

重点内容：

- FastDFS存储原理
- 上传下载原理分析
- 代码实现文件上传

源码：tc-mini3

## 0 上一节课遗留问题

busy\_ 初始化

```
67 *  
68 CHttpConn::CHttpConn() {  
69     .... busy_ = false; 注意初始化  
70     .... socket_handle_ = NETLIB_INVALID_HANDLE;  
71     .... state_ = CONN_STATE_IDLE;  
72 }  
73     .... conn_handle_ = ++g_conn_handle_generator;  
74     .... if (conn_handle_ == 0) {  
75     ....     .... conn_handle_ = ++g_conn_handle_generator;  
76     .... }  
77 }
```

```
68 *  
69 CHttpConn::CHttpConn() {  
70     .... socket_handle_ = NETLIB_INVALID_HANDLE;  
71     .... state_ = CONN_STATE_IDLE;  
72     .... conn_handle_ = ++g_conn_handle_generator;  
73     .... if (conn_handle_ == 0) {  
74     ....     .... conn_handle_ = ++g_conn_handle_generator;  
75     .... }  
76 }
```

send函数优化

```
85 *  
86 int CHttpConn::Send(void *data, int len) {  
87     .... LogInfo("conn_handle_={}, socket_handle_={}", conn_handle_, socket_handle_);  
88     .... if (busy_) {  
89     ....     .... out_buf_.Write(data, len);  
90     ....     .... return len;  
91     .... }  
92     .... 协议没有写缓存空间了  
93     .... 先存着  
94     .... int ret = netlib_send(socket_handle_, data, len);  
95     .... if (ret < 0) {
```

```
82 *  
83 int CHttpConn::Send(void *data, int len) {  
84     .... LogInfo("conn_handle_={}, socket_handle_={}", conn_handle_, socket_handle_);  
85     ....  
86     .... // 调用 send socket 接口  
87     .... int ret = netlib_send(socket_handle_, data, len);  
88     .... if (ret < 0) {
```

Close的时候需要释放引用计数

```
108 void CHttpConn::Close() {  
109     .... LogInfo("conn_handle_={}, socket_handle_={}", conn_handle_, socket_handle_);  
110     .... state_ = CONN_STATE_CLOSED;  
111     ....  
112     .... g_http_conn_map.erase(conn_handle_);  
113     .... netlib_close(socket_handle_);  
114     ....  
115     .... ReleaseRef();  
116 }  
117 }
```

```
101 void CHttpConn::Close() {  
102     .... LogInfo("conn_handle_={}, socket_handle_={}", conn_handle_, socket_handle_);  
103     .... state_ = CONN_STATE_CLOSED;  
104     ....  
105     .... g_http_conn_map.erase(conn_handle_);  
106     .... netlib_close(socket_handle_);  
107     ....  
108     .... ReleaseRef();  
109 }
```

# 1 海量小文件存储和fileid

---

## 1.1 为什么需要小文件存储

---

### 1.1.1 小文件应用场景

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。如社交网站、电子商务、广电、网络视频、高性能计算，这里举几个典型应用场景。

- 著名的社交网站Facebook存储了600亿张以上的图片，推出了专门针对海量小图片定制优化的Haystack进行存储。
- 淘宝目前应该是最大C2C电子商务网站，存储超过200亿张图片，平均大小仅为15KB，也推出了针对小文件优化的TFS文件系统存储这些图片，并且进行了开源。
- 歌华有线可以进行图书和视频的在线点播，图书每页会扫描成一个几十KB大小的图片，总图片数量能够超过20亿；视频会由切片服务器根据视频码流切割成1MB左右的分片文件，100个频道一个星期的点播量，分片文件数量可达到1000万量级。
- 动漫渲染和影视后期制作应用，会使用大量的视频、音频、图像、纹理等原理素材，一部普通的动画电影可能包含超过500万的小文件，平均大小在10-20KB之间。
- 金融票据影像，需要对大量原始票据进行扫描形成图片和描述信息文件，单个文件大小为几KB至几百KB的不等，文件数量达到数千万乃至数亿，并且逐年增长。

应用范例：vivo **FastDFS 海量小文件存储解决之道**[https://mp.weixin.qq.com/s/Tk\\_H-ofrS5\\_kLwT1DZsxyA](https://mp.weixin.qq.com/s/Tk_H-ofrS5_kLwT1DZsxyA)

### 1.1.2 小文件存储带来的问题

Linux通过inode存储文件信息，但inode也会消耗硬盘空间，所以硬盘格式化的时候，操作系统自动将硬盘分成两个区域。

1. 一个是数据区，存放文件数据；
2. 另一个是inode区（inode table），存放inode所包含的信息。

每个inode节点的大小，一般是128字节或256字节。inode节点的总数，在格式化时就给定，一般是每1KB或每2KB就设置一个inode。假定在一块1GB的硬盘中，每个inode节点的大小为128字节，每1KB就设置一个inode，那么inode table的大小就会达到128MB，占整块硬盘的12.8%。

小文件主要有2个问题：

1. 如果小文件都小于<1KB，而系统初始化的时候每个2K设置一个node，则此时一个文件还是至少占用**2K的空间**，最后导致磁盘空间利用率不高，< 50%。
2. 大量的**小文件**，导致在**增加、查找、删除文件**的时候需要遍历过多的node节点，影响效率。

## 1.2 小文件机制配置

---

合并文件存储相关的配置都在**tracker.conf**中。配置完成后，**重启tracker和storage server**。

支持小文件存储，只需要设置tracker的`use_trunk_file=true`，`store_server=1`，其他保持默认即可，但也要注意`slot_max_size`的大小，这样才知道多大的文件触发小文件存储机制。

```
#是否启用trunk存储，缺省false，需要打开配置

use_trunk_file = true

#trunk文件最小分配单元 字节，缺省256，即使上传的文件只有10字节，也会分配这么多空间。

slot_min_size = 256

#trunk内部存储的最大文件，超过该值会被独立存储，缺省16M，超过这个size的文件，不会存储到trunk file中，而是作为一个单独的文件直接存储到文件系统中，这里设置为1M只是为了更方便测试。

slot_max_size = 1MB

#trunk文件大小，默认 64MB，不要配置得过大或者过小，最好不要超过256MB。

trunk_file_size = 64MB
```

## 1.3 合并存储文件命名与文件

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。

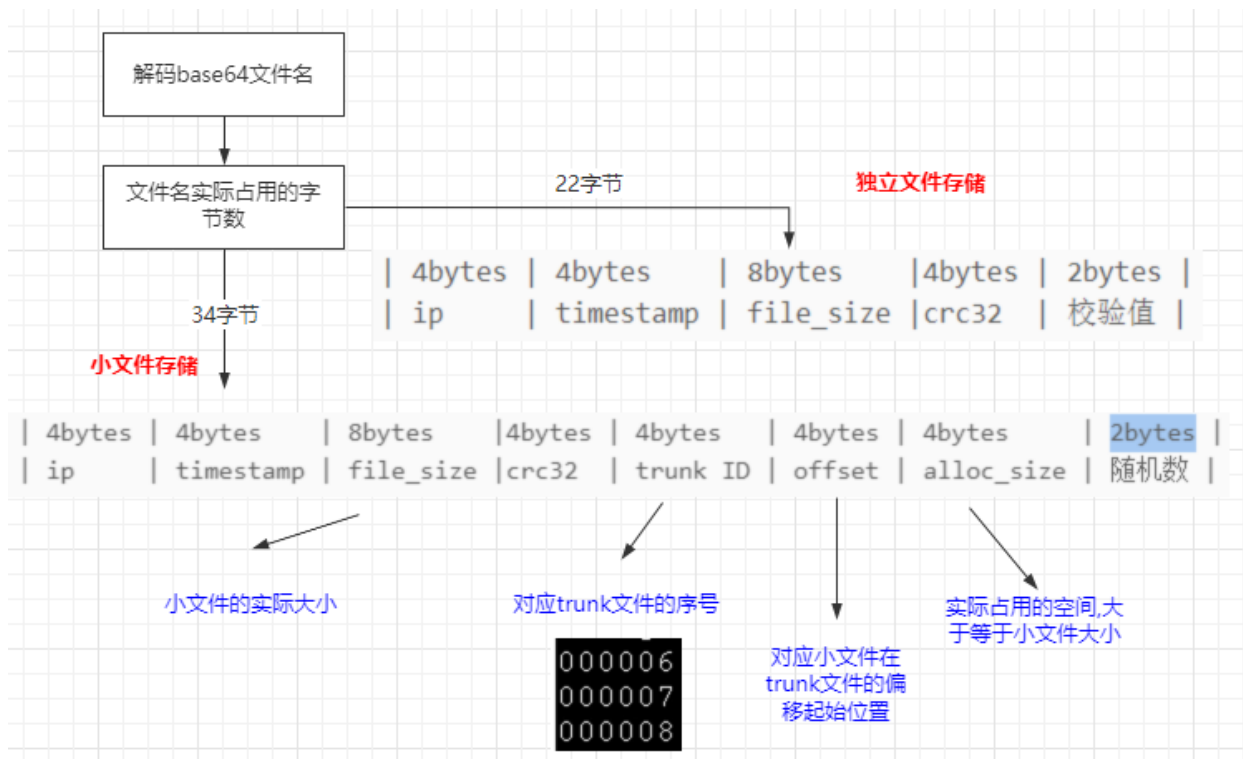
注：下面将采用合并存储后的大文件统称为**Trunk文件**，没有合并存储的文件统称为**源文件**；

请大家注意区分三个概念：

- 1) Trunk文件：storage服务器磁盘上存储的实际文件，默认大小为64MB
- 2) 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；
- 3) 没有合并存储的FileId：表示服务器未启用合并存储时，Upload时返回的FileID

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。

### 1.3.1 启动小文件存储时服务返回给客户端的fileid有变化



## 1. 独立文件存储的file id

文件名（不含后缀名）采用Base64编码，包含如下5个字段（每个字段均为4字节整数）：

group1/M00/00/00/wKgqHV40QQyAbo9YAAAA\_fdSpmg855.txt

这个文件名中，除了.txt为文件后缀，wKgqHV40QQyAbo9YAAAA\_fdSpmg855 这部分是一个base64编码缓冲区，组成如下：

- storage\_id (ip的数值型) 源storage server ID或IP地址
- 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)
- 随机数 (引入随机数的目的是防止生成重名文件)

```
wKgqHV40QQyAbo9YAAAA_fdSpmg855
| 4bytes | 4bytes | 8bytes | 4bytes | 2bytes |
| ip | timestamp | file_size | crc32 | 随机数 |
```

## 2. 小文件存储的file id

如果采用了合并存储，生成的文件ID将变长，文件名后面多了base64文本长度16字符（12个字节）。这部分同样采用Base64编码，包含如下几个字段：

group1/M00/00/00/eBuDxWCwrDqITi98AAAA-3Qtcs8AAAAQAAAgAAAAIA833.txt

采用合并的文件ID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：

- storage\_id (ip的数值型) 源storage server ID或IP地址
- timestamp (文件创建时间戳)
- file\_size: 实际的文件大小
- crc32: 文件内容的crc32码
- trunk file ID: 大文件ID如000001
- offset: 文件内容在**trunk文件中的偏移量**
- alloc\_size: 分配空间, 大于或等于文件大小
- 随机数 (引入随机数的目的是防止生成重名文件)

```
eBuDxWCwrDqITi98AAAA-3Qtcs8AAAAQAAAgAAAAIA833
| 4bytes | 4bytes   | 8bytes   | 4bytes | 4bytes   | 4bytes | 4bytes   | 2bytes |
| ip      | timestamp | file_size | crc32  | trunk ID | offset | alloc_size | 随机数 |
```

### 3.小文件测试文件测试

1. 修完tracker.conf的配置use\_trunk\_file=true, 然后要重启tracker server和storage server

```
# 先重启tracker server
sudo /etc/init.d/fdfs_trackerd restart
# 再重启storage server
sudo /etc/init.d/fdfs_storaged restart
```

2. 使用truncate命令创建测试文件

```
创建一个10M文件
truncate -s 10M testfile10M.bin

创建一个512K文件
truncate -s 512K testfile512K.bin

ll testfile*
看到创建的测试文件
-rw-rw-r-- 1 lqf lqf 10485760 Jul 27 15:43 testfile10M.bin
-rw-rw-r-- 1 lqf lqf 524288 Jul 27 15:43 testfile512K.bin
```

3. 上传测试

```
# 上传10M的文件, 会独立存储
/usr/bin/fdfs_upload_file /etc/fdfs/client.conf ./testfile10M.bin
返回:
group1/M00/00/00/wKgBG2akp02AT-SCAKAAAJ7KKsw455.bin
```

```
# 上传512K的文件，当作小文件存储
/usr/bin/fdfs_upload_file /etc/fdfs/client.conf ./testfile512k.bin
返回：
group1/M00/00/00/wKgBG2akp0-IE7pCAAgAAHvmCqwAAAAAQIAQAACAEA765.bin

#查看存储目录，可以看到trunk文件
lqf@ubuntu:~$ ls -al /home/fastdfs/storage/data/00/00/
total 90712
drwxr-xr-x  2 root root    4096 Jul 27 15:52 .
drwxr-xr-x 258 root root    4096 Jul 22 21:13 ..
-rw-r--r--  1 root root 67108864 Jul 27 15:52 000001  这个是trunk文件，大小为64M
....
-rw-r--r--  1 root root 10485760 Jul 27 15:52 wKgBG2akp02AT-SCAKAAAJ7KKsw455.bin 这个
是上传的10M文件
```

## 1.3.2 Trunk文件存储结构

### 1.3.2.1 磁盘数据内部结构

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 -----|
```

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。

**即是：每次创建一个trunk文件时，是安装既定的文件大小去创建该文件trunk\_file\_size，当要删除trunk文件时，需要该trunk文件没有存储小文件才能删除。**

### 1.3.2.2 小文件存储平衡树

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）。
- **进一步阅读：** storage\trunk\_mgr\trunk\_mem.c 搜索 avl\_tree\_insert、avl\_tree\_find等函数。

## 2 fastdfs交互协议格式

FastDFS采用二进制TCP通信协议。一个数据包由 包头（header）和包体（body）组成。包头只有10个字节，格式如下：

@ pkg\_len: 8字节整数，body长度，不包含header，只是body的长度

@ cmd: 1字节整数，命令码

@ status: 1字节整数，状态码，0表示成功，非0失败（UNIX错误码）

```
tracker\tracker_proto.h TrackerHeader
#define FDFS_PROTO_PKG_LEN_SIZE      8 //8字节

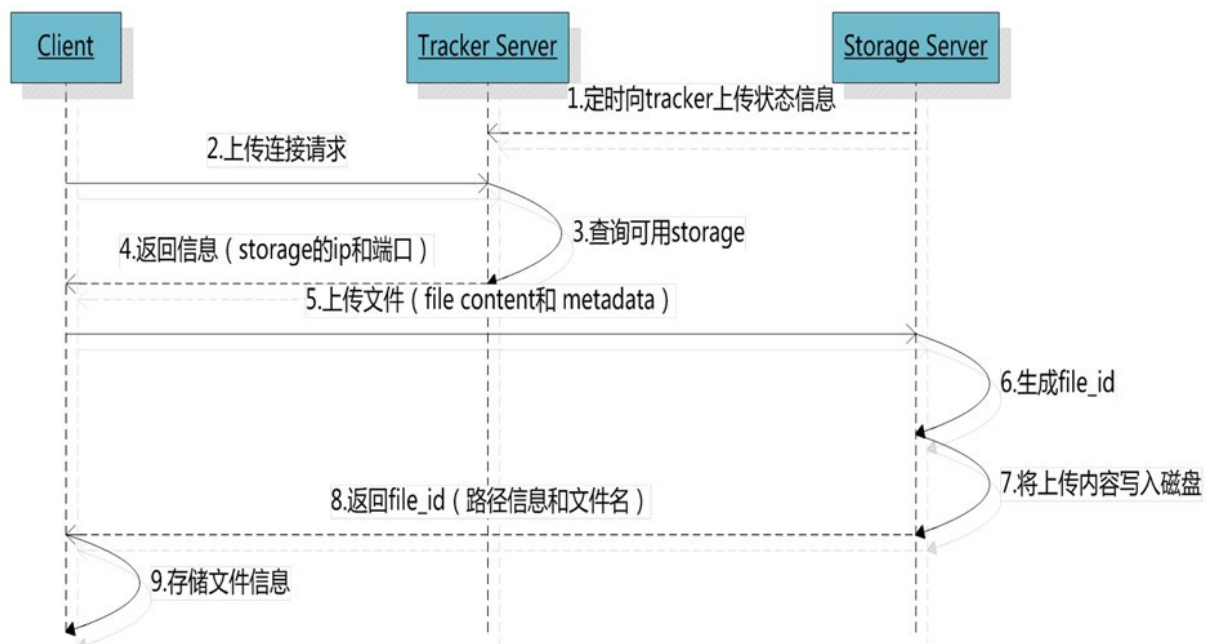
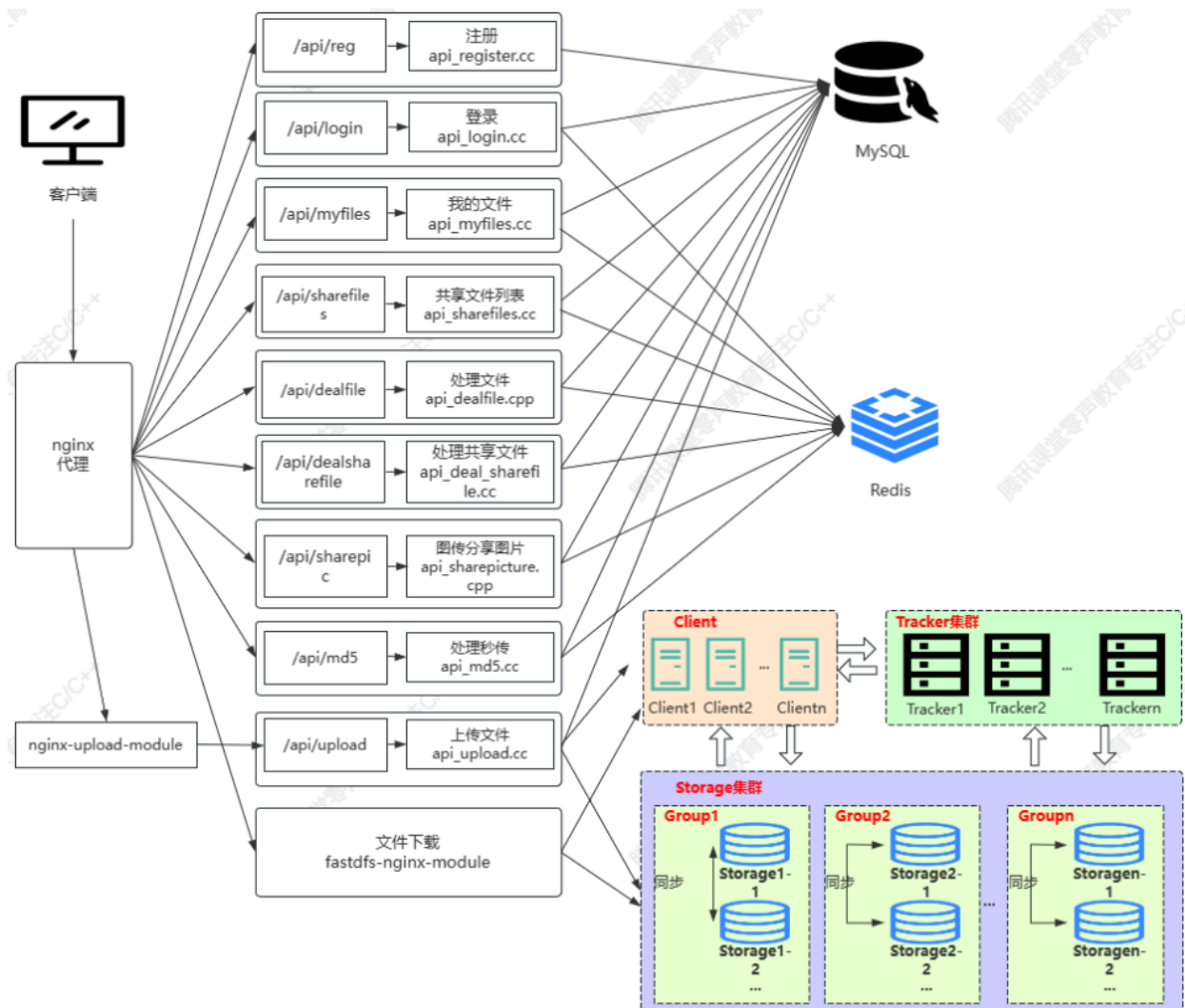
typedef fstruct
{
    char pkg_len[FDFS_PROTO_PKG_LEN_SIZE]; // body长度，不包括header
    FDFS_PROTO_PKG_LEN_SIZE 8
    char cmd; //command 命令
    char status; //status code for response 响应的状态码
} TrackerHeader;
```

即是头部固定10字节，body长度通过pkg\_len给出。

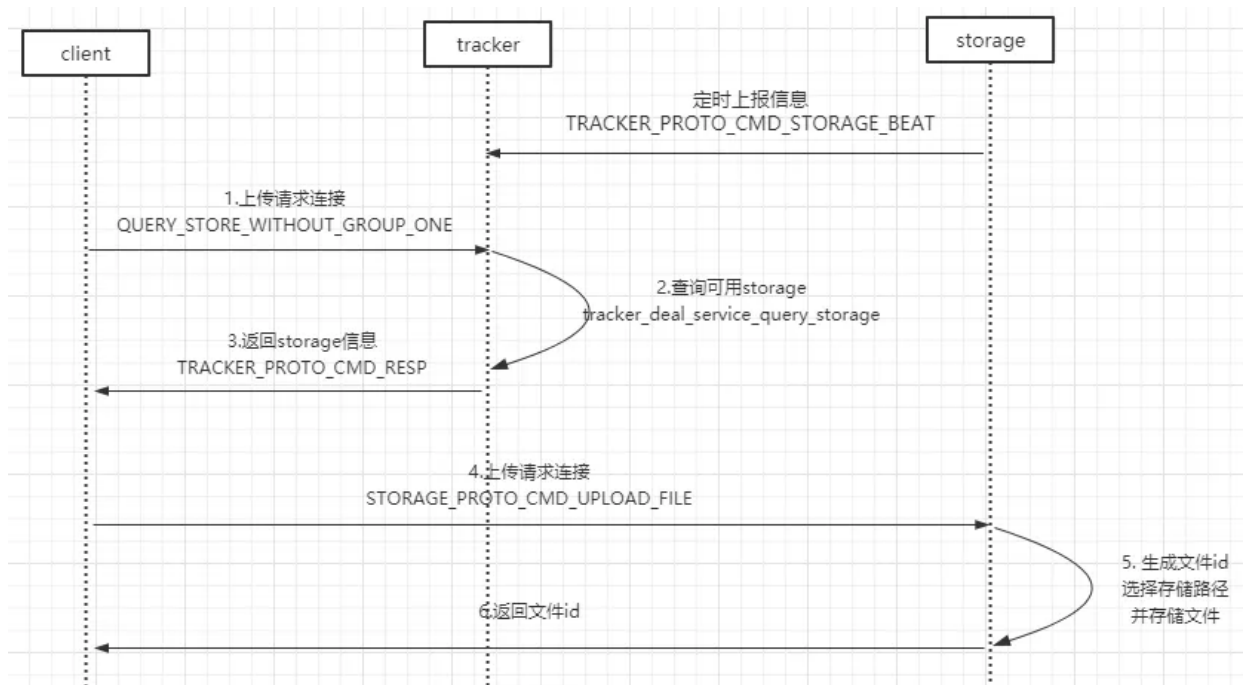
即是一帧完整的协议为 TrackerHeader + body数据（可以为0）。

这里的设计和我们之前在第三专栏讲的应用层协议设计的header + body方式是类似的设计。

## 3 文件上传原理和负载均衡方法







STORAGE\_PROTO\_CMD\_UPLOAD\_FILE: 上传普通文件

# 请求body:

@store\_path\_index: 1字节整数, 基于0的存储路径顺序号

@meta\_data\_length: 8字节整数, meta data (文件附加属性) 长度, 可以为0

@file\_size: 8字节整数, 文件大小

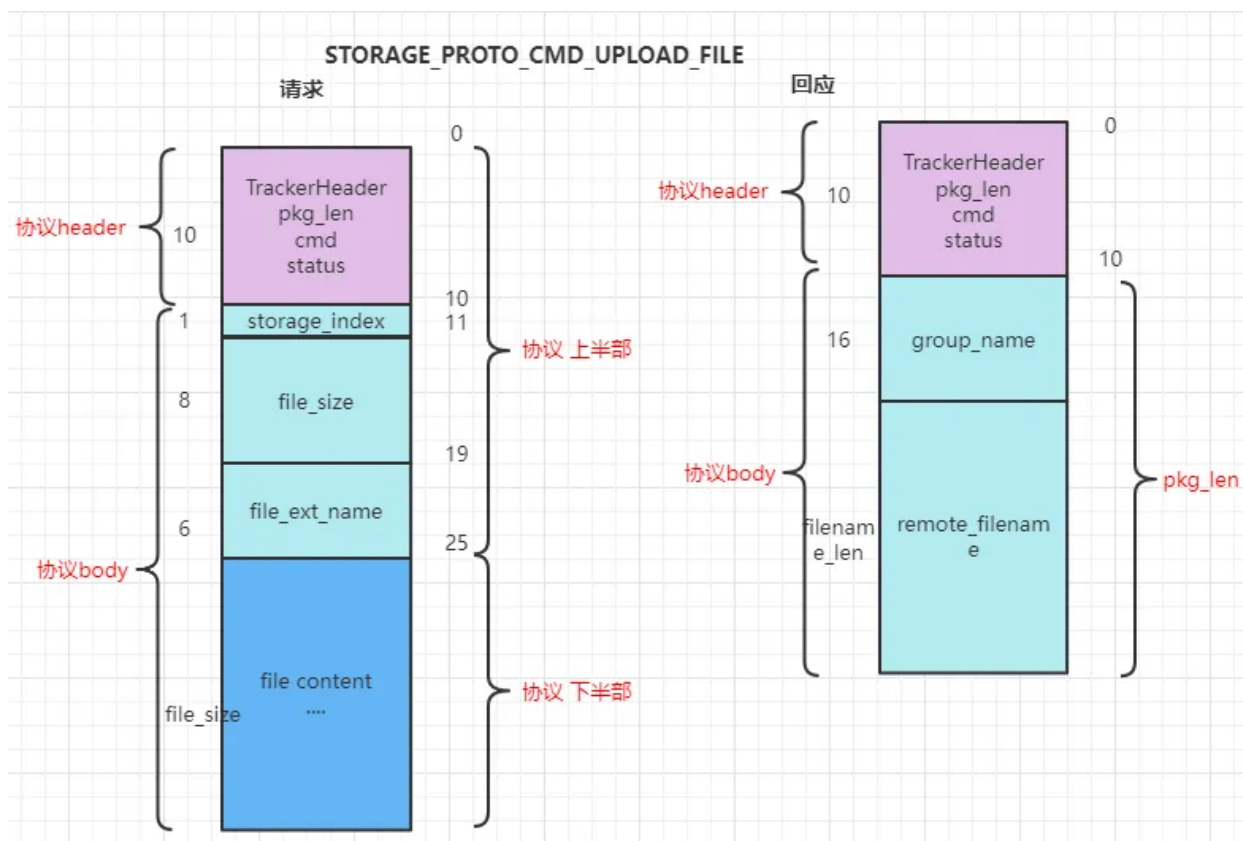
@file\_ext\_name: 6字节字符串, 不包括小数点的文件扩展名, 例如 jpeg、tar.gz

@meta\_data: meta\_data\_length字节字符串, 文件附加属性, 每个属性用字符 \x01分隔, 名称key和取值value之间用字符 \x02分隔

@file content: file\_size字节二进制内容, 文件内容 # 响应body:

@group\_name: 16字节字符串, 组名

@ filename: 不定长字符串, 文件名



### 3.1 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。 --》高可用，通过冗余的方式提供服务。

### 3.2 选择存储的group

(注意不同的负载均衡算法)

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, 选择最大剩余空间的组上传文件

### 3.3 选择storage server

(注意不同的负载均衡算法)

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）

## 3.4 选择storage path

---

(注意不同的负载均衡算法)

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:

1. Round robin, 多个存储目录间轮询
2. 剩余存储空间最多的优先

## 3.5 生成Fileid

---

选定存储目录之后, storage会为文件生一个Fileid, 由:

- storage server ip
- 文件创建时间
- 文件大小
- 文件crc32
- 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。

### 3.5.1 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fileid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。

一个目录存储100百万文件, 单个子目录平均 $100\text{百万} / (256 * 256) = 15.2$ 个文件

### 3.5.2 生成文件名

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、fileid、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。

组名 磁盘 目录 文件名

group1/M00/00/00/eBuDxWCeIFCAEFUrAAAAKTlQHvk462.txt

文件名规则：

- storage\_id (ip的数值型) 源storage server ID或IP地址
- 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)

随机数 (引入随机数的目的是防止生成重名文件)

```
eBuDxWcb2qmAQ89yAAAAKeR1iIo162
| 4bytes | 4bytes | 8bytes | 4bytes | 2bytes |
| ip | timestamp | file_size | crc32 | 校验值 |
```

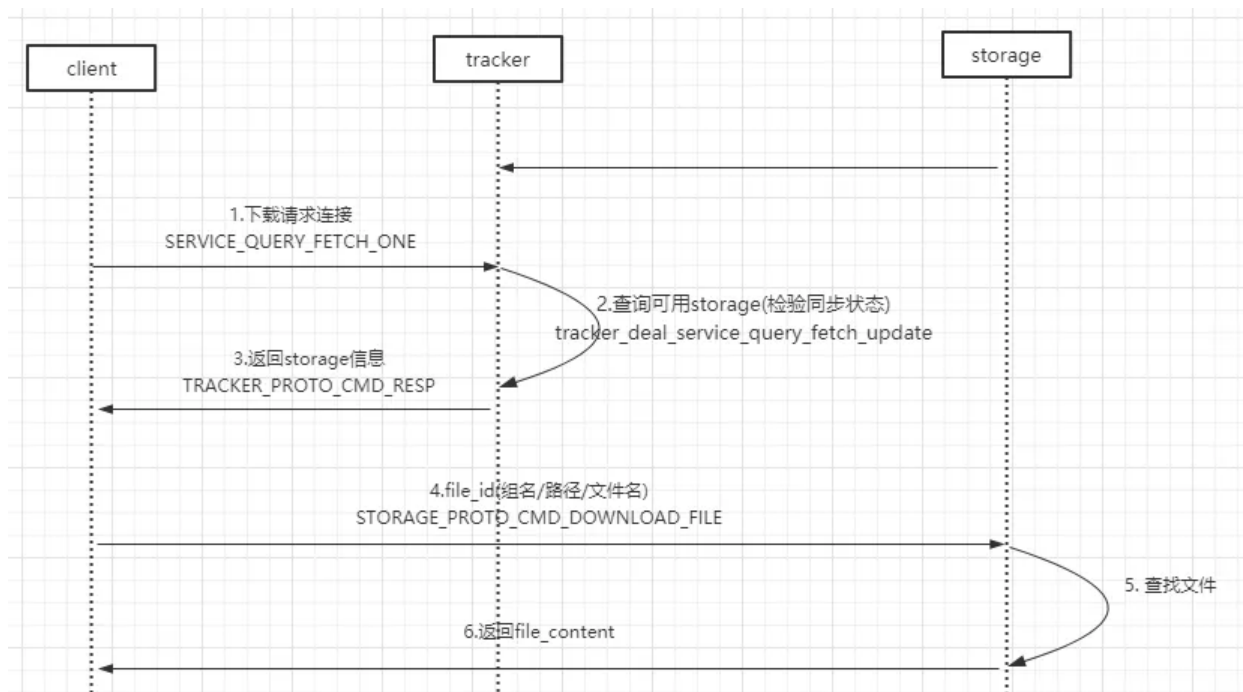
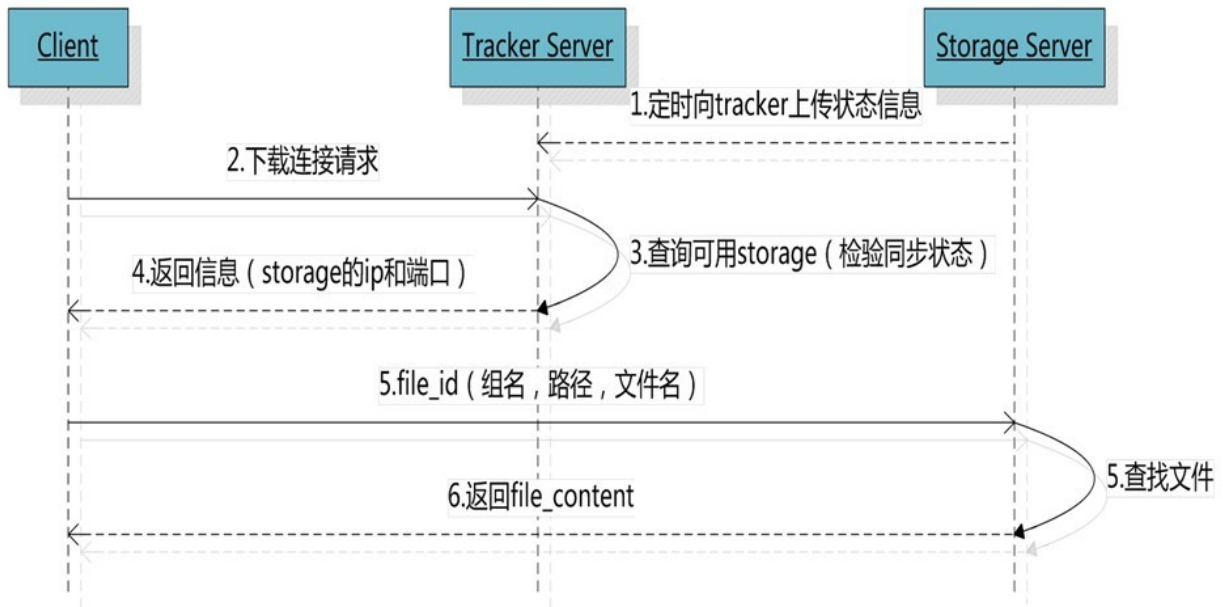
## 4 下载文件逻辑

### 4.1 fastdfs客户端如何下载文件

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。

通过fastdfs客户端下载, 比如:

```
fdfs_download_file /etc/fdfs/client.conf
group1/M00/00/00/ctepQmIWJTCAcldiAAHuj79dAY04.conf
```



STORAGE\_PROTO\_CMD\_DOWNLOAD\_FILE: 下载文件

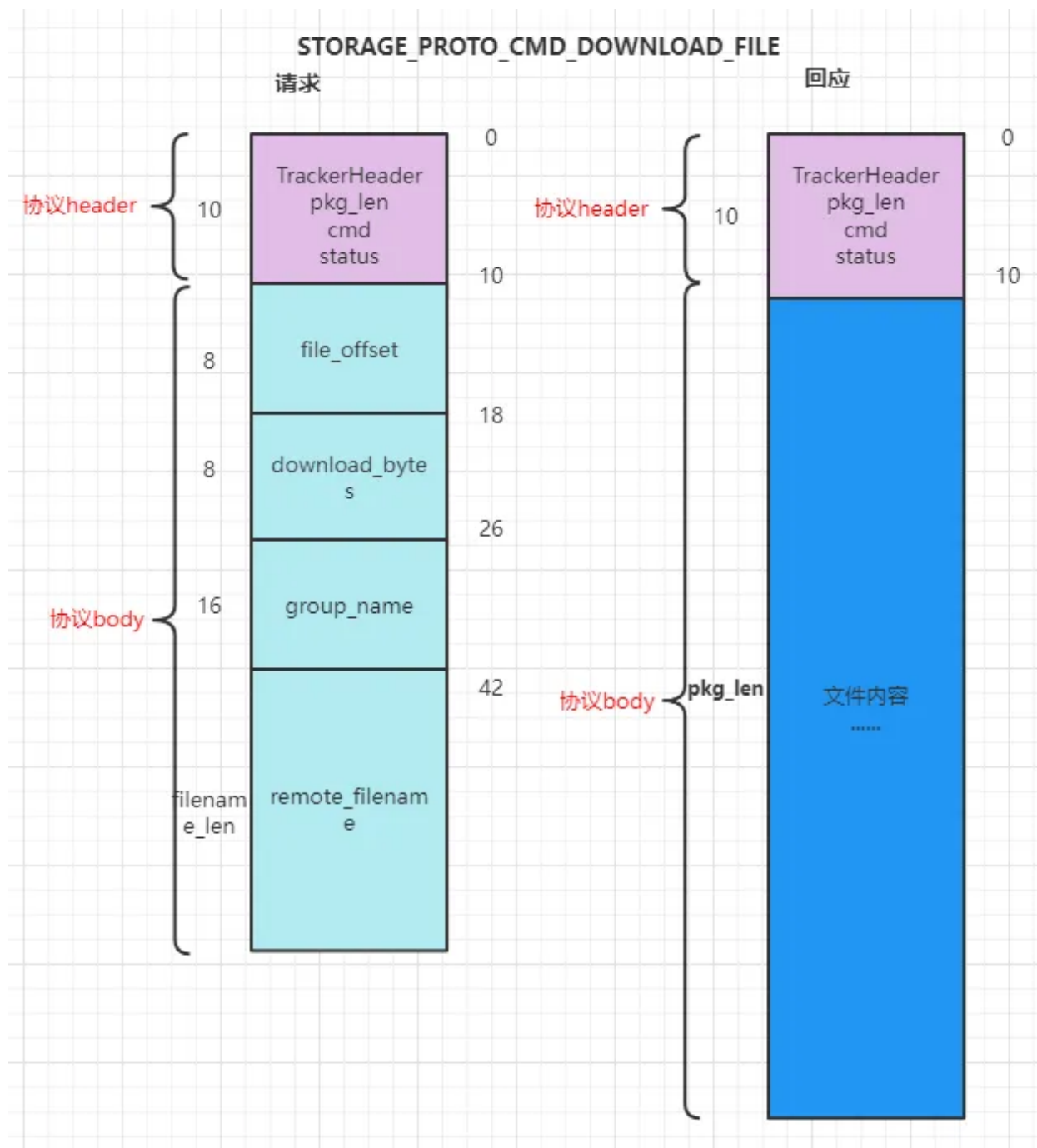
# 请求body:

@file\_offset: 8字节整数, 文件偏移量

@download\_bytes: 8字节整数, 下载字节数

@group name: 16字节字符串, 组名 @filename: 不定长字符串, 文件名

# 响应body: @file\_content: 不定长二进制内容, 文件内容



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

tracker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、大小、创建时间**等信息，然后为该请求选择一个storage用来服务读请求。

由于group内的文件同步时在后台异步进行的，所以有可能出现在读到时候，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（IP）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) > 同步延迟阈值 (如一天)。 经过同步延迟阈值时间, 认为文件肯定已经同步了。

在第五节课讲集群同步的时候进一步讲解。

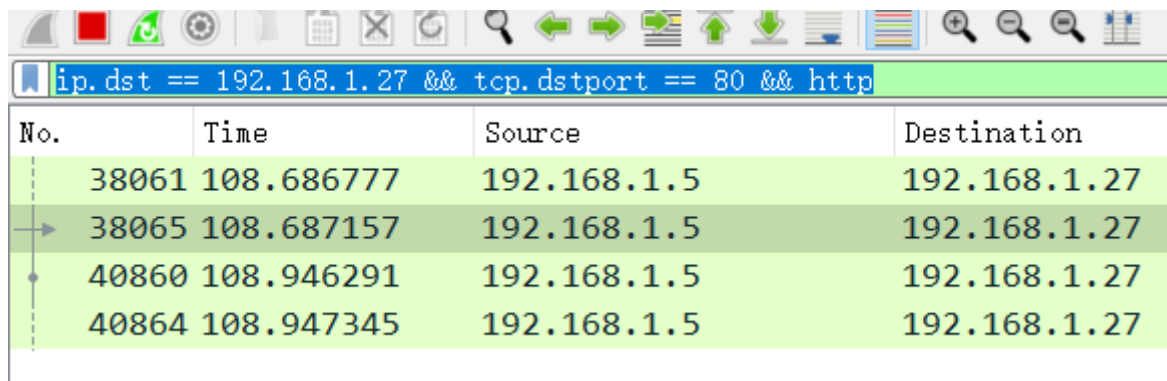
## 4.2 自己写的客户端如何通过fastdfs-nginx-module下载文件

就是http下载原理, 在公开课有讲过, 可以参考下这里的链接, 这个不作为VIP内容, 但有问题可以找老师沟通交流。

# 5 图床文件上传功能实现

## 5.0 wireshark http传输抓包

```
ip.dst == 192.168.1.27 && tcp.dstport == 80 && http
```



The image shows the Wireshark network protocol analyzer interface. At the top, the packet capture filter is set to `ip.dst == 192.168.1.27 && tcp.dstport == 80 && http`. Below the filter, a list of captured packets is displayed with the following columns: No., Time, Source, and Destination.

No.	Time	Source	Destination
38061	108.686777	192.168.1.5	192.168.1.27
38065	108.687157	192.168.1.5	192.168.1.27
40860	108.946291	192.168.1.5	192.168.1.27
40864	108.947345	192.168.1.5	192.168.1.27

```
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/113.0.0.0 Sa
Content-Type: multipart/form-data; boundary=----WebKitFormBoundaryF31lIZUwW40EjPZv\r\n
Origin: http://192.168.1.27\r\n
Referer: http://192.168.1.27\r\n
Accept-Encoding: gzip, deflate\r\n
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8,en-GB;q=0.7,en-US;q=0.6\r\n
✓ Cookie: userName=qingfu; token=nwlrbbmqbhcdarzowkkyhiddqscdxrjm\r\n
  Cookie pair: userName=qingfu
  Cookie pair: token=nwlrbbmqbhcdarzowkkyhiddqscdxrjm
\r\n
[Full request URI: http://192.168.1.27/api/upload]
[HTTP request 2/3]
[Prev request in frame: 85776]
[Response in frame: 85824]
[Next request in frame: 85828]
File Data: 5852 bytes
✓ MIME Multipart Media Encapsulation, Type: multipart/form-data, Boundary: "----WebKitFormBoundaryF31lIZUwW40EjPZv"
  [Type: multipart/form-data]
  First boundary: -----WebKitFormBoundaryF31lIZUwW40EjPZv\r\n
  ✓ Encapsulated multipart part: (text/plain)
    Content-Disposition: form-data; name="file"; filename="20220301.txt"\r\n
    Content-Type: text/plain\r\n\r\n
    > Line-based text data: text/plain (202 lines)
    Boundary: \r\n-----WebKitFormBoundaryF31lIZUwW40EjPZv\r\n
  ✓ Encapsulated multipart part:
    Content-Disposition: form-data; name="user"\r\n\r\n
    > Data (6 bytes)
    Boundary: \r\n-----WebKitFormBoundaryF31lIZUwW40EjPZv\r\n
  ✓ Encapsulated multipart part:
    Content-Disposition: form-data; name="md5"\r\n\r\n
    > Data (32 bytes)
    Boundary: \r\n-----WebKitFormBoundaryF31lIZUwW40EjPZv\r\n
  ✓ Encapsulated multipart part:
    Content-Disposition: form-data; name="size"lala
```

## 5.1 秒传文件api\_md5.cc

用于秒传文件的请求。

上传文件的时候：

- 先调用/api/md5接口判断服务器是否有该文件，如果/api/md5返回成功，则说明服务器已经存在该文件，客户端不需要再去调用/api/upload接口上传文件。
- 如果不成功则客户端继续调用/api/upload接口上传文件。

### /api/md5 请求对比数据库文件信息的md5

#### 请求和应答

请求URL

URL	<a href="http://192.168.1.27/api/md5">http://192.168.1.27/api/md5</a>
请求方式	POST
HTTP版本	1.1
Content-Type	application/json

请求参数



参数名	含义	规则说明	是否必须	缺省值
token	令牌	同上	必填	无
md5	md5值	不能超过32个字符	必填	无
filename	文件名称	不能超过128个字符	必填	无
user	用户名称	不能超过32个字符	必填	无

返回结果参数说明

名称	含义	规则说明
code	结果值	0: 秒传成功 1: 秒传失败4: token校验失败5: 文件已存在

服务示例

调用接口

<http://192.168.1.27/api/md5>

参数

```
{
  "filename": "ui_buttongroup.h",
  "md5": "a89390d867d5da18c8b1a95908d7c653",
  "token": "3a58ca22317e637797f8bcad5c047446",
  "user": "qingfu"
}
```

返回结果

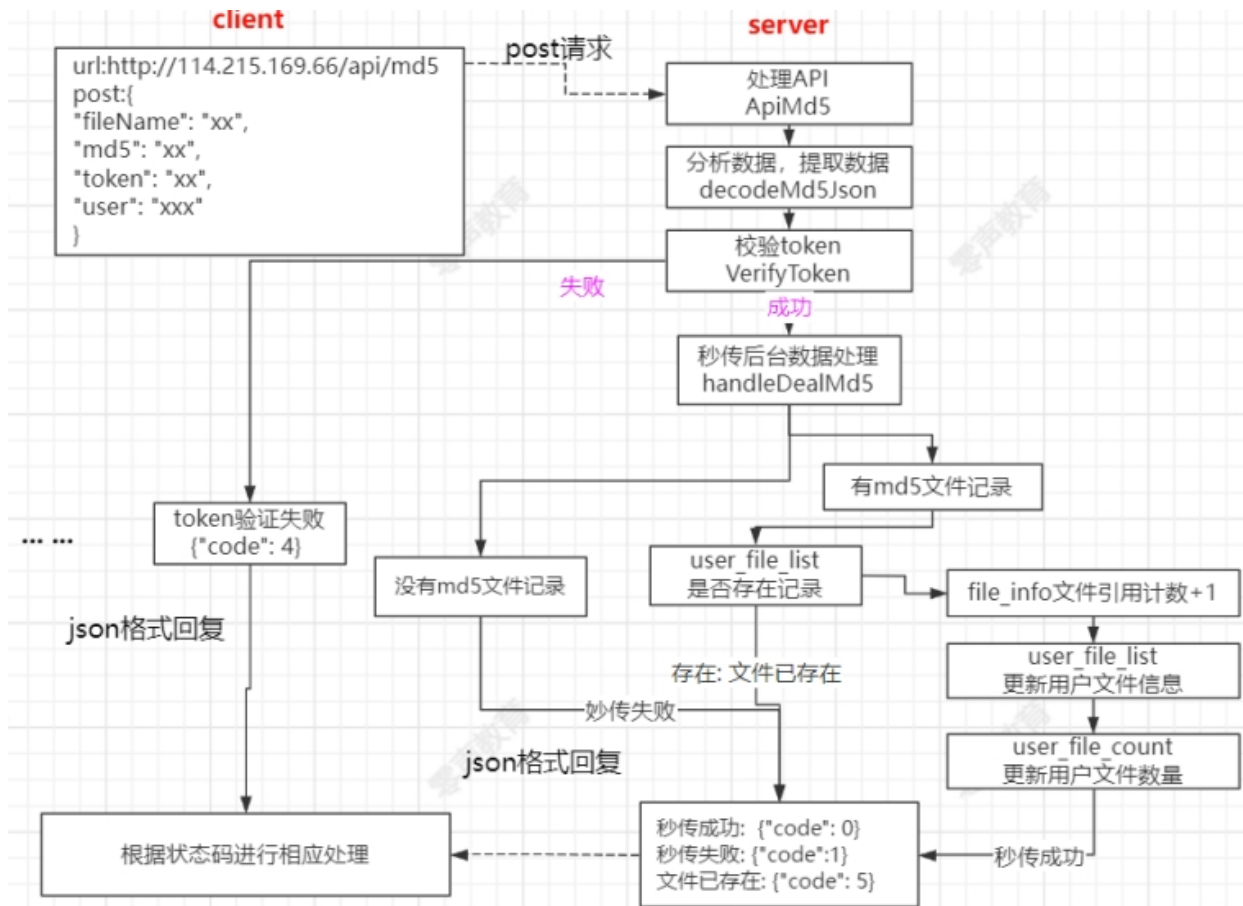
```
{
  "code": 0
}
```

## 处理逻辑

每个文件都有一个唯一的MD5值（比如2bf8170b42cc7124b04a8886c83a9c6f），就好比每个人的指纹都是唯一的一样，校验MD5就是用来确保文件在传输过程中未被修改过。

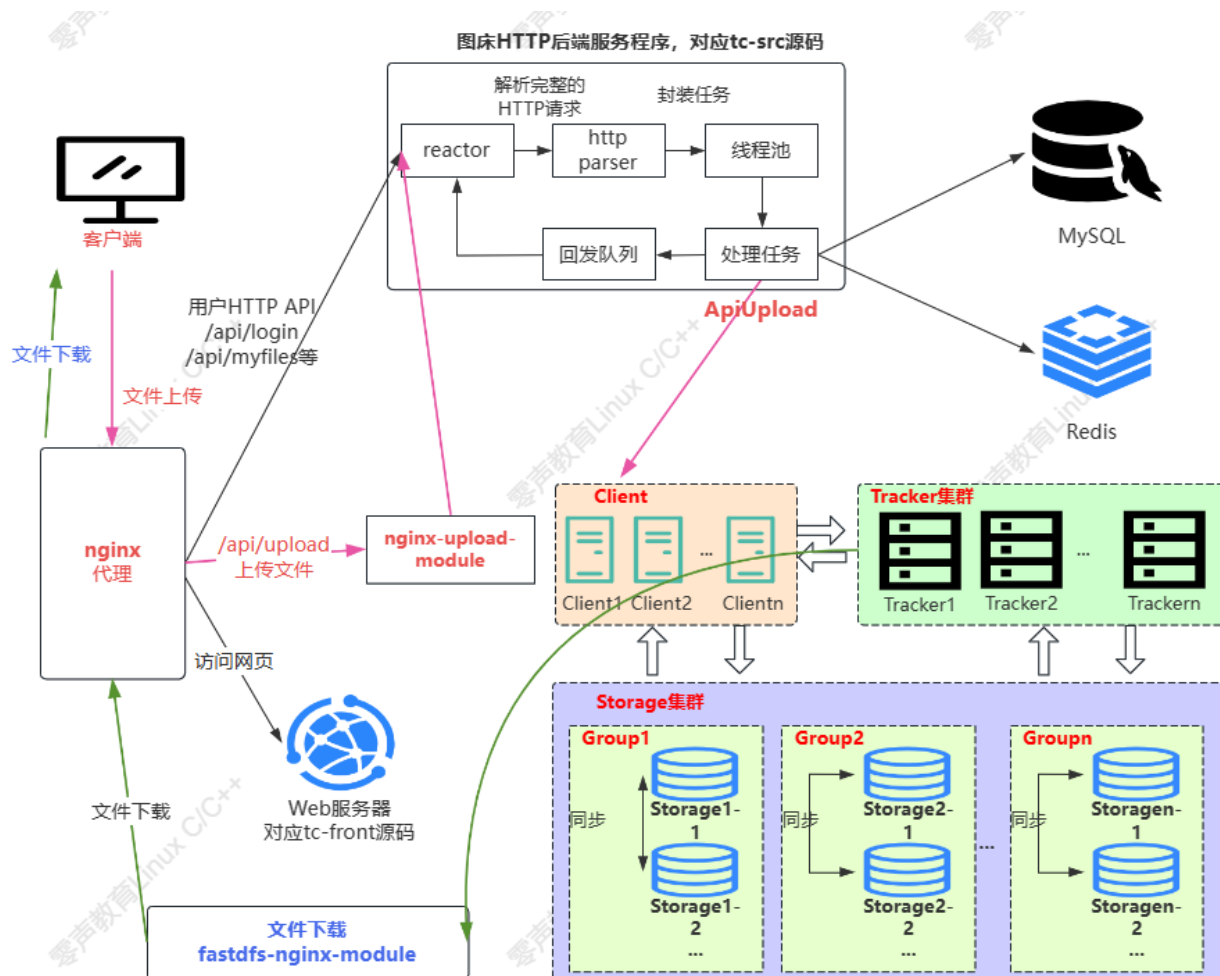
- 客户端在上传文件之前将文件的MD5码上传到服务器
- 服务器端判断是否已存在此MD5码，如果存在，说明该文件已存在，则此文件无需再上传，在此文件的计数器加1，说明此文件多了一个用户共用。
  - file\_info count
  - 插入用户文件列表
  - 用户文件计数+1

- 如果服务器没有此MD5码，说明上传的文件是新文件，则真正上传此文件



## 5.2 上传文件api\_upload.cc

流程图



- 客户端在上传文件之前将文件的MD5码上传到服务器
- 服务器端判断是否已存在此MD5码，如果存在，说明该文件已存在，则此文件无需再上传，在此文件的计数器加1，说明此文件多了一个用户共用。
  - file\_info count
  - 插入user\_file\_list用户文件列表
  - 用户文件计数+1（后续操作Redis）
- 如果服务器没有此MD5码，说明上传的文件是新文件，则真正上传此文件。本流程就是处理文件上传的逻辑。

上传文件的总体逻辑：

1. 先通过nginx-upload-module模块上传文件到临时目录
2. nginx-upload-module模块上传完文件后通知/api/upload后端处理程序：
3. 后端处理程序ApiUpload函数解析文件信息，然后将临时文件上传到fastdfs
4. 更新数据库记录：file\_info, user\_file\_list

# /api/upload上传文件

## 请求和应答

请求URL

URL	<a href="http://192.168.1.27/api/upload">http://192.168.1.27/api/upload</a>
请求方式	POST
HTTP版本	1.1
Content-Type	application/octet-stream

返回结果参数说明

名称	含义	规则说明
code	结果值	0: 上传成功1: 上传失败

## 处理逻辑

客户端上传文件信息具有一定的格式，先把文件上传到nginx

```
-----WebKitFormBoundary88asdgewtgewx\r\n\nContent-Disposition: form-data; user = "milo"; filename = "xxx.jpg"; md5 = "xxxx";\nsize = 1024\r\n\nContent-Type: application/octet-stream\r\n\n\r\n\n真正的文件内容\r\n\n-----WebKitFormBoundary88asdgewtgewx
```

