

《搭建 Node.js 开发环境》

本课程假设大家都是在 Linux 或者 Mac 下面。至于使用 Windows 并坚持玩新技术的同学，我坚信他们一定有着过人的、甚至是不可告人的兼容性 bug 处理能力，所以这部分同学麻烦在课程无法继续时，自行兼容一下。

不久前公司刚发一台新 Mac 给我，所以我对于在新环境中安装 Node.js 的过程还是记忆犹新的。

其实这过程特别简单：

先安装一个 nvm (<https://github.com/creationix/nvm>)

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.25.2/install.sh | bash
```

nvm 的全称是 Node Version Manager，之所以需要这个工具，是因为 Node.js 的各种特性都没有稳定下来，所以我们经常由于老项目或尝新的原因，需要切换各种版本。

安装完成后，你的 shell 里面应该就有个 nvm 命令了，调用它试试

```
$ nvm
```

当看到有输出时，则 nvm 安装成功。

安装 Node.js

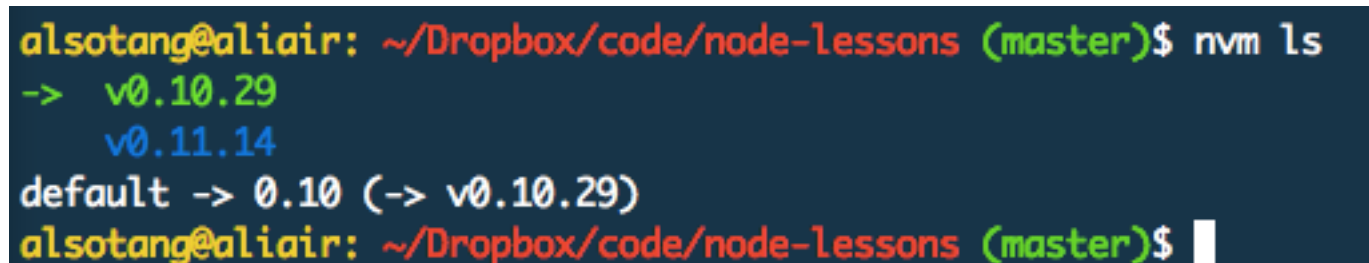
使用 nvm 的命令安装 Node.js 最新稳定版，现在是 v0.12.3。

```
$ nvm install 0.12
```

安装完成后，查看一下

```
$ nvm ls
```

这时候可以看到自己安装的所有 Node.js 版本，输出应如下：

A terminal window screenshot with a dark background. The prompt is 'alsotang@aliair: ~/Dropbox/code/node-lessons (master)'. The command 'nvm ls' has been executed. The output shows two installed versions: 'v0.10.29' with a green arrow pointing to it, and 'v0.11.14'. Below this, it says 'default -> 0.10 (-> v0.10.29)'. The prompt is repeated at the bottom.

(图 1)

那个绿色小箭头的意思就是现在正在使用的版本，我这里是 v0.10.29。我还安装了 v0.11.14，但它并非我当前使用的版本。

如果你那里没有出现绿色小箭头的話，告诉 nvm 你要使用 0.12.x 版本

```
$ nvm use 0.12
```

然后再次查看，这时候小箭头应该出现了。

OK，我们在终端中输入

```
$ node
```

REPL(read-eval-print loop) 应该就出来了，那我们就成功了。

随便敲两行命令玩玩吧。

比如 > while (true) {}, 这时你的 CPU 应该会飆高。

完善安装

上述过程完成后，有时会出现，当开启一个新的 shell 窗口时，找不到 node 命令的情况。

这种情况一般来自两个原因

一、shell 不知道 nvm 的存在

二、nvm 已经存在，但是没有 default 的 Node.js 版本可用。

解决方式：

一、检查 ~/.profile 或者 ~/.bash_profile 中有没有这样两句

```
export NVM_DIR="/Users/YOURUSERNAME/.nvm"  
[ -s "$NVM_DIR/nvm.sh" ] && . "$NVM_DIR/nvm.sh" # This loads nvm
```

没有的话，加进去。

这两句会在 bash 启动的时候被调用，然后注册 nvm 命令。

二、

调用

```
$ nvm ls
```

看看像不像上述图 1 中一样，有 default 的指向。

如果没有的话，执行

```
$ nvm alias default 0.12
```

再

```
$ nvm ls
```

看一下

《一个最简单的 express 应用》

目标

建立一个 lesson1 项目，在其中编写代码。当在浏览器中访问 <http://localhost:3000/> 时，输出 Hello World。

挑战

访问 <http://localhost:3000/> 时，输出 你好，世界。

知识点

1. 包管理器 npm 。使用 npm 安装包，并自动安装所需依赖。
2. 框架 express 。学习新建 express 实例，并定义 routes ，产生输出。

课程内容

按照惯例，我们来个 helloworld 入门。

包管理器 npm

npm 可以自动管理包的依赖. 只需要安装你想要的包, 不必考虑这个包的依赖包.

在 PHP 中, 包管理使用的 **Composer**, python 中, 包管理使用 **easy_install** 或者 **pip**, ruby 中我们使用 **gem**。而在 Node.js 中, 对应就是 **npm**, npm 是 **Node.js Package Manager** 的意思。

框架 Express

express 是 Node.js 应用最广泛的 web 框架, 现在是 4.x 版本, 它非常薄。跟 Rails 比起来, 完全两个极端。

express 的官网是 <http://expressjs.com/>, 我常常上去看它的 API。

首先我们需要得到一个 express。

不同于 ruby 的 gem 装在全局, Node.js 的依赖是以项目为单位管理的, 直接就安装在项目的 **node_modules** 目录下, 而且每个依赖都可以有指定版本的其他依赖, 这些依赖像一棵树一样。根据我自己的使用经验来说, npm 的体验在 pip 和 gem 之上。

OK, 新建一个文件夹叫 lesson1 的, 进去里面安装 express

```
$ mkdir lesson1 && cd lesson1
# 这里没有从官方 npm 安装, 而是使用了大淘宝的 npm 镜像
$ npm install express --registry=https://registry.npm.taobao.org
```

安装完成后, 我们的 lesson1 目录下应该会出现一个 **node_modules** 文件夹, ls 看看

```
$ ls node_modules
```

里面如果出现 express 文件夹则说明安装成功。

或者 npm 命令提供更清晰直观的显示:

```
$ npm list
```

我们继续应用程序的编写。

新建一个 app.js 文件

```
$ touch app.js
```

copy 进去这些代码

```
// 这句的意思就是引入 `express` 模块, 并将它赋予 `express` 这个变量等待使用。
```

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

```
// 调用 express 实例, 它是一个函数, 不带参数调用时, 会返回一个 express 实例, 将这个变量赋予 app 变量。
```

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

```
// app 本身有很多方法, 其中包括最常用的 get、post、put/patch、delete, 在这里我们调用其中的 get 方法, 为我们的 `/'
```

```
// 这个 handler 函数会接收 req 和 res 两个对象, 他们分别是请求的 request 和 response。
```

```
// request 中包含了浏览器传来的各种信息, 比如 query 啊, body 啊, headers 啊之类的, 都可以通过 req 对象访问到。
```

```
// res 对象, 我们一般不从里面取信息, 而是通过它来定制我们向浏览器输出的信息, 比如 header 信息, 比如想要向浏览器输出的
```

```
app.get('/', function (req, res) {
```

```
  res.send('Hello World');
```

```
});
```

```
// 定义好我们 app 的行为之后, 让它监听本地的 3000 端口。这里的第二个函数是个回调函数, 会在 listen 动作成功后执行, 我
```

```
app.listen(3000, function () {
```

```
  console.log('app is listening at port 3000');
```

```
});
```

执行

```
$ node app.js
```

这时候我们的 app 就跑起来了, 终端中会输出 **app is listening at port 3000**。这时我们打开浏览器, 访问 **http://localhost:3000/**, 会出现 **Hello World**。如果没有出现的话, 肯定是上述哪一步弄错了, 自己调试一下。

补充知识

在这个例子中, node 代码监听了 3000 端口, 用户通过访问 `http://localhost:3000/` 得到了内容, 为什么呢?

端口

端口的作用: 通过端口来区分出同一电脑内不同应用或者进程, 从而实现一条物理网线 (通过分组交换技术-比如 **internet**) 同时链接多个程序 Port_(computer_networking)

端口号是一个 16 位的 uint, 所以其范围为 1 to 65535 (对 TCP 来说, port 0 被保留, 不能被使用. 对于 UDP 来说, source 端的端口号是可选的, 为 0 时表示无端口).

`app.listen(3000)`, 进程就被打标, 电脑接收到的 3000 端口的网络消息就会被发送给我们启动的这个进程

URL

RFC1738 定义的 url 格式笼统版本 `<scheme>:<scheme-specific-part>`, scheme 有我们很熟悉的 `http`、`https`、`ftp`, 以及著名的 `ed2k`, `thunder`。

通常我们熟悉的 url 定义成这个样子

`<scheme>://<user>:<password>@<host>:<port>/<url-path>`

用过 ftp 的估计能体会这么长的, 网页上很少带 auth 信息, 所以就精简成这样:

`<scheme>://<host>:<port>/<url-path>`

在上面的例子中, `scheme=http`, `host=localhost`, `port=3000`, `url-path=/`, 再联想对照一下浏览器端 `window.location` 对象。著名的 `localhost`, 你可以在电脑的 hosts 文件上找到

在这篇文章中提到: **URI schemes are frequently and incorrectly referred to as "protocols", or specifically as URI protocols or URL protocols, since most were originally designed to be used with a particular protocol, and often have the same name**, 比较认同这个观点, 尤其是今天移动设备的时代里, android 和 ios 的开发中大量使用 uri 作为跨 app 通讯通道, 把 scheme 理解为协议略狭隘了。

尾声

在了解完端口和 url 之后, 再去看例子代码, 相信应该好理解很多。有必要的話, 还可以在解剖一下 express 的 use 逻辑, 对峙 `http.createServer`, 相信还有火花, :)

《学习使用外部模块》

目标

建立一个 lesson2 项目, 在其中编写代码。

当在浏览器中访问 `http://localhost:3000/?q=alsotang` 时, 输出 alsotang 的 md5 值, 即 `bdd5e57b5c0040f9dc23d430846e68a3`。

挑战

访问 `http://localhost:3000/?q=alsotang` 时, 输出 alsotang 的 sha1 值, 即 `e3c766d71667567e18f77869c65cd62f6a1b9ab9`。

知识点

1. 学习 `req.query` 的用法
2. 学习建立 `package.json` 来管理 Node.js 项目。

课程内容

卧槽，不写 package.json 就写项目我觉得好不爽啊，所以这个 lesson2 我就得跟大家介绍一下 package.json 这个文件的用法了。

简单说来呢，这个 package.json 文件就是定义了项目的各种元信息，包括项目的名称，git repo 的地址，作者等等。最重要的是，其中定义了我们项目的依赖，这样这个项目在部署时，我们就不必将 `node_modules` 目录也上传到服务器，服务器在拿到我们的项目时，只需要执行 `npm install`，则 npm 会自动读取 package.json 中的依赖并安装在项目的 `node_modules` 下面，然后程序就可以在服务器上跑起来了。

本课程的每个 lesson 里面的示例代码都会带上一份 package.json，大家可以去看看它的大概样子。

我们来新建一个 lesson2 项目，并生成一份它的 package.json。

```
$ mkdir lesson2 && cd lesson2
$ npm init
```

OK，这时会要求我们输入一些信息，乱填就好了，反正这个地方也不用填依赖关系。

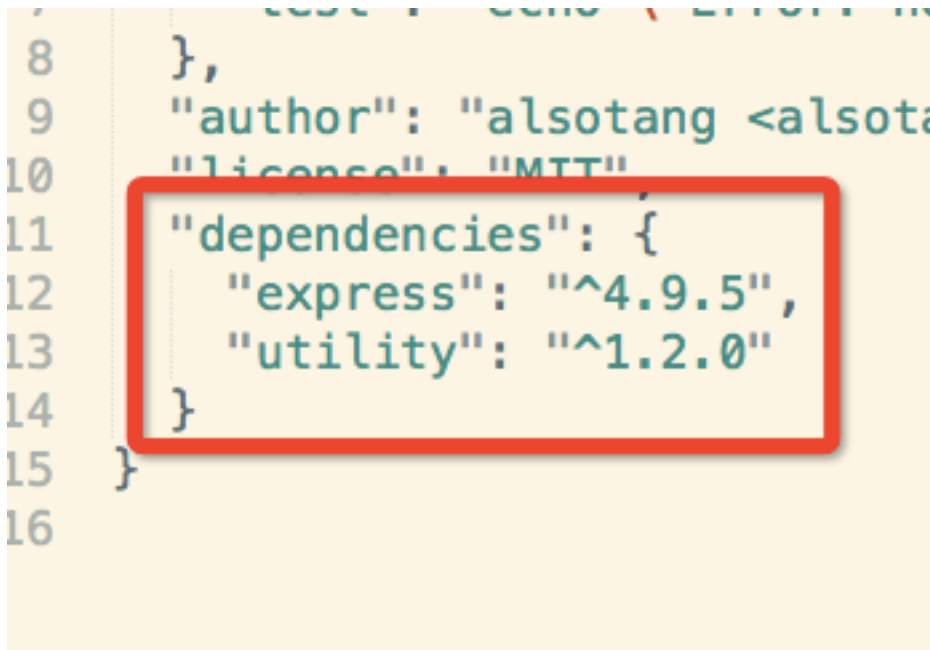
`npm init` 这个命令的作用就是帮我们交互式地生成一份最简单的 package.json 文件，`init` 是 `initialize` 的意思，初始化。

当乱填信息完毕之后，我们的目录下就会有 `package.json` 文件了。

这时我们来安装依赖，这次的应用，我们依赖 `express` 和 `utility` 两个模块。

```
$ npm install express utility --save
```

这次的安装命令与上节课的命令有两点不同，一是没有指定 `registry`，没有指定的情况下，默认从 npm 官方安装，上次我们是从淘宝的源安装的。二是多了个 `--save` 参数，这个参数的作用，就是会在你安装依赖的同时，自动把这些依赖写入 package.json。命令执行完成之后，查看 package.json，会发现多了一个 `dependencies` 字段，如下图：



```
8 },
9 "author": "alsotang <alsota
10 "license": "MIT"
11 "dependencies": {
12   "express": "^4.9.5",
13   "utility": "^1.2.0"
14 }
15 }
16
```

这时查看 `node_modules` 目录，会发现有两个文件夹，分别是 `express` 和 `utility`

```
alsotang@aliair: ~/Dropbox/code/node-lessons/lesson2 (master)$ ls -l node_modules/
total 0
drwxr-xr-x 10 alsotang staff 340 10 6 00:26 express
drwxr-xr-x 11 alsotang staff 374 10 6 00:25 utility
```

我们开始写应用层的代码，建立一个 `app.js` 文件，复制以下代码进去：

```
// 引入依赖
var express = require('express');
var utility = require('utility');

// 建立 express 实例
```

```

var app = express();

app.get('/', function (req, res) {
  // 从 req.query 中取出我们的 q 参数。
  // 如果是 post 传来的 body 数据，则是在 req.body 里面，不过 express 默认不处理 body 中的信息，需要引入 https://
  // 如果分不清什么是 query，什么是 body 的话，那就需要补一下 http 的知识了
  var q = req.query.q;

  // 调用 utility.md5 方法，得到 md5 之后的值
  // 之所以使用 utility 这个库来生成 md5 值，其实只是习惯问题。每个人都有自己习惯的技术堆栈，
  // 我刚入职阿里的时候跟着苏千和朴灵混，所以也混到了不少他们的技术堆栈，仅此而已。
  // utility 的 github 地址: https://github.com/node-modules/utility
  // 里面定义了很多常用且比较杂的辅助方法，可以去看看
  var md5Value = utility.md5(q);

  res.send(md5Value);
});

```

```

app.listen(3000, function (req, res) {
  console.log('app is running at port 3000');
});

```

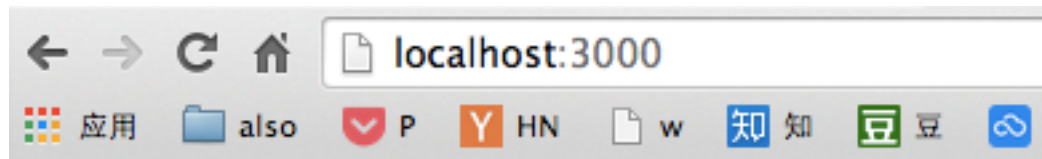
OK，运行我们的程序

```
$ node app.js
```

访问 <http://localhost:3000/?q=alsotang>，完成。

题外话

如果直接访问 <http://localhost:3000/> 会抛错



```

TypeError: Not a string or buffer
    at Hash.update (crypto.js:209:17)
    at Object.hash (/Users/alsotang/Dropbox/code/
    at Object.md5 (/Users/alsotang/Dropbox/code/
    at /Users/alsotang/Dropbox/code/node-lessons,
    at Layer.handle [as handle_request] (/Users/also
    at next (/Users/alsotang/Dropbox/code/node-le
    at Route.dispatch (/Users/alsotang/Dropbox/co
    at Layer.handle [as handle_request] (/Users/also
    at /Users/alsotang/Dropbox/code/node-lessons,
    at Function.proto.process_params (/Users/alsota

```

可以看到，这个错误是从 `crypto.js` 中抛出的。

这是因为，当我们不传入 `q` 参数时，`req.query.q` 取到的值是 `undefined`，`utility.md5` 直接使用了这个空值，导致下层的 `crypto` 抛错。

《使用 superagent 与 cheerio 完成简单爬虫》

目标

建立一个 lesson3 项目，在其中编写代码。

当在浏览器中访问 `http://localhost:3000/` 时，输出 CNode(`https://cnodejs.org/`) 社区首页的所有帖子标题和链接，以 json 的形式。

输出示例：

```
[
  {
    "title": "【公告】发招聘帖的同学留意一下这里",
    "href": "http://cnodejs.org/topic/541ed2d05e28155f24676a12"
  },
  {
    "title": " 发布一款 Sublime Text 下的 JavaScript 语法高亮插件",
    "href": "http://cnodejs.org/topic/54207e2efffeb6de3d61f68f"
  }
]
```

挑战

访问 `http://localhost:3000/` 时，输出包括主题的作者，

示例：

```
[
  {
    "title": "【公告】发招聘帖的同学留意一下这里",
    "href": "http://cnodejs.org/topic/541ed2d05e28155f24676a12",
    "author": "alsotang"
  },
  {
    "title": " 发布一款 Sublime Text 下的 JavaScript 语法高亮插件",
    "href": "http://cnodejs.org/topic/54207e2efffeb6de3d61f68f",
    "author": "otheruser"
  }
]
```

知识点

1. 学习使用 superagent 抓取网页
2. 学习使用 cheerio 分析网页

课程内容

Node.js 总是吹牛逼说自己异步特性多么多么厉害，但是对于初学者来说，要找一个能好好利用异步的场景不容易。我想来想去，爬虫的场景就比较适合，没事就异步并发地爬几个网站玩玩。

本来想教大家怎么爬 github 的 api 的，但是 github 有 rate limit 的限制，所以只好牺牲一下 CNode 社区（国内最专业的 Node.js 开源技术社区），教大家怎么去爬它了。

我们这回需要用到三个依赖，分别是 express, superagent 和 cheerio。

先介绍一下，

superagent(`http://visionmedia.github.io/superagent/`) 是个 http 方面的库，可以发起 get 或 post 请求。

cheerio(<https://github.com/cheeriojs/cheerio>) 大家可以理解成一个 Node.js 版的 jquery, 用来从网页中以 css selector 取数据, 使用方式跟 jquery 一样一样的。

还记得我们怎么新建一个项目吗?

1. 新建一个文件夹, 进去之后 `npm init`
2. 安装依赖 `npm install --save PACKAGE_NAME`
3. 写应用逻辑

我们应用的核心逻辑长这样

```
app.get('/', function (req, res, next) {
  // 用 superagent 去抓取 https://cnodejs.org/ 的内容
  superagent.get('https://cnodejs.org/')
    .end(function (err, sres) {
      // 常规的错误处理
      if (err) {
        return next(err);
      }
      // sres.text 里面存储着网页的 html 内容, 将它传给 cheerio.load 之后
      // 就可以得到一个实现了 jquery 接口的变量, 我们习惯性地将它命名为 `$`
      // 剩下就都是 jquery 的内容了
      var $ = cheerio.load(sres.text);
      var items = [];
      $('#topic_list .topic_title').each(function (idx, element) {
        var $element = $(element);
        items.push({
          title: $element.attr('title'),
          href: $element.attr('href')
        });
      });
      res.send(items);
    });
});
```

OK, 一个简单的爬虫就是这么简单。这里我们还没有利用到 Node.js 的异步并发特性。不过下两章内容都是关于异步控制的。记得好好看看 superagent 的 API, 它把链式调用的风格玩到了极致。

《使用 eventproxy 控制并发》

目标

建立一个 lesson4 项目, 在其中编写代码。

代码的入口是 `app.js`, 当调用 `node app.js` 时, 它会输出 CNode(<https://cnodejs.org/>) 社区首页的所有主题的标题, 链接和第一条评论, 以 json 的格式。

输出示例:

```
[
  {
    "title": "【公告】发招聘帖的同学留意一下这里",
    "href": "http://cnodejs.org/topic/541ed2d05e28155f24676a12",
    "comment1": "呵呵呵呵"
  },
  {
    "title": "发布一款 Sublime Text 下的 JavaScript 语法高亮插件",
    "href": "http://cnodejs.org/topic/54207e2efffeb6de3d61f68f",
    "comment1": "沙发!"
  }
]
```


挑战

以上文目标为基础，输出 `comment1` 的作者，以及他在 `cnode` 社区的积分值。

示例：

```
[
  {
    "title": "【公告】发招聘帖的同学留意一下这里",
    "href": "http://cnodejs.org/topic/541ed2d05e28155f24676a12",
    "comment1": " 呵呵呵呵",
    "author1": "auser",
    "score1": 80
  },
  ...
]
```

知识点

1. 体会 Node.js 的 `callback hell` 之美
2. 学习使用 `eventproxy` 这一利器控制并发

课程内容

注意，`cnodejs.org` 网站有并发连接数的限制，所以当请求发送太快的时候会导致返回值为空或报错。建议一次抓取 3 个主题即可。文中的 40 只是为了方便讲解

这一章我们来到了 Node.js 最牛逼的地方——异步并发的内容了。

上一课我们介绍了如何使用 `superagent` 和 `cheerio` 来取主页内容，那只需要发起一次 `http get` 请求就能办到。但这次，我们需要取出每个主题的第一条评论，这就要求我们对每个主题的连接发起请求，并用 `cheerio` 去取出其中的第一条评论。

CNode 目前每一页有 40 个主题，于是我们就需要发起 $1 + 40$ 个请求，来达到我们这一课的目标。

后者的 40 个请求，我们并发地发起:)，而且不会遇到多线程啊锁什么的，Node.js 的并发模型跟多线程不同，抛却那些观念。更具体一点的话，比如异步到底为何异步，Node.js 为何单线程却能并发这类走近科学的问题，我就不打算讲了。对于这方面有兴趣的同学，强烈推荐 @ 朴灵的《九浅一深 Node.js》：<http://book.douban.com/subject/25768396/>。

有些逼格比较高的朋友可能听说过 `promise` 和 `generator` 这类概念。不过我呢，只会讲 `callback`，主要原因是我个人只喜欢 `callback`。

这次课程我们需要用到三个库：`superagent` `cheerio` `eventproxy`(<https://github.com/JacksonTian/eventproxy>)

手脚架的工作各位自己来，我们一步一步来一起写出这个程序。

首先 `app.js` 应该长这样

```
var eventproxy = require('eventproxy');
var superagent = require('superagent');
var cheerio = require('cheerio');
// url 模块是 Node.js 标准库里面的
// http://nodejs.org/api/url.html
var url = require('url');

var cnodeUrl = 'https://cnodejs.org/';

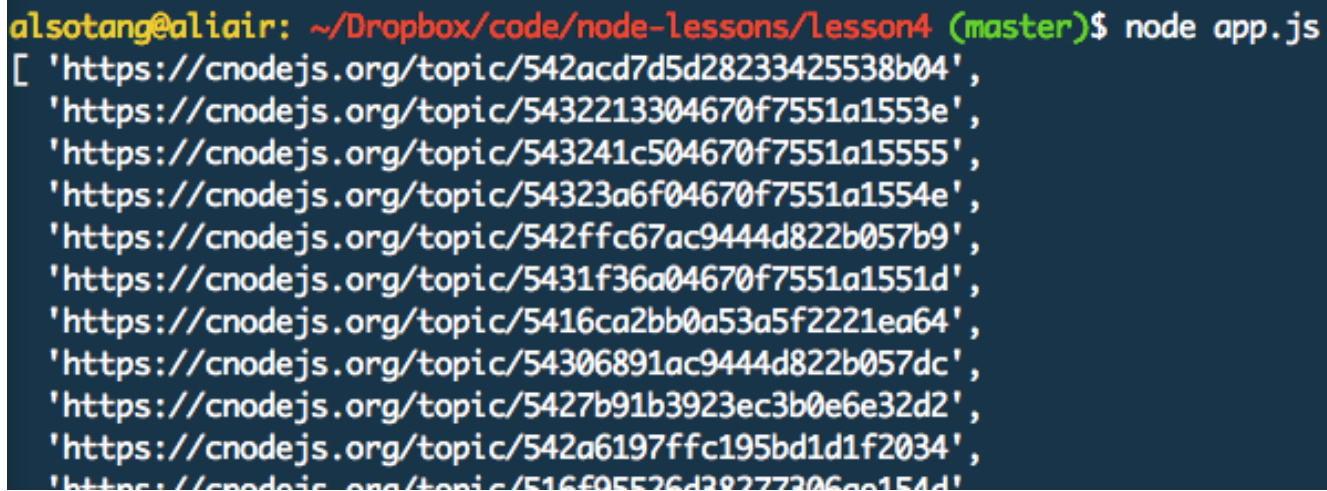
superagent.get(cnodeUrl)
  .end(function (err, res) {
    if (err) {
      return console.error(err);
    }
    var topicUrls = [];
    var $ = cheerio.load(res.text);
```

```
// 获取首页所有的链接
$('#topic_list .topic_title').each(function (idx, element) {
  var $element = $(element);
  // $element.attr('href') 本来的样子是 /topic/542acd7d5d28233425538b04
  // 我们用 url.resolve 来自动推断出完整 url，变成
  // https://cnodejs.org/topic/542acd7d5d28233425538b04 的形式
  // 具体请看 http://nodejs.org/api/url.html#url\_url\_resolve\_from\_to 的示例
  var href = url.resolve(cnodeUrl, $element.attr('href'));
  topicUrls.push(href);
});

console.log(topicUrls);
});
```

运行 node app.js

输出如下图：



```
alsotang@aliar: ~/Dropbox/code/node-lessons/lesson4 (master)$ node app.js
[ 'https://cnodejs.org/topic/542acd7d5d28233425538b04',
  'https://cnodejs.org/topic/5432213304670f7551a1553e',
  'https://cnodejs.org/topic/543241c504670f7551a15555',
  'https://cnodejs.org/topic/54323a6f04670f7551a1554e',
  'https://cnodejs.org/topic/542ffc67ac9444d822b057b9',
  'https://cnodejs.org/topic/5431f36a04670f7551a1551d',
  'https://cnodejs.org/topic/5416ca2bb0a53a5f2221ea64',
  'https://cnodejs.org/topic/54306891ac9444d822b057dc',
  'https://cnodejs.org/topic/5427b91b3923ec3b0e6e32d2',
  'https://cnodejs.org/topic/542a6197ffc195bd1d1f2034',
  'https://cnodejs.org/topic/5416f05526d38277206e0154d'
```

OK，这时候我们已经得到所有 url 的地址了，接下来，我们把这些地址都抓取一遍，就完成了，Node.js 就是这么简单。

抓取之前，还是得介绍一下 eventproxy 这个库。

用 js 写过异步的同学应该都知道，如果你要并发异步获取两三个地址的数据，并且要在获取到数据之后，对这些数据一起进行利用的话，常规的写法是自己维护一个计数器。

先定义一个 `var count = 0`，然后每次抓取成功以后，就 `count++`。如果你是要抓取三个源的数据，由于你根本不知道这些异步操作到底谁先完成，那么每次当抓取成功的时候，就判断一下 `count === 3`。当值为真时，使用另一个函数继续完成操作。

而 eventproxy 就起到了这个计数器的作用，它来帮你管理到底这些异步操作是否完成，完成之后，它会自动调用你提供的处理函数，并将抓取到的数据当参数传过来。

假设我们不使用 eventproxy 也不使用计数器时，抓取三个源的写法是这样的：

```
// 参考 jquery 的 $.get 的方法
$.get("http://data1_source", function (data1) {
  // something
  $.get("http://data2_source", function (data2) {
    // something
    $.get("http://data3_source", function (data3) {
      // something
      var html = fuck(data1, data2, data3);
      render(html);
    });
  });
});
```

上述的代码大家都写过吧。先获取 data1，获取完成之后获取 data2，然后再获取 data3，然后 fuck 它们，进行输出。

但大家应该也想到了，其实这三个源的数据，是可以并行去获取的，data2 的获取并不依赖 data1 的完成，data3 同理也不依赖 data2。

于是我们用计数器来写，会写成这样：

```
(function () {  
    var count = 0;  
    var result = {};  
  
    $.get('http://data1_source', function (data) {  
        result.data1 = data;  
        count++;  
        handle();  
    });  
    $.get('http://data2_source', function (data) {  
        result.data2 = data;  
        count++;  
        handle();  
    });  
    $.get('http://data3_source', function (data) {  
        result.data3 = data;  
        count++;  
        handle();  
    });  
  
    function handle() {  
        if (count === 3) {  
            var html = fuck(result.data1, result.data2, result.data3);  
            render(html);  
        }  
    }  
})();
```

丑的一逼，

也不算丑，主要我写代码好看。

如果我们用 eventproxy，写出来是这样的：

```
var ep = new eventproxy();  
ep.all('data1_event', 'data2_event', 'data3_event', function (data1, data2, data3) {  
    var html = fuck(data1, data2, data3);  
    render(html);  
});  
  
$.get('http://data1_source', function (data) {  
    ep.emit('data1_event', data);  
});  
  
$.get('http://data2_source', function (data) {  
    ep.emit('data2_event', data);  
});  
  
$.get('http://data3_source', function (data) {  
    ep.emit('data3_event', data);  
});
```

好看多了是吧，也就是个高等计数器嘛。

```
ep.all('data1_event', 'data2_event', 'data3_event', function (data1, data2, data3) {});
```

这一句，监听了三个事件，分别是 data1_event, data2_event, data3_event，每次当一个源的数据抓取完成时，就通过 ep.emit() 来告诉 ep 自己，某某事件已经完成了。

当三个事件未同时完成时，ep.emit() 调用之后不会做任何事；当三个事件都完成的时候，就会调用末尾的那个回调函数，来对它们进行统一处理。

eventproxy 提供了不少其他场景所需的 API，但最最常用的用法就是以上的这种，即：

1. 先 var ep = new eventproxy(); 得到一个 eventproxy 实例。


```

Finder
fetch https://cnodejs.org/topic/542acd7d5d28233425538b04 successful
fetch https://cnodejs.org/topic/51ee453af4963ade0ebde85e successful
fetch https://cnodejs.org/topic/542e5c2119dda0dc1d8924bc successful
fetch https://cnodejs.org/topic/5430602cac9444d822b057db successful
fetch https://cnodejs.org/topic/50f90d8edf9e9fcc58a5ee0b successful
fetch https://cnodejs.org/topic/54196ee14566f99164399cdd successful
final:
[ { title: '精华\n\n\n\n      《使用 angular 和 socket.io 搭建聊天室》，求大家拍砖！',
  href: 'https://cnodejs.org/topic/52c3be5f8a716e0b15e28e8c',
  comment1: 'thx' },
  { title: '精华\n\n\n\n      新手献丑，晒几个自己写的小玩意，欢迎拍砖',
  href: 'https://cnodejs.org/topic/5431f36a04670f7551a1551d',
  comment1: '支持原创' },
  { title: 'cnode 社区如果在阿里云上搭一个私有 npm 服务，是否有创业公司会使用呢？',
  href: 'https://cnodejs.org/topic/5427b91b3923ec3b0e6e32d2',
  comment1: '-1' },
  { title: 'npm 安装 express，命令不能找到',
  href: 'https://cnodejs.org/topic/54323a6f04670f7551a1554e',
  comment1: '' },
  { title: 'coffee 为什么每次都会生成 return 语句'

```

完整的代码请查看 lesson4 目录下的 app.js 文件

《使用 async 控制并发》

目标

建立一个 lesson5 项目，在其中编写代码。

代码的入口是 `app.js`，当调用 `node app.js` 时，它会输出 CNode(<https://cnodejs.org/>) 社区首页的所有主题的标题，链接和第一条评论，以 json 的格式。

注意：与上节课不同，并发连接数需要控制在 5 个。

输出示例：

```

[
  {
    "title": "【公告】发招聘帖的同学留意一下这里",
    "href": "http://cnodejs.org/topic/541ed2d05e28155f24676a12",
    "comment1": "呵呵呵呵"
  },
  {
    "title": "发布一款 Sublime Text 下的 JavaScript 语法高亮插件",
    "href": "http://cnodejs.org/topic/54207e2efffeb6de3d61f68f",
    "comment1": "沙发！"
  }
]

```

知识点

1. 学习 `async`(<https://github.com/caolan/async>) 的使用。这里有个详细的 `async` demo 演示：https://github.com/alsotang/async_demo
2. 学习使用 `async` 来控制并发连接数。

课程内容

lesson4 的代码其实是不完美的。为什么这么说，是因为在 lesson4 中，我们一次性发了 40 个并发请求出去，要知道，除去 CNode 的话，别的网站有可能会因为你发出的并发连接数太多而当你是恶意请求，把你的 IP 封掉。

我们在写爬虫的时候，如果有 1000 个链接要去爬，那么不可能同时发出 1000 个并发链接出去对不对？我们需要控制一下并发的数量，比如并发 10 个就好，然后慢慢抓完这 1000 个链接。

用 async 来做这件事很简单。

这次我们要介绍的是 async 的 `mapLimit(arr, limit, iterator, callback)` 接口。另外，还有个常用的控制并发连接数的接口是 `queue(worker, concurrency)`，大家可以去 <https://github.com/caolan/async#queueworker-concurrency> 看看说明。

这回我就不带大家爬网站了，我们来专注知识点：并发连接数控制。

对了，还有个问题是，什么时候用 eventproxy，什么时候使用 async 呢？它们不都是用来做异步流程控制的吗？

我的答案是：

当你需要去多个源（一般是小于 10 个）汇总数据的时候，用 eventproxy 方便；当你需要用到队列，需要控制并发数，或者你喜欢函数式编程思维时，使用 async。大部分场景是前者，所以我个人大部分时间是用 eventproxy 的。

正题开始。

首先，我们伪造一个 `fetchUrl(url, callback)` 函数，这个函数的作用就是，当你通过

```
fetchUrl('http://www.baidu.com', function (err, content) {  
  // do something with `content`  
});
```

调用它时，它会返回 `http://www.baidu.com` 的页面内容回来。

当然，我们这里的返回内容是假的，返回延时是随机的。并且在它被调用时，会告诉你它现在一共被多少个地方并发调用着。

// 并发连接数的计数器

```
var concurrencyCount = 0;  
var fetchUrl = function (url, callback) {  
  // delay 的值在 2000 以内，是个随机的整数  
  var delay = parseInt((Math.random() * 10000000) % 2000, 10);  
  concurrencyCount++;  
  console.log(' 现在的并发数是', concurrencyCount, ', 正在抓取的是', url, ', 耗时' + delay + ' 毫秒');  
  setTimeout(function () {  
    concurrencyCount--;  
    callback(null, url + ' html content');  
  }, delay);  
};
```

我们接着来伪造一组链接

```
var urls = [];  
for(var i = 0; i < 30; i++) {  
  urls.push('http://datasource_' + i);  
}
```

这组链接的长这样：


```
alsotang@aliar: ~/Dropbox/code/node-lessons/lesson5 (master)$ node app.js
[ 'http://datasource_0',
  'http://datasource_1',
  'http://datasource_2',
  'http://datasource_3',
  'http://datasource_4',
  'http://datasource_5',
  'http://datasource_6',
  'http://datasource_7',
  'http://datasource_8',
```

接着，我们使用 `async.mapLimit` 来并发抓取，并获取结果。

```
async.mapLimit(urls, 5, function (url, callback) {
  fetchUrl(url, callback);
}, function (err, result) {
  console.log('final:');
  console.log(result);
});
```

运行输出是这样的：

```
alsotang@aliar: ~/Dropbox/code/node-lessons/lesson5 (master)$ node app.js
现在的并发数是 1，正在抓取的是 http://datasource_0，耗时543毫秒
现在的并发数是 2，正在抓取的是 http://datasource_1，耗时842毫秒
现在的并发数是 3，正在抓取的是 http://datasource_2，耗时1552毫秒
现在的并发数是 4，正在抓取的是 http://datasource_3，耗时743毫秒
现在的并发数是 5，正在抓取的是 http://datasource_4，耗时908毫秒
现在的并发数是 5，正在抓取的是 http://datasource_5，耗时755毫秒
现在的并发数是 5，正在抓取的是 http://datasource_6，耗时1033毫秒
现在的并发数是 5，正在抓取的是 http://datasource_7，耗时69毫秒
现在的并发数是 5，正在抓取的是 http://datasource_8，耗时547毫秒
现在的并发数是 5，正在抓取的是 http://datasource_9，耗时891毫秒
现在的并发数是 5，正在抓取的是 http://datasource_10，耗时1269毫秒
现在的并发数是 5，正在抓取的是 http://datasource_11，耗时1253毫秒
现在的并发数是 5，正在抓取的是 http://datasource_12，耗时154毫秒
```

可以看到，一开始，并发链接数是从 1 开始增长的，增长到 5 时，就不再增加。当其中有任务完成时，再继续抓取。并发连接数始终控制在 5 个。

完整代码请参见 `app.js` 文件。

《测试用例：mocha, should, istanbul》

目标

建立一个 `lesson6` 项目，在其中编写代码。

`main.js`: 其中有个 `fibonacci` 函数。`fibonacci` 的介绍见：http://en.wikipedia.org/wiki/Fibonacci_number。

此函数的定义为 `int fibonacci(int n)`

- 当 `n === 0` 时，返回 0；`n === 1` 时，返回 1；

- $n > 1$ 时, 返回 `fibonacci(n) === fibonacci(n-1) + fibonacci(n-2)`, 如 `fibonacci(10) === 55`;
- n 不可大于 10, 否则抛错, 因为 Node.js 的计算性能没那么强。
- n 也不可小于 0, 否则抛错, 因为没意义。
- n 不为数字时, 抛错。

test/main.test.js: 对 main 函数进行测试, 并使行覆盖率和分支覆盖率都达到 100%。

知识点

1. 学习使用测试框架 mocha : <http://mochajs.org/>
2. 学习使用断言库 should : <https://github.com/tj/should.js>
3. 学习使用测试率覆盖工具 istanbul : <https://github.com/gotwarlost/istanbul>
4. 简单 Makefile 的编写: <http://blog.csdn.net/haol/article/details/2886>

课程内容

首先, 作为一个 Node.js 项目, 先执行 `npm init` 创建 package.json。

其次, 建立我们的 main.js 文件, 编写 fibonacci 函数。

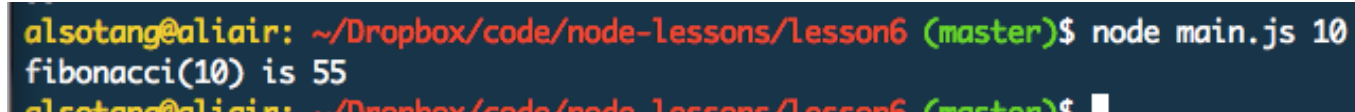
```
var fibonacci = function (n) {
  if (n === 0) {
    return 0;
  }
  if (n === 1) {
    return 1;
  }
  return fibonacci(n-1) + fibonacci(n-2);
};

if (require.main === module) {
  // 如果是直接执行 main.js, 则进入此处
  // 如果 main.js 被其他文件 require, 则此处不会执行。
  var n = Number(process.argv[2]);
  console.log('fibonacci(' + n + ') is', fibonacci(n));
}
```

OK, 这只是个简单的实现。

我们可以执行试试

```
$ node main.js 10
```



```
alstang@aliair: ~/Dropbox/code/node-lessons/lesson6 (master)$ node main.js 10
fibonacci(10) is 55
alstang@aliair: ~/Dropbox/code/node-lessons/lesson6 (master)$
```

嗯, 结果是 55, 符合预期。

接下来我们开始测试驱动开发, 现在简单的实现已经完成, 那我们就对它进行一下简单测试吧。

我们先得把 main.js 里面的 fibonacci 暴露出来, 这个简单。加一句

```
exports.fibonacci = fibonacci; (要是看不懂这句就去补补 Node.js 的基础知识吧)
```

就好了。

然后我们在 test/main.test.js 中引用我们的 main.js, 并开始一个简单的测试。

```
// file: test/main.test.js
var main = require('../main');
var should = require('should');

describe('test/main.test.js', function () {
```

```

    it('should equal 55 when n === 10', function () {
      main.fibonacci(10).should.equal(55);
    });
  });
});

```

把测试先跑通，我们再讲这段测试代码的含义。

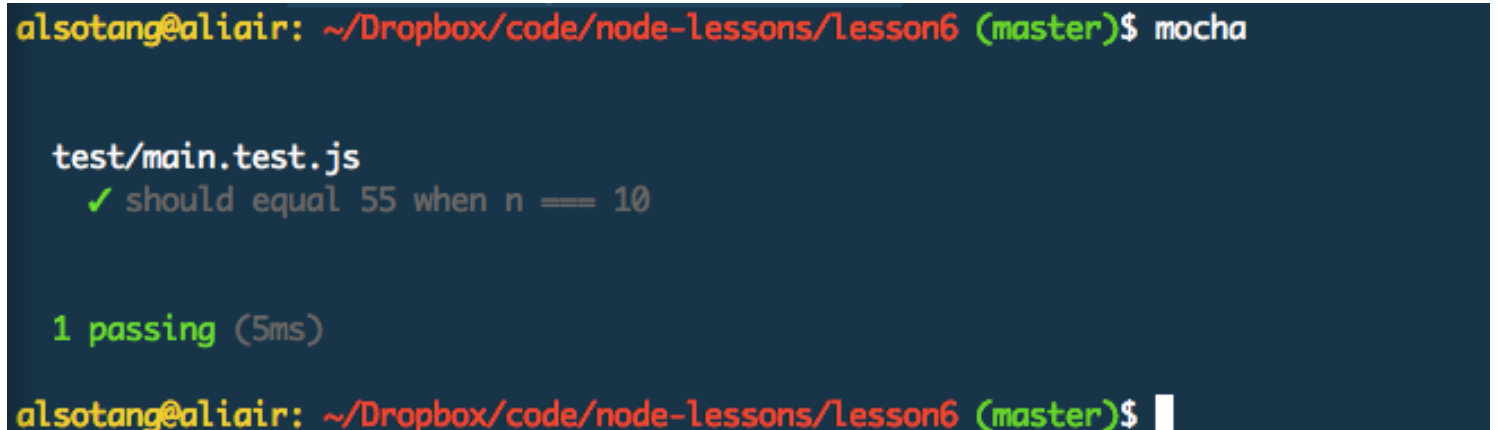
装个全局的 mocha: `$ npm install mocha -g`。

`-g` 与非 `-g` 的区别，就是安装位置的区别，`g` 是 `global` 的意思。如果不加的话，则安装 `mocha` 在你的项目目录下；如果加了，则这个 `mocha` 是安装在全局的，如果 `mocha` 有可执行命令的话，那么这个命令也会自动加入到你系统 `$PATH` 中的某个地方（在我的系统中，是这里 `/Users/alsotang/.nvm/v0.10.29/bin`）

在 `lesson6` 目录下，直接执行

`$ mocha`

输出应如下



```

alsotang@aliair: ~/Dropbox/code/node-lessons/lesson6 (master)$ mocha

test/main.test.js
  ✓ should equal 55 when n === 10

1 passing (5ms)

alsotang@aliair: ~/Dropbox/code/node-lessons/lesson6 (master)$

```

那么，代码中的 `describe` 和 `it` 是什么意思呢？其实就是 BDD 中的那些意思，把它们当做语法来记就好了。

大家来看看 `nodeclub` 中，关于 `topicController` 的测试文件：

<https://github.com/cnodejs/nodeclub/blob/master/test/controllers/topic.test.js>

这文件的内容没有超出之前课程的范围吧。

`describe` 中的字符串，用来描述你要测的主体是什么；`it` 当中，描述具体的 `case` 内容。

而引入的那个 `should` 模块，是个断言库。玩过 `ruby` 的同学应该知道 `rspec`，`rspec` 它把测试框架和断言库的事情一起做了，而在 `Node.js` 中，这两样东西的作用分别是 `mocha` 和 `should` 在协作完成。

`should` 在 `js` 的 `Object` “基类”上注入了一个 `#should` 属性，这个属性中，又有着许许多多的属性可以被访问。

比如测试一个数是不是大于 3，则是 `(5).should.above(3)`；测试一个字符串是否有特定前缀：`'foobar'.should.startsWith('foo')`
`should.js` API 在：<https://github.com/tj/should.js>

`should.js` 如果现在还是 `version 3` 的话，我倒是推荐大家去看看它的 API 和源码；现在 `should` 是 `version 4` 了，API 丑得很，但为了不掉队，我还是一直用着它。我觉得 `expect` 麻烦，所以不用 `expect`，对了，`expect` 也是一个断言库：<https://github.com/LearnBoost/expect.js/>。

回到正题，还记得我们 `fibonacci` 函数的几个要求吗？

- * 当 `n === 0` 时，返回 `0`；`n === 1` 时，返回 `1`；
- * `n > 1` 时，返回 ``fibonacci(n) === fibonacci(n-1) + fibonacci(n-2)``，如 ``fibonacci(10) === 55``；
- * `n` 不可大于 `10`，否则抛错，因为 `Node.js` 的计算性能没那么强。
- * `n` 也不可小于 `0`，否则抛错，因为没意义。
- * `n` 不为数字时，抛错。

我们用测试用例来描述一下这几个要求，更新后的 `main.test.js` 如下：

```

var main = require('../main');
var should = require('should');

describe('test/main.test.js', function () {
  it('should equal 0 when n === 0', function () {
    main.fibonacci(0).should.equal(0);
  });
});

```

```

});

it('should equal 1 when n === 1', function () {
  main.fibonacci(1).should.equal(1);
});

it('should equal 55 when n === 10', function () {
  main.fibonacci(10).should.equal(55);
});

it('should throw when n > 10', function () {
  (function () {
    main.fibonacci(11);
  }).should.throw('n should <= 10');
});

it('should throw when n < 0', function () {
  (function () {
    main.fibonacci(-1);
  }).should.throw('n should >= 0');
});

it('should throw when n isnt Number', function () {
  (function () {
    main.fibonacci(' 呵呵');
  }).should.throw('n should be a Number');
});
});

```

还是比较清晰的吧？

我们这时候跑一下 `$ mocha`，会发现后三个 case 都没过。

于是我们更新 fibonacci 的实现：

```

var fibonacci = function (n) {
  if (typeof n !== 'number') {
    throw new Error('n should be a Number');
  }
  if (n < 0) {
    throw new Error('n should >= 0');
  }
  if (n > 10) {
    throw new Error('n should <= 10');
  }
  if (n === 0) {
    return 0;
  }
  if (n === 1) {
    return 1;
  }

  return fibonacci(n-1) + fibonacci(n-2);
};

```

再跑一次 `$ mocha`，就过了。这就是传说中的测试驱动开发：先把要达到的目的都描述清楚，然后让现有的程序跑不过 case，再修补程序，让 case 通过。

安装一个 istanbul：`$ npm i istanbul -g`

执行 `$ istanbul cover _mocha`

这会比直接使用 mocha 多一行覆盖率的输出，

```
✓ should equal 55 when n == 10
✓ should throw when n > 10
✓ should throw when n < 0
✓ should throw when n isnt Number
```

6 passing (6ms)

Writing coverage object [/Users/alsotang/Dropbox/code/node-lessons/lesson6/coverage/coverage.json]

Writing coverage reports at [/Users/alsotang/Dropbox/code/node-lessons/lesson6/coverage]

Coverage summary

Statements : 87.5% (14/16)

Branches : 91.67% (11/12)

Functions : 100% (1/1)

Lines : 87.5% (14/16)

可以看到，我们其中的分支覆盖率是 91.67%，行覆盖率是 87.5%。

打开 `open coverage/lcov-report/index.html` 看看

Code coverage report for lesson6/main.js

Statements: **87.5% (14 / 16)**

Branches: **91.67% (11 / 12)**

Functions: **100% (1 / 1)**

Lines: **87.5%**

[All files](#) » [lesson6/](#) » [main.js](#)

```
1 1 var fibonacci = function (n) {
2 182   if (typeof n !== 'number') {
3 1     throw new Error('n should be a Number');
4     }
5 181   if (n < 0) {
6 1     throw new Error('n should >= 0')
7     }
8 180   if (n > 10) {
9 1     throw new Error('n should <= 10');
10    }
11 179   if (n === 0) {
12 35     return 0;
13   }
14 144   if (n === 1) {
15 56     return 1;
16   }
17
18 88   return fibonacci(n-1) + fibonacci(n-2);
19 };
20
21 1 exports.fibonacci = fibonacci;
22
23 1 I if (require.main === module) {
24   var n = Number(process.argv[2]);
25   console.log('fibonacci(' + n + ') is', fibonacci(n));
26 }
27
```

Generated by [istanbul](#) at Tu

其实这覆盖率是 100% 的，24 25 两行没法测。

mocha 和 istanbul 的结合是相当无缝的，只要 mocha 跑得动，那么 istanbul 就接得进来。

到此这门课其实就完了，剩下要说的内容，都是些比较细节的。比较懒的同学可以踩坑了之后再回来看。

上面的课程，不完美的地方就在于 mocha 和 istanbul 版本依赖的问题，但为了不引入不必要的复杂性，所以上面就没提到这点了。

假设你有一个项目 A，用到了 mocha 的 version 3，其他人有个项目 B，用到了 mocha 的 version 10，那么如果你 `npm i mocha -g` 装的是 version 3 的话，你用 `$ mocha` 是不兼容 B 项目的。因为 mocha 版本改变之后，很可能语法也变了，对吧。

这时，跑测试用例的正确方法，应该是

1. `$ npm i mocha --save-dev`，装个 mocha 到项目目录中去
2. `$./node_modules/.bin/mocha`，用刚才安装的这个特定版本的 mocha，来跑项目的测试代码。

`./node_modules/.bin` 这个目录下放着我们所有依赖自带的那些可执行文件。

每次输入这个很麻烦对吧？所以我们要引入 Makefile，让 Makefile 帮我们记住复杂的配置。

test:

`./node_modules/.bin/mocha`

cov test-cov:

```
./node_modules/.bin/istanbul cover _mocha
```

```
.PHONY: test cov test-cov
```

这时，我们只需要调用 `make test` 或者 `make cov`，就可以跑我们相应的测试了。

至于 Makefile 怎么写？以及 .PHONY 是什么意思，请看这里：<http://blog.csdn.net/haoel/article/details/2886>，左耳朵耗子陈皓 2004 年的文章。

《浏览器端测试：mocha, chai, phantomjs》

目标

建立一个 lesson7 项目，在其中编写代码，我们暂时命名为 vendor 根据下面的步骤，最终的项目结构应该长这样

这次我们测试的对象是上文提到的 fibonacci 函数

此函数的定义为 `int fibonacci(int n)`

- 当 `n === 0` 时，返回 0；`n === 1` 时，返回 1；
- `n > 1` 时，返回 `fibonacci(n) === fibonacci(n-1) + fibonacci(n-2)`，如 `fibonacci(10) === 55`；

知识点

1. 学习使用测试框架 mocha 进行前端测试: <http://mochajs.org/>
2. 了解全栈的断言库 chai: <http://chaijs.com/>
3. 了解 headless 浏览器 phantomjs: <http://phantomjs.org/>

前端脚本单元测试

lesson6 的内容都是针对后端环境中 node 的一些单元测试方案，出于应用健壮性的考量，针对前端 js 脚本的单元测试也非常重要。而前后端通吃，也是 mocha 的一大特点。

首先，前端脚本的单元测试主要有两个困难需要解决。

1. 运行环境应当在浏览器中，可以操纵浏览器的 DOM 对象，且可以随意定义执行时的 html 上下文。
2. 测试结果应当可以直接反馈给 mocha，判断测试是否通过。

浏览器环境执行

我们首先搭建一个测试原型，用 mocha 自带的脚手架可以自动生成。

```
cd vendor          # 进入我们的项目文件夹
npm i -g mocha     # 安装全局的 mocha 命令行工具
mocha init .       # 生成脚手架
```

mocha 就会自动帮我们生成一个简单的测试原型，目录结构如下

```
.
├── index.html      # 这是前端单元测试的入口
├── mocha.css
├── mocha.js
└── tests.js       # 我们的单元测试代码将在这里编写
```

其中 index.html 是单元测试的入口，tests.js 是我们的测试用例文件。

我们直接在 index.html 插入上述示例的 fibonacci 函数以及断言库 chaijs。

```

<div id="mocha"></div>
<script src='https://cdn.rawgit.com/chaijs/chai/master/chai.js'></script>
<script>
  var fibonacci = function (n) {
    if (n === 0) {
      return 0;
    }
    if (n === 1) {
      return 1;
    }
    return fibonacci(n-1) + fibonacci(n-2);
  };
</script>

```

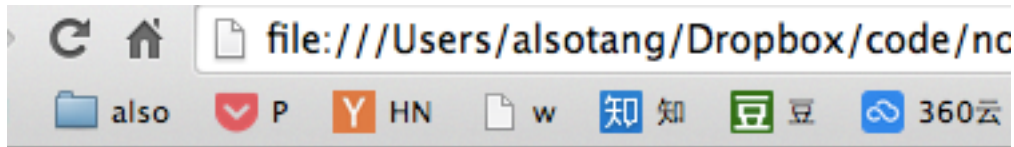
然后在 tests.js 中写入对应测试用例

```

var should = chai.should();
describe('simple test', function () {
  it('should equal 0 when n === 0', function () {
    window.fibonacci(0).should.equal(0);
  });
});

```

这时打开 index.html，可以发现测试结果，我们完成了浏览器端的脚本测试（注意我们调用了 window 对象）



simple test

✓ should equal 0 when n === 0

```
window.fibonacci(0).should.equal(0);
```

测试反馈

mocha 没有提供一个命令行的前端脚本测试环境（因为我们的脚本文件需要运行在浏览器环境中），因此我们使用 phantomjs 帮助我们搭建一个模拟环境。不重复制造轮子，这里直接使用 mocha-phantomjs 帮助我们在命令行运行测试。

首先安装 mocha-phantomjs

```
npm i -g mocha-phantomjs
```

然后在 index.html 的页面下加上这段兼容代码

```
<script>mocha.run()</script>
```


改为

```
<script>
  if (window.initMochaPhantomJS && window.location.search.indexOf('skip') === -1) {
    initMochaPhantomJS()
  }
  mocha.ui('bdd');
  expect = chai.expect;

  mocha.run();
</script>
```

这时候, 我们在命令行中运行

```
mocha-phantomjs index.html --ssl-protocol=any --ignore-ssl-errors=true
```

结果展现是不是和后端代码测试很类似:smile:

更进一步, 我们可以直接在 package.json 的 scripts 中添加 (package.json 通过 npm init 生成, 这里不再赘述)

```
"scripts": {
  "test": "mocha-phantomjs index.html --ssl-protocol=any --ignore-ssl-errors=true"
},
```

将 mocha-phantomjs 作为依赖

```
npm i mocha-phantomjs --save-dev
```

直接运行

```
npm test
```

运行结果如下:

至此, 我们实现了前端脚本的单元测试, 基于 phantomjs 你几乎可以调用所有的浏览器方法, 而 mocha-phantomjs 也可以很便捷地将测试结果反馈到 mocha, 便于后续的持续集成。

《测试用例: supertest》

目标

建立一个 lesson8 项目, 在其中编写代码。

app.js: 其中有个 fibonacci 接口。fibonacci 的介绍见: http://en.wikipedia.org/wiki/Fibonacci_number。

fibonacci 函数的定义为 `int fibonacci(int n)`, 调用函数的路径是 `/fib?n=10`, 然后这个接口会返回 `'55'`。函数的行为定义如下:

- 当 `n === 0` 时, 返回 0; `n === 1` 时, 返回 1;
- `n > 1` 时, 返回 `fibonacci(n) === fibonacci(n-1) + fibonacci(n-2)`, 如 `fibonacci(10) === 55`;
- `n` 不可大于 10, 否则抛错, http status 500, 因为 Node.js 的计算性能没那么强。
- `n` 也不可小于 0, 否则抛错, 500, 因为没意义。
- `n` 不为数字时, 抛错, 500。

test/main.test.js: 对 app 的接口进行测试, 覆盖以上所有情况。

知识点

1. 学习 supertest 的使用 (<https://github.com/tj/supertest>)
2. 复习 mocha, should 的使用

课程内容

这是连续第三节课讲测试了.. 我自己都烦.. 看着烦的可以考虑跳下一课。

OK, 基础知识前面都讲得很多了, 这节课我不会事无巨细地写过程了。

噢, 对了, 说到 fibonacci, Node 中文圈的大神 @ 苏千 (<https://github.com/fengmk2>) 写过一个页面, 对各种语言的 fibonacci 效率进行了测试: <http://fengmk2.cnpmjs.org/blog/2011/fibonacci/nodejs-python-php-ruby-lua.html>。其中, Node 的表现不知道比 Python 和 Ruby 高到哪里去了, 与 CPU 谈笑风生。怀疑 js 的人啊, 都 too simple, sometimes naive。

先来介绍一下 supertest。supertest 是 superagent 的孪生库。他的作者叫 tj, 这是个在 Node.js 的历史上会永远被记住的名字, 因为他一个人撑起了 npm 的半边天。别误会成他是 npm 的开发者, 他的贡献是在 Node.js 的方方面面都贡献了非常高质量和口碑的库, 比如 mocha 是他的, superagent 是他的, express 是他的, should 也是他的, 还有其他很多很多, 比如 koa, 都是他的。如果你更详细点了解一些 Node 圈内的八卦, 一定也会像我一样对 tj 佩服得五体投地。他的 github 首页是: <https://github.com/tj>。

假使你作为一个有志之士, 想要以他为榜样, 跟随他前进的步伐, 那么我指条明路给你, 不收费的: <http://tour.golang.org/>

为什么说 supertest 是 superagent 的孪生库呢, 因为他们的 API 是一模一样的。superagent 是用来抓取页面用的, 而 supertest, 是专门用来配合 express (准确来说是所有兼容 connect 的 web 框架) 进行集成测试的。

将使你有一个 app: `var app = express();`, 想对它的 get 啊, post 接口啊之类的进行测试, 那么只要把它传给 supertest: `var request = require('supertest')(app)`。之后调用 `request.get('/path')` 时, 就可以对 app 的 path 路径进行访问了。它的 API 参照 superagent 的来就好了: <http://visionmedia.github.io/superagent/>。

我们来新建一个项目

```
$ npm init # ..一阵乱填
```

然后安装我们的依赖 (记得去弄清楚 `npm i --save` 与 `npm i --save-dev` 的区别):

```
"devDependencies": {
  "mocha": "^1.21.4",
  "should": "^4.0.4",
  "supertest": "^0.14.0"
},
"dependencies": {
  "express": "^4.9.6"
}
```

接着, 编写 app.js

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

```
// 与之前一样
```

```
var fibonacci = function (n) {
  // typeof NaN === 'number' 是成立的, 所以要判断 NaN
  if (typeof n !== 'number' || isNaN(n)) {
    throw new Error('n should be a Number');
  }
  if (n < 0) {
    throw new Error('n should >= 0');
  }
  if (n > 10) {
    throw new Error('n should <= 10');
  }
  if (n === 0) {
    return 0;
  }
  if (n === 1) {
    return 1;
  }

  return fibonacci(n-1) + fibonacci(n-2);
};
// END 与之前一样
```

```

var app = express();

app.get('/fib', function (req, res) {
  // http 传来的东西默认都是没有类型的，都是 String，所以我们要手动转换类型
  var n = Number(req.query.n);
  try {
    // 为何使用 String 做类型转换，是因为如果你直接给个数字给 res.send 的话，
    // 它会当成是你给了它一个 http 状态码，所以我们明确给 String
    res.send(String(fibonacci(n)));
  } catch (e) {
    // 如果 fibonacci 抛错的话，错误信息会记录在 err 对象的 .message 属性中。
    // 拓展阅读: https://www.joyent.com/developers/node/design/errors
    res
      .status(500)
      .send(e.message);
  }
});

```

// 暴露 app 出去。module.exports 与 exports 的区别请看《深入浅出 Node.js》

```
module.exports = app;
```

```

app.listen(3000, function () {
  console.log('app is listening at port 3000');
});

```

好了，启动一下看看。

```
$ node app.js
```

然后访问 <http://localhost:3000/fib?n=10>，看到 55 就说明启动成功了。再访问 <http://localhost:3000/fib?n=111>，会看到 n should <= 10。

对了，大家去装个 nodemon <https://github.com/remy/nodemon>。

```
$ npm i -g nodemon
```

这个库是专门调试时候使用的，它会自动检测 node.js 代码的改动，然后帮你自动重启应用。在调试时可以完全用 nodemon 命令代替 node 命令。

\$ nodemon app.js 启动我们的应用试试，然后随便改两行代码，就可以看到 nodemon 帮我们重启应用了。

那么 app 写完了，接着开始测试，测试代码在 test/app.test.js。

```

var app = require('../app');
var supertest = require('supertest');
// 看下面这句，这是关键一句。得到的 request 对象可以直接按照
// superagent 的 API 进行调用
var request = supertest(app);

var should = require('should');

describe('test/app.test.js', function () {
  // 我们的第一个测试用例，好好理解一下
  it('should return 55 when n is 10', function (done) {
    // 之所以这个测试的 function 要接受一个 done 函数，是因为我们的测试内容
    // 涉及了异步调用，而 mocha 是无法感知异步调用完成的。所以我们主动接受它提供
    // 的 done 函数，在测试完毕时，自行调用一下，以示结束。
    // mocha 可以感知到我们的测试函数是否接受 done 参数。js 中，function
    // 对象是有长度的，它的长度由它的参数数量决定
    // (function (a, b, c, d) {} ).length === 4
    // 所以 mocha 通过我们测试函数的长度就可以确定我们是否是异步测试。

    request.get('/fib')
      // .query 方法用来传 querystring，.send 方法用来传 body。
      // 它们都可以传 Object 对象进去。
      // 在这里，我们等于访问的是 /fib?n=10

```

```

    .query({n: 10})
    .end(function (err, res) {
        // 由于 http 返回的是 String, 所以我要传入 '55'。
        res.text.should.equal('55');

        // done(err) 这种用法写起来很鸡肋, 是因为偷懒不想测 err 的值
        // 如果勤快点, 这里应该写成
        /*
        should.not.exist(err);
        res.text.should.equal('55');
        */
        done(err);
    });
});

```

```

// 下面我们对于各种边界条件都进行测试, 由于它们的代码雷同,
// 所以我抽象出来了一个 testFib 方法。
var testFib = function (n, statusCode, expect, done) {
    request.get('/fib')
        .query({n: n})
        .expect(statusCode)
        .end(function (err, res) {
            res.text.should.equal(expect);
            done(err);
        });
};

it('should return 0 when n === 0', function (done) {
    testFib(0, 200, '0', done);
});

it('should equal 1 when n === 1', function (done) {
    testFib(1, 200, '1', done);
});

it('should equal 55 when n === 10', function (done) {
    testFib(10, 200, '55', done);
});

it('should throw when n > 10', function (done) {
    testFib(11, 500, 'n should <= 10', done);
});

it('should throw when n < 0', function (done) {
    testFib(-1, 500, 'n should >= 0', done);
});

it('should throw when n isnt Number', function (done) {
    testFib('good', 500, 'n should be a Number', done);
});

// 单独测试一下返回码 500
it('should status 500 when error', function (done) {
    request.get('/fib')
        .query({n: 100})
        .expect(500)
        .end(function (err, res) {
            done(err);
        });
});
});

```

完。

关于 cookie 持久化

有两种思路

1. 在 supertest 中, 可以通过 `var agent = supertest.agent(app)` 获取一个 agent 对象, 这个对象的 API 跟直接在 superagent 上调用各种方法是一样的。agent 对象在被多次调用 `get` 和 `post` 之后, 可以一路把 cookie 都保存下来。

```
var supertest = require('supertest');
var app = express();
var agent = supertest.agent(app);

agent.post('login').end(...);
// then ..
agent.post('create_topic').end(...); // 此时的 agent 中有用户登陆后的 cookie
```

2. 在发起请求时, 调用 `.set('Cookie', 'a cookie string')` 这样的方式。

```
var supertest = require('supertest');
var userCookie;
supertest.post('login').end(function (err, res) {
  userCookie = res.headers['set-cookie']
});
// then ..

supertest.post('create_topic')
  .set('cookie', userCookie)
  .end(...)
```

这里有个相关讨论: <https://github.com/tj/supertest/issues/46>

拓展学习

Nodeclub 里面的测试使用的技术跟前面介绍的是一样的, should mocha supertest 那套, 应该是很容易看懂的:

<https://github.com/cnodejs/nodeclub/blob/master/test/controllers/topic.test.js>

《正则表达式》

目标

```
var web_development = "python php ruby javascript jsonp perhapsphpisoutdated";
```

找出其中包含 p 但不包含 ph 的所有单词, 即

```
[ 'python', 'javascript', 'jsonp' ]
```

知识点

1. 正则表达式的使用
2. js 中的正则表达式与 pcre(http://en.wikipedia.org/wiki/Perl-Compatible_Regular_Expressions) 的区别

课程内容

开始这门课之前, 大家先去看两篇文章。

《正则表达式 30 分钟入门教程》: <http://www.cnblogs.com/deerchao/archive/2006/08/24/zhengzhe30fengzhongjiaocheng.html>

上面这篇介绍了正则表达式的基础知识, 但是对于零宽断言没有展开来讲, 零宽断言看下面这篇:

《正则表达式之：零宽断言不『消费』》：<http://fxck.it/post/50558232873>

好了。

在很久很久以前，有一门语言一度是字符串处理领域的王者，叫 perl。

伴随着 perl，有一个类似正则表达式的标准被实现了出来，叫 pcre: Perl Compatible Regular Expressions。

不遗憾的是，js 里面的正则与 pcre 不是兼容的。很多语言都这样。

如果需要测试你自己写的正则表达式，建议上这里：<http://refiddle.com/>，可以所见即所得地调试。

接下来我们主要讲讲 js 中需要注意的地方，至于正则表达式的内容，上面那两篇文章足够学习了。

第一，

js 中，对于四种零宽断言，只支持零宽度正预测先行断言和零宽度负预测先行断言这两种。

第二，

js 中，正则表达式后面可以跟三个 flag，比如 `/something/igm`。

他们的意义分别是，

- i 的意义是不区分大小写
- g 的意义是，匹配多个
- m 的意义是，是 `^` 和 `$` 可以匹配每一行的开头。

分别举个例子：

```
/a/.test('A') // => false
/a/i.test('A') // => true
```

```
'hello hell hoo'.match(/h.*?\b/) // => [ 'hello', index: 0, input: 'hello hell hoo' ]
'hello hell hoo'.match(/h.*?\b/g) // => [ 'hello', 'hell', 'hoo' ]
```

```
'aaa\nbbb\nccc'.match(/^[\s\S]*$/g) // => [ 'aaa\nbbb\nccc' ]
'aaa\nbbb\nccc'.match(/^[\s\S]*$/gm) // => [ 'aaa', 'bbb', 'ccc' ]
```

与 m 意义相关的，还有 `\A`, `\Z` 和 `\z`

他们的意义分别是：

- `\A` 字符串开头(类似`^`，但不受处理多行选项的影响)
- `\Z` 字符串结尾或行尾(不受处理多行选项的影响)
- `\z` 字符串结尾(类似`$`，但不受处理多行选项的影响)

在 js 中，g flag 会影响 `String.prototype.match()` 和 `RegExp.prototype.exec()` 的行为

`String.prototype.match()` 中，返回数据的格式会不一样，加 g 会返回数组，不加 g 则返回比较详细的信息

```
> 'hello hell'.match(/h(?:.*?)\b/g)
[ 'hello', 'hell' ]
```

```
> 'hello hell'.match(/h(?:.*?)\b/)
[ 'hello',
  'ello',
  index: 0,
  input: 'hello hell' ]
```

`RegExp.prototype.exec()` 中，加 g 之后，如果你的正则不是字面量的正则，而是存储在变量中的话，特么的这个变量就会变得有记忆!!

```
> /h(?:.*?)\b/g.exec('hello hell')
[ 'hello',
  'ello',
  index: 0,
  input: 'hello hell' ]
> /h(?:.*?)\b/g.exec('hello hell')
[ 'hello',
  'ello',
  index: 0,
```

```
input: 'hello hell' ]
```

```
> var re = /h(?:.*?)\b/g;
undefined
> re.exec('hello hell')
[ 'hello',
  'ello',
  index: 0,
  input: 'hello hell' ]
> re.exec('hello hell')
[ 'hell',
  'ell',
  index: 6,
  input: 'hello hell' ]
>
```

第三,

大家知道, . 是不可以匹配 \n 的。如果我们想匹配的数据涉及到了跨行, 比如下面这样的。

```
var multiline = require('multiline');

var text = multiline.stripIndent(function () {
  /*
    head
    ````
 code code2 .code3````
    ````
    foot
  */
});
```

如果我们想把两个 “`” 中包含的内容取出来, 应该怎么办?

直接用 . 匹配不到 \n, 所以我们需要找到一个原子, 能匹配包括 \n 在内的所有字符。

这个原子的惯用写法就是 [\s\S]

```
var match1 = text.match(/^````[\s\S]+?^````/gm);
console.log(match1) // => [ '````\ncode code2 code3````\n````' ]

// 这里有一种很骚的写法, [^] 与 [\s\S] 等价
var match2 = text.match(/^````[^]+?^````/gm)
console.log(match2) // => [ '````\ncode code2 .code3````\n````' ]
```

完。

《benchmark 怎么写》

目标

有一个字符串 `var number = '100'`, 我们要将它转换成 `Number` 类型的 100。

目前有三个选项: +, `parseInt`, `Number`

请测试哪个方法更快。

知识点

1. 学习使用 `benchmark` 库
2. 学习使用 <http://jsperf.com/> 分享你的 benchmark

课程内容

首先去弄个 benchmark 库，<https://github.com/bestiejs/benchmark.js>。

这个库已经两年没有更新了，两年前发了个 1.0.0 版本，直到现在。

这个库的最新版本是 2.1.0

用法也特别简单，照着官网的 copy 下来就好。

我们先来实现这三个函数：

```
var int1 = function (str) {  
  return +str;  
};
```

```
var int2 = function (str) {  
  return parseInt(str, 10);  
};
```

```
var int3 = function (str) {  
  return Number(str);  
};
```

然后照着官方的模板写 benchmark suite：

```
var number = '100';  
  
// 添加测试  
suite  
  .add('+', function() {  
    int1(number);  
  })  
  .add('parseInt', function() {  
    int2(number);  
  })  
  .add('Number', function () {  
    int3(number);  
  })  
  .on('cycle', function(event) {  
    console.log(String(event.target));  
  })  
  .on('complete', function() {  
    console.log('Fastest is ' + this.filter('fastest').map('name'));  
  })  
  // 这里的 async 不是 mocha 测试那个 async 的意思，这个选项与它的时间计算有关，默认勾上就好了。  
  .run({ 'async': true });
```

直接运行：

```
alsotang@aliair: ~/Dropbox/code/node-lessons/lesson10 (master)$ node main.js  
+ x 16,773,424 ops/sec ±1.81% (89 runs sampled)  
parseInt x 31,970,765 ops/sec ±1.60% (80 runs sampled)  
Number x 19,689,845 ops/sec ±1.47% (90 runs sampled)  
Fastest is parseInt
```

可以看到，parseInt 是最快的。

在线分享

如果想要在线分享你的 js benchmark，用这个网站：<http://jsperf.com/>。

比如我在上面测试 `Math.log` 的效率：

<http://jsperf.com/math-perf-alsotang>

进入之后点击那个 **Run tests** 按钮，就可以在浏览器中看到它们的效率差异了，毕竟浏览器也是可以跑 js 的。

点击这里：<http://jsperf.com/math-perf-alsotang/edit>，就可以看到这个 benchmark 是怎么配置的，很简单。

《作用域与闭包：this, var, (function () {})》

目标

无具体目标

知识点

1. 理解 js 中 var 的作用域
2. 了解闭包的概念
3. 理解 this 的指向

课程内容

es6 中新增了 let 关键词，与块级作用域，相关知识参考：<http://es6.ruanyifeng.com/#docs/let>

var 作用域

先来看个简单的例子：

```
var parent = function () {
  var name = "parent_name";
  var age = 13;

  var child = function () {
    var name = "child_name";
    var childAge = 0.3;

    // => child_name 13 0.3
    console.log(name, age, childAge);
  };

  child();

  // will throw Error
  // ReferenceError: childAge is not defined
  console.log(name, age, childAge);
};

parent();
```

直觉地，内部函数可以访问外部函数的变量，外部不能访问内部函数的变量。上面的例子中内部函数 child 可以访问变量 age，而外部函数 parent 不可以访问 child 中的变量 childAge，因此会抛出没有定义变量的异常。

有个重要的事，如果忘记 var，那么变量就被声明为全局变量了。

```
function foo() {
  value = "hello";
}
```

```
foo();
console.log(value); // 输出 hello
console.log(global.value) // 输出 hello
```

这个例子可以很正常的输出 `hello`，是因为 `value` 变量在定义时，没有使用 `var` 关键词，所以被定义成了全局变量。在 Node 中，全局变量会被定义在 `global` 对象下；在浏览器中，全局变量会被定义在 `window` 对象下。

如果你确实要定义一个全局变量的话，请显示地定义在 `global` 或者 `window` 对象上。

这类不小心定义全局变量的问题可以被 `jshint` 检测出来，如果你使用 `sublime` 编辑器的话，记得装一个 `SublimeLinter` 插件，这是插件支持多语言的语法错误检测，js 的检测是原生支持的。

JavaScript 中，变量的局部作用域是函数级别的。不同于 C 语言，在 C 语言中，作用域是块级别的。JavaScript 中没有块级作用域。

js 中，函数中声明的变量在整个函数中都有定义。比如如下代码段，变量 `i` 和 `value` 虽然是在 `for` 循环代码块中被定义，但在代码块外仍可以访问 `i` 和 `value`。

```
function foo() {
  for (var i = 0; i < 10; i++) {
    var value = "hello world";
  }
  console.log(i); //输出 10
  console.log(value); //输出 hello world
}
foo();
```

所以有种说法是：应该提前声明函数中需要用到的变量，即，在函数体的顶部声明可能用到的变量，这样就可以避免出现一些奇奇怪怪的 bug。

但我个人不喜欢遵守这一点，一般都是现用现声明的。这类错误的检测交给 `jshint` 来做就好了。

闭包

闭包这个概念，在函数式编程里很常见，简单的说，就是使内部函数可以访问定义在外部函数中的变量。

假如我们要实现一系列的函数：`add10`，`add20`，它们的定义是 `int add10(int n)`。

为此我们构造了一个名为 `adder` 的构造器，如下：

```
var adder = function (x) {
  var base = x;
  return function (n) {
    return n + base;
  };
};

var add10 = adder(10);
console.log(add10(5));

var add20 = adder(20);
console.log(add20(5));
```

每次调用 `adder` 时，`adder` 都会返回一个函数给我们。我们传给 `adder` 的值，会保存在一个名为 `base` 的变量中。由于返回的函数在其中引用了 `base` 的值，于是 `base` 的引用计数被 +1。当返回函数不被垃圾回收时，则 `base` 也会一直存在。

我暂时想不出什么实用的例子来，如果想深入理解这块，可以看看这篇 <http://coolshell.cn/articles/6731.html>

闭包的一个坑

```
for (var i = 0; i < 5; i++) {
  setTimeout(function () {
    console.log(i);
  }, 5);
}
```

上面这个代码块会打印五个 5 出来，而我们预想的结果是打印 0 1 2 3 4。

之所以会这样，是因为 `setTimeout` 中的 `i` 是对外层 `i` 的引用。当 `setTimeout` 的代码被解释的时候，运行时只是记录了 `i` 的引用，而不是值。而当 `setTimeout` 被触发时，五个 `setTimeout` 中的 `i` 同时被取值，由于它们都指向了外层的同一个 `i`，而那个 `i` 的值在迭代完成时为 5，所以打印了五次 5。

为了得到我们预想的结果，我们可以把 `i` 赋值成一个局部的变量，从而摆脱外层迭代的影响。

```
for (var i = 0; i < 5; i++) {
  (function (idx) {
    setTimeout(function () {
      console.log(idx);
    }, 5);
  })(i);
}
```

this

在函数执行时，`this` 总是指向调用该函数的对象。要判断 `this` 的指向，其实就是判断 `this` 所在的函数属于谁。

在《JavaScript 语言精粹》这本书中，把 `this` 出现的场景分为四类，简单的说就是：

- 有对象就指向调用对象
- 没调用对象就指向全局对象
- 用 `new` 构造就指向新对象
- 通过 `apply` 或 `call` 或 `bind` 来改变 `this` 的所指。

1) 函数有所属对象时：指向所属对象

函数有所属对象时，通常通过 `.` 表达式调用，这时 `this` 自然指向所属对象。比如下面的例子：

```
var myObject = {value: 100};
myObject.getValue = function () {
  console.log(this.value); // 输出 100

  // 输出 { value: 100, getValue: [Function] },
  // 其实就是 myObject 对象本身
  console.log(this);

  return this.value;
};
```

```
console.log(myObject.getValue()); // => 100
```

`getValue()` 属于对象 `myObject`，并由 `myObject` 进行 `.` 调用，因此 `this` 指向对象 `myObject`。

2) 函数没有所属对象：指向全局对象

```
var myObject = {value: 100};
myObject.getValue = function () {
  var foo = function () {
    console.log(this.value) // => undefined
    console.log(this); // 输出全局对象 global
  };

  foo();

  return this.value;
};
```

```
console.log(myObject.getValue()); // => 100
```

在上述代码块中，`foo` 函数虽然定义在 `getValue` 的函数体内，但实际上它既不属于 `getValue` 也不属于 `myObject`。`foo` 并没有被绑定在任何对象上，所以当调用时，它的 `this` 指针指向了全局对象 `global`。

据说这是个设计错误。

3) 构造器中的 this: 指向新对象

js 中, 我们通过 new 关键词来调用构造函数, 此时 this 会绑定在该新对象上。

```
var SomeClass = function(){
  this.value = 100;
}

var myCreate = new SomeClass();

console.log(myCreate.value); // 输出 100
```

顺便说一句, 在 js 中, 构造函数、普通函数、对象方法、闭包, 这四者没有明确界线。界线都在人的心中。

4) apply 和 call 调用以及 bind 绑定: 指向绑定的对象

apply() 方法接受两个参数第一个是函数运行的作用域, 另外一个是一个参数数组 (arguments)。

call() 方法第一个参数的意义与 apply() 方法相同, 只是其他的参数需要一个个列举出来。

简单来说, call 的方式更接近我们平时调用函数, 而 apply 需要我们传递 Array 形式的数组给它。它们是可以互相转换的。

```
var myObject = {value: 100};

var foo = function(){
  console.log(this);
};

foo(); // 全局变量 global
foo.apply(myObject); // { value: 100 }
foo.call(myObject); // { value: 100 }

var newFoo = foo.bind(myObject);
newFoo(); // { value: 100 }
```

完。

《线上部署: heroku》

目标

将 <https://github.com/Ricardo-Li/node-practice-2> (这个项目已经被删了。参照 <https://github.com/alsotang/node-lessons/tree/master/lesson3> 的代码自己操作一下吧。) 这个项目部署上 heroku, 成为一个线上项目

我部署的在这里 <http://serene-falls-9294.herokuapp.com/>

知识点

1. 学习 heroku 的线上部署 (<https://www.heroku.com/>)

课程内容

什么是 heroku

heroku 是弄 ruby 的 paas 起家, 现在支持多种语言环境, 更甚的是它强大的 add-on 服务。

paas 平台相信大家都不陌生。Google 有 gae, 国内新浪有 sae。paas 平台相对 vps 来说, 不需要你配置服务器, 不需要装数据库, 也不需要理会负载均衡。这一切都可以在平台上直接获取。

你只要专注自己的业务，把应用的逻辑写好，然后发布上去，应用自然就上线了。数据库方面，如果你用 mysql，那么你可以从平台商那里得到一个 mysql 的地址、账号和密码，直接连接就能用。如果应用的流量增大，需要横向拓展，则只用去到 paas 平台的管理页面，增大服务器实例的数量即可，负载均衡会自动帮你完成。

说起来，我之所以对于 web 开发产生兴趣也是因为当年 gae 的关系。那时候除了 gae 之外，没有别的 paas 平台，gae 是横空出世的。有款翻墙的软件，叫 gappproxy(<https://code.google.com/p/gappproxy/>)——可以认为是 goagent 的前身——就是搭建在 gae 上面的，不仅快，而且免费。于是我就很想弄懂这样一个程序是如何开发的。好在 gappproxy 是开源的，于是我下了源码来看，那时候才大一，只学过 c，看到那些 python 代码就凌乱。于是转头也去学 python，后来渐渐发现了 web 开发的乐趣，于是 ruby 和 node.js 也碰碰。后来 goagent 火起来了，我又去看了看它的代码，发现非常难看，就自己写了个 <https://github.com/alsotang/keepagent>。不过现在回想起来，还是 goagent 的实现比较稳定以及效率高。

heroku 的免费额度还是足够的，对于 demo 应用来说，放上去是绰绰有余的。各位搞 web 开发的大学生朋友，一定要试着让你开发的项目尽可能早地去线上跑，这样你的项目可以被其他人看到，能够促使你更有热情地进行进一步开发。这回我们放的是 cnode 社区的爬虫上去，你其实可以试着为你们学院或者学校的新闻站点写个爬虫，提供 json api，然后去申请个微信公众平台，每天推送学院网站的新闻。这东西辅导员是有需求的，可以做个给他们用。

好了，我们先 clone <https://github.com/Ricardo-Li/node-practice-2> 这个项目。由于我们这回讲部署，所以代码就用现成的了，代码的内容就是 lesson 3 (<https://github.com/alsotang/node-lessons/tree/master/lesson3>) 里面的那个爬虫。

```
alsotang@aliar: ~/code$ git clone git@github.com:Ricardo-Li/node-practice-2.git
Cloning into 'node-practice-2'...
remote: Counting objects: 12, done.
Receiving objects: 100% (12/12), done.
Resolving deltas: 100% (3/3), done.
remote: Total 12 (delta 0), reused 0 (delta 0)
Checking connectivity... done.
```

clone 下来以后，我们去看看代码。代码中有两个特殊的地方，

一个是一个叫 Procfile 的文件，内容是：

```
web: node app.js
```

一个是 app.js 里面，

```
app.listen(process.env.PORT || 5000);
```

这两者都是为了部署 heroku 所做的。

大家有没有想过，当部署一个应用上 paas 平台以后，paas 要为我们干些什么？

首先，平台要有我们语言的运行时；

然后，对于 node.js 来说，它要帮我们安装 package.json 里面的依赖；

然后呢？然后需要启动我们的项目；

然后把外界的流量导入我们的项目，让我们的项目提供服务。

上面那两处特殊的地方，一个是启动项目的，一个是导流量的。

heroku 虽然能推测出你的应用是 node.js 应用，但它不懂你的主程序是哪个，所以我们提供了 Procfile 来指导它启动我们的程序。

而我们的程序，本来是监听 5000 端口的，但是 heroku 并不知道。当然，你也可以在 Procfile 中告诉 heroku，可如果大家都监听 5000 端口，这时候不就有冲突了吗？所以这个地方，heroku 使用了主动的策略，主动提供一个环境变量 `process.env.PORT` 来供我们监听。

这样的话，一个简单 app 的配置就完成了。

我们去 <https://www.heroku.com/> 申请个账号，然后下载它的工具包 <https://toolbelt.heroku.com/>，然后再在命令行里面，通过 `heroku login` 来登录。

上述步骤完成后，我们进入 `node-practice-2` 的目录，执行 `heroku create`。这时候，heroku 会为我们随机取一个应用名字，并提供一个 git 仓库给我们。


```
alsotang@aliair: ~/code/node-practice-2 (master)$ heroku create
Creating serene-falls-9294... done, stack is cedar
http://serene-falls-9294.herokuapp.com/ | git@heroku.com:serene-falls-9294.git
Git remote heroku added
alsotang@aliair: ~/code/node-practice-2 (master)$ git remote -v
heroku  git@heroku.com:serene-falls-9294.git (fetch)
heroku  git@heroku.com:serene-falls-9294.git (push)
origin  git@github.com:Ricardo-Li/node-practice-2.git (fetch)
origin  git@github.com:Ricardo-Li/node-practice-2.git (push)
```

接着，往 heroku 这个远端地址推送我们的 master 分支：

```
alsotang@aliair: ~/code/node-practice-2 (master)$ git push heroku master
Initializing repository, done.
Counting objects: 12, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (12/12), 1.31 KiB | 0 bytes/s, done.
Total 12 (delta 3), reused 12 (delta 3)

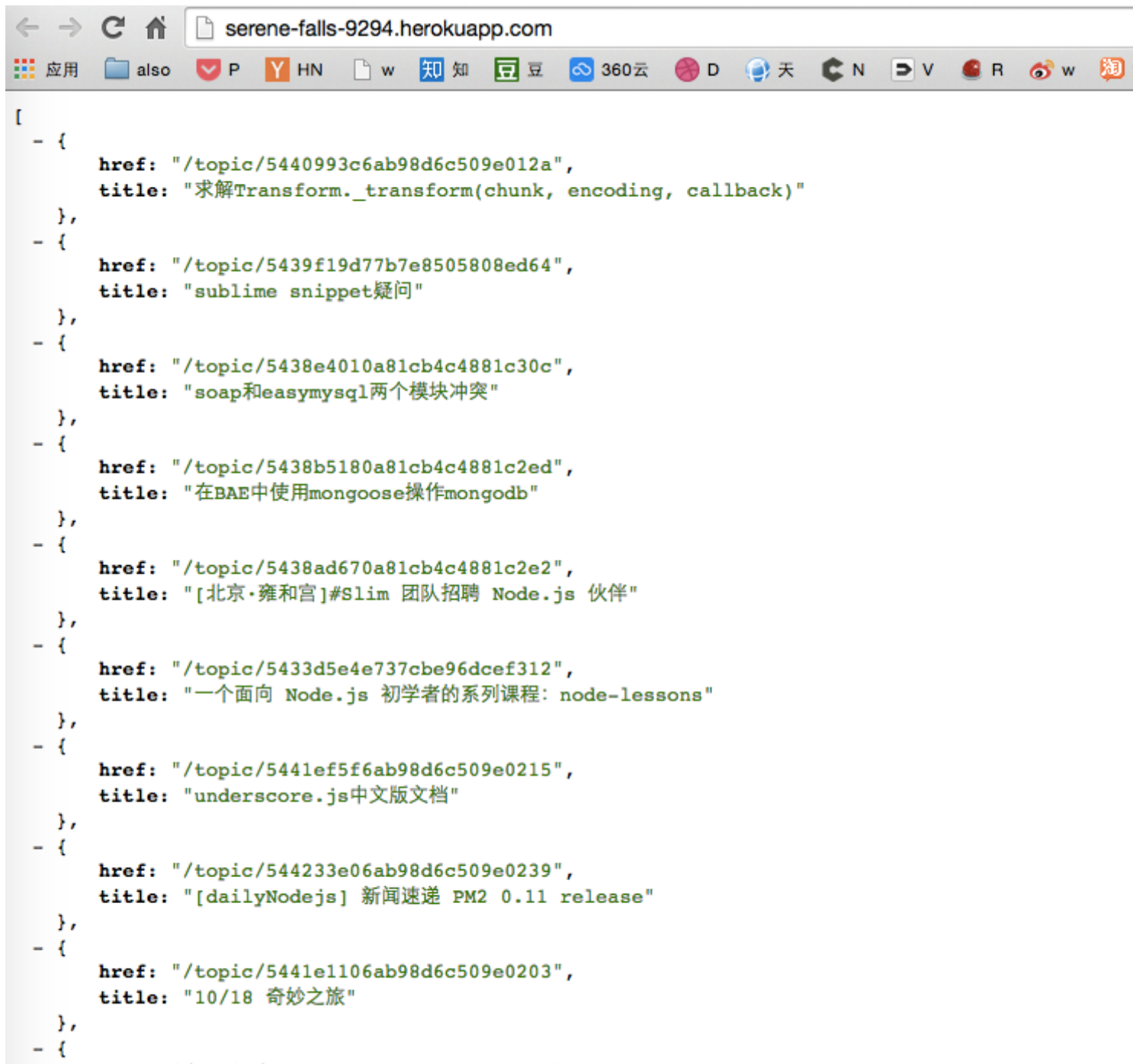
-----> Node.js app detected

      PRO TIP: Specify a node version in package.json
      See https://devcenter.heroku.com/articles/nodejs-support

-----> Defaulting to latest stable node: 0.10.32
-----> Downloading and installing node
-----> Exporting config vars to environment
-----> Installing dependencies
npm WARN package.json node-practice-2@0.0.1 No description
npm WARN package.json node-practice-2@0.0.1 No repository field.
```

heroku 会自动检测出我们是 node.js 程序，并安装依赖，然后按照 Procfile 进行启动。

push 完成后，在命令行输入 `heroku open`，则 heroku 会自动打开浏览器带我们去到相应的网址：



到此课程也就结束了。

随便聊聊 heroku 的 addon 吧。这个 addon 确实是个神奇的东西，反正在 heroku 之外我还没怎么见到这类概念。这些 addon 提供商，有些提供 redis 的服务，有些提供 mongodb，有些提供 mysql。你可以直接在 heroku 上面进行购买，然后 heroku 就会提供一段相应服务的地址和账号密码给你用来连接。

大家可以去 <https://addons.heroku.com/> 这个页面看看，琳琅满目各种应用可以方便接入。之所以这类服务有市场，也是因为亚马逊的 aws 非常牛逼。为什么这么说呢，因为网络速度啊。如果现在在国内，你在 ucloud 买个主机，然后用个阿里云的 rds，那么应用的响应速度会因为 mysql 连接的问题卡得动不了。但在 heroku 这里，提供商们，包括 heroku 自己，都是构建在 aws 上面，这样一来，各种服务的互通其实走的是内网，速度很可以接受，于是各种 addon 提供商就做起来了。

国内的话，其实在阿里云上面也可以考虑这么搞一搞。

完。

《持续集成平台：travis》

目标

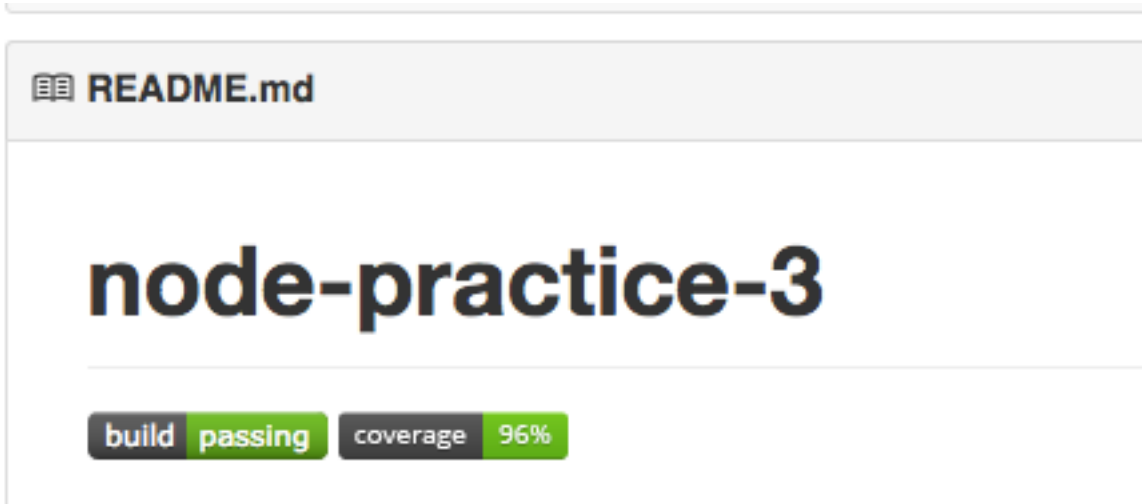
无明确目标

知识点

- 1. 学习使用 travis-ci 对项目进行持续集成测试 (<https://travis-ci.org/>)

课程内容

首先来看看这个项目：<https://github.com/Ricardo-Li/node-practice-3>



(图 1)

类似这样的 badges，在很多项目中都可以看到。前者是告诉我们，这个项目的测试目前是通过的；后者是告诉我们，这个测试的行覆盖率是多少。行覆盖率当然是越多越好。测试的重要性我就不说了。

为什么要使用 travis 这样的平台，是因为它可以让你明白自己的项目在一个“空白环境”中，是否能正确运行；也可以让你知道，用不同的 Node.js 版本运行的话，有没有兼容性问题。

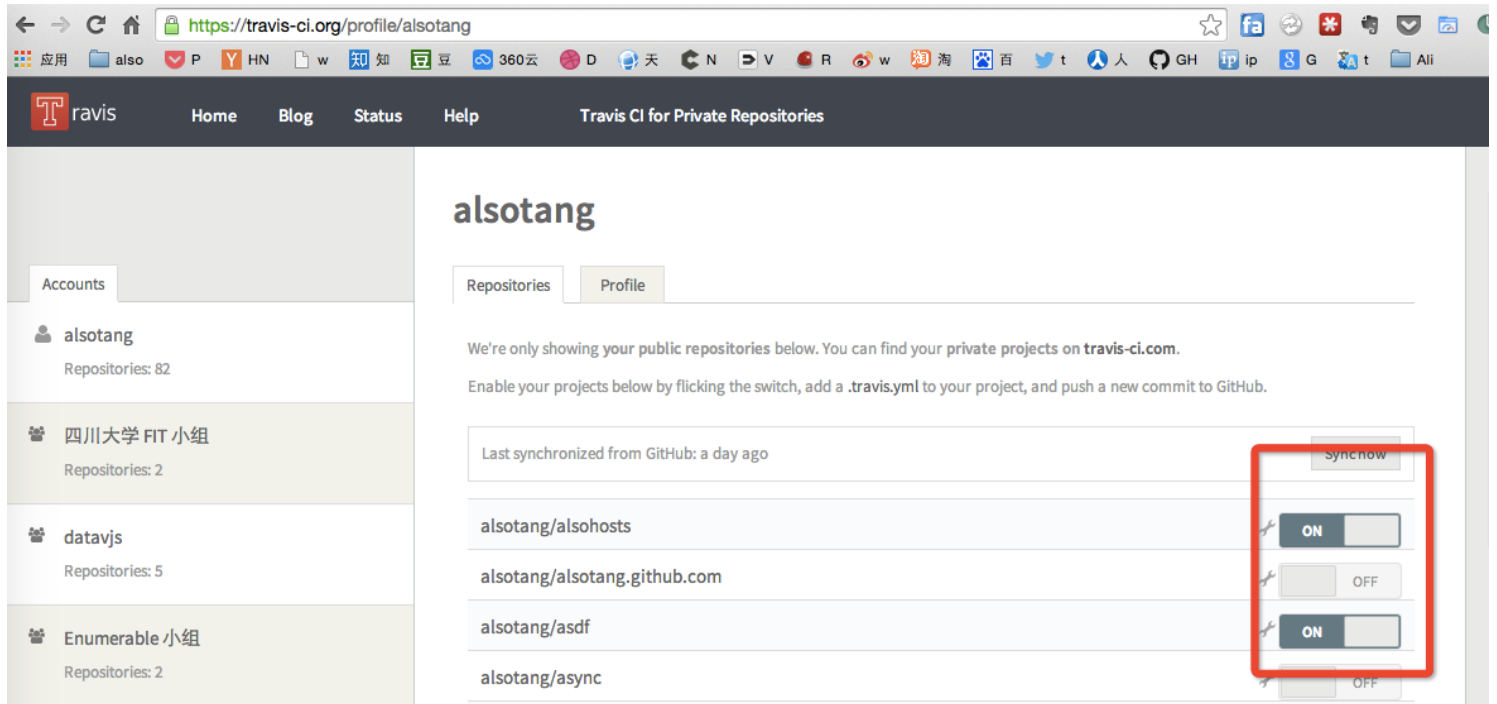
当你在自己的机器上跑测试的时候，你用着特定的 Node.js 版本，比如 0.10，如果测试过了，你也还是不懂在 0.11 下，你的测试能不能通过。你可以手动地切换 node 版本再跑一次，也可以选择让 travis 帮你把不同的 node 版本跑一次。而且有时候，我们 npm 安装了某个包，但却忘记将它写入 package.json 里面了，在自己的机器上，测试没问题，但当别的用户安装你的包时，会发现依赖缺失。

travis 应该是把虚拟机的技术玩得比较好，它每次跑测试时，都会提供一个空白的环境。这个环境只有 Linux 基本的 **build-essential** 和 **wget**、**git** 那些依赖。连 Node.js 的运行都是现跑现安装的。

travis 默认带有的那些依赖，都是每个用户的机器上都会有的，所以一旦你的应用在 travis 上面能够跑通，那么就不用担心别的用户安装不上了。

我们来讲接入 travis 的步骤。

travis 的价格是免费的，对于 github 上的开源项目来说。它默认当然不可能帮 github 的每个用户都跑测试，所以你需要去注册一下 travis，然后告诉它你需要开启集成测试的仓库。



比如上图，可以看到我帮自己的 `alsohosts` 项目以及 `asdf` 项目开启了测试。

当你在 travis 授权了仓库之后，每当你 push 代码到 github，travis 都会自动帮你跑测试。

travis 通过授权，可以知道你的项目在什么地方，于是它就可以把项目 clone 过去。但问题又来了，它不懂你的测试怎么跑啊。用 `npm test` 还是 `make test` 还是 `jake test` 呢？

所以我们需要给出一些配置信息，配置信息以 `.travis.yml` 文件的形式放在项目根目录，比如一个简单的 `.travis.yml`。

```
language: node_js
node_js:
  - '0.8'
  - '0.10'
  - '0.11'
```

```
script: make test
```

这个文件传递的信息是：

- 这是一个 node.js 应用
- 这个测试需要用 0.8、0.10 以及 0.11 三个版本来跑
- 跑测试的命令是 `make test`

将这个文件添加到项目的根目录下，再 push 上 github，这时候 travis 就会被触发了。

travis 接着会做的事情是：

1. 安装一个 node.js 运行时。由于我们指定了三个不同版本，于是 travis 会使用三个机器，分别安装三个版本的 node.js
2. 这些机器在完成运行时安装后，会进入项目目录执行 `npm install` 来安装依赖。
3. 当依赖安装完成后，执行我们指定的 script，在这里也就是 `make test`

如果测试通过的话，`make` 命令的返回码会是 0（如果不懂什么是返回码，则需要补补 shell 的知识），则测试通过；如果测试有不通过的 case，则返回码不会为 0，travis 则判断测试失败。

每一个 travis 上面的项目，都可以得到一个图片地址，这个地址上的图片会显示你项目当前的测试通过状态，把这个图片添加到自己项目的 README 中，就可以得到我们图 1 的那种逼格了。

对了，行覆盖率的那个 badge 是由一个叫 coveralls(<https://coveralls.io/>) 的服务提供的。大家可以试着自己接入。

补充说明：

如果你的应用有使用到数据库，需要在 `.travis.yml` 中添加一些内容。

以 MongoDB 为例：

services:
mongodb

其它数据库详细内容参考travis 官方文档

《js 中的那些最佳实践》

这个章节需要大家帮忙补充，一次性想不完那么多

JavaScript 语言精粹

<http://book.douban.com/subject/3590768/>

豆瓣读书

书名、作者、ISBN

分类浏览

购物车 购书单

电子图书 NEW



作者: Douglas Crockford

出版社: 电子工业出版社

译者: 赵泽欣 / 鄢学鹏

出版年: 2009年

页数: 155

定价: 35.00

装帧: 平装

丛书: 博文视点O'reilly系列

ISBN: 9787121084379

★★★★★ 9.1

(1021人评价)

★★★★★ 61.4%

★★★★☆ 32.2%

★★★☆☆ 6.0%

★★☆☆☆ 0.4%

★☆☆☆☆ 0.0%

想读

在读

读过

评价: ☆☆☆☆☆

写笔记

写书评

加入购书单

分享到

推荐

这本书很薄，只有 155 页，但该讲的几乎都讲了。大家想办法搞来看看吧（我总不能很没节操地给个电子版 PDF 链接在这里吧）。

js 这门语言，水很浅。没有太复杂的地方可以钻，但特么的坑又多。

上面的那本书是一定要看的。这本书专注在讲 js 语法，其他 js 的书都过多地涉及了浏览器知识。

- 其一: <http://fxck.it/post/72326363595>
- 其二: <http://fxck.it/post/73513189448>

继承

js 前端不懂有什么好办法, 后端的话, 很方便。

用 node 官方的 `util` 库, 下面是直接从官网摘抄来的:

```
var util = require("util");
var events = require("events");

function MyStream() {
  events.EventEmitter.call(this);
}

util.inherits(MyStream, events.EventEmitter);

MyStream.prototype.write = function(data) {
  this.emit("data", data);
}

var stream = new MyStream();

console.log(stream instanceof events.EventEmitter); // true
console.log(MyStream.super_ === events.EventEmitter); // true

stream.on("data", function(data) {
  console.log('Received data: ' + data + '');
})
stream.write("It works!"); // Received data: "It works!"
```

js 是面向对象的, 但是是“基于原型的面向对象”, 没有类。没有多重继承, 没有接口。没有结构体, 没有枚举类型。

但它的字面量哈希和 `function` 都足够灵活, 拼拼凑凑, 上面那些东西都能“模拟”着用。

说到没有类的这个问题, 很多人总是要纠正其他人关于 js 原型的理解的。我觉得这是没有必要的。基于原型又不是很牛逼, ES6 不是照样给出了 `class` 关键字吗。不管类还是原型都是为了抽象, 烂的东西始终烂, 不好理解的始终不好理解。

最近学习 ios 的 `swift`, 看见里面有不少相比 `objc` 舒服的改进。比如 `objc` 的“方法调用”, 学的是 `smalltalk` 那一套, 那不叫方法调用, 而是消息传递。结果 `swift` 里面不照样是方法调用的形式?

callback hell

用 `eventproxy` 和 `async` 已经能解决大部分问题。剩下的小部分问题, 肯定是设计错误。:)

参见:

- 《使用 `eventproxy` 控制并发》: <https://github.com/alsotang/node-lessons/tree/master/lesson4>
- 《使用 `async` 控制并发》: <https://github.com/alsotang/node-lessons/tree/master/lesson5>

数据类型

写 js 很少去定义类。Object 的便利在多数其他语言需要定义类的场景下都能直接用。

js 中, 用好 `Number`, `String`, `Array`, `Object` 和 `Function` 就够了。有时用用 `RegExp`。

用于 js 这门语言本身的残废, 大多数时候都采用“约定胜于配置”的思想来交互合作。

控制流

很常规，C 语言那套。

基本运算符

C 语言那套。二进制操作并不会降低效率，V8 很聪明的。

计算型属性

也就是帮一个对象的属性定义 get 和 set 方法，通过 `obj.value` 和 `obj.value=` 的形式来调用。
koa(<http://koajs.com/>) 把这套玩得炉火纯青。

运算符重载

无

类型转换

手动帮你需要转换的类型的类定义 `.toxxx` 方法，比如 `.toString`, `.toJSON`, `toNumber`。
js 的隐式类型转换用一次坑一次。

相等比较

在 js 中，务必使用 `===` 三个等于号来比较对象，或者自定义方法来比较，不要使用 `==`。

我最近做一个项目，从数据库中取出的数据，虽然应该是字符型的，但有时它们是 String 的表示，有时是 Number 的表示。为了省事，会有人直接用 `==` 来对它们进行比较。这种时候，建议在比较时，把它们都转成 String 类型，然后用 `===` 来比较。

比如 `var x = 31243; var y = '31243'`，比较时，这么做：`String(x) === String(y)`

嵌套类型

随便弄。

function 构造函数、闭包、字面量哈希，都可以混在一起写，多少层都行，无限制。

拓展

当无法接触一个类的源码，又想帮这个类新增方法的时候。操作它的 prototype 就好了。但不推荐！

函数式编程

js 中，匿名函数非常的方便，有效利用函数式编程的特性可以使人写代码时心情愉悦。

使用 lodash: <https://lodash.com/docs>

泛型

类型都经常忽略还泛型！every parameter is 泛型 in js

权限控制

类定义中，没有 public private 等关键词，都靠约定。而且经常有人突破约定。

有些 http 方面的库，时不时就去 stub 原生 http 库的方法，0.11 时的 node.js 完全不按章法出牌，所以很多这些库都出现兼容性问题。

设计模式

《解密设计模式-王垠》

<https://github.com/alsotang/node-lessons/blob/master/lesson14/%E8%A7%A3%E5%AF%86%E8%AE%BE%E8%AE%A1%E7%8E%8B%E5%9E%A0.md>

构建大型项目

从 npm 上面寻找质量高的库，并用质量高的方式拼凑起来。

《Mongodb 与 Mongoose 的使用》

目标

无明确目标

知识点

1. 了解 mongodb (<http://www.mongodb.org/>)
2. 学习 mongoose 的使用 (<http://mongoosejs.com/>)

课程内容

mongodb

mongodb 这个名词相信大家不会陌生吧。有段时间 nosql 的概念炒得特别火，其中 hbase redis mongodb couchdb 之类的名词都相继进入了大众的视野。

hbase 和 redis 和 mongodb 和 couchdb 虽然都属于 nosql 的大范畴。但它们关注的领域是不一样的。hbase 是存海量数据的，redis 用来做缓存，而 mongodb 和 couchdb 则试图取代一些使用 mysql 的场景。

mongodb 的官网是这样介绍自己的：

MongoDB (from “humongous”) is an open-source document database, and the leading NoSQL database. Written in C++

开源、文档型、nosql。

其中文档型是个重要的概念需要理解。

在 sql 中，我们的数据层级是：数据库 (db) -> 表 (table) -> 记录 (record) -> 字段；在 mongodb 中，数据的层级是：数据库 -> collection -> document -> 字段。这四个概念可以对应得上。

文档型数据这个名字中，“文档”两个字很容易误解。其实这个文档就是 bson 的意思。bson 是 json 的超集，比如 json 中没法储存二进制类型，而 bson 拓展了类型，提供了二进制支持。mongodb 中存储的一条条记录都可以用 bson 来表示。所以你也可以认为，mongodb 是个存 bson 数据的数据库，或是存哈希数据的数据库。

mongodb 相对于它的竞争对手们来说——比如 couchdb，它的一大优势就是尽可能提供与 sql 对应的概念。之前说了，sql 中的记录对应 mongodb 中的 document，记录这东西是一维的，而 document 可以嵌套很多层。在某些场景下，比如存储一个文章的 tags，mongodb 中的字段可以轻松存储数组类型，而 sql 中就需要设计个一对多的表关系出来。

假设有一个 blog 应用，其中有张 Post 表，表中有用户发表的一些博客内容 (post)。

这些 post 文档的样子大概会是这样：

```
var post = {
  title: ' 呵呵的一天',
  author: 'alsotang',
  content: ' 今天网速很差',
  tags: [' 呵呵', ' 网速', ' 差'],
};
```

mongodb 中有个最亮眼的特性，就是 Auto-Sharding，sharding 的意思可以理解成我们 scale sql 时的分表。

在 mongodb 中，表与表之间是没有联系的，不像 sql 中一样，可以设定外键，可以进行表连接。mongodb 中，也无法支持事务。

所以这样的表，无债一身轻。可以很轻易地 scale 至多个实例（假设实例都有不同的物理位置）上。在 mongodb 中，实时的那些查询，也就只能进行条件查询：某某大于一个值或某某等于一个值。而 sql 中，如果一张表的数据存在了多个实例上的话，当与其他表 join 时候，表之间的运来运去会是个很慢的过程，具体我也不太懂。

反正使用 mongodb 时，一定要思考的两点就是：表 join 到底要不要，事务支持到底要不要。

mongodb 中的索引特性跟 sql 中差不多，只是它对于嵌套的数据类型也提供了支持。在建立复合索引时，mongodb 可以指定不同字段的排序，比如两个字段 `is_top` (置顶) 和 `create_time` (创建时间) 要建立复合索引，我们可以指定 `is_top` 按正序排，`create_time` 按逆序排。mysql 说是有计划支持这个特性，不过目前也没什么消息。不过这点不重要。

mongodb 中，collection 是 schema-less 的。在 sql 中，我们需要用建表语句来表明数据应该具有的形式，而 mongodb 中，可以在同一张里存各种各样不同的形式的数据。同一个 collection 中，可以有些 document 具有 100 个字段，而另一些，则只具有 5 个字段。如果你分不清这个特性的使用场景，那么请像使用 sql 一样的，尽可能保证一个 collection 中数据格式是统一的。这个 schema-less 的特性，有个比较典型的场景是用来存储日志类型的数据，可以搜搜看这方面的典型场景。

mongodb 和 mysql 要我选的话，无关紧要的应用我会选择 mongodb，就当个简单的存 json 数据的数据库来用；如果是线上应用，肯定还是会选择 mysql。毕竟 sql 比较成熟，而且各种常用场景的最佳实践都有先例了。

我所在的阿里巴巴数据平台，有各种各样的大数据系统。有些做离线计算，一算就是几个小时，算出来的结果被缓存起来，查询时候就可以实时得到结果，只是数据一致性上，不可避免会有 delay；有些做实时运算，可以在 1s 内从几千万条数据中算出一个复杂条件的结果。但它们都提供了 sql 的接口，也就是说，无论底层他们是如何让几百台机器 mapreduce，都让你可以用已有的 sql 知识进行查询。所以还是选择 sql 省事啊。

这里还有个很好玩的网站：<http://www.mongodb-is-web-scale.com/>

顺便说说 mongodb 与 redis 的不同。mongodb 是用来存非临时数据的，可以认为是存在硬盘上，而 redis 的数据可以认为都在内存中，存储临时数据，丢了也无所谓。对于稍微复杂的查询，redis 支持的查询方式太少太少了，几乎可以认为是 key-value 的。据说 instagram 的数据就全部存在 redis 中，用了好几个几十 G 内存的 aws ec2 机器在存。redis 也是支持把数据写入硬盘的，aof 貌似都过时了，好久没关注了。

mongodb 与 hbase 的区别。如果说你已经在考虑使用 hbase 了的话，应该也不用我介绍它们的区别了吧..

主题所限，就不展开讲了。这之间的选择和权衡，说起来真的是个很大的话题。

我对这方面的话题很感兴趣，如果要讨论这方面话题的话，可以去 <https://cnodejs.org/> 发个帖，详细描述一下场景然后 at 我 (@alsotang)。

mongodb 的官网中有一些特性介绍：

- **Document-Oriented Storage »** ←

JSON-style documents with dynamic schemas offer simplicity and power.

- **Full Index Support »** ←

Index on any attribute, just like you're used to.

- **Replication & High Availability »** ←

Mirror across LANs and WANs for scale and peace of mind.

- **Auto-Sharding »** ←

Scale horizontally without compromising functionality.

- **Querying »** ←

Rich, document-based queries.

- **Fast In-Place Updates »** ←

Atomic modifiers for contention-free performance.

- **Map/Reduce »** ○

Flexible aggregation and data processing.

- **GridFS »** ✗

Store files of any size without complicating your stack.

- **MongoDB Management Service »** ✗

Monitoring and backup designed for MongoDB.

- **Partner with MongoDB »** ✗

Reduce cost, accelerate time to market, and mitigate risk with proactive support and enterprise-grade capabilities.

其中标有箭头的是基本概念，圆圈的是进阶概念，画叉的不必了解。

安装 mongodb

课程到这，一直忘记说 mongodb 的安装了。

ubuntu: <http://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/>

mac: `$ brew install mongodb`

装好以后，在命令行 `$ mongod`，然后另外开个 shell 窗口，输入 `$ mongo` 就能使用了。

mongoose

mongoose 是个 odm。odm 的概念对应 sql 中的 orm。也就是 ruby on rails 中的 activerecord 那一层。orm 全称是 **Object-Relational Mapping**，对象关系映射；而 odm 是 **Object-Document Mapping**，对象文档映射。

它的作用就是，在程序代码中，定义一下数据库中的数据格式，然后取数据时通过它们，可以把数据库中的 document 映射成程序中的一个对象，这个对象有 .save .update 等一系列方法，和 .title .author 等一系列属性。在调用这些方法时，odm 会根据你调用时所用的条件，自动转换成相应的 mongodb shell 语句帮你发送出去。自然地，在程序中链式调用一个个的方法要比手写数据库操作语句具有更大的灵活性和便利性。

mongoose 的官网给出了类似这样一个示例，我改造了一下：

```
// 首先引入 mongoose 这个模块
var mongoose = require('mongoose');
// 然后连接对应的数据库: mongodb://localhost/test
// 其中，前面那个 mongodb 是 protocol scheme 的名称; localhost 是 mongod 所在的地址;
// 端口号省略则默认连接 27017; test 是数据库的名称
// mongodb 中不需要建立数据库，当你需要连接的数据库不存在时，会自动创建一个出来。
// 关于 mongodb 的安全性，mongodb 我印象中安全机制很残废，用户名密码那套都做得不好，更
// 别提细致的用户权限控制了。不过不用担心，mongodb 的默认配置只接受来自本机的请求，内网都连不上。
// 当需要在内网中为其他机器提供 mongodb 服务时，或许可以去看看 iptables 相关的东西。
mongoose.connect('mongodb://localhost/test');

// 上面说了，我推荐在同一个 collection 中使用固定的数据形式。
// 在这里，我们创建了一个名为 Cat 的 model，它在数据库中的名字根据传给 mongoose.model 的第一个参数决定，mongoose
// 这个 model 的定义是，有一个 String 类型的 name，String 数组类型的 friends，Number 类型的 age。
// mongodb 中大多数的数据类型都可以用 js 的原生类型来表示。至于说 String 的长度是多少，Number 的精度是多少。String
// 这里可以看到各种示例: http://mongoosejs.com/docs/schematypes.html
var Cat = mongoose.model('Cat', {
  name: String,
  friends: [String],
  age: Number,
});

// new 一个新对象，名叫 kitty
// 接着为 kitty 的属性们赋值
var kitty = new Cat({ name: 'Zildjian', friends: ['tom', 'jerry']});
kitty.age = 3;

// 调用 .save 方法后，mongoose 会去你的 mongodb 中的 test 数据库里，存入一条记录。
kitty.save(function (err) {
  if (err) // ...
    console.log('meow');
});
```

我们可以验证一下

```
$ mongo
MongoDB shell version: 2.6.4
connecting to: test
> show dbs
> use test
> show collections
> db.cats.find()
```

会发现里面就有一条记录了。

设计个简单博客程序

如果要写个博客程序练手。数据库可以这样设计

```
var Post = mongoose.model('Post', {
  title: String,
  content: String,
  author: String,
  create_at: Date,
});
```

评论就不要自己做了，接入多说：<http://duoshuo.com/>

编辑器就纯文本好了，用 markdown 写。

用户系统也不做，硬编码几个管理员账号在配置文件中，然后用 http basic auth：<https://github.com/jshttp/basic-auth> 来做验证。

示例程序

Nodeclub 是使用 Node.js 和 MongoDB 开发的社区系统

<https://github.com/cnodejs/nodeclub>

完。

《cookie 和 session》

众所周知，HTTP 是一个无状态协议，所以客户端每次发出请求时，下一次请求无法得知上一次请求所包含的状态数据，如何能把一个用户的状态数据关联起来呢？

比如在淘宝的某个页面中，你进行了登陆操作。当你跳转到商品页时，服务端如何知道你是已经登陆的状态？

cookie

首先产生了 cookie 这门技术来解决这个问题，cookie 是 http 协议的一部分，它的处理分为如下几步：

- 服务器向客户端发送 cookie。
 - 通常使用 HTTP 协议规定的 set-cookie 头操作。
 - 规范规定 cookie 的格式为 name = value 格式，且必须包含这部分。
- 浏览器将 cookie 保存。
- 每次请求浏览器都会将 cookie 发向服务器。

其他可选的 cookie 参数会影响将 cookie 发送给服务器端的过程，主要有以下几种：

- path：表示 cookie 影响到的路径，匹配该路径才发送这个 cookie。
- expires 和 maxAge：告诉浏览器这个 cookie 什么时候过期，expires 是 UTC 格式时间，maxAge 是 cookie 多久后过期的相对时间。当不设置这两个选项时，会产生 session cookie，session cookie 是 transient 的，当用户关闭浏览器时，就被清除。一般用来保存 session 的 session_id。
- secure：当 secure 值为 true 时，cookie 在 HTTP 中是无效，在 HTTPS 中才有效。
- httpOnly：浏览器不允许脚本操作 document.cookie 去更改 cookie。一般情况下都应该设置这个为 true，这样可以避免被 xss 攻击拿到 cookie。

express 中的 cookie

express 在 4.x 版本之后，session 管理和 cookies 等许多模块都不再直接包含在 express 中，而是需要单独添加相应模块。

express4 中操作 cookie 使用 cookie-parser 模块 (<https://github.com/expressjs/cookie-parser>)。

```
var express = require('express');
// 首先引入 cookie-parser 这个模块
var cookieParser = require('cookie-parser');

var app = express();
app.listen(3000);

// 使用 cookieParser 中间件，cookieParser(secret, options)
// 其中 secret 用来加密 cookie 字符串（下面会提到 signedCookies）
// options 传入上面介绍的 cookie 可选参数
app.use(cookieParser());
```

```

app.get('/', function (req, res) {
  // 如果请求中的 cookie 存在 isVisit, 则输出 cookie
  // 否则, 设置 cookie 字段 isVisit, 并设置过期时间为 1 分钟
  if (req.cookies.isVisit) {
    console.log(req.cookies);
    res.send(" 再次欢迎访问");
  } else {
    res.cookie('isVisit', 1, {maxAge: 60 * 1000});
    res.send(" 欢迎第一次访问");
  }
});

```

session

cookie 虽然很方便, 但是使用 cookie 有一个很大的弊端, cookie 中的所有数据在客户端就可以被修改, 数据非常容易被伪造, 那么一些重要的数据就不能存放在 cookie 中了, 而且如果 cookie 中数据字段太多会影响传输效率。为了解决这些问题, 就产生了 session, session 中的数据是保留在服务器端的。

session 的运作通过一个 `session_id` 来进行。`session_id` 通常是存放在客户端的 cookie 中, 比如在 express 中, 默认是 `connect.sid` 这个字段, 当请求到来时, 服务端检查 cookie 中保存的 `session_id` 并通过这个 `session_id` 与服务器端的 session data 关联起来, 进行数据的保存和修改。

这意思就是说, 当你浏览一个网页时, 服务端随机产生一个 1024 比特长的字符串, 然后存在你 cookie 中的 `connect.sid` 字段中。当你下次访问时, cookie 会带有这个字符串, 然后浏览器就知道你是上次访问过的某某某, 然后从服务器的存储中取出上次记录在你身上的数据。由于字符串是随机产生的, 而且位数足够多, 所以也不担心有人能够伪造。伪造成功的概率比坐在家编程时被邻居家的狗突然闯入并咬死的几率还低。

session 可以存放在 1) 内存、2) cookie 本身、3) redis 或 memcached 等缓存中, 或者 4) 数据库中。线上来说, 缓存的方案比较常见, 存数据库的话, 查询效率相比前三者都太低, 不推荐; cookie session 有安全性问题, 下面会提到。

express 中操作 session 要用到 `express-session` (<https://github.com/expressjs/session>) 这个模块, 主要的方法就是 `session(options)`, 其中 options 中包含可选参数, 主要有:

- name: 设置 cookie 中, 保存 session 的字段名称, 默认为 `connect.sid`。
- store: session 的存储方式, 默认存放在内存中, 也可以使用 redis, mongodb 等。express 生态中都有相应模块的支持。
- secret: 通过设置的 secret 字符串, 来计算 hash 值并放在 cookie 中, 使产生的 signedCookie 防篡改。
- cookie: 设置存放 session id 的 cookie 的相关选项, 默认为
 - (default: { path: '/', httpOnly: true, secure: false, maxAge: null })
- genid: 产生一个新的 session_id 时, 所使用的函数, 默认使用 `uid2` 这个 npm 包。
- rolling: 每个请求都重新设置一个 cookie, 默认为 false。
- resave: 即使 session 没有被修改, 也保存 session 值, 默认为 true。

1) 在内存中存储 session

`express-session` 默认使用内存来存 session, 对于开发调试来说很方便。

```

var express = require('express');
// 首先引入 express-session 这个模块
var session = require('express-session');

var app = express();
app.listen(5000);

// 按照上面的解释, 设置 session 的可选参数
app.use(session({
  secret: 'recommand 128 bytes random string', // 建议使用 128 个字符的随机字符串
  cookie: { maxAge: 60 * 1000 }
}));

app.get('/', function (req, res) {
  // 检查 session 中的 isVisit 字段
  // 如果存在则增加一次, 否则为 session 设置 isVisit 字段, 并初始化为 1。
  if(req.session.isVisit) {

```



```

    req.session.isVisit++;
    res.send('<p> 第 ' + req.session.isVisit + ' 次来此页面</p>');
  } else {
    req.session.isVisit = 1;
    res.send(" 欢迎第一次来这里");
    console.log(req.session);
  }
});

```

2) 在 redis 中存储 session

session 存放在内存中不方便进程间共享，因此可以使用 redis 等缓存来存储 session。

假设你的机器是 4 核的，你使用了 4 个进程在跑同一个 node web 服务，当用户访问进程 1 时，他被设置了一些数据当做 session 存在内存中。而下一次访问时，他被负载均衡到了进程 2，则此时进程 2 的内存中没有他的信息，认为他是个新用户。这就会导致用户在我们服务中的状态不一致。

使用 redis 作为缓存，可以使用 `connect-redis` 模块 (<https://github.com/tj/connect-redis>) 来得到 redis 连接实例，然后在 session 中设置存储方式为该实例。

```

var express = require('express');
var session = require('express-session');
var redisStore = require('connect-redis')(session);

var app = express();
app.listen(5000);

app.use(session({
  // 假如你不想使用 redis 而想要使用 memcached 的话，代码改动也不会超过 5 行。
  // 这些 store 都遵循着统一的接口，凡是实现了那些接口的库，都可以作为 session 的 store 使用，比如都需要实现 .get()
  // 编写自己的 store 也很简单
  store: new redisStore(),
  secret: 'somesecrettoken'
}));

app.get('/', function (req, res) {
  if(req.session.isVisit) {
    req.session.isVisit++;
    res.send('<p> 第 ' + req.session.isVisit + ' 次来到此页面</p>');
  } else {
    req.session.isVisit = 1;
    res.send(' 欢迎第一次来这里');
  }
});

```

我们可以运行 `redis-cli` 查看结果，如图可以看到 redis 中缓存结果。

```

Ricardo-Li:~ Ricardo_Li$ redis-cli
127.0.0.1:6379> KEYS *
1) "sess:zjC61f9dqIKuzHsZtViBIc0X7pr3c0-g"
127.0.0.1:6379> MGET "sess:zjC61f9dqIKuzHsZtViBIc0X7pr3c0-g"
1) "{\"cookie\":{\"originalMaxAge\":null,\"expires\":null,\"httpOnly\":true,\"path\":\"/\"},\"isVisit\":3}"
127.0.0.1:6379> exit
Ricardo-Li:~ Ricardo_Li$

```

各种存储的利弊

上面我们说到，session 的 store 有四个常用选项：1) 内存 2) cookie 3) 缓存 4) 数据库

其中，开发环境存内存就好了。一般的小程序为了省事，如果不涉及状态共享的问题，用内存 session 也没问题。但内存 session 除了省事之外，没有别的好处。

cookie session 我们下面会提到，现在说说利弊。用 cookie session 的话，是不用担心状态共享问题的，因为 session 的 data 不是由服务器来保存，而是保存在用户浏览器端，每次用户访问时，都会主动带上他自己的信息。当然在这里，安全性之类的，只要遵照最佳实践来，也是有保证的。它的弊端是增大了数据量传输，利端是方便。

缓存方式是最常用的方式了，即快，又能共享状态。相比 cookie session 来说，当 session data 比较大的时候，可以节省网络传输。推荐使用。

数据库 session。除非你很熟悉这一块，知道自己要什么，否则还是老老实实用缓存吧。

signedCookie

上面都是讲基础，现在讲一些专业点的。

上面有提到

cookie 虽然很方便，但是使用 cookie 有一个很大的弊端，cookie 中的所有数据在客户端就可以被修改，数据非常容易被伪造

其实不是这样的，那只是为了方便理解才那么写。要知道，计算机领域有个名词叫 签名，专业点说，叫 信息摘要算法。

比如我们现在面临着一个菜鸟开发的网站，他用 cookie 来记录登陆的用户凭证。相应的 cookie 长这样：`dotcom_user=alsotang`，它说明现在的用户是 alsotang 这个用户。如果我在浏览器中装个插件，把它改成 `dotcom_user=ricardo`，服务器一读取，就会误认为我是 ricardo。然后我就可以进行 ricardo 才能进行的操作了。之前 web 开发不成熟的时候，用这招甚至可以黑个网站下来，把 cookie 改成 `dotcom_user=admin` 就行了，唉，那是个玩黑客的黄金年代啊。

OK，现在我有一些数据，不想存在 session 中，想存在 cookie 中，怎么保证不被篡改呢？答案很简单，签个名。

假设我的服务器有个秘密字符串，是 `this_is_my_secret_and_fuck_you_all`，我为用户 cookie 的 `dotcom_user` 字段设置了个值 `alsotang`。cookie 本应是

```
{dotcom_user: 'alsotang'}
```

这样的。

而如果我们签个名，比如把 `dotcom_user` 的值跟我的 `secret_string` 做个 sha1

```
sha1('this_is_my_secret_and_fuck_you_all' + 'alsotang') === '4850a42e3bc0d39c978770392cbd8dc2923e3d1d'
```

然后把 cookie 变成这样

```
{
  dotcom_user: 'alsotang',
  'dotcom_user.sig': '4850a42e3bc0d39c978770392cbd8dc2923e3d1d',
}
```

这样一来，用户就没法伪造信息了。一旦它更改了 cookie 中的信息，则服务器会发现 hash 校验的不一致。

毕竟他不懂我们的 `secret_string` 是什么，而暴力破解哈希值的成本太高。

cookie-session

上面一直提到 session 可以存在 cookie 中，现在来讲讲具体的思路。这里所涉及的专业名词叫做对称加密。

假设我们想在用户的 cookie 中存 session data，使用一个名为 `session_data` 的字段。

存

```
var sessionData = {username: 'alsotang', age: 22, company: 'alibaba', location: 'hangzhou'}
```

这段信息的话，可以将 `sessionData` 与我们的 `secret_string` 一起做个对称加密，存到 cookie 的 `session_data` 字段中，只要你的 `secret_string` 足够长，那么攻击者也是无法获取实际 session 内容的。对称加密之后的内容对于攻击者来说相当于一段乱码。

而当用户下次访问时，我们就可以用 `secret_string` 来解密 `sessionData`，得到我们需要的 session data。

signedCookies 跟 cookie-session 还是有区别的：

1) 是前者信息可见不可篡改, 后者不可见也不可篡改

2) 是前者一般是长期保存, 而后者是 session cookie

cookie-session 的实现跟 signedCookies 差不多。

不过 cookie-session 我个人建议不要使用, 有受到回放攻击的危险。

回放攻击指的是, 比如一个用户, 它现在有 100 积分, 积分存在 session 中, session 保存在 cookie 中。他先复制下现在的这段 cookie, 然后去发个帖子, 扣掉了 20 积分, 于是他就只有 80 积分了。而他现在可以将之前复制下的那段 cookie 再粘贴回去浏览器中, 于是服务器在一些场景下会认为他又有了 100 积分。

如果避免这种攻击呢? 这就需要引入一个第三方的手段来验证 cookie session, 而验证所需的信息, 一定不能存在 cookie 中。这么一来, 避免了这种攻击后, 使用 cookie session 的好处就荡然无存了。如果为了避免攻击而引入了缓存使用的话, 那不如把 cookie session 也一起放进缓存中。

session cookie

初学者容易犯的一个错误是, 忘记了 session_id 在 cookie 中的存储方式是 session cookie。即, 当用户一关闭浏览器, 浏览器 cookie 中的 session_id 字段就会消失。

常见的场景就是在开发用户登陆状态保持时。

假如用户在之前登陆了你的网站, 你在他对应的 session 中存了信息, 当他关闭浏览器再次访问时, 你还是不懂他是谁。所以我们要在 cookie 中, 也保存一份关于用户身份的信息。

比如有这样一个用户

```
{username: 'alsotang', age: 22, company: 'alibaba', location: 'hangzhou'}
```

我们可以考虑把这四个字段的信息都存在 session 中, 而在 cookie, 我们用 signedCookies 来存个 username。

登陆的检验过程伪代码如下:

```
if (req.session.user) {
  // 获取 user 并进行下一步
  next()
} else if (req.signedCookies['username']) {
  // 如果存在则从数据库中获取这个 username 的信息, 并保存到 session 中
  getuser(function (err, user) {
    req.session.user = user;
    next();
  });
} else {
  // 当做为登陆用户处理
  next();
}
```

etag 当做 session, 保存 http 会话

很黑客的一种玩法: <https://cnnodejs.org/topic/5212d82d0a746c580b43d948>

《使用 promise 替代回调函数》

知识点

1. 理解 Promise 概念, 为什么需要 promise
2. 学习 q 的 API, 利用 q 来替代回调函数 (<https://github.com/kriskowal/q>)

课程内容

第五课 (<https://github.com/alsotang/node-lessons/tree/master/lesson5>) 讲述了如何使用 `async` 来控制并发。`async` 的本质是一个流程控制。其实在异步编程中，还有一个更为经典的模型，叫做 `Promise/Deferred` 模型。

本节我们就来学习这个模型的代表实现：`q`

首先，我们思考一个典型的异步编程模型，考虑这样一个题目：读取一个文件，在控制台输出这个文件内容。

```
var fs = require('fs');
fs.readFile('sample.txt', 'utf8', function (err, data) {
  console.log(data);
});
```

看起来很简单，再进一步：读取两个文件，在控制台输出这两个文件内容。

```
var fs = require('fs');
fs.readFile('sample01.txt', 'utf8', function (err, data) {
  console.log(data);
  fs.readFile('sample02.txt', 'utf8', function (err, data) {
    console.log(data);
  });
});
```

要是读取更多的文件呢？

```
var fs = require('fs');
fs.readFile('sample01.txt', 'utf8', function (err, data) {
  fs.readFile('sample02.txt', 'utf8', function (err, data) {
    fs.readFile('sample03.txt', 'utf8', function (err, data) {
      fs.readFile('sample04.txt', 'utf8', function (err, data) {
        // ...
      });
    });
  });
});
```

这段代码就是臭名昭著的邪恶金字塔 (Pyramid of Doom)。可以使用 `async` 来改善这段代码，但是在本课中我们要用 `promise/defer` 来改善它。

promise 基本概念

先学习 `promise` 的基本概念。

- `promise` 只有三种状态，未完成，完成 (`fulfilled`) 和失败 (`rejected`)。
- `promise` 的状态可以由未完成转换成完成，或者未完成转换成失败。
- `promise` 的状态转换只发生一次

`promise` 有一个 `then` 方法，`then` 方法可以接受 3 个函数作为参数。前两个函数对应 `promise` 的两种状态 `fulfilled`, `rejected` 的回调函数。第三个函数用于处理进度信息。

```
promiseSomething().then(function(fulfilled){
  //当 promise 状态变成 fulfilled 时，调用此函数
}, function(rejected){
  //当 promise 状态变成 rejected 时，调用此函数
}, function(progress){
  //当返回进度信息时，调用此函数
});
```

学习一个简单的例子：

```
var Q = require('q');
var defer = Q.defer();
/**
 * 获取初始 promise
 * @private
```

```

*/
function getInitialPromise() {
    return defer.promise;
}
/**
 * 为 promise 设置三种状态的回调函数
 */
getInitialPromise().then(function(success){
    console.log(success);
},function(error){
    console.log(error);
},function(progress){
    console.log(progress);
});
defer.notify('in progress');//控制台打印 in progress
defer.resolve('resolve');//控制台打印 resolve
defer.reject('reject');//没有输出。promise 的状态只能改变一次

```

promise 的传递

then 方法会返回一个 promise，在下面这个例子中，我们用 outputPromise 指向 then 返回的 promise。

```

var outputPromise = getInputPromise().then(function (fulfilled) {
}, function (rejected) {
});

```

现在 outputPromise 就变成了受 function(fulfilled) 或者 function(rejected) 控制状态的 promise 了。怎么理解这句话呢？

- 当 function(fulfilled) 或者 function(rejected) 返回一个值，比如一个字符串，数组，对象等等，那么 outputPromise 的状态就会变成 fulfilled。

在下面这个例子中，我们可以看到，当我们把 inputPromise 的状态通过 defer.resolve() 变成 fulfilled 时，控制台输出 fulfilled。

当我们把 inputPromise 的状态通过 defer.reject() 变成 rejected，控制台输出 rejected

```

var Q = require('q');
var defer = Q.defer();
/**
 * 通过 defer 获得 promise
 * @private
 */
function getInputPromise() {
    return defer.promise;
}

/**
 * 当 inputPromise 状态由未完成变成 fulfil 时，调用 function(fulfilled)
 * 当 inputPromise 状态由未完成变成 rejected 时，调用 function(rejected)
 * 将 then 返回的 promise 赋给 outputPromise
 * function(fulfilled) 和 function(rejected) 通过返回字符串将 outputPromise 的状态由
 * 未完成改变为 fulfilled
 * @private
 */
var outputPromise = getInputPromise().then(function(fulfilled){
    return 'fulfilled';
},function(rejected){
    return 'rejected';
});

/**
 * 当 outputPromise 状态由未完成变成 fulfil 时，调用 function(fulfilled)，控制台打印'fulfilled: fulfilled'。
 * 当 outputPromise 状态由未完成变成 rejected，调用 function(rejected)，控制台打印'rejected: rejected'。

```

```

*/
outputPromise.then(function(fulfilled){
    console.log('fulfilled: ' + fulfilled);
},function(rejected){
    console.log('rejected: ' + rejected);
});

/**
 * 将 inputPromise 的状态由未完成变成 rejected
 */
defer.reject(); //输出 fulfilled: rejected

/**
 * 将 inputPromise 的状态由未完成变成 fulfilled
 */
//defer.resolve(); //输出 fulfilled: fulfilled
    • 当 function(fulfilled) 或者 function(rejected) 抛出异常时, 那么 outputPromise 的状态就会变成 rejected

var Q = require('q');
var fs = require('fs');
var defer = Q.defer();

/**
 * 通过 defer 获得 promise
 * @private
 */
function getInputPromise() {
    return defer.promise;
}

/**
 * 当 inputPromise 状态由未完成变成 fulfil 时, 调用 function(fulfilled)
 * 当 inputPromise 状态由未完成变成 rejected 时, 调用 function(rejected)
 * 将 then 返回的 promise 赋给 outputPromise
 * function(fulfilled) 和 function(rejected) 通过抛出异常将 outputPromise 的状态由
 * 未完成改变为 reject
 * @private
 */
var outputPromise = getInputPromise().then(function(fulfilled){
    throw new Error('fulfilled');
},function(rejected){
    throw new Error('rejected');
});

/**
 * 当 outputPromise 状态由未完成变成 fulfil 时, 调用 function(fulfilled)。
 * 当 outputPromise 状态由未完成变成 rejected, 调用 function(rejected)。
 */
outputPromise.then(function(fulfilled){
    console.log('fulfilled: ' + fulfilled);
},function(rejected){
    console.log('rejected: ' + rejected);
});

/**
 * 将 inputPromise 的状态由未完成变成 rejected
 */
defer.reject(); //控制台打印 rejected [Error:rejected]

/**
 * 将 inputPromise 的状态由未完成变成 fulfilled

```

```
*/
//defer.resolve(); //控制台打印 rejected [Error:fulfilled]
```

• 当 function(fulfilled) 或者 function(rejected) 返回一个 promise 时，outputPromise 就会成为这个新的 promise。这样做有什么意义呢？主要在于聚合结果 (Q.all)，管理延时，异常恢复等等

比如说我们想要读取一个文件的内容，然后把这些内容打印出来。可能会写出这样的代码：

```
//错误的写法
```

```
var outputPromise = getInputPromise().then(function(fulfilled){
    fs.readFile('test.txt','utf8',function(err,data){
        return data;
    });
});
```

然而这样写是错误的，因为 function(fulfilled) 并没有返回任何值。需要下面的方式：

```
var Q = require('q');
var fs = require('fs');
var defer = Q.defer();
```

```
/**
 * 通过 defer 获得 promise
 * @private
 */
```

```
function getInputPromise() {
    return defer.promise;
}
```

```
/**
 * 当 inputPromise 状态由未完成变成 fulfil 时，调用 function(fulfilled)
 * 当 inputPromise 状态由未完成变成 rejected 时，调用 function(rejected)
 * 将 then 返回的 promise 赋给 outputPromise
 * function(fulfilled) 将新的 promise 赋给 outputPromise
 * 未完成改变为 reject
 * @private
 */
```

```
var outputPromise = getInputPromise().then(function(fulfilled){
    var myDefer = Q.defer();
    fs.readFile('test.txt','utf8',function(err,data){
        if(!err && data) {
            myDefer.resolve(data);
        }
    });
    return myDefer.promise;
},function(rejected){
    throw new Error('rejected');
});
```

```
/**
 * 当 outputPromise 状态由未完成变成 fulfil 时，调用 function(fulfilled)，控制台打印 test.txt 文件内容。
 *
 */
```

```
outputPromise.then(function(fulfilled){
    console.log(fulfilled);
},function(rejected){
    console.log(rejected);
});
```

```
/**
 * 将 inputPromise 的状态由未完成变成 rejected
 */
//defer.reject();
```

```

/**
 * 将 inputPromise 的状态由未完成变成 fulfilled
 */
defer.resolve(); //控制台打印出 test.txt 的内容

```

方法传递

方法传递有些类似于 Java 中的 try 和 catch。当一个异常没有响应的捕获时，这个异常会接着往下传递。方法传递的含义是当一个状态没有响应的回调函数，就会沿着 then 往下找。

- 没有提供 function(rejected)

```
var outputPromise = getInputPromise().then(function(fulfilled){})
```

如果 inputPromise 的状态由未完成变成 rejected, 此时对 rejected 的处理会由 outputPromise 来完成。

```

var Q = require('q');
var fs = require('fs');
var defer = Q.defer();

```

```

/**
 * 通过 defer 获得 promise
 * @private
 */

```

```

function getInputPromise() {
    return defer.promise;
}

```

```

/**
 * 当 inputPromise 状态由未完成变成 fulfil 时，调用 function(fulfilled)
 * 当 inputPromise 状态由未完成变成 rejected 时，这个 rejected 会传向 outputPromise
 */

```

```

var outputPromise = getInputPromise().then(function(fulfilled){
    return 'fulfilled'
});
outputPromise.then(function(fulfilled){
    console.log('fulfilled: ' + fulfilled);
},function(rejected){
    console.log('rejected: ' + rejected);
});

```

```

/**
 * 将 inputPromise 的状态由未完成变成 rejected
 */

```

```
defer.reject('inputpromise rejected'); //控制台打印 rejected: inputpromise rejected
```

```

/**
 * 将 inputPromise 的状态由未完成变成 fulfilled
 */
//defer.resolve();

```

- 没有提供 function(fulfilled)

```
var outputPromise = getInputPromise().then(null,function(rejected){})
```

如果 inputPromise 的状态由未完成变成 fulfilled, 此时对 fulfil 的处理会由 outputPromise 来完成。

```

var Q = require('q');
var fs = require('fs');
var defer = Q.defer();

```

```

/**

```

```

* 通过 defer 获得 promise
* @private
*/
function getInputPromise() {
    return defer.promise;
}

/**
 * 当 inputPromise 状态由未完成变成 fulfilled 时, 传递给 outputPromise
 * 当 inputPromise 状态由未完成变成 rejected 时, 调用 function(rejected)
 * function(fulfilled) 将新的 promise 赋给 outputPromise
 * 未完成改变为 reject
 * @private
 */
var outputPromise = getInputPromise().then(null,function(rejected){
    return 'rejected';
});

outputPromise.then(function(fulfilled){
    console.log('fulfilled: ' + fulfilled);
},function(rejected){
    console.log('rejected: ' + rejected);
});

/**
 * 将 inputPromise 的状态由未完成变成 rejected
 */
//defer.reject('inputpromise rejected');

/**
 * 将 inputPromise 的状态由未完成变成 fulfilled
 */
defer.resolve('inputpromise fulfilled'); //控制台打印 fulfilled: inputpromise fulfilled
    • 可以使用 fail(function(error)) 来专门针对错误处理, 而不是使用 then(null,function(error))

    var outputPromise = getInputPromise().fail(function(error){})

```

看这个例子

```

var Q = require('q');
var fs = require('fs');
var defer = Q.defer();

/**
 * 通过 defer 获得 promise
 * @private
 */
function getInputPromise() {
    return defer.promise;
}

/**
 * 当 inputPromise 状态由未完成变成 fulfilled 时, 调用 then(function(fulfilled))
 * 当 inputPromise 状态由未完成变成 rejected 时, 调用 fail(function(error))
 * function(fulfilled) 将新的 promise 赋给 outputPromise
 * 未完成改变为 reject
 * @private
 */
var outputPromise = getInputPromise().then(function(fulfilled){
    return fulfilled;
}).fail(function(error){
    console.log('fail: ' + error);
});

```



```

});
/**
 * 将 inputPromise 的状态由未完成变成 rejected
 */
defer.reject('inputpromise rejected');//控制台打印 fail: inputpromise rejected

/**
 * 将 inputPromise 的状态由未完成变成 fulfilled
 */
//defer.resolve('inputpromise fulfilled');
    • 可以使用 progress(function(progress)) 来专门针对进度信息进行处理,而不是使用 then(function(success){},function(erro
var Q = require('q');
var defer = Q.defer();
/**
 * 获取初始 promise
 * @private
 */
function getInitialPromise() {
    return defer.promise;
}
/**
 * 为 promise 设置 progress 信息处理函数
 */
var outputPromise = getInitialPromise().then(function(success){

}).progress(function(progress){
    console.log(progress);
});

defer.notify(1);
defer.notify(2); //控制台打印 1, 2

```

promise 链

promise 链提供了一种让函数顺序执行的方法。

函数顺序执行是很重要的一个功能。比如知道用户名, 需要根据用户名从数据库中找到相应的用户, 然后将用户信息传给下一个函数进行处理。

```

var Q = require('q');
var defer = Q.defer();

//一个模拟数据库
var users = [{ 'name': 'andrew', 'passwd': 'password' }];

function getUsername() {
    return defer.promise;
}

function getUser(username){
    var user;
    users.forEach(function(element){
        if(element.name === username) {
            user = element;
        }
    });
    return user;
}

//promise 链

```

```

    getUsername().then(function(username){
        return getUser(username);
    }).then(function(user){
        console.log(user);
    });

```

```
defer.resolve('andrew');
```

我们通过两个 then 达到让函数顺序执行的目的。

then 的数量其实是没有限制的。当然，then 的数量过多，要手动把他们链接起来是很麻烦的。比如

```
foo(initialVal).then(bar).then(baz).then(qux)
```

这时我们需要用代码来动态制造 promise 链

```

var funcs = [foo,bar,baz,qux]
var result = Q(initialVal)
funcs.forEach(function(func){
    result = result.then(func)
})
return result

```

当然，我们可以再简洁一点

```

var funcs = [foo,bar,baz,qux]
funcs.reduce(function(pre,current),Q(initialVal){
    return pre.then(current)
})

```

看一个具体的例子

```

function foo(result) {
    console.log(result);
    return result+result;
}
//手动链接
Q('hello').then(foo).then(foo).then(foo);

```

```

//控制台输出:  hello
//              hellohello
//              hellohellohello

```

```

//动态链接
var funcs = [foo,foo,foo];
var result = Q('hello');
funcs.forEach(function(func){
    result = result.then(func);
});

```

```

//精简后的动态链接
funcs.reduce(function(prev,current){
    return prev.then(current);
},Q('hello'));

```

对于 promise 链，最重要的是需要理解为什么这个链能够顺序执行。如果能够理解这点，那么以后自己写 promise 链可以说是轻车熟路啊。

promise 组合

回到我们一开始读取文件内容的例子。如果现在让我们把它改写成 promise 链，是不是很简单呢？

```

var Q = require('q'),
    fs = require('fs');
function printFileContent(fileName) {
    return function(){
        var defer = Q.defer();
        fs.readFile(fileName,'utf8',function(err,data){

```

```

        if(!err && data) {
            console.log(data);
            defer.resolve();
        }
    })
    return defer.promise;
}
}
//手动链接
printFileContent('sample01.txt')()
    .then(printFileContent('sample02.txt'))
    .then(printFileContent('sample03.txt'))
    .then(printFileContent('sample04.txt')); //控制台顺序打印 sample01 到 sample04 的内容

```

很有成就感是不是。然而如果仔细分析，我们会发现为什么要他们顺序执行呢，如果他们能够并行执行不是更好吗？我们只需要在他们都执行完成之后，得到他们的执行结果就可以了。

我们可以通过 `Q.all([promise1,promise2...])` 将多个 promise 组合成一个 promise 返回。注意：

1. 当 all 里面所有的 promise 都 fulfil 时，Q.all 返回的 promise 状态变成 fulfil
2. 当任意一个 promise 被 reject 时，Q.all 返回的 promise 状态立即变成 reject

我们来把上面读取文件内容的例子改成并行执行吧

```

var Q = require('q');
var fs = require('fs');
/**
 * 读取文件内容
 * @private
 */
function printFileContent(fileName) {
    //Todo: 这段代码不够简洁。可以使用 Q.denodeify 来简化
    var defer = Q.defer();
    fs.readFile(fileName,'utf8',function(err,data){
        if(!err && data) {
            console.log(data);
            defer.resolve(fileName + ' success ');
        }else {
            defer.reject(fileName + ' fail ');
        }
    })
    return defer.promise;
}

```

```

Q.all([printFileContent('sample01.txt'),printFileContent('sample02.txt'),printFileContent('sample03.txt'),p
    .then(function(success){
        console.log(success);
    }); //控制台打印各个文件内容 顺序不一定

```

现在知道 Q.all 会在任意一个 promise 进入 reject 状态后立即进入 reject 状态。如果我们需要等到所有的 promise 都发生状态后 (有的 fulfil, 有的 reject)，再转换 Q.all 的状态, 这时我们可以使用 `Q.allSettled`

```

var Q = require('q'),
    fs = require('fs');
/**
 * 读取文件内容
 * @private
 */
function printFileContent(fileName) {
    //Todo: 这段代码不够简洁。可以使用 Q.denodeify 来简化
    var defer = Q.defer();
    fs.readFile(fileName,'utf8',function(err,data){
        if(!err && data) {
            console.log(data);
            defer.resolve(fileName + ' success ');
        }
    })
    return defer.promise;
}

```

```

    }else {
      defer.reject(fileName + ' fail ');
    }
  })
  return defer.promise;
}

Q.allSettled([printFileContent('nosuchfile.txt'),printFileContent('sample02.txt'),printFileContent('sample03.txt')])
  .then(function(results){
    results.forEach(
      function(result) {
        console.log(result.state);
      }
    );
  });
});

```

结束 promise 链

通常，对于一个 promise 链，有两种结束的方式。第一种方式是返回最后一个 promise

如 `return foo().then(bar);`

第二种方式就是通过 `done` 来结束 promise 链

如 `foo().then(bar).done()`

为什么需要通过 `done` 来结束一个 promise 链呢？如果在我们的链中有错误没有被处理，那么在一个正确结束的 promise 链中，这个没被处理的错误会通过异常抛出。

```

var Q = require('q');
/**
 * @private
 */
function getPromise(msg,timeout,opt) {
  var defer = Q.defer();
  setTimeout(function(){
    console.log(msg);
    if(opt)
      defer.reject(msg);
    else
      defer.resolve(msg);
  },timeout);
  return defer.promise;
}
/**
 * 没有用 done() 结束的 promise 链
 * 由于 getPromise('2',2000,'opt') 返回 rejected, getPromise('3',1000) 就没有执行
 * 然后这个异常并没有任何提醒，是一个潜在的 bug
 */
getPromise('1',3000)
  .then(function(){return getPromise('2',2000,'opt')})
  .then(function(){return getPromise('3',1000)});
/**
 * 用 done() 结束的 promise 链
 * 有异常抛出
 */
getPromise('1',3000)
  .then(function(){return getPromise('2',2000,'opt')})
  .then(function(){return getPromise('3',1000)})
  .done();

```

结束语

当你理解完上面所有的知识点时，你就会正确高效的使用 promise 了。本节只是讲了 promise 的原理和几个基本的 API，不过你掌握了这些之后，再去看 q 的文档，应该很容易就能理解各个 api 的意图。

《何为 connect 中间件》

目标

1. 理解中间件的概念
2. 了解 Connect 的实现

课程内容

1. 原生 httpServer 遇到的问题
2. 中间件思想
3. Connect 实现
4. Express 简介

这是从 httpServer 到 Express 的升级过程。

HTTP

Nodejs 的经典 httpServer 代码

```
var http = require('http');

var server = http.createServer(requestHandler);
function requestHandler(req, res) {
  res.end('hello visitor!');
}
server.listen(3000);
```

里面的函数 `requestHandler` 就是所有 http 请求的响应函数，即所有的请求都经过这个函数的处理，是所有请求的入口函数。

通过 `requestHandler` 函数我们能写一些简单的 http 逻辑，比如上面的例子，所有请求都返回 `hello visitor!`。

然而，我们的业务逻辑不可能这么简单。例如：需要实现一个接口，要做的是当请求过来时，先判断来源的请求是否包含请求体，然后判断请求体中的 `id` 是不是在数据库中存在，最后若存在则返回 `true`，不存在则返回 `false`。

1. 检测请求中请求体是否存在，若存在则解析请求体；
1. 查看请求体中的 `id` 是否存在，若存在则去数据库查询；
1. 根据数据库结果返回约定的值；

我们首先想到的，抽离函数，每个逻辑一个函数，简单好实现低耦合好维护。

实现代码：

```
function parseBody(req, callback) {
  //根据 http 协议从 req 中解析 body
  callback(null, body);
}
function checkIdInDatabase(body, callback) {
  //根据 body.id 在 Database 中检测，返回结果
  callback(null, dbResult);
}
function returnResult(dbResult, res) {
  if (dbResult && dbResult.length > 0) {
    res.end('true');
  } else {
```

```

    res.end('false')
  }
}
function requestHandler(req, res) {
  parseBody(req, function(err, body) {
    checkIdInDatabase(body, function(err, dbResult) {
      returnResult(dbResult, res);
    });
  });
});
}

```

上面的解决方案解决了包含三个步骤的业务问题，出现了 3 个 `});`；还有 3 个 `err` 需要处理，上面的写法可以得达到预期效果。

然而，业务逻辑越来越复杂，会出发展成 30 个回调逻辑，那么就出现了 30 个 `});`；及 30 个 `err` 异常。更严重的是，到时候写代码根本看不清自己写的逻辑在 30 层中的哪一层，极其容易出现 多次返回或返回地方不对等问题，这就是 回调金字塔问题了。

大多数同学应该能想到解决回调金字塔的办法，朴灵的《深入浅出 Node.js》里讲到的三种方法。下面列举了这三种方法加上 ES6 新增的 Generator，共四种解决办法。

- EventProxy —— 事件发布订阅模式（第四课讲到）
- BlueBird —— Promise 方案（第十七课讲到）
- Async —— 异步流程控制库（第五课讲到）
- Generator —— ES6 原生 Generator

理论上，这四种都能解决回调金字塔问题。而 Connect 和 Express 用的是 类似异步流程控制的思想。

关于异步流程控制库下面简要介绍下，或移步 [@ 第五课] (<https://github.com/alsotang/node-lessons/tree/master/lesson5>)。异步流程控制库首先要求用户传入待执行的函数列表，记为 `funlist`。流程控制库的任务是让这些函数 顺序执行。

`callback` 是控制顺序执行的关键，`funlist` 里的函数每当调用 `callback` 会执行下一个 `funlist` 里的函数

我们动手实现一个类似的链式调用，其中 `funlist` 更名为 `middlewares`、`callback` 更名为 `next`，代码如下：

```

var middlewares = [
  function fun1(req, res, next) {
    parseBody(req, function(err, body) {
      if (err) return next(err);
      req.body = body;
      next();
    });
  },
  function fun2(req, res, next) {
    checkIdInDatabase(req.body.id, function(err, rows) {
      if (err) return next(err);
      res.dbResult = rows;
      next();
    });
  },
  function fun3(req, res, next) {
    if (res.dbResult && res.dbResult.length > 0) {
      res.end('true');
    }
    else {
      res.end('false');
    }
    next();
  }
]

function requestHandler(req, res) {
  var i=0;

  //由 middlewares 链式调用
  function next(err) {

    if (err) {

```

```

    return res.end('error:', err.toString());
  }

  if (i < middlewares.length) {
    middlewares[i++](req, res, next);
  } else {
    return ;
  }
}

//触发第一个 middleware
next();
}

```

上面用 middlewares+next 完成了业务逻辑的 链式调用，而 middlewares 里的每个函数，都是一个 中间件。

整体思路是：

1. 将所有 处理逻辑函数（中间件）存储在一个 list 中；
2. 请求到达时 循环调用 list 中的 处理逻辑函数（中间件）；

Connect的实现

Connect 的思想跟上面阐述的思想基本一样，先将处理逻辑存起来，然后循环调用。

Connect 中主要有五个函数 PS: Connect 的核心代码是 200+ 行，建议对照源码看下面的函数介绍。

函数名	作用
createServer	包装 httpServer 形成 app
listen	监听端口函数
use	向 middlewares 里面放入业务逻辑
handle	上一章的 requestHandler 函数增强版
call	业务逻辑的真正执行者

createServer()

输入：

无

执行过程：

1. app 是一个函数对象（包含 handle 方法）
2. app 具有 Event 所有属性（详见utils-merge，十行代码）
3. app 有 route 属性（路由）、和 stack 属性（用于存储中间件，类似上面的middlewares）

输出：

```

    app is function(req, res, next) {...};
      |
+---+---+
| has |
| route | stack

```

app.use(route, fn)

作用是向 stack 中添加 逻辑处理函数（中间件）。

输入：

1. route 可省略，默认 '/'

2. fn 具体的业务处理逻辑

tips:

上面的 fn 表示处理逻辑，它可以是

1. 一个普通的 `function(req,res[,next]){};`
2. 一个 `httpServer`;
3. 另一个 `connect` 的 `app` 对象 (sub app 特性);

由于它们的本质都是 处理逻辑，都可以用一个 `function(req,res,next){}` 将它们概括起来，Connect 把他们都转化为这个函数，然后把它们存起来。

如何将这三种分别转换为 `function(req, res, next) {}` 的形式呢？

1. 不用转换;
2. `httpServer` 的定义是“对事件‘request’后 handler 的对象”，我们可以从 `httpServer.listeners('request')` 中得到这个函数;
3. 另一个 `connect` 对象，而 `connect()` 返回的 `app` 就是 `function(req, res, out) {}`;

执行过程:

1. 将三种处理逻辑统一转换为 `function(req,res,next){}` 的形式表示。
2. 把这个处理逻辑与 `route` 一起，放入 `stack` 中 (存储处理逻辑，`route` 用来匹配路由)

核心代码片段

```
//route 是路由路径, handle 是一个 `function(req, res, next) {...}`形式的业务逻辑
this.stack.push({ route: path, handle: handle });
```

返回:

```
//返回自己, 可以完成链式调用
return this;
```

总结::

```
var app = connect();
app.use('/api', function(req, res, next) {});
```

等价于

```
var app = connect();
app.stack.push({route: '/api', handle: function(req, res, next) {}});
```

最后，`app.stack` 里 顺序存储了所有的 逻辑处理函数 (中间件)。

```
app.stack = [function1, function2, function3, ... function30];
```

```
app.handle(req, res, out)
```

这个函数就是请求到达时，负责 顺序调用我们存储在 `stack` 中的 逻辑处理函数 (中间件) 函数，类似上一章的 `requestHandler`。

输入:

1. `req` 是 Nodejs 本身的可读流，不做过多介绍
2. `res` 是 Nodejs 本身的可写流，不做过多介绍
3. `out` 是为了 Connect 的 sub app 特性而设计的参数，这个特性可以暂时忽略，这个参数我们暂时不关心

处理过程:

可以回头看一下上面的`requestHandler` 函数，`handle` 的实现是这个函数的增强版

1. 取得 `stack`(存储逻辑处理函数列表)，`index`(列表下标)
2. 构建 `next` 函数，`next` 的作用是执行下一个逻辑处理函数
3. 触发第一个 `next`，触发链式调用

`next` 函数实现:

`next` 函数实现在 `handle` 函数体内，用来顺序执行处理逻辑，它是异步流程控制库的核心，不明白它的作用请看上面的异步流程控制库简介

`path` 是请求路径，`route` 是逻辑处理函数自带的属性。

1. 取得下一个逻辑处理函数;
2. 若路由不匹配, 跳过此逻辑;
3. 若路由匹配下面的 call 执行匹配到的逻辑处理函数

tips: 跟上一章最后的代码一样, 每个逻辑处理函数调用 `next` 来让后面的函数执行, 存储在 `stack` 中的函数就实现了链式调用。不一定所有的函数都在返回的时候才调用 `next`, 为了不影响效率, 有的函数可能先调用 `next`, 然而自己还没有返回, 继续做自己的事情。

核心代码:

```
//取下一个逻辑逻辑处理函数
1: var layer = stack[index++];
//不匹配时跳过
2: if (path.toLowerCase().substr(0, route.length) !== route.toLowerCase()) {
    return next(err);
}
//匹配时执行
3: call(layer.handle, route, err, req, res, next);
```

返回:

无

总结:

画图总结

```
request come
|
v
middleware1 : 不匹配路由, skip
|
v
middleware2 : 匹配路由, 执行
|
v
middleware3 : 匹配路由, 执行
|
v
middleware4 : 不匹配路由, skip
|
v
end
```

call(handle, route, err, req, res, next)

这里有个比较有趣的知识, `console.log(Function.length)` 会返回函数定义的参数个数。值跟在函数体内执行 `arguments.length` 一样。

Connect 中规定 `function(err, req, res, next) {}` 形式为错误处理函数, `function(req, res, next) {}` 为正常的业务逻辑处理函数。那么, 可以根据 `Function.length` 以判断它是否为错误处理函数。

输入:

参数名	描述
handle	逻辑处理函数
route	路由
err	是否发生过错误
req	Nodejs 对象
res	Nodejs 对象
next	next 函数

处理过程:

1. 是否有错误, 本次 handle 是否是错误处理函数;
2. 若有错误且 handle 为错误处理函数, 则执行 handle, 本函数返回;

3. 若没错误且 handle 不是错误处理函数，则执行 handle，本函数返回；
4. 如果上面两个都不满足，不执行 handle，本函数调用 next，返回；

返回:

无

总结:

call 函数是一个执行者，根据当前错误情况和 handle 类型决定是否执行当前的 handle。

listen

创建一个 httpServer，将 Connect 自己的业务逻辑作为 requestHandler，监听端口

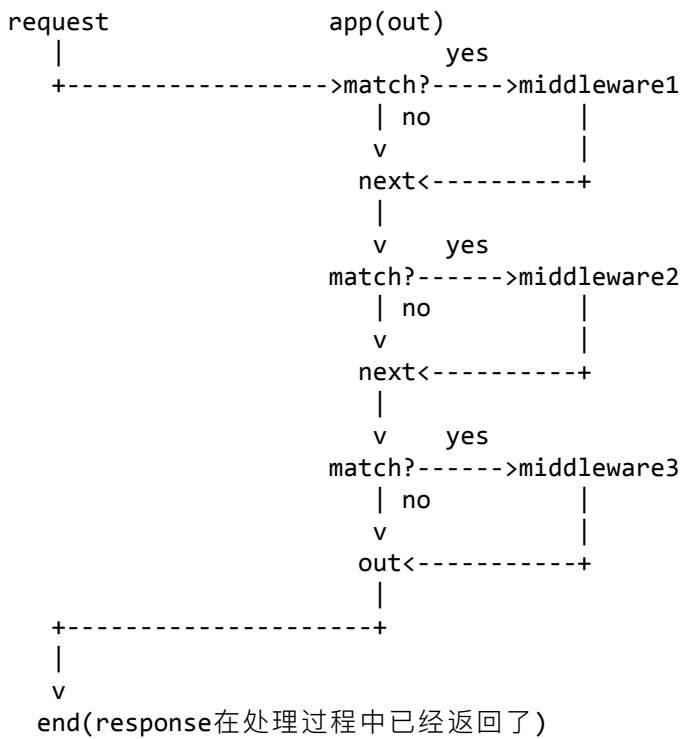
代码

```
var server = http.createServer(this);
return server.listen.apply(server, arguments);
```

图解 Connect

Connect 将中间件存储在 app.stack 中，通过构造 handle 中的 next 函数在请求到来时依次调用这些中间件。

图形总结



Connect 的 subapp 特性

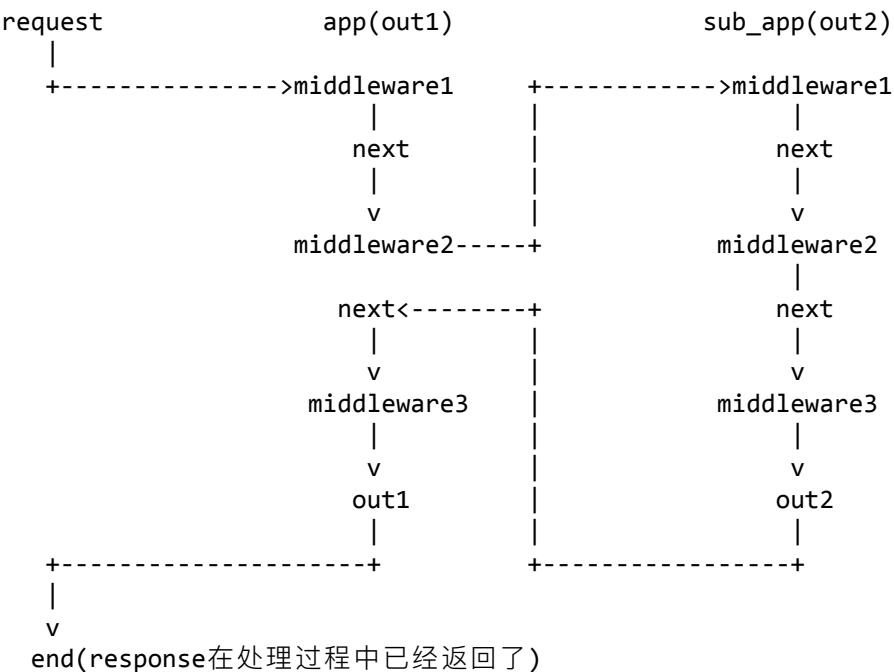
我们再看看 Connect 是怎么实现 subapp 的，比较有趣。

什么是 subapp?

```
var sub_app = connect();
var app = connect();
```

```
app.use('/route1', sub_app);
// request path: '/route1/route2'
// 由app接收到请求后，切割 path为'/route2'转交给sub_app的处理逻辑处理
```

// 再由sub_app返回到app，由app继续向下执行处理逻辑
结合上面的函数画图



完成上面的 sub_app 只需要做到两点：

- 1. 从 app 的调用链进入到 sub_app 的调用链中；
- 2. 从 sub_app 的逻辑回到 app 的调用链中；

connect 在 handle 函数中的第三个参数 out 为这个特性实现提供可能。out 的特点是在 middlewares 链式调用完成以后调用。那么将 app 的 next 作为 sub_app 的 out 传入 sub_app 的 handle 中可以做到 sub_app 自己的业务逻辑处理完后调用 out，即处理权回到了本 app 的 next 手里。

上面图中的 sub_app.out2===app.next，所以能完成逻辑的交接和 sub app 调用。

Express

大家都知道 Express 是 Connect 的升级版。

Express 不只是 Connect 的升级版，它还封装了很多对象来方便业务逻辑处理。Express 里的 Router 是 Connect 的升级版。Express 大概可以分为几个模块

模块	描述
router	路由模块是 Connect 升级版
request	经过 Express 封装的 req 对象
response	经过 Express 封装的 res 对象
application	app 上面的各种默认设置

简要介绍一下每个模块

Router

在 Connect 中间件特性的基础上，加入了如下特性，是 Connect 的升级版

- 1. 正则匹配 route；
- 2. 进行将 http 的方法在 route 中分解开；

Request

在 Request 中集成了 `http.IncomingMessage`(可读流 + 事件), 并在其上增加了新的属性, 方便使用, 我们最常用的应该是 `req.param`。

Response

在 Response 中集成了 `http.ServerResponse`(可写流 + 事件), 并在其上增加了很多方便返回的函数, 有我们熟悉的 `res.json`、`res.render`、`res.redirect`、`res.sendFile` 等等。

我们可以拓展它写一个 `res.sendPersonInfoById`。

关于流的题外话: `req.pipe(res)` 的形式可以“完成发什么就返回什么”, 而 `req.pipe(mylogic).pipe(res)` 可以添加自己的逻辑, 我们的业务逻辑是把流读为 `String/Object` 再进行逻辑处理, 处理完再推送给另一个 `stream`, 有没有可能在流的层面进行逻辑解耦提供服务呢? 求大神解答了...至少这种写法在大流量、逻辑简单的情况下是有用的。

Application

除了上面的三个模块以外, 还需要有个地方存储整个 app 的属性、设置等。比较常用的是 `app.engine` 函数设置模板引擎。

Express 小结

Express 是一个中间件机制的 `httpServer` 框架, 它本身实现了中间件机制, 它也包含了中间件。比如 3.x 版本的 Express 本身自带 `bodyParser`、`cookieSession` 等中间件, 而在 4.x 中去掉了。包括 TJ 也写了很多中间件, 比如 `node-querystring`、`connect-redis` 等。

实现业务逻辑解耦时, 中间件是从纵向的方面进行的逻辑分解, 前面的中间件处理的结果可以给后面用, 比如 `bodyParser` 把解析 `body` 的结果放在 `req.body` 中, 后面的逻辑都可以从 `req.body` 中取值。由于中间件是顺序执行的, `errHandler` 一般都放在最后, 而 `log` 类的中间件则放在比较前面。

总结

Connect 用流程控制库的回调函数及中间件的思想来解耦回调逻辑; Koa 用 Generator 方法解决回调问题;

我们应该也可以用事件、Promise 的方式实现;

PS: 用事件来实现的话还挺期待的, 能形成网状的相互调用。