* + - * Node是一个基于Chrome V8引擎的Javascript运行环境
* 打开cmd，输入node，进入node工作环境
* 运行JS文件：进入文件同级目录，执行命令node example.js
* 大多数 fs 函数允许省略回调参数，即省略callback参数。在这种情况下，会使用默认回调，重新抛出错误。
* 他妈了个逼的我严重怀疑node官方文档上的POSIX系统就是指的linux
* windows以前记事本保存.txt文件的时候在前面自动加了三个字节，现在打开记事本，另存为“带有BOM的UTF-8”，还能看到这种效果。假如文件本身内容是“你好世界”，<Buffer e4 bd a0 e5 a5 bd e4 b8 96 e7 95 8c>，他保存成了<Buffer ef bb bf e4 bd a0 e5 a5 bd e4 b8 96 e7 95 8c>。ZERO WIDTH NO-BREAK SPACE的utf8编码是ef bb bf

编码问题Uint8Array，mysql，url模块，js传值还是传引用，this，node登录等等

options请求。get行post不行。最后加上几个行了，那个中间件。连接池

application/x-www-form-urlencoded form-data request-payload到底有什么区别？？？

application/json在angular里设置就可以，以request payload方式请求。在postman里只能是application/x-www-form-urlencoded，如果改成application/json就会报错。用了express得到请求参数很简单，用原生node到现在还不成功

res.send()只能是对象不能是字符串。能不能变成text/plain然后传纯字符串？

http/https 如何加密传输http报文 MD5还是AES？公钥私钥？adlxfhtxmzdqbcja

svg/cavas

antdesign中使用其他图标？

## npm

npm默认全局安装的包位置C:\\Users\\10332\\AppData\\Roaming\\npm

﻿删除全局模块

npm uninstall @vue/cli -g

## fileSystem

const fs = require('fs');

filename参数：

* 字符串。绝对或相对路径都可以
* 使用 Buffer 指定。主要用于将文件路径视为不透明字节序列的某些 POSIX 操作系统。 在这样的系统上，单个文件路径可以包含多种字符编码

fs.open(Buffer.from('/open/some/file.txt'), 'r', (err, fd) => {});

* 使用 file: 协议的WHATWG URL对象，形如

const fileUrl = new URL('file:///C:/tmp/hello');

fs.readFileSync(fileUrl);

const data = new Uint8Array(Buffer.from('Node.js中文网'));

fs.writeFile('文件.txt', data, (err) => {

if (err) throw err;

console.log('文件已被保存');

});

异步地将数据写入到一个文件，如果文件已存在则覆盖该文件

如果 options 是一个字符串，则它指定字符编码：

fs.writeFile('文件.txt', 'Node.js中文网', 'utf8', (err)=>{});

fs.open() 方法用于分配新的文件描述符。一旦被分配，则文件描述符可用于从文件读取数据、向文件写入数据、或请求关于文件的信息。

fs.open('/open/some/file.txt', 'r', (err, fd) => {

if (err) throw err;

fs.fstat(fd, (err, stat) => {

if (err) throw err;

// 使用文件属性。

// 始终关闭文件描述符！

fs.close(fd, (err) => {

if (err) throw err;

});

});

});

操作系统对于同时打开的文件描述符的数量有限制，因此当操作完成时关闭描述符至关重要。如果不这样做将导致内存泄漏，最终导致应用程序崩溃。

fs.access(path[, mode], callback)

* 测试用户对指定文件或目录的权限。
* mode参数，是一个整数。指定要检查什么权限。
* fs.constants.F\_OK 文件对调用进程可见。可以用于判断文件是否存在。如果没有mode，则fs.constants.F\_OK是默认值
* fs.constants.R\_OK 进程可以读取文件。
* fs.constants.W\_OK 进程可以写入文件。
* fs.constants.X\_OK 进程可以执行文件。在 Windows 上无效。

可以fs.constants.W\_OK | fs.constants.R\_OK

* 不建议在调用 fs.open()、 fs.readFile() 或 fs.writeFile() 之前使用 fs.access() 检查文件的可访问性。因为其他进程可能在access和read之间更改文件的状态。应该直接打开、读取或写入文件，如果文件无法访问则处理error。

fs.appendFile(path, data[, options], (err)=>{})

* 将数据追加到文件，如果文件尚不存在则创建该文件。
* data 可以是字符串或 Buffer。
* options指定字符编码：

fs.appendFile('m.txt', '追加的数据', 'utf8', callback);

默认也是utf8

* 黑马程序员讲师的憨皮，用的方法是读取整个文件，data+=追加的数据。

fs.chmod(path, mode, (err)=>{})

更改文件的权限

Windows 上，只能更改写入权限，并且不会实现群组、所有者或其他人的权限之间的区别。

fs.chown(path, uid, gid, (err)=>{})

同步地更改文件的所有者和群组。

fs.copyFile(src, dest[, flags], (err)=>{})

* 将 src 拷贝到 dest
* 默认情况下，如果 dest 已经存在，则覆盖它。不存在则创建
* flags 是一个可选的整数，指定拷贝操作的行为。可以用 | 连起来
* fs.constants.COPYFILE\_EXCL 如果 dest 已存在，则拷贝操作将失败。有用。
* fs.constants.COPYFILE\_FICLONE 拷贝操作将尝试创建写时拷贝（copy-on-write）链接。如果平台不支持写时拷贝，则使用后备的拷贝机制。看不懂
* fs.constants.COPYFILE\_FICLONE\_FORCE 拷贝操作将尝试创建写时拷贝链接。如果平台不支持写时拷贝，则拷贝操作将失败。

fs.copyFile('./a.txt', './b.txt', (err) => {

if (err) throw err;

console.log('源文件已拷贝到目标文件');

});

fs.createReadStream(path[, options])

创建读取流，用于比如16个字节16个字节的读

fs.mkdir(path[, options], (err, folder) => {})

创建目录

fs.open(path[, flags[, mode]], (err, fd) => {})

打开文件

mode 用于设置文件模式（权限和粘滞位permission and sticky bits），但仅限于创建文件时。 在 Windows 上，只能操作写权限，参阅 fs.chmod()。看不懂

fs.opendir(path[, options], callback)

创建一个 fs.Dir，其中包含所有用于更进一步读取和清理目录的的函数。

fs.readdir(path[, options], (err, files) => {})

* 读取目录的内容。
* files 是目录中文件和文件夹名的数组，不包括 '.' 和 '..'。形如

['a.txt', 'index.js', 'node\_modules', 'package.json']

* 可选的 options 参数可以是字符串，指定编码。也可以是一个对象，具有 encoding 属性，该属性指定文件名的字符编码。例如如果 encoding 设置为 'buffer'，则返回的文件名是 Buffer 对象。
* 如果 options.withFileTypes 设置为 true，则 files 数组将包含 fs.Dirent 对象。

fs.readFile(path[, options], (err, data)=>{})

* 异步地读取文件的全部内容。
* 如果没有options，则返回原始的 buffer。形如<Buffer 7b 0a 20 20 22 ... 3829 more bytes>

如果 options 是字符串，则指定字符编码：

fs.readFile('/a.txt', 'utf8', (err, data)=>{});

然后返回的数据就正常了

* 也可以不指定编码，通过data.toString()得到正常的数据
* 当path是目录时，报错
* fs.readFile() 函数会缓冲整个文件。 为了最小化内存成本，尽可能通过 fs.createReadStream() 进行流式传输。
* 另外，js官方教程没说if语句能不能不加大括号。测试是可以不加的，执行离if最近的语句。但是最好写在一行上，如果不加{}。所以if(err) throw err;会大量出现

fs.realpath(path[, options], (err, resolvedPath)=>{})

通过解析 .、 .. 和符号链接异步地计算规范路径名。

说的是规范路径名，我看了半天，也没发现和绝对路径有啥区别

fs.rename('a.txt', 'b.txt', (err)=>{})

重命名

fs.stat(path, (err, stats)=>{

})

回调函数的参数stats 是一个 fs.Stats 对象。形如

Stats {

dev: 281966685,

mode: 33206,

nlink: 1,

uid: 0,

gid: 0,

rdev: 0,

blksize: 4096,

ino: 5910974511159195,

size: 3,

blocks: 0,

atimeMs: 1575726940378.4343,

mtimeMs: 1575726903020.9878,

ctimeMs: 1575732654710.6191,

birthtimeMs: 1575724379953.591,

atime: 2019-12-07T13:55:40.378Z,

mtime: 2019-12-07T13:55:03.021Z,

ctime: 2019-12-07T15:30:54.711Z,

birthtime: 2019-12-07T13:12:59.954Z

}

* atimeMs、mtimeMs、ctimeMs和birthtimeMs属性保存相应时间，以毫秒为单位。
* atimeNs、mtimeNs、ctimeNs和birthtimeNs属性是保存相应时间，以纳秒为单位。
* atime、mtime、ctime和birthtime 是对应时间的 Date 对象。 Date 值和数值没有关联性。赋值新的数值、或者改变 Date 的值，都将不会影响到对应的属性。
* atime 访问时间，上次访问文件数据的时间。
* mtime 修改时间，上次修改文件数据的时间。
* ctime 更改时间，上次更改文件状态（修改索引节点数据）的时间。
* birthtime 创建时间，创建文件的时间。
* stats.isDirectory() 是否是目录
* stats.size 文件的大小，以字节为单位
* stats.isFile() 如果fs.Stats 对象描述常规文件，则返回 true。
* stats.isSocket() 如果 fs.Stats 对象描述套接字，则返回 true。

fs.watch(filename[, options][, (eventType, filename)=>{} ])

* options如果字符串，则它指定 encoding。 否则，options 应传入对象。
* persistent <boolean> 指示如果文件已正被监视，进程是否应继续运行。默认值: true。
* recursive <boolean> 指示应该监视所有子目录，还是仅监视当前目录。默认值: false。
* encoding <string> 指定用于传给监听器的文件名的字符编码。默认'utf8'。
* 监视 filename 的更改，filename是文件或目录。
* 返回fs.FSWatcher 对象
* 回调函数，eventType是'rename'或'change'，本机是rename。
* filename是触发事件的文件的名称。和watch函数中filename参数一样。每当文件名在目录中出现或消失时，就会触发事件和回调。比如a.txt改成b.txt，回调触发；b.txt改为c.txt，不触发；c.txt改为d.txt，不触发；d.txt改回a.txt，触发
* 如果监听目录，则目录中每个文件名发生改变都触发。
* 使用 fs.watch() 比 fs.watchFile() 和 fs.unwatchFile() 更高效。 应尽可能使用 fs.watch() 代替 fs.watchFile() 和 fs.unwatchFile()。
* 但是不知道如何解除监听，试了一下unwatchFile不能接触watch绑定的监听器，而又没有unwatch方法

监听器回调绑定在由 fs.FSWatcher 触发的 'change' 事件上，但它与 eventType 的 'change' 值不是一回事。

fs.unwatchFile(filename[, listener])

* 停止监视 filename 的变化。如果指定了 listener，仅移除此特定监听器，否则，移除filename上所有的监听器。

## http

const http = require('http');

var server = http.createServer();

server.listen('8000');

* http.createServer([options][, requestListener])返回新建的 http.Server 实例。

server.listen()

启动一个服务器来监听连接

* server.listen(handle[, backlog][, callback])
* server.listen(options[, callback])
* server.listen(path[, backlog][, callback]) 用于 IPC 服务器。
* server.listen([port[, host[, backlog]]][, callback]) 用于 TCP 服务器。

server.on('request', (request, response)=>{

console.log(request.url);

response.write('world');

response.write('11111');

response.end();

});

* server绑定request事件
* request请求对象 response响应对象，给客户端发送消息
* response.end([data[, encoding]][, callback])

必须在每个响应上调用response.end() 方法

也可以有参数response.end('zaijian')，相当于response.write('zaijian')之后再response.end()

callback无任何参数，在end结束后执行

* response.write()可以使用多次
* request.url 拿到请求路径/ /nihao/123 /nihao/123/

qq浏览器会自动请求127.0.0.1:8000/favicon.ico

* request.method 拿到请求方法GET POST

response.write('你好');

write响应中文的时候，浏览器显示乱码，浣犲ソ，应该是因为默认按GBK编码的html内容。但是postman能正确显示，检查Network里边的response也可以，应该是默认utf-8编码。

response.setHeader('Content-type', 'text/plain; charset=utf-8');

然后显示就正确了

* response.setHeader('Content-type', 'text/html; charset=utf-8');

text/plain按纯文本解析，也可以按html解析，换成text/html既可

* response的Headers正常情况下有这三个：
* Connection: keep-alive
* Content-Length: 871
* Date: Sun, 01 Dec 2019 06:31:43 GMT

setHeader以后会加上一条

* Content-type: text/html; charset=utf-8

如果响应的HTML文件里包含图片，那么浏览器解析HTML解析到图片时，会另外发送一个请求。所以响应时应该判断发起的请求是请求什么，分为html和其他资源两类

if(request.url=='' || request.url=='/'){ //如果url什么都没有

response.setHeader('Content-type', 'text/html; charset=utf-8');

fs.readFile('./ca.html', (err, data)=>{

if(err){

throw err;

}

response.end(data);

})

}else{ //如果url请求其他资源

fs.readFile('.'+request.url, (err, data)=>{

if(err){

throw err;

}

response.end(data); //响应其他资源时不需要设置header

})

}

响应图片时不需要设置header

解析GET参数

const url = require('url');

url 模块用于处理与解析 URL

url.parse(myURL)返回一个对象

url.parse(myURL).query返回GET的参数，形如id=123&name=456

url.parse(myURL, true).query 返回形如{id: '12', name: 'feiwu'}

url.parse(myURL, true),pathname

url.parse()、urlObject对象等，新版本node已全部废弃！！！！！！！

一个完整的post请求体。即console.log(req)

IncomingMessage {

\_readableState: ReadableState {

objectMode: false,

highWaterMark: 16384,

buffer: BufferList { head: null, tail: null, length: 0 },

length: 0,

pipes: null,

pipesCount: 0,

flowing: null,

ended: false,

endEmitted: false,

reading: false,

sync: true,

needReadable: false,

emittedReadable: false,

readableListening: false,

resumeScheduled: false,

paused: true,

emitClose: true,

autoDestroy: false,

destroyed: false,

defaultEncoding: 'utf8',

awaitDrain: 0,

readingMore: true,

decoder: null,

encoding: null

},

readable: true,

\_events: [Object: null prototype] {

end: [Function: resetHeadersTimeoutOnReqEnd]

},

\_eventsCount: 1,

\_maxListeners: undefined,

socket: Socket {

connecting: false,

\_hadError: false,

\_parent: null,

\_host: null,

\_readableState: ReadableState {

objectMode: false,

highWaterMark: 16384,

buffer: BufferList { head: null, tail: null, length: 0 },

length: 0,

pipes: null,

pipesCount: 0,

flowing: true,

ended: false,

endEmitted: false,

reading: true,

sync: false,

needReadable: true,

emittedReadable: false,

readableListening: false,

resumeScheduled: false,

paused: false,

emitClose: false,

autoDestroy: false,

destroyed: false,

defaultEncoding: 'utf8',

awaitDrain: 0,

readingMore: false,

decoder: null,

encoding: null

},

readable: true,

\_events: [Object: null prototype] {

end: [Array],

timeout: [Function: socketOnTimeout],

data: [Function: bound socketOnData],

error: [Function: socketOnError],

close: [Array],

drain: [Function: bound socketOnDrain],

resume: [Function: onSocketResume],

pause: [Function: onSocketPause]

},

\_eventsCount: 8,

\_maxListeners: undefined,

\_writableState: WritableState {

objectMode: false,

highWaterMark: 16384,

finalCalled: false,

needDrain: false,

ending: false,

ended: false,

finished: false,

destroyed: false,

decodeStrings: false,

defaultEncoding: 'utf8',

length: 0,

writing: false,

corked: 0,

sync: false,

bufferProcessing: false,

onwrite: [Function: bound onwrite],

writecb: null,

writelen: 0,

bufferedRequest: null,

lastBufferedRequest: null,

pendingcb: 0,

prefinished: false,

errorEmitted: false,

emitClose: false,

autoDestroy: false,

bufferedRequestCount: 0,

corkedRequestsFree: [Object]

},

writable: true,

allowHalfOpen: true,

\_sockname: null,

\_pendingData: null,

\_pendingEncoding: '',

server: Server {

\_events: [Object: null prototype],

\_eventsCount: 2,

\_maxListeners: undefined,

\_connections: 1,

\_handle: [TCP],

\_usingWorkers: false,

\_workers: [],

\_unref: false,

allowHalfOpen: true,

pauseOnConnect: false,

httpAllowHalfOpen: false,

timeout: 120000,

keepAliveTimeout: 5000,

maxHeadersCount: null,

headersTimeout: 40000,

\_connectionKey: '6::::3000',

[Symbol(IncomingMessage)]: [Function: IncomingMessage],

[Symbol(ServerResponse)]: [Function: ServerResponse],

[Symbol(asyncId)]: 5

},

\_server: Server {

\_events: [Object: null prototype],

\_eventsCount: 2,

\_maxListeners: undefined,

\_connections: 1,

\_handle: [TCP],

\_usingWorkers: false,

\_workers: [],

\_unref: false,

allowHalfOpen: true,

pauseOnConnect: false,

httpAllowHalfOpen: false,

timeout: 120000,

keepAliveTimeout: 5000,

maxHeadersCount: null,

headersTimeout: 40000,

\_connectionKey: '6::::3000',

[Symbol(IncomingMessage)]: [Function: IncomingMessage],

[Symbol(ServerResponse)]: [Function: ServerResponse],

[Symbol(asyncId)]: 5

},

timeout: 120000,

parser: HTTPParser {

'0': [Function: parserOnHeaders],

'1': [Function: parserOnHeadersComplete],

'2': [Function: parserOnBody],

'3': [Function: parserOnMessageComplete],

'4': [Function: bound onParserExecute],

\_headers: [],

\_url: '',

socket: [Circular],

incoming: [Circular],

outgoing: null,

maxHeaderPairs: 2000,

\_consumed: true,

onIncoming: [Function: bound parserOnIncoming],

parsingHeadersStart: 0

},

on: [Function: socketListenerWrap],

addListener: [Function: socketListenerWrap],

prependListener: [Function: socketListenerWrap],

\_paused: false,

\_httpMessage: ServerResponse {

\_events: [Object: null prototype],

\_eventsCount: 1,

\_maxListeners: undefined,

outputData: [],

outputSize: 0,

writable: true,

\_last: false,

chunkedEncoding: false,

shouldKeepAlive: true,

useChunkedEncodingByDefault: true,

sendDate: true,

\_removedConnection: false,

\_removedContLen: false,

\_removedTE: false,

\_contentLength: null,

\_hasBody: true,

\_trailer: '',

finished: false,

\_headerSent: false,

socket: [Circular],

connection: [Circular],

\_header: null,

\_onPendingData: [Function: bound updateOutgoingData],

\_sent100: false,

\_expect\_continue: false,

req: [Circular],

locals: [Object: null prototype] {},

[Symbol(kNeedDrain)]: false,

[Symbol(isCorked)]: false,

[Symbol(kOutHeaders)]: [Object: null prototype]

},

[Symbol(asyncId)]: 36,

[Symbol(kHandle)]: TCP {

reading: true,

onconnection: null,

\_consumed: true,

[Symbol(owner)]: [Circular]

},

[Symbol(lastWriteQueueSize)]: 0,

[Symbol(timeout)]: Timeout {

\_idleTimeout: 120000,

\_idlePrev: [TimersList],

\_idleNext: [TimersList],

\_idleStart: 4817,

\_onTimeout: [Function: bound ],

\_timerArgs: undefined,

\_repeat: null,

\_destroyed: false,

[Symbol(refed)]: false,

[Symbol(asyncId)]: 51,

[Symbol(triggerId)]: 38

},

[Symbol(kBuffer)]: null,

[Symbol(kBufferCb)]: null,

[Symbol(kBufferGen)]: null,

[Symbol(kBytesRead)]: 0,

[Symbol(kBytesWritten)]: 0

},

connection: Socket {

connecting: false,

\_hadError: false,

\_parent: null,

\_host: null,

\_readableState: ReadableState {

objectMode: false,

highWaterMark: 16384,

buffer: BufferList { head: null, tail: null, length: 0 },

length: 0,

pipes: null,

pipesCount: 0,

flowing: true,

ended: false,

endEmitted: false,

reading: true,

sync: false,

needReadable: true,

emittedReadable: false,

readableListening: false,

resumeScheduled: false,

paused: false,

emitClose: false,

autoDestroy: false,

destroyed: false,

defaultEncoding: 'utf8',

awaitDrain: 0,

readingMore: false,

decoder: null,

encoding: null

},

readable: true,

\_events: [Object: null prototype] {

end: [Array],

timeout: [Function: socketOnTimeout],

data: [Function: bound socketOnData],

error: [Function: socketOnError],

close: [Array],

drain: [Function: bound socketOnDrain],

resume: [Function: onSocketResume],

pause: [Function: onSocketPause]

},

\_eventsCount: 8,

\_maxListeners: undefined,

\_writableState: WritableState {

objectMode: false,

highWaterMark: 16384,

finalCalled: false,

needDrain: false,

ending: false,

ended: false,

finished: false,

destroyed: false,

decodeStrings: false,

defaultEncoding: 'utf8',

length: 0,

writing: false,

corked: 0,

sync: false,

bufferProcessing: false,

onwrite: [Function: bound onwrite],

writecb: null,

writelen: 0,

bufferedRequest: null,

lastBufferedRequest: null,

pendingcb: 0,

prefinished: false,

errorEmitted: false,

emitClose: false,

autoDestroy: false,

bufferedRequestCount: 0,

corkedRequestsFree: [Object]

},

writable: true,

allowHalfOpen: true,

\_sockname: null,

\_pendingData: null,

\_pendingEncoding: '',

server: Server {

\_events: [Object: null prototype],

\_eventsCount: 2,

\_maxListeners: undefined,

\_connections: 1,

\_handle: [TCP],

\_usingWorkers: false,

\_workers: [],

\_unref: false,

allowHalfOpen: true,

pauseOnConnect: false,

httpAllowHalfOpen: false,

timeout: 120000,

keepAliveTimeout: 5000,

maxHeadersCount: null,

headersTimeout: 40000,

\_connectionKey: '6::::3000',

[Symbol(IncomingMessage)]: [Function: IncomingMessage],

[Symbol(ServerResponse)]: [Function: ServerResponse],

[Symbol(asyncId)]: 5

},

\_server: Server {

\_events: [Object: null prototype],

\_eventsCount: 2,

\_maxListeners: undefined,

\_connections: 1,

\_handle: [TCP],

\_usingWorkers: false,

\_workers: [],

\_unref: false,

allowHalfOpen: true,

pauseOnConnect: false,

httpAllowHalfOpen: false,

timeout: 120000,

keepAliveTimeout: 5000,

maxHeadersCount: null,

headersTimeout: 40000,

\_connectionKey: '6::::3000',

[Symbol(IncomingMessage)]: [Function: IncomingMessage],

[Symbol(ServerResponse)]: [Function: ServerResponse],

[Symbol(asyncId)]: 5

},

timeout: 120000,

parser: HTTPParser {

'0': [Function: parserOnHeaders],

'1': [Function: parserOnHeadersComplete],

'2': [Function: parserOnBody],

'3': [Function: parserOnMessageComplete],

'4': [Function: bound onParserExecute],

\_headers: [],

\_url: '',

socket: [Circular],

incoming: [Circular],

outgoing: null,

maxHeaderPairs: 2000,

\_consumed: true,

onIncoming: [Function: bound parserOnIncoming],

parsingHeadersStart: 0

},

on: [Function: socketListenerWrap],

addListener: [Function: socketListenerWrap],

prependListener: [Function: socketListenerWrap],

\_paused: false,

\_httpMessage: ServerResponse {

\_events: [Object: null prototype],

\_eventsCount: 1,

\_maxListeners: undefined,

outputData: [],

outputSize: 0,

writable: true,

\_last: false,

chunkedEncoding: false,

shouldKeepAlive: true,

useChunkedEncodingByDefault: true,

sendDate: true,

\_removedConnection: false,

\_removedContLen: false,

\_removedTE: false,

\_contentLength: null,

\_hasBody: true,

\_trailer: '',

finished: false,

\_headerSent: false,

socket: [Circular],

connection: [Circular],

\_header: null,

\_onPendingData: [Function: bound updateOutgoingData],

\_sent100: false,

\_expect\_continue: false,

req: [Circular],

locals: [Object: null prototype] {},

[Symbol(kNeedDrain)]: false,

[Symbol(isCorked)]: false,

[Symbol(kOutHeaders)]: [Object: null prototype]

},

[Symbol(asyncId)]: 36,

[Symbol(kHandle)]: TCP {

reading: true,

onconnection: null,

\_consumed: true,

[Symbol(owner)]: [Circular]

},

[Symbol(lastWriteQueueSize)]: 0,

[Symbol(timeout)]: Timeout {

\_idleTimeout: 120000,

\_idlePrev: [TimersList],

\_idleNext: [TimersList],

\_idleStart: 4817,

\_onTimeout: [Function: bound ],

\_timerArgs: undefined,

\_repeat: null,

\_destroyed: false,

[Symbol(refed)]: false,

[Symbol(asyncId)]: 51,

[Symbol(triggerId)]: 38

},

[Symbol(kBuffer)]: null,

[Symbol(kBufferCb)]: null,

[Symbol(kBufferGen)]: null,

[Symbol(kBytesRead)]: 0,

[Symbol(kBytesWritten)]: 0

},

httpVersionMajor: 1,

httpVersionMinor: 1,

httpVersion: '1.1',

complete: false,

headers: {

host: 'localhost:3000',

connection: 'keep-alive',

'content-length': '52',

accept: 'application/json, text/plain, \*/\*',

origin: 'http://localhost:4200',

'user-agent': 'Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/70.0.3538.25 Safari/537.36 Core/1.70.3732.400 QQBrowser/10.5.3819.400',

'content-type': 'application/json',

referer: 'http://localhost:4200/login',

'accept-encoding': 'gzip, deflate, br',

'accept-language': 'zh-CN,zh;q=0.9'

},

rawHeaders: [

'Host',

'localhost:3000',

'Connection',

'keep-alive',

'Content-Length',

'52',

'Accept',

'application/json, text/plain, \*/\*',

'Origin',

'http://localhost:4200',

'User-Agent',

'Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/70.0.3538.25 Safari/537.36 Core/1.70.3732.400 QQBrowser/10.5.3819.400',

'Content-Type',

'application/json',

'Referer',

'http://localhost:4200/login',

'Accept-Encoding',

'gzip, deflate, br',

'Accept-Language',

'zh-CN,zh;q=0.9'

],

trailers: {},

rawTrailers: [],

aborted: false,

upgrade: false,

url: '/login',

method: 'POST',

statusCode: null,

statusMessage: null,

client: Socket {

connecting: false,

\_hadError: false,

\_parent: null,

\_host: null,

\_readableState: ReadableState {

objectMode: false,

highWaterMark: 16384,

buffer: BufferList { head: null, tail: null, length: 0 },

length: 0,

pipes: null,

pipesCount: 0,

flowing: true,

ended: false,

endEmitted: false,

reading: true,

sync: false,

needReadable: true,

emittedReadable: false,

readableListening: false,

resumeScheduled: false,

paused: false,

emitClose: false,

autoDestroy: false,

destroyed: false,

defaultEncoding: 'utf8',

awaitDrain: 0,

readingMore: false,

decoder: null,

encoding: null

},

readable: true,

\_events: [Object: null prototype] {

end: [Array],

timeout: [Function: socketOnTimeout],

data: [Function: bound socketOnData],

error: [Function: socketOnError],

close: [Array],

drain: [Function: bound socketOnDrain],

resume: [Function: onSocketResume],

pause: [Function: onSocketPause]

},

\_eventsCount: 8,

\_maxListeners: undefined,

\_writableState: WritableState {

objectMode: false,

highWaterMark: 16384,

finalCalled: false,

needDrain: false,

ending: false,

ended: false,

finished: false,

destroyed: false,

decodeStrings: false,

defaultEncoding: 'utf8',

length: 0,

writing: false,

corked: 0,

sync: false,

bufferProcessing: false,

onwrite: [Function: bound onwrite],

writecb: null,

writelen: 0,

bufferedRequest: null,

lastBufferedRequest: null,

pendingcb: 0,

prefinished: false,

errorEmitted: false,

emitClose: false,

autoDestroy: false,

bufferedRequestCount: 0,

corkedRequestsFree: [Object]

},

writable: true,

allowHalfOpen: true,

\_sockname: null,

\_pendingData: null,

\_pendingEncoding: '',

server: Server {

\_events: [Object: null prototype],

\_eventsCount: 2,

\_maxListeners: undefined,

\_connections: 1,

\_handle: [TCP],

\_usingWorkers: false,

\_workers: [],

\_unref: false,

allowHalfOpen: true,

pauseOnConnect: false,

httpAllowHalfOpen: false,

timeout: 120000,

keepAliveTimeout: 5000,

maxHeadersCount: null,

headersTimeout: 40000,

\_connectionKey: '6::::3000',

[Symbol(IncomingMessage)]: [Function: IncomingMessage],

[Symbol(ServerResponse)]: [Function: ServerResponse],

[Symbol(asyncId)]: 5

},

\_server: Server {

\_events: [Object: null prototype],

\_eventsCount: 2,

\_maxListeners: undefined,

\_connections: 1,

\_handle: [TCP],

\_usingWorkers: false,

\_workers: [],

\_unref: false,

allowHalfOpen: true,

pauseOnConnect: false,

httpAllowHalfOpen: false,

timeout: 120000,

keepAliveTimeout: 5000,

maxHeadersCount: null,

headersTimeout: 40000,

\_connectionKey: '6::::3000',

[Symbol(IncomingMessage)]: [Function: IncomingMessage],

[Symbol(ServerResponse)]: [Function: ServerResponse],

[Symbol(asyncId)]: 5

},

timeout: 120000,

parser: HTTPParser {

'0': [Function: parserOnHeaders],

'1': [Function: parserOnHeadersComplete],

'2': [Function: parserOnBody],

'3': [Function: parserOnMessageComplete],

'4': [Function: bound onParserExecute],

\_headers: [],

\_url: '',

socket: [Circular],

incoming: [Circular],

outgoing: null,

maxHeaderPairs: 2000,

\_consumed: true,

onIncoming: [Function: bound parserOnIncoming],

parsingHeadersStart: 0

},

on: [Function: socketListenerWrap],

addListener: [Function: socketListenerWrap],

prependListener: [Function: socketListenerWrap],

\_paused: false,

\_httpMessage: ServerResponse {

\_events: [Object: null prototype],

\_eventsCount: 1,

\_maxListeners: undefined,

outputData: [],

outputSize: 0,

writable: true,

\_last: false,

chunkedEncoding: false,

shouldKeepAlive: true,

useChunkedEncodingByDefault: true,

sendDate: true,

\_removedConnection: false,

\_removedContLen: false,

\_removedTE: false,

\_contentLength: null,

\_hasBody: true,

\_trailer: '',

finished: false,

\_headerSent: false,

socket: [Circular],

connection: [Circular],

\_header: null,

\_onPendingData: [Function: bound updateOutgoingData],

\_sent100: false,

\_expect\_continue: false,

req: [Circular],

locals: [Object: null prototype] {},

[Symbol(kNeedDrain)]: false,

[Symbol(isCorked)]: false,

[Symbol(kOutHeaders)]: [Object: null prototype]

},

[Symbol(asyncId)]: 36,

[Symbol(kHandle)]: TCP {

reading: true,

onconnection: null,

\_consumed: true,

[Symbol(owner)]: [Circular]

},

[Symbol(lastWriteQueueSize)]: 0,

[Symbol(timeout)]: Timeout {

\_idleTimeout: 120000,

\_idlePrev: [TimersList],

\_idleNext: [TimersList],

\_idleStart: 4817,

\_onTimeout: [Function: bound ],

\_timerArgs: undefined,

\_repeat: null,

\_destroyed: false,

[Symbol(refed)]: false,

[Symbol(asyncId)]: 51,

[Symbol(triggerId)]: 38

},

[Symbol(kBuffer)]: null,

[Symbol(kBufferCb)]: null,

[Symbol(kBufferGen)]: null,

[Symbol(kBytesRead)]: 0,

[Symbol(kBytesWritten)]: 0

},

\_consuming: false,

\_dumped: false,

next: [Function: next],

baseUrl: '',

originalUrl: '/login',

\_parsedUrl: Url {

protocol: null,

slashes: null,

auth: null,

host: null,

port: null,

hostname: null,

hash: null,

search: null,

query: null,

pathname: '/login',

path: '/login',

href: '/login',

\_raw: '/login'

},

params: {},

query: {},

res: ServerResponse {

\_events: [Object: null prototype] { finish: [Function: bound resOnFinish] },

\_eventsCount: 1,

\_maxListeners: undefined,

outputData: [],

outputSize: 0,

writable: true,

\_last: false,

chunkedEncoding: false,

shouldKeepAlive: true,

useChunkedEncodingByDefault: true,

sendDate: true,

\_removedConnection: false,

\_removedContLen: false,

\_removedTE: false,

\_contentLength: null,

\_hasBody: true,

\_trailer: '',

finished: false,

\_headerSent: false,

socket: Socket {

connecting: false,

\_hadError: false,

\_parent: null,

\_host: null,

\_readableState: [ReadableState],

readable: true,

\_events: [Object: null prototype],

\_eventsCount: 8,

\_maxListeners: undefined,

\_writableState: [WritableState],

writable: true,

allowHalfOpen: true,

\_sockname: null,

\_pendingData: null,

\_pendingEncoding: '',

server: [Server],

\_server: [Server],

timeout: 120000,

parser: [HTTPParser],

on: [Function: socketListenerWrap],

addListener: [Function: socketListenerWrap],

prependListener: [Function: socketListenerWrap],

\_paused: false,

\_httpMessage: [Circular],

[Symbol(asyncId)]: 36,

[Symbol(kHandle)]: [TCP],

[Symbol(lastWriteQueueSize)]: 0,

[Symbol(timeout)]: Timeout {

\_idleTimeout: 120000,

\_idlePrev: [TimersList],

\_idleNext: [TimersList],

\_idleStart: 4817,

\_onTimeout: [Function: bound ],

\_timerArgs: undefined,

\_repeat: null,

\_destroyed: false,

[Symbol(refed)]: false,

[Symbol(asyncId)]: 51,

[Symbol(triggerId)]: 38

},

[Symbol(kBuffer)]: null,

[Symbol(kBufferCb)]: null,

[Symbol(kBufferGen)]: null,

[Symbol(kBytesRead)]: 0,

[Symbol(kBytesWritten)]: 0

},

connection: Socket {

connecting: false,

\_hadError: false,

\_parent: null,

\_host: null,

\_readableState: [ReadableState],

readable: true,

\_events: [Object: null prototype],

\_eventsCount: 8,

\_maxListeners: undefined,

\_writableState: [WritableState],

writable: true,

allowHalfOpen: true,

\_sockname: null,

\_pendingData: null,

\_pendingEncoding: '',

server: [Server],

\_server: [Server],

timeout: 120000,

parser: [HTTPParser],

on: [Function: socketListenerWrap],

addListener: [Function: socketListenerWrap],

prependListener: [Function: socketListenerWrap],

\_paused: false,

\_httpMessage: [Circular],

[Symbol(asyncId)]: 36,

[Symbol(kHandle)]: [TCP],

[Symbol(lastWriteQueueSize)]: 0,

[Symbol(timeout)]: Timeout {

\_idleTimeout: 120000,

\_idlePrev: [TimersList],

\_idleNext: [TimersList],

\_idleStart: 4817,

\_onTimeout: [Function: bound ],

\_timerArgs: undefined,

\_repeat: null,

\_destroyed: false,

[Symbol(refed)]: false,

[Symbol(asyncId)]: 51,

[Symbol(triggerId)]: 38

},

[Symbol(kBuffer)]: null,

[Symbol(kBufferCb)]: null,

[Symbol(kBufferGen)]: null,

[Symbol(kBytesRead)]: 0,

[Symbol(kBytesWritten)]: 0

},

\_header: null,

\_onPendingData: [Function: bound updateOutgoingData],

\_sent100: false,

\_expect\_continue: false,

req: [Circular],

locals: [Object: null prototype] {},

[Symbol(kNeedDrain)]: false,

[Symbol(isCorked)]: false,

[Symbol(kOutHeaders)]: [Object: null prototype] {

'x-powered-by': [Array],

'access-control-allow-origin': [Array],

'access-control-allow-headers': [Array],

'access-control-allow-methods': [Array]

}

},

route: Route { path: '/login', stack: [ [Layer] ], methods: { post: true } }

}

## url

const url=require('url');

new URL(input[, base])

input <string>是要解析的URL。如果 input 是相对路径，则需要 base。如果 input 是绝对路径，则忽略 base。

const myURL = new URL('/nihao', 'https://example.org/');

const myURL = new URL('nihao', 'https://example.org/');

结果都是https://example.org/nihao

const myURL = new URL('//nihao', 'https://example.org/');

相当于写错了，结果是https://nihao

const myURL = new URL('/nihao#section1', 'https://example.org:8080/');

* url.hash #section1
* url.host example.org:8080

host包括端口号，hostname不包括

* url.hostname example.org
* url.href https://example.org:8080/nihao#section1

获取序列化的 URL

等同于调用 url.toString()。

可以修改，相当于使用 new URL创建新的URL对象。 此URL 对象的每个属性都将被修改

* url.origin https://example.org:8080
* url.pathname /nihao
* url.port 8080
* url.protocol https:

package.json

package-lock.json就是锁定安装时的包的版本号，并且需要上传到git，以保证其他人在npm install时大家的依赖能保证一致。

CommonJS

exports.a=1;

此JS文件可当做一个模块，导出变量a

const nihao=require('./ew.js');

console.log(nihao.a);

其他模块导入

* require的参数为绝对路径或相对路径，相对路径必须加../或./，可以不写后缀名.js
* 没有指名路径，就是加载了核心模块或者第三方模块。指明路径就是自定义模块
* 每个模块，即每个js文件里有一个module对象，代表模块自身。module对象有一个exports属性，exports.a=1这个exports就是module.exports的引用。也可以通过module.exports.a=1导出数据
* 模块加载顺序：遇到第三方模块，找node\_modules文件夹，从当前文件的同级目录开始找，找不到就到上一级目录，找不到就一直往上找，直到E盘、根目录…

## 数据库

官方文档中无任何关于数据库的

npm install mysql

第三方模块

var mysql=require('mysql')

教程上的写法：链式操作

Jquery可以这么写：$('#a').css('background', 'red').width('23').css('background', 'red)

原理：链式操作，return this

数据库中的操作可以这么写

db.where('id=19').select();

模仿jQuery链式操作

* 首先明确，id=19需要保存下来传给select,而这个方法返回this，所以需要在this中添加属性condition，

module.exports={

condition:undifined,

where: (condition) => {

this.condition=condition;

return this;

}

select: () => {

if(this.condition === undefined){

let sql='select \* from xx';

}else{

let sql='select \* from xx where '+this.condition;

}

this.condition = undefined; //为了不影响以后的操作

}

}

* 注意：不能用箭头函数。箭头函数体内的this对象，就是定义时所在的对象，而不是使用时所在的对象。如果使用箭头函数，&）￥（）（@%&）#@￥\*）（#@%
* 但是也有缺点。如果只db.where('id=19')，就给condition赋了值，如果此时db.select()，本来想查找所有的，结果只找到id=19这一项。所以where不能单独使用。又没有办法强制它不单独使用。
* select方法千万不能return this。如果return this，则可以db.select().where('id=19')，然后什么都不操作，condition又被赋值了。即，链式操作的终结环节不能return this

综上所述，这样写其实挺垃圾的。限制很多，也没方便多少。

接受POST请求

需要node.js网络模块net

如果这一次请求中接收到了数据，那么data事件触发

数据全部接受完毕，end事件触发

post请求的参数在请求体中，get在请求头中

server.on('request', function(request, response) {

if (request.method == 'POST') {

var data = '';

request.on('data', function(tempData) {

data += tempData

});

request.end('end', function() {

console.log(data);

});

}

})

const querystring = require('querystring')

let obj=querystring.parse(data\_post)

querystring模块

把形如name=%E9%90%E2&age=12&hobby=%2D%6C转成{name: '李四', age:12, hobby: '唱'}的对象

response.end("<script>alert('nihao')</script>")

还能这么用

## 借助formidable实现文件上传

var formidable = new formidable.IncomingForm();

formidable.parse(request, (err, fields, files)=>{

fs.rename(files.imgs.path, './imgs/'+files.imgs.name, (err)=>{

}

})

imgs应该是传上来的时候表单的name

## cookie-session

第三方组件在express中实现注册登录功能

## koa

## package.json

"name": "SB-Admin-BS4-Angular-5",

"version": "4.0.0",

package.json 中有非常多的属性，其中必须填写的只有两个：name 和 version ，这两个属性组成一个 npm 模块的唯一标识。

在 package.json 文件中scripts属性增加脚本

"scripts": {

"ng": "ng",

"start": "ng serve",

"build": "ng build --prod",

},

有了这句话，就可以运行

npm start

效果等于ng serve

# express

const express=require('express');

const app=express();

## 路由处理器

app.get('/', function(request, response){

response.send('helloworld');

})

app.get('/nihao', function(request, response){

response.send({ user: 'tobi' });

})

app.listen(8080, function(){

console.log('监听127.0.0.1:8080')

})

* get方法监听get请求，参数是url和回调函数。get可以被替换为post等
* 这种风格是express的内置路由，相当于http模块和路由模块合并了
* app.listen(8080) 开始监听8080端口。官方文档监听的是3000。不过最终也要监听80
* 可以send对象，以json格式传输

router.get('/', (req, res, next)=>{

res.send('结束1')

})

router.get('/', (req, res, next)=>{

res.send('结束2')

})

如果有两个router.get()处理同一个路由，那么后一个不起作用。所以要使用中间件

app.all('/secret', function (req, res, next) {

console.log('Accessing the secret section ...')

next()

})

app.all()监听所有http请求方法

app.route('/book')

.get(function (req, res) {

res.send('Get a random book')

})

.post(function (req, res) {

res.send('Add a book')

})

可以用app.route()方法，支持链式调用

## 中间件

var myLogger = function (req, res, next) {

console.log('LOGGED')

next()

}

app.use(myLogger);

* app.use()指定中间件middleware。myLogger中间件在路由到根路径(/)之前运行，在get()回调之前运行
* app.use()和app.get()按声明的顺序执行。
* 如果use()在get()之后，而且get()方法没有next参数，则路径的路由处理程序终止请求响应周期，中间件不会运行。所以如果app.use()在app.get()之后，必须在app.get()回调函数中也加上next参数，函数体中运行next()，切换到下一个回调控制器
* 调用next()将请求传递给堆栈中的下一个中间件函数。
* 可以利用中间件修改requst和response对象，因为所有中间件函数共用一个request和response
* app.use()不指定http方法，所以所有http方法的响应都要经过myLogger
* 路由路径

路径可以是strings字符串, string patterns字符串模式？或者regular expressions正则表达式

app.get('/random.text', function (req, res) {

res.send('random.text')

})

匹配/random.text

连字符-和点.还是字面量，但是不作为req.params的属性名也就是路由参数名的组成部分

/ab?cd

匹配abcd和acd

?表示有或没有

/ab+cd

匹配abcd,abbcd,abbbcd

/ab\*cd

abcd,abxcd,abRANDOMcd,ab123cd

/ab(cd)?e

/abe，/abcde

app.get(/a/, function (req, res) {

})

anything with an “a” in it.

app.get(/.\*fly$/, function (req, res) {

})

butterflyanddragonfly, 但是notbutterflyman,dragonflyman不行

以上是正则表达式的例子

* 路由参数

app.get('/users/:userId/books/:bookId', function (req, res) {

})

Request URL: http://localhost:3000/users/34/books/8989

req.params: { "userId": "34", "bookId": "8989" }

/plantae/:genus.:species

Request URL: http://localhost:3000/plantae/Prunus.persica

req.params: { "genus": "Prunus", "species": "persica" }

因为点.不参与组成路由参数名，所以在这里相当于分隔符。-同理

Express4.0有异常，不推荐使用\*应使用{0,}

* next

app.get('/example/b', function (req, res, next) {

next()

}, function (req, res) {

})

app.get()可以有一组处理器函数，也可以是一个函数数组

var cb0 = function (req, res, next) {

console.log('CB0')

next('cb2')

}

var cb1 = function (req, res, next) {

console.log('CB1')

next()

}

var cb2 = function (req, res) {

res.send('Hello cb2!')

}

app.get('/example/c', [cb0, cb1, cb2])

next('cb2')可以跳过cb1

router.get('/user/:id', function (req, res, next) {

if (req.params.id === '0') next('route')

else next()

}, function (req, res, next) {

res.send('regular')

})

router.get('/user/:id', function (req, res, next) {

res.send('special')

})

路由级中间件的next

next('route')直接跳到下一个router，而不是下一个路由处理器

仔细观察，有两个router.get('/user/:id')，一般情况下后一个无效了，但是前一个的处理器有next('route')语句直接将控制权转交给下一个router。

这也侧面反映了路由控制器函数名不能叫router

* 外置路由风格：

route.js

var router=express.Router();

router.get('/', (request, response)=>{

response.end('123');

})

router.post('/about', (request, response)=>{

response.end('321');

})

module.exports = router;

express.Router().get() post() use() 和app.get() post() use()看样子差不多

router.use(function timeLog (req, res, next) {

console.log('Time: ', Date.now())

next()

})

同样可以使用中间件

http.js

var router=require(./router);

app.use(router);

导入之后需要用pp.use()方法使用路由

当然这里边的路由和Angular的路由不同，不负责页面的跳转，只是监听不同的方法和url，来选择不同的回调函数处理。

app.use('/birds', birds)

响应/birds和/birds/about

Express.Router()也能使用链式操作

router

.get('/', (requst, response) => {})

.get('/nihao', (requst, response) => {})

听说这种写法很优雅

* Application-level middleware

应用级中间件，比如myLogger

* Router-level middleware

路由器级别的中间件，使用next('router')切换

* Error-handling middleware
* Built-in middleware
* Third-party middleware

**错误处理**

app.get("/", function (req, res) {

throw new Error("BROKEN"); // Express will catch this on its own.

});

直接抛出。CMD显示

Error: BROKEN

at E:\Codes\Node.js\temp1\index.js:58:9

app.get("/", function (req, res, next) {

fs.readFile("/file-does-not-exist", function (err, data) {

if (err) {

next(err); // Pass errors to Express.

}

else {

res.send(data);

}

});

});

将error传给express中间件。这么传怎么用？？？？？？？？？？

**Express 应用生成器**

Express 应用生成器，相当于angular-cli，可以生成一个 Express 应用的“框架”。

npm install express-generator -g

创建一个Express 应用：

express helloworld --view=pug

直接

express

会把当前目录变为一个express应用

安装所有依赖：

$ cd helloworld

$ npm install

然后运行这个应用（Windows 环境）：

SET DEBUG=helloworld:\* & npm start

（Linux/macOS 环境）：

DEBUG=helloworld:\* npm start

DEBUG 命令可以在控制台展示应用运行时有用的日志信息

因为package.json文件中script属性有

"start": "node ./bin/www"

所以npm start其实执行的是 node ./bin/www

只有重启服务器才能看到 Express 网站所做的改动。每次改动后手动启停服务器实在太烦人了，有必要花点时间让这项工作自动化。

$ sudo npm install -g nodemon

这里还可以把它作为开发依赖将安装在本地，于是使用这个项目的开发人员只要安装这个应用就能自动获得。通过以下命令将其安装在骨架项目的根目录：

$ npm install --save-dev nodemon

项目的 package.json 文件将自动添加一个新的属性：

"devDependencies": {

"nodemon": "^1.18.9"

}

如果没有全局安装该工具，就无法从命令行启动它（除非我们将其添加到路径中），但是可以在 NPM 脚本中调用它，因为 NPM 掌握所有已安装包的信息。找到 package.json 的 scripts 部分。在 "start" 一行的末尾添加逗号，并在新的一行中添加 "devstart"，如下所示：

"scripts": {

"start": "node ./bin/www",

"devstart": "nodemon ./bin/www"

},

现在可以用新建的 devstart 命令启动服务器：

$ DEBUG=express-locallibrary-tutorial:\* npm run devstart

art-template还有一个express版本。更加简单。

利用Express托管静态文件

express.static(root, [options])

app.use(express.static('public'))

创建一个public文件夹，然后就可以访问public文件夹下的所有文件。不知道这样做的意义何在

enctype="multipart/form-data"

如果想上传文件，表单from标签中必须有这个东西

input type="file"

application/x-www-from-urlencoded

post表单中提交的数据需要经过urlencoded编码之后传输

form-data

express里边是send不用end

# mysql in node.js

npm install mysql

安装mysql的驱动

没用，别安了。8.0以后版本不支持这个了，得用mysql自带的

## 建立链接

const mysqlx = require('@mysql/xdevapi');

mysqlx

.getSession('mysqlx://root:123456@localhost:33060/project')

.then(session => {

console.log(session.inspect());

})

123456是密码

33060是这个驱动的端口号，固定值

project是数据库名，可选。区别就是session的\_properties属性的schema属性是数据库名还是undefined

通常用session.inspect()方法检查是否成功建立连接

* 用配置选项建立链接

const config = {

password: 'passwd',

user: 'root',

host: 'localhost',

port: 33060,

schema: 'project'

};

mysqlx.getSession(config)

The API provides support for managing database sessions and schemas, working with document-store collections and using raw SQL statements.

The following is an example encompassing the different sort of CRUD operations using the document-store:

.then(() => {

return session.getSchema('mySchema');

})

schema应该就是数据库，即create database那个database。collection相当于表？

如果想要链式调用必须在then的参数里返回一个promise。getSchema()、createCollection()等等都返回一个Promise对象。但是这个promise对象是不是都是同一个session对象？不确定，所以为了保证整个过程都同一个session，链式操作写在第一个then方法参数的return里，而且以后所有的then的参数的参数都为空，形如

mysqlx.getSession('mysqlx://root:123456@localhost:33060/project').then(

session => {

return session.getSchema('mySchema')

.then(() => {

return session.getSchema('mySchema').createCollection('myCollection');

})

.then(() => {

return session.getSchema('mySchema').getCollection('myCollection')

.add([{ name: 'foo', age: 42 }])

.execute()

})

.then(() => {

return session.close();

})

.catch(err => {

return session.close()

.then(() => {

throw err;

})

.catch(err => {

throw err;

});

});

}

)

.catch( err => console.log(err))

如上，得到第一个session之后，这个session要长期使用，所以再也没有平行的then，只有后面一个catch。

进入第一个then的参数，里面是紧跟着的链式操作。注意这里边的then回调函数都没有参数，因为要使用那个全局的session，不敢保证createCollection、getCollection等方法返回的还是同一个session。所以这些操作都要写在return里，以返回一个promise，进行链式的异步操作，但是下一个then的参数不能有参数。

从上述代码可以看出session.close()，.execute()肯定返回一个promise。getTable、getSession都不返回promise

## 建立数据库

session.createSchema('mySchema')

获得数据库

session.getSchema('mySchema')

## 建表

session.sql('CREATE TABLE myTable (\_id SERIAL, age TINYINT)').execute();

mysqlx.getSession('mysqlx://localhost:33060')

.then(session => {

return session.sql('CREATE TABLE schemaName.tableName (column INT)')

.execute()

.then(() => {

return session.getSchema('schemaName').getTable('tableName');

});

})

.then(table => {

// work with the Table object

})

插入数据

session.getSchema('mySchema').getTable('myTable')

.insert(['name', 'age'])

.values(['foo', 42])

.execute();

查询

return session.getSchema('mySchema').getTable('table1')

.select(['name', 'id'])

.execute(row => {

console.log(row);

})

return session.getSchema('mySchema').getTable('myTable')

.select(['name', 'age'])

.where('name = :value')

.bind('value', 'bar')

.execute(row => {

console.log(row);

});

链接查询

return session.sql('SELECT tasks.fileId, tasks.receiverId, tasks.senderId, files.fenhao FROM tasks INNER JOIN files on tasks.fileId=files.id;')

.execute(row => {

console.log(row)

queryResult.push(row);

})

.then(() => {

res.send({nihao:1})

})

.catch(err => console.log(err))

也就是说用session.sql也可以得到结果。select只是封装了，而且封装的不好，没有连接查询的方法

排序

const table = session.getSchema('testSchema').getTable('testTable');

return table.select().orderBy('age ASC').execute();

修改

return session.getSchema('mySchema').getTable('myTable')

.update()

.where('age = 42')

.set('name', 'bar')

.execute();

session.close();

关闭会话

return session.getSchema('mySchema').createCollection('myCollection');

})

.then(() => {

return session.getSchema('mySchema').getCollection('myCollection')

.add([{ name: 'foo', age: 42 }])

.execute()

})

.then(() => {

return session.getSchema('mySchema').getCollection('myCollection')

.find()

.fields(['name', 'age'])

.execute(row => {

console.log(row); // { name: 'foo', age: 42 }

});

})

.then(() => {

return session.getSchema('mySchema').getCollection('myCollection')

.modify('age = :value')

.bind('value', 42);

.set('name', 'bar')

.execute();

})

.then(() => {

return session.getSchema('mySchema').getCollection('myCollection')

.find()

.fields(['name', 'age'])

.execute(row => {

console.log(row); // { name: 'bar', age: 42 }

});

const table = session.getSchema('testSchema').getTable('testTable');

return table.update().where('name = "bar"').set('age', 50)

.execute()

.then(() => {

return table.select().orderBy('name ASC').execute();

});