09 Node.js

从本章开始，我们就正式开启JavaScript的后端开发之旅。

Node.js是目前非常火热的技术，但是它的诞生经历却很奇特。

众所周知，在Netscape设计出JavaScript后的短短几个月，JavaScript事实上已经是前端开发的唯一标准。

后来，微软通过IE击败了Netscape后一统桌面，结果几年时间，浏览器毫无进步。（2001年推出的古老的IE 6到今天仍然有人在使用！）

没有竞争就没有发展。微软认为IE6浏览器已经非常完善，几乎没有可改进之处，然后解散了IE6开发团队！而Google却认为支持现代Web应用的新一代浏览器才刚刚起步，尤其是浏览器负责运行JavaScript的引擎性能还可提升10倍。

先是Mozilla借助已壮烈牺牲的Netscape遗产在2002年推出了Firefox浏览器，紧接着Apple于2003年在开源的KHTML浏览器的基础上推出了WebKit内核的Safari浏览器，不过仅限于Mac平台。

随后，Google也开始创建自家的浏览器。他们也看中了WebKit内核，于是基于WebKit内核推出了Chrome浏览器。

Chrome浏览器是跨Windows和Mac平台的，并且，Google认为要运行现代Web应用，浏览器必须有一个性能非常强劲的JavaScript引擎，于是Google自己开发了一个高性能JavaScript引擎，名字叫V8，以BSD许可证开源。

现代浏览器大战让微软的IE浏览器远远地落后了，因为他们解散了最有经验、战斗力最强的浏览器团队！回过头再追赶却发现，支持HTML5的WebKit已经成为手机端的标准了，IE浏览器从此与主流移动端设备绝缘。

浏览器大战和Node有何关系？

话说有个叫Ryan Dahl的歪果仁，他的工作是用C/C++写高性能Web服务。对于高性能，异步IO、事件驱动是基本原则，但是用C/C++写就太痛苦了。于是这位仁兄开始设想用高级语言开发Web服务。他评估了很多种高级语言，发现很多语言虽然同时提供了同步IO和异步IO，但是开发人员一旦用了同步IO，他们就再也懒得写异步IO了，所以，最终，Ryan瞄向了JavaScript。

因为JavaScript是单线程执行，根本不能进行同步IO操作，所以，JavaScript的这一“缺陷”导致了它只能使用异步IO。

选定了开发语言，还要有运行时引擎。这位仁兄曾考虑过自己写一个，不过明智地放弃了，因为V8就是开源的JavaScript引擎。让Google投资去优化V8，咱只负责改造一下拿来用，还不用付钱，这个买卖很划算。

于是在2009年，Ryan正式推出了基于JavaScript语言和V8引擎的开源Web服务器项目，命名为Node.js。虽然名字很土，但是，Node第一次把JavaScript带入到后端服务器开发，加上世界上已经有无数的JavaScript开发人员，所以Node一下子就火了起来。

在Node上运行的JavaScript相比其他后端开发语言有何优势？

最大的优势是借助JavaScript天生的事件驱动机制加V8高性能引擎，使编写高性能Web服务轻而易举。

其次，JavaScript语言本身是完善的函数式语言，在前端开发时，开发人员往往写得比较随意，让人感觉JavaScript就是个“玩具语言”。但是，在Node环境下，通过模块化的JavaScript代码，加上函数式编程，并且无需考虑浏览器兼容性问题，直接使用最新的ECMAScript 6标准，可以完全满足工程上的需求。

> *我还听说过io.js，这又是什么鬼？*

因为Node.js是开源项目，虽然由社区推动，但幕后一直由Joyent公司资助。由于一群开发者对Joyent公司的策略不满，于2014年从Node.js项目fork出了io.js项目，决定单独发展，但两者实际上是兼容的。

然而中国有句古话，叫做“分久必合，合久必分”。分家后没多久，Joyent公司表示要和解，于是，io.js项目又决定回归Node.js。

具体做法是将来io.js将首先添加新的特性，如果大家测试用得爽，就把新特性加入Node.js。io.js是“尝鲜版”，而Node.js是线上稳定版，相当于Fedora Linux和RHEL的关系。

本章教程的所有代码都在Node.js上调试通过。如果你要尝试io.js也是可以的，不过两者如果遇到一些区别请自行查看io.js的文档。

安装Node.js和npm

由于Node.js平台是在后端运行JavaScript代码，所以，必须首先在本机安装Node环境。

安装Node.js

目前Node.js的最新版本是7.6.x。首先，从Node.js官网下载对应平台的安装程序，网速慢的童鞋请移步国内镜像。

在Windows上安装时务必选择全部组件，包括勾选`Add to Path`。

安装完成后，在Windows环境下，请打开命令提示符，然后输入`node -v`，如果安装正常，你应该看到`v7.6.0`这样的输出：

```
C:\Users\IEUser>node -v
v7.6.0
```

继续在命令提示符输入`node`，此刻你将进入Node.js的交互环境。在交互环境下，你可以输入任意JavaScript语句，例如`100+200`，回车后将得到输出结果。

要退出Node.js环境，连按两次Ctrl+C。

在Mac或Linux环境下，请打开终端，然后输入`node -v`，你应该看到如下输出：

```
$ node -v
v7.6.0
```

如果版本号小于`v7.6.0`，说明Node.js版本不对，后面章节的代码不保证能正常运行，请重新安装最新版本。

npm

在正式开始Node.js学习之前，我们先认识一下npm。

npm是什么东东？npm其实是Node.js的包管理工具（package manager）。

为啥我们需要一个包管理工具呢？因为我们在Node.js上开发时，会用到很多别人写的JavaScript代码。如果我们要使用别人写的某个包，每次都根据名称搜索一下官方网站，下载代码，解压，再使用，非常繁琐。于是一个集中管理的工具应运而生：大家都把自己开发的模块打包后放到npm官网上，如果要使用，直接通过npm安装就可以直接用，不用管代码存在哪，应该从哪下载。

更重要的是，如果我们要使用模块A，而模块A又依赖于模块B，模块B又依赖于模块X和模块Y，npm可以根据依赖关系，把所有依赖的包都下载下来并管理起来。否则，靠我们自己手动管理，肯定既麻烦又容易出错。

讲了这么多，npm究竟在哪？

其实npm已经在Node.js安装的时候顺带装好了。我们在命令提示符或者终端输入`npm -v`，应该看到类似的输出：

```
C:\>npm -v
4.1.2
```

如果直接输入`npm`，你会看到类似下面的输出：

```
C:\> npm

Usage: npm <command>

where <command> is one of:
    ...
```

上面的一大堆文字告诉你，`npm`需要跟上命令。现在我们不用关心这些命令，后面会一一讲到。目前，你只需要确保npm正确安装了，能运行就行。

小结

请在本机安装Node.js环境，并确保`node`和`npm`能正常运行。

第一个Node程序

在前面的所有章节中，我们编写的JavaScript代码都是在浏览器中运行的，因此，我们可以直接在浏览器中敲代码，然后直接运行。

从本章开始，我们编写的JavaScript代码将*不能*在浏览器环境中执行了，而是在Node环境中执行，因此，JavaScript代码将直接在你的计算机上以命令行的方式运行，所以，我们要先选择一个文本编辑器来编写JavaScript代码，并且把它保存到本地硬盘的某个目录，才能够执行。

那么问题来了：文本编辑器到底哪家强？

首先，请注意，绝对不能用Word和写字板。Word和写字板保存的不是纯文本文件。如果我们要用记事本来编写JavaScript代码，要务必注意，记事本以UTF-8格式保存文件时，会自作聪明地在文件开始的地方加上几个特殊字符（UTF-8 BOM），结果经常会导致程序运行出现莫名其妙的错误。

所以，用记事本写代码时请注意，保存文件时使用ANSI编码，并且暂时不要输入中文。

如果你的电脑上已经安装了Sublime Text，或者Notepad++，也可以用来编写JavaScript代码，注意用UTF-8格式保存。

输入以下代码：

```
'use strict';

console.log('Hello, world.');
```

第一行总是写上`'use strict';`是因为我们总是以严格模式运行JavaScript代码，避免各种潜在陷阱。

然后，选择一个目录，例如`C:\Workspace`，把文件保存为`hello.js`，就可以打开命令行窗口，把当前目录切换到`hello.js`所在目录，然后输入以下命令运行这个程序了：

```
C:\Workspace>node hello.js
Hello, world.
```

也可以保存为别的名字，比如`first.js`，但是必须要以`.js`结尾。此外，文件名只能是英文字母、数字和下划线的组合。

如果当前目录下没有`hello.js`这个文件，运行`node hello.js`就会报错：

```
C:\Workspace>node hello.js
module.js:338
    throw err;
          ^
Error: Cannot find module 'C:\Workspace\hello.js'
    at Function.Module._resolveFilename
    at Function.Module._load
    at Function.Module.runMain
    at startup
    at node.js
```

报错的意思就是，没有找到`hello.js`这个文件，因为文件不存在。这个时候，就要检查一下当前目录下是否有这个文件了。

命令行模式和Node交互模式

请注意区分命令行模式和Node交互模式。

看到类似`C:\>`是在Windows提供的命令行模式：

在命令行模式下，可以执行`node`进入Node交互式环境，也可以执行`node hello.js`运行一个`.js`文件。

看到`>`是在Node交互式环境下：

在Node交互式环境下，我们可以输入JavaScript代码并立刻执行。

此外，在命令行模式运行`.js`文件和在Node交互式环境下直接运行JavaScript代码有所不同。Node交互式环境会把每一行JavaScript代码的结果自动打印出来，但是，直接运行JavaScript文件却不会。

例如，在Node交互式环境下，输入：

```
> 100 + 200 + 300;
600
```

直接可以看到结果`600`。

但是，写一个`calc.js`的文件，内容如下：

```
100 + 200 + 300;
```

然后在命令行模式下执行：

```
C:\Workspace>node calc.js
```

发现什么输出都没有。

这是正常的。想要输出结果，必须自己用`console.log()`打印出来。把`calc.js`改造一下：

```
console.log(100 + 200 + 300);
```

再执行，就可以看到结果：

```
C:\Workspace>node calc.js
600
```

使用严格模式

如果在JavaScript文件开头写上`'use strict';`，那么Node在执行该JavaScript时将使用严格模式。但是，在服务器环境下，如果有很多JavaScript文件，每个文件都写上`'use strict';`很麻烦。我们可以给Nodejs传递一个参数，让Node直接为所有js文件开启严格模式：

```
node --use_strict calc.js
```

后续代码，如无特殊说明，我们都会直接给Node传递`--use_strict`参数来开启严格模式。

小结

用文本编辑器写JavaScript程序，然后保存为后缀为`.js`的文件，就可以用node直接运行这个程序了。

Node的交互模式和直接运行`.js`文件有什么区别呢？

直接输入`node`进入交互模式，相当于启动了Node解释器，但是等待你一行一行地输入源代码，每输入一行就执行一行。

直接运行`node hello.js`文件相当于启动了Node解释器，然后一次性把`hello.js`文件的源代码给执行了，你是没有机会以交互的方式输入源代码的。

在编写JavaScript代码的时候，完全可以一边在文本编辑器里写代码，一边开一个Node交互式命令窗口，在写代码的过程中，把部分代码粘到命令行去验证，事半功倍！前提是得有个27'的超大显示器！

搭建Node开发环境

使用文本编辑器来开发Node程序，最大的缺点是效率太低，运行Node程序还需要在命令行单独敲命令。如果还需要调试程序，就更加麻烦了。

所以我们需要一个IDE集成开发环境，让我们能在一个环境里编码、运行、调试，这样就可以大大提升开发效率。

Java的集成开发环境有Eclipse，Intellij idea等，C#的集成开发环境有Visual Studio，那么问题又来了：Node.js的集成开发环境到底哪家强？

考察Node.js的集成开发环境，重点放在启动速度快，执行简单，调试方便这三点上。当然，免费使用是一个加分项。

综合考察后，我们隆重向大家推荐Node.js集成开发环境：

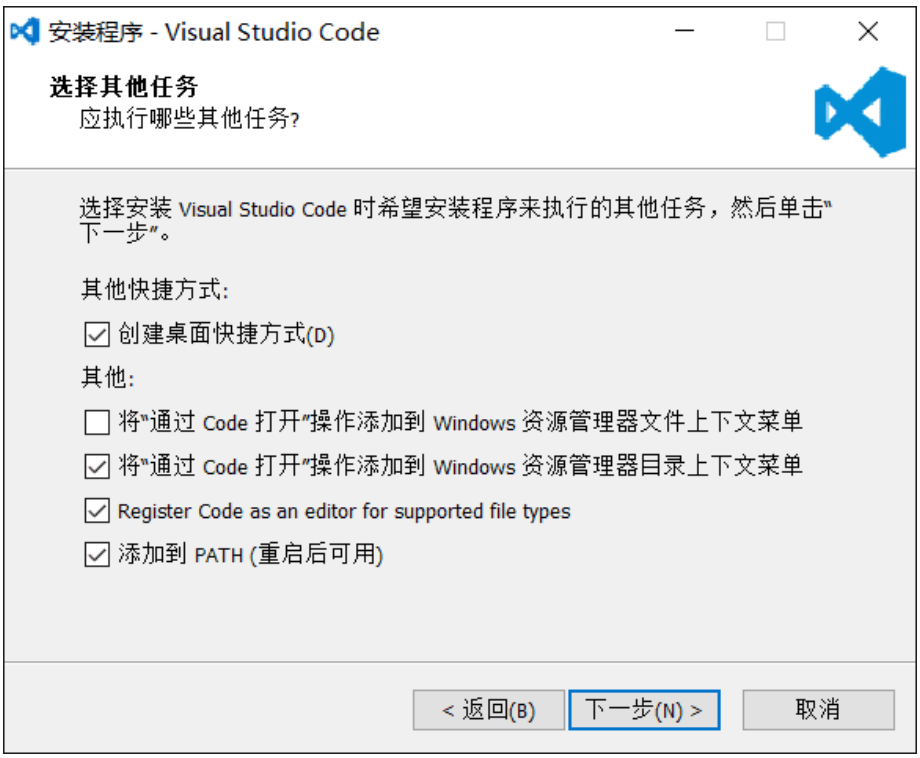
Visual Studio Code

Visual Studio Code由微软出品，但它不是那个大块头的Visual Studio，它是一个精简版的迷你Visual Studio，并且，Visual Studio Code可以跨！平！台！Windows、Mac和Linux通用。

安装Visual Studio Code

可以从Visual Studio Code的官方网站下载并安装最新的版本。

安装过程中，请务必钩上以下选项：

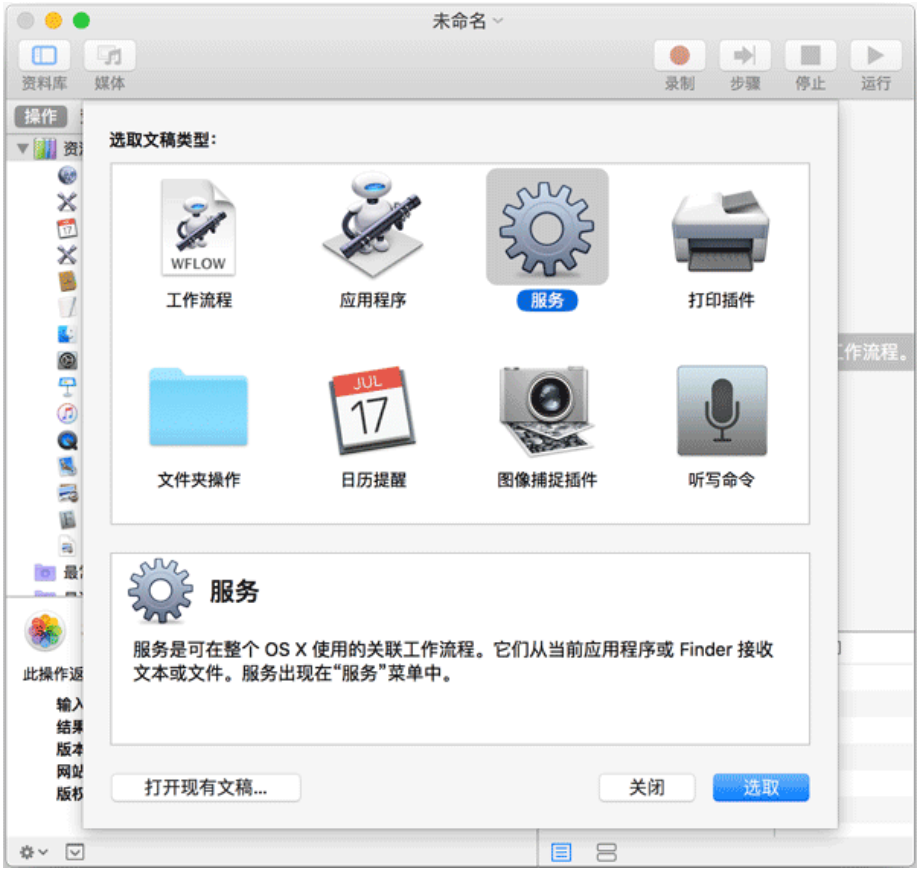


将“通过Code打开”操作添加到Windows资源管理器目录上下文菜单

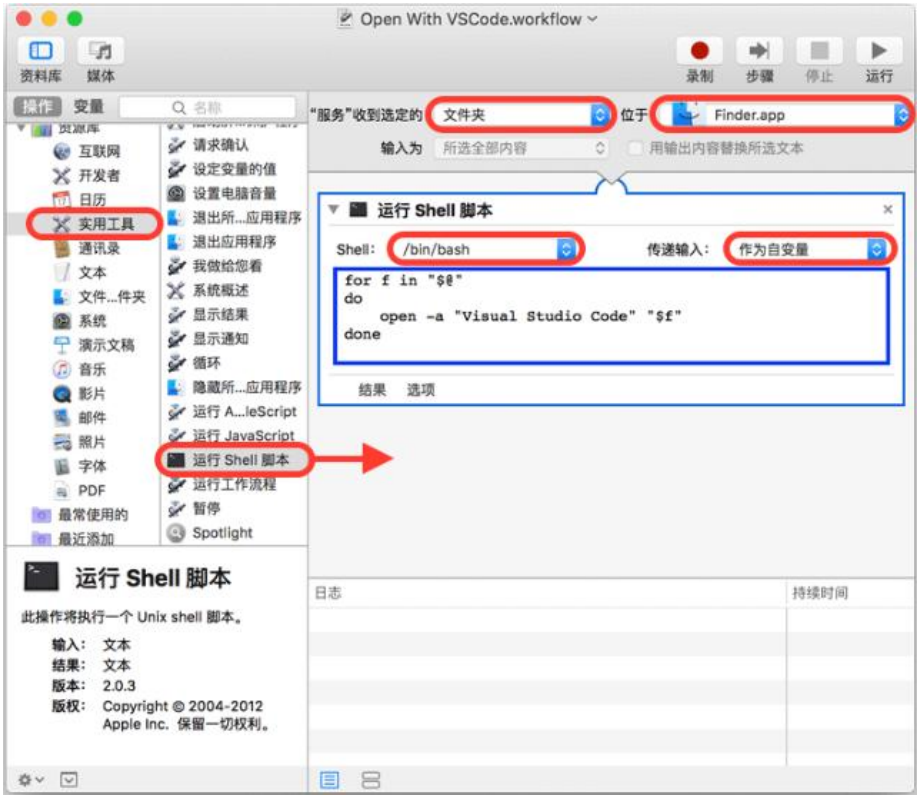
这将大大提升将来的操作快捷度。

在Mac系统上，Finder选中一个目录，右键菜单并没有“通过Code打开”这个操作。不过我们可以通过Automator自己添加这个操作。

先运行Automator，选择“服务”：



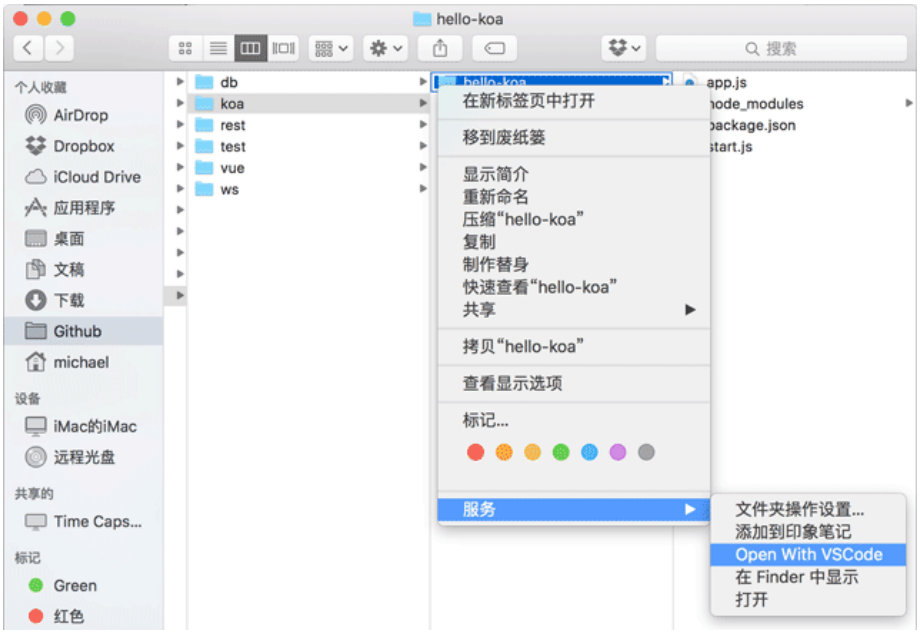
然后，执行以下操作：



1. 在右侧面板选择“服务”收到选定的“文件夹”，位于“Finder.app“，该选项是为了从Finder中接收一个文件夹；
2. 在左侧面板选择”实用工具“，然后找到”运行Shell脚本“，把它拽到右侧面板里；
3. 在右侧”运行Shell脚本“的面板里，选择Shell”/bin/bash“，传递输入“作为自变量”，然后修改Shell脚本如下：

```
for f in "$@"
do
    open -a "Visual Studio Code" "$f"
done
```

保存为“Open With VSCode”后，打开Finder，选中一个文件夹，点击右键，“服务”，就可以看到“Open With VSCode”菜单：



运行和调试JavaScript

在VS Code中，我们可以非常方便地运行JavaScript文件。

VS Code以文件夹作为工程目录（Workspace Dir），所有的JavaScript文件都存放在该目录下。此外，VS Code在工程目录下还需要一个`.vscode`的配置目录，里面存放里VS Code需要的配置文件。

假设我们在`C:\work\`目录下创建了一个`hello`目录作为工程目录，并编写了一个`hello.js`文件，则该工程目录的结构如下：

```
hello/ <-- workspace dir
|
+- hello.js <-- JavaScript file
|
+- .vscode/  <-- VS Code config
   |
   +- launch.json <-- VS Code config file for JavaScript
```

可以用VS Code快速创建`launch.json`，然后修改如下：

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Run hello.js",
            "type": "node",
            "request": "launch",
            "program": "${workspaceRoot}/hello.js",
            "stopOnEntry": false,
            "args": [],
            "cwd": "${workspaceRoot}",
            "preLaunchTask": null,
            "runtimeExecutable": null,
            "runtimeArgs": [
                "--nolazy"
            ],
            "env": {
                "NODE_ENV": "development"
            },
            "externalConsole": false,
            "sourceMaps": false,
            "outDir": null
        }
    ]
}
```

有了配置文件，即可使用VS Code调试JavaScript。

视频演示：

模块

在计算机程序的开发过程中，随着程序代码越写越多，在一个文件里代码就会越来越长，越来越不容易维护。

为了编写可维护的代码，我们把很多函数分组，分别放到不同的文件里，这样，每个文件包含的代码就相对较少，很多编程语言都采用这种组织代码的方式。在Node环境中，一个.js文件就称之为一个模块（module）。

使用模块有什么好处？

最大的好处是大大提高了代码的可维护性。其次，编写代码不必从零开始。当一个模块编写完毕，就可以被其他地方引用。我们在编写程序的时候，也经常引用其他模块，包括Node内置的模块和来自第三方的模块。

使用模块还可以避免函数名和变量名冲突。相同名字的函数和变量完全可以分别存在不同的模块中，因此，我们自己在编写模块时，不必考虑名字会与其他模块冲突。

在上一节，我们编写了一个`hello.js`文件，这个`hello.js`文件就是一个模块，模块的名字就是文件名（去掉`.js`后缀），所以`hello.js`文件就是名为`hello`的模块。

我们把`hello.js`改造一下，创建一个函数，这样我们就可以在其他地方调用这个函数：

```
'use strict';

var s = 'Hello';

function greet(name) {
    console.log(s + ', ' + name + '!');
}

module.exports = greet;
```

函数`greet()`是我们在`hello`模块中定义的，你可能注意到最后一行是一个奇怪的赋值语句，它的意思是，把函数`greet`作为模块的输出暴露出去，这样其他模块就可以使用`greet`函数了。

问题是其他模块怎么使用`hello`模块的这个`greet`函数呢？我们再编写一个`main.js`文件，调用`hello`模块的`greet`函数：

```
'use strict';

// 引入hello模块:
var greet = require('./hello');

var s = 'Michael';

greet(s); // Hello, Michael!
```

注意到引入`hello`模块用Node提供的`require`函数：

```
var greet = require('./hello');
```

引入的模块作为变量保存在`greet`变量中，那`greet`变量到底是什么东西？其实变量`greet`就是在`hello.js`中我们用`module.exports = greet;`输出的`greet`函数。所以，`main.js`就成功地引用了`hello.js`模块中定义的`greet()`函数，接下来就可以直接使用它了。

在使用`require()`引入模块的时候，请注意模块的相对路径。因为`main.js`和`hello.js`位于同一个目录，所以我们用了当前目录`.`：

```
var greet = require('./hello'); // 不要忘了写相对目录!
```

如果只写模块名：

```
var greet = require('hello');
```

则Node会依次在内置模块、全局模块和当前模块下查找`hello.js`，你很可能会得到一个错误：

```
module.js
    throw err;
          ^
Error: Cannot find module 'hello'
    at Function.Module._resolveFilename
    at Function.Module._load
    ...
    at Function.Module._load
    at Function.Module.runMain
```

遇到这个错误，你要检查：

- 模块名是否写对了；
- 模块文件是否存在；
- 相对路径是否写对了。

CommonJS规范

这种模块加载机制被称为CommonJS规范。在这个规范下，每个`.js`文件都是一个模块，它们内部各自使用的变量名和函数名都互不冲突，例如，`hello.js`和`main.js`都申明了全局变量`var s = 'xxx'`，但互不影响。

一个模块想要对外暴露变量（函数也是变量），可以用`module.exports = variable;`，一个模块要引用其他模块暴露的变量，用`var ref = require('module_name');`就拿到了引用模块的变量。

结论

要在模块中对外输出变量，用：

```
module.exports = variable;
```

输出的变量可以是任意对象、函数、数组等等。

要引入其他模块输出的对象，用：

```
var foo = require('other_module');
```

引入的对象具体是什么，取决于引入模块输出的对象。

深入了解模块原理

如果你想详细地了解CommonJS的模块实现原理，请继续往下阅读。如果不想了解，请直接跳到最后做练习。

当我们编写JavaScript代码时，我们可以申明全局变量：

```
var s = 'global';
```

在浏览器中，大量使用全局变量可不好。如果你在`a.js`中使用了全局变量`s`，那么，在`b.js`中也使用全局变量`s`，将造成冲突，`b.js`中对`s`赋值会改变`a.js`的运行逻辑。

也就是说，JavaScript语言本身并没有一种模块机制来保证不同模块可以使用相同的变量名。

那Node.js是如何实现这一点的？

其实要实现“模块”这个功能，并不需要语法层面的支持。Node.js也并不会增加任何JavaScript语法。实现“模块”功能的奥妙就在于JavaScript是一种函数式编程语言，它支持闭包。如果我们把一段JavaScript代码用一个函数包装起来，这段代码的所有“全局”变量就变成了函数内部的局部变量。

请注意我们编写的`hello.js`代码是这样的：

```
var s = 'Hello';
var name = 'world';

console.log(s + ' ' + name + '!');
```

Node.js加载了`hello.js`后，它可以把代码包装一下，变成这样执行：

```
(function () {
    // 读取的hello.js代码:
    var s = 'Hello';
    var name = 'world';

    console.log(s + ' ' + name + '!');
    // hello.js代码结束
})();
```

这样一来，原来的全局变量`s`现在变成了匿名函数内部的局部变量。如果Node.js继续加载其他模块，这些模块中定义的“全局”变量`s`也互不干扰。

所以，Node利用JavaScript的函数式编程的特性，轻而易举地实现了模块的隔离。

但是，模块的输出`module.exports`怎么实现？

这个也很容易实现，Node可以先准备一个对象`module`：

```
// 准备module对象:
var module = {
    id: 'hello',
    exports: {}
};
var load = function (module) {
    // 读取的hello.js代码:
    function greet(name) {
        console.log('Hello, ' + name + '!');
    }

    module.exports = greet;
    // hello.js代码结束
    return module.exports;
};
var exported = load(module);
// 保存module:
save(module, exported);
```

可见，变量`module`是Node在加载js文件前准备的一个变量，并将其传入加载函数，我们在`hello.js`中可以直接使用变量`module`原因就在于它实际上是函数的一个参数：

```
module.exports = greet;
```

通过把参数`module`传递给`load()`函数，`hello.js`就顺利地把一个变量传递给了Node执行环境，Node会把`module`变量保存到某个地方。

由于Node保存了所有导入的`module`，当我们用`require()`获取module时，Node找到对应的`module`，把这个`module`的`exports`变量返回，这样，另一个模块就顺利拿到了模块的输出：

```
var greet = require('./hello');
```

以上是Node实现JavaScript模块的一个简单的原理介绍。

module.exports vs exports

很多时候，你会看到，在Node环境中，有两种方法可以在一个模块中输出变量：

方法一：对module.exports赋值：

```
// hello.js

function hello() {
    console.log('Hello, world!');
}

function greet(name) {
    console.log('Hello, ' + name + '!');
}

module.exports = {
    hello: hello,
    greet: greet
};
```

方法二：直接使用exports：

```
// hello.js

function hello() {
    console.log('Hello, world!');
}

function greet(name) {
    console.log('Hello, ' + name + '!');
}

function hello() {
    console.log('Hello, world!');
}

exports.hello = hello;
exports.greet = greet;
```

但是你不可以直接对`exports`赋值：

```
// 代码可以执行，但是模块并没有输出任何变量:
exports = {
    hello: hello,
    greet: greet
};
```

如果你对上面的写法感到十分困惑，不要着急，我们来分析Node的加载机制：

首先，Node会把整个待加载的`hello.js`文件放入一个包装函数`load`中执行。在执行这个`load()`函数前，Node准备好了module变量：

```
var module = {
    id: 'hello',
    exports: {}
};
```

`load()`函数最终返回`module.exports`：

```
var load = function (exports, module) {
    // hello.js的文件内容
    ...
    // load函数返回:
    return module.exports;
};

var exported = load(module.exports, module);
```

也就是说，默认情况下，Node准备的`exports`变量和`module.exports`变量实际上是同一个变量，并且初始化为空对象`{}`，于是，我们可以写：

```
exports.foo = function () { return 'foo'; };
exports.bar = function () { return 'bar'; };
```

也可以写：

```
module.exports.foo = function () { return 'foo'; };
module.exports.bar = function () { return 'bar'; };
```

换句话说，Node默认给你准备了一个空对象`{}`，这样你可以直接往里面加东西。

但是，如果我们要输出的是一个函数或数组，那么，只能给`module.exports`赋值：

```
module.exports = function () { return 'foo'; };
```

给`exports`赋值是无效的，因为赋值后，`module.exports`仍然是空对象`{}`。

结论

如果要输出一个键值对象`{}`，可以利用`exports`这个已存在的空对象`{}`，并继续在上面添加新的键值；

如果要输出一个函数或数组，必须直接对`module.exports`对象赋值。

所以我们可以得出结论：直接对`module.exports`赋值，可以应对任何情况：

```
module.exports = {
    foo: function () { return 'foo'; }
};
```

或者：

```
module.exports = function () { return 'foo'; };
```

最终，我们*强烈建议*使用`module.exports = xxx`的方式来输出模块变量，这样，你只需要记忆一种方法。

练习

编写`hello.js`，输出一个或多个函数；

编写`main.js`，引入`hello`模块，调用其函数。

基本模块

因为Node.js是运行在服务区端的JavaScript环境，服务器程序和浏览器程序相比，最大的特点是没有浏览器的安全限制了，而且，服务器程序必须能接收网络请求，读写文件，处理二进制内容，所以，Node.js内置的常用模块就是为了实现基本的服务器功能。这些模块在浏览器环境中是无法被执行的，因为它们的底层代码是用C/C++在Node.js运行环境中实现的。

global

在前面的JavaScript课程中，我们已经知道，JavaScript有且仅有一个全局对象，在浏览器中，叫`window`对象。而在Node.js环境中，也有唯一的全局对象，但不叫`window`，而叫`global`，这个对象的属性和方法也和浏览器环境的`window`不同。进入Node.js交互环境，可以直接输入：

```
> global.console
Console {
  log: [Function: bound ],
  info: [Function: bound ],
  warn: [Function: bound ],
  error: [Function: bound ],
  dir: [Function: bound ],
  time: [Function: bound ],
  timeEnd: [Function: bound ],
  trace: [Function: bound trace],
  assert: [Function: bound ],
  Console: [Function: Console] }
```

process

`process`也是Node.js提供的一个对象，它代表当前Node.js进程。通过`process`对象可以拿到许多有用信息：

```
> process === global.process;
true
> process.version;
'v5.2.0'
> process.platform;
'darwin'
> process.arch;
'x64'
> process.cwd(); //返回当前工作目录
'/Users/michael'
> process.chdir('/private/tmp'); // 切换当前工作目录
undefined
> process.cwd();
'/private/tmp'
```

JavaScript程序是由事件驱动执行的单线程模型，Node.js也不例外。Node.js不断执行响应事件的JavaScript函数，直到没有任何响应事件的函数可以执行时，Node.js就退出了。

如果我们想要在下一次事件响应中执行代码，可以调用`process.nextTick()`：

```
// test.js

// process.nextTick()将在下一轮事件循环中调用:
process.nextTick(function () {
    console.log('nextTick callback!');
});
console.log('nextTick was set!');
```

用Node执行上面的代码`node test.js`，你会看到，打印输出是：

```
nextTick was set!
nextTick callback!
```

这说明传入`process.nextTick()`的函数不是立刻执行，而是要等到下一次事件循环。

Node.js进程本身的事件就由`process`对象来处理。如果我们响应`exit`事件，就可以在程序即将退出时执行某个回调函数：

```
// 程序即将退出时的回调函数:
process.on('exit', function (code) {
    console.log('about to exit with code: ' + code);
});
```

判断JavaScript执行环境

有很多JavaScript代码既能在浏览器中执行，也能在Node环境执行，但有些时候，程序本身需要判断自己到底是在什么环境下执行的，常用的方式就是根据浏览器和Node环境提供的全局变量名称来判断：

```
if (typeof(window) === 'undefined') {
    console.log('node.js');
} else {
    console.log('browser');
}
```

后面，我们将介绍Node.js的常用内置模块。

fs

Node.js内置的`fs`模块就是文件系统模块，负责读写文件。

和所有其它JavaScript模块不同的是，`fs`模块同时提供了异步和同步的方法。

回顾一下什么是异步方法。因为JavaScript的单线程模型，执行IO操作时，JavaScript代码无需等待，而是传入回调函数后，继续执行后续JavaScript代码。比如jQuery提供的`getJSON()`操作：

```
$.getJSON('http://example.com/ajax', function (data) {
    console.log('IO结果返回后执行...');
});
console.log('不等待IO结果直接执行后续代码...');
```

而同步的IO操作则需要等待函数返回：

```
// 根据网络耗时，函数将执行几十毫秒~几秒不等:
var data = getJSONSync('http://example.com/ajax');
```

同步操作的好处是代码简单，缺点是程序将等待IO操作，在等待时间内，无法响应其它任何事件。而异步读取不用等待IO操作，但代码较麻烦。

异步读文件

按照JavaScript的标准，异步读取一个文本文件的代码如下：

```
'use strict';

var fs = require('fs');

fs.readFile('sample.txt', 'utf-8', function (err, data) {
    if (err) {
        console.log(err);
    } else {
        console.log(data);
    }
});
```

请注意，`sample.txt`文件必须在当前目录下，且文件编码为`utf-8`。

异步读取时，传入的回调函数接收两个参数，当正常读取时，`err`参数为`null`，`data`参数为读取到的String。当读取发生错误时，`err`参数代表一个错误对象，`data`为`undefined`。这也是Node.js标准的回调函数：第一个参数代表错误信息，第二个参数代表结果。后面我们还会经常编写这种回调函数。

由于`err`是否为`null`就是判断是否出错的标志，所以通常的判断逻辑总是：

```
if (err) {
    // 出错了
} else {
    // 正常
}
```

如果我们要读取的文件不是文本文件，而是二进制文件，怎么办？

下面的例子演示了如何读取一个图片文件：

```
'use strict';

var fs = require('fs');

fs.readFile('sample.png', function (err, data) {
    if (err) {
        console.log(err);
    } else {
        console.log(data);
        console.log(data.length + ' bytes');
    }
});
```

当读取二进制文件时，不传入文件编码时，回调函数的`data`参数将返回一个`Buffer`对象。在Node.js中，`Buffer`对象就是一个包含零个或任意个字节的数组（注意和Array不同）。

`Buffer`对象可以和String作转换，例如，把一个`Buffer`对象转换成String：

```
// Buffer -> String
var text = data.toString('utf-8');
console.log(text);
```

或者把一个String转换成`Buffer`：

```
// String -> Buffer
var buf = Buffer.from(text, 'utf-8');
console.log(buf);
```

同步读文件

除了标准的异步读取模式外，`fs`也提供相应的同步读取函数。同步读取的函数和异步函数相比，多了一个`Sync`后缀，并且不接收回调函数，函数直接返回结果。

用`fs`模块同步读取一个文本文件的代码如下：

```
'use strict';

var fs = require('fs');

var data = fs.readFileSync('sample.txt', 'utf-8');
console.log(data);
```

可见，原异步调用的回调函数的`data`被函数直接返回，函数名需要改为`readFileSync`，其它参数不变。

如果同步读取文件发生错误，则需要用`try...catch`捕获该错误：

```
try {
    var data = fs.readFileSync('sample.txt', 'utf-8');
    console.log(data);
} catch (err) {
    // 出错了
}
```

写文件

将数据写入文件是通过`fs.writeFile()`实现的：

```
'use strict';

var fs = require('fs');

var data = 'Hello, Node.js';
fs.writeFile('output.txt', data, function (err) {
    if (err) {
        console.log(err);
    } else {
        console.log('ok.');
    }
});
```

`writeFile()`的参数依次为文件名、数据和回调函数。如果传入的数据是String，默认按UTF-8编码写入文本文件，如果传入的参数是`Buffer`，则写入的是二进制文件。回调函数由于只关心成功与否，因此只需要一个`err`参数。

和`readFile()`类似，`writeFile()`也有一个同步方法，叫`writeFileSync()`：

```
'use strict';

var fs = require('fs');

var data = 'Hello, Node.js';
fs.writeFileSync('output.txt', data);
```

stat

如果我们要获取文件大小，创建时间等信息，可以使用`fs.stat()`，它返回一个`Stat`对象，能告诉我们文件或目录的详细信息：

```
'use strict';

var fs = require('fs');

fs.stat('sample.txt', function (err, stat) {
    if (err) {
        console.log(err);
    } else {
        // 是否是文件:
        console.log('isFile: ' + stat.isFile());
        // 是否是目录:
        console.log('isDirectory: ' + stat.isDirectory());
        if (stat.isFile()) {
            // 文件大小:
            console.log('size: ' + stat.size);
            // 创建时间, Date对象:
            console.log('birth time: ' + stat.birthtime);
            // 修改时间, Date对象:
            console.log('modified time: ' + stat.mtime);
```

```
        }
    }
});
```

运行结果如下：

```
isFile: true
isDirectory: false
size: 181
birth time: Fri Dec 11 2015 09:43:41 GMT+0800 (CST)
modified time: Fri Dec 11 2015 12:09:00 GMT+0800 (CST)
```

`stat()`也有一个对应的同步函数`statSync()`，请试着改写上述异步代码为同步代码。

异步还是同步

在`fs`模块中，提供同步方法是为了方便使用。那我们到底是应该用异步方法还是同步方法呢？

由于Node环境执行的JavaScript代码是服务器端代码，所以，绝大部分需要在服务器运行期反复执行业务逻辑的代码，*必须使用异步代码*，否则，同步代码在执行时期，服务器将停止响应，因为JavaScript只有一个执行线程。

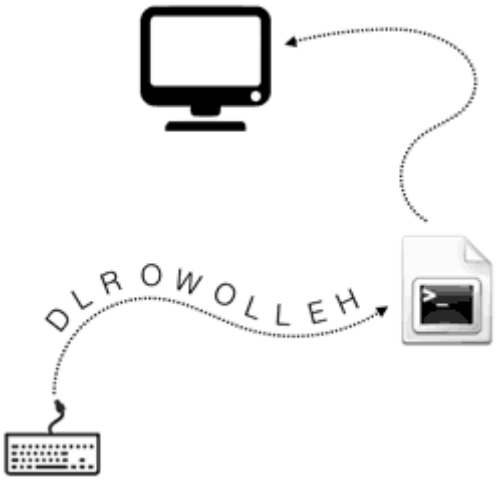
服务器启动时如果需要读取配置文件，或者结束时需要写入到状态文件时，可以使用同步代码，因为这些代码只在启动和结束时执行一次，不影响服务器正常运行时的异步执行。

stream

`stream`是Node.js提供的又一个仅在服务区端可用的模块，目的是支持“流”这种数据结构。

什么是流？流是一种抽象的数据结构。想象水流，当在水管中流动时，就可以从某个地方（例如自来水厂）源源不断地到达另一个地方（比如你家的洗手池）。我们也可以把数据看成是数据流，比如你敲键盘的时候，就可以把每个字符依次连起来，看成字符流。这个流是从键盘输入到应用程序，实际上它还对应着一个名字：标准输入流（stdin）。

如果应用程序把字符一个一个输出到显示器上，这也可以看成是一个流，这个流也有名字：标准输出流（stdout）。流的特点是数据是有序的，而且必须依次读取，或者依次写入，不能像Array那样随机定位。



有些流用来读取数据，比如从文件读取数据时，可以打开一个文件流，然后从文件流中不断地读取数据。有些流用来写入数据，比如向文件写入数据时，只需要把数据不断地往文件流中写进去就可以了。

在Node.js中，流也是一个对象，我们只需要响应流的事件就可以了：`data`事件表示流的数据已经可以读取了，`end`事件表示这个流已经到末尾了，没有数据可以读取了，`error`事件表示出错了。

下面是一个从文件流读取文本内容的示例：

```
'use strict';

var fs = require('fs');

// 打开一个流:
var rs = fs.createReadStream('sample.txt', 'utf-8');

rs.on('data', function (chunk) {
    console.log('DATA:')
    console.log(chunk);
});

rs.on('end', function () {
    console.log('END');
});

rs.on('error', function (err) {
    console.log('ERROR: ' + err);
});
```

要注意，`data`事件可能会有多次，每次传递的`chunk`是流的一部分数据。

要以流的形式写入文件，只需要不断调用`write()`方法，最后以`end()`结束：

```
'use strict';

var fs = require('fs');

var ws1 = fs.createWriteStream('output1.txt', 'utf-8');
ws1.write('使用Stream写入文本数据...\n');
ws1.write('END.');
ws1.end();

var ws2 = fs.createWriteStream('output2.txt');
ws2.write(new Buffer('使用Stream写入二进制数据...\n', 'utf-
8'));
ws2.write(new Buffer('END.', 'utf-8'));
ws2.end();
```

所有可以读取数据的流都继承自`stream.Readable`，所有可以写入的流都继承自`stream.Writable`。

pipe

就像可以把两个水管串成一个更长的水管一样，两个流也可以串起来。一个`Readable`流和一个`Writable`流串起来后，所有的数据自动从`Readable`流进入`Writable`流，这种操作叫`pipe`。

在Node.js中，`Readable`流有一个`pipe()`方法，就是用来干这件事的。

让我们用`pipe()`把一个文件流和另一个文件流串起来，这样源文件的所有数据就自动写入到目标文件里了，所以，这实际上是一个复制文件的程序：

```
'use strict';

var fs = require('fs');

var rs = fs.createReadStream('sample.txt');
var ws = fs.createWriteStream('copied.txt');

rs.pipe(ws);
```

默认情况下，当`Readable`流的数据读取完毕，`end`事件触发后，将自动关闭`Writable`流。如果我们不希望自动关闭`Writable`流，需要传入参数：

```
readable.pipe(writable, { end: false });
```

http

Node.js开发的目的就是为了用JavaScript编写Web服务器程序。因为JavaScript实际上已经统治了浏览器端的脚本，其优势就是有世界上数量最多的前端开发人员。如果已经掌握了JavaScript前端开发，再学习一下如何将JavaScript应用在后端开发，就是名副其实的*全栈*了。

HTTP协议

要理解Web服务器程序的工作原理，首先，我们要对HTTP协议有基本的了解。如果你对HTTP协议不太熟悉，先看一看HTTP协议简介。

HTTP服务器

要开发HTTP服务器程序，从头处理TCP连接，解析HTTP是不现实的。这些工作实际上已经由Node.js自带的`http`模块完成了。应用程序并不直接和HTTP协议打交道，而是操作`http`模块提供的`request`和`response`对象。

`request`对象封装了HTTP请求，我们调用`request`对象的属性和方法就可以拿到所有HTTP请求的信息；

`response`对象封装了HTTP响应，我们操作`response`对象的方法，就可以把HTTP响应返回给浏览器。

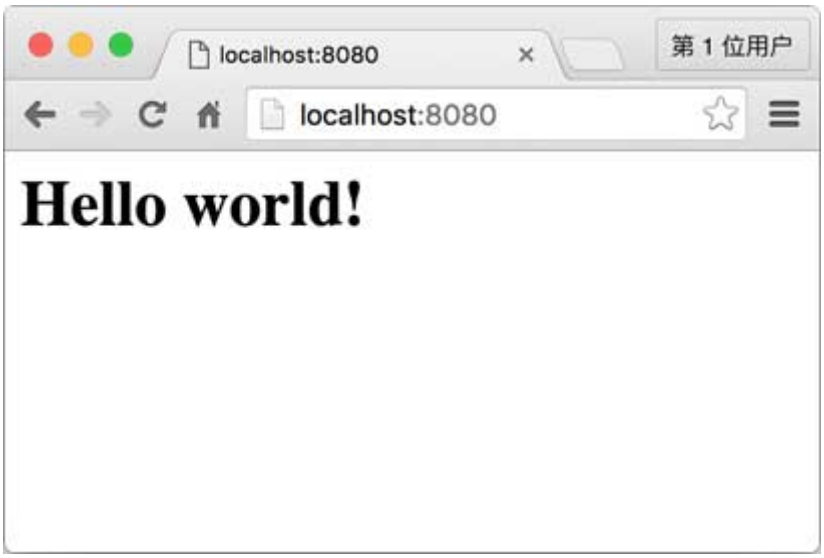
用Node.js实现一个HTTP服务器程序非常简单。我们来实现一个最简单的Web程序`hello.js`，它对于所有请求，都返回`Hello world!`：

```
'use strict';

// 导入http模块:
var http = require('http');

// 创建http server，并传入回调函数:
var server = http.createServer(function (request,
response) {
    // 回调函数接收request和response对象，
    // 获得HTTP请求的method和url:
    console.log(request.method + ': ' + request.url);
    // 将HTTP响应200写入response，同时设置Content-Type:
text/html:
    response.writeHead(200, {'Content-Type':
'text/html'});
    // 将HTTP响应的HTML内容写入response:
    response.end('<h1>Hello world!</h1>');
});

// 让服务器监听8080端口:
server.listen(8080);

console.log('Server is running at
http://127.0.0.1:8080/');
```

在命令提示符下运行该程序，可以看到以下输出：

```
$ node hello.js
Server is running at http://127.0.0.1:8080/
```

不要关闭命令提示符，直接打开浏览器输入`http://localhost:8080`，即可看到服务器响应的内容：



同时，在命令提示符窗口，可以看到程序打印的请求信息：

```
GET: /
GET: /favicon.ico
```

这就是我们编写的第一个HTTP服务器程序！

文件服务器

让我们继续扩展一下上面的Web程序。我们可以设定一个目录，然后让Web程序变成一个文件服务器。要实现这一点，我们只需要解析`request.url`中的路径，然后在本地找到对应的文件，把文件内容发送出去就可以了。

解析URL需要用到Node.js提供的`url`模块，它使用起来非常简单，通过`parse()`将一个字符串解析为一个`url`对象：

```
'use strict';

var url = require('url');

console.log(url.parse('http://user:pass@host.com:8080/path
/to/file?query=string#hash'));
```

结果如下：

```
Url {
  protocol: 'http:',
  slashes: true,
  auth: 'user:pass',
  host: 'host.com:8080',
  port: '8080',
  hostname: 'host.com',
  hash: '#hash',
  search: '?query=string',
  query: 'query=string',
  pathname: '/path/to/file',
  path: '/path/to/file?query=string',
  href: 'http://user:pass@host.com:8080/path/to/file?
query=string#hash' }
```

处理本地文件目录需要使用Node.js提供的`path`模块，它可以方便地构造目录：

```
'use strict';

var path = require('path');

// 解析当前目录:
var workDir = path.resolve('.'); // '/Users/michael'

// 组合完整的文件路径:当前目录+'pub'+'index.html':
var filePath = path.join(workDir, 'pub', 'index.html');
// '/Users/michael/pub/index.html'
```

使用`path`模块可以正确处理操作系统相关的文件路径。在Windows系统下，返回的路径类似于`C:\Users\michael\static\index.html`，这样，我们就不关心怎么拼接路径了。

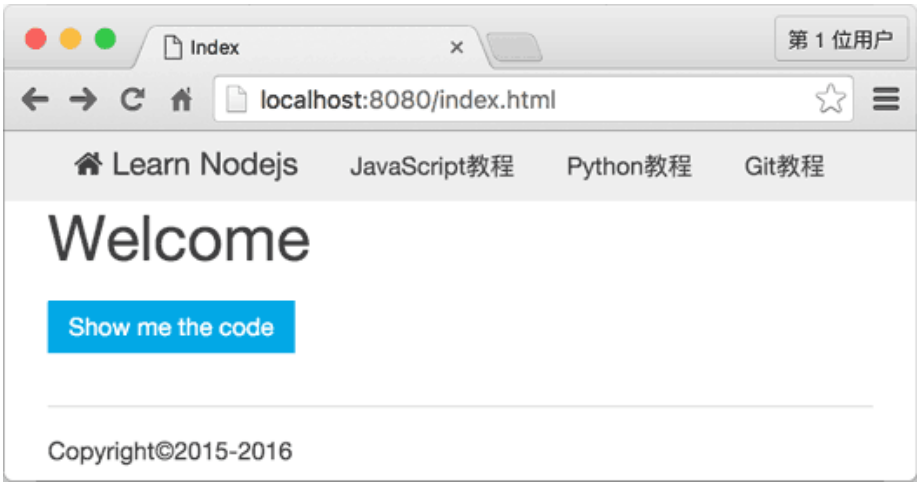
最后，我们实现一个文件服务器`file_server.js`：

```
'use strict';

var
    fs = require('fs'),
    url = require('url'),
    path = require('path'),
    http = require('http');

// 从命令行参数获取root目录，默认是当前目录:
var root = path.resolve(process.argv[2] || '.');

console.log('Static root dir: ' + root);

// 创建服务器:
var server = http.createServer(function (request,
response) {
```

```
    // 获得URL的path，类似 '/css/bootstrap.css':
    var pathname = url.parse(request.url).pathname;
    // 获得对应的本地文件路径，类似
'/srv/www/css/bootstrap.css':
    var filepath = path.join(root, pathname);
    // 获取文件状态:
    fs.stat(filepath, function (err, stats) {
        if (!err && stats.isFile()) {
            // 没有出错并且文件存在:
            console.log('200 ' + request.url);
            // 发送200响应:
            response.writeHead(200);
            // 将文件流导向response:
            fs.createReadStream(filepath).pipe(response);
        } else {
            // 出错了或者文件不存在:
            console.log('404 ' + request.url);
            // 发送404响应:
            response.writeHead(404);
            response.end('404 Not Found');
        }
    });
});

server.listen(8080);

console.log('Server is running at
http://127.0.0.1:8080/');
```

没有必要手动读取文件内容。由于`response`对象本身是一个`Writable Stream`，直接用`pipe()`方法就实现了自动读取文件内容并输出到HTTP响应。

在命令行运行`node file_server.js /path/to/dir`，把`/path/to/dir`改成你本地的一个有效的目录，然后在浏览器中输入`http://localhost:8080/index.html`：



只要当前目录下存在文件`index.html`，服务器就可以把文件内容发送给浏览器。观察控制台输出：

```
200 /index.html
200 /css/uikit.min.css
200 /js/jquery.min.js
200 /fonts/fontawesome-webfont.woff2
```

第一个请求是浏览器请求`index.html`页面，后续请求是浏览器解析HTML后发送的其它资源请求。

练习

在浏览器输入`http://localhost:8080/`时，会返回404，原因是程序识别出HTTP请求的不是文件，而是目录。请修改`file_server.js`，如果遇到请求的路径是目录，则自动在目录下依次搜索`index.html`、`default.html`，如果找到了，就返回HTML文件的内容。

crypto

crypto模块的目的是为了提供通用的加密和哈希算法。用纯JavaScript代码实现这些功能不是不可能，但速度会非常慢。Nodejs用C/C++实现这些算法后，通过cypto这个模块暴露为JavaScript接口，这样用起来方便，运行速度也快。

MD5和SHA1

MD5是一种常用的哈希算法，用于给任意数据一个“签名”。这个签名通常用一个十六进制的字符串表示：

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

const hash = crypto.createHash('md5');

// 可任意多次调用update():
hash.update('Hello, world!');
hash.update('Hello, nodejs!');

console.log(hash.digest('hex')); //
7e1977739c748beac0c0fd14fd26a544
```

`update()`方法默认字符串编码为`UTF-8`，也可以传入Buffer。

如果要计算SHA1，只需要把`'md5'`改成`'sha1'`，就可以得到SHA1的结果`1f32b9c9932c02227819a4151feed43e131aca40`。

还可以使用更安全的`sha256`和`sha512`。

Hmac

Hmac算法也是一种哈希算法，它可以利用MD5或SHA1等哈希算法。不同的是，Hmac还需要一个密钥：

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

const hmac = crypto.createHmac('sha256', 'secret-key');

hmac.update('Hello, world!');
hmac.update('Hello, nodejs!');

console.log(hmac.digest('hex')); // 80f7e22570...
```

只要密钥发生了变化，那么同样的输入数据也会得到不同的签名，因此，可以把Hmac理解为用随机数“增强”的哈希算法。

AES

AES是一种常用的对称加密算法，加解密都用同一个密钥。crypto模块提供了AES支持，但是需要自己封装好函数，便于使用：

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

function aesEncrypt(data, key) {
    const cipher = crypto.createCipher('aes192', key);
    var crypted = cipher.update(data, 'utf8', 'hex');
    crypted += cipher.final('hex');
    return crypted;
}

function aesDecrypt(encrypted, key) {
    const decipher = crypto.createDecipher('aes192', key);
    var decrypted = decipher.update(encrypted, 'hex',
'utf8');
    decrypted += decipher.final('utf8');
    return decrypted;
}

var data = 'Hello, this is a secret message!';
var key = 'Password!';
var encrypted = aesEncrypt(data, key);
var decrypted = aesDecrypt(encrypted, key);

console.log('Plain text: ' + data);
console.log('Encrypted text: ' + encrypted);
console.log('Decrypted text: ' + decrypted);
```

运行结果如下：

```
Plain text: Hello, this is a secret message!
Encrypted text: 8a944d97bdabc157a5b7a40cb180e7...
Decrypted text: Hello, this is a secret message!
```

可以看出，加密后的字符串通过解密又得到了原始内容。

注意到AES有很多不同的算法，如`aes192`，`aes-128-ecb`，`aes-256-cbc`等，AES除了密钥外还可以指定IV（Initial Vector），不同的系统只要IV不同，用相同的密钥加密相同的数据得到的加密结果也是不同的。加密结果通常有两种表示方法：hex和base64，这些功能Nodejs全部都支持，但是在应用中要注意，如果加解密双方一方用Nodejs，另一方用Java、PHP等其它语言，需要仔细测试。如果无法正确解密，要确认双方是否遵循同样的AES算法，字符串密钥和IV是否相同，加密后的数据是否统一为hex或base64格式。

Diffie-Hellman

DH算法是一种密钥交换协议，它可以让双方在不泄漏密钥的情况下协商出一个密钥来。DH算法基于数学原理，比如小明和小红想要协商一个密钥，可以这么做：

1. 小明先选一个素数和一个底数，例如，素数`p=23`，底数`g=5`（底数可以任选），再选择一个秘密整数`a=6`，计算`A=g^a mod p=8`，然后大声告诉小红：`p=23, g=5, A=8`；
2. 小红收到小明发来的`p`，`g`，`A`后，也选一个秘密整数`b=15`，然后计算`B=g^b mod p=19`，并大声告诉小明：`B=19`；
3. 小明自己计算出`s=B^a mod p=2`，小红也自己计算出`s=A^b mod p=2`，因此，最终协商的密钥`s`为`2`。

在这个过程中，密钥`2`并不是小明告诉小红的，也不是小红告诉小明的，而是双方协商计算出来的。第三方只能知道`p=23`，`g=5`，`A=8`，`B=19`，由于不知道双方选的秘密整数`a=6`和`b=15`，因此无法计算出密钥`2`。

用crypto模块实现DH算法如下：

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

// xiaoming's keys:
var ming = crypto.createDiffieHellman(512);
var ming_keys = ming.generateKeys();

var prime = ming.getPrime();
var generator = ming.getGenerator();

console.log('Prime: ' + prime.toString('hex'));
console.log('Generator: ' + generator.toString('hex'));

// xiaohong's keys:
var hong = crypto.createDiffieHellman(prime, generator);
var hong_keys = hong.generateKeys();

// exchange and generate secret:
```

```
var ming_secret = ming.computeSecret(hong_keys);
var hong_secret = hong.computeSecret(ming_keys);

// print secret:
console.log('Secret of Xiao Ming: ' +
ming_secret.toString('hex'));
console.log('Secret of Xiao Hong: ' +
hong_secret.toString('hex'));
```

运行后，可以得到如下输出：

```
$ node dh.js
Prime: a8224c...deead3
Generator: 02
Secret of Xiao Ming: 695308...d519be
Secret of Xiao Hong: 695308...d519be
```

注意每次输出都不一样，因为素数的选择是随机的。

RSA

RSA算法是一种非对称加密算法，即由一个私钥和一个公钥构成的密钥对，通过私钥加密，公钥解密，或者通过公钥加密，私钥解密。其中，公钥可以公开，私钥必须保密。

RSA算法是1977年由Ron Rivest、Adi Shamir和Leonard Adleman共同提出的，所以以他们三人的姓氏的头字母命名。

当小明给小红发送信息时，可以用小明自己的私钥加密，小红用小明的公钥解密，也可以用小红的公钥加密，小红用她自己的私钥解密，这就是非对称加密。相比对称加密，非对称加密只需要每个人各自持有自己的私钥，同时公开自己的公钥，不需要像AES那样由两个人共享同一个密钥。

在使用Node进行RSA加密前，我们先要准备好私钥和公钥。

首先，在命令行执行以下命令以生成一个RSA密钥对：

```
openssl genrsa -aes256 -out rsa-key.pem 2048
```

根据提示输入密码，这个密码是用来加密RSA密钥的，加密方式指定为AES256，生成的RSA的密钥长度是2048位。执行成功后，我们获得了加密的`rsa-key.pem`文件。

第二步，通过上面的`rsa-key.pem`加密文件，我们可以导出原始的私钥，命令如下：

```
openssl rsa -in rsa-key.pem -outform PEM -out rsa-prv.pem
```

输入第一步的密码，我们获得了解密后的私钥。

类似的，我们用下面的命令导出原始的公钥：

```
openssl rsa -in rsa-key.pem -outform PEM -pubout -out rsa-pub.pem
```

这样，我们就准备好了原始私钥文件`rsa-prv.pem`和原始公钥文件`rsa-pub.pem`，编码格式均为PEM。

下面，使用`crypto`模块提供的方法，即可实现非对称加解密。

首先，我们用私钥加密，公钥解密：

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

// 从文件加载key:
function loadKey(file) {
    // key实际上就是PEM编码的字符串:
    return fs.readFileSync(file, 'utf8');
}

let
    prvKey = loadKey('./rsa-prv.pem'),
    pubKey = loadKey('./rsa-pub.pem'),
    message = 'Hello, world!';

// 使用私钥加密:
let enc_by_prv = crypto.privateEncrypt(prvKey,
Buffer.from(message, 'utf8'));
console.log('encrypted by private key: ' +
enc_by_prv.toString('hex'));


let dec_by_pub = crypto.publicDecrypt(pubKey, enc_by_prv);
console.log('decrypted by public key: ' +
dec_by_pub.toString('utf8'));
```

执行后，可以得到解密后的消息，与原始消息相同。

接下来我们使用公钥加密，私钥解密：

```
// 使用公钥加密:
let enc_by_pub = crypto.publicEncrypt(pubKey,
Buffer.from(message, 'utf8'));
console.log('encrypted by public key: ' +
enc_by_pub.toString('hex'));

// 使用私钥解密:
let dec_by_prv = crypto.privateDecrypt(prvKey,
enc_by_pub);
console.log('decrypted by private key: ' +
dec_by_prv.toString('utf8'));
```

执行得到的解密后的消息仍与原始消息相同。

如果我们把`message`字符串的长度增加到很长，例如1M，这时，执行RSA加密会得到一个类似这样的错误：`data too large for key size`，这是因为RSA加密的原始信息必须小于Key的长度。那如何用RSA加密一个很长的消息呢？实际上，RSA并不适合加密大数据，而是先生成一个随机的AES密码，用AES加密原始信息，然后用RSA加密AES口令，这样，实际使用RSA时，给对方传的密文分两部分，一部分是AES加密的密文，另一部分是RSA加密的AES口令。对方用RSA先解密出AES口令，再用AES解密密文，即可获得明文。

证书

crypto模块也可以处理数字证书。数字证书通常用在SSL连接，也就是Web的https连接。一般情况下，https连接只需要处理服务器端的单向认证，如无特殊需求（例如自己作为Root给客户发认证证书），建议用反向代理服务器如Nginx等Web服务器去处理证书。

Web开发

最早的软件都是运行在大型机上的，软件使用者通过“哑终端”登陆到大型机上去运行软件。后来随着PC机的兴起，软件开始主要运行在桌面上，而数据库这样的软件运行在服务器端，这种Client/Server模式简称CS架构。

随着互联网的兴起，人们发现，CS架构不适合Web，最大的原因是Web应用程序的修改和升级非常迅速，而CS架构需要每个客户端逐个升级桌面App，因此，Browser/Server模式开始流行，简称BS架构。

在BS架构下，客户端只需要浏览器，应用程序的逻辑和数据都存储在服务器端。浏览器只需要请求服务器，获取Web页面，并把Web页面展示给用户即可。

当然，Web页面也具有极强的交互性。由于Web页面是用HTML编写的，而HTML具备超强的表现力，并且，服务器端升级后，客户端无需任何部署就可以使用到新的版本，因此，BS架构迅速流行起来。

今天，除了重量级的软件如Office，Photoshop等，大部分软件都以Web形式提供。比如，新浪提供的新闻、博客、微博等服务，均是Web应用。

Web应用开发可以说是目前软件开发中最重要的部分。Web开发也经历了好几个阶段：

静态Web页面：由文本编辑器直接编辑并生成静态的HTML页面，如果要修改Web页面的内容，就需要再次编辑HTML源文件，早期的互联网Web页面就是静态的；

CGI：由于静态Web页面无法与用户交互，比如用户填写了一个注册表单，静态Web页面就无法处理。要处理用户发送的动态数据，出现了Common Gateway Interface，简称CGI，用C/C++编写。

ASP/JSP/PHP：由于Web应用特点是修改频繁，用C/C++这样的低级语言非常不适合Web开发，而脚本语言由于开发效率高，与HTML结合紧密，因此，迅速取代了CGI模式。ASP是微软推出的用VBScript脚本编程的Web开发技术，而JSP用Java来编写脚本，PHP本身则是开源的脚本语言。

MVC：为了解决直接用脚本语言嵌入HTML导致的可维护性差的问题，Web应用也引入了Model-View-Controller的模式，来简化Web开发。ASP发展为ASP.Net，JSP和PHP也有一大堆MVC框架。

目前，Web开发技术仍在快速发展中，异步开发、新的MVVM前端技术层出不穷。

由于Node.js把JavaScript引入了服务器端，因此，原来必须使用PHP/Java/C#/Python/Ruby等其他语言来开发服务器端程序，现在可以使用Node.js开发了！

用Node.js开发Web服务器端，有几个显著的优势：

一是后端语言也是JavaScript，以前掌握了前端JavaScript的开发人员，现在可以同时编写后端代码；

二是前后端统一使用JavaScript，就没有切换语言的障碍了；

三是速度快，非常快！这得益于Node.js天生是异步的。

在Node.js诞生后的短短几年里，出现了无数种Web框架、ORM框架、模版引擎、测试框架、自动化构建工具，数量之多，即使是JavaScript老司机，也不免眼花缭乱。

常见的Web框架包括：Express，Sails.js，koa，Meteor，DerbyJS，Total.js，restify……

ORM框架比Web框架要少一些：Sequelize，ORM2，Bookshelf.js，Objection.js……

模版引擎PK：Jade，EJS，Swig，Nunjucks，doT.js……

测试框架包括：Mocha，Expresso，Unit.js，Karma……

构建工具有：Grunt，Gulp，Webpack……

目前，在npm上已发布的开源Node.js模块数量超过了30万个。

有选择恐惧症的朋友，看到这里可以洗洗睡了。

好消息是这个教程已经帮你选好了，你只需要跟着教程一条道走到黑就可以了。

koa

koa是Express的下一代基于Node.js的web框架，目前有1.x和2.0两个版本。

历史

1. Express

Express是第一代最流行的web框架，它对Node.js的http进行了封装，用起来如下：

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
    res.send('Hello World!');
});

app.listen(3000, function () {
    console.log('Example app listening on port 3000!');
});
```

虽然Express的API很简单，但是它是基于ES5的语法，要实现异步代码，只有一个方法：回调。如果异步嵌套层次过多，代码写起来就非常难看：

```
app.get('/test', function (req, res) {
    fs.readFile('/file1', function (err, data) {
        if (err) {
            res.status(500).send('read file1 error');
        }
        fs.readFile('/file2', function (err, data) {
            if (err) {
                res.status(500).send('read file2 error');
            }
            res.type('text/plain');
            res.send(data);
        });
    });
});
```

虽然可以用async这样的库来组织异步代码，但是用回调写异步实在是太痛苦了！

2. koa 1.0

随着新版Node.js开始支持ES6，Express的团队又基于ES6的generator重新编写了下一代web框架koa。和Express相比，koa 1.0使用generator实现异步，代码看起来像同步的：

```
var koa = require('koa');
var app = koa();

app.use('/test', function *() {
    yield doReadFile1();
    var data = yield doReadFile2();
    this.body = data;
});

app.listen(3000);
```

用generator实现异步比回调简单了不少，但是generator的本意并不是异步。Promise才是为异步设计的，但是Promise的写法……想想就复杂。为了简化异步代码，ES7（目前是草案，还没有发布）引入了新的关键字`async`和`await`，可以轻松地把一个function变为异步模式：

```
async function () {
    var data = await fs.read('/file1');
}
```

这是JavaScript未来标准的异步代码，非常简洁，并且易于使用。

3. koa2

koa团队并没有止步于koa 1.0，他们非常超前地基于ES7开发了koa2，和koa 1相比，koa2完全使用Promise并配合`async`来实现异步。

koa2的代码看上去像这样：

```
app.use(async (ctx, next) => {
    await next();
    var data = await doReadFile();
    ctx.response.type = 'text/plain';
    ctx.response.body = data;
});
```

出于兼容性考虑，目前koa2仍支持generator的写法，但下一个版本将会去掉。

选择哪个版本？

目前JavaScript处于高速进化中，ES7是大势所趋。为了紧跟时代潮流，教程将使用最新的koa2开发！

koa入门

创建koa2工程

首先，我们创建一个目录`hello-koa`并作为工程目录用VS Code打开。然后，我们创建`app.js`，输入以下代码：

```
// 导入koa，和koa 1.x不同，在koa2中，我们导入的是一个class，因此用大写的Koa表示：
const Koa = require('koa');

// 创建一个Koa对象表示web app本身：
const app = new Koa();

// 对于任何请求，app将调用该异步函数处理请求：
app.use(async (ctx, next) => {
    await next();
    ctx.response.type = 'text/html';
    ctx.response.body = '<h1>Hello, koa2!</h1>';
});

// 在端口3000监听：
app.listen(3000);
console.log('app started at port 3000...');
```

对于每一个http请求，koa将调用我们传入的异步函数来处理：

```
async (ctx, next) => {
    await next();
    // 设置response的Content-Type:
    ctx.response.type = 'text/html';
    // 设置response的内容:
    ctx.response.body = '<h1>Hello, koa2!</h1>';
}
```

其中，参数`ctx`是由koa传入的封装了request和response的变量，我们可以通过它访问request和response，`next`是koa传入的将要处理的下一个异步函数。

上面的异步函数中，我们首先用`await next();`处理下一个异步函数，然后，设置response的Content-Type和内容。

由`async`标记的函数称为异步函数，在异步函数中，可以用`await`调用另一个异步函数，这两个关键字将在ES7中引入。

现在我们遇到第一个问题：koa这个包怎么装，`app.js`才能正常导入它？

方法一：可以用npm命令直接安装koa。先打开命令提示符，务必把当前目录切换到`hello-koa`这个目录，然后执行命令：

```
C:\...\hello-koa> npm install koa@2.0.0
```

npm会把koa2以及koa2依赖的所有包全部安装到当前目录的node_modules目录下。

方法二：在`hello-koa`这个目录下创建一个`package.json`，这个文件描述了我们的`hello-koa`工程会用到哪些包。完整的文件内容如下：

```
{
    "name": "hello-koa2",
    "version": "1.0.0",
    "description": "Hello Koa 2 example with async",
    "main": "app.js",
    "scripts": {
        "start": "node app.js"
    },
    "keywords": [
        "koa",
        "async"
    ],
    "author": "Michael Liao",
    "license": "Apache-2.0",
    "repository": {
        "type": "git",
        "url": "https://github.com/michaelliao/learn-
javascript.git"
    },
    "dependencies": {
        "koa": "2.0.0"
    }
}
```

其中，`dependencies`描述了我们的工程依赖的包以及版本号。其他字段均用来描述项目信息，可任意填写。

然后，我们在`hello-koa`目录下执行`npm install`就可以把所需包以及依赖包一次性全部装好：

```
C:\...\hello-koa> npm install
```

很显然，第二个方法更靠谱，因为我们只要在`package.json`正确设置了依赖，npm就会把所有用到的包都装好。

注意，任何时候都可以直接删除整个`node_modules`目录，因为用`npm install`命令可以完整地重新下载所有依赖。并且，这个目录不应该被放入版本控制中。

现在，我们的工程结构如下：

```
hello-koa/
|
+- .vscode/
|  |
|  +- launch.json <-- VSCode 配置文件
|
+- app.js <-- 使用koa的js
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
```

紧接着，我们在`package.json`中添加依赖包：

```
"dependencies": {
    "koa": "2.0.0"
}
```

然后使用`npm install`命令安装后，在VS Code中执行`app.js`，调试控制台输出如下：

```
node --debug-brk=40645 --nolazy app.js
Debugger listening on port 40645
app started at port 3000...
```

我们打开浏览器，输入`http://localhost:3000`，即可看到效果：

还可以直接用命令`node app.js`在命令行启动程序，或者用`npm start`启动。`npm start`命令会让npm执行定义在`package.json`文件中的start对应命令：

```
"scripts": {
    "start": "node app.js"
}
```

koa middleware

让我们再仔细看看koa的执行逻辑。核心代码是：

```
app.use(async (ctx, next) => {
    await next();
    ctx.response.type = 'text/html';
    ctx.response.body = '<h1>Hello, koa2!</h1>';
});
```

每收到一个http请求，koa就会调用通过`app.use()`注册的async函数，并传入`ctx`和`next`参数。

我们可以对`ctx`操作，并设置返回内容。但是为什么要调用`await next()`？

原因是koa把很多async函数组成一个处理链，每个async函数都可以做一些自己的事情，然后用`await next()`来调用下一个async函数。我们把每个async函数称为middleware，这些middleware可以组合起来，完成很多有用的功能。

例如，可以用以下3个middleware组成处理链，依次打印日志，记录处理时间，输出HTML：

```
app.use(async (ctx, next) => {
    console.log(`${ctx.request.method}
${ctx.request.url}`); // 打印URL
    await next(); // 调用下一个middleware
});

app.use(async (ctx, next) => {
    const start = new Date().getTime(); // 当前时间
    await next(); // 调用下一个middleware
    const ms = new Date().getTime() - start; // 耗费时间
    console.log(`Time: ${ms}ms`); // 打印耗费时间
});

app.use(async (ctx, next) => {
    await next();
    ctx.response.type = 'text/html';
    ctx.response.body = '<h1>Hello, koa2!</h1>';
});
```

middleware的顺序很重要，也就是调用`app.use()`的顺序决定了middleware的顺序。

此外，如果一个middleware没有调用`await next()`，会怎么办？答案是后续的middleware将不再执行了。这种情况也很常见，例如，一个检测用户权限的middleware可以决定是否继续处理请求，还是直接返回403错误：

```
app.use(async (ctx, next) => {
    if (await checkUserPermission(ctx)) {
        await next();
    } else {
        ctx.response.status = 403;
    }
});
```

理解了middleware，我们就已经会用koa了！

最后注意`ctx`对象有一些简写的方法，例如`ctx.url`相当于`ctx.request.url`，`ctx.type`相当于`ctx.response.type`。

处理URL

在hello-koa工程中，我们处理http请求一律返回相同的HTML，这样虽然非常简单，但是用浏览器一测，随便输入任何URL都会返回相同的网页。

正常情况下，我们应该对不同的URL调用不同的处理函数，这样才能返回不同的结果。例如像这样写：

```
app.use(async (ctx, next) => {
    if (ctx.request.path === '/') {
        ctx.response.body = 'index page';
    } else {
        await next();
    }
});

app.use(async (ctx, next) => {
    if (ctx.request.path === '/test') {
        ctx.response.body = 'TEST page';
    } else {
        await next();
    }
});

app.use(async (ctx, next) => {
    if (ctx.request.path === '/error') {
        ctx.response.body = 'ERROR page';
    } else {
        await next();
    }
});
```

这么写是可以运行的，但是好像有点蠢。

应该有一个能集中处理URL的middleware，它根据不同的URL调用不同的处理函数，这样，我们才能专心为每个URL编写处理函数。

koa-router

为了处理URL，我们需要引入`koa-router`这个middleware，让它负责处理URL映射。

我们把上一节的`hello-koa`工程复制一份，重命名为`url-koa`。

先在`package.json`中添加依赖项：

```
"koa-router": "7.0.0"
```

然后用`npm install`安装。

接下来，我们修改`app.js`，使用`koa-router`来处理URL：

```
const Koa = require('koa');

// 注意require('koa-router')返回的是函数:
const router = require('koa-router')();

const app = new Koa();

// log request URL:
app.use(async (ctx, next) => {
    console.log(`Process ${ctx.request.method}
${ctx.request.url}...`);
    await next();
});

// add url-route:
router.get('/hello/:name', async (ctx, next) => {
    var name = ctx.params.name;
    ctx.response.body = `<h1>Hello, ${name}!</h1>`;
});

router.get('/', async (ctx, next) => {
    ctx.response.body = '<h1>Index</h1>';
});

// add router middleware:
app.use(router.routes());

app.listen(3000);
console.log('app started at port 3000...');
```

注意导入`koa-router`的语句最后的`()`是函数调用：

```
const router = require('koa-router')();
```

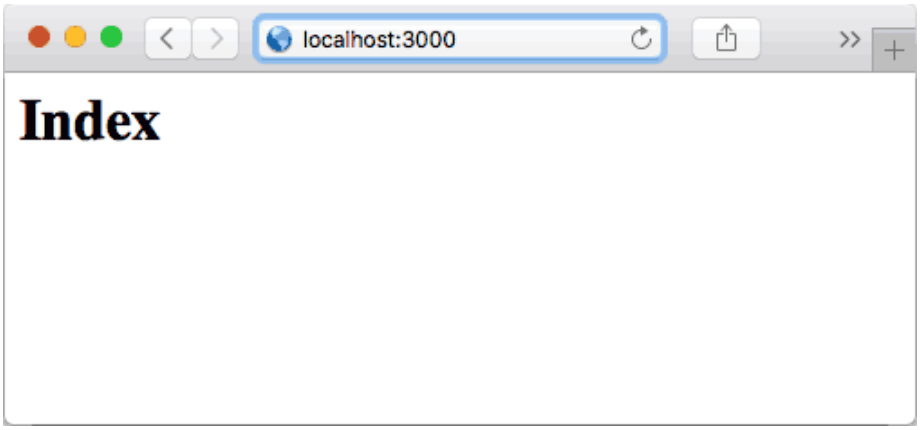
相当于：

```
const fn_router = require('koa-router');
const router = fn_router();
```

然后，我们使用`router.get('/path', async fn)`来注册一个GET请求。可以在请求路径中使用带变量的`/hello/:name`，变量可以通过`ctx.params.name`访问。

再运行`app.js`，我们就可以测试不同的URL：

输入首页：http://localhost:3000/



输入：http://localhost:3000/hello/koa

处理post请求

用`router.get('/path', async fn)`处理的是get请求。如果要处理post请求，可以用`router.post('/path', async fn)`。

用post请求处理URL时，我们会遇到一个问题：post请求通常会发送一个表单，或者JSON，它作为request的body发送，但无论是Node.js提供的原始request对象，还是koa提供的request对象，都*不提供*解析request的body的功能！

所以，我们又需要引入另一个middleware来解析原始request请求，然后，把解析后的参数，绑定到`ctx.request.body`中。

`koa-bodyparser`就是用来干这个活的。

我们在`package.json`中添加依赖项：

```
"koa-bodyparser": "3.2.0"
```

然后使用`npm install`安装。

下面，修改`app.js`，引入`koa-bodyparser`：

```
const bodyParser = require('koa-bodyparser');
```

在合适的位置加上：

```
app.use(bodyParser());
```

由于middleware的顺序很重要，这个`koa-bodyparser`必须在`router`之前被注册到`app`对象上。

现在我们就可以处理post请求了。写一个简单的登录表单：

```
router.get('/', async (ctx, next) => {
    ctx.response.body = `<h1>Index</h1>
        <form action="/signin" method="post">
            <p>Name: <input name="name" value="koa"></p>
            <p>Password: <input name="password"
type="password"></p>
            <p><input type="submit" value="Submit"></p>
        </form>`;
});

router.post('/signin', async (ctx, next) => {
    var
        name = ctx.request.body.name || '',
        password = ctx.request.body.password || '';
    console.log(`signin with name: ${name}, password:
${password}`);
    if (name === 'koa' && password === '12345') {
        ctx.response.body = `<h1>Welcome, ${name}!</h1>`;
    } else {
        ctx.response.body = `<h1>Login failed!</h1>
        <p><a href="/">Try again</a></p>`;
    }
});
```

注意到我们用`var name = ctx.request.body.name || ''`拿到表单的`name`字段，如果该字段不存在，默认值设置为`''`。

类似的，put、delete、head请求也可以由router处理。

重构

现在，我们已经可以处理不同的URL了，但是看看`app.js`，总觉得还是有点不对劲。

所有的URL处理函数都放到`app.js`里显得很乱，而且，每加一个URL，就需要修改`app.js`。随着URL越来越多，`app.js`就会越来越长。

如果能把URL处理函数集中到某个js文件，或者某几个js文件中就好了，然后让`app.js`自动导入所有处理URL的函数。这样，代码一分离，逻辑就显得清楚了。最好是这样：

```
url2-koa/
|
+- .vscode/
|  |
|  +- launch.json <-- VSCode 配置文件
|
+- controllers/
|  |
|  +- login.js <-- 处理login相关URL
|  |
|  +- users.js <-- 处理用户管理相关URL
|
+- app.js <-- 使用koa的js
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
```

于是我们把`url-koa`复制一份，重命名为`url2-koa`，准备重构这个项目。

我们先在`controllers`目录下编写`index.js`：

```
var fn_index = async (ctx, next) => {
    ctx.response.body = `<h1>Index</h1>
        <form action="/signin" method="post">
            <p>Name: <input name="name" value="koa"></p>
```

```
            <p>Password: <input name="password"
type="password"></p>
            <p><input type="submit" value="Submit"></p>
        </form>`;
};

var fn_signin = async (ctx, next) => {
    var
        name = ctx.request.body.name || '',
        password = ctx.request.body.password || '';
    console.log(`signin with name: ${name}, password:
${password}`);
    if (name === 'koa' && password === '12345') {
        ctx.response.body = `<h1>Welcome, ${name}!</h1>`;
    } else {
        ctx.response.body = `<h1>Login failed!</h1>
        <p><a href="/">Try again</a></p>`;
    }
};

module.exports = {
    'GET /': fn_index,
    'POST /signin': fn_signin
};
```

这个`index.js`通过`module.exports`把两个URL处理函数暴露出来。

类似的，`hello.js`把一个URL处理函数暴露出来：

```
var fn_hello = async (ctx, next) => {
    var name = ctx.params.name;
    ctx.response.body = `<h1>Hello, ${name}!</h1>`;
};

module.exports = {
    'GET /hello/:name': fn_hello
};
```

现在，我们修改`app.js`，让它自动扫描`controllers`目录，找到所有`js`文件，导入，然后注册每个URL：

```
// 先导入fs模块，然后用readdirSync列出文件
// 这里可以用sync是因为启动时只运行一次，不存在性能问题:
var files = fs.readdirSync(__dirname + '/controllers');

// 过滤出.js文件:
var js_files = files.filter((f)=>{
    return f.endsWith('.js');
});
```

```
// 处理每个js文件:
for (var f of js_files) {
    console.log(`process controller: ${f}...`);
    // 导入js文件:
    let mapping = require(__dirname + '/controllers/' +
f);
    for (var url in mapping) {
        if (url.startsWith('GET ')) {
            // 如果url类似"GET xxx":
            var path = url.substring(4);
            router.get(path, mapping[url]);
            console.log(`register URL mapping: GET
${path}`);
        } else if (url.startsWith('POST ')) {
            // 如果url类似"POST xxx":
            var path = url.substring(5);
            router.post(path, mapping[url]);
            console.log(`register URL mapping: POST
${path}`);
        } else {
            // 无效的URL:
            console.log(`invalid URL: ${url}`);
        }
    }
}
```

如果上面的大段代码看起来还是有点费劲，那就把它拆成更小单元的函数：

```
function addMapping(router, mapping) {
    for (var url in mapping) {
        if (url.startsWith('GET ')) {
            var path = url.substring(4);
            router.get(path, mapping[url]);
            console.log(`register URL mapping: GET
${path}`);
        } else if (url.startsWith('POST ')) {
            var path = url.substring(5);
            router.post(path, mapping[url]);
            console.log(`register URL mapping: POST
${path}`);
        } else {
            console.log(`invalid URL: ${url}`);
        }
    }
}

function addControllers(router) {
    var files = fs.readdirSync(__dirname +
'/controllers');
    var js_files = files.filter((f) => {
        return f.endsWith('.js');
```

```
    });

    for (var f of js_files) {
        console.log(`process controller: ${f}...`);
        let mapping = require(__dirname + '/controllers/'
+ f);
        addMapping(router, mapping);
    }
}

addControllers(router);
```

确保每个函数功能非常简单，一眼能看明白，是代码可维护的关键。

Controller Middleware

最后，我们把扫描`controllers`目录和创建`router`的代码从`app.js`中提取出来，作为一个简单的middleware使用，命名为`controller.js`：

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

function addMapping(router, mapping) {
    ...
}

function addControllers(router, dir) {
    ...
}

module.exports = function (dir) {
    let
        controllers_dir = dir || 'controllers', // 如果不传
参数，扫描目录默认为'controllers'
        router = require('koa-router')();
    addControllers(router, controllers_dir);
    return router.routes();
};
```

这样一来，我们在`app.js`的代码又简化了：

```
...

// 导入controller middleware:
const controller = require('./controller');

...

// 使用middleware:
app.use(controller());

...
```

经过重新整理后的工程`url2-koa`目前具备非常好的模块化，所有处理URL的函数按功能组存放在`controllers`目录，今后我们也只需要不断往这个目录下加东西就可以了，`app.js`保持不变。

使用Nunjucks

Nunjucks

Nunjucks是什么东东？其实它是一个模板引擎。

那什么是模板引擎？

模板引擎就是基于模板配合数据构造出字符串输出的一个组件。比如下面的函数就是一个模板引擎：

```
function examResult (data) {
    return `${data.name}同学一年级期末考试语文${data.chinese}分，数学${data.math}分，位于年级第${data.ranking}名。`
}
```

如果我们输入数据如下：

```
examResult({
    name: '小明',
    chinese: 78,
    math: 87,
    ranking: 999
});
```

该模板引擎把模板字符串里面对应的变量替换以后，就可以得到以下输出：

```
小明同学一年级期末考试语文78分，数学87分，位于年级第999名。
```

模板引擎最常见的输出就是输出网页，也就是HTML文本。当然，也可以输出任意格式的文本，比如Text，XML，Markdown等等。

有同学要问了：既然JavaScript的模板字符串可以实现模板功能，那为什么我们还需要另外的模板引擎？

因为JavaScript的模板字符串必须写在JavaScript代码中，要想写出新浪首页这样复杂的页面，是非常困难的。

输出HTML有几个特别重要的问题需要考虑：

转义

对特殊字符要转义，避免受到XSS攻击。比如，如果变量`name`的值不是`小明`，而是`小明...`，模板引擎输出的HTML到了浏览器，就会自动执行恶意JavaScript代码。

格式化

对不同类型的变量要格式化，比如，货币需要变成`12,345.00`这样的格式，日期需要变成`2016-01-01`这样的格式。

简单逻辑

模板还需要能执行一些简单逻辑，比如，要按条件输出内容，需要if实现如下输出：

```
{{ name }}同学，
{% if score >= 90 %}
    成绩优秀，应该奖励
{% elif score >=60 %}
    成绩良好，继续努力
{% else %}
    不及格，建议回家打屁股
{% endif %}
```

所以，我们需要一个功能强大的模板引擎，来完成页面输出的功能。

Nunjucks

我们选择Nunjucks作为模板引擎。Nunjucks是Mozilla开发的一个纯JavaScript编写的模板引擎，既可以用在Node环境下，又可以运行在浏览器端。但是，主要还是运行在Node环境下，因为浏览器端有更好的模板解决方案，例如MVVM框架。

如果你使用过Python的模板引擎jinja2，那么使用Nunjucks就非常简单，两者的语法几乎是一模一样的，因为Nunjucks就是用JavaScript重新实现了jinjia2。

从上面的例子我们可以看到，虽然模板引擎内部可能非常复杂，但是使用一个模板引擎是非常简单的，因为本质上我们只需要构造这样一个函数：

```
function render(view, model) {
    // TODO:...
}
```

其中，`view`是模板的名称（又称为视图），因为可能存在多个模板，需要选择其中一个。`model`就是数据，在JavaScript中，它就是一个简单的Object。`render`函数返回一个字符串，就是模板的输出。

下面我们来使用Nunjucks这个模板引擎来编写几个HTML模板，并且用实际数据来渲染模板并获得最终的HTML输出。

我们创建一个`use-nunjucks`的VS Code工程结构如下：

```
use-nunjucks/
|
+- .vscode/
|  |
|  +- launch.json <-- VSCode 配置文件
|
+- views/
|  |
|  +- hello.html <-- HTML模板文件
|
+- app.js <-- 入口js
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
```

其中，模板文件存放在`views`目录中。

我们先在`package.json`中添加`nunjucks`的依赖：

```
"nunjucks": "2.4.2"
```

注意，模板引擎是可以独立使用的，并不需要依赖koa。用`npm install`安装所有依赖包。

紧接着，我们要编写使用Nunjucks的函数`render`。怎么写？方法是查看Nunjucks的官方文档，仔细阅读后，在`app.js`中编写代码如下：

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

function createEnv(path, opts) {
    var
        autoescape = opts.autoescape === undefined ? true
: opts.autoescape,
        noCache = opts.noCache || false,
        watch = opts.watch || false,
        throwOnUndefined = opts.throwOnUndefined || false,
        env = new nunjucks.Environment(
            new nunjucks.FileSystemLoader('views', {
                noCache: noCache,
                watch: watch,
            }), {
```

```
                autoescape: autoescape,
                throwOnUndefined: throwOnUndefined
            });
    if (opts.filters) {
        for (var f in opts.filters) {
            env.addFilter(f, opts.filters[f]);
        }
    }
    return env;
}

var env = createEnv('views', {
    watch: true,
    filters: {
        hex: function (n) {
            return '0x' + n.toString(16);
        }
    }
});
```

变量`env`就表示Nunjucks模板引擎对象，它有一个`render(view, model)`方法，正好传入`view`和`model`两个参数，并返回字符串。

创建`env`需要的参数可以查看文档获知。我们用`autoescape = opts.autoescape && true`这样的代码给每个参数加上默认值，最后使用`new nunjucks.FileSystemLoader('views')`创建一个文件系统加载器，从`views`目录读取模板。

我们编写一个`hello.html`模板文件，放到`views`目录下，内容如下：

```
<h1>Hello {{ name }}</h1>
```

然后，我们就可以用下面的代码来渲染这个模板：

```
var s = env.render('hello.html', { name: '小明' });
console.log(s);
```

获得输出如下：

```
<h1>Hello 小明</h1>
```

咋一看，这和使用JavaScript模板字符串没啥区别嘛。不过，试试：

```
var s = env.render('hello.html', { name:
'<script>alert("小明")</script>' });
console.log(s);
```

获得输出如下：

```
<h1>Hello &lt;script&gt;alert("小明")&lt;/script&gt;</h1>
```

这样就避免了输出恶意脚本。

此外，可以使用Nunjucks提供的功能强大的tag，编写条件判断、循环等功能，例如：

```
<!-- 循环输出名字 -->
<body>
    <h3>Fruits List</h3>
    {% for f in fruits %}
    <p>{{ f }}</p>
    {% endfor %}
</body>
```

Nunjucks模板引擎最强大的功能在于模板的继承。仔细观察各种网站可以发现，网站的结构实际上是类似的，头部、尾部都是固定格式，只有中间页面部分内容不同。如果每个模板都重复头尾，一旦要修改头部或尾部，那就需要改动所有模板。

更好的方式是使用继承。先定义一个基本的网页框架`base.html`：

```
<html><body>
{% block header %} <h3>Unnamed</h3> {% endblock %}
{% block body %} <div>No body</div> {% endblock %}
{% block footer %} <div>copyright</div> {% endblock %}
</body>
```

`base.html`定义了三个可编辑的块，分别命名为`header`、`body`和`footer`。子模板可以有选择地对块进行重新定义：

```
{% extends 'base.html' %}

{% block header %}<h1>{{ header }}</h1>{% endblock %}

{% block body %}<p>{{ body }}</p>{% endblock %}
```

然后，我们对子模板进行渲染：

```
console.log(env.render('extend.html', {
    header: 'Hello',
    body: 'bla bla bla...'
}));
```

输出HTML如下：

```
<html><body>
<h1>Hello</h1>
<p>bla bla bla...</p>
<div>copyright</div> <-- footer没有重定义，所以仍使用父模板的内容
</body>
```

性能

最后我们要考虑一下Nunjucks的性能。

对于模板渲染本身来说，速度是非常非常快的，因为就是拼字符串嘛，纯CPU操作。

性能问题主要出现在从文件读取模板内容这一步。这是一个IO操作，在Node.js环境中，我们知道，单线程的JavaScript最不能忍受的就是同步IO，但Nunjucks默认就使用同步IO读取模板文件。

好消息是Nunjucks会缓存已读取的文件内容，也就是说，模板文件最多读取一次，就会放在内存中，后面的请求是不会再次读取文件的，只要我们指定了`noCache: false`这个参数。

在开发环境下，可以关闭cache，这样每次重新加载模板，便于实时修改模板。在生产环境下，一定要打开cache，这样就不会有性能问题。

Nunjucks也提供了异步读取的方式，但是这样写起来很麻烦，有简单的写法我们就不会考虑复杂的写法。保持代码简单是可维护性的关键。

使用MVC

MVC

我们已经可以用koa处理不同的URL，还可以用Nunjucks渲染模板。现在，是时候把这两者结合起来了！

当用户通过浏览器请求一个URL时，koa将调用某个异步函数处理该URL。在这个异步函数内部，我们用一行代码：

```
ctx.render('home.html', { name: 'Michael' });
```

通过Nunjucks把数据用指定的模板渲染成HTML，然后输出给浏览器，用户就可以看到渲染后的页面了：

这就是传说中的MVC：Model-View-Controller，中文名“模型-视图-控制器”。

异步函数是C：Controller，Controller负责业务逻辑，比如检查用户名是否存在，取出用户信息等等；

包含变量`{{ name }}`的模板就是V：View，View负责显示逻辑，通过简单地替换一些变量，View最终输出的就是用户看到的HTML。

MVC中的Model在哪？Model是用来传给View的，这样View在替换变量的时候，就可以从Model中取出相应的数据。

上面的例子中，Model就是一个JavaScript对象：

```
{ name: 'Michael' }
```

下面，我们根据原来的`url2-koa`创建工程`view-koa`，把koa2、Nunjucks整合起来，然后，把原来直接输出字符串的方式，改为`ctx.render(view, model)`的方式。

工程`view-koa`结构如下：

```
view-koa/
|
+- .vscode/
|  |
|  +- launch.json <-- VSCode 配置文件
|
+- controllers/ <-- Controller
|
+- views/ <-- html模板文件
|
+- static/ <-- 静态资源文件
```

```
|
+- controller.js <-- 扫描注册Controller
|
+- app.js <-- 使用koa的js
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
```

在`package.json`中，我们将要用到的依赖包有：

```
"koa": "2.0.0",
"koa-bodyparser": "3.2.0",
"koa-router": "7.0.0",
"nunjucks": "2.4.2",
"mime": "1.3.4",
"mz": "2.4.0"
```

先用`npm install`安装依赖包。

然后，我们准备编写以下两个Controller：

处理首页 GET /

我们定义一个async函数处理首页URL`/`：

```
async (ctx, next) => {
    ctx.render('index.html', {
        title: 'Welcome'
    });
}
```

注意到koa并没有在`ctx`对象上提供`render`方法，这里我们假设应该这么使用，这样，我们在编写Controller的时候，最后一步调用`ctx.render(view, model)`就完成了页面输出。

处理登录请求 POST /signin

我们再定义一个async函数处理登录请求`/signin`：

```
async (ctx, next) => {
    var
        email = ctx.request.body.email || '',
        password = ctx.request.body.password || '';
    if (email === 'admin@example.com' && password ===
'123456') {
        // 登录成功:
        ctx.render('signin-ok.html', {
            title: 'Sign In OK',
            name: 'Mr Node'
```

```
        });
    } else {
        // 登录失败:
        ctx.render('signin-failed.html', {
            title: 'Sign In Failed'
        });
    }
}
```

由于登录请求是一个POST，我们就用`ctx.request.body.`拿到POST请求的数据，并给一个默认值。

登录成功时我们用`signin-ok.html`渲染，登录失败时我们用`signin-failed.html`渲染，所以，我们一共需要以下3个View：

- index.html
- signin-ok.html
- signin-failed.html

编写View

在编写View的时候，我们实际上是在编写HTML页。为了让页面看起来美观大方，使用一个现成的CSS框架是非常有必要的。我们用Bootstrap这个CSS框架。从首页下载zip包后解压，我们把所有静态资源文件放到`/static`目录下：

```
view-koa/
|
+- static/
   |
   +- css/ <- 存放bootstrap.css等
   |
   +- fonts/ <- 存放字体文件
   |
   +- js/ <- 存放bootstrap.js等
```

这样我们在编写HTML的时候，可以直接用Bootstrap的CSS，像这样：

```
<link rel="stylesheet" href="/static/css/bootstrap.css">
```

现在，在使用MVC之前，第一个问题来了，如何处理静态文件？

我们把所有静态资源文件全部放入`/static`目录，目的就是能统一处理静态文件。在koa中，我们需要编写一个middleware，处理以`/static/`开头的URL。

编写middleware

我们来编写一个处理静态文件的middleware。编写middleware实际上一点也不复杂。我们先创建一个`static-files.js`的文件，编写一个能处理静态文件的middleware：

```
```

```
const path = require('path');
const mime = require('mime');
const fs = require('mz/fs');

// url: 类似 '/static/'
// dir: 类似 __dirname + '/static'
function staticFiles(url, dir) {
    return async (ctx, next) => {
        let rpath = ctx.request.path;
        // 判断是否以指定的url开头:
        if (rpath.startsWith(url)) {
            // 获取文件完整路径:
            let fp = path.join(dir,
rpath.substring(url.length));
            // 判断文件是否存在:
            if (await fs.exists(fp)) {
                // 查找文件的mime:
                ctx.response.type = mime.lookup(rpath);
                // 读取文件内容并赋值给response.body:
                ctx.response.body = await fs.readFile(fp);
            } else {
                // 文件不存在:
                ctx.response.status = 404;
            }
        } else {
            // 不是指定前缀的URL，继续处理下一个middleware:
            await next();
        }
    };
}

module.exports = staticFiles;
```

`staticFiles`是一个普通函数，它接收两个参数：URL前缀和一个目录，然后返回一个async函数。这个async函数会判断当前的URL是否以指定前缀开头，如果是，就把URL的路径视为文件，并发送文件内容。如果不是，这个async函数就不做任何事情，而是简单地调用`await next()`让下一个middleware去处理请求。

我们使用了一个`mz`的包，并通过`require('mz/fs');`导入。`mz`提供的API和Node.js的`fs`模块完全相同，但`fs`模块使用回调，而`mz`封装了`fs`对应的函数，并改为Promise。这样，我们就可以非常简单的用`await`调用`mz`的函数，而不需要任何回调。

所有的第三方包都可以通过npm官网搜索并查看其文档：

https://www.npmjs.com/

最后，这个middleware使用起来也很简单，在`app.js`里加一行代码：

```
let staticFiles = require('./static-files');
app.use(staticFiles('/static/', __dirname + '/static'));
```

注意：也可以去npm搜索能用于koa2的处理静态文件的包并直接使用。

集成Nunjucks

集成Nunjucks实际上也是编写一个middleware，这个middleware的作用是给`ctx`对象绑定一个`render(view, model)`的方法，这样，后面的Controller就可以调用这个方法来渲染模板了。

我们创建一个`templating.js`来实现这个middleware：

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

function createEnv(path, opts) {
    var
        autoescape = opts.autoescape === undefined ? true
: opts.autoescape,
        noCache = opts.noCache || false,
        watch = opts.watch || false,
        throwOnUndefined = opts.throwOnUndefined || false,
        env = new nunjucks.Environment(
            new nunjucks.FileSystemLoader(path || 'views',
{
                noCache: noCache,
                watch: watch,
            }), {
                autoescape: autoescape,
                throwOnUndefined: throwOnUndefined
            });
    if (opts.filters) {
        for (var f in opts.filters) {
            env.addFilter(f, opts.filters[f]);
        }
    }
    return env;
}

function templating(path, opts) {
    // 创建Nunjucks的env对象:
    var env = createEnv(path, opts);
    return async (ctx, next) => {
        // 给ctx绑定render函数:
        ctx.render = function (view, model) {
            // 把render后的内容赋值给response.body:
            ctx.response.body = env.render(view,
Object.assign({}, ctx.state || {}, model || {}));
            // 设置Content-Type:
            ctx.response.type = 'text/html';
        };
```

```
        // 继续处理请求:
        await next();
    };
}

module.exports = templating;
```

注意到`createEnv()`函数和前面使用Nunjucks时编写的函数是一模一样的。我们主要关心`tempating()`函数，它会返回一个middleware，在这个middleware中，我们只给`ctx`“安装”了一个`render()`函数，其他什么事情也没干，就继续调用下一个middleware。

使用的时候，我们在`app.js`添加如下代码：

```
const isProduction = process.env.NODE_ENV ===
'production';

app.use(templating('views', {
    noCache: !isProduction,
    watch: !isProduction
}));
```

这里我们定义了一个常量`isProduction`，它判断当前环境是否是production环境。如果是，就使用缓存，如果不是，就关闭缓存。在开发环境下，关闭缓存后，我们修改View，可以直接刷新浏览器看到效果，否则，每次修改都必须重启Node程序，会极大地降低开发效率。

Node.js在全局变量`process`中定义了一个环境变量`env.NODE_ENV`，为什么要使用该环境变量？因为我们在开发的时候，环境变量应该设置为`'development'`，而部署到服务器时，环境变量应该设置为`'production'`。在编写代码的时候，要根据当前环境作不同的判断。

注意：生产环境上必须配置环境变量`NODE_ENV = 'production'`，而开发环境不需要配置，实际上`NODE_ENV`可能是`undefined`，所以判断的时候，不要用`NODE_ENV === 'development'`。

类似的，我们在使用上面编写的处理静态文件的middleware时，也可以根据环境变量判断：

```
if (! isProduction) {
    let staticFiles = require('./static-files');
    app.use(staticFiles('/static/', __dirname +
'/static'));
}
```

这是因为在生产环境下，静态文件是由部署在最前面的反向代理服务器（如Nginx）处理的，Node程序不需要处理静态文件。而在开发环境下，我们希望koa能顺带处理静态文件，否则，就必须手动配置一个反向代理服务器，这样会导致开发环境非常复杂。

编写View

在编写View的时候，非常有必要先编写一个`base.html`作为骨架，其他模板都继承自`base.html`，这样，才能大大减少重复工作。

编写HTML不在本教程的讨论范围之内。这里我们参考Bootstrap的官网简单编写了`base.html`。

运行

一切顺利的话，这个`view-koa`工程应该可以顺利运行。运行前，我们再检查一下`app.js`里的middleware的顺序：

第一个middleware是记录URL以及页面执行时间：

```
app.use(async (ctx, next) => {
    console.log(`Process ${ctx.request.method}
${ctx.request.url}...`);
    var
        start = new Date().getTime(),
        execTime;
    await next();
    execTime = new Date().getTime() - start;
    ctx.response.set('X-Response-Time', `${execTime}ms`);
});
```

第二个middleware处理静态文件：

```
if (! isProduction) {
    let staticFiles = require('./static-files');
    app.use(staticFiles('/static/', __dirname +
'/static'));
}
```

第三个middleware解析POST请求：

```
app.use(bodyParser());
```

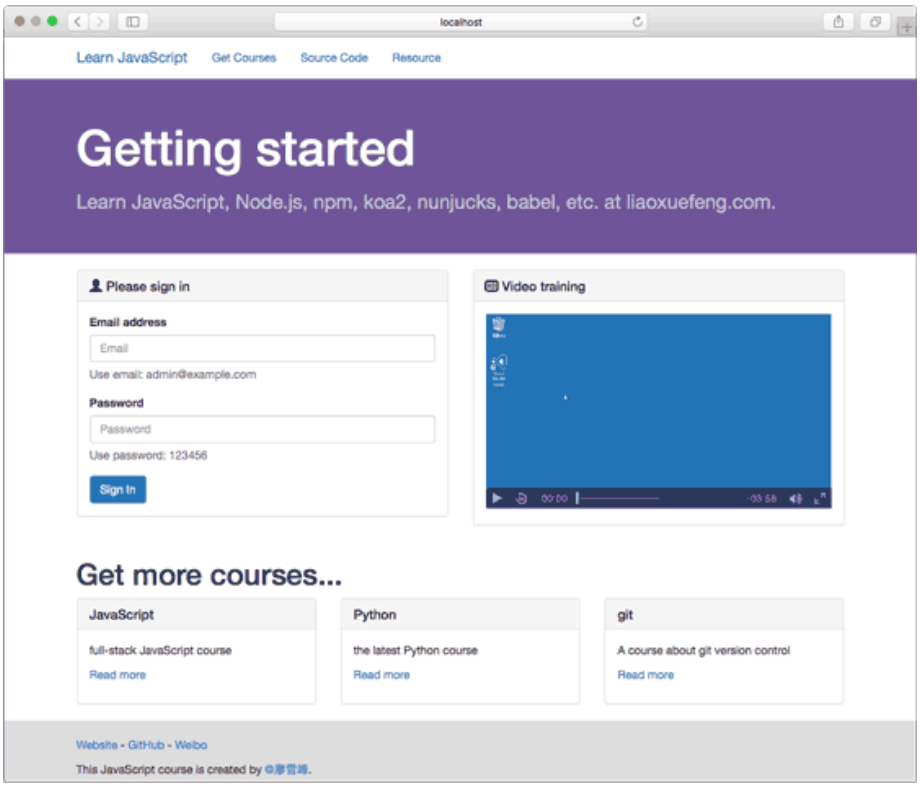
第四个middleware负责给`ctx`加上`render()`来使用Nunjucks：

```
app.use(templating('view', {
    noCache: !isProduction,
    watch: !isProduction
}));
```

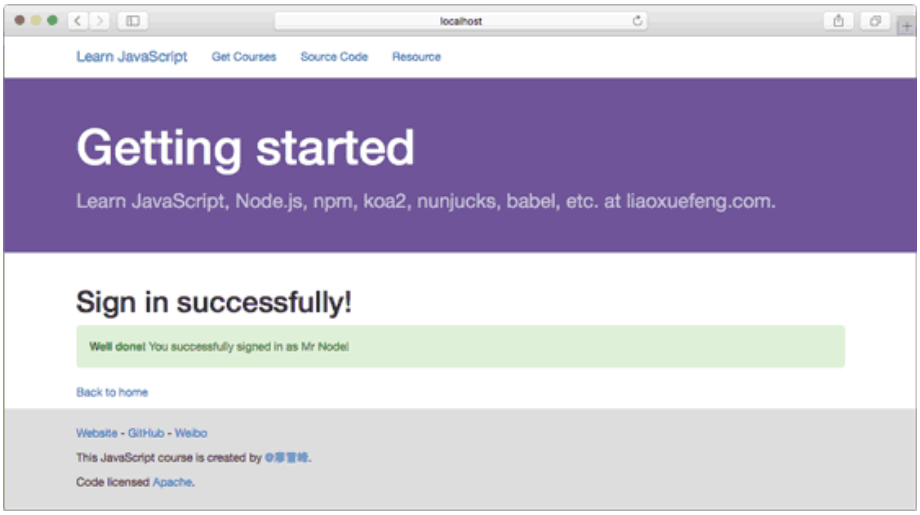
最后一个middleware处理URL路由：

```
app.use(controller());
```

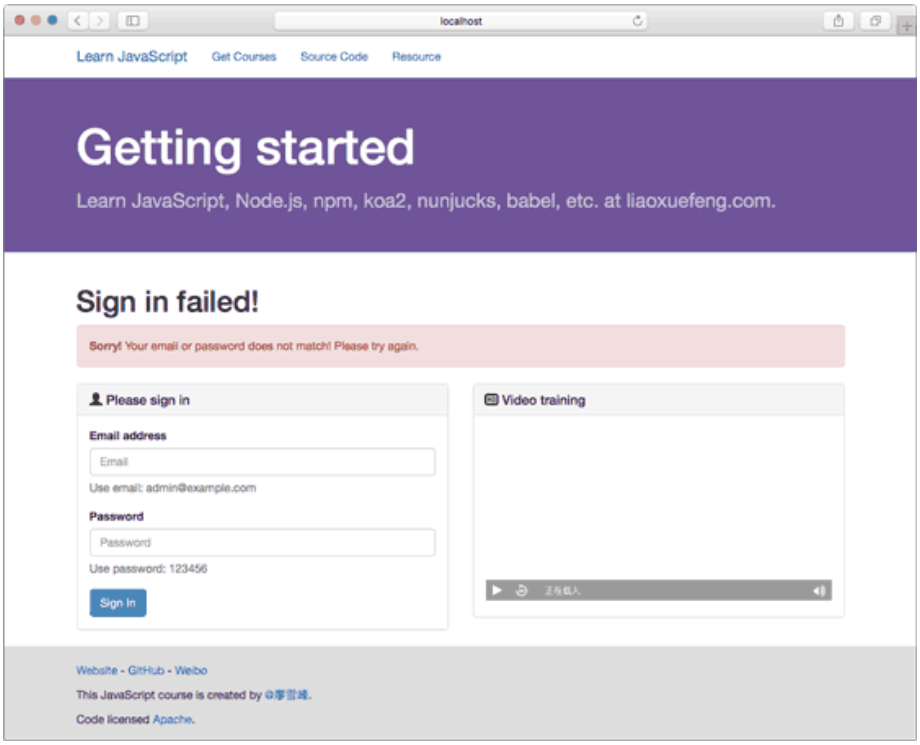
现在，在VS Code中运行代码，不出意外的话，在浏览器输入`localhost:3000/`，可以看到首页内容：



直接在首页登录，如果输入正确的Email和Password，进入登录成功的页面：



如果输入的Email和Password不正确，进入登录失败的页面：



怎么判断正确的Email和Password？目前我们在`signin.js`中是这么判断的：

```
if (email === 'admin@example.com' && password === '123456') {
    ...
}
```

当然，真实的网站会根据用户输入的Email和Password去数据库查询并判断登录是否成功，不过这需要涉及到Node.js环境如何操作数据库，我们后面再讨论。

扩展

注意到`ctx.render`内部渲染模板时，Model对象并不是传入的model变量，而是：

```
Object.assign({}, ctx.state || {}, model || {})
```

这个小技巧是为了扩展。

首先，`model || {}`确保了即使传入`undefined`，model也会变为默认值`{}`。`Object.assign()`会把除第一个参数外的其他参数的所有属性复制到第一个参数中。第二个参数是`ctx.state || {}`，这个目的是为了能把一些公共的变量放入`ctx.state`并传给View。

例如，某个middleware负责检查用户权限，它可以把当前用户放入`ctx.state`中：

```
app.use(async (ctx, next) => {
    var user = tryGetUserFromCookie(ctx.request);
    if (user) {
        ctx.state.user = user;
        await next();
    } else {
        ctx.response.status = 403;
    }
});
```

这样就没有必要在每个Controller的async函数中都把user变量放入model中。

mysql

访问数据库

程序运行的时候，数据都是在内存中的。当程序终止的时候，通常都需要将数据保存到磁盘上，无论是保存到本地磁盘，还是通过网络保存到服务器上，最终都会将数据写入磁盘文件。

而如何定义数据的存储格式就是一个大问题。如果我们自己来定义存储格式，比如保存一个班级所有学生的成绩单：

名字	成绩
Michael	99
Bob	85
Bart	59
Lisa	87

你可以用一个文本文件保存，一行保存一个学生，用`,`隔开：

```
Michael,99
Bob,85
Bart,59
Lisa,87
```

你还可以用JSON格式保存，也是文本文件：

```
[
    {"name":"Michael","score":99},
    {"name":"Bob","score":85},
    {"name":"Bart","score":59},
    {"name":"Lisa","score":87}
]
```

你还可以定义各种保存格式，但是问题来了：

存储和读取需要自己实现，JSON还是标准，自己定义的格式就各式各样了；

不能做快速查询，只有把数据全部读到内存中才能自己遍历，但有时候数据的大小远远超过了内存（比如蓝光电影，40GB的数据），根本无法全部读入内存。

为了便于程序保存和读取数据，而且，能直接通过条件快速查询到指定的数据，就出现了数据库（Database）这种专门用于集中存储和查询的软件。

数据库软件诞生的历史非常久远，早在1950年数据库就诞生了。经历了网状数据库，层次数据库，我们现在广泛使用的关系数据库是20世纪70年代基于关系模型的基础上诞生的。

关系模型有一套复杂的数学理论，但是从概念上是十分容易理解的。举个学校的例子：

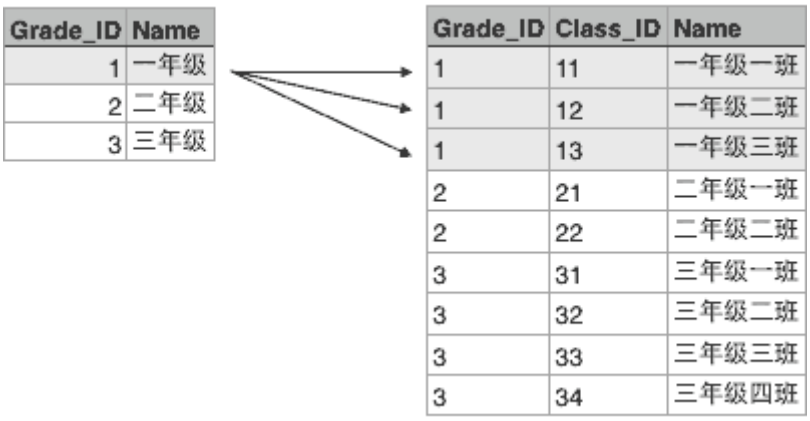
假设某个XX省YY市ZZ县第一实验小学有3个年级，要表示出这3个年级，可以在Excel中用一个表格画出来：

Grade_ID	Name
1	一年级
2	二年级
3	三年级

每个年级又有若干个班级，要把所有班级表示出来，可以在Excel中再画一个表格：

Grade_ID	Class_ID	Name
1	11	一年级一班
1	12	一年级二班
1	13	一年级三班
2	21	二年级一班
2	22	二年级二班
3	31	三年级一班
3	32	三年级二班
3	33	三年级三班
3	34	三年级四班

这两个表格有个映射关系，就是根据Grade_ID可以在班级表中查找到对应的所有班级：



Grade_ID	Name
1	一年级
2	二年级
3	三年级

Grade_ID	Class_ID	Name
1	11	一年级一班
1	12	一年级二班
1	13	一年级三班
2	21	二年级一班
2	22	二年级二班
3	31	三年级一班
3	32	三年级二班
3	33	三年级三班
3	34	三年级四班

也就是Grade表的每一行对应Class表的多行，在关系数据库中，这种基于表（Table）的一对多的关系就是关系数据库的基础。

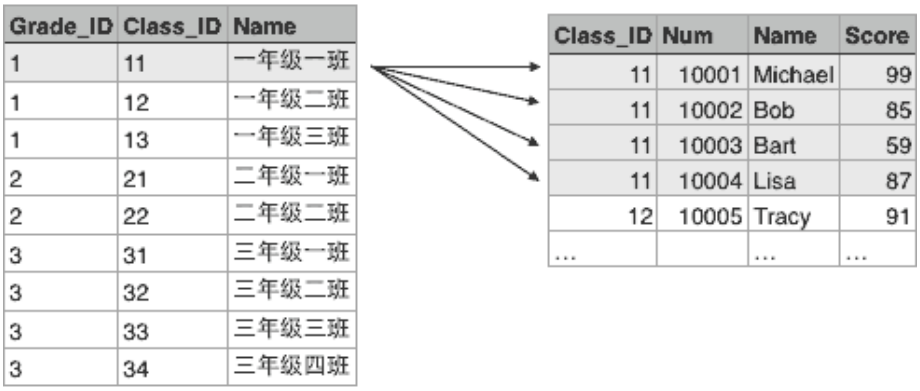
根据某个年级的ID就可以查找所有班级的行，这种查询语句在关系数据库中称为SQL语句，可以写成：

```
SELECT * FROM classes WHERE grade_id = '1';
```

结果也是一个表：

```
---------+----------+----------
grade_id | class_id | name
---------+----------+----------
1        | 11       | 一年级一班
---------+----------+----------
1        | 12       | 一年级二班
---------+----------+----------
1        | 13       | 一年级三班
---------+----------+----------
```

类似的，Class表的一行记录又可以关联到Student表的多行记录：



Grade_ID	Class_ID	Name
1	11	一年级一班
1	12	一年级二班
1	13	一年级三班
2	21	二年级一班
2	22	二年级二班
3	31	三年级一班
3	32	三年级二班
3	33	三年级三班
3	34	三年级四班

Class_ID	Num	Name	Score
11	10001	Michael	99
11	10002	Bob	85
11	10003	Bart	59
11	10004	Lisa	87
12	10005	Tracy	91
...		...	...

由于本教程不涉及到关系数据库的详细内容，如果你想从零学习关系数据库和基本的SQL语句，请参考SQL教程。

NoSQL

你也许还听说过NoSQL数据库，很多NoSQL宣传其速度和规模远远超过关系数据库，所以很多同学觉得有了NoSQL是否就不需要SQL了呢？千万不要被他们忽悠了，连SQL都不明白怎么可能搞明白NoSQL呢？

数据库类别

既然我们要使用关系数据库，就必须选择一个关系数据库。目前广泛使用的关系数据库也就这么几种：

付费的商用数据库：

- Oracle，典型的高富帅；
- SQL Server，微软自家产品，Windows定制专款；
- DB2，IBM的产品，听起来挺高端；
- Sybase，曾经跟微软是好基友，后来关系破裂，现在家境惨淡。

这些数据库都是不开源而且付费的，最大的好处是花了钱出了问题可以找厂家解决，不过在Web的世界里，常常需要部署成千上万的数据库服务器，当然不能把大把大把的银子扔给厂家，所以，无论是Google、Facebook，还是国内的BAT，无一例外都选择了免费的开源数据库：

- MySQL，大家都在用，一般错不了；
- PostgreSQL，学术气息有点重，其实挺不错，但知名度没有MySQL高；
- sqlite，嵌入式数据库，适合桌面和移动应用。

作为一个JavaScript全栈工程师，选择哪个免费数据库呢？当然是MySQL。因为MySQL普及率最高，出了错，可以很容易找到解决方法。而且，围绕MySQL有一大堆监控和运维的工具，安装和使用很方便。

安装MySQL

为了能继续后面的学习，你需要从MySQL官方网站下载并安装MySQL Community Server 5.6，这个版本是免费的，其他高级版本是要收钱的（请放心，收钱的功能我们用不上）。MySQL是跨平台的，选择对应的平台下载安装文件，安装即可。

安装时，MySQL会提示输入`root`用户的口令，请务必记清楚。如果怕记不住，就把口令设置为`password`。

在Windows上，安装时请选择`UTF-8`编码，以便正确地处理中文。

在Mac或Linux上，需要编辑MySQL的配置文件，把数据库默认的编码全部改为UTF-8。MySQL的配置文件默认存放在`/etc/my.cnf`或者`/etc/mysql/my.cnf`：

```
[client]
default-character-set = utf8

[mysqld]
default-storage-engine = INNODB
character-set-server = utf8
collation-server = utf8_general_ci
```

重启MySQL后，可以通过MySQL的客户端命令行检查编码：

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor...
...

mysql> show variables like '%char%';
+--------------------------+------------------------------
--------------------------+
| Variable_name            | Value
                         |
+--------------------------+------------------------------
--------------------------+
| character_set_client     | utf8
                         |
| character_set_connection | utf8
                         |
| character_set_database   | utf8
                         |
| character_set_filesystem | binary
                         |
| character_set_results    | utf8
                         |
| character_set_server     | utf8
                         |
| character_set_system     | utf8
                         |
| character_sets_dir       | /usr/local/mysql-5.1.65-
osx10.6-x86_64/share/charsets/ |
+--------------------------+------------------------------
--------------------------+
8 rows in set (0.00 sec)
```

看到`utf8`字样就表示编码设置正确。

注：如果MySQL的版本≥5.5.3，可以把编码设置为`utf8mb4`，`utf8mb4`和`utf8`完全兼容，但它支持最新的Unicode标准，可以显示emoji字符。

使用**Sequelize**

访问MySQL

当我们安装好MySQL后，Node.js程序如何访问MySQL数据库呢？

访问MySQL数据库只有一种方法，就是通过网络发送SQL命令，然后，MySQL服务器执行后返回结果。

我们可以在命令行窗口输入`mysql -u root -p`，然后输入root口令后，就连接到了MySQL服务器。因为没有指定`--host`参数，所以我们连接到的是`localhost`，也就是本机的MySQL服务器。

在命令行窗口下，我们可以输入命令，操作MySQL服务器：

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| test               |
+--------------------+
4 rows in set (0.05 sec)
```

输入`exit`退出MySQL命令行模式。

对于Node.js程序，访问MySQL也是通过网络发送SQL命令给MySQL服务器。这个访问MySQL服务器的软件包通常称为MySQL驱动程序。不同的编程语言需要实现自己的驱动，MySQL官方提供了Java、.Net、Python、Node.js、C++和C的驱动程序，官方的Node.js驱动目前仅支持5.7以上版本，而我们上面使用的命令行程序实际上用的就是C驱动。

目前使用最广泛的MySQL Node.js驱动程序是开源的`mysql`，可以直接使用npm安装。

ORM

如果直接使用`mysql`包提供的接口，我们编写的代码就比较底层，例如，查询代码：

```
connection.query('SELECT * FROM users WHERE id = ?',
['123'], function(err, rows) {
    if (err) {
        // error
    } else {
        for (let row in rows) {
            processRow(row);
        }
    }
});
```

考虑到数据库表是一个二维表，包含多行多列，例如一个`pets`的表：

```
mysql> select * from pets;
+----+--------+------------+
| id | name   | birth      |
+----+--------+------------+
|  1 | Gaffey | 2007-07-07 |
|  2 | Odie   | 2008-08-08 |
+----+--------+------------+
2 rows in set (0.00 sec)
```

每一行可以用一个JavaScript对象表示，例如第一行：

```
{
    "id": 1,
    "name": "Gaffey",
    "birth": "2007-07-07"
}
```

这就是传说中的ORM技术：Object-Relational Mapping，把关系数据库的表结构映射到对象上。是不是很简单？

但是由谁来做这个转换呢？所以ORM框架应运而生。

我们选择Node的ORM框架Sequelize来操作数据库。这样，我们读写的都是JavaScript对象，Sequelize帮我们把对象变成数据库中的行。

用Sequelize查询`pets`表，代码像这样：

```
Pet.findAll()
    .then(function (pets) {
        for (let pet in pets) {
            console.log(`${pet.id}: ${pet.name}`);
        }
    }).catch(function (err) {
        // error
    });
```

因为Sequelize返回的对象是Promise，所以我们可以用`then()`和`catch()`分别异步响应成功和失败。

但是用`then()`和`catch()`仍然比较麻烦。有没有更简单的方法呢？

可以用ES7的await来调用任何一个Promise对象，这样我们写出来的代码就变成了：

```
var pets = await Pet.findAll();
```

真的就是这么简单！

await只有一个限制，就是必须在async函数中调用。上面的代码直接运行还差一点，我们可以改成：

```
(async () => {
    var pets = await Pet.findAll();
})();
```

考虑到koa的处理函数都是async函数，所以我们实际上将来在koa的async函数中直接写await访问数据库就可以了！

这也是为什么我们选择Sequelize的原因：只要API返回Promise，就可以用await调用，写代码就非常简单！

实战

在使用Sequlize操作数据库之前，我们先在MySQL中创建一个表来测试。我们可以在`test`数据库中创建一个`pets`表。`test`数据库是MySQL安装后自动创建的用于测试的数据库。在MySQL命令行执行下列命令：

```
grant all privileges on test.* to 'www'@'%' identified by
'www';

use test;

create table pets (
    id varchar(50) not null,
    name varchar(100) not null,
    gender bool not null,
    birth varchar(10) not null,
    createdAt bigint not null,
    updatedAt bigint not null,
    version bigint not null,
    primary key (id)
) engine=innodb;
```

第一条`grant`命令是创建MySQL的用户名和口令，均为`www`，并赋予操作`test`数据库的所有权限。

第二条`use`命令把当前数据库切换为`test`。

第三条命令创建了`pets`表。

然后，我们根据前面的工程结构创建`hello-sequelize`工程，结构如下：

```
hello-sequelize/
|
+- .vscode/
|  |
|  +- launch.json <-- VSCode 配置文件
|
+- init.txt <-- 初始化SQL命令
```

```
|
+- config.js <-- MySQL配置文件
|
+- app.js <-- 使用koa的js
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
```

然后，添加如下依赖包：

```
"sequelize": "3.24.1",
"mysql": "2.11.1"
```

注意`mysql`是驱动，我们不直接使用，但是`sequelize`会用。

用`npm install`安装。

`config.js`实际上是一个简单的配置文件：

```
var config = {
    database: 'test', // 使用哪个数据库
    username: 'www', // 用户名
    password: 'www', // 口令
    host: 'localhost', // 主机名
    port: 3306 // 端口号，MySQL默认3306
};

module.exports = config;
```

下面，我们就可以在`app.js`中操作数据库了。使用Sequelize操作MySQL需要先做两件准备工作：

第一步，创建一个sequelize对象实例：

```
const Sequelize = require('sequelize');
const config = require('./config');

var sequelize = new Sequelize(config.database,
config.username, config.password, {
    host: config.host,
    dialect: 'mysql',
    pool: {
        max: 5,
        min: 0,
        idle: 30000
    }
});
```

第二步，定义模型Pet，告诉Sequelize如何映射数据库表：

```
var Pet = sequelize.define('pet', {
    id: {
        type: Sequelize.STRING(50),
        primaryKey: true
    },
    name: Sequelize.STRING(100),
    gender: Sequelize.BOOLEAN,
    birth: Sequelize.STRING(10),
    createdAt: Sequelize.BIGINT,
    updatedAt: Sequelize.BIGINT,
    version: Sequelize.BIGINT
}, {
        timestamps: false
    });
```

用`sequelize.define()`定义Model时，传入名称`pet`，默认的表名就是`pets`。第二个参数指定列名和数据类型，如果是主键，需要更详细地指定。第三个参数是额外的配置，我们传入`{ timestamps: false }`是为了关闭Sequelize的自动添加timestamp的功能。所有的ORM框架都有一种很不好的风气，总是自作聪明地加上所谓“自动化”的功能，但是会让人感到完全摸不着头脑。

接下来，我们就可以往数据库中塞一些数据了。我们可以用Promise的方式写：

```
var now = Date.now();

Pet.create({
    id: 'g-' + now,
    name: 'Gaffey',
    gender: false,
    birth: '2007-07-07',
    createdAt: now,
    updatedAt: now,
    version: 0
}).then(function (p) {
    console.log('created.' + JSON.stringify(p));
}).catch(function (err) {
    console.log('failed: ' + err);
});
```

也可以用await写：

```
(async () => {
    var dog = await Pet.create({
        id: 'd-' + now,
        name: 'Odie',
        gender: false,
        birth: '2008-08-08',
        createdAt: now,
        updatedAt: now,
        version: 0
    });
    console.log('created: ' + JSON.stringify(dog));
})();
```

显然await代码更胜一筹。

查询数据时，用await写法如下：

```
(async () => {
    var pets = await Pet.findAll({
        where: {
            name: 'Gaffey'
        }
    });
    console.log(`find ${pets.length} pets:`);
    for (let p of pets) {
        console.log(JSON.stringify(p));
    }
})();
```

如果要更新数据，可以对查询到的实例调用`save()`方法：

```
(async () => {
    var p = await queryFromSomewhere();
    p.gender = true;
    p.updatedAt = Date.now();
    p.version ++;
    await p.save();
})();
```

如果要删除数据，可以对查询到的实例调用`destroy()`方法：

```
(async () => {
    var p = await queryFromSomewhere();
    await p.destroy();
})();
```

运行代码，可以看到Sequelize打印出的每一个SQL语句，便于我们查看：

```
Executing (default): INSERT INTO `pets`
(`id`,`name`,`gender`,`birth`,`createdAt`,`updatedAt`,`ver
sion`) VALUES ('g-1471961204219','Gaffey',false,'2007-07-
07',1471961204219,1471961204219,0);
```

Model

我们把通过`sequelize.define()`返回的`Pet`称为Model，它表示一个数据模型。

我们把通过`Pet.findAll()`返回的一个或一组对象称为Model实例，每个实例都可以直接通过`JSON.stringify`序列化为JSON字符串。但是它们和普通JSON对象相比，多了一些由Sequelize添加的方法，比如`save()`和`destroy()`。调用这些方法我们可以执行更新或者删除操作。

所以，使用Sequelize操作数据库的一般步骤就是：

首先，通过某个Model对象的`findAll()`方法获取实例；

如果要更新实例，先对实例属性赋新值，再调用`save()`方法；

如果要删除实例，直接调用`destroy()`方法。

注意`findAll()`方法可以接收`where`、`order`这些参数，这和将要生成的SQL语句是对应的。

文档

Sequelize的API可以参考官方文档。

建立Model

直接使用Sequelize虽然可以，但是存在一些问题。

团队开发时，有人喜欢自己加timestamp：

```
var Pet = sequelize.define('pet', {
    id: {
        type: Sequelize.STRING(50),
        primaryKey: true
    },
    name: Sequelize.STRING(100),
    createdAt: Sequelize.BIGINT,
    updatedAt: Sequelize.BIGINT
}, {
        timestamps: false
    });
```

有人又喜欢自增主键，并且自定义表名：

```
var Pet = sequelize.define('pet', {
    id: {
        type: Sequelize.INTEGER,
        autoIncrement: true,
        primaryKey: true
    },
    name: Sequelize.STRING(100)
}, {
        tableName: 't_pet'
    });
```

一个大型Web App通常都有几十个映射表，一个映射表就是一个Model。如果按照各自喜好，那业务代码就不好写。Model不统一，很多代码也无法复用。

所以我们需要一个统一的模型，强迫所有Model都遵守同一个规范，这样不但实现简单，而且容易统一风格。

Model

我们首先要定义的就是Model存放的文件夹必须在`models`内，并且以Model名字命名，例如：`Pet.js`，`User.js`等等。

其次，每个Model必须遵守一套规范：

1. 统一主键，名称必须是`id`，类型必须是`STRING(50)`；
2. 主键可以自己指定，也可以由框架自动生成（如果为null或undefined）；
3. 所有字段默认为`NOT NULL`，除非显式指定；
4. 统一timestamp机制，每个Model必须有`createdAt`、`updatedAt`和`version`，分别记录创建时间、修改时间和版本号。其中，`createdAt`和`updatedAt`以`BIGINT`存储时间戳，最大的好处是无需处理时区，排序方便。`version`每次修改时自增。

所以，我们不要直接使用Sequelize的API，而是通过`db.js`间接地定义Model。例如，`User.js`应该定义如下：

```
const db = require('../db');

module.exports = db.defineModel('users', {
    email: {
        type: db.STRING(100),
        unique: true
    },
    passwd: db.STRING(100),
    name: db.STRING(100),
    gender: db.BOOLEAN
});
```

这样，User就具有`email`、`passwd`、`name`和`gender`这4个业务字段。`id`、`createdAt`、`updatedAt`和`version`应该自动加上，而不是每个Model都去重复定义。

所以，`db.js`的作用就是统一Model的定义：

```
const Sequelize = require('sequelize');

console.log('init sequelize...');

var sequelize = new Sequelize('dbname', 'username',
'password', {
    host: 'localhost',
    dialect: 'mysql',
    pool: {
        max: 5,
        min: 0,
        idle: 10000
    }
});

const ID_TYPE = Sequelize.STRING(50);

function defineModel(name, attributes) {
    var attrs = {};
    for (let key in attributes) {
        let value = attributes[key];
        if (typeof value === 'object' && value['type']) {
            value.allowNull = value.allowNull || false;
            attrs[key] = value;
        } else {
            attrs[key] = {
                type: value,
                allowNull: false
            };
        }
    }
    attrs.id = {
        type: ID_TYPE,
        primaryKey: true
    };
    attrs.createdAt = {
        type: Sequelize.BIGINT,
        allowNull: false
    };
    attrs.updatedAt = {
        type: Sequelize.BIGINT,
        allowNull: false
    };
    attrs.version = {
        type: Sequelize.BIGINT,
        allowNull: false
    };
    return sequelize.define(name, attrs, {
        tableName: name,
        timestamps: false,
```

```
        hooks: {
            beforeValidate: function (obj) {
                let now = Date.now();
                if (obj.isNewRecord) {
                    if (!obj.id) {
                        obj.id = generateId();
                    }
                    obj.createdAt = now;
                    obj.updatedAt = now;
                    obj.version = 0;
                } else {
                    obj.updatedAt = Date.now();
                    obj.version++;
                }
            }
        }
    });
}
```

我们定义的`defineModel`就是为了强制实现上述规则。

Sequelize在创建、修改Entity时会调用我们指定的函数，这些函数通过`hooks`在定义Model时设定。我们在`beforeValidate`这个事件中根据是否是`isNewRecord`设置主键（如果主键为`null`或`undefined`）、设置时间戳和版本号。

这么一来，Model定义的时候就可以大大简化。

数据库配置

接下来，我们把简单的`config.js`拆成3个配置文件：

- config-default.js：存储默认的配置；
- config-override.js：存储特定的配置；
- config-test.js：存储用于测试的配置。

例如，默认的`config-default.js`可以配置如下：

```
var config = {
    dialect: 'mysql',
    database: 'nodejs',
    username: 'www',
    password: 'www',
    host: 'localhost',
    port: 3306
};

module.exports = config;
```

而`config-override.js`可应用实际配置：

```
var config = {
    database: 'production',
    username: 'www',
    password: 'secret-password',
    host: '192.168.1.199'
};

module.exports = config;
```

`config-test.js`可应用测试环境的配置：

```
var config = {
    database: 'test'
};

module.exports = config;
```

读取配置的时候，我们用`config.js`实现不同环境读取不同的配置文件：

```
const defaultConfig = './config-default.js';
// 可设定为绝对路径，如 /opt/product/config-override.js
const overrideConfig = './config-override.js';
const testConfig = './config-test.js';

const fs = require('fs');

var config = null;

if (process.env.NODE_ENV === 'test') {
    console.log(`Load ${testConfig}...`);
    config = require(testConfig);
} else {
    console.log(`Load ${defaultConfig}...`);
    config = require(defaultConfig);
    try {
        if (fs.statSync(overrideConfig).isFile()) {
            console.log(`Load ${overrideConfig}...`);
            config = Object.assign(config,
require(overrideConfig));
        }
    } catch (err) {
        console.log(`Cannot load ${overrideConfig}.`);
    }
}

module.exports = config;
```

具体的规则是：

1. 先读取`config-default.js`：

2. 如果不是测试环境，就读取`config-override.js`，如果文件不存在，就忽略。
3. 如果是测试环境，就读取`config-test.js`。

这样做的好处是，开发环境下，团队统一使用默认的配置，并且无需`config-override.js`。部署到服务器时，由运维团队配置好`config-override.js`，以覆盖`config-override.js`的默认设置。测试环境下，本地和CI服务器统一使用`config-test.js`，测试数据库可以反复清空，不会影响开发。

配置文件表面上写起来很容易，但是，既要保证开发效率，又要避免服务器配置文件泄漏，还要能方便地执行测试，就需要一开始搭建出好的结构，才能提升工程能力。

使用Model

要使用Model，就需要引入对应的Model文件，例如：`user.js`。一旦Model多了起来，如何引用也是一件麻烦事。

自动化永远比手工做效率高，而且更可靠。我们写一个`model.js`，自动扫描并导入所有Model：

```
const fs = require('fs');
const db = require('./db');

let files = fs.readdirSync(__dirname + '/models');

let js_files = files.filter((f)=>{
    return f.endsWith('.js');
}, files);

module.exports = {};

for (let f of js_files) {
    console.log(`import model from file ${f}...`);
    let name = f.substring(0, f.length - 3);
    module.exports[name] = require(__dirname + '/models/'
+ f);
}

module.exports.sync = () => {
    db.sync();
};
```

这样，需要用的时候，写起来就像这样：

```
const model = require('./model');

let
    Pet = model.Pet,
    User = model.User;

var pet = await Pet.create({ ... });
```

工程结构

最终，我们创建的工程`model-sequelize`结构如下：

```
model-sequelize/
|
+- .vscode/
|  |
|  +- launch.json <-- VSCode 配置文件
|
+- models/ <-- 存放所有Model
|  |
|  +- Pet.js <-- Pet
|  |
|  +- User.js <-- User
|
+- config.js <-- 配置文件入口
|
+- config-default.js <-- 默认配置文件
|
+- config-test.js <-- 测试配置文件
|
+- db.js <-- 如何定义Model
|
+- model.js <-- 如何导入Model
|
+- init-db.js <-- 初始化数据库
|
+- app.js <-- 业务代码
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
```

注意到我们其实不需要创建表的SQL，因为Sequelize提供了一个`sync()`方法，可以自动创建数据库。这个功能在开发和生产环境中没有什么用，但是在测试环境中非常有用。测试时，我们可以用`sync()`方法自动创建出表结构，而不是自己维护SQL脚本。这样，可以随时修改Model的定义，并立刻运行测试。开发环境下，首次使用`sync()`也可以自动创建出表结构，避免了手动运行SQL的问题。

`init-db.js`的代码非常简单：

```
const model = require('./model.js');
model.sync();

console.log('init db ok.');
process.exit(0);
```

它最大的好处是避免了手动维护一个SQL脚本。

mocha

如果你听说过“测试驱动开发”（TDD：Test-Driven Development），单元测试就不陌生。

单元测试是用来对一个模块、一个函数或者一个类来进行正确性检验的测试工作。

比如对函数abs()，我们可以编写出以下几个测试用例：

输入正数，比如1、1.2、0.99，期待返回值与输入相同；

输入负数，比如-1、-1.2、-0.99，期待返回值与输入相反；

输入0，期待返回0；

输入非数值类型，比如null、[]、{}，期待抛出Error。

把上面的测试用例放到一个测试模块里，就是一个完整的单元测试。

如果单元测试通过，说明我们测试的这个函数能够正常工作。如果单元测试不通过，要么函数有bug，要么测试条件输入不正确，总之，需要修复使单元测试能够通过。

单元测试通过后有什么意义呢？如果我们对abs()函数代码做了修改，只需要再跑一遍单元测试，如果通过，说明我们的修改不会对abs()函数原有的行为造成影响，如果测试不通过，说明我们的修改与原有行为不一致，要么修改代码，要么修改测试。

这种以测试为驱动的开发模式最大的好处就是确保一个程序模块的行为符合我们设计的测试用例。在将来修改的时候，可以极大程度地保证该模块行为仍然是正确的。

mocha

mocha是JavaScript的一种单元测试框架，既可以在浏览器环境下运行，也可以在Node.js环境下运行。

使用mocha，我们就只需要专注于编写单元测试本身，然后，让mocha去自动运行所有的测试，并给出测试结果。

mocha的特点主要有：

1. 既可以测试简单的JavaScript函数，又可以测试异步代码，因为异步是JavaScript的特性之一；
2. 可以自动运行所有测试，也可以只运行特定的测试；
3. 可以支持before、after、beforeEach和afterEach来编写初始化代码。

我们会详细讲解如何使用mocha编写自动化测试，以及如何测试异步代码。

编写测试

假设我们编写了一个`hello.js`，并且输出一个简单的求和函数：

```
// hello.js

module.exports = function (...rest) {
    var sum = 0;
    for (let n of rest) {
        sum += n;
    }
    return sum;
};
```

这个函数非常简单，就是对输入的任意参数求和并返回结果。

如果我们想对这个函数进行测试，可以写一个`test.js`，然后使用Node.js提供的`assert`模块进行断言：

```
// test.js

const assert = require('assert');
const sum = require('./hello');

assert.strictEqual(sum(), 0);
assert.strictEqual(sum(1), 1);
assert.strictEqual(sum(1, 2), 3);
assert.strictEqual(sum(1, 2, 3), 6);
```

`assert`模块非常简单，它断言一个表达式为true。如果断言失败，就抛出Error。可以在Node.js文档中查看`assert`模块的所有API。

单独写一个`test.js`的缺点是没法自动运行测试，而且，如果第一个assert报错，后面的测试也执行不了了。

如果有很多测试需要运行，就必须把这些测试全部组织起来，然后统一执行，并且得到执行结果。这就是我们为什么要用mocha来编写并运行测试。

mocha test

我们创建`hello-test`工程来编写`hello.js`以及相关测试。工程结构如下：

```
hello-test/
|
+- .vscode/
|  |
|  +- launch.json <-- VSCode 配置文件
|
+- hello.js <-- 待测试js文件
|
+- test/ <-- 存放所有test
|  |
|  +- hello-test.js <-- 测试文件
```

```
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
```

我们首先在`package.json`中添加mocha的依赖包。和其他依赖包不同，这次我们并没有把依赖包添加到`"dependencies"`中，而是`"devDependencies"`：

```
{
  ...

  "dependencies": {},
  "devDependencies": {
    "mocha": "3.0.2"
  }
}
```

如果一个模块在运行的时候并不需要，仅仅在开发时才需要，就可以放到`devDependencies`中。这样，正式打包发布时，`devDependencies`的包不会被包含进来。

然后使用`npm install`安装。

注意，很多文章会让你用命令`npm install -g mocha`把mocha安装到全局module中。这是不需要的。尽量不要安装全局模块，因为全局模块会影响到所有Node.js的工程。

紧接着，我们在`test`目录下创建`hello-test.js`来编写测试。

mocha默认会执行`test`目录下的所有测试，不要去改变默认目录。

`hello-test.js`内容如下：

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

const sum = require('../hello');

describe('#hello.js', () => {

    describe('#sum()', () => {
        it('sum() should return 0', () => {
            assert.strictEqual(sum(), 0);
        });

        it('sum(1) should return 1', () => {
            assert.strictEqual(sum(1), 1);
        });

        it('sum(1, 2) should return 3', () => {
            assert.strictEqual(sum(1, 2), 3);
        });
```

```
        it('sum(1, 2, 3) should return 6', () => {
            assert.strictEqual(sum(1, 2, 3), 6);
        });
    });
});
```

这里我们使用mocha默认的BDD-style的测试。`describe`可以任意嵌套，以便把相关测试看成一组测试。

每个`it("name", function() {...})`就代表一个测试。例如，为了测试`sum(1, 2)`，我们这样写：

```
it('sum(1, 2) should return 3', () => {
    assert.strictEqual(sum(1, 2), 3);
});
```

编写测试的原则是，一次只测一种情况，且测试代码要非常简单。我们编写多个测试来分别测试不同的输入，并使用`assert`判断输出是否是我们所期望的。

运行测试

下一步，我们就可以用mocha运行测试了。

如何运行？有三种方法。

方法一，可以打开命令提示符，切换到`hello-test`目录，然后执行命令：

```
C:\...\hello-test> node_modules\mocha\bin\mocha
```

mocha就会自动执行所有测试，然后输出如下：

```
  #hello.js
    #sum()
      ✓ sum() should return 0
      ✓ sum(1) should return 1
      ✓ sum(1, 2) should return 3
      ✓ sum(1, 2, 3) should return 6
  4 passing (7ms)
```

这说明我们编写的4个测试全部通过。如果没有通过，要么修改测试代码，要么修改`hello.js`，直到测试全部通过为止。

方法二，我们在`package.json`中添加npm命令：

```
{
  ...

  "scripts": {
    "test": "mocha"
  },

  ...
}
```

然后在`hello-test`目录下执行命令：

```
C:\...\hello-test> npm test
```

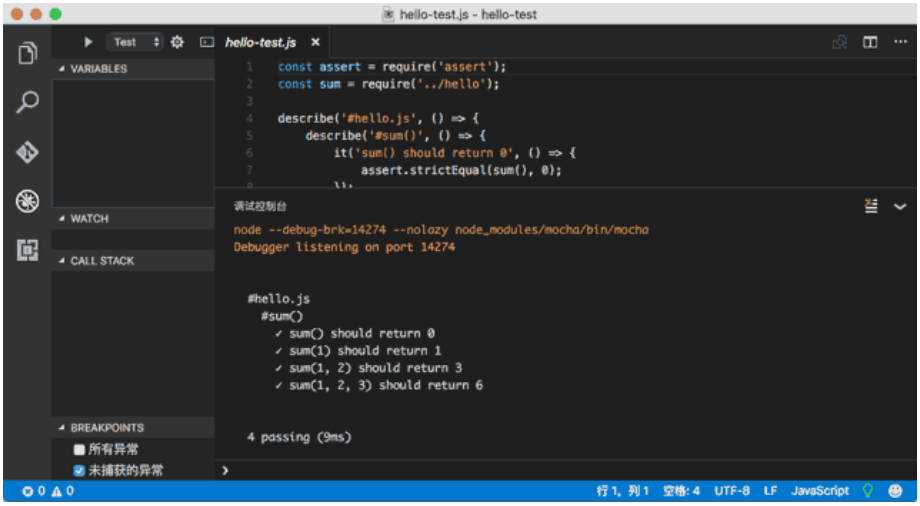
可以得到和上面一样的输出。这种方式通过npm执行命令，输入的命令比较简单。

方法三，我们在VS Code中创建配置文件`.vscode/launch.json`，然后编写两个配置选项：

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Run",
            "type": "node",
            "request": "launch",
            "program": "${workspaceRoot}/hello.js",
            "stopOnEntry": false,
            "args": [],
            "cwd": "${workspaceRoot}",
            "preLaunchTask": null,
            "runtimeExecutable": null,
            "runtimeArgs": [
                "--nolazy"
            ],
            "env": {
                "NODE_ENV": "development"
            },
            "externalConsole": false,
            "sourceMaps": false,
            "outDir": null
        },
        {
            "name": "Test",
            "type": "node",
            "request": "launch",
            "program":
"${workspaceRoot}/node_modules/mocha/bin/mocha",
            "stopOnEntry": false,
            "args": [],
```

```
            "cwd": "${workspaceRoot}",
            "preLaunchTask": null,
            "runtimeExecutable": null,
            "runtimeArgs": [
                "--nolazy"
            ],
            "env": {
                "NODE_ENV": "test"
            },
            "externalConsole": false,
            "sourceMaps": false,
            "outDir": null
        }
    ]
}
```

注意第一个配置选项`Run`是正常执行一个.js文件，第二个配置选项`Test`我们填入`"program": "${workspaceRoot}/node_modules/mocha/bin/mocha"`，并设置`env`为`"NODE_ENV": "test"`，这样，就可以在VS Code中打开Debug面板，选择`Test`，运行，即可在Console面板中看到测试结果：



before和after

在测试前初始化资源，测试后释放资源是非常常见的。mocha提供了before、after、beforeEach和afterEach来实现这些功能。

我们把`hello-test.js`改为：

```
const assert = require('assert');
const sum = require('../hello');

describe('#hello.js', () => {
    describe('#sum()', () => {
        before(function () {
            console.log('before:');
        });

        after(function () {
```

```
            console.log('after.');
        });

        beforeEach(function () {
            console.log('  beforeEach:');
        });

        afterEach(function () {
            console.log('  afterEach.');
        });

        it('sum() should return 0', () => {
            assert.strictEqual(sum(), 0);
        });

        it('sum(1) should return 1', () => {
            assert.strictEqual(sum(1), 1);
        });

        it('sum(1, 2) should return 3', () => {
            assert.strictEqual(sum(1, 2), 3);
        });

        it('sum(1, 2, 3) should return 6', () => {
            assert.strictEqual(sum(1, 2, 3), 6);
        });
    });
});
```

再次运行，可以看到每个test执行前后会分别执行`beforeEach()`和`afterEach()`，以及一组test执行前后会分别执行`before()`和`after()`：

```
  #hello.js
    #sum()
before:
  beforeEach:
      ✓ sum() should return 0
  afterEach.
  beforeEach:
      ✓ sum(1) should return 1
  afterEach.
  beforeEach:
      ✓ sum(1, 2) should return 3
  afterEach.
  beforeEach:
      ✓ sum(1, 2, 3) should return 6
  afterEach.
after.
  4 passing (8ms)
```

异步测试

用mocha测试一个函数是非常简单的，但是，在JavaScript的世界中，更多的时候，我们编写的是异步代码，所以，我们需要用mocha测试异步函数。

我们把上一节的`hello-test`工程复制一份，重命名为`async-test`，然后，把`hello.js`改造为异步函数：

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

// a simple async function:
module.exports = async () => {
    let expression = await fs.readFile('./data.txt', 'utf-8');
    let fn = new Function('return ' + expression);
    let r = fn();
    console.log(`Calculate: ${expression} = ${r}`);
    return r;
};
```

这个async函数通过读取`data.txt`的内容获取表达式，这样它就变成了异步。我们编写一个`data.txt`文件，内容如下：

```
1 + (2 + 4) * (9 - 2) / 3
```

别忘了在`package.json`中添加依赖包：

```
"dependencies": {
    "mz": "2.4.0"
},
```

紧接着，我们在`test`目录中添加一个`await-test.js`，测试`hello.js`的async函数。

我们先看看mocha如何实现异步测试。

如果要测试同步函数，我们传入无参数函数即可：

```
it('test sync function', function () {
    // TODO:
    assert(true);
});
```

如果要测试异步函数，我们要传入的函数需要带一个参数，通常命名为`done`：

```
it('test async function', function (done) {
    fs.readFile('filepath', function (err, data) {
        if (err) {
            done(err);
        } else {
            done();
        }
    });
});
```

测试异步函数需要在函数内部手动调用`done()`表示测试成功，`done(err)`表示测试出错。

对于用ES7的async编写的函数，我们可以这么写：

```
it('#async with done', (done) => {
    (async function () {
        try {
            let r = await hello();
            assert.strictEqual(r, 15);
            done();
        } catch (err) {
            done(err);
        }
    })();
});
```

但是用try...catch太麻烦。还有一种更简单的写法，就是直接把async函数当成同步函数来测试：

```
it('#async function', async () => {
    let r = await hello();
    assert.strictEqual(r, 15);
});
```

这么写异步测试，太简单了有木有！

我们把上一个`hello-test`工程复制为`async-test`，结构如下：

```
async-test/
|
+- .vscode/
|  |
|  +- launch.json <-- VSCode 配置文件
|
+- hello.js <-- 待测试js文件
|
+- data.txt <-- 数据文件
|
+- test/ <-- 存放所有test
```

```
|  |
|  +- await-test.js <-- 异步测试
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
```

现在，在命令行窗口运行命令`node_modules\mocha\bin\mocha`，测试就可以正常执行：

```
  #async hello
    #asyncCalculate()
Calculate: 1 + (2 + 4) * (9 - 2) / 3 = 15
      ✓ #async function
  1 passing (11ms)
```

第二种方法是在`package.json`中把`script`改为：

```
"scripts": {
    "test": "mocha"
}
```

这样就可以在命令行窗口通过`npm test`执行测试。

第三种方法是在VS Code配置文件中把`program`改为：

```
"program": "${workspaceRoot}/node_modules/mocha/bin/mocha"
```

这样就可以在VS Code中直接运行测试。

编写异步代码时，我们要坚持使用`async`和`await`关键字，这样，编写测试也同样简单。

Http测试

用mocha测试一个async函数是非常方便的。现在，当我们有了一个koa的Web应用程序时，我们怎么用mocha来自动化测试Web应用程序呢？

一个简单的想法就是在测试前启动koa的app，然后运行async测试，在测试代码中发送http请求，收到响应后检查结果，这样，一个基于http接口的测试就可以自动运行。

我们先创建一个最简单的koa应用，结构如下：

```
koa-test/
|
+- .vscode/
|  |
```

```
|  +- launch.json <-- VSCode 配置文件
|
+- app.js <-- koa app文件
|
+- start.js <-- app启动入口
|
+- test/ <-- 存放所有test
|  |
|  +- app-test.js <-- 异步测试
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
```

这个koa应用和前面的koa应用稍有不同的是，`app.js`只负责创建`app`实例，并不监听端口：

```
// app.js

const Koa = require('koa');

const app = new Koa();

app.use(async (ctx, next) => {
    const start = new Date().getTime();
    await next();
    const ms = new Date().getTime() - start;
    console.log(`${ctx.request.method} ${ctx.request.url}:
${ms}ms`);
    ctx.response.set('X-Response-Time', `${ms}ms`);
});

app.use(async (ctx, next) => {
    var name = ctx.request.query.name || 'world';
    ctx.response.type = 'text/html';
    ctx.response.body = `<h1>Hello, ${name}!</h1>`;
});

module.exports = app;
```

而`start.js`负责真正启动应用：

```
// start.js

const app = require('./app');

app.listen(3000);
console.log('app started at port 3000...');
```

这样做的目的是便于后面的测试。

紧接着，我们在`test`目录下创建`app-test.js`，来测试这个koa应用。

在测试前，我们在`package.json`中添加`devDependencies`，除了mocha外，我们还需要一个简单而强大的测试模块`supertest`：

```
{
    ...
    "devDependencies": {
        "mocha": "3.0.2",
        "supertest": "3.0.0"
    }
}
```

运行`npm install`后，我们开始编写测试：

```
// app-test.js

const
    request = require('supertest'),
    app = require('../app');

describe('#test koa app', () => {

    let server = app.listen(9900);

    describe('#test server', () => {

        it('#test GET /', async () => {
            let res = await request(server)
                .get('/')
                .expect('Content-Type', /text\/html/)
                .expect(200, '<h1>Hello, world!</h1>');
        });

        it('#test GET /path?name=Bob', async () => {
            let res = await request(server)
                .get('/path?name=Bob')
                .expect('Content-Type', /text\/html/)
                .expect(200, '<h1>Hello, Bob!</h1>');
        });
    });
});
```

在测试中，我们首先导入`supertest`模块，然后导入`app`模块，注意我们已经在`app.js`中移除了`app.listen(3000);`语句，所以，这里我们用：

```
let server = app.listen(9900);
```

让`app`实例监听在`9900`端口上，并且获得返回的`server`实例。

在测试代码中，我们使用：

```
let res = await request(server).get('/');
```

就可以构造一个GET请求，发送给koa的应用，然后获得响应。

可以手动检查响应对象，例如，`res.body`，还可以利用`supertest`提供的`expect()`更方便地断言响应的HTTP代码、返回内容和HTTP头。断言HTTP头时可用使用正则表达式。例如，下面的断言：

```
.expect('Content-Type', /text\/html/)
```

可用成功匹配到`Content-Type`为`text/html`、`text/html; charset=utf-8`等值。

当所有测试运行结束后，`app`实例会自动关闭，无需清理。

利用mocha的异步测试，配合supertest，我们可以用简单的代码编写端到端的HTTP自动化测试。

WebSocket

WebSocket是HTML5新增的协议，它的目的是在浏览器和服务器之间建立一个不受限的双向通信的通道，比如说，服务器可以在任意时刻发送消息给浏览器。

为什么传统的HTTP协议不能做到WebSocket实现的功能？这是因为HTTP协议是一个请求－响应协议，请求必须先由浏览器发给服务器，服务器才能响应这个请求，再把数据发送给浏览器。换句话说，浏览器不主动请求，服务器是没法主动发数据给浏览器的。

这样一来，要在浏览器中搞一个实时聊天，在线炒股（不鼓励），或者在线多人游戏的话就没法实现了，只能借助Flash这些插件。

也有人说，HTTP协议其实也能实现啊，比如用轮询或者Comet。轮询是指浏览器通过JavaScript启动一个定时器，然后以固定的间隔给服务器发请求，询问服务器有没有新消息。这个机制的缺点一是实时性不够，二是频繁的请求会给服务器带来极大的压力。

Comet本质上也是轮询，但是在没有消息的情况下，服务器先拖一段时间，等到有消息了再回复。这个机制暂时地解决了实时性问题，但是它带来了新的问题：以多线程模式运行的服务器会让大部分线程大部分时间都处于挂起状态，极大地浪费服务器资源。另外，一个HTTP连接在长时间没有数据传输的情况下，链路上的任何一个网关都可能关闭这个连接，而网关是我们不可控的，这就要求Comet连接必须定期发一些ping数据表示连接“正常工作”。

以上两种机制都治标不治本，所以，HTML5推出了WebSocket标准，让浏览器和服务器之间可以建立无限制的全双工通信，任何一方都可以主动发消息给对方。

WebSocket协议

WebSocket并不是全新的协议，而是利用了HTTP协议来建立连接。我们来看看WebSocket连接是如何创建的。

首先，WebSocket连接必须由浏览器发起，因为请求协议是一个标准的HTTP请求，格式如下：

```
GET ws://localhost:3000/ws/chat HTTP/1.1
Host: localhost
Upgrade: websocket
Connection: Upgrade
Origin: http://localhost:3000
Sec-WebSocket-Key: client-random-string
Sec-WebSocket-Version: 13
```

该请求和普通的HTTP请求有几点不同：

1. GET请求的地址不是类似`/path/`，而是以`ws://`开头的地址；
2. 请求头`Upgrade: websocket`和`Connection: Upgrade`表示这个连接将要被转换为WebSocket连接；
3. `Sec-WebSocket-Key`是用于标识这个连接，并非用于加密数据；
4. `Sec-WebSocket-Version`指定了WebSocket的协议版本。

随后，服务器如果接受该请求，就会返回如下响应：

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: server-random-string
```

该响应代码`101`表示本次连接的HTTP协议即将被更改，更改后的协议就是`Upgrade: websocket`指定的WebSocket协议。

版本号和子协议规定了双方能理解的数据格式，以及是否支持压缩等等。如果仅使用WebSocket的API，就不需要关心这些。

现在，一个WebSocket连接就建立成功，浏览器和服务器就可以随时主动发送消息给对方。消息有两种，一种是文本，一种是二进制数据。通常，我们可以发送JSON格式的文本，这样，在浏览器处理起来就十分容易。

为什么WebSocket连接可以实现全双工通信而HTTP连接不行呢？实际上HTTP协议是建立在TCP协议之上的，TCP协议本身就实现了全双工通信，但是HTTP协议的请求－应答机制限制了全双工通信。WebSocket连接建立以后，其实只是简单规定了一下：接下来，咱们通信就不使用HTTP协议了，直接互相发数据吧。

安全的WebSocket连接机制和HTTPS类似。首先，浏览器用`wss://xxx`创建WebSocket连接时，会先通过HTTPS创建安全的连接，然后，该HTTPS连接升级为WebSocket连接，底层通信走的仍然是安全的SSL/TLS协议。

浏览器

很显然，要支持WebSocket通信，浏览器得支持这个协议，这样才能发出`ws://xxx`的请求。目前，支持WebSocket的主流浏览器如下：

- Chrome
- Firefox
- IE >= 10
- Sarafi >= 6
- Android >= 4.4
- iOS >= 8

服务器

由于WebSocket是一个协议，服务器具体怎么实现，取决于所用编程语言和框架本身。Node.js本身支持的协议包括TCP协议和HTTP协议，要支持WebSocket协议，需要对Node.js提供的HTTPServer做额外的开发。已经有若干基于Node.js的稳定可靠的WebSocket实现，我们直接用npm安装使用即可。

使用ws

要使用WebSocket，关键在于服务器端支持，这样，我们才有可能用支持WebSocket的浏览器使用WebSocket。

ws模块

在Node.js中，使用最广泛的WebSocket模块是`ws`，我们创建一个`hello-ws`的VS Code工程，然后在`package.json`中添加`ws`的依赖：

```
"dependencies": {
    "ws": "1.1.1"
}
```

整个工程结构如下：

```
hello-ws/
|
+- .vscode/
|  |
|  +- launch.json <-- VSCode 配置文件
|
+- app.js <-- 启动js文件
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
```

运行`npm install`后，我们就可以在`app.js`中编写WebSocket的服务器端代码。

创建一个WebSocket的服务器实例非常容易：

```
// 导入WebSocket模块:
const WebSocket = require('ws');

// 引用Server类:
const WebSocketServer = WebSocket.Server;

// 实例化:
const wss = new WebSocketServer({
    port: 3000
});
```

这样，我们就在3000端口上打开了一个WebSocket Server，该实例由变量`wss`引用。

接下来，如果有WebSocket请求接入，`wss`对象可以响应`connection`事件来处理这个WebSocket：

```
wss.on('connection', function (ws) {
    console.log(`[SERVER] connection()`);
    ws.on('message', function (message) {
        console.log(`[SERVER] Received: ${message}`);
        ws.send(`ECHO: ${message}`, (err) => {
            if (err) {
                console.log(`[SERVER] error: ${err}`);
            }
        });
    })
});
```

在`connection`事件中，回调函数会传入一个`WebSocket`的实例，表示这个WebSocket连接。对于每个WebSocket连接，我们都要对它绑定某些事件方法来处理不同的事件。这里，我们通过响应`message`事件，在收到消息后再返回一个`ECHO: xxx`的消息给客户端。

创建WebSocket连接

现在，这个简单的服务器端WebSocket程序就编写好了。如何真正创建WebSocket并且给服务器发消息呢？方法是在浏览器中写JavaScript代码。

先在VS Code中执行`app.js`，或者在命令行用`npm start`执行。然后，在当前页面下，直接打开可以执行JavaScript代码的浏览器Console，依次输入代码：

```
// 打开一个WebSocket:
var ws = new WebSocket('ws://localhost:3000/test');
// 响应onmessage事件:
ws.onmessage = function(msg) { console.log(msg); };
// 给服务器发送一个字符串:
ws.send('Hello!');
```

一切正常的话，可以看到Console的输出如下：

```
MessageEvent {isTrusted: true, data: "ECHO: Hello!",
origin: "ws://localhost:3000", lastEventId: "", source:
null…}
```

这样，我们就在浏览器中成功地收到了服务器发送的消息！

如果嫌在浏览器中输入JavaScript代码比较麻烦，我们还可以直接用`ws`模块提供的`WebSocket`来充当客户端。换句话说，`ws`模块既包含了服务器端，又包含了客户端。

`ws`的`WebSocket`就表示客户端，它其实就是WebSocketServer响应`connection`事件时回调函数传入的变量`ws`的类型。

客户端的写法如下：

```
let ws = new WebSocket('ws://localhost:3000/test');

// 打开WebSocket连接后立刻发送一条消息:
ws.on('open', function () {
    console.log(`[CLIENT] open()`);
    ws.send('Hello!');
});

// 响应收到的消息:
ws.on('message', function (message) {
    console.log(`[CLIENT] Received: ${message}`);
}
```

在Node环境下，`ws`模块的客户端可以用于测试服务器端代码，否则，每次都必须在浏览器执行JavaScript代码。

同源策略

从上面的测试可以看出，WebSocket协议本身不要求同源策略（Same-origin Policy），也就是某个地址为`http://a.com`的网页可以通过WebSocket连接到`ws://b.com`。但是，浏览器会发送`Origin`的HTTP头给服务器，服务器可以根据`Origin`拒绝这个WebSocket请求。所以，是否要求同源要看服务器端如何检查。

路由

还需要注意到服务器在响应`connection`事件时并未检查请求的路径，因此，在客户端打开`ws://localhost:3000/any/path`可以写任意的路径。

实际应用中还需要根据不同的路径实现不同的功能。

编写聊天室

上一节我们用`ws`模块创建了一个WebSocket应用。但是它只能简单地响应`ECHO: xxx`消息，还属于Hello, world级别的应用。

要创建真正的WebSocket应用，首先，得有一个基于MVC的Web应用，也就是我们在前面用koa2和Nunjucks创建的Web，在此基础上，把WebSocket添加进来，才算完整。

因此，本节的目标是基于WebSocket创建一个在线聊天室。

首先，我们把前面编写的MVC工程复制一份，先创建一个完整的MVC的Web应用，结构如下：

```
ws-with-koa/
|
+- .vscode/
|  |
|  +- launch.json <-- VSCode 配置文件
|
+- controllers/ <-- Controller
|
+- views/ <-- html模板文件
|
+- static/ <-- 静态资源文件
|
+- app.js <-- 使用koa的js
|
+- controller.js <-- 扫描注册Controller
|
+- static-files.js <-- 处理静态文件
|
+- templating.js <-- 模版引擎入口
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
```

然后，把我们需要的依赖包添加到`package.json`：

```
"dependencies": {
    "ws": "1.1.1",
    "koa": "2.0.0",
    "koa-bodyparser": "3.2.0",
    "koa-router": "7.0.0",
    "nunjucks": "2.4.2",
    "mime": "1.3.4",
    "mz": "2.4.0"
}
```

使用`npm install`安装后，我们首先得到了一个标准的基于MVC的koa2应用。该应用的核心是一个代表koa应用的`app`变量：

```
const app = new Koa();

// TODO: app.use(...);

app.listen(3000);
```

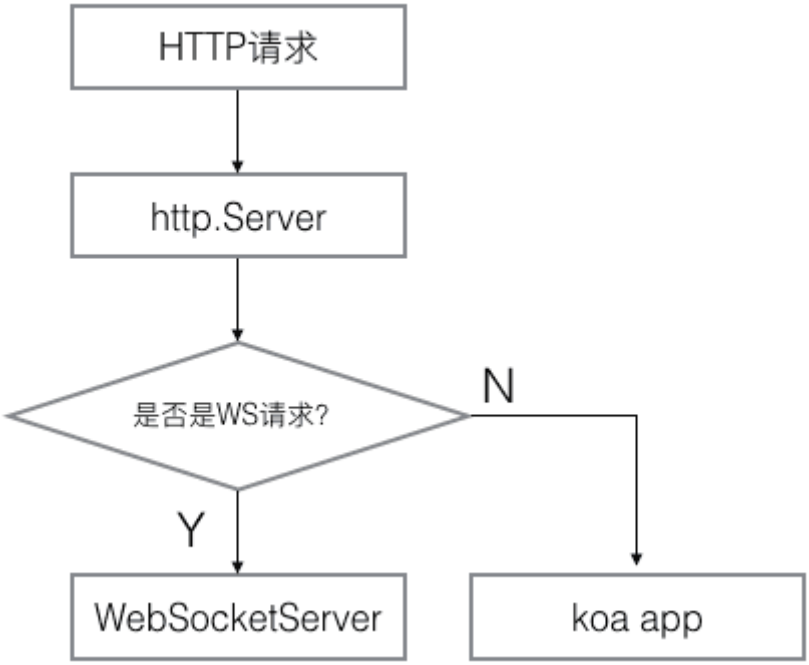
现在第一个问题来了：koa通过3000端口响应HTTP，我们要新加的WebSocketServer还能否使用3000端口？

答案是肯定的。虽然WebSocketServer可以使用别的端口，但是，统一端口有个最大的好处：

实际应用中，HTTP和WebSocket都使用标准的80和443端口，不需要暴露新的端口，也不需要修改防火墙规则。

在3000端口被koa占用后，WebSocketServer如何使用该端口？

实际上，3000端口并非由koa监听，而是koa调用Node标准的http模块创建的http.Server监听的。koa只是把响应函数注册到该http.Server中了。类似的，WebSocketServer也可以把自己的响应函数注册到http.Server中，这样，同一个端口，根据协议，可以分别由koa和ws处理：



把WebSocketServer绑定到同一个端口的关键代码是先获取koa创建的`http.Server`的引用，再根据`http.Server`创建WebSocketServer：

```
// koa app的listen()方法返回http.Server:
let server = app.listen(3000);

// 创建WebSocketServer:
let wss = new WebSocketServer({
    server: server
});
```

要始终注意，浏览器创建WebSocket时发送的仍然是标准的HTTP请求。无论是WebSocket请求，还是普通HTTP请求，都会被http.Server处理。具体的处理方式则是由koa和WebSocketServer注入的回调函数实现的。WebSocketServer会首先判断请求是不是WS请求，如果是，它将处理该请求，如果不是，该请求仍由koa处理。

所以，WS请求会直接由WebSocketServer处理，它根本不会经过koa，koa的任何middleware都没有机会处理该请求。

现在第二个问题来了：在koa应用中，可以很容易地认证用户，例如，通过session或者cookie，但是，在响应WebSocket请求时，如何识别用户身份？

一个简单可行的方案是把用户登录后的身份写入Cookie，在koa中，可以使用middleware解析Cookie，把用户绑定到`ctx.state.user`上。

WS请求也是标准的HTTP请求，所以，服务器也会把Cookie发送过来，这样，我们在用WebSocketServer处理WS请求时，就可以根据Cookie识别用户身份。

先把识别用户身份的逻辑提取为一个单独的函数：

```
function parseUser(obj) {
    if (!obj) {
        return;
    }
    console.log('try parse: ' + obj);
    let s = '';
    if (typeof obj === 'string') {
        s = obj;
    } else if (obj.headers) {
        let cookies = new Cookies(obj, null);
        s = cookies.get('name');
    }
    if (s) {
        try {
            let user = JSON.parse(Buffer.from(s,
'base64').toString());
            console.log(`User: ${user.name}, ID:
${user.id}`);
            return user;
        } catch (e) {
            // ignore
        }
    }
}
```

注意：出于演示目的，该Cookie并没有作Hash处理，实际上它就是一个JSON字符串。

在koa的middleware中，我们很容易识别用户：

```
app.use(async (ctx, next) => {
    ctx.state.user = parseUser(ctx.cookies.get('name') ||
'');
    await next();
});
```

在WebSocketServer中，就需要响应`connection`事件，然后识别用户：

```
wss.on('connection', function (ws) {
    // ws.upgradeReq是一个request对象:
    let user = parseUser(ws.upgradeReq);
    if (!user) {
        // Cookie不存在或无效，直接关闭WebSocket:
        ws.close(4001, 'Invalid user');
    }
    // 识别成功，把user绑定到该WebSocket对象:
    ws.user = user;
    // 绑定WebSocketServer对象:
    ws.wss = wss;
});
```

紧接着，我们要对每个创建成功的WebSocket绑定`message`、`close`、`error`等事件处理函数。对于聊天应用来说，每收到一条消息，就需要把该消息广播到所有WebSocket连接上。

先为`wss`对象添加一个`broadcase()`方法：

```
wss.broadcast = function (data) {
    wss.clients.forEach(function (client) {
        client.send(data);
    });
};
```

在某个WebSocket收到消息后，就可以调用`wss.broadcast()`进行广播了：

```
ws.on('message', function (message) {
    console.log(message);
    if (message && message.trim()) {
        let msg = createMessage('chat', this.user,
message.trim());
        this.wss.broadcast(msg);
    }
});
```

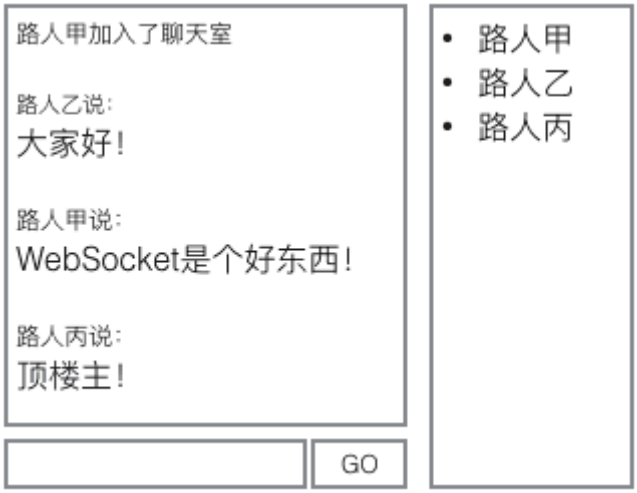
消息有很多类型，不一定是聊天的消息，还可以有获取用户列表、用户加入、用户退出等多种消息。所以我们用`createMessage()`创建一个JSON格式的字符串，发送给浏览器，浏览器端的JavaScript就可以直接使用：

```
// 消息ID:
var messageIndex = 0;

function createMessage(type, user, data) {
    messageIndex ++;
    return JSON.stringify({
        id: messageIndex,
        type: type,
        user: user,
        data: data
    });
}
```

编写页面

相比服务器端的代码，页面的JavaScript代码会更复杂。

聊天室页面可以划分为左侧会话列表和右侧用户列表两部分：



这里的DOM需要动态更新，因此，状态管理是页面逻辑的核心。

为了简化状态管理，我们用Vue控制左右两个列表：

```
var vmMessageList = new Vue({
    el: '#message-list',
    data: {
        messages: []
    }
});

var vmUserList = new Vue({
    el: '#user-list',
    data: {
        users: []
    }
});
```

会话列表和用户列表初始化为空数组。

紧接着，创建WebSocket连接，响应服务器消息，并且更新会话列表和用户列表：

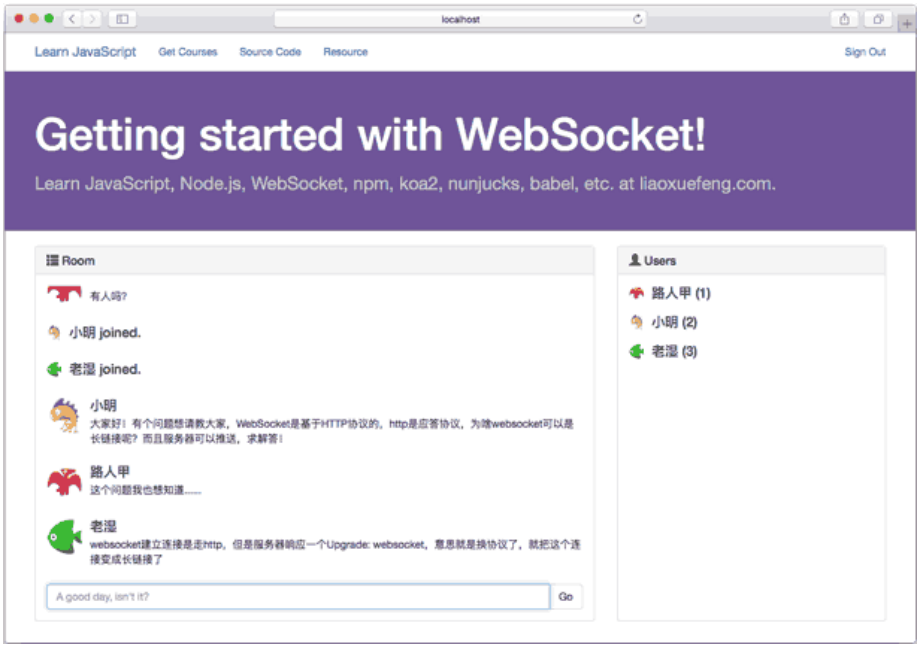
```
var ws = new WebSocket('ws://localhost:3000/ws/chat');

ws.onmessage = function(event) {
    var data = event.data;
    console.log(data);
    var msg = JSON.parse(data);
    if (msg.type === 'list') {
        vmUserList.users = msg.data;
    } else if (msg.type === 'join') {
        addToUserList(vmUserList.users, msg.user);
        addMessage(vmMessageList.messages, msg);
    } else if (msg.type === 'left') {
        removeFromUserList(vmUserList.users, msg.user);
        addMessage(vmMessageList.messages, msg);
    } else if (msg.type === 'chat') {
        addMessage(vmMessageList.messages, msg);
    }
};
```

这样，JavaScript负责更新状态，Vue负责根据状态刷新DOM。以用户列表为例，HTML代码如下：

```
<div id="user-list">
    <div class="media" v-for="user in users">
        <div class="media-left">
            <img class="media-object"
src="/static/user.png">
        </div>
        <div class="media-body">
            <h4 class="media-heading" v-text="user.name">
</h4>
        </div>
    </div>
</div>
```

测试的时候，如果在本机测试，需要同时用几个不同的浏览器，这样Cookie互不干扰。

最终的聊天室效果如下：



配置反向代理

如果网站配置了反向代理，例如Nginx，则HTTP和WebSocket都必须通过反向代理连接Node服务器。HTTP的反向代理非常简单，但是要正常连接WebSocket，代理服务器必须支持WebSocket协议。

我们以Nginx为例，编写一个简单的反向代理配置文件。

详细的配置可以参考Nginx的官方博客：Using NGINX as a WebSocket Proxy

首先要保证Nginx版本>=1.3，然后，通过`proxy_set_header`指令，设定：

```
proxy_set_header Upgrade $http_upgrade;
proxy_set_header Connection "upgrade";
```

Nginx即可理解该连接将使用WebSocket协议。

一个示例配置文件内容如下：

```
server {
    listen       80;
    server_name localhost;

    # 处理静态资源文件:
    location ^~ /static/ {
        root /path/to/ws-with-koa;
    }

    # 处理WebSocket连接:
    location ^~ /ws/ {
        proxy_pass         http://127.0.0.1:3000;
        proxy_http_version 1.1;
        proxy_set_header   Upgrade $http_upgrade;
        proxy_set_header   Connection "upgrade";
    }
```

```
    # 其他所有请求:
    location / {
        proxy_pass       http://127.0.0.1:3000;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For
$proxy_add_x_forwarded_for;
    }
}
```

REST

自从Roy Fielding博士在2000年他的博士论文中提出REST（Representational State Transfer）风格的软件架构模式后，REST就基本上迅速取代了复杂而笨重的SOAP，成为Web API的标准了。

什么是Web API呢？

如果我们想要获取某个电商网站的某个商品，输入`http://localhost:3000/products/123`，就可以看到id为123的商品页面，但这个结果是HTML页面，它同时混合包含了Product的数据和Product的展示两个部分。对于用户来说，阅读起来没有问题，但是，如果机器读取，就很难从HTML中解析出Product的数据。

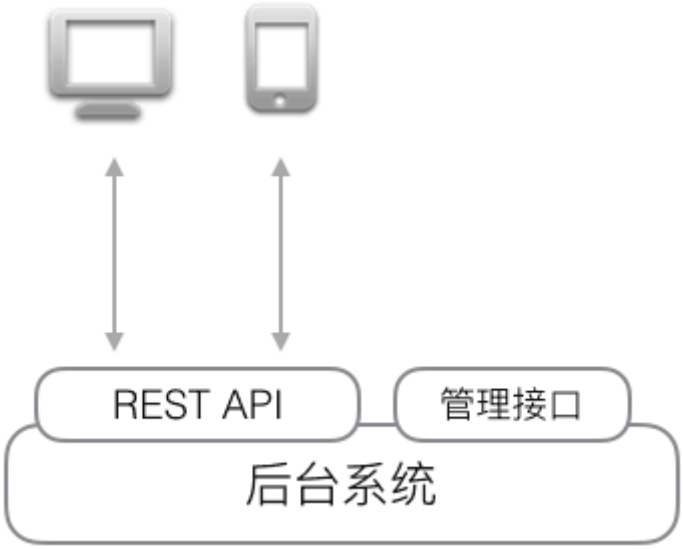
如果一个URL返回的不是HTML，而是机器能直接解析的数据，这个URL就可以看成是一个Web API。比如，读取`http://localhost:3000/api/products/123`，如果能直接返回Product的数据，那么机器就可以直接读取。

REST就是一种设计API的模式。最常用的数据格式是JSON。由于JSON能直接被JavaScript读取，所以，以JSON格式编写的REST风格的API具有简单、易读、易用的特点。

编写API有什么好处呢？由于API就是把Web App的功能全部封装了，所以，通过API操作数据，可以极大地把前端和后端的代码隔离，使得后端代码易于测试，前端代码编写更简单。

此外，如果我们把前端页面看作是一种用于展示的客户端，那么API就是为客户端提供数据、操作数据的接口。这种设计可以获得极高的扩展性。例如，当用户需要在手机上购买商品时，只需要开发针对iOS和Android的两个客户端，通过客户端访问API，就可以完成通过浏览器页面提供的功能，而后端代码基本无需改动。

当一个Web应用以API的形式对外提供功能时，整个应用的结构就扩展为：



把网页视为一种客户端，是REST架构可扩展的一个关键。

编写REST API

REST API规范

编写REST API，实际上就是编写处理HTTP请求的async函数，不过，REST请求和普通的HTTP请求有几个特殊的地方：

1. REST请求仍然是标准的HTTP请求，但是，除了GET请求外，POST、PUT等请求的body是JSON数据格式，请求的`Content-Type`为`application/json`；
2. REST响应返回的结果是JSON数据格式，因此，响应的`Content-Type`也是`application/json`。

REST规范定义了资源的通用访问格式，虽然它不是一个强制要求，但遵守该规范可以让人易于理解。

例如，商品Product就是一种资源。获取所有Product的URL如下：

```
GET /api/products
```

而获取某个指定的Product，例如，id为`123`的Product，其URL如下：

```
GET /api/products/123
```

新建一个Product使用POST请求，JSON数据包含在body中，URL如下：

```
POST /api/products
```

更新一个Product使用PUT请求，例如，更新id为`123`的Product，其URL如下：

```
PUT /api/products/123
```

删除一个Product使用DELETE请求，例如，删除id为`123`的Product，其URL如下：

```
DELETE /api/products/123
```

资源还可以按层次组织。例如，获取某个Product的所有评论，使用：

```
GET /api/products/123/reviews
```

当我们只需要获取部分数据时，可通过参数限制返回的结果集，例如，返回第2页评论，每页10项，按时间排序：

```
GET /api/products/123/reviews?page=2&size=10&sort=time
```

koa处理REST

既然我们已经使用koa作为Web框架处理HTTP请求，因此，我们仍然可以在koa中响应并处理REST请求。

我们先创建一个`rest-hello`的工程，结构如下：

```
rest-hello/
|
+- .vscode/
|  |
|  +- launch.json <-- VSCode 配置文件
|
+- controllers/
|  |
|  +- api.js <-- REST API
|
+- app.js <-- 使用koa的js
|
+- controller.js <-- 扫描注册Controller
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
```

在`package.json`中，我们需要如下依赖包：

```
"dependencies": {
    "koa": "2.0.0",
    "koa-bodyparser": "3.2.0",
    "koa-router": "7.0.0"
}
```

运行`npm install`安装依赖包。

在`app.js`中，我们仍然使用标准的koa组件，并自动扫描加载`controllers`目录下的所有js文件：

```
const app = new Koa();

const controller = require('./controller');

// parse request body:
app.use(bodyParser());

// add controller:
app.use(controller());

app.listen(3000);
console.log('app started at port 3000...');
```

注意到`app.use(bodyParser());`这个语句，它给koa安装了一个解析HTTP请求body的处理函数。如果HTTP请求是JSON数据，我们就可以通过`ctx.request.body`直接访问解析后的JavaScript对象。

下面我们编写`api.js`，添加一个GET请求：

```
// 存储Product列表，相当于模拟数据库:
var products = [{
    name: 'iPhone',
    price: 6999
}, {
    name: 'Kindle',
    price: 999
}];

module.exports = {
    'GET /api/products': async (ctx, next) => {
        // 设置Content-Type:
        ctx.response.type = 'application/json';
        // 设置Response Body:
        ctx.response.body = {
            products: products
        };
    }
}
```

在koa中，我们只需要给`ctx.response.body`赋值一个JavaScript对象，koa会自动把该对象序列化为JSON并输出到客户端。

我们在浏览器中访问`http://localhost:3000/api/products`，可以得到如下输出：

```
{"products":[{"name":"iPhone","price":6999},
{"name":"Kindle","price":999}]}
```

紧接着，我们再添加一个创建Product的API：

```
module.exports = {
    'GET /api/products': async (ctx, next) => {
        ...
    },

    'POST /api/products': async (ctx, next) => {
        var p = {
            name: ctx.request.body.name,
            price: ctx.request.body.price
        };
        products.push(p);
        ctx.response.type = 'application/json';
        ctx.response.body = p;
    }
};
```

这个POST请求无法在浏览器中直接测试。但是我们可以通过`curl`命令在命令提示符窗口测试这个API。我们输入如下命令：

```
curl -H 'Content-Type: application/json' -X POST -d
'{"name":"XBox","price":3999}'
http://localhost:3000/api/products
```

得到的返回内容如下：

```
{"name":"XBox","price":3999}
```

我们再在浏览器中访问`http://localhost:3000/api/products`，可以得到更新后的输出如下：

```
{"products":[{"name":"iPhone","price":6999},
{"name":"Kindle","price":999},
{"name":"XBox","price":3999}]}
```

可见，在koa中处理REST请求是非常简单的。`bodyParser()`这个middleware可以解析请求的JSON数据并绑定到`ctx.request.body`上，输出JSON时我们先指定`ctx.response.type = 'application/json'`，然后把JavaScript对象赋值给`ctx.response.body`就完成了REST请求的处理。

开发REST API

在上一节中，我们演示了如何在koa项目中使用REST。其实，使用REST和使用MVC是类似的，不同的是，提供REST的Controller处理函数最后不调用`render()`去渲染模板，而是把结果直接用JSON序列化返回给客户端。

使用REST虽然非常简单，但是，设计一套合理的REST框架却需要仔细考虑很多问题。

问题一：如何组织URL

在实际工程中，一个Web应用既有REST，还有MVC，可能还需要集成其他第三方系统。如何组织URL？

一个简单的方法是通过固定的前缀区分。例如，`/static/`开头的URL是静态资源文件，类似的，`/api/`开头的URL就是REST API，其他URL是普通的MVC请求。

使用不同的子域名也可以区分，但对于中小项目来说配置麻烦。随着项目的扩大，将来仍然可以把单域名拆成多域名。

问题二：如何统一输出REST

如果每个异步函数都编写下面这样的代码：

```
// 设置Content-Type:
ctx.response.type = 'application/json';
// 设置Response Body:
ctx.response.body = {
    products: products
};
```

很显然，这样的重复代码很容易导致错误，例如，写错了字符串`'application/json'`，或者漏写了`ctx.response.type = 'application/json'`，都会导致浏览器得不到JSON数据。

回忆我们集成Nunjucks模板引擎的方法：通过一个middleware给`ctx`添加一个`render()`方法，Controller就可以直接使用`ctx.render('view', model)`来渲染模板，不必编写重复的代码。

类似的，我们也可以通过一个middleware给`ctx`添加一个`rest()`方法，直接输出JSON数据。

由于我们给所有REST API一个固定的URL前缀`/api/`，所以，这个middleware还需要根据path来判断当前请求是否是一个REST请求，如果是，我们才给`ctx`绑定`rest()`方法。

我们把这个middleware先写出来，命名为`rest.js`：

```
module.exports = {
    restify: (pathPrefix) => {
        // REST API前缀，默认为/api/:
        pathPrefix = pathPrefix || '/api/';
        return async (ctx, next) => {
            // 是否是REST API前缀?
            if (ctx.request.path.startsWith(pathPrefix)) {
                // 绑定rest()方法:
                ctx.rest = (data) => {
```

```
                    ctx.response.type =
'application/json';
                    ctx.response.body = data;
                }
                await next();
            } else {
                await next();
            }
        };
    }
};
```

这样，任何支持REST的异步函数只需要简单地调用：

```
ctx.rest({
    data: 123
});
```

就完成了REST请求的处理。

问题三：如何处理错误

这个问题实际上有两部分。

第一，当REST API请求出错时，我们如何返回错误信息？

第二，当客户端收到REST响应后，如何判断是成功还是错误？

这两部分还必须统一考虑。

REST架构本身对错误处理并没有统一的规定。实际应用时，各种各样的错误处理机制都有。有的设计得比较合理，有的设计得不合理，导致客户端尤其是手机客户端处理API简直就是噩梦。

在涉及到REST API的错误时，我们必须先意识到，客户端会遇到两种类型的REST API错误。

一类是类似403，404，500等错误，这些错误实际上是HTTP请求可能发生的错误。REST请求只是一种请求类型和响应类型均为JSON的HTTP请求，因此，这些错误在REST请求中也会发生。

针对这种类型的错误，客户端除了提示用户“出现了网络错误，稍后重试”以外，并无法获得具体的错误信息。

另一类错误是业务逻辑的错误，例如，输入了不合法的Email地址，试图删除一个不存在的Product，等等。这种类型的错误完全可以通过JSON返回给客户端，这样，客户端可以根据错误信息提示用户“Email不合法”等，以便用户修复后重新请求API。

问题的关键在于客户端必须能区分出这两种类型的错误。

第一类的错误实际上客户端可以识别，并且我们也无法操控HTTP服务器的错误码。

第二类的错误信息是一个JSON字符串，例如：

```
{
    "code": "10000",
    "message": "Bad email address"
}
```

但是HTTP的返回码应该用啥？

有的Web应用使用`200`，这样客户端在识别出第一类错误后，如果遇到`200`响应，则根据响应的JSON判断是否有错误。这种方式对于动态语言（例如，JavaScript，Python等）非常容易：

```
var result = JSON.parse(response.data);
if (result.code) {
    // 有错误:
    alert(result.message);
} else {
    // 没有错误
}
```

但是，对于静态语言（例如，Java）就比较麻烦，很多时候，不得不做两次序列化：

```
APIError err = objectMapper.readValue(jsonString,
APIError.class);
if (err.code == null) {
    // 没有错误，还需要重新转换:
    User user = objectMapper.readValue(jsonString,
User.class);
} else {
    // 有错误:
}
```

有的Web应用对正确的REST响应使用`200`，对错误的REST响应使用`400`，这样，客户端即是静态语言，也可以根据HTTP响应码判断是否出错，出错时直接把结果反序列化为`APIError`对象。

两种方式各有优劣。我们选择第二种，`200`表示成功响应，`400`表示失败响应。

但是，要注意，*绝不能*混合其他HTTP错误码。例如，使用`401`响应“登录失败”，使用`403`响应“权限不够”。这会使客户端无法有效识别HTTP错误码和业务错误，其原因在于HTTP协议定义的错误码十分偏向底层，而REST API属于“高层”协议，不应该复用底层的错误码。

问题四：如何定义错误码

REST架构本身同样没有标准的错误码定义一说，因此，有的Web应用使用数字`1000`、`1001`.....作为错误码，例如Twitter和新浪微博，有的Web应用使用字符串作为错误码，例如YouTube。到底哪一种比较好呢？

我们强烈建议使用字符串作为错误码。原因在于，使用数字作为错误码时，API提供者需要维护一份错误码代码说明表，并且，该文档必须时刻与API发布同步，否则，客户端开发者遇到一个文档上没有写明的错误码，就完全不知道发生了什么错误。

使用字符串作为错误码，最大的好处在于不用查表，根据字面意思也能猜个八九不离十。例如，YouTube API如果返回一个错误`authError`，基本上能猜到是因为认证失败。

我们定义的REST API错误格式如下：

```
{
    "code": "错误代码",
    "message": "错误描述信息"
}
```

其中，错误代码命名规范为`大类:子类`，例如，口令不匹配的登录错误代码为`auth:bad_password`，用户名不存在的登录错误代码为`auth:user_not_found`。这样，客户端既可以简单匹配某个类别的错误，也可以精确匹配某个特定的错误。

问题五：如何返回错误

如果一个REST异步函数想要返回错误，一个直观的想法是调用`ctx.rest()`：

```
user = processLogin(username, password);
if (user != null) {
    ctx.rest(user);
} else {
    ctx.response.status = 400;
    ctx.rest({
        code: 'auth:user_not_found',
        message: 'user not found'
    });
}
```

这种方式不好，因为控制流程会混乱，而且，错误只能在Controller函数中输出。

更好的方式是异步函数直接用`throw`语句抛出错误，让middleware去处理错误：

```
user = processLogin(username, password);
if (user != null) {
    ctx.rest(user);
} else {
    throw new APIError('auth:user_not_found', 'user not
found');
}
```

这种方式可以在异步函数的任何地方抛出错误，包括调用的子函数内部。

我们只需要稍稍改写一个middleware就可以处理错误：

```
module.exports = {
    APIError: function (code, message) {
        this.code = code || 'internal:unknown_error';
        this.message = message || '';
    },
    restify: (pathPrefix) => {
        pathPrefix = pathPrefix || '/api/';
        return async (ctx, next) => {
            if (ctx.request.path.startsWith(pathPrefix)) {
                // 绑定rest()方法:
                ctx.rest = (data) => {
                    ctx.response.type =
'application/json';
                    ctx.response.body = data;
                }
                try {
                    await next();
                } catch (e) {
                    // 返回错误:
                    ctx.response.status = 400;
                    ctx.response.type =
'application/json';
                    ctx.response.body = {
                        code: e.code ||
'internal:unknown_error',
                        message: e.message || ''
                    };
                }
            } else {
                await next();
            }
        };
    }
};
```

这个错误处理的好处在于，不但简化了Controller的错误处理（只需要throw，其他不管），并且，在遇到非APIError的错误时，自动转换错误码为`internal:unknown_error`。

受益于async/await语法，我们在middleware中可以直接用`try...catch`捕获异常。如果是callback模式，就无法用`try...catch`捕获，代码结构将混乱得多。

最后，顺便把`APIError`这个对象export出去。

开发REST API

我们先根据`rest-hello`和`view-koa`来创建一个`rest-hello`的工程，结构如下：

```
rest-koa/
|
+- .vscode/
|  |
|  +- launch.json <-- VSCode 配置文件
|
+- controllers/
|  |
|  +- api.js <-- REST API
|  |
|  +- index.js <-- MVC Controllers
|
+- products.js <-- 集中处理Product
|
+- rest.js <-- 支持REST的middleware
|
+- app.js <-- 使用koa的js
|
+- controller.js <-- 扫描注册Controller
|
+- static-files.js <-- 支持静态文件的middleware
|
+- templating.js <-- 支持Nunjucks的middleware
|
+- package.json <-- 项目描述文件
|
+- views/ <-- Nunjucks模板
|
+- static/ <-- 静态资源文件
|
+- node_modules/ <-- npm安装的所有依赖包
```

在`package.json`中，我们需要如下依赖包：

```
"dependencies": {
    "koa": "2.0.0",
    "koa-bodyparser": "3.2.0",
    "koa-router": "7.0.0",
    "nunjucks": "2.4.2",
    "mime": "1.3.4",
    "mz": "2.4.0"
}
```

运行`npm install`安装依赖包。

我们在这个工程中约定了如下规范：

1. REST API的返回值全部是object对象，而不是简单的number、boolean、null或者数组；
2. REST API必须使用前缀`/api/`。

第一条规则实际上是为了方便客户端处理结果。如果返回结果不是object，则客户端反序列化后还需要判断类型。以Objective-C为例，可以直接返回`NSDictionary*`：

```
NSDictionary* dict = [NSJSONSerialization
JSONObjectWithData:jsonData options:0 error:&err];
```

如果返回值可能是number、boolean、null或者数组，则客户端的工作量会大大增加。

Service

为了操作Product，我们用`products.js`封装所有操作，可以把它视为一个Service：

```
var id = 0;

function nextId() {
    id++;
    return 'p' + id;
}

function Product(name, manufacturer, price) {
    this.id = nextId();
    this.name = name;
    this.manufacturer = manufacturer;
    this.price = price;
}

var products = [
    new Product('iPhone 7', 'Apple', 6800),
    new Product('ThinkPad T440', 'Lenovo', 5999),
    new Product('LBP2900', 'Canon', 1099)
];
```

```
module.exports = {
    getProducts: () => {
        return products;
    },

    getProduct: (id) => {
        var i;
        for (i = 0; i < products.length; i++) {
            if (products[i].id === id) {
                return products[i];
            }
        }
        return null;
    },

    createProduct: (name, manufacturer, price) => {
        var p = new Product(name, manufacturer, price);
        products.push(p);
        return p;
    },

    deleteProduct: (id) => {
        var
            index = -1,
            i;
        for (i = 0; i < products.length; i++) {
            if (products[i].id === id) {
                index = i;
                break;
            }
        }
        if (index >= 0) {
            // remove products[index]:
            return products.splice(index, 1)[0];
        }
        return null;
    }
};
```

变量`products`相当于在内存中模拟了数据库，这里是为了简化逻辑。

API

紧接着，我们编写`api.js`，并放到`controllers`目录下：

```
const products = require('../products');

const APIError = require('../rest').APIError;

module.exports = {
```

```
    'GET /api/products': async (ctx, next) => {
        ctx.rest({
            products: products.getProducts()
        });
    },

    'POST /api/products': async (ctx, next) => {
        var p =
products.createProduct(ctx.request.body.name,
ctx.request.body.manufacturer,
parseFloat(ctx.request.body.price));
        ctx.rest(p);
    },

    'DELETE /api/products/:id': async (ctx, next) => {
        console.log(`delete product ${ctx.params.id}...`);
        var p = products.deleteProduct(ctx.params.id);
        if (p) {
            ctx.rest(p);
        } else {
            throw new APIError('product:not_found',
'product not found by id.');
        }
    }
};
```

该API支持GET、POST和DELETE这三个请求。当然，还可以添加更多的API。

编写API时，需要注意：

如果客户端传递了JSON格式的数据（例如，POST请求），则async函数可以通过`ctx.request.body`直接访问已经反序列化的JavaScript对象。这是由`bodyParser()`这个middleware完成的。如果`ctx.request.body`为`undefined`，说明缺少middleware，或者middleware没有正确配置。

如果API路径带有参数，参数必须用`:`表示，例如，`DELETE /api/products/:id`，客户端传递的URL可能就是`/api/products/A001`，参数`id`对应的值就是`A001`，要获得这个参数，我们用`ctx.params.id`。

类似的，如果API路径有多个参数，例如，`/api/products/:pid/reviews/:rid`，则这两个参数分别用`ctx.params.pid`和`ctx.params.rid`获取。

这个功能由koa-router这个middleware提供。

请注意：API路径的参数永远是字符串！

MVC

有了API以后，我们就可以编写MVC，在页面上调用API完成操作。

先在`controllers`目录下创建`index.js`，编写页面入口函数：

```
module.exports = {
    'GET /': async (ctx, next) => {
        ctx.render('index.html');
    }
};
```

然后，我们在`views`目录下创建`index.html`，编写JavaScript代码读取Products：

```
$(function () {
    var vm = new Vue({
        el: '#product-list',
        data: {
            products: []
        }
    });

    $.getJSON('/api/products').done(function (data) {
        vm.products = data.products;
    }).fail(function (jqXHR, textStatus) {
        alert('Error: ' + jqXHR.status);
    });
});
```

与VM对应的HTML如下：

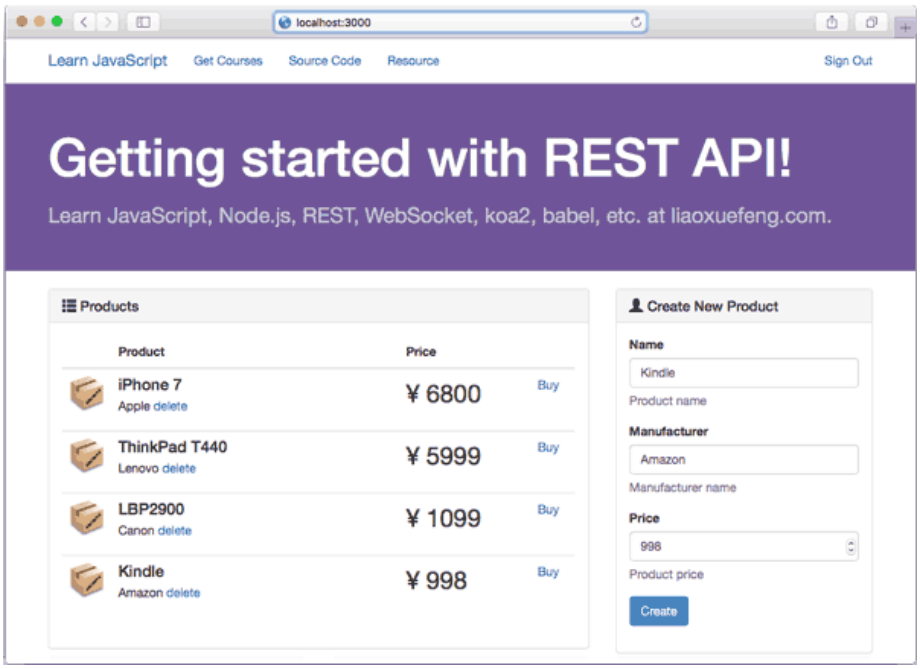
```
<table id="product-list" class="table table-hover">
    <thead>
        <tr>
            <th style="width:50px"></th>
            <th>Product</th>
            <th style="width:150px">Price</th>
        </tr>
    </thead>
    <tbody>
        <tr v-for="p in products">
            <td>
                <img class="media-object"
style="width:40px; height:40px;"
src="/static/images/icon.png">
            </td>
            <td>
                <h4 class="media-heading" v-text="p.name">
</h4>
                <p><span v-text="p.manufacturer"></span>
</p>
            </td>
            <td>
                <p style="font-size:2em">¥ <span v-
text="p.price"></span></p>
```

```
            </td>
        </tr>
    </tbody>
</table>
```

当products变化时，Vue会自动更新表格的内容。

类似的，可以添加创建和删除Product的功能，并且刷新变量`products`的内容，就可以实时更新Product列表。

最终的页面效果如下：



右侧可以通过`POST /api/products`创建新的Product，左侧可以通过`GET /api/products`列出所有Product，并且还可以通过`DELETE /api/products/`来删除某个Product。

MVVM

什么是MVVM？MVVM是Model-View-ViewModel的缩写。

要编写可维护的前端代码绝非易事。我们已经用MVC模式通过koa实现了后端数据、模板页面和控制器的分离，但是，对于前端来说，还不够。

这里有童鞋会问，不是讲Node后端开发吗？怎么又回到前端开发了？

对于一个全栈开发工程师来说，懂前端才会开发出更好的后端程序（不懂前端的后端工程师会设计出非常难用的API），懂后端才会开发出更好的前端程序。程序设计的基本思想在前后端都是通用的，两者并无本质的区别。这和“不想当厨子的裁缝不是好司机”是一个道理。

当我们用Node.js有了一整套后端开发模型后，我们对前端开发也会有新的认识。由于前端开发混合了HTML、CSS和JavaScript，而且页面众多，所以，代码的组织和维护难度其实更加复杂，这就是MVVM出现的原因。

在了解MVVM之前，我们先回顾一下前端发展的历史。

在上个世纪的1989年，欧洲核子研究中心的物理学家Tim Berners-Lee发明了超文本标记语言（HyperText Markup Language），简称HTML，并在1993年成为互联网草案。从此，互联网开始迅速商业化，诞生了一大批商业网站。

最早的HTML页面是完全静态的网页，它们是预先编写好的存放在Web服务器上的html文件。浏览器请求某个URL时，Web服务器把对应的html文件扔给浏览器，就可以显示html文件的内容了。

如果要针对不同的用户显示不同的页面，显然不可能给成千上万的用户准备好成千上万的不同的html文件，所以，服务器就需要针对不同的用户，动态生成不同的html文件。一个最直接的想法就是利用C、C++这些编程语言，直接向浏览器输出拼接后的字符串。这种技术被称为CGI：Common Gateway Interface。

很显然，像新浪首页这样的复杂的HTML是不可能通过拼字符串得到的。于是，人们又发现，其实拼字符串的时候，大多数字符串都是HTML片段，是不变的，变化的只有少数和用户相关的数据，所以，又出现了新的创建动态HTML的方式：ASP、JSP和PHP——分别由微软、SUN和开源社区开发。

在ASP中，一个asp文件就是一个HTML，但是，需要替换的变量用特殊的`<%=var%>`标记出来了，再配合循环、条件判断，创建动态HTML就比CGI要容易得多。

但是，一旦浏览器显示了一个HTML页面，要更新页面内容，唯一的方法就是重新向服务器获取一份新的HTML内容。如果浏览器想要自己修改HTML页面的内容，就需要等到1995年年底，JavaScript被引入到浏览器。

有了JavaScript后，浏览器就可以运行JavaScript，然后，对页面进行一些修改。JavaScript还可以通过修改HTML的DOM结构和CSS来实现一些动画效果，而这些功能没法通过服务器完成，必须在浏览器实现。

用JavaScript在浏览器中操作HTML，经历了若干发展阶段：

第一阶段，直接用JavaScript操作DOM节点，使用浏览器提供的原生API：

```
var dom = document.getElementById('name');
dom.innerHTML = 'Homer';
dom.style.color = 'red';
```

第二阶段，由于原生API不好用，还要考虑浏览器兼容性，jQuery横空出世，以简洁的API迅速俘获了前端开发者的芳心：

```
$('#name').text('Homer').css('color', 'red');
```

第三阶段，MVC模式，需要服务器端配合，JavaScript可以在前端修改服务器渲染后的数据。

现在，随着前端页面越来越复杂，用户对于交互性要求也越来越高，想要写出Gmail这样的页面，仅仅用jQuery是远远不够的。MVVM模型应运而生。

MVVM最早由微软提出来，它借鉴了桌面应用程序的MVC思想，在前端页面中，把Model用纯JavaScript对象表示，View负责显示，两者做到了最大限度的分离。

把Model和View关联起来的就是ViewModel。ViewModel负责把Model的数据同步到View显示出来，还负责把View的修改同步回Model。

ViewModel如何编写？需要用JavaScript编写一个通用的ViewModel，这样，就可以复用整个MVVM模型了。

一个MVVM框架和jQuery操作DOM相比有什么区别？

我们先看用jQuery实现的修改两个DOM节点的例子：

```
<!-- HTML -->
<p>Hello, <span id="name">Bart</span>!</p>
<p>You are <span id="age">12</span>.</p>
```

Hello, **Bart**!

You are **12**.

用jQuery修改name和age节点的内容：

```
var name = 'Homer';
var age = 51;

$('#name').text(name);
$('#age').text(age);
// 执行代码并观察页面变化
```

如果我们使用MVVM框架来实现同样的功能，我们首先并不关心DOM的结构，而是关心数据如何存储。最简单的数据存储方式是使用JavaScript对象：

```
var person = {
    name: 'Bart',
    age: 12
};
```

我们把变量`person`看作Model，把HTML某些DOM节点看作View，并假定它们之间被关联起来了。

要把显示的name从`Bart`改为`Homer`，把显示的age从`12`改为`51`，我们并不操作DOM，而是直接修改JavaScript对象：

```
'use strict';
person.name = 'Homer';
person.age = 51;
// 执行代码并观察页面变化
```

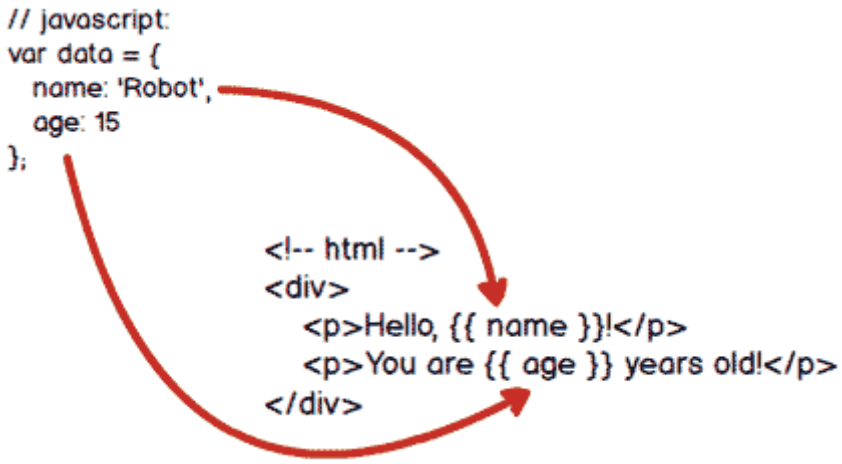
执行上面的代码，我们惊讶地发现，改变JavaScript对象的状态，会导致DOM结构作出对应的变化！这让我们的关注点从如何操作DOM变成了如何更新JavaScript对象的状态，而操作JavaScript对象比DOM简单多了！

这就是MVVM的设计思想：关注Model的变化，让MVVM框架去自动更新DOM的状态，从而把开发者从操作DOM的繁琐步骤中解脱出来！

单向绑定

MVVM就是在前端页面上，应用了扩展的MVC模式，我们关心Model的变化，MVVM框架自动把Model的变化映射到DOM结构上，这样，用户看到的页面内容就会随着Model的变化而更新。

例如，我们定义好一个JavaScript对象作为Model，并且把这个Model的两个属性绑定到DOM节点上：



经过MVVM框架的自动转换，浏览器就可以直接显示Model的数据了：

现在问题来了：MVVM框架哪家强？

目前，常用的MVVM框架有：

Angular：Google出品，名气大，但是很难用；

Backbone.js：入门非常困难，因为自身API太多；

Ember：一个大而全的框架，想写个Hello world都很困难。

我们选择MVVM的目标应该是入门容易，安装简单，能直接在页面写JavaScript，需要更复杂的功能时又能扩展支持。

所以，综合考察，最佳选择是尤雨溪大神开发的MVVM框架：Vue.js

目前，Vue.js的最新版本是2.0，我们会使用2.0版本作为示例。

我们首先创建基于koa的Node.js项目。虽然目前我们只需要在HTML静态页面中编写MVVM，但是很快我们就需要和后端API进行交互，因此，要创建基于koa的项目结构如下：

```
hello-vue/
|
+- .vscode/
|  |
|  +- launch.json <-- VSCode 配置文件
|
+- app.js <-- koa app
|
+- static-files.js <-- 支持静态文件的koa middleware
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
|
+- static/ <-- 存放静态资源文件
   |
   +- css/ <-- 存放bootstrap.css等
   |
   +- fonts/ <-- 存放字体文件
   |
   +- js/ <-- 存放各种js文件
   |
   +- index.html <-- 使用MVVM的静态页面
```

这个Node.js项目的主要目的是作为服务器输出静态网页，因此，`package.json`仅需要如下依赖包：

```
"dependencies": {
    "koa": "2.0.0",
    "mime": "1.3.4",
    "mz": "2.4.0"
}
```

使用`npm install`安装好依赖包，然后启动`app.js`，在`index.html`文件中随便写点内容，确保浏览器可以通过`http://localhost:3000/static/index.html`访问到该静态文件。

紧接着，我们在`index.html`中用Vue实现MVVM的一个简单例子。

安装Vue

安装Vue有很多方法，可以用npm或者webpack。但是我们现在的目标是尽快用起来，所以最简单的方法是直接在HTML代码中像引用jQuery一样引用Vue。可以直接使用CDN的地址，例如：

```
<script src="https://unpkg.com/vue@2.0.1/dist/vue.js">
</script>
```

也可以把`vue.js`文件下载下来，放到项目的`/static/js`文件夹中，使用本地路径：

```
<script src="/static/js/vue.js"></script>
```

这里需要注意，`vue.js`是未压缩的用于开发的版本，它会在浏览器console中输出很多有用的信息，帮助我们调试代码。当开发完毕，需要真正发布到服务器时，应该使用压缩过的`vue.min.js`，它会移除所有调试信息，并且文件体积更小。

编写MVVM

下一步，我们就可以在HTML页面中编写JavaScript代码了。我们的Model是一个JavaScript对象，它包含两个属性：

```
{
    name: 'Robot',
    age: 15
}
```

而负责显示的是DOM节点可以用`{{ name }}`和`{{ age}}`来引用Model的属性：

```
<div id="vm">
    <p>Hello, {{ name }}!</p>
    <p>You are {{ age }} years old!</p>
</div>
```

最后一步是用Vue把两者关联起来。*要特别注意的是*，在`内部编写的JavaScript代码，需要用jQuery把MVVM的初始化代码推迟到页面加载完毕后执行，否则，直接在`内执行MVVM代码时，DOM节点尚未被浏览器加载，初始化将失败。正确的写法如下：

```
<html>
<head>

<!-- 引用jQuery -->
<script src="/static/js/jquery.min.js"></script>

<!-- 引用Vue -->
<script src="/static/js/vue.js"></script>
```

```
<script>
// 初始化代码:
$(function () {
    var vm = new Vue({
        el: '#vm',
        data: {
            name: 'Robot',
            age: 15
        }
    });
    window.vm = vm;
});
</script>

</head>

<body>

    <div id="vm">
        <p>Hello, {{ name }}!</p>
        <p>You are {{ age }} years old!</p>
    </div>

</body>
<html>
```

我们创建一个VM的核心代码如下：

```
var vm = new Vue({
    el: '#vm',
    data: {
        name: 'Robot',
        age: 15
    }
});
```

其中，`el`指定了要把Model绑定到哪个DOM根节点上，语法和jQuery类似。这里的`'#vm'`对应ID为`vm`的一个``节点：

```
<div id="vm">
    ...
</div>
```

在该节点以及该节点内部，就是Vue可以操作的View。Vue可以自动把Model的状态映射到View上，但是*不能*操作View范围之外的其他DOM节点。

然后，`data`属性指定了Model，我们初始化了Model的两个属性`name`和`age`，在View内部的`节点上，可以直接用{{ name }}`引用Model的某个属性。

一切正常的话，我们在浏览器中访问`http://localhost:3000/static/index.html`，可以看到页面输出为：

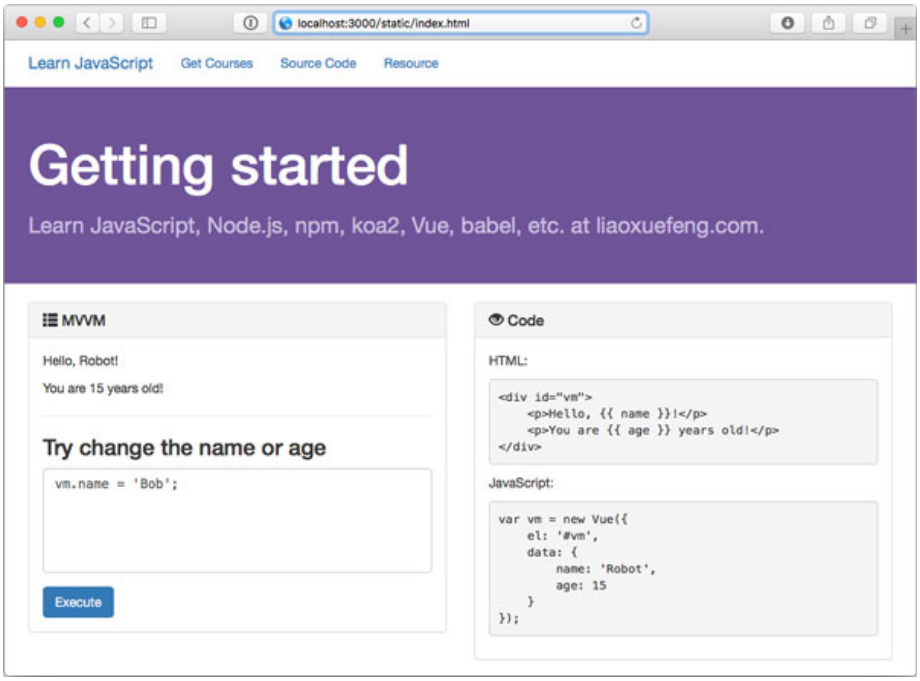
```
Hello, Robot!
You are 15 years old!
```

如果打开浏览器console，因为我们用代码`window.vm = vm`，把VM变量绑定到了window对象上，所以，可以直接修改VM的Model：

```
window.vm.name = 'Bob'
```

执行上述代码，可以观察到页面立刻发生了变化，原来的`Hello, Robot!`自动变成了`Hello, Bob!`。Vue作为MVVM框架会自动监听Model的任何变化，在Model数据变化时，更新View的显示。这种Model到View的绑定我们称为单向绑定。

经过CSS修饰后的页面如下：



可以在页面直接输入JavaScript代码改变Model，并观察页面变化。

单向绑定

在Vue中，可以直接写`{{ name }}`绑定某个属性。如果属性关联的是对象，还可以用多个`.`引用，例如，`{{ address.zipcode }}`。

另一种单向绑定的方法是使用Vue的指令`v-text`，写法如下：

```
<p>Hello, <span v-text="name"></span>!</p>
```

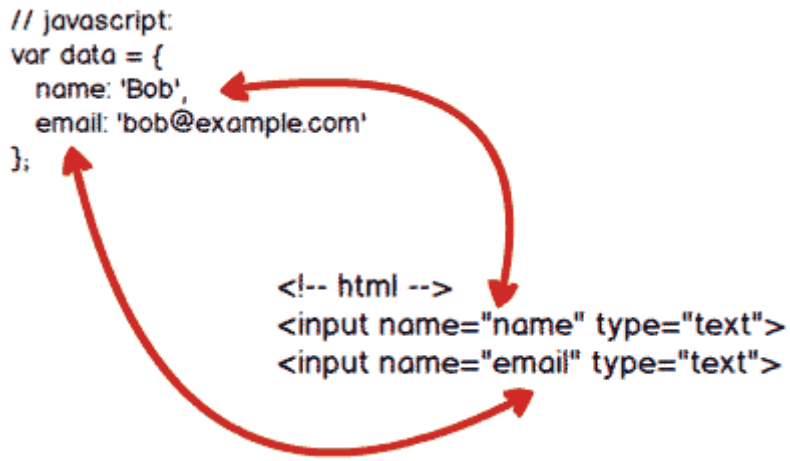
这种写法是把指令写在HTML节点的属性上，它会被Vue解析，该节点的文本内容会被绑定为Model的指定属性，注意不能再写双花括号`{{ }}`。

双向绑定

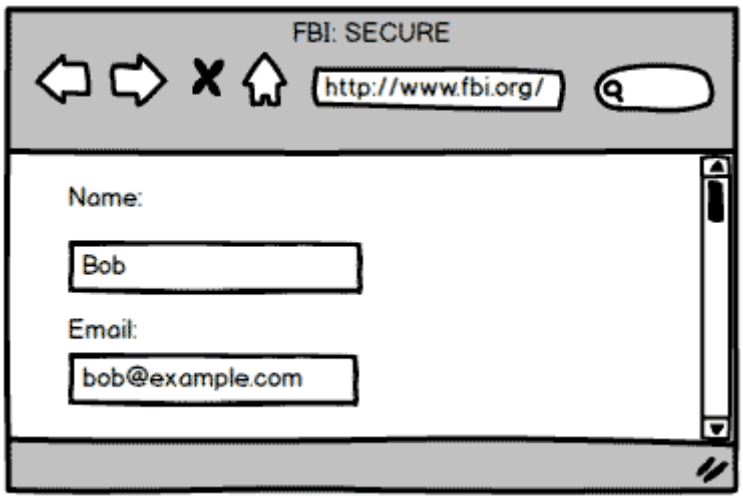
单向绑定非常简单，就是把Model绑定到View，当我们用JavaScript代码更新Model时，View就会自动更新。

有单向绑定，就有双向绑定。如果用户更新了View，Model的数据也自动被更新了，这种情况就是双向绑定。

什么情况下用户可以更新View呢？填写表单就是一个最直接的例子。当用户填写表单时，View的状态就被更新了，如果此时MVVM框架可以自动更新Model的状态，那就相当于我们把Model和View做了双向绑定：



在浏览器中，当用户修改了表单的内容时，我们绑定的Model会自动更新：



在Vue中，使用双向绑定非常容易，我们仍然先创建一个VM实例：

```
$(function () {
    var vm = new Vue({
        el: '#vm',
        data: {
            email: '',
            name: ''
        }
    });
    window.vm = vm;
});
```

然后，编写一个HTML FORM表单，并用`v-model`指令把某个``和Model的某个属性作双向绑定：

```
<form id="vm" action="#">
    <p><input v-model="email"></p>
    <p><input v-model="name"></p>
</form>
```

我们可以在表单中输入内容，然后在浏览器console中用`window.vm.$data`查看Model的内容，也可以用`window.vm.name`查看Model的`name`属性，它的值和FORM表单对应的``是一致的。

如果在浏览器console中用JavaScript更新Model，例如，执行`window.vm.name='Bob'`，表单对应的``内容就会立刻更新。

除了`可以和字符串类型的属性绑定外，其他类型的`也可以和相应数据类型绑定：

多个checkbox可以和数组绑定：

```
<label><input type="checkbox" v-model="language"
value="zh"> Chinese</label>
<label><input type="checkbox" v-model="language"
value="en"> English</label>
<label><input type="checkbox" v-model="language"
value="fr"> French</label>
```

对应的Model为：

```
language: ['zh', 'en']
```

单个checkbox可以和boolean类型变量绑定：

```
<input type="checkbox" v-model="subscribe">
```

对应的Model为：

```
subscribe: true; // 根据checkbox是否选中为true/false
```

下拉框``绑定的是字符串，但是要注意，绑定的是value而非用户看到的文本：

```
<select v-model="city">
    <option value="bj">Beijing</option>
    <option value="sh">Shanghai</option>
    <option value="gz">Guangzhou</option>
</select>
```

对应的Model为：

```
city: 'bj' // 对应option的value
```

双向绑定最大的好处是我们不再需要用jQuery去查询表单的状态，而是直接获得了用JavaScript对象表示的Model。

处理事件

当用户提交表单时，传统的做法是响应`onsubmit`事件，用jQuery获取表单内容，检查输入是否有效，最后提交表单，或者用AJAX提交表单。

现在，获取表单内容已经不需要了，因为双向绑定直接让我们获得了表单内容，并且获得了合适的数据类型。

响应`onsubmit`事件也可以放到VM中。我们在``元素上使用指令：

```
<form id="vm" v-on:submit.prevent="register">
```

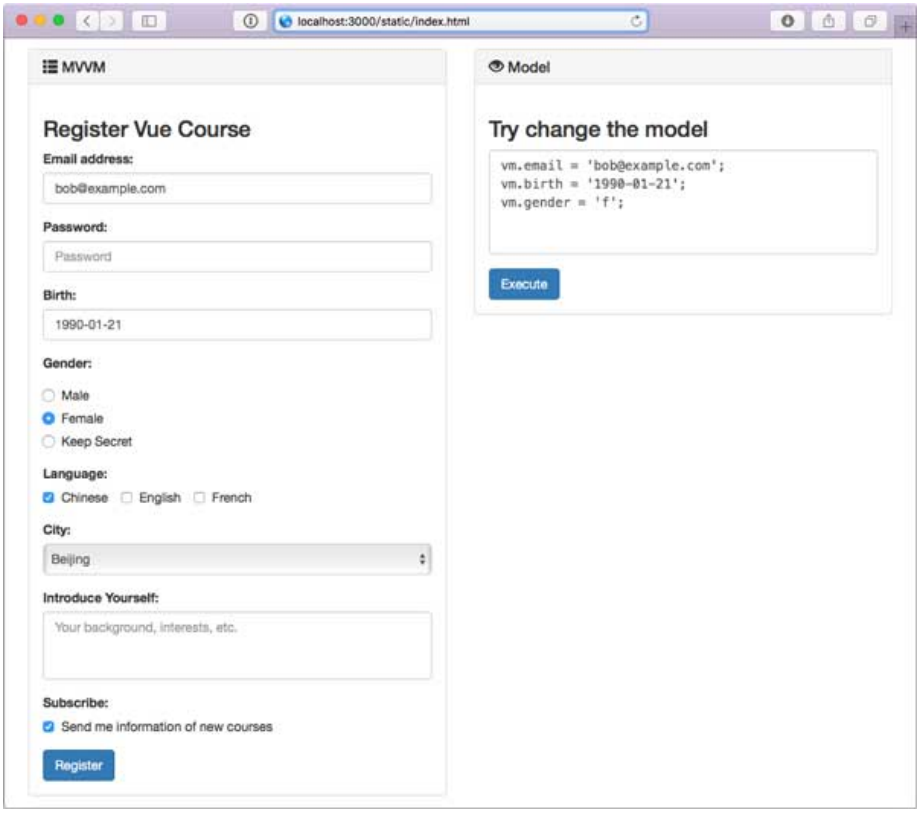
其中，`v-on:submit="register"`指令就会自动监听表单的`submit`事件，并调用`register`方法处理该事件。使用`.prevent`表示阻止事件冒泡，这样，浏览器不再处理`的submit`事件。

因为我们指定了事件处理函数是`register`，所以需要在创建VM时添加一个`register`函数：

```
var vm = new Vue({
    el: '#vm',
    data: {
        ...
    },
    methods: {
        register: function () {
            // 显示JSON格式的Model:
            alert(JSON.stringify(this.$data));
            // TODO: AJAX POST...
        }
    }
});
```

在`register()`函数内部，我们可以用AJAX把JSON格式的Model发送给服务器，就完成了用户注册的功能。

使用CSS修饰后的页面效果如下：



同步DOM结构

除了简单的单向绑定和双向绑定，MVVM还有一个重要的用途，就是让Model和DOM的结构保持同步。

我们用一个TODO的列表作为示例，从用户角度看，一个TODO列表在DOM结构的表现形式就是一组``节点：

```
<ol>
    <li>
        <dl>
            <dt>产品评审</dt>
            <dd>新款iPhone上市前评审</dd>
        </dl>
    </li>
    <li>
        <dl>
            <dt>开发计划</dt>
            <dd>与PM确定下一版Android开发计划</dd>
        </dl>
    </li>
    <li>
        <dl>
            <dt>VC会议</dt>
            <dd>敲定C轮5000万美元融资</dd>
        </dl>
    </li>
</ol>
```

而对应的Model可以用JavaScript数组表示：

```
todos: [
    {
        name: '产品评审',
        description: '新款iPhone上市前评审'
    },
    {
        name: '开发计划',
        description: '与PM确定下一版Android开发计划'
    },
    {
        name: 'VC会议',
        description: '敲定C轮5000万美元融资'
    }
]
```

使用MVVM时，当我们更新Model时，DOM结构会随着Model的变化而自动更新。当`todos`数组增加或删除元素时，相应的DOM节点会增加`或者删除`节点。

在Vue中，可以使用`v-for`指令来实现：

```
<ol>
    <li v-for="t in todos">
        <dl>
            <dt>{{ t.name }}</dt>
            <dd>{{ t.description }}</dd>
        </dl>
    </li>
</ol>
```

`v-for`指令把数组和一组`元素绑定了。在`元素内部，用循环变量`t`引用某个属性，例如，`{{ t.name }}`。这样，我们只关心如何更新Model，不关心如何增删DOM节点，大大简化了整个页面的逻辑。

我们可以在浏览器console中用`window.vm.todos[0].name='计划有变'`查看View的变化，或者通过`window.vm.todos.push({name:'新计划',description:'blabla...'})`来增加一个数组元素，从而自动添加一个``元素。

需要注意的是，Vue之所以能够监听Model状态的变化，是因为JavaScript语言本身提供了Proxy或者Object.observe()机制来监听对象状态的变化。但是，对于数组元素的赋值，却没有办法直接监听，因此，如果我们直接对数组元素赋值：

```
vm.todos[0] = {
    name: 'New name',
    description: 'New description'
};
```

会导致Vue无法更新View。

正确的方法是不要对数组元素赋值，而是更新：

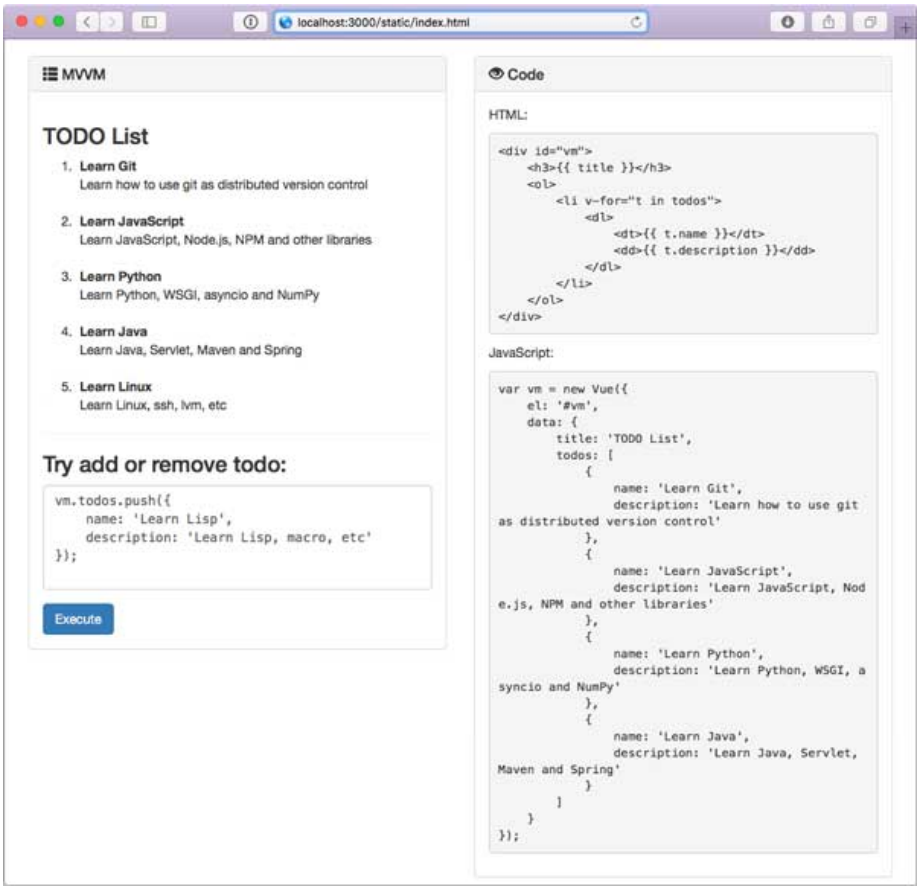
```
vm.todos[0].name = 'New name';
vm.todos[0].description = 'New description';
```

或者，通过`splice()`方法，删除某个元素后，再添加一个元素，达到“赋值”的效果：

```
var index = 0;
var newElement = {...};
vm.todos.splice(index, 1, newElement);
```

Vue可以监听数组的`splice`、`push`、`unshift`等方法调用，所以，上述代码可以正确更新View。

用CSS修饰后的页面效果如下：



集成API

在上一节中，我们用Vue实现了一个简单的TODO应用。通过对Model的更新，DOM结构可以同步更新。

现在，如果要把这个简单的TODO应用变成一个用户能使用的Web应用，我们需要解决几个问题：

1. 用户的TODO数据应该从后台读取；
2. 对TODO的增删改必须同步到服务器后端；

3. 用户在View上必须能够修改TODO。

第1个和第2个问题都是和API相关的。只要我们实现了合适的API接口，就可以在MVVM内部更新Model的同时，通过API把数据更新反映到服务器端，这样，用户数据就保存到了服务器端，下次打开页面时就可以读取TODO列表。

我们在`vue-todo`的基础上创建`vue-todo-2`项目，结构如下：

```
vue-todo-2/
|
+- .vscode/
|  |
|  +- launch.json <-- VSCode 配置文件
|
+- app.js <-- koa app
|
+- static-files.js <-- 支持静态文件的koa middleware
|
+- controller.js <-- 支持路由的koa middleware
|
+- rest.js <-- 支持REST的koa middleware
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
|
+- controllers/ <-- 存放Controller
|  |
|  +- api.js <-- REST API
|
+- static/ <-- 存放静态资源文件
   |
   +- css/ <-- 存放bootstrap.css等
   |
   +- fonts/ <-- 存放字体文件
   |
   +- js/ <-- 存放各种js文件
   |
   +- index.html <-- 使用MVVM的静态页面
```

在`api.js`文件中，我们用数组在服务器端模拟一个数据库，然后实现以下4个API：

- GET /api/todos：返回所有TODO列表；
- POST /api/todos：创建一个新的TODO，并返回创建后的对象；
- PUT /api/todos/:id：更新一个TODO，并返回更新后的对象；
- DELETE /api/todos/:id：删除一个TODO。

和上一节的TODO数据结构相比，我们需要增加一个`id`属性，来唯一标识一个TODO。

准备好API后，在Vue中，我们如何把Model的更新同步到服务器端？

有两个方法，一是直接用jQuery的AJAX调用REST API，不过这种方式比较麻烦。

第二个方法是使用vue-resource这个针对Vue的扩展，它可以给VM对象加上一个`$resource`属性，通过`$resource`来方便地操作API。

使用vue-resource只需要在导入vue.js后，加一行`导入vue-resource.min.js`文件即可。可以直接使用CDN的地址：

```
<script
src="https://cdn.jsdelivr.net/vue.resource/1.0.3/vue-
resource.min.js"></script>
```

我们给VM增加一个`init()`方法，读取TODO列表：

```
var vm = new Vue({
    el: '#vm',
    data: {
        title: 'TODO List',
        todos: []
    },
    created: function () {
        this.init();
    },
    methods: {
        init: function () {
            var that = this;

 that.$resource('/api/todos').get().then(function (resp) {
                // 调用API成功时调用json()异步返回结果:
                resp.json().then(function (result) {
                    // 更新VM的todos:
                    that.todos = result.todos;
                });
            }, function (resp) {
                // 调用API失败:
                alert('error');
            });
        }
    }
});
```

注意到创建VM时，`created`指定了当VM初始化成功后的回调函数，这样，`init()`方法会被自动调用。

类似的，对于添加、修改、删除的操作，我们也需要往VM中添加对应的函数。以添加为例：

```
var vm = new Vue({
    ...
    methods: {
```

```
        ...
        create: function (todo) {
            var that = this;

 that.$resource('/api/todos').save(todo).then(function
(resp) {
                resp.json().then(function (result) {
                    that.todos.push(result);
                });
            }, showError);
        },
        update: function (todo, prop, e) {
            ...
        },
        remove: function (todo) {
            ...
        }
    }
});
```

添加操作需要一个额外的表单，我们可以创建另一个VM对象`vmAdd`来对表单作双向绑定，然后，在提交表单的事件中调用`vm`对象的`create`方法：

```
var vmAdd = new Vue({
    el: '#vmAdd',
    data: {
        name: '',
        description: ''
    },
    methods: {
        submit: function () {
            vm.create(this.$data);
        }
    }
});
```

`vmAdd`和FORM表单绑定：

```
<form id="vmAdd" action="#0" v-on:submit.prevent="submit">
    <p><input type="text" v-model="name"></p>
    <p><input type="text" v-model="description"></p>
    <p><button type="submit">Add</button></p>
</form>
```

最后，把`vm`绑定到对应的DOM：

```
<div id="vm">
    <h3>{{ title }}</h3>
    <ol>
        <li v-for="t in todos">
            <dl>
                <dt contenteditable="true" v-
on:blur="update(t, 'name', $event)">{{ t.name }}</dt>
                <dd contenteditable="true" v-
on:blur="update(t, 'description', $event)">{{
t.description }}</dd>
                <dd><a href="#0" v-
on:click="remove(t)">Delete</a></dd>
            </dl>
        </li>
    </ol>
</div>
```

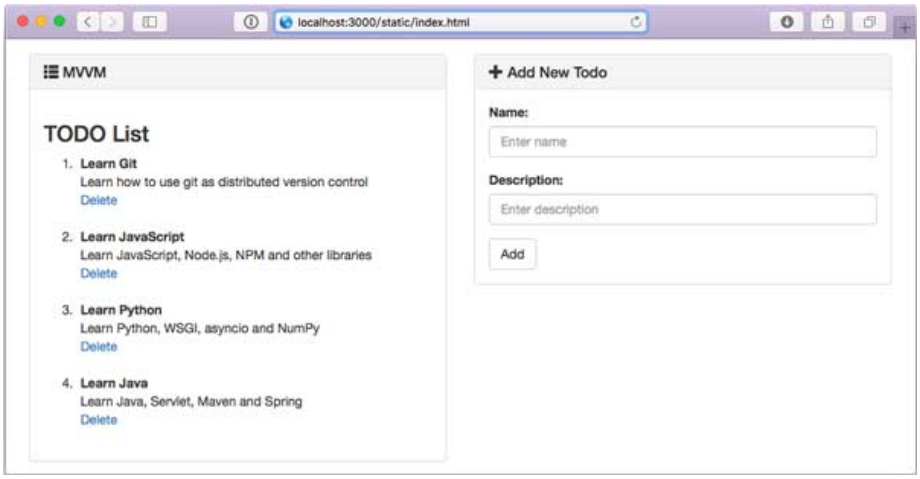
这里我们用`contenteditable="true"`让DOM节点变成可编辑的，用`v-on:blur="update(t, 'name', $event)"`在编辑结束时调用`update()`方法并传入参数，特殊变量`$event`表示DOM事件本身。

删除TODO是通过对`节点绑定v-on:click事件并调用remove()`方法实现的。

如果一切无误，我们就可以先启动服务器，然后在浏览器中访问`http://localhost:3000/static/index.html`，对TODO进行增删改等操作，操作结果会保存在服务器端。

通过Vue和vue-resource插件，我们用简单的几十行代码就实现了一个真正可用的TODO应用。

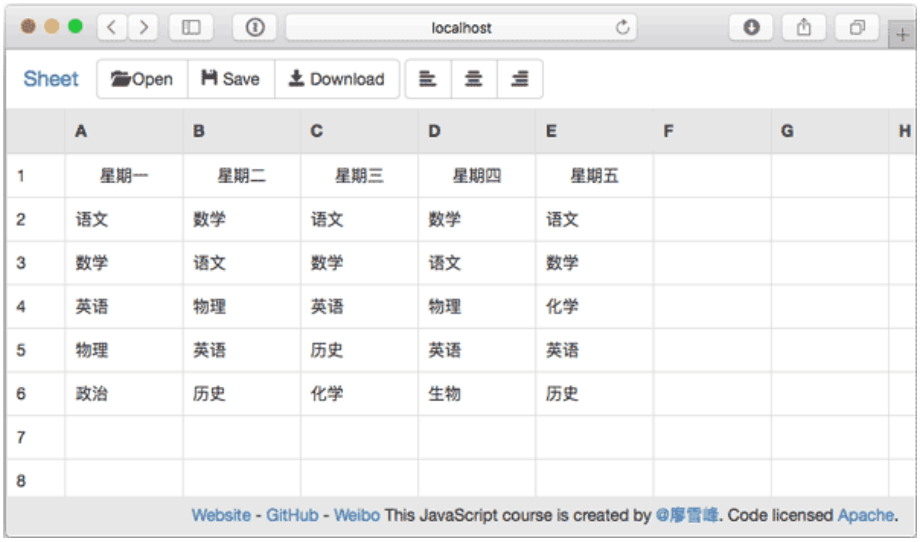
使用CSS修饰后的页面效果如下：



在线电子表格

利用MVVM，很多非常复杂的前端页面编写起来就非常容易了。这得益于我们把注意力放在Model的结构上，而不怎么关心DOM的操作。

本节我们演示如何利用Vue快速创建一个在线电子表格：



	A	B	C	D	E	F	G	H
1	星期一	星期二	星期三	星期四	星期五			
2	语文	数学	语文	数学	语文			
3	数学	语文	数学	语文	数学			
4	英语	物理	英语	物理	化学			
5	物理	英语	历史	英语	英语			
6	政治	历史	化学	生物	历史			
7								
8								

首先，我们定义Model的结构，它的主要数据就是一个二维数组，每个单元格用一个JavaScript对象表示：

```
data: {
    title: 'New Sheet',
    header: [ // 对应首行 A, B, C...
        { row: 0, col: 0, text: '' },
        { row: 0, col: 1, text: 'A' },
        { row: 0, col: 2, text: 'B' },
        { row: 0, col: 3, text: 'C' },
        ...
        { row: 0, col: 10, text: 'J' }
    ],
    rows: [
        [
            { row: 1, col: 0, text: '1' },
            { row: 1, col: 1, text: '' },
            { row: 1, col: 2, text: '' },
            ...
            { row: 1, col: 10, text: '' },
        ],
        [
            { row: 2, col: 0, text: '2' },
            { row: 2, col: 1, text: '' },
            { row: 2, col: 2, text: '' },
            ...
            { row: 2, col: 10, text: '' },
        ],
        ...
        [
            { row: 10, col: 0, text: '10' },
            { row: 10, col: 1, text: '' },
            { row: 10, col: 2, text: '' },
            ...
            { row: 10, col: 10, text: '' },
        ]
    ],
    selectedRowIndex: 0, // 当前活动单元格的row
```

```
    selectedColIndex: 0 // 当前活动单元格的col
}
```

紧接着，我们就可以把Model的结构映射到一个``上：

```
<table id="sheet">
    <thead>
        <tr>
            <th v-for="cell in header" v-text="cell.text">
</th>
        </tr>
    </thead>
    <tbody>
        <tr v-for="tr in rows">
            <td v-for="cell in tr" v-text="cell.text">
</td>
        </tr>
    </tbody>
</table>
```

现在，用Vue把Model和View关联起来，这个电子表格的原型已经可以运行了！

下一步，我们想在单元格内输入一些文本，怎么办？

因为不是所有单元格都可以被编辑，首行和首列不行。首行对应的是，默认是不可编辑的，首列对应的是第一列的，所以，需要判断某个是否可编辑，我们用v-bind`指令给某个DOM元素绑定对应的HTML属性：

```
<td v-for="cell in tr" v-
bind:contenteditable="cell.contentEditable" v-
text="cell.text"></td>
```

在Model中给每个单元格对象加上`contentEditable`属性，就可以决定哪些单元格可编辑。

最后，给`绑定click事件，记录当前活动单元格的row和col，再绑定blur事件，在单元格内容编辑结束后更新Model：

```
<td v-for="cell in tr" v-on:click="focus(cell)" v-
on:blur="change" ...></td>
```

对应的两个方法要添加到VM中：

```
var vm = new Vue({
    ...
    methods: {
        focus: function (cell) {
            this.selectedRowIndex = cell.row;
            this.selectedColIndex = cell.col;
```

```
        },
        change: function (e) {
            // change事件传入的e是DOM事件
            var
                rowIndex = this.selectedRowIndex,
                colIndex = this.selectedColIndex,
                text;
            if (rowIndex > 0 && colIndex > 0) {
                text = e.target.innerText; // 获取td的
innerText
                this.rows[rowIndex - 1][colIndex].text =
text;
            }
        }
    }
});
```

现在，单元格已经可以编辑，并且用户的输入会自动更新到Model中。

如果我们要给单元格的文本添加格式，例如，左对齐或右对齐，可以给Model对应的对象添加一个`align`属性，然后用`v-bind:style`绑定到`上：

```
<td v-for="cell in tr" ... v-bind:style="{ textAlign:
cell.align }"></td>
```

然后，创建工具栏，给左对齐、居中对齐和右对齐按钮编写`click`事件代码，调用`setAlign()`函数：

```
function setAlign(align) {
    var
        rowIndex = vm.selectedRowIndex,
        colIndex = vm.selectedColIndex,
        row, cell;
    if (rowIndex > 0 && colIndex > 0) {
        row = vm.rows[rowIndex - 1];
        cell = row[colIndex];
        cell.align = align;
    }
}

// 给按钮绑定事件:
$('#cmd-left').click(function () { setAlign('left'); });
$('#cmd-center').click(function () { setAlign('center');
});
$('#cmd-right').click(function () { setAlign('right'); });
```

现在，点击某个单元格，再点击右对齐按钮，单元格文本就变成右对齐了。

类似的，可以继续添加其他样式，例如字体、字号等。

MVVM的适用范围

从几个例子我们可以看到，MVVM最大的优势是编写前端逻辑非常复杂的页面，尤其是需要大量DOM操作的逻辑，利用MVVM可以极大地简化前端页面的逻辑。

但是MVVM不是万能的，它的目的是为了解决复杂的前端逻辑。对于以展示逻辑为主的页面，例如，新闻，博客、文档等，*不能*使用MVVM展示数据，因为这些页面需要被搜索引擎索引，而搜索引擎无法获取使用MVVM并通过API加载的数据。

所以，需要SEO（Search Engine Optimization）的页面，不能使用MVVM展示数据。不需要SEO的页面，如果前端逻辑复杂，就适合使用MVVM展示数据，例如，工具类页面，复杂的表单页面，用户登录后才能操作的页面等等。

自动化工具