愿景："让编程不在难学，让技术与生活更加有趣"

第一章 初识NodeJs

第1集 node.js课程介绍及案例演示

简介：介绍nodejs课程大纲及实战项目演示

第2集 nodejs环境安装配置

简介：讲解windows和mac系统下nodejs环境安装配置

更换淘宝镜像：

```
export NVM_NODEJS_ORG_MIRROR=https://npm.taobao.org/mirrors/node
```

nvm 的bash_profile文件里写入以下配置：

```
export NVM_DIR="$([ -z "${XDG_CONFIG_HOME-}" ] && printf %s "${HOME}/.nvm" ||
printf %s "${XDG_CONFIG_HOME}/nvm")"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh" # This loads nvm
```

第3集 vscode编辑器和插件安装

简介：vscode安装步骤及常用插件安装

- vscode官方下载地址：https://code.visualstudio.com/
- 安装后在插件库里搜索Chinese选择Chinese (Simplified) Language Pack for Visual Studio Code

汉化软件

第4集 初建NodeJs应用及调试

简介：创建一个nodejs应用以及如何去调试nodejs应用

- nodejs和JavaScript有什么区别?

  nodejs是一个JavaScript运行环境（平台），JavaScript是编程语言。

- nodejs中无法运行alert方法，DOM和BOM这类方法也无法在node中运行。

第5集 深入理解commonjs模块规范

简介：讲解使用require和module引入、导出模块

- commonjs规范

  每一个文件相当于一个模块，有自己的作用域，其模块里的变量、函数以及类都是私有的，对外不可见的。

- module.exports模块导出

```
function add(a,b){
  console.log(a+b)
}

function decrease(a,b){
  console.log(a-b)
}

module.exports = {
  add,
  decrease
}
```

- require模块引用

```
let cal = require('./calculate')

cal.add(10,5)
cal.decrease(100,50)
```

- loadsh

```
它是一个一致性、模块化、高性能的 JavaScript 实用工具库
```

初始化项目

```
npm init -y
```

安装命令

```
npm i loadsh --save/cnpm i loadsh --save
```

下载速度慢可替换成淘宝镜像

```
npm install -g cnpm --registry=https://registry.npm.taobao.org
```

- nodejs中的全局对象是global，定义全局变量用global对象来定义

```
global.a = 2  //定义全局变量，可在其他模块中直接使用
```

愿景："让编程不在难学，让技术与生活更加有趣"

第二章 NodeJs核心模块api-基础

第1集 Buffer缓冲器常用api （一）

简介：介绍buffer缓冲器类常用的api使用

- buffer用于处理二进制数据，在v8堆外分配物理内存，buffer实例类似0-255之间的整数数组，显示的数据为十六进制，大小是固定的，无法修改。
- 创建buffer
  - **Buffer.alloc(size[, fill[, encoding]])**：
    - `size` 新 `Buffer` 的所需长度。
    - `fill` | | | 用于预填充新 `Buffer` 的值。**默认值:** `0` 。
  - `encoding` 如果 `fill` 是一个字符串，则这是它的字符编码。**默认值:** `'utf8'` 。

```
console.log(Buffer.alloc(10));//创建一个长度为10的Buffer，默认填充0
console.log(Buffer.alloc(10,2));//创建一个长度为10的Buffer，填充2
console.log(Buffer.alloc(10,100));//创建一个长度为10的Buffer，填充1000
console.log(Buffer.alloc(10,-1));//创建一个长度为10的Buffer，-1强制转换成
255
```

  - **Buffer.allocUnsafe(size)**：
    - `size` 新 `Buffer` 的所需长度。

```
console.log(Buffer.allocUnsafe(10));//创建一个长度为10未初始化的buffer
```

  - **Buffer.from(array)**：
    - `array` 整数数组

```
console.log(Buffer.from([1,2,3]));//创建buffer，填充[1,2,3]
```

  - **Buffer.from(string[, encoding])**：
    - `string` 要编码的字符串。
    - `encoding` `string` 的字符编码。**默认值:** `'utf8'` 。

```
console.log(Buffer.from('eric')); //创建buffer，填充字符串

console.log(Buffer.from('eric','base64')); //创建buffer，填充bese64编码的字符串
```

- Buffer类上常用的属性、方法
  - **Buffer.byteLength:** 返回字符串的字节长度

```
console.log(Buffer.byteLength('eric'));//返回字符串的字节长度，4

console.log(Buffer.byteLength('中文'));//6，一个文字代表3个字节
```

  - **Buffer.isBuffer(obj):** 判断是否是buffer

```
console.log(Buffer.isBuffer({}));

console.log(Buffer.isBuffer(Buffer.from('eric')));
```

  - **Buffer.concat(list[, totalLength])**： 合并buffer

```
const buf = Buffer.from('hello');
const buf2 = Buffer.from('eric');

console.log(buf);
console.log(buf2);
console.log(Buffer.concat([buf,buf2]));
console.log(Buffer.concat([buf,buf2],10));
```

官网文档地址：http://nodejs.cn/api/buffer.html

第2集 Buffer缓冲器常用api （二）

简介：介绍buffer缓冲器实例常用的api使用

- buffer实例常用的属性、方法
  - **buf.write(string[, offset[, length]][, encoding])** 将字符写入buffer,返回已经写入的字节数
    - `string` 要写入 `buf` 的字符串。
    - `offset` 从指定索引下写入。**默认值:** `0` 。

- `length` 要写入的字节数。**默认值:** `buf.length - offset`。
- `encoding` 字符串的字符编码。**默认值:** `'utf8'`。

```
const buf = Buffer.allocUnsafe(20);
console.log(buf);

// console.log(buf.write('buffer'));
console.log(buf.write('buffer',5,3));
console.log(buf);
```

- **buf.fill(value[, offset[, end]][, encoding])** 填充buffer
  - `value` | | | 用来填充 `buf` 的值。
  - `offset` 开始填充 `buf` 的索引。**默认值:** `0`。
  - `end` 结束填充 `buf` 的索引（不包含）。**默认值:** `buf.length`。
  - `encoding` 如果 `value` 是字符串，则指定 `value` 的字符编码。**默认值:** `'utf8'`。

```
console.log(buf.fill('eric',5,10));
```

- **buf.length** buffer的长度
- **buf.toString([encoding[, start[, end]]])** 将buffer解码成字符串形式
  - `encoding` 使用的字符编码。**默认值:** `'utf8'`。
  - `start` 开始解码的字节索引。**默认值:** `0`。
  - `end` 结束解码的字节索引（不包含）。**默认值:** `buf.length`。

```
const buf = Buffer.from('test');
console.log(buf.toString('utf8',1,3));
```

- **buf.toJSON** 返回 buffer 的 JSON 格式

```
const buf = Buffer.from('test');
console.log(buf.toJSON());
```

- **buf.equals（otherBuffer）** 对比其它buffer是否具有完全相同的字节
  - `otherBuffer` 要对比的buffer

```
const buf1 = Buffer.from('ABC');
const buf2 = Buffer.from('414243', 'hex');
const buf3 = Buffer.from('ABCD');

console.log(buf1);
console.log(buf2);

console.log(buf1.equals(buf2));
// 打印: true
console.log(buf1.equals(buf3));
```

- **buf.indexOf/lastIndexOf** 查找指定的值对应的索引
- **buf.slice([start[, end]])** 切割buffer
  - `start` 新 `Buffer` 开始的位置。**默认值:** `0`。
  - `end` 新 `Buffer` 结束的位置（不包含）。**默认值:** `buf.length`。

```
const buf = Buffer.from('abcdefghi');
console.log(buf.slice(2,7).toString());//cdefg
```

- **buf.copy(target[, targetStart[, sourceStart[, sourceEnd]]])** 拷贝buffer
  - `target` | 要拷贝进的 `Buffer` 或 `Uint8Array`。
  - `targetStart` 目标`buffer`中开始写入之前要跳过的字节数。**默认值:** `0`。
  - `sourceStart` 来源`buffer` 中开始拷贝的索引。**默认值:** `0`。
  - `sourceEnd` 来源`buffer` 中结束拷贝的索引（不包含）。**默认值:** `buf.length`。

```
const buf = Buffer.from('abcdefghi');
const buf2 = Buffer.from('test');
// console.log(buf.slice(2,7).toString());

// console.log(buf.copy(buf2));
// console.log(buf2.toString());
console.log(buf2.copy(buf,2,1,3));
console.log(buf.toString());
```

第3集 node.js文件系统模块常用api操作（一）

简介：讲解一些文件的常用api操作

- 引入文件系统模块fs

```
const fs = require('fs')
```

- **fs.readFile(path[, options], callback)** 读取文件
  - `path` | | 文件路径
  - `callback` 回调函数
    - `err`
    - `data` | 读取的数据

```
fs.readFile('./hello.txt','utf8',(err,data) => {
  if(err) throw err;
  console.log(data);
})
```

- **fs.writeFile(file, data[, options], callback)** 写入文件
  - `file` | | | 文件名或文件描述符。
  - `data` | | | 写入的数据
  - `options` |
    - `encoding` | 写入字符串的编码格式 **默认值:** `'utf8'`。
    - `mode` 文件模式(权限) **默认值:** `0o666`。
    - `flag` 参阅支持的文件系统标志。**默认值:** `'w'`。
  - `callback` 回调函数
    - `err`

```
fs.writeFile('./hello.txt','this is a test',err => {
  if(err) throw err;
  console.log('写入成功');
})
```

- **fs.appendFile(path, data[, options], callback)** 追加数据到文件
  - `path` | | | 文件名或文件描述符。
  - `data` | 追加的数据
  - `options` |
    - `encoding` | 写入字符串的编码格式 **默认值:** `'utf8'`。
    - `mode` 文件模式(权限) **默认值:** `0o666`。
    - `flag` 参阅支持的文件系统标志。**默认值:** `'a'`。
  - `callback` 回调函数
    - `err`

```
const buf = Buffer.from('hello world!')
fs.appendFile('./hello.txt',buf,(err) => {
  if(err) throw err;
  console.log('追加成功');
})
```

- **fs.stat(path[, options], callback)** 获取文件信息，判断文件状态（是文件还是文件夹）
  - `path` | |
  - `options`
    - `bigint` 返回的 `fs.Stats` 对象中的数值是否应为 `bigint` 型。**默认值:** `false`。
  - `callback`
    - `err`
    - `stats` <fs.Stats> 文件信息

```
fs.stat('./hello.txtt',(err,stats) => {
  if(err){
    console.log('文件不存在');
    return;
  }
  console.log(stats);
  console.log(stats.isFile());
  console.log(stats.isDirectory());
})
```

- **fs.rename(oldPath, newPath, callback)** 重命名文件
  - `oldPath` | | 旧文件路径名字
  - `newPath` | | 新文件路径名字
  - `callback` 回调函数
    - `err`

```
fs.rename('./hello.txt','./test.txt',err => {
  if(err) throw err;
  console.log('重命名成功');
})
```

- **fs.unlink(path, callback)** 删除文件
  - `path` | |
  - `callback`
    - `err`

```
fs.unlink('./test.txt',err => {
  if(err) throw err;
  console.log('删除成功');
})
```

官方文档：http://nodejs.cn/api/fs.html#fs_file_system_flags

第4集 node.js文件系统模块常用api操作（二）

简介：讲解如何使用文件系统操作文件夹

- fs.mkdir(path[, options], callback) 创建文件夹
  - `path` | |
  - `options` |
    - `recursive` 是否递归创建 **默认值:** `false`。
    - `mode` 文件模式（权限）Windows 上不支持。**默认值:** `0o777`。
  - `callback`
    - `err`

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

// fs.mkdir('./a',err => {
//   if(err) throw err;
//   console.log('创建文件夹成功');
// })

fs.mkdir('./b/c',{
  recursive:true
},err => {
  if(err) throw err;
  console.log('创建文件夹成功');
})
```

- fs.readdir (path[, options], callback) 读取文件夹
  - `path` | |
  - `options` |
    - `encoding` **默认值:** `'utf8'`。

        - `withFileTypes` **默认值:** `false`。
    - `callback`
        - `err`
        - `files` <string[]> | <buffer[]> | <fs.Dirent[]>

```
fs.readdir('./',{
  encoding:'buffer',  //设置buffer，files返回文件名为buffer对象
  withFileTypes:true //单上文件类型
},(err,files) => {
  if(err) throw err;
  console.log(files)
})
```

- fs.rmdir(path[, options], callback) 删除文件夹
    - `path` | |
    - `options`
        - `maxRetries` 重试次数。出现这类错误`EBUSY`、`EMFILE`、`ENFILE`、`ENOTEMPTY`或者EPERM，每一个重试会根据设置的重试间隔重试操作。如果`recursive`不为true则忽略. **默认值:** `0`。
        - `retryDelay` 重试的间隔，如果`recursive`不为true则忽略. **默认值:** `100`。
        - `recursive` 如果为 `true`，则执行递归的目录删除。在递归模式中，如果 `path` 不存在则不报告错误，并且在失败时重试操作。**默认值:** `false`。
    - `callback`
        - `err`

```
fs.rmdir('./b',{
  recursive:true
},err => {
  if(err) throw err;
  console.log('删除文件夹成功');
})
```

    - 错误代码意义：https://blog.csdn.net/a8039974/article/details/25830705
- 监听文件变化 chokidar
    - 安装chokidar

```
npm install chokidar --save-dev
```

    - Chokidar.watch(path,[options])

```
chokidar.watch('./',{
  ignored:'./node_modules'
}).on('all',(event,path) => {
  console.log(event,path)
})
```

第5集 核心知识之文件流讲解

简介：讲解如何创建读取文件流和创建写入文件流

- Node.js 中有四种基本的流类型：
  - `Writable` - 可写入数据的流（例如 `fs.createWriteStream()`）。
  - `Readable` - 可读取数据的流（例如 `fs.createReadStream()`）。
  - `Duplex` - 可读又可写的流（例如 `net.Socket`）。
  - `Transform` - 在读写过程中可以修改或转换数据的 `Duplex` 流（例如 `zlib.createDeflate()`）。
- 创建读取文件流fs.createReadStream(path[, options])

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


let rs = fs.createReadStream('./streamTest.js',{
  highWaterMark:100  //每次on data的一个数据量
});

let count = 1;
rs.on('data',chunk => {
  console.log(chunk.toString())
  console.log(count++)
})

rs.on('end',()=>{
  console.log('读取完成')
});
```

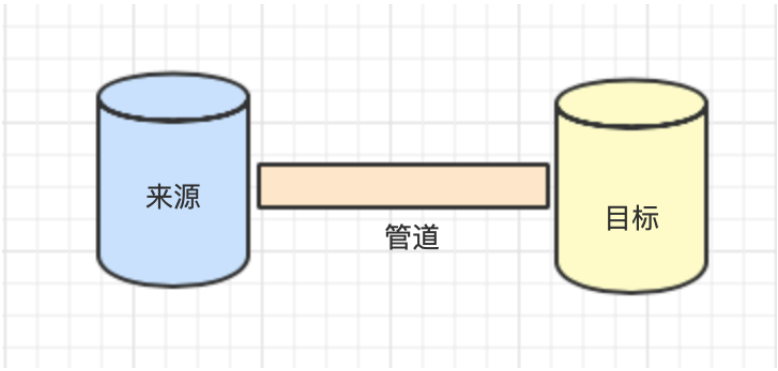
- 创建写入文件流fs.createWriteStream(path[, options])

```
let ws = fs.createWriteStream('./a.txt');
```

```
let num = 1;
let timer = setInterval(() => {
  if(num < 10){
    ws.write(num + '')
    num++
  }else{
    ws.end("写入完成");
    clearInterval(timer)
  }
}, 200);


ws.on('finish',()=>{
  console.log('写入完成')
})
```

- 管道流



从数据流来源中一段一段通过管道流向目标。

- readable.pipe(destination[, options])

```
let rs = fs.createReadStream('./streamTest.js');
let ws = fs.createWriteStream('./a.txt');

rs.pipe(ws)
```

第6集 基础模块path常用api

简介：讲解path模块常用的一些api

- **path.basename(path[,ext])** 返回path的最后一部分

- **path.dirname(path)** 返回path的目录名
- **path.extname(path)** 返回path的扩展名
- **path.join([...paths])** 路径拼接
- **path.normalize(path)** 规范化路径
- **path.resolve([...paths])** 将路径解析为绝对路径
- **path.format(pathObject)** 从对象中返回路径字符串
- **path.parse(path)** 返回一个对象，包含path的属性
- **path.sep** 返回系统特定的路径片段分隔符
- **path.win32** 可以实现访问windows的path方法
- **__filename** 表示当前正在执行的脚本的文件名
- **__dirname** 表示当前执行脚本所在的目录

```
const {basename,dirname,extname,join,normalize,resolve,format,parse,sep,win32}
= require('path');

// console.log(basename('/nodejs/2-6/index.js','.js'))
// console.log(dirname('/nodejs/2-6/index.js'))
// console.log(extname('index.'))
// console.log(join('/nodejs/','/index.js'))
// console.log(normalize('/nodejs/test/../index.js'))
// console.log(resolve('./pathTest.js'))
// let pathObj = parse('/nodejs/test/index.js');
// console.log(pathObj)

// console.log(format(pathObj))
// console.log("当前系统下分隔符   " + sep)
// console.log("windows下分隔符   " + win32.sep)

console.log("filename   " + __filename)
// console.log(__dirname)
console.log("resolve   " + resolve('./pathTest.js'))
```

第7集 深度讲解node.js事件触发器

简介：讲解事件触发器events的使用方法

- **eventEmitter.on(eventName, listener)**注册监听器
  - `eventName` | 事件名称
  - `listener` 回调函数。
- **eventEmitter.emit(eventName[, ...args])** 触发事件
  - `eventName` | 事件名称

    - `...args` 参数
- **eventEmitter.once(eventName, listener)** 绑定的事件只能触发一次
- **emitter.removeListener(eventName,listener)** 从名为 `eventName` 的事件的监听器数组中移除指定的 `listener`。
- **emitter.removeAllListener([eventName])** 移除全部监听器或指定的 `eventName` 事件的监听器

```
const EventEmitter = require('events');

class MyEmitter extends EventEmitter{}

let myEmitter = new MyEmitter();

function fn1(a,b){
    console.log('触发了事件，带参 ',a+b)
}

function fn2(){
  console.log("触发了事件，不带参")
}

myEmitter.on('hi',fn1)

myEmitter.on('hi',fn2)

myEmitter.once('hello',()=>{
  console.log('触发了hello事件')
})

myEmitter.emit('hi',1,8);
// myEmitter.emit('hello')
// myEmitter.emit('hello')

// myEmitter.removeListener('hi',fn1)
myEmitter.removeAllListeners('hi');
myEmitter.emit('hi',1,8);
```

第8集 核心模块util常用工具

简介：讲解util模块里常用的工具

- **util.callbackify(original)** 将 `async` 异步函数（或者一个返回值为 `Promise` 的函数）转换成遵循异常优先的回调风格的函数

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

async function hello(){
  return 'hello world'
}

let helloCb = util.callbackify(hello);

helloCb((err,res) => {
  if(err) throw err;
  console.log(res)
})
```

- **util.promisify(original)** 转换成 promise 版本的函数

```
let stat = util.promisify(fs.stat)

// stat('./utilTest.js').then((data) => {
//   console.log(data)
// }).catch((err) => {
//   console.log(err)
// })

async function statFn () {
  try {
    let stats = await stat('./utilTest.js');
    console.log(stats)
  } catch (e) {
    console.log(e)
  }
}

statFn();
```

- **util.types.isDate(value)** 判断是否为date数据

```
console.log(util.types.isDate(new Date()))
```

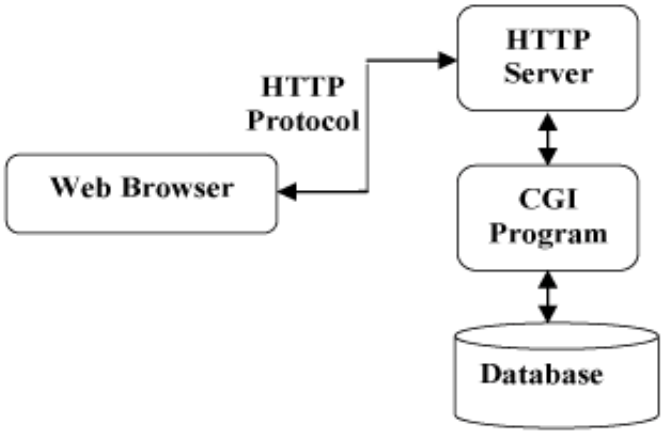
小滴课堂 愿景："让编程不在难学，让技术与生活更加有趣"

第三章 http全面解析

第1集 http的发展历史

简介：讲解什么是http

- http是什么？ http://www.xxx.com
  - http协议（HyperText Transfer Protocol，超文本传输协议）是一种应用广泛的网络传输协议。
  - http是一个基于TCP/IP通讯协议来传递数据（HTML文件，图片文件，查询结果等）。
- http工作原理
  - http协议工作在客户端-服务端之间
  - 主流的三个web服务器：Apache、 Nginx 、IIS。
  - http默认端口为80
  - http协议通信流程



- 输入url发生了什么?
  - DNS解析
  - TCP连接
  - 发送http请求
  - 服务器处理请求
  - 浏览器解析渲染页面
  - 连接结束

第2集 走进http之请求方法和响应头信息

简介：讲解http的请求方法和响应头信息

- http请求方法
  - GET 请求指定的页面信息，并返回实体主体
  - HEAD 类似于get请求，只不过返回的响应中没有具体的内容，用于获取报头
  - POST 向指定资源提交数据进行处理请求。数据被包含在请求体中。
  - PUT 从客户端向服务器传送的数据取代指定的文档的内容
  - DELETE 请求服务器删除指定的页面
  - CONNECT HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。
  - OPTIONS 允许客户端查看服务器的性能
  - TRACE 回显服务器收到的请求，主要用于测试或诊断
- HTTP响应头信息

应答头	说明
Allow	服务器支持哪些请求方法（如get、post等）
Content-Encoding	文档的编码方法。只有在解码之后才可以得到Content-Type头指定的内容类型。利用gzip压缩能减少HTML文档的下载时间。
Content-Length	表示内容长度。只有当浏览器使用持久http连接时才需要这个数据。
Content-Type	表示文档属于什么MIME类型。
Date	当前的GMT时间。
Expires	资源什么时候过期，不再缓存
Last-Modified	文档最后改动时间。
Location	重定向的地址
Server	服务器的名字
Set-Cookie	设置和页面关联的Cookie
WWW-Authenticate	定义了使用何种验证方式去获取对资源的链接

第3集 走进http之状态码和content-type

简介：讲解http的状态码和content-type

- 常见的http状态码
  - 200 请求成功
  - 301 资源被永久转移到其他URL
  - 404 请求的资源（网页等）不存在
  - 500 内部服务器错误
- http状态码分为5类：

分类	分类描述
1**	信息，服务器收到请求，需要请求者继续执行操作
2**	成功，操作被成功接收并处理
3**	重定向，需要进一步的操作以完成请求
4**	客户端错误，请求包含语法错误或无法完成请求
5**	服务器错误，服务器在处理请求的过程中发生了错误

- **Content-Type** 内容类型
  - 常见的媒体格式类型如下
    - text/html:HTML格式
    - text/plain:纯文本格式
    - text/xml:XML格式
    - image/gif:gif图片格式
    - image/jpeg:jpg图片格式
    - image/png:png图片格式
    - multipart/form-data:需要在表单中进行文件上传时，就需要使用该格式
  - 以application开头的媒体格式类型：
    - application/xhtml+xml:XHTML格式
    - application/xml:XML数据格式
    - application/atom+xml:Atom XML聚合格式
    - application/json:JSON数据格式
    - application/pdf:pdf格式
    - application/msword:Word文档格式
    - application/octet-stream:二进制流数据（常见的文件下载)
    - application/x-www-form-urlencoded:表单中默认的encType,表单数据被编码为key/value格式发送到服务器

第4集 搭建自己的第一个http服务器

简介：讲解如何使用nodejs中的http模块搭建服务器

- 引入http模块

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

- 创建http服务器

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

const server = http.createServer((req,res)=>{
  res.writeHead(200,{'content-type':'text/html'});
  res.end('<h1>hello world</h1>');
})

server.listen(3000,()=>{
  console.log('监听了3000端口')
})
```

第5集 实战案例之nodejs简易爬虫

简介：讲解如何使用http模块做一个简单的爬虫

- 简单爬虫实现

```
const https = require('https');
const fs = require('fs');

https.get('https://xdclass.net/#/index',(res)=>{
  res.setEncoding('utf8');
  let html = '';
  res.on('data',chunk => {
    html += chunk;
  })
  res.on('end',()=>{
    console.log(html)
    fs.writeFile('./index.txt',html,(err) => {
```

```
      if(err) throw err;
      console.log('写入成功')
    })
  })
})
```

- cheerio实现dom操作
  - 安装cheerio

    ```
    npm install cheerio --save-dev
    ```

  - 引入cheerio

    ```
    const cheerio = require('cheerio')
    ```

  - 通过title元素获取其内容文本

    ```
    const $ = cheerio.load(html);//把html代码加载进去，就可以实现jq的dom操作
    console.log($('title').text());
    ```

愿景："让编程不在难学，让技术与生活更加有趣"

第四章 NodeJs核心模块api-路由与接口

第1集 如何处理客户端get/post请求

简介：讲解如何处理从页面中传来的参数

- url.parse(urlString[, parseQueryString[, slashesDenoteHost]])
  - urlString url字符串
  - parseQueryString 是否解析
  - slashesDenoteHost
    - -默认为false，//foo/bar 形式的字符串将被解释成 { pathname: '//foo/bar' }
    - -如果设置成true，//foo/bar 形式的字符串将被解释成 { host: 'foo', pathname: '/bar' }

```
console.log(url.parse('https://api.xdclass.net/pub/api/v1/web/product/find_list_by_type?type=2',true,true))
```

- get请求用于客户端向服务端获取数据，post请求是客户端传递数据给服务端
- 处理get请求

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

//
console.log(url.parse('https://api.xdclass.net/pub/api/v1/web/product/find_list_by_type?type=2',true,true))

const server = http.createServer((req,res) => {
  let urlObj = url.parse(req.url,true);
  res.end(JSON.stringify(urlObj.query))
})

server.listen(3000,()=>{
  console.log('监听3000端口')
})
```

- 处理post请求

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

//
console.log(url.parse('https://api.xdclass.net/pub/api/v1/web/product/find_list_by_type?type=2',true,true))

const server = http.createServer((req,res) => {
  let postData = '';
  req.on('data',chunk => {
    postData += chunk;
  })
  req.on('end',()=>{
    console.log(postData)
  })
  res.end(JSON.stringify({
    data:'请求成功',
    code:0
  }))
})

server.listen(3000,()=>{
  console.log('监听3000端口')
```

```
})
```

- 整合get/post请求

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

//
console.log(url.parse('https://api.xdclass.net/pub/api/v1/web/product/find_list_by_type?type=2',true,true))

const server = http.createServer((req, res) => {
  if (req.method === 'GET') {
    let urlObj = url.parse(req.url, true);
    res.end(JSON.stringify(urlObj.query))
  } else if (req.method === 'POST') {
    let postData = '';
    req.on('data', chunk => {
      postData += chunk;
    })
    req.on('end', () => {
      console.log(postData)
    })
    res.end(JSON.stringify({
      data: '请求成功',
      code: 0
    }))
  }

})

server.listen(3000, () => {
  console.log('监听3000端口')
})
```

第2集 nodemon自动重启工具安装配置

简介：讲解nodemon安装以及配置

- nodemon安装

```
npm install -g  nodemon
```

- 替换淘宝镜像

```
$ npm install -g cnpm --registry=https://registry.npm.taobao.org
```

第3集 讲解初始化路由及接口开发

简介：讲解如何开发一个接口以及路由编写

- 通过pathname判断请求地址

```
//server.js    服务器文件
const http = require('http');
const routerModal = require('./router/index')


const server = http.createServer((req,res)=>{
  res.writeHead(200,{'content-type':'application/json;charset=UTF-8'})
    let resultData = routerModal(req,res);
    if(resultData){
      res.end(JSON.stringify(resultData))
    }else{
      res.writeHead(404,{'content-type':'text/html'})
      res.end('404 not found')
    }
})

server.listen(3000,() => {
  console.log('监听3000端口')
})
```

```
//router/index.js   路由文件
const url = require('url')
function handleRequest(req,res) {
  let urlObj = url.parse(req.url,true);
  console.log(urlObj)
  if(urlObj.pathname === '/api/getMsg'&&req.method === 'GET'){
    return {
      msg:'获取成功'
    }
  }
  if(urlObj.pathname === '/api/updateData'&&req.method === 'POST'){
    return {
      msg:'更新成功'
```

```
    }
  }
}

module.exports = handleRequest
```

第4集 案例实战用户列表增删改查

简介：讲解用户列表增删改查接口开发

- 使用promise处理post请求

```
const getPostData = (req) => {
  return new Promise((resolve, reject) => {
    if (req.method !== 'POST') {
      resolve({})
      return
    }
    let postData = '';
    req.on('data', chunk => {
      postData += chunk;
    })
    req.on('end', () => {
      console.log(postData)
      resolve(JSON.parse(postData))
    })
  })
}


const server = http.createServer((req, res) => {
  res.writeHead(200, { 'content-type': 'application/json;charset=UTF-8' })
  //获取post请求数据
  getPostData(req).then((data) => {
    req.body = data
    let resultData = routerModal(req, res);
    if (resultData) {
      res.end(JSON.stringify(resultData))
    } else {
      res.writeHead(404, { 'content-type': 'text/html' })
      res.end('404 not found')
    }
  })
```

```
})
```

- 处理数据的文件controller/user.js

```
module.exports = {
  getUserList(){
    return [
      {
        id:1,
        name:'eric',
        city:'北京'
      },
      {
        id:2,
        name:'xiaoming',
        city:'广州'
      },
      {
        id:3,
        name:'小红',
        city:'上海'
      }
    ]
  },
  addUser(userObj){
    console.log(userObj);
    return {
      code:0,
      msg:'新增成功',
      data:null
    }
  },
  delectUser(id){
    console.log(id)
    return {
      code:0,
      msg:'删除成功',
      data:null
    }
  },
  updateUser(id,userObj){
    console.log(id,userObj);
    return {
      code:0,
      msg:'更新成功',
      data:null
```

```
    }
  }
}
```

- 路由编写文件

```
const url = require('url')
const {getUserList,addUser,delectUser,updateUser} =
require('../controller/user')
function handleRequest(req,res) {
  let urlObj = url.parse(req.url,true);
  console.log(urlObj)
  if(urlObj.pathname === '/api/getUserList'&&req.method === 'GET'){
    let resultData = getUserList()
    console.log(resultData)
    return resultData;
  }
  if(urlObj.pathname === '/api/addUser'&&req.method === 'POST'){
    let resultData = addUser(req.body);
    return resultData;
  }
  if(urlObj.pathname === '/api/delectUser'&&req.method === 'POST'){
    let resultData = delectUser(urlObj.query.id);
    return resultData;
  }
  if(urlObj.pathname === '/api/updateUser'&&req.method === 'POST'){
    let resultData = updateUser(urlObj.query.id,req.body);
    return resultData;
  }
}

module.exports = handleRequest
```

第5集 教你轻松解决接口跨域问题

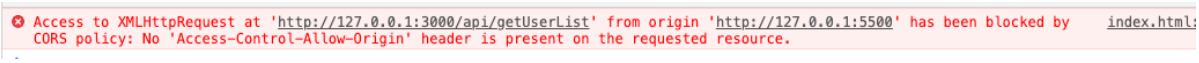
简介：讲解如何利用cors解决跨域问题

- **什么是跨域?**

```
浏览器同源策略：协议+域名+端口三者相同就是同源。
http://www.baidu.com/a.js       http://www.baidu.com/b.js
https://www.baidu.com/a.js      http://www.baidu.com/a.js    协议不同
https://www.baidu.com:8080/a.js      https://www.baidu.com/a.js   端口不同
https://www.baidu.com:8080/a.js      https://www.a.com:8080/a.js   域名不同


跨域：协议、域名、端口三者任意一个不同就是跨域。
```

- 前端请求跨域提示



- 跨域的解决方法cors方法

```
//设置允许跨域的域名，*代表允许任意域名跨域
  res.setHeader("Access-Control-Allow-Origin","*");
```

愿景："让编程不在难学，让技术与生活更加有趣"

第五章 NodeJS连接Mysql

第1集 mysql介绍

简介：介绍mysql及mysql软件推荐

- 什么是mysql?

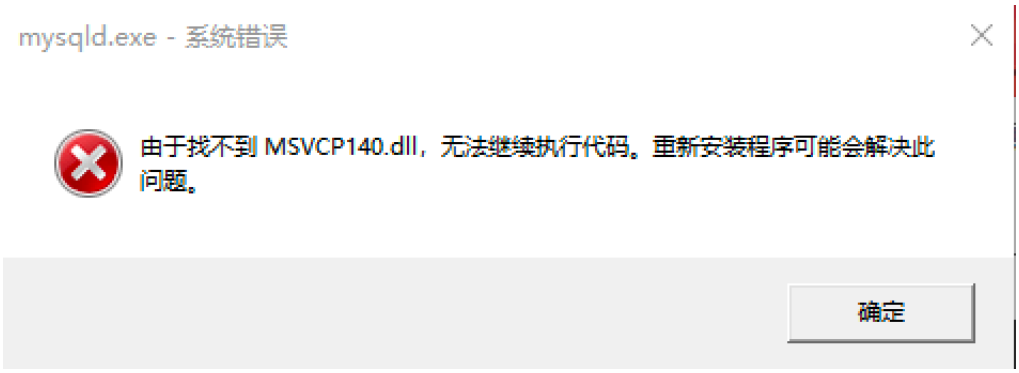
  mysql是一个数据库管理系统。数据库是存储、管理数据的仓库。

- mysql环境安装配置
  - **windows安装及配置**

```
windows安装配置教程
https://blog.csdn.net/qq_37350706/article/details/81707862
```

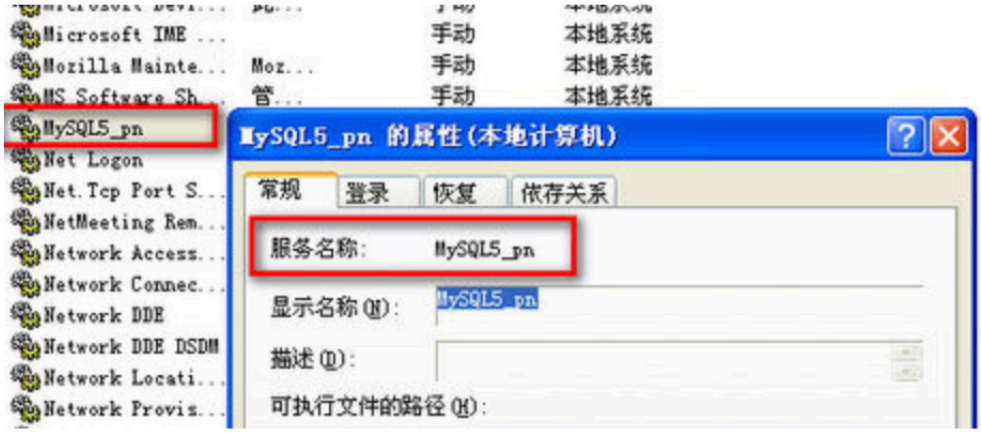
安装过程出现错误总结：

- 执行mysqld出现以下错误，可能是电脑缺少VC++ 2015运行库，安装一下就可以了

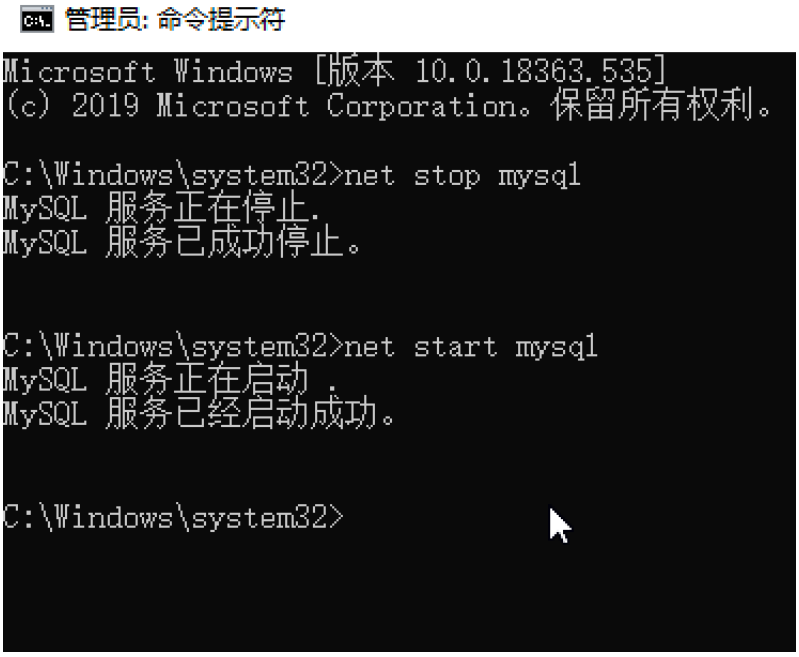


VC++2015下载地址：https://www.microsoft.com/zh-CN/download/details.aspx?id=48145

- net start mysql服务名无效
  1. win+R打开运行窗口，输入services.msc
  2. 在其中查看mysql的服务名，找到mysql开头的，如下图



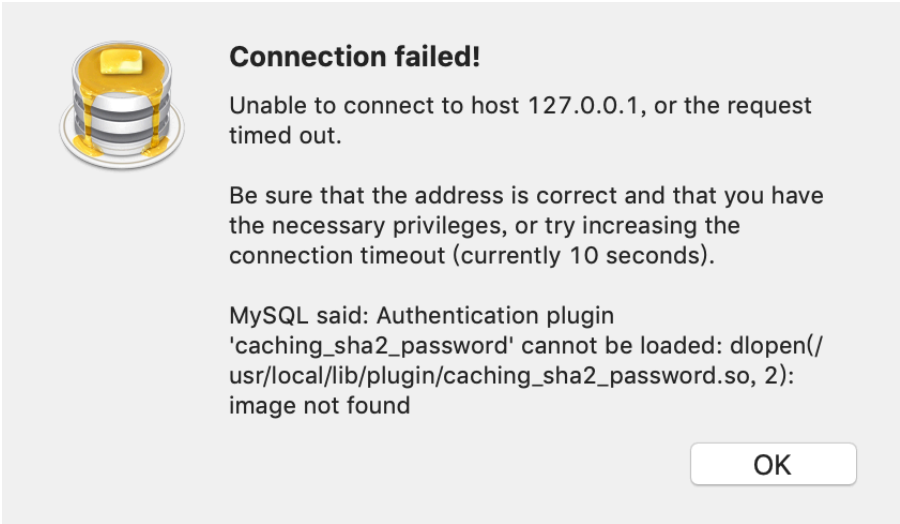
  3. 以管理员身份打开cmd，输入net start mysql服务名
  4. 出现下图表示启动成功



        - 出现拒绝访问，denied等字样，表示需使用管理员运行命令行
    - mac安装及配置

      mac下mysql8安装教程：https://blog.csdn.net/luzhensmart/article/details/82948133

- 客户端连接报错



  客户端不支持mysql8的新密码格式，修改密码加密规则之后重新修改密码即可。

  解决方案地址：https://blog.csdn.net/u011182575/article/details/80821418

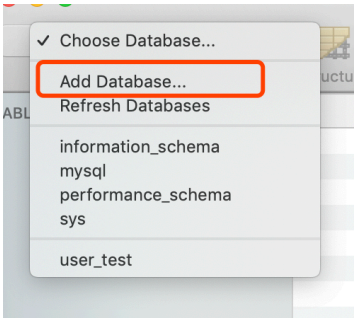
- mysql数据库管理工具
    - MySQL Workbench：下载地址：https://dev.mysql.com/downloads/workbench/

- navicat
- sequal pro：下载地址https://sequelpro.com/test-builds

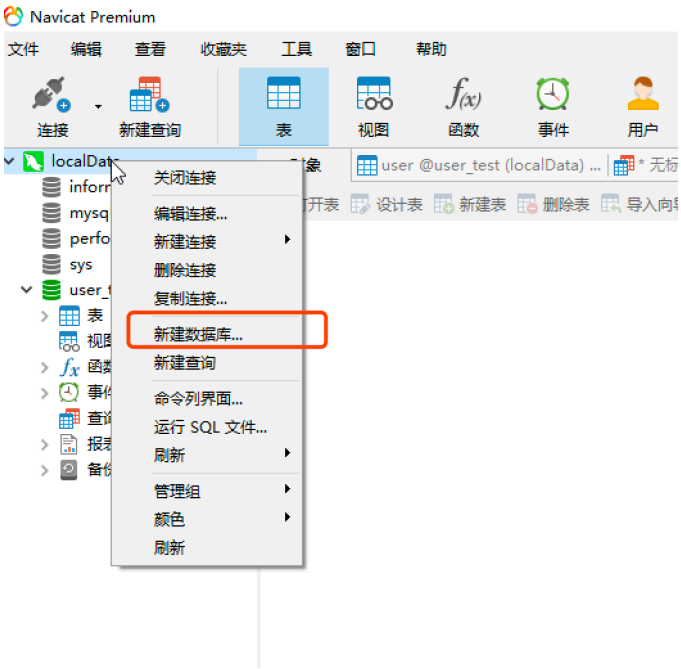
第2集 开发前准备之mysql数据库设计

简介：讲解如何去创建一个数据库

- 创建数据库
  - Sequel pro



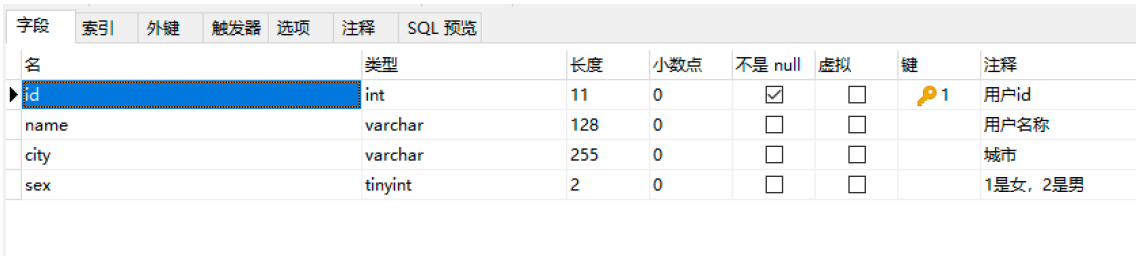
  - Navicat



- 创建表
  - Sequel pro

- Navicat

- 设计表
  - navicat



名	类型	长度	小数点	不是 null	虚拟	键	注释
id	int	11	0	☑	☐	1	用户id
name	varchar	128	0	☐	☐		用户名称
city	varchar	255	0	☐	☐		城市
sex	tinyint	2	0	☐	☐		1是女，2是男

  - sequel pro

Field	Type	Length	Unsign...	Zero...	Bina...	Allow Null	Key	Default	Extra	Encoding	Collation	Comment
id	INT	11	☑	☐	☐	☐	PRI		auto_incre...			用户id
name	VARCHAR	128	☐	☐	☐	☑		NULL	None	UTF-8 Uni...	utf8_gener...	用户名称
city	VARCHAR	128	☐	☐	☐	☑		NULL	None	UTF-8 Uni...	utf8_gener...	城市
sex	TINYINT	2	☐	☐	☐	☑		NULL	None			1是女，2...

- id设为主键（主键表示该字段为唯一标识，不能为空），自动递增
- varchar为字符类型，长度可根据业务需求设置
- int，tinyint为整数类型，tinyint占用1个字节，int占用4字节，tinyint允许从0到255的所有数字，int允许从-2 147 483 648～ 2 147 483 647

第3集 mysql常用数据库操作语句

简介：讲解数据库中常用的操作语句

- 增加表格数据

```
INSERT INTO table_name ( field1, field2,...fieldN ) VALUES (value1,
value2,...valueN)

//往user表插入一条数据
insert into user (name,city,sex) values ('小小','北京',1)
```

- 删除表格数据

```
DELETE FROM table_name [WHERE Clause]

//删除用户id为5的用户
delete from user where id = 5
```

- 修改表格数据

```
UPDATE table_name SET field1=new-value1, field2=new-value2 [WHERE Clause]

//修改用户id为5的信息
update user set name='小天',city='深圳' where id = 5
```

- 查询表格数据

```
SELECT column_name,column_name FROM table_name [WHERE Clause]

//查询所有用户信息，*表示显示所有字段信息
select * from user

//查询所有用户信息，只显示name和city信息
select name,city from user

//查询id为4的用户
select name,city from user where id = 4

//同时满足两个条件用and
select name,city from user where city = '北京' and sex = 1
```

- 排序

```
SELECT field1, field2,...fieldN FROM table_name1, table_name2...
ORDER BY field1 [ASC [DESC][默认 ASC]], [field2...] [ASC [DESC][默认 ASC]]

默认asc升序排序，desc降序排序

//根据id进行降序排序
select * from user order by id desc
```

- 模糊查询

```
SELECT field1, field2,...fieldN FROM table_name WHERE field1 LIKE
condition1

//查询名字带有红的用户
select * from user where name like '%红%'
```

第4集 NodeJs连接mysql数据库讲解

简介：讲解nodejs安装mysql及连接mysql数据库方式

- mysql模块安装

```
npm install mysql --save
```

- 连接数据库

```
const mysql = require('mysql')

//创建连接
const conn = mysql.createConnection({
  host:'localhost',
  user:'root',
  password:'123456789',
  port:'3306',
  database:'user_test'
})

//建立连接
conn.connect()


let sql = 'select * from user where id = ?'

//执行sql语句
conn.query(sql,[4],(err,result)=>{
  if(err) throw err
  console.log(result)
})

//关闭连接
conn.end()
```

- 通过占位符实现传参,query方法第二参数就是会填充sql语句里的?

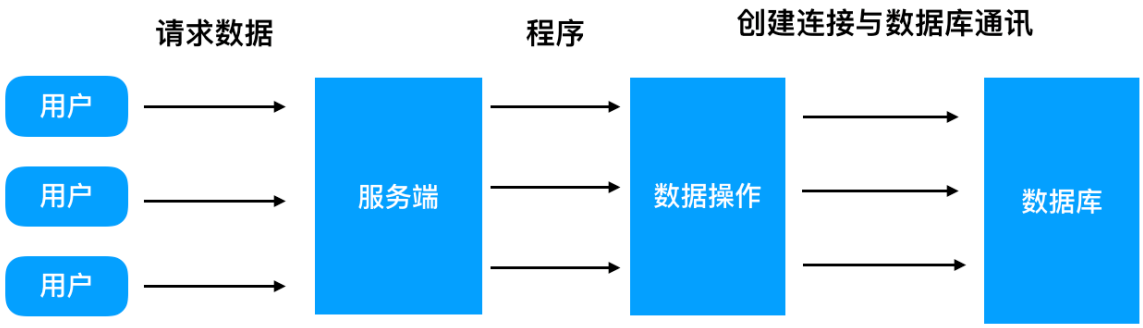
```
let sql = 'select * from user where id = ?'

//执行sql语句
conn.query(sql,[4],(err,result)=>{
  if(err) throw err
  console.log(result)
})
```

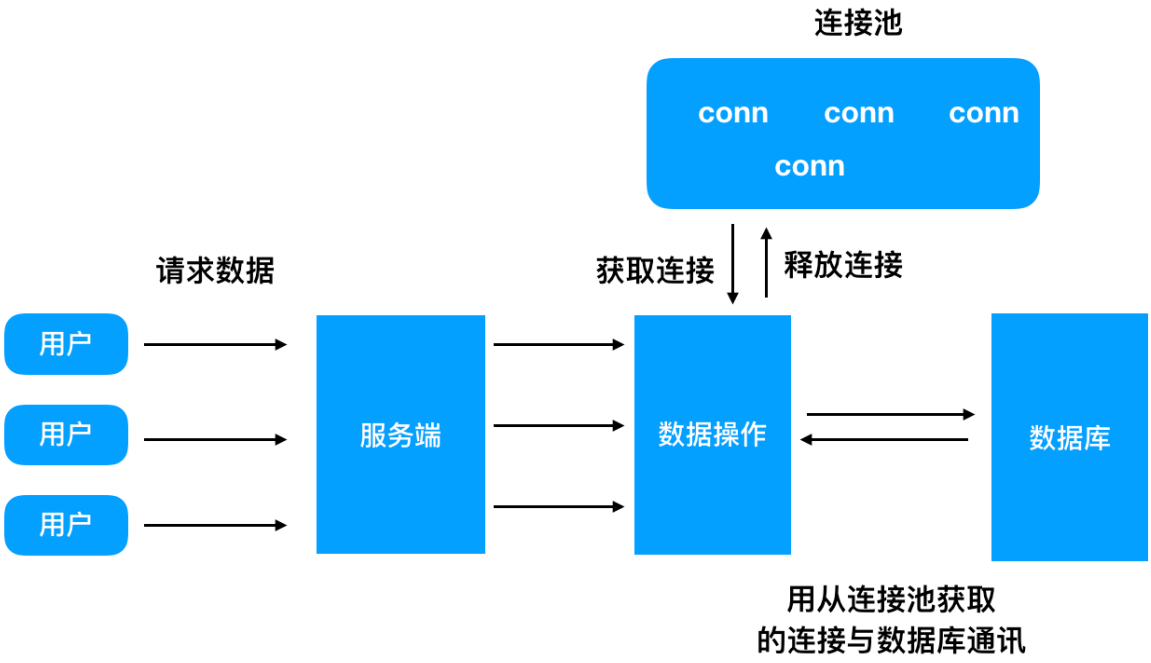
第5集 深度讲解mysql连接池

简介：mysql连接池与普通连接的区别以及它的使用方式

- 频繁的创建、关闭连接会减低系统的性能，提高系统的开销



- 连接池可以有效的管理连接，达到连接复用的效果



- 连接池的使用

```
const mysql = require('mysql')

//创建连接池
const pool = mysql.createPool({
  connectionLimit: 10,
  host: 'localhost',
  user: 'root',
  password: '123456789',
  port: '3306',
  database: 'user_test'
})

//获取连接
pool.getConnection((err, conn) => {
  if (err) throw err
  let sql = 'select * from user where city = ?'

  //执行sql语句
  conn.query(sql, ['广州'], (err, result) => {
    conn.release()
    if (err) throw err
    console.log(result)
  })
})
```

第6集 结合数据库改造用户列表接口（增）

简介：改造接口，通过sql语句操作数据库增加用户

- 数据库配置

```
let dbOption

dbOption = {
  connectionLimit: 10,
  host: 'localhost',
  user: 'root',
  password: '123456789',
  port: '3306',
  database: 'user_test'
}

module.exports = dbOption
```

- 数据库连接，以及query方法封装

```
const mysql = require('mysql')
const dbOption = require('../config/db_config')

//创建连接池
const pool = mysql.createPool(dbOption)


function query (sql,params) {
  return new Promise((resolve, reject) => {
    //获取连接
    pool.getConnection((err, conn) => {
      if (err){
        reject(err)
        return
      }
      //执行sql语句
      conn.query(sql, params, (err, result) => {
        conn.release()
        if (err) {
          reject(err)
          return
        }
        resolve(result)
      })
    })
  })
}


module.exports = query
```

- server通过获取的promise结果来获取数据，并且返回结果

```
let result = routerModal(req, res);
    if (result) {
      result.then(resultData =>{
        res.end(JSON.stringify(resultData))
      })
    } else {
      res.writeHead(404, { 'content-type': 'text/html' })
      res.end('404 not found')
    }
```

第7集 结合数据库改造用户列表接口（删改）

简介：讲解用户列表删除更新接口的改造编写

- 更新用户接口

```
async updateUser(id,userObj){
    // console.log(id,userObj);
    let {name,city,sex} = userObj
    let sql = 'update user set name = ?,city = ?,sex = ? where id = ?'
    let resultData = await query(sql,[name,city,sex,id])
    if(resultData.affectedRows > 0){
      return {
        msg:'更新成功'
      }
    }else{
      return {
        msg:'更新失败'
      }
    }

  }
```

- 删除用户接口

```
async delectUser(id){
    let sql = 'delete from user where id = ?'
    let resultData = await query(sql,[id])
    if(resultData.affectedRows > 0){
      return {
        msg:'删除成功'
      }
    }else{
      return {
        msg:'删除失败'
      }
    }
  }
```

第8集 结合数据库改造用户列表接口（动态查询）

简介：讲解通过sql语句操作数据库实现动态查询

```
async getUserList(urlParams){
    let {name,city} = urlParams
    let sql = 'select * from user where 1=1 '
    if(name){
      sql += 'and name = ?'
    }
    if(city){
      sql += 'and city = ?'
    }
    let resultData = await query(sql,[name,city])
    return resultData
}
```

愿景："让编程不在难学，让技术与生活更加有趣"

第六章 分布式文件储存数据库MongoDB

第1集 MongoDB的介绍及安装

简介：介绍MongoDB及安装方式

- MongoDB是用C++语言编写的非关系型数据库
- MongoDB与mysql的区别

	MongoDB	mysql
数据库模型	非关系型	关系型
表	collection集合	table二维表
表的行数据	document文档	row记录
数据结构	虚拟内存+持久化	不同引擎不同储存方式
查询语句	mongodb查询方式（类似js函数）	sql语句
数据处理	将热数据存储在物理内存中，从而达到快速读写	不同引擎有自己的特点
事务性	不支持	支持事务
占用空间	占用空间大	占用空间小
join操作	没有join	支持join

- mongodb数据库软件及可视化软件安装

  数据库软件安装地址：https://www.mongodb.com/download-center/community

  可视化软件安装地址：https://www.mongodb.com/download-center/compass

  mac安装数据库教程
  - 安装brew教程：https://www.jianshu.com/p/dea776e7effb
  - 安装mongodb教程https://www.cnblogs.com/georgeleoo/p/11479409.html

- 启动mongodb命令
  - mac

```
// 启动
brew services start mongodb-community@4.2
// 关闭
brew services stop mongodb-community@4.2
```

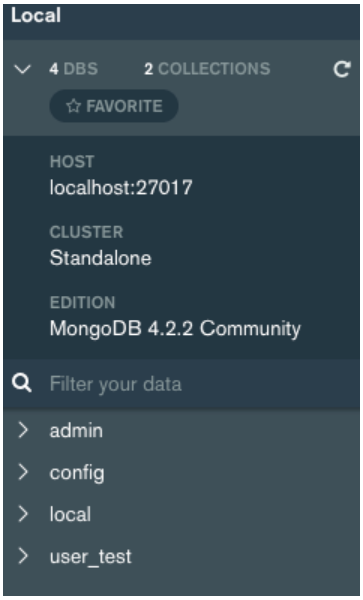
- windows

```
//启动
net start mongodb
//关闭
net stop mongodb
```

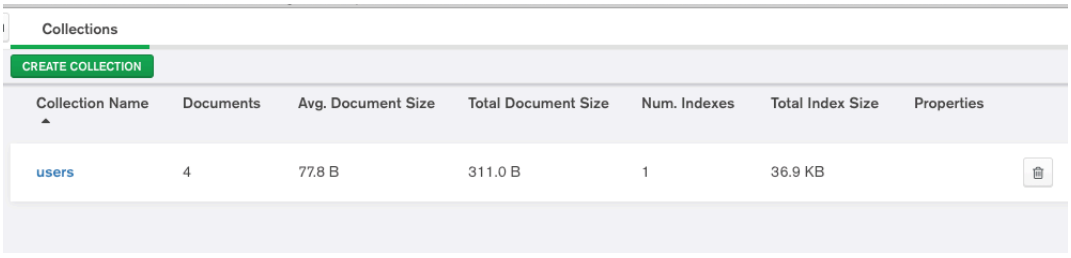
第2集 玩转MongoDB可视化工具

简介：讲解如何使用MongoDB可视化工具

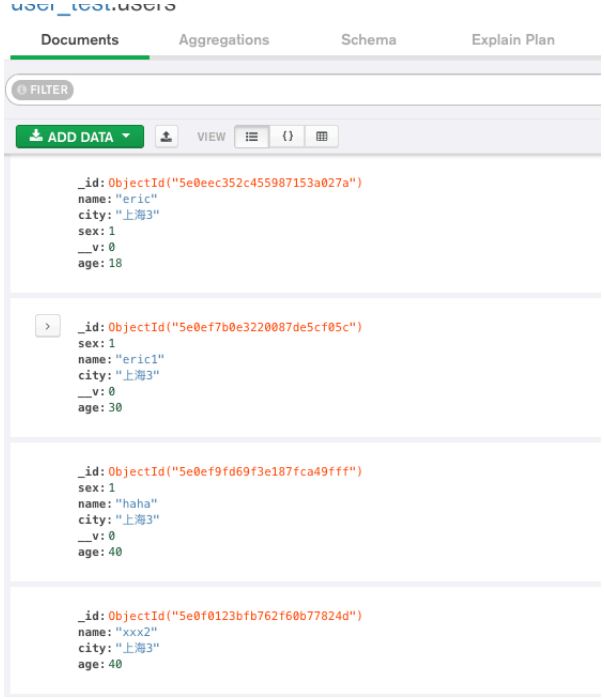
- 数据库列表



- 集合列表,这里相当于mysql的表



- 集合里的文档,相当于mysql的一条数据



第3集 讲解第三方包mongoose的使用

简介：详细讲解如何使用mongoose连接数据库

- 安装mongoose包

```
npm install mongoose
```

- 数据库连接

```
const mongoose = require('mongoose')

//连接数据库,返回promise
mongoose.connect('mongodb://localhost/user_test',{
  useNewUrlParser: true,
  useUnifiedTopology: true
}).then(()=>{
  console.log('连接数据库成功')
}).catch(err => {
  console.log(err,'连接数据库失败')
})
```

官方文档地址：https://mongoosejs.com/docs/connections.html

第4集 MongoDB常用数据库操作之创建集合、文档

简介：讲解MongoDB创建集合、创建文档

- mongodb不需要显示创建数据库，如果数据库不存在，它会自动创建。
- 创建集合

```
const Schema = new mongoose.Schema(options) //创建集合结构（规则）  options集合的结构（规则）
const Model = mongoose.model(modelName, schema)  //modalName集合名称 schema集合结构

//案例代码

//创建集合结构
const userSchema = new Schema({
  name:String,
  city:String,
  sex:Number
})

//创建集合
const Model = mongoose.model('user',userSchema)
```

- 创建文档
  - 第一种方式

```
const doc = new Model(options) //options符合集合规则的文档数据，以对象形式
doc.save()  //将文档插入数据库中


//案例代码
  //创建文档
const doc = new Model({
    name:'eric',
    city:'深圳',
    sex:2
  })

  //将文档插入数据库中
doc.save()
```

- 第二种方式

```
//options符合集合规则的文档数据，以对象形式,err为报错信息，doc是插入的文档
Model.create(options,(err,doc))


//案例代码
//创建文档并把文档插入数据库中
Model.create({
  name:'嘻嘻',
  city:'广州',
  sex:1
},(err,doc) => {
  if(err) throw err
  console.log(doc)
})
```

第5集 讲解MongoDB如何导入文件数据

简介：讲解MongoDB里如何导入文件数据到数据库

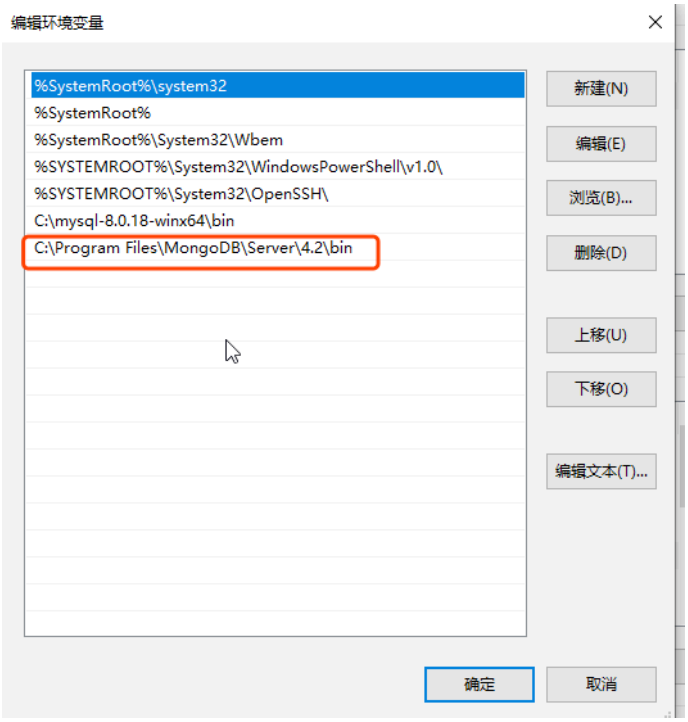
- 数据库导入数据

```
mongoimport -d 数据库名称 -c 集合名称 --file 导入的数据文件路径

mongoimport -d test -c users --file ./user.json
```

- mongoimport环境变量配置
  - mac

    如果使用不了命令，访问该连接，用教程里的第二个方式配置环境变量https://www.cnblogs.com/georgeleoo/p/11479409.html
  - window



配置系统环境变量，将mongodb安装路径添加进去（要进到bin文件夹）

第6集 MongoDB常用数据库操作之查询文档

简介：讲解MongoDB查询文档的多种用法

- 查询文档

```
Model.find(条件)  //根据条件查询文档，条件为空则查询所有文档，          返回数组
Model.findOne(条件)  //默认返回当前集合中的第一条文档          返回对象

Model.find({name:'6'}).then(res => {
   console.log(res)
})

Model.findOne({name:'2'}).then(res => {
   console.log(res)
})
```

- 区间查询

```
{key:{$gt:value,$lt:value}}   gt大于 lt小于    gte大于等于   lte小于等于

Model.find({age:{$gte:18,$lte:38}}).then(res => {
   console.log(res)
})
```

- 模糊查询

```
{key:正则表达式}

Model.find({city:/上/}).then(res => {
   console.log(res)
})
```

- 选择要查询的字段

```
Modal.find().select(arg)  //arg为要操作的字段      字段前加上-表示不查询该字段

Model.find().select('-name').then(res => {
   console.log(res)
})
```

- 排序

```
Modal.find().sort(arg)  //arg为要操作的字段       字段前加-表示降序排列

Model.find().sort('-age').then(res => {
   console.log(res)
})
```

- 跳过多少条数据、限制查询数量

```
Modal.find().skip(num).limit(num)   //skip跳过多少条数据，limit限制查询数量


Model.find().skip(2).limit(3).then(res => {
  console.log(res)
})
```

第7集 MongoDB常用数据库操作之更新文档

简介：讲解MongoDB更新文档的用法

- 更新单个文档

```
//找到一个文档并更新,如果查询多个文档，则更新第一个匹配文档      返回值为该文档
Modal.findOneAndUpdate(条件,更新的值)
//更新指定条件文档,如果查询多个文档，则更新第一个匹配文档
Modal.updateOne(条件，更新的值)

//全局配置
mongoose.set('useFindAndModify',false)

Model.findOneAndUpdate({name:'2'},{city:'深圳'}).then(res => {
  console.log(res)
})


Model.updateOne({name:'2'},{city:'上海'}).then(res => {
  console.log(res)
})
```

- 更新多个文档

```
Modal.updateMany(条件，更新的值)  //如果条件为空，则会更新全部文档


Model.updateMany({name:'2'},{city:"深圳"}).then(res => {
  console.log(res)
})
```

第8集 MongoDB常用数据库操作之删除文档

简介：讲解MongoDB删除文档的用法

- 删除单个文档

```
//找到一个文档并删除,如果查询多个文档，则删除第一个匹配文档      返回值为该文档
Modal.findOneAndDelete(条件)
//删除指定条件文档,如果查询多个文档，则删除第一个匹配文档          返回值是一个成功对象
Modal.delectOne(条件)


Model.findOneAndDelete({name:'xx'}).then(res => {
   console.log(res)
})

Model.deleteOne({name:'2'}).then(res => {
  console.log(res)
})
```

- 删除多个文档

```
Modal.deleteMany(条件)    //如果条件为空，则会删除全部文档


Model.deleteMany().then(res => {
  console.log(res)
})
```

第9集 深度讲解MongoDB字段验证

简介：讲解集合中字段的验证

- required 验证字段是否为必须输入，值为boolean

```
name:{
   type:String,
   required:[true,'该字段为必选字段'],
}
```

- minlength，maxlength 验证字符的值的最小长度和最大长度

```
name:{
    type:String,
    required:[true,'该字段为必选字段'],
    minlength:[2,'输入值长度小于最小长度'],
    maxlength:[6,'输入值长度大于最大长度'],
 }
```

- trim 去除字符串首尾空格，值为boolean

```
name:{
  type:String,
  required:[true,'该字段为必选字段'],
  minlength:[2,'输入值长度小于最小长度'],
  maxlength:[6,'输入值长度大于最大长度'],
  trim:true
}
```

- min、max 验证最小最大数字

```
age:{
  type:Number,
  min:18,
  max:30
},
```

- default 默认值

```
createTime:{
  type:Date,
  default:Date.now
},
```

- enum 规定输入的值

```
hobbies:{
    type:String,
    enum:{
      values:['唱','跳','Rap'],
      message:'该值不在设定的值当中'
    }
  },
```

- validate 根据自定义条件验证，通过validator函数处理输入值，message为自定义错误信息

```
validate:{
```

```
  validator:v => {
    //todo 返回布尔值验证输入值是否有效
  }
}


score:{
    type:Number,
    validate:{
      validator:v => {
        //返回布尔值
        return v&&v>0&&v<100
      },
      message:'不是有效的分数'
    }
  }
```

- 通过catch获取errors对象，遍历对象从中获取对应字段自定义报错信息

```
Modal.create().then().catch(error => {
  //获取错误信息对象
  const errs = error.errors;
  //循环错误信息对象
  for(var i in errs ){
    console.log(err[i].message)
  }
})
```

愿景："让编程不在难学，让技术与生活更加有趣"

第七章 NodeJs进阶知识线程与进程

第八章 留言板实战

第九章 上线部署

第十章 总结