**Nodejs**

**建立web服务器**

**nodejs的全局变量 process 在命令行输入**

**模块的分类和引用**



**模块的流程**



**url的解析**

**url 解析 定位所寻资源**

**> url.parse('http://www.imooc.com/')**

**Url {**

**protocol: 'http:', 指定底层使用的协议**

**slashes: true, 协议双斜线**

**auth: null,**

**host: 'www.imooc.com', 服务器域名**

**port: null, 端口**

**hostname: 'www.imooc.com', 主机名**

**hash: null, 哈希值（通常是页面锚点）**

**search: null, 查询字符串参数**

**query: null, 发给http服务器的数据（通常等号分隔开的键值叫参数串）**

**pathname: '/', 访问资源路径名**

**path: '/', 路径**

**href: 'http://www.imooc.com/' } 未解析的超链接**



**QueryString 参数处理 （序列化）**

**通过url解析，将url的query解析成一个对象**

**我们常需要传递不同用途的参数追加到url地址中发送给服务器，服务器拿到的是字符串的参数串，它不知道个数、参数名和参数值，需要一个途径将参数一个一个放到对象里，我们可以遍历这个对象，也可以获取对象里的值。**

**就像每次打开慕课网课程视频，这个页面会自动向后台去拿一些跟当前视频有关的一些视频信息。这就依赖于路径结合参数来告知服务器，需要的是当前的哪一节课程。**



命令行：querystring.stringify({name:’scott’,course:[‘jade’,’node’],from:’’})

// ‘name=scott&course=jade&course=node&from=’

querystring.parse( ‘name=scott&course=jade&course=node&from=’)

// {name:’scott’,course:[‘jade’,’node’],from:’’}

**querystring .escape(‘<哈哈>’) 参数处理之 转译和反转译**

**querystring .unescape(‘ ’)**

**事件模块 events**

**var eventEmitter = require(‘events’).EventEmitter**

**obj.emit(‘事件’ , ’参数’)**

**查看事件是否被监听过**

**.emit() 时会返回一个布尔值 true则被监听过**

**移除一个事件 removeListener**

**obj.removeListener(‘事件’ , 函数名)**

**obj.removeAllListeners(‘事件’)**

**查询监听 个数**

**obj.listeners(‘事件’) .length**

**eventEmitter .listenerCount(obj , ‘事件’)**

**HTTP get / request**

**http.request( options[,callback] )**

**get就是对于request的一个封装，所以get能做的事情，request都能做**

**get和request可以从后台发起一个http的请求，从而可以获取远程资源，甚至是更新和同**

**步远程的资源。本质上request返回的是一个clientRequest类的一个实例，clientRequest实例是**

**一个可写的流，如果你需要使用POST请求上传一个文件，这个文件就会被写入到ClientRequest**

**的对象里面。**

**ClientRequest接受2个参数，**

1. **option 可以是字符串或对象，如果是字符串，这个字符串会被url模块的**

**parse方法，解析为一个对象；如果是对象，就可以进行一系列的配置，来定制我们需要发出的请求的格式。**

1. **回调函数 （可选参数） 通过回调函数可以接收到远端服务器的响应数据，也**

**就是response**

**常用参数：**

**host 是我们将要请求的服务器域名或IP地址**

**hostname 是host别名**

**port 是远端服务器的端口，默认值是80**

**localAddress 是用于绑定网络连接的本地接口，看文档对应的翻译**

**socketPath**

**method 指定http请求方法的字符串，默认是get**

**path 请求的路径，默认是根路径 / ,如果是有查询的字符串，就 需要在后面追加一些参数**

**headers 是包含请求头的一个对象**

**auth 用于计算认证头的基本的认证，一般是user跟上一个 password**

**agent 控制agent的行为，也就是一个代理**

**keepAlive 保持资源池周围的套接字在未来能被继续用于其它请 求，默认的值是false**

**keepAliveMsecs**

**这些参数不是每一个都需要传递，大部分是默认值，我们常用的有：主机、端口、路径、请求方法、请求头和认证。**

**实例：request怎么实现一个评论的提交？**

**Promise 摆脱callback回调函数的阅读调试的痛苦**

**npm install bluebird**

**引入文件：/bluebird/js/brower/bluebird.js**

**例：**

**var promise = window.Promise**

**function promiseAnimate(ball,distance ){**

**return new promise(function(resolve,reject){**

**...**

**})**

**}**

**promiseAnimate(ball1,100)**

**.then(function(){**

**return promiseAnimate(ball2,200)**

**})**

**.then(function(){**

**return promiseAnimate(ball3,300)**

**})**

**...**

**Promise : 是JavaScript针对异步操作场景的解决方案**

**是一个对象，一种规范，跟异步操作约定了统一的接口，表示一个异步操作的最终结果以同步的方式来写代码，执行的操作是异步的，但是又保证程序的执行顺序是同步的，**

**promise 对象有三种状态：**

**未完成（pending）**

**已完成（fulfilled）**

**失败（rejected）**

**这三种状态的变化可以从 未完成 到 已完成 || 从 未完成 到 失败 ，过程只能发生一次，并且是不可逆的。**

**更优雅的异步编程**

**http:// HTTP TCP IP**

**https:// HTTP SSL/TLS TCP IP**

**专门用于处理加密访问，搭建服务器时比http多个证书**

**模拟搭建https服务器**

**Buffer：缓冲，在nodejs里用来处理二进制的数据**

**（javascript的字符串是以utf-8的编码存储的，处理二进制的能力很弱，而网络层对于不同资源的请求，响应，文件等都是由二进制的方式进行交互的，所以nodejs有了这么个接口，来创建一个专门存放二进制数据的缓存区，并提供了一些方法来对这些缓存区的数据进行进一步的处理，buffer在nodejs里是可以全局访问的，不需要用require关键字来引入和加载它。）**

**buffer：方法——实例化——实例化后对象上所用的属性和方法**

**命令行：**

**node //输入node来进入nodejs的rep on 模式**

**>Buffer**

**Buffer类是随node内核一起发布的核心库，作为一种存储语言式数据的方法，借助Buffer，Nodejs不仅能处理TCP连接中发送和接收的数据，也能处理图像或压缩文件，读取到的二进制数据，甚至文件系统里面读取到的数据，包括网络中的这种二进制数据流。只要是nodejs里能处理的I/O操作中移动的数据，都有可能用到Buffer。**

> Buffer 是一个构造函数，同时也是一个对象

{ [Function: Buffer]

poolSize: 8192, 内存载体的容量

isBuffer: [Function: isBuffer], 判断这个对象是否是Buffer类型对象

compare: [Function: compare], 判断两个Buffer对象的相对位置 一般来做按字符串排序

isEncoding: [Function], 判断Nodejs是否支持某种编码

concat: [Function], 将几个Buffer对象连接创建为一个新的Buffer对象

byteLength: [Function: byteLength] } 用来获得指定编码下字符串所占的字节数

**Buffer实例方法**

**在js里对数组操作的能力挪到了Buffer里面，用来对二进制数据进行操作。**

**buffer[index]**

**buffer.length**

**buffer.write(string, offset=0, length, encoding=’utf-8’) 写入字符串，位置，长度，编码格式**

**buffer.toString(encoding, start=0, end=buffer.length) 输出字符串编码，开始位置**

**buffer.copy(target, tStart, sStart, sEnd=buffer.length) 复制的目标，**

**buffer.slice(start,end)**

**buffer.compare(otherBuffer)**

**buffer.fill(value, offset, end)**

$ node

> Buffer

{ [Function: Buffer]

poolSize: 8192,

isBuffer: [Function: isBuffer],

compare: [Function: compare],

isEncoding: [Function],

concat: [Function],

byteLength: [Function: byteLength] }

> var buf = new Buffer('Hello MUKEWANG')

undefined

> buf.length

14

> buf.write('Hi MUKEWANG')

11

> buf.toString()

'Hi MUKEWANGANG'

> buf.length

14

> buf.write(' ImoocImoocImooc',2,16)  **写入16个字符 偏移量2**

12 **只写入了12个**

> buf.toString() **打印一下**，**输出字符串**

'Hi ImoocImoocI'

**COPY**

> var buf = new Buffer('Hello Imooc')

undefined

> buf.length

11

> var buf2 = new Buffer(5)

undefined

> buf.copy(buf2)

5

> buf2.toString()

'Hello'

> buf.copy(buf2,0,6,11)  **copy**

5

> buf2.toString()

'Imooc'

**编码**

$ node

> var buf = new Buffer('imooc')

undefined

> var str = buf.toString('base64')

undefined

> str

'aW1vb2M='

> var buf = new Buffer('aW1vb2M=','base64')

undefined

> var str = buf.toString()

undefined

> str

'imooc'

> var buf = new Buffer('aW1vb2M=','base64')

undefined

> var str = buf.toString('hex')

undefined

> str

'696d6f6f63'

> var buf = new Buffer('696d6f6f63','hex')

undefined

> var str = buf.toString('utf8')

undefined

> str

'imooc'

Stream 流 暂存和移动数据，与Buffer结合使用

Readable Writable/Readable Writable

Source —— Transform —— Destination

—— Data flows through pipeline ——

在unix操作系统中，流是一个标准的概念，有标准的输入输出和标准的错误

命令行： cat \*.js | grep http | 是unix管道

//打印出所有的js文件，交给grep过滤出包含http的文件内容

进行很多数据读写 最好边读边写 避免内存爆了 ，借助流来完成

nodejs涉及流的模块

在nodejs里，流是一个抽象接口，实现在node . js中各种对象。流以Buffer形式存在

var fs = require(‘fs’) 是文件系统

var readStrem = fs.createReadStream(‘ ’) //创建可读的流

readStream.on(‘data’, function(chunk){ //data事件 数据传递时触发

console.log(‘data emits’) //确认触发了data事件

console.log(Buffer.isBuffer(chunk)) //验证是否是Buffer对象

console.log(chunk.toString(‘’utf8)) //打印读到的文件的内容 字符串

})

.on(‘readable’, function(){ }) //readable事件，表示这个文件是可读的

.on(‘end’, function(){ }) //end事件，数据传递完成后触发,同时触发可写流的end 事件，目标就变成不再可写了

.on(‘close’,function(){ }) //close事件，整个流传输结束/关闭的时候触发

.on(’error’,function(e){ //error事件，读取过程中发生错误或不存在时触发

console.log(‘data read error’ + e)

})

.pipe

Readable : 可读流，用来提供数据，外部来源的数据会被存储到内部的Buffer 里面缓存起来 两种模式：流动模式：暂停模式：pause

Writable : 可写流，用来消费数据，从可读流获取数据后，对得到的Buffer 数据进行处理

Duplex : 双攻流，它实现了readable和writable的两个接口

Transform :转换流，可读可写，它也实现了readable和writable的两个接 口，只不过它不保存数据，只负责处理流经它的数据