学校代码： 10246
学　　号： 15212010038

硕 士 学 位 论 文

（专业学位）

云存储中间件关键技术优化设计与实现

Design and Implementation of Backup-Cloud Oriented Optimized Middleware

院　　系：　软件学院

专业学位类别（领域）：　软件工程

姓　　名：　刘武

指 导 教 师：　韩伟力　教授

完 成 日 期：　2017 年 月 日

指导小组成员名单

韩伟力 教授

目录

摘要

近年来，云存储服务因其海量的存储空间、便携的存取方式、可靠的安全特性，逐渐取代传统的数据存储模式，成为人们首选的数据备份方式。国家“核高基”科技重大专项[1]为顺应这一发展趋势，设立了云存储与云备份子课题，构建了一款自主研发的云存储服务平台。该系统的中间件[2]作为服务器与客户端之间的交互枢纽，实现了数据传输、数据压缩、数据加密以及安全会话等基本功能。然而，持续升温的云存储行业在近几年面临着诸多的挑战，衍生出了如安全性、稳定性、可扩展性等问题，用户对云存储平台提出了更高的要求。在这样的环境下，各大云存储供应商纷纷对云存储的关键技术进行革新，而现有的云备份中间件系统仅实现了基础的数据传输、存储功能，这已无法适应当前错综复杂的云环境，因此亟需对其关键技术进行优化，以提升自身的竞争力。

本论文主要通过对当前云存储行业分析，研究云储存服务中的关键技术，同时基于前期的成果，对现有云存储中间件进行优化，以适应云存储行业的发展趋势。本文拟在 Linux 平台上，通过物理隔离和数据加密的方式，保证缓存在中间件中的用户信息和会话记录的安全性；同时，借助大文件分块传输和断点续传技术，解决大文件上传过程中遇到网络中断或服务器死机等故障而引发的数据重传问题；利用本地存储和服务器提供的高级接口，减少数据在上传下载过程中产生的冗余数据，并提供快速上传的功能，以提高数据传输性能；此外，优化并完善平台框架和 API 支持机制，解决不同应用和云存储服务之间适配的问题，以支持更多客户端类型使用云备份与云存储服务。

关键词：

Abstract

With the rapid development of Internet technology, the network has penetrated into every corner of our lives. It needs an efficient way to store the massive resulting data. However, the traditional data storage methods has long been unable to meet the needs of users because of its limited capacity, the use of inconvenient, easy to lose data and other limitations, so cloud storage technology came into being. In just a few years, there are dozens of cloud storage products launch on the market. From abroad iCloud, Google Drive, Dropbox to the domestic Baidu cloud, Ali Cloud, Tencent Cloud, many companies who are optimistic about the emerging markets want to be in this crowd. Continued warming of the cloud storage industry in recent years is facing fierce competition, In order to occupy more market share, the challenge has also spurred the new cloud storage platforms to introduce new technologies to solve the issues derived from cloud storage like security, stability and network problems.

This paper mainly studies the key technology in cloud storage service, and optimizes a self-developed cloud storage middleware. Based on previous results, in the Linux platform, exploiting the local storage and computing power to reduce data upload and download I / O-intensive services, to improve service performance. Through the form of encryption to ensure that user information and data security. At the same time, this paper uses the technology of block upload and breakpoint resume to avoid data retransmission caused by network problem or server problem. In addition, the platform framework and API support mechanisms are optimized and refined to address the adaptation issues between different applications and multiple cloud storage services to support more desktop OS components using cloud backup and cloud storage services.

Keywords：

第一章 引言

1.1 背景介绍

随着互联网技术的飞速发展，网络数据呈现出井喷式的增长，传统的数据存储方式存在太多的局限性，已经无法满足人们的要求。在这样的背景下，云存储服务[3]逐渐成为主流的数据存储方式。因其海量的存储空间、便携的存取方式、可靠的安全特性，越来越受大众的青睐，大型的机构如政府、医院、企业等也逐步将数据转移到云端。

为适应当前数据存储的新趋势，国家“核高基”科技重大专项设立了云存储与云备份子课题，拟构建支持云存储服务的底层框架，研制一款高效的云存储服务平台。至目前为止，经过充分的市场调研和系统的需求分析，设计出了一套完整的体系结构，实现了数据传输、数据压缩、数据加密以及安全会话等基本功能。系统以中间件作为服务器与客户端之间通信的桥梁，有效地分离了客户端云存储工具与基础设施之间的绑定，提高了系统的稳定性，并为系统的跨平台性提供了良好的基础[4]。

然而，大范围普及的云存储服务在近几年迎来了巨大的挑战，除了提供云端数据存储这项基本的功能之外，人们对云存储服务平台的安全性、容灾性、稳定性以及可扩展性投入了更多的关注。在传统的数据存储方式中，数据保存在本地，对用户来说是可管控的，而将数据存储在云端则可能产生如个人数据泄露、商业信息遭到窃取、科研成果外泄等问题，这可能对用户产生不可逆的后果。此外，云存储服务以网络作为基础设施，而网络又时常受限于硬件、软件和地域等多方面的因素，一旦出现网络波动或中断，将会影响数据在网络层的传输，尤其对较大的文件传输来说，遇到网络问题只能进行重传，这将极大地降低用户的体验度。同时，随着云端数据不断地积累，产生了大量重复冗余的数据。一方面这些冗余的数据会占用服务器大量的存储空间，另一方面用户存储在服务器的文件只是一个映射的链接，如果重复上传服务器中已存在的文件，将会占用大量的网络带宽。更进一步，介于移动互联网和智能手机的热度持续走高，人们的娱乐、办公环境逐渐趋向于平台化，针对单一应用的云存储服务已不适应当前平台多元化的格局，实现云端数据在多平台的共享，成为了云存储行业的共识。

目前，国内外成熟的云存储服务供应商为应对这一系列新的挑战，纷纷推出了新的技术或是对原有技术进行了革新，以最大化程度的满足用户需求。我们选取了四个国内外较为知名的云存储服务供应商进行了调研，国

内的百度网盘、360 云盘，国外的 Google Drive、Dropbox，分别对他们的技术特性进行了全面的分析。表 1.1 罗列出了详细的信息。

表 1.1 国内外知名云存储服务平台技术特性分析

	加密算法	数据去冗余	大文件上传	断点续传	平台数
百度网盘	128 位 SSL	全局	20G / 4M	支持	6 种
360 云盘	128 位 SSL	全局	5G / 4M	支持	5 种
Google Drive	128 位 AES SSL/TLS	无	5TB / 字节流	支持	5 种
Dropbox	256 位 SSL/TLS	单一用户	不限 / 8M	支持	7 种

数据加密是一种强有力的安全措施来保护个人信息不被窃取或篡改，根据调研结果，所有的云存储服务供应商都采用了类似的方式对数据在传输过程中及存储在云端时两种状态进行了加密。在传输过程中，百度网盘和 360 云盘使用的是 128 位的 SSL[5]加密技术，Google Drive 和 Dropbox 分别采用了的 128 位/256 位的 AES SSL/TLS[6] 加密技术。数据在云端时，百度网盘和 360 云盘未强制对数据进行加密，而是由用户在上传之前决定是否要将数据以加密的形式存储。反观 Google Drive 和 Dropbox，它们选择了对上传至云端数据进行强制加密，Dropbox 使用的是 256 位的 AES[7]加密方式，而 Google Drive 采取了规模相对较小的 128 位 AES 加密技术。此外，考虑到网络的不稳定因素，各大供应商都对大文件上传技术进行了优化，以避免因网络原因而导致的大文件重传问题。百度云盘支持单个文件上传的大小上限为 20G，360 云盘为 5G，Google Drive 为 5TB，而 Dropbox 不限制单个文件的大小。在处理大文件上传的方案上，它们都将文件以固定大小的文件流形式，通过设定偏移量来对大文件进行分块传输，这样做的好处是遇到网络故障时，只需记录已上传数据流的偏移量，重传时可以根据偏移量进行文件续传而无需从头开始。针对大量数据冗余的问题，国内外的云存储服务平台选择了不一样的策略。减少数据冗余除了能为服务器节省大量的存储空间外，还能为用户提供一种新的功能，这种功能称之为“秒传技术”。所谓的“秒传技术”指的是当服务器存在当前用户待上传的文件时（通过文件的 MD5 值进行校验[7]），不会真实地上传该文件，而是会将这个文件的链接拷贝到用户的网盘中，这样能有效地解决冗余数据重复上传的问题。百度网盘和 360 云盘都支持全局的重复文件校验，为用户提供了“秒传”这项功能。Dropbox 采取的是一种相对折中的方式，它不会在全局文件系统中校验该文件是否存在，只针对单一用户进行去重处理，这样能在很大程度上地减少由去重而导致的数据安全问题，如侧信道攻击[8] 。Google Drive 对待“秒传技术”极其谨慎，尽管这样做能够大幅度地提升上传的效率，但出于安全性方面的考虑，Google Drive 未在云盘

中使用该技术。同时，这些成熟的云存储供应商都会针对主流的操作系统和平台，提供不同类型的客户端，这样使得用户可以跨平台地使用云存储服务。常见的平台有 Windows、Mac、Android、iPhone、Web，以上四个云存储供应商都提供了这些平台的客户端，其中百度云盘还支持 Windows Phone，Dropbox 是唯一支持 Linux 和 BlackBerry 智能手机的平台。

1.2 研究内容及意义

云存储关键技术作为产品的核心竞争力，已成为了各大供应商的研究的重点。但是，由国家“核高基”科技重大专项自主研制的云存储服务平台还停留在提供基础的云端数据存储服务上，而各大云存储服务器平台的关键技术都是不对外公开的，因此只能通过对现有的中间件系统进行全面分析，同时借鉴业内成熟产品的技术特点，在中间件的关键技术上进行自主研发，以符合当前云存储行业的发展趋势。

此外，尽管成熟的云存储服务产品在市场上取得了很大的成功，然而仍存在一些问题。在功能上，经过系统的调研，以上四大云存储供应商都没有提供本地缓存的功能，也就是说它们只做到了服务器端的去重，当服务器中存在用户待上传的文件时，可以通过在服务器端拷贝文件的方式实现“秒传”功能。然而重复地从服务器端下载同一文件也是一种常见的用户行为，如果在本地存储用户下载的文件，那么我们也可以利用本地缓存技术，节省下载时所需的网络带宽，从而实现快速下载功能。在安全性上，成熟云存储平台也存在一些漏洞，有些漏洞被黑客利用后对用户利益造成了严重的损伤，如百度云 SDK 虫洞漏洞[8] ，安装了百度云的 Android 手机遭到攻击后能被黑客任意操作。再如 Dropbox CVE-2014-8889 的漏洞[9]，攻击者利用该漏洞可未经用户同意直接把应用程序和 Dropbox 账户连接。只要用户安装的应用程序使用了含有漏洞的 Dropbox SDK，则其敏感信息就可能被攻击者窃取。因此云存储平台的安全性是至关重要重要的，本文将对原有中间件系统的安全性进行全面优化。基于以上云存储平台都支持多种类型的客户端，本文拟设计一款基于 Firefox 的应用扩展，支持用户以浏览器插件的方式使用云储存服务，相较于常见的 Web 版应用，插件更加轻量级，同时还支持用户将网页上浏览的任何形式的资源随时上传到服务器。

本论文主要研究的内容如下：

1. 持久会话管理：云存储中间件在客户端和服务器之间工作，多个客户端的命令通过中间件提交给服务器。用户的命令会以会话的形式保存在中间件，当中间件系统出现异常时（如网络中

断、中间件死机），会在下次重启中间件时自动恢复执行未完成的会话。此外，对会话信息进行了物理隔离，并且以加密的形式存储在数据库中，以提高用户信息的安全性。

2. 大文件上传：利用服务器提供的高级功能，实现了大文件分块上传功能，在受到网络状况受到影响的情况下，若文件上传中断，会自动启用断点续传功能，避免数据重传。
3. 本地缓存：利用现代计算机普遍较大的存储空间以及高性能的计算能力，实现了本地缓存管理，进一步减少了系统 I/O 操作，提高了中间件性能。同时，利用服务器提供的个人存储在云端的数据信息，实现了针对单一用户的“秒传”功能。
4. 多应用适配：原有的中间件系统只支持单一的 CLI 客户端，为实现对不同客户端请求的适配，支持更多的客户端类型，对系统的跨平台性做了优化。此外，开发了一款基于 Firefox 浏览器的扩展，用户可以利用扩展使用云存储服务，并提供了 Java 平台的软件开发包（SDK）[9]，Java 开发者可以利用该开发包提供的应用接口 API 来集成我们的云存储服务。

1.3 论文组织结构

本文将分七个章节对论文进行阐述。第一章介绍本文研究内容的背景以及相关的研究工作。第二章介绍本文在研究和实现过程中使用到的核心技术，包括 AES 加密算法、SQLite 数据库、断点续传技术、WebSocket、Java SDK、LRU 替换策略以及 WebExtension。第三章将分析原有云备份中间件的不足，指出本文需研究和实现的内容。第四章详细阐述云备份中间件系统的整体设计，包括优化后的系统框架、类图设计、流程设计以及接口设计。第五章介绍云备份中间件实现的细节。第六章将对云备份中间件优化后的结果进行评估，以实际的测试结果作为评估的标准。第七章对全文进行总结，并对未来进行展望。

第二章 核心技术

2.1 SQLite

SQLite 是 D.Richard Hipp 用 C 语言编写的开源嵌入式数据库引擎。它是一款轻量级的关系型数据库，绝大多数主流的操作系统上都能够运行 SQLite，同时它为大多数编程语言，如 C#、PHP、Java 等提供了编程接口。此外，它还支持大多数的 SQL92 标准，且源代码不受版权限制。目前，由于其占用的内存低、性能较好以及零成本管理的特点，被广泛地应用于嵌入式应用的开发中，像 Android、IOS 等平台都为开发人员提供了内置 SQLite 数据库。

在存储容量方面，SQLite 虽然是一款轻量级的数据库，但是最高可支持高达 2TB 大小的数据存储，并且每个数据库都是以单个文件的形式存在的，数据都是以 B-Tree 的数据结构形式存储在磁盘上。

在事务处理方面，SQLite 支持多个进程可以同时读取同一个数据，但只有一个进程可以执行写操作。通过数据库级上的独占性和共享锁机制，使得某个进程或线程想要往数据库写入数据之前，必须获得独占锁，而其他进程则无法执行写操作。

在数据类型方面，SQLite 支持 NULL、INTEGER、REAL、TEXT 和 BLOB 数据类型，分别代表空值、整型值、浮点值、字符串文本、二进制对象。同时，SQLite 采用动态数据类型的机制，当某个值插入到数据库时，首先会对它的类型进行检验，如果该类型与关联的列不匹配，SQLite 则会尝试将其自动转换为对应列的类型，如果不能转换，则将该值将作为本身的类型。

2.2 断点续传

断点续传技术是一种结合本地存储和网络存储的技术，主要用来解决网络失效时的数据丢失问题。具体来说是在上传文件时，将文件以固定的大小，划分为若干部分，或以固定长度的字节流的形式上，对每部分都单独采用一个线程进行上传。服务器端监听数据传输请求并接收数据，记录数据传输进程。如果遇到网络故障，则可通过捕获网络异常的形式，将此时的断点信息记录到数据库中。等待网络恢复正常后，根据断点所记录的已上传的信息续传。通过此项技术，可以帮助用户在遇到网络故障时，节省因重传而浪费的大量时间，提升数据传输的效率。在云存储行业，网络作为重要的基础设施，通常会受到各种因素的影响，因此断点续传技术成为了提升数据传输性能必不可少的一个环节。

断点续传是HTTP 1.1协议的一部分，只要利用了HTTP协议[1]中的Range、Accept-Ranges以及Content-Ranges字段来和服务器端交互，就可以实现文件下载的断点续传。

Range字段用户客户端到服务器的请求报文中，通过该字段可以指定传输文件的某一段，典型格式如下：

Range : bytes=0-1024 //第0-1024个字节范围的内容

Accept-Ranges用于服务器端给客户端的应答，通过该字段可以获取服务器是否支持断点续传的信息，典型格式如下：

Accept-Ranges: bytes　//支持以bytes为单位进行传输。

Accept-Ranges: none　//不支持以bytes为单位进行传输。

Content-Ranges用于服务器到客户端的应答，通过该字段可以获取传输的资源文件的具体范围以及资源的总大小，典型格式如下：

Content-Ranges: bytes 0-1024/5645 //大小为5645字节的前1024字节。

请求报文如下所示：

```
GET /test.docx HTTP/1.1
Connection: close
Host: 10.131.1.63
Range: bytes=0-1024
```

响应报文如下所示：

```
HTTP/1.1 200 OK
Content-Length: 256823
Content-Type: text/html
Content-Range: bytes 0-1024/256823
```

在断点续传过程中，首先需要将上传的文件拆分为固定大小的部分，随后逐一发送PUT请求到服务器。请求报文中需要包含Range字段，表示当前请求上传数据的字节范围。一旦发生网络中断，下次连接时首先发送一个GET请求到服务器，在请求的header中放一个Content-Range字段，通过服务器返回的信息，从 Content-Range 字段中获取已上传的字节数，之后再将剩余字节的上传至服务器。

2.3 WebSocket

WebSocket主要用于 Web 浏览器和服务器之间进行双向数据传输，相较于Ajax技术需要客户端发起请求，WebSocket服务器和客户端可以彼此

[1] http://www.ietf.org/rfc/rfc2616.txt

相互推送信息，建立持续的连接。其协议基于 TCP 协议实现，包含初始的握手过程，以及后续的多次数据帧双向传输过程。其目的是在 Web 应用和后台服务器进行双向通信时，可以使服务器避免打开多个 HTTP 连接进行工作来节约资源，提高工作效率和资源利用率。

WebSocket 是一种类似 TCP/IP 的 socket 技术，其协议为应用层协议，定义在 TCP/IP 协议栈之上。WebSocket 连接服务器的 URI 以"ws"或者"wss"开头。ws 开头的默认 TCP 端口为 80，wss 开头的默认端口为 443。具体来说，共分为握手阶段和数据通信两个阶段。

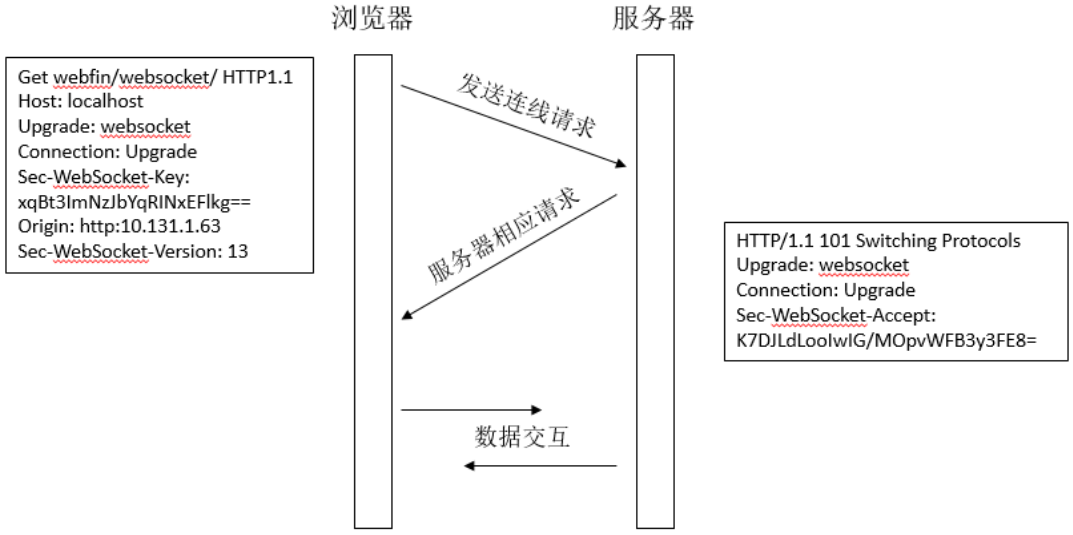

图 2-1 握手阶段示意图

如图 2-1 所示，在握手阶段，户端和服务器建立 TCP 连接之后，客户端向服务器发送握手请求，随后服务器向客户端发送握手响应，随后连接建立，双方可以互通信息。

在数据通信阶段，数据被组织为依次序的一串数据帧(data frame)，然后进行传送。帧的类型分为两类：数据帧(data frame)和控制帧(Control frame)。数据帧可以携带文本数据或者二进制数据；控制帧包含关闭帧和 Ping/Pong 帧。如要关闭连接，只需任何一端发送关闭数据帧给对方即可，关闭连接以状态码进行标识。

2.4 LRU 替换策略

LRU（Least Recently Used）是“最近最少使用”的意思，算法根据数据的历史访问记录来淘汰缓存中的数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高。这种算法来自操作系统的内存管理，常用于页面置换算法，是为虚拟页式存储管理服务的。对于在内存中存在但又不用的数据块（内存块）叫做 LRU，操作系统会根据哪些数据属于 LRU

而将其移出内存而腾出空间来加载另外的数据。

现在这种算法常用于管理缓存中，简单的说就是缓存一定量的数据，当超过设定的阈值时就把一些过期的数据删除掉。LRU 算法就是根据数据的历史访问记录，删除最近没有被使用到的文件，从而保证缓存中的数据不会过期，进行实时更新。

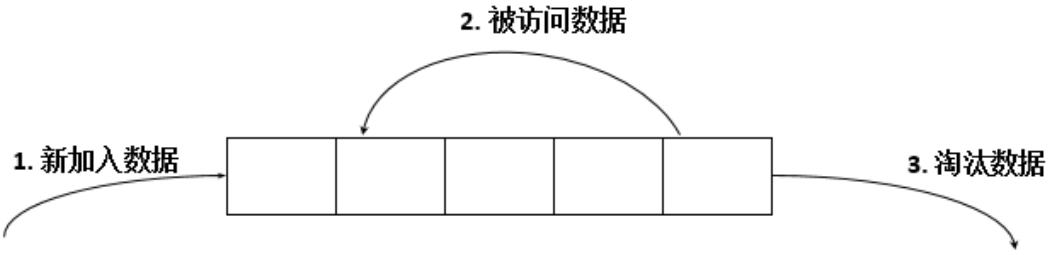

图 2-2 LRU 替换策略示意图

图 2-2 所示的是 LRU 替换策略的逻辑。当缓存中的存储空间足够时，新加入的数据以队列的方式加入缓存中，如果某条数据被访问了，那么它在队列中的位置将会往后移动。当缓存中的存储空间被使用完毕后，位于队首的数据将会被淘汰。

2.5 Java SDK

SDK（software development kit）即“软件开发工具包”是由第三方服务商提供的实现软件产品某项功能的工具包。一般以集合 API 和文档、范例、工具的形式出现。

通常 SDK 是由专业性质的公司提供专业服务的集合，比如提供安卓开发工具、或者基于硬件开发的服务等。也有针对某项软件功能的 SDK，如推送技术、图像识别技术、移动支付技术等，同时资源优势类的公司也提供资源共享的 SDK，如一些广告 SDK 提供盈利渠道，分发 SDK 提供产品下载渠道。

随着国内互联网环境的大发展，大部分的 SDK 都是免费的 但已经有一部分功能性 SDK 已经被当作一个产品来运营，这依赖于人们开发互联网产品理念的变化和云计算技术的发展。开发者不再需要对产品每个功能进行开发，选择合适、稳定的 SDK 服务并花费很少的精力即可在产品中集成某项功能。

2.6 WebExtension

基于 Firefox 附加组件(Add-ons)扩展是一种具有新功能的加载项。它们使用标准的 JavaScript、Html、CSS 等网页技术编写，再加上一些专用的

javascrip API 进行开发。另一方面，附加组件可以为浏览器增加新的特性或者改变某些网站的外观。

WebExtensions（扩展）是跨浏览器的用于开发附加组件的工具。在很大程度上，与谷歌浏览器 Chrome 和欧朋浏览器 Opera 所支持的扩展 API 兼容。为这些浏览器所写的扩展在大多数情况下只需少量修改的便可在火狐浏览器 FireFox 和 Microsoft Edge 浏览器上运行。这些 API 与多线程 Firefox 完全兼容。

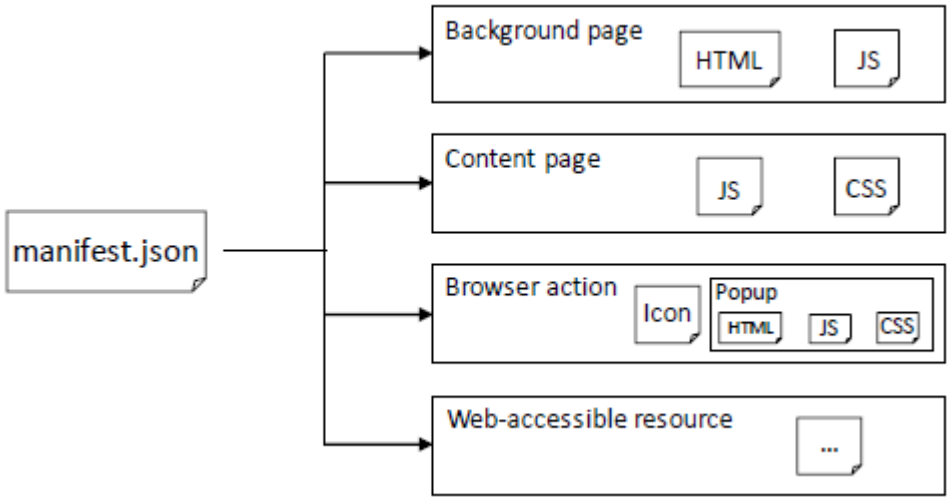

图 2-3 WebExtension 框架图

WebExtensions 的配置信息保存在一个名为 manifest 的文档中，以 json 的数据格式保存各项配置信息。这是惟一一个在每个 WebExtension 里面必须存在的文件。包含了关于这个扩展插件基本的元数据（metadata），比如它的名字、版本和所需权限。 它需要版本信息与权限。并且，他也对 WebExtension 中其他文件进行了链接。本文用到的配置信息以及整体框架图 2-3 所示。

Background page： WebExtension 常常需要在浏览器窗口或特定网页的存活期中独立的维持一种长期的状态或者执行长期的操作，比如与服务器建立持久的连接，这个时候就需要 Background page 中的 JavaScript 脚本来实现。运行在后台的文件不会因为当前界面关闭也停止，它的生命周期直到浏览器关闭才结束。

Content page：该文件夹下保存的文件称之为 Content scripts，被用来获取、操作页面。这些脚本会被加载到页面中并运行在页面的特定环境下，可以像普通脚本一样获取、操作页面的 DOM。

Browser action：浏览器动作是指通过在工具栏上添加扩展的图标，使得用户可以点击并与之交互。该文件加下保存了扩展的图标以及初始界面的 html 文件等资源。

Web-accessible resources: 是指像图片、HTML、CSS 和 JavaScript 之类的，引入插件并且想要获得访问权限的内容脚本和页面脚本。成为 web-accessible 的资源可以在页面脚本和内容脚本中通过使用特定的 URL 方案来引用。

第三章 云存储中间件系统优化分析

本章主要通过对现有的中间件系统进行全面分析，详细阐述其已实现的功能，并在此基础上结合对国内外云存储服务平台的调研结果，指出当前中间件系统的不足之处，由此分析出需要优化的具体内容。

3.1 现有中间件系统分析

现有的云备份中间件系统是基于 Linux 系统的，它提供了客户端与存储服务器之间的数据交互，有效地分离了客户端和服务器之间的绑定。同时，云备份中间件也提供给云备份客户端与云备份服务器，身份认证服务器以及密钥管理服务器之间稳定可靠的数据传输[4]。

3.1.1 系统框架分析

图 3-1 现有云存储中间件系统架构图

图 3-1 展示了现有的云存储中间件系统框架图。中间件作为客户端与服务器之间的交互枢纽，负责对用户从客户端发送的请求和数据进行处理，然后转发给服务器。中间件系统一共分为三层，底层结构包括操作系统平台、本地库、内存与本地存储三部分，它提供了中间件可以正常运作的基础软硬件；功能层由四大核心模块组成，分别是安全会话模块、数据传输模块、数据加密模

块，以及数据压缩模块；接口层包括了为客户端提供的所有中间件服务，这些服务可以分成三大类，分别是信息处理类、文件操作类，和回收站操作类。

3.1.2 功能分析

目前，云存储中间件在功能上主要包括四个模块：数据传输模块、数据加密模块、数据压缩模块、安全会话管理模块，各模块的详细说明如下：

1. 数据传输模块：实现了服务器与客户端之间的数据交互，主要提供数据的上传与下载功能，这也是云存储中间件系统的基本功能。
2. 数据加密模块：为了保证中间件系统能够抵御各种恶意网络攻击，中间件系统会将数据以加密的形式进行网络传输。使用的是高级加密标准（Advanced Encryption Standard，AES）对称加密算法，该算法是当前最流行的对称加密算法。
3. 数据压缩模块：支持对大文件的压缩功能。当上传大文件时，用户可以通过 COMPRESS 参数指定对文件进行压缩，以减少数据传输的大小，降低所需的网络带宽，提高传输的效率。此外，整个压缩过程是透明，用户通过压缩上传的文件，下载后是解压的格式。
4. 安全会话模块：中间件作为服务器与客户端之间的桥梁，需要保存用户的登录信息，以方便客户端与服务器之间建立持续的连接。原有的中间件系统采取一种安全的方式以确保用户的信息的安全性。

3.2.3 缺陷分析

从功能上看，现有的中间件系统实现数据传输、加密、压缩和安全会话等功能，能够有效地将用户的数据传输到云端，并且具备一定的安全机制。然而，根据目前行业的发展趋势，现有的中间件系统存在很多的不足之处，具体来说有以下几个方面。

首先，现有的中间件系统没有考虑到数据传输模块是以网络作为基础的，默认网络在数据传输过程中是绝对稳定的。但是，在实际生活中，我们的网络状况常常受限于硬件、软件以及地域等多方面的因素，网络波动或网络中断是常见的问题。由于我们的网络带宽是有限的，因此上传文件，尤其是较大的文件，通常需要花费一定的时间，在此期间存在因网络故障而导致文件传输失败的问题。在现有的中间件系统里，用户只能重传文件，

倘若多次遇到网络故障，则可能导致无休止的重传，这对用户来说是极其不友好的体验。

其次，为保证用户数据的安全性，现有的中间件系统在文件传输过程中，系统采用了 128 位的 SSL 加密技术，在文件的存储形式上，让用户主动选择是否加密，使用的是 AES 加密算法，这样的加密策略与主流的云存储服务平台使用的方式基本一致。然而中间件作为服务器与客户端之间的桥梁，需要保存一定的用户信息，以作为交互的保障。用户的登录信息、会话信息等敏感数据在现有中间系统中是以缓存的形式保存的，这些信息以明文的形式保存在 SQLite 数据库中，同时也未设置数据库的访问权限。因此，用户的私密信息很容易被窃取。此外，由于会话等信息是以缓存的形式存储在中间件系统中的，若中间件系统出现故障，则有可能造成用户会话信息的丢失。

再者，现有的中间件系统未对冗余的数据进行处理。当用户上传一个已有的文件到服务器时，服务器不会对文件进行去重检验，而是会重命名该文件，并以文件的原始形式保存。当冗余的文件过多时，会占用服务器大量的存储空间。去重技术虽然有一定的安全隐患，但是我们可以借鉴 Dropbox 针对单一用户的去重策略，这样既能缓解服务器的存储压力，又能为在一定程度上保证数据的安全。

最后，现有的中间件系统只支持 Linux 平台下的 CLI（Command Line Interface）客户端。考虑到主流的云存储供应商都提供了多平台数据共享的服务，我们需要对现有的中间件系统进行重构，解决来自多平台的不同客户端与云存储服务器的适配问题，使其具备可扩展性以支撑更多的客户端类型。

3.3 中间件系统优化需求分析

通过上文对现有中间件系统的缺陷分析，并结合业内同类产品的调研结果， 本文将从持久会话管理、大文件上传、本地缓存、多应用适配四个方面进行优化，具体分析见下文。

3.3.1 持久会话管理

云备份中间件在客户端和服务器之间工作，多个客户端的命令通过中间件提交给服务器，用户在访问存储服务器上的资源时，需要提供身份和授权信息，因此中间件也需要保存用户的相关私密信息。由于存在多用户同时将命令提交给中间件的可能，因此用户的命令不会马上被执行，而是以缓存的形式存储在数据库

中，只有被处理请求的进程轮询到时才会生效。在原有的系统中，用户的会话信息是以明文的形式保存的，因此需要设计一种行之有效的策略来保证用户私密信息的安全。具体来说，可以从以下几个方面进行优化：

1. 不同用户的数据，按照会话进行逻辑隔离，一个用户只能访问自己所属会话的数据。
2. 明文形式的秘密信息，只能作为会话数据在内存中存在。
3. 用户的持久私密信息以加密的形式存储，通过会话管理模块只能获取到加密形式的数据。

此外，客户端可以批量提交任务给中间件，即使用户注销后，中间件仍要在后台继续执行任务。更进一步，在中间件系统重启后，中间件也需要自动执行未完成的任务。因此，中间件需要持久化保存多个客户端的请求，以可靠地执行客户端提交的任务。同时，我们需要设计一套基于会话信息的接口，便于用户查询已提交的请求，如果请求没有执行，用户可以撤销该任务。

3.3.2 大文件上传

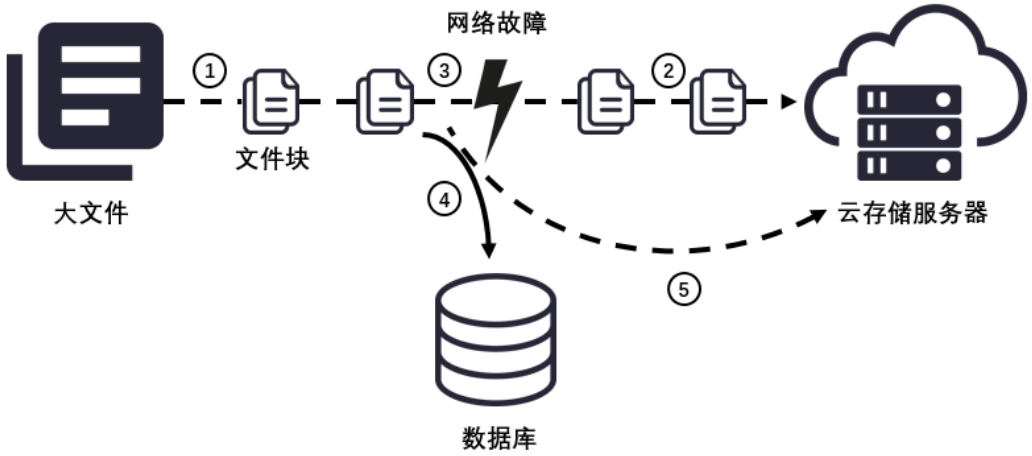

图 3-1 大文件上传及断点续传过程分析图

在很多情况下，客户端和云备份服务器分布在不同的地域，网络传输速度受各种条件制约。在现有的中间件系统中，一个文件如果传输不成功，下一次只能重传，这浪费了大量的网络带宽和时间。事实上，服务器支持大文件分块传输的功能，也就是说客户端可以传输指定大小的文件数据。因此，中间件可以根据网络数据传输的质量，确定文件分块传输的大小，并在每次上传时记录文件以上传的块数，即已上传文件的偏移量。如果遇到网络故障，可以根据偏移量实现断点续传的功能。这样做能够很大程度上地减少因网络因素导致的数据重传问题，提

高大文件的传输成功率。

图 3-1 详细描述了大文件上传的过程。

1. 将大文件分割成固定大小的文件块；
2. 依次将文件块上传至服务器；
3. 遇到网络故障；
4. 将当前已传送文件块的信息记录到数据库；
5. 查询数据库中的断点信息，实现文件续传，并调用合并文件接口组合文件。

3.3.3 本地缓存

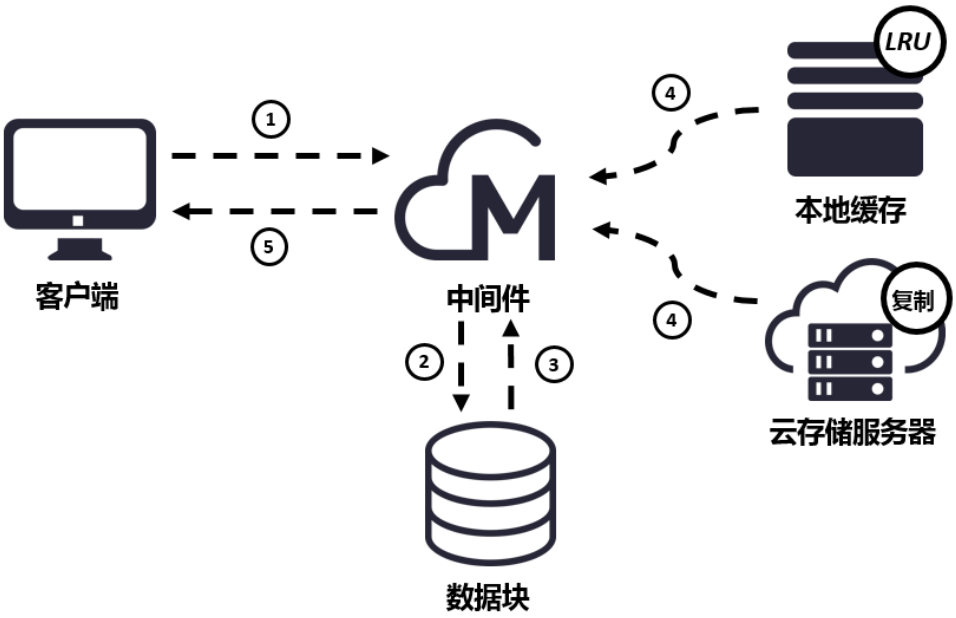

图 3-1 本地缓存示意图

云存储是存储和网络 I/O 密集的服务[10]，大量的 I/O 会引起机器性能下降，也可能会干扰其它的应用。现代计算机通常有很大的本地存储空间，并且计算能力普遍过剩。因此，可以通过利用本地存储和计算能力，如优化本地缓存的利用，进一步减少 I/O，提高服务性能。

现有的中间件系统在接受到用户文件上传和下载时，首先将文件存储在临时区域，该区域只能被中间件访问。由于客户端有远程访问文件的功能，所以用户可能会反复访问同一个文件，如果每次都从远程下载该文件，则是对网络带宽资源的浪费。因此，我们需要高效的临时存储区域，以降低下载文件时所需要的 I/O 请求。考虑到本地缓存的具有一定的容量限制，当缓存容量达到上限时，可以采用 LRU 替换策略[11]以保证缓存文件的时效性。

此外， 同一个用户备份数据时会有相同的文件。在上传文件时，同一用户的

相同文件是可以复用的，我们可以利用服务器提供的复制文件接口，直接在服务器操作，以减少上传文件所需的网络带宽，从而实现“秒传”功能。

图 3-2 详细描述了本地缓存的过程。

1. 用户发送上传/下载请求至中间件；
2. 中间件查询服务器/本地缓存中是否存在用户待上传/下载的文件；
3. 返回数据库查询结果；
4. 上传文件上，若服务器存在该文件，则将服务器中的文件复制到用户的上传路径；下载文件时，若本地缓存存在该文件，则将缓存中的文件复制到用户的下载路径；若不存在缓存文件，则执行正常的上传/下载流程。
5. 将请求结果返回给用户。

3.3.4 多应用适配

作为中间件，其主要的目标之一是完成不同客户端和服务器之间的适配，以使平台具备一定的可扩展性。由于平台和操作系统之间的差异性，加上不同客户端在设计时考虑的情形和目标不一样，客户端和服务端通常会定义不同的接口，导致服务器和客户端不匹配。

为了减少客户端和服务端之间的差异，可以通过中间件来对双方的调用方式进行转换，适配分为两种情况：

1. 提供一个标准的接口，客户端调用中间件给出的接口；
2. 在中间件中引入新的适配模块，以较小的代价将不同的客户端和服务端整合在一起。

第一种情况适合于规范新开发的客户端，从用户使用的角度来统一定义一个访问接口。第二种情况，实际上是第一种情况的强化形式，需要中间件来设计更灵活的机制来支持更一般化的适配。在这里，我们选择了第二种方式作为中间件多应用适配的方案。

为了适配客户端发出的请求（输出形式）和服务端的接收请求（输入形式）之间的差异，中间件需要对来自不同客户端的请求进行适配，以服务器的标准接口规范作为基准，转化为统一的、服务器可识别的形式。同时，由于目前版本的中间件只有 CLI 一个客户端，因此我们需要开发其他类型的客户端来支撑多应用适配，在这里，我们选择了开发基于 Firefox 浏览器的扩展和 Java SDK 开发工具。

第四章 云存储中间件系统优化设计

4.1 中间件优化系统框架设计

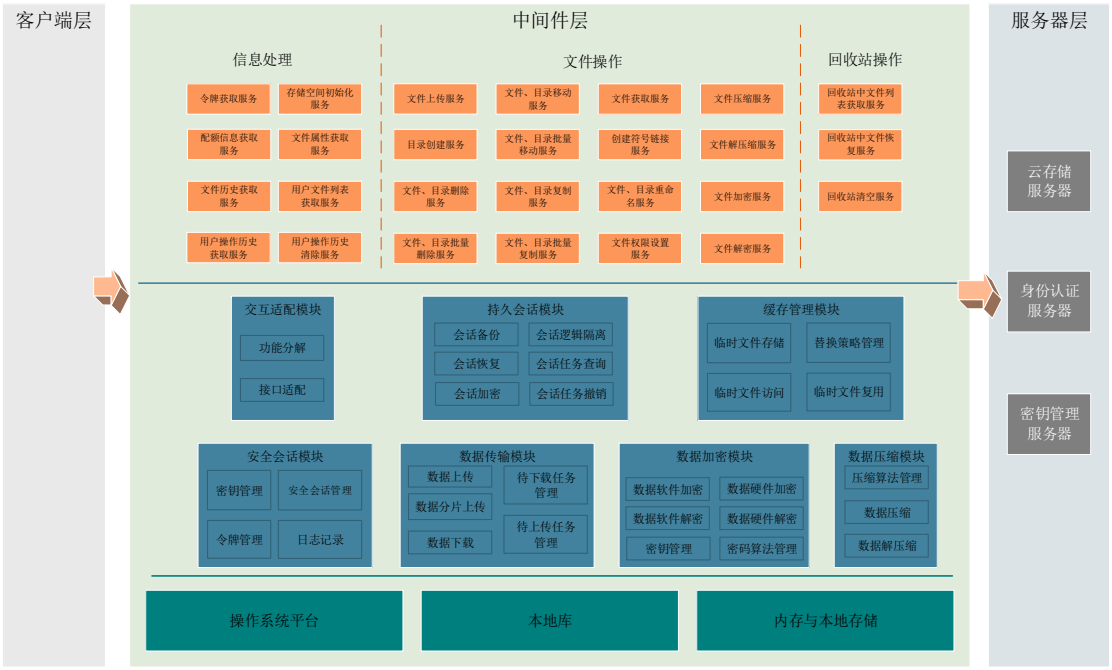

图 4-1 云存储中间件系统结构图

图 4-1 展示了优化后的中间件系统框架设计图，在原有系统的基础上，主要针对第二层架构进行重构，新增了持久会话模块、大文件上传模块、本地缓存模块以及多应用适配模块。

4.2 类图设计

我们对原有的中间件系统源代码进行了重构，因此，在类图的设计上也做了很大了改变。由于原有的中间件系统只实现了数据上传、加密、压缩这三项功能，因此类图设计相对比较简单，由 mwServer 类接收用户的请求，该类的主要作用是启动系统的配置文件，并在接收到用户请求后为其分配一个线程，该线程通过调用 mwController 类将用户请求发送给服务器，并将服务器返回的结果返回给 mwServer 类，最终由 mwServer 类将结果返回给客户端。此外，原有的中间件系统提供对数据的加密和压缩功能，这两个功能分别由 mwEncrypter 类和 mwCompresser 类实现。

图 4-2 展示的是优化后的云存储中间件系统 UML 类图，相对于原有的中间件系统类图，增加了许多类，具体来说有以下几个类：

mwChunk：该类主要用与大文件传输，将超过规定大小的文件拆分为

固定大小的文件块，并将按当前用户的信息以及时间戳信息自动命名，保存在本地的缓存文件中。此外，该类还需记录上传文件的上传状态信息，如断点信息，为后续的断点续传技术实现提供支持。

mwConstant: 该类的主要作用的为系统提供全局的常量，如字符串常量、整型的数字常量（主要用来标识状态或异常信息），并对这些常量进行统一管理。在原有的中间件系统中，常量都是以单独的形式存在于各个类中，这样存在大量重复重复冗余的常量信息，且对源码进行维护时，难以找全所有的常量进行统一修改，这对开发和维护工作带来了很大的困扰。

mwDao: 该类主要负责数据库层接口的定义与实现，实现业务逻辑与数据库操作的分离。在原有的中间件系统中，数据库操作都是在需要的时候创建连接，并执行所需的操作。这样的做法使得对数据库的操作十分混乱，并且难于管理，因此在优化的过程中，我们将数据库层的操作与业务层的操作进行了分离。在 nwDao 类中，提供了大量的接口如 add，delete，update，query 等，这些接口使得系统的实现层次更加清晰，并且便于修改和管理。

mwFileUtil: 该类的作用是处理与文件相关的请求，如获取文件的大小、MD5 值、复制文件等。这些操作对于云存储服务来说十分常见的，因此我们单独此类方法统一封装在 mwFileUtil 类中，以方便在后续的开发过程中使用。

mwUserUtil: 该类的主要作用是获取用户的信息，如用户的 id、用户的 token、查询用户登录状态是否过期。这些在 service 层中都是经常使用到的信息，因此我们也对这些功能进行了封装。

mwFirefoxHelper: 该类的主要作用是处理来自 firefox 扩展的请求。由于 Firefox 扩展请求的特殊性，使用到了 WebSocket 作为服务器与客户端之间的通行方式，与 Socket 通信方式不同的是，它需要进行一次握手协议，因此，我们需要对来 Firefox 的请求进行单独处理。

mwParser: 该类的主要目的是适配来自不同客户端的请求，并将请求转化为了服务器可识别的标准形式。此外该类预留了一定的接口，使得中间件系统具备一定的可扩展性，为以后开发更多种类的云存储客户端奠定基础。

图 4-2 展示了类图信息及他们之间的关联关系。

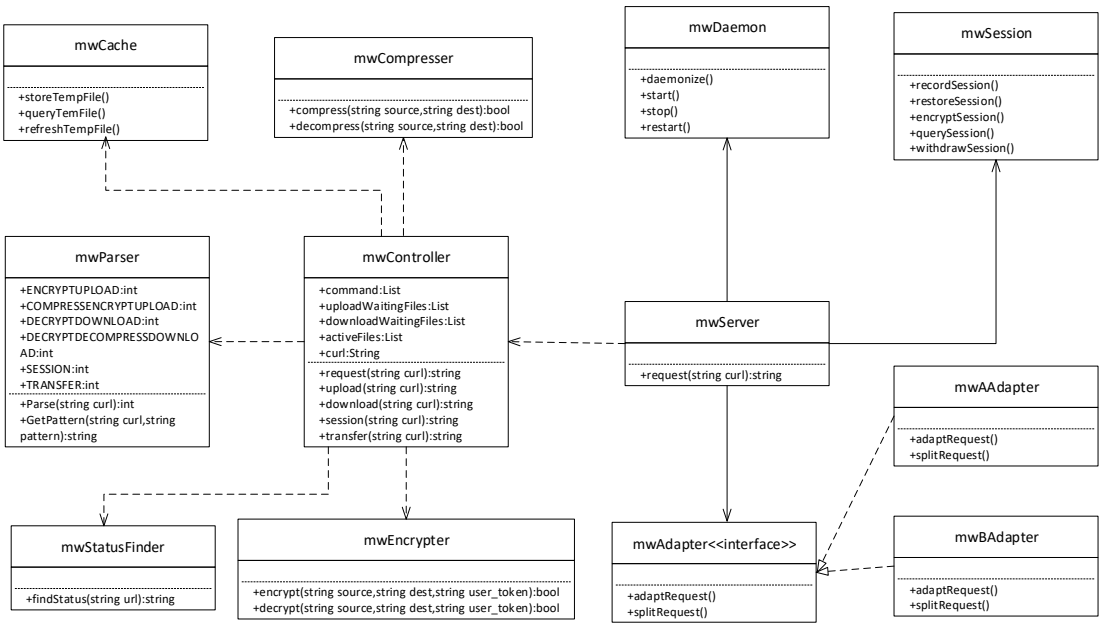

图 4-2 云存储中间件系统 UML 类图

4.3 关键流程设计

4.3.1 系统总流程设计

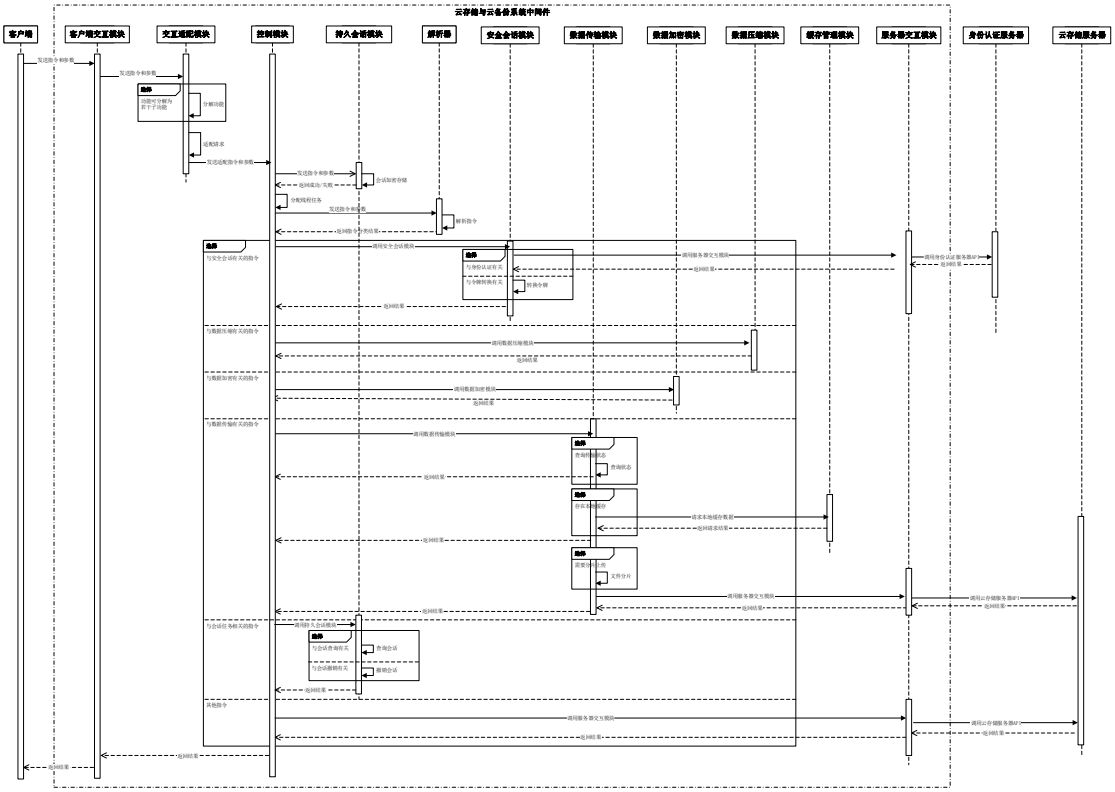

图 4-3 云存储系统中间件总体流程图

从系统的总流程上来看，优化后的中间件系统并未改变原有的系统的整体流程，我们只是通过对关键技术的优化，对每个步骤进行了细化。

客户端向中间件发送请求，在原有的中间件系统中，只需将用户的请求通过解析器模块进行格式化操作，即可传递给服务器。而在优化后的中间件系统中，我们将进行持久会话管理操作，当用户的请求到达中间件后，首先会将用户的登录信息 id、token 等以及请求的详细信息保存在 SQLite 数据库中，同时由于这些信息中包含用户的隐私，因此我们会首先对这些信息加密，然后再存入数据库中。此外，由于用户的请求可能来自不同的客户端，它们可能使用不同的通信协议，不同的请求信息，因此，我们需要对来自不同客户端的用户请求进行适配。此外，当用户的请求为上传或下载文件时，我们需要首先判断本地缓存中或者服务器端有没有当前文件，如果存在，那么执行文件复制操作，若不存在，则执行正常的文件长传下载操作。同时，我们需要判断上传文件的大小，如果超过规定的文件块大小，那么需要执行大文件分块传输。

中间件系统优化具体的流程如下：

4.3.2 持久会话管理流程设计

云备份中间件在客户端和服务器之间工作，多个客户端的命令通过中间件提交给服务器。当用户注销或者在机器重启的情况下，仍然需要中间件执行未完成的任务。因此，中间件需要持久化保存多个客户端的请求，以可靠地执行客户端提交的任务。同时，用户在访问存储服务器上的资源时，需要提供身份和授权信息，因此中间件也需要保存用户的相关私密信息。以下展示的是会话持久化管理的流程设计，包括会话持久化与会话加密管理流程设计。

1）会话持久化流程设计

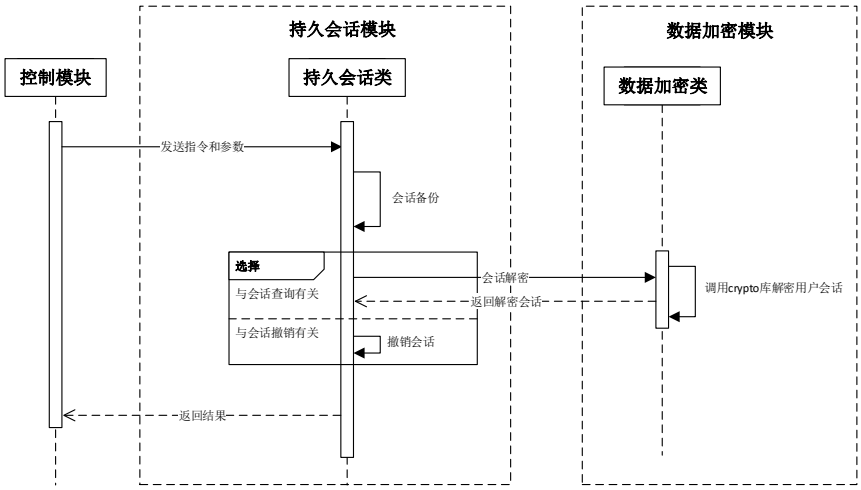

图 4-4 云存储与云备份系统中间件会话持久化流程图

控制模块接收到客户端交互模块的用户指令后，首先将指令发送给持久会

话模块进行会话备份。若请求中含有与会话查询有关的指令，持久会话模块会从数据库中查询该条指令，并调用数据加密模块进行相应的解密操作，然后将结果返回给客户端；若请求中含有与会话撤销有关的指令，则直接在数据库中将相应的会话删除，并将删除结果返回给客户端。

2） 会话加密流程设计

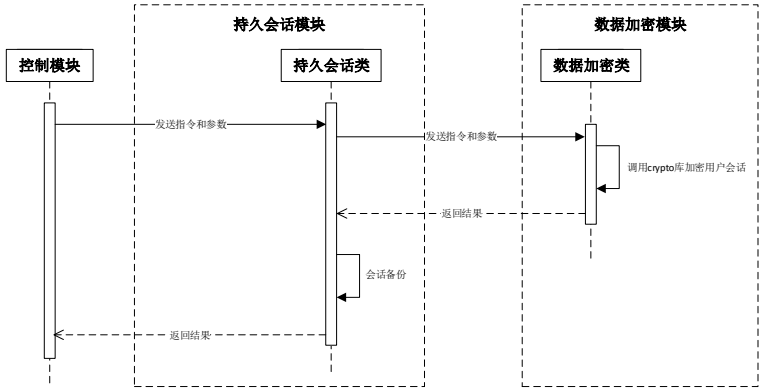

图 4-5 云存储与云备份系统中间件会话加密流程图

控制模块接收到客户端交互模块的用户指令后，需要对会话进行备份。然而以明文的形式对会话进行存储可能会导致用户信息泄露，因此在对会话进行存储之前，控制模块会调用数据加密模块对用户会话进行加密，然后再存入数据库，并将结果返回给客户端。

4.3.3 大文件上传流程设计

系统优化包括优化中间件，充分利用服务器的高级功能，以提高系统性能；同时可以利用现代计算机的处理器，通常都有多个处理器或者多个处理核，多任务可以充分发挥系统性能；此外，在原有系统中，客户端提交任务如果长时间不活跃，则会导致大量任务的失败和重新提交，浪费了资源，用户体验也不够好。结合会话信息管理，我们可以进一步支持用户对中间件的授权，从而解决身份认证和授权信息过期所引发的问题。以下展示的是系统优化的流程设计，包括大文件上传下载流程设计、并发任务执行流程设计以及中间件授权流程设计。

1） 大文件下载流程设计

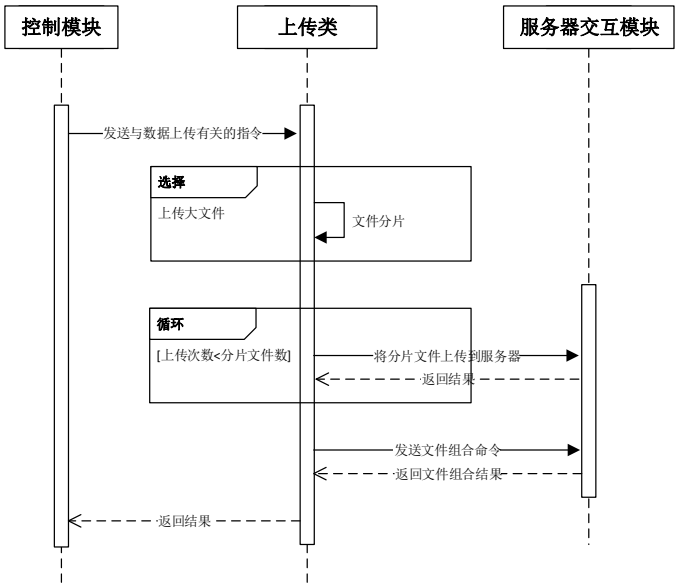

图 4-6 云存储与云备份系统中间件大文件上传流程图

控制模块接收到数据上传的指令后，首先会对文件大小进行判断。若是大文件则调用分片上传函数，将大文件分成固定大小的临时小文件后逐一分片上传，并将上传结果返回。由于服务器不会自动将分片的文件进行整合，而是提供一个整合分片文件的接口，所以在文件上传完毕之后需要向服务器发送文件整合的命令将分片上传的文件进行整合，并将执行结果返回给控制模块。

2） 大文件下载流程设计

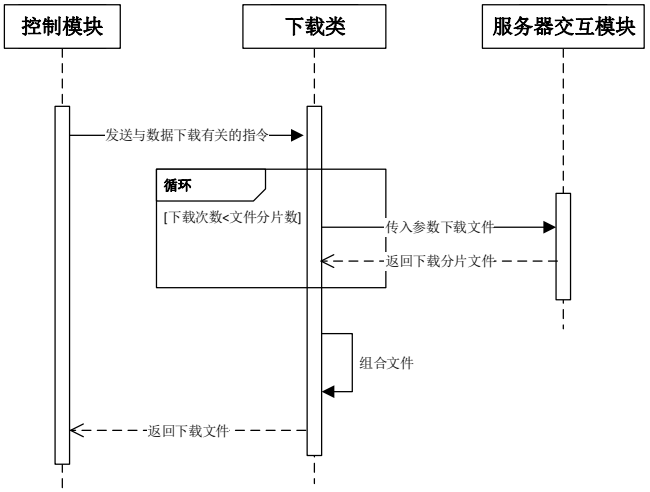

图 4-7 云存储与云备份系统中间件大文件下载流程图

控制模块接收到数据下载的指令后，首先会对文件大小进行判断。若是大文件则调用分片下载函数，每次请求包含分片文件的起止索引和大小，通过多次请求服务器完成大文件的下载。将所有分片文件下载完成后，对分片文件进行组合并返回给控制模块。

4.3.4 本地缓存流程设计

用户在文件上传和下载时，可能会反复访问同一文件，这时可用充分利用临时存储区域，将反复访问的文件保存在临时存储区域。通过分析请求和文件内容，可以进一步在客户端和服务器端上降低系统 I/O。以下展示的是本地缓存管理模块的流程设计，包括数据上传和下载缓存管理流程设计。

1） 数据上传本地缓存流程设计

控制模块接收到用户上传文件的指令时，首先获取待上传文件的信息，包括文件大小和 MD5 值等信息，根据文件信息判断缓存中是否存在该文件。若缓存中存在该文件，则向缓存管理模块请求该文件并上传到服务器，并将上传结果返回给客户端；若缓存中不存在该文件，则将用户文件直接上传到服务器，并返回上传结果。

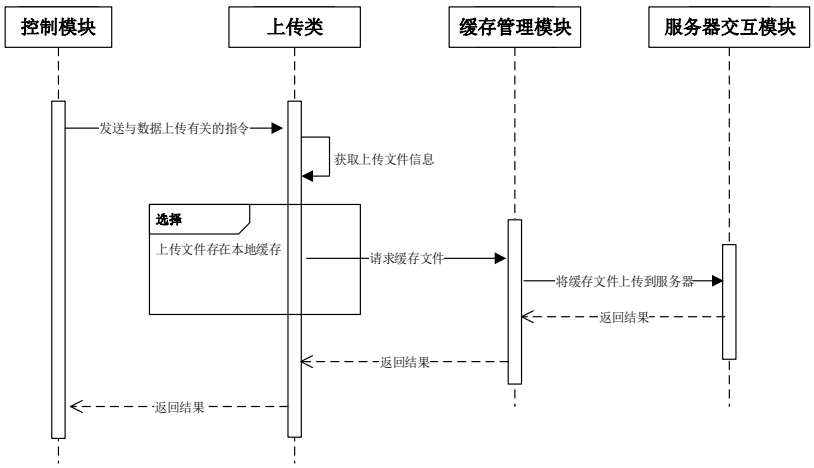

图 4-8 云存储与云备份系统中间件数据上传缓存管理流程图

2） 数据下载缓存流程设计

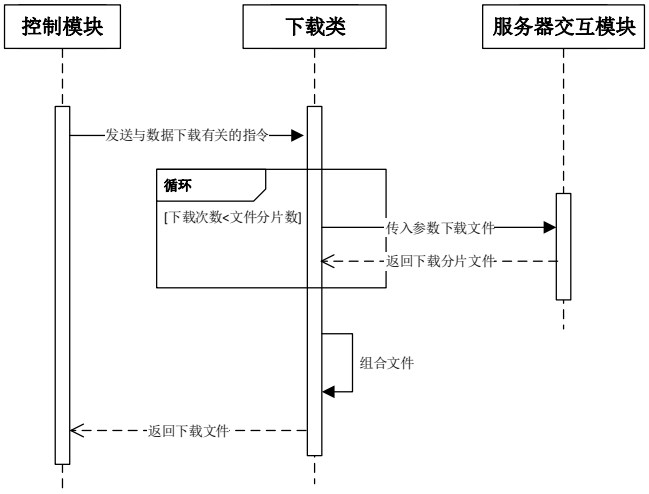

图 4-9 云存储与云备份系统中间件大文件下载流程图

控制模块接收到数据下载的指令后，首先会对文件大小进行判断。若是大文件则调用分片下载函数，每次请求包含分片文件的起止索引和大小，通过多次请求服务器完成大文件的下载。将所有分片文件下载完成后，对分片文件进行组合并返回给控制模块。

4.3.5 多应用适配流程设计

由于设计时考虑的情形和目标不一样，客户端和服务端通常会定义不同的接口，导致服务器和客户端不匹配。交互适配管理模块的主要的目标是完成不同客户端和服务之间的适配。以下展示的是交互适配模块进行命令接口转换时的流程设计。

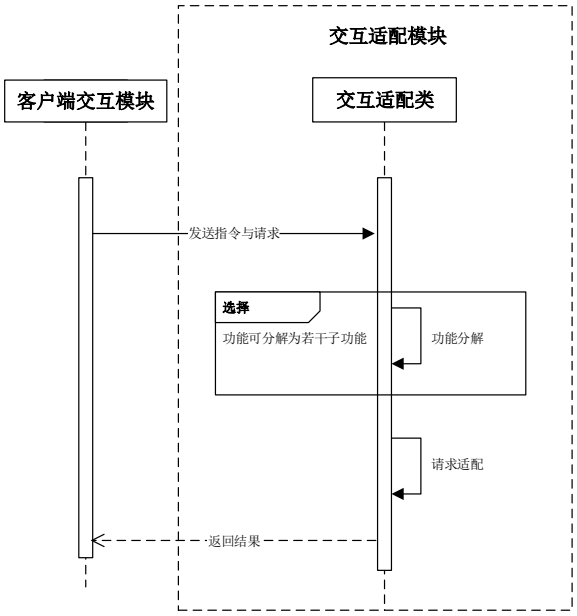

图 4-10 云存储与云备份系统中间件交互适配模块流程图

交互适配模块接收到客户端交互模块发送的指令后，首先判断该指令中所请求的功能是否可以分解为若干个子功能，若能分解为若干个子功能，则先将指令分解然后逐一适配，并将适配后的指令交给控制模块处理，最后将处理后的结果进行整合返回给客户端。

4.4 数据库设计

中间件系统优化涉及到许多数据存储的问题，如对用户信息加密存储、记录大文件上传断点信息、记录本地和服务器文件的缓存信息，这些信息都需要保存

在数据库中。在实现的过程中，我们使用的是 SQLite 关系型数据库，之所以选择 SQLite，是因为它是一款轻量级的数据库，且可以在几乎所有的主流操作系统上运行，并支持大多数的 SQL92 标准，源代码不受版权限制。此外，最大支持 2TB 的存储空间也足够存储缓存信息。在处理事务的能力上，也具备一定的优势。

我们在系统的数据库中，一共设计了三张表，分别用来保存用户的请求信息、缓存信息以及断点信息。详细信息如下：

用户请求信息表：

```
create table if not exists Log(
    request text not null,          //请求内容
    uuid char[33] not null,         //请求标识
    state char[20] not null,        //请求状态
    time char[20] not null,         //请求时间
    userid varchar[100] not null,       //请求用户 id
    result text not null);              //请求执行结果
```

缓存信息表：

```
create table if not exists fileCache(
    user_id varchar[100] not null,              //用户 id
    file_name varchar[100] not null,            //文件名
    length bigint not null,                     //文件长度
    MD5 varchar[100] not null,                  //文件 MD5 值
    use_time int not null,                      //使用次数
    modified_time varchar[100] not null);       //最后一次修改时间
```

断线信息表：

```
create table if not exists breakpoint(
    user_id varchar[100] not null,              //用户 id
    MD5 varchar[100] not null,                  //文件 MD5 值
    server_path varchar[100] not null,          //文件在服务器的位置
    point int not null default 0,               //断点偏移量
    flag bit not null default 0);               //标识是否需要断点
```

4.5 系统优化接口设计

程序接口是操作系统为用户提供的两类接口之一，编程人员在程序中通过程序接口来请求操作系统提供服务。原有的中间件系统为用户提供了大量的接口，满足了用户使用云存储服务的基本功能。优化后的中间件系统为用户提供了更多

的高级功能，这些功能都是以接口的形式呈现给用户和开发人员，下面我们将从函数原型、功能、参数说明、返回值说明、使用方法这五个方面，对优化后提供的接口进行详细的阐述。

4.5.1 持久会话管理接口设计

1）新增会话

接口编号	接口名称	新增会话
Cloud-Opt-01	接口原型	bool add_session(String request, String userid)
	接口参数	request：用户请求 userid：用户 id
	返回结果	True：新增会话成功 False：新增会话失败

2）删除会话

接口编号	接口名称	删除会话
Cloud-Opt-02	接口原型	bool delete_session(String sessionid, String userid)
	接口参数	sessionid：会话 id userid：用户 id
	返回结果	True：删除会话成功 False：删除会话失败

3）查询会话

接口编号	接口名称	查询会话
Cloud-Opt-03	接口原型	String query_session(String sessionid, String userid)
	接口参数	sessionid：会话 id userid：用户 id
	返回结果	{ "status": "0", "request": "curl -k -X POST -d '{"password": "123456", "email":"zhu__feng006@163.com"}' https://localhost:443/oauth/access_token", "state": "finished", "time": "1432653661.66864", "userid": "kaeyika@163.com"

		}

4）会话加密

接口编号	接口名称	会话加密
Cloud-Opt-04	接口原型	bool encrypt_session(String request)
	接口参数	request：用户请求
	返回结果	True：会话加密成功 False：会话加密失败

5）查询登录状态

接口编号	接口名称	查询登录状态
Cloud-Opt-05	接口原型	String query_session(String userid)
	接口参数	userid：用户 id

	返回结果	{
		"status": "0",
		"expired": "false",
		"time": "1432653661.66864",
		"userid": "kaeyika@163.com"
		}

4.5.1 大文件上传接口设计

1）获取文件信息

接口编号	接口名称	获取文件信息
Cloud-Opt-06	接口原型	String get_file_info(String file_path)
	接口参数	file_path：文件路径

	返回结果	{
		"status": "0",
		"Content-Length": "200",
		"X-File-Type": "f",
		"ETag":
		"7add118c851212736b362105e6235b84",
		"X-Timestamp": "1438679161.39779",
		"X-Object-Permisson": "500",
		"mode": "COMPRESS"
		}

2）大文件分块

接口编号	接口名称	大文件分块
Cloud-Opt-07	接口原型	bool split_file(String from, String to)
	接口参数	From 目标文件路径 userid：临时分块文件保存路径
	返回结果	True：文件分割成功 False：文件分割失败

3）大文件分块上传

接口编号	接口名称	大文件分块上传
Cloud-Opt-08	接口原型	String chunk_upload(String file_path, String server_path, String user_id)
	接口参数	file_path：文件路径 server_path：服务器路径 userid：用户 id

	返回结果	{
		"status": "0",
		"X-Object-Permisson": "500",
		"path":" “/temp”,
		"size":" 200",
		"mtime":"1438679161.39779",
		"ctime":" 1438679161.39779",
		"md5":"
		7add118c851212736b362105e6235b84",
		}

2）获取上传文件断点

接口编号	接口名称	大文件分块
Cloud-Opt-09	接口原型	bool get_breakpoint(String file_path, String MD5, String userid)
	接口参数	file_pat：文件路径 MD5：文件 MD5 值 userid：用户 ID
	返回结果	{ "status": "0", "X-Object-Permisson": "500", "path":" “/temp”,

		"size":" 200", "md5":" 7add118c851212736b362105e6235b84", "breakpoint":" 15", }

4.5.1 本地缓存接口设计

1）是否有本地缓存

接口编号	接口名称	是否有本地缓存
Cloud-Opt-09	接口原型	bool has_local_cache(String file_path, String MD5, String userid)
	接口参数	file_pat：文件路径 MD5：文件 MD5 值 userid：用户 ID

	返回结果	{
		"status": "0",
		"cached": "true",
		"path":" “/temp/test.docxs”,
		"size":" 200",
		"md5":"
		7add118c851212736b362105e6235b84",
		}

1）是否有服务器缓存

接口编号	接口名称	是否有服务器缓存
Cloud-Opt-09	接口原型	bool has_local_cache(String server_path, String MD5, String userid)
	接口参数	file_pat：服务器端文件路径 MD5：文件 MD5 值 userid：用户 ID
	返回结果	{ "status": "0", "cached": "true", "path":" “server/test.docxs”, "size":" 200",

		"md5":" 7add118c851212736b362105e6235b84", }

4.5.1 多应用适配接口设计

1）请求适配

接口编号	接口名称	是否有服务器缓存
Cloud-Opt-09	接口原型	bool adapt_request(String request)
	接口参数	request：原始请求
	返回结果	String：adapted_request //适配后的请求

第五章 云存储中间件系统优化实现

5.1 开发环境

表 5-1 所示的是中间件优化的开发环境

表 5-1 中间件优化开发环境表

类别	具体参数
操作系统	Ubuntu 14.04
开发语言	Python Java
开发工具	Atom， WebExtension
数据库	SQLite

5.2 关键技术

5.2.1 加密方式

缓存在中间件的用户登录信息是基于 libsecrypto 库的接口进行加密的，该接口由密钥服务器提供，使用的是 128 位的 AES 加密算法。由于该接口只提供文件级别的数据加密接口，不适应用户信息这类字符串级别的数据加密，因此我们对原有的接口进行了修改，新增了对字符串的加解密接口，具体实现如下文所示：

```
def sAES_str_encrypt(req, access_token):
    result = ''
    dll = CDLL("/usr/lib/libsecrypto.so")
    user_token = '{"access_token": "#"}'
    user_token = user_token.replace("#", access_token)
    offset = 0
    lens = len(req)
    tag = True
    while tag:
        buff = create_string_buffer(1024)
        if lens < 1024:
            chunk = req[offset:]
            chunk = "%s%s" % (chunk,'\001' * (1024 - lens))
            tag = False
        else:
```

```
            chunk = req[offset:offset+1024]
            offset = offset + 1024
            lens = lens - 1024
            inputs = c_char_p(chunk)
            token = c_char_p(user_token)
            dll.sAES_encrypt(inputs,buff,token)
            result = result + buff.raw
        return result
```

该接口的是需要传入两个参数，req 表示用户请求，access_token 表示用户登录 token。通过引入/usr/lib/libsecrypto.so 模块，对字符串进行加密。该模块提供的接口每次只能加密 1024 个字节的数据，因此我们需要对字符串的长度进行判断，如果超过了 1024 个字节，那么需要对字符串进行分割，逐一进行加密后再整合形成最后的加密结果。数据的解密算法是加密算法的逆过程，从数据库获取到加密后的会话信息后，需要提供用户的 token 来进行解密。同理，若是请求的长度超过了 1024 个字节，那么也需要逐一进行解密，最后组合得到解密信息。加密和解密接口的使用方式如下：

```
//加密算法
enreq = mwEncrypter.sAES_str_encrypt(clientrequest, access_token)
//解密算法
dereq = mwEncrypter.sAES_str_decrypt(enreq, access_token)
```

5.2.2 中间件授权代理

在原有系统中，客户端提交任务如果长时间不活跃，则会导致大量任务的失败和重新提交，浪费了资源，用户体验也不够好。结合会话信息管理，我们可以进一步支持用户对中间件的授权，从而解决身份认证和授权信息过期所引发的问题。

中间件系统授权代理的流程是当用户的请求发送到服务器时，服务器返回给用户的结果是用户登录信息过期，那么我们需要在中间件进行自动授权代理，重新登录后再重新发送用户的请求。

```
//生成用户 token 认证请求
req = __VERIFY_TOKEN__.replace("#token#", token);
//判断用户请求是否过期
```

```
data = os.popen(req)
data = data.read()
if(data.find('"status": "0"')!=-1):
    return True
else:
    return False
```

上述代码实现的是用户登录信息认证，若服务器返回的状态不为“0”，则表示用户的登录信息过期，需要通过中间件代理授权登录。

```
//生成登录请求
req = __LOGIN__.replace("#email#", userid).replace("#password#", password)
//更新用户 token
data = os.popen(req)
data = data.read()
rstjson = json.loads(data)
newToken = rstjson["access_token"]
```

中间件授权代理即使用用户的用户名和密码重新登录，在登录成功后，更新系统中保存的该用户的 token。对于改用的其他的请求，也需要替换其原有的过期 token。

5.2.3 大文件上传

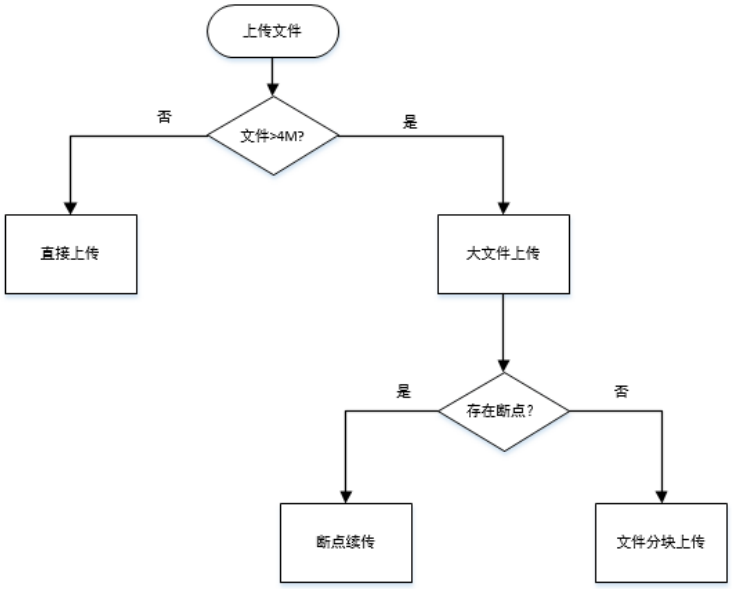

图 5-1 大文件上传流程图

图 5-1 所示的是大文件上传以及断点续传的流程图，当文件的小于超过 4M 时，执行正常的长传文件操作。当文件的大小超过 4M 时，需要对大文件进行分块上传，每个文件块的大小均为 4M，系统为每个文件块的上传分配一个进程。当文件块在上传的过程中遇到网络故障时，将断点信息保存在数据库中，下次上传该文件时，若存在断点，则从断点的位置继续上传。

```
//判断文件大小是否超过 4M
filesize = os.path.getsize(path)
if(filesize > __CHUNKSIZE__):
    result = self.chunkupload(rq, path, chunksize)
else:
    result = self.upload(rq)
//将文件分割成文件块
fileUtil.mwSplitFile(path, tempfile, chunksize)
//获取断点信息
if(breakpoint>0):
    breakpoint = breakpoint - 1
//将文件块逐一上传
for filepart in files:
    rq = rq.replace(tempfilename + '?op',filepart + '?op')
    result = self.upload(rq)
//在服务器端合并文件
for filepart in files:
    rq = rq.replace(tempfilename + '?op',filepart + '?op')
    result = self.upload(rq)
```

5.2.4 本地缓存

本地缓存的是指将客户机本地的物理内存划分出一部分空间用来存储用户频繁使用的文件，当用户反复下载同一个文件，如果每次都从远程下载该文件，则是对网络带宽资源的浪费。因此，我们需要高效的临时存储区域，以降低下载文件时所需要的 I/O 请求。此外， 同一个用户备份数据时会有相同的文件。在上传文件时，同一用户的相同文件是可以复用的，我们可以利用服务器提供的复制文件接口，直接在服务器操作，以减少上传文件所需的网络带宽，从而实现“秒

传”功能。本地缓存实现的核心代码如下：

1）下载文件本地缓存实现

```
//判断是否存在本地缓存
result = self.doCache(rq, n)
//查询本地缓存数据库
cx = sqlite3.connect('/var/log/mwcache.db')
fname_result = cx.execute("select file_name from localCache where
MD5 = '" + MD5 + "';")
//若存在本地缓存，则将缓存中文件拷贝到下载路径
shutil.copy(cache_path, destination)
```

2）上传文件实现“秒传功能”

```
//判断是否存在本地缓存
result = self.doCache(rq, n)
//查询服务器端是否存在待上传文件
cx = sqlite3.connect('/var/log/mwcache.db')
spath_result = cx.execute("select server_path from serverCache where
user_id = '" + user_id + "' and MD5 = '" + MD5 + "';")
//若服务器存在该文件，调用服务器端复制文件接口
cp_req = __COPY__.replace("#src_path#", src_path)
cp_req = cp_req.replace("#des_path#", des_path)
data = os.popen(cp_req)
```

上述代码中，使用到了了两张表分别保存本地缓存数据信息和存储在服务器端的数据信息，数据库的详细设计参见 4.4 数据库设计。表 localCache 用来保存保存在本地的缓存文件信息，用文件 MD5 进行唯一标识；表 serverCache 用来保存存储在服务器端的数据信息。由于服务器端不提供去重服务，因为我们需要组合服务器提供的接口，以实现“秒传”功能，通过获取文件属性的接口获取服务器端文件的 MD5 值，并通过该值与 serverCache 中 MD5 值进行匹配，若存在相同的 MD5 值，则说明服务器端存在该文件，此时，我们就调用服务器提供的复制文件接口，提升上传文件的效率。

5.2.5 LRU 替换策略

LRU 缓存利用了这样的一种思想。LRU 是 Least Recently Used 的缩写，即“最近最少使用”，也就是说，LRU 缓存把最近最少使用的数据移除，让给最新读取的数据。而往往最常读取的，也是读取次数最多的，所以，利用 LRU 缓存，我们能

够提高系统的性能。

要实现 LRU 缓存，我们需要用一个数据库记录每个文件的使用频率，每当用户上传或者下载一个文件时，将会更新数据中关于该文件使用次数的字段，自动加一。当本地缓存达到存储空间的上限时，我们会将使用次数最少的文件替换掉，如果同时存在两个使用次数相同的文件，那么我们会根据最近使用的时间来判断替换哪个文件。LRU 替换策略的核心代码如下：

```
//计算当前缓存文件的总大小
for fsize in file_size_result:
    cache_size += fsize[0]
cache_size += os.path.getsize(path)
//如果超过缓存空间的最大容量，执行 LRU 替换策略
if(cache_size > __CACHESIZE__):
    self.doLRU(rq, cx, path, MD5)
//获取使用次数最少的文件
fcache_result = cx.execute("select * from fileCache where use_time
<= %d"%use_time + " order by use_time, modified_time;")
//删除使用次数最少的文件
cx.execute("delete from fileCache where MD5 = '" + fcache[3] + "';")
os.remove(sys.path[0] + "/mwcache/cache/" + fcache[1])
//将当前文件拷贝至缓存文件夹
shutil.copy(path, sys.path[0] + "/mwcache/cache/" + file_name)
```

5.2.6 Firefox 扩展

浏览器扩展是一种用来修改 web 浏览器功能的工具。它们使用 JavaScript、Html、CSS 等标准的 web 技术，再加上一些专用的 javascrip API 进行编写。火狐扩展(Extensions for Firefox)由 WebExtensions API 构建而成。 在很大程度上，与谷歌浏览器 Chrome 和欧朋浏览器 Opera 所支持的扩展 API 兼容。通过浏览器扩展可以为浏览器增加新的特性或者改变某些网站的外观和内容。本文主要是为验证优化后的中间件系统具备多应用适配的特性，因此开发了一款基于 Firefox 的扩展，使得用户可以通过扩展来使用云存储服务。

1）组织结构

WebExtension 中包含的组件如下：

- background pages：执行一个长时间运行的逻辑

- content scripts：与网页进行交互(与 JS 在页面中的<script>元素不一样)
- browser action files：在工具栏中添加按钮
- page action files：在地址栏添加按钮
- options pages：为用户定义一个可浏览的 UI 界面，可以改变曾经的设置
- web-accessible resources：是打包好的内容可用于网页与目录脚本

manifest.json 是唯一一个在每个 WebExtension 中必须存在的文件。包含了关于这个扩展扩展基本的元数据。比如扩展的名字，版本和所需权限。以及扩张需要的版本信息与权限。并且，也对 WebExtension 中其他文件进行了链接。本文中 manifest.json 中的关键内容如下：

```
{
  //版本信息
  "manifest_version": 2,
  "name": "Middleware",
  "version": "1.0",
  //所需的权限信息
  "permissions": [
    "activeTab",
    "tabs",
    "cookies",
    "<all_urls>"
  ],
  //图标信息与初始界面
  "browser_action": {
    "default_icon": "icons/beasts-32.png",
    "default_title": "Middleware",
    "default_popup": "popup/middleware_login.html"
  },
}
```

上述代码描述的是 manifest.json 中一些较为关键的内容，从配置文件中可以清晰地看到扩展的名称、版本信息以及所需的权限信息。作为扩展的入口，我们需要在浏览器的导航栏里定义一个图标，以及一个初始界面，browser_action 参数中的 default_icon 属性指定扩展的图标，而 default_popup 指定了初始界面。

2）通信方式

WebExtension 使用的是网页开发的标准技术，因此与服务器的通讯方式也与

一般网页开发的方式相同，使用的是 Websocket。Websocket 需要通过一次握手协议才能与服务器建立连接，其实现的核心代码如下：

Firefox 扩展客户端核心代码：

```
//设置服务器地址和端口号，并新建 WebSocket 对象
var host = "ws://127.0.0.1:10000/"
socket = new WebSocket(host);
//连接建立时
socket.onopen = function (msg) {
    send(request);
};
//收到服务器返回的消息时
socket.onmessage = function (msg) {
    pyresult = pyresult + msg.data
    var JResult = JSON.parse(pyresult);
    portFromCS.postMessage({"JResult": JResult });
};
//连接关闭后
socket.onclose = function (msg) {
    portFromCS.postMessage({"mstatus": "connection closed"});
    login = false;
};
```

Python 服务器端核心代码：

```
//判断是否为来自 Firefox 的请求
if(recvdata.find("websocket")!=-1):
    clientrequest = handle_WebRequest(sock, recvdata)
//返回握手协议信息
def handle_WebRequest(sock, request):
    sock.send('\
    HTTP/1.1 101 WebSocket Protocol Hybi-10\r\n\
    Upgrade: WebSocket\r\n\
    Connection: Upgrade\r\n\
    Sec-WebSocket-Accept: %s\r\n\r\n' % token)
//处理来自 Firefox 客户端的请求
data = sock.recv(RECV_BUFFER)
```

在 Firefox 扩展客户端中，首先需要通过服务器的地址和端口号建立连接，在创建 WebSocket 对象时，会主动向服务器发送握手协议，报文由 WebSocket 自动生成。建立连接之后，WebSocket 通过三个回调函数 onopen、onmessage、onclose 分别来处理连接建立时、收到服务器返回的消息时、连接关闭后的请求和数据。

在 Python 服务器端，接收用户的请求之前需要完成握手协议。WebSocket 有一套标准的格式来规定服务器返回的信息，通过此报文以完成握手协议。随后，通过 recv 方法接收用户的请求。

3）部署与实现

在 Firefox 浏览器中输入“about:debugging” 页面，点击 "临时加载附加组件按钮" 并选择附加组件目录，即选择 manifest.json 文件，如下图所示：

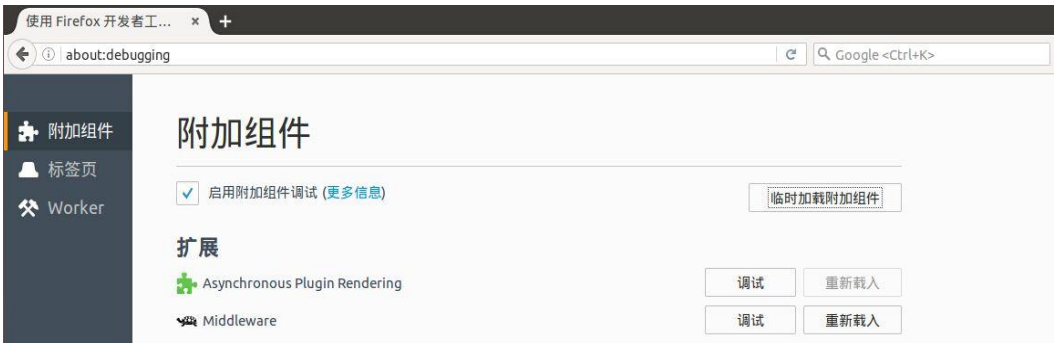

图 5-2 Firefox 扩展部署

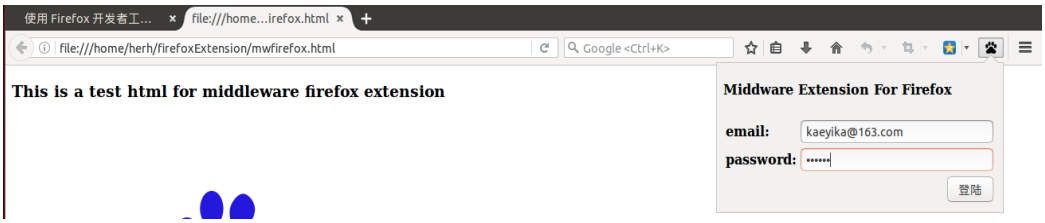

图 5-3 Firefox 扩展登录后查询存储空间效果图

图 5-2 展示的是将开发好的 Firefox 扩展导入到 Firefox 浏览器中，从上图中我们可以看到，名为 Middleware 的扩展已经被部署到了浏览器中。5-3 展示的是使用浏览器扩展登录云存储服务器，然后查询存储空间的效果图。

5.2.7 Java SDK

优化的中间件系统具备一定的可扩展性，即支持多平台的应用使用云存储服务。除了提供基于 Firefox 浏览器的中间件扩展以外，我们还开发了一款基于 Java 平台的 SDK。Java SDK 是提供给 Java 开发人员建立应用时使用的开发工具集合，开发人员通过调用 SDK 提供的接口，便可以与云存储服务器进行连接，从而使用云存储服务。Java 语言能够适用于当前很多应用的开发，如 Web 开发、Android

开发等，因此我们想通过提供 SDK 的形式，让现有的中间件系统支持更多的应用。

1）通信方式

服务器端的代码是由 Python 语言编写的，Java 与 Python 可以直接 Socket 进行通信，因此可以使用 Java 自带的 java.net.Socket 工具包。相较于 WebSocket 来说，Socket 与服务器进行连接无需通过握手协议，因此，可以在建立连接后直接向服务器发送请求。用户在使用 Java SDK 时，每调用一个接口便会开启一个线程来处理用户的请求，由于需要处理服务器反馈回来的信息，因此我们选用了 java.util.concurrent.Callable 类作为线程类的接口。实现的核心代码如下：2）

```
//通过服务器地址和端口号简历 socket 连接
socket= new Socket(SOCKET_HOST, SOCKET_PORT);
//向服务器发送请求
PrintWriter mwWriter = new PrintWriter(socket.getOutputStream());
mwWriter.write(this.request);
mwWriter.flush();
//接收来自服务器的响应
InputStream mwInputStream=mwSocket.getInputStream();
InputStreamReader isr = new InputStreamReader(mwInputStream);
BufferedReader mwBufferedReader = new BufferedReader();
String info=mwBufferedReader.readLine();
```

2）接口设计与实现

中间件 Java SDK 的设计目的是使 Java 开发人员集成我们云存储服务器到 Java 应用中去，从而使应用能够使用云存储服务。在设计 Java SDK 的接口时，我们参照现有的中间件 CLI 客户端接口，确保了 Java SDK 开发工具提供的接口与其保持一致，除了原有的中间件系统提供的基本接口外，还加入了优化后实现的一些高级接口，接口的定义和实现如下所示：

```
//接口定义
public interface IMiddlewareAPI {
    //用户登录接口
    String getAccessToken(String userID, String pwd);
    ……
}
```

```
//接口实现
```

```
public class MiddlewareAPI implements IMiddlewareAPI{
   @Override
     public String getAccessToken(String userID, String pwd) {
          //构建登录请求
          String request =
         Constant.GET_ACCESS_TOKEN.replace("#email#",
         userID).replace("#password#", pwd);
          //发送请求
          String result = this.sendRequest(request)
          //处理请求
          JSONObject jsonObject = JSONObject.fromObject(result);
          token = (String) jsonObject.get("access_token");
          return token;
     }
}
```

在 IMiddlewareAPI 类中定义了 Java SDK 提供的所有接口，包含了 CLI 客户端所有的功能以及优化后扩展的功能。为保证与 CLI 客户端的特性一致，相同功能的接口参数和返回值与其保持一致。在 MiddlewareAPI 类中实现了所有接口，实现接口一般分为三个步奏，首先构建请求，通过引用 Constant 常量类中预定义好的请求模式，替换相应的参数值；然后新建一个线程，发送用户请求；最后处理服务器返回的信息，将其转化为标准的 Json 格式。

3）部署与使用

Java SDK 的使用一共分为五个步骤：

1. 新建一个 Java 项目，并创建 lib 文件夹；
2. 将 Java SDK MiddlewareSDK.jar 拷贝纸 lib 目录下；
3. 将 lib 目录下的 MiddlewareSDK.jar 添加到 Build Path 中；
4. 实例化 MiddlewareAPI 对象

   // MiddlewareAPI mwAPI = new MiddlewareAPI("kaeyika@163.com", "12356");

5. 调用 API 接口

   // String result = mwAPI.getQuota();

、

第六章 云存储中间件系统优化测试分析

本章主要介绍中间件系统优化实现后，对系统进行全面测试。测试主要分为两个部分，一个是功能测试，即保证分析中所提到的优化项能够正常运行。另一个是优化后的系统性能测试，主要测试的是系统的稳定性和健壮性，同时与原有的中间件系统进行纵向比较，体现优化后系统的特性。同时与同类型的产品性能进行横向比较，凸显优化后中间件系统的优势。

6.1 测试环境部署

硬件环境	具体参数
计算机型号	Dell Inc. OptiPlex 990
处理器型号	Intel(R) Core(TM) i5-2500S CPU @ 2.7GHz
内存大小	10.0 GB RAM
硬盘总大小	750 GB
操作系统	Ubuntu 14.04
测试软件名称	middleware
测试软件版本号	0.94
测试服务器	IP：192.168.7.62

图 6-1 测试环境表

6.2 测试目的

本文对中间件优化的内容为基于前期的成果，在 Linux 平台上，针对需要改善的问题，综合运用前期的技术积累，优化并完善平台框架和 API 支持机制，以支持更多的桌面 OS 组件使用云备份与云存储服务。为了确保优化在功能需求与性能需要，对云存储与云备份中间件进行了测试。

具体测试目的如下：

1. 测试已实现的中间件是否达到任务及设计的要求，包括：各个功能点是否以实现，业务流程是否正确；
2. 任务规定的操作和运行稳定；
3. Bug 数和缺陷率控制在可接收的范围之内。

6.3 功能测试

中间件系统的功能测试就是对各优化项进行验证，设计一套完备的测试用例，逐项测试，检查是否达到设计要求的功能。由于优化的功能复杂且测试用例较多，所以我们选取了每个优化项中具有代表性的核心功能进行分析。

6.3.1 持久会话管理功能测试

持久会话管理的主要功能是将用户的命令会以会话的形式保存在中间件，当中间件系统出现异常时（如网络中断、中间件死机），会在下次重启中间件时自动恢复执行未完成的会话。此外，对会话信息进行了物理隔离，并且以加密的形式存储在数据库中，以提高用户信息的安全性。

测试用例一：

表 6-2 会话存储及加密测试

测试编号	Cloud-Opt-Test-001
测试内容	用户的请求是否以加密的形式存储在数据库中
测试用例	request = '-k -X POST -d \'{"password": "123456", "email":"kaeyika@163.com"}\' https://192.168.7.62:443/oauth/access_token'
测试结果	用户的请求以加密的形式存储在了 SQLite 数据库中

本测试主要对持久会话管理模块中会话存储及加密进行了测试，测试用例来自用户的请求，该用例表示的是用户名为 kaeyika163com，密码为 123456 的用户请求登录。优化后的中间件系统首先会调用请求加密函数，对会话进行加密，然后调用数据库层的服务，将加密后的请求存入到数据库。测试结果图下图所示：

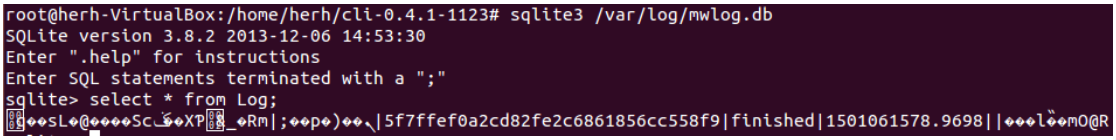

图 6-1 会话信息加密测试图

从测试结果的截图中可以看出，请求已经被存在 SQLite 数据中，数据库的路径为/va/log/mwlog.db。我们可以看到，用户的请求是以乱码的形式存储在数据库中的，这很好地验证了对数据进行加密存储的需求。

6.3.2 大文件上传功能测试

大文件上传利用服务器提供的高级功能，实现了大文件分块上传功能，在受到网络状况受到影响的情况下，若文件上传中断，会自动启用断点续传功能，避免数据重传。测试中，我们分别对网络状况正常和出现异常两种情况进行测试。

在正常的网络状态下，超过规定大小的文件将被分割成固定大小的文件块，然后逐一上传至服务器，等所有文件块上传完之后，调用服务器提供的合并文件结构，完成大文件在服务器端的合并。在网络异常的状态下，中间件需要记录上传的断点信息，然后在下次网络连接正常后，根据断点信息实现续传。

测试用例二：

表 6-3 大文件分块上传测试

测试编号	Cloud-Opt-Test-002
测试内容	大文件分块上传
测试用例	request = -k -X PUT -T /home/herh/test.docx "https://192.168.7.62:443/v1/AUTH_kaeyika163com/end1/bigtest.docx?op=CREATE&overwrite=&metadata=&mode=ENCRYPT" -H "X-Auth-Token:G6i7w90RCH6DoL0ZNv85rORwNBsThSF6wyouYq8s"
测试结果	大文件分块后逐一上传至服务器，并在服务器端合并

本测试主要针对大文件分块上传功能，请求表示用户名为 kaeyika163com 的用户上传一个大文件 test.py 文件至服务器，上传至服务器的路径为/biye/test.py。测试的结果如下图所示：

图 6-2 大文件分块上传测试图

从测试结果图中输出的日志文件我们可以看出，test.py 文件按用户 id 信息及时间信息被分成大小相等的 4 个文件块，最后一个文件块的大小不足 4M。每个文件块都由一个上传文件的进程单独上传到了服务器，并且最后调用的服务器端的合并文件接口。通过对服务器端文件的查询，我们可以看到，文件块被合并成了一个完成的文件。

6.3.3 本地缓存功能测试

本地缓存主要利用现代计算机普遍较大的存储空间以及高性能的计算能力，实现了本地缓存管理，进一步减少了系统 I/O 操作，提高了中间件性能。同时，

利用服务器提供的个人存储在云端的数据信息，实现了针对单一用户的“秒传”功能。测试中，我们主要针对存在本地或是服务器缓存的情况下，记录文件下载或上传过程中调用的接口和传输的效率。

测试用例三：

表 6-4 本地缓存测试

测试编号	Cloud-Opt-Test-002
测试内容	本地缓存测试
测试用例	request = -k -L "https://192.168.7.62:443/v1/AUTH_kaeyika163com/end/test.docx?op=OPEN&offset=&length=&version=&mode=" -H "X-Auth-Token:G6i7w90RCH6DoL0ZNv85rORwNBsThSF6wyouYq8s" -o /home/herh/end.docx request = -k -L "https://192.168.7.62:443/v1/AUTH_kaeyika163com/end/test.docx?op=OPEN&offset=&length=&version=&mode=" -H "X-Auth-Token:G6i7w90RCH6DoL0ZNv85rORwNBsThSF6wyouYq8s" -o /home/herh/mend.docx
测试结果	第一次下载文件时正常下传，第二次从本地复制

本测试针对本地缓存设计，测试用例由两个相同的下载文件请求构成。该请求表示用户 kaeyika163com 下载服务器端的/end/test.docx 文件至本地的/home/herh/end.docx。测试的结果如下图所示：

```
-----mwDownloadCache-----
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   197    0   197    0     0    983      0 --:--:-- --:--:-- --:--:--   989
has cache and copy......

-------------------------------------

There is a copy locally and copy it to the download path!
```

图 6-3 本地缓存测试图

从测试结果图中输出的日志文件我们可以看出，第一次下载文件时，正常调用了下载文件接口，并将文件保存到了下载路径。再次下载该文件时，并没有调用下载文件接口，而是直接从本地缓存中将该文件拷贝到了下载路径，下载所需的时间相较于正常下载任务来说，可以忽略不计，这大大地提升了文件下载的效率。

测试用例四：

表 6-5 服务器去重测试

测试编号	Cloud-Opt-Test-002
测试内容	本地缓存测试
测试用例	request = -k -X PUT -T /home/herh/test.docx "https://192.168.7.62:443/v1/AUTH_kaeyika163com/end/test.docx?op=CREATE&overwrite=&metadata=&mode=" -H "X-Auth-Token:G6i7w90RCH6DoL0ZNv85rORwNBsThSF6wyouYq8s" request = -k -X PUT -T /home/herh/test.docx "https://192.168.7.62:443/v1/AUTH_kaeyika163com/end1/test.docx?op=CREATE&overwrite=&metadata=&mode=" -H "X-Auth-Token:G6i7w90RCH6DoL0ZNv85rORwNBsThSF6wyouYq8s"
测试结果	第一次上传文件时正常上传，第二次从服务器复制

本测试针对服务器端的去重测试，测试用例由两个相同的上传文件请求构成。该请求表示用户 kaeyika163com 上传本地文件/home/herh/test.py 至服务器端/biye/test.py。测试结果如下图所示：

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   114  100   114    0     0    910      0 --:--:-- --:--:-- --:--:--   919
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    60  100    60    0     0    299      0 --:--:-- --:--:-- --:--:--   303
has cache and copy......

-----------------------------------

{"status": "0", "msg": "txe5abf1f629cc457da111ea028a75272c"}
```

图 6-4 服务器端去重测试图

从测试结果图中输出的日志文件我们可以看出，第一次上传文件时，正常调用了上传文件接口，将文件上传至了服务器端的指定路径。再次上传该文件时，并没有调用上传文件接口，通过服务器端去重校验，发现服务器存在该文件，所以调用的是服务器端复制文件的接口。该接口的执行时间很短，因此实现了“秒传”功能。

6.3.2 多应用适配功能测试

多应用适配为实现中间件系统对不同客户端请求的适配，支持更多的客户端

类型，实现系统的跨平台性。我们开发了一款基于 Firefox 浏览器的扩展工具，用户可以利用扩展使用云存储服务，并提供了 Java 平台的软件开发包（SDK），Java 开发者可以利用该开发包提供的应用接口 API 来集成我们的云存储服务。测试中，我们对通过 Firefox 扩展和 Java 开发包用户的功能测试，验证优化后的中间件系统能够适配各种类型的客户端。

测试用例五：

表 6-6 多应用适配测试

测试编号	Cloud-Opt-Test-002
测试内容	本地缓存测试

测试用例	//Firefox 登录测试
	firefox_request = '-k -X POST -d \'{"password": "123456", "email":"kaeyika@163.com"}\' https://192.168.7.62:443/oauth/access_token'
	//Java SDK 登录测试
	MiddlewareAPI middlewareAPI = new MiddlewareAPI("kaeyika@163.com", "123456");

测试结果	Firefox 浏览器与 Java 开发包登录成功

本测试同时测试了 Firefox 扩展以及 Java SDK 的登录功能。Firefox 通过 WebSocket 向服务器发送请求，Firefox 扩展的请求由“firefox_”标识。Java SDK 通过 Socket 向服务器发送请求，并有“java_”标识。通过身份认证之后，便可使用云云存储平台提供的接口。测试结果如下图所示：

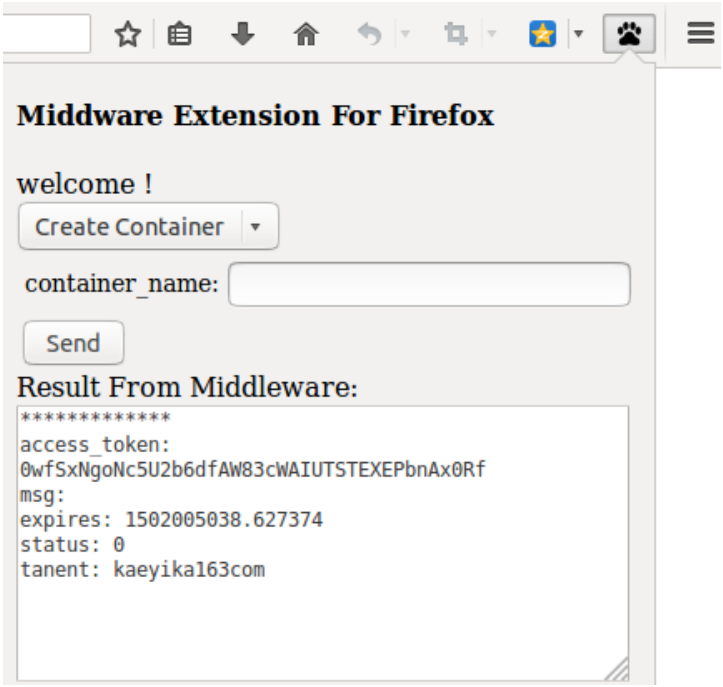

图 5-3 Firefox 扩展登录测试图

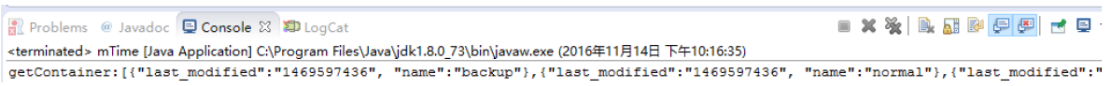

图 5-3 Java SDK 测试图

功能都通过了测试，能够正常登录并连接云存储服务器。此外，我们对这两个客户端的其他接口都进行了详细的测试。图 5-3 所示的是 Firefox 扩展登陆功能，用户点击插件图标后，会弹出登陆界面，输入正确的账户信息后会会将服务器返回的登陆成功信息展示在插件界面上。图 5-3 所示的是 Java SDK 的测试结果，开发人员将 Middleware.jar 导入到工程中后，调用登陆接口创建中间件接口实例对象，然后调用获取存储空间接口，返回的是 Json 格式的数据。同时，Firefox 扩展以及 Java SDK 实现了与原有的 CLI 相同的功能，且具有一定稳定性、安全性和鲁棒性。

6.4 性能分析

6.4.1 安全性分析

1）会话信息安全性

为确保中间件系统的安全性，原有的中间件系统在文件传输过程中，系统采用了 128 位的 SSL 加密技术，在文件的存储形式上，让用户主动选择是否加密，使用的是 AES 加密算法。然而原有中间件系统是以明文的形式保存用户的登录信息、会话信息等敏感数据的。这很有可能造成用户隐私的泄露。

优化后的中间件系统对系统的安全性进行了更为全面的考虑。为了保证存储在数据库中用户会话信息不被窃取，我们对会话信息进行了加密。具体实现如下：

```
//对用户的请求进行加密处理
enreq = mwEncrypter.sAES_str_encrypt(clientrequest, uuid)
//构架插入数据的 SQL 语句
sqlString = "insert into Log values(?,?,?,?,?,?);"
//调用 Dao 的 save 方法，将加密后的会话信息存入数据库
dao.save('/var/log/mwlog.db', sqlString, (buffer(enreq), uuid,
"unfinished", t, userid, ""))
```

2）本地缓存安全性

优化后的中间件系统新增了本地缓存的功能，旨在解决同一用户多次从服务器下载同一文件而导致的网络带宽浪费的问题。在现有的计算机系统中，存在一台终端被多人使用的情况，因此我们需要对不同的用户的缓存数据进行物理隔离，

防止某一用户的缓存信息被其他用户窃取。具体实现如下：

```
//根据用户名创建缓存文件夹名称
cache_path = sys.path[0] + "/mwcache/cache" + "/" + userid
if not os.path.exists(cache_path):
//文件夹不存在时创建
    os.makedirs(cache_path)
//修改文件权限，S_IRWXU 代表该文件所有者具有可读、可写及可执行的权限
    os.chmod(cache_path, stat.S_IRWXU)
```

3）数据库安全性

原有的中间件系统在执行数据库操作时，采取的方式是建立临时的数据库连接，并编写相应的 SQL 语句，最后执行数据库操作。

```
//建立数据库连接
sql_connection = sqlite3.connect('/var/log/mwlog.db')
//执行数据库操作
result = sql_conn.execute("select request from Log where state = 'unfinished' and userid='" + userid + "' and time<'" + t + "';")
```

这样的操作存在一定的安全隐患，如 SQL 注入攻击。SQL 注入攻击是黑客对数据库进行攻击的常用手段之一，程序员在编写代码的时候，没有对用户输入数据的合法性进行判断，使应用程序存在安全隐患。用户可以提交一段数据库查询代码，根据程序返回的结果，获得某些他想得知的数据，这就是所谓的 SQL Injection，即 SQL 注入[12]。优化后的中间件系统对数据库的操作进行的全面优化，不仅分离了数据库层操作，更是改变了执行数据库操作的方式。优化后的中间件系统使用了占位符的方式来抵御 SQL 注入攻击。

```
//使用占位符的方式构建 SQL 语句
sqlString = "insert into Log values(?,?,?,?,?,?);"
//调用 Dao 的 save 方法，执行 SQL 语句
dao.save('/var/log/mwlog.db', sqlString, (buffer(enreq), uuid, "unfinished", t, userid, ""))
```

6.4.2 稳定性分析

1）系统稳定性

中间件系统可以同时处理来自不同用户的请求，同时用户也能批量地发送请求给中间件。因此用户的请求不会立马被执行，而是以缓存的形式保存的。当用

户注销或是中间件系统遇到故障之后，优化后的中间件系统会在中间件系统重启后，自动执行未完成的任务。保证用户提供的请求不会丢失。此外，我们通过大量的压力测试、边界值测试增强了中间件系统的鲁棒性。当用户执行错误的操作时，会抛出相应的异常，而不是强制退出。

2）传输稳定性

网络环境受到很多因素的制约，云存储平台是以网络作为基础的，尤其是上传和下载操作。优化后的中间件系统充分地考虑到了网络的不稳定因素，当出现网络故障时，会记录断点信息，下次上传文件的时候，会根据断点信息进行续传，这大大地提升了数据传输的稳定性。

6.5 对比分析

中间件优化的主要目的是基于前期的中间件系统，同时研究云储存服务中的关键技术，对其功能和性能两方面进行优化。本文将从纵向和横向两个方面，将优化后的中间件系统与原有的中间件系统和市场上同类产品进行比较分析。

6.5.1 原中间系统对比分析

表 6-7 与原中间件系统对比分析表

	原有中间件系统	优化后中间件系统
安全性	传输、存储加密。	传输、存储、会话加密； 本地缓存权限管理； 数据库操作防注入攻击。
稳定性	网络不稳定时无处理； 中间件异常会话丢失； 能处理一定的异常。	对网络异常进行了处理； 中间件异常会话保留，且在下次启动时自动自行未完成的会话； 优化了异常处理机制。
功能特性	数据传输； 数据加密； 数据压缩。	数据传输； 数据加密； 数据压缩； 持久会话管理； 大文件上传； 本地缓存； 多应用适配。

表 6-7 展示了原有的中间件系统与优化后的中间件系统的对比结果。从安全性上来说，原有的中间件系统对数据在传输过程中和存储在服务器时两种状态进行了加密，确保了一定的安全性。优化后的中间件系统在此基础上做了进一步的改进，对保存在中间件系统中用户信息进行了加密存储。此外，对用户存储在本

地的缓存文件通过权限设置进行了物理隔离，确保了缓存数据的安全性。同时，规范了数据库层的操作，可以抵御一定的攻击。

在系统的稳定性上，原有的中间件系统只做了简单的异常处理。优化后的中间件系统对网络状况、系统自身的稳定性、异常处理三个方面进行了改进，能够在遇到网络异常后 断点续传、中间件系统出现异常后用户请求不会丢失、提供更多的异常处理机制。

功能上，原有的中间件系统主要提供基础的数据传输功能，一共分为三个模块，分别为数据传输模块、数据加密模块、数据压缩模块。优化后的中间件系统通过对云存储关键技术的调研，新增了持久会话管理、大文件上传、本地缓存以及多应用适配四个模块，使得当前的中间件系统更加符合云存储服务的发展趋势。

6.5.1 同类产品对比分析

中间件系统关键技术的优化主要基于对同类产品的调研，我们对国内外的一些云存储平台进行了详细的分析，从而得出了原有的中间件系统需要优化的内容。经过设计与开发，现有的中间件系统在一些重要的功能上已经达到了成熟产品的要求，此外，一些新的特性具备一定的优势。

在系统的安全性上，优化后的系统对缓存在中间件的用户登录信息、会话信息进行了加密，使用的是与其他同类产品类似的加密方式，采取了 128 位的 AES 加密算法。该加密算法相较于 258 位的 AES 加密算法，128 位的加密算法能够节省大约 40%的加密时间，同时能够保证数据的安全性，因为它足够复杂无法被暴力破解。

在数据去重方面，百度网盘和 360 云盘都采取了全局去重的方式，Dropbox 使用了单一用户去重，而 Google Drive 未进行系统去重处理。优化后的中间件借鉴了 Dropbox 的去重方式，系统只支持单一用户的去重，也就是说即使服务器中的其他用户存在当前用户待上传的文件，也不会将其他用户的文件拷贝至当前用户的上传路径下。这样能避免由服务器端全局去重而导致的侧信道攻击文集，保证了系统在一定程度上能够缓解服务器存储压力，又确保了系统的安全性。

优化后的中间件系统支持大文件的断点续传功能，当遇到网络故障的时候，会在下次上传任务开始时，根据断点信息进行续传，当前主流的云存储平台都支持此项功能。

调研的结果显示，目前百度网盘、360 云盘已经 Dropbox 和 Google Drive 都不支持本地缓存。也就是说当用户反复下载同一文件时，需要多次从服务器端下载同一文件，这极大的浪费了网络带宽。优化后的中间件系统在客户端本地建立了一个缓存文件，记录文件的下载信息和数据，以解决多次重复下载同一文件的

问题。

为适应当前平台多元化的格局，我们开发了两款云存储客户端，分别是 Java SDK 和 Firefox 扩展工具。同类的云存储服务器平台大多支持主流的操作系统的平台，如 Windos，IOS，Linux 等。优化后的中间件系统同样具备一定的可扩展性，Java SDK 可用于与 Java 相关的应用，如 Web 开发和 Android 开发。Firefox 扩展工具相较与其他客户端来说，更加轻量级，且访问更加便利。

第七章 总结和展望

7.1 总结

随着云存储行业技术的不断革新，云存储服务在安全性、稳定性以及可扩展性上有了大幅度的提升。然而由国家“核高基”科技重大专项自主研发的一款云存储服务平台只提供了基础的数据存储服务，已经跟不上运存储领域发展的趋势。本文在通过对原有的中间件系统进行了系统的分析，结合业内成熟的产品，对云存储中间件的关键技术进行了设计和优化。

本文首先对原有的中间件系统和市场上成熟的产品进行了分析，得出了中间件优化的意义和必要性，确立了研究的内容。随后，对实现过程中需要用的技术进行了详细的介绍，为后续的系统实现提供了技术支持。紧接着，详细地阐述了原有的中间件系统的不足，并引出了需要进行优化的具体内容。接下来的工作中，我们细致地介绍了中间件系统核心技术优化设计过程，包括系统设计的整体框架、流程设计和接口设计等，并通过核心代码的分析介绍了系统实现的细节。最后，搭建了系统测试环境，分析了中间件系统优化后的性能。

云存储中间件关键技术的优化只要是对原有的中间件系统进行改进，内容包括四个方面。持久会话管理主要解决存储在中间件的用户缓存信息，包括用户登录状态信息和用户个人信息，本系统使用了 128 位的 AES 加密算法对这些隐私信息进行了加密，并保存在本地的 SQLite 数据库中，同时提供给用户增、删、改、查会话的接口；大文件上传解决了因网络故障而导致的频繁的重传问题，可以根据断点信息进行文件续传；本地缓存主要包含两部分的内容，客户端缓存和服务器端缓存。客户端缓存利用本地计算机的存储能力和计算能力，对下载的文件进行缓存管理，使得用户频繁地下载同一文件时，可以利用本地缓存节省网络带宽。服务器端缓存利用服务器提供的存储数据信息，实现了“秒传功能”；多应用适配模块解决了原有中间件系统缺乏可扩展性的问题，本系统同时研发了两款客户端，Java SDK 和 Firefox 扩展，以支持更多平台的可扩展性。

7.2 展望

本文针对云存储中间件关键技术做出的优化，大体符合了当前云存储行业发展的趋势，与同类成熟的云存储产品在功能上基本保持了一致性。然而，仍存在一些挑战，需要在后续的工作中对中间件系统进行不断的完善。

目前，该款云存储服务平台没有投入商用，只是适用于内部小范围的使用。与其他同类产品相比，没有经历过市场和用户的检验，而测试受硬件、人员条件的限制，虽然对基本功能进行了较为完备的测试，然而在压力测试、边界测试上具有一定的局限性，因此需要在后续的工作中通过搜集用户使用时的数据以及反

馈信息，对中间件系统进行持续优化，通过系统的稳定性。

此外，优化的后的中间件系统虽然具备了一定的可扩展性和跨平台性，并提供了 CLI、Java SDK 和 Firefox 扩展三个客户端，分别支持在 Linux 平台下、移动端和 Web 端使用中间件，但是相较于其他同类产品，支持的客户端类型较少，在未来的工作中，可以考虑实现 Android 和 IOS 两大当今主流的客户端，以支持更多的用户使用。此外，当前支持的客户端只实现了基本的功能，在性能方面需要进行更为深入的研究。

最后，本文只对云存储服务中最为关键的技术进行了优化，随着云存储技术的不断革新，以及人们日益增长的需求，云存储服务平台的高级功能越来越多，如在线预览、视频播放、自动备份等，这要求云存储中间件系统做持续的更新和优化，在未来的工作中，我们会持续调研云存储行业关键技术的发展趋势，使这款自主研发的云存储系统能最大程度地满足用户的需求。

参考文献

致　谢

能够顺利完成毕业论文，首先想要感谢的是我的研究生导师韩伟力老师。从论文的选题到项目的实施，以及最后毕业论文的撰写，韩老师都给予了我最大的支持和帮助。在项目实施的过程中，韩老师悉心地对我进行了指导，每当我遇到问题时，都会为我指明方向，提供技术支持。在论文的撰写过程中，韩老师严谨的学术作风，一丝不苟的工作态度一直敦促着我不断修正、完善论文。除了在学习上对我的指导以外，韩老师在生活上也给予了我细致的关怀，在这里，我想再次向韩老师表示由衷的感谢。

其次，我还要感谢实验室的小伙伴们，在两年半的学习生活中，你们给了我太多的支持和感动。生活中我们一起谈天说地，学术上我们一起共同探讨，特别是在我撰写论文期间，无私地给予我帮助，让我感受到了实验室的友爱。能够与你们同处一个实验室，是一件很幸运的事。

此外，我想要感谢复旦大学以及学校的各位老师，给了我一个很好的平台，让我有机会参与一些重要的项目和会议，这对我的影响十分深远。此外，我还能够与其他优秀的同学一起学习，一起进步。

最后，感谢各位评阅老师对本文给出的宝贵意见。

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：刘武　　日期：2017.07.02

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：刘武　导师签名：韩伟力　日期：2017.07.02