# 基于分布式存储的污染源数据平台的设计与实现

摘要：目前随着互联网的发展，用户，企业已经实现了资源的共享，共享的内容包括了计算能力，存储能力，网络传输，数据服务等。随即出现了云服务，相比较而言，云服务比个体服务的搭建有诸多的优势，如果一个中小企业需要相应的云服务，通过自己构建机房搭建平台，无疑不是个明智的选择，因为一套可以上线的服务需要付出巨大的开发成本和维护成本以及硬件成本，明显已经超过了服务所能创造的价值，当互联网共享云服务出现了，给企业带来了很大的便利，用最少的成本创造最大的价值。

云平台主要使用Hadoop开源组件，首先Hadoop实现了一个分布式文件系统，具有较高的容错能力和高可用性能，云存储主要使用了Hadoop的HDFS海量数据存储的能力，将许多廉价的机器经过整合管理变成具有超大处理性能的云存储系统，在互联网中，云存储是广大客户可以共享的一种存储服务，面对的客户大到企业，小到个人，提供的服务具有较好的安全性，可靠性，易管理性等等优点；Hadoop是通过java语言实现，我们在开发过程中可以通过java应用程序调用相关的API，并封装成相应的功能，实现云存储服务。

本研究的主要意义在于如何将现有的分布式存储运用到实际应用当中，在世纪生产环境当中，仍然有很多从云存储基本功能基础上延伸出来很多的新功能，比如ftp服务器对接云存储，这项功能主要是让原有的ftp服务器底层对接windows系统，通过该写相应的特性，让底层的文件系统对接云存储文件系统，而对于客户来说，仍然觉得云存储文件系统和windows文件系统一样，由文件和子文件以及属性一样，实际上在云存储中也不存在云存储文件系统，这是就需要约定一套规定来对云数据进行翻译，从而实现和客户进行完美的交互；其次是对于海量数据的存储，如何设计性能更好，更符合生产环境下的存储系统。

关键字：云存储，Hadoop，HDFS，ftp。

／／xie

第一章引言

1.1背景研究

1.1.1背景介绍

大数据时代，我们需要存储的数据已经急剧增长，企业以及互联网的数据都在以每年50%的速率在增长，经过预测，到2020年，全世界的数据已将达到35ZB，等于80亿块4TB的硬盘，面对如此庞大的数据，我们的存储结构将面临巨大的挑战，这时非结构化数据存储将占很大一部分比例。互联网的发展，让数据产生的主体由企业转向了个人，对于个人来说，产生的很多数据都是图片，文档，视频等非结构化数据，这些数据都需要使用非结构化数据库来存储。对于这种分布式存储，物理介质分不到不同的地理位置，数据上传，可以进行就近上传，数据存盘时，根据映射算法，找到合适的IP，把数据刷到指定的磁盘。这种分布式数据存储，有效地解决了因为数据存储和访问造成的IO瓶颈，下载数据的时候，可以同时从多个数据盘上去取数据，上传数据的时候能够高并发存储，从单个数据流存储数据上看，映射算法会使用到负载均衡原理，有效地选择经验值最高的IP进行存储，大大提高了存储的效率。而且随着数据量的增加，我们需要从硬件上去扩大存储能力，这在集中式存储系统中是很难做到的，换更大的磁盘和内存盘，从硬件上来说，过程繁琐，操作艰难，从软件层面上来说，数据的转移相当困难，经济上也会付出更大的代价，大容量单盘比小容量多盘的成本高，于是分布式存储给使用者带来了福音，当我们需要增加平台的存储能力时，我们仅仅需要增加一台设备放到局域网中，通过管理软件Zookeeper的API把新的设备添加到集群中，并启动相应的负载均衡功能，让新的设备均摊集群承载的数据量，这些优良的性能和数据时代的大环境让分布式存储的盛行成为了必然。

1.1.2相关课题介绍

目前云存储平台主要是给其他应用程序提供数据的最底层的数据存储和数据产查询的服务，其实就是数据库的分布式实现，对外提供相应的接口供开发人员访问，对于本平台的研究的同时，我们还解决了对接云存储数据的ftp服务器，底层采用的是FTP协议，顾名思义就是文件传输协议，实现服务端的文件共享，在互联网上的计算机可能会使用不同的操作系统，计算机Unix，Dos，Windows，MACOS等等，为了实现不同系统之间的文件交流，我们就需要建立一个统一的文件传输协议，这就是FTP协议，这样用户就可以从别的用户那里获得文件，或者把文件传送给其他人。目前FTP主要采用了客户机／服务器系统，在服务端搭建一个提供文件共享的服务进程，用户通过一个支持FTP协议的客户端程序，连接服务端，发出相应的指令，和服务端实现具体的交互。本课题研究相关的FTP服务器就是将底层的数据存储对接在云端，对于云端，数据都以流的形式存在，这时候，文件系统的管理需要由HBASE来管理，每个流对应一项元数据，包括这一分片的大小，序列值，存储路径等等相关的属性，而对于真实数据流则直接实现刷盘，存在磁盘的目录树中，这样便实现了数据的高效率存储和管理，同时也充分利用了HBASE的列可以动态增加，如果列空，则不占用空间，同时，列数据增加，会进行自动的切割，使数据可以水平扩展，同时HBASE可以提供高并发读写操作的支持。

1.2研究现状

1.2.1国外研究现状

目前国外的云存储已经有一定的成功案例，首先Amazon公司是最先研究关于云存储系统的企业，目前在存储体系上也是最完备最成功的，云存储是一项能够给广大用户提供服务的平台，对于Amazon公司来说，它拥有远远超过它本身所需要的硬件资源，于是在06年将存储服务对外放开，作为AWS的一部分。目前Amazon提供的云存储服务价格低廉，性能稳定，是业界最受欢迎的存储服务，目前Amazon的简单存储服务支持上传的单个文件不可以大于5T；IBM在09年也推出了企业级智能云存储服务，主要通过存储虚拟化和基于私有云的存储技术，目前在提供应用程序方面得到了广泛的支持，每个企业的数据管理模式存在一定的差别，IBM可以根据企业的本身框架进行调整，通过虚拟化和自动化技术，构建属于企业自身的云存储系统，充分协调企业自身的硬件和软件资源，让资源统一分配，管理，部署，备份和监控，更加有效地实现资源的共享；EMC公司拥有一套基于策略的存储管理系统ATMOS，这是一套首次能把容量提高到PB字节的存储方案，通过全世界的云存储环境，把客户的那个大量的非结构化数据自动管理起来，并且为其提供自动的信息配置功能，使客户安全地构建和实现云存储服务，ATMOS的优势在于可以提供信息配送和处理，使用一套策略来建立云存储的不同层级，目前主要容量为120T，240T和360T；2010年夏天Google提出了GSD服务，该服务实现了数据可以在美国的若干个数据中心之间复制数据，组织方式提出了Bucket的概念，而且Google向外提供了管理工具和网络用户界面，完全放开存储服务；为了应对云存储的发展，HP提出了云存储融合架构，将网络，软件，资源和服务器统一管理起来，形成资源池，并通过模块化存储来实现容量和性能的提高。

1.2.2国内研究现状

目前国内的各大高校都在积极开展云存储的相关方案的研究，首先对于清华大学的Corsair系统，实现了分布式存储和数据共享，主要为学校师生提供个人数据存储，社区资源共享和公共资源的下载等，这个存储平台目前有几百个集团用户和一万六千多个人用户，大概使用100T左的存储空间；解放军理工大学的Mass云，这个云平台陈本较低，性能高校，在存储较多数据量的时候，性价比较低，而且通用API方便开发人员进行二次开发，使用成本较低。

国内的产业界也纷纷推出自己的云存储方案：

网易NOS深度整合了IaaS，PaaS以及各种容器技术，服务还包括了弹性计算Devops工具链和微服务基础等服务；腾讯COS对于开发者来说非常方便，性能安全，稳定，高效，访问主要是基于HTTP的方式进行访问；青云QingStor 对象存储服务提供了无限扩展的存储空间，存储高效，性能可靠；阿里OSS主要适合存放任意文件类型，对企业个性化服务来说是个不错的选择；华为云最大的特点是在于IT基础设施资源能够随着用户使用情况作弹性伸缩，而且华为云是经过行业认证和授权的专业平台，网络接入到配备管理具有7层安全防护，具有极高的稳定性和安全性。

1.3平台主要内容和结构

1.3.1实验需要解决问题

一直以来，我们做项目使用的都是关系型数据库，存储也采用的集中式，对于关系型数据库，处理起来非常方便，而且使用集中式存储一般来说硬件资源几种一个活着几个设备上，但是随着数据量的增加，动态资源的调整，这种集中式存储越来越不能满足我们的需求，实验的数据是实时从设备上采集而来，数据量也越越来越大，读数据的访问也越来越频繁，而且数据的安全性和扩展性要求也越来越高，这时候集中式的关系行数据库已经远远不能满足我们的需求，于是出现了分布式存储，将组件分布在网络计算机上，很多台计算机一起为外部提供服务，而且在同一个分布式系统中，计算机在空间部署上可以随意分布的，这些计算机可以放在不同的机柜，不同的机房甚至是不同的城市，因此这样的分布式系统基本没有逻辑上的约束；由于单节点的安全性得不到有效的保证，而分布式存储实现了三幅本保存，当一个节点的数据丢失的时候，可以通过心跳得知，并通过Namenode组织发出任务，从其他的副本中复制数据完成形成丢失的副本；除此之外，随着访问量的增加，数据访问出现了瓶颈IO，对于分布式系统，对于资源的请求可以实现多线程高并发访问，同时从不同的设备上访问多项数据，大大增加了系统的访问性能，从以上这些特性来看，分布式存储将会给我们的应用服务带来更加稳定高效的支持。

1.3.2论文主要研究内容

1.3.3论文的结构

第2章 云存储的介绍

21定义

云存储是从云计算中发展而来的一个提供网络存储技术，云存储涉及集群系统，网络技术和分布式文件系统，以往的单节点存储已经不能满足时代的需求，随即出现了一套能够将网络中的各种各样不同类型的存储设备管理起来，提供对外的数据存储和业务访问功能。它不是单一的某台设备，而是一整套完整的服务，通过分布式技术把一个个小的存储设备转化成对外提供的服务，这套服务包含了网络设备，存储设别，服务器，应用软件，公共访问接口，接入网和客户端程序等等多个部分。

22优势

有了云存储，传统的集中式存储在很多行业已经渐渐被代替了，云存储有着传统存储不可代替的优势。

221可扩展性

云存储的可扩展性，目前大多数企业在业务刚刚兴起的时候，很难预估未来的业务扩展量，不管是计算需求还是存储需求，增长速率也不可预测，这时候对存储服务的扩展性要求很高，对于云存储来说，集群是通过Zookeeper来管理的，Zookeeper提供对外API，能够方便快捷通过API调用相应的进程，把新的设备IP添加进来，这台设备只需要安装相应的操作系统和应用软件即可，并在网络中运行，这种动态容量扩充非常方便，不需要对原有的服务造成任何的影响，通过内部封装好的均衡负载功能，将新加进的设备均衡负载整体的存储量。

222安全管理

整个存储系统是一大批设备，传统的存储模式对于设备的管理尤其困难，对于每个设备的使用情况，各种指标的指数，尤其是当服务器或者磁盘出现故障了，这时候对数据的读取和写入会受到很大的影响，甚至造成数据的丢失，有了云存储久不存在这样的问题了，所有的数据都存在多个副本，当某个磁盘的数据发生丢失，这是可以通过心跳感知设备的异常，当发现异常，便可以调动相应的程序自动从其他的盘中拷贝数据形成新的副本，这样给管理人员带来了很大的便利，减轻了工作负担，大大提高了存储系统的维护效率；同时在进行设备升级的时候，可以在软件层面上操作，把旧设备上的数据进行拷贝转移，这样就可以在提供稳定服务的同时卸下旧的设备，把新的设备替换上，再将数据迁移回来，实现动态升级且保持服务的连续性。

223 设备类型广泛

目前设备的更新换代尤其快，一种新的设备在出厂后2年可能就被淘汰停产了，很难再买到相同型号的设备，以往的存储系统中，对磁盘的一致性要求非常高，在容量，厂家，型号方面必须完全一样，不然一起运行过程中就会出现不兼容的现象，这在互联网行业是一个极大的弊端，互联网行业是个业务不断复杂化，数据量不断增加的一个行业，这是我们就需要充分考虑增加业务量时付出的代价，对于非集中式存储，首先我们需要把数据拷贝到新的设备上，同时单台设备的性能提升需要付出高昂的代价，相对而言，相同的性能，多台设备比单台设备要廉价很多，而且添加新设备的同时，我们不需要限制新设备的规格，这样就可以选择大量性价比较高的设备，降低服务的成本，新增设备的时候，旧的设备仍然能够提供稳定的服务。

23应用场景

目前可以说云存储的应用场景比比皆是，典型是音视频应用场景，首先对于音视频点播类应用场景，对服务的性能要求比较高，播放需要非常流畅，而且在音视频处理上也有着很大的需求，对于音视频文件在服务端是需要进行相关的转码操作，有些类型的视频适合在web端播放，有些视频适合在移动端上传，这时我们需要把食品在服务端进行转码，对于上传的文件通常有些比较大，这时需要采用断电续传的原理保证上传的速度和稳定性，同时为音视频服务提供性能保障的是CDN节点可以提供缓存功能，对于某一点的高频访问数据，可以放在缓存中，这里的缓存有2层含义，首先对于单点访问，一级频繁访问数据会放在周围边缘节点的内存中，调用的速度最快，其次二级频繁访问数据会放在边缘节点的磁盘上，层层上升，节点数据量也越来越大，数据越来越全；数据传输过程中，数据会暴露在网络中，对于私有资源会进行加密，访问数据时需要下载相应的凭证，保证资源的私密性。

24 存储结构

云存储系统平台从顶层到底层主要包含了4个层次：

数据存储层、数据管理层、数据服务层、和用户访问层。

241数据存储层

首选数据存储层是服务平台的最底层进行数据处理的层，这层直接接触的是存储设备和网络设备，对于这些设备还对应了一个用来管理的系统，将这些硬件设备同意管理起来，包括设备的使用情况，设备的性能等等，以及设备的维护升级。整个云存储系统可以对外提供不同的存储服务，对于不同的数据类型，统一放在一个海量的存储池中。所有的数据都是对象，文件的结构由HBASE来管理，真实数据存在磁盘目录中，所有的数据的查找都是通过IP和对象的目录来查找的，一个文件的所有数据可能存放在不同的设备，不同的磁盘中，目前单服务器肯定是难以满足高并发访问的需求，P2P的数据结构要求比较高，在网络中需要大量的节点和科学的算法来保证数据传输的可靠性和质量。因此这种多元化存储方法能够解决这样的问题，构建不同地区的数据中心，在不同的区域可以访问周边的边缘节点的数据，当用户数量增大时，可以缓解中心服务器的IO压力，为客户带来更好的服务。

242数据管理层

数据管理层是整个云存储的核心，也是最复杂的部分。该层一般都是通过集群管理，存储方式通过分布式存储。在这一层中针对用户具体的需求，作出相应的扩展性功能，对一些私密性较强的数据，可以对数据进行加密，加密有不同的加密方式，针对不同的数据类型，可以采取不同的加密方式，以及备份和冗余等等。存储层层包含了很多重要的功能，对上层提供的服务也是各种各样，如果不进行有效的管理，上层调用数据层的功能很不方便，通过数据管理层的包装，可以把各种功能进行统一的管理，诸如用户管理，安全管理，副本管理以及策略管理等来实现对公共数据的统一管理，如此就屏蔽了存储底层的功能调用方式和实现机制，实现上层和底层的完美对接，让多个存储设备之间能够协调工作起来，更好地提供服务。

243**数据服务层**

这一层是存储服务直接面向客户的一层，在这一层**根**据具体的需求灵活地对服务进行相应的扩展。在这一层，存储平台暴露的接口已经形成了一定的标准，根据用户的不同，可以对接口做不同的实现，最终提供相应的服务。例如果数据存储服务，主要为用户分配相应的用户和空间，提供对象操作，根据用户使用服务的时间和对象操作的次数来进行相关的计费；公共资源服务，提供所有用户对一部分资源的访问可达性；多用户共享服务，比如一个公司的不同部门可能对公司的一部分数据需要共享，这时会根据用户定义共享组；数据备份服务，这里主要是为数据服务提供安全保障，当机器发生故障的时候，系统的可用性能够不受到影响。

244用户访问层

这一层是用户主要是用户能够真实可以接触的一层，关于云存储的相关的功能在这一层已经完全包装好，用户使用的仅仅是客户端或者URL，服务端分配不同用户相应的权限，而用户即可在任意点通过互联网登录系统，通过权限验证，进入存储系统，访问相关已经被授权的服务。

第三章 需求分析

31需求分析

32主要功能分析

33用例图分析

第4章云存储的核心技术

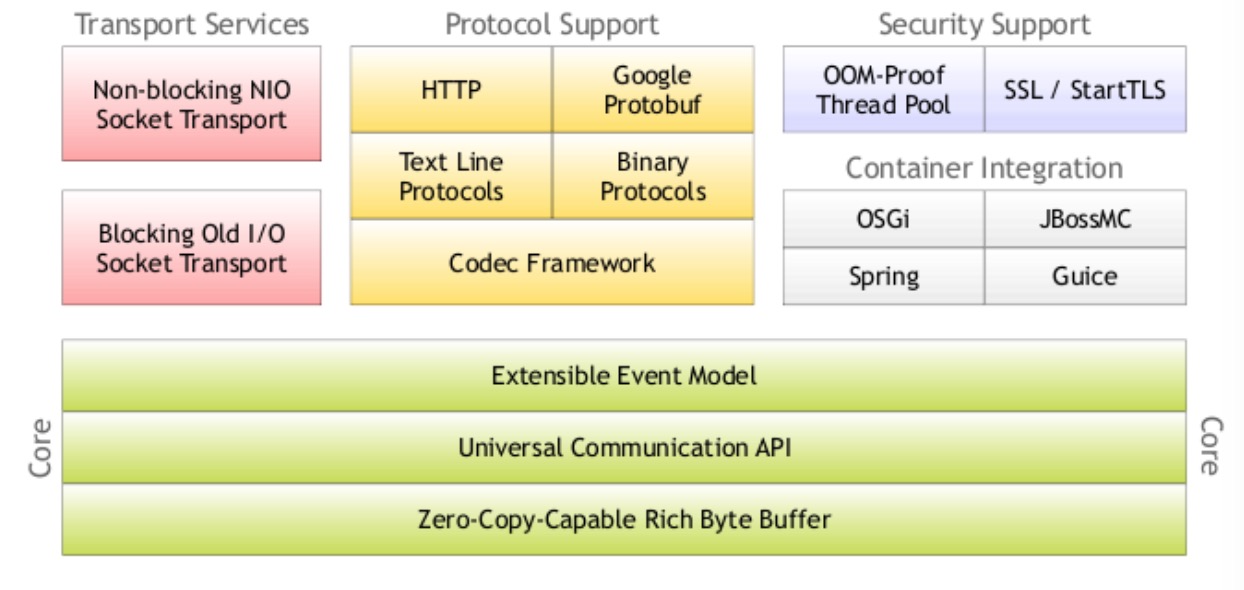
／／虚拟化，分布式存储，备份，压缩，安全，容错

第5章具体框架设计

4.1 netty框架的应用

4.1.1 netty 框架介绍

Netty是高效的Java NIO开发框架。它的总体结构如下：

从图中的最上层传输数据服务分为：阻塞模式和非阻塞模式。

阻塞模式基于TCP/IP协议，完整的服务包括服务端进程和客户端进程，在服务端申请ServerSocket进行，并注册监听端口，此时客户端可以建立Socket服务发起请求，自定对应服务的IP和端口，这时服务端可以通过ServerSocket的accept方法从请求队列中取出socket，如果此时没有建立好的链接，这时进程需要阻塞等待客户端的请求，对于已经建立好的链接，服务端和客户端可以通过输入输出流传送数据，对于这样的非阻塞模式一次只能处理一个请求，如果需要建立多个请求的时候，需要为每个请求分贝建立一个线程，这种模式可能会有一些线程处于等待状态造成资源浪费，线程数量越来越多，线程之间的切换也会带来越来越多的性能开销。

非阻塞模式是单线程操作，Selector用来实现非阻塞，其中的时间通知API可以确定一系列套接字中哪些已经处于就绪状态。由于在任意时间可以检查任意读操作和写操作的完成状态，这样就可以通过单个线程同时操作多个并发的链接，大大提高了内存的使用效率，使用较少的线程处理较多的请求，节省了上下文切换造成资源消耗，而且在高负载的情况下，可靠和高效地处理和调度IO操作不容易出错，这些复杂的处理都交给了Netty来处理。

现在Netty是一种采用基于NIO类库的异步通信框架，采用非阻塞模式，基于事件驱动，高性能，高可靠性的通信模式。

NIO采用Reactor模式，一个Reactor线程聚集了多路复用器Selector，它可以同时注册监听和轮训上百上千个Channel，一个线程就可以同时轮训处理多个客户端链接，线程的数量也可以是多个，但一般不会超过CPU个数+1，通过这种方式可以大大控制I/O线程之间上下文的切换和竞争，CPU的的利用率的到了很大的提高。所有I/O

操作是异步的，业务线程直接进行IO操作，也不会发生阻塞，系统不再依赖于外部的网络环境和外部程序的处理性能。这种非阻塞模式可以给系统带来很大的可靠性和性能提升。

Netty从逻辑上分为三层：

第一层是Reactor通信调度层，由一系列辅助类组成，包括Reactor线程NioEventLoop及其父类，NioSocketChannel／NioServerSocketChannel及其父类，ByteBuffer以及衍生出来的各种Buffer，Unsafe以及衍生出来的各种子类。

第二层是职责链ChannelPipeLine，主要负责上层调度时间在职责链中的传播，可以动态编排职责链，职责链可以拦截制定的时间，对其他的IO和事件忽略，Handler对inbound和outbound事件都支持。

第三层是业务逻辑编排层，这一层一般有单纯的业务逻辑编排和应用层协议插件，用于协议相关的编解码和链路管理。

Netty还可以实现数据在内存之间的零拷贝，即通常用户调用文件资源的时候，首先需要把文件拷贝到用户空间，而Netty可以直接在内核空间中传输到网络。Netty在进行接收和发送ByteBuffer使用的是直接buffers，使用非堆内存进行Socket的读写，不需要进行字节缓冲区的拷贝。如果使用传统的堆内存进行Socket的读写，就需要虚拟机首先把堆内存拷贝到直接内存中，然后才写入Socket中，这就多了一次缓冲区的内存拷贝。Netty使用了组合Buffer对象，可以聚合若干个ByteBuffer对象，用户可以像操作Buffer一样对组合Buffer进行操作，可以避免通过传统的方式将多个小的Buffer和并成一个大的Buffer。Netty允许我们将多段数据和行为一段虚拟数据供用户使用，这个过程也不会发生拷贝操作。Netty对数据的访问可以随机访问，根据index定位数据的位置，实现随机读写，同时为需要传输的数据拟定了统一的ChannelBuffer接口。

Netty的零拷贝是通过CompositeChannelBuffer来实现的，它的作用是将多个ChannelBuffer组成一个虚拟的ChannelBuffer来进行操作，这种虚拟的ChannelBuffer并没有将多个ChannelBuffer真正地合并起来，知识保存了数据的引用，这样就避免了数据拷贝；其中readerIndex和writeIndex是从AbstractChannelBuffer继承而来，components是一个ChannelBuffer类型的数组，它可以保存这个虚拟的Buffer的各个子Buffer，indices是一个int类型的数组，它保存的是各个Buffer的索引值，lastAccessedComponentId保存最后一次访问的子Buffer的ID，总的来说CompositeChannelBuffer就是一系列buffer通过数据结构包装起来，通过接口对外提供服务，让整体的Buffe组看起来就像是单个Buffer一样。

文件传输采用的是transterTo，它可以将文件缓冲区的数据发送到目标Channel中去，避免了传统的通过循环write方式导致的数据拷贝问题。

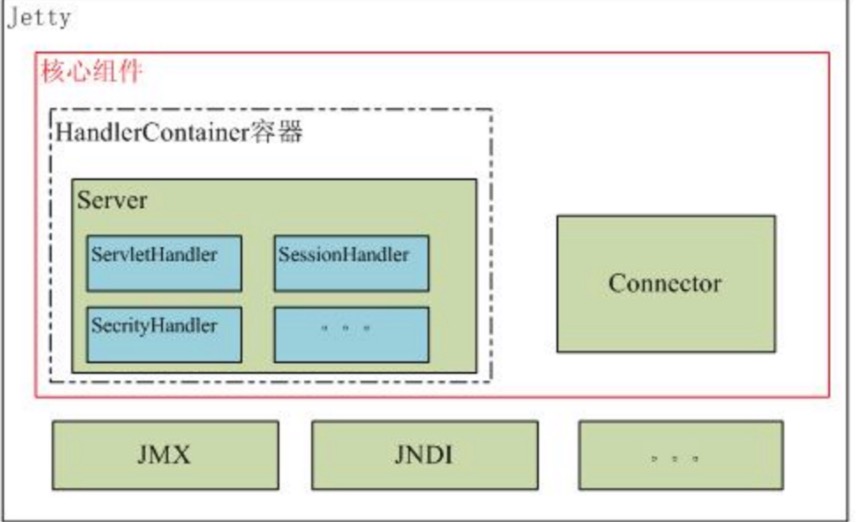
4.1.2 netty框架的应用

在我们的云存储中，Netty主要担任数据流读写的功能，客户发送的请求首先经过Cos-server的处理，将对应的数据切片，刷盘等交给IO-Server处理，这里包含了数据配片，高并发刷盘，消息队列，路由选择算法等，通过这一套服务，大大提高数据流的存储性能。对于每个大对象会进行分片，每个小的分片作为一个存储单位，并且存三份在不同的磁盘上，有效防止当机造成的数据丢失；对于设备的选择，使用了一套科学的路由选择算法，每次存储分片，会计算磁盘的存储经验值，选择相对来说读写性能较优的磁盘，例如读写较频繁的磁盘或者出现问题次数较多的磁盘，此时它对应的经验值较低，这样可以大大提高数据的操作性能。消息队列的使用可以提高客户的使用满意度，例如在删除数据的时候，拿到每个分片的URL和IP，生成任务放到MQ中，然后刷到Hbase的Task表中，这是就已经可以给客户返回操作成功的消息，实际的任务并没有做完，只是保存在了Hbase中，系统还有一个叫BaseRecover的进程，这个进程会定期扫描Task表，进行后续的任务处理。

4.2 jetty框架的应用

421 jetty框架的介绍

Jetty作为一个轻便的Web服务器，架构比较简单，有很强的扩展性，对于不同的业务可以灵活修改，Jetty有如下2个核心组件，Server组件是急于Handler容易工作，同时需要Connector的配合，负责在客户端发送请求是提供有效的连接，请求会被分配到一个处理队列中，当然也有一些其他的组件如JMX，JNDI等。



Jetty服务的主要是围绕着Server类来构建的，主要用来为Web程序提供访问服务，Jetty的Server扩展实现了一个个Handler，实现许多不同的逻辑处理。Jetty的Handler有2种类型，一种是HandlerWrapper，把Handler委托给另一个类去执行，如果把Handler加入到Jetty中去，这时我们就需要通过Server去调用，同时配合ScopeHandler类可以拦截Handler的执行，在Handler的前后可以添加一系列的操作，比如日志的打印，如果有数据库多步骤的操作，此时我们可以加进去事务，使程序的业务逻辑更加完善。另一种是HandlerCollection，这种方式是把多个Handler组装到一起，构成一个Handler链。

一个Jetty服务需要初始化Server服务，并注册web应用程序的Context，申明对应的路径，下面是一个最简单服务所需要的内容：

Server server = new Server(8080);

ServletContextHandler context = new ServletContextHandler(ServletContextHandler.SESSIONS);

context.setContextPath("/");

server.setHandler(context);

context.addServlet(new ServletHolder(new HelloServlet()), "/hello");

context.addServlet(new ServletHolder(new HelloServlet("Hello Kongxx!")), "/hello/kongxx");

context.addServlet(new ServletHolder(new GoodbyeServlet()), "/goodbye");

context.addServlet(new ServletHolder(new GoodbyeServlet("Goodbye kongxx!")), "/goodbye/kongxx");

server.start();

server.join();

在这里context可以添加任意多的业务处理Servlet，同时对于context我们也可以添加多个，保存各自的上下文：

Server server = new Server(8080);

ServletContextHandler context1 = new ServletContextHandler(ServletContextHandler.SESSIONS);

context1.setContextPath("/hello");

context1.setResourceBase(".");

context1.setClassLoader(Thread.currentThread().getContextClassLoader());

context1.addServlet(new ServletHolder(new HelloServlet("Hello Kongxx!")), "/kongxx");

ServletContextHandler context2 = new ServletContextHandler(ServletContextHandler.SESSIONS);

context2.setContextPath("/goodbye");

context2.setResourceBase(".");

context2.setClassLoader(Thread.currentThread().getContextClassLoader());

context2.addServlet(new ServletHolder(new GoodbyeServlet("Goodbye kongxx!")), "/kongxx");

ContextHandlerCollection contexts = new ContextHandlerCollection();

contexts.setHandlers(new Handler[] { context1, context2 });

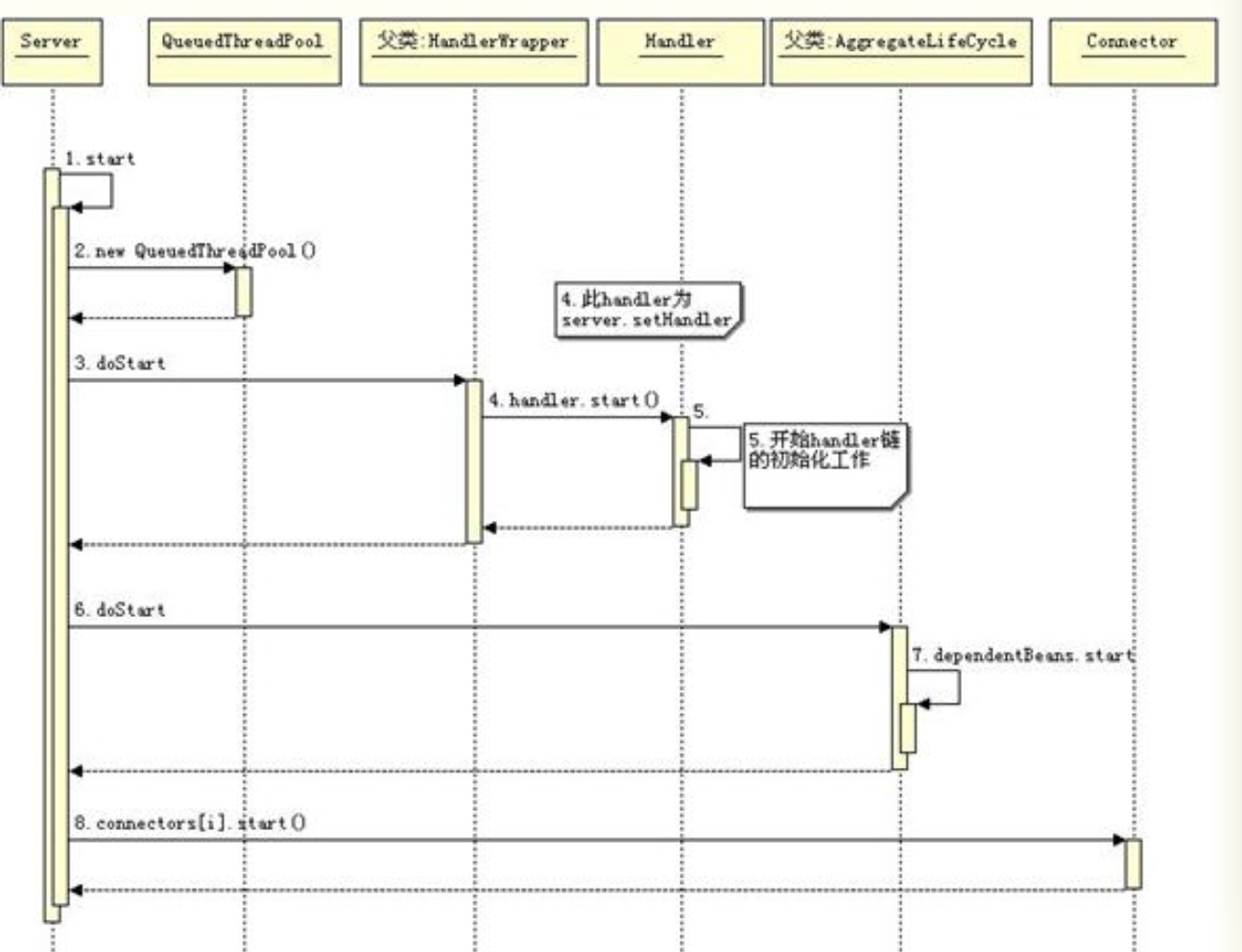
server.setHandler(contexts);

server.start();

server.join();

Jetty启动原理：

下图是启动以及服务调用的流程图：



Jetty服务的启动从Server的start函数开始，这个函数执行完后，代表Jetty能够对外提供服务了，至于需要哪些服务，可以对Jetty做不同的配置，Jetty中所有的组件都是继承自LifeCycle，在调用start函数的时候，会启动所有的已经注册的组件。首先Server会把注册的Handler启动，通常这个Handler会有很多的子Handler，这些子Handler形成Handler链，Server会依次把链上的所有Handler启动起来，然后让已经注册在Server上的JMX的Mbean启动启动起来配合Handler的相关工作，上述过程完成后会启动Connector，通过指定端口接受客户端的请求。

客户端和服务器的交互一般采用HTTP协议，当Jetty接收到客户发送过来的请求的时候，必须按照HTTP协议解析请求，从Header中获取相关的数据，进入到指定的类执行相应的业务逻辑，然后封装返回的数据。

处理客户端的链接实现类是org.eclipse.jetty.server.bi.SocketConnector，让Jetty以BIO的方式工作，BIO工作环境在启动的时候就会创建，并由HttpConnection 类来解析和封装Http协议；ConnectorEndPoint类是以BIO的方式处理链接请求，ServerSocket通过建立socket连接进行数据的交互；

Executor是处理connect的线程池，负责处理请求队列中的任务，acceptThread监听连接请求，下面是建立连接的过程：

首先创建一个队列线程池，用来处理每个建立连接产生的任务，线程池可以通过用户来指定；再建立ServerSocet，准备接受客户端的socket请求；最后再创建监听线程，这样可以用来监听访问端口是否有连接建立。

Acceptor线程会为客户端的请求创建ConnectorEndPoint，HttpConnection用来表示连接是一个http协议的连接，HttpPars类负责解析HTTP协议，同时创建Request和Response对象，然后把线程放到线程池中去执行。

基于AJP工作

一般来说，web网站的后台服务端不是将java程序的服务端直接暴露给用户的，比如说在Jboss的前面加一个web服务器，像apache或nginx都是常用的服务器，通过这种方式可以增加一些方便的功能如：日志分析，均衡负载，权限控制，防止外部的恶意请求以及一些静态资源的预加载等等。通过这种架构，servlet引擎就不再需要对http协议做处理，这些工作都在web服务器端完成了，极大地加快了请求的相应速度。

422jetty框架在云平台中的应用

对于整个云平台来说，jetty负责处理用户的第一层请求的，解析http协议，获取请求信息，web服务使用通过jetty和外部实现交互，这一层在云存储平台称之为cos-server，sdk可以通过裸写http请求申请云存储服务，也可以开发应用程序，调用cos-server的Api，所有用户需要实现的功能必须通过http请求通过cos-server向底层发送并获得服务，可以称之为云服务的包装层，整个云平台唯一可以跨越jetty服务层的是存储后台服务程序，这是云服务内部服务，不具有向其他外部开发者开放性，可以分配给若干个用户访问使用，程序功能较单一，没有扩展性，存储后台服务程序直接通过调用IO-server的Api调用调用相关的服务，这其实也是云平台对外提供存储服务的实例程序。

43Hbase框架

431Hbase框架原理

4311Hbase定义

Hbase是一个可靠，性能良好，列式存储的数据库，使用过程中可以根据需要动态扩大列的数量，通过Hbase可以通过大规模的廉价机器构建超大存储量的集群。Hbase的实现主要采用了Google Bigtable的原理，存储利用Hadoop HDFS作为文件存储系统，处理海量数据时使用Hadoop MapReduce来处理，集群中的海量机器通过zookeeper来管理。

4312 Hbase特性

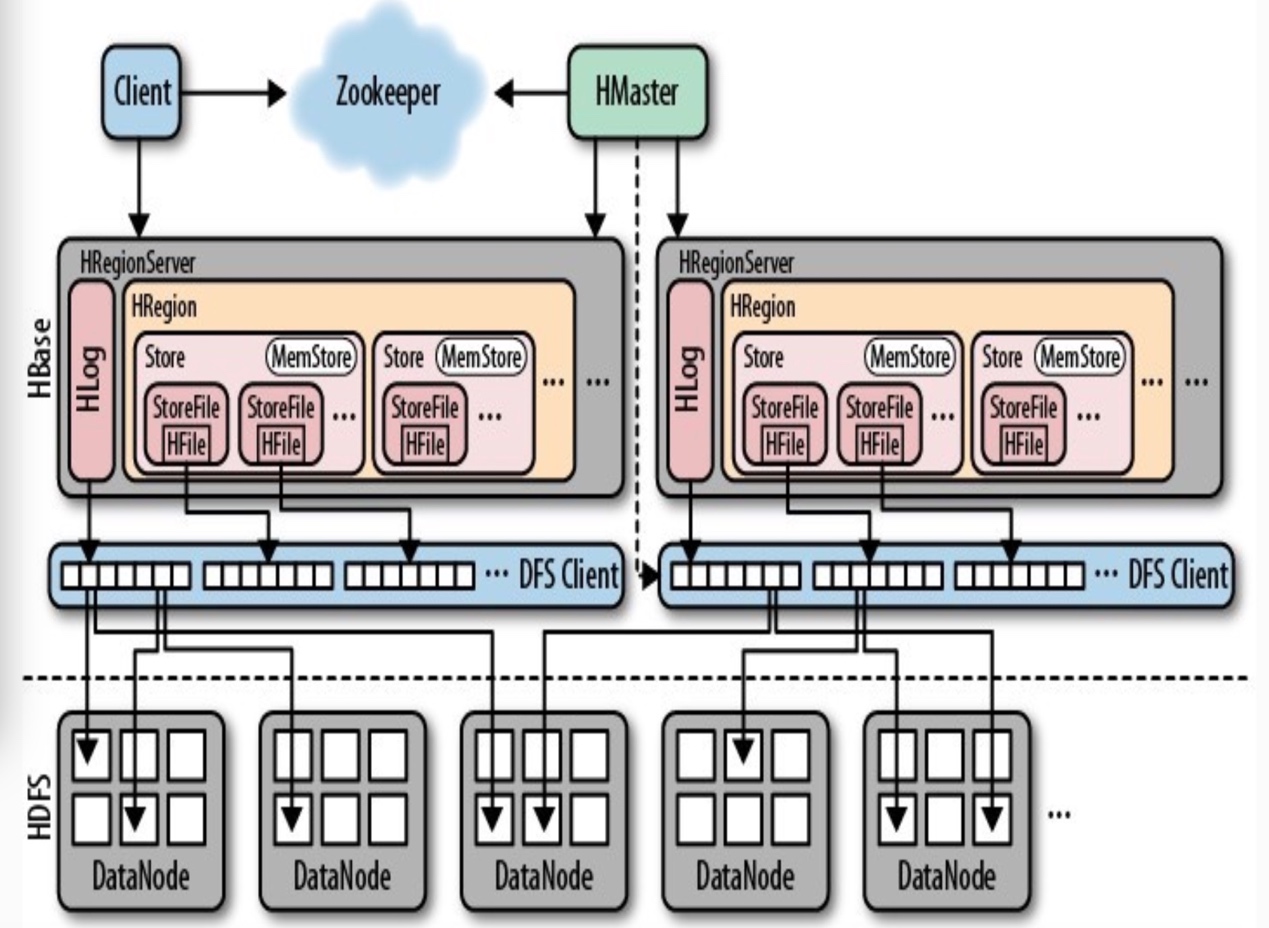
Hbase的表结构非常大，有上亿行和上百万列，且列可以动态增加，它的权限控制面对列簇，数据呈现列式块存储，列或者列簇检索信息独立，和传统的关系型数据库不同，空列不占用存储空间，这样表可以设计得非常稀疏。

4313Hbase访问接口

Hbase最常规和最有效的访问方式是Native Java API，适用于任务的并行处理；Hbase Shell ，命令行工具，一般来说不涉及复杂的业务，比如查询数据库所有的表，扫描表的数据，添加数据，增加列等等；Thrift Gateway，可以支持多种语言如C++，JS等的访问；REST Gateway，支持Http的API访问Hbase服务，不受客户端语言的限制，在Hbase服务端解析相应的Http请求即可；Pig可以使用Pig Latin流失编程语言调用库里的数据，它的原理就是通过程序生成MapReduce Job来处理Hbase的表数据，对海量数据的处理有很大的优势；Hive，通过编写类似的SQL语句来访问库里的数据。

4314 Hbase 的结构

下图是Hbase的存储结构模型图：



Hbase的主要包含Hmaster，HregionServer，HRegion，Store，MemStore，StoreFile，Hfile，Hlog等模块，下面将对各个模块作详细的介绍。

Hmaster： 可以为Region server分配region，资源使用不均衡是，调整不同的server之间的负载均衡，当有的设备出现故障，此时可以将此server上的region重新分配到别的设备上，随着业务额扩展，文件系统中会逐渐初心啊一些垃圾文件，Hmaster会负责这部分文件的回收，释放出来这部分存储资源，对于一些schema的请求也是H master处理的。

HRegionServer：主要管理被分配的Region，处理在其上的IO请求，随着存储数据量的增加，Region逐渐扩大，需要把Region切分，这是HregionServer的一个很重要的功能。通常用户在访问Hbase中表的数据时，并不需要master作处理，对于客户的请求，首先需要访问zookeeper和region server，然后访问数据通过region server，master只需要处理好表和region的元数据信息即可，负载比较低。HregionServer存取一个表时，会创建一个HRegion对象，表的列簇是查询的单位，每个列簇对应着一个Store实例，这其中有一个MenStore和若干个StoreFile与之对应，每个StoreFile又对应一个HFile，这里的HFile是存储在磁盘中的文件。

Hregion：一个表上有有若干个Region，Region是分布式存储的最小单位，Region可以放在不同的设备上，一个Region必须存储在同一个设备上，对于Region来说，Hbase会给Region的大小设置一个阀值，随着数据存储量的增加，Region包含的信息越来越多，当超过了这个阀值，此时就会被分成2个新的region。Hbase定位region的过程为：首先通过zookeeper的存储region位置的文件找到-ROOT-表的位置，这里-ROOT-表是.META.的第一个Region，而.META.表中的每一个region在-ROOT-表都有一条记录，这时可以通过.META.表找到所要的用户表region的位置。

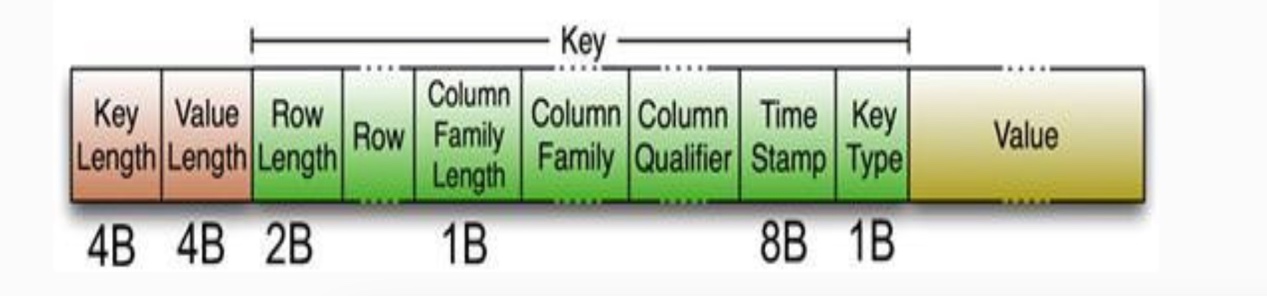
一般来说一个表开始只有一个region，随着数据量的增加，region会进行分裂，每个region都有startkey和endkey参数。

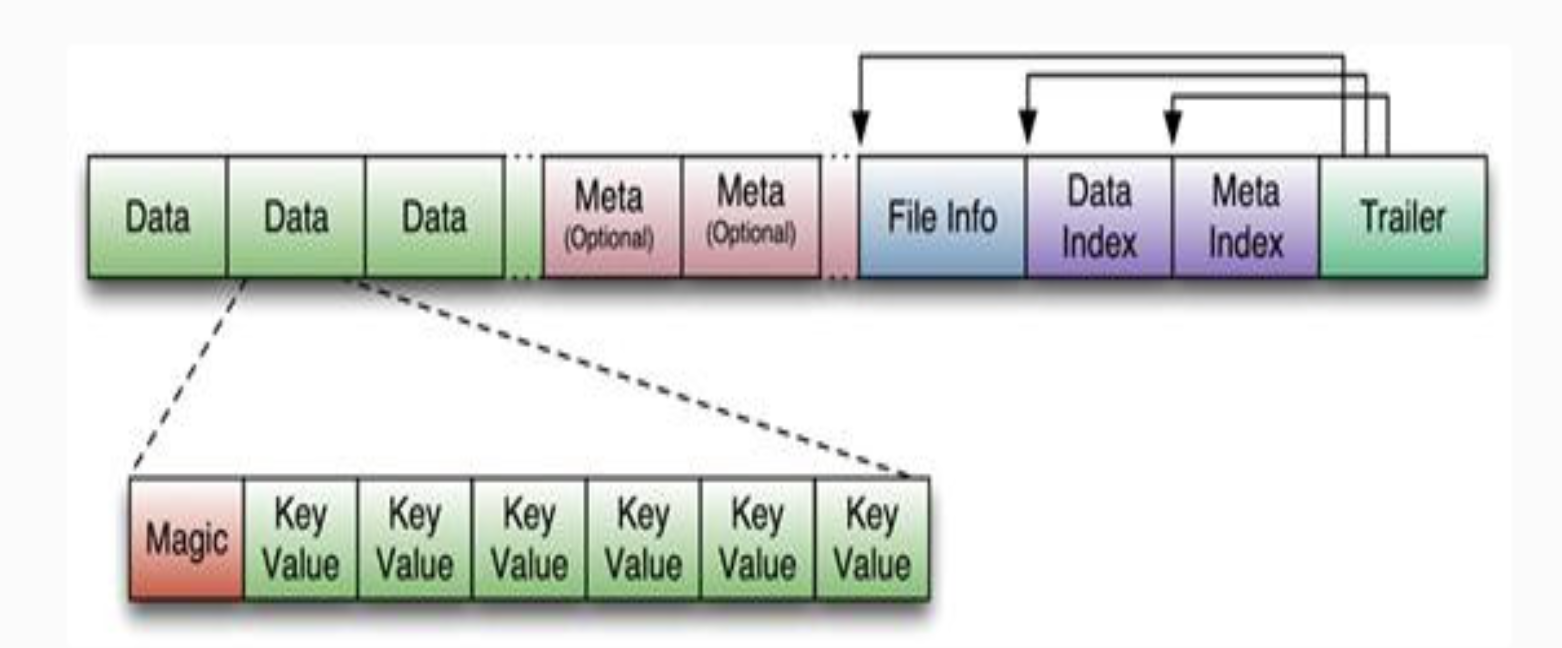
Store：region主要是由若干个store组成的，每个列簇建一个store，而一个store由一个memStore和0或者多个StoreFile组成。HBase以store的大小来判断是否需要切分region。

MemStore：memstore放在内存里，数据以keyValues的形式，由于存储在HDFS上的数据需要按照行键进行排序，HDFS本身就是顺序读写，不能修改，这样的话Hbase不能高效写数据，写到Hbase的数据是不会被排序的，将来检索性能大大降低了，于是Hbase为了解决这个问题，就在内存中多了一步处理，完成数据的排序，这样写入到磁盘的数据是顺序的，为将来的数据检索带来了性能上的提升。

StoreFile：memStore内存中的数据写到文件后就是StoreFile，在底层以HFile的形式存在。

HFile：在Hbase中，数据主要以键值对的形式存储，是二进制文件。HFile的文件长度长短不一，其中有2部分是固定的长度：Trailer和FileInfo，Trailer中含有其他的数据块的起始位置信息，FileInfo记录了文件的一些元数据信息，数据块是Hbase进行输入输出的最小单位，对于数据块的大小可以在建表的时候指定，如果不指定，一般来说，默认大小是64KB，块的大小各有利弊，较大的块方便顺序扫描，小的块有利于随机查询。每个块由一个Magic和多个键值对组成，Magic储存一些随机化的数字来保证数据的完整性的，每个KeyValue包含了键长，值的长度，行的长度，列簇长度，列的长度，列属性，时间戳以及key类型等。数据块主要是用来保存表中的数据的，数据块可以压缩。元数据段用来保存一些自定义的键值对，这部分可以压缩，而文件信息段主要用来保存HFile的元信息的，则不可以压缩，data index用来保存meta块的索引，Trailer主要是用来保存每一段的偏移量，读取HFile的时候，首先会读取Trailer，这样就找到了data index，进而不需要遍历整个文件就能找到对应的数据块，再找出最终的Value。现在的存储模式，数据块和元数据块基本都采用的压缩的方式，压缩可以大大减少IO的时间，不过这样也又一个弊端就是压缩和解压耗费了一定的资源，常用的压缩方式有gzip和lzo。





Hlog：Hlog文件就是一个普通的序列化文件，序化化文件的value的key是HlogKey对象，记录了写入数据的归属信息，存储的信息包括了table名，region名，时间戳（写入时间），对应的value是keyvalue对象，也就是HFile中的键值对。Hlog主要是数据写入钱，先把日志保存下来，当机器发生故障的时候，可以从Hlog中调出数据变更的记录，可以通过这些记录进行恢复。

LogFlusher：存储数据的时候，首先把日志写下，然后再写入序列文件，通常来说，数据流进入文件系统，采用缓存机制，保证存储的性能，可能有时一部分数据对于客户端来说已经显示存储完成，实际上，这部分的数据在服务端并没有真正刷到磁盘，而是存在内存中。这就需要处理类LogFlusher，通过调用optionalSync()方法，在hbase还有一个设置默认定时刷日志的设置项hbase.regionserver.optionallogflushinterval，一般情况下默认是10s，这个参数定期调用Hlog.sync()，进行数据刷盘；还有一个doWrite函数也会定期调用sync()，实现刷盘功能。

LogRoller：Log的大小通过配置文件hbase.regionserver.logroll.period来设置，不设置的话，默认是一个小时，每过60分钟，便会产生一个新的log文件。这样随着时间的推移，会形成很多很多的log文件需要定期整理，rollerWriter定期滚动日志，有一些日志也可以定期清除。每个region server都会维护一个Hlog，这样，不同的region的日志就混杂在了一起，这样在追加单个文件的时候，可以同时写多个文件，大大减少了磁盘的寻址次数，table的写性能得到了很大的提高。这样也有相应的弊端，如果某台设备出现故障了，对数据进行恢复时，需要把日志进行拆分，然后发送到其他的region server进行恢复。

Hbase 结构设计

Hbase的数据库表是一张很大的表，从整体上看，行记录有三个基本类型，包括：RowKey，Time Stamp，Column。RowKey是数据在表中的唯一标识，时间戳是数据版本的标志，列由列簇和列名组成，列簇的修改类似与需要修改表结构，而列名可以动态增加，这也是Hbase能够动态增加列的体现；列示存储使得读写性能得到很大的优化，相同的family的数据存储位置比较近，对于这样的存储特性，在我们处理具体访问需求时，我们可以通过业务逻辑层加以利用。

在Hbase数据库中，行键用来检索数据的，我们可以通过单个row key进行访问，全表扫描访问，也可以使用行键的前缀进行访问，在数据底层，row key保存为字节数组，而存储的时，按照字典顺序进行存储，设计Key的时候，我们需要充分利用这个特性，把经常一起读写的数据放在同一个列簇中。

列簇包含若干个表，列簇是Schema的一部分，在定义表时就需要指定好，而列就不需要，是可以动态调整的属性，每个列都有其所属的列簇，对于访问的控制，磁盘和内存的使用统计都在列簇的层面上的，访问控制可以允许一些应用有修改数据的权限，或者只有浏览数据的权限等等。

通过行和列簇可以确定存储单元。Cell的数据没有具体的类型，全部以字节码形式存储，每个Cell都保存着多个版本，一些数据会进行覆盖和内容的修改，进行数据变动时，系统会指定此时的时间戳，也可以自行添加时间，Cell的不同的版本倒序排列，最新的数据在前，一般访问数据时，只需要最新的那一版数据即可，对于数据的管理版本，目前有2总常用的数据管理方法，一种是按照最近的时间去管理，比如根据存储能力的大小，数据版本更新的频率，数据量的大小等，规定一个可以存在多版本数据可控范围的时间区间，比如一月，这样，所有一个月内的数据增加和修改版本都会保存下来；另一种方法是规定版本的数量，每条数据指保存N个最近的版本，增加一个新的版本，如果此时已经达到存储的数量上限，此时需要存储最新的版本，删除最老的那一版数据。

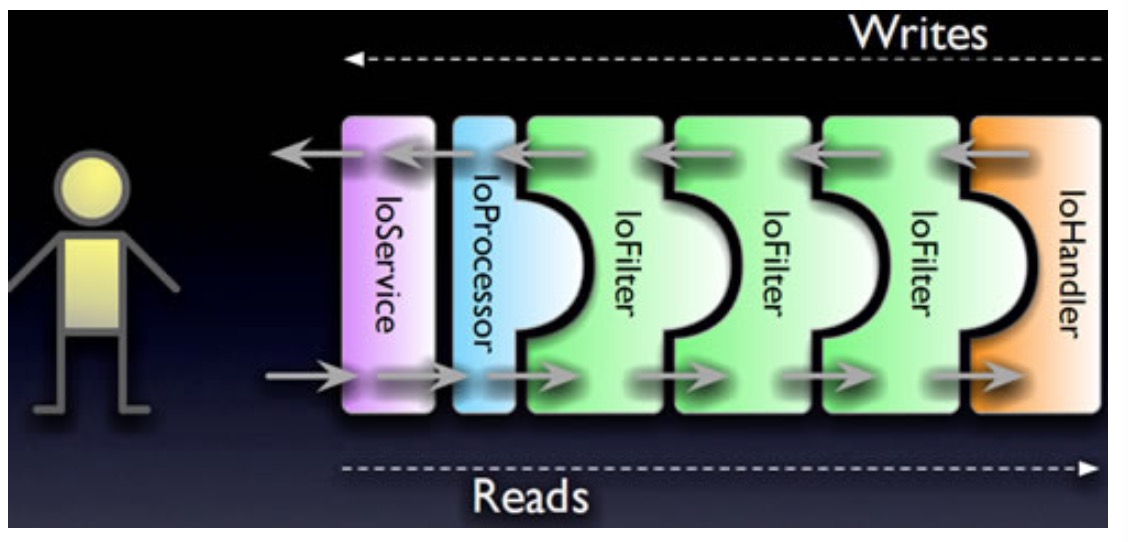
432 Hbase的应用

作为大型的云存储平台，对数据的扩展性和密集访问性有着很高的要求，高峰时期，可能有每秒上千次的访问量，对这样的高频访问，对于普通的关系型数据库来所，不可能完成，得益于Hbase的行键访问机制，能够迅速从海量数据中挑选出我们想要的数据，所以选择Hbase作为云平台的元数据存储，Hbase依赖于HDFS，本平台的流数据存储在HDFS上，这样也能通过应用程序获得元数据的同时，快速获得真实数据。目前Hbase已经开源，随着功能和性能的完善,目前Hbase的性能也相对稳定，具有很强的可扩展性，数据库表对应实现的实例类，很方便实现了对数据库的操作。对于数据的版本，我们可以保存多份最近修改的数据，方便对历史版本的调用。

对于Hbase的行键搜索是通过顺序搜索的，这样对于大对象的存储，采用分片存储的功能，不同的分片采用相同的Hash值加对应的序号，对于大对象进行切割，在存储数据时，可以实现多个分片同时上传到不同的磁盘上，在检索的时候，通过对应的Hash值模糊查询出所有的分片，然后组合成完整的对象，再传给客户端。对于像批量删除大批数据这样的操作，采用在Hbase中建各种task表，失败表，未处理表，处理失败的表，这些表用来暂时存储客户发过来的任务，可以在人物还没有真正完成的时候，给客户一个提前的反馈，然后系统可以选择一个合适的时间去执行这个任务，并在对应的位置，如bucket上加相应的标志表明某文件已经被删除，这样在刷新bucket的时候，就不会再把不需要的文件也罗列出来。在Hbase中还设计了流量监控表，对不同的用户访问情况作相应的统计，对后期的收费问题，需要一些参数比如某个用户的访问次数，上传次数，下载次数，流量大小等等，这些都是消费大小的指标，这些服务功能都通过Hbase高效管理地了起来。

44 MINA框架

441MINA框架介绍



Apache Mina Server是一个应用与网络通信的框架，底层通信协议可以是TCP/IP,UDP/IP协议，当我们的应用程序需要较高的处理性能和较高的扩展性，MINA框架是一个不错的选择，MINA提供了事件驱动操作，底层使用NIO机制，刚刚发行是1.x版本，现在我们所说的MINA框架一般都是指2.x版本，框架基本已经把底层的通信机制和业务逻辑隔离开来了，作为程序开发人员来说，只需要关注发送的数据和业务逻辑就可以了，下面将对框架的各层作详细的讲解：

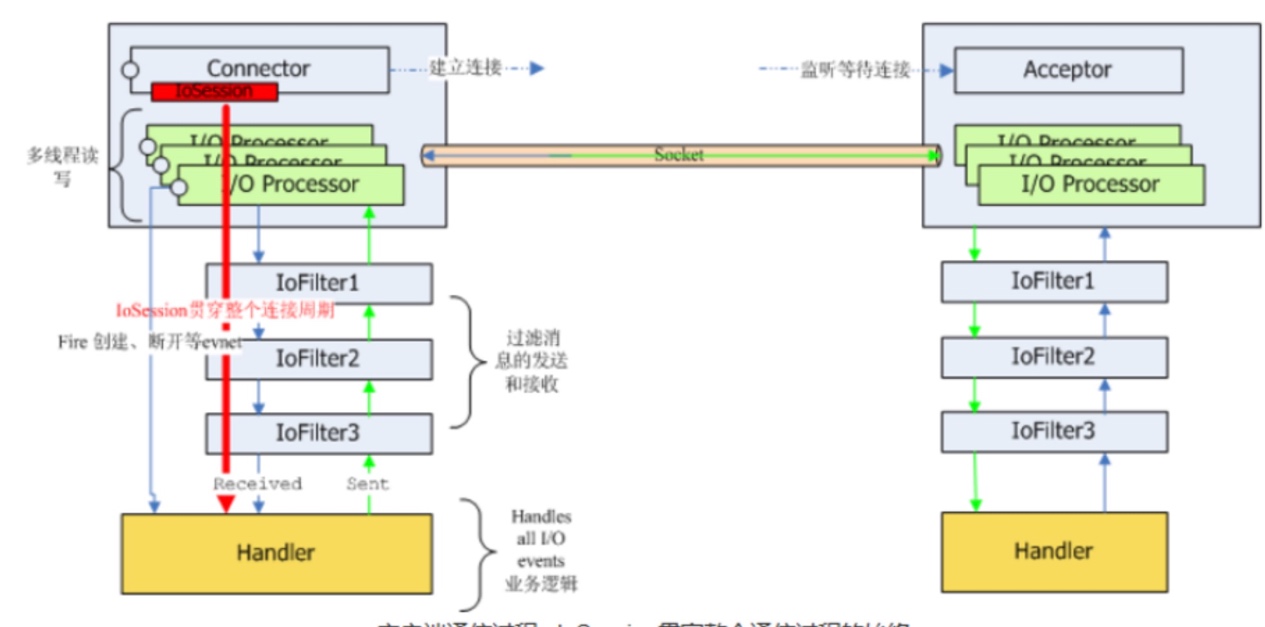
IoService: 这是框架的最底层，主要用来处理IO相关的工作的。IOSocketAcceptor对应TCP协议下面的服务端IOService，而IOSocketCHannel对应TCP协议下的客户端IOService，IOService主要用于隐藏底层IO操作的细节，对上层提供统一的基于事件的异步IO接口。数据到达时，IOService调用底层IO接口读取数据，然后封装成IoBuffer，然后以事件的形式传到上层，这样底层的同步IO转化成了异步IO，low level IO也经过IOService变成了IO Event。

IoProcessor：系统需要为此接口单独起一个线程，负责检查是否有数据在通道上读写，这样它就拥有了自己的Selector，这与通常的NiO编码不一样，一般来说只有一个Selector， IoProcessor负责调用注册在IoService上的过滤器，然后调用IOHandler。

IoFilter：这个借口定义一组拦截器，比如说一些日志的输出，系统设置的一些限制用户，对数据进行编码和数据解码等相关功能，在开发过程中，很多时候会遇到数据的编码解码问题，这对数据的传输有着至关重要的作用，在内存中和网络中，数据的传输完全不一样，所以一个合格的开发人员需要考虑的不仅仅是功能的实现，更加需要关注程序运行的效率，这就尤其需要我们关注数据的编码。

IoHandler：这一层是开人员需要重点开发的一层，也是数据接收发送的层，在这里我们会对发送过来的资源进行相关的处理，这一层的具体实现需要开发人员作详细设计和处理，这一层是框架处理流程的最后一站，通畅在这一层会对业务做详细的包装，对于一个指令的执行，做一系列周到的判断，指令执行的条件是否满足，调用相关的类方法，在本流程业务处理过程中，对于不合理的操作，或者操作条件不满足，会有一系列异常机制，并将异常信息反馈给客户端，对于这一层有完善的继承实现机制，严格按照框架指定的继承方式去实现相关的功能，每个IoService都需要指定一个IoHandler。

IoSession：这个类是对底层连接的封装，当新的连接建立的时候，会立刻生成一个context和session，用来存储全局信息，保证这些信息在执行的过程中能够随之拿出这些信息作相关的应用，发送数据是一个异步的过程，业务层首先会逆向穿过Filter链，到达IoService，此时并没有直接调用底层的接口实现相关的数据操作，而是将这次调用封装成一个写请求，然后放到session的请求队列中去，通过IoProcessor线程统一flush出去，这样就不会造成线程的等待情况。



MINA框架提供服务时，首先时服务端和客户端建立连接，I/O的读写交给了I/O Processor线程，读取的数据经过了一些的IoFilter，在这里进行一系列操作，包括消息过滤和格式转换等等，最后把数据交给了Handler进行处理，最后按照原来的路径逆向传输，最后通过Processor线程把数据写出输出通道。

一 Mina有很强的扩展性，作为扩展开发的程序员，我们需要对各层的接口有深刻的认识。

1. IoService层： getTransportMetadata()主要用来获取传输方式的云数据描述信息；addListener用来添加监听器，监听连接的建立，空闲，活动周期，消亡等等；removeListener用户移除程序添加的监听器；setHandler用来注册处理逻辑的Handler；getManagedSessions用来获取IoService管理的所有session；getSessionConfig()可以获得配置对象来设置连接的一些属性设置。
2. IoAcceptor：

这是服务端发起服务的接口，bind()方法用来监听服务机器上的某端口，Mina框架中的监听接口机制和其他的有所不同，可以多次调用bind()方法，同时监听不同的端口来接收服务。

1. IoConnector

这是客户端建立连接的端口，connect方法可以饥饿服务端建立连接，也可以同时访问多个服务端，进行不同的业务交互。

1. IoSession

这个接口表示服务端和客户端建立的连接，其中有许多属性在程序运行的不同位置可能需要调用，传入Handler的时候，可以把对应的实现类传进来，这个接口有很多常用的方法：write用来把数据写出去；close方法用来关闭连接；setAttribure让我们可以在会话中添加一些属性，会话的整个过程中都可以调出来使用，底层通过HashMap来存储我们自己定义的属性；getRemoteAddress方法用来获取远程连接的套接字地址；suspendwrite用语挂起写操作，同时也有恢复写操作的resumeWrite与之对应；

read()用于读取数据，这种读取数据的方法是将数据读取到一个阻塞队列中，对于服务端不太适用；getService()用来获取当前的IoService实例。

1. IoSessionConfig

这个方法用于指定会话的配置，setReadBufferSize()用来设置读取的字节数，通常不用，IoProcessor会自动调整缓冲的大小，也可以通过setMinTeadBufferSize(),setMaxReadBufferSize()可以设置自动调整的区间；setIdleTime()，通道上的事件在设置时间内没有发生,这时通道就会进入空闲状态；setWriteTimeout()用于设置写操作的超时时间；setUseReadOperation()用于设置read方法是否可用。

1. IlHandler

这个接口是程序员操作最多的地方，数据的读取和发出都是在这里编写的，sessionCreate()在Session对象被创建的时候调用；sessionOpend()在连接打开时调用，在这里可以做一些认证操作，数据的流入流出操作；sessionClosed()在连接被关闭时，调用这个方法；sessionIdle()在通道进入空闲状态时调用；exceptionCaught()在程序执行过程中出现异常时回调；messageReceived()接收到消息时调用的方法，一般来说message是一个IoBuffer类；messageSent()当发送消息成功时调用这个方法，其实这个方法不是用来发送数据的，真正发送数据是IoSession.write()方法完成的。

1. Iobuffer

本接口是对ByteBuffer的封装，byteBuffer只有基本的数据读写操作，不可以直接对字符串等对象类型进行操作。其中的allocate使用的是SimpleBufferAllocator创建一个具有初始容量的实例，第二个参数指定缓冲区的类型；free()用来释放缓冲区；setAutoExpand()可以把本身设置成长度可变的自动扩展容量；setAutoShrink()用来裁减掉一些闲置的空间；order()设置区域的位置；asReadOnlyBuffer()用来设置缓存是否是只读的；

1. IoFuture

在Mina的很多操作中，可以看到返回值是IoFuture的子类，这个接口和Future有很大的相似性，对于异步执行，可以返回结果。addListener用于添加监听器，异步执行时返回结果时，会触发回调方法；removeListener移除指定的监听器；getSession()返回当前的IoSession，连接成功时，可能对IoSession进行相关的操作。

二 日志配置：

Mina框架使用的是SLF4J来打印日志，对于一些精细的业务，有时我们需要跟踪程序的运行细节，这时可以配置过滤链，对于Session从建立，打开，空闲等，把跟踪信息打印出来，对于日志，可以设置不同的级别，DEBUG调试应用程序有很大的帮助，INFO级别在程序运行都会打印相关的信息，WARN会打出一些潜在的错误，ERROR指出一些不严重的错误类型，但是并不影响程序的继续运行，FATAL指出严重的错误事件，这样的错误会导致程序的退出，ALL是级别最低的，用于打开所有的日志记录，OFF是最高等级的，用于关闭所有的日志记录。

三 过滤器：

Mina框架中主要有日志过滤器和协议编码过滤器，前者主要用来负责日志的输出的，后者负责数据的编码解码，这里我们主要介绍协议编解码器，数据在网络中传输是采用的二进制byte，程序中是JAVA对象，这就需要在进行发送数据的时候讲对象编码成二进制数据，接收数据时需要把二进制数据转化成JAVA对象，这需要客户端和服务端同时遵守一套相同的协议，举个简单的例子：对于数据传输编码过程可以定义一定的规则，比如对于一段数据流，第一行代表协议名字以及版本，第二行代表源IP，第三行代表目的IP，第四行代表字节数，最后一行代表传输的内容，行的末尾以10作为标志，也就是ASCII中的换行符，这样的数据流在接受的时候，不知道何时数据接收结束，对于数据的结束需要自己找到结束位置，目前采用的方法有设置分隔符的方法，当接受数据遇到某个分隔符就终止数据的接受，或者可以通过信息中的内容长度来判断数据的长度，这种方式较为科学，对于接收端知道自己需要接受多长的数据然后终止接收。这是解码的过程，编码相对来说比较简单，对于需要发送的数据，只需要打开数据流，按照指定的格式把数据添加到IoBuffer里，数据添加完整后只需要通过write方法把缓存刷向通道即可。

四 线程模型配置

Mina框架为了提高性能，在很多环节使用了多线程机制，下面我们介绍一下shying线程的三个地方：

IoAcceptor：作为服务端等待客户端的连接，对于每个端口，设置一个线程处理连接，

当有请求到来，便为此线程创建一个IoSession对象。

IoConnector：用于和服务端建立连接，每连接一个服务端，就新起一个线程，连接建立后，为此连接在客户端创建一个IoSession对象。

IoProcessor：用来作真正的IO操作，默认的线程个数时CPU的核数+1，IoAcceptor和IoConnector都会对应着一个IoProcessor池。比如8核CPU，就有9个IoProcessor，IO操作是很好资源的操作，采用这种线程池的方式操作数据，能大大提高操作的性能。

对于IoAcceptor的数量在IoAcceptor定义的时候可以设置，Mina 的解码器要使用IoSession来保存状态变量，而不是解码器本身，主要是因为连接可以长期存在，而数据的传输却是随时可嫩停止的，这时如果线程仍然被粘人用，就会造成资源的浪费，对于多线程框架，采用的是在IoSession处于空闲的状态时，可以把处理线程放回池中，处理线程仍然可以被其他的IoSession随时调用，大大提高了资源的重复使用性，如果原来的IoSession再次处于繁忙状态时，线程池慧给它再分配一个IoProcessor实例，这时可能已经不是上次执行的IlProcessor了。线程池对象在默认情况下会创造一个无界的线程池，当然也可以根据具体情况，设置线程池的相关参数，不过这些参数需要作科学的计算，否则会造成性能的大幅度下降。

442MINA框架应用

随着云存储业务的扩展，客户在使用云服务的过程中，对服务质量的要求也越来越高，开发初期，云服务通过Web面向客户，这种方式访问初期对于程序员来说很方便，编写后台程序时就可以编写相关的访问服务，对于云存储来说，处理的频度非常高，而且在Web端的稳定性有一定的限制，FTP时文本传输协议，可以实现互联网的文件双向传输，本地的PC机可以和互联网上的很多使用FTP协议的服务器连接进行数据传输。

FTP服务器使用C/S的方式，作为服务端需要发起一个基于FTP协议的服务，并添加响应的端口监听客户端的请求，客户端一般通过图形化界面进行业务操作，FTP的命令非常繁多复杂，客户使用命令的学习成本很高，图形化界面操作能让客户很快熟悉云服务的流程和操作，经过服务端的授权，得到IP，用户名和密码，输入这些信息建立连接，即可看到此用户拥有的权限下的文件信息，后续的操作和Windows下文件操纵基本一致，在客户端通过类似Windows鼠标操作，会把响应的指令解释成FTP可以处理的指令，包括文件的上传，下载，删除，重命名等。这样只要客户端能连上互联网，在世界各地能随时随地稳定安全访问云存储的服务。

45 HDFS

451 HDFS的原理

HDFS是分布式服务的存储基础，是基于流式数据的访问，对于超大文件的处理有着独特的优势，存储的硬件是多设备集群共同承担存储业务，可以把大量的相对廉价的设备投入使用，能以较小的成本开发出高服务质量的平台。HDFS具有容错，可靠，高吞吐率，高扩展性等特性，为海量数据的存储提供了可靠的保障。

HDFS的出现并不是一个新的发现，早起Google发表一篇Google File System的论文，准确来说，HDFS就是基于这篇文章扩展而来的。

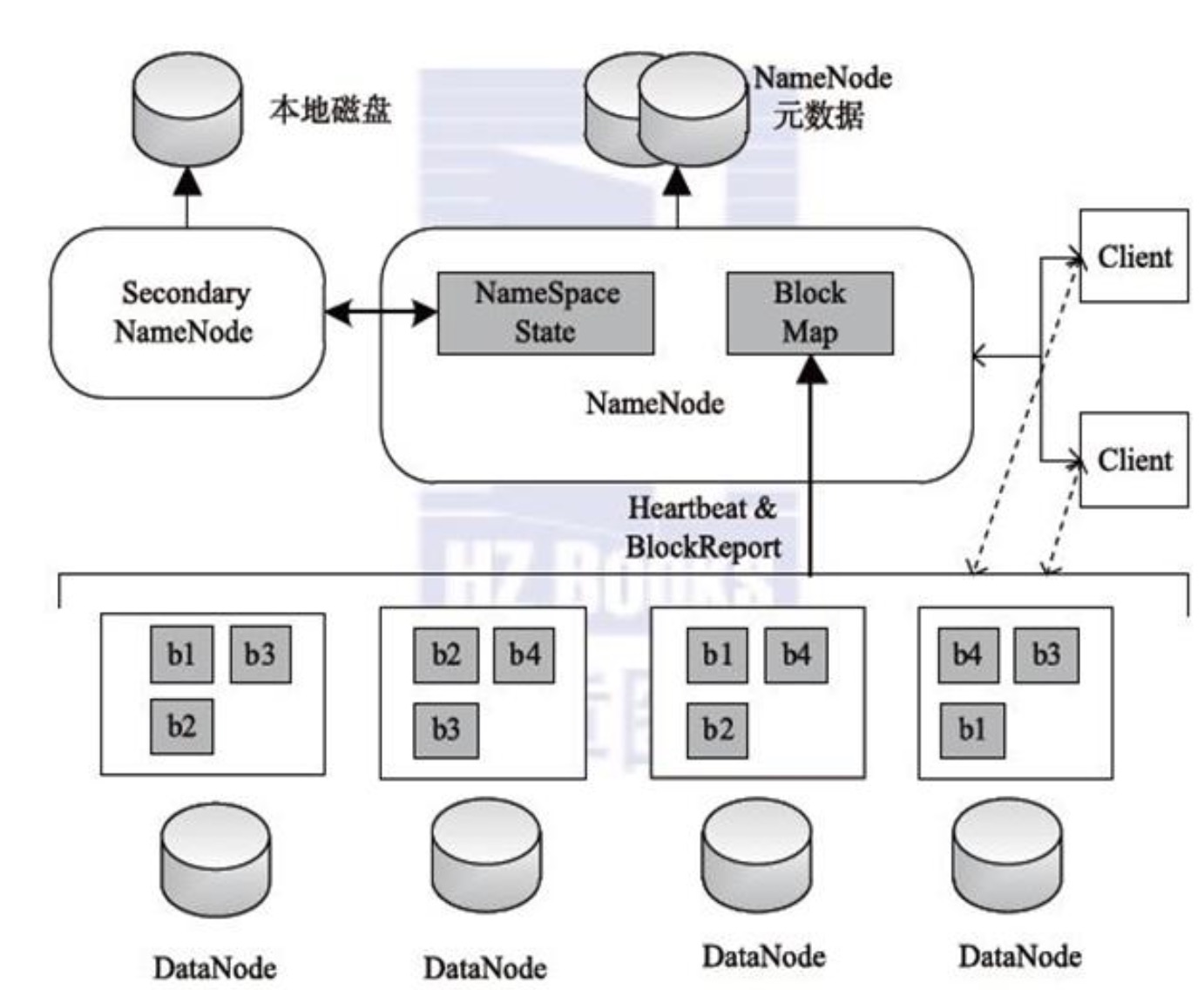
4511 HDFS的优势

高容错性：每一份都有多个副本，充分保证数据的容错性，如果一个副本丢失，可以自动恢复；比较适合批量处理，使用移动过计算而不是移动数据，数据存储的位置交给框架去处理；处理数据都是TB，PB级别的数据，处理的文件数量也是百万规模以上，节点数量达到10K的规模；流式文件的访问一次性写入，以后不可以修改，只能追加，可以通过自己的方式保证数据的一致性；可以通过软件的管理，添加多样化机器，降低硬件上的成本。

4512 HDFS 的弊端

虽然HDFS有以上这些优势，优势也是相对的，对有些业务场景也是不适合的，比如对于一些要求低延时的业务，存储和读取数据的事件限制在毫秒级别，HDFS做不到，能够提供较大的吞吐量，同时存储大量的数据，不具有较高的实时性；对于HDFS来说，小文件的存储不利于存储的性能，这里的小文件大小是指小于64M的文件，每个文件的存储都在NameNode中有相应的文件信息，目录信息，块信息存到内存中，而且对于小文件的存储的寻址事件超过了读取时间，所以不适合小文件的存储；对于文件的写操作是安全写操作，文件只支持数据追加。

4513 HDFS存储原理



HDFS的架构图

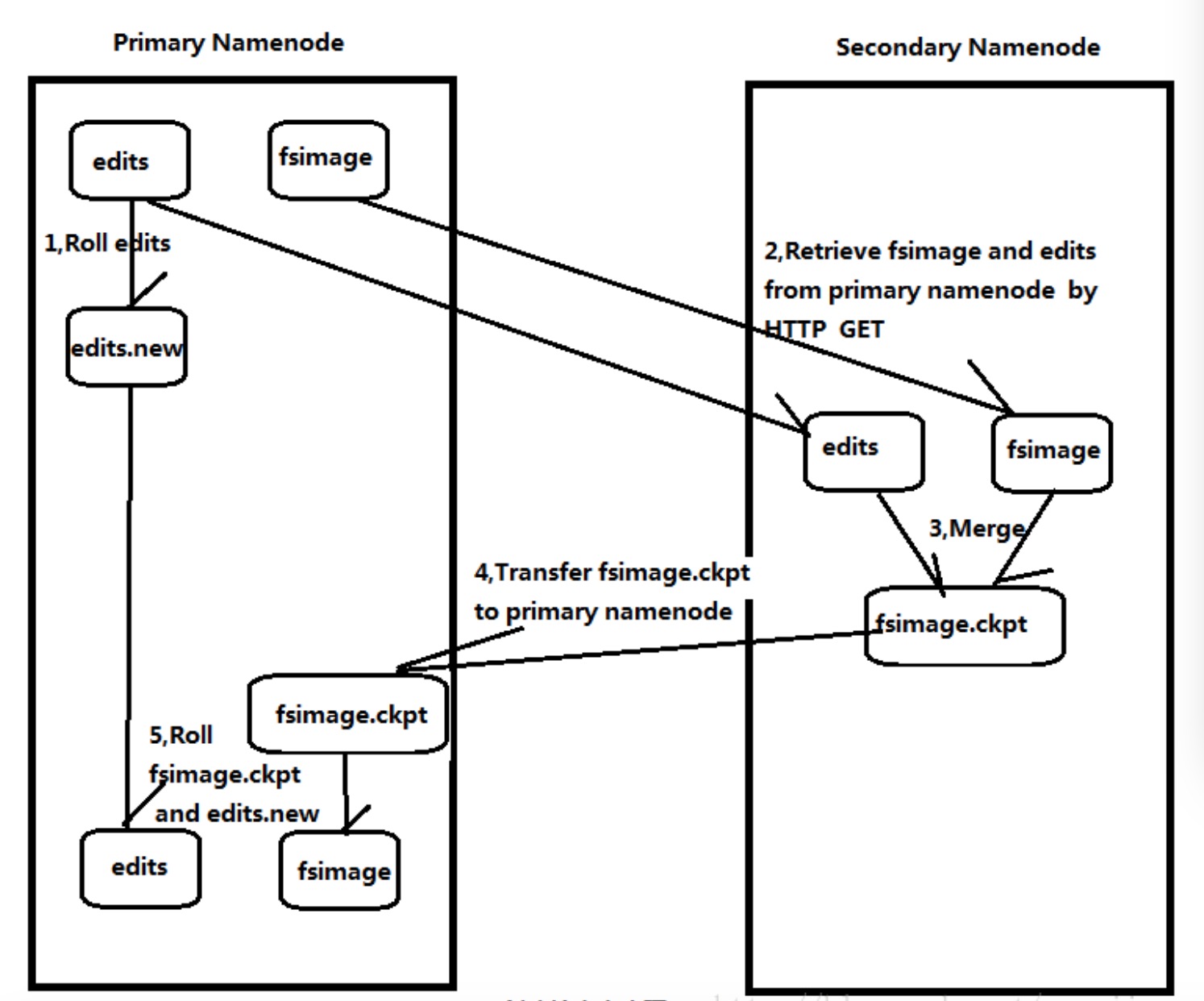
HDFS使用Master/Slave的架构来存储数据，主要分为四个部分，分别是Client、NameNode、DataNode和Secondary NameNode，每个部分支持不同的功能，下面将对不同的部分作详细的分析：

Client：最为客户端，主要站在系统的外层调用包装好的API。在进行文件上传的时候，客户端负责把文件进行切分成若干个Block，然后进行存储，客户端访问数据，首需要和NameNode进行交互，获取数据的位置信息，通过位置信息读取或者写入到指定的位置，客户端还可以通过指令来对HDFS进行管理，比如启动HDFS，上传文件，下载文件，删除文件，文件夹操作，关闭 HDFS等操作。

NameNode：它是整个文件系统的管理节点，它负责维护整个文件系统的目录树，文件或者目录的元信息和每个文件对应的数据块列表，接收并处理用户的请求，然后返回用户所需要的信息。其中有几个重要的文件分别是fsimage，edits，fstime，首先fsimage是元数据镜像文件，存储着某个时间段NameNode内存的元数据信息；edits保存操作日志文件；fstime保存着最近一次checpoint的时间；总的来说NameNode维护着2张表，一个是文件系统的目录结构，以及元数据信息，另一个是文件由许多数据块组成，这就需要一个对应关系，集群的所有元数据信息都存放在fsimage中，每次运行HDFS时，需要把fsimage中的信息都加载到内存中，在内存中的数据操作比较快，有利于数据的读写；

Datanode：提供真实文件数据的存储服务，整个集群中，Datanode有很多，用来存储海量的数据块，在Datanode节点中数据块是最基本的存储单位，每个文件都有固定的长度，在存储数据时，我们读取文件从位置0开始，按照固定的大小，顺序对文件进行编号，每个固定的块作为大文件的最小存储单位；HDFS的默认块大小是64M，如果一个文件小于一个数据块的大小，并不占用整个存储空间，设置较大数据块可以减轻namenode的压力，不然文件过多，这些块和文件的对应关系都存在namenode的内存中，对主节点的压力太大，影响整体的访问性能；数据副本默认是三份完整副本，随着存储框架的完善，对于这种对存储数据利用率只有1/3的情况，业界也在不断推敲新的方法，n个数据块不一定需要完全复制3n份才能保证数据的安全性，现在有一种叫做纠删码的方法，对于给定的n个数据块，可以根据相关的算法生成m检验块数据，这里已经包含了数据和检测数据，任何m数据丢失，都可以使用对应的重构算法回复出来原来的n块数据，这对存储的利用率有了很大的提高，不过对于这样的算法仍然在很多方面需要提高，包括丢失对于数据的恢复也加大了计算机的负担，在实际生产环境下，目前还不具有很强的实用性。

Secondary NameNode：首先需要明确的是它并不是namenode的副本，namenode是整个集群的管理者，主要通过fsimage和edits文件来实现管理者这个角色，在管理过程中请求不断发送给namenode，日志写入edits，对于存进hdfs的数据，edits和fsimage需要定期和并形成新的fsimage，而对于hdfs来说，只有重启才会实现2者的合并，重启也会带来很多的问题，比如edits较大，合并花费较多的时间，造成服务的延迟，而且内存中的数据很不安全，如果此时服务发生故障，前一次重启到现在出现故障这段时间存取的数据会丢失，基于以上这些弊端，于是有了Secondary NameNode，它主要用来解决namenode的日志和映射文件同步以及镜像的备份问题，解决方法如图：



从图上我们可以看出，对于edits和fsimage 会定向Secondary NameNode发送一本数据,注意此时在namenode节点生成edits.new文件，保存新的操作记录，这样可以保证namenode发生故障不至于丢失数据，这其实也就是对namenode的一种备份机制，在Secondary NameNode中会进行日志和fsimage的合并操作，在规定的时间会把fsimag.ckpt发送给namenode，然后覆盖fsimage，再将edits.new改成edits即可完成一次操作记录和镜像映射文件的合并，发送如果namenode出现故障， 然后重新启动，这时启动时，会加载Secondary NameNode里的fsimage。

从工作过程来看，Secondary NameNode主要是定期把namenode中的fsimage和edits合并后覆盖fsimage，避免edits不断积累变大，Secondary NameNode在另一台设备上运行，而且它和namenode需要具有相同容量的内存空间，对于合并后的fsimage其实就是namenode发生故障可以使用的副本。

第5章云存储服务平台的详细实现

本平台作为大型存储服务类平台，使用了这些高性能框架，能够理解这些框架的底层原理是开发程序最重要的一环，框架与框架之间能配合起来，理解框架的功能作用，在平台中担任的角色，在实现的过程中可能会出现各种各样的问题，我们有了前面对底层的原理的理解，这些知识流程上的bug，通过调试便能很快解决，下面我们将对各个模块的实现作详细的分析。

51 COS-SERVER模块

511模块介绍

这一模块是整个云存储平台的核心，

512功能介绍

513代码设计

52 WEB-SERVER

521模块设计

522功能介绍

523代码设计

524效果展示

525数据库设计

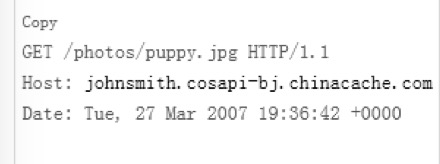
53 SDK模块

531 SDK模块介绍

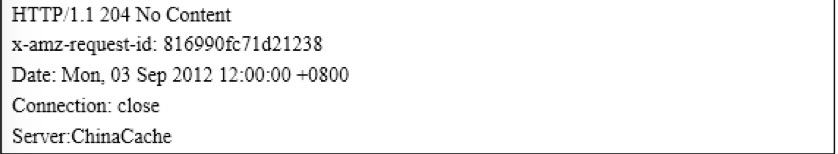
这个模块是面向程序员的模块，对于我们的云存储服务一方面我们向客户提供存储服务，另一方面，我们也可以向公司提供服务，对于公司拥有的海量数据需要存储，这时候，我们简单地提供Web访问界面可能已经远远不能满足企业用户的需求，对企业来说，海量的数据，丰富的需求类型，很多功能仅仅通过Web界面是难以满足的，数据的上传量较多的时候，需要企业用户自己跟我我们的云存储服务提供的SDK扩展和自己业务相配套的程序，也就是在我们的云存储服务的上层再包装一层，让存储服务可以通过程序来调用，对于我们开发的SDK，二次开发程序员只需要把响应的包添加到自己编写的应用程序当中去，设定响应的用户参数，就可以一程序的形式实现对云存储的访问，对于一些海量操作，如把目标资源池的数据拉取到本客户的云存储池中去，我们只需要调用目的接口，获取相应的信息，就可以循环拉取数据，这就大大提高了生产力，把一些本来需要人力重复性操作的事情，让程序按照指定的规则去执行。

532 SDK通信协议

云平台采用的是亚马逊S3协议，类似Http 协议，目前已经成为行业的标准，基本可以解决互联网行业的Web服务的所有需求，开发程序员按照协议的标准可以快速实现程序的各种请求处理，S3协议可靠，容易扩展，低延迟，是程序员实现处理请求的最佳协议标准，而且S3协议存储包括图片，视频，音乐，文档等等，具有广泛的存储类型，请求和返回的格式和Http协议类似，如下图：



请求格式图



返回格式图

533 SDK模块具体的实现

对于SDK的实现语言有很多种，云存储面向的开发语言需要覆盖大部分的应用程序会用到的语言，如Java，C，C++，Python，JS等，对于不同的语言在SDK的实现上没有太大的区别，主要在实现语法上存在一定差异，通过不同的语言包装HTTP请求，发送到服务端，获得相应的内容进行解析，底层解析原理我们饮用不同语言发行的包即可，我们只需要调用相关的包，把业务组合起来即可，下面我们对Java进行举例说明。

对于Java-SDK的各个功能，下面通过简单的调用实例来体现如何通过SDK访问云存储的服务的。对于包装的SDK，我们主要需要实现SDK的根据S3协议发送和接收信息，对于底层的处理细节，目前对于各种语言已经有了很完善的发行包了，对于http-core，http-client等包，可以对请求进行包装发送，S3协议的格式和Http协议本身就具有很大的相似度，具体的区别主要在协议处理层的规定上增加了很多头字段类型，头字段所具有的属性也有所不同，所以在设置这些头信息时，要严格根据亚马逊S3协议的官方文档去设计，对于协议调用的逻辑流程，我们采用了阿里发行的公开版本，它们在处理流程上相对来说已经比较完善，下面我们将对于不同请求类型作详细介绍。

1定义客户端

public static AmazonS3Client getClient(boolean v4) {

ClientConfiguration configuration = new ClientConfiguration();

if (v4) {

configuration.setSignerOverride("AWS4SignerType");

}

configuration.setMaxErrorRetry(0);

AmazonS3Client client = new AmazonS3Client(new AWSCredentials() {

@Override

public String getAWSSecretKey() {

return accessKeySecret; //hbase

}

@Override

public String getAWSAccessKeyId() {

return accessKeyId;//hbase

}

}, configuration);

client.setEndpoint(endpoint);

return client;

}

这个函数是在客户端访问云存储服务的时候，首先需要根据用户的信息定义客户端对象，这里包括了AK，SK，EndPoint，后续的所有访问操作都从这个对象发起。

2 设定bucket的权限

CreateBucketRequest createBucketRequest = new CreateBucketRequest(newBucket);

createBucketRequest.setCannedAcl(CannedAccessControlList.PublicReadWrite);

对于不同bucket可以设置不同的访问权限，分别有可读权限，可写权限，可读写权限。

3 上传对象

String key = dirKey + "chinacache" + i;

ObjectMetadata meta = new ObjectMetadata();

meta.setHeader("x-amz-aaaa", 200);

meta.setHeader("x-amz-ddddd", 1);

meta.setHeader("x1-amz-ddddd", 1);

PutObjectRequest putObjectRequest2 = new PutObjectRequest(bucketName, key, createSampleFile());

putObjectRequest2.setMetadata(meta);

lient.putObject(putObjectRequest2);

申请对象请求，设置桶名和URI，通过createSampleFile()函数创建文件，并注入到请求中，对于一个文件对象，可以根据ObjectMetadata()对象设置文件对象的属性，比如对象的长度，修改的时间，MD5值，对象的编码，请求的过期时间等等，这些属性是用来描述对象的，在云存储中的存储，真实数据都以流存在，对于流的访问必须先访问流的元数据再定位到流进行操作。

4 获得对象

String key = "dir/chinacache2";

GetObjectRequest GET = new GetObjectRequest(bucketName, key);

S3Object o = client.getObject(getObjectRequest2);

try (InputStream in = o.getObjectContent()) {

int r = -1;

while ((r = in.read()) != -1) {

//可以对流内容进行操作；

}

} catch (IOException e) {

获取对象使用GetObjectRequest()函数，通过bucket参数和key参数获取相关对象，通过客户端调取获得S3对象，对象中有不同的元数据信息，一般需要通过getObjectContent获得对象的真实流数据，在进行传输或者刷盘等操作。

5 复制对象

CopyObjectRequest copyObjectRequest = new CopyObjectRequest(bucketName, "dir/chinacache1", bucketName, "dir/chinacacheCOS1");

client.copyObject(copyObjectRequest);

在云存储中的CopyObjectRequest()对象用来再云存储中进行对象复制，复制可以在不同的bucket之间进行。

6 创建目录

PutObjectRequest putObjectRequest = new PutObjectRequest(bucketName, dirKey, "");

ObjectMetadata meta = new ObjectMetadata();

putObjectRequest.setMetadata(meta);

client.putObject(putObjectRequest);

对于PutObjectRequest()，不制定dirKey对应的文件，此时执行的操作会建立一个空目录。

7 列出目录下的内容

ListObjectsRequest listObjectsRequest = new ListObjectsRequest();

listObjectsRequest.setBucketName(bucketName);

listObjectsRequest.setPrefix(dirKey);

listObjectsRequest.setDelimiter("/");

ObjectListing list = client.listObjects(listObjectsRequest);

System.out.println("getBucketName="+list.getBucketName());

System.out.println("getDelimiter="+ list.getDelimiter());

System.out.println("getMarker=" + list.getMarker());

System.out.println("getMaxKeys=" + list.getMaxKeys());

System.out.println("getPrefix=" + list.getPrefix());

System.out.println("isTruncated=" + list.isTruncated());

System.out.println("getCommonPrefixes=" + list.getCommonPrefixes());

System.out.println("getNextMarker=" + list.getNextMarker());

List<S3ObjectSummary> objs = list.getObjectSummaries();

System.out.println("list 成功！");

for (S3ObjectSummary o : objs) {

System.out.println(o.getBucketName());

System.out.println(o.getKey());

System.out.println(o.getSize());

System.out.println(o.getStorageClass());

System.out.println(o.getETag());

System.out.println();

}

首先通过ListObjectsRequest()函数获取所有的文件和文件夹，对于获取的list内容都包装在了ObjectListing()中，list中有bucket信息方法getBucketName，目录分割符方法getDelimiter，一次最多列出文件数getMaxKeys，下一次列出文件其实文件getMarker，文件是否是最后一个文件isTruncated，列出文件夹使用getCommonPrefixes，列出文件使用getObjectSummaries。

5332

8 删除对象

DeleteObjectRequest deleteObjectRequest = new DeleteObjectRequest(bucketName, dir);

client.deleteObject(deleteObjectRequest);

DeleteObjectsRequest deleteObjectRequest1 = new DeleteObjectsRequest(bucketName).withKeys(dirKey + prefixe + 1).withKeys(dirKey + prefixe + 2);

client.deleteObjects(deleteObjectRequest1);

System.out.println("delete 成功！");

通过DeleteObjectRequest()设置删除的对象属性包括bucket和URI，如果需要同时删除多个对象时，可以使用withKeys()方法添加多个需要删除的对象的信息，客户端调用deleteObjects()方法进行批量删除操作。

54 FTP模块

541 FTP模块介绍

FTP模块是为客户提供更稳定更快速数据传输的应用应用程序，程序开发是在已经发行的FTP程序基础上开发而来，对于内部的通信协议不作修改，对外层提供的服务仍然是FTP服务，客户端可以沿用FTP客户端程序的大部分操作，只要支持FTP协议的客户端软件都可以通过指定的IP和用户登陆信息对服务端资源的访问。对于云存储客户来说，访问数据的频度会很高，而且对于一些大文件的传输往往要求较高的传输速度，本地网络有时也会出现不稳定的情况，大文件的上传可能会出现间断的情况，如果此时直接对文件进行重传，软件的使用效率会很低，此时通过客户端软件配合云存储实现断点续传，对于大文件进行切割，然后分片存储，在存储的过程中，分片的信息会进行统计，客户端设计一个数据结构用来存储分片信息，当到达某一分片时断了，这时发起重传请求，对于这个文件已经有过传输，此时不会重头开始传输，调用客户端软件的关联信息，找出下一次流数据切割的偏移量，执行后续的分片的存储，大大提高了程序的执行效率。

542 FTP模块具体实现

总的来说个个模块的实现其实是各种抽象类和接口的继承实现过程，一般来说发起服务时会注入默认的实现类，对接到云存储时，需要对用户的验证机制，文件系统，云文件，指令的操作等进行重新注入，实现自己的功能方法，启动服务后，即可实现对接云存储的云服务。

下面将对各个模块的具体实现做详细的分析：

5421 CosFtpServer

作为整个服务的主线程，对于服务所需要初始化的模块都进行了相关的调用，主要代码如下：

MyListenerFactory listenerFactory = new MyListenerFactory();  
listenerFactory.setPort(21);  
CommandFactoryFactory c = new CommandFactoryFactory();  
c.setUseDefaultCommands(false);  
c.addCommand("SITE\_DESCUSER", new SITE\_DESCUSER());  
c.addCommand("SITE\_HELP", new SITE\_HELP());  
c.addCommand("SITE\_STAT", new SITE\_STAT());  
c.addCommand("SITE\_WHO", new SITE\_WHO());  
c.addCommand("SITE\_ZONE", new SITE\_ZONE());  
FtpServerFactory ftpServerFactory = new FtpServerFactory();  
ftpServerFactory.addListener("default", listenerFactory.createListener());  
ftpServerFactory.setUserManager(new MyUserManager());  
ftpServerFactory.setCommandFactory(c.createCommandFactory());  
ftpServerFactory.setFileSystem(new MyFileSystemFactory());  
FtpServer ftpServer = ftpServerFactory.createServer();

ftpServer.start();

首先定义MyListenerFactory()函数用来创建监听器实例的工厂类，通过setPort()函数设置监听端口，再通过createListener()函数调用创建监听的实例；

CommandFactoryFactory()用来设置指令，如果我们通过setUseDefaultCommands()把属性设置成true，调用原生的包里的一系列指令，这种实现是基于最基础的指令操作，对于普通环境下，包括文件系统也是常用的操作系统情况下，是不需要对这一层进行修改的，对于云存储的特殊需求，需要对执行的指令进行修改来适应访问云存储的各种操作，此时通过setUseDefaultCommands()把属性设置成false，然后调用addCommand()函数添加自己重新实现的指令；

MyUserManager()实现云存储的用户验证，由于用户表在云存储中的Hbase表里，而对于我们这个小型应用程序，如果把Hbase加入到程序中来，而仅仅用来做用户验证使用，机器的大部分性能会被浪费，这里我们采用的是自己实现一个用户管理类调用云服务接口，获取用户信息和登陆信息进行匹配。

MyFileSystemFactory()用来创建文件系统，对于云存储不存在像windows下文件的文件系统实现类，云中的对象和属性分开存储，需要我们设计一个虚拟的文件系统类提供FTP服务底层的支持。

FtpServerFactory()是FTP服务最核心的类，以上这些实现类都是通过这个类调用相关的注入方法把这些类注入主服务，调用时抽象类时，使用的便是这些注入的实现类，最后通过start()函数发起服务。整个服务启动，对外提供服务。

5422 用户设计

用户模块主要实现了MyBaseUser和MyUserManager，MyBaseUser是一个用户实体类，其中定义了用户的UserInfo和AmazonS3Client。MyUserManager通过MyBaseUser的调用，实现用户的验证是否通过。

MyBaseUser的实现主要通过如下的方法：

public static UserInfo doGetUserInfo(String name) throws IOException, JSONException {  
    URL url = new URL("http://cos-wz.chinacache.com/doInside");  
    HttpURLConnection httpUrlConnection = (HttpURLConnection) url.openConnection();  
    httpUrlConnection.setConnectTimeout(10000);  
    httpUrlConnection.setReadTimeout(10000);  
    httpUrlConnection.setRequestMethod("POST");  
    httpUrlConnection.setDoOutput(true);  
    httpUrlConnection.connect();  
    try(OutputStream os = httpUrlConnection.getOutputStream()){  
        JSONObject jo = new JSONObject();  
        jo.put("action", "queryAksk");  
        jo.put("name", name);  
        os.write(jo.toString().getBytes("UTF-8"));  
        os.flush();  
    }  
    if(httpUrlConnection.getResponseCode() == 200) {  
        try (InputStream in = httpUrlConnection.getInputStream()) {  
            String respBody = IOUtils.toString(in);  
            JSONObject jo = new JSONObject(respBody);  
  
            int code = jo.getInt("code");  
            if (code == 100){  
                String info = jo.getString("info");  
                info = Utils.decode(info);  
  
                int idx = info.indexOf("@");  
                String ak = info.substring(0, idx);  
  
                int idx2 = info.indexOf("@", idx+1);  
                String sk = info.substring(idx+1, idx2);  
  
                String md5Passwd = info.substring(idx2+1);  
  
                UserInfo userInfo = new UserInfo(ak, sk, md5Passwd);  
                return userInfo;  
            }else if(code == 404){  
                return NOT\_EXIST\_USER;  
            }  
        }  
    }else{  
        log.error("failed to get ak sk of user:" + name + ", error code:" + httpUrlConnection.getResponseCode());  
    }  
  
    return null;  
}

云存储中有获取用户信息的相关接口，通过裸写Http请求可以实现用户信息的获取，对于返回的respBody进行解析，得到UserInfo中的每个字段的信息，然后包装到一个对象中。

MyUserManager是用户信息验证的管理层，在原生的Ftp服务器中有用户抽象类AbstractUserManager，我们只需要继承这个抽象类，把主要的方法进行重写，就可以满足现有的验证机制，实现方法如下：

public User getUserByName(String username) throws FtpException {  
    if(Utils.noRealValue(username)){  
        return null;  
    }  
    try {  
        MyBaseUser user = new MyBaseUser();  
        user.setEnabled(true);  
        user.setName(username);  
        user.initUserInfo();  
        user.setPassword(user.userInfo.md5Passwd);  
        user.setHomeDirectory("./");  
        user.setMaxIdleTime(100);  
        List<Authority> authorities = new ArrayList<Authority>();  
        authorities.add(new WritePermission());  
        authorities.add(new ConcurrentLoginPermission(1000, 1000));  
        authorities.add(new TransferRatePermission(1000000, 1000000));  
        user.setAuthorities(authorities);  
        return user;  
    }catch (IOException | JSONException e) {  
        throw new FtpException(e);  
    }  
}

getUserByName通过调用MyBaseUser方法获取需要验证的标准用户信息，同时赋予相关的读写和传输权限，并设置对应的过期时间，为下一步的验证操作提供user对象。

public User authenticate(Authentication authentication) throws AuthenticationFailedException {  
    if (authentication instanceof UsernamePasswordAuthentication) {  
        UsernamePasswordAuthentication upauth = (UsernamePasswordAuthentication) authentication;  
  
        String userName = upauth.getUsername();  
        String password = upauth.getPassword();  
  
        if (userName == null) {  
            throw new AuthenticationFailedException("Authentication failed");  
        }  
        if (password == null) {  
            password = "";  
        }  
  
        try {  
            User user = getUserByName(userName);  
            if(user == null || !getPasswordEncryptor().matches(password, user.getPassword())){  
                throw new AuthenticationFailedException("Authentication failed, failed to match password");  
            }  
            return user;  
        }catch(FtpException e){  
            throw new AuthenticationFailedException("Authentication failed", e);  
        }  
    }else {  
        throw new IllegalArgumentException(  
                "Authentication not supported by this user manager");  
    }

authenticate方法把客户端传过来的用户信息和从云存储库里拿到的信息进行对比，再包装好一系列的异常处理，让验证逻辑更加完整，至此，Ftp服务器的用户验证就顺利对接到了云存储平台。

5423文件系统和指令操作设计

文件系统部分需要对客户返回的数据流进行过滤，对于云存储的返回的目录结构原理和用户有所区别，所以需要设计一套让用户如同在本地的文件系统上进行操作一样的结构，也就是虚拟的云文件系统，主要由MyFileSystemFactory，MyFileSystemView，MyFtpFile组成，下面对这部分的设计作详细的介绍：

1. MyFileSystemFactory

这个类是服务启动是添加文件系统类型的，默认情况下，使用自带的文件系统，在我们的程序中，我们修改成了重新实现的文件系统，注册此系统的过程中，需要对本段代码加锁，防止同时设置多个文件系统，发生文件系统的覆盖，除了指定使用的文件系统外，还需要对初始目录作相应的验证，确保初始目录的正确性，代码如下：

synchronized (user) {  
    if (createHome) {  
        String homeDirStr = user.getHomeDirectory();  
        File homeDir = new File(homeDirStr);  
        if (homeDir.isFile()) {  
            LOG.warn("Not a directory :: " + homeDirStr);  
            throw new FtpException("Not a directory :: " + homeDirStr);  
        }  
        if ((!homeDir.exists()) && (!homeDir.mkdirs())) {  
            LOG.warn("Cannot create user home :: " + homeDirStr);  
            throw new FtpException("Cannot create user home :: "  
                    + homeDirStr);  
        }  
    }  
    FileSystemView fsView = new MyFileSystemView(user,  
            caseInsensitive);  
    return fsView;  
}

（2）MyFileSystemView

文件系统的主要实现都在这个类中，对于用户的每一次操作，都涉及到当前操作对应的根目录，当前目录，当前文件，当前用户，这些信息需要用一个类统一去管理，一般来说，一个请求到达服务端，经过一系列处理，到达指令类中，通过MyFileSystemView调用具体的方法执行相应的操作，这个类最重要的方法是根据用户信息获得MyFtpFile的getFile方法，具体实现如下：

public MyFtpFile getFile(String file) {  
  
    // get actual file object  
    boolean isFile = false;  
    String physicalName = getPhysicalName(rootDir,  
            currDir, file, caseInsensitive);  
  
    if ( !file.endsWith("/")) {  
        isFile = true;  
    }  
    // strip the root directory and return  
    String userFileName = physicalName.substring(rootDir.length() - 1);  
  
    Pair<String, String> bucketAndObject = S3File.parseBucketAndObject(userFileName);  
    S3File s3file;  
    if(bucketAndObject.equals(ROOT\_FILE)){  
        s3file = ROOT\_S3\_FILE;  
    }else if(bucketAndObject.second() != null) {  
         s3file = S3File.S3FileCache.getS3File(bucketAndObject.first(), bucketAndObject.second(), isFile);  
    }else{  
        s3file = S3File.S3FileCache.getS3File(bucketAndObject.first(), isFile);  
    }  
  
    return new MyFtpFile(userFileName, s3file, user);  
}

（3）MyFtpFile

这个类是FTP服务器的核心操作类，大部分操作最终都会传递到MyFtpFile，在这里执行真正的操作，比如数据的删除，修改，上传，下载等等，类的基本信息包括fileName，file，user，这些信息可能在进行相关操作时会使用，具体的文件操作需要借助于云存储的SDK实现，下面是具体操作的例子：

move方法实现文件的移动：

 public boolean move(final FtpFile dest) {  
        try {  
            MyFtpFile myDest = (MyFtpFile) dest;  
            System.out.println("prefix:" + file.objectName);  
            getObjectList(file.objectName + "/"); //get objectList  
            List<BatchMoveRequest.KeyVersion> keys = new LinkedList<>();  
            for (String object : objectList) {  
                String key = object.substring(file.objectName.length()+1);  
                System.out.println(key);  
                keys.add(new BatchMoveRequest.KeyVersion(key, null, file.objectName+ "/", dest+"/"));  
            }  
            BatchMoveRequest batchMoveRequest = new BatchMoveRequest(file.bucketName,file.bucketName);  
            batchMoveRequest.setKeys(keys);  
            DeleteObjectsResult deleteObjectsResult =  getS3Client().batchMoveObjects(batchMoveRequest);      
            return true;  
        }catch (Throwable e){  
            LOG.error(e.getMessage(),e);  
            return false;  
        }  
    }

getFtpFile可以实现把当前目录下的文件和文件夹以集合的形式返回到客户端：

public List<FtpFile>  getFtpFile(ObjectListing objectListing) {  
    FtpFile[] virtualObjects = new FtpFile[objectListing.getObjectSummaries().size() + objectListing.getCommonPrefixes().size()];  
    int index = 0;  
    for(String commonPrefixe : objectListing.getCommonPrefixes()) {  
        S3File s3file = new S3File(file.bucketName, commonPrefixe, false,  
                0, System.currentTimeMillis());  
        S3File.S3FileCache.addCache(file.bucketName, s3file);  
        virtualObjects[index] = new MyFtpFile("/" + commonPrefixe, s3file, user);  
        index = index + 1;  
    }  
    if ( objectListing.getObjectSummaries().size() > 0 ) {  
        for (S3ObjectSummary objectSummarie : objectListing.getObjectSummaries()) {  
            S3File s3file = new S3File(file.bucketName, objectSummarie.getKey(), true,  
                    objectSummarie.getSize(), objectSummarie.getLastModified().getTime());  
            S3File.S3FileCache.addCache(file.bucketName, s3file);  
            virtualObjects[index] = new MyFtpFile("/" + objectSummarie.getKey(), s3file, user);  
            index = index + 1;  
        }  
    }  
  
    return Collections.unmodifiableList(Arrays.asList(virtualObjects));  
}

（4）MyFtpHandlerAdapter

这个类主要用来分配逻辑处理的适配器，如各个处理方法中MyFtpIoSession的注入，比如：

public void messageReceived(IoSession session, Object message)  
        throws Exception {  
    MyFtpIoSession ftpSession = new MyFtpIoSession(session, context);  
    FtpRequest request = new DefaultFtpRequest(message.toString());  
  
    ftpHandler.messageReceived(ftpSession, request);  
}

（5）MyNioListener

这是整个服务启动过程中仅次于CosFtpServer的一个重要类，首先注册acceptor，配置acceptor的相关属性如缓冲大小，过期时间，然后添加过滤链，实现数据过滤，日志过滤以及编码的转换等等，然后注入处理适配器和上下文，就可以开始监听某一段口了，当有请求来到这个端口，服务器就可以对请求进行处理了，实现如下：

public synchronized void start(FtpServerContext context) {  
    if(!isStopped()) {  
        // listener already started, don't allow  
        throw new IllegalStateException("Listener already started");  
    }  
    try {  
        this.context = context;  
        acceptor = new NioSocketAcceptor(Runtime.getRuntime().availableProcessors());  
        if (getServerAddress() != null) {  
            address = new InetSocketAddress(getServerAddress(), getPort());  
        } else {  
            address = new InetSocketAddress(getPort());  
        }  
        acceptor.setReuseAddress(true);  
        acceptor.getSessionConfig().setReadBufferSize(2048);  
        acceptor.getSessionConfig().setIdleTime(IdleStatus.BOTH\_IDLE, getIdleTimeout());  
        acceptor.getSessionConfig().setReceiveBufferSize(512);  
        MdcInjectionFilter mdcFilter = new MdcInjectionFilter();  
        acceptor.getFilterChain().addLast("mdcFilter", mdcFilter);  
        SessionFilter sessionFilter = getSessionFilter();  
        if (sessionFilter != null) {  
            // add and IP filter to the filter chain.  
            acceptor.getFilterChain().addLast("sessionFilter",  
                    new MinaSessionFilter(sessionFilter));  
        }  
        acceptor.getFilterChain().addLast("threadPool",  
                new ExecutorFilter(context.getThreadPoolExecutor()));  
        acceptor.getFilterChain().addLast("codec",  
                new ProtocolCodecFilter(new FtpServerProtocolCodecFactory()));  
        acceptor.getFilterChain().addLast("mdcFilter2", mdcFilter);  
        acceptor.getFilterChain().addLast("logger", new FtpLoggingFilter());  
        if (isImplicitSsl()) {  
            SslConfiguration ssl = getSslConfiguration();  
            SslFilter sslFilter;  
            try {  
                sslFilter = new SslFilter(ssl.getSSLContext());  
            } catch (GeneralSecurityException e) {  
                throw new FtpServerConfigurationException("SSL could not be initialized, check configuration");  
            }  
            if (ssl.getClientAuth() == ClientAuth.NEED) {  
                sslFilter.setNeedClientAuth(true);  
            } else if (ssl.getClientAuth() == ClientAuth.WANT) {  
                sslFilter.setWantClientAuth(true);  
            }  
            if (ssl.getEnabledCipherSuites() != null) {  
                sslFilter.setEnabledCipherSuites(ssl.getEnabledCipherSuites());  
            }  
            acceptor.getFilterChain().addFirst("sslFilter", sslFilter);  
        }  
        handler.init(context, this);  
        acceptor.setHandler(new MyFtpHandlerAdapter(context, handler));  
        try {  
            acceptor.bind(address);  
        } catch (IOException e) {  
            throw new FtpServerConfigurationException("Failed to bind to address " + address + ", check configuration", e);  
        }  
        updatePort();  
    } catch(RuntimeException e) {  
        stop();  
        throw e;  
    }  
}

（6）Command和MyIODataConnection

Ftp对接云存储需要通过一系列不同的指令触发不同的操作，Ftp原生的指令很多，在这个程序中用到的并不多，我们需要改写如下指令MyDELE，MyMKD，MyMLSD，MyRETR，MyRMD，MyRNTO，MySTOR分别执行删除文件，新建目录，列表，下载，删除目录，移动文件，上传文件，下面我们贴出上传文件的主要代码部分：

MyIODataConnection dataConnection;  
try {  
    dataConnection = (MyIODataConnection) session.getDataConnection().openConnection();  
} catch (Exception e) {  
    LOG.debug("Exception getting the input data stream", e);  
    session.write(LocalizedDataTransferFtpReply.translate(session, request, context,  
            FtpReply.REPLY\_425\_CANT\_OPEN\_DATA\_CONNECTION, "STOR",  
            fileName, file));  
    return;  
}  
  
// transfer data  
boolean failure = false;  
OutputStream outStream = null;  
long transSz = 0L;

try {  
    //skiplen ignored  
    if ( file.file.objectName != "")//不可以直接向根目录下传文件  
        transSz = dataConnection.transferFromClient(session.getFtpletSession(), file);  
    else  
        throw new IOException();  
    // attempt to close the output stream so that errors in  
    // closing it will return an error to the client (FTPSERVER-119)  
    if(outStream != null) {  
        outStream.close();  
    }  
  
    LOG.info("File uploaded {}", fileName);  
  
    // notify the statistics component  
    ServerFtpStatistics ftpStat = (ServerFtpStatistics) context  
            .getFtpStatistics();  
    ftpStat.setUpload(session, file, transSz);  
  
} catch (SocketException ex) {  
    LOG.debug("Socket exception during data transfer", ex);  
    failure = true;  
    session.write(LocalizedDataTransferFtpReply.translate(session, request, context,  
            FtpReply.REPLY\_426\_CONNECTION\_CLOSED\_TRANSFER\_ABORTED,  
            "STOR", fileName, file));  
} catch (IOException ex) {  
    LOG.debug("IOException during data transfer", ex);  
    failure = true;  
    session  
            .write(LocalizedDataTransferFtpReply  
                    .translate(  
                            session,  
                            request,  
                            context,  
                            FtpReply.REPLY\_551\_REQUESTED\_ACTION\_ABORTED\_PAGE\_TYPE\_UNKNOWN,  
                            "STOR", fileName, file));  
} finally {  
    // make sure we really close the output stream  
    IoUtils.close(outStream);  
}

指令层主要负责封装客户的请求，对于每个步骤都有完善的异常捕获机制，保证每个步骤的执行是科学的，关键操作是transferFromClient函数，调用真正的上传操作，函数设计如下：

public final long transferFromClient(FtpSession session,  
                                     final MyFtpFile out) throws IOException {  
    TransferRateRequest transferRateRequest = new TransferRateRequest();  
    transferRateRequest = (TransferRateRequest) session.getUser()  
            .authorize(transferRateRequest);  
    int maxRate = 0;  
    if (transferRateRequest != null) {  
        maxRate = transferRateRequest.getMaxUploadRate();  
    }  
  
    InputStream is = getDataInputStream();  
    try {  
        return transfer(session, false, is, out, maxRate);  
    } finally {  
        IoUtils.close(is);  
    }  
}

MyIODataConnection主要是建立数据通道，发起请求获得相应的数据，通过连接把客户端的数据上传到云存储系统中，在这一层中所有的数据将以流的形式执行后续的相关操作，这里再一次把数据包装成传输请求，发起transfer函数，实现如下：

private final long transfer(FtpSession session, boolean isWrite,  
                            final InputStream in, final MyFtpFile out, final int maxRate)  
        throws IOException {  
  
    BufferedInputStream bis = IoUtils.getBufferedInputStream(in);  
    String tmpFileNamePrefix = MD5Hash.digest(out.file.bucketName + "/" + out.file.objectName).toString();  
    File f = File.createTempFile(tmpFileNamePrefix, UUID.randomUUID().toString(), tmpFileDir);  
  
    try {  
        try (FileOutputStream fos = new FileOutputStream(f)) {  
            Utils.streamCopy(bis, fos);  
        }  
  
        AmazonS3Client s3Client = out.getS3Client();  
        ObjectMetadata meta = new ObjectMetadata();  
        meta.setContentLength(f.length());  
        try(FileInputStream fin = new FileInputStream(f)) {  
            PutObjectRequest request = new PutObjectRequest(out.file.bucketName, out.file.objectName, fin, meta);  
            s3Client.putObject(request);  
        }  
    }finally {  
        if(! f.delete()){  
            LOG.error("failed to delete tmp file:" + f.getAbsolutePath());  
        }  
    }  
    return 0L;  
}

这是上传的最底层操作，也是云存储系统的最表层操作，在这一层拿到数据流和元数据信息，组织S3请求，实现上传操作。

55污染源数据展示模块

551模块介绍

对于这一模块是对Hbase数据库的一个应用，我们需要通过相关的监测设备收集船只的排放污染数据， 风速风向数据以及附近船只数据，将设备安装用于通信的SIM卡，发射数据信号，也就相当于设备可以通过SIM卡连上互联网，我们的数据集群安装在单位的内网中，此时数据采集设备需要把数据发送到分配给单位的唯一外网IP和制定的端口，在单位的局域网做一个ip和端口映射，这样就可以把数据采集到，通过Hbase的API进行数据的存取，从而实现对这一模块数据的管理。

本论文所介绍的云存储系统的Hbase部分可以为我们的污染源平台提供底层的数据支持，对于污染源的数据具有单挑数据量小，访问频繁，数据跨度大等特点，所以我们可以充分利用Hbase这样键值型的数据库进行数据频繁调用和显示，对于每条数据，我们都可以通过本条数据的键找到，Hbase本身的存储特性决定了它可以通过键很快找到对应的数据，在传统的数据库中，超多用户同时向服务发起请求时，关系型数据库很难承受这种高频访问的压力，而数据库通过Hbase来存储，对客户的反馈就能大大地提高，而且Hbase底层以来HDFS来存储其数据块，其本身的多副本保存，让污染源数据得到更加安全的保存。

552功能介绍

（1）污染源曲线

这一部分主要是把数据库中按照指定的条件筛选出来的数据展示在前台界面上，这一部分主要使用的是echarts控件，不同的污染物类型通过不同的曲线展示出来，控件可以对数据的范围按照时间进行扩大和缩小，纵轴的左边和右边分别代表不同的污染物数据的坐标。

（2）污染元素S含量的计算

对于图中的曲线是有一定的波动的，大部分的数据呈现大致水平线，小部分的数据会形成波峰，这部分波峰是由于船只通过监测设备的附近，排放了大量的污染物，造成空气中的不同类型污染物的浓度突然变大，这时我们可以通过选择指定的时间段计算这段时间的不同污染物的平均浓度和附近的背景浓度，比如对于初期我们主要测量CO2和SO2的浓度来计算船只的排放是否超标，计算公式为：0.232（尾气SO2-平均SO2）／（尾气CO2-平均CO2），算出这个参数提供用户去分析船只的排污量是否达标。

（3）嫌疑船只的锁定

如果某个波峰此时我们断定污染物排放已经超标，这时可以点击对应的曲线的某个时间点，跳转到这个时刻的附近1.5KM的所有船只地图，对于这个地图中心位置放置箭头标示此时的风速风向，根据风速风向可以判断出哪些船只是嫌疑船只，此时我们通过点击相应的船只就可以锁定嫌疑船只，这样就能将这个船只的信息保存下来，存到嫌疑船只列表中去，可供我们在后续使用的过程中查看，船只地图默认显示某一时刻到距此刻前5分钟的所有船只，界面也可以设置更多的选项，显示其他时间间隔内的船只轨迹。

（4）嫌疑船只数据的展示

这部分是嫌疑船只信息表的数据展示，对于这个部分，首先需要对表中的数据按照指定的时间段进行筛选，然后行进行显示，每一行显示若干主要字段，末尾有详细信息展示和数据删除操作供用户使用。

553主要代码设计(Hbase层的代码写出来即可)

(1)逻辑处理层

这一层是这一模块最重要的一个子部分，对于客户发送的请求，通过各级网络，最终到达处理层，在这里我们可以解析请求request中的信息，然后根据请求中的操作类型执行相应的操作，下面对本模块的几个主要的处理类进行详细分析。

（a）AisDynamicAction

这层主要是获取船只的数据的，根据发来的请求，筛选出所需要的数据，按照指定的格式把数据返回给前端，最常用的处理方法是listMmsi:

**public** String listMmsi() {

String flag=**null**;

Date dateBeg=**null**;

Date dateEnd=**null**;

SimpleDateFormat formatter=**new** SimpleDateFormat("yyyy/MM/dd hh:mm:ss");

System.*out*.println(checktime);

**try** {

dateEnd=formatter.parse(checktime);

} **catch** (ParseException e1) {

// **TODO** Auto-generated catch block

e1.printStackTrace();

}

//minute

//Long time=Long.parseLong(checktime);

dateBeg=**new** Date(dateEnd.getTime()-Integer.*parseInt*(minute)\*60000);

**try** {

JSONArray jsonArray=**new** JSONArray();

JSONArray jsonArray\_mmsi=**new** JSONArray();

JSONArray jsonArray\_length=**new** JSONArray();

JSONArray jsonArray\_local=**new** JSONArray();

//JSONArray jsonArray\_CO2=new JSONArray();

AisDynamicQuery query = newQuery(AisDynamicQuery.**class**,"ptime asc");

query.setPtimeBegin(dateBeg);

query.setPtimeEnd(dateEnd);

List<AisDynamic> list=aisDynamicManager.findPage(query).getResult();

Iterator<AisDynamic> it=list.iterator();

ArrayList<Double[]> arrayListdouble = **new** ArrayList<Double[]>();

ArrayList<Long> arrayListlongArray = **new** ArrayList<Long>();

TreeMap<Long,ArrayList<Double[]>> tree\_double\_mssi= **new** TreeMap<Long,ArrayList<Double[]>>();

**while**(it.hasNext()){

AisDynamic aisDynamic=(AisDynamic)it.next();

Double Longitude=aisDynamic.getLongitude();

Double Latitude=aisDynamic.getLatitude();

Double[] local={Longitude,Latitude};

//map.put(aisDynamic.getMmsi(), local);

arrayListdouble.add(local);

arrayListlongArray.add(aisDynamic.getMmsi());

**if** (tree\_double\_mssi.containsKey(aisDynamic.getMmsi())) {

tree\_double\_mssi.get(aisDynamic.getMmsi()).add(local);

}**else** {

ArrayList<Double[]> doubles = **new** ArrayList<Double[]>();

doubles.add(0, local);

tree\_double\_mssi.put(aisDynamic.getMmsi(), doubles);

}

}

Iterator ietIterator = tree\_double\_mssi.entrySet().iterator();

**for** (**int** i =0;i<tree\_double\_mssi.size();i++) {

Map.Entry ent=(Map.Entry)ietIterator.next();

Long mmsi=(Long) ent.getKey();

ArrayList<Double[]> arrayLists =(ArrayList<Double[]>) ent.getValue();

AisStaticQuery queryaisSt = newQuery(AisStaticQuery.**class**,*DEFAULT\_SORT\_COLUMNS*);

queryaisSt.setMmsi(mmsi);

AisStatic aisStatic=(AisStatic)aisStaticManager.findPage(queryaisSt).getResult().get(0);

Double length=aisStatic.getLength();

          // String keyt=ent.getKey().toString();

            //String valuet=ent.getValue().toString();

Double longtitude = **null**;

Double latitude = **null**;Double distance = **null**;Double Longdis = **null**;Double Latdis=**null**;

**for**(**int** j= 0 ;j<arrayLists.size();j++){

jsonArray\_mmsi.add(mmsi);

jsonArray\_length.add(length);

longtitude = arrayLists.get(j)[0];

latitude = arrayLists.get(j)[1];

distance=*getDistance*(longtitude, latitude,*localLongitude*,*localLatitude*);

Longdis=(longtitude-*localLongitude*)\*111000;

Latdis=Math.*sqrt*(distance\*distance-Longdis\*Longdis);

**if**(latitude<*localLatitude*){

Latdis=-Latdis;

}

**int**[] fouce={(**new** Double(Longdis)).intValue(),(**new** Double(Latdis)).intValue()};

jsonArray\_local.add(fouce);

}

//Double longitude=entry.getValue()[0];

//Double latitude=entry.getValue()[1];

}

jsonArray.add(jsonArray\_mmsi);

jsonArray.add(jsonArray\_local);

jsonArray.add(jsonArray\_length);

// }

// }

HttpServletResponse response = ServletActionContext.*getResponse*();

        response.setContentType("text/html");

        response.setCharacterEncoding("UTF-8");

response.getWriter().print(jsonArray.toString());

**return** **null**;

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

//e.printStackTrace();

flag="无运算结果可以查询";

JSONObject JSONflag = **new** JSONObject()

.element( "flag",flag);

HttpServletResponse response = ServletActionContext.*getResponse*();

        response.setContentType("text/html");

        response.setCharacterEncoding("UTF-8");

**return** **null**;

}

}

对于解析请求的参数不做赘述，主要把海量的船只数据进行了统计和分类，在这里每一条数据包括船只编号，船只的经纬度，船的长度等，船只的数据需要在前台中按照每一条船只的轨迹进行显示，而一条船只某段时间的一串数据并不是连续的，这时我们需要根据编号对船只的数据进行分类，我们通过TreeMap先把相同的船只的所有数据放到一个数据结构里，每个数据结构包含相同船只的所有数据，然后再按照顺序添加到对应的数组中，这样穿到前台时，遍历数据显示时就可以根据当前这条数据和下一条数据的Mmsi是否相同来决定是否显示新的船只分类，这样在前端可以把相同的船只通过一个图形标示进行显示。

(2)GasAction

这个类用来处理所有的污染源数据，这一模块最主要就是对污染物CO2和SO2的显示和处理以及计算，对于list实现如下：

**public** String list() {

String flag = **null**;

Date dateBeg = **null**;

Date dateEnd = **null**;

Long time = Long.*parseLong*(checktime);

dateEnd = **new** Date(time);

dateBeg = **new** Date(time - 36 \* 60000);

**try** {

JSONArray jsonArray = **new** JSONArray();

JSONArray jsonArray\_time = **new** JSONArray();

JSONArray jsonArray\_SO2 = **new** JSONArray();

JSONArray jsonArray\_CO2 = **new** JSONArray();

SimpleDateFormat formatter = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

Date start\_request = **null**, stop\_request = **null**;

**if** (!(start\_researchs.length() == 1 || stop\_researchs.length() == 1)) {

start\_request = formatter.parse(start\_researchs);

stop\_request = formatter.parse(stop\_researchs);

} **else** {

start\_request = dateBeg;

stop\_request = dateEnd;

}

**boolean** time\_toolong=**false**;

**if** (stop\_request.getTime() - start\_request.getTime() > 60000 \* 60 \* 24||

stop\_request.getTime() - start\_request.getTime()<0

) {

start\_request = dateBeg;

stop\_request = dateEnd;

time\_toolong = **true**;

}

GasQuery query = newQuery(GasQuery.**class**, "ptime asc");

query.setPtimeBegin(start\_request);

query.setPtimeEnd(stop\_request);

List<Gas> list = gasManager.findPage(query).getResult();

Iterator itc = list.iterator();

**while** (itc.hasNext()) {

Gas gas = (Gas) itc.next();

**if** (gas.getName().equals("SO2")) {

jsonArray\_time.add(formatter.format(gas.getPtime()));

Double value = gas.getValue();

**if** (value > 0) {

jsonArray\_SO2.add(value);

} **else** {

jsonArray\_SO2.add(0);

}

} **else** {

//jsonArray\_time.add(formatter.format(gas.getPtime()));

Double value = gas.getValue();

**if** (value > 0) {

jsonArray\_CO2.add(value);

} **else** {

jsonArray\_CO2.add(0);

}

}

}

jsonArray.add(jsonArray\_time);

jsonArray.add(jsonArray\_SO2);

jsonArray.add(jsonArray\_CO2);

jsonArray.add(time\_toolong);

HttpServletResponse response = ServletActionContext.*getResponse*();

response.setContentType("text/html");

response.setCharacterEncoding("UTF-8");

response.getWriter().print(jsonArray.toString());

**return** **null**;

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

// e.printStackTrace();

flag = "无运算结果可以查询";

JSONObject JSONflag = **new** JSONObject().element("flag", flag);

HttpServletResponse response = ServletActionContext.*getResponse*();

response.setContentType("text/html");

response.setCharacterEncoding("UTF-8");

**return** **null**;

}

}

数据筛选条件有一定的逻辑判断，默认时当前时间到前36分钟的污染物数据，如果前端传过来了相应的时间区间，我们可以根据时间条件进行筛选,时间点和不同的污染物分别用json数据存储。

(2)ComputeSo2

这个方法针对某一个波峰或者曲线的背景数据计算污染物的平均浓度，对于客户来说，这些浓度平均值再结合排放标准可以判断此刻船只的排放是否符合国家的排放标准，代码实现如下：

// 计算硫硫含量；

**public** String ComputeSo2() **throws** IOException, ParseException {

SimpleDateFormat formatter = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

Date startmills = formatter.parse(starttime);

Date stopmills = formatter.parse(stoptime);

System.*out*.println(starttime);

HttpServletResponse response = ServletActionContext.*getResponse*();

response.setContentType("text/html");

response.setCharacterEncoding("UTF-8");

JSONObject JSONresult;

**double** so2sum = 0, co2sum = 0;

GasQuery queryco2 = newQuery(GasQuery.**class**, "ptime asc");

queryco2.setPtimeBegin(startmills);

queryco2.setPtimeEnd(stopmills);

List<Gas> listco2 = gasManager.findPage(queryco2).getResult();

**int** sizec = 0;

**int** sizes = 0;

Iterator itco2 = listco2.iterator();

**while** (itco2.hasNext()) {

Gas gas = (Gas) itco2.next();

**if** (gas.getName().equals("CO2")) {

co2sum += gas.getValue();sizec++;

} **else** {

so2sum += gas.getValue();sizes++;

}

}

**if**(num.equals("1")) {

// 是背景

JSONresult = **new** JSONObject().element("co2", co2sum/sizec).element("so2",

so2sum/sizes).element("status", 0);

} **else** {

// 是波峰

JSONresult = **new** JSONObject().element("co2", co2sum/sizec).element(

"so2", so2sum/sizes).element("status", 1);

}

response.getWriter().print(JSONresult.toString());

**return** **null**;

}

(3)底层Hbase调用接口

对于每项操作由操作表的Db类和被操作的Table类组成，Db是表的管理类，实现对表的增删改查，Table类关联每个具体的表信息。

**（a）Db类**public Connection conn;  
  
private static Db2 db = new Db2();  
public static Db2 getInstance() {  
    return db;  
}  
public Db2() {  
    try {  
        conn = ConnectionFactory.createConnection(MetaConfig.hbaseConfig);  
    } catch (IOException e) {  
        throw new RuntimeException(e);  
    }  
}

public void insertBucketAcl(DbBucketAcl bucketAcl) throws IOException, JSONException {  
    try (Table table = conn.getTable(TableName.valueOf(DbBucketAcl.getTableName()))) {  
        bucketAcl.insert(table);  
    }  
}  
  
public boolean selectBucketAcl(DbBucketAcl ucketAcl) throws IOException, JSONException {  
    try (Table table = conn.getTable(TableName.valueOf(DbBucketAcl.getTableName()))) {  
        return ucketAcl.select(table);  
    }  
}  
  
public void updateBucketAcl(DbBucketAcl bucketAcl) throws IOException, JSONException {  
    try (Table table = conn.getTable(TableName.valueOf(DbBucketAcl.getTableName()))) {  
        bucketAcl.update(table);  
    }  
}  
  
public void deleteBucketAcl(DbBucketAcl bucketAcl) throws IOException, JSONException {  
    try (Table table = conn.getTable(TableName.valueOf(DbBucketAcl.getTableName()))) {  
        bucketAcl.delete(table);  
    }  
}

上面贴出了具体的简历操作链接和具体操作类型的部分实现代码。

（b）DbAksk类

此类实现JdbcRow接口，这个接口中有和数据库字段信息进行转换的方法和具体表操作方法，实现如下：

private static final byte[] FAMILY = Bytes.toBytes("fy");  
    private static final byte[] COL = Bytes.toBytes("v1");  
//    private static final byte[] COL2 = Bytes.toBytes("v2");  
    public static final int ORDINARY = 0;  
    public static final int PRIMARY = 1;  
    public static final int INTERNAL = 2; //内部使用，用户不可删除  
    public static final int STATUS\_INACTIVE = 0;  
    public static final int STATUS\_ACTIVE = 1;  
    public String accessKey = "";  
    public String secretKey = "";  
    public int status;  
    public int isPrimary;  
    public long ownerId;

这些地段是表操作所需要的必要信息，包括了表所属的列和列簇信息，所属用户，ak和sk验证信息，这些都是对应Hbase中的键值信息的。

public String toStringJson() throws JSONException{  
    JSONObject jo = new JSONObject();  
    jo.put("ak", accessKey);  
    jo.put("sk", secretKey);  
    jo.put("st", status);  
    jo.put("pri", isPrimary);  
    jo.put("ownerId", ownerId);  
    return jo.toString();  
}  
  
@Override  
public void deserialize(byte[] value, int offset, int length) throws JSONException {  
    JSONObject jo = new JSONObject(new String(value, offset, length, Consts.CS\_UTF8));  
    accessKey = jo.getString("ak");  
    secretKey = jo.getString("sk");  
    status = jo.getInt("st");  
    isPrimary = jo.getInt("pri");  
    ownerId = jo.getLong("ownerId");  
}

@Override  
public void insert(Table table) throws JSONException, IOException {  
    Put put = new Put(getKey());  
    byte[] value = getValue();  
    put.addColumn(FAMILY, COL, value);  
  
    Put put2 = new Put(getKey4List());  
    put2.addColumn(FAMILY, COL, value);  
  
    List<Put> puts = new ArrayList<>();  
    puts.add(put);  
    puts.add(put2);  
    table.put(puts);  
}

554效果演示

第6章 部署与实施

（1）污染数据平台的部署

这是一个java-web项目，我们的部署系统是windows，数据库存储把IP参数修改成目标机器的IP即可，然后在本地安装java环境，使用jdk1.8.0，配置JAVA\_HOME=安装的路径，PATH=%JAVA\_HOME%\bin;%JAVA\_HOME%\jre\bin;CLASSPAT=.;%JAVA\_HOME%\lib\dt.jar;%JAVA\_HOME%\lib\tools.jar;配知道这些参数后，安装相应的tomcat，在webapp下拷贝进去我们的web项目，在bin下启动tomcat即可在浏览器中访问数据平台服务了。

（2）COS存储服务的部署

我们的平台部署在linux系统中，我们使用的是centos6.5版本，发起进程并不像windows下运行IDE自动起jvm就可以，首先需要把相关的应用程序打成jar包，打包时注意指定整个应用程序的主函数，设置好依赖的一系列jar的名称，在linux中安装java环境jdk1.8.0，然后把相关的jar包和主函数包都拷贝到服务器上，此时我们通过java -cp .;c:\classes\myClass.jar;d:\classes\\*.jar packname.mainclassname指令即可起云存储的主服务，主进程部署完成。

COS服务底层依赖的是HDFS，也就是我们还需要部署集群，首先需要若干台位于同一局域网的机器，这些机器都是centos6.5系统，机器都部署了Hbase1.1.3，集群的监控通过zookeeper来管理，对于设备的增添，更换，zookeeper提供了很好的包装，我们只需要调用相关的接口，实现设备数据的拷贝，然后更换相应的设备即可让集群实现升级。

第7章 总结

7.1 总结

／／遇到的问题，开发的难度

7.2 展望

附录A

HBASE数据库

参考文献

致谢