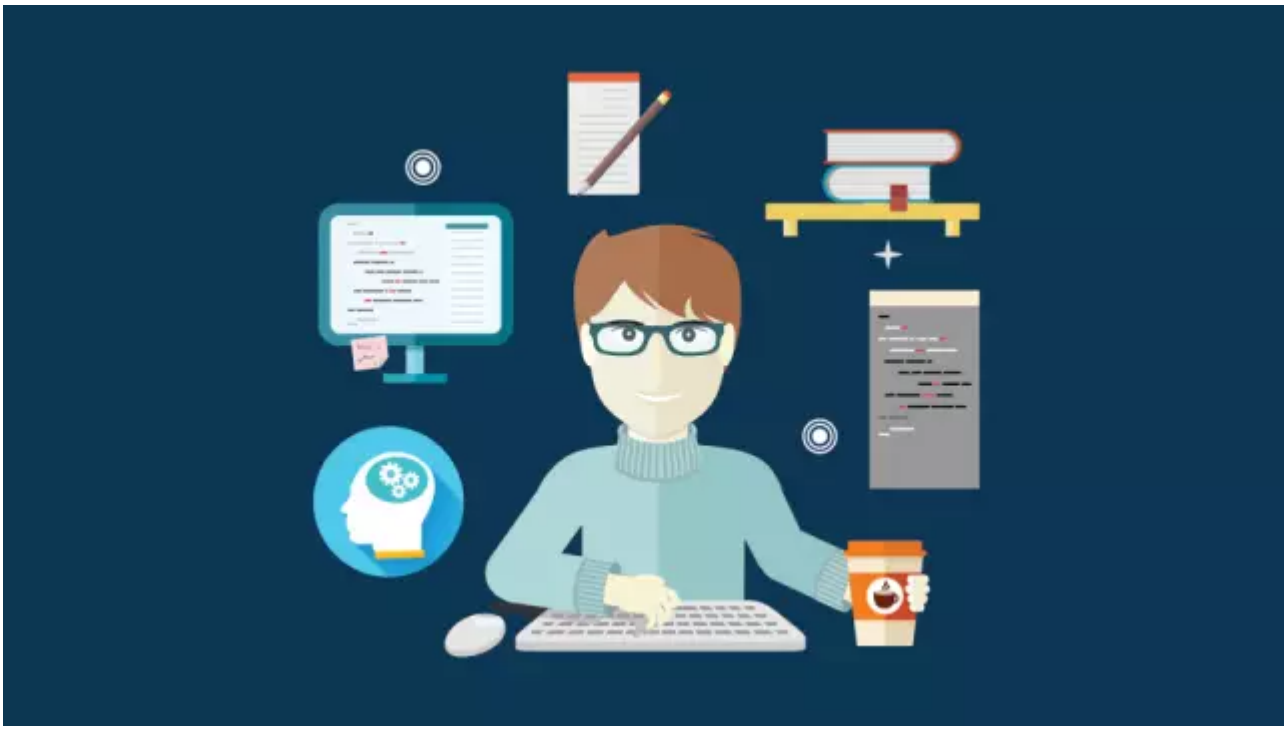
Vue.js+Node.js 入门实战开发

原创 2017-08-16 赵梦欢 GitChat



作者/分享人：赵梦欢，目前就任于北京赞同科技研究院，参与赞同移动云生态产品研发，目前在写《Vue.js+Node.js全栈开发》一书。

前言

随着前端技术的快速迭代，JavaScript 的关注度越来越高，相关的生态链和前后端分离的解决方案愈趋完善，JavaScript 发挥着越来越重要的作用。Vue.js 的火热程度丝毫不亚于曾经的 jQuery，相关的生态也发展得越来越好。Node.js 经过这么多年的发展也越来越强大，虽然 Node.js 在短期内不会取代基于 Java 、PHP 等传统的后端语言，然而 Node.js 已经被越来越多的人用于项目开发中。基于这种考虑，数月前我打算搞点事情，尝试做一个 Vue.js + Node.js 全栈开发社区，专注于探索和分享 Vue.js 和 Node.js 开发的相关技术。之前申请的 vuenode.com 域名备案上个月通过了，于是开始实施了，本次 Chat 作为一个入门级别的总结，将以构建一个简单的技术博客系统为例，说明 Vue.js 结合 Node.js 实战的例子帮助前端新人进阶学习，掌握后端必备的一些基础技能。本文的主要内容包括：

- Vue.js / vue-router / vuex 基本用法；
- Vue 的 PWA的解决方案 Lavas 介绍；
- Node.js 框架 Egg.js 的基本用法；
- Linux 云服务器搭建部署 Node 服务。

Vue.js / vue-router / vuex 基本用法

Vue.js 的核心特征

学习一种新技术我们首先需要抓住主要内容，其他的工作无非是一些需要反复去研读文档去实践验证的细节。学习 Vue.js 我不建议新手直接用 Vue Cli 跑单文件组件工程，虽然这样可以很快写出一个简单项目的原型，但是通过简单的demo学习，更能够理解问题的本质。

响应式系统

在上一期的讲座中我们学习了Vue.js数据双向绑定的实现原理，其主要特性是一种基于数据驱动的思想构成的响应式系统，模型层(model)只是普通 JavaScript 对象，修改它则更新视图(view)。这是需要去重点关注的第一个重点，与之相关的知识点是**生命周期**、**模板语法**（**插值**、**指令**、**过滤器**）等。这部分的内容基本上是需要我们对于数据绑定相关的内容有一定的理解。

看到很多人在写 Vue.js 的时候依然还是之前DOM操作的一些思想，先入为主的一定要拿到需要控制的对象进行操作，而不是想着我们的数据模型是响应式的，通过改变数据模型，视图会自动变动。改变思想才能更好的利用工具，以数据和状态为中心，在某些场合下可能比对象事件更简单。

这部分的内容基本上以文档为主，本文不作赘述。

组件化系统

用官方文档的话说就是，组件（Component）是 Vue.js 最强大的功能之一，组件可以扩展 HTML 元素，封装可重用的代码，在较高层面上，组件是自定义元素， Vue.js 的编译器为它添加特殊功能，在有些情况下，组件也可以是原生 HTML 元素的形式，以 is 特性扩展。

要想理解组件化的意义，我觉得有必要了解一下 Web Components 标准，Web Components 由这四种技术组成：

- 自定义元素 (Custom Elements)
- HTML模板 (HTML Templates)
- 影子DOM (Shadow DOM)
- HTML导入 (HTML Imports)

为了深入理解 Vue.js 背后设计的思想，我们先通过一个自定义 Dialog 的实例看看 Web Components 如何使用。我们想实现通过调用自定义元素 `<web-components-dialog></<web-components-dialog>` 的形式调用我们自定义的dialog组件。

web 开发中需要解决的经典问题——封装。如何保护组件的样式不被外部 css 样式侵入，如何保护组件的 dom 结构不被页面的其他 javascript 脚本修改。

那么如何实现呢？首先我们需要理解几个基本概念。

Shadow DOM

Shadow DOM 很好的解决了这个问题，我们通过 Element.createShadowRoot()

为自定义元素创造一个"隔离区间"，浏览器渲染显示结果如下图：

显示 shadow dom 需要开启 Chrome 开发者工具的 'Show user agent shadow DOM'：

HTML Templates

以前我们写 html 模板一般会写在 `script` 元素或者 `textarea` 元素中，比如原来的模板形式：

```
<script type="text/template">
    <div>
        this is your template content.
    </div>
</script>
```

而现在我们可以直接使用语义化更好的 `template` 元素，我们可以通过 content 属性访问 template 标签的内容。

```
<template id="dialog-template">
  <style>
    /* web-components-dialog style */
  </style>

  <div class="web-components-dialog-mask">
    <div class="web-components-dialog-wrapper">
      <div class="web-components-dialog">
        <div class="web-components-dialog-title"></div>
        <div class="web-components-dialog-content"></div>
        <div class="web-components-dialog-btns"></div>
      </div>
    </div>
  </div>
</template>
```

读取模板内容：

```
var template = document.querySelector('#dialog-template').content;
```

template 元素主要有四个特性：

- 惰性：在使用前不会被渲染；
- 无副作用：在使用前，模板内部的各种脚本不会运行、图像不会加载等；
- 内容不可见：模板的内容不存在于文档中，使用选择器无法获取；
- 可被放置于任意位置：即使是 HTML 解析器不允许出现的位置，例如作为 `<select>` 的子元素。

Custom Elements

自定义元素允许开发者定义新的 HTML 元素类型，带来以下特性：

- 定义新元素
- 元素继承

- 扩展原生 DOM 元素的 API

如果使用非标准的自定义元素，会有什么结果?

```
<style>
  mydialog {
    color: red;
  }
</style>
<mydialog>hello world!</mydialog>
```

我们会发现浏览器没有过滤这个元素，浏览器对待自定义元素，就像对待标准元素一样，只是没有默认的样式和行为，这种处理方式是写入HTML5的。

自定义元素有几种形式：

- 使用非标准元素， HTMLUnknownElement接口的实例，例如标签；
- 使用标准的自定义元素，通过Document.registerElement() 方法，此方法向浏览器注册一个新元素，该元素默认使用 HTMLElement 接口；
- 使用Custom Elements 标准通过customElements.define定义一个Custom Elements，自定义元素的名字必须包含一个破折号（-），自定义元素的定义可以使用 ES6 的class语法；
- 可以在例如 `<button>` 这样的原生元素的基础上创建自定义元素，不过如此一来就不能使用自定义标签名，比如 `<my-button>` ，而要使用 `<button is="my-button">` 这样的语法。

自定义元素可以使用以下生命周期回调函数：

- createdCallback – 注册元素时执行
- attachedCallback – 元素插入DOM时执行
- detachedCallback – 元素被移除DOM时执行
- attributeChangedCallback – 元素的属性被增、删、改时执行

HTML Imports

通过标签来引入 HTML 文件，使得我们可以用不同的物理文件来组织代码。

```
<link rel="import" href="http://example.com/component.html" >
```

注意：受浏览器同源策略限制，跨域资源的 import 需要服务器端开启 CORS。

通过import引入的 HTML 文件是一个包含了 html, css, javascript 的独立 component。

```
<template>
    <style>
        .coloured {
            color: red;
        }
    </style>
    <p>My favorite colour is: <strong class="coloured">Red</strong></p>
</template>
<script>
    (function() {
        var element = Object.create(HTMLElement.prototype);
        var template = document.currentScript.ownerDocument.querySelector('templ
        element.createdCallback = function() {
            var shadowRoot = this.createShadowRoot();
            var clone = document.importNode(template, true);
            shadowRoot.appendChild(clone);
        };
        document.registerElement('favorite-colour', {
            prototype: element
        });
    }());
</script>
```

下面我们完成 Dialog 的例子，我们要在浏览器注册我们的自定义元素 `web-components-dialog` ：

```
var template = document.querySelector('#dialog-template').content;
class CustomDialog extends HTMLElement {
  createdCallback() {
    // 拷贝template
    var clone = document.importNode(template, true);
    // 设置标题
    var titleElem = clone.querySelector('.web-components-dialog-title');
    titleElem.textContent = this.getAttribute('data-title');
    titleElem.style.cssText = this.getAttribute('data-title-style');
    // 设置内容
    var contentElem = clone.querySelector('.web-components-dialog-content');
```

```
    contentElem.textContent = this.getAttribute('data-content');
    contentElem.style.cssText = this.getAttribute('data-content-style');
    // 设置按钮
    var fragment = document.createDocumentFragment();
    var btns = this.getAttribute('data-btns').split(',');
    var callback = this.getAttribute('data-callback');
    for (var i = 0; i < btns.length; i++) {
      var btn = document.createElement("button");
      btn.innerText = btns[i];
      btn.index = i;
      btn.addEventListener('click', function() {
        (new Function("index", callback))(this.index);
      });
      fragment.appendChild(btn);
    }
    clone.querySelector('.web-components-dialog-btns').appendChild(fragment);
    // Create a Shadow Root
    this.createShadowRoot().appendChild(clone);
  }
}
document.registerElement('web-components-dialog', CustomDialog);
```

然后就可以这样调用了：

```
<web-components-dialog
  id="dialog"
  data-title="我是标题"
  data-title-style="background-color: rgb(141, 206, 22); color: rgb(255, 255, 25
  data-content="我是内容"
  data-content-style="color: #f00"
  data-btns="取消, 确认"
  data-callback="alert(index);">
</web-components-dialog>
```

我们可以使用 Vue.js 的组件来实现这个例子：

```
Vue.component('vue-dialog', {
  template: '\
    <transition name="vue-dialog">\
      <div class="vue-dialog-mask" v-if="show">\
```

```
        <div class="vue-dialog-wrapper">\
          <div class="vue-dialog">\
            <div v-if="!!options.title" class="vue-dialog-title" v-bind:style="c
              <h5>{{options.title.text}}</h5>\
              <i class="vue-icon" @click="cancel">×</i>\
            </div>\
            <div v-if="!!options.content" class="vue-dialog-content" v-bind:styl
            <div v-if="!!options.btns" class="vue-dialog-btns">\
              <button v-for="(item, index) in options.btns"  @click="callback(in
            </div>\
          </div>\
        </div>\
      </div>\
    </transition>\
  ',
  props: ['show', 'options', 'callback'],
  methods: {
    cancel: function() {
      this.$emit('update:show', false);
    }
  }
});
```

然后就可以这样使用：

```
<vue-dialog
  :show.sync="showModal"
  :options="optionsObj"
  :callback="modelCallback">
</vue-dialog>
```

看到这里我们发现 Vue.js 组件的思想和 web components 很类似，我们首先通过 Vue.component 或 components注册一个组件，然后可以在Vue实例或组件实例中使用，将整个应用程序划分为组件，以使开发可管理。

```
<div id="app">
  <app-nav></app-nav>
  <app-view>
    <app-sidebar></app-sidebar>
    <app-content></app-content>
```

```
  </app-view>
</div>
```

组件化最大的好处就是实现了分治，组件之间相互隔离，互不干扰，有利于我们抽象和复用。

> Polymer 是另一个由谷歌赞助的项目，事实上也是 Vue 的一个灵感来源。Vue 的组件可以粗略的类比于 Polymer 的自定义元素，并且两者具有相似的开发风格。最大的不同之处在于，Polymer 是基于最新版的 Web Components 标准之上，并且需要重量级的 polyfills 来帮助工作（性能下降），浏览器本身并不支持这些功能。相比而言，Vue 在支持到 IE9 的情况下并不需要依赖 polyfills 来工作。—— 对比其他框架#Polymer

这是 Vue.js 官方文档的一段话，在 web components 规范普遍普及之前，Vue.js 这种框架或许是 web 组件化更好的解决方案。这里花很大的篇幅说明 web components 组件化的实现，更多的是希望与 Vue.js 进行对比，去思考其中相同的地方，或许在技术实现上有很大的差异，但是在思想上是有很多共性的地方的。

另外 Vue.js 的单文件组件也是一个亮点，通过 vue-loader 将 Vue 组件转换为 JavaScript 模块，不过本质上仍然是组件的思想，只是形式上更加友好，配合预处理器（如 Pug，Babel，和 Stylus）可以构建简洁和功能更丰富的组件。这部分的学习我觉得读者有必要自己从零开始构建一个工程，或许自己的配置不够完备，但是对于理解整个过程，熟悉预处理很有必要，在此不做赘述，感兴趣的可以看看我之前的一篇文章：从0到1搭建webpack2+vue2自定义模板详细教程。

vue-router

不知从何时起，各种前端框架都会默认给一个官方 router 的解决方案，都会提供一个脚手架跑一个装着各种 loader 的工程。比如 Vue 社区，很多人一上去直接利用 Vue Cli 跑 Vue 全家桶，不管什么类型的项目，搞得不用 vue-router 都不是正确的做法一样，我们真的明白 vue-router 的场景吗？我们是否真的理解 vue-router？我们是否真的需要 vue-router ？

Vue 官方文档上给出了一个从零开始简单的路由，这是一个新手很容易忽略的例子，但是如果仔细研究一下，我们会发现这个例子很有价值。

```
const Home = { template: '<main-layout><p>home page</p></main-layout>' }
const About = { template: '<main-layout><p>about page</p></main-layout>' }
const NotFound = { template: '<main-layout><p>Page not found</p></main-layout>'

const routes = {
  '/': Home,
  '/about': About
}
```

```
Vue.component('v-link', {
  template: `
    <a
      v-bind:href="href"
      v-bind:class="{ active: isActive }"
      v-on:click="go">
      <slot></slot>
    </a>
  `,
  props: {
    href: {
      type:String,
      required: true
    }
  },
  computed: {
    isActive () {
      return this.href === this.$root.currentRoute
    }
  },
  methods: {
    go (event) {
      event.preventDefault()
      this.$root.currentRoute = this.href
      window.history.pushState(
        null,
        routes[this.href],
        this.href
      )
    }
  }
})

Vue.component('main-layout', {
  template: `
    <div>
      <v-link href="/">Home</v-link>
      <v-link href="/about">About</v-link>
      <slot></slot>
    </div>
  `
})
```

```
new Vue({
  el: '#app',
  data: {
    currentRoute: window.location.pathname
  },
  computed: {
    ViewComponent () {
      return routes[this.currentRoute] || NotFound
    }
  },
  render (h) { return h(this.ViewComponent) }
})

window.addEventListener('popstate', () => {
  app.currentRoute = window.location.pathname
})
```

这里利用 HTML5 History API 提供了对浏览器历史的访问，通过 `window.history.pushState` 创建新的历史记录条目，当用户导航到新的状态，popstate事件就会被触发，通过监听 `popstate` 事件进行路由监听从而切换视图。对于一些较为简单的业务场景，这也不失为一种很好的处理方法。其实 vue-router history 模式底层实现也是基于 HTML5 History API，无非是做了更加完备的处理，封装了更加方便使用的 API 。

路由是单页应用的核心，而结合组件，路由变得更加方便灵活。路由的切换本质上是组件动态切换，理解到这一点我们很容易发现路由和组件有紧密的联系。

- 路由切换动画效果实际上是动态组件的过渡状态；
- vue-router 的 组件是通过渲染成 A 标签进行导航，如果要在组件内进行编程式导航我们可以通过 `this.$router.push('...')` 实现；
- 路由懒加载是通过结合 Vue 的 异步组件 和 Webpack 的 code splitting feature 实现。

vuex

前面我们说明了如何构建一个组件，这部分我们重点说说组件之间数据如何共享，这是开发中必须要学习的内容。vue.js 组件实例的作用域是孤立的，这意味着不能并且不应该在子组件的模板内直接引用父组件的数据，可以使用 props 把数据传给子组件。另外vue.js中提倡单向数据流：当父组件的属性变化时，将传导给子组件，但是不会反过来。这是为了防止子组件无意修改了父组件的状态——这会让应用

的数据流难以理解。当子组件需要更新父组件的状态时，我们可以通过事件触发。首先我们看一个例子：

```
<div id="app">
  {{ message }}
  <button v-on:click="parentAtion">parentAtion</button>
  <child v-bind:param="message" v-on:childfn="fromChild"></child>
</div>

<script type="text/javascript">
  Vue.component('child', {
    template: `
      <div>
        <span>{{param}}</span>
        <button v-on:click="childAtion">childAtion</button>
      </div>
    `,
    props: ['param'],
    methods: {
      childAtion: function () {
        // 触发事件
        this.$emit('childfn', 'child component');
      }
    }
  });

  var app = new Vue({
    el: '#app',
    data: {
      message: 'Hello vuejs!'
    },
    methods: {
      parentAtion: function () {
        this.message = 'parent component';
      },
      fromChild: function (msg) {
        this.message = msg;
      }
    }
  })
</script>
```

每个 Vue 实例都实现了事件接口(Events interface)，即：

- 使用 $on(eventName) 监听事件
- 使用 $emit(eventName) 触发事件

Vue的事件系统分离自浏览器的EventTarget API。尽管它们的运行类似，但是 `$on` 和 `$emit` 不是addEventListener和 dispatchEvent的别名。另外，父组件可以在使用子组件的地方直接用 v-on 来监听子组件触发的事件，不能用 `$on` 侦听子组件抛出的事件，而必须在模板里直接用 v-on 绑定。`vm.$on( event, callback )` 监听当前实例上的自定义事件。

父子组件可以通过 **Prop** 和**事件接口**进行通信传递数据，对于非父子组件之间的通信，在简单的场景下，使用一个空的 Vue 实例作为中央事件总线：

```
var bus = new Vue()
// 触发组件 A 中的事件
bus.$emit('id-selected', 1)
// 在组件 B 创建的钩子中监听事件
bus.$on('id-selected', function (id) {
  // ...
})
```

在一般的情况下这些就足够我们完成组件之间的通信，在相对较为复杂的情况下我们才考虑引入 vuex 这种专门用作状态管理的库。

Vue 的 PWA 的解决方案 —— Lavas

vue 的生态如今还算比较完备，类似于Atwood定律（“任何可以使用JavaScript来编写的应用，最终会由JavaScript编写。”）一样，能够被js写的都会拿 Vue 来写一套，Vue 生态越来越多组件库和解决方案：

- 原生应用跨平台开发方案 —— weex
- 服务端渲染 —— vue-server-renderer / Nuxt.js
- PWA 解决方案(开发工具) —— Lavas

原生应用和服务端渲染相对来说内容比较多，本文不便于展开讲解，后续再做探讨。这里就简单聊聊 PWA 以及 Lavas 工具。

Progressive Web App, 简称 PWA，是提升 Web App 的体验的一种新方法，能给用户原生应用的体验。PWA 能做到原生应用的体验不是靠特指某一项技术，而是经过应用一些新技术进行改进，在安

全、性能和体验三个方面都有很大提升，PWA 本质上是 Web App，借助一些新技术也具备了 Native App 的一些特性，兼具 Web App 和 Native App 的优点。PWA 的主要特点包括下面三点：

- 可靠 – 即使在不稳定的网络环境下，也能瞬间加载并展现
- 体验 – 快速响应，并且有平滑的动画响应用户的操作
- 粘性 – 像设备上的原生应用，具有沉浸式的用户体验，用户可以添加到桌面

PWA 本身强调渐进式，并不要求一次性达到安全、性能和体验上的所有要求，开发者可以通过 PWA Checklist 查看现有的特征：

- 使用 HTTPS
- 用户界面响应式
- 将站点添加至首屏
- Service Worker离线和缓存
- Push Notification
- App Shell 模型
- 凭证管理 API
- PRPL 模式
- 使用Payment Request API 进行支付
- 使用Web Share API 调用 Android 系统原生分享
- 使用Network Information API提示网络状态
- 使用 Element.scrollIntoView()和Element.scrollIntoViewIfNeeded()避免input输入时屏幕被键盘遮住
- ……

这里大致上列举了一下 PWA 的一些核心特性，感兴趣的同学可以做深入探讨。就目前的情况看，国外 PWA 活跃度更高，由于国内的特殊性，PWA 可能暂时还需要一段时间过渡。不过目前有些团队在做升级了，Lavas 是百度提出的一套解决方案，相信 PWA 在改善 web 体验（特别是 Android ）上还是有一定的优势的。

PWA 中比较有特点的一个特性是 Service Worker，可以让网页的用户体验做到极致，大家可以体验一下这个例子：https://weatherpwa.baidu.com/。打开页面后，关闭网络刷新浏览器会发现页面依然能

够加载，利用这种特性我们可以很好的做离线应用。

Lavas 对 PWA 新特性做了一些封装与降级处理，可以说为我们提供了一些非常不错的解决方案，结合 Vue.js 社区的组件库做了一些设计，提供了脚手架工具，可以很方便的跑一个 PWA 工程。限于篇幅，这里暂不做深入探讨。

Node.js 框架 Egg.js 的基本用法

Node.js 本身的内容十分多，如果要展开讲解，恐怕得一本书才能更好的诠释，不可能因为一篇文章就搞懂所有的内容，只能说希望给新手学习 Node 提供一种思路。先掌握最核心的内容然后自己再拓展学习。

首先说明为啥会选择 Egg.js？ Node 相关的框架也有很多，其他的比如还有Express、Koa、thinkjs 等。Express 是比较经典的 Node 框架，Koa 是由 Express 原班人马打造的，都属于比较小巧的框架。对于一般的项目而言，Express 和 Koa 绰绰有余，但是就**开发效率、可维护性、运行性能及稳定性**，think.js 和 Egg.js 更有优势。Egg 继承于 Koa，奉行『**约定优于配置**』，按照一套统一的约定进行应用开发，插件机制也比较完善。

当然最重要的是看着顺眼，哈哈，没有别的原因。

之前简书上写过两篇 Node 相关的文章，为了便于阅读移到 segmentfault，可以在这里阅读：

- node学习之路（一）—— 网络请求
- node学习之路（二）—— Node.js 连接 MongoDB

egg.js 使用指南

我们从实例的角度，一步步地搭建出一个 Egg.js 应用，让你能快速的入门 Egg.js，首先下载脚手架，快速生成一个最简单的项目骨架：

```
$ npm i egg-init -g
$ egg-init learn-eggjs --type=simple
$ cd learn-eggjs
$ npm i
```

`npm run dev` 开启服务，浏览器打开 `http://127.0.0.1:7001/` 即可看到提示： `hi, egg` 。

此时目录结构如下：

```
learn-eggjs
├─── app
│    ├─── controller
│    │    └─── home.js
│    ├─── public
│    └─── router.js
├─── config
│    └─── plugin.js
│    └─── config.default.js
└─── package.json
```

完整的目录结构规范参见目录结构。

app 目录是我们的开发目录，config 是配置目录，我们在app 目录下还可以再建立 middleware 、service 、extend、view、model、schedule这几个目录。

常用目录及文件说明：

- app/router.js

  用于配置 URL 路由规则，具体参见 Router。

- app/controller/

  用于解析用户的输入，处理后返回相应的结果，具体参见 Controller。

- app/service/

  用于编写业务逻辑层，可选，建议使用，具体参见 Service。

- app/middleware/

  用于编写中间件，可选，具体参见 Middleware。

- app/public/

  用于放置静态资源，可选，具体参见内置插件 egg-static。

- app/schedule/

  用于定时任务，可选，具体参见定时任务。

- app/extend/

  用于框架的扩展，可选，具体参见框架扩展。

- app/view/

  用于放置模板文件，可选，由模板插件约定，具体参见模板渲染。

- app/model/

  用于放置领域模型，可选，由领域类相关插件约定，如 egg-sequelize。

- config/config.{env}.js

  用于编写配置文件，具体参见配置。

- config/plugin.js

  用于配置需要加载的插件，具体参见插件开发。

- test/

  用于单元测试，具体参见单元测试。

- app.js 和 agent.js

  用于自定义启动时的初始化工作，可选，具体参见启动自定义。关于agent.js的作用参见Agent机制。

我们接下来重点看看 controller 、service、middleware、model、config 以及 router.js。

Router 路由

我们首先看看router.js，

```
'use strict';

module.exports = app => {
  app.get('/', 'home.index');
};
```

这是定义一个最简单的 URL 路由规则，当浏览器get请求 `http://127.0.0.1:7001/` 时，app/controller/home.js里面的index 方法执行，还可以写成： `app.get('/', app.controller.home.index);` 。

Router 详细定义说明：

根据参数的不同有以下几种方式：

```
app.verb('path-match', app.controller.controller.action);
app.verb('router-name', 'path-match', app.controller.controller.action);
app.verb('path-match', middleware1, ..., middlewareN, app.controller.controller.
app.verb('router-name', 'path-match', middleware1, ..., middlewareN, app.control
```

- verb – 用户触发动作：
  - app.head – HEAD
  - app.options – OPTIONS
  - app.get – GET
  - app.put – PUT
  - app.post – POST
  - app.patch – PATCH
  - app.delete – DELETE
  - app.del – delete方法的别名
  - app.redirect – 重定向
- router-name – 路由别名
- path-match – 路由 URL 路径
- middleware1, ..., middlewareN – 中间件
- controller – 控制器

我们在编写 API 的时候通常遵循 restful 设计风格。REST（即Representational State Transfer），翻译为“表现层状态转化”。因为HTTP协议是一个无状态协议，这意味着，所有的状态都保存在服务器端。因此，如果客户端想要操作服务器，必须通过某种手段，让服务器端发生"状态转化"（State Transfer）。而这种转化是建立在表现层之上的，所以就是"表现层状态转化"。客户端用到的手段，只能是HTTP协议。具体来说，就是HTTP协议里面，四个表示操作方式的动词：GET、POST、PUT、DELETE。它们分别对应四种基本操作：**GET用来获取资源，POST用来新建资源（也可以用于更新资源），PUT用来更新资源，DELETE用来删除资源。**

Egg.js对RESTful路由定义做了简化，在构建restful API时很有优势。如果想通过 RESTful 的方式来定义路由， 可以使用 `app.resources('routerName', 'pathMatch', controller)` 快速在一个路径上生成 CRUD 路由结构。例如我们在定义Blogs的API时，可以这样定义：

```
app.get('/api/v1/blogs', 'v1.blogs.index'); // 获取博文列表
app.post('/api/v1/blogs/new', 'v1.blogs.create'); // 新建博文内容
app.get('/api/v1/blogs/:id', 'v1.blogs.show'); // 展示博文详情
app.put('/api/v1/blogs/:id', 'v1.blogs.update'); // 更新博文内容
app.delete('/api/v1/blogs/:id', 'v1.blogs.delete'); // 删除博文内容
...
```

注：我们通常为了区分API的版本，可以在controller文件夹下建立一个名为v1版本号的文件夹加以区分。

我们也可以直接使用 `app.resources('blogs', '/api/v1/blogs', app.controller.v1.blogs);` 即可，然后在app/controller/v1/blogs.js 里面定义预定义的方法就可以：

```
exports.index = function* () {};
exports.new = function* () {};
exports.create = function* () {};
exports.show = function* () {};
exports.edit = function* () {};
exports.update = function* () {};
exports.destroy = function* () {};
```

对应关系可以看 router mapping：

Method	Path	Route Name	Controller.Action
GET	/posts	posts	app.controller.posts.index
GET	/posts/new	new_post	app.controller.posts.new
GET	/posts/:id	post	app.controller.posts.show
GET	/posts/:id/edit	edit_post	app.controller.posts.edit
POST	/posts	posts	app.controller.posts.create
PUT	/posts/:id	post	app.controller.posts.update
DELETE	/posts/:id	post	app.controller.posts.destroy

RESTful路由关键在于约定，当我们定义好了router，接下来就是去实现controller里面的方法。

Controller 控制器

简单的说 Controller 负责解析用户的输入，处理后返回相应的结果。在 RESTful 接口中，Controller 接受用户的参数，从数据库中查找内容返回给用户或者将用户的请求更新到数据库中。在 HTML 页面请求中，Controller 根据用户访问不同的 URL，渲染不同的模板得到 HTML 返回给用户。在代理服务器中，Controller 将用户的请求转发到其他服务器上，并将其他服务器的处理结果返回给用户。

Controller 有两种写法，Controller 方法和Controller 类。Controller 方法不推荐使用，只是为了兼容。每一个 Controller 都是一个 generator function，它的 this 指向请求的上下文 Context 对象的实例，通过它我们可以拿到框架封装好的各种便捷属性和方法。例如：

```
exports.index = function* () {
  this.body = 'hello egg.js';
  this.status = 200;
};
```

也可以将上下文Context作为参数ctx传入，通过ctx对象调用Context的方法，我们可以将ctx在页面展示出来 `this.body = this` ，然后可以看看上下文Context对象有哪一些属性方法。

官方推荐按照类的方式编写 Controller，不仅可以让我们更好的对 Controller 层代码进行抽象（例如将一些统一的处理抽象成一些私有方法），还可以通过自定义 Controller 基类的方式封装应用中常用的方法。

```
// app/controller/v1/user.js
module.exports = app => {
```

```
  return class UserController extends app.Controller {
    * index() {
      this.ctx.body = this;
    }
  };
};
```


框架中内置的一些基础对象，包括从 Koa 继承而来的 4 个对象（Application, Context, Request, Response) 以及框架扩展的一些对象（Controller, Service, Helper, Config, Logger）。

app对象指的是 Koa 的全局应用对象，全局只有一个，在应用启动时被创建，有三种访问方式：

- ctx.app.
- Controller，Middleware，Helper，Service 中都可以通过 this.app 访问到 Application 对象，例如 this.app.config 访问配置对象。
- 在 app.js 中 app 对象会作为第一个参数注入到入口函数中。

```
// app.js
module.exports = app => {
  // 使用 app 对象
};
```

其他对象类似，在此不做赘述，详细看文档介绍：框架内置基础对象。

HTTP 请求过程通常会传递参数，主要分为以下几种方式：

- 获取 URL Query String，可以通过 `context.query` 拿到解析过后的这个参数体。
- 获取 Router params，这些参数都可以通过 `context.params` 。
- 获取 HTTP 请求报文 body，可以通过 `context.request.body` 。
- 获取上传的文件，可以通过 `context.getFileStream*()` 接口。

- 获取 header 参数，可以通过

  `context.headers` ， `context.header` ， `context.request.headers` ，`context.request.header` 获取整个 header 对象，context.get(name)，context.request.get(name)：获取请求 header 中的一个字段的值，如果这个字段不存在，会返回空字符串。

- 管理 Cookie，服务端可以通过响应头（set-cookie）将少量数据响应给客户端，浏览器会遵循协议将数据保存，并在下次请求同一个服务的时候带上（浏览器也会遵循协议，只在访问符合 Cookie 指定规则的网站时带上对应的 Cookie 来保证安全性）。 `context.cookies` ，我们可以在 Controller 中便捷、安全的设置和读取 Cookie。

- 通过 `context.session` 来访问或者修改当前用户 Session。

Egg.js 框架内置了 CSRF 防范方案，默认是开启状态，为了方便接口调用，我们这里选择关闭，通过设置白名单的方式访问接口。

`{app_root}/config/config.default.js` 填加配置:

```
config.security = {
  csrf: {
    enable: false,
  },
  domainWhiteList: [
    'http://127.0.0.1:8080',
    'http://localhost:8080',
  ],
};
```

我们前端工程可以在通过8080端口访问，下面我们接着写我们的控制器逻辑。

我们这里以新建用户为例说明。我们需要输入用户名和密码，然后通过POST提交信息注册用户。在获取到用户请求的参数后，不可避免的要对参数进行一些校验。这里我们通过 `egg-validate` 插件进行校验。

安装:

```
npm i egg-validate --save
```

配置 `{app_root}/config/plugin.js` ：

```
exports.validate = {
  enable: true,
  package: 'egg-validate',
};
```

```
// {app_root}/app/controller/v1/user.js

'use strict';

const bcrypt = require('bcrypt');

module.exports = app => {
  return class UserController extends app.Controller {
    * create() {
      const ctx = this.ctx;
      let { name, password } = ctx.request.body;
      // 校验参数
      try {
        ctx.validate({
          name: { type: 'string' },
          password: { type: 'string' },
        });
      } catch (err) {
        ctx.logger.warn(err.errors);
        ctx.body = err.errors;
        return;
      }
      // 密码加密
      let salt = bcrypt.genSaltSync(10);
      password = bcrypt.hashSync('password', salt);
      ctx.body = {
        name,
        password
      };
      // service 服务层处理
      // ctx.body = yield ctx.service.user.create({
      //     name,
      //     password
      // });
      ctx.status = 200;
    }
```

```
  };
};
```

这里我使用 Postman 调试接口，很明显我们的密码使用bcrypt模块加密。

很显然我们不可能都是输出固定内容，我们需要操作数据库，在 Controller中调用service层操作数据库。

Service 服务

Service 就是在复杂业务场景下用于做业务逻辑封装的一个抽象层，提供这个抽象有以下几个好处：

- 保持 Controller 中的逻辑更加简洁。
- 保持业务逻辑的独立性，抽象出来的 Service 可以被多个 Controller 重复调用。
- 将逻辑和展现分离，更容易编写测试用例

下面我们开始介绍数据库的基本操作，这里以MongoDB数据库为例加以说明。我们首先要在本地环境安装好MongoDB数据库，不熟悉的可以看一下我之前的一篇文章：node学习之路（二）—— Node.js 连接 MongoDB。

启动数据库后，我们接下来继续在 egg 工程中进行，我们需要下载 egg-mongoose 插件，官方地址 => egg-mongoose。

```
npm i egg-mongoose --save
```

配置 {app_root}/config/plugin.js ：

```
exports.mongoose = {
  enable: true,
  package: 'egg-mongoose',
};
```

配置 {app_root}/config/config.default.js ：

```
module.exports = appInfo => {
  const config = {};

  // mongodb config
  config.mongoose = {
    url: 'mongodb://127.0.0.1/zhaomenghuan',
    options: {},
  };

  return config;
};
```

mongoose 是个 odm，odm 的概念对应 sql 中的 orm。orm 全称是 Object-Relational Mapping，对象关系映射；而 odm 是 Object-Document Mapping，对象文档映射。

mongoose 官方文档：http://mongoosejs.com/

我们在app目录下建立一个model文件夹用于定义schema，建立一个service文件夹用户管理后端数据库操作。

{app_root}/app/model/user.js ：

```
'use strict';

module.exports = app => {
  const mongoose = app.mongoose;

  const UserSchema = new mongoose.Schema({
    name: {
      type: String,
      unique: true, // 不可重复约束
```

```
      require: true // 不可为空约束
    },
    password: {
      type: String,
      require: true // 不可为空约束
    }
  });

  return mongoose.model('User', UserSchema);
};
```

{app_root}/app/service/user.js ：

```
'use strict';

module.exports = app => {
  class UserService extends app.Service {
    * create(options) {
      const ctx = this.ctx;
      let result = yield new ctx.model.User(options).save();
      return result;
    }
  }
  return UserService;
};
```

可以通过 Robomongo 查看数据库存的内容：

这里只是讲解了数据库操作，RESTful API 基本设计，很有很多内容有待我们去实现。

Linux 云服务器搭建部署 Node 服务

当我们的系统开发完之后，我们需要部署到服务器上使用，目前主流的云主机是Linux，Linux 部署代码的基本技能需要掌握，大家可以看看我之前的一篇文章：Linux CentOS7 搭建node服务详细教程，主要涉及下面几方面的内容：

- 登录Linux云服务器
- Linux 系统目录结构
- Linux 文件基本属性
- Linux 文件与目录管理
- Linux 磁盘管理
- Linux vim
- Linux Yum 包管理器
- 安装及启动nginx
- Linux 源码编译安装node.js
- 通过Filezilla 进行文件上传下载
- Linux Nginx ssl证书部署
- Linux node服务nginx配置
- Linux 安装 MongDB 数据库

这里不做重复赘述，只说明一下我们部署代码的时候可以使用如下脚本部署：

```
// ssh 登录
ssh xxxxxxx C:/Users/Administrator/.ssh/id_rsa
// 设置 node 环境变量
export PATH=/opt/node-v7.7.4-linux-x64/bin/:$PATH
// 服务端目录
cd /var/www
// 从 git 仓库拉资源
git pull origin master
// 启动
npm start
```

对于前端而言配置 nginx，部署环境，掌握基本的 Linux 操作还是很有必要的，这里只是抛砖引玉，希望共同进步。

参考

- Custom Elements 中文规范
- Web Components 可用性调研
- 跟 Web Components 打个啵
- lavas PWA 文档
- Google Web Fundamentals
- Node.js+MongoDB对于RestfulApi中用户token认证实践

GitChat 是一种全新的阅读/写作互动体验产品。一场 Chat 包含一篇文章和一场为文章的读者和作者定制的专属线上交流。本文出自 Chat 话题《JavaScrip 进阶之 Vue.js + Node.js 入门实战开发》。