# 空间管理设计文档

# 需求分析

## 1.1 项目概述

对于大型的图片分享网站，或者其它的资源分享平台，都离不开对资源文件的管理，最简单的操作包括添加、删除、修改和查询。虽然各个平台的前端展示方式有所不同，但后台的管理机制大同小异。从文件存储的角度来分，一种传统的方式是将资源的信息存储关系型数据库系统（RDBMS）中，而把资源文件存储在文件系统中；另一种逐渐流行的方式是利用各种不同数据库的优势，在RDBMS同样存储资源的信息，而将文件存储在类似于MongoDB的NoSQL数据库中。

在本项目中，主要采用前一种经典的存储方式，对图片（Picture）、文档（Post）或其它资源（Resource）进行统一的管理，借助操作系统本身的文件系统来存储资源文件，用关系型数据库来保存资源的相关信息。本项目旨在编写一套完善的存储机制，为上层应用模块提供通用的、跨平台的API，通过简单的配置，便能使得这些应用不用再重复实现类似的资源管理功能。

## 1.2 功能分析

对于一个比较完善的资源管理来说，在功能上可以大体分为三大模块，如图1所示：

空间管理

资源管理模块

文件同步模块

系统监控模块

图1 空间管理的功能模块

* 资源管理模块

核心模块，主要提供对资源的增删改查功能，实现一套合理的存储机制，并对系统资源进行控制，比如磁盘大小、块大小（Block）、文件大小等。

* 文件同步模块

对于一般的主从服务器，网站每天的数据都保存于主服务器的数据库和文件系统中，而备份服务器则定时与主服务器的数据进行同步。该模块实现一套文件同步与数据库同步机制，通过简单的配置，便能定期同步服务器上的数据。

* 系统监控模块

针对系统在运行中或文件同步中遇到的错误或中断，系统应能自动发现并采取相应的补救措施。

## 1.3 难点分析

### 1.3.1 接口设计

该项目旨在为上层应用模块提供简单易用的API，封装底层空间管理的细节，因此该项目的难点之一是设计合理的公共API接口。由于目前采用的存储机制是RDBMS与文件系统结合的方式，如果以后需要换成RDBMS+NoSQL的话，必须在接口保持不变的情况下，改变具体实现的细节，从而不影响上层应用调用的兼容性。同时，为了使接口能够适应各种类型的系统，接口本身必须具备良好的扩张性。毕竟不能在一次设计中满足所有系统的需求，所以应当为开发人员提供方便的扩展接口，使开发人员能够迅速将该项目整合至其自己的项目。同时要考虑到二次开发的复杂性，如果过于复杂，则失去了该项目的意义，即开发人员不如自行开发一套全新的存储系统。

### 1.3.2 文件同步算法

由于本项目适用于各种规模的上层应用，所以往往免不了会涉及大量文件的存储，当文件数量规模达到一定程度的时候，主从服务器间的同步机制会愈加重要。因此，不能在每次同步时将所有文件重新更新（除非在真的必要时），因为这样会大量增加网络带宽和系统资源的消耗（硬盘读写、内存使用率等）。合理的做法应当是只同步更新的或不同的部分，而且应当在适当时间（夜晚使用率较低的时候）进行同步。

### 1.3.3 文件存储

本项目直接利用操作系统本身的文件系统保存文件，所以在系统运行时必定会涉及大量的读写操作（I/O），如何有效降低I/O是一个值得研究的方向。相比使用NoSQL数据库存储文件，在文件系统上存储文件的好处是便于管理员直接查看资源的内容（比如直接点击打开图片），而不是通过额外的操作访问数据库。而且在迁移和同步过程也相对容易，也给管理员提供更加直观的方式进行查看。但是，弊端也十分明显，即资源文件检索的速率，此时需要一种合理高效的方式来读写文件，比如建立索引等。

### 1.3.4 异常处理

系统在运行过程中难免会发生异常，比如网络中断，容量不足等。这时需要让系统自行发现问题所在，在能够自动解决的前提下，修复异常带来的损耗；如果涉及人为因素等，则必须记录下相应的异常，待管理员手工解决。而这里的难点是即使系统发生异常，也应当尽量保持系统能够继续正常使用，否则必定会导致不可估计的损失。

# 系统架构

## 开发环境

|  |  |
| --- | --- |
| 操作系统 | Windows，Linux，OS X |
| 开发语言 | Java（JDK 1.7+） |
| IDE | Eclipse |
| 数据库 | MySQL 5.6 |

## 2.2系统架构

该项目的系统架构共分为三层，如图2所示：

**Public API Interface**

**Core Manager**

**Database Manager**

**File System Manager**

DB

DB

DB

**File System**

**Infrastructure Layer**

**Software Layer**

**Interface Layer**

图2 系统总体架构图

**Logging System**

**System Monitor**

### 2.2.1 Interface Layer

该层用于为外界应用提供调用接口，同时封装下层所有的细节实现。从功能的角度来看，对外暴露的接口主要为资源的读写和系统配置，对于系统异常处理的结果均保存于日志系统。当然，对于日志系统的读写暂时还未提供外部API，因为管理员可以直接于文件系统下查看日志，如果以后需要，可以直接在该层中添加对日志的管理。

### 2.2.2 Software Layer

该层为整个项目的核心部分，主要由5个子模块组成：

* Core Manager

该子模块充当协调者的角色，一方面整合其它四个子模块的功能，调用相应的API以实现对应的功能，比如资源的读取，需要同时调用Database Manager和File System Manager中的API，并加上资源读取的业务逻辑，从而实现完整的调用流程。另一方面，该子模块为仅有的与Interface Layer进行交互的部分，从而最大限度剥离了模块间的耦合度。

* Database Manager

该子模块提供了程序与数据库间交互的功能，封装了不同数据库间的不同，提供统一的访问接口。目前的版本中，该模块仅提供与MySQL的交互，同时利用JDBC的API重新实现了数据库读写操作，包括将SQL语句隐藏在该层中，以及一个简单易用的数据库连接池。

* File System Manager

该子模块的功能与Database Manager类似，即封装不同文件系统的读写操作。由于Java本身便提供了很好的跨平台API，因此该子模块在很大程度上依赖于Java的I/O接口。

* System Monitor

该子模块用于监控程序运行时的异常错误，且不是单独独立存在，而是嵌入在各个模块中。当其它子模块在运行时抛出异常时，往往调用该子模块提供的API，即相应的补救措施。在目前版本中，该子模块将主要的错误信息存储在日志系统中，同时也需要调用Logging System提供的API。

* Logging System

该子模块用于一套完整的日志读写功能，主要为System Monitor提供相应的API。在目前版本中，还未提供一个GUI的界面供管理员查看日志文件，但管理员可以直接在操作系统的文件系统中直接检查。

### 2.2.3 Infrastructure Layer

该层为基础设施的部署和设置，主要涉及异构数据库和不同操作系统的文件系统的管理。对于数据库部分，目前使用的是比较流行的MySQL，虽然会部署在不同的操作系统上，但其都为Java提供相应的驱动程序，所以在跨平台上的运行没有问题。针对其他待添加的数据库，同样需要考虑到是否能够兼容各大平台。对于文件系统部分，Java虽然提供了通用的I/O接口，但是不同操作系统的文件系统目录结构是不一样的，对于Windows和Linux就需要考虑到不同的文件格式、目录路径等。

# 资源管理模块

## 3.1 资源类型

资源主要为图片（Picture）和文章（Post），其中图片的格式没有限制，当系统接收到上层应用传入的图片文件时（也可以让该模块去读取存在临时目录中的图片文件），直接将图片文件复制到相应的存储目录，而不会读取图片本身包含的信息，该部分信息应该由上层应用自行保存。换句话说，该系统只保存图片文件的最基本信息，比如图片名、创建的时间、保存的地址等。对于文章，一般为纯文本文件，比如可以是由富文本编辑器生成的HTML文档，其中链接的图片可以当作Picture存储，而文本则作为Post存储。而如果是由Word或其它客户端生成的doc或pdf文档时，应当把该文件当作其它资源处理（因为程序不可能一一将所有格式的文档做处理）。剩下的资源则都归类为其它类型，其存储方式和前两者类似，只是包含的信息更为简单（因为要涉及不同格式的资源，所以只保存共有的属性）。

## 3.2 数据库设计

### 3.2.1 Resource Table

资源表用于存储所有资源，包括图片，文档和其它。Type属性标识了不同的资源类型，status属性标识了资源的状态。

|  |  |  |
| --- | --- | --- |
| 列名 | 类型 | 说明 |
| id | INTEGER | PK，资源的唯一标示符 |
| block\_id | INTEGER | 指向资源所在的Block |
| name | VARCHAR(100) | 资源原始名称 |
| format | VARCHAR(20) | 资源的格式 |
| type | VARCHAR(20) | 资源的类型：picture, post, other |
| status | VARCHAR(20) | 资源的状态：normal, deleted |
| size | INTEGER | 资源大小 |
| created | DATETIME | 创建时间 |
| modified | DATETIME | 修改时间 |
| description | VARCHAR(1000) | 资源描述 |

### 3.2.2 Block Table

Block表用于存储所有block，block的id无论资源类型一律自增。Vdisk\_id指向VDisk，该信息由配置文件提供，不用数据库存储。

|  |  |  |
| --- | --- | --- |
| 列名 | 类型 | 说明 |
| id | INTEGER | PK，Block的唯一标示符 |
| vdisk\_id | VARCHAR(100) | Block所在的VDisk |
| type | INTEGER | Block存储的资源类型 |
| status | VARCHAR(20) | Block状态：full, use |
| left\_space | VARCHAR(20) | Block剩余的空间 |
| created | DATETIME | 创建时间 |
| modified | DATETIME | 修改时间 |

## 3.3 功能设计

### 3.3.1 资源存储结构

为了使文件存储和读写更加高效，本系统建立了一套自定义的文件存储结构，基于不同的操作系统，统一了文件所在的目录结构。如图3所示，最外层的Server为实体的服务器或是专业的存储设备，一般为装有操作系统的硬件。里面一层的VDisk为虚拟磁盘，实际则为文件系统下用于存放资源的根目录，同一个Server上可以有多个VDisk，且不同的VDisk路径可能会依赖于操作系统本身的目录结构，比如Unix为树状结构，而Windows则为分区。不过，只要VDisk的根路径确定下来之后，便不存在异构文件系统不兼容的问题（除个别文件系统格式的兼容性问题，如NTFS和EXT3等）。在VDisk中存放的是资源类型目录，也即为图片目录pictures，文档目录posts和其它目录，这样分类的目的是为了让管理员能够直观地浏览不同类型的资源。对于每种类型目录中，是以Block为单位的块目录结构，每个Block中存放固定数量上限的资源文件，这是考虑到当一个目录下存放过多文件时，会严重拖慢文件搜索的速度，因此每个Block下的文件数目是有上限的，而不关心Block已经占用的空间。

因此，定位每个资源文件的话，需要知道VDisk、Resource Type、Block和文件本身，那么一个完整的路径是可能是：[vdisk\_root\_path]/[resource\_type]/[block\_id]/[resource\_id]。至于定位到文件是存在哪个Server上时，可以通过VDisk数据结构来获得，因为VDisk和Server的对应关系是多对一的，也即一个Server上可以有多个VDisk目录，但一个VDisk目录必定存在一个Server上，而不可能分布在多个Server上。所以一旦知道了VDisk，也就知道了其所在的Server。

Server

VDisk

Resource Type

图3 文件存储结构层次

Block

Resource File

### 3.3.2 资源文件操作

* 读文件

只要知道资源在数据库中唯一标识的ID就能找到文件所在的位置。因此完整的过程是先根据ID从数据库中获取资源的信息（主要是VDisk ID和Block ID），然后将信息拼接成正确的URL，从而读取资源文件。如果没有在数据库中或者文件系统中找到指定资源，则仅返回空值。

* 写文件

写文件操作需要考虑到多方面因素。第一是保存的信息为数据库和文件系统两部分，数据库中为基本信息，文件系统中保存实际的资源文件。但是，在保存进数据库之前，系统会根据目前使用情况来分配可用的VDisk和Block。对于VDisk来说，只要当前服务器的硬盘还有存储空间，或者未达到VDisk预先设定的最大阈值，那么VDisk总是指向当前正在使用的那个。而对于Block来说，需要考虑到当前Block是否已经达到最大容量，如果还有容量，则直接分配当前Block，如果没有，则需要建立新的Block，而当其所在的VDisk容量达到最大时，则又需要通知系统建立新的VDisk和对应的Block，然后才将文件执行保存操作。

注意的是，每次存储新的资源文件时，不会存在以前旧的Block中，即便这些旧的Block中仍然有空间。

* 更新文件

为了降低系统的复杂度，规定文件一旦保存后就不能移动，因此文件的更新一般为更新数据库中的内容，且不包括VDisk和Block的值。但对于文档文件来说，更新文档的内容有时是十分必要的，因此此时的做法是删除原来的文档文件，再按照写文件的方式重新将新的文档文件保存至原来的位置，最后修改数据库中记录的modified属性。

* 删除文件

文件删除分为临时删除和永久性删除。对于临时删除，只是将数据库中对应记录的status属性置为删除状态。而永久性删除，则是同时删除数据库记录和文件系统中的实体文件。在删除文件过程中，如果发现其所在的Block下已经没有任何文件（Empty Block），并且该Block不是最新使用之时，则直接将该Block删除，因为以后也不会再用到了。

### 3.3.3 目录结构的调整

目前资源文件的存储结构为上述讨论的，如果用户还想对当前目录层次进行增加时，则可以在类型目录下继续添加，比如按照地域分类，此时的目录结构为VDisk 🡪 Type 🡪 Region 🡪 Block 🡪 File，同时修改相应的配置即可。

### 3.3.4 系统配置

上层应用的开发人员可以通过动态配置来适应各种不同的需求，配置文件形式为XML，主要配置的内容有：

* 不同的数据库

即配置不同数据库的数据源，同时需要载入相应的JDBC驱动。

* VDisk目录

VDisk为根目录路径，所在服务器的IP、端口、访问方式（FTP，Samba）、必要时提供用户名和密码等。

* 主从服务器的设置

这里的服务器概念就是VDisk，而实际的硬件服务器已经被抽象成底层的基础设施，位于Infrastructure Layer，而这里的主从服务器位于Software Layer。

# 文件同步模块

## 4.1 增量同步算法

对于增量同步算法，本系统借鉴了Linux下的一个同步文件的高效算法rsync，该算法也同样应用与许多网盘的更新中。Rsync能同步更新两处计算机的文件与目录，并适当利用查找文件中的不同块以减少数据传输。其中一项与其他大部分类似程序或协定中所未见的重要特性是镜像是只对有变更的部分进行传送。Rsync可拷贝或显示目录属性，以及拷贝文件，并可选择性的压缩以及递归拷贝。Rsync的具体做法可以参考<http://en.wikipedia.org/wiki/Rsync>，下面简单说明一下大体的思想：

* 分块Checksum算法

把目标文件（fileDst）平均切分成若干个小块，比如每块512个字节（最后一块会小于这个数），然后对每块计算两个校验值（checksum），一个叫滚动校验（[rolling checksum](http://en.wikipedia.org/wiki/Rolling_hash)），是弱checksum，32位的checksum，其使用的是Mark Adler发明的[adler-32](http://en.wikipedia.org/wiki/Adler-32" \o "Adler-32)算法；另一个是强checksum，128位的，以前用md4，现在用md5 hash算法。因为以前的硬件上跑md4算法太慢了，所以，需要一个快算法来鉴别文件块的不同，但是弱的adler32算法碰撞概率太高了，所以还要引入强的checksum算法以保证两文件块是相同的。换句话说，弱的checksum是用来区别不同，而强的是用来确认相同。

* 传输算法

同步目标端会把fileDst的一个checksum列表传给同步源，这个列表里包括了三个东西，rolling checksum(32bits)，md5 checksume(128bits)，文件块编号。同步源机器拿到了这个列表后，会对源文件（fileSrc）做同样的checksum，然后和fileDst的checksum做对比，这样就知道哪些文件块改变了。

* checksum查找算法

同步源端拿到fileDst的checksum数组后，会把这个数据存到一个hash table中，用rolling checksum做hash，以便获得O(1)时间复杂度的查找性能。这个hash table是16bits的，所以，hash table的尺寸是2的16次方，对rolling checksum的hash会被散列到0 到 2^16 – 1中的某个整数值。

* 比对算法

这是最关键的算法，细节如下：

（1）取fileSrc的第一个文件块（假设是512个长度），也就是从fileSrc的第1个字节到第512个字节，取出来后做rolling checksum计算。计算好的值到hash表中查。

（2）如果查到了，说明发现在fileDst中有潜在相同的文件块，于是就再比较md5的checksum，因为rolling checksume太弱了，可能发生碰撞。于是还要算md5的128bits的checksum，这样一来，就有 2^-(32+128) = 2^-160的概率发生碰撞，这太小了可以忽略。如果rolling checksum和md5 checksum都相同，这说明在fileDst中有相同的块，需要记下这一块在fileDst下的文件编号。

（3）如果fileSrc的rolling checksum 没有在hash table中找到，那就不用算md5 checksum了。表示这一块中有不同的信息。总之，只要rolling checksum 或 md5 checksum 其中有一个在fileDst的checksum hash表中找不到匹配项，那么就会触发算法对fileSrc的rolling动作。于是，算法会住后step 1个字节，取fileSrc中字节2-513的文件块要做checksum，go to 1) - 现在你明白什么叫rolling checksum了吧。

（4）这样，我们就可以找出fileSrc相邻两次匹配中的那些文本字符，这些就是我们要往同步目标端传的文件内容了。

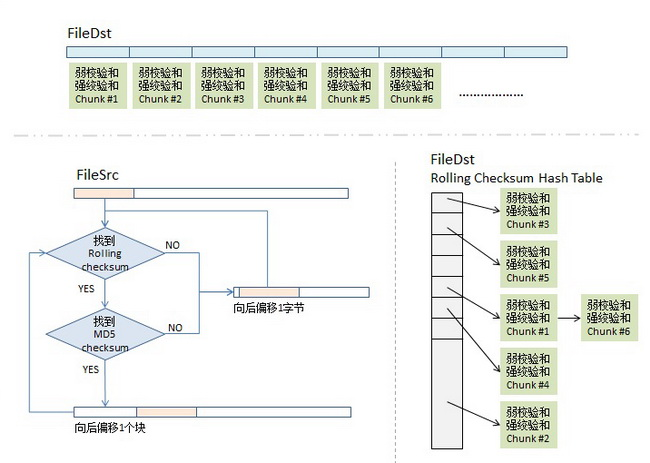


图4 rsync算法

如图4所示，在同步源这端，rsync算法可能会得到下面这个样子的一个数据数组，图中，红色块表示在目标端已匹配上，不用传输，而白色的地方就是需要传输的内容（这些白色的块是不定长的），这样，同步源这端把这个数组（白色的就是实际内容，红色的就放一个标号）压缩传到目的端，在目的端的rsync会根据这个表重新生成文件，这样，同步完成。

## 4.2 文件同步机制

虽然使用增量同步算法可以大大降低网络带宽的占用率，但是在实际使用中还是会消耗不少系统资源，因此该同步过程往往是在晚上用户使用率较低的情况下来运行。同时，对于不同的资源文件来说，同步的内容也是不一样的，对于图片或者其它资源来说，由于是并非可读的文本，所以主服务器和备份服务器（这里应该叫Master VDisk和Backup VDisk更加确切，因为同一个Server上可以同时存放两种VDisk，虽然实际应用价值并不高）之间的区别就在于单个文件的差异，即最小粒度是文件本身。然而，对于文档文件来说，主从服务器上会存在同名的文档，但是文本内容确有所不同，所以此时增量同步算法便有用武之地了，即只更新修改的那部分。当然，还有一种情况是无论什么类型的文件，更新时改变了整个文件（比如替换了一张图片，但其它信息不变），此时也只能同步完整的文件。

# 系统监控模块

## 5.1 文件存储异常处理

* 资源文件不存在

如果给定的资源ID没有存在数据库中，或者数据库中存在记录，而文件系统中没有实体文件时，统一返回给用户空值。同时，针对后一种情况，系统直接将记录下当前的操作，然后从备份服务器中寻找相应的文件是否存在，如果两者中都未能找到文件，则删除数据库中的记录，否则将备份服务器中的数据同步至主服务器中。（后一种发生的可能是有人直接访问文件系统中的目录，将资源文件删除，因此服务器的安全性需要系统管理员考虑）

* 文件无法写入磁盘

此时发生的情况可能是磁盘损坏或者文件被锁，这种条件下，系统应当事先指定一个可随意读写的临时目录，先将文件写入该目录，随后再尝试写入正常指定的目录。如果发生特殊情况时，应当记录在日志文件中，待管理员来处理，并且返回上层应用文件保存失败的通知。

* 磁盘已满

如果待写入的磁盘已满，系统自动分配其它可用服务器的磁盘（包括备份服务器），如果所有配置过的磁盘都没有空间，则在日志中记录下该错误，然后待系统管理员来解决。

## 5.2 网络通信异常处理

* 同步传输中断

如果同步过程中由于网络等外部因素而终止，该过程理论上并未受到影响。因为采用的是增量同步算法，所以只需要在网络情况恢复后，继续同步即可。

* 文件传输中断

如果在单个或个别文件传输过程中出现中断，系统则重复向目标服务器请求重新传输，如果这些请求都失败的话，则说明需要人为力量来解决，因此先将该错误记录与日志系统中，之后由管理员手工解决文件的复制。

# 技术优化方案

## 6.1 NoSQL数据库

通过简单的实验可以发现，使用MongoDB之类的NoSQL数据库与文件系统相比，在I/O速率上有很好的提升，但是瓶颈在CPU的使用上，因此随着系统规模的增加，如果能够合理利用这些数据库的话，何尝不是一种很好的优化方法呢。以下是摘自现有的一个实验结果，可供参考：

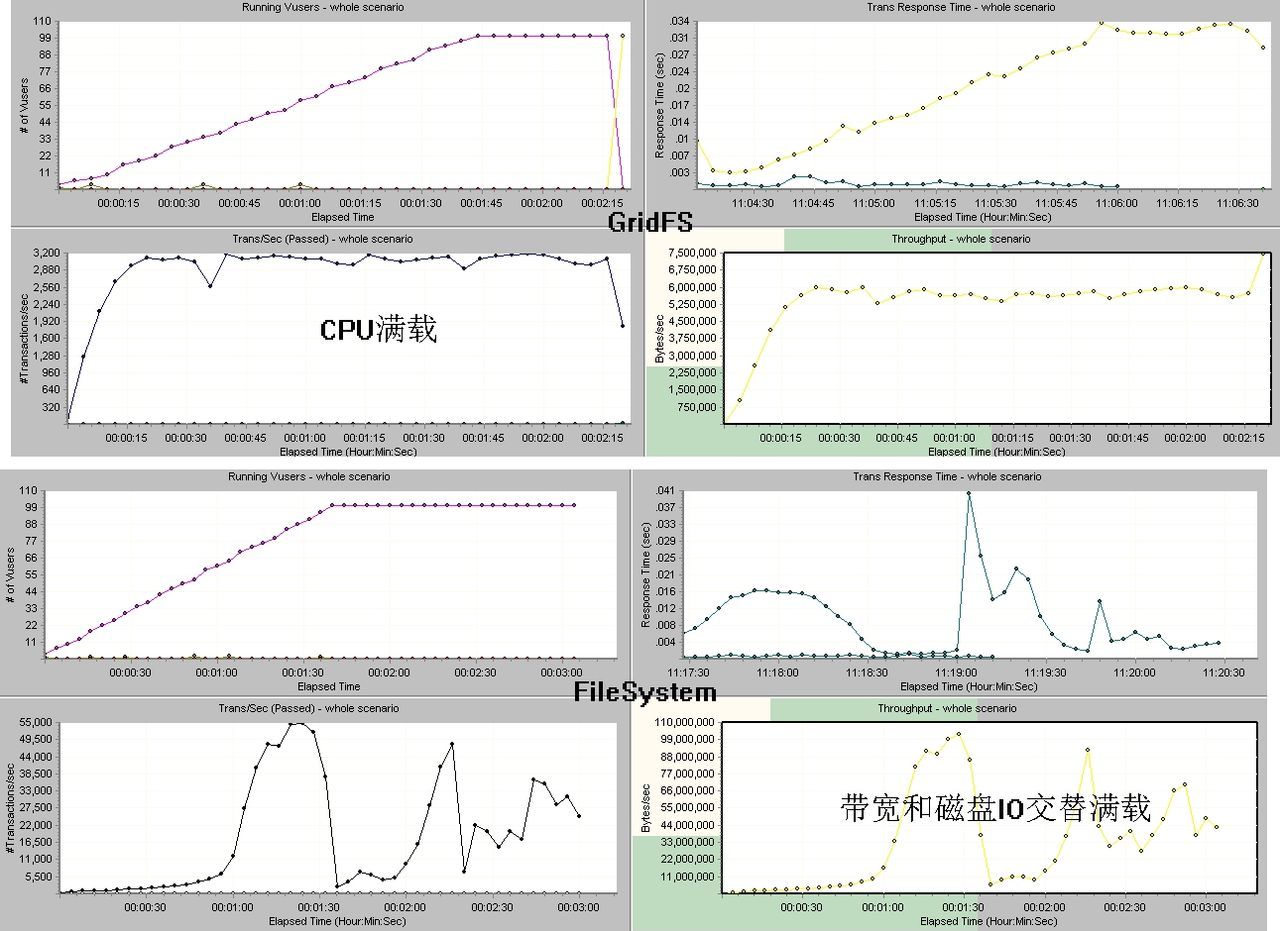


图5 Gridfs和文件系统的性能对比

如图5所示，  该测试场景为：向文件系统和Gridfs各存入100万张约1.2KB的图片，随机访问前3万张图片。结果表明，MongoDB的瓶颈在CPU，而文件系统的瓶颈交替出现在带宽和磁盘IO上。由于被查询的数据少，MongoDB完全将其缓存到内存里，完全没有磁盘I/O上的消耗。从现象看，文件系统的缓存不间断的被置入和移出内存，导致性能不稳定。命中缓存则带宽瓶颈，否则磁盘IO跑满。此场景下，文件系统性能明显优于MongoDB。

实验结果来自<http://hi.baidu.com/higkoo/item/02d33cc790832960f7c95de1>。

## 6.2 Memcached缓存

当下，NoSQL数据库由于在K/V存储的优异性能表现，广为流行，然而他们或多或少都有自己的某些缺陷，比如存在单点、数据安全持久化等等。然而这些随着新的技术和思路的在MySQL上面产品化，这些东西会被慢慢取代，MySQL重回王者地位。一种比较流行的架构方式是MySQL+InnoDB with Memcached，如图6所示：

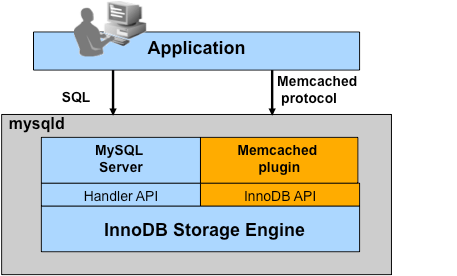


图6 MySQL+Memcached架构

这样做的好处是可以充分利用现有的memcached client，减少应用程序的变动，解决了memcached不具备的数据存储持久性问题。同时提供了memcached不具备的crash-safe和ACID特性、复查查询的支持、丰富的管理和展示工具，解决了平时单独使用memcached时，cache和数据库数据一致性的问题，提供更高的性能。并且还能减少数据冗余，充分利用好内存，降低硬件投入成本，在提供高效的k/v访问的同时，还提供复查的查询功能，比如排序、join等。

## 6.3 GUI

目前的日志系统以及系统配置部分仅仅提供了API接口，没有提供一个带有图形化界面的工具，主要考虑到的是为了与上层应用有更好的集成（如果提供了Web风格的界面或者Java Swing的界面会显得很冗余）。但是换个角度会发现，如果单独将该项目作为一个可执行程序，则可以提供一个带有GUI和Lightweight Web Server的独立程序，从而管理员可以直接通过GUI工具进行配置和日志管理。

# 源代码分析

## 包

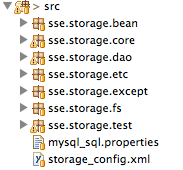


图7 项目包结构

项目完整包结构如图7所示。Package bean为Java Bean，用于与数据库交互的bean，或者配置文件中生成的实体。Package core中为核心接口，外部调用一般只用该package中的类。Package dao为与访问数据库的Data Access Object。Package etc为环境配置变量，或者易用的工具接口。Package except为异常处理的类。Package fs为文件系统的读/写接口，包括本地的文件读写，以及用SMB协议访问的接口。

## 7.2 sse.storage.etc.Config

该类从配置文件读取配置信息，包括cluster, vdisk, database等。一套完整的存储系统包括一个master cluster和至少一个backup cluster；一个cluster（无论master还是backup）是由一个集中式的database和至少一个vdisk构成；每个vdisk中存有一些block，每个block中才是资源文件；在同一个cluster下的所有block都有着唯一标识ID，这些block都存储在对应的database中。所以，对应的配置文件内容可能如图8所示。

Config类中有三个变量：clusters, vdisks和databases，可以知道分别存储配置文件中读取的信息。如果一个cluster是master类型，则旗下的database和vdisk都为master，同样对于backup类型的cluster。该类中还提供了一些get接口，供外部程序使用。



图8 storage\_config.xml部分内容

## 7.3 sse.storage.dao.BaseDao

该类为DAO的抽象类，用于提供数据库的IO操作。对于数据库持久层和ORM，本项目使用了Yank库，一个十分lightweight的框架。BaseDao提供了增删改查（insert, delete, update, read/find/findAll）等接口，子类仅需继承该类，设置一些变量，就能使用这些方法。

具体的实现利用了Java的反射机制，详情请直接阅读source code。

## 7.4 sse.storage.bean.VDisk

该类为较为复杂的Java Bean，保存VDisk的相关信息，以及提供一些文件系统的IO操作，作用类似于Java中的File类。同时，完全封装了本地或者SMB访问的操作。

公共接口中，isValid方法用于检查VDisk的root path字符串是否合法；isdir方法用于判断给定路径是否为目录；ls方法列出给定路径下的所有文件或目录名；lsdirs方法列出给定路径下的所有目录名；mkdirs方法用于在给定路径下完整建立目录；rm方法用于删除给定路径的文件或目录。

## 7.5 sse.storage.fs.Writer

该类以及Reader类都用于文件系统上的读写，封装了本地和SMB远程的实现，为外部程序提供了方便的接口，而不用关心到底是什么服务器。

## 7.6 sse.storage.core.BlockManager

每个BlockManager类用于管理一个cluster下的同种类型（picture，post，other）的所有block，所以clusterId和ResourceType组合可以唯一确定一个BlockManager。换句话说，BlockManager和Cluster是多对一的关系，一个BlockManager也唯一对应一个Database（但反过来不是）。

在该类初始化的时候，首先需要从对应的database中载入所有对应的block，然后在这些block中间找到状态是use的block，即表示该block为当前使用（一个block只有两种状态，use和full，并且状态只会从use变为full，反过来则不行）。最后，如果发现use状态的block，则需要检查该block的情况是否相符（DB中的状态和文件系统中的状态一致性），调用checkCurrBlock方法。该方法首先检查block是否在文件系统中存在，然后检查block实际是否已满（文件数量到达最大），最后核对剩余文件数量是否一致。

在该类中，还提供了一些方便的公共接口，mkurl方法用于构造在当前cluster中给定Resource的完整路径；allocateBlock方法用于分配block中的一个资源空间，该方法会更新数据库中block的信息；getAvailBlock方法返回当前正在使用的block。

## 7.6 sse.storage.core.ClusterManager

该类为单例模式，用于管理所有配置过的cluster以及包含的BlockManager。主要提供一些get接口，包括给定clusterId和ResourceType来返回对应的BlockManager等。

## 7.7 sse.storage.core.ResourceManager

该类为用户可以直接调用的接口，每个ResourceManager对应一个cluster，所以在初始化时需要将clusterId传入。该类提供了资源读写的接口，且读写操作都是在给定的cluster中进行的。接口使用十分简单，这里不再赘述。

## 7.8 sse.storage.core.SyncManager

该类用于两个cluster之间的同步，唯一的接口为sync方法。该算法的实现没有利用rsync算法，而是一种比较简单的实现。首先从两个database的block表中获取不同的block记录（不同的意思是id和modified时间不同，其它不考虑），从而得到一个待更新的block list。然后对这个block list进行数据库上的同步。接着，用同样的方法获取一个待更新的resource list，最后对这个resource list进行同步，包括数据库和文件同步。这里的文件同步因为考虑到本地和远程之间的通信，直接使用了Wrtier和Reader类里的读写方法，在性能上可能会有影响，一个可以改进的传输方式是用socket来交换数据。