Node.js 基础

课程目标

- 了解 nodejs 特点和应用场景
- 掌握 node 模块系统使用
- 掌握核心 api 使用
- 实战一个简版 Express 服务器

学习 Node 的目标

- 优秀的前端，可以和后端有效沟通
- 敏捷的全栈 --快速开发全站应用
- 架构师-- 践行工程化思想

NodeJS 是什么

简单的说 Node.js 就是运行在服务端的 JavaScript。

https://nodejs.org/en/
http://nodejs.cn/api/

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行时环境。

Node.js 是一个事件驱动 I/O 服务端 JavaScript 环境，基于 Google 的 V8 引擎，V8 引擎执行 Javascript 的速度非常快，性能非常好。

运行时环境 runtime: 就是程序运行时， js 是解释性语言， 有些是编译性语言

node.js 特性：

Node.js 的强大功能体现在很多方面，如事件驱动、异步处理、非阻塞 I/O 等。在这里将介绍 Node.js 具备的不同于其它框架的特点。

● 单线程

这里的单线程是指主线程为“单线程”，所有的阻塞部分交给部分的线程池处理，然后这个主线程通过一个队列跟线程池协作，Node.js 以单线程为基础的，这个正是 Node.js 保持轻量级和高性能的关键。

单线程的弱点：

1. 无法利用多核 CPU

2.一个用户造成线程奔溃，整个服务都奔溃

3.大量计算占用 CPU 导致无法继续调用异步 I/O

单线程的优点：

1.操作系统完全不再有线程创建，销毁的时间开销

2.减少了内存的开销，操作系统的内存换页

3.不用像多线程编程一样处处在意状态的同步问题

4.一个 8G 的内存服务器，可以同时处理超过四万用户的连接

● 异步、非阻塞 I/O

异步 I/O 机制，因此在执行访问数据库的代码之后，将立即去执行其后面的代码，把数据库返回结果的处理代码放在回调函数中，每个调用之间无需等待之前的 I/O 调用结束，提高了程序的执行效率。

一个异步 I/O 的大致流程：

（1） 发起 I/O 调用

- 用户通过 JavaScript 代码调用 Node 核心模块，将参数和回调函数传入核心模块。
- Node 核心模块会将传入的参数和回调函数封装成一个请求对象。
- 将这个请求对象推入 I/O 线程池等待执行。
- JavaScript 发起的异步调用结束，JavaScript 线程继续执行后续操作。

（2）执行回调

- I/O 操作完成后会将结果存储到请求对象的 result 属性上，并发出操作完成的通知。
- 每次事件循环时会检查是否有完成的 I/O 操作，如果有就将请求对象加入观察者队列中，之后当做事件处理。
- 处理 I/O 观察者事件时会取出之前封装在请求对象中的回调函数，执行这个回调函数，并将 result 当做参数，以完成 JavaScript 回调的目的。

```
var fs=require('fs');
fs.readFile('/path',function(err,file){
console.log('读取文件完成')
});
console.log('发起读取文件')
```

这里的“发起读取文件”是在“读取文件完成”之前输出的，同样，“读取文件完成”的执行也取决于读取文件的异步调用何时结束。

● 事件与回调函数（事件驱动）

在 node.js 中，一个时刻只能执行一个事件回调函数，但在执行一个事件回调函数的过程中，可以转而处理其他事情（比如又有新用户连接），然后返回继续执行原事件的回调函数，这种处理机制称为“事件环”机制。

事件驱动的优势在于充分利用了系统资源，执行代码无须等待某种操作完成，有限的资源可以用于其他的任务。Node.js 的目标是为后端的网络服务编程,在服务器的开发中，并发的请求处理是一个大问题，阻塞式的函数会导致资源的浪费和时间的延迟。通过事件的注册、异步函数，开发人员可以提高资源的利用率，性能也会改善。

准备工作

运行 node 程序 node 01-runnode.js

监听配置 nodemon： 它的作用是监听代码文件的变动，当代码改变之后，自动重启

npm install -g nodemon 或者 yarn global add nodemon

安装在全局。

调试 node 程序：Debug - Start Debugging

util.promisify 的那些事儿

util.promisify 是在 **node.js 8.x** 版本中新增的一个工具，用于将老式的 Error first callback 转换为 Promise 对象，让老项目改造变得更为轻松。

在官方推出这个工具之前，民间已经有很多类似的工具了，比如 es6-promisify、thenify、bluebird.promisify。

手动实现一个 promisify

es6 原生支持了 promise 规范，可以使用 new Promise()方法得到一个 promise 对象

参考：

复习：

```
function getName() {
	let p
	p = new Promise(function (resolve, reject) {
		setTimeout(function() {
			resolve('jianyong')
		}, 2000)
	})
	return p
}
getName().then(function (name) {
	console.log(name)
})
```

异步操作 promise 化

nodejs 中，很多函数都是异步执行，我们需写一个回调函数作为异步函数的最后一个参数传入。比如 fs 包中的 readFile 这样的异步函数。

```
require('fs').readFile('./test.js', 'utf-8', function (err, data) {
		if (err) {
			console.log(err)
		}
```

```
        console.log(data)
})
```

使用 promise 改造 fs.readFile

```
function readfile() {
    return new Promise(function (resolve, reject) {
        require('fs').readFile('./test.js', 'utf-8', function (err, data) {
            if (err) {
                reject(err)
            }
            resolve(data)
        })
    })
}
readfile().then(function (data) {console.log(data)})
```

promise 化

如果每次遇到异步操作就需要像上面例子那样包裹改造一下岂不是很麻烦？！

可以使用下面脚本改造需要 promisify 的异步函数：

```
function promisify(f) {
    return function () {
        let args = Array.prototype.slice.call(arguments);
        return new Promise(function (resolve, reject) {
            args.push(function (err, result) {
                if (err) reject(err);
                else resolve(result);
            });
            f.apply(null, args);
        });
    }
}
```

使用示例：

```
stat = promisify(require('fs').stat)
```

stat('./test.js').then(data => console.log(data)).catch(err => console.log(err))

内置的 promisify 转换后函数

如果你的 Node 版本使用 10.x 以上的，还可以从很多内置的模块中找到类似.promises 的子模块，这里边包含了该模块中常用的回调函数的 Promise 版本（都是 async 函数），无需再手动进行 promisify 转换了。

拿 fs 模块进行举例：

```
// 之前引入一些 fs 相关的 API 是这样做的
const { readFile, stat } = require('fs')

// 而现在可以很简单的改为
const { readFile, stat } = require('fs').promises
// 或者
const { promises: { readFile, stat } } = require('fs')

readFile('./package.json').then((res)=>{
   console.log('从回调函数得到的数据：',res.toString());
})
```

使用模块(module)

node 内建模块

```
require('os')
```

第三方模块 ， 需要安装 cpu-stat

```
require("cpu-stat");
```

自定义模块：

```
// 导出
```

```
module.exports = {}
// 导入
require('./conf')
```

核心 API

fs

```
const fs = require('fs');
fs.readFile('./笔记.md', (err, data) => {
if (err) throw err;
console.log(data);
});
```

Buffer（缓冲区）

JavaScript 语言自身只有字符串数据类型，没有二进制数据类型。

但在处理像 TCP 流或文件流时，必须使用到二进制数据。因此在 Node.js 中，定义了一个 Buffer 类，该类用来创建一个专门存放二进制数据的缓存区。

在 Node.js 中，Buffer 类是随 Node 内核一起发布的核心库。Buffer 库为 Node.js 带来了一种存储原始数据的方法，可以让 Node.js 处理二进制数据，每当需要在 Node.js 中处理 I/O 操作中移动的数据时，就有可能使用 Buffer 库。原始数据存储在 Buffer 类的实例中。一个 Buffer 类似于一个整数数组，但它对应于 V8 堆内存之外的一块原始内存。

在 v6.0 之前创建 Buffer 对象直接使用 new Buffer()构造函数来创建对象实例，但是 Buffer 对内存的权限操作相比很大，可以直接捕获一些敏感信息，所以在 v6.0 以后，官方文档里面建议使用 ***Buffer.from()*** *接口去创建 Buffer 对象。*

创建 Buffer 类

Buffer 提供了以下 API 来创建 Buffer 类：

Buffer.alloc(size[, fill[, encoding]])： 返回一个指定大小的 Buffer 实例，如果没有设置 fill，则默认填满 0

Buffer.allocUnsafe(size)： 返回一个指定大小的 Buffer 实例，但是它不会被初始化，所以它可能包含敏感的数据

Buffer.from(array)：返回一个被 array 的值初始化的新的 Buffer 实例(传入的 array 的元素只能是数字，不然就会自动被 0 覆盖)

Buffer.from(arrayBuffer[, byteOffset[, length]])：返回一个新建的与给定的 ArrayBuffer 共享同一内存的 Buffer。

Buffer.from(buffer)： 复制传入的 Buffer 实例的数据，并返回一个新的 Buffer 实例

Buffer.from(string[, encoding])： 返回一个被 string 的值初始化的新的 Buffer 实例

```
// 创建
const buf1 = Buffer.alloc(10);
const buf2 = Buffer.from([1, 2, 3]);
const buf3 = Buffer.from('Buffer 创建方法');

// 写入缓冲区
buf1.write('hello');
// 从缓冲区读取数据

console.log(buf3.toString());  //默认为 'utf8' 。

// 合并
const buf4 = Buffer.concat([buf1, buf3]);
```

http

Node 的网络应用都需要先创建一个网络服务对象，这里我们通过 createServer 来实现。传入 createServer 的 function 在每次 HTTP 请求时都将被调用执行，因此这个 function 也被称为请求的处理者。事实上通过 createServer 返回的 Server 对象是一个 **EventEmitter**，我们需要做的仅仅是在这里保存这个 server 对象，并在之后对其添加监听器。

```
var http = require('http');
var server = http.createServer(function(request, response) {
// handle your request
});
```

传入 createServer 的 function 在每次 HTTP 请求时都将被调用执行，因此这个 function 也被称为请求的处理者。事实上通过 createServer 返回的 Server 对象是一个 EventEmitter，我们需要做的仅仅是在这里保存这个 server 对象，并在之后对其添加监听器。

```
var http = require('http');
var server = http.createServer();
server.on('request', function(request, response) {
// handle your request
});
```

当 HTTP 请求这个服务时，node 调用请求处理者 function 并传入一些用于处理事务相关的对象：request 和 response。我们可以非常方便的获得这两个对象。

```
var http = require('http');
var server = http.createServer();
server.on('request', function(request, response) {
// handle your request
}).listen(8080);
```

为了对实际的请求提供服务，在 server 对象上需要调用 listen 方法。绝大多数情况你需要传入 listen 你想要服务监听的端口号，这里也存在很多其他的可选方案

EventEmitter 类

events 模块只提供了一个对象： events.EventEmitter。EventEmitter 的核心就是事件触发与事件监听器功能的封装。

你可以通过 require("events");来访问该模块。

都绑定了一个监听器，eventEmitter.on()用于监听事件，eventEmitter.emit()用于触发事件。

继承 EventEmitter

大多数时候我们不会直接使用 EventEmitter，而是在对象中继承它。包括 fs、net、 http 在内的，只要是支持事件响应的核心模块都是 EventEmitter 的子类。

为什么要这样做呢？原因有两点：

首先，具有某个实体功能的对象实现事件符合语义， 事件的监听和发生应该是一个对象的方法。

其次 JavaScript 的对象机制是基于原型的，支持 部分多重继承，继承 EventEmitter 不会打乱对象原有的继承关系。

一.从 EventEmitter 类继承

问题

你希望通过事件驱动的手段来解决问题。假如你有一个音乐播放器类，你希望在异步事件或者接口成功发生的时候来操作它。

解决办法

你需要创建一个基于 EventEmitter 类的自定义类，基于 EventEmitter 类得到的示例，都绑定了一个监听器，eventEmitter.on()用于监听事件，eventEmitter.emit()用于触发事件。

下面是一个音乐播放器的实例：

首先实现对 EventEmitter 类的继承

```
const EventEmitter = require('events');
//所有的构造函数都必须继承自 EventEmitter 类;
class MusicPlayer extends EventEmitter{};
//再通过这个构造函数来创建触发事件的对象
let musicPlayer = new MusicPlayer();

// 通过继承创建的实例对象有绑定监听器，可以调用 on,emit 方法
// 订阅 发布
//音响设备 AudioDevice
let AudioDevice = {
    play:function(track){
        //
        console.log(track);
        console.log('audio play');
    },
    stop:function(){
        //
        console.log('audio stop');
    }
}
//监听事件
musicPlayer.on('play',function(track){
    this.playing = true;
    AudioDevice.play(track);
})
//监听事件
musicPlayer.on('stop',function(track){
    this.playing = false;
    AudioDevice.stop();
});

musicPlayer.emit('play','The Roots - The Fire');
```

```
setTimeout(function(){
//emit 触发事件
    musicPlayer.emit('stop')
},3000);
```

二.添加多个监听器

我们可以给事件添加多个监听器，比如上面的音乐播放器，我们在 play 触发时需要做些其他的事情比如用户界面需要更新等。对 play 事件添加一个新的监听器就能轻松实现。

```
musicPlayer.on('play',function(track){
   this.playing = true;
   AudioDevice.play(track);});//添加新的监听器
musicPlayer.on('play',function(track){
   console.log('添加新的监听器')});
```

三.移除监听器

```
eventEmitter.removeListener(eventname,fn):移除一个监听器
emitter.removeAllListeners([eventName]):移除所有的监听器

let playFn1 = function(track){
   this.playing = true;
   AudioDevice.play(track);}
musicPlayer.on('play',playFn1);//移除监听器
musicPlayer.removeEventListener('play',playFn1())
```

四.错误处理

通过监听 error 事件，来进行错误处理。

```
//错误处理 let playFn1 = function(track){
    this.playing = true;
    AudioDevice.play(track);
    //这里如果出现错误，就触发 error 事件
    this.emit('error','unable to play')}
        musicPlayer.on('play',playFn1);
        musicPlayer.on('error',function(err){
        console.log(err);})
```

拓展： Vue 父子组件通信之$emit

$emit 的作用

在 Vue 中，父组件监听子组件触发的事件，需要在子组件用使用 $emit 触发，在父组件中使用 v-on: / @ 自定义事件监听。

stream

Stream 是 Node.js 最基本的概念之一，Node.js 内部的大部分与 IO 相关的模块，比如 http、net、fs，都是建立在各种 Stream 之上的。

下面这个经典的例子应该大部分人都知道，对于大文件，我们不需要把它完全读入内存，而是使用 Stream 流式地把它发送出去:

在业务代码中合理地使用 Stream 能很大程度地提升性能，当然是但实际的业务中我们很可能会忽略这一点

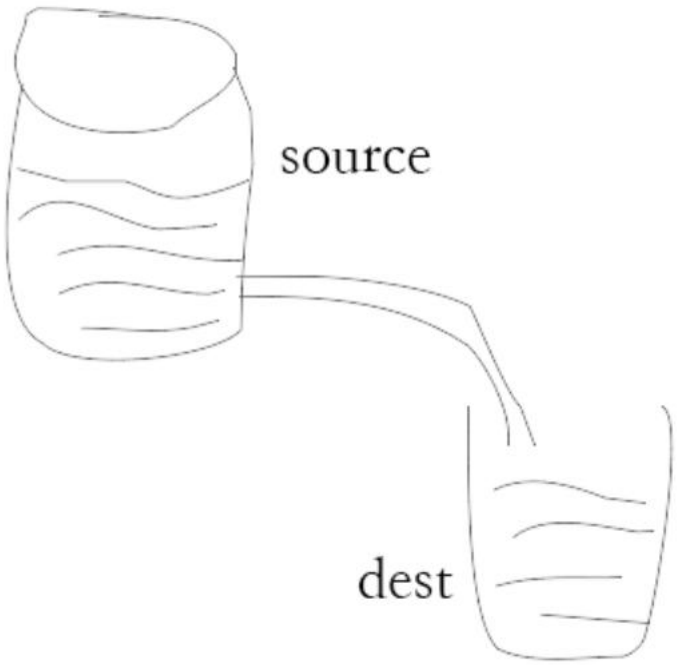
```
创建可读流
const rs = fs.createReadStream('./data.txt')
// 设置编码为 utf8。
readerStream.setEncoding('UTF8');


//写入流
var data = '菜鸟教程官网地址：www.runoob.com';
```

```
// 创建一个可以写入的流，写入到文件 data.txt 中
var ws= fs.createWriteStream('output.txt');
// 使用 utf8 编码写入数据
ws.write(data,'UTF8');
```

管道流

管道提供了一个输出流到输入流的机制。通常我们用于从一个流中获取数据并将数据传递到另外一个流中。



如上面的图片所示，我们把文件比作装水的桶，而水就是文件里的内容，我们用一根管子(pipe)连接两个桶使得水从一个桶流入另一个桶，这样就慢慢的实现了大文件的复制过程。

以下实例我们通过读取一个文件内容并将内容写入到另外一个文件中。

设置 input.txt 文件内容如下：

创建 main.js 文件，代码如下：

```
var fs = require("fs");
// 创建一个可读流
```

```
var readerStream = fs.createReadStream('input.txt');
// 创建一个可写流
var writerStream = fs.createWriteStream('output.txt');
// 管道读写操作
// 读取 input.txt 文件内容，并将内容写入到 output.txt 文件中
readerStream.pipe(writerStream);

console.log("程序执行完毕");
```

仿写一个简版 Express

体验 express

```
// npm i express
// const express = require('express');

const express = require('./06-myexpress.js');
const app = express();
app.get('/',(req,res) => {
    res.end('Hello world')
})
app.get('/users',(req,res) => {
    res.end(JSON.stringify({name:'abc'}))
})
app.listen(3000 , () => {
    console.log('Example listen at 3000')
})
```

实现 myexpress

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

let router = []

class Application {
    get(path, handler) {
```

```
            router.push({
                path,
                method: 'get',
                handler
            })
        }
        listen() {
            const server = http.createServer((req, res) => {
                const { pathname } = url.parse(req.url, true)
                // for (const item of router) {
                //     const { path, method, handler } = item
                //     if (pathname == path && req.method.toLowerCase() == method) {
                //         return handler(req, res)
                //     }
                // }
                router
                    .find(v => pathname === v.path && req.method.toLowerCase() ==
v.method)
                    .handler(req,res)
            })
            //在 Application 原型上添加 listen 方法匹配路径，执行对应的 handler 方法
            server.listen(...arguments)
        }

}
module.exports = function createApplication(){
    return new Application()
}
```