Nodejs网络 HTTP & 部署

#2021#

网络

TCP/IP网络协议

聊TCP/IP协议之前, 咱们先看一下OSI七层模型.

第 7 层：应用层 为操作系统或网络应用程序提供访问网络服务的接口。应用层协议的代表包括： HTTP，HTTPS，FTP，TELNET，SSH，SMTP，POP3等。
第 6 层：表示层 把数据转换为接受者能够兼容并且适合传输的内容，比如数据加密，压缩，格式转换等。
第 5 层：会话层 负责数据传输中设置和维持网络设备之间的通信连接。管理主机之间的会话进程，还可以利用在数据中插入校验点来实现数据的同步。
第 4 层：传输层 把传输表头加至数据形成数据包，完成端到端的数据传输。传输表头包含了协议等信息，比如: TCP，UDP 等。
第 3 层：网络层 负责对子网间的数据包进行寻址和路由选择，还可以实现拥塞控制，网际互联等功能。网络层的协议包括：IP，IPX 等。比如路由器就在这
第 2 层：数据链路层 在不可靠的物理介质上提供可靠的传输，主要主要为：物理地址寻址、数据封装成帧、流量控制、数据校验、重发等。比如交换机就在这
第 1 层：物理层 在局域网上传送数据帧，负责电脑通信设备与网络媒体之间的互通，包括针脚，电压，线缆规范，集线器，网卡，主机适配等。

1. html在哪一层?

   应用层

2. 那么咱们一直提到的协议是啥意思? 协议是什么?

   通俗点说：是协议定义了每一层的作用是什么, 每一层的职责是什么, 类似于规范和约束.

3. TCP/IP协议具体指什么?

   有的文章里就是具体指的TCP协议和IP协议, 但是大多数时候提到的TCP/IP协议, 可以理解为互联网通信所需要的协议, 是一个协议家族, 以TCP、IP协议为核心, 包含HTTP、SMTP、TELNET等各种协议。

4. TCP/IP参考模型?

   TCP/IP参考模型是一个抽象的分层模型，这个模型中，所有的TCP/IP系列网络协议都归类到4个抽象的"层"中.

   可以看一下图 **assets/OSI和TCPIP概念模型.png**

5. 我们常说的数据包是什么?

   数据包是网络层及以上分层中包的单位.

   每个分层都会对发送的数据添加一个首部, 首部包含了该层协议相关的信息, 而真正要发送的内容称之为数据.

   也就是说每一个数据包都由首部 + 数据组成.

   而对于下层来说, 上层发送过来的全部内容, 都会当做本层的数据, 举个例子:

   传输层 TCP包：TCP包首部 + 数据
   网络层 IP包：IP包首部 + (TCP包首部 + 数据)
   数据链路层 以太网包：以太网包首部 + (IP包首部 + (TCP包首部 + 数据))

   这里可以看图 **assets/TCP_IP_数据包.png**

6. 每层在接收到数据后除了添加首部, 还要做什么呢?

   用户1

- 传输层：TCP模块为保证数据的可靠传输, 需要添加TCP首部
- 网络层：IP包生成后，参考路由控制表决定接受此 IP 包的路由或主机。
- 数据链路层：生成的以太网数据包将通过物理层传输给接收端

   用户2

- 数据链路层：主机收到以太网包后，首先从以太网包首部找到 MAC 地址判断是否为发送给自己的包，若不是则丢弃数据。
  如果是发送给自己的包，则从以太网包首部中的类型确定数据类型，再传给相应的模块，比如IP.
- 网络层：从包首部中判断此 IP 地址是否与自己的 IP 地址匹配，如果匹配则根据首部的协议类型将数据发送给对应的模块，比如TCP

- 传输层：在 TCP 模块中，首先会计算一下校验和，判断数据是否被破坏。然后检查是否在按照顺序接收数据。最后检查端口号，确定具体的应用程序。数据被完整地接收以后，会传给由端口号识别的应用程序。

总结一下几个地址：

- 数据链路层的是MAC地址, 用来识别同一链路中的不同计算机
- 网络层的是IP地址, 用来识别TCP/IP 网络中互连的主机和路由器
- 传输层的是端口号(程序地址), 用来识别同一台计算机中进行通信的不同应用程序

7. 那么我们通过这三个地址就可以识别一次通信了吗?

答案是否定的.

我们需要通过以下这几个数据综合来识别一次通信：

- IP首部：源IP地址
- IP首部：目标IP地址
- 协议号, TCP或者UDP
- TCP首部：源端口号
- TCP首部：目标端口号

8. 我们常说的TCP/UDP他们的区别是什么？分别适合用在什么场景?

- UDP是无连接的，TCP必须三次握手建立连接
- UDP是面向报文，没有拥塞控制，所以速度快，适合多媒体通信要求，比如及时聊天，支持一对一，一队多。多对一，多对多。就像牛客网的视频面试就是用的UDP
- TCP只能是一对一的可靠性传输

那么咱们的直播底层是什么协议呢?

其实现在常见的rtmp和hls直播, 都是基于TCP的, 希望能提供稳定的直播环境.

9. TCP通过什么方式提供可靠性?

- 超时重发，发出报文段要是没有收到及时的确认，会重发。
- 数据包的校验，也就是校验首部数据和。
- 对失序的数据重新排序
- 进行流量控制，防止缓冲区溢出
- 快重传和快恢复

- TCP会将数据截断为合理的长度

10. TCP如何控制拥塞?

拥塞控制就是防止过多的数据注入网络中，这样可以使网络中的路由器或链路不致过载。
发送方维持一个叫做拥塞窗口cwnd（congestion window）的状态变量。
为了防止cwnd增长过大引起网络拥塞，还需设置一个慢开始门限ssthresh状态变量。ssthresh的用法如下：
当cwnd<ssthresh时，使用慢开始算法。也就是乘法算法
当cwnd>ssthresh时，改用拥塞避免算法。也就是加法算法
当cwnd=ssthresh时，慢开始与拥塞避免算法任意。
当出现拥塞的时候就把心的门限值设为此时窗口大小的一般，窗口大小设置为1，再重新执行上面的步骤。
当收到连续三个重传的时候这就需要快重传和快恢复了，当收到连续三个重传 这个时候发送方就要重传自己的信息，然后门限减半但是这个时候并不是网络阻塞，窗口只会减半执行拥塞避免算法。

11. TCP协议的一次数据传输, 从建立连接到断开连接都有哪些流程?

可以看一下图 **assets/TCP数据传输流程.jpeg**

第一次握手：建立连接。客户端发送连接请求报文段，将SYN位置为1，Sequence Number为x；然后，客户端进入SYN_SEND状态，等待服务器的确认；
第二次握手：服务器收到客户端的SYN报文段，需要对这个SYN报文段进行确认，设置Acknowledgment Number为x+1(Sequence Number+1)；同时，自己自己还要发送SYN请求信息，将SYN位置为1，Sequence Number为y；服务器端将上述所有信息放到一个报文段（即SYN+ACK报文段）中，一并发送给客户端，此时服务器进入SYN_RECV状态；
第三次握手：客户端收到服务器的SYN+ACK报文段。然后将Acknowledgment Number设置为y+1，向服务器发送ACK报文段，这个报文段发送完毕以后，客户端和服务器端都进入ESTABLISHED状态，完成TCP三次握手。

完成了三次握手，客户端和服务器端就可以开始传送数据。以上就是TCP三次握手的总体介绍。通信结束客户端和服务端就断开连接，需要经过四次分手确认。

第一次分手：主机1（可以使客户端，也可以是服务器端），设置Sequence Number和Acknowledgment Number，向主机2发送一个FIN报文段；此时，主机1进入FIN_WAIT_1状态；这表示主机1没有数据要发送给主机2了；
第二次分手：主机2收到了主机1发送的FIN报文段，向主机1回一个ACK报文段，Acknowledgment Number为Sequence Number加1；主机1进入FIN_WAIT_2状态；主机2告

诉主机1，我"同意"你的关闭请求；
第三次分手：主机2向主机1发送FIN报文段，请求关闭连接，同时主机2进入LAST_ACK状态；
第四次分手：主机1收到主机2发送的FIN报文段，向主机2发送ACK报文段，然后主机1进入TIME_WAIT状态；主机2收到主机1的ACK报文段以后，就关闭连接；此时，主机1等待2MSL后依然没有收到回复，则证明Server端已正常关闭，那好，主机1也可以关闭连接了。

1. IP地址

IP 地址（IPv4 地址）由32位正整数来表示，在计算机内部以二进制方式被处理。日常生活中，我们将32位的 IP 地址以每8位为一组，分成4组，每组以 "." 隔开，再将每组数转换成十进制数

IP地址包含网络标识和主机标识, 比如172.112.110.11

172.112.110就是网络标识, 同一网段内网络标识必须相同
11就是主机标识, 同一网段内主机标识不能重复

12. IPv6

IPv6（IP version 6）是为了根本解决 IPv4 地址耗尽的问题而被标准化的网际协议。IPv4 的地址长度为 4 个 8 位字节，即 32 比特。而 IPv6 的地址长度则是原来的 4 倍，即 128 比特，一般写成 8 个 16 位字节。

13. DNS

我们平时访问一个网站, 一个应用程序, 并不是用ip来访问的, 而是用一个域名. 那么域名是怎么和ip地址建立联系的呢?

就是通过dns, Domain Name System. 比如wiki上的一个例子

以访问 zh.wikipedia.org 为例：

客户端发送查询报文"query zh.wikipedia.org"至DNS服务器，DNS服务器首先检查自身缓存，如果存在记录则直接返回结果。
如果记录老化或不存在，则：
DNS服务器向根域名服务器发送查询报文"query zh.wikipedia.org"，根域名服务器返回顶级域 .org 的顶级域名服务器地址。
DNS服务器向 .org 域的顶级域名服务器发送查询报文"query zh.wikipedia.org"，得到二级

域 .wikipedia.org 的权威域名服务器地址。
DNS服务器向 .wikipedia.org 域的权威域名服务器发送查询报文"query zh.wikipedia.org"，得到主机 zh 的A记录，存入自身缓存并返回给客户端。

如何使用Nodejs来创建一个TCP服务?

在这之前咱们要先来了解一下Socket的概念,

我们经常把socket翻译为套接字，socket是在应用层和传输层之间的一个抽象层，它把TCP/IP层复杂的操作抽象为几个简单的接口供应用层调用已实现进程在网络中通信, 比如create，listen，accept，connect，read和write等等。

node里有各种网络相关的模块, http为应用层模块，主要按照特定协议编解码数据；net为传输层模块，主要负责传输编码后的应用层数据；https是个综合模块（涵盖了http/tls/crypto等），主要用于确保数据安全性

1. 创建tcp服务端

```
const net = require('net');

const HOST = '127.0.0.1';
const PORT = 7777;

// 创建一个TCP服务器实例，调用listen函数开始监听指定端口
// net.createServer()有一个参数，是监听连接建立的回调
net.createServer((socket) => {
    const remoteName = `${socket.remoteAddress}:${socket.remotePort}`;
    // 建立成功了一个连接，这个回调函数里返回一个socket对象.
    console.log(`${remoteName} 连接到本服务器`);

    // 接收消息
    socket.on('data', (data) => {
        console.log(`${remoteName} - ${data}`)
        // 给客户端发消息
        socket.write(`你刚才说啥? 是${data}吗? `);
    });

    // 关闭
    socket.on('close', (data) => {
```

```
        console.log(`${remoteName} 连接关闭`)
    });

}).listen(PORT, HOST);

console.log(`Server listening on ${HOST}:${PORT}`);
```

2. 创建tcp客户端

```
const net = require('net');

const HOST = '127.0.0.1';
const PORT = 7777;

const client = new net.Socket();
const ServerName = `${HOST}:${PORT}`;
let count = 0;

client.connect(PORT, HOST, () => {
    console.log(`成功连接到 ${ServerName}`);
    // 向服务端发送数据
    const timer = setInterval(() => {
        if (count > 10) {
            client.write('我没事了, 告辞');
            clearInterval(timer);
            return;
        }
        client.write('马冬梅' + count++);
    }, 1000)
});

// 接收消息
client.on('data', (data) => {
    console.log(`${ServerName} - ${data}`);
    // 关闭连接
    // client.destroy();
});
```

```
// 关闭事件
client.on('close', () => {
    console.log('Connection closed');
});

client.on('error', (error) => {
    console.log(error);
})
```

3. 运行一下

```
node tcp-server.js
```

```
node tcp-client.js
```

如何使用NodeJs来创建一个UDP服务?

1. 创建udp服务端

```
const dgram = require('dgram');
const server = dgram.createSocket('udp4');

server.on('message', (msg, remote) => {
    console.log(`${remote.address}:${remote.port} - ${msg}`)
    server.send(`收到! `, remote.port, remote.address);
})

server.on('listening', () => {
    const address = server.address()
    console.log(`Server listening on ${address.address}:${address.port}`);
})


server.bind(44444);
```

2. 创建udp客户端

```
const dgram = require('dgram')
const message = Buffer.alloc(5, 'lubai')
const client = dgram.createSocket('udp4')

client.send(message, 0, message.length, 44444, 'localhost',
    (err, bytes) => {
        console.log(`发送成功${bytes}字节`);
        // client.close()
    }
)

client.on('message', (buffer) => {
    console.log(buffer.toString())
})
```

3. 运行一下

```
node udp-server.js
```

```
node udp-client.js
```

HTTP

HTTP协议（HyperText Transfer Protocol，超文本传输协议）是用于从WWW服务器传输超文本到本地浏览器的传输协议。它可以使浏览器更加高效，使网络传输减少。它不仅保证计算机正确快速地传输超文本文档，还确定传输文档中的哪一部分，以及哪部分内容首先显示(如文本先于图形)等。

HTTP是客户端浏览器或其他程序与Web服务器之间的应用层通信协议。在Internet上的Web服务器上存放的都是超文本信息，客户机需要通过HTTP协议传输所要访问的超文本信息。HTTP包含命令和传输信息，不仅可用于Web访问，也可以用于其他因特网/内联网应用系统之间的通信，从而实现各类应用资源超媒体访问的集成。

我们在浏览器的地址栏里输入的网站地址叫做URL (Uniform Resource Locator，统一资源定位符)。就像每家每户都有一个门牌地址一样，每个网页也都有一个Internet地址。当你在浏览器的地址框中输入一个URL或是单击一个超级链接时，URL就确定了要浏览的地址。浏览器通过超文本传输协议(HTTP)，将Web服务器上站点的网页代码提取出来，并翻译成漂亮的网页。

一次完整的HTTP通信是什么样子的?

1. 建立 TCP 连接
   在HTTP工作开始之前，客户端首先要通过网络与服务器建立连接，该连接是通过 TCP 来完成的。HTTP 是比 TCP 更高层次的应用层协议，根据规则，只有低层协议建立之后，才能进行高层协议的连接，因此，首先要建立 TCP 连接，一般 TCP 连接的端口号是80；

2. 客户端向服务器发送请求命令
   一旦建立了TCP连接，客户端就会向服务器发送请求命令；
   例如：GET/info HTTP/1.1

3. 客户端发送请求头信息
   客户端发送其请求命令之后，还要以头信息的形式向服务器发送一些别的信息，之后客户端发送了一空白行来通知服务器，它已经结束了该头信息的发送；

4. 服务器应答
   客户端向服务器发出请求后，服务器会客户端返回响应；
   例如： HTTP/1.1 200 OK
   响应的第一部分是协议的版本号和响应状态码

5. 服务器返回响应头信息
   正如客户端会随同请求发送关于自身的信息一样，服务器也会随同响应向用户发送关于它自己的数据及被请求的文档；

6. 服务器向客户端发送数据
   服务器向客户端发送头信息后，它会发送一个空白行来表示头信息的发送到此为结束，接着，它就以 Content-Type 响应头信息所描述的格式发送用户所请求的实际数据；

7. 服务器关闭 TCP 连接
   一般情况下，一旦服务器向客户端返回了请求数据，它就要关闭 TCP 连接，然后如果客户端或者服务器在其头信息加入了这行代码 Connection:keep-alive ，TCP 连接在发送后将仍然保持打开状态，于是，客户端可以继续通过相同的连接发送请求。保持连接节省了为每个请求建立新连接所需的时间，还节约了网络带宽。

HTTP协议有哪些特点?

1. 通过请求和响应的交换达成通信

协议规定, 请求从客户端发出, 服务端响应请求并返回.

2. 无状态

   HTTP 是一种无状态协议, 在单纯HTTP这个层面，协议对于发送过的请求或响应都不做持久化处理

3. 使用Cookie做状态管理

   服务端返回的头信息上有可能会携带Set-Cookie, 那么当客户端接收到响应后, 就会在本地种上cookie. 在下一次给服务端发送请求的时候, 就会携带上这些cookie。

4. 通过URL定位资源

   这里区分一下URI和URL的概念.

   URI: 统一资源标识符, 比如你身份证号是xxxxxxx, 在所有人中是独一无二的, 这个身份证号就能标识你的身份, 那么它就是URI
   URL: 统一资源定位符, 比如北京市/朝阳区/xxxx/xxxx/xxxxx, 通过这一串信息可以定位到你, 那么这个就是URL

   URL有点类似于通过定位实现的URI.

   就像有个父类叫做URI, 他要实现的是唯一确定一个id. 有的人喜欢继承URI, 通过location来实现; 有的人喜欢继承URI, 通过name来实现.

5. 通过各种方法来标识自己的意图

   这里指的是各种HTTP方法, 比如GET POST PUT DELETE等.

6. 持久连接

   HTTP 协议的初始版本中，每进行一个 HTTP 通信都要断开一次 TCP 连接，增加了很多没必要的建立连接的开销。
   为了解决上述 TCP 连接的问题，HTTP/1.1 支持持久连接。其特点是，只要任意一端没有明确提出断开连接，则保持 TCP 连接状态。旨在建立一次 TCP 连接后进行多次请求和响应的交互。在 HTTP/1.1 中，所有的连接默认都是持久连接。

   也就是说默认情况下建立 TCP 连接不会断开，只有在请求报头中声明 Connection: close 才会

在请求完成后关闭连接。

7. 管道机制

1.1版本引入pipelining机制, 即在同一个TCP连接里面，客户端可以同时发送多个请求。

举例来说，客户端需要请求两个资源。以前的做法是，在同一个TCP连接里面，先发送A请求，然后等待服务器做出回应，收到后再发出B请求。管道机制则是允许浏览器同时发出A请求和B请求，但是服务器还是按照顺序，先回应A请求，完成后再回应B请求。

但是现代浏览器一般没开启这个配置, 这个机制可能会造成队头阻塞. 因为响应是有顺序的, 如果一个TCP连接中的第一个HTTP请求响应非常慢, 那么就会阻塞后续HTTP请求的响应.

所以现实中默认情况下, 一个TCP连接同一时间只发一个HTTP请求.

有的同学会问, 我怎么听说chrome最大支持6个同域名请求呢?

那是因为chrome最大支持同时开启6个TCP连接。

那么HTTP 1.0/1.1/2.0在并发请求上主要区别是什么?

1. HTTP/1.0
每次TCP连接只能发送一个请求，当服务器响应后就会关闭这次连接，下一个请求需要再次建立TCP连接.

2. HTTP/1.1
默认采用持续连接(TCP连接默认不关闭，可以被多个请求复用，不用声明Connection: keep-alive).
增加了管道机制，在同一个TCP连接里，允许多个请求同时发送，增加了并发性，进一步改善了HTTP协议的效率，
但是同一个TCP连接里，所有的数据通信是按次序进行的。回应慢，会有许多请求排队，造成"队头堵塞"。

3. HTTP/2.0

加了双工模式，即不仅客户端能够同时发送多个请求，服务端也能同时处理多个请求，解决了队头堵塞的问题。
使用了多路复用的技术，做到同一个连接并发处理多个请求，而且并发请求的数量比HTTP1.1大了好几个数量级。

增加服务器推送的功能，不经请求服务端主动向客户端发送数据。

各种Headers

Cache-Control

通过指定首部字段 Cache-Control 的指令，就能操作缓存的工作机制。

1. Cache-Control: public

   当指定使用 public 指令时，则明确表明其他用户也可利用缓存。

2. Cache-Control: private

   当指定 private 指令后，响应只以特定的用户作为对象，这与 public 指令的行为相反。缓存服务器会对该特定用户提供资源缓存的服务，对于其他用户发送过来的请求，代理服务器则不会返回缓存。

3. Cache-Control: no-cache

   可以在客户端存储资源，每次都必须去服务端做过期校验，来决定从服务端获取新的资源（200）还是使用客户端缓存（304）。也就是所谓的协商缓存。

4. Cache-Control: no-store

   永远都不要在客户端存储资源，永远都去原始服务器去获取资源。

5. Cache-Control: max-age=604800（单位：秒）

   当客户端发送的请求中包含 max-age 指令时，如果判定缓存资源的缓存时间数值比指定的时间更小，那么客户端就接收缓存的资源。另外，当指定 max-age 的值为0，那么缓存服务器通常需要将请求转发给源服务器。

   HTTP/1.1 版本的缓存服务器遇到同时存在 Expires 首部字段的情况时，会优先处理 max-age 指令，并忽略掉 Expires 首部字段

6. Cache-Control: s-maxage=604800（单位：秒）

   s-maxage 指令的功能和 max-age 指令的相同，它们的不同点是 s-maxage 指令只适用于供多位用户使用的公共缓存服务器（一般指代理）。

当使用 s-maxage 指令后，则直接忽略对 Expires 首部字段及 max-age 指令的处理

Connection

1. Connection: close

HTTP/1.1 版本的默认连接都是持久连接。当服务器端想明确断开连接时，则指定 Connection 首部字段的值为 close。

2. Connection: Keep-Alive

HTTP/1.1 之前的 HTTP 版本的默认连接都是非持久连接。为此，如果想在旧版本的 HTTP 协议上维持持续连接，则需要指定 Connection 首部字段的值为 Keep-Alive。

Date

表明创建 HTTP 报文的日期和时间。
Date: Mon, 10 Jul 2021 15:50:06 GMT
HTTP/1.1 协议使用在 RFC1123 中规定的日期时间的格式。

Pragma

Pragma 首部字段是 HTTP/1.1 版本之前的历史遗留字段，仅作为与 HTTP/1.0 的向后兼容而定义。

1. Pragma: no-cache

该首部字段属于通用首部字段，但只用在客户端发送的请求中，要求所有的中间服务器不返回缓存的资源。

所有的中间服务器如果都能以 HTTP/1.1 为基准，那直接采用 Cache-Control: no-cache 指定缓存的处理方式最为理想。但是要整体掌握所有中间服务器使用的 HTTP 协议版本却是不现实的，所以，发送的请求会同时包含下面两个首部字段：

```
Cache-Control: no-cache
Pragma: no-cache
```

Accept

1. Accept: text/html, application/xhtml+xml, application/xml;

   Accept 首部字段可通知服务器，用户代理能够处理的媒体类型及媒体类型的相对优先级。可使用 type/subtype 这种形式，一次指定多种媒体类型。

2. Accept-Encoding: gzip, deflate

   Accept-Encoding 首部字段用来告知服务器用户代理支持的内容编码及内容编码的优先顺序，并可一次性指定多种内容编码
   也可使用星号（*）作为通配符，指定任意的编码格式。

   gzip表明实体采用 GNU zip 编码
   compress　表明实体采用 Unix 的文件压缩程序
   deflate　表明实体采用 zlib 的格式压缩的
   identity　表明没有对实体进行编码，当没有 Content-Encoding 首部字段时，默认采用此编码方式

   Host

1. Host: www.baidu.com

- 告知服务器，请求的资源所处的互联网主机和端口号。
- Host 首部字段是 HTTP/1.1 规范内唯一一个必须被包含在请求内的首部字段。
- 若服务器未设定主机名，那直接发送一个空值即可 Host: 。

If-Modified-Since
形如 If-xxx 这种样式的请求首部字段，都可称为条件请求。服务器接收到附带条件的请求后，只有判断指定条件为真时，才会执行请求。

1. If-Modified-Since: Mon, 10 Jul 2021 15:50:06 GMT

用于确认代理或客户端拥有的本地资源的有效性。
在指定 If-Modified-Since 字段值的日期时间之后，如果请求的资源都没有过更新，则返回状态码 304 Not Modified 的响应

ETag

1. ETag: “aaaa-1234”

首部字段 ETag 能告知客户端实体标识。它是一种可将资源以字符串形式做唯一性标识的方式。服务器会为每份资源分配对应的 ETag 值。
另外，当资源更新时，ETag 值也需要更新。生成 ETag 值时，并没有统一的算法规则，而仅仅是由服务器来分配。

If-None-Match

1. If-None-Match: “lubai”

用于指定 If-None-Match 字段值的实体标记（ETag）值与请求资源的 ETag 不一致时，它就告知服务器处理该请求。

User-Agent

1. User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.114 Safari/537.36

首部字段 User-Agent 会将创建请求的浏览器和用户代理名称等信息传达给服务器。
由网络爬虫发起请求时，有可能会在字段内添加爬虫作者的电子邮件地址。此外，如果请求经过代理，那么中间也很可能被添加上代理服务器的名称。

Allow

1. Allow: GET, HEAD

首部字段 Allow 用于通知客户端能够支持 Request-URI 指定资源的所有 HTTP 方法。
当服务器接收到不支持的 HTTP 方法时，会以状态码 405 Method Not Allowed 作为响应返回。与此同时，还会把所有能支持的 HTTP 方法写入首部字段 Allow 后返回。

Content-Encoding

1. Content-Encoding: gzip

首部字段 Content-Encoding 会告知客户端服务器对实体的主体部分选用的内容编码方式。内容编码是指在不丢失实体信息的前提下所进行的压缩。

Content-Type

1. Content-Type: text/html; charset=UTF-8

首部字段 Content-Type 说明了实体主体内对象的媒体类型。和首部字段 Accept 一样，字段值用 type/subtype 形式赋值。

Expires

1. Expires: Mon, 10 Jul 2021 15:50:06 GMT

首部字段 Expires 会将资源失效的日期告知客户端。
缓存服务器在接收到含有首部字段 Expires 的响应后，会以缓存来应答请求，在 Expires 字段值指定的时间之前，响应的副本会一直被保存。当超过指定的时间后，缓存服务器在请求发送过来时，会转向源服务器请求资源。

Set-Cookie

1. Set-Cookie: userId=11111; expires=Mon, 10 Jul 20121 15:50:06 GMT; path=/;

- NAME=VALUE: cookie名称和值
- expires=DATE: Cookie 的有效期（若不明确指定则默认为浏览器关闭前为止）
- path=PATH: 用于限制指定 Cookie 的发送范围的文件目录。
- domain=域名: cookie有效的域名 （若不指定则默认为创建 Cookie的服务器的域名）
- Secure: 仅在 HTTPS 安全通信时才会发送 Cookie
- HttpOnly: 使 Cookie 不能被 JavaScript 脚本访问

如何使用NodeJs创建HTTP服务?

1. http-server.js

```
const http = require('http')
http.createServer(function (req, res) {
    res.writeHead(200, {
        'Content-Type': 'text/plain'
    })
    res.end('Hello World')
}).listen(80, '127.0.0.1')

console.log('Server running at http://127.0.0.1:80/')
```

2. 浏览器访问

   http://127.0.0.1:80/

3. 用curl访问

   curl -v http://127.0.0.1:80

   看一下请求报文

```
// 三次握手
* Rebuilt URL to: http://127.0.0.1:80/
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to 127.0.0.1 (127.0.0.1) port 80 (#0)

// 客户端向服务端发送请求报文
> GET / HTTP/1.1
> Host: 127.0.0.1:80
> User-Agent: curl/7.54.0
> Accept: */*
>

// 服务端响应客户端内容
< HTTP/1.1 200 OK
< Content-Type: text/plain
< Date: Wed, 04 Aug 2021 15:55:55 GMT
< Connection: keep-alive
< Keep-Alive: timeout=5
< Transfer-Encoding: chunked
<
* Connection #0 to host 127.0.0.1 left intact
Hello World%
```

4. htttp-client.js

```
const http = require('http')

const options = {
    hostname: '127.0.0.1',
    port: 80,
    path: '/',
    method: 'GET'
}
const req = http.request(options, (res) => {
    console.log(`Status=${res.statusCode}, Headers=$
{JSON.stringify(res.headers)}`);
    res.setEncoding('utf8')
    res.on('data', (data) => {
        console.log(data)
    })
})
req.end()
```

部署

服务器

谈到部署, 肯定得先有一个自己的服务器. 因为咱们是上课教学, 我就随便找个便宜的演示一下..

https://www.aliyun.com/daily-act/ecs/activity_selection?userCode=fyhp3q4t

选ecs服务器, 按量付费/月/年 都行, 随便选个镜像即可.

Linux安装Nodejs

1. 下载安装包

```
wget https://nodejs.org/dist/v10.9.0/node-v10.9.0-linux-x64.tar.xz
```

2. 解压

```
tar xf  node-v10.9.0-linux-x64.tar.xz
```

3. 设置软链接

```
ln -s /root/node-v10.9.0-linux-x64/bin/node   /usr/local/bin/node
ln -s /root/node-v10.9.0-linux-x64/bin/npm   /usr/local/bin/npm
```

4. 查看Node版本和npm版本

```
node -v
npm -v
```

5. 设置npm源

```
npm config set registry  https://registry.npm.taobao.org
```

6. 服务器安装pm2

```
npm install -g pm2
ln -s /root/node-v10.9.0-linux-x64/bin/pm2 /usr/local/bin/
```

7. 配置ssh

- 本地生成秘钥对: `ssh-keygen -t rsa` demo*id*rsa
- 将公钥放到服务器上: `scp ~/.ssh/demo_id_rsa.pub root@39.107.238.161:/root/.ssh/authorized_keys`
- 修改ssh配置 `vi ~/.ssh/config`

```
Host lubai
HostName 39.107.238.161
User root
Port 22
IdentityFile ~/.ssh/demo_id_rsa
```

- 服务器上修改ssh配置 `vim /etc/ssh/sshd_config`

PubkeyAuthentication yes
AuthorizedKeysFile .ssh/authorized_keys

- 最后就可以ssh登录了! `ssh lubai`

8. 将本地代码同步到服务器

```
rsync -avzp -e "ssh" ./Internet/ lubai:/root/app
```

9. 服务器上启动http

```
pm2 start /root/app/http-server.js
```

10. 本地修改发布命令

10.1 新建deploy.sh文件

```
#!/bin/bash

HOST=lubai

rsync -avzp -e "ssh" ./Internet/ $HOST:/root/app
ssh $HOST "pm2 restart /root/app/http-server.js"

echo 'deploy success'
```

10.2 初始化npm命令

```
npm init
```

新增scripts “deploy”: “./deploy.sh”

10.3 发布

```
npm run deploy
```

11. 修改http-server的监听host

```
const http = require('http')
const host = '0.0.0.0';
const port = 80;
http.createServer(function (req, res) {
    res.writeHead(200, {
        'Content-Type': 'text/plain'
    })
    res.end('Hello World')
}).listen(port, host)

console.log(`Server running at http://${host}:${port}/`)
```

12. ECS安全组添加80端口
13. 查看服务器上是否已正常监听80端口

```
netstat -tpln
```

14. 通过ip+端口访问

    39.107.238.161:80