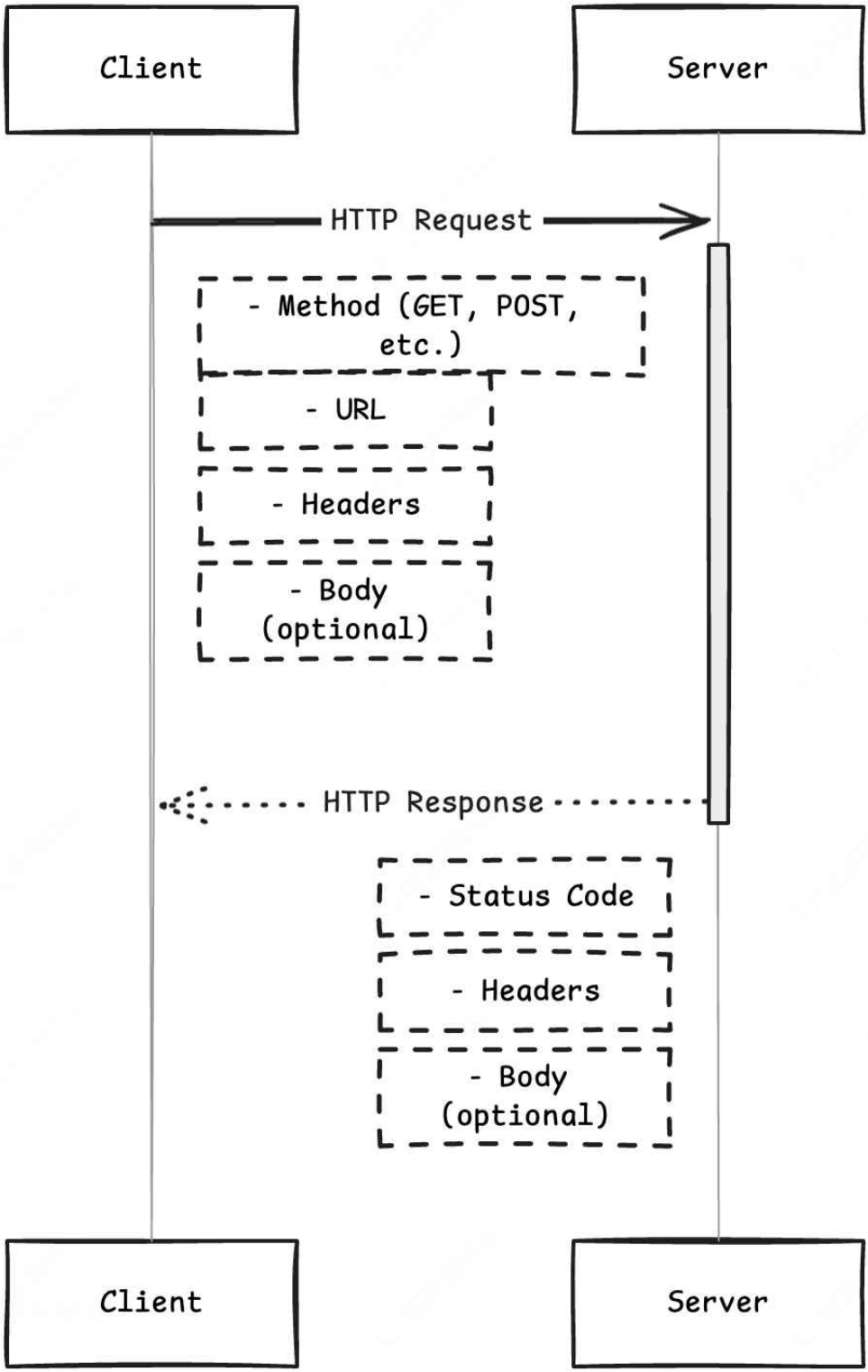
Node.js 的 HTTP 处理



在 Node 里面，起一个 HTTP Server 非常简单，如官网示例：

```
// 加载 http 模块
const http = require('http')
// 定义服务运行的主机名和端口号
const hostname = '127.0.0.1'
const port = 3000
// 通过 http.createServer 方法创建一个服务实例
// 同时传入回调函数，以接管后面进来的请求
// 后面请求进来时，这个回调函数会被调用执行，同时会拿到两个参数，分别是 req 和 res
// req 是可读流（通过 data 事件接收数据），res 是可写流（通过 write 写数据，end 结束输出）
const server = http.createServer((req, res) => {
// 设置返回的状态码 200 表示成功
res.statusCode = 200
// 设置返回的请求头类型 text/plain 表示普通文本
res.setHeader('Content-Type', 'text/plain')
// 对响应写入内容后，关闭可写流
res.end('Hello World\n')
})
// 调用实例的 listen 方法把服务正式启动
server.listen(port, hostname, () => {
  console.log(
Server running at http://${hostname}:${port}/
)
})
```

HTTP 作为整个互联网数据通信中几乎最主流的协议，它本身就是巨大的知识库，无论是工作 1 年还是 10 年的工程师，每一次重温 HTTP 的整体知识相信都会有很多收获，从 HTTP/1.1 到 HTTP/2，从 HTTP 到 HTTPS，从 TCP 的握手到 cookie/session 的状态保持...，我们在接触和 HTTP 的时候，一开始很容易被吓唬到，扎进去学习的时候也确实枯燥乏味，比较好的办法，就是在工作中不断的使用它，不断的练习，随着使用中的一点点深入，我们会对 HTTP 越来越熟悉。

那么这一节，我们就挑 HTTP 模块在 Node 中的几个应用知识来学习，以代码练习为主，主要学习 HTTP 模块在 Node 中的使用。

简单的 HTTP 头常识

一个请求，通常会建立在两个角色之间，一个是客户端，一个是服务端，而且两者的身份可以互换的，比如一台服务器 A 向 服务器 B 发请求，那么 A 就是客户端，B 是服务端，反过来身份就变了，甚至如果 A 这台服务器自己向自己发一个请求，那么 A 里面发请求的程序就是客户端，响应请求的程序就是服务端了，所以大家可以打开思路，不用局限在端的形态上面。

我们简单的看下一个请求从浏览器发出，以及服务器返回，它们的头信息，我们去实现爬虫的时候，有时候需要构造假的请求头，或者解析响应头，这在特定场景下会有一定的参考作用，比如打开 xiaojuSurvey首页，我们针对这个网页 HTML 的 GET 请求，简单学习它里面的头信息知识：

```
// 请求由 A 请求行、B 请求头组成
// A 请求行由 3 端组成，HTTP Verb/URL Path/HTTP Version
// 1. 标明请求方法是 GET，往往用作获取资源（图片、视频、文档等等）
// 2. /timeline 是请求的资源路径，由服务器来决定如何响应该地址
// 3. HTTP 协议版本是 1.1
GET /timeline HTTP/1.1
// B 如下都是请求头
// 去往哪个域名（服务器）去获取资源
Host: xiaojusurvey.didi.cn
// 保持连接，避免连接重新建立，减少通信开销提高效率
Connection: keep-alive
// HTTP 1.0 时代产物，no-cache 禁用缓存
Pragma: no-cache
// HTTP 1.1 时代产物，与 Pragma 一样控制缓存行为
Cache-Control: no-cache
// 浏览器自动升级请求，告诉服务器后续会使用 HTTPS 协议请求
Upgrade-Insecure-Requests: 1
// 上报用户代理的版本信息
User-Agent: Mozilla/5.0 (iPhone; CPU iPhone OS 16_6 like Mac OS X)
AppleWebKit/605.1.15 (KHTML, like Gecko) Version/16.6 Mobile/15E148
Safari/604.1
// 声明接收哪种格式的数据内容
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,i
mage/apng,
*/*
;q=0.8
// 声明所接受编码压缩的格式
Accept-Encoding: gzip, deflate, br, zstd
// 声明所接受的地区语言
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8
```

当然还有 POST，HEAD, PUT, DELETE 等这些请求方法，他们甚至可以多传一些数据包，比如 POST会多一个请求体，我们继续看下上面的这个请求头给到服务端，服务器返回的头是如何的：

```
// 整体上， response header 跟 request header 格式都是类似的
// 响应由 3 部分组成，A 响应行； B 响应头；C 响应体
// A 响应行依然是 HTTP 协议与响应状态码，200 是响应成功
HTTP/1.1 200 OK
// 响应的服务器类型
Server: nginx
// 当前响应的时间
Date: Thu, 08 Aug 2024 12:16:58 GMT
// 返回的数据类型，字符编码集
Content-Type: text/html; charset=utf-8
```

```
// 数据传输模式
Transfer-Encoding: chunked
// 保持连接
Connection: keep-alive
// HTTP 协议的内容协商，比如如何响应缓存
Vary: Accept-Encoding
// 控制该网页在浏览器的 frame 中展示，如果是 DENY 则同域名页面中也不允许嵌套
X-Frame-Options: SAMEORIGIN
// 控制预检请求的结果缓存时长
Access-Control-Max-Age: 86400
// 在规定时间内，网站请求都会重定向走 HTTPS 协议，也属于安全策略
Strict-Transport-Security: max-age=31536000
// 编码压缩格式的约定，gzip 是一种比较节省资源的压缩格式
Content-Encoding: gzip
// 告诉所有的缓存机制是否可以缓存及哪种类型
Cache-control: private
// 服务器输出的标识，不同服务器不同，可以从服务器上关闭不输出
X-Powered-By-Defense: from pon-wyxm-tel-qs-qssec-kd55
```

向别的服务器请求数据 - http.get

我们在 Node 里面，向另外一台服务器发请求，这个请求可能是域名/IP，请求的内容也可能五花八门，那这个请求该怎么构造呢？

这时候可以使用简单的 http.get/https.get 方法，比如我们去请求一个 Node LTS JSON 文件，从浏览器里直接打开就可以自动下载，在 Node 里面就可以这样做：

```
// 加载 https 模块，https 的底层依然是 http
const https = require('https')
// 请求的目标资源
const url = 'https://nodejs.org/dist/index.json'
// 发起 GET 请求，回调函数中会拿到一个来自服务器的响应可读流 res
https.get(url, (res) => {
  // 声明一个 字符串
  let data = ''
  // 每次可读流数据搬运过来，都是一个 buffer 数据块，每次通过 data 事件触发
  res.on('data', (chunk) => {
    // 把所有的 buffer 都拼一起
    data += chunk
  })
  res.on('end', () => {
    // 等待可读流接收完毕，就拿到了完整的 buffer
    // 通过 toString 把 buffer 转成字符串打印出来
    console.log(data.toString())
  })
```

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

会拿到这样的一坨 JSON 字符串

```
[{"version":"v11.2.0","date":"2018-11-15","files":..},...]
```

通过 Promise 包装一个 http.get 请求

上面的这个请求比较简单，代码也比较硬，是通过回调和事件的形式来完成数据的接收，一旦有多个存在依赖关系的异步请求，各种回调函数层层嵌套可读性就会变得很差。如果想让这个代码更优雅一些，我们可以将其改造成Promise:

```
const https = require('https')
const url = 'https://nodejs.org/dist/index.json'
// 声明一个普通函数，它接收 url 参数，执行后返回一个 Promise 实例
const request = (url) => {
  return new Promise((resolve, reject) => {
    https.get(url, (res) => {
      let data = ''
      res.on('data', (chunk) => {
      data += chunk
    })
    res.on('end', () => {
      // 通过 Promise 的 resolve 来回调结果
      resolve(data.toString())
    })
  }).on('error', (e) => {
    reject(e)
  })
})
}
// 在执行时候，可以使用 .then() 方法来链式调用，避免回调嵌套
request(url)
.then(data => {
    console.log(data)
})
```

通过 async function 来替代请求的链式传递

上面有了 Promise 的封装后，我们可以直接用 async function 继续完善下这个 Promise，进一步避免 then 的回调包裹，比如改成这样子：

```
const https = require('https')
const url = 'https://nodejs.org/dist/index.json'
// 改成一个 async 异步函数
const request = async (url) => {
return new Promise((resolve, reject) => {
    https.get(url, (res) => {
      let data = ''
      res.on('data', (chunk) => {
      data += chunk
      })
    res.on('end', () => {
      resolve(data.toString())
    })
    }).on('error', (e) => {
      reject(e)
    })
  })
}
// 声明一个异步函数，await 和 async 要配对使用
async function run () {
  // 以同步的方式来写异步逻辑
  const data = await request(url)
  console.log(data)
}
// run 方法执行后本身也是一个 Promise，可以通过 then 链式调用
run()
```

经过Promise -> async function的改造，代码的调用逻辑也清晰了很多。而且由于浏览器引擎的缘故，async function的性能是比Promise好的，Promise要慢10%左右。

通过三方库 axios/undici 来替代 http.get

以上都是用原生的 API 来直接实现，优点是不依赖三方库，拿到的响应就是天然的 Stream 流，那么一点点不方便的地方是，它回调中的响应 res 是 http.ClientRequest 流对象，需要自己监听 data 事件来手动组装 bufer 块，甚至还需要自己解析 JSON 数据格式，处理解析异常等，另外还不能原生支持 Promise，需要自己包装。在实际的工作场景中，我们为了开发效率，会考虑不深入这么底层的细节，直接采用三方库，比如 axios/undici 等，我们通过 axios 来实现一下：

```
const axios = require('axios')
const url = 'https://nodejs.org/dist/index.json'
```

```
const run = async url => {
  try {
    const res = await axios.get(url)
    console.log(res.data)
  } catch (error) {}
}
run(url)
```

拿到的结果是一个 JSON 化后的数据格式，更加易用友好，简单了解下 API 和三方库，我们来做一些稍微复杂点的练习，比如爬取网页源码。

结合 axios 和 cheerio 来爬取分析网页源码

请求一个网页就是获取一个远程 HTML 文件内容，跟上面我们获取 JSON 没有本质区别，拿到网页源码后，可以通过 cheerio 来加载源码遍历 DOM 节点，选择目标的 HTML 元素，从而获得期望的内容，比如文本或者链接，我们拿 Node 的 Learn 页面为例，分析页面的 DOM 节点，左侧的菜单都是 nav 元素里的 a 标签，我们只需要获取 a 标签就行了，这个通过 cheerio可以很轻松的做到：

```
const axios = require("axios");
const cheerio = require("cheerio");
const url = "https://nodejs.org/en/learn/getting-started/introduction-to-nodejs";
// 这里可以借助 request 三方库实现
async function run() {
  // 拿到网页源码内容
  const { data: html } = await axios.get(url);
  // 遍历查找出目标元素节点
  const
$ = cheerio.load(html);
  const items = $
("nav a");
  const menus = [];
  // 遍历节点对象，调用 text 方法取出文本内容
  items.each(function () {
    menus.push($(this).text());
  });
  console.log(menus);
}
run();
```

打印结果就是菜单列表：

```
[
  '',
  'Learn',
  'About',
  'Download',
  'Blog',
  'Docs',
  'Certification',
  '',
  'Introduction to Node.js',
  ...
]
```