标识符（ID），这个ID在对象的整个生存周期里都维持不变。

Container对象间可以互相嵌套：

* http://cloud.example.com/container/sub-container

通过在Read操作的URI之后加一个“?”字符，然后加上想要读取的Container对象的字段可以只读取个别字段。例如，以下URI将只读取并返回所要读取的Container对象的children字段值：

* http://cloud.example.com/container?children

通过在children字段后指定一个范围，可以只读取children字段值的一个范围，这点与读取Dataobject对象的Value字段类似：

* http://cloud.example.com/container?children:0-2

在单个请求里可以一次指定多个要读取的个别字段，字段名之间用“;”分隔。以下请求将只读取children和metadata字段：

* http://cloud.example.com/container?children;metadata

Container对象中的metadata字段可以包含任意由用户定义的元数据。childrenrange字段的功能与CDMI中的不同，用于保存在数据库中的对象的children字段值实际数组下标范围，非CDMI中的所描述的返回给用户的HTTP响应中children字段值的数组下标范围。

一个Container对象在数据库中存储有objectName、objectURI、objectID、parentURI、capabilitiesURI、metadata、childrenrange、children这8个字段的值，除了childrenrange字段之外，具体功能和CDMI中的同名字段相同。objectName为新增字段，用于保存对象名。除了删除操作外，其余3种操作的HTTP响应主体中都默认会包含全部字段及其字段值。

##### 创建一个Container对象

在URI所指定的位置创建一个新的Dataobject对象：

PUT <root URI>/<ContainerName>/<NewContainerName>

* <root URI>是云存储系统的根路径
* <ContainerName>是0个或多个已经存在的处于中间层次的Container对象的名称
* <NewContainerName>是将要创建的Container对象的名称

一旦创建成功后，新的Container对象将会获得一个ID，之后也可以通过<root URI>/cdmi\_objectid/<objectID>这种按ID访问的URI形式访问这个对象。

具体的参数见文献[1]中的Container部分。但也有些地方和CDMI不同：

1. HTTP请求头部中如果出现“X-CDMI-NoClobber”字段，并且值设置为"true"，则能保证不会覆盖写一个已经存在的对象，PUT请求会被确定为一个Create操作。如果所请求URI所标志的对象已存在，则会返回一个304错误。
2. HTTP请求主体中只保留metadata、reference这2个可用字段，具体功能与使用方法和CDMI中的同名字段相同。

##### 读取一个Container对象

读取请求中指定的URI所对应的Container对象。

GET <root URI>/<ContainerName>/<TheContainerName>

GET <root URI>/<ContainerName>/<TheContainerName>?<fieldname>;<fieldname>;...

GET <root URI>/<ContainerName>/<TheContainerName>?children:<range>;...

GET <root URI>/<ContainerName>/<TheContainerName>?metadata:<prefix>;...

* <root URI>是云存储系统的根路径
* <ContainerName>是0个或多个已经存在的处于中间层次的Container对象的名称
* <TheContainerName>>是将要读取的Container对象的名称
* <fieldname>是读取的字段名，对应的字段值将会出现在HTTP响应主体中
* <range>指定读取children.字段值数组下标的范围
* <prefix>是用于前缀匹配字符串，将返回Metadata字段值中以这个字符串开头的项

对象也可以通过ID访问，URI形式为<root URI>/cdmi\_objectid/<objectID>

具体的参数见文献[1]中的Container部分。

如果在GET请求中只要求读取个别字段的字段值，那在响应主体中只返回这些字段的字段值。要是要求读取的字段名在对象中不存在，则返回一个400 Bad Request错误。

##### 修改一个Container对象

PUT <root URI>/<ContainerName>/<TheContainerName>

PUT <root URI>/<ContainerName>/<TheContainerName>?metadata

* <root URI>是云存储系统的根路径
* <ContainerName>是0个或多个已经存在的处于中间层次的Container对象的名称
* <TheContainerName>是将要修改的Container对象的名称
* 在“?”后可以指定要修改的个别字段名，这个字段名只能是metadata，对应的metadata字段值将会被请求主体中metadata字段的值替换。若指定修改metadata，但请求主体中却没有出现metadata字段，则对象中metadata字段的值将会被清空。若没有指定修改metadata，对象将要被修改的字段依请求主体中出现的字段而定，但只能修改metadata这个字段，主体中出现的其他字段将被忽略。

同样可以通过以ID访问的URI形式修改对象，<root URI>/cdmi\_objectid/<objectID>。修改操作不会改变对象的ID。

具体的参数见文献[1]中的Container部分。和CDMI不同的地方：

1. HTTP请求头部中如果出现“X-CDMI-MustExist”字段，并且值设置为"true"，那么PUT可以确定为一个Update操作，若URI所指定的对象不存在则会返回一个404错误。
2. HTTP请求主体中只保留metadata这1个可用字段，具体功能与使用方法和CDMI中的同名字段相同。

##### 删除一个Container对象

DELETE <root URI>/<ContainerName>/<TheContainerName>

* <root URI>是云存储系统的根路径
* <ContainerName>是0个或多个已经存在的处于中间层次的Container对象的名称
* <TheContainerName>是将要删除的Container对象的名称
* 若这个Container对象包含有子对象，该删除操作将会返回一个409错误

也可以通过ID直接删除对象，<root URI>/cdmi\_objectid/<objectID>

具体的参数见文献[1]中的Container部分。

#### Reference对象

在创建Dataobject或Container类型的对象时，如果在请求主体中出现“reference”字段标明要创建一个引用时，所创建的对象将会自动成为一个Reference类型的对象。Reference类型的对象本身只能被创建或删除，一旦创建后就不能被修改。若对一个Reference类型的对象进行读取或修改操作，将会在HTTP响应头部的“Location”字段值中返回这个Reference对象所指向的URI，并设置302状态码，HTTP响应主体为空。

在创建引用时，不检查所指向的URI是否存在一个有效的对象。

一旦创建成功后，新的Reference对象将会获得一个ID，之后也可以通过<root URI>/cdmi\_objectid/<objectID>这种按ID访问的URI形式访问这个对象。

### 数据模型的设计

这里的数据模型特指MVC模式中的M（Model）,在具体编码上可以用定义类来实现。为了便于理解，以下将以关系型数据库中的概念“表”代指NoSQL中的相似概念“Collection”。

这个接口将设计有4个主要数据模型，分别是Item、Container、Dataobject、Reference，分别是同名的对象概念的实现。Item是新定义的一个数据模型及对象，便于存储和统一管理对象，下文会做具体解释。

这个接口中所有类型的对象将存储在同一个数据库的同一张表中，如果不进行封装直接存储对象，这样表中的每一行不是一个Container就是Dataobject或Reference对象。同一张表中存储的对象类型不完全一致容易造成管理混乱，于是就需要做一个封装处理，把整个需要存储的对象当做一组普通的数据存储到一个统一对象的数据域中。表中直接存储的是这个统一对象，这个统一对象应该有一个字段指出其中封装的是何种类型的对象，便于解封还原出原本的对象。

按照这种思路，Container、Dataobject、Reference这3种对象不能直接作为表中的一行存储，而是整个嵌入到一个Item对象中做一个封装，表中直接存储的是一个个Item对象，每个对象就是表中的一行。Container、Dataobject、Reference这3种类型对象必须要被嵌入到一个Item对象中，一个Item对象也必须嵌有一个且唯一一个Container、Dataobject、Reference其中一个类型的对象。

Item对象中必须定义有一个itemType字段，指示这个Item对象中封装的是何种类型的对象，取值必须是“Container”、“Dataobject”、“Reference”中的一个。

Item对象的数据空间将会是图3.1所示3种可能情况的其中一种。

Item对象的数据空间

Container对象的数据空间

Item对象的数据空间

Dataobject对象的数据空间

 Item对象的数据空间

Reference对象的数据空间

#### Item数据模型

图 3.1 Item对象数据空间的3种可能情况

Item数据模型应包含四个字段：\_id、path、itemType、数据域。字段具体的作用与存储的数据类型如表3.1所示。

表 3.1 Item数据模型

| 字段名 | 存储的数据类型 | 作用 |
| --- | --- | --- |
| \_id | JSON String | Item对象的全局ID，该ID与同处一张数据表中的其他Item对象都不同 |
| path | JSON String | path类似于关系数据库中的主键，存储路径层次形式的URI，  如："/Container1/Container2/new.txt" |
| itemType | JSON String | itemType指示这个Item对象中封装的是何种类型的对象，取值必须是"Container"、"Dataobject"、"Reference"中的一个。 |
| 数据域 | JSON Object | 数据域用于存储所嵌入的整个对象 |

数据表中存储的情形如表3.2所示，每一行都是一个Item对象，Item对象的数据域里封装了一个或Container或Dataobject或Reference类型的对象：

表 3.2数据表存储情形示例

| \_id | path | itemType | 数据域 |
| --- | --- | --- | --- |
| 1 | "/" | "Container" | < path = "/"的Container类型的对象> |
| 2 | "/Container1" | "Container" | < path = "/Container1"的Container类型的对象> |
| 3 | "/Container1/new.txt" | "Dataobject" | < path = "/Container1/new.txt "的Dataobject类型的对象> |

#### Dataobject数据模型

Dataobject数据模型包含objectName、objectURI、objectID、parentURI、capabilitiesURI、metadata、mimetype、value、valuerange这个9个字段。字段功能与存储的数据类型如表3.3所示。

表 3.3 Dataobject数据模型

| 字段名 | 存储的数据类型 | 作用 |
| --- | --- | --- |
| objectName | JSON String | 存储Dataobject对象的名称，如："note.txt" |
| objectURI | JSON String | 存储Dataobject对象的路径层次形式URI |
| objectID | JSON String | 存储Dataobject对象的全局ID，与所嵌入的Item对象的\_id字段值相同 |
| parentURI | JSON String | 存储Dataobject对象的父对象的路径层次形式URI |
| capabilitiesURI | JSON String | 存储Dataobject对象所对应的能力对象的URI |
| metadata | JSON Object | 存储Dataobject对象的元数据 |
| mimetype | JSON String | 存储Dataobject对象的mime类型字符串 |
| value | JSON String | 存储Dataobject对象的value字段值，具体为以JSON格式编码的数据。 |
| valuerange | JSON String | 存储Dataobject对象的valuerange字段值，具体为表示value字段值的字节范围的一个字符串 |

#### Container数据模型

Container数据模型包含objectName、objectURI、objectID、parentURI、capabilitiesURI、metadata、childrenrange、children、subcontainers\_item\_ids、subdataobjects\_item\_ids、subreferences\_item\_ids这个11字段。字段功能与存储的数据类型如表3.4所示。

表 3.4 Container数据模型

| 字段名 | 存储的数据类型 | 作用 |
| --- | --- | --- |
| objectName | JSON String | 存储Container对象的名称，如："container1" |
| objectURI | JSON String | 存储Container对象的路径层次形式URI |
| objectID | JSON String | 存储Container对象的全局ID，与所嵌入的Item对象的\_id字段值相同 |
| parentURI | JSON String | 存储Container对象的父对象的路径层次形式URI |
| capabilitiesURI | JSON String | 存储Container对象所对应的能力对象的URI |
| metadata | JSON Object | 存储Container对象的元数据 |
| childrenrange | JSON String | 存储Container对象的childrenrange字段值，具体为表示children字段值的数组下标范围的一个字符串 |
| children | JSON Array | 存储Container对象的children字段值，具体为Container对象包含的子对象的对象名的一个数组，如果子对象是Container类型，则应该在对象名之后加一个"/"字符 |
| subcontainers\_item\_ids | JSON Array | 存储Container类型的子对象的objectID，与那个子对象所嵌入的Item对象的\_id字段值相同 |
| subdataobjects\_item\_ids | JSON Array | 存储Dataobject类型的子对象的objectID，与那个子对象所嵌入的Item对象的\_id字段值相同 |
| subreferences\_item\_ids | JSON Array | 存储Reference类型的子对象所的objectID，与那个子对象所嵌入的Item对象的\_id字段值相同 |

#### Reference数据模型

Reference数据模型包含objectName、objectURI、objectID、parentURI、referPath这个5字段。字段功能与存储的数据类型如表3.5所示。

表 3.5 Reference数据模型

| 字段名 | 存储的数据类型 | 作用 |
| --- | --- | --- |
| objectName | JSON String | 存储Reference对象的名称 |
| objectURI | JSON String | 存储Reference对象的路径层次形式URI |
| objectID | JSON String | 存储Reference对象的全局ID，与所嵌入的Item对象的\_id字段值相同 |
| parentURI | JSON String | 存储Reference对象的父对象的路径层次形式URI |
| referPath | JSON String | 存储Reference对象引用所指向的URI |

### URI路径部分的设计

URI起着唯一定位一个对象的作用，意义十分重要。本次课题所设计的接口对于URI有一定的要求。主要要求包括以下2个方面：URI中不能出现不能识别的字符，即非法字符、对对象名中能使用的字符做了一定限制。本节最后还对哪些URI能进行哪些操作进行了一个讨论。

#### 有效字符

可以出现在URI中路径部分的有效字符：

* ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-\_,.

'.'若出现，只能在URI的最后一级是Dataobject和Reference对象名的URI中出现，标志着格式名的开始，不能在上层的任何一级目录名中出现。例如：

1. 有效：/containerA/containerB/dataobject.txt *对象名 "dataobject" 格式名 ".txt"*
2. 有效：/containerA/containerB/data,object *对象名 "data,object" 格式名 nil*
3. 有效：/containerA/containerB/2009-6.4db *对象名 "2009-6" 格式名 ".4db"*
4. 无效：/containerA/contai.nerB/2009-6.4db *对象名 "2009-6" 格式名 ".4db"*
5. 有效：/containerA/containerB/\_dataobject *对象名 "\_dataobject" 格式名 nil*
6. 无效：/contai.nerA/containerB/\_dataobject *对象名 "\_dataobject" 格式名 nil*
7. 对象名无效：/containerA/containerB/\_da.tao.bject *对象名 "\_da.tao" 格式名 ".bject"*
8. 对象名无效：/containerA/containerB/-da,tao.bject *对象名 "-da,tao" 格式名 ".bject"*
9. 格式名无效：/containerA/containerB/\_da,tao.b\_ject *对象名 "\_da,tao" 格式名 ".b\_ject"*

对象名可以由：

* ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-\_,

中的任意字符以任意长度混合组成，但有一点限制：**不能以**'-'**或**','**开头或结尾。**

格式名可以由：

* ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789

中的任意字符以任意长度混合组成

#### 格式名的问题

只有Dataobject和Reference对象的对象名中可以有格式名，格式名通常出现在URI的最后一级。

一般来说，Dataobject对象的路径层次形式URI会像这样：

* /ContainerA/ContainB/ContainD/ContainE/dataobject.format

Dataobject对象也可以没有格式名：

* /ContainerA/dataobject

路径层次形式URI通常起着唯一定址一个对象的作用，这个接口假定格式名字符串是与对象的定址是有关的。

* /ContainerA/dataobject.txt
* /ContainerA/dataobject.jpg

以上两个URI指向的是不同的Dataobject对象。

#### 能否执行请求的操作的问题

首先假定根目录是'/'

* 如果一个URI是以'/'结尾，那通常情况下不允许做GET|PUT|DELETE操作（有例外，下文会讲到），POST则可以，如果URI所指对象是一个Container的话POST操作会在下一级目录创建一个对象。例如：

1. POST /ContainerA/ContainerX/ 会创建一个对象(其所属类别根据Http Request Header中的具体内容确定)，其URI为"/ContainerA/ContainerX/newobject"
2. POST /ContainerA/ContainerX/newobject/ POST操作会被允许，但创建object失败，因为虽然newobject存在，但newobject不是一个Container

* 如果一个URI是以除'/'以外的任何一个有效字符结尾，则任何情况下都不允许做POST操作，GET|PUT|DELETE则可以。例如："/ContainerA/../ContainerX/Object"
* 如果一个URI是以'/'结尾，并且这个URI就是根目录'/'，则允许做GET|PUT|POST操作，对应的过程为Read根目录、Update根目录、在根目录下Create一个对象，例如：

1. POST / 会Create一个object，其URI为"/newobject"

## **接口的实现**

本章节在内容上主要分为四个部分：接口的开发与运行环境的介绍、接口关键技术的实现、接口的主要构成模块、实际运行。

### 接口开发环境

由于这个云存储管理接口将处理http请求，因此这是一个Web服务，属于Web开发的服务器端部分，见文献[21]。这个Web服务向外界暴露出一个能够通过Web进行调用的API，客户端能够用编程的方法通过Web来调用。

Rails是一个基于Ruby语言的Web开发框架，最近的Rails 2版本倾向于使用REST风格的开发方式开发Web服务，这与CDMI的REST设计风格十分合拍，很适合作为本课题的开发框架，见文献[24,25,26,27]。

Rails采用了MVC（Model|View|Controller）设计模式，将这个Web程序分成3个部分：

* Model封装了对象的数据逻辑结构，可以为CDMI的每种基本对象定义一个Model结构，对CDMI每种对象的操作在Rails中就转化为对相应Model的操作。
* View表示返回给用户的数据表现形式，这里是SNIA的JSON格式
* Controller负责将数据送进送出Model，处理用户的HTTP请求，与Model互动后输出View



图4.1描述了这个管理接口处理一个用户请求的完整流程，执行步骤是：

图 4.1 MVC模式处理请求的流程

1. 用户发出HTTP请求，一个请求中若存在主体部分，该部分必须是JSON格式的数据
2. HTTP请求首先传递给一个Controller，负责处理的Controller操作Model数据结构
3. Model存取数据库，处理从数据库得到的数据，将处理后的数据发给Controller
4. Controller将得到的数据输出给View样板，View渲染出最终结果
5. 回传最终结果成品给用户，这里是JSON格式的数据

Controller是最先接触到用户的HTTP请求的，因此Controller需要对整个HTTP请求数据包进行预处理，判断请求中的数据格式是否符合要求，是否有非法字符，操作方法是否被接口支持等等。顺利通过预处理后，就可以确认这个HTTP请求的各个字段都是符合接口的格式规范的，下一步把HTTP请求中的各个参数解析提取出来，然后将参数传递给Model。

通过Model操作CDMI对象，具体来说CDMI对象是存储在数据库中的，用Model对数据库中的数据表进行读取或写入操作，这要依请求中具体的HTTP方法而定，例如：Get方法要对数据库执行一次读取过程，Put则要对数据库执行一次写入过程。

Model整理要将要返回给用户的数据后，就可以把数据喂给View了，View就是表现数据的一个形式。一组结构化的数据可以有多种表现形式，如html，XML，JSON等。CDMI选用的是JSON，JSON（JavaScript Object Notation）是一种轻量级的数据交换格式，易于人阅读和编写，同时也易于机器解析和生成，见文献[28,29]。本课题选用JSON作为View的默认渲染格式，将给用户返回JSON格式的数据。

数据库方面，由于传统的关系型数据库不利于水平扩展，不能应付对数据库高并发读写的需求，而这两个要求都是搭建和运行一个云存储系统所必须的，故而新兴的非关系型数据库成为了云存储系统的首选数据库。

MongoDB是一个面向文档的的非关系数据库，自带了一个出色的分布式文件系统GridFS，可以支持海量的数据存储，它在保证海量数据存储的同时，还具有良好的查询性能，见文献[30]。云存储系统既然是一个存储系统，那就得有处理海量数据的能力，MongoDB很好的解决了这个需求，因此本课题选用MongoDB数据库。另外，MongoDB也有一个ruby的项目MongoMapper，MongoMapper是一个在Ruby环境下访问MongoDB的接口。使用MongoMapper在Rails中操作MongoDB十分方便，这也是选择MongoDB的一个重要的理由。

### 在数据库中存储树形结构的实现

在每个Item对象的path字段里存储这个Item对象中所嵌入的接口基本对象的objectURI，这个objectURI为路径层次形式。然后将path字段设置为数据表的主键。

### 对象ID的实现

MongoDB数据库会为每个对象自动分配一个ID，这里的对象可以是Item、Container、Dataobject、Reference。但由于Container、Dataobject、Reference是嵌入到Item对象中的，每个Item对象都能指代一个或Container或Dataobject或Reference类型的对象，因此MongoDB分配给Container、Dataobject、Reference对象的ID将是多余的，使用Item对象的ID就能访问到这3个对象。

Item对象从MongoDB得到的ID将作为这3个对象的objectID，可以直接用objectID访问这些接口基本对象：

* <root URI>/cdmi\_objectid/<objectID>

### Container类型包含子对象的实现

用户申请在特定URI创建一个对象，不妨假设这个对象的URI是“/a/b/c.txt”，接口的PUT请求处理模块会根据这个URI找到这个对象的父对象的URI，这里是“/a/b”。然后根据父对象的URI在数据库中找有无path字段值为“/a/b”的Item对象，若没有找到，则创建失败，因为不能在一个不存在的目录下创建子对象；若找到了，则进一步判断这个Item对象的itemType字段值是不是Container，若是则创建成功，若不是则创建失败，因为只有Container类型才能包含子对象。

### 四个主要处理模块

这个接口共有4个主要模块组成：预处理模块、PUT请求处理模块、GET请求处理模块、DELETE请求处理模块。一个请求的完整处理最多需要经过2个阶段与2个模块：第一阶段必须要经过的预处理模块，第二阶段的PUT请求处理模块、GET请求处理模块、DELETE请求处理模块中的其中一个模块。图4.2表现了请求的处理流程与几个模块间的关系。

其中，HTTP方法为PUT时，第2阶段将进入PUT请求处理模块；HTTP方法为GET时，第2阶段将进入GET请求处理模块；HTTP方法为DELETE时，第2阶段将进入PUT请求处理模块。

第二阶段结束后，请求处理成功完成，给用户返回相应的HTTP响应。

图 4.2 请求的处理流程与模块间的关系



#### 预处理模块

预处理模块的主要功能是对整个HTTP请求进行检测，测试当前URI是否是根目录、是否以ID访问对象、有无URI格式错误、有无字段名错误、当前的URI能否执行HTTP请求中的HTTP方法等。若没有检测到错误，则认为这个HTTP请求符合接口规范，收集请求中的有用信息，准备进入到下一个处理阶段。处理流程图如图4.3。



图 4.3预处理模块流程图

#### PUT请求处理模块

PUT请求处理模块将处理HTTP方法为PUT的请求。这个模块会得到输入预处理模块分析得到的有用信息，进行第二阶段的处理。PUT对应的是对象的创建或修改操作，处理流程如图4.4。

图 4.4 PUT请求处理模块流程图



#### GET请求处理模块

GET请求处理模块将处理HTTP方法为GET的请求。GET请求对应的是对象的读取操作，处理流程如图4.5。



图 4.5 GET请求处理模块流程图

#### DELETE请求处理模块

DELETE请求处理模块将处理HTTP方法为DELETE的请求，DELETE对应的是对象的删除操作。特别指出的是不能删除一个当前含有子对象的Container类型的对象，另外在删除对象后还要在父对象的子对象列表里删去本对象。处理流程如图4.6。



图 4.6 DELETE请求处理模块流程图

### 实际运行

先在本地启动MongoDB数据库服务：

PS F:\mongodb\bin> .\mongod.exe --dbpath "F:\mongodb\data\db"

启动Rails项目：

PS M:\Devl\Project\Eclipse Workspace\CAPI> ruby .\script\server

此时云管理接口运行在本地3000端口，http://localhost:3000/是云存储系统的根目录。

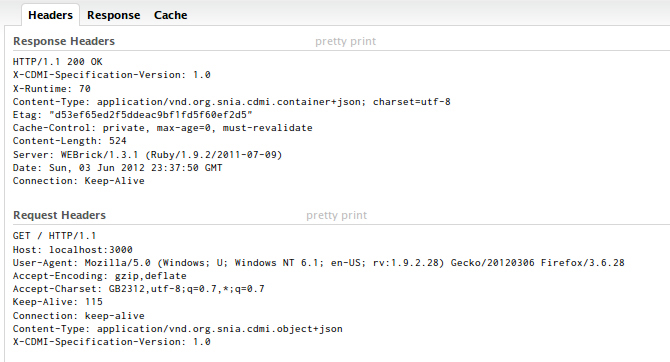
在Firefox地址栏里输入about:config，将network.http.accept.default字段值清空，禁止发送默认Accept头部。之后用Firefox插件RESTClient发送自定义HTTP请求读取根目录。Firebug截获的http请求头部和响应头部信息如图4.7。

图 4.7 Firebug截获的读取根目录的HTTP请求头部与响应头部屏幕截图

从图4.7中可以看到发送的请求头部的关键部分：

GET / HTTP/1.1

Host: localhost:3000

Content-Type: application/vnd.org.snia.cdmi.object+json

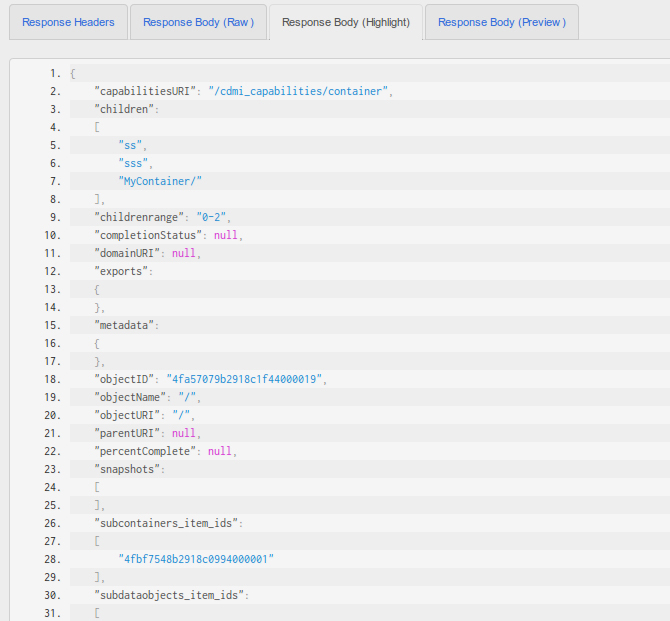
X-CDMI-Specification-Version: 1.0

HTTP响应头部的关键部分：

HTTP/1.1 200 OK

X-CDMI-Specification-Version: 1.0

Content-Type: application/vnd.org.snia.cdmi.container+json

RESTClient插件获得的HTTP响应主体，如图4.8所示。

从图4.8可以看出，根目录下有3个子对象，其中MyContainer是一个Container类型的对象。

图 4.8 RESTClient获得的读取根目录的HTTP响应屏幕截图

现在在MyContainer目录下创建一个名为new.txt的Dataobject类型的对象。Firebug截获的http请求头部和响应头部信息如图4.9。

图4.9中HTTP请求头部的关键部分：

图 4.9 Firebug截获的创建new.txt的HTTP请求头部与响应头部屏幕截图

PUT /MyContainer/new.txt HTTP/1.1

Host: localhost:3000

Accept: application/vnd.org.snia.cdmi.dataobject+json

Content-Type: application/vnd.org.snia.cdmi.dataobject+json

X-CDMI-Specification-Version: 1.0

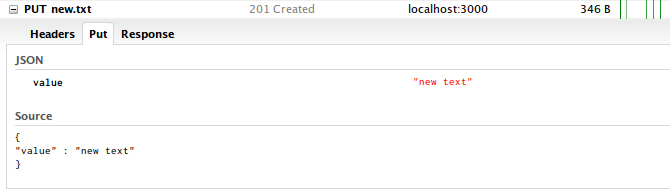
其中的“Content-Type: application/vnd.org.snia.cdmi.dataobject+json”明确指出要创建的对象类型为Dataobject。

HTTP响应头部的关键部分：

HTTP/1.1 201 Created

X-CDMI-Specification-Version: 1.0

Content-Type: application/vnd.org.snia.cdmi.dataobject+json

Firebug截获的http请求主体信息，如图4.10所示。

请求主体中设置要创建的Dataobject对象value字段值为“new text”。

图 4.10 Firebug截获的创建new.txt的HTTP请求主体屏幕截图

RESTClient获得的HTTP响应主体，如图4.11所示。可以看出对象现在的value字段值与HTTP请求主体中设置的值一致。id字段是MongoDB自动分配给Dataobject的一个全局ID，对现有功能来说是完全无用的。objectID是所嵌入Item对象的ID。如果要按ID访问对象，应该使用objectID。

图 4.11 RESTClient获得的创建new.txt的HTTP响应屏幕截图

现在使用new.txt对象的objectID直接访问对象，Firebug截获的http请求头部和响应头部如图4.12。

图4.12中HTTP请求头部的关键部分：

图 4.12 Firebug截获的按ID访问对象的HTTP请求头部与响应头部屏幕截图

GET /cdmi\_objectid/4fcbfd7eb2918c09a4000003 HTTP/1.1

Host: localhost:3000

Accept: application/vnd.org.snia.cdmi.dataobject+json

Content-Type: application/vnd.org.snia.cdmi.object+json

X-CDMI-Specification-Version: 1.0

HTTP响应头部的关键部分：

HTTP/1.1 200 OK

X-CDMI-Specification-Version: 1.0

Content-Type: application/vnd.org.snia.cdmi.dataobject+json

RESTClient获得的HTTP响应主体，如图4.13。可以看出读到的正是刚创建的Dataobject对象。

图 4.13 RESTClient获得的按ID访问对象的HTTP响应屏幕截图

### 遗留问题

虽然在本次设计的接口中用户能设置对象的metadata字段值，但metadata字段值只被当做普通的数据，其在CDMI中所定义的设置个别对象能力属性的功能没有实现，不过在读取对象时能用查询参数metadata:<prefix>实现metadata的前缀匹配功能。

虽然对象存储有capabilityURI字段，并且也能返回正确的capabilityURI，但其实Capability对象的功能没有实现。

POST处理模块没有实现。因为POST操作是在某个Container类型的对象下随机创建一个子对象，CDMI没有进一步详述，不能准确确定新创建的对象类型以及名称，这样的对象无法存储到数据库中。当然，可以通过在POST请求的HTTP请求主体中提供新创建的对象类型以及名称完成通过POST方法创建对象的目的，不过通常情况下PUT方法也可以实现创建对象的目的。

## **总结与展望**

本文主要完成了以下工作：

1. 以SNIA提出的CDMI为参考标准，对云存储中的数据管理接口功能需求进行了系统的分析，提出了具体实现上的思路，完成了基于CDMI的云数据管理接口的设计。
2. 对CDMI中的一些美中不足与遗漏之处做出了一些设计上的改进与补充。增加了新对象Reference，增加了objectName字段，对URI的设计做了一个详细的讨论。
3. 在Rails+MongoDB环境下实现了一个云数据管理接口。对CDMI所描述的功能做出一些取舍，实现了Container和Dataobject对象的创建、读取、修改、删除操作，并增加了Reference对象以实现引用功能。

由于时间上的原因，本课题实现的接口还有很多不足之处，主要包括以下这些方面：

1. POST请求处理模块没有实现。
2. 对象的metadata字段在CDMI中可以设置个别对象的能力属性，这是个很有用的功能，但是本课题实现的接口中没有实现这个功能。

通过本文的研究，实现了一个只支持最基本功能的云数据管理接口。当前，该云数据管理接口还是一个原型版本，将来的工作是在当前系统的基础上持续完善系统的框架和规范，在系统的稳定性、可扩展性、安全性上做更多的开发和实践。相信本文能对这方面的研究提供一条思路、一个基石，对后续的研究人员与开发人员来说具有一定的参考价值。

## **致谢**

在4年的学习生活期间，得到了很多老师和同学的关心和帮助。在此对他们表示衷心感谢！首先感谢我的指导老师王桦老师，她严谨的治学作风、求实的科研精神都是我学习的榜样！同时还要感谢王冲学长给我提供的帮助，我能顺利完成本次课题与他的细致讲解和对我的鼓励密不可分！

感谢华中科技大学IBM俱乐部的全体成员，感谢计算机0806班一起奋斗4年的兄弟姐妹们！

最后感谢我的父母，在漫漫人生道路和求学生涯中他们给了我在学习和工作上莫大的支持和关怀！

## **参考文献**

1. SNIA. ＂CDMI SNIA Architecture v1.0＂. in *http://snia.org/sites/default/files/CDMI\_SNIA\_Architecture\_v1.0.pdf*, April 2010
2. Brad Calder. ＂Windows Azure Storage: a highly available cloud storage service with strong consistency＂. SOSP '11 Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, September 2011
3. Amazon. ＂Dynamo: Amazon’s Highly Available Key-value Store＂. SOSP’07, October 14–17, 2007
4. Amazon. Amazon elastic compute cloud (Amazon EC2). 2009.

*http://aws.amazon.com/ec2/*

1. Amazon Simple Storage Service Developer Guide API Version,

*http://aws.amazon.com/documentation/s3/*

1. 王洁萍，李海波，宋杰，杜宇建.＂云数据存储和管理标准化研究＂.《信息技术与标准化》2011 年第9 期
2. 韩峰，闫振兴，王洁萍，李海波，宋杰.＂基于Key-Value的云数据管理应用接口标准化研究＂.《信息技术与标准化》2011 年第12期
3. 陈 康，郑纬民. ＂云计算:系统实例与研究现状＂. Journal of Software, Vol.20, No.5, May 2009
4. 毛文波，EMC中国实验室. ＂关于云存储标准的讨论＂. 中国互联网大会“云计算产业高峰论坛”.2010年
5. 李锐, 林艳萍, 徐正全, 冯蔚. ＂空间数据存储对象的元数据可伸缩性管理＂. 《计算机应用研究》第28卷 第12期, 2011年12月
6. Andy Edmonds, Thijs Metsch, Eugene Luster.＂An Open, Interoperable Cloud＂，*http://www.infoq.com/articles/open-interoperable-cloud*
7. 徐海峰，毛华坚，肖侬，卢宇彤.＂一种基于Wukong的云存储接口虚拟化服务＂. 华中科技大学学报（自然科学版）2011年6月第39卷 增刊I
8. 郅斌. ＂一种私有云存储系统的设计与实现＂: [硕士学位论文] . 北京邮电大学, 2011年5月
9. 龙志德，徐鹏. ＂基于云计算平台通用容器管理接口的设计与实现＂. 中国科技论文在线2011-12-06
10. Leonard Ricbardson, San Ruby.《RESTful Web Services》. 徐涵, 胡伟. 电子工业出版社, 2008
11. Wikipedia *http://en.wikipedia.org/wiki/Rest*
12. REST和SOAP Web Service的比较，

*http://hi.baidu.com/nansheng2/blog/item/b916cac6756adfc8d0006023.html*

1. NOSQL 背後的共通原則，*http://oss-tw.blogspot.com/2010/04/nosql.html*
2. F. Chang et al., "Bigtable: A Distributed Storage System for Structured Data," in OSDI, 2006
3. [雲端計算] NOSQL背後的共通原則, *http://oss-tw.blogspot.com/2010/04/nosql.html*
4. Cloud API，*http://en.wikipedia.org/wiki/Cloud\_API*
5. Deltacloud，*http://deltacloud.apache.org/*
6. 再谈最终一致，*http://www.infoq.com/cn/news/2009/01/EventuallyConsistent,*
7. Peter coope., 《Ruby从入门到精通》. 仲田. 机械工业出版社, 2009
8. Ruby on Rails Documentation*, http://railsapi.com/doc/v2.3.2/*
9. Building a Platform API on Rails,

*http://blog.gomiso.com/2011/06/27/building-a-platform-api-on-rails/*

1. Using Rails for API-only Apps,

*https://github.com/rails/rails/blob/efd557a60cd976ac17be9e238111a551599caeb5/railties/guides/source/api\_app.textile*

1. JSON, *http://www.json.org/*
2. Posting JSON with headers in Ruby with Net::HTTP,

*http://www.thedwick.com/2012/03/posting-json-with-headers-in-ruby-with-nethttp/*

1. MongoDB, *http://www.mongodb.org*