码农 the code maker

18

Node 生态系统：成长 · 协作 · 分享

异步的本质以及其他

指令式 Callback，函数式 Promise：一声叹息

用 Q 实现 Promise

Node 编码规范

朴灵：打破限制，从前端到全栈

TED 总策划 Chris Anderson：永不退休的人

算法的乐趣：从乐趣出发阐述算法

iTuring.cn

目　录

专题：Hello Node

人物

动手

践行

鲜阅

九卦

书榜

成书手记

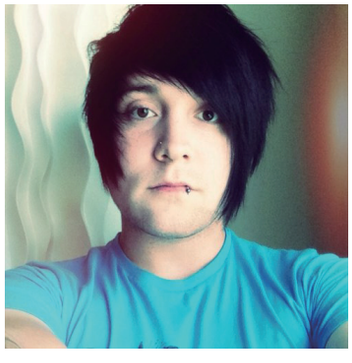
妙评

专题：Hello Node

Node生态系统：成长，协作，分享

作者 / Mike Cantelon

Node.js核心框架贡献者、Node社区活跃分子、资深培训师和演讲人。



作者 / T.J. Holowaychuk

开发建立了很多Node.js模块，包括流行的Express框架。

要从Node开发中获得最大收益，你得知道到哪里寻求帮助，以及如何跟社区中的其他人分享你的成果。

跟大多数开源社区一样，Node和相关项目的开发都是通过在线协作完成的。很多开发人员合作提交和审核代码，做项目文档，报告bug。当开发人员准备好发布Node的新版本时，会把它发布在Node的官网上。当一个值得发布的第三方模块被创建出来时，可以把它发布到npm库中，这样其他人安装起来更容易。在线资源为你提供了使用Node及相关项目所需的支持。

图1阐明了如何用在线资源做Node相关的开发、分发和支持。

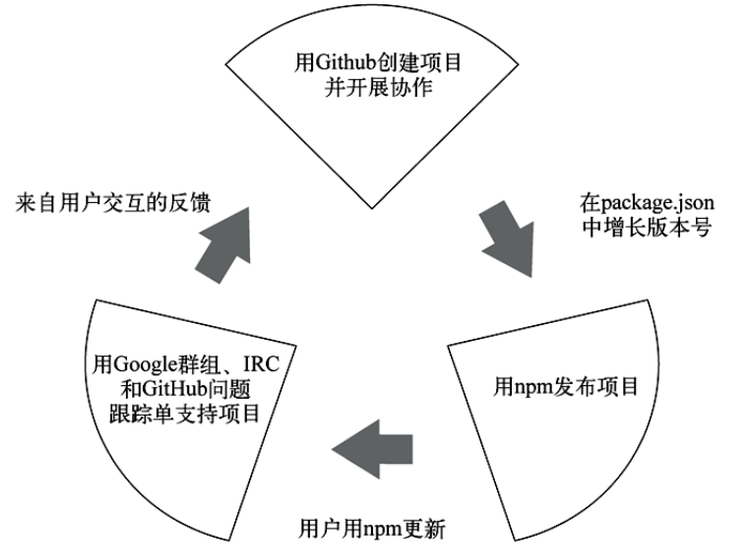


图1　Node相关的项目是通过在线协作创建的，一般是通过GitHub网站。然后发布到npm中，通过在线资源提供文档和支持

在协作之前，你很可能先需要支持，所以我们先来看一下网上有哪些地方可以为你提供帮助。

给Node开发人员的在线资源

Node的世界日新月异，所以只能在网上找到最新的参考资料。你将面对数不清的网站、在线讨论组和聊天室，并从中找到你需要的信息。

Node和模块的参考资料

表1列出了一些与Node相关的在线参考资料和资源。学习Node API和了解可用的第三方模块最实用的网站分别是Node.js和npm的首页。

表1　实用的Node.js参考资料

资　源	URL
Node.js首页	http://nodejs.org/
最新的Nodejs核心文挡	http://nodejs.org/api/
Node.js博客	http://blog.nodejs.org/
Node.js职位公告板	http://jobs.nodejs.org/
Node.js包管理器（npm）的首页	http://npmjs.org/

当你尝试用Node，或它的任何内置模块做些东西时，Node的首页是一个宝贵的资源。这个网站（如图14-2所示）有Node框架的完整文档，包括它的每个API。你总能在这个网站上找到最新版本的Node文档。官方博客还记录了Node的最新进展，分享重要的社区新闻。这里甚至还有个职位公告板。

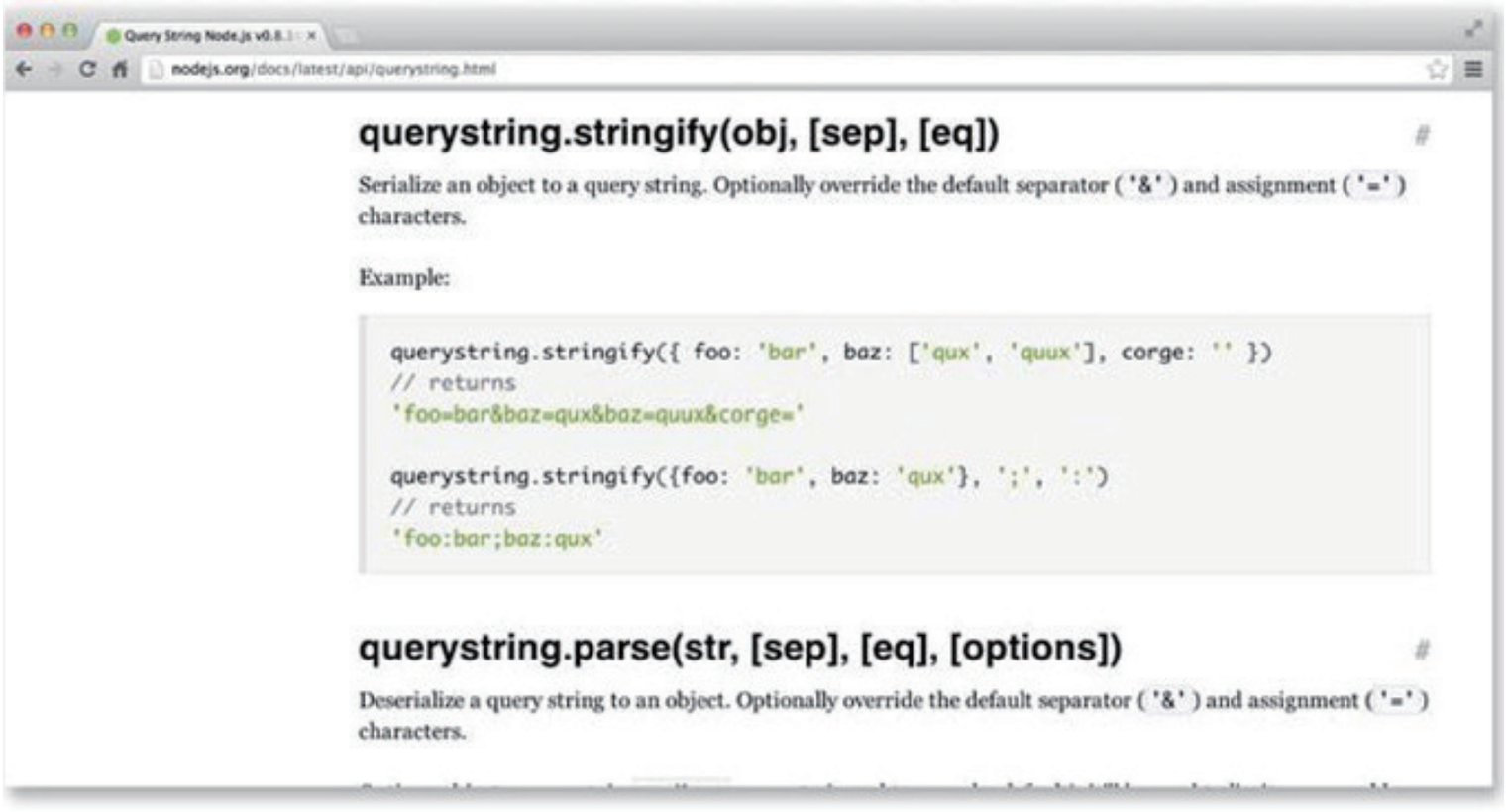


图2　除了提供与Node相关的实用资源的链接，nodejs.org还提供了Node各个版本API的权威文档

如果你要选购第三方的功能，应该去npm库的搜索页面。你可以用关键字在 npm中的上千个模块中进行搜索。如果你找到了一个你想要签出的模块，点击模块的名字进入它的详细页面，你会在那里看到指向模块项目主页的链接，如果有的话，以及依赖该模块的其他npm包，这个模块的依赖项，跟哪个版本的Node兼容，以及版权信息。

无论如何，这些网站可能无法回答你关于如何使用Node或其他第三方模块的所有问题。我们再去看一些可以给予你莫大帮助的其他地方。

Google群组

Node和一些流行的第三方模块，包括npm、Express、node-mongodb-native和Mongoose已经有Google群组了。

Google群组适合讨论困难的，或有深度的问题。比如说，如果你不知道如何用node-mongodb-native模块删除MongoDB文档，可以到node-mongodb-native的Google群组（https://groups.google.com/forum/?fromgroups#!forum/node-mongodb-native）中搜一下，看看其他人有没有相同的问题。如果没有人解决过你遇到的问题，接下来你应该加入Google群组提交你的问题。在Google群组上，你可以发长长的帖子，这对于复杂问题很有帮助，因为这样你才能充分地解释它。

这里没有包含与Node相关的所有Google群组的清单。可能会有些项目文档提到它们，但通常你只能在网上搜一下。比如说，你可以在Google上搜“模块名称node.js google group”，看看有没有这个第三方模块的Google 群组。

Google群组的缺点时你通常要等上几个小时，或几天才能看到反馈，这取决于Google群组。对于需要快速回复的简单问题，你应该考虑找个在线聊天室，通常能很快得到答案。

IRC

互联网中继聊天（IRC）的创建可以回溯到1988年，尽管有人觉得这是个老古董，但它依然生机勃发，并且如果你想问开源软件方面的简单问题，它是得到答案的最佳在线途径。IRC聊天室被称为频道，Node和各种第三方模块都有自己的频道。你也找不到跟Node相关的IRC频道的清单，但第三方模块有时会在它们的文档中提到自己的IRC频道。

要在IRC上得到答案，先连接到IRC网络（http://chatzilla.hacksrus.com/faq/#connect），进入相应的频道，发送你的问题。出于对频道中那些朋友的尊重，你最好事先在网上搜一下，别问那种一下子就能找到答案的问题。

如果你刚接触IRC，最容易的连接办法是用基于Web的客户端。大部分与Node相关的IRC频道都在Freenode上，这个IRC网络有个Web客户端

http://webchat.freenode.net/。要加入频道，在连接表单中填上相应的名称。你不需要注册，并且你可以输入任何想要的昵称。(如果你选的名字已经被其他人占用了，你的昵称后面会加上一个下划线 (_) 以示区别)。

点击连接之后，你就能进入频道，跟房间中的其他用户一样出现在右侧栏的用户列表中。

GitHub 问题列表

如果是在 GitHub 上开发的项目，你还可以到项目的 GitHub 问题列表上找找问题和解决方案。要访问问题列表，先进入项目的 GitHub 主页，点击 Issues 标签栏。你可以用搜索框查找跟你的问题相关的问题。图 14-3 中是一个问题列表的示例。

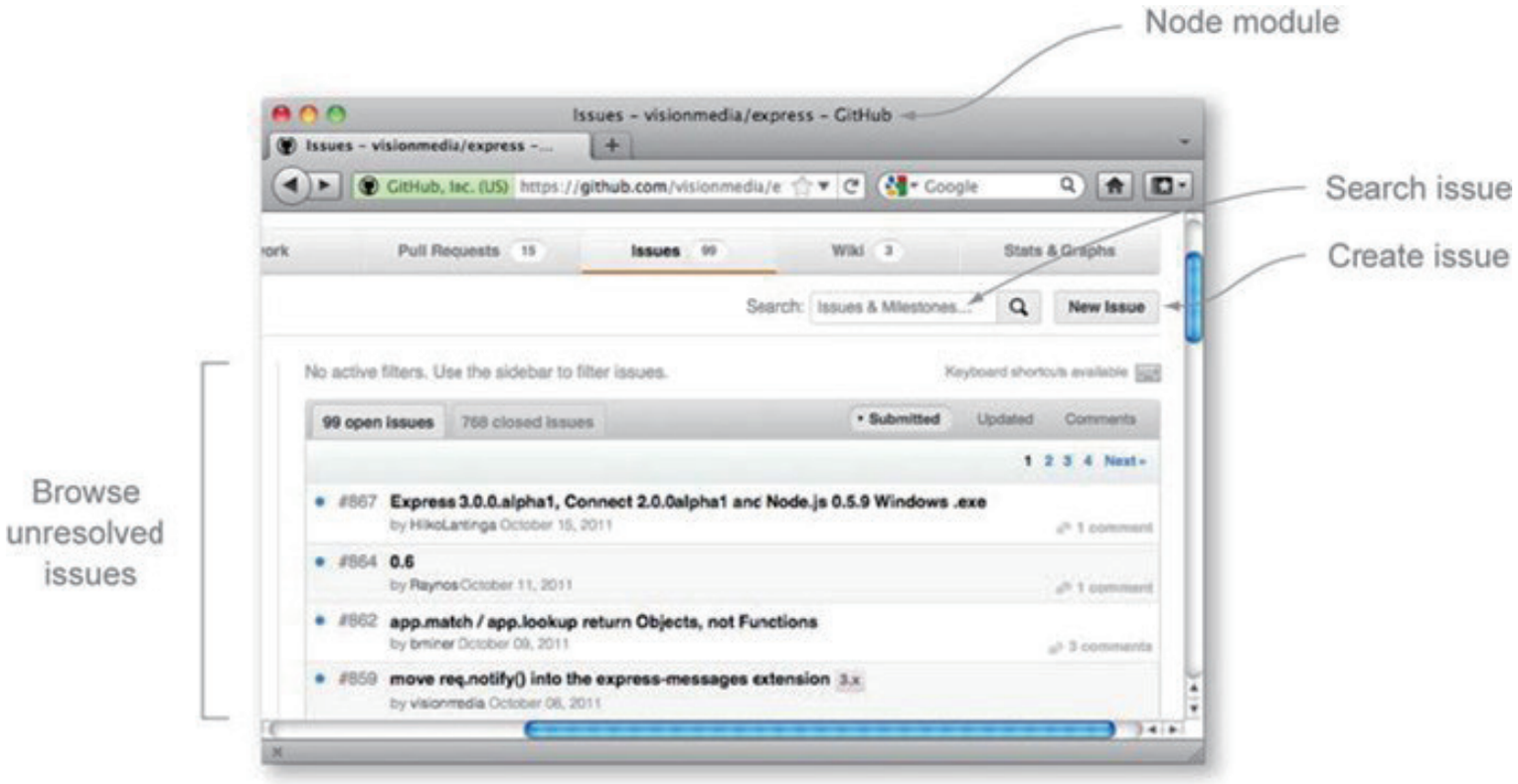


图 3　对于 GitHub 上的项目而言，如果你觉得自己发现了项目代码中的问题，问题列表可以帮到你

如果找不到可以解决你的问题的问题，并且你认为是项目代码中的bug导致了问题的出现，可以点击问题列表页面上的New Issue按钮提交这个 bug，项目的维护者可以在那个问题页面上回复你，或者解决这个问题，或者提出一些疑问以便了解问题出现的原因。

问题跟踪单不是支持论坛

在项目的GitHub问题跟踪单开一个普通的支持性问题可能不太合适，当然，这取决于具体项目。如果项目为用户设置了获取一般性支持的途径，比如Google群组，则通常是这种情况。你最好先看一下项目的README文件，看看它是否有关于一般性支持或问题的偏好说明。

现在你知道到哪里去提交项目的问题了，接下来我们要讨论GitHub的非支持性角色——它是大部分Node开发协作所倚重的网站。

GitHub

GitHub称得上是开源世界的重心，对Node开发人员来说至关重要。GitHub服务提供商提供了Git服务，这是一个强大的版本控制系统（VCS），你还可以通过Web界面轻松浏览Git库。开源项目可以免费使用GitHub。

GIT

Git VCS很受开源项目的青睐。它是一个分布式的版本控制系统（DVCS），跟Subversion和很多其他的VCS不同，你不一定非要连接到服务器上。Git是在2005年发布的，当时是受到了一个叫做BitKeeper的特有VCS的启发。BitKeeper的发布者授权Linux内核开发团队自由使用该软件，但因为怀疑该团队的成员试图探究BitKeeper的内部工作机制，随后又收回了授权。Linux的缔造者Linus Torvalds，决定创建一个功能相似的VCS，个把月后，Linux内核开发团队用上了Git。

除了提供Git访问，GitHub还为项目准备了问题跟踪、维基和Web页面服务等功能。因为npm库中的大多数Node项目都在GitHub上，所以了解 GitHub的使用对你充分利用Node开发很有帮助。在GitHub上很多事情做起来都很方便，浏览代码、检查未解决的bug，如果你想，还可以贡献解决问题的办法，编写文档。

GitHub的另一个用途是监测项目。受到监测的项目发生变化时会给你发送通知。监测项目的人数经常被用来评判项目的普及程度。

GitHub可能很强大，但你要怎么用它呢？接下来我们就要深入研究一下。

GitHub入门

当你有了一个基于Node的项目或第三方模块的想法时，很可能要在GitHub上设置个账号（如果你还没有），以便访问Git服务。设置好后可以添加项目。

因为GitHub要用Git，在继续GitHub之前，需要先配置Git。谢天谢地，GitHub分别为Mac、Windows和Linux准备了帮助页面（https://help.github.com/articles/set-up-git），帮你把Git配置好。Git配置好之后，你还需要配置GitHub，在它的网站上注册，并提供一个安全壳（SSH）公钥。SSH秘钥可以确保你跟GitHub交互的安全性。

注意，这些步骤只需要做一次，不是每次往GitHub中添加项目时都需要。

1. GIT配置和GITHUB注册

要用GitHub，得配置好你的Git工具。你需要用下面这两条命令提供你的姓名和邮箱地址：

```
git config --global user.name "Bob Dobbs"
git config --global user.email subgenius@example.com
```

接下来在GitHub网站上注册。访问注册页面（[https://github.com/signup/free](https://github.com/ signup/free)），填好表单，点击**创建账号**。

2. 给GITHUB一个SSH公钥

注册完之后，你需要给GitHub一个SSH公钥（https://help.github.com/articles/generating-ssh-keys）。你将用这个公钥对Git事务进行验证。按照下面这些步骤操作：

1. 在浏览器中访问https://github.com/settings/ssh；
2. 点击添加SSH秘钥。

到这里后，你需要做什么就取决于你使用的操作系统了。GitHub会检测出你的操作系统，并给出相应的指令。

添加一个项目到GitHub中

在GitHub上安顿好之后，你就可以往自己的账号下添加项目，提交内容了。

为此你需要先为项目创建一个GitHub库，稍后详细介绍。之后在你的本地机器上创建一个Git库，在把作品推送到GitHub库之前你就在那里完成它。图4列出了这个过程。

你还可以用GitHub的Web界面查看项目文件。

1. 创建一个GitHub库

在GitHub上创建库需要下面这些步骤：

1. 在Web浏览器中登入github.com；
2. 访问https://github.com/new；
3. 填好结果表单，描述你的库，然后点击**创建库**；
4. GitHub为你的项目创建了一个空白的Git库和一个问题列表；
5. GitHub会给出你用Git把代码推送到GitHub中所需的步骤。

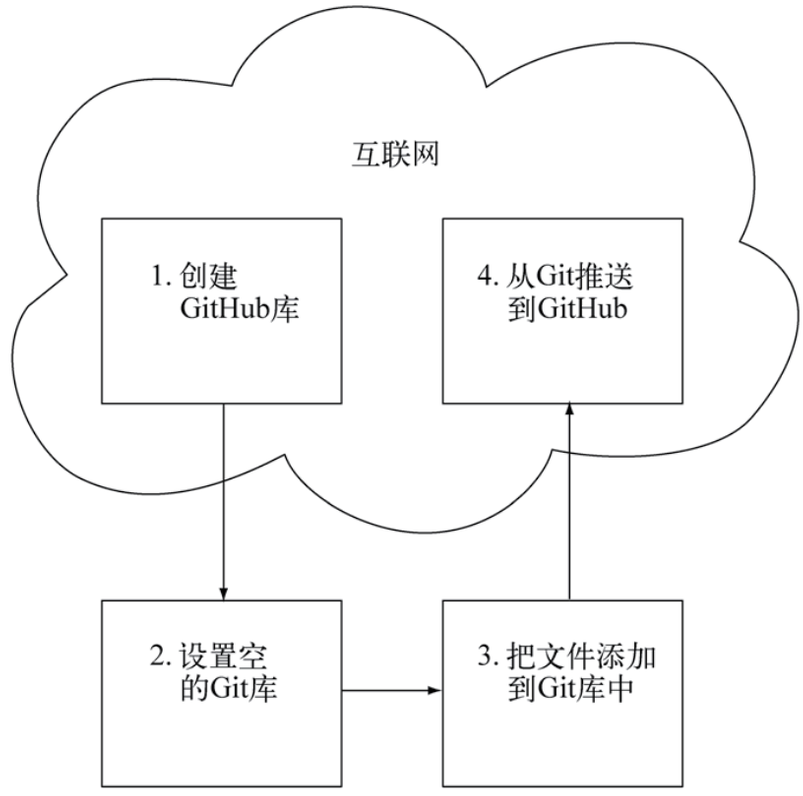


图4　把Node项目添加到GitHub中所需的步骤

理解这些步骤做了什么会对你有帮助的，所以我们会做一个例子来阐明Git最基本的用法。

2. 设置一个空白的Git库

要往GitHub中添加一个示例项目，需要先创建一个Node模块的例子。我们要创建一个能缩短URL的模块，`node-elf`。

先用下面的命令给项目创建一个临时目录：

```
mkdir -p ~/tmp/node-elf
cd ~/tmp/node-elf
```

为了把这个目录当作Git库，需要输入下面的命令（它会创建一个.git目录存放库的元数据）：

```
git init
```

3. 向Git库中添加一个文件

空白库设置好了，你可能想添加一些文件进去。作为例子，我们会添加一个包含URL缩短逻辑的文件。把下面的代码放到这个目录下名为index.js的文件里。

代码清单1　缩短URL的Node模块

```
exports.initPathData = function(pathData) {    // 由 shorten() 和
expand() 隐含调用的初始化函数
  pathData = (pathData) ? pathData : {};
  pathData.count = (pathData.count) ? pathData.count : 0;
  pathData.map   = (pathData.map) ? pathData.map : {};
}
exports.shorten = function(pathData, path) {  //接受一个“path”字符
串，并返回一个跟它对应的短化URL
  exports.initPathData(pathData);
  pathData.count++;
  pathData.map[pathData.count] = path;
  return pathData.count.toString(36);
}
exports.expand = function(pathData, shortened) {  //接受之前短化的
URL并返回展开的URL
  exports.initPathData(pathData);
  var pathIndex = parseInt(shortened, 36);
  return pathData.map[pathIndex];
}
```

接下来，让Git知道你想把这个文件放到库中。git的add命令跟其他的版本控制系统不一样。它不是把文件添加到库中，而是添加到Git的临时区。你可以把临时区看成是一个检查表，指明新添加的文件，或你修改过的文件，要把它们包含在库的下一次修订中：

```
git add index.js
```

这样Git就知道它应该跟踪这个文件。如果你想，还可以向临时区添加其他文件，但现在有这一个文件就够了。

要让Git知道你想在库中做个新修订，包含你放在临时区中修改过的文件，需要用commit命令。跟其他VCS中一样，commit命令可以用命令行选项-m指定一条消息，描述新修订所做的修改：

```
git commit -m "Added URL shortening functionality."
```

你本地机器中的库现在已经包含新的修订了。要查看库修改的清单，请输入下面的命令：

```
git log
```

4. 从Git推送到GitHub

如果这时候你的机器突然被雷劈了，那所有的工作就要丢了。为了防范这种突发性事件，并充分利用GitHub的Web界面提供的好处，你得把本地Git库中的修改送到你的GitHub账号下。但在做这件事情之前，要先让Git知道应该把修改送到哪里去。为此你需要添加一个Git远程库。它们被称为**remotes**。

下面这条命令就是往你的库上添加GitHub远程库的。用你的用户名替换掉username，注意node-elf.git，这是项目的名称：

```
git remote add origin git@github.com:username/node-elf.git
```

远程库添加好，现在你可以把修改发送给GitHub了。在Git的术语表中，发送修改被称为库**推送**。在下面的命令中，你告诉Git把你的工作推送到前面定义的远程库**origin中。所有Git库都可以有一或多个分支，从概念上区分库中的不同工作区。你要把工作推送到分支master**中：

```
git push -u origin master
```

推送命令中的选项-u告诉Git这个远程库是上游的远程库和分支。上游远程库是默认使用的远程库。

在做过一次带-u的推送后，将来再推送时用下面这条命令就行了，它更好记：

```
git push
```

如果到GitHub上去刷新你的库页面，应该能见到你的文件了。创建一个模块并把它放到GitHub上是重用它的快捷办法。比如说，如果你想在项目中使用你的样本模块，可以像下面这个例子一样输入这些命令：

```
mkdir ~/tmp/my_project/node_modules
cd ~/tmp/my_project/node_modules
git clone https://github.com/mcantelon/node-elf.git elf
cd ..
```

然后用`require('elf')`就可以使用这个模块了。注意，在克隆一个库时，命令行中的最后一个参数是你要把它克隆到哪里去的目录名。

你现在已经知道如何把项目添加到GitHub中了，包括如何在GitHub上创建一个库；如何在你的机器上创建Git库，并把文件添加到里面；以及如何把你的机器上的库推送到GitHub中。网上有很多优秀的资源可以支持你继续前行。如果你想寻求全面的Git使用指导，Scott Chacon，GitHub的创建者之一，写了一本非常全面的书，*Pro Git*，你可以买来看看，或者在线免费阅读（http://progit.org/）。如果你更喜欢手把手教学的

方式，Git 官方网站的文档页里列出了帮你起步的教程 (http://git-scm.com/documentation)。

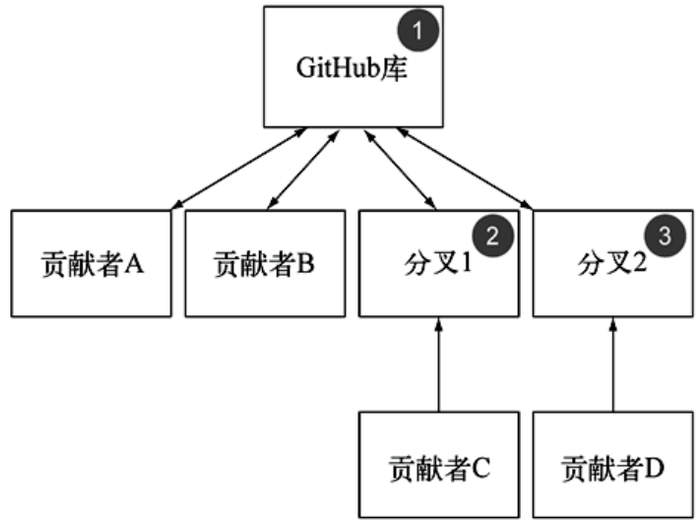
用 GitHub 协作

现在你已经知道如何从头开始创建 GitHub 库了，接下来我们看看如何用 GitHub 跟其他人协作。

假定你正在用一个第三方模块，并且遇到了 bug。你可能会去检查这个模块的源码并找出解决办法，然后你可能会给代码的作者发封邮件，介绍你的解决办法，并把修改过的文件作为附件。但这样那位作者还需要做些繁琐的工作。他 / 她只能比较你的文件和最新的代码，然后再把修订从你的文件中拿出来放到最新的代码中。但如果这位作者用了 GitHub，你可以克隆这个项目库，做些修改，然后通过 GitHub 的 bug 修订通知作者。GitHub 会在 Web 页面上展示你的代码和你复制的版本的差异，并且如果 bug 修订可以接受的话，只需点击一次鼠标就能把你的修订合并到最新的代码中。

按 GitHub 的说法，复制一个库被称为分叉 (forking)。对项目分叉后，你可以在你的副本上做任何事情，不用担心对原始库造成影响。分叉不需要得到原作者的许可：任何人都可以分叉任何项目，并把他们的贡献提交回原始项目中。原作者可能不会认可你的贡献，但你仍然可以拥有你自己的修订版，继续独立地维护和增强它。如果你的分叉越来越受欢迎，其他人可能也会分叉你的分叉，并贡献他们自己的成果。

你对分叉做出修改后，可以用一个**拉动 (pull) 请求**把这些修改提交给原作者，这是一个询问库作者是否拉动变化进来的消息。**拉动**，按 Git 的说法，意是时从分支中引入工作，并合并到自己的工作中。图 5 描绘了 GitHub 协作的场景。



1 一个贡献者A创建了一个GitHub库。贡献者A让朋友B参与到项目中来帮忙。

2 贡献者C决定往项目中添加一个功能，创建了分支1。当原始库被更新时，分支的贡献者们可以“拉动”变化，更新他们分支的代码。贡献者C曾试着让贡献者A和B接受他的功能，但他们没有，因为他们对项目有不同的定位，所以贡献者C的功能只能留在他自己的分支里。

3 贡献者D在这个Web框架里发现了一个bug，她决定花些时间来修订它，所以创建了分叉2。贡献者D的bug修订被贡献者A和B接受了，无论如何，她提交了“拉动请求”到原始库，结果经过贡献者A和B的评审后，她的代码就被“拉到”原始库中了。

图5　典型的GitHub开发场景

现在我们来看一个为了协作对GitHub库进行分叉的例子。这个过程如图6所示。

图6　通过分叉在GitHub上进行协作的过程

分叉把GitHub上的库复制到你的账号下，开启了协作的过程（称为分叉）（A）。然后你把分叉库克隆到自己的机器上（B），对它进行修改，提交这些修改（C），把你的工作推送会GitHub（D），并给原始库的所有者发送一个拉动请求，让他们考虑下你的修改（E）。如果他们想把你的修改纳入他们的库中，他们就会认可你的拉动请求。

比如说你想分叉的node-elf库，添加代码输出模块的版本号。这样模块的用户就可以肯定他们用的是正确的版本了。

首先登入GitHub，进入这个库的主页：https://github.com/mcantelon/node-elf。点击页面上的分叉（Fork）按钮复制该库。结果页面跟原始库的页面类似，不过在库名下有类似“forked from mcantelon/node-elf”的说明。

分叉后，接下来是把库克隆到你的机器上，进行修改，把修改推送给GitHub。下面的命令会对node-elf库做这些操作：

```
mkdir -p ~/tmp/forktest
cd ~/tmp/forktest
git clone git@github.com:chickentown/node-elf.git
cd node-elf
echo "exports.version = '0.0.2';" >> index.js
git add index.js
git commit -m "Added specification of module version."
git push origin master
```

完成修改的推送后，在分叉库的页面上点击拉动请求（Pull Request），输入标题和消息主体，描述你的修改。点击发送拉动请求（Send Pull Request）。图7是包含常见内容的截屏。

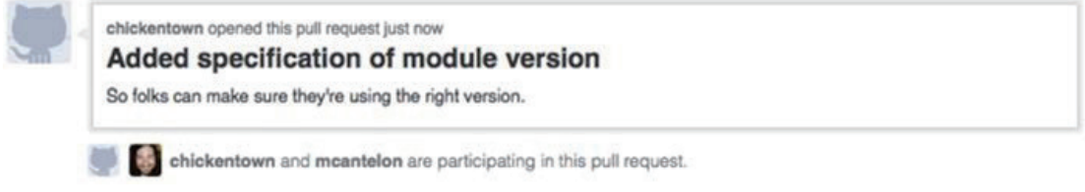


图7　GitHub拉动请求的细节

然后拉动请求会被添加到原始库的问题列表上。原始库的所有者可以评审你的修改，点击合并拉动请求（Merge Pull Request）引入它们，输入一条提交消息，点击确认合并（Confirm Merge）。然后这个问题就被自动关闭了。

在你跟别人合作创建出一个优秀的模块后，接下来就要把它推向全世界。最好的办法是把它添加到npm库中。

为npm库做贡献

假定你这个URL短化的模块已经做了一段时间了，你觉得其他Node用户应该也能用到它。为了推广它，你可以在Node相关的Google群组上发帖，介绍它的功能。但这样你的受众群只是有限的一部分Node用户，并且在人们开始用上你的模块后，他们没办法了解模块的更新情况。

为了解决可发现和提供更新的问题，你可以把它发布到npm上。有了npm，你可以轻松定义项目的依赖项，在安装你的模块时把那些依赖项也给自动安装上。如果你创建了一个专门保存内容（比如博客文章）评论的模块，可能会引入一个处理评论数据到MongoDB存储的模块作为依赖项。或者一个提供命令行工具的模块，会有一个解析命令行参数的辅助模块作为依赖项。

看到这里，你已经用npm装过很多东西了，从测试框架到数据库驱动无所不包，但你还什么都没发布过。下面我们要介绍把作品发布到npm上所需的步骤：

1. 准备包；
2. 编写包规范；
3. 测试包；
4. 发布包。

我们从准备包开始。

准备包

你想跟人分享的任何Node模块都应该搭配上相关资源，比如文档、例子、测试和相关的命令行工具。模块还应该有一个README文件，提供充足的信息让用户能够快速入门。

包目录应该用子目录组织起来。表2列出了常规的子目录——bin、docs、example、lib和test——以及它们都用来做什么。

表2　Node项目中的常规子目录

目　录	用　途
bin	命令行脚本
docs	文档
example	程序的例子
lib	程序的核心功能
test	测试脚本及相关资源

包组织好之后，你应该写一个包规范以便准备好把它发布到npm上。

编写包规范

在把包发布到npm上时，需要包含一个机器可读的包规范文件。这个JSON文件的名称是package.json，其中有模块的相关信息，比如它的名称、描述、版本、依赖项，以及其他特性。Nodejitsu有一个很方便的网站，给出了一个package.json文件样本，当你把鼠标悬停在样本文件的任一部分上时还会显示对该部分内容的解释（http://package.json.nodejitsu.com/）。

在package.json文件中，只有名称和版本是必须的。其他都是可选内容，但有一些，如果定义了，会让你的模块可用性更强。比如说bin，如果定义了的话，npm就知道包中的哪些文件是命令行工具，并让它们全局可用。

下面是一个规范样本：

```
{
    "name": "elf"
  , "version": "0.0.1"
  , "description": "Toy URL shortener"
  , "author": "Mike Cantelon <mcantelon@example.com>"
  , "main": "index"
  , "engines": { "node": "0.4.x" }
}
```

要查看package.json可用选项的完整文档，可以用下面的命令：

```
npm help json
```

因为手工生成JSON并不比手工编码XML有趣多少，所以我们来看一些可以让这个过程更轻松的工具。ngen就是这样的工具，装上这个npm包后，会有一个名为ngen的命令行工具。问几个问题之后，ngen会生成一个 package.json文件。它还会生成npm包中通常会有的一些其他文件，比如 Readme.md文件。

你可以用下面的命令安装ngen：

```
npm install -g ngen
```

装上ngen后，你会有一个全局的ngen命令，如果你在项目的根目录下运行这个命令，它会问你一些与项目相关的问题，并生成一个package.json文件，以及编写Node包通常会有的其他文件。可能有些生成的文件你并不需要，可以删掉。生成的文件包括一个.gitignore文件，指定

一些不应该添加到Git库中的文件和目录。还有一个.npmignore文件，它的作用跟.gitignore文件差不多，让npm知道将包发布到npm上时应该忽略哪些文件。

这里有一个运行ngen命令时的输出样例：

```
Project name: elf
Enter your name: Mike Cantelon
Enter your email: mcantelon@gmail.com
Project description: URL shortening library
create : /Users/mike/programming/js/shorten/node_modules/.
gitignore create : /Users/mike/programming/js/shorten/node_
modules/.npmignore create : /Users/mike/programming/js/shorten/
node_modules/History.md create : /Users/mike/programming/js/
shorten/node_modules/index.js ...
```

生成package.json文件是向npm发布中最难的部分。这一步一完成，你就可以准备发布模块了。

测试和发布包

发布模块到npm上需要三步，我们会逐一介绍：

1. 在本地测试包的安装；
2. 如果你还没有，添加一个npm用户；
3. 把包发布到npm上。

1. 测试包的安装

在模块的根目录下使用npm的`link`命令可以在本地测试包。这个命令让你的包可以在你的机器上全局使用，Node可以像使用由npm安装的包那样使用它。

```
sudo npm link
```

现在你的项目被全局链接了，你可以在link命令后面放上包名把它装在一个单独的测试目录中：

```
npm link elf
```

包装好之后，在Node REPL中执行require函数引入这个模块来测试一下，像下面的代码这样。你应该能在结果中看到模块提供的变量或函数：

```
node
> require('elf');
{ version: '0.0.1',
  initPathData: [Function],
  shorten: [Function],
  expand: [Function] }
```

如果你的包通过了测试，并且你已经结束了它的开发工作，在模块的根目录下执行npm的unlink命令：

```
sudo npm unlink
```

之后你的模块就不再是全局可用的了，但稍后，在你完成模块到npm上的发布之后，你还可以用install命令像平常那样安装它。

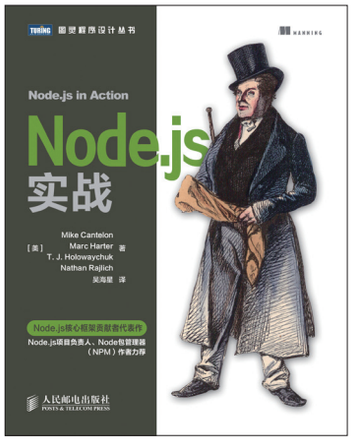
测试过npm包之后，接下来是创建npm发布账号，如果你之前没有设置过的话。

2. 添加npm用户

用下面的命令创建你自己的npm发布账号：

```
npm adduser
```

它会提示你输入用户名、邮箱地址和密码。如果账号添加成功，你不会看到错误消息。



《Node.js 实 战》是 Node.js 的实战教程，涵盖了为开发产品级Node应用程序所需要的一切特性、技巧以及相关理念。从搭建Node开发环境，到一些简单的演示程序，到开发复杂应用程序所必不可少的异步编程。书中还介绍了HTTP API的应用技巧等。本文节选自《Node.js实战》。

3. 发布到npm上

接下来是发布。输入下面的命令把你的包发布到npm上：

```
npm publish
```

你可能会看到警告，“经不安全的通道发送授权”，但如果你没看到其他错误，说明模块已经成功发布了。你可以用npm的`view`命令验证你的发布是否成功：

```
npm view elf description
```

如果你想引入一或多个私有库作为npm包的依赖项，可以。可能你想用一个实用的辅助函数模块，但不把它公开发布到npm上。

要添加私有依赖项，在通常放依赖模块名称的地方，你可以放任何跟其他依赖项的名称不同的名称。在通常放版本号的地方，放Git库的URL。下面的例子是package.json文件的一个片段，最后一个依赖项是私有库：

```
"dependencies" : {
          "optimist" : ">=0.1.3",
          "iniparser" : ">=1.0.1",
          "mingy": ">=0.1.2",
          "elf": "git://github.com/mcantelon/node-elf.git"
},
```

注意，私有模块也应该包含package.json文件。为了确保这些模块不会因为疏忽被发布出去，你可以在package.json文件中将`private`属性设为`true`：

```
 "private": true,
```

现在你已经掌握了设置、测试和发布模块到npm库上的知识。

小结

跟大多数成功的开源项目一样，Node有一个活跃的网上社区，也就是说你会找到充足的在线资源，并且能迅速地从在线参考资料、Google群组、IRC或GitHub问题列表中得到答案。

除了帮项目追踪bug，GitHub还提供了Git服务，并且开源用Web浏览器查看Git库中的代码。借助GitHub，其他开发人员如果想贡献bug修订、添加功能，或者把项目引向新的方向，他们很容易分叉你的开源代码。你也可以轻松地将提交到分叉上的修改带回到原始库中。

一旦Node项目进入了可以跟其他人分享的阶段，你就可以把它提交到Node包管理器的库中。把你的项目纳入npm中，其他人更容易找到它，并且如果你的项目是模块的话，纳入npm意味着模块安装起来更容易。

你知道如何得到帮助，在线协作，以及分享你的作品。Node之所以能变成现在这样，要归功于活跃的，大家都积极参与的社区。我们希望你也变得活跃起来，成为Node社区的一份子！ ■

专题：Hello Node

Node.js 中异步的本质以及其他

作者 / Azat Mardanov

Azat Mardanov 是一位有着 12 年开发经验的资深软件工程师，他曾涉足 web、移动、软件开发领域。他著有 9 本 JavaScript 和 Node.js 技术领域相关书籍，其中的 *Express.js Guide*, *Practical Node.js*，以及《JavaScript 快速全栈开发》在 Amazon.com 专业类别中成为了 #1 的畅销书。作为技术作者，其个人博客 webAppLog.com 一度成为谷歌搜索 “express.js tutorial” 结果中排名第一的教程站点。

> “不要害怕失败，你只需要做对一次。”
>
> ——安德鲁・豪斯顿

在这里我们引入了 Webapplog.com 上一些关于 Node.js 里异步的本质的文章，通过 Mocha 使用 TDD；介绍 Monk、Wintersmith 等框架和库。Webapplog.com 是一个公开的关于 Web 开发的博客。

Node 里的异步

非阻塞 I/O

与使用 Python 或者 Ruby 相比，使用 Node.js 最大的好处是有非阻塞 I/O 机制。为了说明这一点，我们以星巴克咖啡店里的队伍为例。假设每个人排队领取咖啡是一个任务，那么柜台后面所有的人和物，例如收银机、注册以及服务生，都像是服务器或者服务器应用。当我们要一杯普通的咖啡，比如 Pike Place、hot tea 或者 Earl Grey，服务生会制作它。整个队伍咖啡制作过程中都会等待，且每个人付适当的费用。

当然，上述那些饮品（众所周知是 “耗时的瓶颈”）制作上都很简单，只需要倒出液体，就可以做好。但是 choco-mocha-frappe-latte-soy-decafs 怎么办？如果队列中每个人都决定购买这种耗时的饮料呢？这个

队列会停滞不前，并且变得越来越长。咖啡店的经理必须增加更多的服务生，甚至自己开始收银。

这样不太好，对吧？但这个比喻生动地展现了除Node.js以外所有服务器端技术都会遭遇的情况，就像是真正的一家星巴克咖啡店。当你下单的时候，服务生把订单传递给别的雇员，你离开队列。队伍移动，处理器异步处理并且不会阻止队列。

这就是Node.js能够在性能和扩展上打败同类产品（也许底层的C++除外）的原因。使用Node.js，你将不需要大量的CPU和服务器来处理负载。

异步编码方式

异步需要程序员有不同于自己所熟悉的Python、PHP、C或Ruby的思路。因为如果忘记在执行的时候返回正确的表达式，这非常容易引发错误。

下面这个简单的例子描述了这种情况：

```
javascript
var test = function (callback) {
     return callback();
     console.log('test') //不会被打印
}

var test2 = function (callback) {
     callback();
     console.log('test2') //第三个被打印
}

test(function () {
     console.log('callback1') //第一个被打印
     test2(function () {
          console.log('callback2') //第二个被打印
     })
});
```

如果我们不使用return callback()，仅仅使用callback()，字符串test2将会打印出来，test不会。

```
callback1
callback2
test2
```

娱乐一下，我给callback2字符串添加了一个延迟setTimeout()，现在顺序改变了：

```
javascript
var test = function (callback) {
     return callback();
     console.log('test') //不会被打印
}

var test2 = function (callback) {
     callback();
     console.log('test2') //第二个被打印
}

test(function () {
     console.log('callback1') //第一个被打印
     test2(function () {
          setTimeout(function () {
               console.log('callback2') //第三个被打印
          }, 100)
     })
});
```

打印：

```
callback1
tes2
callback2
```

最后的例子展示了两个并行运行的独立函数。较快的函数将会比较慢的那个更早结束。再回到星巴克的例子，这意味着，你可能会比队伍里排在你前面的人更快得到自己的饮料。对用户更友好，对编程也更好！

使用Monk迁移MongoDB

最近，我们的一个高等用户抱怨他的Storify账户不能登录。我们检查了产品数据库，发现可能有人用这个账户的用户名和密码登录并且恶意删除了它。多亏了伟大的MongoHQ服务，我们得以在15分钟内恢复数据库。进行这个操作有两种选择：

1. Mongo shell脚本；
2. Node.js程序。

因为这个Storify用户账户删除了所有相关的对象，如验证、关系（关注、被关注）、喜欢、故事等，我们决定使用第二种方案。它工作得非常棒，这里有一个简洁版，你可以在MongoDB迁移中使用（也放在了gist.github.com/4516139里）。

现在我们来加载所有的模块：Monk、Progress、Async以及MongoDB：

```
JavaScript
var async = require('async');
var ProgressBar = require('progress');
var monk = require('monk');
var ObjectId = require('mongodb').ObjectID;
```

顺便说一下，Monk是由LeanBoost开发，它是Node.js里使用MongoDB的一个简易且对用户友好的封装。

Monk使用下面的连接字符串格式：

```
username:password@dbhost:port/database
```

创建下面的对象：

```
JavaScript
var dest = monk('localhost:27017/storify_localhost');
var backup = monk('localhost:27017/storify_backup');
```

我们需要知道要保存的对象的ID：

```
JavaScript
var userId = ObjectId(YOUR-OBJECT-ID);
```

这是人工输入的`restore()`函数，它可以通过指定特定的查询重复使用已经保存的对象（更多关于MongoDB查询的内容，请查看文章“Querying 20M-Record MongoDB Collection”）。如果想调用它，只需要把集合的名字以一个字符串传入，比如"stories"，并且从主对象里取值关联对象，比如`{userId:user.id}`。进度条可以在终端里形象展示当前进度：

```
JavaScript
var restore = function (collection, query, callback) {
    console.info('restoring from ' + collection);
    var q = query;
    backup.get(collection).count(q, function (e, n) {
        console.log('found ' + n + ' ' + collection);
        if (e) console.error(e);
        var bar = new ProgressBar('[:bar] :current/:total'
        + ':percent :etas'
        , { total: n - 1, width: 40 })
        var tick = function (e) {
            if (e) {
                console.error(e);
                bar.tick();
            }
            else {
                bar.tick();
            }
            if (bar.complete) {
                    console.log('restoring ' + collection + ' is
completed');
```

```
                callback();
            }
        };
        if (n > 0) {
            console.log('adding ' + n + ' ' + collection);
            backup.get(collection).find(q, {
                stream: true
                }).each(function (element) {
                dest.get(collection).insert(element, tick);
            });
        } else {
            callback();
        }
    });
}
```

现在我们使用async来调用上面提到的restore()函数:

```
JavaScript
async.series({
    restoreUser: function (callback) { // 导入用户元素
        backup.get('users').find({_id: userId}, {
            stream: true, limit: 1
        }).each(function (user) {
            dest.get('users').insert(user, function (e) {
                if (e) {
                    console.log(e);
                }
                else {
                            console.log('resored user: ' + user.
username);
                }
                callback();
            });
        });
    },
    restoreIdentity: function (callback) {
        restore('identities', {
```

```
            userid: userId
        }, callback);
    },
    restoreStories: function (callback) {
        restore('stories', {authorid: userId}, callback);
    }

}, function (e) {
    console.log();
    console.log();
    console.log('restoring is completed!');
    process.exit(1);
});
```

完整的代码在gist.github.com/4516139里，下面也是：

```
JavaScript
var async = require('async');
var ProgressBar = require('progress');
var monk = require('monk');
var ms = require('ms');
var ObjectId = require('mongodb').ObjectID;

var dest = monk('localhost:27017/storify_localhost');
var backup = monk('localhost:27017/storify_backup');

var userId = ObjectId(YOUR - OBJECT - ID);
// monk会自动分配，但是我们需要用它来进行查询

var restore = function (collection, query, callback) {
    console.info('restoring from ' + collection);
    var q = query;
    backup.get(collection).count(q, function (e, n) {
        console.log('found ' + n + ' ' + collection);
        if (e) console.error(e);
        var bar = new ProgressBar(
            '[:bar] :current/:total :percent :etas',
            { total: n - 1, width: 40 })
        var tick = function (e) {
```

```
            if (e) {
                console.error(e);
                bar.tick();
            }
            else {
                bar.tick();
            }
            if (bar.complete) {
                console.log();
                    console.log('restoring ' + collection + ' is
completed');
                callback();
            }
        };
        if (n > 0) {
            console.log('adding ' + n + ' ' + collection);
            backup.get(collection).find(q, { stream: true })
                .each(function (element) {
                dest.get(collection).insert(element, tick);
        });
        } else {
            callback();
        }
    });
}

async.series({
    restoreUser: function (callback) {// 导入用户元素
        backup.get('users').find({_id: userId}, {
            stream: true,limit: 1 })
            .each(function (user) {
            dest.get('users').insert(user, function (e) {
            if (e) {
                console.log(e);
            }
            else {
                console.log('resored user: ' + user.username);
            }
                callback();
```

```
            });
        });
    },

    restoreIdentity: function (callback) {
    restore('identities', {
            userid: userId
        }, callback);
    },

    restoreStories: function (callback) {
        restore('stories', {authorid: userId}, callback);
    }

}, function (e) {
    console.log();
    console.log('restoring is completed!');
    process.exit(1);
});
```

运行npm insall/npm update并且修改硬编码的数据库值来运行它。

在Node.js里使用Mocha实践TDD

谁需要使用测试驱动的开发

设想一下你要在一个已经存在的接口上实现一个复杂的功能，比如在评论上添加一个“like”（赞）按钮。在没有测试的情况下，必须人工创建用户，登录，创建文章，创建另一个用户，登录，赞这个文章。很令人厌烦吧？如果你需要重复这个步骤10到20次来发现和修复某些bug呢？如果你新添加的功能破坏了已经有的功能，而且在没有测试的情况下，6个月后才发现这一漏洞，怎么办呢？

不要为一次性的代码写测试，但是针对主代码，请养成测试驱动的习惯。只需要在开始的时候花点儿时间，你和你的团队稍后就可以节约很多时间并且在发布的时候更有自信。测试驱动开发真的是益处多多的好事情！

快速开始指南

请按照这个快速指南使用Mocha来设置测试驱动开发环境。

执行下面的命令，全局安装Mocha：

```
bash
$ sudo npm install -g mocha
```

我们还要使用另外两个库：LearnBoost的Superagent和expect.js。安装它们，在项目目录里使用NPM命令：

```
bash
$ npm install superagent
$ npm install expect.js
```

打开一个新的.js文件，输入：

```
javascript
var request = require('superagent');
var expect = require('expect.js');
```

目前我们已经载入了两个库。测试套件的结构看起来是这样的：

```
javascript
describe('Suite one', function () {
    it(function (done) {
    ...
    });
    it(function (done) { ...
    });
});
describe('Suite two', function () {
    it(function (done) { ...
    });
});
```

在这个封闭包里，我们写一个针对我们的服务器的请求，假设它在http://localhost:8080：

```
javascript
...
it(function (done) {
    request.post('localhost:8080').end(function (res) {
        //TODO 检查响应是否正确
    });
});
...
```

Expect提供了用来检查返回是否正确的便捷函数：

```
javascript
...
expect(res).to.exist;
expect(res.status).to.equal(200);
expect(res.body).to.contain('world');
...
```

最后，我们需要添加done()调用，告诉Mocha，这个异步测试已经完成。我们第一个测试的完整代码如下：

```
javascript
var request = require('superagent');
var expect = require('expect.js');

describe('Suite one', function () {
    it(function (done) {
        request.post('localhost:8080').end(function (res) {
            expect(res).to.exist;
            expect(res.status).to.equal(200);
            expect(res.body).to.contain('world');
            done();
        });
  });
```

如果我们想在请求前处理，可以添加**before**和**beforeEach**钩子，顾名思义，它们在每一个测试（或测试套件）运行前执行：

```
javascript
before(function () {
    //TODO 初始化数据库
});
describe('suite one ', function () {
    beforeEach(function () {
        //TODO 登录测试用户
    });
    it('test one', function (done) {
    ...
    });
});
```

请注意，before和beforeEach可以放在describe里，也可以放在外面。运行这个测试，简单执行：

```
bash
$ mocha test.sj
```

使用不同的报告类型：

```
bash
$ mocha test.js -R list
$ mocah test.js -R spec
```

Wintersmith：静态网站生成器

针对单页网站rapidprototypingwithjs.com，我使用Wintersmith来学习并快速启动了一些东西。Wintersmith是一个Node.js静态网站生成器。它的可扩展和方便的部署带给了我极大的震憾。另外，这里还有几个我最喜欢的工具，比如Markdown、Jade和Underscrore。

为什么选择静态网站生成器

这里有一个文章解释了为什么一般情况下使用静态网站生成器是好主意：An Introduction to Static Site Generators。它依据的是下面几样重要的东西。

模板

可以使用诸如Jade的模板引擎。Jade使用空格来组织级联元素，它的语法和Ruby on Rail's的Haml标记很像。

Markdown

我曾经从自己某本书的介绍章节复制markdown文本，并且没做任何修改直接使用它。Wintersmith使用了marked作为Markdown解析器。更多关于为什么Markdown是很棒的，请参考我的文章：Markdown Goodness。

简单的部署

所需的东西是HTML、CSS和JavaScript，所以你只需要使用一个FTP客户端上传它们，比如Panic的Transmit或Cyberduck。

基本服务

由于任何静态Web服务器都可以正常工作，没必要使用Heroku或者Nodejitsu私有云，甚至PHP/MySQL托管服务。

性能

没有数据库调用，没有服务器端API调用，也不会有CPU或内存过载。

《JavaScript快速全栈开发》涵盖JavaScript快速开发的多项前沿技术，是极其少见的前后端技术集大成之作。本书所涉技术包括Node.js、MongoDB、Twitter Bootstrap、LESS、jQuery、Parse.com、Heroku等，分三部分介绍如何用这些技术快速构建软件原型。第一部分是基础知识，让大家真正认识前后端及敏捷开发，并学会搭建本地及云环境。第二部分与第三部分分别介绍如何构建前端原型和后端原型。作者以前端组件开篇，通过为一个示例聊天应用Chat打造多个版本（Web/移动），将前端和后端结合在一起并给出应用部署方式。本文节选自《JavaScript快速全栈开发》。

灵活性

Wintersmith可以为内容和模板加载插件，你也可以写一个自己的插件。

开始使用Wintersmith

github.com/jnordberg/wintersmith这里有快速指南。

全局安装Wintersmith，使用-g参数和sudo来运行NPM：

```
bash
$ sudo npm install wintersmith -g
```

使用默认的博客模板来运行：

```
bash
$ wintersmith new <path>
```

或者使用一个空网站：

```
bash
$ wintersmith new <path> -template basic
```

或者使用快捷方式：

```
bash
$ wintersmith new <path> -T basic
```

类似Ruby on Rails支架，Wintersmith将生成一个带有**内容**和**模板**目录的基本框架。预览这个网站，运行下面的命令：

```
$ cd <path>
$ wintersmith preview
$ open http://localhost:8080
```

大多数的改变在预览模式下可以自动更新，config.json文件除外。

图像、CSS、JavaScript和其他文件会移动到**contents**文件夹。Wintersmith生成器用的是下面的逻辑：

1. 查找contents目录里的*.md文件；
2. 阅读metadata，例如模板名；
3. 处理每一个*.md文件里后缀名为*.jade的模板里的metadata。

当你创建好了自己的静态网站，运行：

```
bash
$ wintersmith build
```

其他静态网站生成器

这里还有一些别的Node.js静态网站生成器：

- Docpad
- Blacksmith
- Scotch
- Wheat
- Petrify

关于这些静态网站生成器更详细概述参见：Node.js Based Static Site Generators。

其他语言如Rails和PHP的生成器，请查看：“按Github关注数排序的静态网站生成器列表”和“mother of all site generator lists”。 ■

专题：Hello Node

Azat Mardanov：

现在是拥抱Node技术栈的最佳时机

Azat Mardanov是一位有着12年开发经验的资深软件工程师，他曾涉足web、移动、软件开发领域。他著有9本JavaScript和Node.js技术领域相关书籍，其中的*Express.js Guide*，*Practical Node.js*，以及《JavaScript快速全栈开发》在Amazon.com专业类别中成为了#1的畅销书。作为技术作者，其个人博客webAppLog.com一度成为谷歌搜索“express.js tutorial”结果中排名第一的教程站点。

Azat现在在DocuSign任高级工程师，他利用由Node.js, Express.js, Backbone.js, CoffeeScript, Jade, Stylus以及Redis组成的技术栈，带领团队重构了具有5000万用户的DocuSign。Azat还是科技聚会和编程马拉松活动的常客，他曾和FashionMetric.com团队一起在AngelHack活动上12次入围决赛。长期以来，他都是General Assembly、Hack Reactor、pariSOMA和Marakana等机构的讲师，其技术课程获得一致好评。另外，他还开发了很多Node.js开源项目（如ExpressWorks、mongoui以及HackHall等）。

问：Io.js另起炉灶，这将对Node造成什么影响？Io.js和Node分别具有什么优势？

在我看来，Joyent公司构建的Node需要加一把力了。他们需要缩短发布周期提高速度。他们也需要吸引新的代码贡献者。换句话说，为了提高整体轨迹和速度，Node必须要有所改变。

Io.js之所以被创造出来，主要是因为Joyent版Node决策过于缓慢。就目前来说，Io.js更加前沿，贡献者更多，发布周期也更短。Io.js已经到了版本1.6.2而Node仍然处在0.12.1版。

当然，版本号并不是成熟度的绝对指标。因为有一些项目可能只是提交了几个小补丁就跃进了主要版本号。

问：Node.js阵营的分裂你觉得会是一件好事吗？（早期硅谷不少成功的公司都是仙童公司分裂出去的。）

我认为分裂是一件好事。我们拥有的优秀分支越多，这些分支就会变得越好。可能这个例子不太合适，但是看看Linux的各种发行版：Debian，Fedora，openSUSE，Red Hat，Ubuntu等等。这些版本帮助传播了Linux基础的系统，彼此间的竞争也让它们越变越强。

但我并不是100%赞成这个论调。我更愿意集中精力在我自己的项目上，我建议大家远离政治和闹剧。因为只有时间能告诉我们未来会发生什么。分裂也可能是一件坏事，如果Io.js表现平庸的话，有可能会把人们从Node/Io.js阵营赶走。

但是，如果让我预测的话，我认为未来是很光明的，而现在就是拥抱Node/Io.js/JavaScript技术栈的最佳时机。

问：PayPal从Java迁移到Node非常成功。你认为Node会在后端取代Java吗？

是的。Node已经在取代Java了。

大多数Java应用都是很庞大的，所以很多公司把他们的巨型应用拆分成用Node实现的小型web服务。其他一些公司把Node用到前端层，这层的作用相当于Java或.NET的老API之间的中间人，优点是易于迭代（缩短的发布周期），易于构建（一种语言），而且更好扩展（应用更快）。

我推荐大家阅读这篇名为Monolithic Node.js的文章。

另外，企业和大公司需要的技术，其背后必须有一些声誉好的公司和工具。StrongLoop正在帮助大公司使用Node.js/Io.js。

初创企业已经爱上了Node/Io，因为便宜，可扩展，而且也更容易雇到程序员。

问：看起来Node.js比Python的框架Twisted火很多，为什么会这样？对于后端来说，Node.js和Python各有什么优缺点？

我并不是Python及其框架方面的专家。但是我从其他人那里听来的结果是这样：当你编写非阻塞的I/O代码时，Python的框架Twisted更加复杂。因为Python并不是从一开始就设计成非阻塞平台的。而Node从一开始就是为非阻塞和异步类型的代码和架构而设计的。

Node在NPM（Node包管理器）中的模块比Python多。Python标准库比Node更丰富。Python对于JavaScript开发者来说更难学，同时Node对于JavaScript开发者和前端开发者来说更容易学。

Python的结构很严谨，空格和缩进都是语言的一部分。

问：Node.JS+NoSQL的方式衍生出了很多便捷的工具让开发者能够快速响应前端需求，特别是像Parse这类的工具。你觉得在整个敏捷创新的过程中，未来在哪些方面可能会有更多这样的创新？

做预测很难，但是我认为移动开发会变得更加容易。只要看看Ionic和Swift就知道了！

在桌面端，我认为编程会更加聚焦在前端，比如Twitter Bootstrap，LESS这样的框架，而且类似Webflow，WordPress这样的服务会变得越来越流行。

后端所需要的编码越来越少。Node/Io/JS会一直笼络新生代后端开发者的心，同时也会占据PHP，.NET以及Java世界的“市场”。可能未来会产生面向后端的视觉拖放框架。

在部署层，Docker的方式变得越来越流行，因为你在生产环境开发（无差异）。

在2015年之初，我写下了一些预测。并不是所有都和科技有关。但是可能对你来说会很有趣：http://webapplog.com/my-predictions-for - 2015。

问：有哪些设计决策让JavaScript在开发和企业级应用方面一直保持长青？

使用JS，你可以直接解决问题。因为JavaScript是一种表现力很强的语言，这意味着你不用花很多时间来设置，换句话说，更小的额外负担。与之相比，Java的架构师仅仅在创造界面、类，构建梯阶、环境上就要花很多时间。总之：JavaScript更简单，用起来也更有乐趣。

使用Node/Io时，你可以在后端和数据库层使用JavaScript的API或语

言。所以在浏览器和服务器之间代码不需要上下文切换。如果没有上下文切换，那么生产力也会更高。

NPM能够帮助企业切换得更快是因为企业意识到NPM是由很多好用的优秀模块构成的。但是，主要原因还是可扩展性。利用非阻塞I/O，企业应用可以用更少的资源（服务器，内存，CPU）伺服更多页面。

问：作为一种编程语言JavaScript有些“不好的部分”，你认为基于JavaScript开发出的语言CoffeeScript如何？

CoffeeScript很不错，对于企业来说甚至更合适。ECMAScript 6标准从CoffeeScript那里借鉴了很多。

如果你需要一个好结构，那就用CoffeeScript吧！

你可以注册我的免费CoffeeScript线上课程：https://www.udemy.com/coffeescript。

问：JavaScript有很多框架和库，如何才能在众多资源中选择，然后建立属于自己的技术栈？

NPM（Node包管理器）发展很快，应该是你选择过程的最好起点。另外，我创造了“Node框架”网站，在那里我选出了最佳模块。你可以在“Node框架”上选择加入，查询以下简报：

- Node 周报
- JavaScript 周报
- Webapplog.com

问：如今，移动互联网已经变得越来越重要，在这种情况下前端工程师需要面临什么样的机会和挑战？

最开始的挑战在于平台分隔，开发者们试图通过HTML5解决问题。但

是HTML5的问题在于，相比于原生平台，HTML5的工具有些简陋。

现在，像Ionic这样的框架让你可以使用前端技术以及Angular和Backbone框架，同时你也可以利用一些原生的功能。我认为这是一个好的趋势。（Node和JS不仅被用在桌面开发中，也被用在机器人和嵌入式系统中。）

问：在读你的书之前，你建议读者先了解什么知识？在读完你的书后你建议他们读些什么或做些什么？

《JavaScript快速全栈开发》是一本Node栈的入门书，在读完之后我建议阅读*Practical Node.js*。这本书会为你完全综述整个开发过程的全貌：模板引擎、部署、代码组织、安全、数据库、等等。 ■

专题：Hello Node

指令式Callback，函数式Promise：
对Node.js的一声叹息

作者 / James Coglan

Faye, Vault, jstest, Canopy

作者，博客：jcoglan.com。

> 所谓promises，就是不会受不断变化的情况影响。
>
> —— Frank Underwood，‘House of Cards’

人们常说Javascript是'函数式'编程语言。而这仅仅因为函数是它的一等值，可函数式编程的很多其他特性，包括不可变数据，递归比循环更招人待见，代数类型系统，规避副作用等，它都不俱备。尽管把函数作为一等公民确实管用，也让码农可以根据自己的需要决定是否采用函数式的风格编程，但宣称JS是函数式的往往会让JS码农们忽略函数式编程的一个核心理念：用值编程。

'函数式编程'是一个使用不当的词，因为它会让人们以为这是'用函数编程'的意思，把它跟用对象编程相对比。但如果面向对象编程是把一切都当作对象，那函数式编程是把一切都当作值，不仅函数是值，而是一切都是值。这其中当然包括显而易见的数值、字符串、列表和其它数据，还包括我们这些OOP狗一般不会看成值的其它东西：IO操作和其它副作用，GUI事件流，null检查，甚至是函数调用序列的概念。如果你曾听说过'可编程的分号'[1] 这个短语，你应该就能明白我在说什么了。

[1] 指单子。In functional programming, a monad is a structure that represents computations. A type with a monad structure defines what it means to chain operations of that type together. This allows the programmer to build pipelines that process data in steps, in which each action is decorated with additional processing rules provided by the monad. As such, monads have been described as "programmable semicolons"; a semicolon is the operator used to chain together individual statements in many imperative programming languages, thus the expression implies that extra code will be executed between the statements in the pipeline. Monads have been also explained with a physical metaphor as assembly lines, where a conveyor belt transports data between functional units that transform it one step at a time. http://en.wikipedia.org/wiki/Monad(functionalprogramming)

最好的函数式编程是声明式的。在指令式编程中，我们编写指令序列来告诉机器如何做我们想做的事情。在函数式编程中，我们描述值之间的关系，告诉机器我们想计算什么，然后由机器自己产生指令序列完成计算。

用过excel的人都做过函数式编程：在其中通过建模把一个问题描绘成一个值图（如何从一个值推导出另一个）。当插入新值时，Excel负责找出它对图会产生什么影响，并帮你完成所有的更新，而无需你编写指令序列指导它完成这项工作。

有了这个定义做依据，我要指出node.js一个最大的设计失误，最起码我是这样认为的：在最初设计node.js时，在确定提供哪种方式的API式，它选择了基于callback，而不是基于promise。

> 所有人都在用 [callbacks]。如果你发布了一个返回promise的模块，没人会注意到它。人们甚至不会去用那样一个模块。
>
> 如果我要自己写个小库，用来跟Redis交互，并且这是它所做的最后一件事，我可以把传给我的callback转给Redis。而且当我们真地遇到callback hell之类的问题时，我会告诉你一个秘密：这里还有协同hell和单子hell，并且对于你所创建的任何抽象工具，只要你用得足够多，总会遇到某个hell。
>
> 在90%的情况下我们都有这种超级简单的接口，所以当我们需要做某件事的时候，只要小小的缩进一下，就可以搞定了。而在遇到复杂的情况时，你可以像npm里的其它827个模块一样，装上async。
>
> --Mikeal Rogers, LXJS 2012

Node宣称它的设计目标是让码农中的屌丝也能轻松写出反应迅速的并发网络程序，但我认为这个美好的愿望撞墙了。用Promise可以让运行时确定控制流程，而不是让码农绞尽脑汁地明确写出来，所以更容易构建出正确的、并发程度最高的程序。

编写正确的并发程序归根结底是要让尽可能多的操作同步进行，但各操作的执行顺序仍能正确无误。尽管Javascript是单线程的，但由于异步，我们仍然会遇到竞态条件：所有涉及到I/O操作的操作在等待callback时都要把CPU时间让给其他操作。多个并发操作都能访问内存中的相同数据，对数据库或DOM执行重叠的命令序列。借助promise，我们可以像excel那样用值之间的相互关系来描述问题，从而让工具帮你找出最优的解决方案，而不是你亲自去确定控制流。

我希望澄清大家对promise的误解，它的作用不仅是给基于callback的异步实现找一个语法更清晰的写法。promise以一种全新的方式对问题建模；它要比语法层面的变化更深入，实际上是在语义层上改变了解决问题的方式。

我在两年前曾写过一篇文章，promises是异步编程的单子。那篇文章的核心理念是**单子是组建函数的工具**，比如构建一个以上一个函数的输出作为下一个函数输入的管道。这是通过使用值之间的结构化关系来达成的，它的值和彼此之间的关系在这里仍要发挥重要作用。

我仍将借助Haskell的类型声明来阐明问题。在Haskell中，声明`foo::bar`表示“foo是类型为bar的值”。声明`foo :: Bar -> Qux`的意思是"foo是一个函数，以类型Bar的值为参数，返回一个类型为Qux的值"。如果输入/输出的确切类型无关紧要，可以用单个的小写字母表示，`foo :: a -> b`。如果foo的参数不止一个，可以加上更多的箭头，比如`foo :: a -> b -> c`表示foo有两个类型分别为a和b的参数，返回类型为c的值。

我们来看一个Node函数，就以`fs.readFile()`为例吧。这个函数的参数是一个String类型的路径名和一个callback函数，它没有任何返回值。callback函数有两个参数，Error（可能为null）和包含文件内容的Buffer，也是没有任何返回值。我们可以把readFile的类型表示为：

```
readFile :: String -> Callback -> ()
```

() 在 Haskell 中表示 null 类型。callback 本身是另一个函数，它的类型签名是：

```
Callback :: Error -> Buffer -> ()
```

把这些都放到一起，则可以说readFile以一个String和一个带着Buffer调用的函数为参数：

```
readFile :: String -> (Error -> Buffer -> ()) -> ()
```

好，现在请想象一下Node使用promises是什么情况。对于readFile而言，就是简单地接受一个String类型的值，并返回一个Buffer的promise值。

```
readFile :: String -> Promise Buffer
```

说得更概括一点，就是基于callback的函数接受一些输入和一个callback，然后用它的输出调用这个callback函数，而基于promise的函数接受输入，返回输出的promise值：

```
callback :: a -> (Error -> b -> ()) -> ()
promise :: a -> Promise b
```

基于callback的函数返回的那些null值就是基于callback编程之所以艰难的源头：基于callback的函数什么都不返回，所以难以把它们组装到一起。没有返回值的函数，执行它仅仅是因为它的副作用 －－ 没有返回值或副作用的函数就是个黑洞。所以用callback编程天生就是指令式的，是编写以副作用为主的过程的执行顺序，而不是像函数应用那样把输入映射到输出。是手工编排控制流，而不是通过定义值之间的关系来解决问题。因此使编写正确的并发程序变得艰难。

而基于promise的函数与之相反，你总能把函数的结果当作一个与时

间无关的值。在调用基于callback的函数时，在你调用这个函数和它的callback被调用之间要经过一段时间，而在这段时间里，程序中的任何地方都找不到表示结果的值。

```
fs.readFile('file1.txt',
  // 时光流逝...
  function(error, buffer) {
    // 现在，结果突然跌落在凡间
  }
);
```

从基于callback或事件的函数中得到结果基本上就意味着你“要在正确的时间正确的地点”出现。如果你是在事件已经被触发之后才把事件监听器绑定上去，或者把callback放错了位置，那上帝也罩不了你，你只能看着结果从眼前溜走。这对于用Node写HTTP服务器的人来说就像瘟疫一样。如果你搞错了控制流，那你的程序就只能崩溃。

而Promises与之相反，它不关心时间或者顺序。无论你在promise被resolve之前还是之后附上监听器，都没关系，你总能从中得到结果值。因此，返回promises的函数马上就能给你一个表示结果的值，你可以把它当作一等数据来用，也可以把它传给其它函数。不用等着callback，也不会错过任何事件。只要你手中握有promise，你就能从中得到结果值。

```
var p1 = new Promise();
p1.then(console.log);
p1.resolve(42);

var p2 = new Promise();
p2.resolve(2013);
p2.then(console.log);

// prints:
// 42
// 2013
```

所以尽管then()这个方法的名字让人觉得它跟某种顺序化的操作有关，并且那确实是它所承担的职责的副产品，但你真的可以把它当作unwrap来看待。promise是一个存放未知值的容器，而then的任务就是把这个值从promise中提取出来，把它交给另一个函数：从单子的角度来看就是bind函数。在上面的代码中，我们完全看不出来该值何时可用，或代码执行的顺序是什么，它只表达了某种依赖关系：要想在日志中输出某个值，那你必须先知道这个值是什么。程序执行的顺序是从这些依赖信息中推导出来的。两者的区别其实相当微妙，但随着我们讨论的不断深入，到文章末尾的lazy promises时，这个区别就会变得愈加明显。

到目前为止，你看到的都是些无足轻重的东西；一些彼此之间几乎没什么互动的小函数。为了让你了解promises为什么比callback更强大，我们来搞点更需要技巧性的把戏。假设我们要写段代码，用`fs.stat()`取得一堆文件的`mtimes`属性。如果这是异步的，我们只需要调用`paths.map(fs.stat)`，但既然跟异步函数映射难度较大，所以我们把async模块挖出来用一下。

```
var async = require('async'),
    fs    = require('fs');

var paths = ['file1.txt', 'file2.txt', 'file3.txt'];

async.map(paths, fs.stat, function(error, results) {
  // use the results
});
```

(哦，我知道fs的函数都有sync版本，但很多其它I/O操作都没有这种待遇。所以，请淡定地坐下来看我把戏法变完。)

一切都很美好，但是，新需求来了，我们还需要得到file1的size。只要再stat就可以了：

```
var paths = ['file1.txt', 'file2.txt', 'file3.txt'];

async.map(paths, fs.stat, function(error, results) {
  // use the results
});

fs.stat(paths[0], function(error, stat) {
  // use stat.size
});
```

需求满足了，但这个跟size有关的任务要等着前面整个列表中的文件都处理完才会开始。如果前面那个文件列表中的任何一项出错了，很不幸，我们根本就不可能得到第一个文件的size。这可就大大地坏了，所以，我们要试试别的办法：把第一个文件从文件列表中拿出来单独处理。

```
var paths = ['file1.txt', 'file2.txt', 'file3.txt'],
    file1 = paths.shift();

fs.stat(file1, function(error, stat) {
  // use stat.size
  async.map(paths, fs.stat, function(error, results) {
    results.unshift(stat);
    // use the results
  });
});
```

这样也行，但现在我们已经不能把这个程序称为并行化的了：它要用更长的时间，因为在处理完第一个文件之前，文件列表的请求处理得一直等着。之前它们还都是并发运行的。另外我们还不得不处理下数组，以便可以把第一个文件提出来做特别的处理。

Okay，最后的成功一击。我们知道需要得到所有文件的stats，每次命中一个文件，如果成功，则在第一个文件上做些工作，然后如果整个文件列表都成功了，则要在那个列表上做些工作。带着对问题中这些依赖关系的认识，用async把它表示出来。

```
var paths = ['file1.txt', 'file2.txt', 'file3.txt'],
    file1 = paths.shift();

async.parallel([
  function(callback) {
    fs.stat(file1, function(error, stat) {
      // use stat.size
      callback(error, stat);
    });
  },
  function(callback) {
    async.map(paths, fs.stat, callback);
  }
], function(error, results) {
  var stats = [results[0]].concat(results[1]);
  // use the stats
});
```

这就对了：每次一个文件，所有工作都是并行的，第一个文件的结果跟其他的没关系，而相关任务可以尽早执行。Mission accomplished!

好吧，实际上并不尽然。这个太丑了，并且当问题变得更加复杂后，这个显然不易于扩展。为了正确解决问题，要考虑很多东西，而且这个设计意图也不显眼，后期维护时很可能会把它破坏掉，后续任务跟如何完成所需工作的策略混杂在一起，而且我们不得不动用一些比较复杂的数组分割操作来应对这个特殊状况。啊哦！

这些问题的根源都在于我们用控制流作为解决办法的主体，如果用数据间的依赖关系，就不会这样了。我们的思路不是“要运行这个任务，我需要这个数据”，没有把找出最优路径的工作交给运行时，而是明确地向运行时指出哪些应该并行，哪些应该顺行，所以我们得到了一个特别脆弱的解决方案。

那promises怎么帮你脱离困境？嗯，首先要有能返回promises的文件

系统函数，用callback做参数的那套东西不行。但在这里我们不要手工打造一套文件系统函数，通过元编程作个能转换一切函数的东西就行。比如，它应该接受类型为：

```
String -> (Error -> Stat -> ()) -> ()
```

的函数，并返回：

```
String -> Promise Stat
```

下面就是这样一个函数：

```
// promisify :: (a -> (Error -> b -> ()) -> ()) -> (a -> Promise b)
var promisify = function(fn, receiver) {
  return function() {
    var slice   = Array.prototype.slice,
        args    = slice.call(arguments, 0, fn.length - 1),
        promise = new Promise();

    args.push(function() {
      var results = slice.call(arguments),
          error   = results.shift();

      if (error) promise.reject(error);
      else promise.resolve.apply(promise, results);
    });

    fn.apply(receiver, args);
    return promise;
  };
};
```

(这不是特别通用，但对我们来说够了.)

现在我们可以对问题重新建模。我们需要做的全部工作基本就是将一个路径列表映射到一个stats的promises列表上：

```
var fs_stat = promisify(fs.stat);

var paths = ['file1.txt', 'file2.txt', 'file3.txt'];

// [String] -> [Promise Stat]
var statsPromises = paths.map(fs_stat);
```

这已经是付利息了：在用 async.map() 时，在整个列表处理完之前你拿不到任何数据，而用上promises的列表之后，你可以径直挑出第一个文件的stat做些处理：

```
statsPromises[0].then(function(stat) { /* use stat.size */ });
```

所以在用上promise值后，我们已经解决了大部分问题：所有文件的stat都是并发进行的，并且访问所有文件的stat都和其他的无关，可以从数组中直接挑我们想要的任何一个，不止是第一个了。在前面那个方案中，我们必须在代码里明确写明要处理第一个文件，想换文件时改起来不是那么容易，但用promises列表就容易多了。

谜底还没有完全揭晓，在得到所有的stat结果之后，我们该做什么？在之前的程序中，我们最终得到的是一个Stat对象的列表，而现在我们得到的是一个Promise Stat 对象的列表。我们想等着所有这些promises都被兑现（resolve），然后生出一个包含所有stats的列表。换句话说，我们想把一个promises列表变成一个列表的promise。

闲言少叙，我们现在就给这个列表加上promise方法，那这个包含promises的列表就会变成一个promise，当它所包含的所有元素都兑现后，它也就兑现了。

```
// list :: [Promise a] -> Promise [a]
var list = function(promises) {
  var listPromise = new Promise();
  for (var k in listPromise) promises[k] = listPromise[k];

  var results = [], done = 0;

  promises.forEach(function(promise, i) {
    promise.then(function(result) {
      results[i] = result;
      done += 1;
      if (done === promises.length) promises.resolve(results);
    }, function(error) {
      promises.reject(error);
    });
  });

  if (promises.length === 0) promises.resolve(results);
  return promises;
};
```

(这个函数跟 jQuery.when() 类似，以一个promises列表为参数，返回一个新的promise，当参数中的所有promises都兑现后，这个新的promise就兑现了.)

只需把数组打包在promise里，我们就可以等着所有结果出来了：

```
list(statsPromises).then(function(stats) { /* use the stats */ });
```

我们最终的解决方案就被削减成了下面这样：

```
var fs_stat = promisify(fs.stat);

var paths = ['file1.txt', 'file2.txt', 'file3.txt'],
    statsPromises = list(paths.map(fs_stat));

statsPromises[0].then(function(stat) {
```

```
  // use stat.size
});

statsPromises.then(function(stats) {
  // use the stats
});
```

该方案的这种表示方式看起来要清楚得多了。借助一点通用的粘合剂（我们的promise辅助函数），以及已有的数组方法，我们就能用正确、有效、修改起来非常容易的办法解决这个问题。不需要async模块特制的集合方法，只是让promises和数组两者的思想各自保持独立，然后以非常强大的方式把它们整合到一起。

特别要注意这个程序是如何避免了跟并行或顺序相关的字眼出现。它只是说我们想做什么，然后说明任务之间的依赖关系是什么样的，其他的事情就交给promise类库去做了。

实际上，async集合模块中的很多东西都可以用promises列表上的操作轻松代替。前面已经看到map的例子了：

```
async.map(inputs, fn, function(error, results) {});
```

相当于：

```
list(inputs.map(promisify(fn))).then(
    function(results) {},
    function(error) {}
);
```

async.each() 跟 async.map() 实质上是一样的，只不过each()只是要执行效果，不关心返回值。完全可以用map()代替。

async.mapSeries()（如前所述，包括async.eachSeries()）相当

于在promises列表上调用reduce()。也就是说，你拿到输入列表，并用reduce产生一个promise，每个操作都依赖于之前的操作是否成功。我们来看一个例子：基于fs.rmdir()实现 rm -rf 。代码如下：

```
var dirs = ['a/b/c', 'a/b', 'a'];
async.mapSeries(dirs, fs.rmdir, function(error) {});
```

相当于：

```
var dirs     = ['a/b/c', 'a/b', 'a'],
    fs_rmdir = promisify(fs.rmdir);

var rm_rf = dirs.reduce(function(promise, path) {
  return promise.then(function() { return fs_rmdir(path) });
}, unit());

rm_rf.then(
    function() {},
    function(error) {}
);
```

其中的 unit() 只是为了产生一个已解决的promise已启动操作链（如果你知道monads，这就是给promises的return 函数）：

```
// unit :: a -> Promise a
var unit = function(a) {
  var promise = new Promise();
  promise.resolve(a);
  return promise;
};
```

用reduce()只是取出路径列表中的每对目录，用promise.then()根据上一步操作是否成功来执行路径删除操作。这样可以处理非空目录：如果上一个promise由于某种错误被rejecte了，操作链就会终止。用值之间的依赖关系限定执行顺序是函数式语言借助monads处理副作用的核心思想。

最后这个例子的代码比async版本繁琐得多，但不要被它骗了。关键是领会精神，要将彼此不相干的promise值和list操作结合起来组装程序，而不是依赖定制的流程控制库。如您所见，前一种方式写出来的程序更容易理解。

准确地讲，它之所以容易理解，是因为我们把一部分思考的过程交给机器了。如果用async模块，我们的思考过程是这样的：

- A.程序中这些任务间的依赖关系是这样的
- B.因此各操作的顺序必须是这样
- C.然后我们把B所表达的意思写成代码吧

用promises依赖图可以跳过步骤B。代码只要表达任务之间的依赖关系，然后让电脑去设定控制流。换种说法，callback用显式的控制流把很多细小的值粘到一起，而promises用显式的值间关系把很多细小的控制流粘到一起。Callback是指令式的，promises是函数式的。

如果最终没有一个完整的promises应用，并且是体现函数式编程核心思想 laziness的应用，我们对这个话题的讨论就不算完整。Haskell是一门懒语言，也就是说它不会把程序当成从头运行到尾的脚本，而是从定义程序输出的表达式开始，向stdio、数据库中写了什么等等，以此向后推导。它寻找最终表达式的输入所依赖的那些表达式，按图反向探索，直到计算出程序产生输出所需的一切。只有程序为完成任务而需要计算的东西才会计算。

解决计算机科学问题的最佳解决方案通常都是找到可以对其建模的准确数据结构。Javascript有一个与之非常相似的问题：模块加载。我们只想加载程序真正需要的模块，而且想尽可能高效地完成这个任务。

在 CommonJS 和 AMD出现之前，我们确实就已经有依赖的概念了，

脚本加载库有一大把。大多数的工作方式都跟前面的例子差不多，明确告诉脚本加载器哪些文件可以并行下载，哪些必须按顺序来。基本上都必须写出下载策略，要想做到正确高效，那是相当困难，跟简单描述脚本间的依赖关系，让加载器自己决定顺序比起来简直太坑人了。

接下来开始介绍LazyPromise的概念。这是一个promise对象，其中会包含一个可能做异步工作的函数。这个函数只在调用promise的then()时才会被调用一次：即只在需要它的结果时才开始计算它。这是通过重写then()实现的，如果工作还没开始，就启动它。

```
var Promise = require('rsvp').Promise,
    util    = require('util');

var LazyPromise = function(factory) {
  this._factory = factory;
  this._started = false;
};
util.inherits(LazyPromise, Promise);

LazyPromise.prototype.then = function() {
  if (!this._started) {
    this._started = true;
    var self = this;

    this._factory(function(error, result) {
      if (error) self.reject(error);
      else self.resolve(result);
    });
  }
  return Promise.prototype.then.apply(this, arguments);
};
```

比如下面这个程序，它什么也不做：因为我们根本没要过promise的结果，所以不用干活：

```
var delayed = new LazyPromise(function(callback) {
  console.log('Started');
  setTimeout(function() {
    console.log('Done');
    callback(null, 42);
  }, 1000);
});
```

但如果加上下面这行，程序就会输出Started，过了一秒后，在输出Done和42:

```
delayed.then(console.log);
```

但既然这个工作只做一次，调用then()会多次输出结构，但并不会每次都执行任务:

```
delayed.then(console.log);
delayed.then(console.log);
delayed.then(console.log);

// prints:
// Started
// -- 1 second delay --
// Done
// 42
// 42
// 42
```

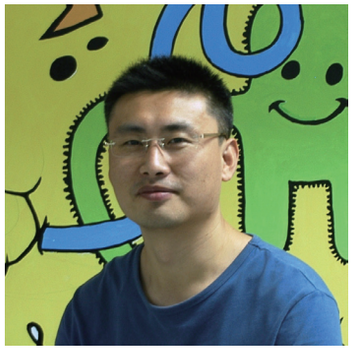
用这个非常简单的通用抽象，我们可以随时搭建一个优化模块系统。假定我们要像下面这样创建一堆模块：每个模块都有一个名字，一个依赖模块列表，以及一个传入依赖项，返回模块API的工厂函数。跟AMD的工作方式非常像。

```
var A = new Module('A', [], function() {
  return {
```

```
    logBase: function(x, y) {
      return Math.log(x) / Math.log(y);
    }
  };
});

var B = new Module('B', [A], function(a) {
  return {
    doMath: function(x, y) {
      return 'B result is: ' + a.logBase(x, y);
    }
  };
});

var C = new Module('C', [A], function(a) {
  return {
    doMath: function(x, y) {
      return 'C result is: ' + a.logBase(y, x);
    }
  };
});

var D = new Module('D', [B, C], function(b, c) {
  return {
    run: function(x, y) {
      console.log(b.doMath(x, y));
      console.log(c.doMath(x, y));
    }
  };
});
```

这里出了一个钻石的形状：D依赖于B和C，而它们每个都依赖于A。也就是说我们可以加载A，然后并行加载B和C，两个都到位后加载D。但是，我们希望工具能自己找出这个顺序，而不是由我们自己写出来。

这很容易实现，我们把模块当作LazyPromise的子类型来建模。它的工厂只要用我们前面那个list promise辅助函数得到依赖项的值，然后再经过一段模拟的加载时间后用那些依赖项构建模块。



译者 / 吴海星

2001 年毕业于南京理工大学，现任北京北控文化体育有限公司技术总监。熟悉 Java 及 Scala 语言，熟悉 web 应用系统开发，熟悉 IC 卡应用系统建设，目前主要对商业智能方面的内容感兴趣。译有《Node.js 实战》，《Node 与 Express 开发》，《JavaScript 编程实战》等书。图灵社区 ID: 海兴

```
var DELAY = 1000;

var Module = function(name, deps, factory) {
  this._factory = function(callback) {
    list(deps).then(function(apis) {
      console.log('-- module LOAD: ' + name);
      setTimeout(function() {
        console.log('-- module done: ' + name);
        var api = factory.apply(this, apis);
        callback(null, api);
      }, DELAY);
    });
  };
};
util.inherits(Module, LazyPromise);
```

因为 Module 是 LazyPromise, 只是像上面那样定义模块不会加载。我们只在需要用这些模块的时候加载它们:

```
D.then(function(d) { d.run(1000, 2) });

// prints:
//
// -- module LOAD: A
// -- module done: A
// -- module LOAD: B
// -- module LOAD: C
// -- module done: B
// -- module done: C
// -- module LOAD: D
// -- module done: D
// B result is: 9.965784284662087
// C result is: 0.10034333188799373
```

如上所示，最先加载的是A，完成后同时开始下载B和C，在两个都完成后加载D，跟我们想的一样。如果调用`C.then(function() {})`，那就只会加载A和C；不在依赖关系图中的模块不会加载。

所以我们几乎没怎么写代码就创建了一个正确的优化模块加载器，只要用lazy promises的图就行了。我们用函数式编程中值间关系的方式代替了显式声明控制流的方式，比我们自己写控制流容易得多。对于任何一个非循环得依赖关系图，这个库都能用来替你优化控制流。

这就是promises真正强大的地方。它不仅能在语法层面上规避缩进金字塔，还能让你在更高层次上对问题建模，而把底层工作交给工具完成。真的，那应该是我们所有码农对我们的软件提出的要求。如果Node真的想让并发编程更容易，他们应该再好好看看promises。■

阅读图灵社区原文

英文原文：Callbacks are imperative, promises are functional: Node's biggest missed opportunity

专题：Hello Node

用Q实现Promise：
Callbacks之外的另一种选择

作者 / Marc Harter

《Node.js实战》作者，Node.js核心框架贡献者。

怎么写异步代码？相对原始的callbacks而言，promises无疑是更好的选择。可掌握promises的概念及其用法可能不太容易，而且很有可能你已经放弃它了。但经过一大波码农的努力，promise的美终于以一种可互操、可验证的方式现于世间。这一努力的结果就是Promises/A+规范，它以自己的方式影响了各种promises库，甚至DOM。

扯了这么多，promises到底是什么？写Node程序时它能帮上什么忙？

Promises是一个。。。抽象

我们先来聊聊promise的行为模式，让你对他是什么，能怎么用他有个直观的感受。在本文的后半段，我们会以Q为例讲一下在程序里怎么创建和使用promise。

那promise究竟是什么呢？请看定义：

> promise是对异步编程的一种抽象。它是一个代理对象，代表一个必须进行异步处理的函数返回的值或抛出的异常。
>
> —— Kris Kowal on JSJ

callback 是编写 Javascript 异步代码最最最简单的机制。可用这种原始的 callback 必须以牺牲控制流、异常处理和函数语义为代价，而我们在同步代码中已经习惯了它们的存在，不适应！ Promises 能带它们回来。

promise 对象的核心部件是它的 then 方法。我们可以用这个方法从异步操作中得到返回值 (传说中的履约值)，或抛出的异常 (传说中的拒绝的理由)。then 方法有两个可选的参数，都是 callback 函数，分别是 `onFulfilled` 和 `onRejected`:

```
var promise = doSomethingAync()
promise.then(onFulfilled, onRejected)
```

promise 被解决时 (异步处理已经完成) 会调用 `onFulfilled` 和 `onRejected`。因为只会有一种结果，所以这两个函数中仅有一个会被触发。

从 Callbacks 到 promises

看过这个 promises 的基础知识后，我们再来看一个经典的异步 Node callback:

```
readFile(function (err, data) {
  if (err) return console.error(err)
  console.log(data)
})
```

如果函数 `readFile` 返回的是 promise，我们可以这样写:

```
var promise = readFile()
promise.then(console.log, console.error)
```

乍一看这没什么实质性变化。但实际上现在我们得到了一个代表异步操作的值 (promise)。我们可以传递 promise，不管异步操作完成与否，

所有能访问到promise的代码都可以用then使用这个异步操作的处理结果。而且我们还得到保证，异步操作的结果不会发生某种变化，因为promise只会被解决一次（或履约，或被拒）。

> 把then当成对promise解包以得到异步操作结果（或异常）的函数对理解promise更有帮助，不要把它当成只是带两个callback（onFulfilled 和 onRejected）的普通函数。详情请见此文

promise的链接及内嵌

then方法返回的还是promise。

```
var promise = readFile()
var promise2 = promise.then(readAnotherFile, console.error)
```

这个promise表示onFulfilled或onRejected的返回结果。既然结果只能是其中之一，所以不管是什么结果，promise都会转发调用：

```
var promise = readFile()
var promise2 = promise.then(function (data) {
   return readAnotherFile() // if readFile was successful, let's
readAnotherFile
}, function (err) {
   console.error(err) // if readFile was unsuccessful, let's log
it but still readAnotherFile
  return readAnotherFile()
})
promise2.then(console.log, console.error) // the result of
readAnotherFile
```

因为 then 返回的是 promise，所以 promise 可以形成调用链，避免出现 callback 大坑：

```
readFile()
  .then(readAnotherFile)
  .then(doSomethingElse)
  .then(...)
```

如果非要保持闭包，promise 也可以嵌套：

```
readFile()
  .then(function (data) {
    return readAnotherFile().then(function () {
      // do something with `data`
    })
  })
```

Promise 与同步函数

Promises 有几种编写同步函数的办法。其中之一是用返回代替调用。在前面的例子中，返回 `readAnotherFile()` 是一个信号，表明在 `readFile` 完成之后做什么。

如果返回 promise，它会在异步操作完成后发信号给下一个 then。返回值并不是非 promise 不可，不管返回什么，都会传给下一个 onFulfilled 做参数：

```
readFile()
  .then(function (buf) {
    return JSON.parse(buf.toString())
  })
  .then(function (data) {
    // do something with `data`
  })
```

promise的错误处理

除了return，还可以用关键字throw和try/catch语法。这可以算是promises最强的一个特性了。下面我们来看一段同步代码：

```
try {
  doThis()
  doThat()
} catch (err) {
  console.error(err)
}
```

在上例中，如果doThis()或doThat()抛出了异常，异常会被捕获并输出错误日志。既然try/catch允许多个操作放到一起，我们就不用单独处理每个操作可能出现的错误。用promises的异步代码也可以这样：

```
doThisAsync()
  .then(doThatAsync)
  .then(null, console.error)
```

如果doThisAsync()没有成功，它的promise会被拒，处理链下一个then上的onRejected会被调用。在上例中就是函数console.error。而且跟 try/catch 一样，doThatAsync() 根本就不会被调用。对于原始的callback那种每一步里都要显式处理错误的方式而言，这是巨大的进步。

实际上它比这还要好！任何被抛出的异常，隐式的或显式的，then的回调函数中的也会处理：

```
doThisAsync()
  .then(function (data) {
    data.foo.baz = 'bar' // throws a ReferenceError as foo is not
defined
  })
  .then(null, console.error)
```

上例中抛出的ReferenceError会被处理链中下一个onRejected捕获。相当漂亮！当然，这对显式抛出的异常也有效：

```
doThisAsync()
  .then(function (data) {
    if (!data.baz) throw new Error('Expected baz to be there')
  })
  .then(null, console.error)
```

对错误处理的重要提示

我们在前面已经说过了，promises模拟了try/catch。在try/catch中，可以不对异常做显式的处理，屏蔽它：

```
try {
  throw new Error('never will know this happened')
} catch (e) {}
```

对promise来说也是如此：

```
readFile()
  .then(function (data) {
    throw new Error('never will know this happened')
  })
```

要处理被屏蔽的错误，可以在promise处理链的最后加一个.then(null, onRejected)：

```
readFile()
  .then(function (data) {
    throw new Error('now I know this happened')
  })
  .then(null, console.error)
```

各种函数库中还包括暴露被屏蔽错误的其他选项。比如Q中的done方法可以重新向上抛出错误。

promise的具体应用

前面的例子都是返回空方法，只是为了阐明Promises/A+中的then方法。接下来我们要看一些更具体的例子。

将callbacks 变成 promises

你可能在想promise最初是从哪蹦出来的。Promise/A+规范中没有规定创建promise的API，因为它不会影响互操作性。因此不同promise库的实现可能是不同的。我们的例子用的是Q（`npm install q`）。

Node 核心异步函数不会返回promises；它们采用了callbacks的方式。然而用Q可以很容易地让它们返回promises：

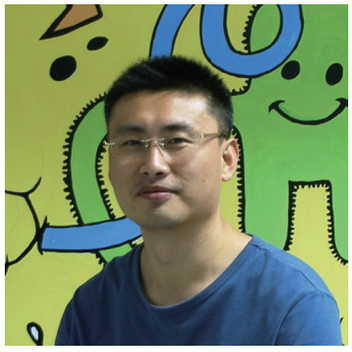
```
var fs_readFile = Q.denodify(fs.readFile)
var promise = fs_readFile('myfile.txt')
promise.then(console.log, console.error)
```

> Q提供了一些辅助函数，可以将Node和其他环境适配为promise可用的。请参见readme和API documentation了解详情。

创建原始的promise

用`Q.defer`可以手动创建promise。比如将`fs.readFile`手工封装成promise的（基本上就是`Q.denodify`做的事情）

```
function fs_readFile (file, encoding) {
  var deferred = Q.defer()
  fs.readFile(file, encoding, function (err, data) {
     if (err) deferred.reject(err) // rejects the promise with
`er` as the reason
```



译者 / 吴海星

2001 年毕业于南京理工大学，现任北京北控文化体育有限公司技术总监。熟悉 Java 及 Scala 语言，熟悉 web 应用系统开发，熟悉 IC 卡应用系统建设，目前主要对商业智能方面的内容感兴趣。译有《Node.js实战》，《Node 与 Express 开 发》，《JavaScript 编 程 实 战》等书。图灵社区 ID: 海兴

```
      else deferred.resolve(data) // fulfills the promise with
`data` as the value
  })
  return deferred.promise // the promise is returned
}
fs_readFile('myfile.txt').then(console.log, console.error)
```

做同时支持 callbacks 和 promises 的 APIs

我们已经见过两种将 callback 代码变成 promise 代码的办法了。其实还能做出同时提供 promise 和 callback 接口的 APIs。下面我们就把 `fs.readFile` 变成这样的 API:

```
function fs_readFile (file, encoding, callback) {
  var deferred = Q.defer()
  fs.readFile(function (err, data) {
      if (err) deferred.reject(err) // rejects the promise with
`er` as the reason
      else deferred.resolve(data) // fulfills the promise with
`data` as the value
  })
   return deferred.promise.nodeify(callback) // the promise is
returned
}
```

如果提供了 callback，当 promise 被拒或被解决时，会用标准 Node 风格的 (err, result) 参数调用它。

```
fs_readFile('myfile.txt', 'utf8', function (er, data) {
  // ...
})
```

用 promise 执行并行操作

我们前面聊的都是顺序的异步操作。对于并行操作，Q 提供了 Q.all 方法，它以一个 promises 数组作为输入，返回一个新的 promise。 在数组中

的所有操作都成功完成后，这个promise就会履约。如果任何一个操作失败，这个新的promise就会被拒。

```
var allPromise = Q.all([ fs_readFile('file1.txt'), fs_readFile
('file2.txt') ])
allPromise.then(console.log, console.error)
```

不得不强调一下，promise在模仿函数。函数只有一个返回值。当传给Q.all两个成功完成的promises时，调用onFulfilled只会有一个参数（一个包含两个结果的数组）。你可能会对此感到吃惊；然而跟同步保持一致是promise的一个重要保证。如果你想把结果展开成多个参数，可以用Q.spread。

让promise更具体

要想真正理解promise，最好的办法就是用一用。下面是几个帮你开始的主意：

1. 封装一些基本的Node流程，将callbacks 变成 promises
2. 重写一个async方法，变成使用promise的
3. 写一些递归使用promises的东西（目录树应该是个不错的开端）
4. 写一个过得去的 Promise A+实现

阅读图灵社区原文

英文原文：Promises in Node.js with Q - An Alternative to Callbacks

专题：Hello Node

Node编码规范（优秀是一种习惯）

作者 / 朴灵

真名田永强，文艺型码农，就职于阿里巴巴数据平台，资深工程师，Node.js布道者，写了多篇文章介绍Node.js的细节。活跃于CNode社区，是线下会议NodeParty的组织者和JSConf China（沪JS和京JS）的组织者之一。热爱开源，多个Node.js模块的作者。个人GitHub地址：http://github.com/JacksonTian。**叩首问路，码梦为生。**

根源

JavaScript作为一门编程语言，在语法上可谓是最为灵活的语言了。有人喜欢它的灵活，也有人讨厌它的混乱。无论它的灵活也好，混乱也罢，都离不开其诞生的历史。Brendan Eich在1995年里花了10天设计出了这门语言，其后微软在1996年也发布了支持JavaScript的浏览器IE 3.0。网景公司为了保护自己，在1996年11月将JavaScript提交给ECMA标准化组织，次年6月第一版标准发布，命名为ECMAScript，编号262。

早年的JavaScript编写十分混乱。它的灵活性和容忍度都非常高，使得开发者可以毫无顾忌地编码，最终导致它在一定程度上臭名昭著。在编码规范上，一个重要的人物是Douglas Crockford，他是JavaScript开发社区最知名的权威，是JSON、JSLint、JSMin和ADSafe之父，其中JSLint现在仍然是最重要的JavaScript质量检测工具。他出版的JavaScript: The Good Parts一书对于JavaScript社区影响深远。

通常，一门语言的发展要经历十多年的锤炼才能为大众所接受。由于历史原因，JavaScript在短短的时间内就被标准化定型，这样它的优点和缺点都暴露在大众之下。Douglas Crockford的JSLint和JavaScript: The Good Parts对JavaScript的贡献在于，他让我们能够甄别语言中的精华和糟粕，写出更好的代码。

与其他语言（比如Python或Ruby）的程序员相比，JavaScript程序员需要更多的自律才能够写出易读、易维护的代码。为避免这个问题，部分开发者选择TypeScript或CoffeeScript来编写应用。但我认为了解一门语言为何是当下这种情况是有必要的。编码规范的目的是在一定程度上约束程序员，使之能够在团队中易维护并且避免低级错误。

尽管JavaScript规范已经相当成熟，利用JSLint能够解决大部分问题，但是随着Node的流行，带来了一些新的变化，这些需要引起我们注意。本附录是在总结了JavaScript的编码规范的基础上，根据Node的特殊环境和社区的习惯进行改进而成。

编码规范

空格与格式

1.缩进

采用2个空格缩进，而不是tab缩进。 空格在编辑器中与字符是等宽的，而tab可能因编辑器的设置不同。2个空格会让代码看起来更紧凑、明快。

2.变量声明

永远用var声明变量，不加var时会将其变成全局变量，这样可能会意外污染上下文，或是被意外污染。 在ECMAScript 5的strict模式下，未声明的变量将会直接抛出ReferenceError异常。

需要说明的是，每行声明都应该带上var，而不是只有一个var，示例代码如下：

```
var assert = require('assert');
var fork = require('child_process').fork;
var net = require('net');
var EventEmitter = require('events').EventEmitter;
```

错误示例如下所示：

```
var assert = require('assert')
  , fork = require('child_process').fork
  , net = require('net')
  , EventEmitter = require('events').EventEmitter;
```

3. 空格

在操作符前后需要加空格，比如+、-、*、%、=等操作符前后都应该存在一个空格，示例如下：

```
var foo = 'bar' + baz;
```

错误的示例如下所示：

```
var foo='bar'+baz;
```

此外，在小括号前后应该存在空格，如：

```
if (true) {
  // some code
}
```

错误的示例如下所示：

```
if(true){
  // some code
}
```

4. 单双引号的使用

由于双引号在别的场景下使用较多，在Node中使用字符串时尽量使用单引号，这样无需转义，如：

```
var html = '<a href="http://cnodejs.org">CNode</a>';
```

而在JSON中，严格的规范是要求字符串用双引号，内容中出现双引号时，需要转义。

5.大括号的位置

一般情况下，大括号无需另起一行，如

```
if (true) {
  // some code
}
```

错误的示例如下：

```
if (true)
{
  // some code
}
```

6.逗号

逗号用于变量声明的分隔或是元素的分隔。如果逗号不在行结尾，前面需要一个空格。此外，逗号不允许出现在行首，比如：var foo = 'hello', bar = 'world'; // 或是var hello = { foo: 'hello', bar: 'world' }; // 或是 var world = ['hello', 'world'];错误示例如下：

```
var foo = 'hello'
  , bar = 'world';
// 或是
var hello = {foo: 'hello'
  , bar: 'world'
};
// 或是
var world = [
  'hello'
  , 'world'
];
```

7.分号

给表达式结尾添加分号。尽管JavaScript编译器会自动给行尾添加分号，但还是会带来一些误解，示例如下：

```
function add() {
  var a = 1, b = 2
  return
    a + b
}
```

将会得到undefined的返回值。因为自动加入分号后会变成如下的样子：

```
function add() {
  var a = 1, b = 2;
  return;
    a + b;
}
```

后续的a + b将不会执行。

而如下的代码：

```
x = y
(function () {
}())
```

执行时会得到：

```
x = y(function () {}())
```

由于自动添加分号可能带来未预期的结果，所以添加上分号有助于避免误会。

命名规范

在编码过程中，命名是重头戏。好的命名可以令代码赏心悦目，带来愉悦的阅读享受，令代码具有良好的可维护性。命令的主要范畴有变量、常量、方法、类、文件、包等。

1.变量命名

变量名都采用小驼峰式命名，即除了第一个单词的首字母不大写外，每个单词的首字母都大写，词与词之间没有任何符号，如：

```
var adminUser = {};
```

错误的示例如下：

```
var admin_user = {};
```

2.方法命名

方法命名与变量命名一样，采用小驼峰式命名。与变量不同的是，方法名尽量采用动词或判断性词汇，如：

```
var getUser = function () {};
var isAdmin = function () {};
User.prototype.getInfo = function () {};
```

错误示例如下：

```
var get_user = function () {};
var is_admin = function () {};
User.prototype.get_info = function () {};
```

3. 类命名

类名采用大驼峰式命名，即所有单词的首字母都大写，如：

```
function User {
}
```

4. 常量命名

作为常量时，单词的所有字母都大写，并用下划线分割，如：

```
var PINK_COLOR = "pink";
```

5. 文件命名

命名文件时，请尽量采用下划线分割单词，比如child_process.js和string_decode.js。如果你不想将文件暴露给其他用户，可以约定以下划线开头，如_linklist.js。

6. 包名

也许你有贡献模块并将其打包发布到NPM上。在包名中，尽量不要包含js或node的字样，它是重复的。包名应当适当短且有意义的，如：

var express = require('express');

比较操作

在比较操作中，如果是无容忍的场景，请尽量使用===代替==，否则你会遇到下面这样不符合逻辑的结果：

```
'0' == 0; // true
'' == 0 // true
'0' === '' // false
```

此外，当判断容忍假值时，可以无需使用 === 或 ==。在下面的代码中，当 foo 是 0、undefined、null、false、'' 时，都会进入分支：

```
if (!foo) {
  // some code
}
```

字面量

请尽量使用 {}、[] 代替 new Object()、new Array()，不要使用 string、bool、number 对象类型，即不要调用 new String、new Boolean 和 new Number。

作用域

在 JavaScript 中，需要注意一个关键字和一个方法，它们是 with 和 eval()，容易引起作用域混乱。

1. 慎用 with

示例代码如下：

```
with (obj) {
  foo = bar;
}
```

它的结果有可能是如下四种之一：obj.foo = obj.bar;、obj.foo = bar;、foo = bar;、foo = obj.bar;，这些结果取决于它的作用域。如果作用域链上没有导致冲突的变量存在，使用它则是安全的。但在多人合作的项目中，这并不容易保证，所以要慎用 with。

2. 慎用 eval()

慎用 eval() 的原因与 with 相同。如果不影响作用域上已存在的变量，用它是安全的。另外，利用 eval() 的这个特性，也可以玩出一些好玩

的特性来，比如wind.js利用它实现了流程控制。在大多数情况下，基本上轮不到eval()来完成特殊使命。示例代码如下：

```
var obj = {
  foo: 'hello',
  bar: 'world'
};
var key = (Math.round(Math.random() * 100) % 2 === 0) ? 'foo' :
'bar';
var value = eval('(obj.' + key + ')');
```

上述代码多出现在新手中，实际只要如下一行代码即可完成：

```
var value = obj[key];
```

数组与对象

在JavaScript中，数组其实也是对象，但是两者在使用时有些细节需要注意。

1.字面量格式

创建对象或者数组时，注意在结尾用逗号分隔。如果分行，一行只能一个元素，示例代码如下：

```
var foo = ['hello', 'world'];
var bar = {
  hello: 'world',
  pretty: 'code'
};
```

错误示例如下所示：

```
var foo = ['hello',
'world'];
var bar = {
  hello: 'world', pretty: 'code'
};
```

2. for in循环

使用for in循环时，请对对象使用，不要对数组使用，示例代码如下：

```
var foo = [];
foo[100] = 100;
for (var i in foo) {
  console.log(i);
}
for (var i = 0; i < foo.length; i++) {
  console.log(i);
}
```

在上述代码中，第一个循环只打印一次，而第二个循环则打印0~100，这并不满足预期值。

3.不要把数组当做对象使用

尽管在JavaScript内部实现中可以把数组当做对象来使用，如下所示：

```
var foo = [1, 2, 3];
foo['hello'] = 'world';
```

这在for in迭代时，会得到所有值：

```
for (var i in foo) {
  console.log(foo[i]);
}
```

也许你只是想得到hello而已。

异步

在Node中，异步使用非常广泛并且在实践过程中形成了一些约定，这是以往不曾在意的点。

1.异步回调函数的第一个参数应该是错误指示

并不是所有回调函数都需要将第一个参数设计为错误对象。但是一旦涉及异步，将会导致try catch无法捕获到异步回调期的异常。将第一个参数设计为错误对象，告知调用方是一个不错的约定。示例代码如下：

```
function (err, data) {
};
```

这个约定被很多流程控制库所采用。遵循这个约定，可以享受社区流程控制库带来的业务编写便利。

2.执行传入的回调函数

在异步方法中一旦有回调函数传入，就一定要执行它，且不能多次执行。如果不执行，可能造成调用一直等待不结束，多次执行也可能会造成未期望的结果。

类与模块

关于如何在JavaScript中实现继承，有各种各样的方式，但在Node中我们只推荐一种，那就是类继承的方式。另外，在Node中，如果要将一个类作为一个模块，就需要在意它的导出 方式。

1.类继承

一般情况下，我们采用Node推荐的类继承方式，示例代码如下：

```
function Socket(options) {
  // ...
  stream.Stream.call(this);
  // ...
}

util.inherits(Socket, stream.Stream);
```

2.导出

所有供外部调用的方法或变量均需挂载在exports变量上。当需要将文件当做一个类导出时，需要通过如下的方式挂载：

```
module.exprots = Class;
```

而不是通过

```
exports = Class;
```

私有方法无需因为测试等原因导出给外部，所以无须挂载。

注解规范

一般情况下，我们会对每个方法编写注释，这里采用dox的推荐注释，示例如下：

```
/**
 * Queries some records
 * Examples:
 * ```
 * query('SELECT * FROM table', function (err, data) {
 * // some code
 * });
 * ```
 * @param {String} sql Queries
 * @param {Function} callback Callback
 */
exports.query = function (sql, callback) {
  // ...
};
```

dox的注释规范源自于JSDoc。可以通过注释生成对应的API文档。

最佳实践

细致的编码规范有很多，有争议的也少，但这并不阻碍我们找到共同点。

冲突的解决原则

如果你要贡献部分代码给某个开源项目，而它的编码规范与你并不相同，这种情况下需要采用入乡随俗的原则，尽量遵循开源项目本身的编码规范而不是自己的编码规范。

给编辑器设置检测工具

实际上，现在的编辑器基本上都可以通过安装插件的方式将JSLint或者JSHint这样的代码质量扫描工具集成进开发环境中，这样编码完成后就可以及时得到提示。

如果采用的是Sublime Text 2编辑器，在安装好插件后，可以在项目中配置.jshintrc文件，每次保存都会在编辑器中提醒不规范的信息。

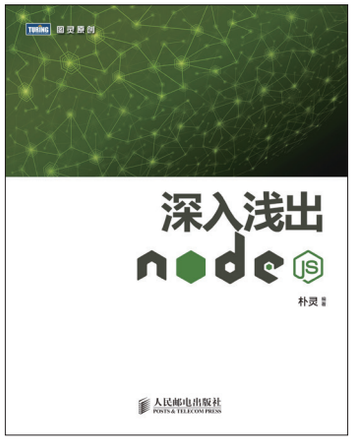
如下是我某个项目的.jshintrc文件，仅供参考：

```
{
  "predef": [
    "document",
    "module",
    "require",
    "__dirname",
    "process",
    "console",
    "it",
    "xit",
    "describe",
    "xdescribe",
```

```
    "before",
    "beforeEach",
    "after",
    "afterEach"
  ],
  "node": true,
  "es5": true,
  "bitwise": true,
  "curly": true,
  "eqeqeq": true,
  "forin": false,
  "immed": true,
  "latedef": true,
  "newcap": false,
  "noarg": true,
  "noempty": true,
  "nonew": true,
  "plusplus": false,
  "undef": true,
  "strict": false,
  "trailing": false,
  "globalstrict": true,
  "nonstandard": true,
  "white": true,
  "indent": 2,
  "expr": true,
  "multistr": true,
  "onevar": false,
  "unused": "vars",
  "swindent": false
}
```

版本控制中的hook

另一种最佳实践是在版本控制工具中完成的。无论SVN还是Git，都有precommit这样的钩子脚本，通过在提交时实现代码质量的检查。如果质量不达标，将停止提交。



《深入浅出Node.js》从不同的视角介绍了Node内在的特点和结构。书中并非完全按照顺序递进式介绍，首先简要介绍了Node，接着深入探讨了模块机制、异步I/O和异步编程，然后讨论了内存控制和Buffer相关的内容，接着探讨了网络编程、Node Web开发、进程、测试和产品化等内容，最后的附录介绍了Node的安装、调试、编码规范和NPM仓库搭建等内容。本文节选自《深入浅出Node.js》。

持续集成

持续集成包含两个方面：一方面仍是代码质量的扫描，可以选择定时扫描，或是触发式扫描；另一方面可以通过集中的平台统计代码质量的好坏变化趋势。根据统计结果可以判定团队中的个人对编码规范的执行情况，决定用宽松的质量管理方式还是严格的方式。

总结

代码质量关乎产品的质量，最容易改进的地方即是编码规范，收效也是最高的，它远比单元测试要容易付诸实践。一旦团队制定了编码规范，就应该严格执行，严格杜绝团队中编码规范拖后腿的现象。

也许可以采用CoffeeScript的方式来避免编码规范的问题，但是我相信在使用CoffeeScript之前，了解这些规范会更好地帮助你理解CoffeeScript。

如果你还采用非编译式JavaScript来编写你的应用，请记住这些编码规范。尽管因为历史原因无法一步到位改进这些缺点，但是既然知晓何为优秀，何为糟粕，就应该将优秀当做一种习惯。

参考资源

本附录参考的资源如下：

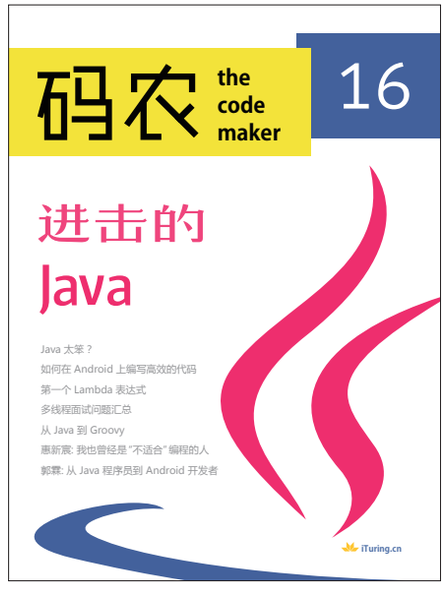
- http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml
- http://caolanmcmahon.com/posts/nodejsstyleand_structure/
- http://nodeguide.com/style.html Felix's Node.js
- https://npmjs.org/doc/coding-style.html NPM

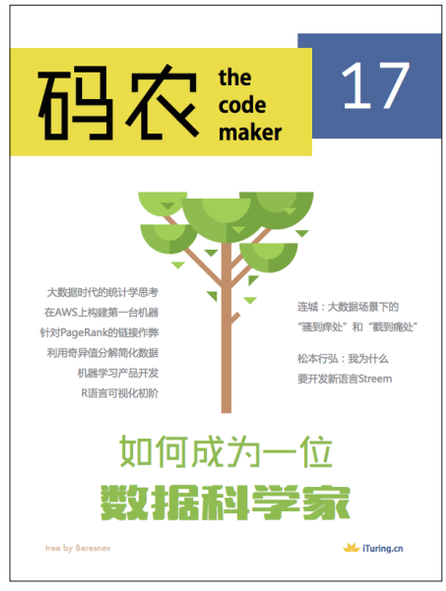
读《码农》吐吐槽 还能赚银子！

《码农》电子刊如今慢慢悠悠已经出版了18期，从一开始的好评如潮，到现在的“怎么这么抽象啊”“赶脚内容没有以前好了？”“一下就翻完了，没什么内容啊”……小编表示压力很大，现在的码农读者太难伺候了，明明是本免费杂志，￥%#%&*%@#￥*@￥#@！

但是，一看到好评，盼盼姐还是会热泪盈眶，热血沸腾，各种正能量啊“这么好质量的杂志还免费，天上掉馅饼呀”“希望一直出！每一版都读了，很值得推荐！！”……

所以《码农》还是会一直做下去，但是，是好是坏，您得给个话儿啊！





活动规则：在吐槽贴中留言，选出任何一期中你最喜欢的文章和最不喜欢的文章，即可获得图灵社区银子2两！凡吐槽吐得掷地有声者，加赠银子3两（共5两）！（怎样使用银子兑换图书）

活动时间：本活动长期有效。

朴灵：打破限制，从前端到全栈

朴灵，真名田永强，文艺型码农，Node.js布道者。现就职于阿里巴巴数据平台，任资深工程师，著有《深入浅出Node.js》。朴灵活跃于CNode社区，是线下会议NodeParty的组织者，同时也是JSConf China（沪JS、京JS，以及杭JS）的组织者之一。朴灵热爱开源，是多个Node.js模块的作者，个人GitHub地址：http://github.com/JacksonTian。**叩首问路，码梦为生。**

从前端到全栈

当时我们团队的Leader不会去限制工程师做什么事情，而是说这件事是你在负责，所以你就要从前做到后。

问：你从什么时候开始编程的？

我高二的时候才刚刚接触互联网，那时候看别人搭一个BBS我心里就特别崇拜，我很好奇这个过程是怎么实现的。以前有一个叫“中国博客网”的网站，上面提供了可以自己编辑模板的功能。在那个年代，脚本特效很流行，我经常把代码拷到那个模板里面去，测试效果。后来上大学之后我才知道，这个论坛的构建者其实就是利用了一个类似于Discuz的系统，搭这个网站并没有想象中那么复杂。

问：听说你在大学学的是Java，在这过程中你的兴趣点有没有转移？

学校教了很多东西，但是我的重心一直放在前端技术上。虽然学校教了C和C++这样的语言，但是教了之后老师也不告诉我们这个东西到底有什么用，所以我当时仍然痴迷于网页。工作几年以后才发现学校教的很多东西真的很有用，所以现在再回去补。

问：你是怎么变成所谓的全栈工程师的？是你自己的兴趣使然还是工作环境给了你这样一个机会？

我一般不说我是全栈工程师（笑）。我刚进大学的时候，目标其实很明确，就是想看看那些JS特效是怎么实现的。2009年毕业之后我开始工作，我就侧重去做前端。那时候前端也才刚刚兴起，后来我就成了一个真正的前端工程师。

去了淘宝之后，我碰到了Node.js这样一个我很感兴趣的东西，我用了两年时间完成了从前端工程师到全栈工程师的演变。我的个人习惯就是，如果我对什么感兴趣，就会在这块投入精力去研究。所以通过这一两年的学习，我可以完成所谓的后端的工作了。当时我们那个团队由空无和玄澄带领，他们不会去限制工程师做什么事，而是说这件事情是你在负责，所以你就要从前做到后。

问：在你看来全栈意味着什么？你在QCon主持了这个专题之后有没有带给你一些新的思考？

以前我认为全栈就是一个开发者能力的提升。如果你在某个领域非常专，有一个点很擅长，这时候别人才称你是专家；如果你有多个点比较擅长，那别人才有可能称你是全栈工程师。另外，责任也很重要。一个人不再只是负责在前端写CSS，也不只是在大工程里面做一小块东西，而是有了一个全局的视野。我以前对于全栈工程师的理解可能就是集中在工程师个人能力以及责任这两点上，通过主持Qcon的全栈工程师专题，我觉得还有一点，就是环境对全栈的影响。

个人能力是不太好复制的，有的工程师能力强，有的工程师能力没那么强，这种情况下你对能力稍微弱一点的工程师做这种要求，是很难达到目标的。Coding的分享给我带来了这个想法：基础设施建设对全栈的影响也是很大的。如果你不能帮助一个工程师成长，那就应该在环境上做出改变，你要去打造适合这样的人才成长的环境才行，这样才是正向循环。

问：淘宝现在使用Node.js的范围有没有扩大，主站有没有在用？

主站也有在用。现在几乎每个BU都在用Node做尝试，不管它做的是大系统还是小系统。我看到有很多团队尝到了甜头，但我也看到有很多团队不愿做改变，不愿去尝试新技术。

问：方便说一下你现在的工作内容是什么吗？

我最近半年在社区都不怎么活跃，是因为我做的内容才刚刚开始，现在才有了一些进展，将来我们有计划把我们做的东西对外产品化。我现在的主要工作就是改Node.js和V8。在Node.js上改动的话我会直接接到Io.js的仓库里面，现在已经提了很多。

我可以举个例子解释一下我们现在做的事情。在Node.js社区上有一些性能调优的工具，比如Chrome上有个CPU调优工具，它可以帮你找出哪段代码运行得比较慢。我们在这个基础上做了更多的工作，我们做了一个性能调优的工具，目标是不仅要知道它慢，还要知道它为什么慢。用我的工具扫过之后，我就知道怎么去改，其实现在我打的很多补丁都是通过这个工具发现的。造成性能慢的原因可能有很多，我们尽量去发掘这些东西。 另外一件我在做的事情就是帮助公司内的一些Node.js开发者。比如，这些开发者可能做业务开发时没有什么问题，但是运维或者运行的时候就会出现问题。我们工作几年来，并没有因为使用Node.js而造成了什么故障，高手写代码不太容易出现问题。但是对于每个工程师来说要求是不一样的，我现在的工作重点就是：出问题的时候能够马上解决。线上代码慢的原因是什么，为什么会有内存泄露，我们做了一些工具来解决这件事。

虽然用一个新技术的时候可能感觉很爽，但是它有什么后果或者副作用，可能对于很多人来说都是没底的。为了让阿里的技术环境能够更健康地发展，我们的工作就是要让大家知道，我们不怕这样的问题出现，能出现我们也能解决。现在我们努力在公司以内创造更好的环境，在未来我们有计划把这些东西推出来，希望国内其他用Node.js的公司也能从我们的工作中得到一些好处，独乐乐不如众乐乐。我们希望自己能够做更多的事情，有朝一日能成为Io.js或者Node.js项目里面的核心贡献者，然后去影响它。

问：淘宝UED正在实践前后端分离，应该也有其他团队在做类似的事情，你怎么看？

我个人没有直接参与这样的事情。我觉得前后端分离是否能完成的核心在于服务化的程度。如果系统都是面向服务去设计的，那么在这个过程中解耦可能很容易完成。这样当你真正做一个前端应用层的东西时，就不需要Java的同学来参与帮你写模板，而这个应用层可能是跟业务相关的。

这里的一个概念就是产品应该有两个部分：一个部分是系统层，另一个部分是应用层。应用层的需求可能天天变，但是这种改动基本上不怎么影响底层系统的稳定性。系统层应该是趋向于稳定的，除非业务模型上有一些大的变化的时候才会去改动。而平常的日常迭代，也就是应用层，会有频繁的发布和改动。

如果把两层东西都混在一起的话，整个系统就会比较庞大，界限又不清晰。后端的人会干预前端的东西，前端有需求的时候可能后端的同学也不能理解。在这过程中我觉得可以把应用层独立出来，让前端的同学能够在中间有一个缓冲地带，既能适应应用层的频繁发布和需求改动，又能让后端稳定。

问：JavaScript有很多框架和库，对于初级学习者来说，怎么能在这些资源中选择合适的来创建属于自己的技术栈？

我觉得JavaScript最核心、最源头的东西就是它的规范。当然，如果你纯粹去读规范的话也没有什么目的性，但那是一个核心。当你遇到困惑的时候就不妨去规范上找一找，通常你都能找到答案。这对于我来说是核武器一样的东西，一般不能用的（笑）。

另外，ECMAScript不算是纯粹的API，它定义的实现就不能更改，对于那些没定义的实现，你要去看那些API是怎么去做的。我基本上主要是靠这两个东西。现在社区里面也有很多资料，还有一些网站专门教你怎么去用。比如有一个项目叫做NodeSchool，它会设计一些非常小的任务给你，让你由浅入深地去完成。

他眼中的Node

> 有一个圈子，有一个社区，其实你会收获很多东西。

问：能不能简要介绍一下Node生态环境的发展现状？

这个话题其实我已经很久不再提了，因为现状已经太好了。前几年的时候模块有几个了、有几千个了，然后过了一年这个数字就上万了，再过一年就蹦到十万以上了。

我是蛮喜欢这样一个形态的，在跨过一个相对比较低的起点之后，大家可能都有能力来完成一个模块。但是，每个工程师的能力有强有弱，每个人的意识都是不同的，所以我觉得这个生态环境特别真实。这里面的东西多，水平参差不齐，但你会发现在很多情况下，如果一个模块是活跃的，它就一直是活跃的；如果它的质量不好的，它永远都是质量不好的。说不定几年以后，你会发现越来越多的模块都是以不断迭代更新的状态生存的，总是有最好的模块来适应当下最需要的事情。

问：我之前听你在采访里说，你当时推广Node的初衷是因为你看到国内外对Node的理解存在很大差距，现在还有没有这种差距？

现在还是有。我觉得最大的差距就是在国内推一个技术的时候，不管怎么样，出来的时候就会有很多人来黑。这些人可能也没有什么前提，就是嘴巴爽了再说。另外，国内外环境还有一个差距，就是大家主动刷新自己知识的能力还不是太强。你会发现有一些工程师可能已经能完成当前的工作了，但是他在持续学习上可能不会投入太多。

问：你认为前端工程师和后端工程师谁更应该学习Node.js？谁学习Node获益会更多？

其实无论什么工程师学习Node都可以有收获，但是我觉得前端工程师的收获会更大。

为什么要有前端这个工种？我不知道国外是不是这样，但是国内很多人会说前端工程师就是写HTML和CSS的，这种工作就像流水线一样，已经被限定在一个划定的圈里面去发展。当前端工程师发展到某一个阶段的时候，他会发现这个圈子对他是一个限制。我觉得前端工程师需要打破这个圈子，打破之后就会发现更开阔的视野。当我们想要解决一个问题的时候，就不只是再用已有的熟悉的办法来解决，而是发现视野以外，能够更好地解决问题的方法。反过来说，如果你总是不想去了解服务器端的一些技术，总是用前端的方法来解决问题，那成本也可能很高，做出来的方案也可能不理想。

前后端有Node.js连接以后，你就会发现自己的工具链已经完成了一次革命。很多工程师可能会用Java或Ruby来写一些工具，前端工程师不熟悉Java，也不熟悉Ruby，他要去完成目标的时候就会特别困难。现在Node.js出来以后，你就发现很多工具直接用Node.js就可以写了，比如说CoffeScript，还有一些像LESS,Sass之类的工具。前端工程师用自己熟悉的东西就能够改良自己的工具，这是第一步。改良工具以后视野就会放开，你会发现还有很多其他东西你可以出手去做。

问：你怎么看LinkedIn放弃Node和Scala？

做任何一件事情都会有最适合的方法。我觉得像LinkedIn这么大的系统，其实是一种很复杂、很复合的模式。我们在淘宝推广Node.js的时候，我们也不可能去用Node把那些最底层最核心的东西替换掉，这是不现实的。如果说在某个系统里面，Node.js不能胜任某些工作，那就不要用它好了。

据说LinkedIn也并没有完全放弃Node，只是某一部分弃用。我觉得Node.js是有它擅长的地方的，你只要能够在适合的地方去用好就行了。

问：Io.js和Node.js分离开来，是因为Node更新太慢造成的吗?

大部分原因我觉得可能在于，虽然Joyent公司高管对这个项目还算重视，但是他们对开发进度没有实质的帮助。

从2014年7月份开始转岗以后，我的工作重点就是Node.js的内核，所以我对Node.js和Io.js的提交列表和PR都是比较熟悉的。我发现当我提交一个补丁上去的时候，在Node.js下面的处理速度特别慢。我刚进去的时候看到PR列表大概只有两百多个，现在Io.js分出去以后PR列表更长了。那些在这个项目上做贡献的人，基本上80%已经不在这家公司了。所以现在出力的人不是这家公司的，而是社区的一些开发者。如果这家公司只想享受商标带来的好处，贡献者们就不会同意，所以他们就独立出来了。

目前来说我自己是比较喜欢Io.js这个项目的。我跟他们提一个问题上去，很快就会有反应。他们的态度也更开放，就是说不管你的贡献是大是小，只要对这个项目有好处的，他们都会接受。并且他们对新事物的接受速度也更快，比如现在Node.js的V8版本应该处于3.2，而Io.js里面的V8已经是4.2了。如果更新慢的话，ES6的一些特性就没办法引进来。因为Io.js中有这些新的特性，所以开发者就更愿意进入这个项目。

但是Io.js也给我造成了一些困扰。以前一个版本发布至少也要一两个月，但是现在的版本发布速度已经是以周为单位的了，几乎每周都能出一版。他们更新速度太快逼着我要去做很多工作，需要不停地打版本。话说回来，实际上企业内部升级版本的速度要求不是太快，可能升级一个大版本以后，如果不需要接下来小版本的功能，我可以先忍忍。

问：你觉得Node阵营的分裂是一件好事吗？

我觉得是好事。我记得很久以前就有人特别想给Node.js的异步IO全部加上Promise，如果不分裂，这件事情永远不可能。虽然现在的Io.js也不能完成这件事，但是只有分裂才有更多的可能性，才能产生更好的东西。

问：很多大公司比如Paypal，从Java转换到Node.js非常成功，在后端Node.js会取代Java吗？

不太可能，Node.js还是在应用层上更有优势。无论是我们在系统层所做的尝试，还是现在已有的案例，都没有能够证明Node.js能够取代Java。我举个简单的例子，现在没有人能拿Node.js来完成缓存或者涉及到集群的大计算，但是Java已经能够做到了。

但是Node.js在应用层上是有优势的，它的最初设计目的就是要优化IO和CPU的关系。以前的IO都是要阻塞CPU计算的，Node把所有IO相关的东西全部从主线程上剥离以后，主线程就变得比较高效。而我们在Java里面去实现异步是比较麻烦的，可能需要启用多线程。虽然Java也有框架去做这件事，但是对已有的开发者来说，由于他们对原有的模式已经很熟悉，所以不会愿意去做改变。应用层就应该能够快速运行并且面向很多系统。我可以用Node.js快速地开发，调用各个系统。

问：刚才你也提到，有些人认为Node.js在设计上最大的失败就是它的API是基于Callback，而不是基于Promise的，你同意吗？

当时的情况就是那样，ES6没有出来，Generator也没有出来，如果让Node.js创始人Ryan Dahl去做这件事，他不仅要去改上层的东西，还要去改JavaScript。但是现在的情况已经有变化了，ES6已经出来了，它里面有一个东西叫Generator，运行过程中我们让程序调用栈停下来，然后在某个时间再重新把它唤起来。新的框架Koa就能够利用好这个特

性，写程序的时候你会发现程序已经顺序执行了，但是背后的实质还是异步。我觉得这些变化将来可能会慢慢影响Node.js本身的设计，甚至将来Io.js的API可能也会慢慢去改。

另外，之所以当时会有Callback这样的设计，就是因为当时的Callback特别适合异步调用。这样做的原因也跟当时选JavaScript有很大的关系。Ryan Dahl去调研过Java、Ruby之类的语言，他发现这些语言提供的API有很重的历史负担。这时候如果给开发者提供新的API，大家也不会去用，因为这会改变他们的思维模式。他在设计这个模型的过程中，发现事件循环里面有阻塞IO的话就会导致效率急剧下降。而JavaScript特别适合完成这项工作，它不主动提供同步的API给你，所以阻塞的方式就不存在。

问：今年你还会不会举办京JS或者杭JS？

今年的会叫深JS。我们今年没有主动去办会，而是由之前我们的一个合作方去承接了这件事。他们是上海的一家外企，对JS社区有比较高的热情，过去三年举办的会他们都帮助我们做了很多事情，这次交给他们办也是顺理成章的。

问：以后你还会再举办这样的活动吗？你在这几次办会的过程中遇到过什么困难？

我们以后不排除仍然会举办这样的活动。有一次在北京办会，我们三个人都身在外地，所有的事情都是通过远程操作，在外地办会感到资源上的限制。另外的一个困难在于沟通，比如怎么去邀请国外的讲师或者跟我们国外的主办方沟通，中西方的文化差异会造成一些困扰。

我觉得办会最重要的还是内容，对内容的挑选和审核，讲师怎么邀请，主题要怎么规划。不能某个主题今年讲，明年讲，后年讲，有的公司可能出于商业目的有这样的需求，但有些东西是不能去妥协的。

问：纵然有这么多困难，但是你还是一直在坚持做这件事，举办这样的活动对社区来说最大的收获是什么？

其实在办活动这件事上，我有一个老师叫周裕波，我在上海工作的时候就遇到过他，他经常会办一些前端的活动。对于个人开发者而言，他们很少愿意主动出来组织这些活动，但周裕波做到了，我觉得我应该也能为Node做一些事情。当时我看到Node没有这样的氛围，国内的很多开发者都不愿意出来，当时的社区也关注不到这个东西，所以我们办了很多次Party。

这个过程其实收益还是很多的。首先我对这个社区更加了解了，跟很多高手都很熟悉。另外，我接触了很多赞助商，各种资源其实都是大家互相需要的。没做过这件事的人可能会觉得非常难，但做了以后你会发现没有想象中那么难。因为你总是会得到一些帮助，来自各个方面的帮助，你在经济上也没有什么损失，还会结交一些的朋友。有一个圈子，有一个社区，其实你会收获很多东西。■

动　手

用Express框架创建草地鹨旅行社网站

作者 / Ethan Brown

Ethan Brown是美国俄勒冈州Pop Art公司的高级软件工程师，负责网站及Web服务架构的设计与实现。拥有20多年编程经验，从事过嵌入式开发和Web开发，他相信JavaScript技术栈是未来的Web平台。

脚手架

脚手架并不是一个新想法，但很多人（包括我自己）都是通过Ruby才接触到这个概念的。这个想法很简单：大多数项目都需要一定数量的“套路化”代码，谁会想每次开始新项目时都重新写一次这些代码呢？对此有个简单的方法，那就是创建一个通用的项目骨架，每次开始新项目时，只需复制这个骨架，或者说是模板。

RoR把这个概念向前推进了一步，它提供了一个可以自动生成脚手架的程序。相对于从一堆模板中作出选择，这种方式的优点是可以生成更复杂的框架。

Express借鉴了RoR的这一做法，提供了一个生成脚手架的工具，从而可以让你开始一个新的Express项目。

尽管Express有可用的脚手架工具，但它目前并不能生成我推荐使用的框架。特别是它不支持我所选择的模板语言（Handlebars），也没有遵循我所偏好的命名规则（尽管这很容易解决）。

尽管我们不用这个脚手架工具，但我还是建议你看一下它：到那时，你就能够充分了解它生成的脚手架是否对你有用了。

套路化对最终发送到客户端的真正HTML也是有用的。我推荐非常出色

的HTML5 Boilerplate（http://html5boilerplate.com/），它能生成一个很不错的空白HTML5网站。最近HTML5 Boilerplate又新增加了可定制的功能，其中一个定制选项包含Twitter Bootstrap，这个是我高度推荐的前端框架。

草地鹨旅行社网站

本文以一个可运行的网站为例：假想的草地鹨旅行社网站，该旅行社是一家为到俄勒冈州旅游的人提供服务的公司。如果你对创建REST应用程序更感兴趣，不用担心，因为草地鹨旅行社网站除了作为功能性网站外，也提供REST服务。

初始步骤

先给你的项目创建一个新目录，这将作为项目的根目录。本文中，凡提到“项目目录”“程序目录”或“项目根路径”，指的都是这个目录。

提示 或许你会把Web程序文件跟项目相关的其他文件全都分开存放，比如会议纪要、文档等。因此，我建议你把项目根路径作为项目目录的子目录。比如，对于草地鹨旅行社网站而言，我会把项目放在~/projects/meadowlark，而项目根路径放在~/projects/meadowlark/site。

npm在package.json文件中管理项目的依赖项以及项目的元数据。要创建这个文件，最简单的办法是运行`npm init`：它会问一系列的问题，然后为你生成一个package.json文件帮你起步（对于“入口点”的问题，用meadowlark.js或项目的名字作为答案）。

提示 如果你的package.json文件中没有指定一个存储库的URL，以及一个非空的README.md文件，那么你每次运行npm时都会看到警告信息。package.json文件中的元数据只有在发布到npm存储库时才是真正必要的，但为了消除npm的警告信息，做这些工作依然是值得的。

第一步是安装Express。运行下面这条npm命令：

```
npm install --save express
```

运行npm install会把指定名称的包安装到node*modules*目录下。如果你用了--save选项，它还会更新package.json文件。因为nodemodules随时都可以用npm重新生成，所以我们不会把这个目录保存在我们的代码库中。为了确保不把它添加到代码库中，我们可以创建一个.gitignore文件：

```
# ignore packages installed by npm
node_modules

# put any other files you don't want to check in here,
# such as .DS_Store (OSX), *.bak, etc.
```

接下来创建meadowlark.js文件，这是我们项目的入口。本文中将这个文件简单称为“程序文件”：

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

var app = express();

app.set('port', process.env.PORT || 3000);

// 定制404页面
app.use(function(req, res){
            res.type('text/plain');
            res.status(404);
            res.send('404 - Not Found');
});

//定制500页面
app.use(function(err, req, res, next){
            console.error(err.stack);
            res.type('text/plain');
            res.status(500);
```

```
            res.send('500 - Server Error');
});

app.listen(app.get('port'), function(){
     console.log( 'Express started on http://localhost:' +
        app.get('port') + '; press Ctrl-C to terminate.' );
});
```

提示 很多教程，甚至是Express的脚手架生成器会建议你把主文件命名为app.js（或者有时是index.js或server.js）。除非你用的托管服务或部署系统对程序主文件的名称有特定的要求，否则我认为这么做是没有道理的，我更倾向于按照项目命名主文件。凡是曾在编辑器里见过一堆index.html标签的人都会立刻明白这样做的好处。`npm init`默认是用index.js，如果要使用其他的主文件名，要记得修改package.json文件中的`main`属性。

现在你有了一个非常精简的Express服务器。你可以启动这个服务器（node meadowlark.js），然后访问http://localhost:3000。结果可能会让你失望，因为你还没给Express任何路由信息，所以它会返回一个404页面，表示你访问的页面不存在。

注释 注意我们指定程序端口的方式：`app.set(port, process.env.PORT || 3000)`。这样我们可以在启动服务器前通过设置环境变量覆盖端口。如果你在运行这个案例时发现它监听的不是3000端口，检查一下是否设置了环境变量PORT。

提示 我高度推荐你安装一个能显示HTTP请求状态码和所有重定向的浏览器插件。这样在解决重定向问题或者不正确的状态码时会更加容易，它们经常被忽视。对于Chrome来说，Ayima的Redirect Path特别好用。在大多数浏览器中，都能在开发者工具的网络部分看到状态码。

我们来给首页和关于页面加上路由。在404处理器之前加上两个新路由：

```
app.get('/', function(req, res){
            res.type('text/plain');
            res.send('Meadowlark Travel');
});
```

```
app.get('/about', function(req, res){
            res.type('text/plain');
            res.send('About Meadowlark Travel');
});

// 定制404页面
app.use(function(req, res, next){
            res.type('text/plain');
            res.status(404);
            res.send('404 - Not Found');
});
```

app.get是我们添加路由的方法。在Express文档中写的是app.VERB。这并不意味着存在一个叫VERB的方法，它是用来指代HTTP动词的（最常见的是“get”和“post”）。这个方法有两个参数：一个路径和一个函数。

路由就是由这个路径定义的。app.VERB帮我们做了很多工作：它默认忽略了大小写或反斜杠，并且在进行匹配时也不考虑查询字符串。所以针对关于页面的路由对于/about、/About、/about/、/about?foo=bar、/about/?foo=bar等路径都适用。

路由匹配上之后就会调用你提供的函数，并把请求和响应对象作为参数传给这个函数。现在我们只是返回了状态码为200的普通文本（Express默认的状态码是200，不用显式指定）。

我们这次使用的不是Node的res.end，而是换成了Express的扩展res.send。我们还用res.set和res.status替换了Node的res.writeHead。Express还提供了一个res.type方法，可以方便地设置响应头Content-Type。尽管仍然可以使用res.writeHead和res.end，但没有必要也不作推荐。

注意，我们对定制的404和500页面的处理与对普通页面的处理应有所区别：用的不是app.get，而是app.use。app.use是Express添加

中间件的一种方法。你可以把它看作处理所有没有路由匹配路径的处理器。这里涉及一个非常重要的知识点：在Express中，路由和中间件的添加顺序至关重要。如果我们把404处理器放在所有路由上面，那首页和关于页面就不能用了，访问这些URL得到的都是404。现在我们的路由相当简单，但其实它们还能支持通配符，这会导致顺序上的问题。比如说，如果要给关于页面添加子页面，比如/about/contact和/about/directions会怎么样呢？下面这段代码是达不到预期效果的：

```
app.get('/about*',function(req,res){
            // 发送内容....
})
app.get('/about/contact',function(req,res){
            // 发送内容....
})
app.get('/about/directions',function(req,res){
            // 发送内容....
})
```

本例中的/about/contact和/about/directions处理器永远无法匹配到这些路径，因为第一个处理器的路径中用了通配符：/about*。

Express能根据回调函数中参数的个数区分404和500处理器。

你可以再次启动服务器，现在首页和关于页面都可以运行了。

截至目前我们所做的事情，即使不用Express也很容易完成，但Express所提供的一些功能并非那么显而易见。还记得如何规范化req.url来确定所请求的资源吗？我们必须手动剥离查询字符串和反斜杠，并转化为小写。而Express的路由器会自动帮我们处理好这些细节。尽管目前看起来这并非什么大不了的事情，但这只是Express路由器能力的冰山一角。

视图和布局

如果你熟知“模型-视图-控制器”模式，那你对视图这个概念应该不会感到陌生。视图本质上是要发送给用户的东西。对网站而言，视图通常就是HTML，尽管也会发送PNG或PDF，或者其他任何能被客户端渲染的东西。

视图与静态资源（比如图片或CSS文件）的区别是它不一定是静态的：HTML可以动态构建，为每个请求提供定制的页面。

Express支持多种不同的视图引擎，它们有不同层次的抽象。Express比较偏好的视图引擎是Jade（因为它也是TJ Holowaychuk开发的）。Jade所采用的方式非常精简：你写的根本不像是HTML，因为没有尖括号和结束标签，这样可以少敲好多次键盘。然后，Jade引擎会将其转换成HTML。

Jade是非常吸引人的，但这种程度的抽象也是有代价的。如果你是一名前端开发人员，即便你实际上是用Jade编写视图，也必须理解HTML，并且有足够深入的认识。我认识的大多数前端开发人员都不喜欢他们主要的标记语言被抽象化处理。因此我推荐使用另外一个抽象程度较低的模板框架Handlebars。Handlebars（基于与语言无关的流行模板语言Mustache）不会试图对HTML进行抽象：你编写的是带特殊标签的HTML，Handlebars可以借此插入内容。

为了支持Handlebars，我们要用到Eric Ferraiuolo的`express3-handlebars`包（尽管名字中是express3，但这个包在Express 4.0中也可以使用）。在你的项目目录下执行：

```
npm install --save express3-handlebars
```

然后在创建app之后，把下面的代码加到meadowlark.js中：

```
var app = express();

// 设置handlebars视图引擎
var handlebars = require('express3-handlebars')
            .create({ defaultLayout:'main' });
app.engine('handlebars', handlebars.engine);
app.set('view engine', 'handlebars');
```

这段代码创建了一个视图引擎，并对Express进行了配置，将其作为默认的视图引擎。接下来创建views目录，在其中创建一个子目录layouts。如果你是一位经验丰富的Web开发人员，可能已经熟悉**布局**的概念了（有时也被称为“母版页”）。在开发网站时，每个页面上肯定有一定数量的HTML是相同的，或者非常相近。在每个页面上重复写这些代码不仅非常繁琐，还会导致潜在的维护困境：如果你想在每个页面上做一些修改，那就要修改所有文件。布局可以解决这个问题，它为网站上的所有页面提供了一个通用的框架。

所以我们要给网站创建一个模板。接下来我们创建一个views/layouts/main.handlebars文件：

```
<!doctype html>
<html>
<head>
      <title>Meadowlark Travel</title>
</head>
<body>
      {{{body}}}
</body>
</html>
```

以上内容你未曾见过的可能只有{{{body}}}。这个表达式会被每个视图自己的HTML取代。在创建Handlebars实例时，我们指明了默认布局（`defaultLayout:'main'`）。这就意味着除非你特别指明，否则所有视图用的都是这个布局。

接下来我们给首页创建视图页面，views/home.handlebars：

```
<h1>Welcome to Meadowlark Travel</h1>
```

关于页面，views/about.handlebars：

```
<h1>About Meadowlark Travel</h1>
```

未找到页面，views/404.handlebars：

```
<h1>404 - Not Found</h1>
```

最后是服务器错误页面，views/500.handlebars：

```
<h1>500 - Server Error</h1>
```

提示 你或许想在编辑器中把.handlebars和.hbs（另外一种常见的Handlebars文件扩展名）跟HTML相关联，以便启用语法高亮和其他编辑器特性。如果是vim，你可以在~/.vimrc文件中加上一行`au BufNewFile,BufRead *.handlebars set filetype=html`。其他编辑器请参考相关文档。

现在视图已经设置好了，接下来我们必须将使用这些视图的新路由替换旧路由：

```
app.get('/', function(req, res) {
    res.render('home');
});
app.get('/about', function(req, res) {
    res.render('about');
});

// 404 catch-all处理器（中间件）
app.use(function(req, res, next){
    res.status(404);
```

```
    res.render('404');
});

// 500错误处理器（中间件）
app.use(function(err, req, res, next){
    console.error(err.stack);
    res.status(500);
    res.render('500');
});
```

需要注意，我们已经不再指定内容类型和状态码了：视图引擎默认会返回text/html的内容类型和200的状态码。在catch-all处理器（提供定制的404页面）以及500处理器中，我们必须明确设定状态码。

如果你再次启动服务器检查首页和关于页面，将会看到那些视图已呈现出来。如果你检查源码，将会看到views/layouts/main.handlebars中的套路化HTML。

视图和静态文件

Express靠中间件处理静态文件和视图。只需了解中间件是一种模块化手段，它使得请求的处理更加容易。

static中间件可以将一个或多个目录指派为包含静态资源的目录，其中的资源不经过任何特殊处理直接发送到客户端。你可以在其中放图片、CSS文件、客户端JavaScript文件之类的资源。

在项目目录下创建名为public的子目录（因为这个目录中的所有文件都会直接对外开放，所以我们称这个目录为public）。接下来，你应该把`static`中间件加在所有路由之前：

```
app.use(express.static(__dirname + '/public'));
```

static 中间件相当于给你想要发送的所有静态文件创建了一个路由，渲染文件并发送给客户端。接下来我们在 public 下面创建一个子目录 img，并把 logo.png 文件放在其中。

现在我们可以直接指向 /img/logo.png（注意：路径中没有 public，这个目录对客户端来说是隐形的），static 中间件会返回这个文件，并正确设定内容类型。接下来我们修改一下布局文件，以便让我们的 logo 出现在所有页面上：

```
<body>
    <header>
    <img src="/img/logo.png" alt="Meadowlark Travel Logo">
    </header>
  {{{body}}}
</body>
```

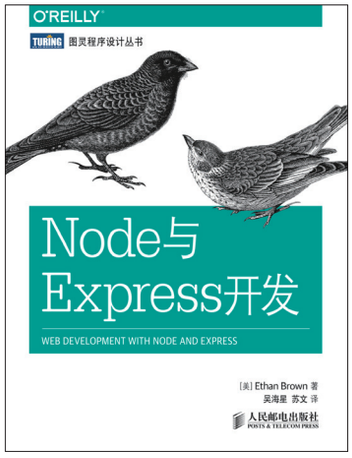
注释 `是 HTML5 中引入的元素，它出现在页面顶部，提供一些与内容有关的额外语义信息，比如 logo、标题文本或导航等。

视图中的动态内容

视图并不只是一种传递静态 HTML 的复杂方式（尽管它们当然能做到）。视图真正的强大之处在于它可以包含动态信息。

比如在关于页面上发送“虚拟幸运饼干”。我们在 meadowlark.js 中定义一个幸运饼干数组：

```
var fortunes = [
  "Conquer your fears or they will conquer you.",
  "Rivers need springs.",
  "Do not fear what you don't know.",
  "You will have a pleasant surprise.", "Whenever possible, keep
it simple.",
];
```



Express在根本没有框架和有一个健壮的框架之间找到了平衡，让你自由选择架构。通过《Node与Express开发》，熟悉JavaScript的前端和后端工程师会发现一种新的Web开发视角。本文节选自《Node与Express开发》。

修改视图（/views/about.handlebars）以显示幸运饼干：

```
<h1>About Meadowlark Travel</h1>

<p>Your fortune for the day:</p>
<blockquote>{{fortune}}</blockquote>
```

接下来修改路由/about，随机发送幸运饼干：

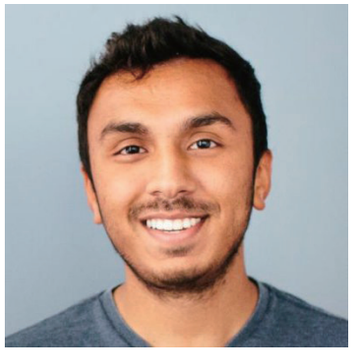
```
app.get('/about', function(req, res){
  var randomFortune =
          fortunes[Math.floor(Math.random() * fortunes.length)];
  res.render('about', { fortune: randomFortune });
  });
```

重启服务器，加载/about页面，你会看到一个随机发放的幸运饼干。模板真的是非常有用。

小结

我们刚用Express创建了一个非常基本的网站。尽管简单，但这个网站包含了功能完备的网站所需的一切。 ■

新手学习编程的最佳方式是什么？



作者 / Roshan Choxi

美国在线编程学习公司Bloc联合创始人兼CEO。

回答这个问题是我最近两年来唯一的关注点。我觉得此处提及的许多资源尽管都很不错，然而我却注意到，成功的学生，无论使用哪种资源，往往都会在以下三个方面，比其他人有着更好的表现。

- 聚焦习惯而不是目标
- 独自学习非常痛苦
- 项目实践

聚焦习惯而不是目标

聚焦习惯而不是目标听起来似乎不合常理，但是请听我把话说完 - 这是一个有关平衡的问题。凡是和我共过事的人都知道，我有时就像傻子一样，在一天内，引用《蝙蝠侠：侠影之谜》中 Ra's Al Ghul 的话多达 3-4 次。

译者 / 傅俊杰

Qingniu/ 青牛，码农，复唧唧联合创始人，业余时间喜欢研究中国传统文化。博客@明珠夜话。

Bloc 公司的投资者们对于我在董事会上频繁地引用《蝙蝠侠：侠影之谜》中的片段作为开始，早已经烦透了。

R'As 告诉 Bruce:

“（当你的胳膊快要冻僵的时候，）按摩你的胸口，你的胳膊自然会暖和起来。”

如果你将精力放在在每星期编程 20-30 个小时的习惯培养上，成为一名 Web 开发者的目标很快就可以实现。如果你将目标设定为在数月内成为一名 Web 开发者，在什么时候能可以达成目标以及距离目标还有多远等不确定性的压力之下，你反而可能会一无所获。聚焦于习惯而不是目标。按摩你的胸口，你的胳膊自然会暖和起来。

因此，你现在应该做的是：在你的日程安排上，每天花 15 分钟时间用于编程。不要多于 15 分钟，每天只要 15 分钟就够了。如果你能在一星期之内坚持做下来而不找任何借口，请试着把时间延长到一天 20 分钟。不要试图通过每天一小时的编程来过度扩展自己的能力，编程就是一场

10, 000 小时的马拉松，因此我们应该将精力集中在培养习惯上。与其在一天内花费大量时间学习编程，不如每天花费少量时间，但是能够天天坚持，因为这样更有效。

独自学习非常痛苦

当我在学习 Web 开发时，在我的学习过程中，拥有一位导师和加入一个社区是两个最大的组成要素。

- 拥有一位导师

大学期间，我曾在一家名叫 merge.fm 的小型创业公司工作。暑假期间，我与该公司的一位联合创始人一起工作，我在此期间学到的东西比我之前一年在大学学到的还要多。跟随一名经验丰富的专业人士一起工作，可以真正地加速你学习的速度，你会了解到他们如何思考问题的，同时，也会发现自己的不足之处。这就是为什么师徒制通常成为学习一门新技艺的默认标准的一个原因吧，因为它非常有效。

- 加入一个社区

我加入了的两个社区，一个是 Illini Entrepreneurship Network (我们学校的一个学生组织)，另一个是 Hacker News (一个面向黑客和创业者的大型在线社区)。

我没有从 Hacker News 学到任何有关对象和类的知识，但是我学到了一些别的东西。我知道了没人喜欢 JavaScript。我知道了 Ruby 程序员是编程领域的潮人。我还知道了 Bret Taylor, Rich Hickey, and John Carmack 都是编程世界的领袖人物，还有就是，那些真正关心员工的软件公司，它们公司的厨房看上去就像兴奋剂实验室。

总之，我学会了如何说行话。当你和其他程序员一同工作的时候，这一点非常重要，也正因如此，才使你觉得自己就是一名程序员。

项目实践

在学习 Web 开发的第一年，我动手实现了以下项目：

- 一个 Digg 的克隆版（来自 Sitepoint 书上的 Rails 例子，我想现在它已经过时了）
- 一个在线购物应用程序（来自 Agile Web Development with Rails 4）
- 一个 GeekSquad-esque 应用程序（个人项目）
- 一个实时在线课堂应用程序（个人项目）
- 一个外语学习应用程序（课程项目）

我认为构建真实项目之所以非常重要的原因有很多，就我而言，最重要的一个原因是它充满了乐趣。这恰恰是在传统教育中严重缺失的，也正因如此，它才成为了诸多原因中最重要的一项。寻找如何开展项目实践的参考资源，http://ruby.railstutorial.org/ 是一个不错的选择。

要具有“小强”精神

对于那些能坚持读到此处的读者们，我偷偷地增加了第四项内容。 Paul Graham 曾经对 Airbnb 的创始人说过：

> “你们这些家伙是不会失败的，因为你们就像打不死的‘小强’一样”。

一段时期，你可能会有放弃学习编程的想法。就像任何具有同样价值的事情一样，学习编程真的很难，有时你会觉得自己真的很笨。这就是为什么第一项策略如此重要的原因 - 不要过度担心是否已经取得进步，或者需要花费多长时间达成目标。你要做的就是每周坚持 10-30 个小时的编程。就像执着的’小强’一样，你就不会失败。

多年以前我说过一句话 - “成功贵在坚持（注解：亦可译为‘生活中80%的成功源于坚持’）”，这句话经常被其他人引用。人们常常向我提及，他们想编写一个剧本，他们想制作一部电影，他们想撰写一本小说。那些最终成功实现目标的人，80%的做法是先行动起来。那些最终失败的人，他们连这一点都做不到。这正是他们不能做成一件事情的原因，他们没有去做。一旦你去做了，如果你真地去编写电影剧本，或者撰写小说，实际上，你的成功之路已走过大半。这就是我可以告诉你的我人生最大的成功经验。其它的都是失败的教训。

—— Woody Allen（美国好莱坞著名电影导演）

图灵社区原文

英文原文：What are the best ways for a complete beginner to learn programming_

前端工程师入职两年的工作和技术总结

作者 / 揭秋明，1989 年出生于江西赣州，本科、硕士毕业于中国地质大学（武汉）软件工程系，现就职于科大讯飞（合肥总部）。关注前端开发、.NET 分布式计算，常泡图书馆，爱好阅读 IT 技术书籍，喜欢阅读英文技术文章；周末也会动手做点美食犒劳自己，习惯在刷碗的时候思考人生。

来公司将近两年，从一开始的前端开发，到现在的 ASP.NET Web 开发、.NET 客户端工具开发，跨度比较大。目前的状态是技术研究不够深，多数停留在表层，没能强制自己做好学习总结工作。项目工作内容也没用一个传承接代的过程，项目中需要你做什么，那你就得去干，下面主要梳理下这两年的工作内容和技术成长过程。

智能电子书项目

刚来公司接触的第一个项目就是教学软件的 html5 化，一开始面对 HTML5/CSS3/JS，这些当时以为比较 low 的技术，难免有些心浮气躁，但是接触到前端模块化开发后，我的观念马上就转变了，不管哪门技术，都有它的适用场景和优点，不能一棒子打死。前端模块化（SeaJS）让 JavaScript 变得跟 C++ 等其他高级语言一样可以方便的进行团队协同开发，一下子提高了生产力，纠正了我以前认为的 JavaScript 仅能实现某些特效的错误思想。还有 jQuery 的出现，帮助前端开发人员解决了浏览器兼容性的头疼问题。

这个时候我恶补 JavaScript、jQuery 的知识，网络收集 jQuery 的各种教程，其中从零开始学习 jQuery 系列让我快速上手 jQuery 开发，同时也学习 Google JavaScript 编码规范指南。

同时阅读相关的经典书籍也不能落下，在图书馆借来的书单如下：

1.《JavaScript DOM编程艺术》
2.《编写可维护的JavaScript》
3.《精彩绝伦的CSS》
4.《编写高质量代码--Web前端开发修炼之道》
5.《响应式Web设计实践》

这一阶段算是比较轻松的经过了3个月的试用期，在转正答辩会上，我也提出希望能有更多挑战的工作机会。结果研发经理就安排我去了现在的考试产品研发线，开始了苦逼的模考之旅。

模考数据可视化软件开发

调入新的考试业务线做研发后，感觉项目的紧张程度比以前更高了，先了解了模考数据可视化软件的功能后，我发现该项目算是大系统中的一个子项目，可能就你一人负责，相比之前多人同时开发一个项目，任务更重了。该项目基于Chromium 嵌入式框架（CEF）内置网页的HybridApp，需要对考试任务数据进行分析和读取，然后根据可视化要求及计算公式，得到最终的数据，最后利用可视化图表库Highcharts进行展现。

该项目有多个奇葩的地方：

1.前端代码表现、行为、内容没有做到完全分离；
2.前端JS代码没有进行模块化组织；
3.前端JS代码无法调试（CEF控件集成有误，不支持JS调试）。

我的做法：

1.针对目前已有的代码不进行重构工作，但是在增加的代码中尽量做到前端代码的HTML/CSS/JS分离；

2. 不进行 SeaJS 前端代码改造，改用闭包的形式分离代码模块；
3. 通过打印关键对象值的调试信息进行调试。

做完这个项目后，发现已有的业务逻辑代码较多，自己也无更大的精力去改造。有空要研究下 **JavaScript 的数据处理功能**，包括性能以及实现方式。目前知道的方式有：

1. 将数据处理放在后端，并结合 HTML5 中的 WebWorker，防止 UI 卡死。
2. 对于不支持 HTML5 的浏览器，则将需要大量处理数据的过程分割成很多小段，然后通过 JavaScript 的**计时器来分别执行**，就可以防止浏览器假死，详情见该链接内容。

基于互联网的自动化评分系统

这个项目是考试业务线的大项目，主要分两阶段：

1. 考试数据收集，上传至公司服务器；
2. 对当前的两个评分系统（人工评分和机器评分系统）进行自动化调度，根据考生音频评测情况自动分发打分任务，提高考试成绩发布速度。

我在第一阶段的任务是：

1. 开发数据收集 Restful 接口，提供给考试客户端调用，利用 ASP.NET WebAPI、对象关系模型（ORM）框架 Entity Framework，这一阶段已经逐渐转向后台开发了，重温了 SQL 以及 C#。
2. 同时开发后台监管系统，也是基于 ASP.NET MVC 框架搭建。
3. 为了适应开发的需要，我买了一本《ASP.NET MVC4 Web 编程》，断断续续读完了，该书较完整的介绍了利用 ASP.NET MVC 开发网站的技术，涉及前后台的设计与开发，还包括性能优化、web 安全等方面的知识。

第二阶段的主要任务是：

1. 开发评分系统调用服务，利用**Redis**通信来调度人工评分系统和机器评分系统，我们将评分流程分为基础数据准备阶段、机器评分阶段、人工评分阶段、成绩汇总阶段，每个阶段对应一个服务，同时还有一个流程监控调度服务，负责消息的转发和控制。涉及到的知识点有：.NET多线程、sqlite数据库操作、报表操作等。
2. 改进后台监管系统，增加试卷资源的管理，并基于Bootstrap和SeaJS改造系统界面，特别针对大量列表格数据的呈现进行可视化优化和用户体验，即合并列数据、数据分层分级、显示最重要的数据、隐藏不太重要的数据。
3. 这段时间重点看的书是介绍前端MVC的书《基于MVC的JavaScript Web富应用开发》，不过有些概念理解的不是很深刻，该书较全面的介绍了MVC在前端开发中的应用。

统一模考平台

进入2014年11月，新的项目又开始了，主要是对口考网进行改版，并且将考试操作搬到Web上，方便教师进行考试，且可以查看成绩报告等；在这个项目中我的主要工作是负责前端框架的搭建和设计，协调其他两名前端开发人员进行网站的前端开发，另外还涉及到后台接口的设计与需求讨论。

在实际开发过程中，也出现了许多问题：

1. 需求不具体，花费太多时间讨论需求，导致开发进度一度落后延迟。
2. 代码质量没有得到很好的保证，单元测试没有做好，后来尝试使用前端测试框架QUnit。
3. 与UI的配合不够默契，界面优化工作耗时费力。

主要使用到的技术有：

1. ASP.NET MVC框架，前端还是使用seajs、jQuery模块化开发，同时利用模板引擎artTemplate生成表格数据。
2. 后台使用 Redis缓存，数据库索引技术。
3. 后期尝试使用Grunt打包上线网站，不过还没有全部完成。

模考试卷制作工具

加入考试业务线之后，就一直跟试卷制作打交道，发现里面坑很深，牵涉到语音资源部和其他开发小组同事，一开始制作试卷均是手动操作，后来我开发工具进行快速转换，主要是文本资源内容的拷贝、替换，音频的转换，图片和ppr文件的拷贝，甚至还有sqlite数据库的读写。

目前已经支持全国各地几十种试卷资源的转换，代码急需重构，重构方向有如下：

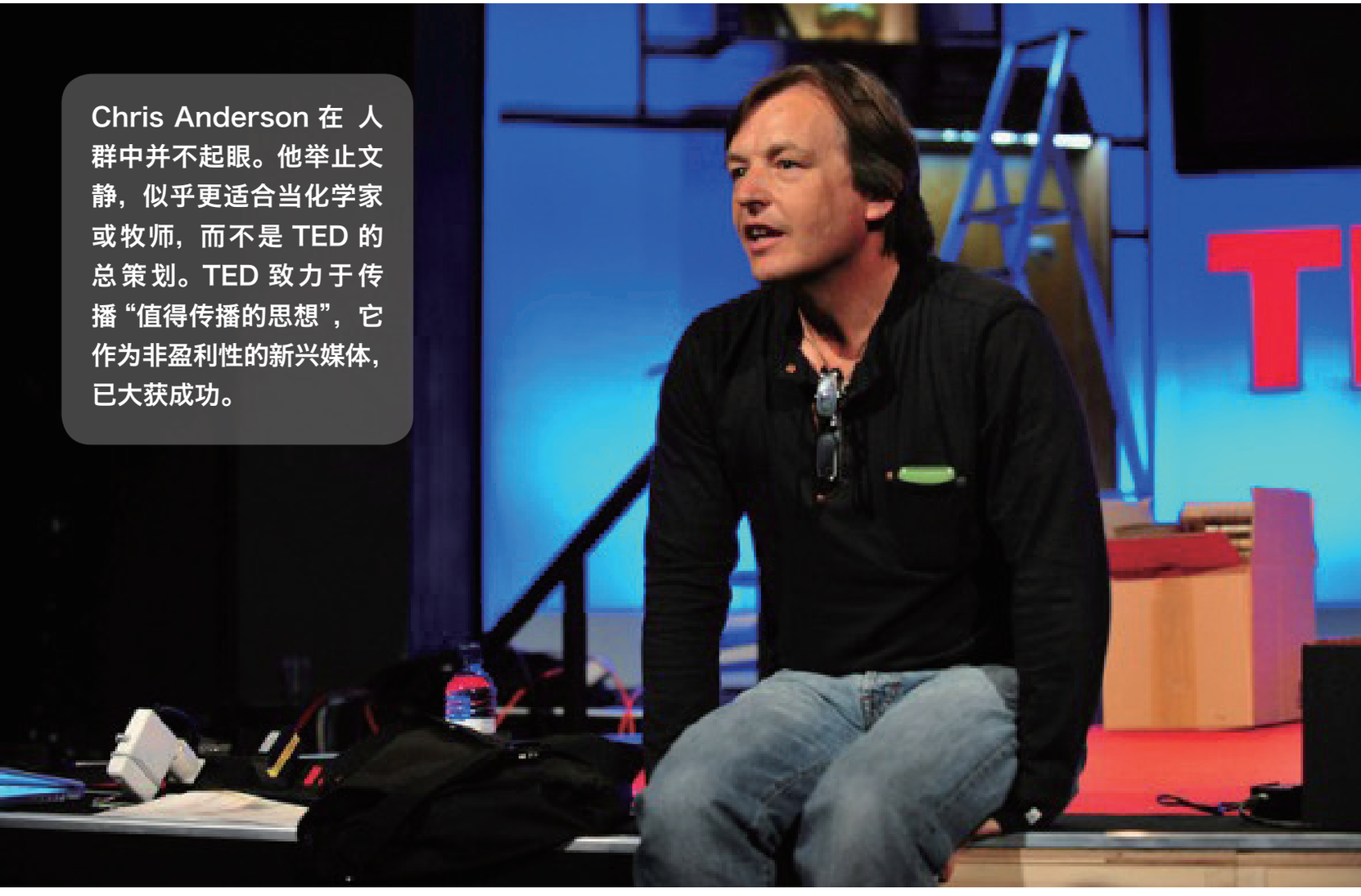
1. 将对试卷模板xml的解析按照模块划分，下次制作资源时直接根据模板来组装。
2. 利用策略者模式或者是单例模式来改造代码，梳理出试卷转换操作的主要流程，并做到自动化。
3. 另外试卷原始资源制作及验证过程主要交由语言资源部同事来进行，容易出现一些细节错误，这时就得我这边来寻找问题，经常会占用许多工作时间。

总结

从以上项目及技术实践看来，前后端技术都接触到了，不过主要的精力都投入到业务开发中，需及时总结遇到的技术难点和解决方案，后续我会逐步梳理出可以总结的部分，以博文的形式发布出来，方便记忆。■

图灵社区原文

TED总策划Chris Anderson：
我是个永不退休的人



Chris Anderson 在人群中并不起眼。他举止文静，似乎更适合当化学家或牧师，而不是 TED 的总策划。TED 致力于传播“值得传播的思想”，它作为非盈利性的新兴媒体，已大获成功。

作者 / Maria Giudice

创新者、画家、女性领导者，知名设计公司 Hot Studio 创始人，2013 年 3 月 Facebook 并购 Hot Studio 后，Maria 开始担任 Facebook 产品设计总监。

我们在安德森位于纽约的办公室里采访了他。房间中的现代家具和暖色调的硬木地板搭配在一起，让人感觉十分舒适。屋外传来城市的喧嚣，在这样的背景音下，安德森向我们讲述了他的事业——只有 DEO 才可能设计出的事业。

问：你还是个孩子的时候，觉得自己长大了会是什么样？

我那时候想成为一名兽医。我的父亲是医生，我想跟他不一样，而且我很爱我的狗。大概就是这样一个逻辑。但这种想法没有持续很久，后来我基本上听从命运安排，认为自己会成为一名医生。从长远来看，帮助人们好像比帮助动物更有用一些。

让人高兴的是，在差不多 18 岁时，我明白自己要是当医生的话，肯定是个庸医。搞清楚这一点让我欣喜若狂。我根本记不住那么多细节，对病人也没耐心。我对生物学很感兴趣，但是一想到还要经过三四年培训，然后无休无止地跟一个个病人打交道，我就感到非常、非常有压力。

大众传媒在那时候看起来是个激动人心的行业，这可能是因为我生性内向，以为搞传媒可以减少与他人一对一的接触。新闻工作在我眼中就是写出很多文字，如果文章够好，就可以通过富有魔力的印刷媒体让潜在的数百万读者看到，这一点光是想想就让人心潮澎湃。我一直都徘徊在文字和科学之间，能将两者有趣地结合起来的职业，我还真是没找到几个。

家庭电脑最早出现在英国市场大约是在 1981 年，那年我一把电脑买回家就被它彻底吸引了。我发现了自己内在的极客特质——事实上，这种特质一直都没有消失。我刚开始时做了几种不同的新闻工作，后来成为了一份电脑杂志的编辑，我感到十分兴奋。干了大约有一年，然后就出来创业了。

问：你从编辑变身企业家，这种转变是有意为之还是事出偶然？

一开始并不是有意为之。我的家族里就没出过商人，我脑子里也从没蹦出过要成为企业家的想法。在成为电脑杂志编辑以后，我突然发现可以去做一些更小规模的杂志，这深深地吸引了我。这些小杂志可以用电脑上的排版文件打印出来。如此一来，如果三个孩子有台家庭电脑的话，实际上就可以设计并印出一本杂志。

事情就这样开始了。我离开了最后一份杂志的工作，不再给人打工。三个月后，我们自办的小杂志就在英国开始卖了。每次想到这里我都觉得不可思议。那时候还没有互联网，人们的兴趣只是以各种小众话题的形式爆发出来，尤其是围绕着电脑的一些兴趣。我们都有过那种疯狂的商业经历：第一年开始赚钱，第二年规模、员工和利润翻倍，然后在之后的七年里年年如此。我们能够聚集起一帮非凡人物，他们都乐意为彼此工作，同时还吸引着其他优秀的人，这种经历真是妙不可言。

问：那个时候对你来说，最重要的是什么？

我们是一家小公司，每个人都沉浸在激情中。公司当时的口号就是“有激情的媒体”。作为传统媒体的叛军，我们想要与那些大出版商较量一番。我们的秘密武器就是对读者真正的阅读需求有深刻的、发自内心的理解。我们招聘的时候，不会因为对方做过传统新闻工作就录用他。我们雇佣的人都是对某些主题有激情的人，比如电脑极客或者电子游戏狂人。我们公司处处都充斥着人们的激情。

一方面，作为传统媒体的局外人，挑战那些规模大得多、资源丰富得多的大公司十分令人兴奋。我们有读者也能感受到的一腔热血，还有几个别人没想出来的好点子，比如把杂志和软件打包出售。对于我们来说，读者为什么乐意花双倍价钱买我们的产品套装而不去买传统杂志，这是显而易见的。但是对于其他出版商而言，弄明白个中缘由真是花了很长的时间。

问：你是从什么时候开始把自己看作一位领导的？

我们开始创业的一年后，我才意识到这一点。那时候我们推出的第二份杂志已经很成功了。看到读者对第二份杂志的反应，我们都禁不住惊呼："不会吧！"于是我心中盘算："这样下去肯定没问题，而且以后还会有更多杂志问世。"那时候，我们就是在做自己酷爱的事情。我以前从没想过能够实现，当时兴奋得脑袋都要炸了。

问：你那时候的领导风格是怎样的？

我肯定不是那种发号施令、控制欲强的领导。我努力用自己的热情来领导别人。你要知道，谁的想法最好谁就胜出。

我会说："伙计们、伙计们、伙计们！我们可以做这个！"或者"这样子出来绝对棒，为什么呢？是这样……"或者"如果你这样……这样……这样做，人们会非常喜欢！"

经常会有些想法在我脑子里狂呼乱叫，结果说出来会碰一鼻子灰，或者最终事实证明这些想法并不成功。但是各种选题的点子绝对是层出不穷。我们不断发布。在第一个 7 年里，我们发布了 32 份大获成功的杂志——总共发布了 40 份，但是其中 8 份失败了。32 份杂志活了下来，而且一直赢利。整个过程让人热血沸腾。我们所做的就是"去争取""去尝试"。

问：在工作中，你有哪些强项？

我用想法激发人们的兴奋点。不论是个人层面还是社交层面，我都不认为自己在领导别人。世上几乎所有人都能做得比我出色。但是如果说到"这里有个绝佳的点子，我来告诉你原因"，我绝对是个鼓动人心的好手。

很多时候你会碰到这么一瞬间，点子"啪"地卡对了位。有很多词适合用在这儿：冒险、设计、想象。人类就是有这样的本事，可以玩转未来，

可以在脑海中形成未来某个东西的模型，然后在脑子里不断修改完善，直到“啪”的一声有个点子卡对了位置。这时候不管原因是什么，你都会说：“就是这样。”

内向的人或者说更沉浸于自我的人有一个好处，那就是你在点子卡进的位置停留得更久，你会在内心世界不断玩味未来。我是一个非常以未来为导向的人，一直都向前看而不向后看。如果有个东西合乎未来的口味，那么它就对我的胃口。典型情况就是把不同来源的想法放在一起。这些不同的东西在你脑子里盘旋，但你就是感觉哪里不太对头。直到突然间，有东西非常非常对你的胃口，就好像“啪”的一声，恰好卡在了你的脑子里。

> 我肯定不是那种发号施令、控制欲强的领导。我努力用自己的热情来领导别人。

你有时候凌晨三点醒来，兴奋异常。你马上跑去把你的想法解释给别人听，甚至解释时结结巴巴、不修边幅，但是你眉飞色舞、激情澎湃，于是这个点子胜出。人们接着说“哇！那绝对靠谱，真是酷！”，一群人都兴奋起来，最终你就能实现这个想法。

问：你如何保持创造力？

你越是把自己置于各种意想不到的刺激之下，就越有可能获得那种“啪”的重要时刻。你肯定听说过，史蒂夫·乔布斯在很大程度上将他的成功归功于里德学院的书法课。

当然，TED 最美妙的时刻就是你听到开头内容，感觉这个人绝对不怎么有意思，然后“叮！”，产生强烈共鸣。我就有过这种体验，那是多年前看舞蹈团表演的时候，听到有些人在说：“是啊，舞蹈也在变呢。多亏了 YouTube，小孩子能与世界另一边的同龄人互相学习了。”我想：“好家伙，

这不正是 TED 演讲者之间发生的事情吗？”人们正在互相学习——认识到这一点竟然是在这么毫不相关的场合。你必须在锅里放很多不同的原料，然后再搅拌，才会产生奇妙的效果。

问：你是否遵循某种固定做法？

如果说我有固定的做法，那就是把有创造力的人聚集到一起。以前我们办各种演讲的时候，June Cohen 非常积极地参与进来，一些外部人士还建议我们在网上放点东西。一开始的时候是当作试验。最初的 6 个演讲上线之后，我看到了人们非常强烈的反应，感到非常吃惊。有了人们的热情回应，我们也变得兴奋起来。基于这些观众的电子邮件，我们立即将 TED 从原来单纯的实体大会转换成网络盛会。

如果你是个比较传统且有控制欲的人，会希望自己可以衡量、知晓所有事情，不愿冒任何风险，那么你肯定不会像我们这么做。你会觉得身上的压力难以承受。所以你要解决的核心问题是在一定程度上接受混乱状态，同时认识到在这段时间里，一点点“放手”的心态就能够产生难以置信的力量。有舍才能有得。有时，“放手”处处让人心神不宁，不过你要学会适应。

女士们、先生们，如果你想要一个毫无错误的维基百科，那你就根本不会有维基百科。接受现实，然后继续前进。我的意思是说，是的，要有规则，把这些规则尽量公布给大众。让社群来帮助你执行这些规则，不断迭代改进。不需要告诉人们不能做什么，但是你可以向他们展示怎样做才能变得更棒。我们帮助人们了解怎样才能成功创新，怎样才能成功指导演讲者。我们尽快把知识传播出去，让人们彼此学习。

问：你有没有过失败经历？

整个 TEDx 实验就是建立在失败或者说一种失败的形式之上的。我们举办了盘古日活动 (Pangea Day)，让世界各地的人在同一时间看电影。那是个很棒的活动，在很多方面都产生了美妙的体验，但是从财务角度来看，

《创意型领袖：从CEO到DEO》讲解如何将设计能力和思维融入企业管理和创新。随着互联网时代的到来，人类社会无时无刻不在发生天翻地覆的变化，人们对于管理的认识也在变化。设计能力被视为未来管理者的一项重大素质。本书两位作者以设计师和管理者的双重身份、以多年的亲身体会讲述设计在企业管理和创新中的应用。本文节选自《创意型领袖：从CEO到DEO》。

却是个惨痛的失败。我们没有足够的赞助商，那个项目真是难以想象地烧钱。但是其中有一点经验既不昂贵也很成功，那就是允许人们以自组织的方式独立组织活动。有一千五百人已经这样做了。

我们注意到了这一点，进而思考也许可以将 TED 体验带到草根阶层，但又担心造成品牌的滥用——我们很珍惜自己的品牌，绝不会滥用。所以，我们聚在一起进行头脑风暴。突然某一刻，一个点子“啪”地出现，这就是 TEDx。TEDx 感觉上与 TED 非常相近，人们会争先恐后地参加这些活动，但是又与 TED 有一定差别，因为它是自组织的。这种感觉很对头，我们强烈感觉到应该试上一试。

一个优秀的协作团队，共同努力创造 TED 体验

> 接受混乱状态，同时认识到在这段时间里，一点点‘放手’的心态就能够产生难以置信的力量。

问：你希望别人都知道你的哪三点？

我希望别人知道我是个永不退休的人。我会一直参与到世界上各种各样的想法中去，直到死去。当一个人不能再参与其中的时候，就该离开这个世界了。我希望别人认为我是个慷慨的人。在这个相互关联的世界里，你有理由认为大方一些其实是明智之举，因为很多事物传播得很快。你对别人慷慨，别人也更乐意回报给你慷慨。慷慨是有传染性的，而且我认为它也是 TED 能够取得成功的核心。最后，我希望别人把我视为一个梦想家，一个对新鲜想法着迷的家伙。 ■

九　卦

橡皮鸭子解决问题法

作者 / Jeff Atwood

程序员，著名博主，Stack Overflow/Stack Exchange & Discourse 联合创始人及发起者。

在 Stack Exchange 上，**我们一直强调，提交问题的人应该在提问前多花点时间研究一下他们的问题，而且我们对此非常偏执。**就是说，当你提交问题时，你应该…

- 描述问题要足够详细，以便我们能跟上你的思路。提供必要的背景信息，帮助我们理解发生了什么事情，即使我们不是你所在领域的专家。
- 告诉我们为什么你需要知道这个问题的答案。是什么让你找到这儿来寻找答案的？你提的问题是出于好奇心，还是在某个项目上遇到了阻碍？我们不需要知道你全部的故事细节，这样做，只是需要在这儿给我们一些上下文的提示。
- 分享你在该问题上面所做的研究。迄今为止你发现了什么？为什么它不能有效工作？如果你没有做任何研究 … 你应该提交这个问题吗？如果你邀请我们花费宝贵的时间帮助你，你应该花同样合理的宝贵时间设计一个像样的问题，唯有这样才是公平的。帮助我们也是帮助你自己！

我们有一个非常好的如何提问页面（How to Ask page）解释了这些事宜。（并且在 Stack Overflow 上，由于问题太多，我们的确要求新用户在提交他们的第一个问题之前去访问那个页面。作为一个新用户在提交第一个问题前，你自己就能看到这个页面。）

我们极力避免的，也是最最重要的，是那些无法回答的问题。这些问题对任何人都没有帮助，但是若任其发展却可以毁掉一个问答网站，将其

译者 / 傅俊杰

Qingniu/ 青牛，码农，复唧唧联合创始人，业余时间喜欢研究中国传统文化。博客@明珠夜话。

变成一个虚拟的鬼城。在 Stack Exchange 上，那些缺乏背景信息和上下文以至于不能被合理回答的问题，将被立即关闭，然后如果情况不能得到改进的话，最终将被删除。

正如我之前所说的，我们对此非常偏执。我们认为，通过教授橡皮鸭子问题解决法 (Rubber Duck problem solving)，是一个明确地帮助你自助解决问题的好理由。而且，这样做一直以来都非常有效。在数年的时间里，我已经从 Stack Overflow 或者 Stack Exchange 其它子站上得到了大量的反馈，就是说，在以这样的方式撰写具体问题的过程中，最终他们想出了自己问题的答案。

这种事已经非常司空见惯了。不信的话你自己看：

当我解决了自己的问题，我该如何感谢社区呢？

迄今为止我只发布过一个问题，而且差点提交了另一个。这两次经历，最终都是在我撰写问题的过程中，我至少部分地解答了自己要问的问题。我之所以能够想出答案，这要归功于社区以及描述问题的过程。当我描述问题时，并没有与答案有关的明确线索，但当问题写完之后，却让我产生了考虑该问题的另一条思路。

为什么正确地描述你的问题往往会自助地产生答案呢？

我不知道这已经发生多少次了：

- 我有一个问题
- 我决定把它放到 stackoverflow 上面
- 我粗略地将问题写下来
- 我知道该问题描述的不好
- 我又花费了 15 分钟时间重新思考该如何描述问题
- 我意识到自己正在一个完全错误的方向上解决问题
- 我再次从头开始，并且迅速找到了问题的解决方案

上述这样的事情是否也发生在你身上呢？有时候，提出正确的问题，似乎问题就已经解决一半了。

开始提交一个问题，实际上是在帮助我调试我自己的问题

开始提出一个问题，实际上是在帮助我调试我自己的问题，尤其为了得到像模像样的的答案时，我们总是会提供足够详细的与问题相关内容。这样的事情以前是否在别的人身上发生过？

这不是一个什么新东西，只要给予足够的时间，每一个互联网社区似乎都能找到自己的解决问题方式，但是“向鸭子提问”的确是一个非常强大的解决问题的技巧和方法。

鲍勃指着办公室的角落，“在那儿”，他说，“有一只鸭子。我希望你向那只鸭子提出你的问题。”

我看着那只鸭子。事实上，它吃的很饱，一动不动。即便它还能动，也不可能是一个有关设计信息的有效来源。我看着鲍勃。鲍勃看起来很认真。当然，他是我的上司，我不想失去这份工作。

我摇摇晃晃地向鸭子走了过去，并且站在了它的旁边。我开始低下头和鸭子交流，看起来有点像在祈祷。“你，”鲍勃问，“在干什么？”

鲍勃的一位上司正巧在他的办公室。他开心地大笑起来。

“安迪，”他说，“我不是让你向鸭子祈祷，我是让你向鸭子问问题。”我舔了舔我的嘴唇。“大声吗？”我说。

“大声，”鲍勃坚定地说。

我清了清嗓子。“鸭子，”我要开始了。

“它的名字叫小鲍勃，”鲍勃的那位上司补充了一句。我冷冷地瞥了他一眼。

“鸭子，”我继续说，“我想知道，当你使用 U 形夹挂钩，在管道头部排水时，怎样防止喷水管弹出 U 形夹，导致管…”

在我向鸭子问问题的过程中，我得到了问题的答案。U 形夹挂钩是悬挂在螺纹杆上面的。如果管道安装工将螺纹杆锯到一定长度，使其紧紧顶在喷水管顶部的话，实际上管子已经被固定在挂钩上了，这样也就防止了管子的突然脱落。

我转头看向鲍勃。鲍勃在点头。“你知道答案了，是这样吗？”他说。

“应该把螺纹杆紧紧顶在管子的顶部，”我说。

“完全正确，”鲍勃说。“下次你再有问题，我还让你来这儿继续问鸭子，而不是问我。大声地问它。如果你仍旧无法得到答案，你再来问我。”

“好的，”我说，然后就回去继续工作了。

我很喜欢这个特殊的故事，因为它讲解地十分清楚 - **解决橡皮鸭问题的关键部分是向这个虚构的人物或静物问一个深入且足够详尽的问题。**是的，

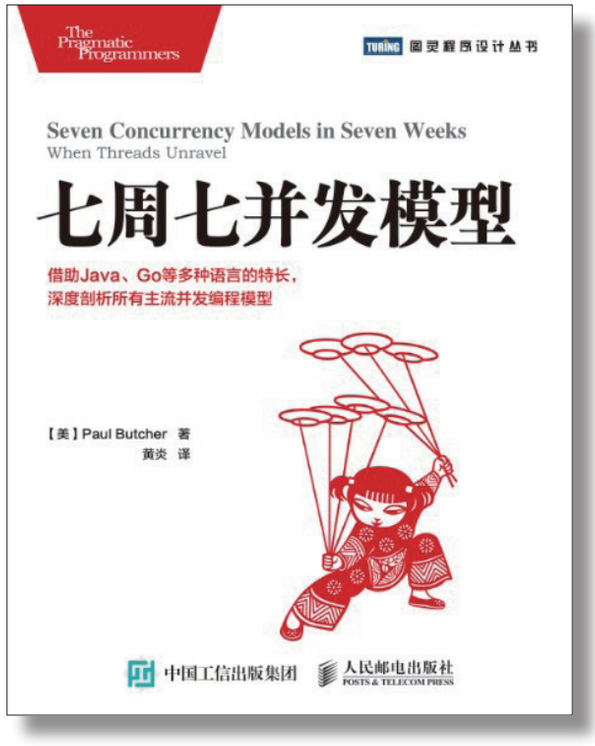
即使你最终没能解决这个问题，起码你可以意识到自己犯了一些愚蠢的错误。向虚构的人物问问题，要一步一步来，并且要尽量详细，这种尝试经常能让你找到问题的答案。如果你不愿意花费精力去完整地说明问题以及试图解决该问题的过程，那么在你询问其他人之前，你就不能得到深度思考你的问题所带来的好处。

如果你在编程上缺少伙伴（但是你绝对应该有），你可以利用橡皮鸭问题解决法这样的技巧找出答案，当然这全部要靠你自己，或者利用伟大的互联网在社区中寻求答案。即使你没有得到你想要的答案，强制自己完整地描述自己的问题 - 最好以书面形式 - 往往就会产生新的认识和发现。■

图灵社区原文

英文原文：Rubber Duck Problem Solving

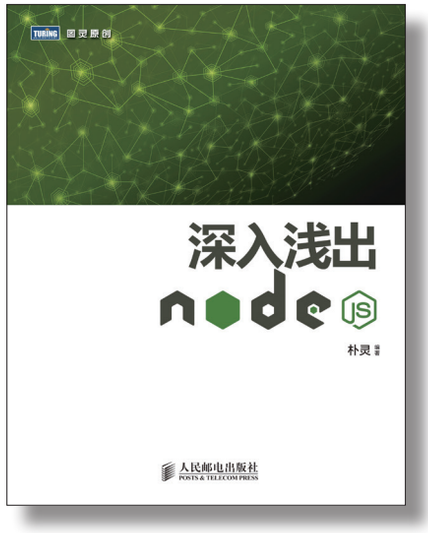
书　榜



七周七并发模型

作者：Paul Butcher
译者：黄炎
书号：978-7-115-38606-9
图灵社区推荐：11

并发编程近年逐渐热起来，Go等并发语言也对并发编程提供了良好的支持，使得并发这个话题受到越来越多人的关注。本书延续了《七周七语言》的写作风格，通过以下七个精选的模型帮助读者了解并发领域的轮廓：线程与锁，函数式编程，Clojure，actor，通信顺序进程，数据级并行，Lambda架构。书中每一章都设计成三天的阅读量。每天阅读结束都会有相关练习，巩固并扩展当天的知识。每一章均有复习，用于概括本章模型的优点和缺陷。本书适合所有想了解并发的程序员。



2 深入浅出Node.js

作者：朴灵

书号：978-7-115-33550-0

图灵社区推荐：18

Node.js让JavaScript在服务器端焕发生机，如果你是前端工程师，这会是你迈向全端工程师的关键一步。——玉伯，支付宝高级技术专家

3 超弦理论：探究时间、空间及宇宙的本原

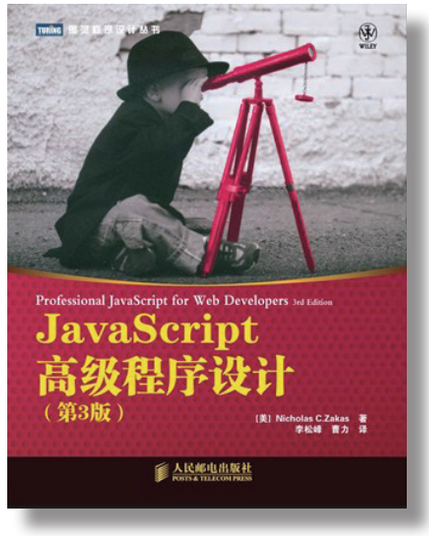
作者：大栗博司

译者：逸宁

书号：978-7-115-37386-1

图灵社区推荐：6

“空间”和“时间”究竟是什么？物质的本原不是点，而是弦？超弦理论是继牛顿力学、爱因斯坦相对论之后，时空概念的“第三次革命”。



4 JavaScript高级程序设计（第3版）

作者：Nicholas C.Zakas

译者：李松峰　曹力

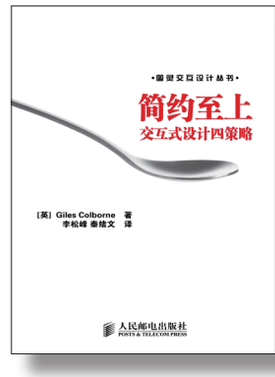
书号：978-7-115-27579-0

图灵社区推荐：38

一幅浓墨重彩的语言画卷，一部推陈出新的技术名著。全能前端人员必读之经典，全面知识更新必备之佳作。

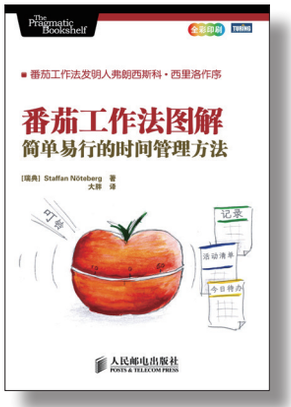
5 Android编程权威指南

作者：Brian Hardy，Bill Phillips
译者：王明发
书号：978-7-115-34643-8
图灵社区推荐：23



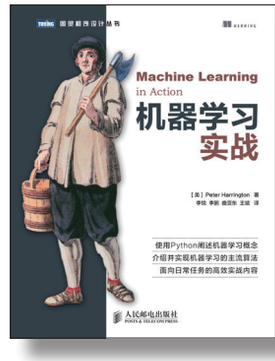
6 简约至上：交互式设计四策略

作者：Giles Colborne
译者：李松峰　秦绪文
书号：978-7-115-24324-9
图灵社区推荐：10



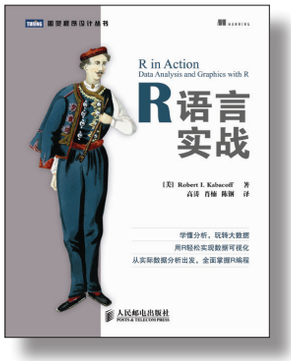
7 番茄工作法图解：简单易行的时间管理方法

作者：Staffan Noteberg
译者：大胖
书号：978-7-115-24669-1
图灵社区推荐：14



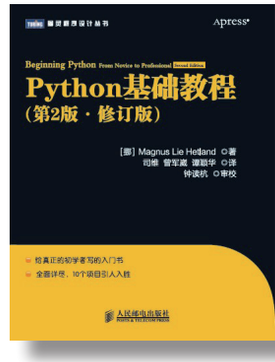
8 机器学习实战

作者：Peter Harrington
译者：李锐　李鹏　曲亚东　王斌
书号：978-7-115-31795-7
图灵社区推荐：34



9 R语言实战

作者：Robert I. Kabacoff
译者：高涛　肖楠　陈钢
书号：978-7-115-29990-1
图灵社区推荐：21



10 Python基础教程（第2版·修订版）

作者：Magnus Lie Hetland
译者：司维　曾军崴　谭颖华
书号：978-7-115-35352-8
图灵社区推荐：4

电子书榜

1. 七周七并发模型
   ——借助Java、Go等多种语言的特长，深度剖析所有主流并发编程模型。

2. AngularJS权威教程
   ——资深全栈工程师的代表性著作，由拥有丰富经验的国内AngularJS技术专家执笔翻译。

3. Node与Express开发
   ——Express在根本没有框架和有一个健壮的框架之间找到了平衡，让你自由选择架构。

4. 深入浅出Node.js
   ——阿里巴巴一线Node开发者最真实的经验。

5. 发布！软件的设计与部署
   ——针对详细展示软件发布前可能出现的种种问题以及相应的解决之道。

6. 你不知道的JavaScript（上卷）
   ——直面当前JavaScript开发者不求甚解的大趋势，深入理解语言内部的机制。

7. HTML5秘籍（第2版）
   ——你的运气不错：这本书涵盖了关于HTML5的一切！

8. 系统化思维导论（25周年纪念版）
   ——软件思想家温伯格的非凡之作，内容涵盖整个软件开发周期。

9. Java 8函数式编程
   ——如果你想尽快了解Java 8新特性，写出简单干净的代码，那么本书不容错过。

10. WEB+DB PRESS 中文版 01
    ——旨在帮助程序员更实时、深入地了解前沿技术，扩大视野，提升技能。

Joel、Apress、网志和网志书

作者 / Gary Cornell
Apress 出版公司创始人。

“很久以前，在一个很遥远、很遥远的星系中……”好吧，实际上没有那么久啦，那是在 2000 年接近年底的时候，Apress 出版公司正式运营刚满一年。当时，我们只是一家非常小的计算机书籍出版商，毫无名气。那一年，我们计划出版的书籍只有很少几本，大概只相当于 Apress 现在一个月的出版量。

那时，我苦苦学习如何成为一个出版商，可能花费了过多的时间，忙于浏览网站和编写程序。有一天，我偶然来到了一个叫作“乔尔谈软件”（Joel on Software）的网站，发现网站的主人是一个观点鲜明的家伙，他的写作风格有点不寻常，很聪明并且还有意挑战一些传统观念。最特别的是，那时他正在写一组系列文章，批评大多数软件的用户界面是多么糟糕。总的来说，这主要是因为程序员们对用户的实际需求几乎毫无所知——用乔尔和我经常使用的话说，这叫作“bupkis”（近乎没有），这是一句来源于意第绪语 的纽约土话。我同许多其他人一样，被乔尔的这组系列文章以及其他几篇随笔吸引住了。

然后，我就冒出了一个想法：我是出版商，我喜欢读他的文章，那么为什么不出书呢？我给乔尔写信，自我介绍了一番。虽然他起初有些怀疑，但是我不知怎地就说服他相信，如果他将那组用户界面的系列文章写成一本书，会有很多人购买，我和他都会赚到很多钱。（当然，那是发生在很久以前的事情，那时 FogBugz 还没有变得像今天这样成功，乔尔也

还不是一个令人羡慕的收入颇丰的演讲者。不过，那时我们都比现在年轻，并且正如你想的那样，比现在穷得多。）

闲话少说，乔尔后来又为新书加入一些新内容，使得它更具吸引力，我觉得也更有销路了。突然之间，Apress就必须考虑如何出版它的第一本全彩书籍了。《面向程序员的用户界面设计》（User Interface Design for Programmers）正式出版是在2001年6月21日。现在，它被公认为有史以来第一本“网志书”（blook）。令计算机图书行业和我本人有些震惊的是，按照当时的畅销标准，它竟然成了一本很优秀的畅销书。顺便说一句，直到今天，它仍然在不断重印，仍然卖得非常好，仍然值得一读。（不过，作为乔尔的出版商，而不是作为朋友，我想对他说：你是不是该考虑出个修订版了？）

不过，还是有人出来说，《面向程序员的用户界面设计》并不是一本纯粹的“网志书”，因为加入了“太多的”网站上没有的新内容，使得这本书看上去更像一个混合体——我的看法是，这正同它的先锋地位相适合。

短短几年之后，“乔尔谈软件”成了全世界程序员中最著名的网志，原因当然是乔尔一直不停地写作那些非常有趣的文章。其中最著名的大概是那篇经典文章《微软公司如何在API战争中失利》（How Microsoft Lost the API War）。据我所知，这篇文章着实把微软的开发部门折腾得够呛。

这样，我就有了另一个想法：将乔尔最好的那些文章收集起来，再出一本书，不作大的变动，除了加上一篇字数很少的前言，只要乔尔觉得合适就可以。这样一本书的名字就叫作Joel on Software 。即使书中98%的内容都能在互联网上找到，即使人们认定Apress出版这样一本书一定是疯了，它还是在2004年底出版了。今天，这本书已经印刷了10次，而且依然是一本畅销书。

为什么呢？人们的阅读习惯并没有改变，在像品尝美味的巧克力糖果一样品味乔尔的文章时，很多人仍然习惯于看书而不是看浏览器。

《软件随想录 卷2》是一部关于软件技术、人才、创业和企业管理的随想文集，作者以诙谐幽默的笔触将自己在软件行业的亲身感悟娓娓道来，观点新颖独特，内容简洁实用。全书分为36讲，每一讲都是一个独立的专题。

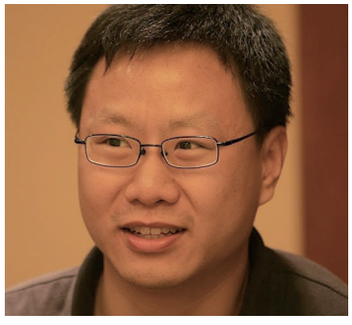
但是，乔尔并没有因此停下来，他依然在努力思索如何才能更好地编程，或者怎样招聘到优秀的程序员，他也没有停止用自己的观点挑战传统看法。所以，我说服他，现在可以出一本续集，收录2004年底上一本书出版之后的那些“乔尔的精华文章”。

结果就是你现在手里拿的这第二本文集，乔尔的观点、随感以及偶尔的夸夸其谈都浓缩在了他才华横溢的文章之中。除了少量的编辑加工，原文几乎毫无变动，但是同显示器屏幕或者Kindle阅读器 相比，你确实以一种非常不同的形式拥有了最新的“乔尔的精华文章”，现在这被称为“网志书”。（我要对乔尔说，我很希望你像中意第一本集子里那些文章那样，中意这本集子里的文章。）

这本书同第一本一样，有着不同寻常的封面和副标题。这是因为乔尔和我都是藏书爱好者（好吧，乔尔才是藏书爱好者，我是藏书狂人）。17世纪和18世纪那些经典著作的印刷商，为了让他们的书变得生动，往往会做一些特别的设计，我们两人都非常喜欢这一类东西。在第一本Joel on Software的封面上，我们向伯顿的《忧郁的剖析》致敬；这一本的封面上，我们向霍布斯的《利维坦》致敬，它的封面很著名，一个巨人由许多个小人组成。乔尔和我都感到这个隐喻很不错，可以暗示程序是如何编写完成的：宏伟的整体由个体组成，并且个体是关键。

最后，是一点很个人化的说明：尽管现在乔尔的名气很大，但他依然是一个很朴实的人，或者再一次用我们共同的土话说，是一个真正的“mensch”（好人）。我非常骄傲，我有这样一个好朋友。 ■

《算法的乐趣》：从乐趣出发阐述算法



作者 / 王益

王益，LinkedIn 高级分析师。他曾在腾讯担任广告算法和策略的技术总监，在此期间他发明了并行机器学习系统“孔雀”，它可以从数十亿的用户行为或文本数据中学习到上百万的潜在主题，该系统被应用在腾讯可计算广告业务中。在此之前，他在 Google 担任软件工程师，并开发了一个分布式机器学习工具，这个工具让他获得了 2008 年的“Google APAC 创新奖”。王益曾在

读《算法的乐趣》的乐趣超出了我的预料。

说到算法，大部分计算机专业的同学的第一反应估计是 MIT 出版社的经典教材《算法导论》(Introduction to Algorithms)。这是一本由浅入深的好书，堪称“神书”——别看书挺厚，但是对初学者来说很难弄懂的问题也娓娓道来，让人看一遍就明白；而且作者用最简单的英语词汇和句法写书，以至于世界各地的学生们，不需要英语很好，即可读懂原版。只是看完这本大部头之后，总有一些意犹未尽的感觉——对我们日常生活中常见的比如音乐播放器里以及电子游戏里的算法并没有太多介绍。而这些正是《算法的乐趣》中主要的部分。

在 Amazon 上，另外两本排名靠前的经典算法教材是 Jon Kleinberg 的《算法设计》(*Algorithm Design*) 和 Steven S. Skiena 的《算法设计手册》(*The Algorithm Design Manual*)。这两本出自名家之手的教材和很多教材一样，按照算法的类型或者背后的设计思路来组织内容。这是教材应该做的，“授人以鱼不如授人以渔”，传授思路而不是算法本身是教材的写作目的。可是算法最有意思的地方首先在于算法本身，因为算法是为了解决实际问题而设计的，所以让大家认识到算法奥妙的自然顺序应该是先展示有趣的问题，再展示优雅的算法，最后归纳设计思路。而这正是《算法的乐趣》吸引人的地方。 说到乐趣，总让我想起我学习和使用数学知识的经历。虽然我的学位是关于统计机器学习的，而且毕业后一直从事相关工作，但是我从小学一年级到博士第三年都对数学毫无兴趣，因为学校的老师和数学成绩好的同学都说不明白数学的用处，

清华大学和香港城市大学学习，并取得了清华大学机器学习和人工智能的博士学位。此外，他还是IEEE的高级会员，著有《推荐系统实践》。

以至于我一直以为数学的作用只是锻炼和展示自己的聪明，博得老师的表扬，成为陈景润那样为国争光的英雄。而这些对我实在没有吸引力，而且我认为恐怕对绝大部分学生都没什么吸引力。

我认识到数学的价值，是因为在博士第三年把研究方向换到了统计机器学习。在读教材的时候，我曾想验证“数学无用”，所以费尽心力地试图写一个程序来判断一个64×64像素的图片里到底是数字“1”还是数字“9”，却发现无论如何也很难写一个有效的程序；可是利用教材里的数学知识却能设计和“训练”一个数学模型，准确地识别任意字符。因为体会到了数学的用处，我兴奋地用了一年的时间复习大学本科的数学课程，然后才读懂了人工智能的专业教材和论文。此后才有所创新，发表论文，到博士毕业。这整个过程用了三年，而效果超过了之前19年数学教育的效果。

在这个过程中，我自然而然地开始注意数学知识的前因（比如为什么人们会关注长度、面积，怎么会有人考虑勾股定理这样的规律）以及后果（今天的数学知识能给物理学和机器智能带来什么样的帮助），也开始归纳和了解各种数学系统背后的规律，能体会哥德尔定理阐述的意思。当然，也破除了“数学是各种科学之母”之类的迷信，数学当然不是“科学之母”，而是“科学之子”，是先有物理学、力学和天文学，才有的数学；先有应用场景后有工具，先有探索后有归纳。

算法也是如此。先有工程问题需要解决，算法是解法，设计算法是寻求解法。虽然算法作为一门科学是归纳寻求解法的思路，但学习这种归纳法的前提是能体会各种具体算法的用处和效果。意识到这一点，自然也就破除了诸如“学好数学才能学好算法”之类的迷信。而把算法解决的各种有趣问题罗列出来，把算法的可爱之处展示给愿意发现和体会生活中点滴乐趣的读者们，正是《算法的乐趣》在技术价值之外的一层社会价值。

十年前，当我们坐在课堂里学习算法的时候，我们学到的是如何用人脑寻求解法，然后把解法写成程序，让计算机照着执行去解决问题。这是“经

《算法的乐趣》从一系列有趣的生活实例出发，全面介绍了构造算法的基础方法及其广泛应用，生动地展现了算法的趣味性和实用性。全书分为两个部分，第一部分介绍了算法的概念、常用的算法结构以及实现方法，第二部分介绍了算法在各个领域的应用，如物理实验、计算机图形学、数字音频处理等。其中，既有各种大名鼎鼎的算法，如神经网络、遗传算法、离散傅里叶变换算法及各种插值算法，也有不起眼的排序和概率计算算法。讲解浅显易懂而不失深度和严谨，对程序员有很大的启发意义。书中所有的示例都与生活息息相关，淋漓尽致地展现了算法解决问题的本质，让你爱上算法，乐在其中。

典算法”。最近十几年，随着Internet产业的兴起，Internet服务在不断取代原来由人提供的服务，这就要求机器拥有一定程度上能取代人的“智能”。在搜索引擎、推荐系统和广告系统等各个领域里，类似上述“识别数字”的问题越来越多，而人工智能和机器学习的应用也越来越深入我们的生活。机器学习算法的设计目标和“经典算法”不同——不是让人来想解法，而是让计算机从数据归纳知识——有了这些知识，计算机就能自己寻求解法。

虽然经典算法和机器学习算法之间的差别大得如同一场革命，但是由经典而入机器学习的过程却是自然而然的。比如《算法的乐趣》中介绍的曲线拟合问题，就是supervised learning（有监督学习），而音乐播放器里常用的傅里叶变换和其他时域频域变换则是unsupervised learning（无监督学习）的技术基础，棋类游戏算法是博弈论和reinforcement learning（强化学习）的经典例子。我常见有朋友从读数学教材开始探索机器学习和人工智能算法，也常看到有人不堪忍受长时间缺乏乐趣的探索以至于半途而废。如果是这样，也许不如从《算法的乐趣》开始这个探索过程。

我曾经以为从乐趣出发阐述算法的书会从西方发芽，没想到先看到了一本中文书。这真超出了我的预料。 ■

图灵社区 出品

出版人：武卫东
编辑：李盼
顾问：杨帆
设计：大胖

地址：北京市朝阳区北苑路13号院领地OFFICE C座603室
电话：010-51095181
微博：weibo.com/ituring
Email: ebook@turingbook.com