

# 5-高负载nginx/fastcgi和文件上传下载原理

---

## 1 FastCGI

### 1.1 CGI

#### 1.1.1 什么是CGI

#### 1.1.2 CGI处理流程

#### 1.1.3 环境变量

#### 1.1.4 标准输入

### 2.2 FastCGI

#### 2.2.1 什么是FastCGI

#### 2.2.2 FastCGI处理流程

#### 2.2.3 进程管理器管理：spawn-fcgi

##### 2.2.3.1 什么是spawn-fcgi

##### 2.2.3.2 编译安装spawn-fcgi

#### 2.2.4 软件开发套件：fcgi

##### 2.2.4.1 编译安装fcgi

##### 2.2.4.2 测试程序

##### 2.2.4.3 有关Nginx的fcgi的配置

## 2 FastDFS-Nginx扩展模块分析

### 2.1 参考架构

### 2.2 实现原理

#### 2.1 源码包说明

#### 2.2 初始化

##### 2.2.1 加载配置文件

##### 2.2.2 读取扩展模块配置

##### 2.2.3 加载服务端配置

#### 2.3 下载过程

##### 2.3.1 解析访问路径

##### 2.3.2 防盗链检查

##### 2.3.3 获取文件元数据

#### 2.3.4 检查本地文件是否存在

#### 3.3.5 文件不存在的处理

#### 2.3.6 输出本地文件

### 3 文件上传下载原理

#### 3.1 文件上传原理

##### 3.1.1 http请求格式

##### 3.1.2 服务器解析

#### 3.2 文件上传类型分析

##### 3.2.1 秒传

###### 1. 什么是秒传

###### 2. 秒传核心逻辑

##### 3.2.2 分片上传

###### 1. 什么是分片上传

###### 2. 分片上传的场景

##### 3.2.3 大文件上传

##### 3.2.4 断点续传

###### 1. 什么是断点续传

###### 2. 应用场景

###### 3. 实现断点续传的核心逻辑

###### 4. 实现流程步骤

#### 3.3 文件下载原理

### 4 文件上传下载实现

#### 4.0 安装nodejs

#### 4.1 普通上传

当前图床项目web上传文件格式

自定义的格式

#### 4.2 分片上传

#### 4.3 断点续传

#### 4.4 多线程下载

异常处理

### 5 参考引申

该文档, 不包括fastdfs部署, fastdfs参考《1-3.1 FastDFS 单机版环境搭建》文档。

redis、nginx、mysql组件如果已经安装不需要重复安装, 只需要根据文档做必要的配置。

# 1 FastCGI

## 1.1 CGI

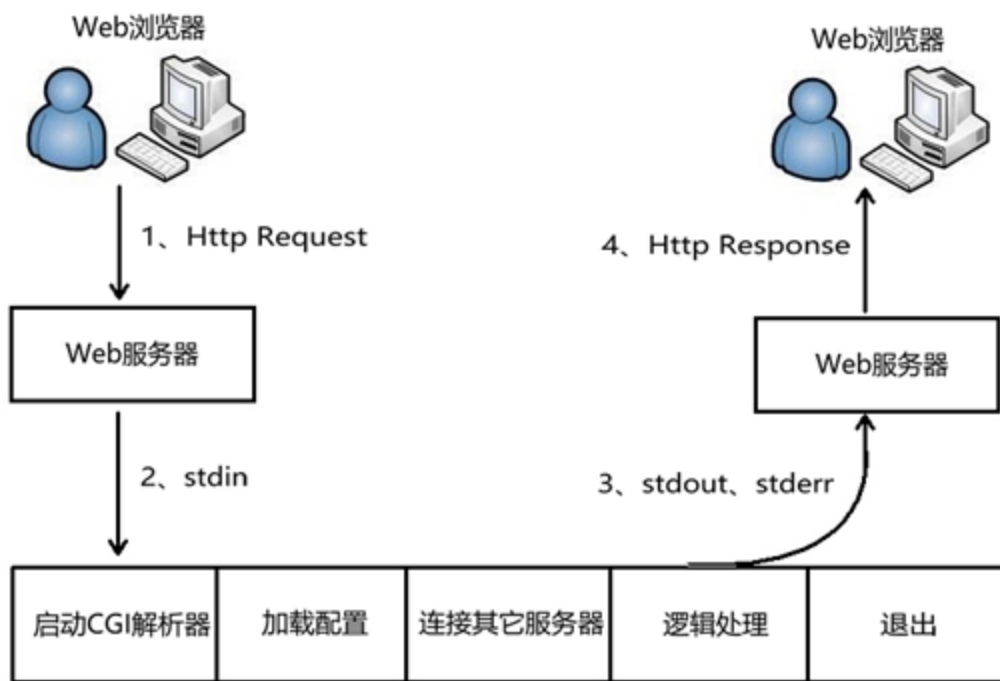
### 1.1.1 什么是CGI

通用网关接口(Common Gateway Interface、CGI)描述了客户端和服务程序之间传输数据的一种标准, 可以让一个客户端, 从网页浏览器向执行在网络服务器上的程序请求数据。

CGI独立于任何语言的, CGI 程序可以用任何脚本语言或者是完全独立编程语言实现, 只要这个语言可以在这个系统上运行。Unix shell script、Python、Ruby、PHP、perl、Tcl、C/C++和 Visual Basic 都可以用来编写 CGI 程序。

最初, CGI 是在 1993 年由美国国家超级电脑应用中心(NCSA)为 NCSA HTTPd Web 服务器开发的。这个 Web 服务器使用了 UNIX shell 环境变量来保存从 Web 服务器传递出去参数, 然后生成一个运行 CGI 的独立的进程。

### 1.1.2 CGI处理流程



- 1.web服务器收到客户端(浏览器)的请求Http Request，启动CGI程序，并通过环境变量、标准输入传递数据
- 2.CGI进程启动解析器、加载配置(如业务相关配置)、连接其它服务器(如数据库服务器)、逻辑处理等
- 3.CGI进程将处理结果通过标准输出、标准错误，传递给web服务器
- 4.web服务器收到CGI返回的结果，构建Http Response返回给客户端，并杀死CGI进程

**web服务器与CGI通过环境变量、标准输入、标准输出、标准错误互相传递数据。**在遇到用户连接请求：

- 先要创建CGI子进程，然后CGI子进程处理请求，处理完事退出这个子进程：fork-and-execute
- CGI方式是客户端有多少个请求，就开辟多少个子进程，每个子进程都需要启动自己的解释器、加载配置，连接其他服务器等初始化工作，这是CGI进程性能低下的主要原因。当用户请求非常多的时候，会占用大量的内存、cpu等资源，造成性能低下。

CGI使外部程序与Web服务器之间交互成为可能。CGI程序运行在独立的进程中，并对每个Web请求建立一个进程，**这种方法非常容易实现，但效率很差**，难以扩展。面对大量请求，进程的大量建立和消亡使操作系统性能大大下降。此外，由于地址空间无法共享，也限制了资源重用。

### 1.1.3 环境变量

GET请求，它将数据打包放置在环境变量QUERY\_STRING中，CGI从环境变量QUERY\_STRING中获取数据。

常见的环境变量如下表所示：

环境变数	含义
AUTH_TYPE	存取认证类型
CONTENT_LENGTH	由标准输入传递给CGI程序的数据长度，以bytes或字元数来计算
CONTENT_TYPE	请求的MIME类型
GATEWAY_INTERFACE	服务器的CGI版本编号
HTTP_ACCEPT	浏览器能直接接收的Content-types, 可以有HTTP Accept header定义
HTTP_USER_AGENT	递交表单的浏览器的名称、版本和其他平台性的附加信息
HTTP_REFERER	递交表单的文本的URL，不是所有的浏览器都发出这个信息，不要依赖它
PATH_INFO	传递给CGI程序的路径信息
QUERY_STRING	传递给CGI程序的请求参数，也就是用"?"隔开，添加在URL后面的字串
REMOTE_ADDR	client端的host名称
REMOTE_HOST	client端的IP位址
REMOTE_USER	client端送出来的使用者名称
REMOTE_METHOD	client端发出请求的方法(如get、post)
SCRIPT_NAME	CGI程序所在的虚拟路径，如/cgi-bin/echo
SERVER_NAME	server的host名称或IP地址
SERVER_PORT	收到request的server端口
SERVER_PROTOCOL	所使用的通讯协定和版本编号
SERVER_SOFTWARE	server程序的名称和版本

#### 1.1.4 标准输入

环境变量的大小是有一定的限制的，当需要传送的数据量大时，储存环境变量的空间可能会不足，造成数据接收不完全，甚至无法执行CGI程序。

因此后来又发展出另外一种方法：POST，也就是利用I/O重新导向的技巧，让CGI程序可以由stdin和stdout直接跟浏览器沟通。

当我们指定用这种方法传递请求的数据时，web服务器收到数据后会先放在一块输入缓冲区中，并且将数据的大小记录在CONTENT\_LENGTH这个环境变量，然后调用CGI程序并将CGI程序的stdin指向这块缓冲区，于是我们就可以很顺利的通过stdin和环境变数CONTENT\_LENGTH得到所有的信息，再没有信息大小的限制了。

## 2.2 FastCGI

### 2.2.1 什么是FastCGI

快速通用网关接口(Fast Common Gateway Interface / FastCGI)是通用网关接口(CGI)的改进，描述了客户端和服务端程序之间传输数据的一种标准。

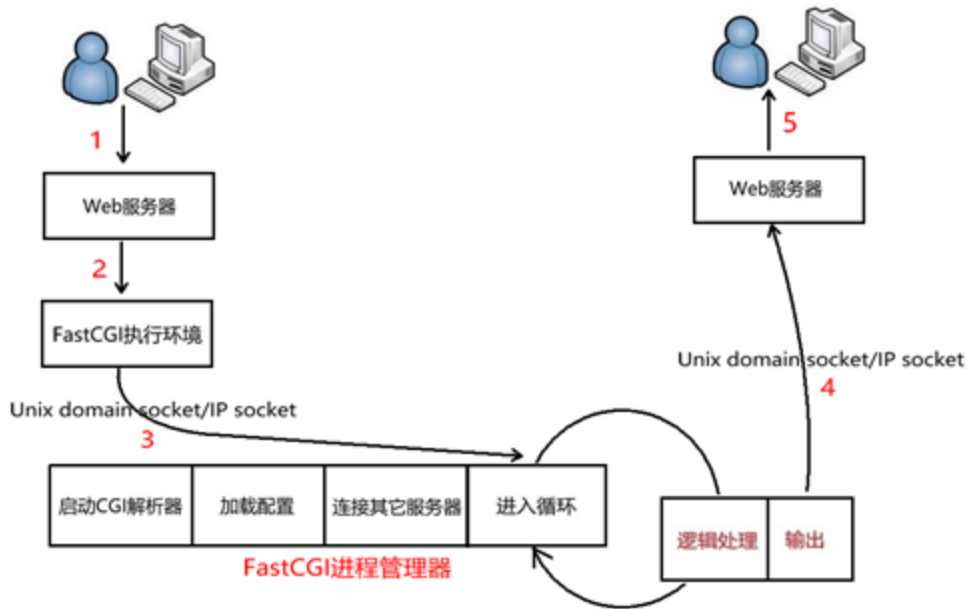
FastCGI致力于减少Web服务器与CGI程式之间互动的开销，从而使服务器可以同时处理更多的Web请求。与为每个请求创建一个新的进程不同，FastCGI使用持续的进程来处理一连串的请求。这些进程由FastCGI进程管理器管理，而不是web服务器。

nginx服务支持FastCGI模式，能够快速高效地处理动态请求。而nginx对应的FastCGI模块为：

- ngx\_http\_fastcgi\_module。
- 

ngx\_http\_fastcgi\_module 模块允许将请求传递给 FastCGI 服务器。

### 2.2.2 FastCGI处理流程



1. Web 服务器启动时载入初始化FastCGI执行环境。例如IIS、ISAPI、apache mod\_fastcgi、nginx ngx\_http\_fastcgi\_module、lighttpd mod\_fastcgi。
2. FastCGI进程管理器自身初始化，启动多个CGI解释器进程并等待来自Web服务器的连接。启动FastCGI进程时，可以配置以ip和UNIX 域socket两种方式启动。
3. 当客户端请求到达Web 服务器时， Web 服务器将请求采用socket方式转发FastCGI主进程，FastCGI主进程选择并连接到一个CGI解释器。Web 服务器将CGI环境变量和标准输入发送到FastCGI子进程。
4. FastCGI子进程完成处理后将标准输出和错误信息从同一socket连接返回Web 服务器。当FastCGI子进程关闭连接时，请求便处理完成。
5. FastCGI子进程接着等待并处理来自Web 服务器的下一个连接。

由于FastCGI程序并不需要不断的产生新进程，可以大大降低服务器的压力并且产生较高的应用效率。它的速度效率最少要比CGI 技术提高 5 倍以上。它还支持分布式的部署，即FastCGI 程序可以在web 服务器以外的主机上执行。

CGI 是所谓的短生存期应用程序，FastCGI 是所谓的长生存期应用程序。FastCGI像是一个常驻(long-live)型的CGI，它可以一直执行着，不会每次都要花费时间去fork一次(这是CGI最为人诟病的fork-and-execute 模式)。

## 2.2.3 进程管理器管理：spawn-fcgi

### 2.2.3.1 什么是spawn-fcgi

Nginx不能像Apache那样直接执行外部可执行程序，但Nginx可以作为代理服务器，将请求转发给后端服务器，这也是Nginx的主要作用之一。其中Nginx就支持FastCGI代理，接收客户端的请求，然后将请求转发给后端FastCGI进程。

由于FastCGI进程由FastCGI进程管理器管理，而不是Nginx。这样就需要一个FastCGI进程管理器，管理我们编写FastCGI程序。

spawn-fcgi是一个通用的FastCGI进程管理器，简单小巧，原先是属于lighttpd的一部分，后来由于使用比较广泛，所以就迁移出来作为独立项目。

spawn-fcgi使用pre-fork 模型，功能主要是打开监听端口，绑定地址，然后fork-and-exec创建我们编写的FastCGI应用程序进程，退出完成工作。FastCGI应用程序初始化，然后进入死循环侦听socket的连接请求。

### 2.2.3.2 编译安装spawn-fcgi

spawn-fcgi源码包下载地址：<http://redmine.lighttpd.net/projects/spawn-fcgi/wiki>

编译和安装spawn-fcgi相关命令：

```
▼ Bash | 复制代码  
1  wget http://download.lighttpd.net/spawn-fcgi/releases-1.6.x/spawn-fcgi-  
   1.6.4.tar.gz  
2  tar -zxf spawn-fcgi-1.6.4.tar.gz  
3  cd spawn-fcgi-1.6.4/  
4  ./configure  
5  make  
6  make install
```

如果遇到以下错误：

```
./autogen.sh: x: autoreconf: not found
```

因为没有安装automake工具，ubuntu用下面的命令安装即可：

```
apt-get install autoconf automake libtool
```

spawn-fcgi的帮助信息可以通过man spawn-fcgi或spawn-fcgi -h获得，下面是部分常用

spawn-fcgi参数信息：



参数	含义
f <fcgiapp>	指定调用FastCGI的进程的执行程序位置
-a <addr>	绑定到地址addr
-p <port>	绑定到端口port
-s <path>	绑定到unix domain socket
-C <chlds>	指定产生的FastCGI的进程数，默认为5(仅用于PHP)
-P <path>	指定产生的进程的PID文件路径
-F <chlds>	指定产生的FastCGI的进程数(C的CGI用这个)
-u和-g FastCGI	使用什么身份(-u用户、-g用户组)运行，CentOS下可以使用apache用户，其他的根据情况配置，如nobody、www-data等

## 2.2.4 软件开发套件：fcgi

### 2.2.4.1 编译安装fcgi

使用C/C++编写FastCGI应用程序，可以使用FastCGI软件开发套件或者其它开发框架，如fcgi。

fcgi下载地址：wget <https://fossies.org/linux/www/old/fcgi-2.4.0.tar.gz>

编译和安装fcgi相关命令：

▼

Bash | 复制代码

```

1  wget https://fossies.org/linux/www/old/fcgi-2.4.0.tar.gz --no-check-
   certificate
2  tar -zxf fcgi-2.4.0.tar.gz
3  cd fcgi-2.4.1-SNAP-0910052249/
4  ./configure
5
6  编译前在libfcgi/fcgio.cpp 文件上添加#include <stdio.h>
7  make
8  make install

```

**如果编译出现下面问题：**

fcgio.cpp: In destructor 'virtual fcgi\_streambuf::~~fcgi\_streambuf()':

fcgio.cpp:50:14: **error:** 'EOF' was not declared in this scope

```
    overflow(EOF);
```

```
    ^
```

fcgio.cpp: In member function 'virtual int fcgi\_streambuf::overflow(int)':

fcgio.cpp:70:72: **error:** 'EOF' was not declared in this scope

```
    if (FCGX_PutStr(pbase(), plen, this->fcgx) != plen) return EOF;
```

```
                                ^
```

fcgio.cpp:75:14: **error:** 'EOF' was not declared in this scope

```
    if (c != EOF)
```

```
    ^
```

fcgio.cpp: In member function 'virtual int fcgi\_streambuf::sync()':

fcgio.cpp:86:18: **error:** 'EOF' was not declared in this scope

```
    if (overflow(EOF)) return EOF;
```

```
    ^
```

fcgio.cpp:87:41: **error:** 'EOF' was not declared in this scope

```
    if (FCGX_FFlush(this->fcgx)) return EOF;
```

```
                                ^
```

fcgio.cpp: In member function 'virtual int fcgi\_streambuf::underflow()':

fcgio.cpp:113:35: **error:** 'EOF' was not declared in this scope

```
    if (glen <= 0) return EOF;
```

**解决方法：** 在fcgio.h/fcgio.cpp 文件上添加#include <stdio.h>

## 2.2.4.2 测试程序

示例代码： vim fcgi.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h>
5  #include "fcgi_stdio.h"
6
7  int main(int argc, char *argv[])
8  {
9      int count = 0;
10
11     //阻塞等待并监听某个端口，等待Nginx将数据发过来
12
13     while (FCGI_Accept() >= 0)
14     {
15         //如果想得到数据，需要从stdin去读，实际上从Nginx上去读
16         //如果想上传数据，需要往stdout写，实际上是给Nginx写数据
17
18         printf("Content-type: text/html\r\n");
19         printf("\r\n");
20         printf("<title>Fast CGI Hello!</title>");
21         printf("<h1>Fast CGI Hello!</h1>");
22         //SERVER_NAME: 得到server的host名称
23         printf("Request number %d running on host <i>%s</i>\n",
24             ++count, getenv("SERVER_NAME"));
25     }
26
27     return 0;
28 }

```

编译代码：

```
1  gcc fcgi.c -o test -lfcgi
```

test就是其中一个针对client一个http请求的业务服务应用程序。需要在后台跑起来，并且让spawn负责管理。

**记得先：ldconfig，否则spawn-fcgi启动异常**

Bash | 复制代码

```
1 root@iZbp1d83xkvoja33dm7ki2Z:~/0voice/cloud-drive/test# ldconfig
2 root@iZbp1d83xkvoja33dm7ki2Z:~/0voice/cloud-drive/test# spawn-fcgi -a
  127.0.0.1 -p 8001 -f ./test
3 spawn-fcgi: child spawned successfully: PID: 24314
```

查看8001是否有被监听

C | 复制代码

```
1 root@iZbp1h2l856zgoegc8rvnhZ:~/tuchuang# lsof -i:8001
2 COMMAND  PID USER  FD  TYPE  DEVICE  SIZE/OFF  NODE NAME
3 test    3370 root   0u   IPv4  5624558      0t0  TCP localhost:8001
  (LISTEN)
```

### 2.2.4.3 有关Nginx的fcgi的配置

- 监听用户的test请求，通过fastcgi\_pass交给本地8001端口处理
- 此时spawn-cgi已经将8001端口交给之前我们写好的test进程处理

SQL | 复制代码

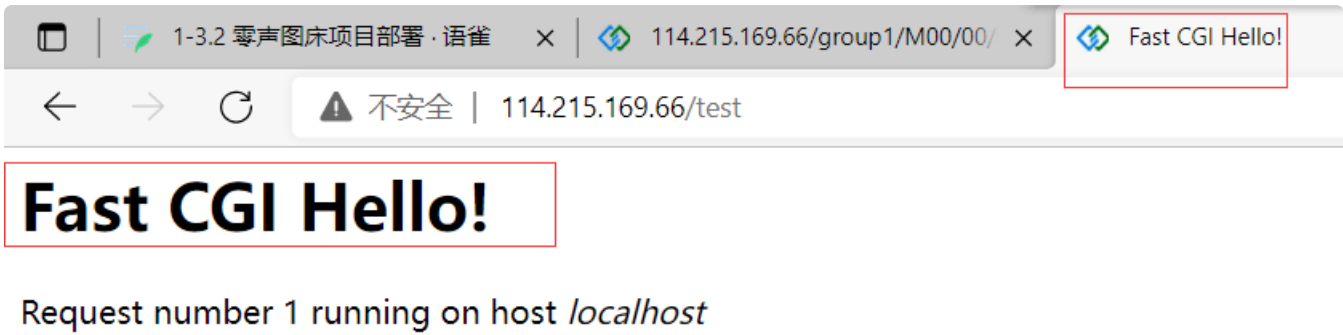
```
1 location /test {
2     fastcgi_pass 127.0.0.1:8001;
3     fastcgi_index test;
4     include fastcgi.conf;
5 }
```

记得nginx先重新加载配置文件

Bash | 复制代码

```
1 /usr/local/nginx/sbin/nginx -s reload
```

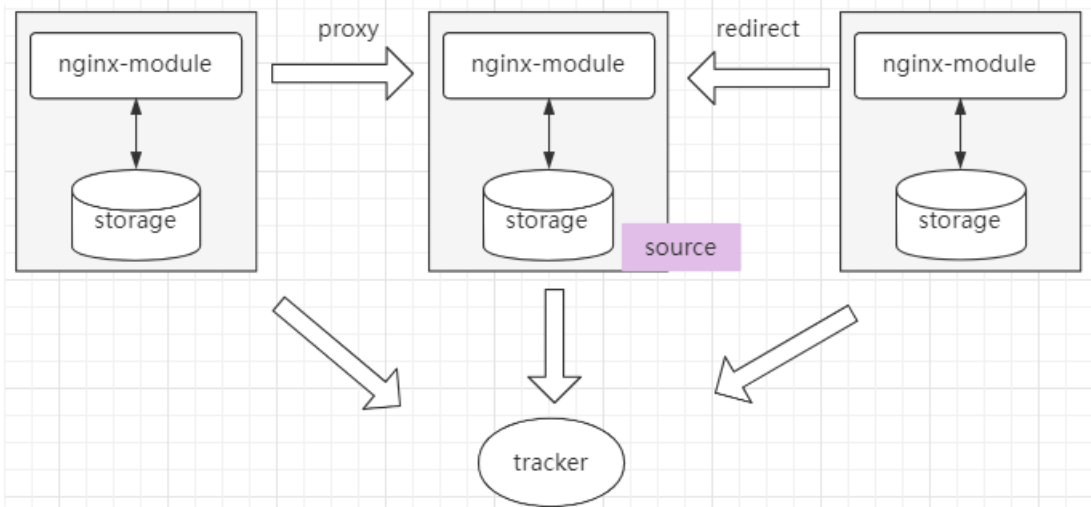
当Nginx收到http://ip/test请求时，比如http://114.215.169.66/test，会匹配到location test块，将请求传到后端的FastCGI应用程序处理：



## 2 FastDFS–Nginx扩展模块分析

### 2.1 参考架构

使用FastDFS整合Nginx的参考架构如下所示



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

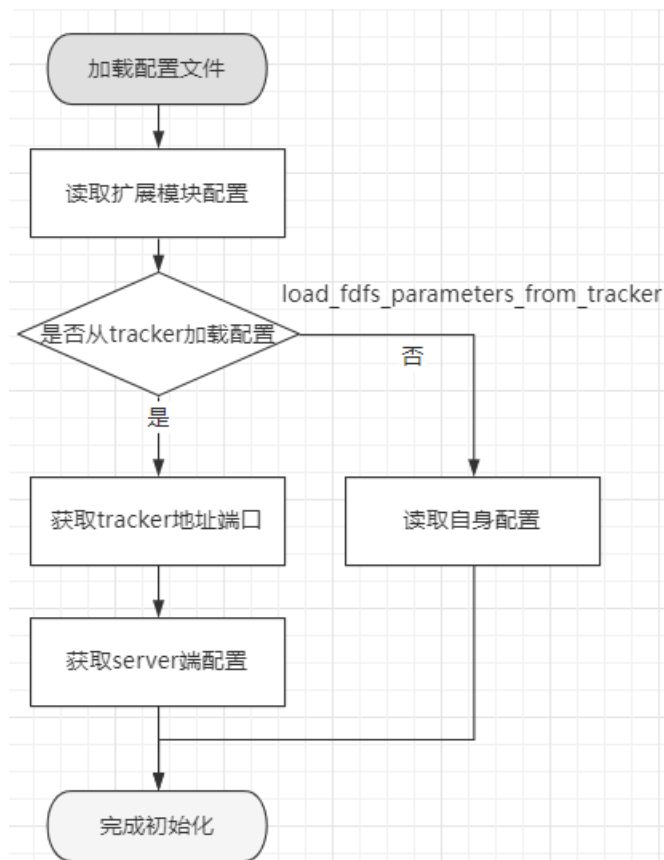
### 2.2 实现原理

## 2.1 源码包说明

📄 复制代码

```
1  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  
2  common.c                    //fastdfs-module核心模块，实现了初始化、文件下载的  
    主要逻辑  
3  common.h                    //对应于common.c的头文件  
4  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  
5  mod_fastdfs.conf            //扩展配置文件的范例
```

## 2.2 初始化



`fdfs_mod_init`

### 2.2.1 加载配置文件

目标文件：/etc/fdfs/mod\_fastdfs.conf 在config的时候写死

## 2.2.2 读取扩展模块配置

一些重要参数包括：

▼ C | 复制代码

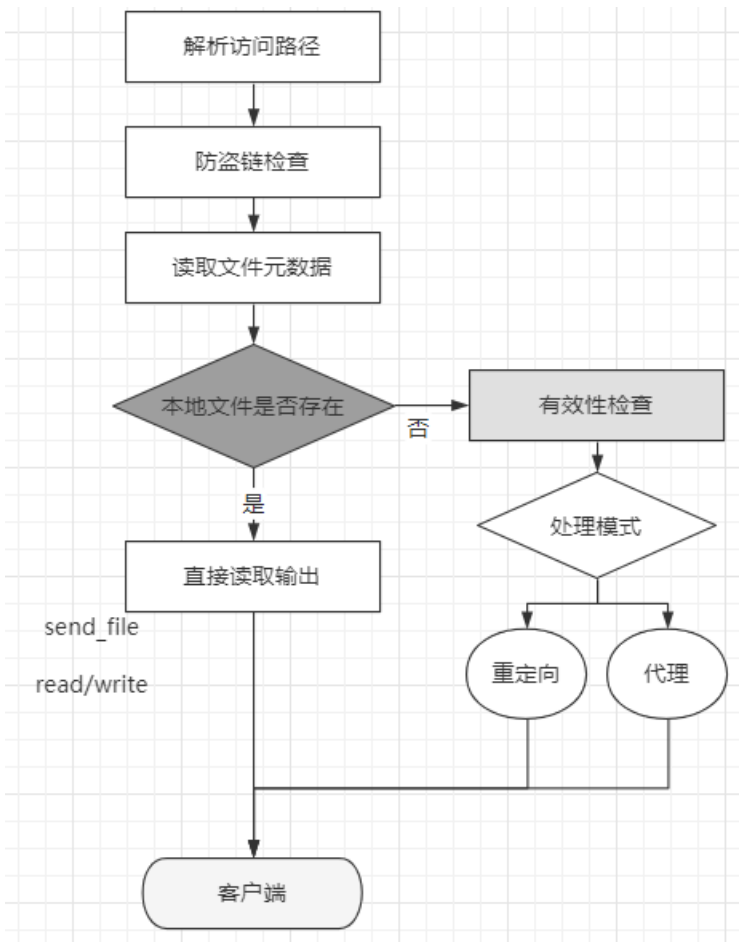
```
1  group_count           //group个数
2  url_have_group_name   //url中是否包含group
3  group.store_path      //group对应的存储路径
4  connect_timeout       //连接超时
5  network_timeout       //接收或发送超时
6  storage_server_port   //storage_server端口，用于在找不到文件情况下连接源storage
    下载文件(该做法已过时)
7  response_mode         //响应模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

## 2.2.3 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:
  - a. 调用fdfs\_load\_tracker\_group\_ex解析tracker连接配置；
  - b. 调用fdfs\_get\_ini\_context\_from\_tracker连接tracker获取配置信息；
  - c. 获取storage\_sync\_file\_max\_delay阈值
  - d. 获取use\_storage\_id
  - e. 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:
  - a. 从mod\_fastdfs.conf加载所需配置：storage\_sync\_file\_max\_delay、use\_storage\_id；
  - b. 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)

## 2.3 下载过程



### 2.3.1 解析访问路径

得到group和file\_id\_without\_group两个参数;

### 2.3.2 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

### 2.3.3 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





复制代码

```
1      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。



复制代码

```
1  fdfs_shared.func.c  
2  -> fdfs_get_server_id_type(ip_addr.s_addr) == FDFS_ID_TYPE_SERVER_ID  
3  ...  
4  if (id > 0 && id <= FDFS_MAX_SERVER_ID) {  
5      return FDFS_ID_TYPE_SERVER_ID;  
6  } else {  
7      return FDFS_ID_TYPE_IP_ADDRESS;  
8  }
```

判断标准为ip段的整数值是否在 0 到  $\rightarrow$  FDFS\_MAX\_SERVER\_ID(见tracker\_types.h)之间；

其中FDFS\_MAX\_SERVER\_ID =  $(1 \ll 24) - 1$ ，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

### 2.3.4 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

```

1  if (bSameGroup)
2  {
3      FDFSTrunkHeader trunkHeader;
4      if ((result=trunk_file_stat_ex1(pStorePaths, store_path_index, \
5          true_filename, filename_len, &file_stat, \
6          &trunkInfo, &trunkHeader, &fd)) != 0)
7      {
8          bFileExists = false;
9      }
10     else
11     {
12         bFileExists = true;
13     }
14 }
15 else
16 {
17     bFileExists = false;
18     memset(&trunkInfo, 0, sizeof(trunkInfo));
19 }

```

### 3.3.5 文件不存在的处理

- 进行有效性检查

检查项有二：

A. 源storage是本机或者当前时间与文件创建时间的差距已经超过阈值，报错；

```

1      if (is_local_host_ip(file_info.source_ip_addr) || \
2          (file_info.create_timestamp > 0 && (time(NULL) - \
3              file_info.create_timestamp >
4              '''storage_sync_file_max_delay'''))

```

B. 如果是redirect后的场景，同样报错；

如果是由其他storage节点redirect过来的请求，其url参数中会存在redirect一项

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)

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

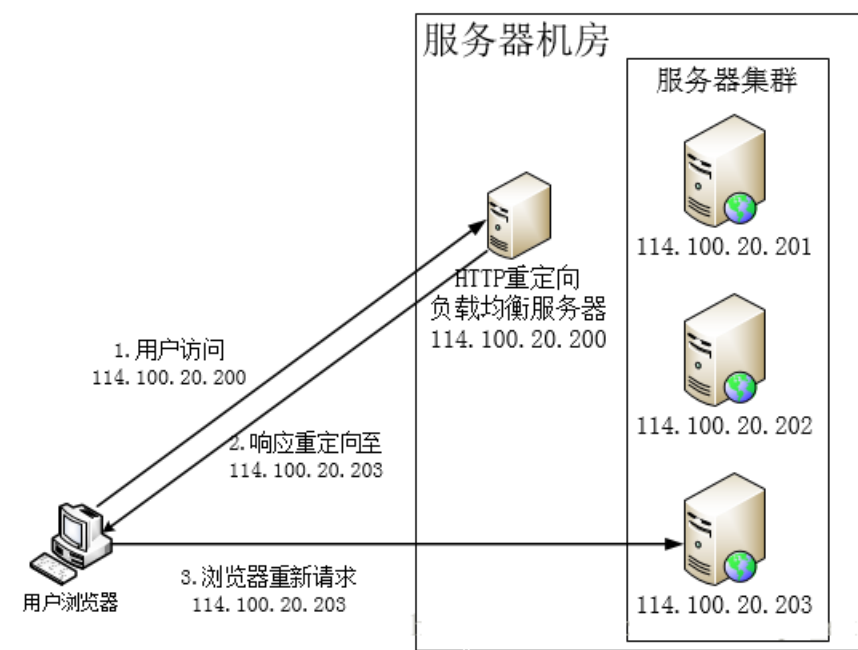
注: 该模式下要求源storage配备公开访问的webserver、同样的端口(一般是80)、同样的path配置。

- 代理模式

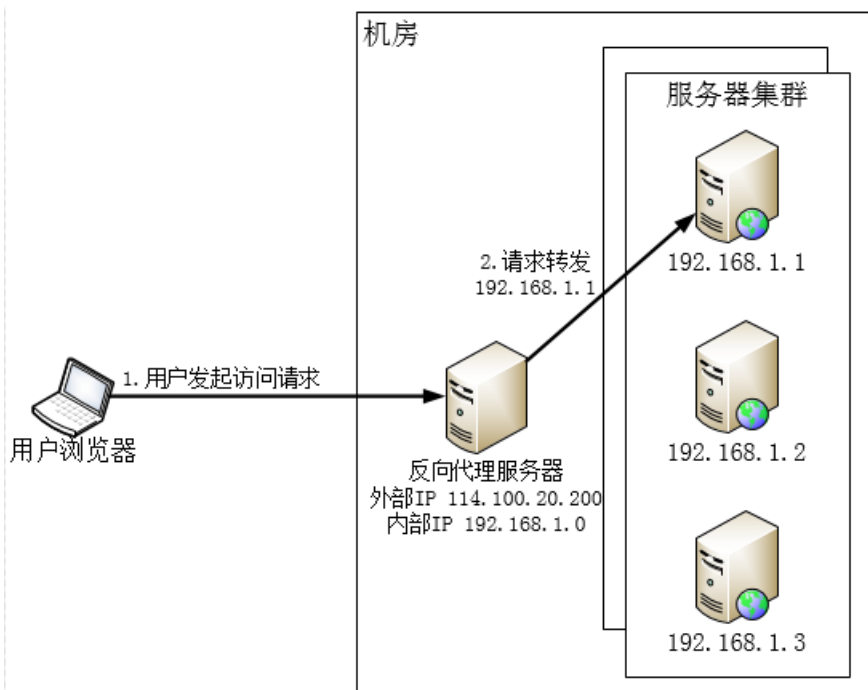
配置项response\_mode = proxy, 该模式的工作原理如同反向代理的做法, 而仅仅使用源storage地址作为代理proxy的host, 其余部分保持不变。

```
1 if (pContext->proxy_handler != NULL)
2 {
3     return pContext->proxy_handler(pContext->arg, \
4         file_info.source_ip_addr);
5 }
6 //其中proxy_handler方法来自ngx_http_fastdfs_module.c文件的
  ngx_http_fastdfs_proxy_handler方法
7 //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:
8 rc = ngx_http_read_client_request_body(r,
  ngx_http_upstream_init); //执行代理请求, 并返回结果
```

重定向和代理的区别:



重定向



反向代理负载均衡工作原理图

反向代理

### 2.3.6 输出本地文件

当本地文件存在时，将直接输出。

- 根据是否trunkfile获取文件名，文件名长度、文件offset;

```

1  bTrunkFile = IS_TRUNK_FILE_BY_ID(trunkInfo);
2  if (bTrunkFile)
3  {
4      trunk_get_full_filename_ex(pStorePaths, &trunkInfo, \
5          full_filename, sizeof(full_filename));
6      full_filename_len = strlen(full_filename);
7      file_offset = TRUNK_FILE_START_OFFSET(trunkInfo) + \
8          pContext->range.start;
9  }
10 else
11 {
12     full_filename_len = snprintf(full_filename, \
13         sizeof(full_filename), "%s/data/%s", \
14         pStorePaths->paths[store_path_index], \
15         true_filename);
16     file_offset = pContext->range.start;
17 }

```

- 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;

```

1  if (pContext->send_file != NULL && !bTrunkFile)
2  {
3      http_status = pContext->if_range ? \
4          HTTP_PARTIAL_CONTENT : HTTP_OK;
5      OUTPUT_HEADERS(pContext, (&response), http_status)
6      .....
7      return pContext->send_file(pContext->arg, full_filename, \
8          full_filename_len, file_offset, download_bytes);
9  }

```

- 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...

```

1  while (remain_bytes > 0)
2  {
3      read_bytes = remain_bytes <= FDFS_OUTPUT_CHUNK_SIZE ? \
4          remain_bytes : FDFS_OUTPUT_CHUNK_SIZE;
5      if (read(fd, file_trunk_buff, read_bytes) != read_bytes)
6      {
7          close(fd);
8          .....
9          return HTTP_INTERNAL_SERVER_ERROR;
10     }
11
12     remain_bytes -= read_bytes;
13     if (pContext->send_reply_chunk(pContext->arg, \
14         (remain_bytes == 0) ? 1: 0, file_trunk_buff, \
15         read_bytes) != 0)
16     {
17         close(fd);
18         return HTTP_INTERNAL_SERVER_ERROR;
19     }
20 }

```

其中chunk大小见config文件配置：-DFDFS\_OUTPUT\_CHUNK\_SIZE='256\*1024'。

## 3 文件上传下载原理

### 3.1 文件上传原理

#### 3.1.1 http请求格式

件上传的是根据 [http](#) 协议的规范和定义，完成请求消息体的封装和消息体的解析，然后将二进制内容保存到文件。

在上传一个文件时，需要把 form 标签的 [enctype](#) 设置为 [multipart/form-data](#),同时 [method](#) 必须为 [post](#) 方法。

那么 [multipart/form-data](#) 表示什么呢？

[multipart/form-data](#) 结构



- 请求头：

`Content-Type: multipart/form-data; boundary=----WebKitFormBoundaryDCntfiXcSkPhS4PN` 表示本次请求要上传文件，其中 `boundary` 表示分隔符，如果要上传多个表单项，就要使用 `boundary` 分割，每个表单项由 `-----XXX` 开始，以 `-----XXX` 结尾。

- 消息体- Form Data 部分

每一个表单项又由 `Content-Type` 和 `Content-Disposition` 组成。

`Content-Disposition: form-data` 为固定值，表示一个表单元素，`name` 表示表单元素的名称，回车换行后面就是 `name` 的值，如果是上传文件就是文件的二进制内容。

`Content-Type`：表示当前的内容的 MIME 类型，是图片还是文本还是二进制数据。

### 3.1.2 服务器解析

客户端发送请求到服务器后，服务器会收到请求的消息体，然后对消息体进行解析，解析出哪些是普通表单哪些是附件。



## 3.2 文件上传类型分析

### 3.2.1 秒传

#### 1. 什么是秒传

通俗的说，你把要上传的东西上传，服务器会先做MD5校验，如果服务器上有一样的东西，它就直接给你个新地址，其实你下载的都是服务器上的同一个文件，想要不秒传，其实只要让MD5改变，就是对文件本身做一下修改（改名字不行），例如一个文本文件，你多加几个字，MD5就变了，就不会秒传了。

#### 2. 秒传核心逻辑

- a、利用redis的set方法存放文件上传状态，其中key为文件上传的md5，value为是否上传完成的标志位，
- b、当标志位true为上传已经完成，此时如果有相同文件上传，则进入秒传逻辑。如果标志位为false，则说明还没上传完成，此时需要在调用set的方法，保存块号文件记录的路径，其中key为上传文件md5加一个固定前缀，value为块号文件记录路径

### 3.2.2 分片上传

#### 1. 什么是分片上传

分片上传，就是将所要上传的文件，按照一定的大小，将整个文件分隔成多个数据块（我们称之为Part）来进行分别上传，上传完之后再由服务端对所有上传的文件进行汇总整合成原始的文件。

#### 2. 分片上传的场景

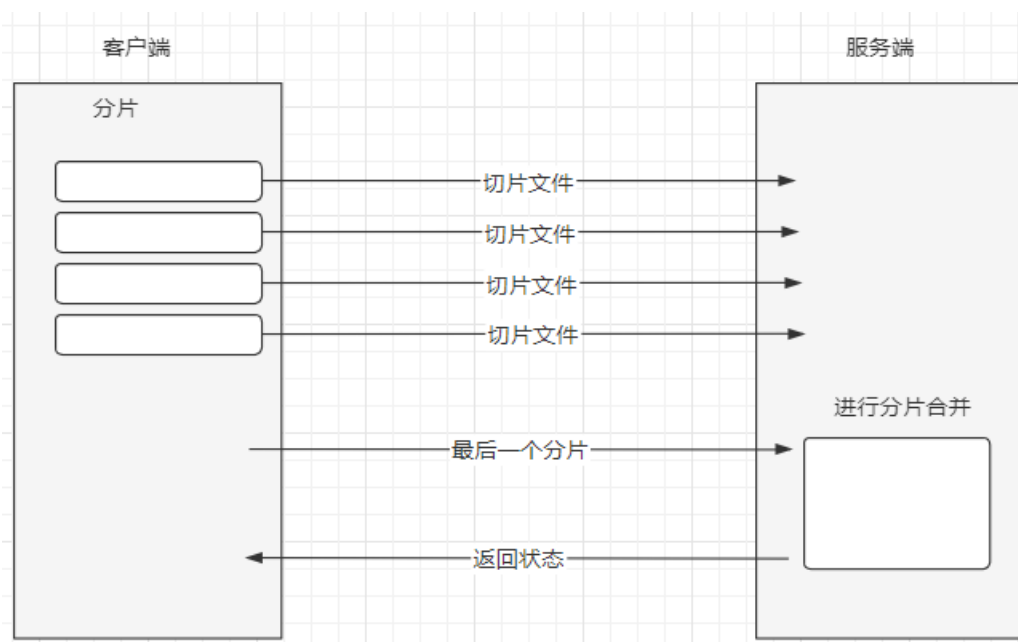


1. 大文件上传
2. 网络环境不好，存在需要重传风险的场景

### 3.2.3 大文件上传

大文件上传一般采用分片上传的方式，这样可以提高文件上传的速度，前端拿到文件流后进行分片，然后与后端进行通讯传输，一般还会结合断点继传，这时后端一般提供三个接口：

- 第一个接口获取已经上传的分片信息
- 第二个接口将前端分片文件进行传输
- 第三个接口是将所有分片上传完成后告诉后端进行文件合并



### 3.2.4 断点续传

#### 1. 什么是断点续传

断点续传是在下载或上传时，将下载或上传任务（一个文件或一个压缩包）人为的划分为几个部分，每一个部分采用一个线程进行上传或下载，如果碰到网络故障，可以从已经上传或下载的部分开始继续上传或者下载未完成的部分，而没有必要从头开始上传或者下载。

#### 2. 应用场景

断点续传可以看成是分片上传的一个衍生，因此可以使用分片上传的场景，都可以使用断点续传。

#### 3. 实现断点续传的核心逻辑

在分片上传的过程中，如果因为系统崩溃或者网络中断等异常因素导致上传中断，这时候客户端需要记录上传的进度。在之后支持再次上传时，可以继续从上次上传中断的地方进行继续上传。

为了避免客户端在上传之后的进度数据被删除而导致重新开始从头上上传的问题，服务端也可以提供相应的接口便于客户端对已经上传的分片数据进行查询，从而使客户端知道已经上传的分片数据，从而从下一个分片数据开始继续上传。

#### 4. 实现流程步骤

##### a、方案一，常规步骤

- 将需要上传的文件按照一定的分割规则，分割成相同大小的数据块；
- 初始化一个分片上传任务，返回本次分片上传唯一标识；
- 按照一定的策略（串行或并行）发送各个分片数据块；
- 发送完成后，服务端根据判断数据上传是否完整，如果完整，则进行数据块合成得到原始文件。

##### b、方案二、实现的步骤

- 前端（客户端）需要根据固定大小对文件进行分片，请求后端（服务端）时要带上分片序号和大小
- 服务端创建conf文件用来记录分块位置，conf文件长度为总分片数，每上传一个分块即向conf文件中写入一个127，那么没上传的位置就是默认的0,已上传的就是Byte.MAX\_VALUE 127（这步是实现断点续传和秒传的核心步骤）
- 服务器按照请求数据中给的分片序号和每片分块大小（分片大小是固定且一样的）算出开始位置，与读取到的文件片段数据，写入文件。

### 3.3 文件下载原理

文件下载比文件上传容易的多。

对于HTTP协议，向服务器请求某个文件时，只要发送类似如下的请求即可：

```
1  GET /Path/FileName HTTP/1.0
2  Host: www.baidu.com:80
3  Accept: */*
4  User-Agent: GeneralDownloadApplication
5  Connection: close
```

第一行中的GET是HTTP协议支持的方法之一，方法名是大小写敏感的。每行用一个“回车换行”分隔，末尾再追加一个“回车换行”作为整个请求的结束。

除第一行以外，其余行都是HTTP头的字段部分。Host字段表示主机名和端口号，如果端口号是默认的80则可以不用写。Accept字段中的\*/\*表示接收任何类型的数据。User-Agent表示用户代理，这个字段可有可无，但强

烈建议加上，因为它是服务器统计、追踪以及识别客户端的依据。Connection字段中的close表示使用非持久连接。

如果服务器成功收到该请求，并且没有出现任何错误，则会返回类似下面的数据：

▼

⌂ | 复制代码

```
1 HTTP/1.0 200 OK
2 Content-Length: 13057672
3 Content-Type: application/octet-stream
4 Last-Modified: Wed, 10 Oct 2005 00:56:34 GMT
5 Accept-Ranges: bytes
6 ETag: "2f38a6cac7cec51:160c"
7 Server: Microsoft-IIS/6.0
8 X-Powered-By: ASP.NET
9 Date: Wed, 16 Nov 2005 01:57:54 GMT
10 Connection: close
```

第一行是协议名称及版本号，空格后面会有一个三位数的数字，是HTTP协议的响应状态码，200表示成功，OK是对状态码的简短文字描述。状态码共有5类：

1xx属于通知类；

2xx属于成功类；

3xx属于重定向类；

4xx属于客户端错误类；

5xx属于服务端错误类。

对于状态码，相信大家对404应该很熟悉，如果向一个服务器请求一个不存在的文件，就会得到该错误，通常浏览器也会显示类似“HTTP 404 – 未找到文件”这样的错误。

第二行Content-Length字段是一个比较重要的字段，它标明了服务器返回数据的长度，这个长度是不包含HTTP头长度的。换句话说，我们的请求中并没有Range字段（后面会说到），表示我们请求的是整个文件，所以Content-Length就是整个文件的大小。其余各字段是一些关于文件和服务器的属性信息。

这段返回数据同样是以最后一行的结束标志（回车换行）和一个额外的回车换行作为结束，即“\r\n\r\n”。

**而“\r\n\r\n”后面紧接的就是文件的内容了**，这样我们就可以找到“\r\n\r\n”，并从它后面的第一个字节开始，源源不断的读取，再写到文件中了。

以上就是通过HTTP协议实现文件下载的全过程。但还不能实现断点续传，而实际上断点续传的实现非常简单，只要在请求中加一个Range字段就可以了。

假如一个文件有1000个字节，那么其范围就是0-999，则：

- Range: bytes=500- 表示读取该文件的500-999字节，共500字节。
- Range: bytes=500-599 表示读取该文件的500-599字节，共100字节。

如果HTTP请求中包含Range字段，那么服务器会返回206（Partial Content），同时HTTP头中也会有一个相应的Content-Range字段，类似下面的格式：

```
Content-Range: bytes 500-999/1000
```

Content-Range字段说明服务器返回了文件的某个范围及文件的总长度。这时Content-Length字段就不是整个文件的大小了，而是对应文件这个范围的字节数，这一点一定要注意。

## 4 文件上传下载实现

### 4.0 安装nodejs

#### 1. 下载nodejs

官方：<https://nodejs.org/en/download/>

如图：




## Downloads

Latest LTS Version: **16.14.0** (includes npm 8.3.1)

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

**LTS**  
Recommended For Most Users

**Current**  
Latest Features

 <b>Windows Installer</b> node-v16.14.0-x64.msi	 <b>macOS Installer</b> node-v16.14.0.pkg	 <b>Source Code</b> node-v16.14.0.tar.gz
--	--	---

Windows Installer (.msi)	32-bit	64-bit
Windows Binary (.zip)	32-bit	64-bit
macOS Installer (.pkg)	64-bit / ARM64	
macOS Binary (.tar.gz)	64-bit	ARM64
Linux Binaries (x64)	64-bit	
Linux Binaries (ARM)	ARMv7	ARMv8
Source Code	node-v16.14.0.tar.gz	

复制下载链接后，从命令行下载:

▼ | C | 复制代码

```
1 wget https://nodejs.org/dist/v16.14.0/node-v16.14.0-linux-x64.tar.xz
```

## 2. 解压node安装包

▼ | C | 复制代码

```
1 xz -d node-v16.14.0-linux-x64.tar.xz
2 tar xf node-v16.14.0-linux-x64.tar
```

## 3. 创建符号链接，供直接从命令行访问无需输入路径（/root/.voice/node-v16.14.0-linux-x64路

径是自己的路径，不要照抄)

C | 复制代码

```
1 ln -s /root/0voice/node-v16.14.0-linux-x64/bin/node /usr/local/bin/node
2 ln -s /root/0voice/node-v16.14.0-linux-x64/bin/npm /usr/local/bin/npm
3
4 打印版本:
5 ~/0voice/node-v16.14.0-linux-x64/bin# node -v
6 v16.14.0
7
8 ~/0voice/node-v16.14.0-linux-x64/bin# npm -v
9 8.3.1
10
```

请输入代码块名称

全名路径，不要写相对路径

C v

Yuque Light v

...

```
1 ln -s /root/0voice/node-v16.14.0-linux-x64/bin/node /usr/local/bin/node
2 ln -s /root/0voice/node-v16.14.0-linux-x64/bin/npm /usr/local/bin/npm
3
4 打印版本:
5 ~/0voice/node-v16.14.0-linux-x64/bin# node -v
6 v16.14.0
7
8 ~/0voice/node-v16.14.0-linux-x64/bin# npm -v
9 8.3.1
10
```

ln -sf

参考

见课程源码分析

## 4.1 普通上传

<https://www.axios-http.cn/> Axios 是一个基于 promise 的网络请求库，可以用于浏览器和 node.js

当前图床项目web上传文件格式

```

1  -----WebKitFormBoundary5hBDa6WqVJEF946U
2  Content-Disposition: form-data; name="file"; filename="tuchuangtest.txt"
3  Content-Type: text/plain
4
5  123456789
6  -----WebKitFormBoundary5hBDa6WqVJEF946U
7  Content-Disposition: form-data; name="user"
8
9  qingfuliao
10 -----WebKitFormBoundary5hBDa6WqVJEF946U
11 Content-Disposition: form-data; name="md5"
12
13 25f9e794323b453885f5181f1b624d0b
14 -----WebKitFormBoundary5hBDa6WqVJEF946U
15 Content-Disposition: form-data; name="size"
16
17 9
18 -----WebKitFormBoundary5hBDa6WqVJEF946U--

```

## 自定义的格式

客户端的上传格式：1-simpleupload.html

```

1  formData.append('user', 'qingfuliao')
2  formData.append('token', '123xxx') // 设置token
3  formData.append('filesize', file.size.toString()) // 设置文件大小
4  formData.append('file', file)

```

启动： node 1-serv\_simpleupload.js

服务端1-serv\_simpleupload.js打印的内容：

```
1  -----WebKitFormBoundary07eSVNWqgP1XNvW0
2  Content-Disposition: form-data; name="user"
3
4  qingfuliao
5  -----WebKitFormBoundary07eSVNWqgP1XNvW0
6  Content-Disposition: form-data; name="token"
7
8  123xxx
9  -----WebKitFormBoundary07eSVNWqgP1XNvW0
10 Content-Disposition: form-data; name="filesize"
11
12 9
13 -----WebKitFormBoundary07eSVNWqgP1XNvW0
14 Content-Disposition: form-data; name="file"; filename="tuchuangtest.txt"
15 Content-Type: text/plain
16
17 123456789
18 -----WebKitFormBoundary07eSVNWqgP1XNvW0--
```

## 4.2 分片上传

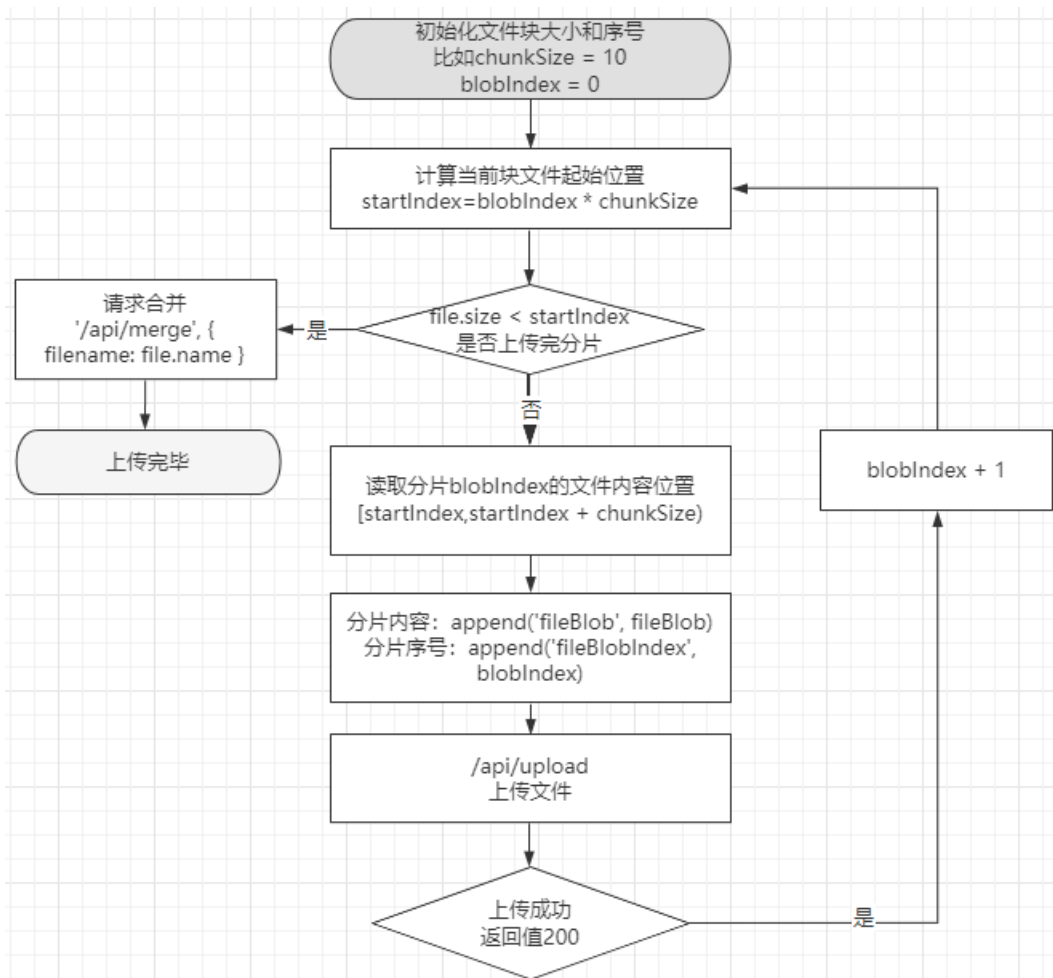
后端保存时以fileBlob为分片根目录，以文件名创建一个文件夹，我们把分片都保存在分片对应的文件名创建的文件夹下。

最终合并的文件保存到fileSave目录。

客户端：2-pieceupload.html

服务端：2-serv\_pieceupload.js





客户端上传分片

## 服务端临时文件夹



## 4.3 断点续传

分片上传已经实现将一个大的文件分割成多个切块储存，在所有切块都上传完成后在服务端进行一次合并操作。那么假如在上传切块的过程中因断网、页面崩溃等原因导致上传过程被迫中断，这时其实服务端已经储存了部分切块数据了，下次再次上传该文件时我们可以不再重复上传已经保存好的切块，而只上传服务端没有的切块，这样就避免了重复上传的麻烦，这就是断点续传。

要实现断点续传我们在上传之前需要知道服务端已经储存了哪些切块，所以可以先发送一个 `/api/check` 请求来进行检查

`/api/check` 请求的目的是通过检查，让服务端告诉前端当前文件是否上传过，是否上传完成，如果没上传完成，上传到第几个切块

```
1  const params = { filename: file.name }
2  const resCheck = await axios.get('/api/check', { params })
3  console.log(resCheck)
```

## 4.4 多线程下载

见http-download 源码。

1. 先获取文件大小 `CDownload::GetSize()`
2. 根据线程数量划分每个线程要下载的文件长度
3. `download_write_cb`负责文件的写入

## 异常处理

```
✖ Access to XMLHttpRequest at 'http://192.168.206.135:3000/api/upload' from origin 'http://127.0.0.1:5500' has been blocked by CORS policy: The 'Access-Control-Allow-Origin' header has a value 'http://localhost:8080' that is not equal to the supplied origin. 2-pieceupload.html:1
✖ POST http://192.168.206.135:3000/api/upload net::ERR_FAILED 200 xhr.js:178
✖ Uncaught (in promise) Error: Network Error
    at createError (createError.js:17:1)
    at XMLHttpRequest.handleError (xhr.js:84:1) createError.js:17
```

html添加

```
<cross-domain-policy>
```

```
<allow-access-from domain="*" />
```

```
</cross-domain-policy>
```

## 5 参考引申

前端采用百度提供的webuploader的插件，<http://fex.baidu.com/webuploader/getting-started.html>

FastDFS-Nginx扩展模块源码分析 <https://www.cnblogs.com/littleatp/p/4361318.html>

面试官：大文件上传如何做断点续传？

[https://blog.csdn.net/weixin\\_44475093/article/details/115191898](https://blog.csdn.net/weixin_44475093/article/details/115191898)