[静态资源服务器] Node 的工具集 – path/util/zlib 等

本节目标：【实现一个静态资源服务器】没有金刚钻，不揽瓷器活，而金刚钻，也需要得力小助手，在 Node **那就是文件路径，网址解析，参数处理，压缩解压等等这些贴心小工具。**

开始之前，我们把这几个模块导进来，看下有哪些小工具：

```
const path = require('path')       // 路径模块
const qs = require('querystring') // 地址参数解析模块
const util = require('util')       // 常用工具方法模块
const url = require('url')         // URL 解析模块
```

Node 默认提供的路径信息

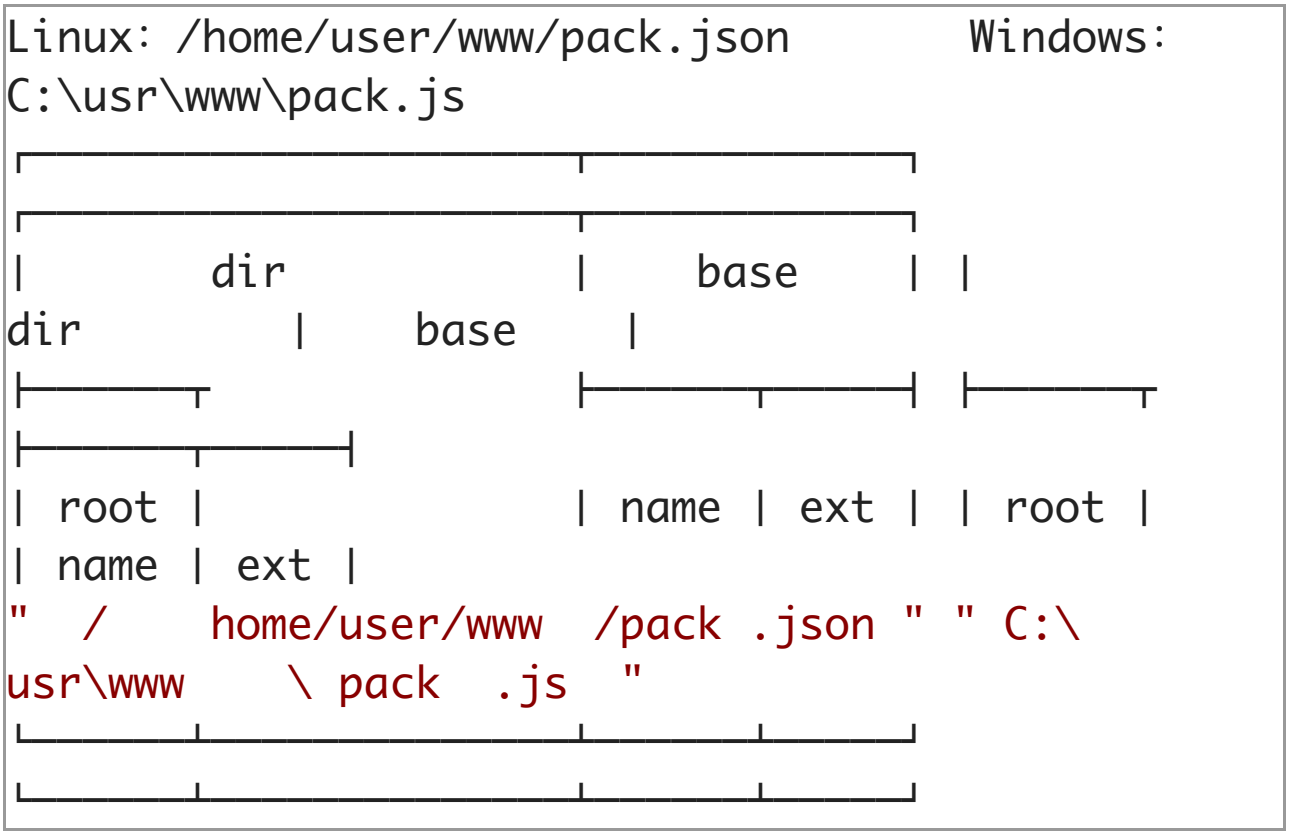
当一个代码开始运行的时候，它是在哪个目录下，以及从什么位置运行，这些地址信息非常重要，那么在 Node 里面有哪些关键的地址信息呢，我们在最初 Node 源码解读和 CommonJS 规范讲解的时候就知道，在 Node 里面所写的任何一个 JS 模块，它都会被被这样的函数包起来：

```
(function (exports, require, module, __filename, __dirname) {
  // 我们的代码都写在这里
});
```

在代码内部就可以拿到传下来的 5 个参数变量，跟地址有关的是最后两个参数 __filename 和 __dirname，在 global 里面还有 process.cwd() 可以直接访问，它们都是跟路径相关的变量，我们来过一下：

```
./              当前文件所在目录，是个相对路径
                - 如：/Users/black/juejin/index.js
                - index.js 的 ./ 是
/Users/black/juejin/
__dirname       当前文件所在目录的完整目录名，也就是绝对路
径
                - 如：/Users/black/juejin/node/
__filename      当前文件所在目录的完整目录路径，含文件名
                -
如：/Users/black/juejin/node/package.json
process.cwd()   当前执行 Node 命令时候的文件夹目录名
                - 如：node index.js 就是 index.js 文
件夹目录
                - 如：node ./lib/index.js 就是 lib 这
一级的目录
```

其中 process.cwd() 这个特殊一些，大家要注意，文件的真实位置和程序启动文件时候所处的位置不一定是相同的，而 process.cwd() 就是指运行程序时候所处的目录，无论是文件还是文件夹，它都有针对当前运行程序的一个相对地址和一个相对于操作系统的绝对路径，其中相对地址随着参照物和运行程序会发生变化，而绝对路径是不变的，参考 [Node Path 的文档](https://nodejs.org/api/path.html#path_path_parse_path)，一个绝对路径的组成是这样的：



- root 文件所在的根目录
- dir 文件所处的目录
- base 由文件名和后缀名组成，是路径的最后一部分
- name 文件名
- ext 文件后缀名

我们在程序里能东奔西跑，就是靠路径的这几个组织层级，至于 path 是怎么组织路径的，我们接下来到 path 里面了解一下。

文件位置穿梭不迷路 - path

我们在文件堆中穿梭的时候，最害怕的就是进错目录，进错层级，还要考虑到 windows 系统和 Linux 系统的差异性，一个文件夹是 C:\\Users\\..，一个是 /Users/black/，目录的格式差别很大。

我们在 windows 上和 Linux 上运行代码的时候，path 会自动判断当前的系统，从而做出相应的处理，比如

- const file = '/Users/black/juejin/node/package.json'
- const file = 'C:\Users\black\juejin\node\package.json'

path 的几个常用的 API 比较简单，整理如下：

方法	使用场景	代码示例
path.basename(path[, ext])	我们想要获取这个目录路径中最后一部分，也就是文件名	`path.basename('./ltsn/ // package.json path.basename('./ltsn/ '.json') // package`
path.dirname(path)	当我们想要拿到当前 JS 文件所在的文件夹名称时	`path.dirname('/usr/bi // /usr/bin/data`
path.extname(path)	当我们想拿到一个文件名的扩展名的时候，也就是文件后缀	`path.extname('about.h // .html // 没有扩展名，则返回一个`
path.normalize(path)	当我们想要把一个不规范的路径，处理成符合当前操作系统格式的标准路径字符	`path.normalize('/usr/ // /usr/local/bin/pac`

	串时	
path.join([...paths])	当我们想要把几个路径合并起来，拼成一个路径字符串的时候	`path.join('./a', 'b/c` `// a/b/c/d/e`
path.resolve([...paths])	当我们想要通过当前文件所在的目录，与另外一个目录之间的关系，来拼接绝对路径，或者寻找某一层级的目录时：	`// 特点：  相当于不断调用` `path.resolve('foo/bar`
path.relative(from, to)	当我们想要获取两个目录路径之间的相对关系时	`path.resolve('/usr/bi` `// /usr/bin/www/b`

path.parse(path)	将一个路径字符串解析为一个对象，里面分别是根目录，文件所在目录，文件名字、后缀名和名字	`path.parse('/usr/bin/`
		`/*`
		` { root: '/',`
		`  dir: '/usr/bin/www'`
		`  base: 'package.json`
		`  ext: '.json',`
		`  name: 'package' }`
		`*/`

	等	

path.format(pathObject)	与 path.parse 相反，把一个对象解析为一个路径字符串	path.format({
		  dir: '/usr/bin/www'
		  base: 'package.json
		  ext: '.json',
		  name: 'package'
		})
		// /usr/bin/www/packa

参数的序列化与反序列化 - querystring

前后端工程师合作时，可能提到最多的就是接口返回的数据格式和请求所传的参数了，前者是从服务端拿到的响应，而后者就是入参，参数一旦获取出错可能导致返回的数据出错，甚至会引发后端的一些程序运行异常，那么参数解析就变得非常关键，一个简单的地址参数可能长这个样子：

```
https://bd.juejin.im/?
utm_campaign=bd&utm_source=web&utm_medium=lin
(https://bd.juejin.im/?
utm_campaign=bd&utm_source=web&utm_medium=lin
```

一个复杂点的可能长这样子：

```
https://srd.simba.taobao.com/rd?
w=mmp4ptest&f=https%3A%2F%2Fre.taobao.com%2Fauc
(https://srd.simba.taobao.com/rd?
w=mmp4ptest&f=https%3A%2F%2Fre.taobao.com%2Fauc
```

稍后我们再来拆解这两个 url，先来看下 querystring 几个主要方法：

方法	使用场景	代码示例
querystring.parse(str[, sep][, eq][, options])	参数串解析为对象	`qs.parse('a=1&b=2')` `// { a: '1', b: '2' }`
querystring.stringify(obj[, sep][, eq][, options])	对象转成参数串	`qs.stringify({a: 1, b:` `// 'a=1&b=2'`
querystring.escape(str)	对参数进行编码转义	`qs.escape('qs.escape('a` `// a%3D%E5%BC%80%20%E5%`
querystring.unescape(str)	解码，是 escape 的逆向	`qs.unescape('a%3D%20%E4` `// a= 不`

我们现在对前面的两个 url 地址参数进行解析：

```
qs.parse('utm_campaign=bd&utm_source=web&utm_medium=link')
// 解析出 3 个参数，解析为一个对象
// { utm_campaign: 'bd', utm_source: 'web', utm_medium: 'link' }
qs.stringify({ utm_campaign: 'bd', utm_source: 'web', utm_medium: 'link' }, '@@')
// 参数的分隔符可以定制，比如 @@
//
utm_campaign=bd@@utm_source=web@@utm_medium=link
```

```
qs.parse('https://srd.simba.taobao.com/rd?
w=mmp4ptest&f=https%3A%2F%2Fre.taobao.com%2Faucti
on%3Fkeyword%3D%26catid%3D50050587%26refpid%3Dtt_
26632537_19294801_67276517%26crtid%3D1189307289%2
6itemid%3D572699375755%26adgrid%3D1016912923%26el
emtid%3D1%26clk1info%3D1210788191%2C64%2CdPGasl8c
WaKf4G0xY6c%252BLZZBixWUMilHWql2oFy2mXu521tWpp6T7
wwyX7D1fRla%26sbid%3D%3B%3B%2C%3B31234%26nick%3D%
25%5Cu6dd8%5Cu5b9d%5Cu6635%5Cu79f0%26qtype%3D5%26
tagvalue%3D645942367902559177 4_0_100%26isf%3D0&k=
eb305a7ca09eeebf&pvid=0a67267d00005bd5b1c04ab3008
26baf&p=tt_26632537_19294801_67276517')

/* 解析出来 w/f/k/pvid/p 这 4 个参数
{ w: 'mmp4ptest',
  f:
   'https://re.taobao.com/auction?
keyword=&catid=50050587&refpid=tt_26632537_192948
01_67276517&crtid=1189307289&itemid=572699375755&
adgrid=1016912923&elemtid=1&clk1info=1210788191,6
4,dPGasl8cWaKf4G0xY6c%2BLZZBixWUMilHWql2oFy2mXu52
1tWpp6T7wwyX7D1fRla&sbid=;;,;31234&nick=%\\u6dd8\
\u5b9d\\u6635\\u79f0&qtype=5&tagvalue=64594236790
25591774_0_100&isf=0',
  k: 'eb305a7ca09eeebf',
  pvid: '0a67267d00005bd5b1c04ab300826baf',
  p: 'tt_26632537_19294801_67276517' }
*/
```

其中 f 这个参数是一个二跳地址，它里面有 %，我们把它的参数解码一下

```
qs.unescape('keyword=&catid=50050587&refpid=tt_266
```

解码后，继续解析参数串：

```
qs.parse('keyword=&catid=50050587&refpid=tt_266325
```

```
/* 解析出来一坨参数
{ keyword: '',
  catid: '50050587',
  refpid: 'tt_26632537_19294801_67276517',
  crtid: '1189307289',
  itemid: '572699375755',
  adgrid: '1016912923',
  elemtid: '1',
  clk1info:
   '1210788191,64,dPGasl8cWaKf4G0xY6c
LZZBixWUMilHWql2oFy2mXu521tWpp6T7wwyX7D1fRla',
  sbid: ';;,;31234',
  nick: '%\\u6dd8\\u5b9d\\u6635\\u79f0',
  qtype: '5',
  tagvalue: '6459423679025591774_0_100',
  isf: '0' }
*/
```

其中 itemid 就是淘宝的商品 ID 了，有一个 nick 应该是用户昵称（这个已被我改过），是一个 utf8 编码的字符串，它从 utf8 转中文，就是淘宝昵称了。

所以在这个复杂点的场景里面，我们可以解析拿到二跳需要跳往的地址，同时带过去相应的参数，这个 url 就可以承载更多的业务逻辑了，比如商品定位，用户信息透传，类目信息包括媒体推广的流量跟

踪等等，非常强大。

实用方法集 - util

util 一开始是给内部模块使用的，提供了很多好用的工具函数，现在作为 Node 的核心 API 暴露给开发者使用，我们也就不用自己实现了。

util.callbackify(original)	**promise 的代码风格转成 callback 风格**
util.promisify(original)	callback 的代码风格转成 promise 风格
util.format(format[, ...args])	字符串格式化处理
util.isDeepStrictEqual(val1, val2)	比对两个变量是否严格相等
util.inherits(constructor, superConstructor)	与 class 的 extends 相同，继承父类的方法属性
util.inspect(object[, options])	对传入对象进行字符串格式化操作
util.types	各种数据类型的判断

当然还有其他的 API，比如对于废弃 API 的提示等，不再一一列举，他们在特定的场景下，都非常有用，特别是 util.types 来判断数据类型，可能是用到非常高频的 API，我们单独看下 promisify，这个是我个人最喜欢的一个工具函数了，可以把 callback 转成 promise，最开始想要把一个 callback 包装成 promise，通常会这么干：

```
const { readFile } = require('fs')
const { join } = require('path')
const filePath = join(__dirname,
'./package.json.')

const readFileAsync = (filePath) => {
  return new Promise((resolve, reject) => {
    readFile(filePath, 'utf8', (err, data) => {
      if (err) reject(err)
      else resolve(data)
    })
  })
}

readFileAsync(filePath)
  .then(data => console.log(data.toString()))
```

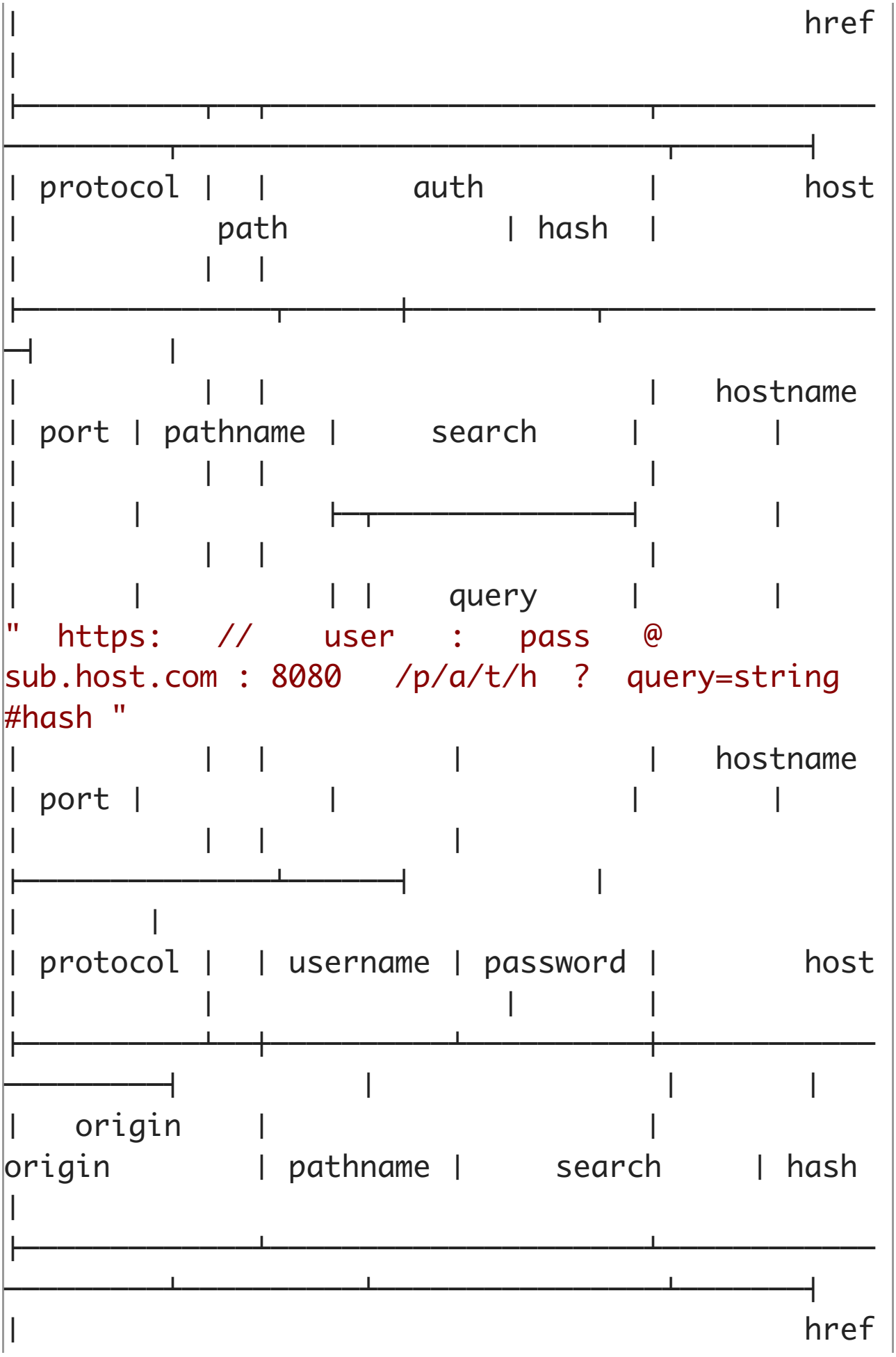
现在有了 promisify，就简单多了：

```
const { readFile } = require('fs')
const { join } = require('path')
const { promisify } = require('util')
const filePath = join(__dirname,
'./package.json')
const readFileAsync = promisify(readFile)

readFileAsync(filePath)
  .then(data => console.log(data.toString()))
```

网址解析利器 - url

```
┌──────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────┐
```


href
protocol
auth
host
path
hash
port
pathname
search
hostname
query
" https: // user : pass @ sub.host.com : 8080 /p/a/t/h ? query=string #hash "
port
hostname
protocol
username
password
host
origin
origin
pathname
search
hash
href

```
|
└──────────────────────────────────────────────────────────────────────────────┘
(all spaces in the "" line should be ignored - they are purely for formatting)
```

结合上图，一个 url 可以是这个样子：

> https://usr:pwd@juejin.com:8080/a/b/c/d?q=js&cat=3&#hash (https://usr:pwd@juejin.com:8080/a/b/c/d?q=js&cat=3&#hash)

在命令行中，我们 url.parse 一下，所解析出来的，跟上图也能对应上：

```
url.parse('https://usr:pwd@juejin.com:8080/a/b/c/d?q=js&cat=3&#hash')
Url {
  // 请求协议，比如 http、https、ftp、file 等
  protocol: 'https:',
  // 协议的 : 号有没有 /
  slashes: true,
  // url 的认证信息，跟上 @ 来区分认证部分和域名部分
  auth: 'usr:pwd',
  // url 的主机名
  host: 'juejin.com:8080',
  // 主机端口号
  port: '8080',
  // 主机名
  hostname: 'juejin.com',
  // 锚点部分，用 # 标识
  hash: '#hash',
  // 查询参数，包含 ?
  search: '?q=js&cat=3&',
  // 查询参数的字符串部分，不包含 ?
  query: 'q=js&cat=3&',
  // url 中的路径部分
  pathname: '/a/b/c/d',
  // 完整路径，由 pathname 和 search 组成
  path: '/a/b/c/d?q=js&cat=3&',
  // 链接地址
  href: 'https://usr:pwd@juejin.com:8080/a/b/c/d?q=js&cat=3&#hash'
}
```

除了 parse，Node 还提供了一个 URL 类，挂到了 global 上面，通过它可以 new 一个 URL 实例出来，来拿到所有特定的属性值，而且还可以修改任意的属性值：

```
const url = new
URL('https://usr:pwd@juejin.com:8080/a/b/c/d?
q=js&cat=3#hash')
console.log(url.hash)
// #hash
url.hash = 'newHash'
url.port = 7000
url.pathname = '/e/f'
console.log(url.href)
// https://usr:pwd@juejin.com:7000/e/f?
q=js&cat=3#newHash
```

除了解析 url，我们反过来也可以根据一个对象生成一个 url 地址：

```
const url = require('url')

const href = url.format({
  protocol: 'https',
  hostname: 'juejin.com',
  port: '8080',
  pathname: '/a/b/c/d',
  auth: 'usr:pwd',
  hash: '#hash',
  query: {
    q: 'js',
    cat: 3
  }
})

// https://usr:pwd@juejin.com:8080/a/b/c/d?
q=js&cat=3#hash
```

或者是通过 resolve 来拼接一个 url 地址，比如：

```
const url = require('url')

const href =
url.resolve('https://juejin.com/book', '/3')
// 'https://juejin.com/book/3'
```

编程练习 - 静态服务器搭建

在 Node 里面，起一个服务器非常简单：

```
const server = http.createServer((req, res) => {
  //
})
```

但是要让这个服务器可以响应不同的静态资源，就需要考虑情况，做必要的判断甚至压缩处理，这时候小工具们就派上用场了，我们来实现一个静态服务器：

```
#!/usr/bin/env node

const fs = require('fs')
const url = require('url')
const http = require('http')
const path = require('path')
const zlib = require('zlib')
const wwwroot = '/home/admin/wwwroot'
const mimeType = {
  '.ico': 'image/x-icon',
  '.md': 'text/plain',
  '.html': 'text/html',
  '.js': 'application/javascript',
  '.json': 'application/json',
  '.css': 'text/css',
```

```
  '.png': 'image/png',
  '.jpg': 'image/jpeg',
  '.wav': 'audio/wav',
  '.mp3': 'audio/mpeg',
  '.svg': 'image/svg+xml',
  '.pdf': 'application/pdf',
  '.doc': 'application/msword',
  '.eot': 'appliaction/vnd.ms-fontobject',
  '.ttf': 'aplication/font-sfnt'
}

const server = http.createServer((req, res) => {
  const { pathname } = url.parse(req.url)
  const filePath = path.join(wwwroot, pathname)
  const ext = path.extname(pathname)
  // 参数合法性校验
  // 1. 非允许后缀的资源不予返回
  if (!mimeType[ext]) {
    res.writeHead(404)
    return res.end()
  }
  // 2. 若后缀合法，判断文件是否存在
  if (!fs.existsSync(filePath)) {
    return (res.statusCode = 404)
  }
  // 3. 若文件存在，判断是否是文件类型
  const fStat = fs.statSync(filePath)
  if (!fStat.isFile()) {
    return (res.statusCode = 404)
  }
  // 4. 若合法存在，判断是否位于 wwwroot 目录下
  if (!filePath.startsWith(wwwroot)) {
    return (res.statusCode = 404)
```

```
  }

  // 5. 304 缓存有效期判断，使用 If-Modified-Since，用 Etag 也可以
  const modified = req.headers['if-modified-since']
  const expectedModified = new Date(fStat.mtime).getTime()
  if (modified && modified == expectedModified) {
    res.statusCode = 304
    res.setHeader('Content-Type', mimeType[ext])
    res.setHeader('Cache-Control', 'max-age=3600')
    res.setHeader('Last-Modified', new Date(expectedModified).toGMTString())
    return
  }

  // 6. 文件头信息设置
  res.statusCode = 200
  res.setHeader('Content-Type', mimeType[ext])
  res.setHeader('Cache-Control', 'max-age=3600')
  res.setHeader('Content-Encoding', 'gzip')
  res.setHeader('Last-Modified', new Date(expectedModified).toGMTString())

  // 7. gzip 压缩后，把文件流 pipe 回去
  const stream = fs.createReadStream(filePath, {
    flags: 'r', encoding: 'utf8'
  })
  stream.on('error', () => {
    res.writeHead(404)
    res.end()
```

```
  })
  stream.pipe(zlib.createGzip()).pipe(res)
})

server.on('error', error => console.log(error))
server.listen(4000, '127.0.0.1')
```