网络编程

主要介绍 http https http2 websocket

OSI模型

OSI模型	实际应用
7 应用层	应用层: TELNET,SSH,HTTP,SMTP,POP,SSL/TLS,FTP,MIME,HTML,SNMP,MIB,SIP,RTP...
6 表示层	
5 会话层	
4 传输层	传输层: TCP,UDP,UDP-Lite,SCTP,DCCP
3 网络层	网络层: ARP,IPv4,IPv6,ICMP,IPsec
2 数据链路层	以太网，无线LAN，PPP... (双文线电缆、无线、光纤...)
1 物理层	

http协议

http协议是基于请求响应的无状态协议，可以使用curl发送http请求，看一些信息

```
# 向baidu.com发送http请求
curl -v http://www.baidu.com
```

执行结果

```
kevindeMacBook-Air:ts_init kevin$ curl -v http://www.baidu.com
* Rebuilt URL to: http://www.baidu.com/
*   Trying 14.215.177.38...
* TCP_NODELAY set
* Connected to www.baidu.com (14.215.177.38) port 80 (#0)
> GET / HTTP/1.1
> Host: www.baidu.com
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Accept-Ranges: bytes
< Cache-Control: private, no-cache, no-store, proxy-revalidate, no-transform
< Connection: keep-alive
< Content-Length: 2381
< Content-Type: text/html
< Date: Sat, 11 Jan 2020 14:17:27 GMT
< Etag: "588604dd-94d"
< Last-Modified: Mon, 23 Jan 2017 13:27:57 GMT
< Pragma: no-cache
```

```
< Server: bfe/1.0.8.18
< Set-Cookie: BDORZ=27315; max-age=86400; domain=.baidu.com; path=/
<
<!DOCTYPE html>
<!--STATUS OK--><html> <head><meta http-equiv=content-type
content=text/html;charset=utf-8><meta http-equiv=X-UA-Compatible
content=IE=Edge><meta content=always name=referrer><link rel=stylesheet
type=text/css href=http://s1.bdstatic.com/r/www/cache/bdorz/baidu.min.css>
<title>百度一下，你就知道</title></head> <body link=#0000cc> <div id=wrapper> <div
id=head> <div class=head_wrapper> <div class=s_form> <div class=s_form_wrapper>
<div id=lg> <img hidefocus=true src=//www.baidu.com/img/bd_logo1.png width=270
height=129> </div> <form id=form name=f action=//www.baidu.com/s class=fm>
<input type=hidden name=bdorz_come value=1> <input type=hidden name=ie
value=utf-8> <input type=hidden name=f value=8> <input type=hidden name=rsv_bp
value=1> <input type=hidden name=rsv_idx value=1> <input type=hidden name=tn
value=baidu><span class="bg s_ipt_wr"><input id=kw name=wd class=s_ipt value
maxlength=255 autocomplete=off autofocus></span><span class="bg s_btn_wr"><input
type=submit id=su value=百度一下 class="bg s_btn"></span> </form> </div> </div>
<div id=u1> <a href=http://news.baidu.com name=tj_trnews class=mnav>新闻</a> <a
href=http://www.hao123.com name=tj_trhao123 class=mnav>hao123</a> <a
href=http://map.baidu.com name=tj_trmap class=mnav>地图</a> <a
href=http://v.baidu.com name=tj_trvideo class=mnav>视频</a> <a
href=http://tieba.baidu.com name=tj_trtieba class=mnav>贴吧</a> <noscript> <a
href=http://www.baidu.com/bdorz/login.gif?
login&amp;tpl=mn&amp;u=http%3A%2F%2Fwww.baidu.com%2f%3fbdorz_come%3d1
name=tj_login class=lb>登录</a> </noscript> <script>document.write('<a
href="http://www.baidu.com/bdorz/login.gif?login&tpl=mn&u='+
encodeURIComponent(window.location.href+ (window.location.search === "" ? "?" :
"&")+ "bdorz_come=1")+ '" name="tj_login" class="lb">登录</a>');</script> <a
href=//www.baidu.com/more/ name=tj_briicon class=bri style="display: block;">更多
产品</a> </div> </div> </div> <div id=ftCon> <div id=ftConw> <p id=lh> <a
href=http://home.baidu.com>关于百度</a> <a href=http://ir.baidu.com>About
Baidu</a> </p> <p id=cp>&copy;2017 Baidu <a
href=http://www.baidu.com/duty/>使用百度前必读</a>  <a
href=http://jianyi.baidu.com/ class=cp-feedback>意见反馈</a> 京ICP证030173号
  <img src=//www.baidu.com/img/gs.gif> </p> </div> </div> </div> </body>
</html>
* Connection #0 to host www.baidu.com left intact
kevindeMacBook-Air:ts_init kevin$
```

特点

无连接、无状态（http协议本身无法保存鉴权、登录状态）、简单快速、灵活

请求部分(request)

上行，curl > 部分内容，向服务器发送请求信息

请求行

- Method `GET`
    - GET 请求获取url所标识的资源
    - POST 在url所标识的资源里附加新的数据
    - HEAD 请求获取url所标识资源的响应消息报头，请求回应的部分里，http头部信息与通过get请求所得到的信息一致，利用head，不必传输整个资源内容，就可以获取url所标识资源的信息，常用于测试超链接的有效性，是否可以访问，以及是否有更新

    - PUT 请求服务器存储一个资源，并用url作为标识
    - DELETE 请求服务器删除url所标识的资源
    - TRACE 请求服务器回送收到的请求消息，主要用于测试或诊断
    - CONNECT 保留将来使用
    - OPTIONS 请求查询服务器的性能，或者查询与资源相关选项和需求
- RequestUrl `www.baidu.com`
- HttpVersion `/ HTTP/1.1`

消息包头

- Accept 指定客户端接受那些类型的消息/MIME
    - text/html html文本
    - image/gif gif图片
- Accept-Chartset 客户端接受的字符集
    - gb2312 中文字符
    - iso-8859-1 西文字符
    - utf-8 多语言字符
- Accept-Encoding 可接受的内容编码
    - gzip,deflate 压缩类型
    - identify 默认
- Accept-Language 浏览器语言 zh-cn
- Authorization 证明客户端有权查看某个资源（已经很少用了）
- Host 指定被请求资源的internet主机和端口号 www.baidu.com:8080
- User-Agent 用户代理 `curl/7.54.0`
    - 操作系统及版本
    - CPU类型
    - 浏览器版本
    - 浏览器渲染引擎
    - 浏览器语言
    - 浏览器插件
- Content-Type Body（请求正文）的编码方式

请求正文

根据头部的Content-Type确定

- application/x-www-form-urlencoded
    - `title=test&a=2`
    - 默认的数据编码方式
- multipart/form-data
    - 既有文本数据，又有文件等二进制数据
    - 允许在数据中包含整个文件，常用于文件上传
- application/json 序列化后的JSON字符串，ajax
- text/xml xml作为编码方式的远程调用规范
- text/plain 纯文本数据

响应部分(response)

下行，curl < 部分内容，从服务器接收数据

状态行

HTTP协议版本(HTTP-Version) 状态码(Status-code) 状态码文本描述(Reason-Phrase) CRLF

< HTTP/1.1 200 OK

状态码

- 1xx 指示信息 -- 表示请求已接收， 继续处理
- 2xx 成功 -- 表示请求已被成功接收，理解、接受
  - 200 OK 请求成
  - 200 客户端发送一个带Range头的GET请求 服务器完成
- 3xx 重定向 -- 一般用于url变更，为了seo或不影响原入口，当访问原url时重定向到新的url
  - 301 表示资源永久性移动到新URL
  - 302 表示资源暂时重定向到新URL
- 4xx 客户端错误 -- 请求有语法错误或请求无法实现
  - 400 Bad Request 客户端请求有语法错误，不能被服务器所理解
  - 401 Unauthorized 请求未经授权
  - 403 Forbidden 服务器收到请求，但拒绝提供服务
  - 404 Not Found 请求资源部存在，比如输错了url
- 5xx 服务端错误 -- 服务端未能实现合法的请求
  - 500 Internal Server Error 服务器发生不可预期的错误
  - 503 Server Unavailable 服务器当前不能处理客户端请求，一段时间后可能恢复正常

消息报头

- 响应报头
  - Location 重定向接受者到一个新的位置
  - WWW-Authenticate 包含在401(未授权)响应消息中，客户端接收到401响应时，并发送Authorization报头域请求服务器对其进行验证时，服务端响应报文就包含该报头域。
  - Server 包含了服务器用来处理请求的软件信息 如：Apache-Coyote/1.1
- 实体报头
- Content-Encoding 媒体类型的修饰符 gzip
- Content-Language 资源所用的语言
- Content-Length 正文的长度，以字节方式存储的十进制数字表示
- Conent-Type 正文媒体类型
  - text/html
  - application/json
- Expirse 响应过期日期和时间 为了让代理服务器或浏览器在一段时间以后更新缓存数据

响应正文

服务器返回的资源、内容

GET和POST区别

- GET回退无害，POST会再次提交
- GET产生URL地址收藏，POST不可以起
- GET请求会被浏览器主动缓存
- GET请求需要URL编码
- GET请求长度限制

- GET参数通过URL传递，POST在request Body中

创建接口

```
// app.js
const http = require('http')
const fs =require('fs')
http.createServer((req, res) => {
  const { method, url } = req
  console.log(url)
  if (method === 'GET' && url === '/') {
    fs.readFile('./index.html', (err, data) => {
      res.setHeader('Content-Type', 'text/html')
      res.end(data)
    })
  } else if (mtthod === 'GET' && url === '/api/users') {
    res.setHeader('Content-Type', 'application/json')
    res.end(JSON.stringify([{name: 'Tom', age: 20}]))
  }
}).listen(3000, () => {
  console.log('服务已开启与3000端口')
})
```

index.html 里使用axios请求接口，通过http://127.0.0.1:3000 可以访问

```
<body>
  <script src="https://unpkg.com/axios/dist/axios.min.js"></script>
  <script>
    (async () => {
      let res = await axios.get('/api/users')
      console.log(JSON.stringify(res, null, 2))

      // 一种埋点方式
      let img = new Image()
      img.src="/api/users?button=123"
    })()
    // res 返回的数据
    // {
    //   "data": [
    //     {
    //       "name": "Tom",
    //       "age": 20
    //     }
    //   ],
    //   "status": 200,
    //   "statusText": "OK",
    //   "headers": {
    //     "connection": "keep-alive",
    //     "content-length": "25",
    //     "content-type": "application/json",
    //     "date": "Sun, 12 Jan 2020 06:27:54 GMT"
    //   },
    //   "config": {
    //     "url": "/api/users",
    //     "method": "get",
    //     "headers": {
```

```
    //       "Accept": "application/json, text/plain, */*"
    //     }
    //     ...
    //   }
    // }
  </script>
</body>
```

前端跨域问题

http://127.0.0.1:3000，跨域是浏览器同源策略引起的接口调用问题，只针对XMLHttpRequest发出的请求

- 协议 `http`
- host `127.0.0.1`
- 端口 `3000`

协议、端口、host 三者有一个不同就会跨域，假设接口地址为 http://127.0.0.1/api/users 下面的三种请求都是跨域:

- 前端发出的请求URL为 https://127.0.0.1/api/users `协议不同`
- 前端发出的请求URL为 https://192.168.1.2/api/users `host不同`
- 前端发出的请求URL为 http://127.0.0.1:3000/api/users `端口不同，默认端口为80`

CORS 跨域资源共享

Cross Origin Resource Sharing, 后端设置响应头，参考之前的笔记：https://www.yuque.com/guoqzuo/js_es6/fcw53h#ac35dda4

```
// 如果是4000端口发过来的请求，允许跨域
res.setHeader('Access-Control-Allow-Origin', 'http://127.0.0.1:4000')
// 允许所有跨域
res.setHeader('Access-Control-Allow-Origin', '*')
```

其他跨域JSONP与img

使用script标签或img标签以加载资源的方式来发送请求，而非xhr(XMLHttpRequest)请求，所以不会跨域, 参考之前的笔记：https://www.yuque.com/guoqzuo/js_es6/fcw53h#JSONP

```
<body>
  <!-- 客户端代码 -->
  <input id="jsonpclick" type="button" value="jsonp test">
  <script>
    function handleRes(response) {
      console.log(response)
      // 这里可以接收到对应的数据
    }
    var jsonpclick = document.getElementById('jsonpclick');
    jsonpclick.onclick = function(){
      console.log('开始测试');
      var script = document.createElement('script');
      script.type="text/javascript"
      script.src = "http://127.0.0.1:8088/gzh_test?callback=handleRes"
      document.body.insertBefore(script, document.body.firstChild)
    }
```

```
  </script>
</body>
```

node服务端代码

```
function gzhM_test(app, data, req, res) {
    console.log('开始执行gzhm_test');
    console.log(req.query)
    if (req.query && req.query.callback) {
        //console.log(params.query.callback);
        var str =  req.query.callback + '(' + "a=2" + ')';//jsonp
        res.end(str);
    } else {
        res.end('b=2');//普通的json
    }
    return;
}
```

preflight 预检请求

/priːˈflaɪt/, 在CORS跨域方法中，设置了允许跨域，就可以跨域访问了，现在在请求里加一个请求头试试

```
(async () => {
  axios.defaults.baseURL = 'http://127.0.0.1:3000';
  let res = await axios.get('/api/users', {headers: {'X-Token': 'test'}})
  console.log(JSON.stringify(res, null, 2))
})()
```

加了请求头后，发现不能跨域了，请求一直在pedding，这就涉及到CORS的 preflight（预检）了。在跨域时，有些情况会发送两次请求，第一次为预检，请求方式为OPTIONS，第二次才是带真实数据的请求

为什么会有预检请求

出于安全考虑，浏览器有同源策略，会限制跨域的请求，浏览器限制跨域有两种方式：

1. 浏览器限制发起跨域请求
2. 跨域请求可以正常发起，但返回的结果被浏览器拦截了

一般浏览器都是使用第二种方式限制跨域请求，跨域请求已经到达服务器，并可能对数据库里的数据进行了操作，但返回的结果被浏览器拦截了，对前端来讲这是一次失败的请求，但可能对数据库里的数据产生了影响，为了防止这种情况发生，对于可能对服务器数据产生副作用的HTTP请求方法，浏览器必先使用OPTIONS方法发起一个预检请求，从而获知服务器是否允许跨域请求：如果允许，就发送带真实的数据请求，如果不允许，则阻止带数据的真实请求。

什么情况会发触发CORS预检请求

- 使用了PUT、DELETE、CONNECT、OPTIONS、TRACE、PATCH请求方法
- 人为设置了CORS安全的请求头之外的其他请求头，下面是安全的请求头列表
    - Accept
    - Accept-Language
    - Content-Language
    - Content-Type

- DPR
- Downlink
- Save-Data
- Viewport-Width
- Width
- Content-Type值为 application/x-www-form-urlencoded、multipart/form-data、text/plain

加上对CORS预检请求的处理

上面的例子中，加了一个CORS非安全的请求头: `X-Token`，所以发触发CORS预检，但我们并没有允许预检的OPTIONS请求跨域，所以需要加上对应的处理：

```
const http = require('http')
const fs =require('fs')
http.createServer((req, res) => {
  const { method, url } = req
  console.log(url)
  if (method === 'GET' && url === '/') {
    fs.readFile('./index.html', (err, data) => {
      res.setHeader('Content-Type', 'text/html')
      res.end(data)
    })
  } else if ((method === 'GET' || method === 'POST') && url === '/api/users') {
    // 如果是4000端口发过来的请求，允许跨域
    // res.setHeader('Access-Control-Allow-Origin', 'http://127.0.0.1:4000')
    res.setHeader('Access-Control-Allow-Origin', '*')
    res.setHeader('Content-Type', 'application/json')
    res.setHeader('Set-Cookie', 'cookie1=test')
    res.end(JSON.stringify([{name: 'Tom', age: 20}]))
  } else if (method === 'OPTIONS' && url === '/api/users') {
    // 预检
    res.writeHead(200, {
      'Access-Control-Allow-Origin': 'http://127.0.0.1:4000',
      'ACCESS-Control-Allow-Headers': 'X-token',
      // 'ACCESS-Control-Allow-Headers': 'X-token,Content-Type',
      // 'Access-Control-Allow-Methods': 'PUT'
    })
    res.end()
  }
}).listen(3000, () => {
  console.log('服务已开启与3000端口')
})
```

如果我们把请求改为POST，axios发送json数据时，Cotent-Type为application/json，非CORS安全请求头，需要允许对应的请求头

```
'ACCESS-Control-Allow-Headers': 'X-token,Content-Type',
// 或者
'ACCESS-Control-Allow-Headers': '*'
```

如果请求携带cookie信息，则请求变为credential请求

```
// axios 跨域请求 默认不发送cookie，如果想发送的请求携带cookie需要将withCredentials设置为true
// `withCredentials` indicates whether or not cross-site Access-Control requests
// should be made using credentials
// withCredentials: false, // default
axios.defaults.withCredentials = true // 允许跨域请求发送cookie

// OPTIONS需要加一个设置
res.setHeader('Access-Control-Allow-Credentials', 'true')

// 在接口响应时，写入Cookie，之后该域下每次发送的请求都会携带这个cookie
res.setHeader('Set-Cookie', 'cookie1=test')
// 之后发送的请求，请求头里会多出 Cookie: cookie1=test 这一项
console.log(req.headers.cookie) // cookie1=test
```

参考：前端 | 浅谈preflight request

服务器代理

除了上面将的跨域方法外，还有使用代理服务器的方法来跨域。就是请求同源服务器，通过该服务器转发请求到目标服务器，得到结果再转发给前端。

webpack，vue.config.js里面 devserver配置中就是使用这种方法。在测试时，使用代理。

正向代理与反向代理

> 参考：https://www.cnblogs.com/xuepei/p/10437114.html

正向代理：很早之前，网络带宽很小，64k 网络专线，那么多电脑想要访问网络，采用的方式是，统一访问一台代理服务器来上网。客户端通过代理服务器访问目标服务器内容，就是正向代理。比如科学上网，对于墙屏蔽的网站，我们会采用ss代理服务器来访问。这就是正向代理。

反向代理：对于比较大流量的站点，一台服务器顶不住，需要在前面有个代理服务器，将请求分发代理到其他服务器来处理。一台服务器将前端发送的请求，转发到另一台服务器处理，再将处理好的数据传给前端，这种情况就是反向代理。一般用于负载均衡。

正向代理与反向代理的区别：

- 正向代理是客户端的代理，帮助客户端访问其无法访问的服务器资源。反向代理则是服务器代理，帮助服务器做负载均衡、安全防护等。
- 正向代理一般是客户端架设的，比如在电脑上装一个ss客户端。反向代理一般在服务端架设，比如在集群中部署一个反向代理服务器
- 正向代理中服务端不知道真正的客户端是谁，以为访问自己的就是真实的客户端。反向代理中，客户端不知道真正的服务器是谁，以为自己访问的就是真实的服务器
- 正向代理和反向代理的作用和目的不同：正向代理主要用于解决访问限制问题，反向代理主要提供负载均衡，安全防护功能。二者均能提高访问速度。

利用反向代理跨域

```
# 使用 http-proxy-middleware 将请求代理到目标服务器
npm i http-proxy-middleware --save-dev
```

示例代码

```
const express = require('express')
const app = new express()
const proxy = require('http-proxy-middleware')
// webpack devServer里proxy就是使用的这个包

app.use(express.static(__dirname + '/'))
app.use('/api', proxy({
  target: 'http://127.0.0.1:3000',
  changeOrigin: true
}))

app.listen(4000)
```

webpack devServer vue.config.js里面的配置

```
module.export = {
  devServer: {
    disableHostCheck: true,
    compress: true,
    port: 5000,
    proxy: {
      '/api': {
        tartget: 'http://127.0.0.1:4000',
        changeOrigin: true
      }
    }
  }
}
```

nginx 代理配置

```
server {
  listen 80;
  location / {
    root /var/www/html;
    index index.html index.htm;
    try_files $uri $uri/ /index.html
  }
}

location /api {
  proxy_pass http://127.0.0.1:3000;
  proxy_redirect off;
  proxy_set_header Host $host;
  proxy_set_header X-Real-IP $remote_addr;
  proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
}
```

Bodyparser

为什么需要使用bodyparser，bodyparser是用来做什么的？先看看一个例子

node接收post发送的数据

node接收post发送的数据时，默认处理方式

```
// 前端发送post请求 index.html
// <form action="/api/save" method="POST">
//   <input name="title" value="abc">
//   <input type="submit" value="save">
// </form>
const Koa = require('koa')
const app = new Koa()

// 静态服务，让访问 127.0.0.1:3000时可以直接访问index
app.use(require('koa-static')(__dirname + '/'))

app.use((ctx, next) => {
  const { req } = ctx.request
  let reqData = []
  let size = 0

  // 处理post请求发送的数据
  req.on('data', data => {
    console.log('>>>req on', data) // >>>req on <Buffer 74 69 74 6c 65 3d 61 62
63>
    reqData.push(data)
    size += data.length
  })

  req.on('end', () => {
    console.log('end')
    const data = Buffer.concat(reqData, size)
    console.log('data:', size, data.toString()) //  data: 9 title=abc
  })
})

app.listen(3000, () => { console.log('服务开启于3000端口') })
```

bodyparser中间件

bodyparser就是将获取post请求内容封装成了一个中间件，后面就可以直接通过ctx.request.body就可以拿到post请求的数据了

```
const Koa = require('koa')
const app = new Koa()

// 静态服务，让访问 127.0.0.1:3000时可以直接访问index
app.use(require('koa-static')(__dirname + '/'))

app.use(require('koa-bodyparser')())
app.use((ctx, next) => {
  console.log(ctx.request.body) // { title: 'abc' }
  ctx.body = ctx.request.body
})

app.listen(3000, () => { console.log('服务开启于3000端口') })
```

axios发送数据Content-Type

```
// application/json
axios.post('/api/save', {a: 1, b: 2})

// application/x-www-form-urlencoded
axios.post('/api/save', 'a=1&n=2', {
  headers: {
    'Content-Type': 'aplication/x-www-form-urlencoded'
  }
})
```

文件上传与下载

主要介绍node处理文件的上传和下载

文件下载

前端demo

```
<a href="/api/download" target="_blank">download</a>
```

下载demo

```
// http.js
const http = require("http");
const fs = require("fs");

const app = http
  .createServer((req, res) => {
      const { method, url } = req;
      if (method == "GET" && url == "/") {
          fs.readFile("./index.html", (err, data) => {
              res.setHeader("Content-Type", "text/html");
              res.end(data);
          });

      } else if (method === "GET" && url === "/api/download") {
          fs.readFile("./file.pdf", (err, data) => {
              res.setHeader("Content-Type", "application/pdf");
              const fileName = encodeURI('中文')
              res.setHeader('Content-Disposition' ,`attachment;
filename="${fileName}.pdf"`)
              res.end(data);
          });

      }


  })
// module.exports = app
app.listen(3000)
```

文件上传

前端demo

```
<input id='file1' type="file" />
<script>
  window.onload=function(){
      var files = document.getElementsByTagName('input'),
      len = files.length,
      file;
      for (var i = 0; i < len; i++) {
          file = files[i];
          if (file.type !== 'file') continue; // 不是文件类型的控件跳过
          file.onchange = function() {
              console.log('change')
              var _files = this.files;
              if (!_files.length) return;
              if (_files.length === 1) { // 选择单个文件
                  var xhr = new XMLHttpRequest();
                  xhr.open('POST', '/api/upload');
                  var filePath = files[0].value;
                  console.log(filePath) // C:\fakepath\2_koa中间件洋葱圈模型.png
                  // 'setRequestHeader' on 'XMLHttpRequest': Value is not a
valid ByteString.
                  // 请求的头信息中不能出现中文或UTF-8码的字符
                  xhr.setRequestHeader('file-name',
filePath.substring(filePath.lastIndexOf('\\') + 1));
                  xhr.send(_files[0]);
              } else { }
          };
      }
  };
</script>
```

node获取上传的图片

```
// 文件上传 3 种处理方式
const Koa = require('koa')
const app = new Koa()
const path = require('path')
const fs = require('fs')

// 静态服务，让访问 127.0.0.1:3000时可以直接访问index
app.use(require('koa-static')(__dirname + '/'))
app.use(require('koa-bodyparser')())

app.use((ctx, next) => {
  // 如果是上传文件，Content-Type为 multipart/form-data，bodyparser会无效
  console.log(ctx.request.body)
  let { req } = ctx.request
  let fileName = req.headers['file-name'] ? req.headers['file-name'] : '123.png'
  const outputFile = path.resolve(__dirname, fileName)

  // 1.使用流的方法
  // const fis = fs.createWriteStream(outputFile)
  // console.log(req.pipe)
  // req.pipe(fis)

  // 2.使用buffer方法
  let chunk = []
  let size = 0
```

```
  req.on('data', data => {
    chunk.push(data)
    size += data.length;
    console.log('data: ', data, size)
  })
  req.on('end', () => {
    console.log('end..')
    const buffer = Buffer.concat(chunk, size)
    size = 0
    fs.writeFileSync(outputFile, buffer)
  })

  // 3.流事件写入
  // const fis = fs.createWriteStream(outputFile)
  // req.on('data', data => {
  //   console.log('data:', data)
  //   fis.write(data)
  // })
  // req.on('end', () => {
  //   fis.end()
  // })

  ctx.body = 'hello'
})

app.listen(3000, () => { console.log('服务开启于3000端口') })
```

TCP协议与UDP协议的区别

TCP协议需要传统的三次握手，而UDP不需要，所以速度会快点

TCP协议 - 实现一个即时通讯IM

socket实现

原理：Net模块能够创建一个基于流的TCP服务器，客户端与服务端建立连接后，服务器可以获得一个全双工Socket对象，服务器可以保存socket对象列表，在接收到某个客户端消息时，再推送给其他客户端。

```
const net = require('net')
const chatServer = net.createServer()
const clientList = []

chatServer.on('connection', client => {
  client.write('Hi\n')
  clientList.push(client)
  client.on('data', data => {
    console.log('recive:', data.toString())
    clientList.forEach(v => {
      v.write(data)
    })
  })
})

chatServer.listen(9000, () => {
```

```
  console.log('服务开启在9000端口')
})
```

使用 `telnet` 连接到服务器，发送消息，其他连接上的客户端就可以收到消息了

```
# mac安装telnet：brew install telnet
# telnet是最早的远程控制工具，由于是明文传输，不安全，现在都改为ssh了
# 使用命链接到服务器
telnet localhost 9000
```