一、问题引导

- 互联网环境中的文件如何存储？(待完善)

  不能存储本地、**NFS（mount挂载）、HDFS、FastDFS、云存储（图床等）**

- 互联网环境中的文件如何进行HTTP访问？(待完善)

  web服务器：Nginx、Apache等

二、FastDFS介绍

1.FastDFS是什么？

- FastDFS 是一个使用C编写的开源的高性能**分布式文件系统（Distributed File System,简称DFS）**。
- 它由**淘宝**开发平台部资深架构师**余庆**开发。FastDFS孵化平台(ChinaUnix)版块http://bbs.chinaunix.net/forum-240-1.html。
- 它对文件进行管理，功能包括：**文件存储、文件同步、文件访问（文件上传、文件下载）等**，解决了**大容量存储和负载均衡**的问题。
- 特别适合以文件为载体的在线服务，如相册网站、视频网站、电商网站等等。特别**适合以中小文件**（建议范围：**4KB < file_size <500MB**）为载体的在线服务。
- FastDFS为互联网量身定制，充分考虑了**冗余备份、负载均衡、线性扩容**等机制，并注重**高可用、高性能**等指标，使用FastDFS很容易搭建一套高性能的文件服务器集群提供文件上传、下载等服务。

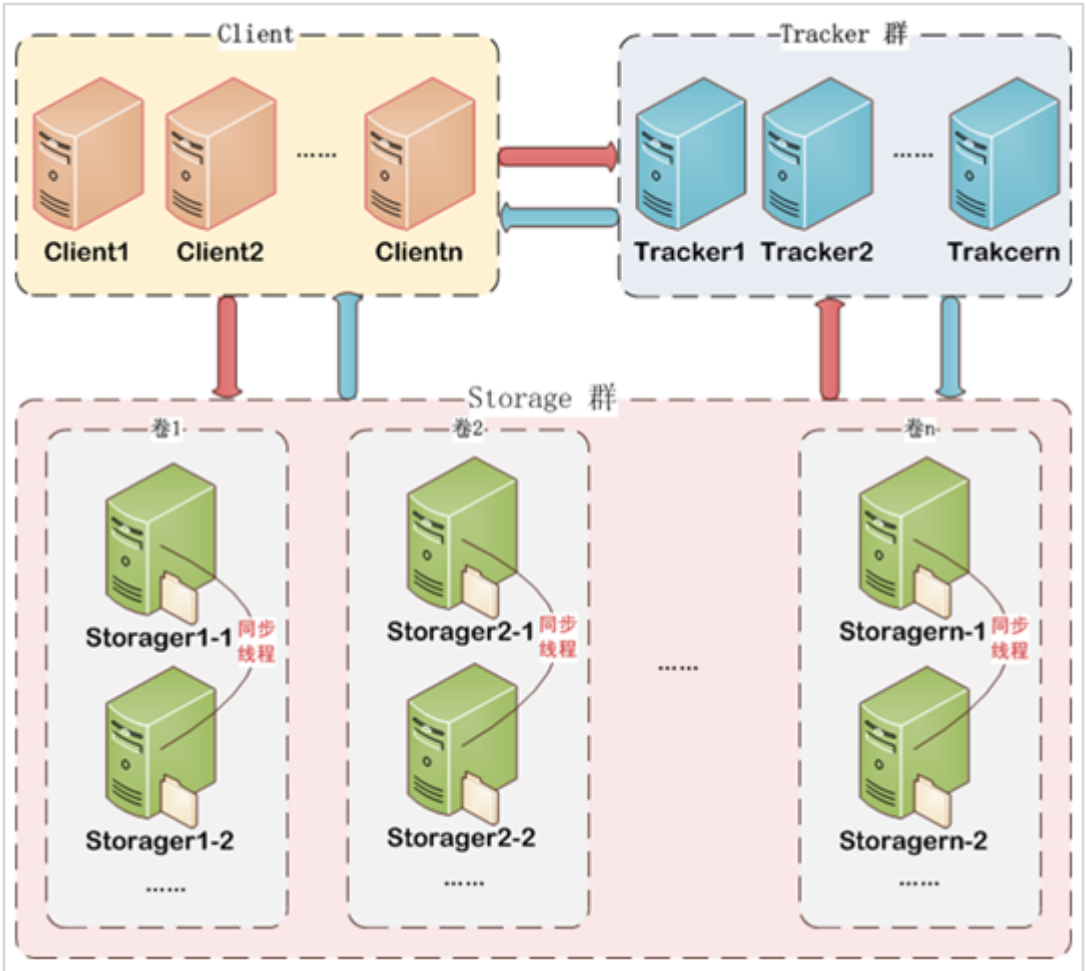
三、FastDFS架构原理分析（重点）

1.架构分析

FastDFS 系统有三个角色：跟踪服务器(Tracker Server)、存储服务器(Storage Server)和客户端(Client)。

- **Tracker Server**：跟踪服务器
  - 主要做调度工作，并对Storage Server起到负载均衡的作用；
  - 负责管理所有的 storage server和 group，每个 storage 在启动后会连接 Tracker，告知自己所属 group 等信息，并保持周期性心跳。
  - Tracker Server可以有多台，**Tracker Server之间是相互平等关系同时提供服务，Tracker Server不存在单点故障。客户端请求Tracker Server采用轮询方式，如果请求的Tracker无法提供服务则换另一个Tracker。**
- **Storage Server**：存储服务器
  - 主要提供容量和备份服务；
  - 以 group 为单位，不同group之间互相独立，每个 group 内可以有多台 storage server，数据互为备份。
  - **采用分组存储方式的好处是灵活、可控性较强。**比如上传文件时，可以由客户端直接指定上传到的组也可以由Tracker进行调度选择。

    - 一个分组的存储服务器访问压力较大时，可以在该组增加存储服务器来扩充服务能力（纵向扩容）。当系统容量不足时，可以增加组来扩充存储容量（横向扩容）。
- **Client**：客户端
    - 上传下载数据的服务器，也就是我们自己的项目所部署在的服务器。



2.存储策略

为了支持大容量，存储节点（服务器）采用了**分卷（或分组）的组织方式。**存储系统由一个或多个卷组成，**卷与卷之间的文件是相互独立的**，所有卷的文件容量累加就是整个存储系统中的文件容量。一个卷可以由一台或多台存储服务器组成，**一个卷下的存储服务器中的文件都是相同的**，卷中的多台存储服务器起到了**冗余备份**和**负载均衡**的作用。

在卷中增加服务器时，同步已有的文件由系统自动完成，同步完成后，系统自动将新增服务器切换到线上提供服务。当存储空间不足或即将耗尽时，可以动态添加卷。只需要增加一台或多台服务器，并将它们配置为一个新的卷，这样就扩大了存储系统的容量。

3.Storage状态收集

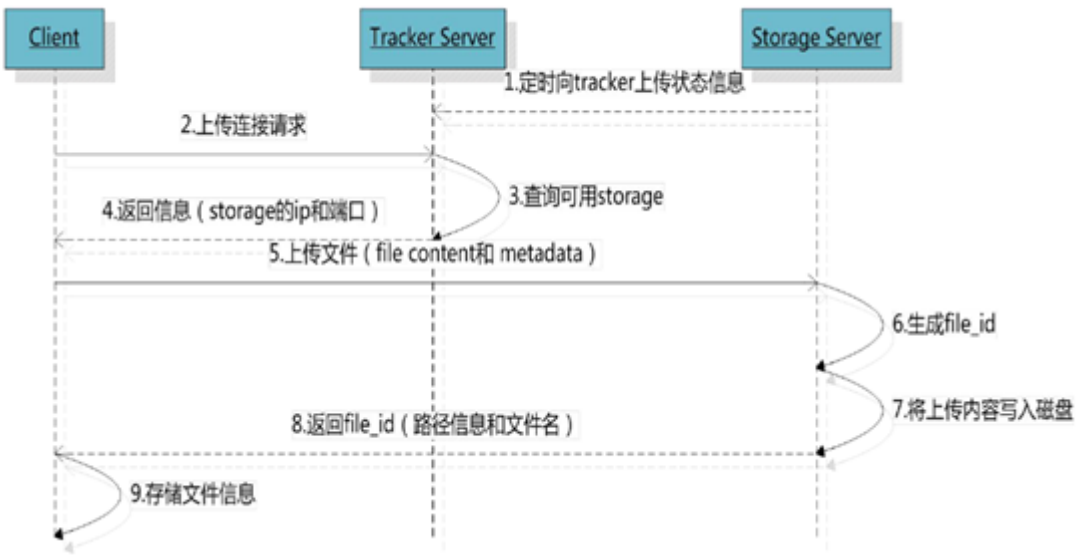
Storage Server会通过配置连接集群中所有的Tracker Server，定时向他们报告自己的状态，包括**磁盘剩余空间、文件同步状况、文件上传下载次数等统计信息。**

storage server有7个状态，如下：

```
FDFS_STORAGE_STATUS_INIT       :初始化，尚未得到同步已有数据的源服务器
FDFS_STORAGE_STATUS_WAIT_SYNC  :等待同步，已得到同步已有数据的源服务器
FDFS_STORAGE_STATUS_SYNCING    :同步中
FDFS_STORAGE_STATUS_DELETED    :已删除，该服务器从本组中摘除（注：本状态的功能尚未实现）
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
```

当storage server的状态为`FDFS_STORAGE_STATUS_ONLINE`时，当该storage server向tracker server发起一次heart beat时，tracker server将其状态更改为`FDFS_STORAGE_STATUS_ACTIVE`。

4.文件上传流程分析



流程说明：

1、tracker server收集storage server的状态信息

```
storage server定时向已知的tracker server（可以是多个）发送磁盘剩余空间、文件同步状况、文件上传下载次数等统计信息
storage server会连接整个集群中所有的Tracker server，向他们报告自己的状态。
```

2、选择tracker server

```
当集群中不止一个tracker server时，由于tracker之间是完全对等的关系，客户端在upload文件时可以任意选择一个trakcer。
```

3、选择存储的group

```
当tracker接收到upload file的请求时，会为该文件分配一个可以存储该文件的group，支持如下选择group的规则：
    1. Round robin，所有的group间轮询
    2. Specified group，指定某一个确定的group
    3. Load balance，剩余存储空间多多group优先
```

4、选择storage server

```
当选定group后，tracker会在group内选择一个storage server给客户端，支持如下选择storage的规则：
    1. Round robin，在group内的所有storage间轮询
    2. First server ordered by ip，按ip排序
    3. First server ordered by priority，按优先级排序（优先级在storage上配置）
```

5、选择storage path

```
当分配好storage server后，客户端将向storage发送写文件请求，storage将会为文件分配一个数据存储目录，支持如下规则（在storage配置文件可以通过配置store_path*参数来设置，该参数可以设置多个，通过*来区别）：
    1. Round robin，多个存储目录间轮询
    2. 剩余存储空间最多的优先
```

6、生成文件名

```
选定存储目录之后，storage会为文件生一个文件名称，由源storage server ip、文件创建时间、文件大小、文件crc32和一个随机数拼接而成，然后将这个二进制串进行base64编码，转换为可打印的字符串。
```

7、选择两级目录

```
当选定存储目录之后，storage会为文件分配一个fileid，每个存储目录下有两级256*256的子目录，storage会按文件名称进行两次hash（猜测），路由到其中一个子目录，然后将文件以fileid为文件名存储到该子目录下。
```

8、生成fileid

```
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。
```

5.文件同步分析

写文件时，客户端将文件写至group内一个storage server即认为**写文件成功**，storage server写完文件后，会由后台线程将文件同步至同group内其他的storage server。

同步规则总结如下：

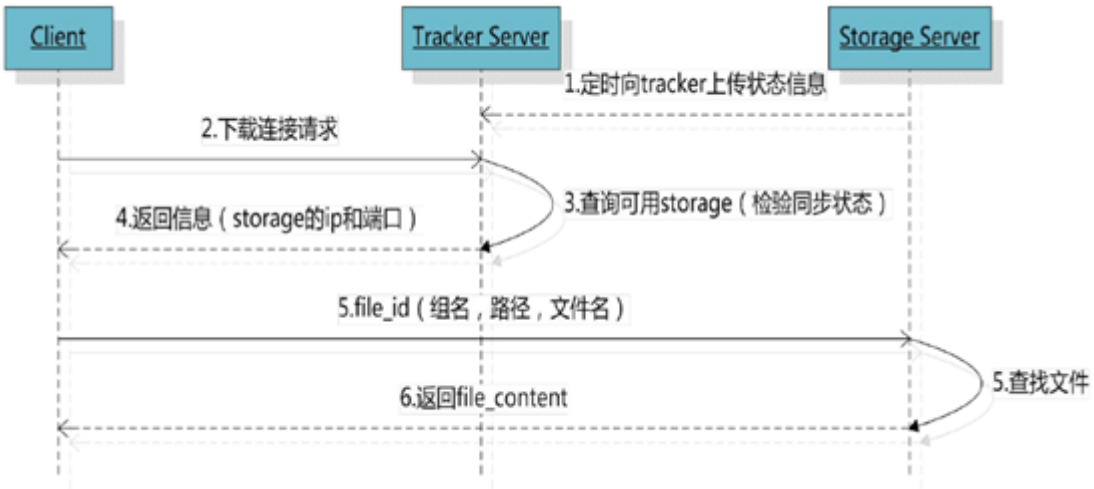
```
1. 只在本组内的storage server之间进行同步；
2. 源头数据才需要同步，备份数据不需要再次同步，否则就构成环路了；
3. 上述第二条规则有个例外，就是新增加一台storage server时，由已有的一台storage server将已有的所有数据（包括源头数据和备份数据）同步给该新增服务器。
```

每个storage写文件后，同时会写一份binlog，binlog里不包含文件数据，只包含文件名等元信息，**这份binlog用于后台同步**，storage会记录向group内其他storage同步的进度，以便重启后能接上次的进度继续同步；**进度以时间戳的方式进行记录**，所以最好能保证集群内所有server的时钟保持同步。

storage的同步进度会作为元数据的一部分汇报到tracker上，tracker在选择读storage的时候会以同步进度作为参考。

> 比如一个group内有A、B、C三个storage server，A向C同步到进度为T1 (T1以前写的文件都已经同步到B上了），B向C同步到时间戳为T2（T2 > T1)，tracker接收到这些同步进度信息时，就会进行整理，将最小的那个做为C的同步时间戳，本例中T1即为C的同步时间戳为T1（即所有T1以前写的数据都已经同步到C上了）；同理，根据上述规则，tracker会为A、B生成一个同步时间戳。

6.文件下载流程分析



客户端upload file成功后，会拿到一个storage生成的文件名，接下来客户端根据这个文件名即可访问到该文件。

流程说明：

1、tracker server收集storage server的状态信息

```
storage server定时向已知的tracker server（可以是多个）发送磁盘剩余空间、文件同步状况、文件上传下载次数等统计信息
storage server会连接整个集群中所有的Tracker server，向他们报告自己的状态。
```

2、选择tracker server

```
跟upload file一样，在download file时客户端可以选择任意tracker server。
```

3、选择可用的storage server

```
client发送download请求给某个tracker，必须带上文件名信息，tracker从文件名中解析出文件的group、路径信息、文件大小、创建时间、源storage server ip等信息，然后为该请求选择一个storage用来服务读请求。
由于group内的文件同步是在后台异步进行的，所以有可能出现在读的时候，文件还没有同步到某些storage server上，为了尽量避免访问到这样的storage，tracker按照如下规则选择group内可读的storage：
    1. 该文件上传到的源头storage - 源头storage只要存活着，肯定包含这个文件，源头的地址被编码在文件名中。
    2. 文件创建时间戳==storage被同步到的时间戳 且(当前时间-文件创建时间戳) > 文件同步最大时间（如5分钟） - 文件创建后，认为经过最大同步时间后，肯定已经同步到其他storage了。
    3. 文件创建时间戳 < storage被同步到的时间戳。 - 同步时间戳之前的文件确定已经同步了
    4. (当前时间-文件创建时间戳) > 同步延迟阀值（如一天）。 - 经过同步延迟阈值时间，认为文件肯定已经同步了。
```

7.新增Storage server分析

组内新增加一台`storage server A`时，由系统自动完成已有数据同步，处理逻辑如下： 1. storage server A连接tracker server，tracker server将storage server A的状态设置为`FDFS_STORAGE_STATUS_INIT`。storage server A询问追加同步的源服务器和追加同步截至时间点，如果该组内只有storage server A或该组内已成功上传的文件数为0，则没有数据需要同步，storage server A就可以提供在线服务，此时tracker将其状态设置为FDFS_STORAGE_STATUS_ONLINE，否则tracker server将其状态设置为FDFS_STORAGE_STATUS_WAIT_SYNC，进入第二步的处理；

2. 假设tracker server分配向storage server A同步已有数据的源storage server为B。同组的storage server和tracker server通讯得知新增了storage server A，将启动同步线程，并向tracker server询问向storage server A追加同步的源服务器和截至时间点。storage server B将把截至时间点之前的所有数据同步给storage server A；而其余的storage server从截至时间点之后进行正常同步，只把源头数据同步给storage server A。到了截至时间点之后，storage server B对storage server A的同步将由追加同步切换为正常同步，只同步源头数据； 3. storage server B向storage server A同步完所有数据，暂时没有数据要同步时，storage server B请求tracker server将storage server A的状态设置为FDFS_STORAGE_STATUS_ONLINE； 4 当storage server A向tracker server发起heart beat时，tracker server将其状态更改为FDFS_STORAGE_STATUS_ACTIVE。

四、FastDFS安装

1.下载

- 下载libfastcommon包
  - github源码

```
https://github.com/happyfish100/libfastcommon/releases
```

- linux wget下载

```
wget https://github.com/happyfish100/libfastcommon/archive/V1.0.39.tar.gz
```

- 下载fastdfs源码包
  - github源码

```
https://github.com/happyfish100/fastdfs/releases
```

- linux wget下载

```
wget https://github.com/happyfish100/fastdfs/archive/V5.11.tar.gz
```

2.需求

- Tracker Server：192.168.10.135
- Storage Server：192.168.10.136、192.168.10.137

3.安装

tracker和storage安装

`tracker server和storage server都有一些相同的安装操作，如下：`

- **安装gcc环境**

```
yum install -y gcc-c++
```

- **安装libevent**，FastDFS依赖libevent库（暂不安装）

```
yum install -y libevent
```

- **安装libfastcommon**，libfastcommon是FastDFS官方提供的包，包含了FastDFS运行所需要的一些基础库。

```
wget https://github.com/happyfish100/libfastcommon/archive/V1.0.39.tar.gz
tar -zxvf  V1.0.39.tar.gz
cd libfastcommon-1.0.39
./make.sh && ./make.sh  install
```

- **拷贝libfastcommon.so文件至/usr/lib目录（新版本不需要此步）**

```
cp  /usr/lib64/libfastcommon.so  /usr/lib/
```

> **注：**
>
> libfastcommon安装好后会自动将库文件拷贝至/usr/lib64下，由于FastDFS程序引用usr/lib目录，所以需要将/usr/lib64下的库文件拷贝至/usr/lib下。

- 下载安装FastDFS，进入FastDFS目录，编译安装

```
wget https://github.com/happyfish100/fastdfs/archive/V5.11.tar.gz
tar -zxvf  V5.11.tar.gz
cd cd fastdfs-5.11
./make.sh && ./make.sh install
```

- **拷贝/root/fastdfs-5.11/conf目录下的文件到/etc/fdfs目录下**

```
cp /kkb/soft/fastdfs-5.11/conf/* /etc/fdfs
```

tracker server配置

注意：base_path目录要存在

- 修改/etc/fdfs/tracker.conf

```
vim /etc/fdfs/tracker.conf
```

修改内容如下：

```
base_path=/kkb/server/fastdfs/tracker
```

- **创建tracker服务器上面的目录**

```
mkdir /kkb/server/fastdfs/tracker -p
```

storage server配置

注意：base_path和store_path0目录要存在

- 修改/etc/fdfs/storage.conf

```
vim /etc/fdfs/storage.conf
```

修改内容如下：

```
#指定storage的组名
group_name=group1
base_path=/kkb/server/fastdfs/storage
store_path0=/kkb/server/fastdfs/storage
#如果有多个挂载磁盘则定义多个store_path，如下
#store_path1=.....
#store_path2=......
#配置tracker服务器IP和端口
tracker_Server=111.231.106.221:22122
#如果有多个则配置多个tracker
#tracker_Server=192.168.101.4:22122
```

- **创建storage服务器上面的目录**

```
mkdir /kkb/server/fastdfs/storage -p
```

4.启动

4.1 Tracker启动命令

```
/usr/bin/fdfs_trackerd /etc/fdfs/tracker.conf
```

4.2 Storage启动命令

```
/usr/bin/fdfs_storaged /etc/fdfs/storage.conf
```

4.3 Tracker开机自启动

```
vim /etc/rc.d/rc.local
```

将运行命令行添加进文件：

```
/usr/bin/fdfs_trackerd /etc/fdfs/tracker.conf
```

4.4 Storage开机自启动

```
vim /etc/rc.d/rc.local
```

将运行命令行添加进文件：

```
/usr/bin/fdfs_storaged /etc/fdfs/storage.conf
```

5.上传图片测试

FastDFS安装成功后可通过【fdfs_test】命令测试上传、下载等操作。

- 修改client.conf

```
vim /etc/fdfs/client.conf
```

修改内容如下：

```
base_path=/kkb/server/fastdfs/client
tracker_Server=111.231.106.221:22122
```

- 创建client的数据目录

```
mkdir -p /kkb/server/fastdfs/client
```

- 使用fdfs_test命令将/home下的tomcat.png上传到FastDFS中

```
/usr/bin/fdfs_test  /etc/fdfs/client.conf  upload  /etc/fdfs/anti-steal.jpg
```

> **说明：**
>
> http://192.168.10.135/group1/M00/00/00/wKhlBVVY2M-AM_9DAAAT7-0xdqM485_big.png就是文件的访问路径。
>
> 对应storage服务器上的磁盘路径：
>
> `/home/fastdfs/fdfs_storage/data/00/00/wKhlBVVY2M-AM_9DAAAT7-0xdqM485_big.png`文件。

6.tracker.conf

6.1 基本配置

```
disable
#func：配置是否生效
#valu：true、false
disable=false
bind_addr
#func：绑定IP
#valu：IP地址
bind_addr=192.168.6.102
port
#func：服务端口
#valu：端口整数值
port=22122
connect_timeout
#func：连接超时
#valu：秒单位正整数值
connect_timeout=30
```

```
network_timeout
#func：网络超时
#valu：秒单位正整数值
network_timeout=60
base_path
#func：Tracker数据/日志目录地址
#valu：路径
base_path=/home/michael/fdfs/base4tracker
max_connections
#func：最大连接数
#valu：正整数值
max_connections=256
work_threads
#func：线程数，通常设置CPU数
#valu：正整数值
work_threads=4
store_lookup
#func：上传文件的选组方式。
#valu：0、1或2。
# 0：表示轮询
# 1：表示指定组
# 2：表示存储负载均衡（选择剩余空间最大的组）
store_lookup=2
store_group
#func：指定上传的组，如果在应用层指定了具体的组，那么这个参数将不会起效。另外如果store_lookup如果是0或2，则此参数无效。
#valu：group1等
store_group=group1
store_Server
#func：上传服务器的选择方式。（一个文件被上传后，这个storage Server就相当于这个文件的storage Server源，会对同组的storage Server推送这个文件达到同步效果）
#valu：0、1或2
# 0: 轮询方式（默认）
# 1: 根据ip 地址进行排序选择第一个服务器（IP地址最小者）
# 2: 根据优先级进行排序（上传优先级由storage Server来设置，参数名为upload_priority），优先级值越小优先级越高。
store_Server=0
store_path
#func：上传路径的选择方式。storage Server可以有多个存放文件的base path（可以理解为多个磁盘）。
#valu：
# 0: 轮流方式，多个目录依次存放文件
# 2: 存储负载均衡。选择剩余空间最大的目录存放文件（注意：剩余磁盘空间是动态的，因此存储到的目录或磁盘可能也是变化的）
store_path=0
download_Server
#func：下载服务器的选择方式。
#valu：
# 0：轮询（默认）
# 1：IP最小者
# 2：优先级排序（值最小的，优先级最高。）
download_Server=0
reserved_storage_space
```

```
#func：保留空间值。如果某个组中的某个服务器的剩余自由空间小于设定值，则文件不会被上传到这个组。
#valu：
# G or g for gigabyte
# M or m for megabyte
# K or k for kilobyte
reserved_storage_space=1GB
log_level
#func：日志级别
#valu：
# emerg for emergency
# alert
# crit for critical
# error
# warn for warning
# notice
# info for information
# debug for debugging
log_level=info
run_by_group / run_by_user
#func：指定运行该程序的用户组
#valu：用户组名或空
run_by_group=

#func：
#valu：
run_by_user=
allow_hosts
#func：可以连接到tracker Server的ip范围。可设定多个值。
#valu
allow_hosts=
check_active_interval
#func：检测 storage Server 存活的时间隔，单位为秒。
#       storage Server定期向tracker Server 发心跳，
#       如果tracker Server在一个check_active_interval内还没有收到storage Server的一次心跳，
#       那边将认为该storage Server已经下线。所以本参数值必须大于storage Server配置的心跳时间间隔。
#       通常配置为storage Server心跳时间间隔的2倍或3倍。
check_active_interval=120
thread_stack_size
#func：设定线程栈的大小。 线程栈越大，一个线程占用的系统资源就越多。
#       如果要启动更多的线程（V1.x对应的参数为max_connections，V2.0为work_threads），可以适当降低本参数值。
#valu：如64KB，默认值为64，tracker Server线程栈不应小于64KB
thread_stack_size=64KB
storage_ip_changed_auto_adjust
#func：这个参数控制当storage Server IP地址改变时，集群是否自动调整。注：只有在storage Server进程重启时才完成自动调整。
#valu：true或false
storage_ip_changed_auto_adjust=true
```

6.2 同步

```
storage_sync_file_max_delay
#func：同组storage服务器之间同步的最大延迟时间。存储服务器之间同步文件的最大延迟时间，根据实际情况进行调整
#valu：秒为单位，默认值为1天（24*3600）
#sinc：v2.0
storage_sync_file_max_delay=86400
storage_sync_file_max_time
#func：存储服务器同步一个文件需要消耗的最大时间，缺省为300s，即5分钟。
#sinc：v2.0
storage_sync_file_max_time=300
sync_log_buff_interval
#func：同步或刷新日志信息到硬盘的时间间隔。注意：tracker Server 的日志不是时时写硬盘的，而是先写内存。
#valu：以秒为单位
sync_log_buff_interval=10
```

6.3 trunk 和 slot

```
#func：是否使用trunk文件来存储几个小文件
#valu：true或false
#sinc：v3.0
use_trunk_file=false

#func：最小slot大小
#valu：<= 4KB，默认为256字节
#sinc：v3.0
slot_min_size=256

#func：最大slot大小
#valu：>= slot_min_size，当小于这个值的时候就存储到trunk file中。默认为16MB。
#sinc：v3.0
slot_max_size=16MB

#func：trunk file的size
#valu：>= 4MB，默认为64MB
#sinc：v3.0
trunk_file_size=64MB
```

6.4 HTTP相关

```
是否启用 HTTP
#func：HTTP是否生效
#valu：true或false
http.disabled=false
HTTP 服务器端口号
#func：tracker Server上的http port
#valu：
#note：只有http.disabled=false时才生效
http.Server_port=7271
检查Storage存活状态的间隔时间（心跳检测）
#func：检查storage http Server存活的间隔时间
```

```
#valu：单位为秒
#note：只有http.disabled=false时才生效
http.check_alive_interval=30
心跳检测使用的协议方式
#func：检查storage http Server存活的方式
#valu：
# tcp：连接到storage Server的http端口，不进行request和response。
# http：storage check alive url must return http status 200.
#note：只有http.disabled=false时才生效
http.check_alive_type=tcp
检查 Storage 状态的 URI
#func：检查storage http Server是否alive的uri/url
#note：只有http.disabled=false时才生效
http.check_alive_uri=/status.html
```

五、FastDFS-Nginx扩展模块源码分析

1. 背景

在大多数业务场景中，往往需要为FastDFS存储的文件提供http下载服务，而尽管FastDFS在其storage及tracker都内置了http服务， 但性能表现却不尽如人意；(余老师在 V4.05 以后的版本就把内置 HTTP服务去掉了) 作者余庆在后来的版本中增加了基于当前主流web服务器的扩展模块(包括nginx/apache)，其用意在于利用web服务器直接对本机storage数据文件提供http服务，以提高文件下载的性能。

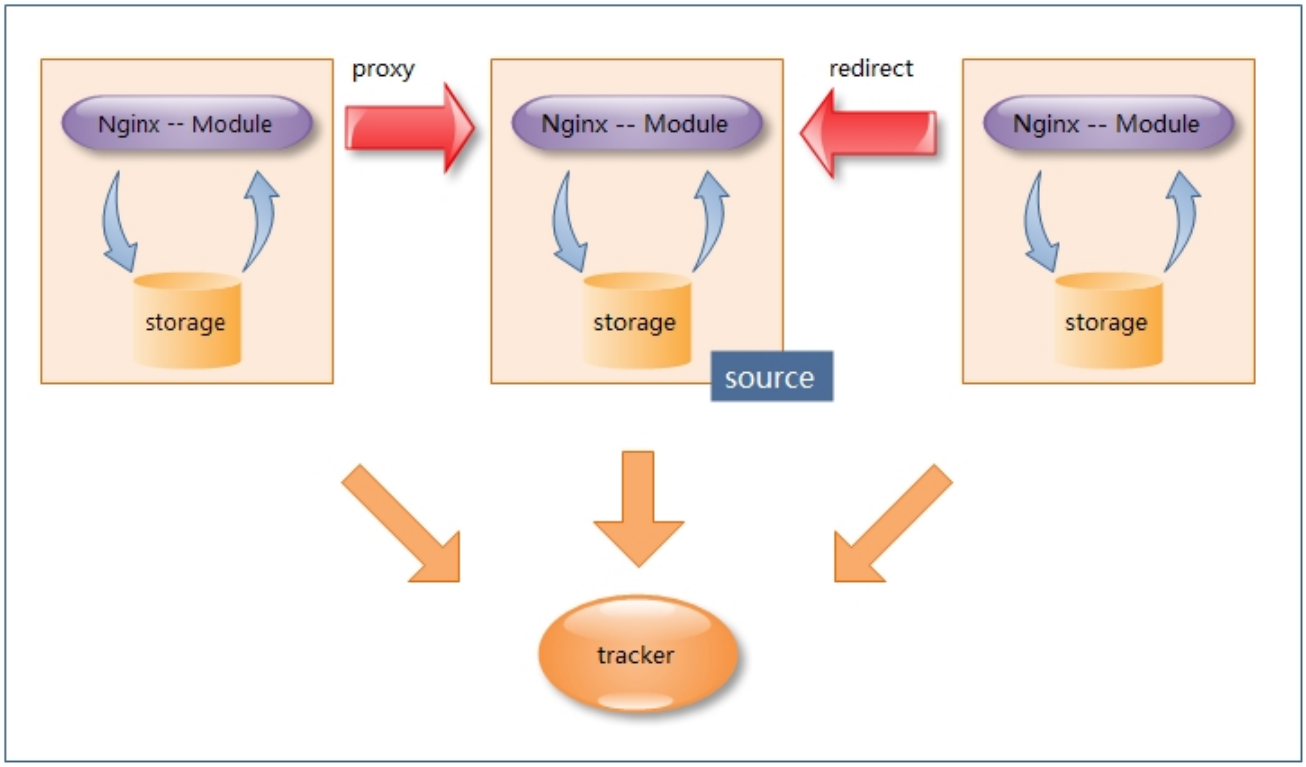
不使用Nginx的扩展模块，只安装web服务器（ Nginx和Apache ），也可以对存储的文件进行访问。

为什么要使用Nginx的扩展模块来访问存储的文件，原因有两个：

- 如果进行文件合并，那么不使用FastDFS的nginx扩展模块，是无法访问到具体的文件的，因为文件合并之后，多个小文件都是存储在一个trunk文件中的，在存储目录下，是看不到具体的小文件的。
- 如果文件未同步成功，那么不使用FastDFS的nginx扩展模块，是无法正常访问到指定的文件的，而使用了FastDFS的nginx扩展模块之后，如果要访问的文件未同步成功，那么会解析出来该文件的源存储服务器ip，然后将该访问请求重定向或者代理到源存储服务器中进行访问。

2. 概要介绍

2.1 FastDFS整合Nginx的参考架构



说明：

在每一台storage服务器主机上部署Nginx及FastDFS扩展模块，由Nginx模块对storage存储的文件提供http下载服务，仅当当前storage节点找不到文件时会向**源storage**主机发起`redirect`或`proxy`动作。

> **注：**
>
> 图中的tracker可能为多个tracker组成的集群；
>
> 且当前FastDFS的Nginx扩展模块支持单机多个group的情况

2.2 几个概念

- **storage_id**：指storage server的id，从FastDFS4.x版本开始，tracker可以对storage定义一组ip到id的映射，以id的形式对storage进行管理。而文件名写入的不再是storage的ip而是id，这样的方式对于数据迁移十分有利。
- **storage_sync_file_max_delay**：指storage节点同步一个文件最大的时间延迟，是一个阈值；如果当前时间与文件创建时间的差距超过该值则认为同步已经完成。
- **anti_steal_token**：指文件ID防盗链的方式，FastDFS采用`token认证`的方式进行文件防盗链检查。

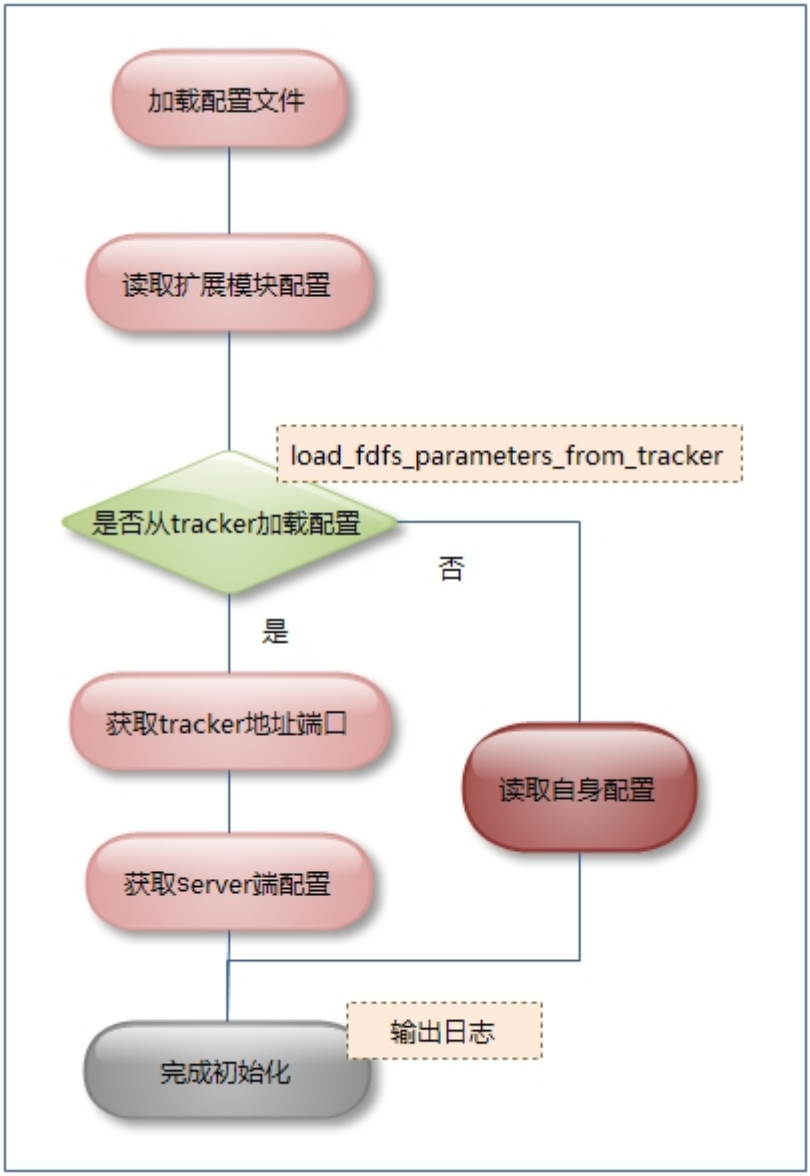
3. 实现原理

3.1 源码包说明

下载后的源码包很小，仅包括以下文件：

```
ngx_http_fastdfs_module.c    //nginx-module接口实现文件，用于接入fastdfs-module核心模块逻辑
common.c                     //fastdfs-module核心模块，实现了初始化、文件下载的主要逻辑
common.h                     //对应于common.c的头文件
config                       //编译模块所用的配置，里面定义了一些重要的常量，如扩展配置文件路径、文件下载chunk大小
mod_fastdfs.conf             //扩展配置文件的demo
```

3.2 初始化



3.2.1 加载配置文件

目标文件：**/etc/fdfs/mod_fastdfs.conf**

3.2.2 读取扩展模块配置

一些重要参数包括：

```
group_count            //group个数
url_have_group_name    //url中是否包含group
group.store_path       //group对应的存储路径
connect_timeout        //连接超时
network_timeout        //接收或发送超时
storage_server_port    //storage_server端口，用于在找不到文件情况下连接源storage下载文件(该做法已过时)
response_mode          //响应模式，proxy或redirect
load_fdfs_parameters_from_tracker //是否从tracker下载服务端配置
```

3.2.3 加载服务端配置

根据`load_fdfs_parameters_from_tracker`参数确定是否从tracker获取server端的配置信息

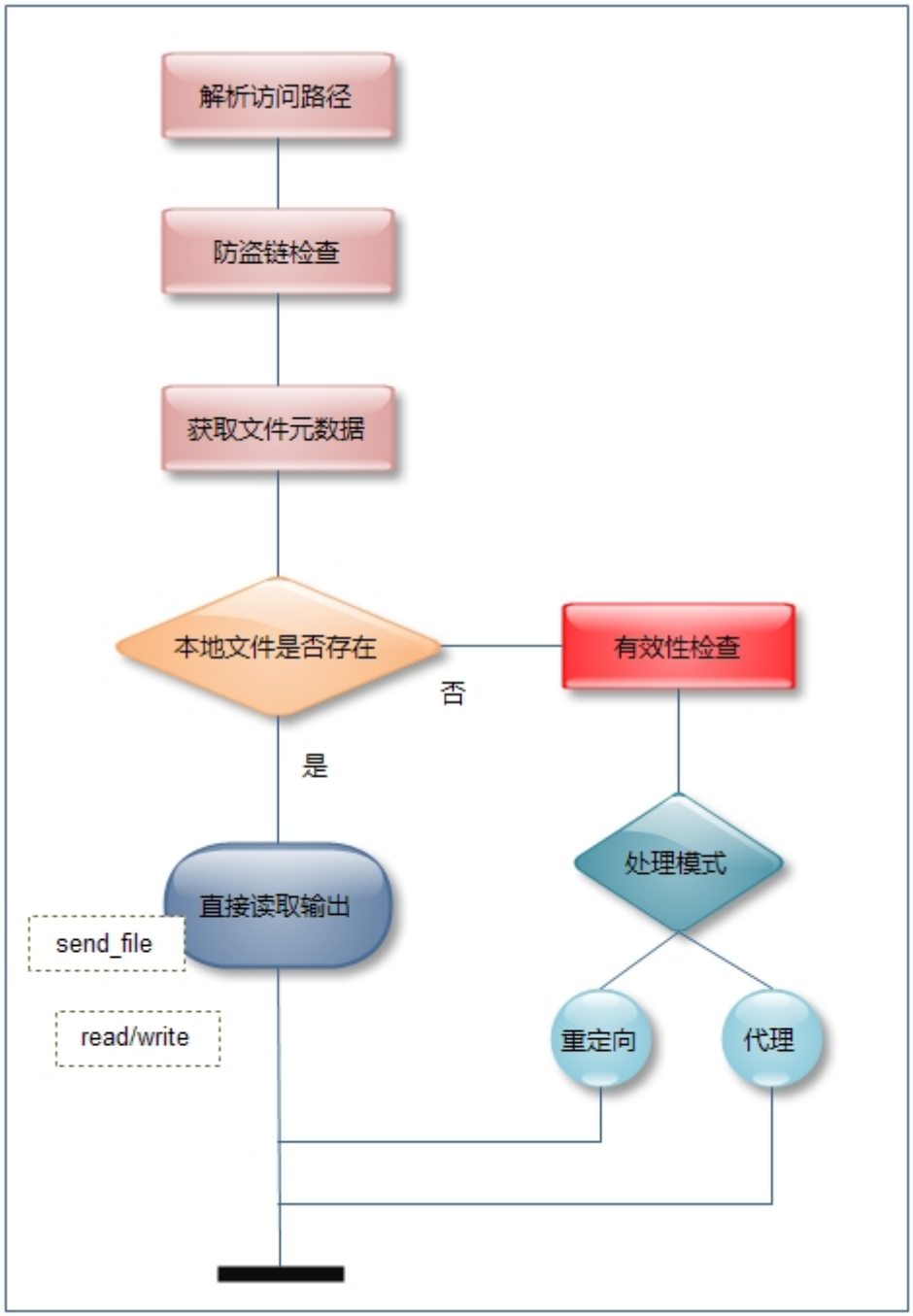
- **load_fdfs_parameters_from_tracker=true:**

1. 调用fdfs_load_tracker_group_ex解析tracker连接配置 ；
2. 调用fdfs_get_ini_context_from_tracker连接tracker获取配置信息；
3. 获取storage_sync_file_max_delay阈值
4. 获取use_storage_id
5. 如果use_storage_id为true，则连接tracker获取storage_ids映射表(调用方法：**fdfs_get_storage_ids_from_tracker_group**)

- **load_fdfs_parameters_from_tracker=false:**

1. 从mod_fastdfs.conf加载所需配置：storage_sync_file_max_delay、use_storage_id;
2. 如果use_storage_id为true，则根据storage_ids_filename获取storage_ids映射表(调用方法：**fdfs_load_storage_ids_from_file**)

3.3 下载过程(重点)



3.3.1 解析访问路径

得到`group`和`file_id_without_group`两个参数;

3.3.2 防盗链检查

- 根据**g_http_params.anti_steal_token**配置(见http.conf文件)，判断是否进行防盗链检查；
- 采用token的方式实现防盗链, 该方式要求下载地址带上token，且token具有时效性(由ts参数指明);

检查方式：

```
   md5(fileid_without_group + privKey + ts) = token; 同时ts没有超过ttl范围 (可参考 JavaClient CommonProtocol)
```

调用方法：**fdfs_http_check_token** 关于FastDFS的防盗链可参考： http://bbs.chinaunix.net/thread-1916999-1-1.html

3.3.3 获取文件元数据

根据文件ID 获取元数据信息, 包括：**源storage ip,文件路径、名称，大小 代码**：

```
    if ((result=fdfs_get_file_info_ex1(file_id, false, &file_info)) != 0)...
```

在**fdfs_get_file_info_ex1** 的实现中，存在一个取巧的逻辑： 当获得文件的ip段之后，仍然需要确定该段落是storage的id还是ip。 **代码**：

```
fdfs_shared.func.c
-> fdfs_get_server_id_type(ip_addr.s_addr) == FDFS_ID_TYPE_SERVER_ID
...
     if (id > 0 && id <= FDFS_MAX_SERVER_ID) {
        return FDFS_ID_TYPE_SERVER_ID;
     } else  {
       return FDFS_ID_TYPE_IP_ADDRESS;
     }
```

判断标准为ip段的整数值是否在 0 到 -> **FDFS_MAX_SERVER_ID**(见tracker_types.h)之间； *其中 FDFS_MAX_SERVER_ID = (1 << 24) - 1，该做法利用了ipv4地址的特点(由4*8个二进制位组成)，即ipv4地址数值务必大于该阈值*

3.3.4 检查本地文件是否存在

调用**trunk_file_stat_ex1**获取本地文件信息，该方法将实现：

1. 辨别当前文件是trunkfile还是singlefile
2. 获得文件句柄fd
3. 如果文件是trunk形式则同时也将相关信息(偏移量/长度)一并获得

代码：

```
    if (bSameGroup)
    {
        FDFSTrunkHeader trunkHeader;
        if ((result=trunk_file_stat_ex1(pStorePaths, store_path_index, \
            true_filename, filename_len, &file_stat, \
            &trunkInfo, &trunkHeader, &fd)) != 0)
        {
            bFileExists = false;
        }
        else
        {
            bFileExists = true;
        }
    }
    else
```

```
    {
        bFileExists = false;
        memset(&trunkInfo, 0, sizeof(trunkInfo));
    }
```

3.3.5 文件不存在的处理

- **进行有效性检查**

检查项有二：

代码：

```
    if (is_local_host_ip(file_info.source_ip_addr) || \
       (file_info.create_timestamp > 0 && (time(NULL) - \
           file_info.create_timestamp > '''storage_sync_file_max_delay''')))
```

在通过有效性检查之后将进行代理或重定向处理

- **重定向模式**

配置项`response_mode = redirect`，此时服务端返回返回302响应码，url如下：

```
http:// {源storage地址} : {当前port} {当前url} {参数"redirect=1"}(标记已重定向过)
```

代码：

```
     response.redirect_url_len = snprintf( \
               response.redirect_url, \
               sizeof(response.redirect_url), \
               "http://%s%s%s%c%s", \
               file_info.source_ip_addr, port_part, \
               path_split_str, url, \
               param_split_char, "redirect=1");
```

注：该模式下要求源storage配备公开访问的webserver、同样的端口(一般是80)、同样的path配置。

- **代理模式**

配置项`response_mode = proxy`，该模式的工作原理如同反向代理的做法，而**仅仅使用源storage地址作为代理proxy的host**，其余部分保持不变。 **代码**：

```
        if (pContext->proxy_handler != NULL)
        {
            return pContext->proxy_handler(pContext->arg, \
                    file_info.source_ip_addr);
        }
        //其中proxy_handler方法来自ngx_http_fastdfs_module.c文件的
ngx_http_fastdfs_proxy_handler方法
        //其实现中设置了大量回调、变量，并最终调用代理请求方法，返回结果：
        rc = ngx_http_read_client_request_body(r, ngx_http_upstream_init);  //执行代理
请求，并返回结果
```

3.3.6 输出本地文件

- 根据是否**trunkfile**获取文件名，文件名长度、文件offset;

代码：

```
    bTrunkFile = IS_TRUNK_FILE_BY_ID(trunkInfo);
    if (bTrunkFile)
    {
        trunk_get_full_filename_ex(pStorePaths, &trunkInfo, \
                full_filename, sizeof(full_filename));
        full_filename_len = strlen(full_filename);
        file_offset = TRUNK_FILE_START_OFFSET(trunkInfo) + \
                pContext->range.start;
    }
    else
    {
        full_filename_len = snprintf(full_filename, \
                sizeof(full_filename), "%s/data/%s", \
                pStorePaths->paths[store_path_index], \
                true_filename);
        file_offset = pContext->range.start;
    }
```

- 若nginx开启了send_file开关而且当前为非chunkFile的情况下尝试**使用sendfile方法以优化性能**;

代码：

```
    if (pContext->send_file != NULL && !bTrunkFile)
    {
        http_status = pContext->if_range ? \
                HTTP_PARTIAL_CONTENT : HTTP_OK;
        OUTPUT_HEADERS(pContext, (&response), http_status)
        ......
        return pContext->send_file(pContext->arg, full_filename, \
                full_filename_len, file_offset, download_bytes);
    }
```

- 否则使用lseek 方式随机访问文件，并输出相应的段;

做法：使用chunk方式循环读，输出... **代码**：

```
    while (remain_bytes > 0)
    {
        read_bytes = remain_bytes <= FDFS_OUTPUT_CHUNK_SIZE ? \
                 remain_bytes : FDFS_OUTPUT_CHUNK_SIZE;
        if (read(fd, file_trunk_buff, read_bytes) != read_bytes)
        {
            close(fd);
            ......
            return HTTP_INTERNAL_SERVER_ERROR;
        }

        remain_bytes -= read_bytes;
        if (pContext->send_reply_chunk(pContext->arg, \
            (remain_bytes == 0) ? 1: 0, file_trunk_buff, \
            read_bytes) != 0)
        {
            close(fd);
            return HTTP_INTERNAL_SERVER_ERROR;
        }
    }
```

其中chunk大小见config文件配置： **-DFDFS_OUTPUT_CHUNK_SIZE='256*1024'**

六、配置FastDFS的Nginx模块

FastDFS的Nginx模块需要安装到每一台storage server中

1.下载文件

```
wget https://github.com/happyfish100/fastdfs-nginx-module/archive/V1.20.tar.gz
```

2.解压缩

```
tar -zxvf V1.20.tar.gz
```

3.修改config文件（特别关键的一步）

修改 `fastdfs-nginx-module/src/config` 文件

```
cd fastdfs-nginx-module-1.20/src
vim config
```

- **修改前的内容如下：**

```
ngx_addon_name=ngx_http_fastdfs_module

if test -n "${ngx_module_link}"; then
    ngx_module_type=HTTP
    ngx_module_name=$ngx_addon_name
    ngx_module_incs="/usr/local/include"
    ngx_module_libs="-lfastcommon -lfdfsclient"
    ngx_module_srcs="$ngx_addon_dir/ngx_http_fastdfs_module.c"
    ngx_module_deps=
    CFLAGS="$CFLAGS -D_FILE_OFFSET_BITS=64 -DFDFS_OUTPUT_CHUNK_SIZE='256*1024' -DFDFS_MOD_CONF_FILENAME='\"/etc/fdfs/mod_fastdfs.conf\"'"
    . auto/module
else
    HTTP_MODULES="$HTTP_MODULES ngx_http_fastdfs_module"
    NGX_ADDON_SRCS="$NGX_ADDON_SRCS $ngx_addon_dir/ngx_http_fastdfs_module.c"
    CORE_INCS="$CORE_INCS /usr/local/include"
    CORE_LIBS="$CORE_LIBS -lfastcommon -lfdfsclient"
    CFLAGS="$CFLAGS -D_FILE_OFFSET_BITS=64 -DFDFS_OUTPUT_CHUNK_SIZE='256*1024' -DFDFS_MOD_CONF_FILENAME='\"/etc/fdfs/mod_fastdfs.conf\"'"
fi
```

- **其中第6行和第15行要进行修改，修改后的内容如下：**

```
ngx_addon_name=ngx_http_fastdfs_module

if test -n "${ngx_module_link}"; then
    ngx_module_type=HTTP
    ngx_module_name=$ngx_addon_name
    ngx_module_incs="/usr/include/fastdfs /usr/include/fastcommon/"
    ngx_module_libs="-lfastcommon -lfdfsclient"
    ngx_module_srcs="$ngx_addon_dir/ngx_http_fastdfs_module.c"
    ngx_module_deps=
    CFLAGS="$CFLAGS -D_FILE_OFFSET_BITS=64 -DFDFS_OUTPUT_CHUNK_SIZE='256*1024' -DFDFS_MOD_CONF_FILENAME='\"/etc/fdfs/mod_fastdfs.conf\"'"
    . auto/module
else
    HTTP_MODULES="$HTTP_MODULES ngx_http_fastdfs_module"
    NGX_ADDON_SRCS="$NGX_ADDON_SRCS $ngx_addon_dir/ngx_http_fastdfs_module.c"
    CORE_INCS="$CORE_INCS /usr/include/fastdfs /usr/include/fastcommon/"
    CORE_LIBS="$CORE_LIBS -lfastcommon -lfdfsclient"
    CFLAGS="$CFLAGS -D_FILE_OFFSET_BITS=64 -DFDFS_OUTPUT_CHUNK_SIZE='256*1024' -DFDFS_MOD_CONF_FILENAME='\"/etc/fdfs/mod_fastdfs.conf\"'"
fi
```

4.拷贝mod_fastdfs.conf

将`fastdfs-nginx-module-1.20/src/mod_fastdfs.conf`拷贝至`/etc/fdfs/`下

```
cp mod_fastdfs.conf /etc/fdfs/
```

5.修改mod_fastdfs.conf

```
vim /etc/fdfs/mod_fastdfs.conf
```

文件内容如下：

```
base_path=/kkb/server/fastdfs/storage
tracker_Server=192.168.10.135:22122 #url中是否包含group名称
url_have_group_name=true    #指定文件存储路径，访问时使用该路径
store_path0=/kkb/server/fastdfs/storage
```

6.拷贝libfdfsclient.so(新版不需要)

将`libfdfsclient.so`拷贝至/usr/lib下

```
cp /usr/lib64/libfdfsclient.so   /usr/lib/
```

7.创建nginx/client目录

```
mkdir -p /var/temp/nginx/client
```

七、安装Nginx（Apache）

Nginx需要每一台storage server都需要安装。

1.下载文件

- 查看GitHub上面最新的nginx release 版本

https://github.com/nginx/nginx/releases

- 下载各个版本的nginx的地址：

http://nginx.org/download/

- 上传nginx压缩包

  ```
  yum install -y lrzsz
  rz
  ```

  通过rz命令上传文件：nginx-1.15.6.tar.gz

2.安装第三方软件

2.1 安装PCRE

PCRE(Perl Compatible Regular Expressions)是一个Perl库，包括 perl 兼容的正则表达式库。Nginx的http模块使用pcre来解析正则表达式，所以需要在linux上安装pcre库。

```
yum install -y pcre-devel
```

> 注：pcre-devel是使用pcre开发的一个二次开发库。Nginx也需要此库。

2.2 安装ZLIB

zlib库提供了很多种压缩和解压缩的方式，Nginx使用zlib对http包的内容进行gzip，所以需要在linux上安装zlib库。

```
yum install -y zlib-devel
```

2.3 安装OPENSSL

OpenSSL 是一个强大的安全套接字层密码库，囊括主要的密码算法、常用的密钥和证书封装管理功能及SSL协议，并提供丰富的应用程序供测试或其它目的使用。

Nginx不仅支持http协议，还支持https（即在ssl协议上传输http），所以需要在linux安装openssl库。

```
yum install -y openssl-devel
```

3.解压缩

```
tar -xf nginx-1.15.6.tar.gz
```

4.执行configure配置

```
cd 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=/kkb/soft/fastdfs-nginx-module-1.20/src
```

> **注意：**
>
> prefix=`/kkb/server/nginx` 中的`/kkb/server/nginx`指的是要安装的nginx的路径
>
> add-module=`/opt/fastdfs-nginx-module/src`中的路径指的是`fastdfs-nginx-module模块`的解压缩路径

5.创建临时目录

上面执行的configure命令，设置了一些配置参数，其中的一些参数指定的目录一定要存在。

```
mkdir /var/temp/nginx -p
```

6.编译安装

```
make && make install
```

如果出现以下错误，请修改fastdfs-nginx-module-1.20/src/config配置文件

```
       /kkb/server/fastdfs-nginx-module-1.20/src/ngx_http_fastdfs_module.c
In file included from /kkb/server/fastdfs-nginx-module-1.20/src/common.c:26:0,
                 from /kkb/server/fastdfs-nginx-module-1.20/src/ngx_http_fastdfs_module.c:6:
/usr/include/fastdfs/fdfs_define.h:15:27: fatal error: common_define.h: No such file or directory
 #include "common_define.h"
                           ^
compilation terminated.
make[1]: *** [objs/addon/src/ngx_http_fastdfs_module.o] Error 1
make[1]: Leaving directory `/kkb/soft/nginx-1.15.6'
make: *** [build] Error 2
```

7.修改nginx.conf

```
vim /kkb/server/nginx/conf/nginx.conf
```

修改内容如下：

```
server {
        listen       80;
        server_name  localhost;
        location /group1/M00/{
                ngx_fastdfs_module;
        }
}
```

> **说明：**
>
> `location /group1/M00/`：以**/group1/M00/**开头的请求，才会正常使用Nginx模块**ngx_fastdfs_module**下载访问图片。

8.启动Nginx

切换到nginx/bin目录

```
./nginx
```

八、合并存储(重点)

1.合并存储简介

在处理海量小文件问题上，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。主要需要面对如下几个问题：

- **元数据管理低效**，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；
- **数据布局低效；**
- **IO访问流程复杂；** 因此一种解决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。

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

> **注：**
>
> FastDFS提供的合并存储功能，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**。

2.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
```

3.合并存储文件命名与文件结构

我们知道向FastDFS上传文件成功时，服务器返回该文件的存取ID叫做fileid，当没有启动合并存储时该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。

3.1 合并存储后fileid的变化

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：**创建时间
>
> **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

3.2 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**，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——|

4.合并存储空闲空间管理

4.1 概述

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。

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。**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）。

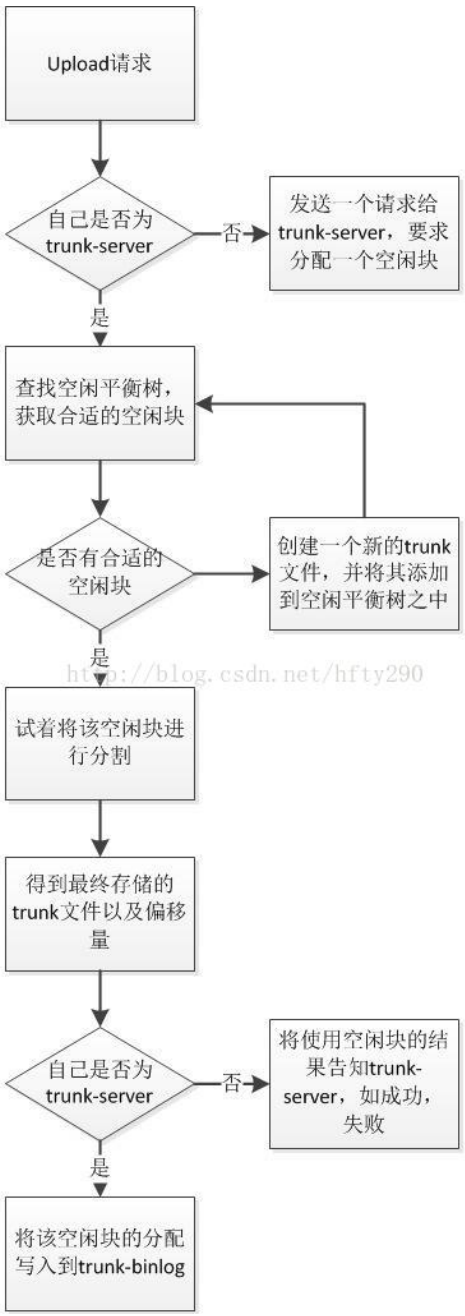
4.2 TrunkServer

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**：Upload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**也接受Upload一个文件B.txt 200KB，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。**

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



4.3 TrunkFile同步

开启了合并存储服务后，除了原本的源文件同步之外，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
```

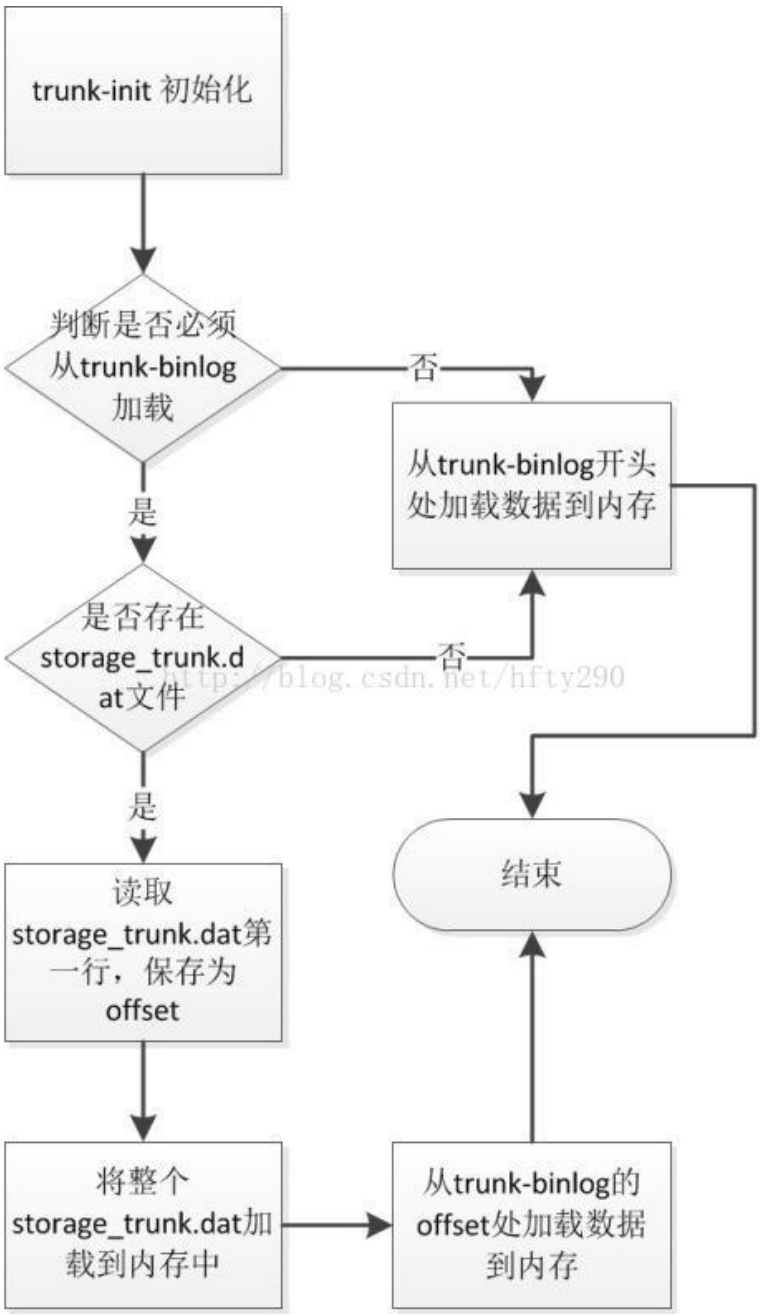
> **各字段含义如下（按照顺序）：**
>
> 时间戳
>
> 操作类型（A：增加，D：删除）
>
> store_path_index
>
> sub_path_high
>
> sub_path_low
>
> file.id（TrunkFile文件名，比如000001）
>
> offset（在TrunkFile文件中的偏移量）
>
> size（占用的大小，按照slot对齐）

4.4 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。

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：



4.5 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，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。

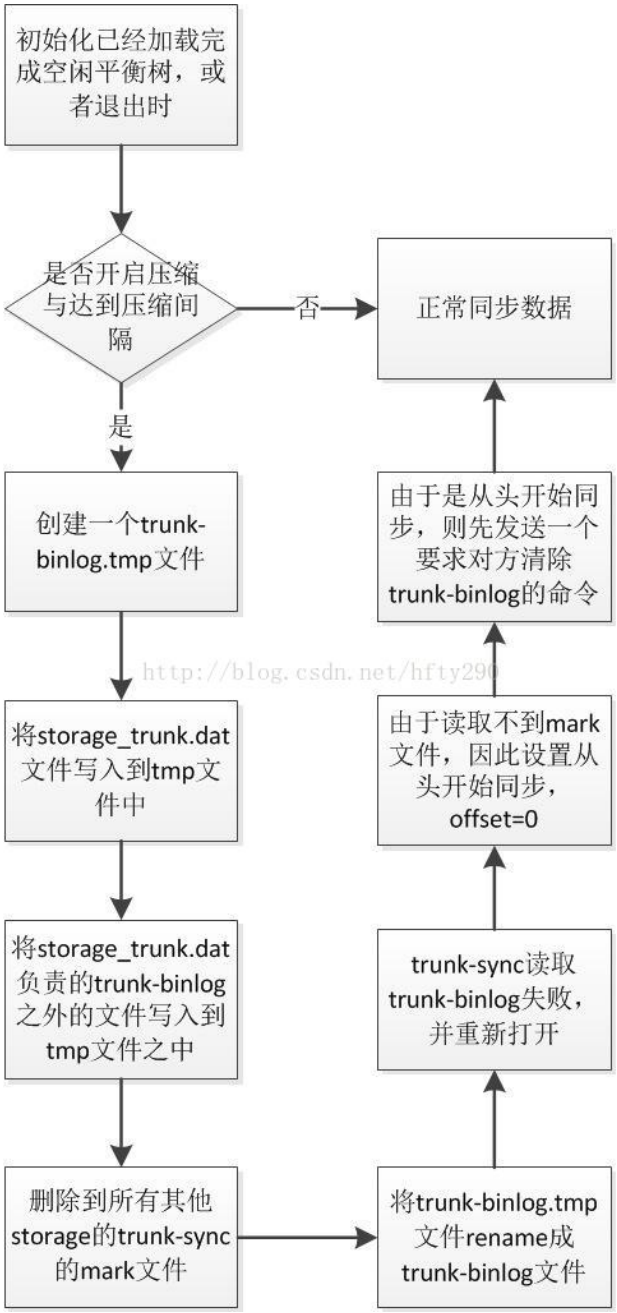
3）当下次TrunkServer要同步TrunkBinlog到B、C时，发现找不到B.mark、C.mark文件，就会自动从头转换成从头开始同步文件。

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：



5.Tracker-Leader选择TrunkServer

5.1 描述

在一个FastDFS集群之中，在开启合并存储时，为了分配空间引入了一个TrunkServer角色，**该TrunkServer是该Group中的一个Storage**，只是该Storage要**负责**为该组内的所有Upload操作**分配空间**。为了避免不同Tracker为该Group选择了不同的TrunkServer，此时引入了Tracker-Leader角色，**也就是TrunkServer最终是由Tracker-Leader来选择的，然后通知给该组内的所有Storage**。

5.2 Tracker -Leader选择TrunkServer时机

1. 当组内的一个Storage状态变成Active时，并且该组还没有指定TrunkServer，在`tracker_mem_active_store_server`函数中触发。
2. 某个Tracker在经过选择，被设置成Leader时，则为当前还没有指定TrunkServer的组选择TrunkServer，在`relationship_select_leader`函数中触发。
3. 在定期指定的任务`tracker_mem_check_alive`函数中，默认该函数100秒指定一次。
    1. 会尝试着为每个当前还没有指定TrunkServer的组选择TrunkServer。
    2. 对已经指定的组检查其TrunkServer是否还处于活动状态（根据TrunkServer与Tracker的最后心跳时间计算），若不处于活动状态，则会尝试着给该组选择一个新的TrunkServer。

5.3 Tracker-Leader选择TrunkServer的过程

该过程由`tracker_mem_find_trunk_server`函数负责，具体操作步骤如下：

1. 依次向组内当前状态为ACTIVE的Storage发送`TRUNK_GET_BINLOG_SIZE`命令的消息，来查询每个Storage当前保存的Trunk-Binlog的文件大小。来`找到Trunk-Binlog文件最大的Storage`。
2. 若该Group的最后一个TrunkServer与要设置的新的TrunkServer并非同一个，则向该新的TrunkServer发送`TRUNK_DELETE_BINLOG_MARKS`命令，让其删除trunk-binlog同步的状态mark文件。（既然这个TrunkServer是新的，那么就要清除同步trunk-binlog的状态，使其从头同步trunk-binlog给组内的其他Storage）。
3. 变更该组的TrunkServer，并将修改写入到`storage_groups_new.dat`文件之中，更新该组的最后TrunkServer设置，设置TrunkServer已经变更标志，该标志使得在与Storage的心跳中通知对方TrunkServer已经变更。

九、存储缩略图

1.FastDFS主从文件

应用背景

使用FastDFS存储一个图片的多个分辨率的备份时，**希望只记录源图的fileid**，并能将其它分辨率的图片与源图关联。可以使用从文件方法。

解决办法

前提：本地先要生成对应的缩略图，然后再使用FastDFS的主从文件完成存储。

名词注解：`主从文件是指文件ID有关联的文件`，一个主文件可以对应多个从文件。

- 主文件ID = 主文件名 + 主文件扩展名
- 从文件ID = 主文件名 + **从文件后缀名（比如_200*200）** + 从文件扩展名

以本场景为例：主文件为原始图片，从文件为该图片的一张或多张缩略图。

流程说明：

1. 先上传主文件（即：原文件），得到**主文件FID**

2. 然后上传从文件（即：缩略图），指定主文件FID和从文件后缀名，上传后得到从文件FID。

Java代码：

```
public static String uploadFile(String filePath) throws Exception {
        String fileId = "";
        String fileExtName = "";
        if (filePath.contains(".")) {
            fileExtName = filePath.substring(filePath.lastIndexOf(".") + 1);
        } else {
            logger.warn("上传失败，无效的扩展名");
            return fileId;
        }

        try {
            fileId = client.upload_file1(filePath, fileExtName, null);
        } catch (Exception e) {
            logger.warn("Upload file \"" + filePath + "\"fails");
        } finally {
            trackerServer.close();
        }
        return fileId;
    }

    /**
     *
     * @param masterFileId      FastDFS服务器返回的主文件的fileid
     * @param prefixName        从文件后缀名（比如_200_200）
     * @param slaveFilePath     从文件所在路径（主从文件在本地都需要有对应的文件）
     * @return
     * @throws Exception
     */
    public static String uploadSlaveFile(String masterFileId, String prefixName,
String slaveFilePath) throws Exception {
        String slaveFileId = "";
        String slaveFileExtName = "";
        if (slaveFilePath.contains(".")) {
            slaveFileExtName =
                slaveFilePath.substring(slaveFilePath.lastIndexOf(".") + 1);
        } else {
            logger.warn("上传失败，无效的扩展名");
            return slaveFileId;
        }
        try {
            slaveFileId = client.upload_file1(masterFileId, prefixName,
                                              slaveFilePath, slaveFileExtName, null);
        } catch (Exception e) {
            logger.warn("Upload file \"" + slaveFilePath + "\"fails");
        } finally {
            trackerServer.close();
        }
        return slaveFileId;
    }
```

```
49
50      public static int download(String fileId, String localFile) throws Exception {
51          int result = 0;
52          // 建立连接
53          TrackerClient tracker = new TrackerClient();
54          TrackerServer trackerServer = tracker.getConnection();
55          StorageServer storageServer = null;
56          StorageClient1 client = new StorageClient1(trackerServer, storageServer);
57          // 上传文件
58          try {
59              result = client.download_file1(fileId, localFile);
60          } catch (Exception e) {
61              logger.warn("Download file \"" + localFile + "\"fails");
62          } finally {
63              trackerServer.close();
64          }
65          return result;
66      }
67
68      public static void main(String[] args) {
69          try {
70              String masterFileId = uploadFile("E:/1.jpg");
71              System.out.println("主文件:" + masterFileId);
72              //下载上传成功的主文件
73              download(masterFileId, "E:/master.jpg");
74              //第三个参数：待上传的从文件（由此可知，必须先本地生成从文件，再上传）
75              String slaveFileId = uploadSlaveFile(masterFileId, "_120x120",
    "E:/2.jpg");
76              System.out.println("从文件:" + slaveFileId);
77              //下载上传成功的从文件
78              download(slaveFileId, "E:/slave.jpg");
79          } catch (Exception e) {
80              logger.error("upload file to FastDFS failed.", e);
81          }
82      }
```

2.Nginx生成缩略图

2.1 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）。

2.1.1 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
[root@localhost img]#
```

2.1.2 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 \
--with-http_stub_status_module \
--with-http_ssl_module \
--with-http_realip_module \
--with-http_image_filter_module


make && make install
```

2.1.3 访问普通图片

需求：

假设我们图片的真实路径是在本地`/kkb/data/img/kkb.jpg`，下面有很多jpg格式的图片，我们希望通过访问`http://ip:port/img/kkb_100x100.jpg`这样的请求路径可以生成宽为100，高也为100的小图，并且请求的宽和高是可变的，那么这时候需要在nginx模块中拦截该请求并返回转换后的小图，在对应的server {}段中进行配置。

nginx.conf配置如下：

```
      location ~* /img/(.*)_(\d+)x(\d+)\.(jpg|gif|png)$ {
            root /;
            set $s $1;
            set $w $2;
            set $h $3;
            set $t $4;
            image_filter resize $w $h;
            image_filter_buffer 10M;
            rewrite ^/img/(.*)$ /kkb/data/img/$s.$t break;
        }
```

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，参数能够包括变量，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415，参数值可包括变量，能够与rotate一起使用，则两个一起生效。


image_filter crop width height;
#按比例降低图像比较大的侧面积和还有一侧多余的载翦边缘，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。参数值能够包括变量。
```

```
35
36
37  image_filter_sharpen 100;
38  #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。参数值能够包括变量。
39
40
41  image_filter_transparency on;
42  #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。
```

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

```
1  /kkb/server/nginx/sbin/nginx -s stop
2  /kkb/server/nginx/sbin/nginx
```

2.1.4 访问FastDFS图片

```
1  location ~ group1/M00/(.+)_(\d+)x(\d+)\.(jpg|gif|png) {
2              # 设置别名（类似于root的用法）
3              alias /kkb/server/fastdfs/storage/data/;
4              # fastdfs中的ngx_fastdfs_module模块
5              ngx_fastdfs_module;
6              set $w $2;
7              set $h $3;
8
9              if ($w != "0") {
10                 rewrite group1/M00(.+)_(\d+)x(\d+)\.(jpg|gif|png)$ group1/M00$1.$4
   break;
11             }
12
13             if ($h != "0") {
14                 rewrite group1/M00(.+)_(\d+)x(\d+)\.(jpg|gif|png)$ group1/M00$1.$4
   break;
15             }
16
17             #根据给定的长宽生成缩略图
18             image_filter resize $w $h;
19
20             #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
21             image_filter_buffer 2M;
22
23             #try_files group1/M00$1.$4 $1.jpg;
24         }
```

重新加载Nginx配置文件

```
/kkb/server/nginx/sbin/nginx -s reload
```

2.2 Nginx Image 缩略图 模块

- 本nginx模块主要功能是对请求的图片进行缩略/水印处理，支持文字水印和图片水印。
- 支持自定义字体，文字大小，水印透明度，水印位置。
- 判断原图是否是否大于指定尺寸才处理。 ....等等
- 支持 jpeg / png / gif (Gif生成后变成静态图片)

2.2.1 安装nginx image模块

- 编译nginx前请确认您的系统已经安装了libcurl-dev libgd2-dev libpcre-dev 依赖库

```
yum install -y gd-devel pcre-devel libcurl-devel
```

- 下载nginx (略)
- 下载nginx image模块，并解压缩

```
wget https://github.com/oupula/ngx_image_thumb/archive/master.tar.gz
tar -xf master.tar.gz
```

- 添加nginx image模块

```
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 \
--with-http_stub_status_module \
--with-http_ssl_module \
--add-module=/kkb/soft/ngx_image_thumb-master \
--add-module=/kkb/soft/fastdfs-nginx-module-1.20/src

make && make install
```

2.2.2 访问普通图片

修改nginx.conf配置文件

```
        location /img/ {
            root /kkb/data/;
            # 开启压缩功能
            image on;
            # 是否不生成图片而直接处理后输出
            image_output on;

            image_water on;
            # 水印类型：0为图片水印，1位文字水印
            image_water_type 0;
            # 水印出现位置
            image_water_pos 9;
            # 水印透明度
            image_water_transparent 80;
            # 水印文件
            image_water_file "/kkb/data/logo.png";
        }
```

访问方式

- nginx服务器地址： http://192.168.10.136
- 访问的源文件地址：http://192.168.10.136/img/1.jpg
- 访问的压缩文件地址：http://192.168.10.136/img/1.jpg`!c300x200.jpg`

  其中 `c` *是生成图片缩略图的参数，* `300` *是生成缩略图的 宽度，* `200` *是生成缩略图的 高度*

参数说明：

一共可以生成四种不同类型的缩略图：

```
C 参数按请求宽高比例从图片高度 10% 处开始截取图片，然后缩放/放大到指定尺寸（ 图片缩略图大小等于请求的宽高 ）

M 参数按请求宽高比例居中截图图片，然后缩放/放大到指定尺寸（ 图片缩略图大小等于请求的宽高 ）

T 参数按请求宽高比例按比例缩放/放大到指定尺寸（ 图片缩略图大小可能小于请求的宽高 ）

W 参数按请求宽高比例缩放/放大到指定尺寸，空白处填充白色背景颜色（ 图片缩略图大小等于请求的宽高 ）
```

2.2.3 访问FastDFS图片

```
        location /group1/M00/{
            alias /kkb/server/fastdfs/storage/data/;

            image on;
            image_output on;
```

```
            image_jpeg_quality 75;

            image_water on;
            image_water_type 0;
            image_water_pos 9;
            image_water_transparent 80;
            image_water_file "/kkb/data/logo.png";

            # image_backend off;
            #配置一个不存在的图片地址，防止查看缩略图时图片不存在，服务器响应慢
            # image_backend_server http://www.baidu.com/img/baidu_jgylogo3.gif;
        }
```

2.2.4 配置参数说明

```
image on/off 是否开启缩略图功能,默认关闭

image_backend on/off 是否开启镜像服务，当开启该功能时，请求目录不存在的图片（判断原图），将自动从镜像服务器地址下载原图

image_backend_server 镜像服务器地址

image_output on/off 是否不生成图片而直接处理后输出 默认off

image_jpeg_quality 75 生成JPEG图片的质量 默认值75

image_water on/off 是否开启水印功能

image_water_type 0/1 水印类型 0:图片水印 1:文字水印

image_water_min 300 300 图片宽度 300 高度 300 的情况才添加水印

image_water_pos 0-9 水印位置 默认值9 0为随机位置,1为顶端居左,2为顶端居中,3为顶端居右,4为中部居左,5为中部居中,6为中部居右,7为底端居左,8为底端居中,9为底端居右

image_water_file 水印文件(jpg/png/gif),绝对路径或者相对路径的水印图片

image_water_transparent 水印透明度,默认20

image_water_text 水印文字 "Power By Vampire"

image_water_font_size 水印大小 默认 5

image_water_font 文字水印字体文件路径

image_water_color 水印文字颜色,默认 #000000
```

十、Java客户端

将jar包install到maven仓库

Maven项目的**第三方依赖**（jar）如果中央仓库没有，怎么办呢？

- 搭建本地私服（nexus），通过私服的操作界面将jar包上传到私服，并产生一个GAV坐标。
- 直接通过以下命令，将jar包上传到本地仓库，同时也是需要指定GAV坐标的。

如果通过eclipse来执行以下命令的时候，是不需要加上`mvn`这个命令的。

```
mvn install:install-file -DgroupId=org.csource.fastdfs -DartifactId=fastdfs  -Dversion=1.25 -Dpackaging=jar -Dfile=E:\01-java-soft\FastDFS\fastdfs_client-1.25.jar
```

配置文件fdfs_client.conf

该配置文件要放到fdfs子目录下，名字叫fdfs_client.conf

```
tracker_server=192.168.10.135:22122
```

编写测试代码

```
public class FastDFSClient {
    private static Logger logger = Logger.getLogger(FastDFSClient.class);

    private static TrackerClient trackerClient = null;
    private static TrackerServer trackerServer = null;
    private static StorageServer storageServer = null;
    private static StorageClient1 client = null;
    // fdfsclient的配置文件的路径
    private static String CONF_NAME = "/fdfs/fdfs_client.conf";

    static {
        try {
            // 配置文件必须指定全路径
            String confName = FastDFSClient.class.getResource(CONF_NAME).getPath();
            // 配置文件全路径中如果有中文，需要进行utf8转码
            confName = URLDecoder.decode(confName, "utf8");

            ClientGlobal.init(confName);
            trackerClient = new TrackerClient();
            trackerServer = trackerClient.getConnection();
            storageServer = null;
            client = new StorageClient1(trackerServer, storageServer);
        } catch (Exception e) {
```

```
            e.printStackTrace();
        }
    }
    /**
     * 上传文件方法
     * <p>
     * Title: uploadFile
     * </p>
     * <p>
     * Description:
     * </p>
     *
     * @param fileName
     *            文件全路径
     * @param extName
     *            文件扩展名，不包含（.）
     * @param metas
     *            文件扩展信息
     * @return
     * @throws Exception
     */
    public static String uploadFile(String fileName, String extName, NameValuePair[]
metas) throws Exception {
        String result = client.upload_file1(fileName, extName, metas);
        return result;
    }

    public static void main(String[] args) {
        try {
            uploadFile("E:/1.jpg","jpg",null);
        } catch (Exception e) {
            logger.error("upload file to FastDFS failed.", e);
        }
    }
```

十一、查看错误日志

- tracker日志

```
tail -100f /kkb/server/fastdfs/tracker/logs/tracker.log
```

其中`/kkb/server/fastdfs/tracker`是在`/etc/fdfs/tracker.conf`中配置的base_path路径

- storage日志

```
tail -100f /kkb/server/fastdfs/storage/logs/storage.log
```

其中`/kkb/server/fastdfs/storage`是在`/etc/fdfs/storage.conf`中配置的base_path路径

- nginx日志

```
tail -100f /var/log/nginx/error.log
```

`/var/log/nginx/error.log` 是通过 `configure脚本` 指定的，或者默认在 `nginx/conf` 目录下。

Nginx附加资料

location配置详解

语法规则： `location [=|~|~*|^~] /uri/ { … }`

- = 开头表示精确匹配
- ^~ 开头表示uri以某个常规字符串开头，理解为匹配 url路径即可。nginx不对url做编码，因此请求为/static/20%/aa，可以被规则^~ /static/ /aa匹配到（注意是空格）。
- **~ 开头表示区分大小写的正则匹配**
- **~* 开头表示不区分大小写的正则匹配**
- **!~和!~*分别为区分大小写不匹配及不区分大小写不匹配 的正则**
- / 通用匹配，任何请求都会匹配到。

多个location配置的情况下匹配顺序：

> 首先匹配 =，其次匹配^~, 其次是按文件中顺序的正则匹配，最后是交给 / 通用匹配。当有匹配成功时候，停止匹配，按当前匹配规则处理请求。

例子，有如下匹配规则：

```
location = / {
   #规则A
}
location = /login {
   #规则B
}
location ^~ /static/ {
   #规则C
}
location ~ \.(gif|jpg|png|js|css)$ {
   #规则D
}
location ~* \.png$ {
   #规则E
}
location !~ \.xhtml$ {
   #规则F
}
location !~* \.xhtml$ {
   #规则G
}
```

```
location / {
   #规则H
}
```

那么产生的效果如下：

```
访问根目录/， 比如http://localhost/ 将匹配规则A

访问 http://localhost/login 将匹配规则B，http://localhost/register 则匹配规则H

访问 http://localhost/static/a.html 将匹配规则C

访问 http://localhost/a.gif, http://localhost/b.jpg 将匹配规则D和规则E，但是规则D顺序优先，规则E不起作用，而 http://localhost/static/c.png 则优先匹配到 规则C

访问 http://localhost/a.PNG 则匹配规则E， 而不会匹配规则D，因为规则E不区分大小写。

访问 http://localhost/a.xhtml 不会匹配规则F和规则G，http://localhost/a.XHTML不会匹配规则G，因为不区分大小写。规则F，规则G属于排除法，符合匹配规则但是不会匹配到，所以想想看实际应用中哪里会用到。

访问 http://localhost/category/id/1111 则最终匹配到规则H，因为以上规则都不匹配，这个时候应该是nginx转发请求给后端应用服务器，比如FastCGI（php），tomcat（jsp），nginx作为方向代理服务器存在。
```

所以实际使用中，通常至少有三个匹配规则定义，如下：

```
#直接匹配网站根，通过域名访问网站首页比较频繁，使用这个会加速处理，官网如是说。
#这里是直接转发给后端应用服务器了，也可以是一个静态首页
# 第一个必选规则
location = / {
    proxy_pass http://tomcat:8080/index
}

# 第二个必选规则是处理静态文件请求，这是nginx作为http服务器的强项
# 有两种配置模式，目录匹配或后缀匹配,任选其一或搭配使用
location ^~ /static/ {
    root /webroot/static/;
}
location ~* \.(gif|jpg|jpeg|png|css|js|ico)$ {
    root /webroot/res/;
}

#第三个规则就是通用规则，用来转发动态请求到后端应用服务器
#非静态文件请求就默认是动态请求，自己根据实际把握
#毕竟目前的一些框架的流行，带.php,.jsp后缀的情况很少了
location / {
    proxy_pass http://tomcat:8080/
}
```

rewrite语法

- **语法命令：**

rewrite [flag];

关键字 正则 替代内容 flag标记

- **命令解释：**
  - 关键字：重写语法关键字
  - 正则：perl兼容正则表达式语句进行规则匹配
  - 替代内容：将正则匹配的内容替换成replacement
  - flag标记：rewrite支持的flag标记
- **flag标记说明：**

```
* last  #本条规则匹配完成后，继续向下匹配新的location URI规则

* break  #本条规则匹配完成即终止，不再匹配后面的任何规则

* redirect  #返回302临时重定向，浏览器地址会显示跳转后的URL地址

* permanent  #返回301永久重定向，浏览器地址栏会显示跳转后的URL地址
```