FastDFS分布式文件系统

# 课程内容大纲

1. 认识海量文件存储、架构、性能瓶颈
2. 能够理解互联网环境中文件服务器选型
3. 能够理解FastDFS分布式文件系统架构、及架构思想
4. 能够理解文件上传、下载、 文件服务器访问的流程
5. 能够完成FastDFS环境搭建（使用docker镜像即可）
6. 能够理解FastDFS&nginx访问流程
7. 能够掌握FastDFS错误日志查看方法
8. 能够理解FastDFS文件同步原理
9. 能够掌握FastDFS文件合并存储机制
10. 能够掌握FastDFS图片压缩机制
11. 能够理解FastDFS快速定位文件机制

后期课程：

1、es

2、RocketMQ

3、授权中心 （非对称加密（公钥私钥），JWT）

4、分布式事务

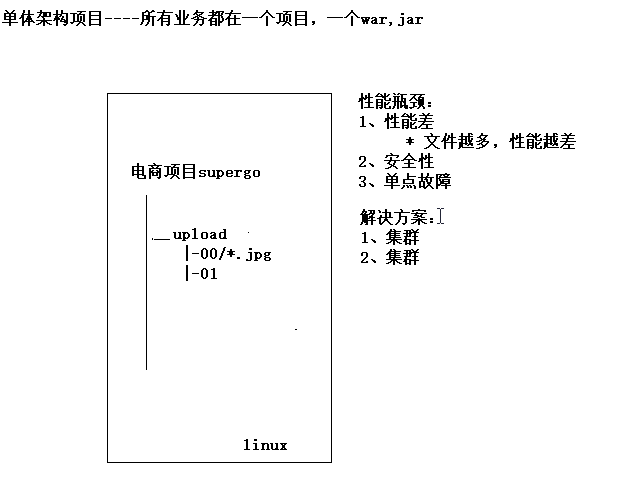
5、分布式锁（控制秒杀库存）

6、搜索

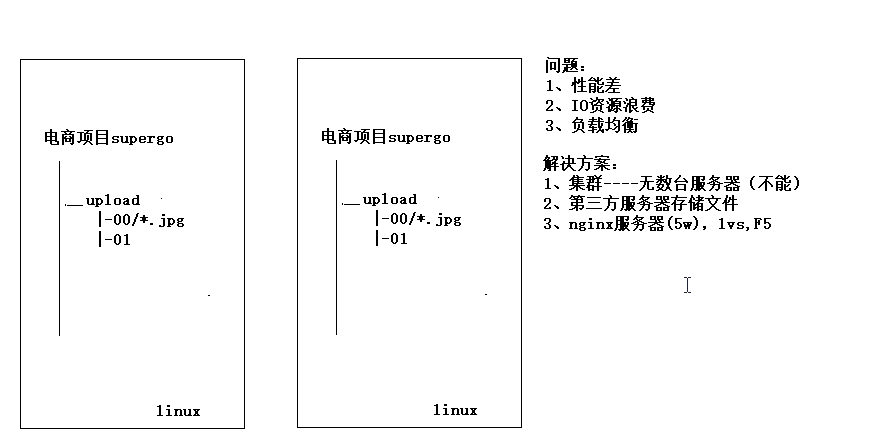
7、项目架构优化

# 海量文件存储性能瓶颈

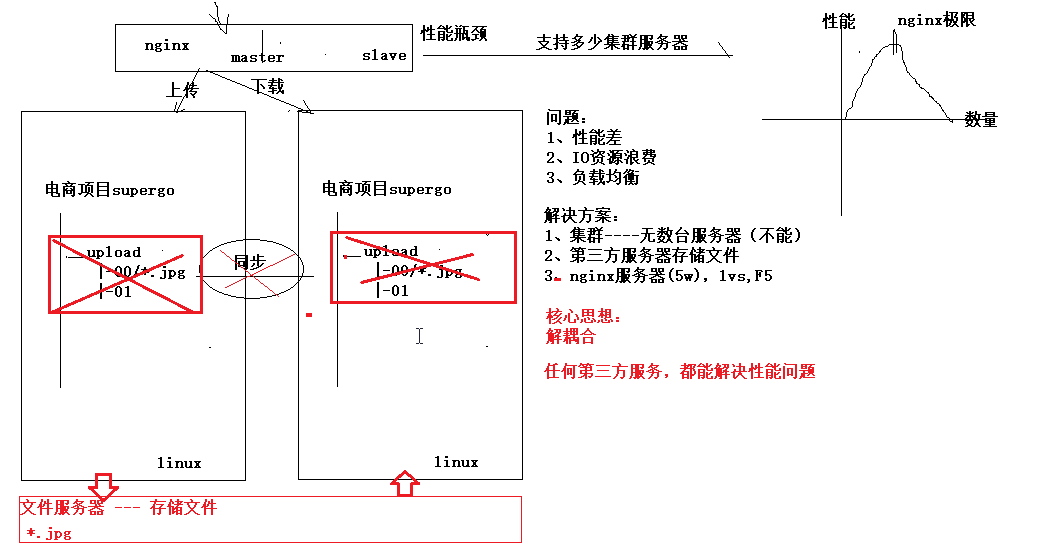
单点架构：



集群架构：



负载均衡、文件服务器：



项目只负责业务的处理，文件服务器只负责文件存储（下载，删除，访问）

# 文件服务器选型

## 问题

1）性能

2）功能支持程度 （上传，下载，删除，水印，压缩,访问..）

3）开发语言

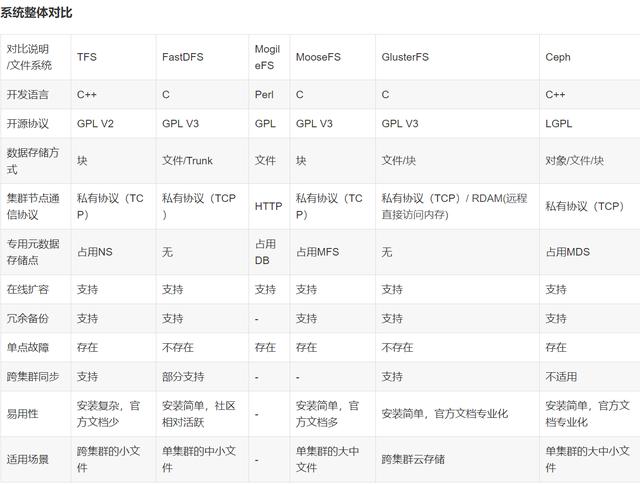
4）有多少公司在使用

5）社区是否活跃

6）稳定性，是否支持集群

7）是否有完善文档

## 文件服务对比



GFS google 文件系统

小型企业：

云存储服务器 --- 500G --- 1000

中大型企业：

FastDFS 分布式文件系统

# FastDFS文件系统

## 简介

FastDFS 特点：

1、c语言开发

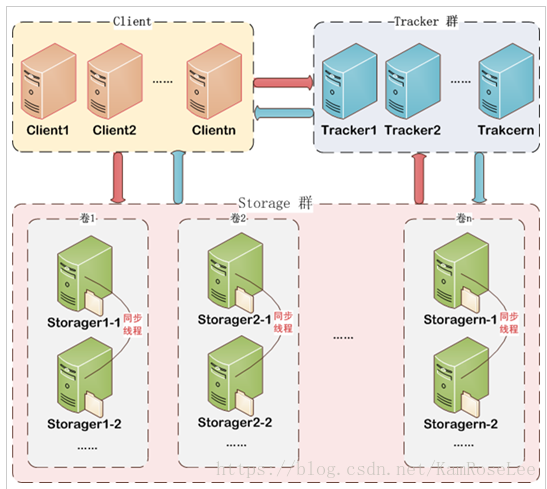
2、专门为互联网量身定制 （存储中小型文件） ---- IDC --- 200亿

3、在线扩容、冗余备份、负载均衡

4、文件存储、文件同步、文件访问，上传，下载

中小型文件：1M以下是小文件， 1M 500M—中文件

## FastDFS架构



Client(客户端：application应用程序)

Tracker: 调度服务器，负载均衡服务器

Storage: 存储文件

Tracker服务器：

1、tracker集群服务器是相互独立的服务器

2、storage 集群服务器定时向tracker汇报自身状态信息（30s）

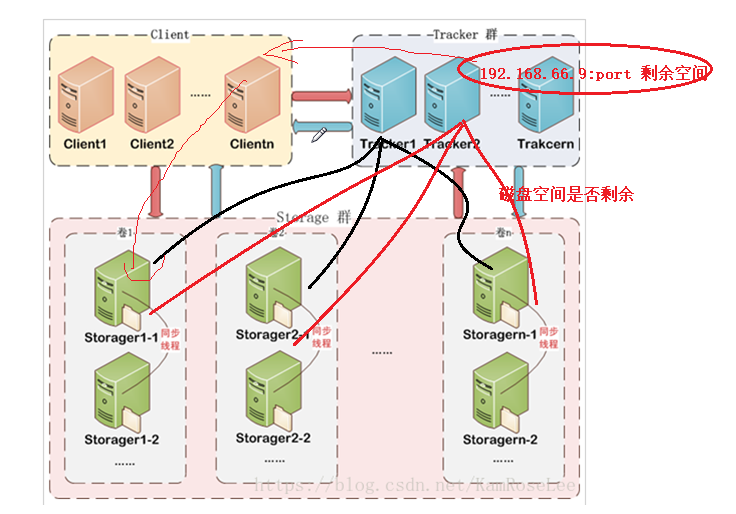
健康信息

同步信息

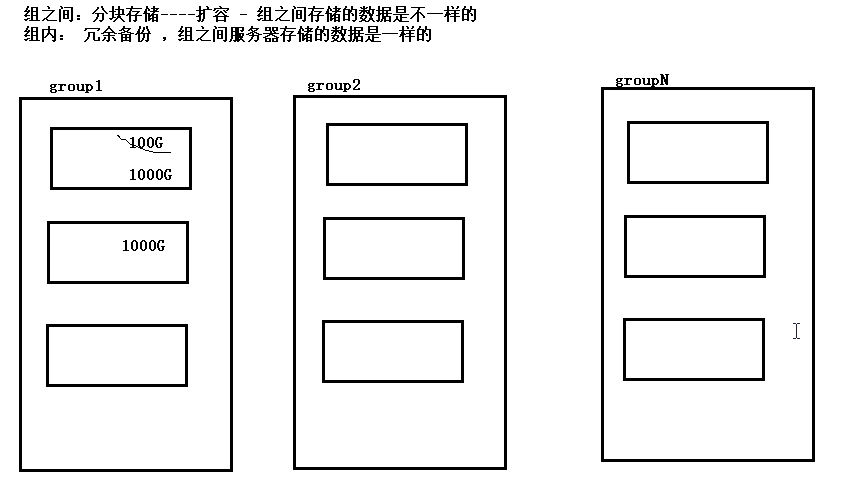
磁盘是否还有空闲空间

IP

3、负载均衡：



Storage存储服务：



同组策略：

1、同组服务器数据是一样的

2、同组服务器数据要相互备份，服务器容量大小必须一致

3、同组之间服务器同步是在后台完成

4、同组集群：冗余备份，负载均衡

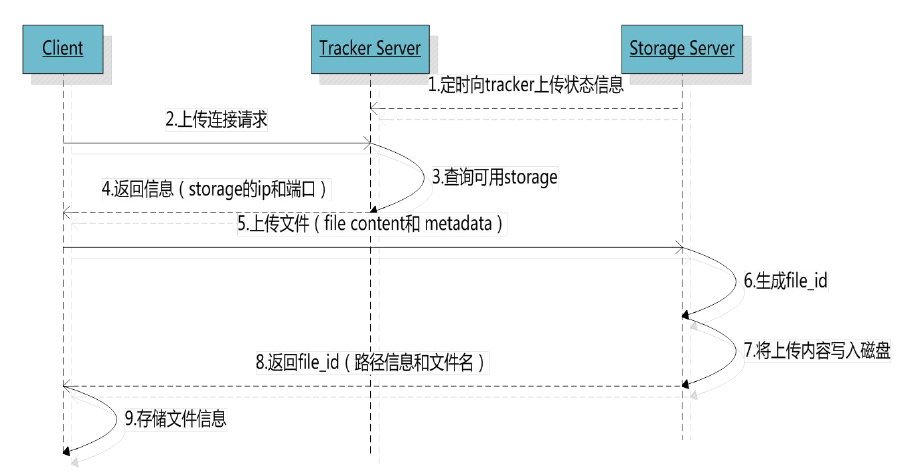
不同组策略：

1、组合组之间的文件是不相同的

2、整个组之间的容量集合就是整个文件系统的文件

3、分组就是文件系统的扩容

## 上传文件



互联网文件访问：



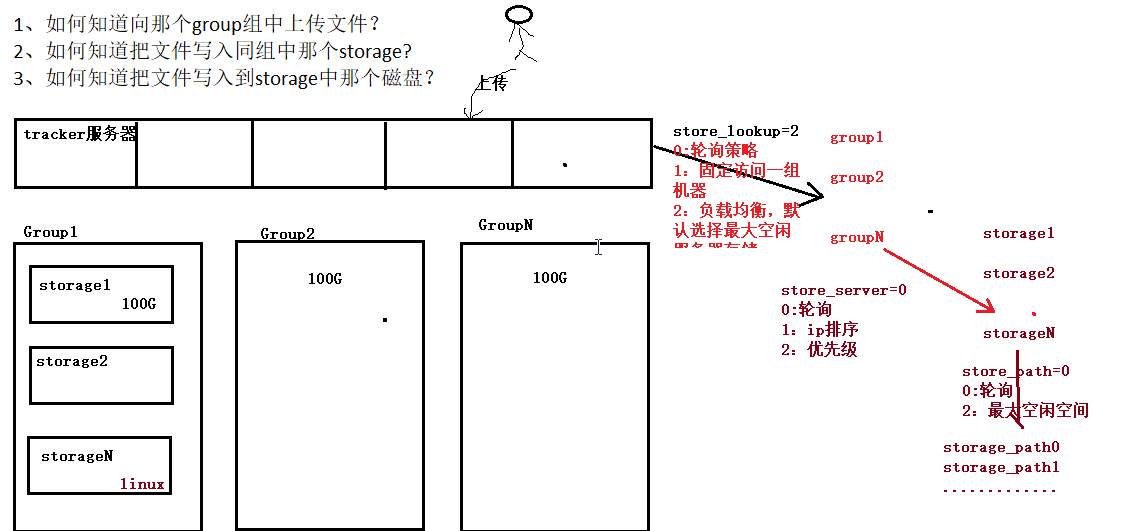
文件上传细节：

问题：

1、如何知道向那个group组中上传文件？

2、如何知道把文件写入同组中那个storage?

3、如何知道把文件写入到storage中那个磁盘？



## 文件名

当文件存储到某个子目录后，即认为该文件存储成功，接下来会为该文件生成一个文件名，文件名由group、存储目录、

两级子目录、文件名、文件后缀名（由客户端指定，主要用于区分文件类型）拼接而成



M00: 虚拟磁盘路径

思考：文件存储一定是存储磁盘上的，-- 磁盘目录—磁盘路径

store\_path0=/kkb/fastdfs/storage ============== M00

store\_path1=/home/yuqing/fastdfs2 ============ M01

目录：16\*16\*16\*16 = 2的16次方

文件名：



1）源服务器IP

2）文件创建时间

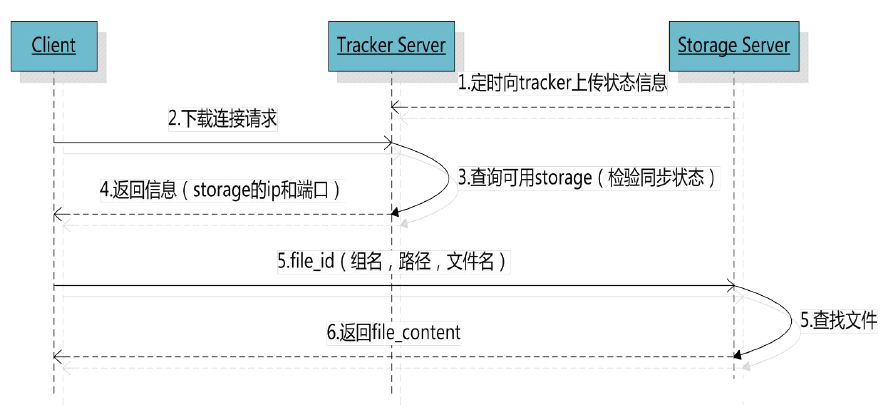
3）文件大小

4）文件crc32

5）随机数

把以上数据Base64进行编码措施。

## 下载文件



## 同步状态

* FDFS\_STORAGE\_STATUS\_INIT 初始化，未得到同步源服务器
* FDFS\_STORAGE\_STATUS\_WAIT\_SYNC 等待同步
* FDFS\_STORAGE\_STATUS\_SYNCING 同步中
* FDFS\_STORAGE\_STATUS\_DELETED 删除
* FDFS\_STORAGE\_STATUS\_OFFLINE 离线
* FDFS\_STORAGE\_STATUS\_ONLINE 在线，尚不能提供服务
* FDFS\_STORAGE\_STATUS\_ACTIVE 在线，可以提供服务

# Docker基本命令

1）下载docker引擎:

yum install docker

2）查询docker

docker -v #查询docker是否下载成功

3）基本命令

Systemctl start docker # 启动docker

Systemctl stop docker # 停止docker

Systemctl restart docker

4）下载镜像

docker pull centos:7

5）查询镜像

docker images

6）创建容器

docker run -di --name=java5-tracker-01 centos:7 /bin/bash

7）查询容器

docker ps

8）登录容器

docker exec -it java5-tracker-01 /bin/bash

9）提交镜像

docker commit demo-tracker-one one-me

10）镜像备份

docker save -o xxx.tar one-me

11）镜像恢复

docker load -i xxx.tar

# FastDFS文件系统搭建

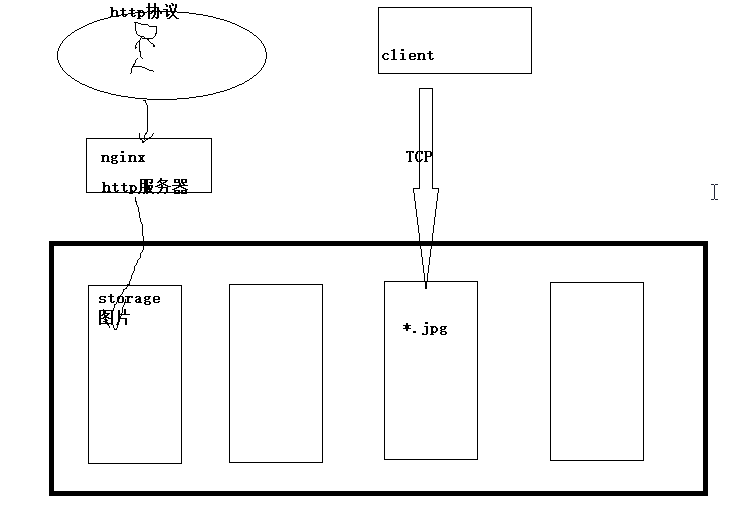
安装FastDFS，参考文档

# 问题解答

问题

（1）、上传文件的时候是nginx直接连接到文件服务器吗？

不是



（2）、client直接连接strorager的场景是什么？

（3）、group1中各节点存储的内容都一样吗？

（4）、每组一般几个节点，2个够吗？

2台服务器已经达到最基本高可用场景

并发量更大，可以适当的添加服务器

（5）、如果文件比较大，同组之间数据同步会不会很慢？

肯定会慢，无法避免

（6）、客户端怎么知道写到哪组，访问哪组？

（7）、会不会存在 上传到节点1，下载时访问节点2？

解析文件名（源ip）转发

（8）、负载写到A组，读的时候怎么知道在A组呢？

（9）、会不会有是一个文件太大，在同步的时间其他服务空间不够的问题呢？

（10）、ip转发是strorager自己完成的吗？

（11）、文件的同步机制和上传机制肯定是会有冲突的啊, 这个是怎么解决的？

（12）、最大空闲机器不是某台机器么，怎么和组关联的？

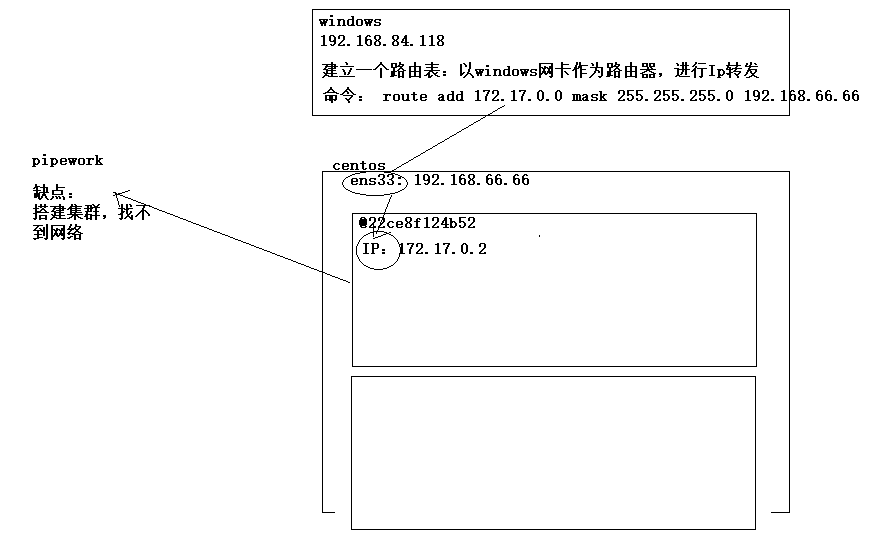
（13）、docker引擎（yum install docker--- docker hub）和镜像(fastdfs.tar)老师能给一个吗？

（14）、docker引擎需要安装吗？

# FastDFS&nginx

参考安装文档，但是必须注意：此时需要把命令放入docker容器进行执行。

# 网络问题



添加路由表：

Route add 172.17.0（跟据网段）.0(必须是0) mask 255.255.255.0 转发ip

注意: 此命令必须在管理员权限下进行设计

# FastDFS集群

集群：集群就是把单机版拷贝几份，构建一个集群网络

1）提交为一个镜像

docker commit java5-tracker-01 java5-fastdfs

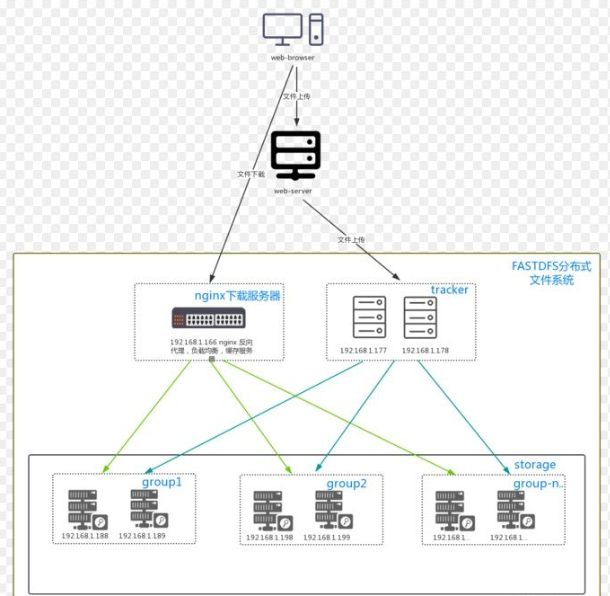
2）镜像备份

docker save -o java5-fastdfs xxxx.tar

3）导入镜像

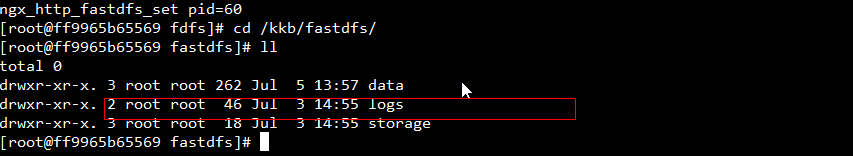
docker load -i xxxx.tar

集群架构：



# 日志查看

Fastdfs日志文件存储在 基础路径中： base\_path

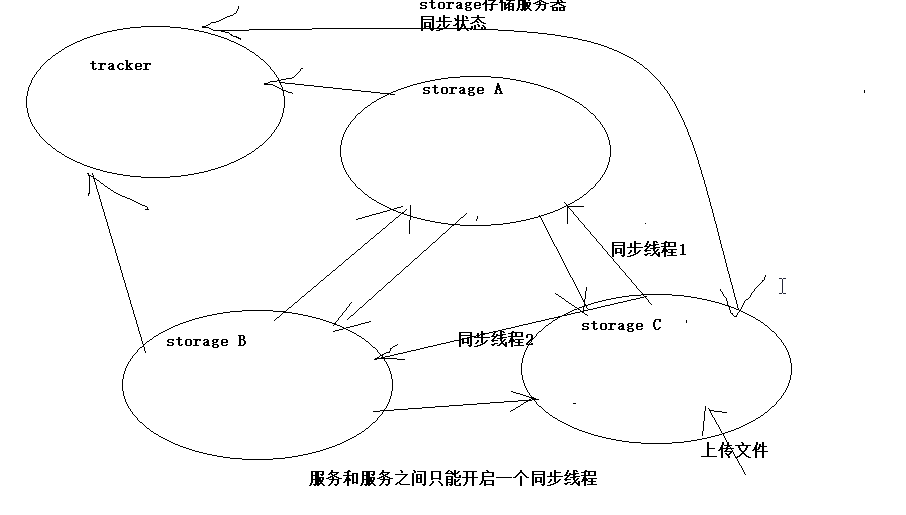


# 文件同步

## 何时开启同步线程

从Tracker Server获取到同组storage server集合后，就会为除了自己之外的其他同组storage server分别开启同步线程。

如果组内有三个Storage Server，那么每台Storage Server都应该开启两个独立同步线程



## 同步规则

1. 只在本组内的storage server之间进行同步；

2. 源头数据才需要同步，备份数据不需要再次同步，否则就构成环路了；

3. 同步采取push方式，即源服务器同步给同组其他服务器；

4. 上述第二条规则有个例外，就是新增加一台storage server时，由已有的一台storage server将已有的所有数据（包括源头数据和备份数据）同步给该新增服务器

## 同步流程

### Binlog 目录结构

当Storaged server启动时会创建一个 **base\_path/data/sync** 同步目录，该目录中的文件都是和同组内的其它 Storaged server之间的同步状态文件，如：

 192.168.1.2\_33450.mark   
 192.168.1.3\_33450.mark   
 binlog.000   
 binlog.index

**文件说明：**

binlog.index 记录当前使用的binlog文件序号，如为10，则表示使用binlog.010

binlog.100 真实地binlog文件

192.168.1.2\_33450.mark 同步状态文件，记录本机到192.168.1.2\_33450的同步状态

### Mark 文件

在Mark文件中内容：

 binlog\_index=0  
 binlog\_offset=1334  
 need\_sync\_old=1  
 sync\_old\_done=1  
 until\_timestamp=1457542256  
 scan\_row\_count=23  
 sync\_row\_count=11

**文件说明：**

binlog\_index：对应于哪个binlog

binlog\_offset：binlog.xxx的偏移量，可以直接这个偏移量获取下一行记录

need\_sync\_old：本storage是否是对侧storage(10.100.66.82)的源结点，同时是否需要从起点同步所有的记录

sync\_old\_done：是否同步完成过

until\_timestamp：上次同步时间结点

scan\_row\_count：总记录数

sync\_row\_count：已同步记录数

### Binlog 文件

Binlog文件内容：在该文件中是以binlog日志组成，比如：

 1470292943 c M00/03/61/QkIPAFdQCL-AQb\_4AAIAi4iqLzk223.jpg  
 1470292948 C M00/03/63/QkIPAFdWPUCAfiraAAG93gO\_2Ew311.png  
 1470292954 d M00/03/62/QkIPAFdWOyeAO3eUAABvALuMG64183.jpg  
 1470292959 C M00/01/23/QUIPAFdVQZ2AL\_o-AAAMRBAMk3s679.jpg  
 1470292964 c M00/03/62/QkIPAFdVOsCAcxeQAAGTdbQsdVs062.jpg  
 1470292969 c M00/03/62/QkIPAFdVOnKAXu1NAABq9pkfsms63.jpeg  
 1470293326 D M00/03/62/QkIPAFdVMnGAZYSZAABq9pkfsms33.jpeg

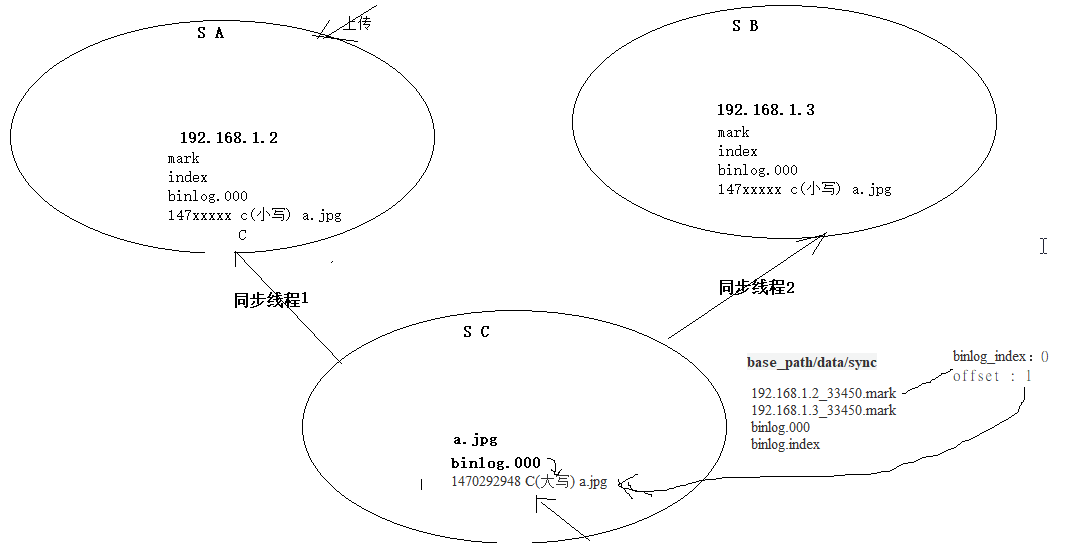
**文件说明：**

其中的每一条记录都是使用[空格符]分成三个字段，分别为：

* 第一个字段 表示文件 upload 时间戳 如：1470292943
* 第二个字段 表示文件执行操作，值为下面几种（其中源表示客户端直接操作的那个 Storage 即为源，其他的 Storage 都为副本）

  C 表示源创建、c 表示副本创建  
 ​  
  A 表示源追加、a 表示副本追加  
 ​  
  D 表示源删除、d 表示副本删除  
 ​  
  T 表示源 Truncate、t 表示副本 Truncate

* 第三个字段 表示文件，如 M00/03/61/QkIPAFdQCL-AQb\_4AAIAi4iqLzk223.jpg



## 同步过程

从fastdfs文件同步原理中我们知道Storaged server之间的同步都是由一个独立线程负责的，这个线程中的所有操作都是以同步方式执行的。比如一组服务器有A、B、C三台机器，那么在每台机器上都有两个线程负责同步，如A机器，线程1负责同步数据到B，线程2负责同步数据到C。每个同步线程负责到一台Storage的同步，以阻塞方式进行。

**1）获取组内的其他 Storage 信息，并启动同步线程**

在Storage.conf配置文件中，只配置了Tracker的IP地址，并没有配置组内其他的Storage。因此同组的其他Storage必须从Tracker获取。具体过程如下：

 1）Storage启动时为每一个配置的Tracker启动一个线程负责与该Tracker的通讯。  
 2）默认每间隔30秒，与Tracker发送一次心跳包，在心跳包的回复中，将会有该组内的其他Storage信息。  
 3）Storage获取到同组的其他Storage信息之后，为组内的每个其他Storage开启一个线程负责同步。

**2）同步线程执行过程**

每个同步线程负责到一台Storage的同步，以阻塞方式进行。

 1）打开对应Storage的mark文件，如负责到10.0.1.1的同步则打开10.0.1.1\_23000.mark文件，从中读取binlog\_index、binlog\_offset两个字段值，如取到值为：1、100，那么就打开binlog.001文件，seek到100这个位置。

 ​  
 2）进入一个while循环，尝试着读取一行，若读取不到则睡眠等待。若读取到一行，并且该行的操作方式为源操作，如C、A、D、T（大写的都是），则将该行指定的操作同步给对方（非源操作不需要同步），同步成功后更新binlog\_offset标志，该值会定期写入到10.0.1.1\_23000.mark文件之中。

## 文件同步时间戳

举个例子：一组内有Storage-A、Storage-B、Storage-C三台机器。对于A这台机器来说，B与C机器都会同步Binlog（包括文件）给他，A在接受同步时会记录每台机器同步给他的最后时间（也就是Binlog中的第一个字段timpstamp）。比如B最后同步给A的Binlog-timestamp为100，C最后同步给A的Binlog-timestamp为200，那么A机器的最后最早被同步时间就为100。

这个值的意义在于，判断一个文件是否存在某个Storage上。比如这里A机器的最后最早被同步时间为100，那么如果一个文件的创建时间为99，就可以肯定这个文件在A上肯定有。为什呢？一个文件会Upload到组内三台机器的任何一台上：

 1）若这个文件是直接Upload到A上，那么A肯定有。  
 2）若这个文件是Upload到B上，由于B同步给A的最后时间为100，也就是说在100之前的文件都已经同步A了，那么A肯定有。  
 3）同理C也一样。

Storage会定期将每台机器同步给他的最后时间告诉给Tracker，Tracker在客户端要下载一个文件时，需要判断一个Storage是否有该文件，只要解析文件的创建时间，然后与该值作比较，若该值大于创建创建时间，说明该Storage存在这个文件，可以从其下载。

Tracker也会定期将该值写入到一个文件之中，storage\_sync\_timestamp.dat，内容如下：

 group1,10.0.0.1,0,1408524351,1408524352  
 group1,10.0.0.2,1408524353,0,1408524354  
 group1,10.0.0.3,1408524355,1408524356,0

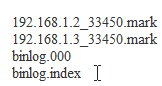
每一行记录了，对应Storage同步给其他Storage的最后时间，如第一行表示10.0.0.1同步给10.0.0.2的最后时间为1408524351，同步给10.0.0.3的最后时间为1408524352；同理可以知道后面两行的含义了。每行中间都有一个0，这个零表示自己同步给自己，没有记录。按照纵列看，取非零最小值就是最后最早同步时间了，如第三纵列为10.0.0.1被同步的时间，其中1408524353就是其最后最早被同步时间

## 问题

问题一：同步到中途，把源文件删除，备份服务器怎么办？

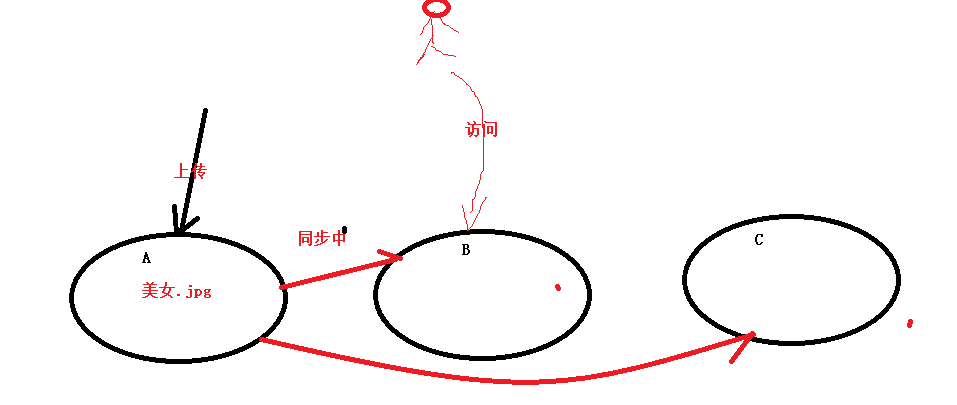
源文件删除，同步就读取不到数据，把中断写入日志，直接同步下一条数据。

问题二：同步到中途，服务器宕机，如何解决？

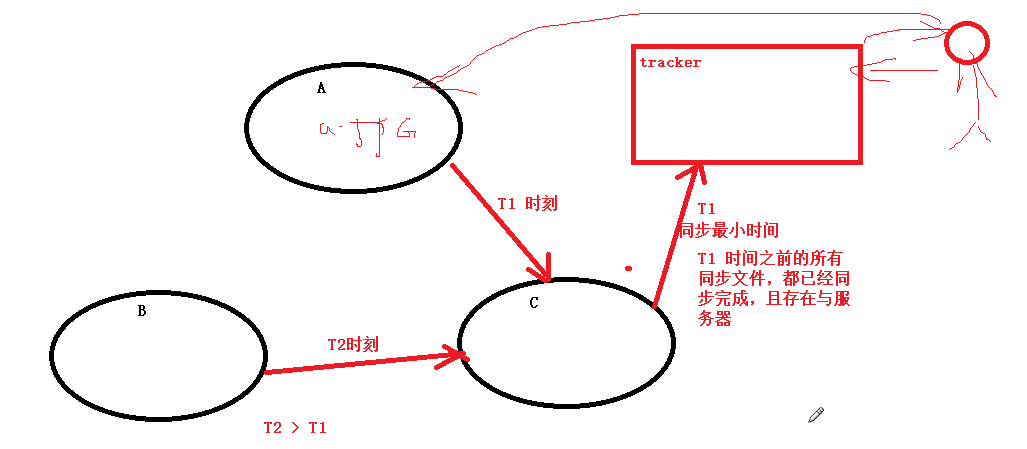


Make,binlog 同步文件，记录同步偏移量，时间戳。 服务器宕机，下次重新启动，重新加载同步文件，发现没有同步完成。

问题三：此时需要下载文件，如何确定从那台storage服务器下载文件呢？ 此时服务器同步未完成？



访问不到文件，如何解决？



# 合并存储

## 为什么要合并存储

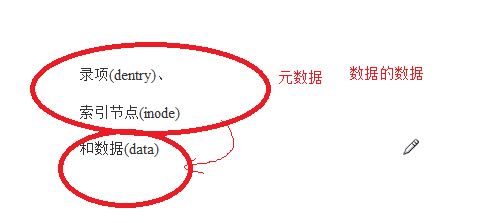
在处理【海量小文件（LOSF）】问题上，文件系统处理性能会受到显著的影响，在读写次数（IOPS）与吞吐量（Throughput）这两个指标上会有不少的下降。

通常我们认为大小在【1MB以内】的文件称为**小文件**，【百万级数量及以上】称为**海量**，由此量化定义海量小文件问题，以下简称**LOSF**（全称lots of small files）

海量文件描述参考自：<http://blog.csdn.net/liuaigui/article/details/9981135>

**海量小文件存储问题：**

* **元数据管理低效**



 由于小文件数据内容较少，因此元数据的访问性能对小文件访问性能影响巨大。当前主流的磁盘文件系统基本都是面向大文件高聚合带宽设计的，而不是小文件的低延迟访问。磁盘文件系统中，目录项(dentry)、索引节点(inode)和数据(data)保存在存储介质的不同位置上。因此，访问一个文件需要经历至少3次独立的访问。这样，并发的小文件访问就转变成了大量的随机访问，而这种访问对于广泛使用的磁盘来说是非常低效的。同时，文件系统通常采用Hash树、 B+树或B\*树来组织和索引目录，这种方法不能在数以亿计的大目录中很好的扩展，海量目录下检索效率会明显下降。正是由于单个目录元数据组织能力的低效，文件系统使用者通常被鼓励把文件分散在多层次的目录中以提高性能。然而，这种方法会进一步加大路径查询的开销

* **数据布局低效**

 磁盘文件系统使用块来组织磁盘数据，并在inode中使用多级指针或hash树来索引文件数据块。数据块通常比较小，一般为1KB、2KB或4KB。当文件需要存储数据时，文件系统根据预定的策略分配数据块，分配策略会综合考虑数据局部性、存储空间利用效率等因素，通常会优先考虑大文件I/O带宽。对于大文件，数据块会尽量进行连续分配，具有比较好的空间局部性。对于小文件，尤其是大文件和小文件混合存储或者经过大量删除和修改后，数据块分配的随机性会进一步加剧，数据块可能零散分布在磁盘上的不同位置，并且会造成大量的磁盘碎片(包括内部碎片和外部碎片)，不仅造成访问性能下降，还导致大量磁盘空间浪费。对于特别小的小文件，比如小于4KB，inode与数据分开存储，这种数据布局也没有充分利用空间局部性，导致随机I/O访问，目前已经有文件系统实现了data in inode。

* **IO访问流程复杂**

 对于小文件的I/O访问过程，读写数据量比较小，这些流程太过复杂，系统调用开销太大，尤其是其中的open()操作占用了大部分的操作时间。当面对海量小文件并发访问，读写之前的准备工作占用了绝大部分系统时间，有效磁盘服务时间非常低，从而导致小I/O性能极度低下。

因此一种【**解决海量小文件**】的途径就是将小文件【**合并存储**】成大文件，使用seek来定位到大文件的指定位置来访问该小文件。

## 合并存储介绍

FastDFS提供了【合并存储】功能，可以将【海量小文件】，合并存储为【大文件】。

默认创建的**大文件为64MB**，然后在**该大文件中存储很多小文件**。大文件中容纳一个小文件的空间称为一个**Slot**，规定**Slot最小值为256字节，最大为16MB**，也就是小于256字节的文件也需要占用256字节，**超过16MB的文件不会合并存储而是创建独立的文件**。

当没有启动合并存储时fileid和磁盘上实际存储的文件一一对应；当采用合并存储时就不再一一对应，而是多个fileid对应的文件被存储成一个大文件。

注：下面将采用合并存储后的大文件统称为trunk文件，没有合并存储的文件统称为源文件；

**请大家注意区分三个概念：**

* **trunk文件**：

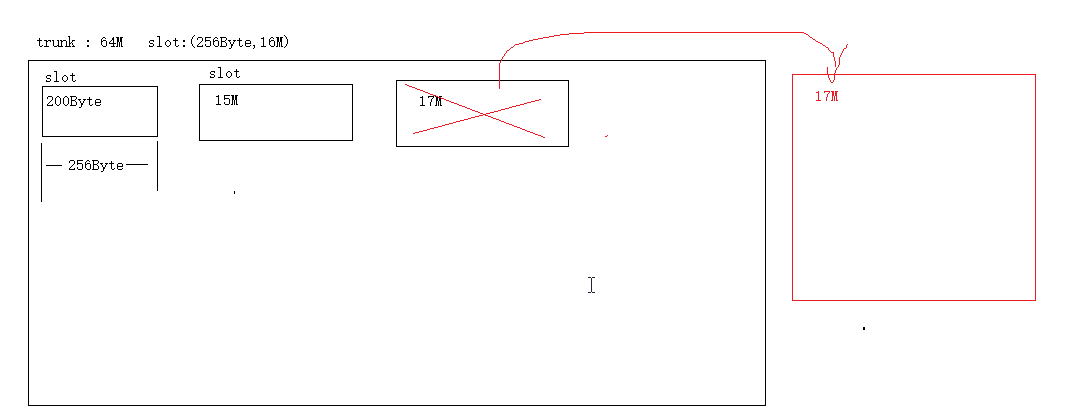
 storage服务器磁盘上存储的合并后的实际文件，默认大小为64MB。  
 Trunk文件文件名格式：fdfs\_storage1/data/00/00/000001 文件名从1开始递增，类型为int；

* **合并存储文件的fileid**：

 表示服务器启用合并存储后，每次上传返回给客户端的fileid，注意此时该fileid与磁盘上的文件没有一一对应关系；

* **没有合并存储的fileid**：

 表示服务器未启用合并存储时，Upload返回的fileid。



## 合并存储前后的fileid介绍

在启动合并存储时服务返回给客户端的fileid也会有所变化，具体如下：

* **合并存储时fileid：**

 group1/M00/00/00/CgAEbFQWWbyIPCu1AAAFr1bq36EAAAAAQAAAAAAAAXH82.conf

可以看出合并存储的fileid更长，因为其中需要加入保存的大文件id以及偏移量，具体包括了如下信息：

**file\_size：**占用大文件的空间（注意按照最小slot-256字节进行对齐）

**mtime：**文件修改时间

**crc32：**文件内容的crc32码

**formatted\_ext\_name：**文件扩展名

**alloc\_size：**文件大小与size相等

**id：**大文件ID如000001

**offset：**文件内容在trunk文件中的偏移量

**size：**文件大小

* **没有合并存储时fileid：**

 group1/M00/00/00/CmQPRlP0T4-AA9\_ECDsoXi21HR0.tar.gz

CmQPRlP0T4-AA9\_ECDsoXi21HR0.tar.gz，这个文件名中，除了.tar.gz 为文件后缀，CmQPRlP0T4-AA9\_ECDsoXi21HR0 这部分是一个base64编码缓冲区，组成如下：

**storage\_id：**ip的数值型

**timestamp：**文件创建时间戳

**file\_size：**若原始值为32位则前面加入一个随机值填充，最终为64位

**crc32：**文件内容的检验码

## Trunk文件内部结构

trunk内部是由多个小文件组成，**每个小文件都会有一个trunkHeader（占用24个字节）**，以及紧跟在其后的真实数据，结构如下：

|||——————————————————— 24bytes———-----———-—————————|||

|—1byte —|— 4bytes —|—4bytes —|—4bytes—|—4bytes—|— 7bytes —|

|—**filetype**—|—**alloc\_size**—|—**filesize**—|—**crc32**—|—**mtime**—|—**formatted\_ext\_name**—|

|||——————file\_data filesize bytes————————————————— —————|||

|———————file\_data———————————————————————— ————|

## 合并存储配置

FastDFS提供了合并存储功能的实现，所有的配置都在**tracker.conf**文件之中，具体摘录如下：

注意：开启合并存储只需要设置**use\_trunk\_file = true**和**store\_server=1**

 # which storage server to upload file  
 # 0: round robin (default)  
 # 1: the first server order by ip address  
 # 2: the first server order by priority (the minimal)  
 # Note: if use\_trunk\_file set to true, must set store\_server to 1 or 2  
 store\_server = 0  
 #是否启用trunk存储  
 use\_trunk\_file = false   
 #trunk文件最小分配单元  
 slot\_min\_size = 256   
 #trunk内部存储的最大文件，超过该值会被独立存储  
 slot\_max\_size = 16MB   
 #trunk文件大小  
 trunk\_file\_size = 64MB  
 #是否预先创建trunk文件  
 trunk\_create\_file\_advance = false  
 #预先创建trunk文件的基准时间  
 trunk\_create\_file\_time\_base = 02:00  
 #预先创建trunk文件的时间间隔  
 trunk\_create\_file\_interval = 86400  
 #trunk创建文件的最大空闲空间  
 trunk\_create\_file\_space\_threshold = 20G  
 #启动时是否检查每个空闲空间列表项已经被使用  
 trunk\_init\_check\_occupying = false  
 #是否纯粹从trunk-binlog重建空闲空间列表  
 trunk\_init\_reload\_from\_binlog = false  
 #对trunk-binlog进行压缩的时间间隔  
 trunk\_compress\_binlog\_min\_interval = 0

## 空闲空间概述

### 为什么产生空余空间

Trunk文件为64MB（默认），因此***每次创建一次Trunk文件总是会产生空余空间***，比如为了存储一个10MB文件，创建一个Trunk文件，那么就会剩下接近54MB的空间（TrunkHeader 会24字节，后面为了方便叙述暂时忽略其所占空间），下次要想再次存储10MB文件时就不需要创建新的文件，存储在已经创建的Trunk文件中即可。

另外***当删除一个存储的文件时，也会产生空余空间***

### 如何管理空余空间

在Storage内部会***为每个store\_path构造一颗以空闲块大小作为关键字的空闲平衡树，相同大小的空闲块保存在链表之中***。

### 如何使用空闲平衡树

每当需要存储一个文件时会***首先到空闲平衡树中查找大于并且最接近的空闲块，然后试着从该空闲块中分割出多余的部分作为一个新的空闲块，加入到空闲平衡树中***。

例如：

要求存储文件为300KB，通过空闲平衡树找到一个350KB的空闲块，那么就会将350KB的空闲块分裂成两块，前面300KB返回用于存储，后面50KB则继续放置到空闲平衡树之中。

假若此时找不到可满足的空闲块，那么就会创建一个新的trunk文件64MB，将其加入到空闲平衡树之中，再次执行上面的查找操作（此时总是能够满足了）

## TrunkServer空闲空间分配

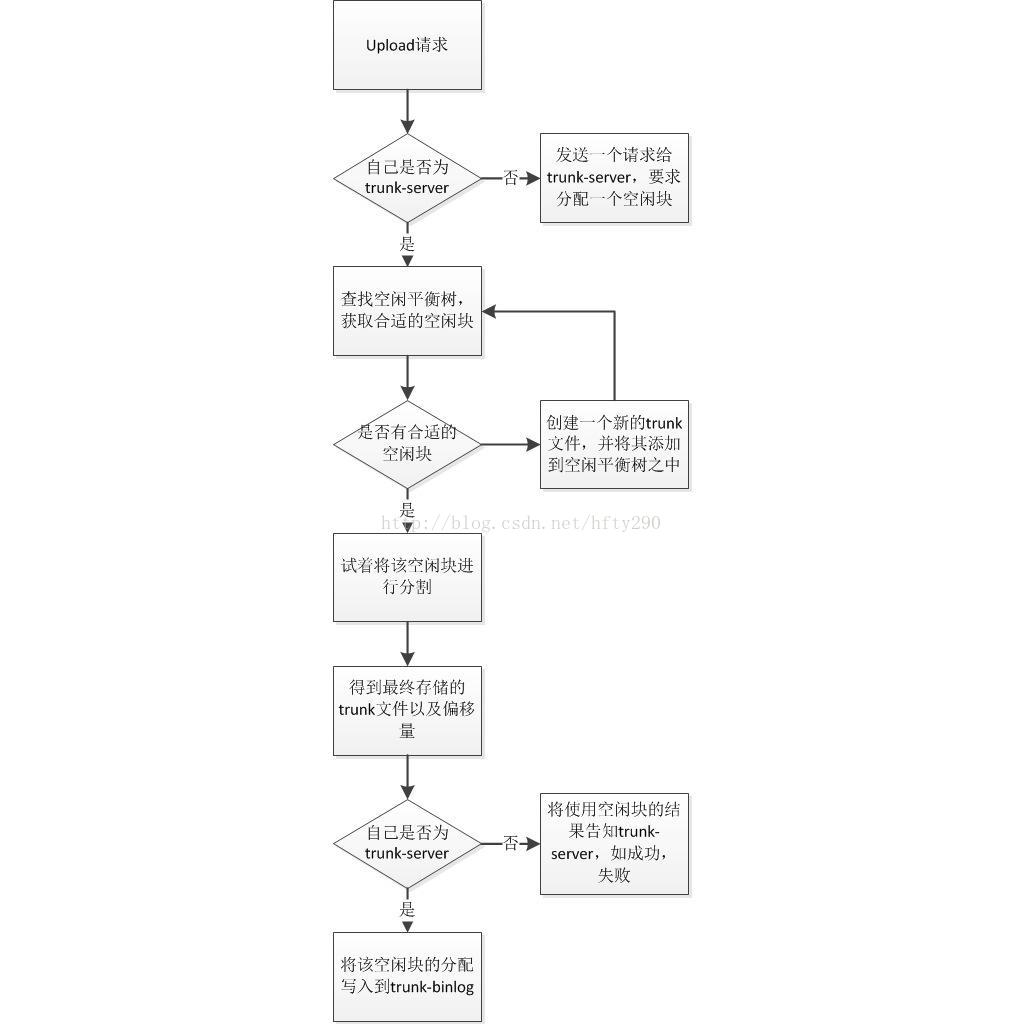
假若所有的Storage都具有分配空闲空间的能力（upload文件时自主决定存储到哪个TrunkFile之中），那么可能会***由于同步延迟导致数据冲突***。

例如：

**Storage-A**中上传了一个文件A.txt 100KB，将其保存到000001这个TrunkFile的***开头***，与此同时，**Storage-B**也上传了一个文件B.txt 200KB，也将其保存在000001这个TrunkFile文件的***开头***，当**Storage-B**收到**Storage-A**的同步信息时，他无法将A.txt 保存在000001这个trunk文件的开头，因此这个位置已经被B.txt占用。

为了处理这种冲突，引入了**TrunkServer**概念，只有TrunkServer才有权限分配空闲空间，决定文件应该保存到哪个TrunkFile的什么位置。**TrunkServer由Tracker指定，并且在心跳信息中通知所有的Storage（此处也会存在问题，可以思考一下）。**

引入TrunkServer之后，一次Upload请求，Storage的处理流程图如下：



### TrunkFile同步

**开启了合并存储服务后，除了原本的源文件（trunk文件和未合并文件）同步之外，TrunkServer还多了TrunkBinlog的同步（非TrunkServer没有TrunkBinlog同步）。**源文件的同步与没有开启合并存储时过程完全一样，都是从binlog触发同步文件

TrunkBinlog记录了TrunkServer所有分配与回收空闲块的操作，由TrunkServer同步给同组中的其他Storage。TrunkServer为同组中的其他Storage各创建一个同步线程，【每秒】将TrunkBinlog的变化同步出去。同组的Storage接收到TrunkBinlog【只是保存到文件中，不做其他任何操作】

**TrunkBinlog**文件文件记录如下：

 1410750754 A 0 0 0 1 0 67108864  
 ​  
 1410750754 D 0 0 0 1 0 67108864

**各字段含义如下（按照顺序）：**

时间戳

操作类型（A：增加，D：删除）

store\_path\_index

sub\_path\_high

sub\_path\_low

fileid（TrunkFile文件名，比如000001）

offset（在TrunkFile文件中的偏移量）

size（占用的大小，按照slot对齐）

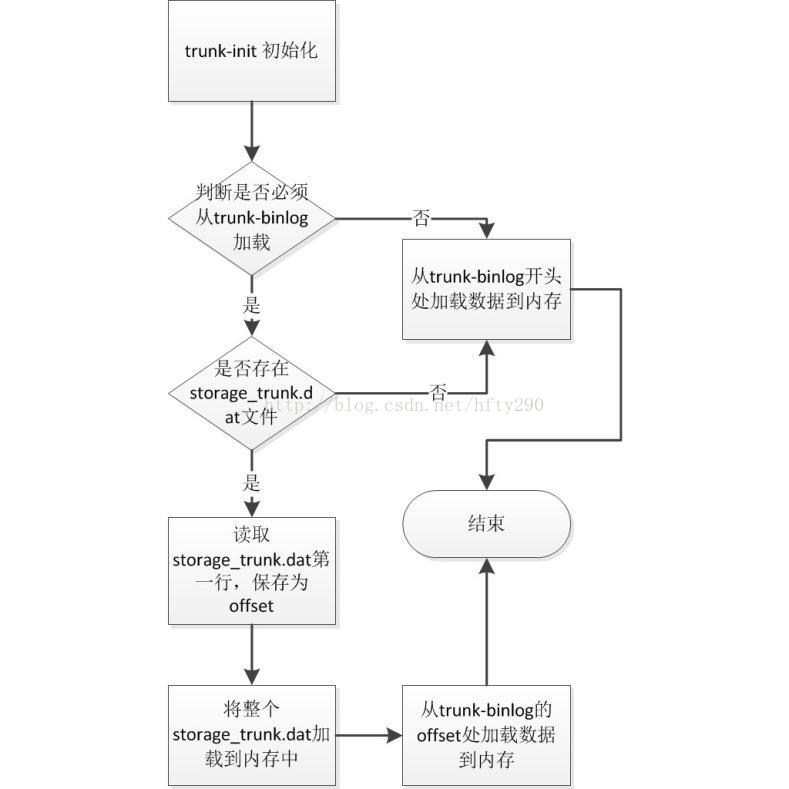
## 空闲平衡树重建

当作为TrunkServer的Storage启动时可以从TrunkBinlog文件中中加载所有的空闲块分配与加入操作，***这个过程就可以实现空闲平衡树的重建***。

### 为什么要重建空闲平衡树？

当长期运行时，随着空闲块的不断删除添加会导致TrunkBinlog文件很大，那么加载时间会很长，FastDFS为了解决这个问题，引入了检查点文件storage\_trunk.dat，每次TrunkServer进程退出时，会将当前内存里的空闲平衡树导出为storage\_trunk.dat文件，该文件的第一行为TrunkBinlog的offset，也就是该检查点文件负责到这个offset为止的TrunkBinlog。也就是说下次TrunkServer启动的时候，先加载storage\_trunk.dat文件，然后继续加载这个offset之后的TrunkBinlog文件内容。

下面为TrunkServer初始化的流程图：



### TrunkBinlog压缩

上文提到的storage\_trunk.dat既是**检查点文件**，其实也是一个**压缩文件**，因为从内存中将整个空闲平衡树直接导出，没有了中间步骤（创建、删除、分割等步骤），因此文件就很小。**这种方式虽然实现了TrunkServer自身重启时快速加载空闲平衡树的目的，但是并没有实际上缩小TrunkBinlog文件的大小。**假如这台TrunkServer宕机后，Tracker会选择另外一台机器作为新的TrunkServer，这台新的TrunkServer就必须从很庞大的TrunkBinlog中加载空闲平衡树，由于TrunkBinlog文件很大，这将是一个很漫长的过程

为了减少TrunkBinlog，可以选择压缩文件，在TrunkServer初始化完成后，或者退出时，可以将storage\_trunk.dat与其负责偏移量之后的TrunkBinlog进行合并，产生一个新的TrunkBinlog。由于此时的TrunkBinlog已经从头到尾整个修改了，就需要将该文件完成的同步给同组内的其他Storage，为了达到该目的，FastDFS使用了如下方法：

1）TrunkServer将TrunkBinlog同步给组内其他Storage时会将同步的最后状态记录到一个mark文件之中，比如同步给A，则记录到A.mark文件（其中包括最后同步成功的TrunkBinlog偏移量）。

2）TrunkServer在将storage\_trunk.dat与TrunkBinlog合并之后，就将本地记录TrunkBinlog最后同步状态的所有mark文件删除，如，一组有A、B、C，其中A为TrunkServer则A此时删除B.mark、C.mark。

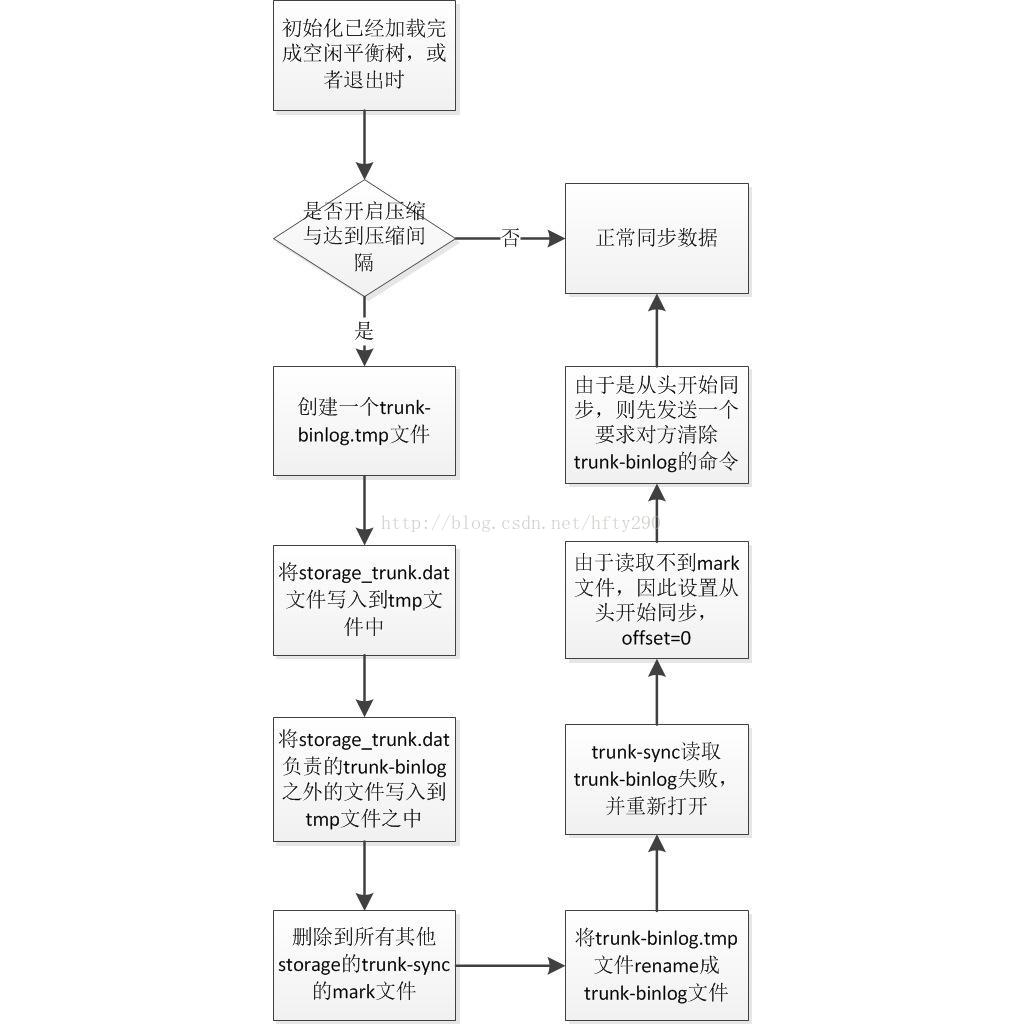
3）当下次TrunkServer要同步TrunkBinlog到B、C时，发现找不到B.mark、C.mark文件，就会自动转换成从头开始同步文件。

4）当TrunkServer判断需要从头开始同步TrunkBinlog，由于担心B、C已经有旧的文件，因此就需要向B、C发送一个删除旧的TrunkBinlog的命令。

5）发送删除命令成功之后，就可以从头开始将TrunkBinlog同步给B、C了。

 大家发现了么，这里的删除TrunkBinlog文件，会有一个时间窗口，就是删除B、C的TrunkBinlog文件之后，与将TrunkBinlog同步给他们之前，假如TrunkBinlog宕机了，那么组内的B、C都会没有TrunkBinlog可使用。

流程图如下：



# 图片压缩

## image\_filter模块

nginx\_http\_image\_filter\_module在nginx 0.7.54以后才出现的，用于对**JPEG, GIF和PNG**图片进行转换处理（**压缩图片、裁剪图片、旋转图片**）。这个模块默认不被编译，所以要在编译nginx源码的时候，加入相关配置信息（下面会给出相关配置）

## 检测nginx模块安装情况

[root@localhost img]# /kkb/server/nginx/sbin/nginx -V

nginx version: nginx/1.15.6

built by gcc 4.8.5 20150623 (Red Hat 4.8.5-28) (GCC)

built with OpenSSL 1.0.2k-fips 26 Jan 2017

TLS SNI support enabled

configure arguments: --prefix=/kkb/server/nginx --with-http\_stub\_status\_module --with-http\_ssl\_module --with-http\_realip\_module

## 安装步骤

* **安装gd，HttpImageFilterModule模块需要依赖gd-devel的支持**

 yum -y install gd-devel

* **将http\_image\_filter\_module包含进来**

 cd /root/nginx-1.15.6  
 ​  
 ​  
 ./configure \  
 --prefix=/kkb/server/nginx \  
 --pid-path=/var/run/nginx/nginx.pid \  
 --lock-path=/var/lock/nginx.lock \  
 --error-log-path=/var/log/nginx/error.log \  
 --http-log-path=/var/log/nginx/access.log \  
 --http-client-body-temp-path=/var/temp/nginx/client \  
 --http-proxy-temp-path=/var/temp/nginx/proxy \  
 --http-fastcgi-temp-path=/var/temp/nginx/fastcgi \  
 --http-uwsgi-temp-path=/var/temp/nginx/uwsgi \  
 --http-scgi-temp-path=/var/temp/nginx/scgi \  
 --with-http\_gzip\_static\_module \  
 --add-module=/opt/fastdfs-nginx-module-1.20/src \ #http模块必须添加  
 --with-http\_stub\_status\_module \  
 --with-http\_ssl\_module \  
 --with-http\_realip\_module \  
 --with-http\_image\_filter\_module  
 ​  
 ​  
 make && make install

注意：重新编译时，必须添加上nginx编译http的扩展模块,否则无法访问fastdfs图片

## 访问FastDFS图片

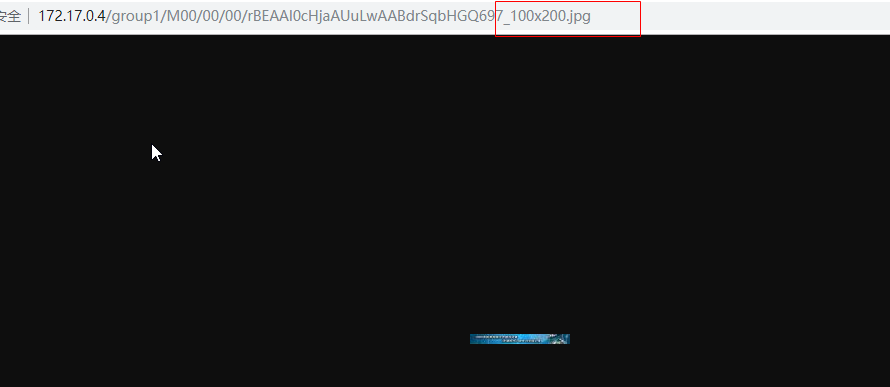
 location ~ group1/M00/(.+)\_(\d+)x(\d+)\.(jpg|gif|png) {  
  # 设置别名（类似于root的用法）  
            alias /kkb/server/fastdfs/storage/data/;  
             # fastdfs中的ngx\_fastdfs\_module模块  
            ngx\_fastdfs\_module;  
             set $w $2;  
             set $h $3;  
 ​  
             if ($w != "0") {  
                rewrite group1/M00(.+)\_(\d+)x(\d+)\.(jpg|gif|png)$ group1/M00$1.$4 break;  
            }  
 ​  
             if ($h != "0") {  
                rewrite group1/M00(.+)\_(\d+)x(\d+)\.(jpg|gif|png)$ group1/M00$1.$4 break;  
            }  
 ​  
             #根据给定的长宽生成缩略图     
            image\_filter resize $w $h;  
 ​  
             #原图最大2M，要裁剪的图片超过2M返回415错误，需要调节参数image\_filter\_buffer      
            image\_filter\_buffer 2M;  
 ​  
             #try\_files group1/M00$1.$4 $1.jpg;  
        }

**重启Nginx**

**注意事项：**由于添加了新的模块，所以nginx需要重新启动，不需要重新加载reload

 /kkb/server/nginx/sbin/nginx -s stop  
 /kkb/server/nginx/sbin/nginx

## 访问方式



image\_filter 模块详细配置说明：

 image\_filter off;  
 #关闭模块  
 ​  
 ​  
 image\_filter test;  
 #确保图片是jpeg gif png否则返415错误  
 ​  
 ​  
 image\_filter size;  
 #输出有关图像的json格式：例如以下显示{ "img" : { "width": 100, "height": 100, "type": "gif" } } 出错显示：{}  
 ​  
 ​  
 image\_filter rotate 90|180|270;  
 #旋转指定度数的图像，參数能够包括变量，单独或一起与resize crop一起使用。  
 ​  
 ​  
 image\_filter resize width height;  
 #按比例降低图像到指定大小，公降低一个能够还有一个用"-"来表示,出错415，參数值可包括变量，能够与rotate一起使用，则两个一起生效。  
 ​  
 ​  
 image\_filter crop width height;  
 #按比例降低图像比較大的側面积和还有一側多余的载翦边缘，其他和rotate一样。没太理解  
 ​  
 ​  
 image\_filter\_buffer 10M;  
 #设置读取图像缓冲的最大大小，超过则415错误。  
 ​  
 ​  
 image\_filter\_interlace on;  
 #假设启用，终于的图像将被交错。对于JPEG，终于的图像将在“渐进式JPEG”格式。  
 ​  
 ​  
 image\_filter\_jpeg\_quality 95;  
 #设置变换的JPEG图像的期望质量。可接受的值是从1到100的范围内。较小的值通常意味着既降低图像质量，降低数据传输，推荐的最大值为95。參数值能够包括变量。  
 ​  
 ​  
 image\_filter\_sharpen 100;  
 #添加了终于图像的清晰度。锐度百分比能够超过100。零值将禁用锐化。參数值能够包括变量。  
 ​  
 ​  
 image\_filter\_transparency on;  
 #定义是否应该透明转换的GIF图像或PNG图像与调色板中指定的颜色时，能够保留。透明度的损失将导致更好的图像质量。在PNG的Alpha通道总是保留透明度。

# 快速访问

# Java客户端

**mvn install:install-file -DgroupId=org.csource.fastdfs -DartifactId=fastdfs -Dversion=1.25 -Dpackaging=jar -Dfile=C:\Users\hubin\Desktop\java5\fastdfs\day02\client\fastdfs-client-java-1.27-RELEASE.jar**