**Node.js**

## 一、nodejs概述

**1 JavaScript 是属于网络的脚本语言！**被数百万计的网页用来改进设计、验证表单、检测浏览器、创建cookies，以及更多的应用。是因特网上最流行的脚本语言。世界上最流行的编程语言。可用于 HTML 和 web，更可广泛用于服务器、PC、笔记本电脑、平板电脑和智能手机等设备。是一种轻量级的编程语言。是可插入 HTML 页面的编程代码。插入 HTML 页面后，可由所有的现代浏览器执行。

**2 简单的说 Node.js 就是运行在服务端的 JavaScript。**Node.js 是一个基于Chrome JavaScript 运行时建立的一个平台。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。

**3 Node.js 是运行在服务端的 JavaScript**，如果你熟悉Javascript，那么你将会很容易的学会Node.js。当然，如果你是后端程序员，想部署一些高性能的服务，那么学习Node.js也是一个非常好的选择

**4 如果我们使用PHP来编写后端的代码**时，需要Apache 或者 Nginx 的HTTP 服务器，并配上 mod\_php5 模块和php-cgi。从这个角度看，整个"接收 HTTP 请求并提供 Web 页面"的需求根本不需 要 PHP 来处理。不过对 Node.js 来说，概念完全不一样了。使用 Node.js 时，我们不仅仅 在实现一个应用，同时还实现了整个 HTTP 服务器。事实上，我们的 Web 应用以及对应的 Web 服务器基本上是一样的。

**5 组成**

**（1）引入 required 模块**：我们可以使用 require 指令来载入 Node.js 模块。

**（2）创建服务器：**服务器可以监听客户端的请求，类似于 Apache 、Nginx 等 HTTP 服务器。

**（3）接收请求与响应请求 服务器很容易创建**，客户端可以使用浏览器或终端发送 HTTP 请求，服务器接收请求后返回响应数据。

**6 事件驱动程序**

Node.js 使用事件驱动模型，当web server接收到请求，就把它关闭然后进行处理，然后去服务下一个web请求。当这个请求完成，它被放回处理队列，当到达队列开头，这个结果被返回给用户。这个模型非常高效可扩展性非常强，因为webserver一直接受请求而不等待任何读写操作。（这也被称之为非阻塞式IO或者事件驱动IO）。在事件驱动模型中，会生成一个主循环来监听事件，当检测到事件时触发回调函数。

（1）Node.js 是单进程单线程应用程序，但是通过事件和回调支持并发，所以性能非常高。

（2）Node.js 的每一个 API 都是异步的，并作为一个独立线程运行，使用异步函数调用，并处理并发。

（3）Node.js 基本上所有的事件机制都是用设计模式中观察者模式实现。

（4）Node.js 单线程类似进入一个while(true)的事件循环，直到没有事件观察者退出，每个异步事件都生成一个事件观察者，如果有事件发生就调用该回调函数。

**7 特点**

（1）它是一个Javascript运行环境

（2）依赖于Chrome V8引擎进行代码解释

（3）事件驱动

（4）非阻塞I/O

（5）轻量、可伸缩，适于实时数据交互应用

（6）单进程，单线程

## 二、知识总结

**1 Windows 安装包(.msi)**

32 位安装包下载地址 :

https://nodejs.org/dist/v4.4.3/node-v4.4.3-x86.msi

64 位安装包下载地址 :

https://nodejs.org/dist/v4.4.3/node-v4.4.3-x64.msi

检测PATH环境变量是否配置了Node.js，点击开始=》运行=》输入"cmd" => 输入命令"path"，输出如下结果：我们可以看到环境变量中已经包含了C:\Program Files\nodejs\

检查Node.js版本》cmd>node --version

**2 编码**

（1）创建服务器

var http = require('http');

http.createServer(function (request, response)

{

// 发送 HTTP 头部

// HTTP 状态值: 200 : OK

// 内容类型: text/plain

response.writeHead(200, {'Content-Type': 'text/plain'});

// 发送响应数据 "Hello World"

response.end('Hello World\n');

}).listen(8888);

// 终端打印如下信息

console.log('Server running at http://127.0.0.1:8888/');

node server.js

Server running at http://127.0.0.1:8888/

打开浏览器访问 http://127.0.0.1:8888/，你会看到一个写着 "Hello World"的网页。

（2）事件编码

// 引入 events 模块

var events = require('events');

// 创建 eventEmitter 对象

var eventEmitter = new events.EventEmitter();

// 创建事件处理程序

var connectHandler = function connected()

{

console.log('连接成功。');

// 触发 data\_received 事件

eventEmitter.emit('data\_received');

}

// 绑定 connection 事件处理程序

eventEmitter.on('connection', connectHandler);

// 使用匿名函数绑定 data\_received 事件

eventEmitter.on('data\_received', function(){

console.log('数据接收成功。');

});

// 触发 connection 事件

eventEmitter.emit('connection');

console.log("程序执行完毕。");

## 三、优点

（1）高并发（最重要的优点）

（2）适合I/O密集型应用

## 四、缺点

（1）不适合CPU密集型应用；CPU密集型应用给Node带来的挑战主要是：由于JavaScript单线程的原因，如果有长时间运行的计算（比如大循环），将会导致CPU时间片不能释放，使得后续I/O无法发起；解决方案：分解大型运算任务为多个小任务，使得运算能够适时释放，不阻塞I/O调用的发起；

（2）只支持单核CPU，不能充分利用CPU

（3）可靠性低，一旦代码某个环节崩溃，整个系统都崩溃（原因：单进程，单线程）。解决方案：（1）Nnigx反向代理，负载均衡，开多个进程，绑定多个端口；开多个进程监听同一个端口，使用cluster模块；

（4）开源组件库质量参差不齐，更新快，向下不兼容

（5）Debug不方便，错误没有stack trace

## 五、使用场景

**1 NodeJS带来的对系统瓶颈的解决方案**

（1）并发连接

举个例子，想象一个场景，我们在银行排队办理业务，我们看看下面两个模型。系统线程模型：服务端只有一个线程，并发请求（用户）到达只能处理一个，其余的要先等待，这就是阻塞，正在享受服务的请求阻塞后面的请求了；

**多线程、线程池模型：**这个模型已经比上一个有所进步，它调节服务端线程的数量来提高对并发请求的接收和响应，但并发量高的时候，请求仍然需要等待，它有个更严重的问题。到代码层面上来讲，我们看看客户端请求与服务端

**通讯的过程：**客户端请求与服务端通讯的过程服务端与客户端每建立一个连接，都要为这个连接分配一套配套的资源，主要体现为系统内存资源，以PHP为例，维护一个连接可能需要20M的内存。这就是为什么一般并发量一大，就需要多开服务器。我们同样是要发起请求，等待服务器端响应；但是与银行例子不同的是，这次我们点完餐后拿到了一个号码，拿到号码，我们往往会在位置上等待，而在我们后面的请求会继续得到处理，同样是拿了一个号码然后到一旁等待，接待员能一直进行处理。等到饭菜做号了，会喊号码，我们拿到了自己的饭菜，进行后续的处理（吃饭）。这个喊号码的动作在NodeJS中叫做回调（Callback），能在事件（烧菜，I/O）处理完成后继续执行后面的逻辑（吃饭），这体现了NodeJS的显著特点，异步机制、事件驱动整个过程没有阻塞新用户的连接（点餐），也不需要维护已经点餐的用户与厨师的连接。基于这样的机制，理论上陆续有用户请求连接，NodeJS都可以进行响应，因此NodeJS能支持比Java、PHP程序更高的并发量虽然维护事件队列也需要成本，再由于NodeJS是单线程，事件队列越长，得到响应的时间就越长，并发量上去还是会力不从心。

（2）I/O阻塞

NodeJS遇到I/O事件会创建一个线程去执行，然后主线程会继续往下执行的，因此，拿profile的动作触发一个I/O事件，马上就会执行拿timeline的动作，两个动作并行执行，假如各需要1S，那么总的时间也就是1S。它们的I/O操作执行完成后，发射一个事件，profile和timeline，事件代理接收后继续往下执行后面的逻辑，这就是NodeJS非阻塞I/O的特点。

**2 RESTful API**

这是NodeJS最理想的应用场景，可以处理数万条连接，本身没有太多的逻辑，只需要请求API，组织数据进行返回即可。它本质上只是从某个数据库中查找一些值并将它们组成一个响应。由于响应是少量文本，入站请求也是少量的文本，因此流量不高，一台机器甚至也可以处理最繁忙的公司的API需求。

**3. 统一Web应用的UI层**

目前MVC的架构，在某种意义上来说，Web开发有两个UI层，一个是在浏览器里面我们最终看到的，另一个在server端，负责生成和拼接页面。不讨论这种架构是好是坏，但是有另外一种实践，面向服务的架构，更好的做前后端的依赖分离。如果所有的关键业务逻辑都封装成REST调用，就意味着在上层只需要考虑如何用这些REST接口构建具体的应用。那些后端程序员们根本不操心具体数据是如何从一个页面传递到另一个页面的，他们也不用管用户数据更新是通过Ajax异步获取的还是通过刷新页面。

**4.大量Ajax请求的应用**

例如个性化应用，每个用户看到的页面都不一样，缓存失效，需要在页面加载的时候发起Ajax请求，NodeJS能响应大量的并发请求。

**总而言之，NodeJS适合运用在高并发、I/O密集、少量业务逻辑的场景。**

## 六、前后端分离

**1 为什么选择前后端分离**

在以前传统的网站开发中，前端一般扮演的只是切图的工作，只是简单地将UI设计师提供的原型图实现成静态的HTML页面，而具体的页面交互逻辑，比如与后台的数据交互工作等，可能都是由后台的开发人员来实现的，或者是前端是紧紧的耦合后台。比如，以前淘宝的Web基本上都是基于MVC框架webx，架构决定了前端只能依赖后端。所以他们的开发模式依然是，前端写好静态demo，后端翻译成VM模版，这种模式的问题就不说了，被吐槽了很久。

而且更有可能后台人员直接兼顾前端的工作，一边实现API接口，一边开发页面，两者互相切换着做，而且根据不同的url动态拼接页面，这也导致后台的开发压力大大增加。前后端工作分配不均。不仅仅开发效率慢，而且代码难以维护。而前后端分离的话，则可以很好的解决前后端分工不均的问题，将更多的交互逻辑分配给前端来处理，而后端则可以专注于其本职工作，比如提供API接口，进行权限控制以及进行运算工作。而前端开发人员则可以利用nodejs来搭建自己的本地服务器，直接在本地开发，然后通过一些插件来将api请求转发到后台，这样就可以完全模拟线上的场景，并且与后台解耦。前端可以独立完成与用户交互的整一个过程，两者都可以同时开工，不互相依赖，开发效率更快，而且分工比较均衡。

**2 如何做到前后端分离**

(以下的内容都是基于我们的电影购票网站来讨论的)

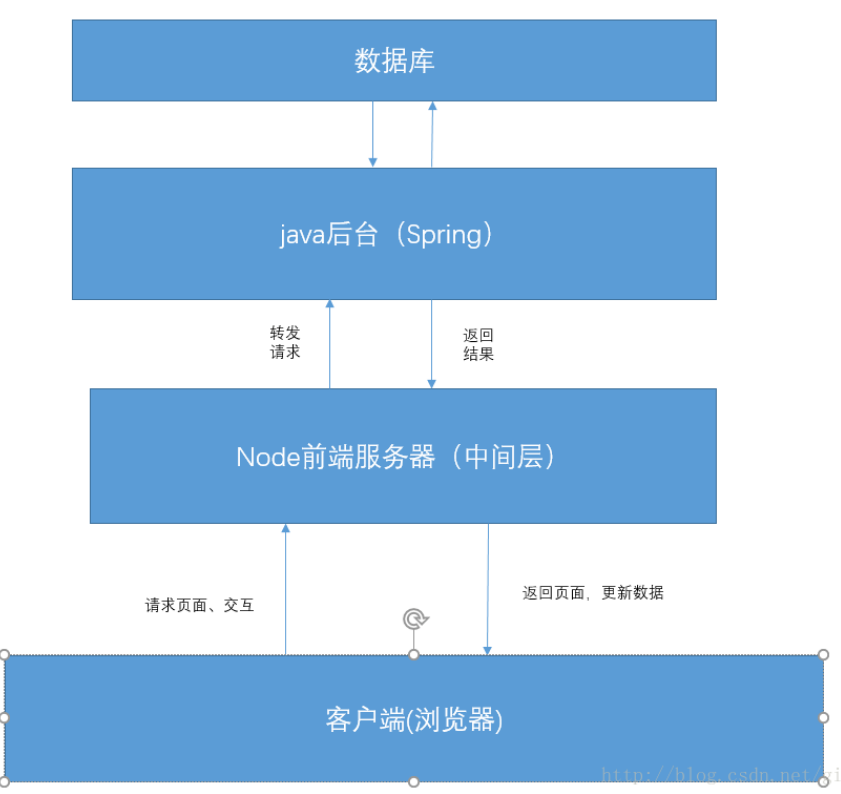
前端的技术框架是: vue全家桶+nodejs+express(实现的是单页面(SPA)应用)

首先，先分清楚前后端的工作

**前端的工作：实现整一个前端页面以及交互逻辑，以及利用ajax与nodejs服务器（中间层)交互**

**后端的工作：提供API接口，利用redis来管理session,与数据库交互**

**我们项目的整一个架构如下:**



（1）一般来说，要实现前后端分离，前端就需要开启一个本地的服务器来运行自己的前端代码，以此来模拟真实的线上环境，并且，也是为了更好的开发。因为你在实际开发中，你不可能要求每一个前端都去搭建一个java(php)环境，并且在java环境下开发，这对于前端来说，学习成本太高了。但如果本地没有开启服务器的话，不仅无法模拟线上的环境，而且还面临到了跨域的问题，因为你如果写静态的html页面，直接在文件目录下打开的话，你是无法发出ajax请求的(浏览器跨域的限制),因此，你需要在本地运行一个服务器，可是又不想搭建陌生而庞大的java环境，怎么办法呢？nodejs正好解决了这个问题。在我们项目中，我们利用nodejs的express框架来开启一个本地的服务器，然后利用nodejs的一个http-proxy-middleware插件将客户端发往nodejs的请求转发给真正的服务器，让nodejs作为一个中间层。这样，前端就可以无忧无虑的开发了

（2）由于前后端分离后，前端和后台同时开发时，就可能遇到前端已经开发好一个页面了，可是却等待后台API接口的情况。比如说A是负责前端，B是负责后台，A可能用了一周做好了基本的结构，并且需要API接口联调后，才能继续开发，而此时B却还没有实现好所需要的接口，这种情况，怎么办呢？在我们这个项目里，我们是通过了mock来提供一些假数据，我们先规定好了API接口，设计出了一套API文档，然后我们就可以通过API文档，利用mock(http://mockjs.com)来返回一些假数据，这样就可以模拟发送API到接受响应的整一个过程，因此前端也不需要依赖于后端开发了，可以独立开发，等到后台的API全部设计完之后，就可以比较快速的联调。

**3 为什么要引入nodejs作为中间层**

前面的我发的项目结构图中，已经表明，在这个项目里，我们将nodejs作为中间层，那么，为什么我们要特地引入nodejs呢？直接用java做不就行了吗？

我觉得引入nodejs主要是为了分层开发，职责划分，nodejs作为前端服务器，由前端开发人员负责，前端开发人员不需要知道java后台是如何实现的，也不需要知道API接口是如何实现的，我们只需要关心我们前端的开发工作，并且管理好nodejs前端服务器，而后台开发人员也不需要考虑如何前端是如何部署的，他只需要做好自己擅长的部分，提供好API接口就可以；

nodejs本身有着独特的异步、非阻塞I/O的特点，这也就意味着他特别适合I/O密集型操作，在处理并发量比较大的请求上能力比较强，因此，利用它来充当前端服务器，向客户端提供静态文件以及响应客户端的请求，我觉得这是一个很不错的选择。

**4 前端服务器如何部署**

（1）nodejs前端服务器的职责

作为静态文件服务器，当用户访问网站的时候，将index.html以及其引入的js、css、fonts以及图片返回给用户

负责将客户端发来的ajax请求转发给后台服务器

（2）其实前端服务器的部署工作是算比较简单的，具体有以下两个点:

将开发完的前端代码，利用webpack打包成静态压缩文件

在服务器上，利用pm2负载均衡器来执行以下的代码来开启服务器:



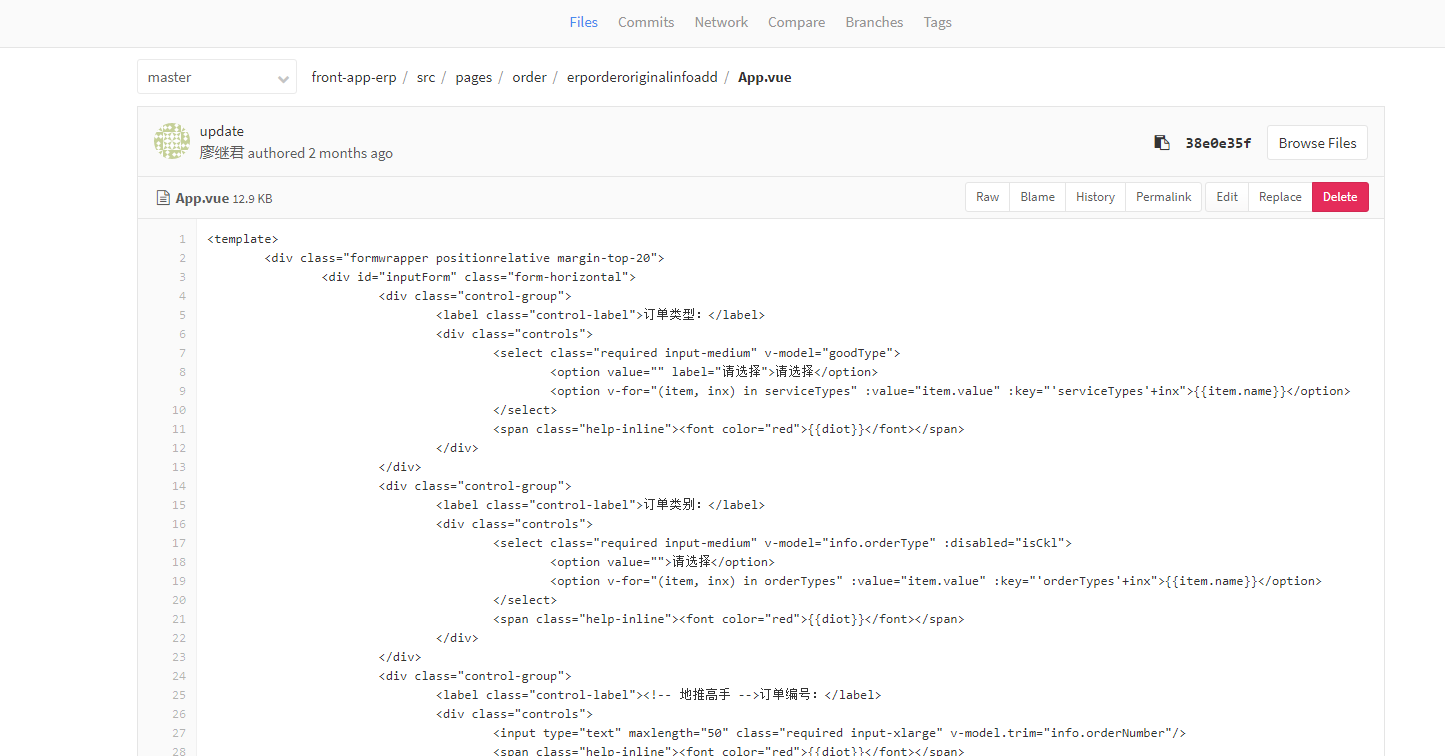
**5 webpack和前后端分离**

# nodejs+webpack+vuejs

Webpack有一个前端工程，把源码写入到vue, 用命令打包成min.js，拷贝到项目中。

Webpack封装了nodejs,前端开发人员启动nodejs,nodejs配置了后台的接口，可以mock一些数据，提高开发效率。不用启动前端tomcat服务器（太笨重）。

front-app-erp/src/pages/order/ erporderoriginalinfoad/App.vue



6 **前后端分离的session问题**

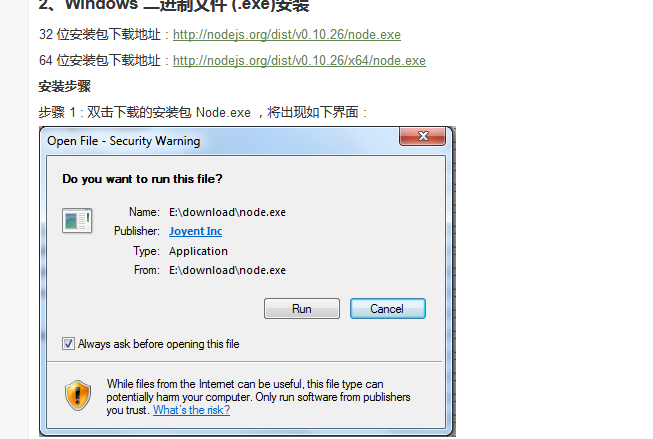
因为前后端分离后，前后端可能会部署在不同的服务器上面，会跨域，前端每次请求时，都会是一次新的请求，所以前端需要记住一个用户的标识，每次请求数据都传给后端。后端才能知道是哪个用户。

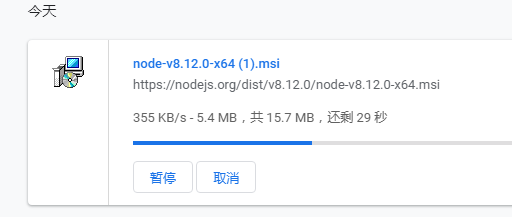
以上代码注释掉的部分是返回登录的token信息，之后每次前端请求的时候，参数都需要带着这个token。   
放开的代码片段是登录后，返回shiro的sessionid，前端在cookie里保存一下这个sessionid，每次请求的时候参数中都需要加上sessionid。   
两种方式都可以使用，但是本人推荐返回sessionid 的方式，因为前端给了sessionid 之后，我们只需要在服务中通过sessionid 获取shiro的session信息，然后就能获取到shiro对应的用户信息。而使用token的这种方式的话，每次前端请求过来，后端都需要根据这个token再登录一下。两者的区别就在这里

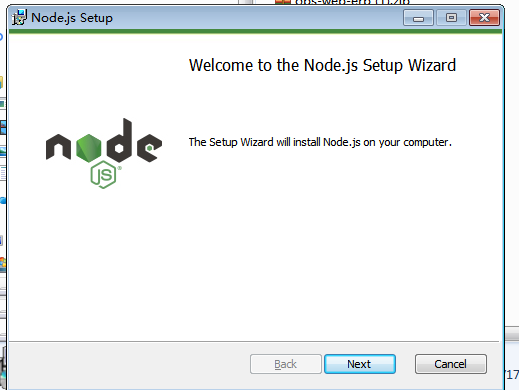
**7 nodejs 安装**

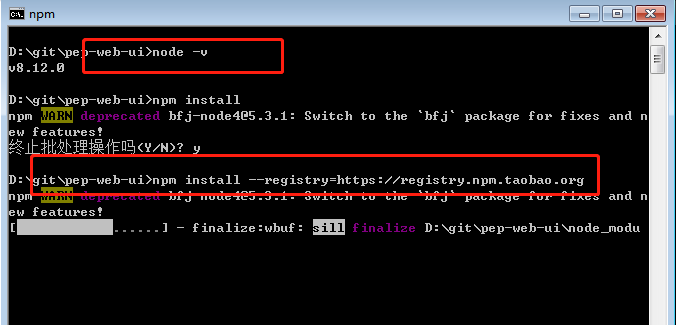
<http://nodejs.org/dist/v0.10.26/x64/node.exe>

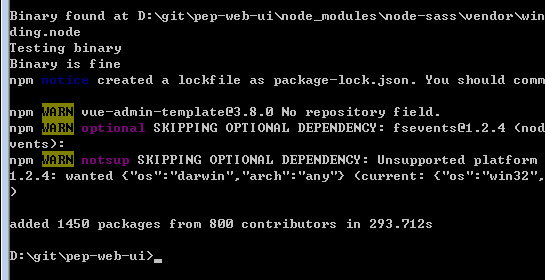








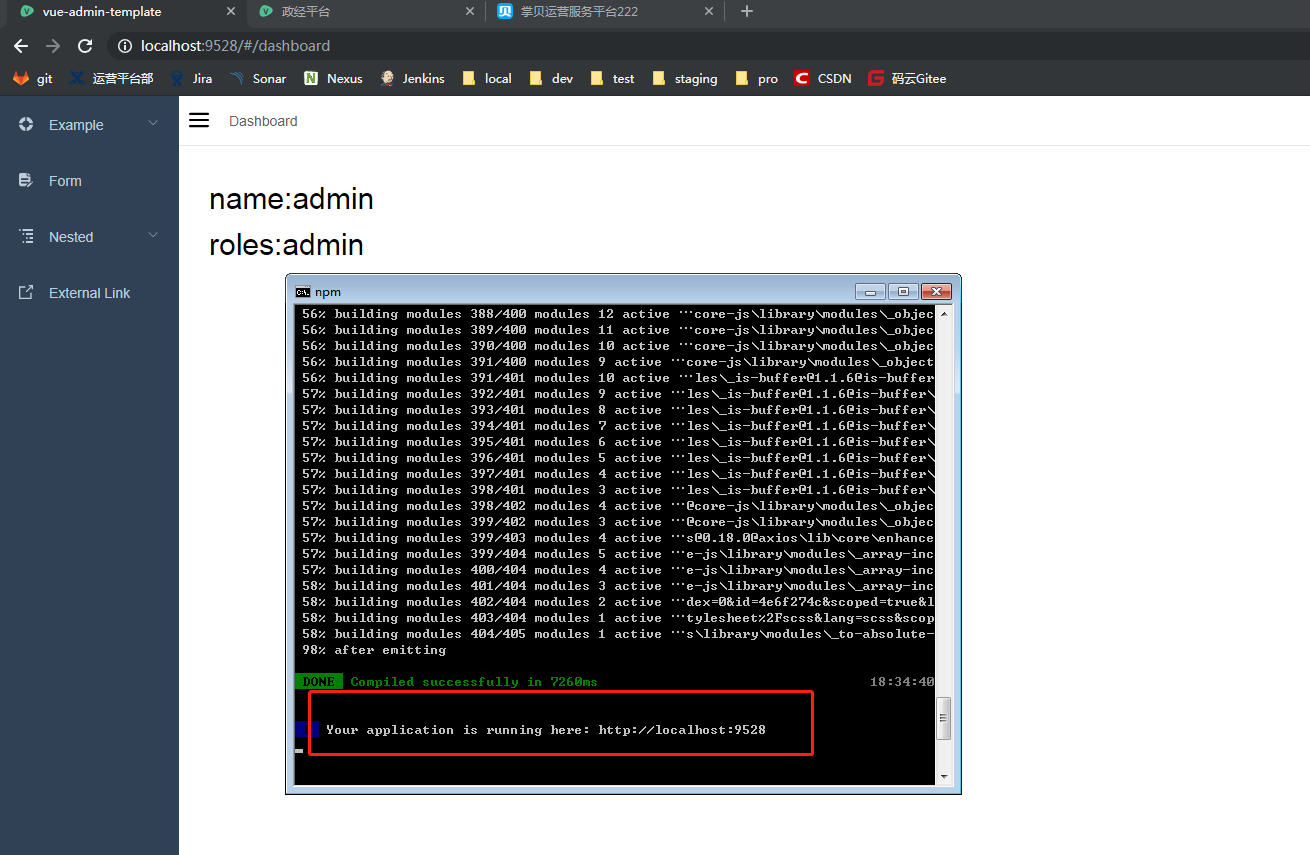




**npm install -g cnpm --registry=https://registry.npm.taobao.org**

**cnpm install**

**npm run dev**

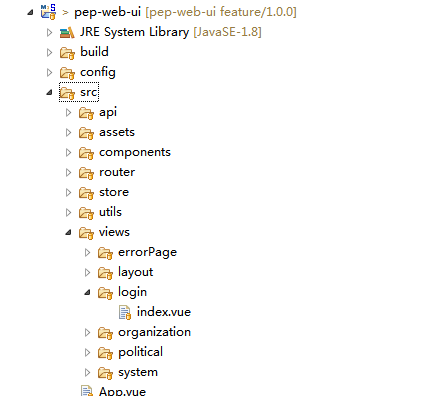


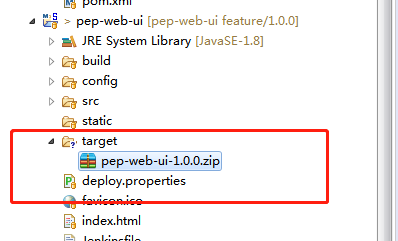
**Nodejs 作为本地开发环境的静态资源服务器； 到测试环境和生产环境用的静态资源服务器是nginx。**

# nodejs、npm、webpack概念解释

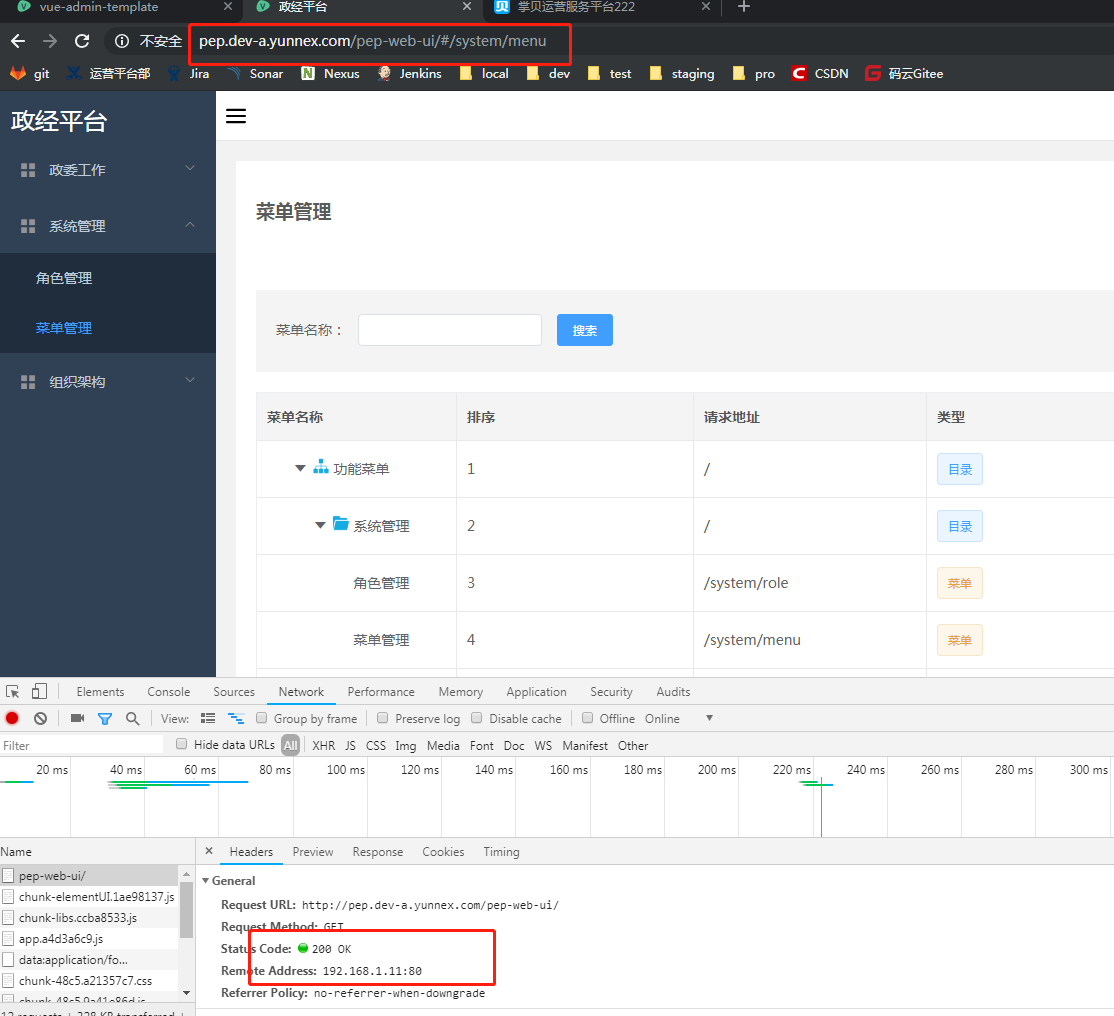
1. **nodejs是js后端运行平台，可以把它看成java体系中对应的jdk，是三个里面最基础的。**
2. **npm是nodejs的包管理工具，可以把它看成maven中包依赖管理那部分。**
3. **webpack是前端工程化打包工具，可以把它看成maven中工程自动化构建那部分。**

**8 webpack + nodejs+ vue.js**





测试环境 部署到nginx



**前台编程及运行环境描述： 使用nginx服务器运行前台代码，使用npm+webpack对前台代码进行管理及编译。**

**1. 安装Nodejs**

目的：安装之后就可以用其自带的npm进行工具引用管理。

安装原理：下载官网编译好的文件：node-v6.10.0-linux-x64.tar.xz， 对其进行解压，因为是二进制文件，因此可以直接用命令使用，但是需要设置全局变量。

安装步骤：

step 0. 下载文件

进入官网： https://nodejs.org/en/download/ 选择Linux Binaries (x86/x64)的64位版本进行下载；

将下载文件拷贝至/opt文件夹下

step 1. 解压tar.xz文件

sudo xz -d node-v6.10.0-linux-x64.tar.xz

sudo tar -xvf node-v6.10.0-linux-x64.tar

step 2. 配置全局变量（实际上是把链接放在/usr的文件夹下）

ln -s /opt/node-v6.10.0-linux-x64/bin/node /usr/local/bin/node

ln -s /opt/node-v6.10.0-linux-x64/bin/npm /usr/local/bin/npm

step 3. 测试

可以直接在命令行输入node -v，打印出版本即说明安装成功

step 4. 修改文件夹权限

如果不修改文件夹权限的话，后面全局安装的过程会出现权限问题。

sudo chmod -R 777 /opt/node-v6.10.0-linux-x64

**2. 安装webpack**

目的： 使用webpack对项目进行打包管理，及时编译前台。

安装原理： 直接使用npm进行全局安装，并设置全局变量。

安装步骤：

step 1. 下载安装全局webpcak

npm install webpack -g

step 2. 配置全局变量

如果不配置的话，会出现“command not found: webpack”的问题（困扰我好几天……）

ln -s /opt/node-v6.10.0-linux-x64/bin/webpcak /usr/local/bin/webpcak

安装成功nodejs以及webpack之后，就可以对项目进行编辑查看了。

**3. 安装ngnix服务器**

目的：因为需要cookie等功能，因此需使用服务器来运行前台代码，

安装原理：下载安装，并且修改配置文件。

安装步骤：

step 1: 下载安装

sudo apt-get update

sudo apt-get install nginx

step 2: 配置参数

配置：在http里面添加：

server {

listen 8000;

server\_name localhost;

location / {

**root /home/hadoop/Projects/code-clone-frontend;**

index index.html;

}

}

**其中：/home/hadoop/Projects/code-clone-frontend为项目的工作目录，8000为端口号。**

