link: null
title: 珠峰架构师成长计划
description: HTTP全称是超文本传输协议，构建于TCP之上，属于应用层协议。
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=58 sentences=162, words=841

1. HTTP服务器

HTTP全称是超文本传输协议，构建于TCP之上，属于应用层协议。

```
let server  = http.createServer([requestListener]);
server.on('request',requestListener);
```

- requestListener 当服务器收到客户端的连接后执行的处理
    - http.IncomingMessage 请求对象
    - http.ServerResponse对象 服务器端响应对象

```
server.listen(port,[host],[backlog],[callback]);
server.on('listening',callback);
```

- port 监听的端口号
- host 监听的地址
- backlog 指定位于等待队列中的客户端连接数

```
let http = require('http');
let server = http.createServer(function(req,res){
}).listen(8080,'127.0.0.1',function(){console.log('服务器端开始监听!')});
```

```
server.close();
server.on('close',function(){});
```

```
let http = require('http');
let server = http.createServer(function(req,res){
});
server.on('close',function(){
    console.log('服务器关闭');
});
server.listen(8080,'127.0.0.1',function(){
    console.log('服务器端开始监听!')
    server.close();
});
```

```
server.on('error',function(){
    if(e.code == 'EADDRINUSE'){
        console.log('端口号已经被占用!);
    }
});
```

```
let server = http.createServer(function(req,res){
});
server.on('connection',function(){
    console.log(客户端连接已经建立);
});
```

设置超时时间，超时后不可再复用已经建立的连接，需要发请求需要重新建立连接。默认超时时间时2分钟

```
server.setTimeout(msecs,callback);
server.on('timeout',function(){
    console.log('连接已经超时');
});
```

- request
    - method 请求的方法
    - url 请求的路径
    - headers 请求头对象
    - httpVersion 客户端的http版本
    - socket 监听客户端请求的socket对象

```
let http = require('http');
let fs = require('fs');
let server = http.createServer(function(req,res){
if(req.url != '/favicon.ico'){
let out = fs.createWriteStream(path.join(__dirname,'request.log'));
out.write('method='+req.method);
out.write('url='+req.url);
out.write('headers='+JSON.stringify(req.headers));
out.write('httpVersion='+req.httpVersion);
}
}).listen(8080,'127.0.0.1);
```

```
let http = require('http');
let fs = require('fs');
let server = http.createServer(function(req,res){
  let body = [];
  req.on('data',function(data){
    body.push(data);
  });
  req.on('end',function(){
      let result = Buffer.concat(body);
      console.log(result.toString());
  });
}).listen(8080,'127.0.0.1);
```

querystring模块用来转换URL字符串和URL中的查询字符串

```
querystring.parse(str,[sep],[eq],[options]);
```

```
querystring.stringify(obj,[sep],[eq]);
```

```
url.parse(urlStr,[parseQueryString]);
```

- href 被转换的原URL字符串
- protocal 客户端发出请求时使用的协议
- slashes 在协议与路径之间是否使用了//分隔符
- host URL字符串中的完整地址和端口号
- auth URL字符串中的认证部分
- hostname URL字符串中的完整地址
- port URL字符串中的端口号
- pathname URL字符串的路径，不包含查询字符串
- search 查询字符串，包含?
- path 路径，包含查询字符串
- query 查询字符串，不包含起始字符串 ?
- hash 散列字符串，包含 #

http.ServerResponse对象表示响应对象

```
response.writeHead(statusCode,[reasonPhrase],[headers]);
```

- content-type 内容类型
- location 将客户端重定向到另外一个URL地址
- content-disposition 指定一个被下载的文件名
- content-length 服务器响应内容的字节数
- set-cookie 在客户端创建Cookie
- content-encoding 指定服务器响应内容的编码方式
- cache-cache 开启缓存机制
- expires 用于制定缓存过期时间
- etag 指定当服务器响应内容没有变化不重新下载数据

设置、获取和删除Header

```
response.setHeader('Content-Type','text/html;charset=utf-8');
response.getHeader('Content-Type');
response.removeHeader('Content-Type');
response.headersSent 判断响应头是否已经发送
```

判断响应头是否已经发送

```
let http = require('http');
let server = http.createServer(function(req,res){
  console.log(resopnse.headersSent?"响应头已经发送":"响应头未发送!");
  res.writeHead(200,'ok);
  console.log(resopnse.headersSent?"响应头已经发送":"响应头未发送!");
});
```

不发送Date

```
res.sendDate = false;
```

可以使用write方法发送响应内容

```
response.write(chunk,[encoding]);
response.end([chunk],[encoding]);
```

可以使用setTimeout方法设置响应让超时时间，如果在指定时间内不响应，则触发timeout事件

```
response.setTimeout(msecs,[callback]);
response.on('timeout',callback);
```

在响应对象的end方法被调用之前，如果连接中断，将触发http.ServerResponse对象的close事件

```
response.on('close',callback);
```

```
net
onconnection

_http_server.js
连接监听
connectionListenerInternal
socketOnData
onParserExecuteCommon
parserOnIncoming
```

2. HTTP客户端

```
let req = http.request(options,callback);
req.on('request',callback);
request.write(chunk,[encoding]);
request.end([chunk],[encoding]);
```

- host 指定目标域名或主机名
- hostname 指定目标域名或主机名，如果和host都指定了，优先使用hostname
- port 指定目标服务器的端口号
- localAddress 本地接口
- socketPath 指定Unix域端口
- method 指定HTTP请求的方式
- path 指定请求路径和查询字符串
- headers 指定客户端请求头对象
- auth 指定认证部分
- agent 用于指定HTTP代理，在Node.js中，使用http.Agent类代表一个HTTP代理，默认使用keep-alive连接，同时使用http.Agent对象来实现所有的HTTP客户端请求

```
let http = require('http');
let options = {
    hostname: 'localhost',
    port: 8080,
    path: '/',
    method: 'GET'
}
let req = http.request(options, function (res) {
    console.log('状态吗:' + res.statusCode);
    console.log('响应头:' + JSON.stringify(res.headers));
    res.setEncoding('utf8');
    res.on('data', function (chunk) {
        console.log('响应内容', chunk);
    });
});
req.end();
```

可以使用abort方法来终止本次请求

```
req.abort();
```

如果请求过程中出错了，会触发error事件

```
request.on('error',function(err){});
```

建立连接过程中，为该连接分配端口时，触发 socket事件

```
req.on('socket',function(socket){
  socket.setTimeout(1000);
  socket.on('timeout',function(){req.abort()});
});
```

可以使用get方法向服务器发送数据

```
http.get(options,callback);
```

可以使用response对象的addTrailers方法在服务器响应尾部追加一个头信息

```
let http = require('http');
let path = require('path');
let crypto = require('crypto');

let server = http.createServer(function (req, res) {
    res.writeHead(200, {
        'Transfer-Encoding': 'chunked',
        'Trailer': 'Content-MD5'
    });
    let rs = require('fs').createReadStream(path.join(__dirname, 'msg.txt'), {
        highWaterMark: 2
    });
    let md5 = crypto.createHash('md5');
    rs.on('data', function (data) {
        console.log(data);
        res.write(data);
        md5.update(data);
    });
    rs.on('end', function () {
        res.addTrailers({
            'Content-MD5': md5.digest('hex')
        });
        res.end();
    });
}).listen(8080);
```

```
let http = require('http');
let options = {
    hostname: 'localhost',
    port: 8080,
    path: '/',
    method: 'GET'
}
let req = http.request(options, function (res) {
    console.log('状态吗:' + res.statusCode);
    console.log('响应头:' + JSON.stringify(res.headers));
    res.setEncoding('utf8');
    res.on('data', function (chunk) {
        console.log('响应内容', chunk);
    });
    res.on('end', function () {
        console.log('trailer', res.trailers);
    });
});
req.end();
```

```
let http = require('http');
let url = require('url');
let server = http.createServer(function (request, response) {
    let {
        path
    } = url.parse(request.url);
    let options = {
        host: 'localhost',
        port: 9090,
        path: path,
        headers: request.headers
    }
    let req = http.get(options, function (res) {
        console.log(res);
        response.writeHead(res.statusCode, res.headers);
        res.pipe(response);
    });
    req.on('error', function (err) {
        console.log(err);
    });
    request.pipe(req);
}).listen(8080);
```