码农 the code maker

12

JavaScript

第二季

iTuring.cn

目　录

九卦

践行

出版的未来

书榜

妙评

编者的话

Live Long and Prosper

编者 / 李盼

Brendan Eich原本进入网景公司是希望做Scheme语言的开发，但是却接到了一个不喜欢的任务，于是他在10天的时间里仓促完成了一门语言的设计。处于Java阴影下的这门语言获得了它最终的名字：JavaScript。它于1999年冻结了，接下来本应走向灭亡。但Ajax的横空出世改变了这一切，JavaScript变成了世界上最重要的编程语言。ECMAScript标准不断推进，令语言更加精炼简洁，不停地去芜存菁。Node不仅满足JavaScript同时运行在前后端，而且性能还十分高效。大量的前后端工程师加入了Node的开发阵营，GitHub上JavaScript成为了最活跃的开发语言。

JavaScript经历了Web相关技术的各种变革，目睹了网景浏览器的衰落，IE的后来居上，Firefox和Chrome的逆袭，各类RIA技术的风起云涌……前端工程师徐飞经历了前端技术快速变革的这10年，《前端开发技术的发展》和《我在中兴软创这9年》这两篇文章一起读来，定会别有一番风味。

在变化中生存的也包括曾经的“黑客教父”万涛。他见证了黑客团体的兴旺衰落，也曾被称为互联网十大恶人之一。如今他找到了一条新路，用公益实现他的黑客信条：自由、平等、分享、互助。但是他的做事原则中从此多了一条：不要因为自己的理想而产生新的社会问题。

本期《码农》群星荟萃，你可以看到Joel Spolsky如何用以色列伞兵的知识破解微软的伎俩；Zakas大神自出版一年的心得体会;语言狂人Ola Bini心目中最有趣的编程语言……无论你是否码农，是否善用JavaScript，都祝您阅读愉快！

专题：JavaScript第二季

前端开发技术的发展

作者 / 徐飞

徐飞，05年至今就职于中兴软创，致力于企业软件前端工程化的发展。徐飞坚信：作为技术人员应当乐于助人，有好东西要主动拿出来分享，资产阶级知识分子垄断电子书的现象再也不能出现了！微博@民工精髓V，图灵社区ID:民工精髓。

前端开发技术，从狭义的定义来看，是指围绕HTML、JavaScript、CSS这样一套体系的开发技术，它的运行宿主是浏览器。从广义的定义来看，包括了：

- 专门为手持终端设计的类似WML这样的类HTML语言，类似WMLScript这样的类JavaScript语言。
- VML和SVG等基于XML的描述图形的语言。
- 从属于XML体系的XML，XPath，DTD等技术。
- 用于支撑后端的ASP，JSP，ASP.net，PHP，nodejs等语言或者技术。
- 被第三方程序打包的一种类似浏览器的宿主环境，比如Adobe AIR和使用HyBird方式的一些开发技术，如PhoneGap（它使用Android中的WebView等技术，让开发人员使用传统Web开发技术来开发本地应用）
- Adobe Flash，Flex，Microsoft Silverlight，Java Applet，JavaFx等RIA开发技术。

本文从狭义的前端定义出发，探讨一下这方面开发技术的发展过程。

从前端开发技术的发展来看，大致可以分为以下几个阶段：

一、刀耕火种

1. 静态页面

最早期的Web界面基本都是在互联网上使用，人们浏览某些内容，填写几个表单，并且提交。当时的界面以浏览为主，基本都是HTML代码，有时候穿插一些JavaScript，作为客户端校验这样的基础功能。代码的组织比较简单，而且CSS的运用也是比较少的。

最简单的是这样一个文件：

```
<html>
    <head>
        <title> 测试一 </title>
    </head>
    <body>
        <h1> 主标题 </h1>
        <p> 段落内容 </p>
    </body>
</html>
```

2. 带有简单逻辑的界面

这个界面带有一段JavaScript代码，用于拼接两个输入框中的字符串，并且弹出窗口显示。

```
<html>
    <head>
        <title> 测试二 </title>
    </head>
    <body>
        <input id="firstNameInput" type="text" />
        <input id="lastNameInput" type="text" />
        <input type="button" onclick="greet()" />
        <script language="JavaScript">
        function greet() {
            var firstName = document.getElementById("firstName
Input").value;
            var lastName = document.getElementById ("lastName
Input").value;
            alert("Hello, " + firstName + "." + lastName);
        }
        </script>
    </body>
</html>
```

3. 结合了服务端技术的混合编程

由于静态界面不能实现保存数据等功能，出现了很多服务端技术，早期的有CGI（Common Gateway Interface，多数用C语言或者Perl实现的），ASP（使用VBScript或者JScript），JSP（使用Java），PHP等等，Python和Ruby等语言也常被用于这类用途。

有了这类技术，在HTML中就可以使用表单的post功能提交数据了，比如：

```
<form method="post" action="username.asp">
      <p>First Name: <input type="text" name="firstName"
/></p>
    <p>Last Name: <input type="text" name="lastName" /></
p>
    <input type="submit" value="Submit" />
</form>
```

在这个阶段，由于客户端和服务端的职责未作明确的划分，比如生成一个字符串，可以由前端的JavaScript做，也可以由服务端语言做，所以通常在一个界面里，会有两种语言混杂在一起，用<%和%>标记的部分会在服务端执行，输出结果，甚至经常有把数据库连接的代码跟页面代码混杂在一起的情况，给维护带来较大的不便。

```
<html>
    <body>
        <p>Hello world!</p>
        <p>
        <%
            response.write("Hello world from server!")
        %>
```

```
        </p>
    </body>
</html>
```

4. 组件化的萌芽

这个时代，也逐渐出现了组件化的萌芽。比较常见的有服务端的组件化，比如把某一类服务端功能单独做成片段，然后其他需要的地方来include进来，典型的有：ASP里面数据库连接的地方，把数据源连接的部分写成conn.asp，然后其他每个需要操作数据库的asp文件包含它。

上面所说的是在服务端做的，浏览器端通常有针对JavaScript的，把某一类的Javascript代码写到单独的js文件中，界面根据需要，引用不同的js文件。针对界面的组件方式，通常利用frameset和iframe这两个标签。某一大块有独立功能的界面写到一个html文件，然后在主界面里面把它当作一个frame来载入，一般的B/S系统集成菜单的方式都是这样的。

此外，还出现了一些基于特定浏览器的客户端组件技术，比如IE浏览器的HTC（HTML Component）。这种技术最初是为了对已有的常用元素附加行为的，后来有些场合也用它来实现控件。微软ASP.net的一些版本里，使用这种技术提供了树形列表，日历，选项卡等功能。HTC的优点是允许用户自行扩展HTML标签，可以在自己的命名空间里定义元素，然后，使用HTML，JavaScript和CSS来实现它的布局、行为和观感。这种技术因为是微软的私有技术，所以逐渐变得不那么流行。

Firefox浏览器里面推出过一种叫XUL的技术，也没有流行起来。

二、铁器时代

这个时代的典型特征是Ajax的出现。

1. AJAX

AJAX其实是一系列已有技术的组合，早在这个名词出现之前，这些技术的使用就已经比较广泛了，GMail因为恰当地应用了这些技术，获得了很好的用户体验。

由于Ajax的出现，规模更大，效果更好的Web程序逐渐出现，在这些程序中，JavaScript代码的数量迅速增加。出于代码组织的需要，“JavaScript框架”这个概念逐步形成，当时的主流是prototype和mootools，这两者各有千秋，提供了各自方式的面向对象组织思路。

2. JavaScript基础库

Prototype框架主要是为JavaScript代码提供了一种组织方式，对一些原生的JavaScript类型提供了一些扩展，比如数组、字符串，又额外提供了一些实用的数据结构，如：枚举，Hash等，除此之外，还对dom操作，事件，表单和Ajax做了一些封装。

Mootools框架的思路跟Prototype很接近，它对JavaScript类型扩展的方式别具一格，所以在这类框架中，经常被称作“最优雅的”对象扩展体系。

从这两个框架的所提供的功能来看，它们的定位是核心库，在使用的时候一般需要配合一些外围的库来完成。

jQuery与这两者有所不同，它着眼于简化DOM相关的代码。例如：

●DOM的选择

jQuery提供了一系列选择器用于选取界面元素，在其他一些框架中也有类似功能，但是一般没有它的简洁、强大。

```
$("*")                  // 选取所有元素
$("#lastname")          // 选取 id 为 lastname 的元素
$(".intro")             // 选取所有 class="intro" 的元素
$("p")                  // 选取所有 &lt;p&gt; 元素
$(".intro.demo")        // 选取所有 class="intro" 且 class="demo" 的元素
```

●链式表达式：

在jQuery中，可以使用链式表达式来连续操作dom，比如下面这个例子：

如果不使用链式表达式，可能我们需要这么写：

```
var neat = $("p.neat");
neat.addClass("ohmy");
neat.show("slow");
```

但是有了链式表达式，我们只需要这么一行代码就可以完成这些：

```
$("p.neat").addClass("ohmy").show("slow");
```

除此之外，jQuery还提供了一些动画方面的特效代码，也有大量的外围库，比如jQuery UI这样的控件库，jQuery mobile这样的移动开发库等等。

3. 模块代码加载方式

以上这些框架提供了代码的组织能力，但是未能提供代码的动态加载能力。动态加载JavaScript为什么重要呢？因为随着Ajax的普及，jQuery等辅助库的出现，Web上可以做很复杂的功能，因此，单页面应用程序（SPA，Single Page Application）也逐渐多了起来。

单个的界面想要做很多功能，需要写的代码是会比较多的，但是，并非所有的功能都需要在界面加载的时候就全部引入，如果能够在需要的时候才加载那些代码，就把加载的压力分担了，在这个背景下，出现了一些用于动态加载JavaScript的框架，也出现了一些定义这类可被动态加载代码的规范。

在这些框架里，知名度比较高的是RequireJS，它遵循一种称为AMD（Asynchronous Module Definition）的规范。

比如下面这段，定义了一个动态的匿名模块，它依赖math模块

```
define(["math"], function(math) {
    return {
        addTen : function(x) {
            return math.add(x, 10);
        }
    };
});
```

假设上面的代码存放于adder.js中，当需要使用这个模块的时候，通过如下代码来引入adder：

```
<script src="require.js"></script>
<script>
    require(["adder"], function(adder) {
```

```
        // 使用这个 adder
    });
</script>
```

RequireJS除了提供异步加载方式，也可以使用同步方式加载模块代码。AMD规范除了使用在前端浏览器环境中，也可以运行于nodejs等服务端环境，nodejs的模块就是基于这套规范定义的。（修订，这里弄错了，nodejs是基于类似的CMD规范的）

三、工业革命

这个时期，随着Web端功能的日益复杂，人们开始考虑这样一些问题：

- 如何更好地模块化开发
- 业务数据如何组织
- 界面和业务数据之间通过何种方式进行交互

在这种背景下，出现了一些前端MVC、MVP、MVVM框架，我们把这些框架统称为MV*框架。这些框架的出现，都是为了解决上面这些问题，具体的实现思路各有不同，主流的有Backbone，AngularJS，Ember，Spine等等，本文主要选用Backbone和AngularJS来讲述以下场景。

1. 数据模型

在这些框架里，定义数据模型的方式与以往有些差异，主要在于数据的get和set更加有意义了，比如说，可以把某个实体的get和set绑定到RESTful的服务上，这样，对某个实体的读写可以更新到数据库中。另外一个特点是，它们一般都提供

一个事件，用于监控数据的变化，这个机制使得数据绑定成为可能。

在一些框架中，数据模型需要在原生的JavaScript类型上做一层封装，比如Backbone的方式是这样：

```
var Todo = Backbone.Model.extend({
    // Default attributes for the todo item.
    defaults : function() {
        return {
            title : "empty todo...",
            order : Todos.nextOrder(),
            done : false
        };
    },

    // Ensure that each todo created has `title`.
    initialize : function() {
        if (!this.get("title")) {
            this.set({
                "title" : this.defaults().title
            });
        }
    },

    // Toggle the 'done' state of this todo item.
    toggle : function() {
        this.save({
            done : !this.get("done")
        });
    }
});
```

上述例子中，defaults方法用于提供模型的默认值，initialize方法用于做一些初始化工作，这两个都是约定的方法，toggle是自定义的，用于保存todo的选中状态。

除了对象，Backbone也支持集合类型，集合类型在定义的时候要通过model属性指定其中的元素类型。

```
// The collection of todos is backed by *localStorage*
instead of a remote server.
var TodoList = Backbone.Collection.extend({
    // Reference to this collection's model.
    model : Todo,

     // Save all of the todo items under the '"todos-
backbone"' namespace.
     localStorage : new Backbone.LocalStorage("todos-
backbone"),

     // Filter down the list of all todo items that are
finished.
    done : function() {
        return this.filter(function(todo) {
            return todo.get('done');
        });
    },

     // Filter down the list to only todo items that are
still not finished.
    remaining : function() {
        return this.without.apply(this, this.done());
    },

     // We keep the Todos in sequential order, despite
being saved by unordered
    //GUID in the database. This generates the next order
number for new items.
    nextOrder : function() {
        if (!this.length)
            return 1;
        return this.last().get('order') + 1;
    },
```

```
     // Todos are sorted by their original insertion
order.
    comparator : function(todo) {
        return todo.get('order');
    }
});
```

数据模型也可以包含一些方法，比如自身的校验，或者跟后端的通讯、数据的存取等等，在上面两个例子中，也都有体现。

AngularJS的模型定义方式与Backbone不同，可以不需要经过一层封装，直接使用原生的JavaScript简单数据、对象、数组，相对来说比较简便。

2. 控制器

在Backbone中，是没有独立的控制器的，它的一些控制的职责都放在了视图里，所以其实这是一种MVP（Model View Presentation）模式，而AngularJS有很清晰的控制器层。

还是以这个todo为例，在AngularJS中，会有一些约定的注入，比如$scope，它是控制器、模型和视图之间的桥梁。在控制器定义的时候，将$scope作为参数，然后，就可以在控制器里面为它添加模型的支持。

```
function TodoCtrl($scope) {
    $scope.todos = [{
        text : 'learn angular',
        done : true
    }, {
        text : 'build an angular app',
        done : false
    }];
```

```
    $scope.addTodo = function() {
        $scope.todos.push({
            text : $scope.todoText,
            done : false
        });
        $scope.todoText = '';
    };

    $scope.remaining = function() {
        var count = 0;
        angular.forEach($scope.todos, function(todo) {
            count += todo.done ? 0 : 1;
        });
        return count;
    };

    $scope.archive = function() {
        var oldTodos = $scope.todos;
        $scope.todos = [];
        angular.forEach(oldTodos, function(todo) {
            if (!todo.done)
                $scope.todos.push(todo);
        });
    };
}
```

本例中为$scope添加了todos这个数组，addTodo，remaining和archive三个方法，然后，可以在视图中对他们进行绑定。

3. 视图

在这些主流的MV*框架中，一般都提供了定义视图的功能。在Backbone中，是这样定义视图的：

```
// The DOM element for a todo item...
var TodoView = Backbone.View.extend({
    //... is a list tag.
    tagName : "li",

    // Cache the template function for a single item.
    template : _.template($('#item-template').html()),

    // The DOM events specific to an item.
    events : {
        "click .toggle" : "toggleDone",
        "dblclick .view" : "edit",
        "click a.destroy" : "clear",
        "keypress .edit" : "updateOnEnter",
        "blur .edit" : "close"
    },

    // The TodoView listens for changes to its model, re-
rendering. Since there's
    // a one-to-one correspondence between a **Todo** and
a **TodoView** in this
     // app, we set a direct reference on the model for
convenience.
    initialize : function() {
        this.listenTo(this.model, 'change', this.render);
           this.listenTo(this.model, 'destroy', this.
remove);
    },

    // Re-render the titles of the todo item.
    render : function() {
            this.$el.html(this.template(this.model.
toJSON()));
            this.$el.toggleClass('done', this.model.
get('done'));
        this.input = this.$('.edit');
        return this;
    },
```

```
    //......

    // Remove the item, destroy the model.
    clear : function() {
        this.model.destroy();
    }
});
```

上面这个例子是一个典型的“部件”视图，它对于界面上的已有元素没有依赖。也有那么一些视图，需要依赖于界面上的已有元素，比如下面这个，它通过el属性，指定了HTML中id为todoapp的元素，并且还在initialize方法中引用了另外一些元素，通常，需要直接放置到界面的顶层试图会采用这种方式，而“部件”视图一般由主视图来创建、布局。

```
// Our overall **AppView** is the top-level piece of UI.
var AppView = Backbone.View.extend({
     // Instead of generating a new element, bind to the
existing skeleton of
    // the App already present in the HTML.
    el : $("#todoapp"),

     // Our template for the line of statistics at the
bottom of the app.
     statsTemplate : _.template($('#stats-template').
html()),

     // Delegated events for creating new items, and
clearing completed ones.
    events : {
        "keypress #new-todo" : "createOnEnter",
        "click #clear-completed" : "clearCompleted",
        "click #toggle-all" : "toggleAllComplete"
    },

     // At initialization we bind to the relevant events
```

```
on the `Todos`
    // collection, when items are added or changed. Kick
things off by
    // loading any preexisting todos that might be saved
in *localStorage*.
    initialize : function() {
        this.input = this.$("#new-todo");
        this.allCheckbox = this.$("#toggle-all")[0];

        this.listenTo(Todos, 'add', this.addOne);
        this.listenTo(Todos, 'reset', this.addAll);
        this.listenTo(Todos, 'all', this.render);

        this.footer = this.$('footer');
        this.main = $('#main');

        Todos.fetch();
    },

     // Re-rendering the App just means refreshing the
statistics -- the rest
    // of the app doesn't change.
    render : function() {
        var done = Todos.done().length;
        var remaining = Todos.remaining().length;

        if (Todos.length) {
            this.main.show();
            this.footer.show();
            this.footer.html(this.statsTemplate({
                done : done,
                remaining : remaining
            }));
        } else {
            this.main.hide();
            this.footer.hide();
        }

        this.allCheckbox.checked = !remaining;
```

```
    },

    //......
});
```

对于AngularJS来说，基本不需要有额外的视图定义，它采用的是直接定义在HTML上的方式，比如：

```
<div ng-controller="TodoCtrl">
    <span>{{remaining()}} of {{todos.length}} remaining</
span>
    <a href="" ng-click="archive()">archive</a>
    <ul class="unstyled">
        <li ng-repeat="todo in todos">
            <input type="checkbox" ng-model="todo.done">
                <span class="done-{{todo.done}}">{{todo.
text}}</span>
        </li>
    </ul>
    <form ng-submit="addTodo()">
        <input type="text" ng-model="todoText"  size="30"
        placeholder="add new todo here">
            <input class="btn-primary" type="submit"
value="add">
    </form>
</div>
```

在这个例子中，使用ng-controller注入了一个TodoCtrl的实例，然后，在TodoCtrl的$scope中附加的那些变量和方法都可以直接访问了。注意到其中的ng-repeat部分，它遍历了todos数组，然后使用其中的单个todo对象创建了一些HTML元素，把相应的值填到里面。这种做法和ng-model一样，都创造了双向绑定，即：

- 改变模型可以随时反映到界面上
- 在界面上做的操作（输入，选择等等）可以实时反映到模型里。

而且，这种绑定都会自动忽略其中可能因为空数据而引起的异常情况。

4. 模板

模板是这个时期一种很典型的解决方案。我们常常有这样的场景：在一个界面上重复展示类似的DOM片段，例如微博。以传统的开发方式，也可以轻松实现出来，比如：

```
var feedsDiv = $("#feedsDiv");

for (var i = 0; i < 5; i++) {
    var feedDiv = $("<div class='post'></div>");

    var authorDiv = $("<div class='author'></div>");
    var authorLink = $("<a></a>")
        .attr("href", "/user.html?user='" + "Test" + "'")
        .html("@" + "Test")
        .appendTo(authorDiv);
    authorDiv.appendTo(feedDiv);

    var contentDiv = $("<div></div>")
        .html("Hello, world!")
        .appendTo(feedDiv);
    var dateDiv = $("<div></div>")
        .html("发布日期: " + new Date().toString())
        .appendTo(feedDiv);

    feedDiv.appendTo(feedsDiv);
}
```

但是使用模板技术，这一切可以更加优雅，以常用的模板框架UnderScore为例，实现这段功能的代码为：

```
var templateStr = '<div class="post">'
    +'<div class="author">'
```

```
      +    '<a href="/user.html?user={{creatorName}}">@
{{creatorName}}</a>'
    +'</div>'
    +'<div>{{content}}</div>'
    +'<div>{{postedDate}}</div>'
    +'</div>';
var template = _.template(templateStr);
template({
    createName : "Xufei",
    content: "Hello, world",
    postedDate: new Date().toString()
});
```

也可以这么定义：

```
<script type="text/template" id="feedTemplate">
<% _.each(feeds, function (item) { %>
    <div class="post">
        <div class="author">
            <a href="/user.html?user=<%= item.creatorName
%>">@<%= item.creatorName %></a>
        </div>
        <div><%= item.content %></div>
        <div><%= item.postedData %></div>
    </div>
<% }); %>
</script>

<script>
$('#feedsDiv').html( _.template($('#feedTemplate').
html(), feeds));
</script>
```

除此之外，UnderScore还提供了一些很方便的集合操作，使得模板的使用更加方便。如果你打算使用BackBone框架，并且需要用到模板功能，那么UnderScore是一个很好的选择，当然，也可以选用其它的模板库，比如Mustache等等。

如果使用AngularJS，可以不需要额外的模板库，它自身就提供了类似的功能，比如上面这个例子可以改写成这样：

```
<div class="post" ng-repeat="post in feeds">
    <div class="author">
            <a  ng-href="/user.html?user={{post.
creatorName}}">@{{post.creatorName}}</a>
    </div>
    <div>{{post.content}}</div>
    <div>
        发布日期: {{post.postedTime | date:'medium'}}
    </div>
</div>
```

主流的模板技术都提供了一些特定的语法，有些功能很强。值得注意的是，他们虽然与JSP之类的代码写法类似甚至相同，但原理差别很大，这些模板框架都是在浏览器端执行的，不依赖任何服务端技术，即使界面文件是.html也可以，而传统比如JSP模板是需要后端支持的，执行时间是在服务端。

5. 路由

通常路由是定义在后端的，但是在这类MV*框架的帮助下，路由可以由前端来解析执行。比如下面这个Backbone的路由示例：

```
var Workspace = Backbone.Router.extend({
    routes: {
        "help":              "help",    // #help
         "search/:query":         "search",  // #search/
kiwis
          "search/:query/p:page": "search"    // #search/
kiwis/p7
    },
```

```
    help: function() {
        ...
    },

    search: function(query, page) {
        ...
    }
});
```

在上述例子中，定义了一些路由的映射关系，那么，在实际访问的时候，如果在地址栏输入"#search/obama/p2"，就会匹配到"search/:query/p:page"这条路由，然后，把"obama"和"2"当作参数，传递给search方法。

AngularJS中定义路由的方式有些区别，它使用一个$routeProvider来提供路由的存取，每一个when表达式配置一条路由信息，otherwise配置默认路由，在配置路由的时候，可以指定一个额外的控制器，用于控制这条路由对应的html界面：

```
app.config(['$routeProvider',
function($routeProvider) {
    $routeProvider.when('/phones', {
        templateUrl : 'partials/phone-list.html',
        controller : PhoneListCtrl
    }).when('/phones/:phoneId', {
        templateUrl : 'partials/phone-detail.html',
        controller : PhoneDetailCtrl
    }).otherwise({
        redirectTo : '/phones'
    });
}]);
```

注意，在AngularJS中，路由的template并非一个完整的html文件，而是其中的一段，文件的头尾都可以不要，也可以

不要那些包含的外部样式和JavaScript文件，这些在主界面中载入就可以了。

6. 自定义标签

用过XAML或者MXML的人一定会对其中的可扩充标签印象深刻，对于前端开发人员而言，基于标签的组件定义方式一定是优于其他任何方式的，看下面这段HTML：

```
<div>
    <input type="text" value="hello, world"/>
    <button>test</button>
</div>
```

即使是刚刚接触这种东西的新手，也能够理解它的意思，并且能够照着做出类似的东西，如果使用传统的面向对象语言去描述界面，效率远远没有这么高，这就是在界面开发领域，声明式编程比命令式编程适合的最重要原因。

但是，HTML的标签是有限的，如果我们需要的功能不在其中，怎么办？在开发过程中，我们可能需要一个选项卡的功能，但是，HTML里面不提供选项卡标签，所以，一般来说，会使用一些li元素和div的组合，加上一些css，来实现选项卡的效果，也有的框架使用JavaScript来完成这些功能。总的来说，这些代码都不够简洁直观。

如果能够有一种技术，能够提供类似这样的方式，该多么好呢？

```
<tabs>
    <tab name="Tab 1">content 1</tab>
    <tab name="Tab 2">content 2</tab>
</tabs>
```

回忆一下，我们在章节1.4 组件化的萌芽 里面，提到过一种叫做HTC的技术，这种技术提供了类似的功能，而且使用起来也比较简便，问题是，它属于一种正在消亡的技术，于是我们的目光投向了更为现代的前端世界，AngularJS拯救了我们。

在AngularJS的首页，可以看到这么一个区块“Create Components”，在它的演示代码里，能够看到类似的一段：

```
<tabs>
    <pane title="Localization">
        ...
    </pane>
    <pane title="Pluralization">
        ...
    </pane>
</tabs>
```

那么，它是怎么做到的呢？秘密在这里：

```
angular.module('components', []).directive('tabs',
function() {
    return {
        restrict : 'E',
        transclude : true,
        scope : {},
        controller : function($scope, $element) {
            var panes = $scope.panes = [];

            $scope.select = function(pane) {
                angular.forEach(panes, function(pane) {
                    pane.selected = false;
                });
                pane.selected = true;
            }

            this.addPane = function(pane) {
```

```
                if (panes.length == 0)
                    $scope.select(pane);
                panes.push(pane);
            }
        },
        template : '<div class="tabbable">'
            + '<ul class="nav nav-tabs">'
                + '<li ng-repeat="pane in panes" ng-
class="{active:pane.selected}">'
            + '<a href="" ng-click="select(pane)">{{pane.
title}}</a>'
            + '</li>'
            + '</ul>'
             + '<div class="tab-content" ng-transclude></
div>'
            + '</div>',
        replace : true
    };
}).directive('pane', function() {
    return {
        require : '^tabs',
        restrict : 'E',
        transclude : true,
        scope : {
            title : '@'
        },
        link : function(scope, element, attrs, tabsCtrl) {
            tabsCtrl.addPane(scope);
        },
            template : '<div class="tab-pane" ng-
class="{active: selected}" ng-transclude>' + '</div>',
        replace : true
    };
})
```

这段代码里，定义了tabs和pane两个标签，并且限定了pane标签不能脱离tabs而单独存在，tabs的controller定义了它的行为，两者的template定义了实际生成的html，通过

这种方式，开发者可以扩展出自己需要的新元素，对于使用者而言，这不会增加任何额外的负担。

四、一些想说的话

关于ExtJS

注意到在本文中，并未提及这样一个比较流行的前端框架，主要是因为他自成一系，思路跟其他框架不同，所做的事情，层次介于文中的二和三之间，所以没有单独列出。

写作目的

在我10多年的Web开发生涯中，经历了Web相关技术的各种变革，从2003年开始，接触并使用到了HTC，VML，XMLHTTP等当时比较先进的技术，目睹了网景浏览器的衰落，IE的后来居上，Firefox和Chrome的逆袭，各类RIA技术的风起云涌，对JavaScript的模块化有过持续的思考。未来究竟是什么样子？我说不清楚，只能凭自己的一些认识，把这些年一些比较主流的发展过程总结一下，供有需要了解的朋友们作个参考，错漏在所难免，欢迎大家指教。

专题：JavaScript第二季

给JavaScript初学者的24条最佳实践

作者/ Jeffrey Way

Jeffrey Way曾经是Nettuts+的编辑和网站开发课程Tuts+ Premium的校长。现在Jeffrey专注于在Laracasts!上教授关于Laravel零零总总，欢迎来laracasts.com看看。

1. 使用 === 代替 ==

JavaScript 使用2种不同的等值运算符：===|!== 和 ==|!=，在比较操作中使用前者是最佳实践。

> “如果两边的操作数具有相同的类型和值，===返回true，!==返回false。”
>
> ——JavaScript：语言精粹

然而，当使用==和！=时，你可能会遇到类型不同的情况，这种情况下，操作数的类型会被强制转换成一样的再做比较，这可能不是你想要的结果。

2.Eval= 邪恶

起初不太熟悉时，“eval”让我们能够访问JavaScript的编译器（译注：这看起来很强大）。从本质上讲，我们可以将字符串传递给eval作为参数，而执行它。

这不仅大幅降低脚本的性能（译注：JIT编译器无法预知字符串内容，而无法预编译和优化），而且这也会带来巨大的安全风险，因为这样付给要执行的文本太高的权限，避而远之。

3. 省略未必省事

从技术上讲，你可以省略大多数花括号和分号。大多数浏览器都能正确理解下面的代码：

```
if(someVariableExists)
   x = false
```

然后，如果像下面这样：

```
if(someVariableExists)
   x = false
   anotherFunctionCall();
```

有人可能会认为上面的代码等价于下面这样：

```
if(someVariableExists) {
   x = false;
   anotherFunctionCall();
}
```

不幸的是，这种理解是错误的。实际上的意思如下：

```
if(someVariableExists) {
   x = false;
}
anotherFunctionCall();
```

你可能注意到了，上面的缩进容易给人花括号的假象。无可非议，这是一种可怕的实践，应不惜一切代价避免。仅有一种情况下，即只有一行的时候，花括号是可以省略的，但这点是饱受争议的。

```
if(2 + 2 === 4) return 'nicely done';
```

未雨绸缪

很可能，有一天你需要在if语句块中添加更多的语句。这样的话，你必须重写这段代码。底线——省略是雷区。

4. 使用 JSLint

JSLint是由大名鼎鼎的道格拉斯（Douglas Crockford）编写的调试器。简单的将你的代码粘贴进JSLint中，它会迅速找出代码中明显的问题和错误。

> “JSLint扫面输入的源代码。如果发现一个问题，它返回一条描述问题和一个代码中的所在位置的消息。问题并不一定是语法错误，尽管通常是这样。JSLint还会查看一些编码风格和程序结构问题。这并不能保证你的程序是正确的。它只是提供了另一双帮助发现问题的眼睛。”
>
> ——JSLing 文档

部署脚本之前，运行JSLint，只是为了确保你没有做出任何愚蠢的错误。

5. 将脚本放在页面的底部

在本系列前面的文章里已经提到过这个技巧，我粘贴信息在这里。

```
    </div><!-- end container-->
    <script type="text/javascript" src="<?php echo bloginfo('template_directory') . '/js/jquery.js';?>"></script>
    <script type="text/javascript" src="<?php echo bloginfo('template_directory') . '/js/siteScripts.js';?>"></script>
  </body>
</html>
```

记住——首要目标是让页面尽可能快的呈献给用户，脚本的夹在是阻塞的，脚本加载并执行完之前，浏览器不能继续渲染下面的内容。因此，用户将被迫等待更长时间。

如果你的js只是用来增强效果——例如，按钮的单击事件——马上将脚本放在body结束之前。这绝对是最佳实践。

建议

```
<p>And now you know my favorite kinds of corn. </p>
<script type="text/javascript" src="path/to/file.js"></
script>
<script type="text/javascript" src="path/to/anotherFile.
js"></script>
</body>
</html>
```

6. 避免在 For 语句内声明变量

当执行冗长的for语句时，要保持语句块的尽量简洁，例如：

糟糕

```
for(var i = 0; i < someArray.length; i++) {
   var container = document.getElementById('container');
   container.innerHtml += 'my number: ' + i;
   console.log(i);
}
```

注意每次循环都要计算数组的长度，并且每次都要遍历dom查询“container”元素——效率严重地下！

建议

```
var container = document.getElementById('container');
```

```
for(var i = 0, len = someArray.length; i < len;  i++) {
   container.innerHtml += 'my number: ' + i;
   console.log(i);
}
```

感兴趣可以思考如何继续优化上面的代码，欢迎留下评论大家分享。

7. 构建字符串的最优方法

当你需要遍历数组或对象的时候，不要总想着“for”语句，要有创造性，总能找到更好的办法，例如，像下面这样。

```
var arr = ['item 1', 'item 2', 'item 3', ...];
var list = '<ul><li>' + arr.join('</li><li>') + '</li></
ul>';
```

> 我不是你心中神，但请你相信我（不信你自己测试）——这是迄今为止最快的方法！使用原生代码（如 join()），不管系统内部做了什么，通常比非原生快很多。
>
> ——James Padolsey, james.padolsey.com

8. 减少全局变量

> 只要把多个全局变量都整理在一个名称空间下，拟将显著降低与其他应用程序、组件或类库之间产生糟糕的相互影响的可能性。
>
> ——Douglas Crockford

```
var name = 'Jeffrey';
var lastName = 'Way';
```

```
function doSomething() {...}

console.log(name); // Jeffrey -- 或 window.name
```

更好的做法

```
var DudeNameSpace = {
   name : 'Jeffrey',
   lastName : 'Way',
   doSomething : function() {...}
}
console.log(DudeNameSpace.name); // Jeffrey
```

注：这里只是简单命名为 "DudeNameSpace"，实际当中要取更合理的名字。

9. 给代码添加注释

似乎没有必要，当请相信我，尽量给你的代码添加更合理的注释。当几个月后，重看你的项目，你可能记不清当初你的思路。或者，假如你的一位同事需要修改你的代码呢？总而言之，给代码添加注释是重要的部分。

```
// 循环数组，输出每项名字（译者注：这样的注释似乎有点多余吧）.
for(var i = 0, len = array.length; i < len; i++) {
   console.log(array[i]);
}
```

10. 拥抱渐进增强

确保javascript被禁用的情况下能平稳退化。我们总是被这样的想法吸引，“大多数我的访客已经启用JavaScript，所以我不必担心。”然而，这是个很大的误区。

你可曾花费片刻查看下你漂亮的页面在javascript被关闭时是什么样的吗？（下载 Web Developer 工具就能很容易做到（译者注：chrome用户在应用商店里自行下载，ie用户在Internet选项中设置）），这有可能让你的网站支离破碎。作为一个经验法则，设计你的网站时假设JavaScript是被禁用的，然后，在此基础上，逐步增强你的网站。

11. 不要给 "setInterval" 或 "setTimeout" 传递字符串参数

考虑下面的代码：

```
setInterval(
"document.getElementById('container').innerHTML += 'My
new number: ' + i", 3000
);
```

不仅效率低下，而且这种做法和"eval"如出一辙。从不给setInterval和setTimeout传递字符串作为参数，而是像下面这样传递函数名。

```
setInterval(someFunction, 3000);
```

12. 不要使用 "with" 语句

乍一看，"with"语句看起来像一个聪明的主意。基本理念是，它可以为访问深度嵌套对象提供缩写，例如……

```
with (being.person.man.bodyparts) {
   arms = true;
   legs = true;
```

```
}
```

而不是像下面这样：

```
being.person.man.bodyparts.arms = true;
being.person.man.bodyparts.legs= true;
```

不幸的是，经过测试后，发现这时“设置新成员时表现得非常糟糕。作为代替，您应该使用变量，像下面这样。

```
var o = being.person.man.bodyparts;
o.arms = true;
o.legs = true;
```

13. 使用 {} 代替 new Ojbect()

在JavaScript中创建对象的方法有多种。可能是传统的方法是使用"new"加构造函数，像下面这样：

```
var o = new Object();
o.name = 'Jeffrey';
o.lastName = 'Way';
o.someFunction = function() {
   console.log(this.name);
}
```

然而，这种方法的受到的诟病不及实际上多。作为代替，我建议你使用更健壮的对象字面量方法。

更好的做法

```
var o = {
   name: 'Jeffrey',
   lastName = 'Way',
   someFunction : function() {
      console.log(this.name);
```

```
    }
};
```

注意，如果你只是想创建一个空对象，{} 更好。

```
var o = {};
```

> “对象字面量使我们能够编写更具特色的代码，而且相对简单的多。不需要直接调用构造函数或维持传递给函数的参数的正确顺序，等”
>
> ——dyn-web.com

14. 使用 [] 代替 new Array()

这同样适用于创建一个新的数组。

例如：

```
var a = new Array();
a[0] = "Joe";
a[1] = 'Plumber';
```

更好的做法：

```
var a = ['Joe','Plumber'];
```

> “javascript程序中常见的错误是在需要对象的时候使用数组，而需要数组的时候却使用对象。规则很简单：当属性名是连续的整数时，你应该使用数组。否则，请使用对象”
>
> ——Douglas Crockford

15. 定义多个变量时，省略 var 关键字，用逗号代替

```
var someItem = 'some string';
var anotherItem = 'another string';
var oneMoreItem = 'one more string';
```

更好的做法

```
var someItem = 'some string',
    anotherItem = 'another string',
    oneMoreItem = 'one more string';
```

…应而不言自明。我怀疑这里真的有所提速，但它能是你的代码更清晰。

17. 谨记，不要省略分号

从技术上讲，大多数浏览器允许你省略分号。

```
var someItem = 'some string'
function doSomething() {
  return 'something'
}
```

已经说过，这是一个非常糟糕的做法可能会导致更大的，难以发现的问题。

更好的做法

```
var someItem = 'some string';
function doSomething() {
```

```
  return 'something';
}
```

18."For in" 语句

当遍历对象的属性时，你可能会发现还会检索方法函数。为了解决这个问题，总在你的代码里包裹在一个if语句来过滤信息。

```
for(key in object) {
   if(object.hasOwnProperty(key) {
      ...then do something...
   }
}
```

参考 JavaScript：语言精粹，道格拉斯（Douglas Crockford）。

19. 使用 Firebug 的 "timer" 功能优化你的代码

在寻找一个快速、简单的方法来确定操作需要多长时间吗？使用Firebug的“timer”功能来记录结果。

```
function TimeTracker(){
 console.time("MyTimer");
 for(x=5000; x > 0; x--){}
 console.timeEnd("MyTimer");
}
```

20. 阅读，阅读，反复阅读

虽然我是一个巨大的web开发博客的粉丝(像这样!)，午餐之余或上床睡觉之前，实在没有什么比一本书更合适了，坚持放

一本web开发方面书在你的床头柜。下面是一些我最喜爱的JavaScript书籍。

- Object-Oriented JavaScript（JavaScript面向对象编程指南 pdf）
- JavaScript: The Good Parts（JavaScript语言精粹 修订版 pdf）
- Learning jQuery 1.3（jQuery基础教程 第4版 pdf）
- Learning JavaScript（JavaScript学习指南 pdf）

读了他们……多次。我仍将继续！

21. 自执行函数

和调用一个函数类似，它很简单的使一个函数在页面加载或父函数被调用时自动运行。简单的将你的函数用圆括号包裹起来，然后添加一个额外的设置，这本质上就是调用函数。

```
(function doSomething() {
   return {
      name: 'jeff',
      lastName: 'way'
   };
})();
```

22. 原生代码永远比库快

JavaScript库，例如jQuery和Mootools等可以节省大量的编码时间，特别是AJAX操作。已经说过，总是记住，库永远不可能比原生JavaScript代码更快(假设你的代码正确)。

jQuery的“each”方法是伟大的循环，但使用原生"for"语句总是更快。

23. 道格拉斯的 JSON.Parse

尽管JavaScript 2（ES5)已经内置了JSON 解析器。但在撰写本文时，我们仍然需要自己实现（兼容性）。道格拉斯（Douglas Crockford），JSON之父，已经创建了一个你可以直接使用的解析器。这里可以下载（链接已坏，可以在这里查看相关信息http://www.json.org/）。

只需简单导入脚本，您将获得一个新的全局JSON对象，然后可以用来解析您的json文件。

```
var response = JSON.parse(xhr.responseText);

var container = document.getElementById('container');
for(var i = 0, len = response.length; i < len; i++) {
  container.innerHTML += '<li>' + response[i].name + ' :
' + response[i].email + '</li>';
}
```

24. 移除 "language" 属性

曾经脚本标签中的“language”属性非常常见。

```
<script type="text/javascript" language="javascript">
...
</script>
```

然而，这个属性早已被弃用，所以请移除（译者注：html5 中已废弃，但如果你喜欢，你仍然可以添加）。

译者 / 颜海镜

90后一枚，活跃在各个技术社区，专注Web前端已有三个年头。关注HTML/CSS/JavaScript等相关技术，目前就职于北京金山软件。坚信Web赢在未来。热爱思考，热爱开源分享，常翻译些外文博客，此外还爱好读书，羽毛球，乒乓球，相声，铁杆纲丝，当然，也非常热爱写代码。个人博客：yanhaijing.com

就这样吧，伙计

现在你已经学到了，24条JavaScript初学者的必备技巧。让我知道你高效技巧吧！感谢你的阅读。本系列的第三部分主题会是什么呢（思索中）？

译者补充

第三部分在这里：高效jQuery的奥秘

本文为翻译文章，原文为“24 JavaScript Best Practices for Beginners”

关于#20 的补充，下面是译者认为的一些好书，有兴趣的读者可以留言讨论

- javascript模式（和上面JavaScript面向对象编程指南同一作者，这本书更好）
- javascript设计模式
- 编写可维护的javascript（尼古拉斯新书）
- 高性能javascript（尼古拉斯 已绝版）
- javascript语言精髓与编程实践
- javascript高级程序设计（尼古拉斯）

专题：JavaScript第二季

Angular、Backbone、CanJS与Ember:
JavaScript MVC框架PK

作者/ Sebastian Porto

最爱JavaScript, Ruby和Go, 没事儿爱写Blog，深度编程强迫症患者。现居于澳大利亚墨尔本。Sebastian's Blog,@Twitter,@sporto on GitHub。

选择JavaScript MVC框架很难。一方面要考虑的因素非常多，另一方面这种框架也非常多，而要从中选择一个合适的，还真得费一番心思。想知道有哪些JavaScript MVC框架可以选择？看看TodoMVC吧。

我用过其中4个框架：**Angular、Backbone、CanJS和Ember**。因此，可以对它们作一比较，供大家参考。本文会涉及框架选型过程中需要考虑的一系列因素，我们逐一讨论。



每一个因素我们都会按照1到5分来打分，1分代表很差，5分代表很好。我会尽量保持客观，但也不敢保证真能“一碗水端平”，毕竟这些分数都是根据我个人经验给出的。

功能

作为构建应用的基础，框架必须具备一些重要的功能。比如，视图绑定、双向绑定、筛选、可计算属性（computed property）、脏属性（dirty attribute）、表单验证，等等。还能罗列出一大堆来。下面比较了一些我认为MVC框架中比较重要的功能：

功能	Angular	Backbone	CanJS	Ember
可观察对象（observable）	是	是	是	是
路由（routing）	是	是	是	是
视图绑定（view binding）	是		是	是
双向绑定（two way binding）	是	-	-	是
部分视图（partial view）	是	-	是	是
筛选列表视图（filtered list view）	是	-	是	是

- **可观察对象：**可以被监听是否发生变化的对象。
- **路由：**把变化通过浏览器URL的参数反映出来，并监听这些变化以便执行相应的操作。
- **视图绑定：**在视图中使用可观察对象，让视图随着可观察对象的变化而自动刷新。
- **双向绑定：**让视图也能把变化（如表单输入）自动推送到可观察对象。
- **部分视图：**包含其他视图的视图。
- **筛选列表视图：**用于显示根据某些条件筛选出来的对象的视图。

得分

根据上述功能，我打出的分数如下：

Angular	Backbone	CanJS	Ember
5	2	4	5

有一点必须指出，使用Backbone也能实现上述大多数功能，只是手工编码量挺大的，有时候还要借助插件。这里的打分只考虑了框架核心是否支持某一功能。

灵活性

有时候，框架配合一些现成的插件和库来使用，可能要比使用框架原生同类功能效果更好，而这种插件和库几乎遍地都是（不下数百个），又各有特色。因此，能够把这些库和插件整合到MVC框架中也非常重要。

Backbone是其中最灵活的一个框架，因为它的约定和主张最少。使用Backbone需要你自己作出很多决定。

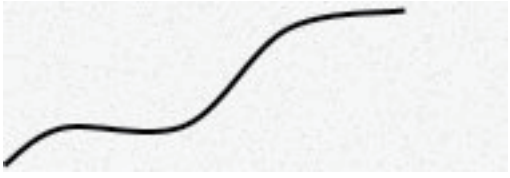
CanJS的灵活性与Backbone差不多，把它跟别的库整合起来很容易。在CanJS中甚至可以更换其他渲染引擎，我在CanJS中就一直用Rivets，没有任何问题。不过，我还是推荐框架自带的组件。

Ember和Angular也都还算灵活，可有时候你会发现，就算不喜欢它们的某些实现方法，你也只能默默忍受。 这是在选择Ember或Angular时必须考虑的。

得分

Angular	Backbone	CanJS	Ember
3	5	4	3

上手难度



Angular

Angular一开始会让人大呼过瘾，因为可以利用它干好多意想不到的事，比如双向绑定，而且学习难度不高。乍一看让人觉得很简单。可是，进了门之后，你会发现后面的路还很长。应该说这个框架比较复杂，而且有不少标新立异之处。想看着它的文档上手并不现实，因为Angular制造的概念很多，而文档中的例子又很少。

Backbone

Backbone的基本概念非常容易理解。但很快你会发现它对怎么更好地组织代码并没有太多主张。为此，你得观摩或阅读一些教程，才能知道在Backbone中怎么编码最好。而且，你会发现在有了Backbone的基础上，还得再找一个库（比如Marionette或Thorax）跟它配合才能得心应手。正因为如此，我不认为Backbone是个容易上手的框架。

CanJS

CanJS相对而言是这里面最容易上手的。看看它只有一页的网站（http://canjs.com/），基本上就知道怎么做效率最高了。当然，还得找其他一些资料看，不过我个人很少有这种需求（比如看其他教程、上论坛或讨论组提问呀什么的）。

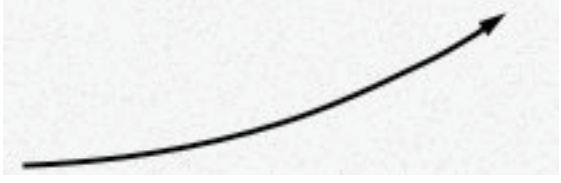
Ember

Ember的上手难度与Angular有一拼，我认为学习Ember比学习Angular总体上容易一些，但它要求你一开始就要先搞懂一批基本概念。而Angular呢，一开始不需要这么费劲也能做一些让人兴奋不已的事儿。Ember缺少这种前期兴奋点。

得分

Angular	Backbone	CanJS	Ember
2	4	5	3

开发效率



比较全面地掌握了一个框架之后，重点就转移到了产出上。什么意思呢？约定啊、戏法啊，反正要尽可能快。

Angular

熟悉Angular之后，你的效率会非常高，这一点毋庸置疑。之所以我没给它打最高分，主要因为我觉得Ember的开发效率似乎更胜一筹。

Backbone

Backbone要求你写很多样板（boilerplate ）代码，而我认为这完全没必要。要我说，这是直接影响效率的一个因素。

CanJS

CanJS的开发效率属于不快不慢的那种。不过，考虑到学习难度很低，因此适合早投入早产出的项目。

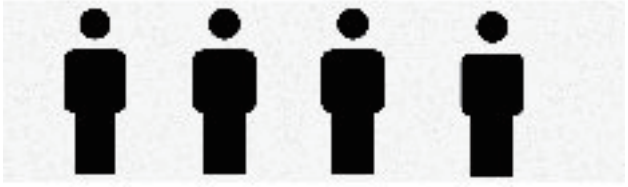
Ember

Ember的开发效率首屈一指。它有很多强制性约束，可以帮你自动完成的事很多。而开发人员要做的，就是学习和应用这些约定，Ember会替你处理到位。

得分

Angular	Backbone	CanJS	Ember
4	2	4	5

社区支持



能轻易找到参考资料和专家帮忙吗?

Backbone的社区很大，这是人所共知的事实。关于Backbone的教程也几乎汗牛充栋，StackOverflow和IRC社区非常热闹。

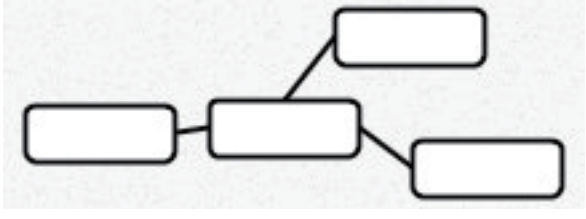
Angular和Ember社区也相当大，教程什么的同样不少，StackOverflow和IRC也很热闹，但还是比不上Backbone。

CanJS社区呢，相对小一些，好在社区成员比较活跃，乐于助人。我倒没发现CanJS社区规模小有什么负面影响。

得分

Angular	Backbone	CanJS	Ember
4	5	3	4

生态系统



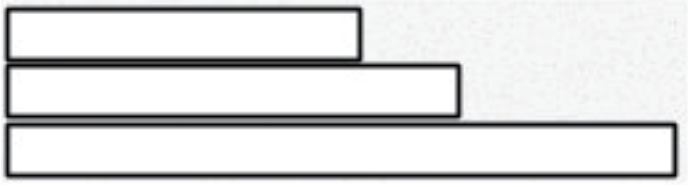
有没有插件或库构成的生态系统?

说起插件和库，Backbone的选择是最多的，可用插件俯拾皆是，这一点让其他框架都望尘莫及。Angular的生态圈加上Angular UI还是很令人瞩目的。我觉得Ember的下游生态虽然欠发达，但Ember本身很受欢迎，所以前景十分乐观。CanJS的下游支脉比较少见。

得分

Angular	Backbone	CanJS	Ember
4	5	2	4

文件大小



这个因素有时候很重要，特别是对于移动开发项目。

自身大小（无依赖，未压缩）

Angular	Backbone	CanJS	Ember
80KB	18KB	33KB	141KB

Backbone最小，这一点也是最为人们所津津乐道的。但不能只看库本身的体积。

包含依赖的大小

80KB的Angular是唯一不需要其他库就能使用的。其他三个框架则都对其他库有依赖。

Backbone至少需要Underscore和Zepto。虽然在Underscore中可以使用最小的模板来渲染视图，但多数情况下，还要借助更好的模板引擎，比如Mustache。这样它就增肥到了61KB。

CanJS至少需要Zepto，因此会达到57KB。

Ember需要jQuery和Handlebars，总共是269KB。

Angular	Backbone	CanJS	Ember
80KB	61KB	57KB	269KB

得分

Angular	Backbone	CanJS	Ember
4	5	5	2

性能

我不认为性能是选择框架的关键因素，因为这些框架在预期应用领域中的性能都不差。当然啦，具体还得看你做什么项目。要是想开发游戏，那性能是个重要因素。

虽然我见过，也亲自做过一些性能对比（比如这个测试），但我并不完全相信测试结果。很难说这种测试的方法和结果与实际项目吻合。

不过，据我所见所闻，CanJS的性能是最高的，而且在视图绑定上格外突出。相对来说，我觉得Angular性能稍差，因为它执行对象的脏检查（dirty checking），这一点就拖了它的后腿了。参见这里。

得分

Angular	Backbone	CanJS	Ember
3	4	5	4

成熟度

这个框架成熟吗，经过实际检验了吗，有多少网站在用它呢？

使用Backbone的网站不计其数。最近两年，它的核心代码没怎么改过，这是成熟的一个重要标志。

Ember已经不是新框架了，但它的重大变更还是经常有，前几月刚刚稳定下来。因此，目前还不能说它是个成熟的框架。

Angular似乎比Ember更稳定，验证的示例也更多，但不能与Backbone相提并论。

CanJS好像还未经任何验证，因为不知道有什么网站在使用它。不过，CanJS其实也没有看起来那么弱不经风，它可是从JavaScriptMVC精简来的。JavaScriptMVC是2008年就出现的一个库，因此会有很多智慧结晶留传下来。

得分

Angular	Backbone	CanJS	Ember
4	5	4	3

内存泄漏隐患

如果你想开发每次打开都得运行很长时间的单页应用，这是必须得考虑的问题。你当然不希望自己的应用导致内存泄漏，这个问题非常现实。不幸的是，内存泄露很容易发生，而自己编写的DOM事件监听器则是重灾区。

只要你守规矩，Angular、CanJS和Ember能把这个问题帮你解决好。Backbone则不然，它需要你自己手工来卸载。

得分

Angular	Backbone	CanJS	Ember
5	3	5	5

个人偏好

这恐怕是选择框架时最重要的一个因素了。

- 你喜欢声明式HTML吗？ -> Angular
- 你喜欢使用模板引擎吗？ -> Backbone、CanJS或Ember

- 你喜欢固执已见的框架吗？ -> Ember
- 你希望框架与最初的SmallTalk MVC模式完全吻合吗？ -> 没有完全吻合的，或许CanJS最接近
- 你希望使用目前看来很酷的框架吗？ -> Ember、Angular

本项没办法打分。

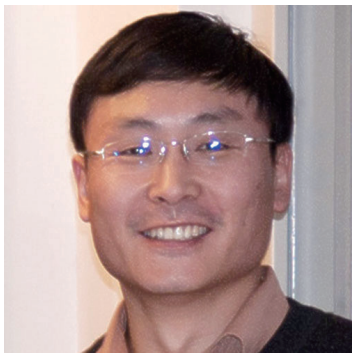
算总分

好啦，把各框架所得分数做个汇总吧。别忘了这只是我个人看法，如果你觉得哪一项打分有失偏颇，请务必告诉我。

（此处只是一个静态图表，单击可以打开）

Result	Relevance	Angular	Backbone	Can	Edit in JSFiddle
Features	- + 1	5	2	4	5
Flexibility	- + 1	3	5	4	3
Learning curve and documentation	- + 1	2	4	5	3
Developer productivity	- + 1	4	2	4	5
Community	- + 1	4	5	3	4
Ecosystem	- + 1	4	5	2	4
Size	- + 1	4	5	5	2
Performance	- + 1	3	4	5	4
Maturity	- + 1	4	5	4	3
Memory leak safety	- + 1	5	3	5	5
		38	40	41	38

如果每一个因素的权重都一样，那么这几个框架确实难分高下。因此，我觉得最终决定很大程度上还是取决于你的个人偏好，或者必须要给每个因素赋予不同的权重才行。



译者 / 李松峰

@ 李松峰，非计算机专业出身的技术爱好者，曾从事 5 年 Web 前后端开发工作，现为北京图灵文化发展有限公司 QA 部主任。2006 年开始涉足计算机相关图书翻译，至今翻译字数超过 458 万字，已出版译著 20 余部。图灵社区 ID: 李松峰

聊聊 Backbone（公平到此为止！）

我在这篇文章里始终尽量做到“一碗水端平”。可是，最后针对 Backbone 我还想再多说两句，因为这些话如鲠在喉，不吐实在不快。

Backbone 在两年前是一个不错的库，但我知道：今天已经有了更好的选择了。我也知道，很多人选择 Backbone 其实仅仅因为使用它的人多，这是个恶性循环。

Backbone 为追求灵活性而抛弃了开发的便利性。但我觉得它实在太过了，功能严重缺乏，导致开发效率很低。没错，有一大堆插件可以实现这样那样的功能，可这样一来你不仅要学习 Backbone，还得不断学习怎么使用那些插件。

Backbone 的大型社区和生态系统也很诱人，但随着其他框架越来越受欢迎，这个优势也将慢慢消失。

正因为如此，我强烈建议大家选择 Backbone 时要三思而后行。

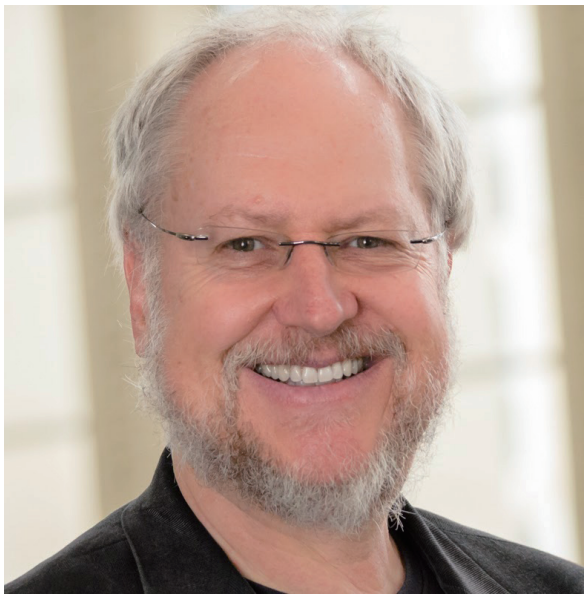
- 原文地址：http://sporto.github.io/.../comparison-angular-backbone-can-ember/
- 相关阅读：JavaScript 宝座：七大框架论剑

专题：JavaScript第二季

Douglas Crockford：
代码阅读和每个人都该学的编程

作者 /Peter Seibel

Peter Seibel是Common Lisp专家，Jolt生产效率大奖图书《实用 Common Lisp 编程》的作者。耶鲁大学英语专业毕业，后投身于互联网行业，曾负责“Mother Jones Magazine”和“Organic Online” 的Perl 专栏以及 WebLogic 的 Java 专栏，并曾在加州大学伯克利分校成人教育学院教授 Java 编程。2003 年辞职专心研究 Lisp 编程，之后即有了那部 Jolt 大奖图书。现在他是 Gigamonkeys Consulting 公司的首席执行官，和家人幸福地生活在加州伯克利。



Douglas Crockford现在供职于Paypal。曾是Yahoo!的资深JavaScript架构师，他在上世纪70年代初求学期间就开始从事程序开发工作了，那时的他主修电视广播专业，但苦于无法进入演播室工作，转而学习了学校开设的Fortran课程。在其职业生涯中，Crockford曾先后供职于Atari、Lucasfilm和Electric Communities，以各种方式联姻计算机与传播媒介。深感于XML的复杂性，他发明了JSON这一广泛用于Ajax应用的数据交换格式。Crockford曾谈到如果能避免使用某些特性的话，JavaScript实际上是一门相当优雅的语言。他强调了以子集方式来管理复杂度的重要性，同时介绍了他所使用的一种代码阅读方法：从清理代码开始。

关于 JavaScript

Seibel：在程序学习之路上有哪些令你后悔的事情?

Crockford：我了解一些语言，但却一直没有机会使用。我花了不少时间学习APL并了解到其衰败的原因，但这门语言真的非常优雅，可我却没有花时间使用它，这太遗憾了。除此以外，我还了解其他一些语言，知道能用它们做什么，但实际上却并没有机会用这些语言思考。

Seibel：我听说你喜欢ES3版本JavaScript的简洁性。

Crockford：嗯，最终无论怎么对语言进行修订，其要义都是希望促进语言的不断成功。语言越成功，修改的代价就越大。随着你的不断成熟，再教育的成本就会变得更大，同时还有潜在的破坏代价，而这些成本和代价也会变得难以接受。如果你确实非常成功，那就更要小心提防所做的任何变化了。反之，如果你尚未成功，那么就有更大的自由空间来改变了。

JavaScript成为世界上最流行的编程语言纯粹是偶然。目前世界上JavaScript处理器的数量要高于任何其他语言。得益于其安全模型带来的种种问题，JavaScript是唯一一门可在任何机器上编写并运行的语言。 这些还嫌不够的话，再看看那么多嵌入了JavaScript的应用吧。Adobe的大多数应用都嵌入了JavaScript，这样就可以在本地编写脚本控制这些应用了。还有其他很多应用，不胜枚举。这么一看，JavaScript已经变得非常流行了。

JavaScript这门语言的问题在于推向市场以及标准化的过程都过于匆忙了。其大多数缺陷都没有出现在目前的实现当

中——只存在于规范中。标准说照错的做，这听起来太吓人了，但这就是JavaScript的状态。它于1999年冻结了，接下来本应走向灭亡。但Ajax的横空出世改变了这一切，JavaScript变成了世界上最重要的编程语言。

于是，我们现在认为应该修复它。但这事应该是在2000年就开始做的，而那时并没有这么做，因为根本没人关注JavaScript。现在它已经长大了。

Web环境下的JavaScript还有一点非常怪异：如果编写服务器端应用、桌面应用或是嵌入式应用，你不仅需要选择语言，还要选择特定的编译器以及特定的运行时。但对JavaScript而言你别无选择，你必须在所有的环境下运行。

由于要在所有环境下运行，bug就没法修复了。如果某个浏览器厂商搞出个bug，他们会说“天啊，搞砸了”，下个月就会发布另一个版本，但我们却不能指望着所有用户都会升级。大多数人一旦在机器里装上IE就再也不会升级了，那些bug就会常年驻留在浏览器上。

Seibel：这就是目前的状况。你希望Web能成为更适合于应用开发的平台。除非所有浏览器都能修复这些bug，否则我们是没法修复的，如果这还不管用，那实在是没办法了。路在何方呢?

Crockford：这正是我努力争取的东西。我心中已经有一套理想的方案了，知道它要成为什么样子。我知道身处何方，也能看到前方的障碍。我正在思考如何才能前行。从某种意义上来说，我们已经身陷囹圄了，因为我们开发出了这些大型系统——我更关注经济系统、社会系统，还有技术系统——它们都依赖于这个并不完善的系统。

毫无疑问，JavaScript的鸡肋就是对全局对象的依赖。它没有链接器，无法在编译单元间隐藏信息。它将这些信息都丢到了一个普通的全局对象中。这样，所有组件都能访问一切内容，对DOM拥有相同的访问权限，对网络也拥有相同的访问权限。如果有人将脚本放到了你的页面上，它就可以访问服务器，看起来就像是你自己的脚本一样，而服务器则根本无法分辩。

这些脚本可以获取屏幕信息，可以访问用户信息，看起来就像是你自己的脚本一样，用户同样也无法分辩。在页面来自于你的服务器，而且无论脚本来自何处都拥有同样权限的情况下，用户所有新式的反钓鱼工具就都派不上用场了。

实际情况比这还要糟，因为脚本还可以通过其他方式进入你的页面。Web架构涉及几种语言——有HTTP、HTML，URL可以看作是一种语言，有CSS，还有脚本语言。它们可以彼此嵌入并且具有不同的引号、转义和注释约定。所有浏览器对这些语言的实现都不一样。加之这些实现的细节并不是在哪里都可以找到，这样恶意用户就能够轻松将脚本放到URL中，放到样式中，放到HTML中，放到其他脚本中。

Seibel：这么说来人们在ES4上所付出的仅仅是机会成本，每个人都花时间来思考这个问题而不是寻找问题的解决办法？

Crockford：没错。ES4的目标根本就不对，它想解决的是人们厌恶JavaScript这个问题。我很欣赏Brendan Eich对待这个问题的立场，他确实做得很棒，但却太着急了，没有管理好这个项目，放出了ES4这么差劲的东西。在过去的12年里，有很多人诋毁和诅咒他，觉得他是个蠢蛋，搞出来的语言也愚蠢至极，但我觉得事实并非如此。这个语言确实有闪亮的地方，而他其实是一个才华横溢的家伙。现在他在极力表明自己是无

辜的，想要证明自己是个聪明人，他想要展示这门语言的优秀特性，将这些特性整合起来，这样它就能好用了。

我觉得这并非现在需要解决的问题，亟需解决的问题是：Web正变得分崩离析，我们需要对其进行修复。这要求我们清楚路在何方。我最反对Brendan的一个地方就是他的方案完全没抓住重点。

我越来越感觉到这是个问题。如果能够模块化，如果能够自己选择编程语言，我们本可以走得更远。但现在这一切都尚未实现，不过现实情况比这还好一些。现在有Caja和ADsafe这样的东西在使用目前的技术解决这些问题。时不我待啊。

ADsafe创建了JavaScript的一个安全子集，它不允许访问任何的全局变量和危险的东西。这么做的结果是这个子集依然可以构成一门有用的语言，因为有强大的lambda作为后盾，它可以完成很多事情。它是一门非传统的语言，它不支持我们今天使用原型的方式，但这个子集却是一个功能完整的lambda语言，非常强大。

Seibel：先不考虑ES4目标的正确与否，仅仅从语言的视角来看，它有没有哪些特性吸引到你呢？

Crockford：有一些bug修复我觉得很不错，值得保留。但ES4中有太多未经验证的东西了。我们使用ES3的经历表明，一旦规范中出现了错误就很难再把它剔除出去。我们并没有ES4的使用经验，没人用它开发过大型应用。

在真正应用到实际工作中之前需要对其进行标准化和部署，我

觉得现在的节奏太快了。如果有多个参考实现，人们也开发出了一些有价值的应用，这才表明语言是没问题的，接下来才对其进行标准化并部署起来。但现在的做法却是反其道而行之。

Seibel：Google的GWT会 将Java编 译 为JavaScript。还有人尝试将其他语言编译为JavaScript。这是未来之路么？

Crockford：看到JavaScript逐渐变为通用运行时还是蛮有趣的一件事，这一点倒是我们不曾想到的。

Seibel：但正如你所说，JavaScript无处不在，它确实是通用运行时。

Crockford：我认为这也进一步说明JavaScript确实该跑步前进。尤其是现在我们正在进入移动产品的时代，但摩尔定律并不适用于电池。JavaScript在解释上所花费的时间愈发显得重要，此外还有周期数。我认为这会更加督促我们来改进运行时的质量。

就GWT和其他转换工具而言，我的态度是实用至上。人们是很难融入到这个环境中的——如果能找到好的解决方案，那非常棒。我自己是害怕使用这些工具的，担心的就是抽象层泄漏（abstraction leakage）。如果你的Java代码、GWT或是其生成的代码存在问题，那可能就没法解决了。特别是在完全不了解JavaScript的情况下就贸然使用这种方法会更加危险，因为GWT对你完全隐藏了JavaScript。在这种情况下一旦出现了问题，那受伤的肯定是你自己。虽然我还没有听说发生过这种事，到目前为止这些工具还不错，但毕竟存在着风险。

Seibel：你希望JavaScript有哪些变化呢?

Crockford：我认为改进JavaScript最好的办法就是瘦身。如果我们能够取其精华，弃其糟粕，那JavaScript会变得更棒。我认为这个办法也适合于HTML、HTTP和CSS。我们应该仔细思考所用的各种标准，搞清楚需要哪些特性，遗漏了哪些特性并重新审视它们，绝不应该盲目地增加新特性。

代码阅读

Seibel：目前你在Yahoo！扮演着JavaScript架构师和布道者的角色，我想你的一部分工作是向Yahoo!的JavaScript开发者们传授“JavaScript应用之道”。那你的工作还涉及一般性的优秀设计实践与编码实践么?

Crockford：我一直在倡导良好的代码阅读方法。我认为这是社区中的开发者多花点儿时间阅读彼此的代码，这对每个人都非常有意义。现今的项目管理有这样一种趋势：让开发者们独立完成工作，接下来进行大规模的整合，如果没问题就发布出去，然后就大功告成了，接着就抛之脑后了。

这么做的一个后果就是一旦碰上差劲的开发者，到最后你才能发现问题，不过这时已经太晚了。项目会因此出现风险，在构建的时候才发现有些代码写得实在是糟糕，肯定会导致项目延期，而这是无法接受的。另外，项目中可能会有一些优秀的开发者，而他们却没有太多机会指导其他成员。代码阅读可以解决上面这两个问题。

Seibel：能否详细谈谈如何进行代码阅读呢?

Crockford：每次开会都让一些人阅读他们各自的代码，他们会引领我们查看其编写的所有内容，其他人则负责检查。对于团队的其他成员来说，这绝对是个学习的好机会，通过这个过程他们就可以知道自己的东西该如何与他人的相配合。

每个人都围坐在桌边，手里拿一叠纸，同时还把代码在屏幕上打出来，大家一起阅读。我们会在编写代码的过程中加上注释。有人会说“我看不懂这个注释”或是“这个注释与代码风马牛不相及”。大家的意见极具价值，因为作为开发者的你是不会阅读自己编写的注释的，你也根本没有意识到自己写的注释误导了读者。有这么多人帮助你编写整洁的代码是多么幸福的一件事啊——你会找到自己根本无法找到的缺陷。

我认为一小时的代码阅读抵得上两周的QA。这种剔除错误的手段真是很高效。如果你让能力很强的同事阅读代码，那么他们周围的新手们就会学到很多东西，而这一切是无法通过其他手段获得的；如果新手来阅读代码，那么他会得到很多极具价值的建议。

但这件事我们不能一直留到最后再做。回忆过去，我们会在项目完成之际安排代码阅读，但这个时候已经太迟了，只好取消。现在我深信代码阅读应该伴随着整个项目的生命周期。我花了很长时间才意识到这一点，这么做的好处不胜枚举。

首先，这么做有助于把控项目，我们能够真切地看到大家的进度，也能及早发现是不是有人已经偏离了轨道。

我曾经管理过一些项目，马上就到最后期限了，有人说“耶，马上就干完了”，然后我拿到了代码，发现里面什么都没有，

有的也是一些垃圾，离完成还远着呢。管理层最厌恶这种事情了，我觉得代码阅读能够有效避免这种窘境的发生。

Seibel：那你需要指导别人如何进行代码阅读么？可以想象，既不想让代码编写者感觉受到侵犯，又能给出颇具价值的意见是很难的。

Crockford：没错，这需要给予团队成员充分的信任，要明确界定好边界。如果团队不和睦，那就别指望这么做了，这会导致团队分崩离析。如果还没有意识到团队的不和睦，那这么做很快就能发现。这个过程会让你学到很多，也会揭示出很多问题。起初会觉得不太自然，但一旦适应后就会觉得再自然不过了。

另外，我们要编写可读性好的代码。大家都知道，整洁很重要，而代码风格也同样如此。所有这一切会提升代码的质量并增强编程社区的能力。

Seibel：如何编写可读性好的代码呢？

Crockford：可读性有几个等级。最简单的一级是与表达保持一致，适当地保持缩进，在适当的地方使用空格。我有一个习惯来自于早年学习Fortran的时候，那就是，我往往会使用过多的单字母变量名，这个习惯可不好。我真的在很努力改掉这个坏习惯，但太难了——这么多年来还在与之斗争。

Seibel：有什么具体的举措可以提升代码的可读性呢？

Crockford：子集的想法非常重要，尤其对于JavaScript来说更是如此，因为这门语言包含了太多的糟粕，当然其他语言也一样。当还是个菜鸟时，我会翻阅语言规范并弄明白每个特性。我知道该如何使用这些特性并一直在用。但事实证明很多特性并非是深思熟虑的结果。

我现在想到的是Fortran，但其实所有语言都难逃这个宿命。有时语言设计者本身就错了。依我看来，C在设计上存在很多不妥之处。

Seibel：你如何阅读别人编写的代码呢?

Crockford：清理。我会把代码放到文本编辑器中并开始修复。首先，我会统一标点符号，适当缩进，等等这类的事情。我有些程序可以完成这些事情，但从长远来看自己完成会更加高效，因为这有助于我加深对代码的理解。Morningstar曾教会我如何完成这些事情。他非常善于重构别人的代码，并且这也是他所使用的方法，很好。

Seibel：你是否遇到过这种情况：看到代码写得一团糟，然后进行清理，但最后发现原来的代码其实写得很不错?

Crockford：还没遇到过。我认为随随便便写出来的代码肯定不好。好的代码意味着可读性要好。在某种程度上，如果我搞不懂代码的意图，那么写得再好也没用，有可能代码在我不关心的方面表现得很好，比如很高效、很紧凑，等等。

代码的可读性是我的第一要义。它比速度还重要，可以与正确性一争高下，可读性是正确性的重要前提。如果可读性不好，那就不是好代码，代码的编写者可能做出了错误的权衡。

Seibel：在阅读代码时你会先排版，那你会对代码进行多大程度上的重构呢?

Crockford：我会重新编排代码以便所有东西都会在使用前声明和创建。有些语言提供了很大的灵活性，让你无需再这么做了。但这种灵活性我不需要。

Seibel：你曾在之前的一次演讲中引用了《出埃及记》第23章第10节和第11节的内容——六年你要耕种田地，收藏土产，只是第七年你要叫地歇息，不耕不种。并建议每次第7个sprint都应该用来清理代码。那什么时候做比较好呢？

Crockford：每6个周期一一不管周期间是什么都该如此。如果你是每月交付，那么我觉得每隔半年都应该跳过一个周期，专门用来清理代码。

每个人的必修课：编程

Seibel：编程是不是越来越容易了，门槛逐渐降低？

Crockford：我对编程的兴趣在于帮助其他人编程、设计特定的语言或编程工具，这样会有越来越多的人能够从事编程工作——这也是Smalltalk的初衷。Smalltalk后来的发展方向出现了变化，但最初的方向确实吸引了我。我们如何设计一门面向儿童的语言，如何为那些并非程序员的人们设计一种语言？

Seibel：你是不是认为每个人都应该学习编程，至少了解一些？

Crockford：没错。当今世界快被计算机控制了，为了保护自己或是让自己更加全面，你应该了解这些东西的工作方式。

Seibel：有些人认为通过编程可以学到一种重要的思考方式，比如阅读和数学就是不同的思考方式，但都非常重要。

Crockford：我以前也这么想。在开始编程时我就有过这种想法：一切都是那么地井然有序，我看到了之前从未接触过的结构等东西。我在想“喔，这太神奇了。每个人都应该学习学习”，因为我突然之间感觉自己变聪明了。但不久之后，在与其他程

序员交流的过程中发现，他们并没有开窍。程序员其实与常人也没什么区别，有时他们也会出现误解。当认识到这一点后我觉得很难过。

Seibel：那编程只是年轻人的专利么？

Crockford：过去我是这么认为的。几年前我患有睡眠呼吸暂停的症状，但没有意识到。我想可能是太累了，年纪也有些大了吧，结果发现自己的注意力很难集中，甚至都没法编程了，因为我的大脑没法承载太多的东西。很多时候编程都需要先在脑子里想好，然后再写出来，但我却不行。

我丧失了这种能力，想当然地认为是年龄太大的缘故。幸好，病情得到了好转，于是我又开始编程了。现在的我编程水平可能比以前还要好，因为我知道如何不过多地依赖于记忆。现在的我更喜欢将代码文档化，因为我不敢保证下一周还能记得写这些代码的意图。事实上，有时我会检查自己的代码，但会惊讶于自己怎么会这么写代码：我压根就不记得自己曾经这么写过，这些代码有的非常丑陋，有的却非常优雅。我实在不知道怎么会这样。

Seibel：Knuth的大部头《计算机程序设计艺术》如何？你是从头到尾读过这本书呢，还是将其作为参考随时翻阅，抑或是把它束之高阁碰也不碰呢？

Crockford：除了你说的最后一种情形之外。在上大学时，有那么几个月我连房租都没交，就是为了买他的书。我读过这些书，从中得到了不少乐趣，比如在第一卷的索引有个关于拖车的笑话就很好玩。我到现在为止还没能把书上的内容全部搞懂。Knuth对某些地方的研究要比我深入得多，但我还是喜欢这些书并把它们当作参考资料。

这是一本访谈笔录，记录了当今最具个人魅力的 15 位软件先驱的编程生涯。包括 Donald Knuth、Jamie Zawinski、Joshua Bloch、Ken Thompson 等在内的业界传奇人物，为我们讲述了他们是怎么学习编程的，在编程过程中发现了什么以及他们对未来的看法，并对诸如应该如何设计软件等长久以来一直困扰很多程序员的问题谈了自己的观点。本文选编自《编程人生：15位软件先驱访谈录》。

Seibel：你是从头到尾逐字阅读，跳过那些不理解的数学部分？

Crockford：是的，我会很快略读过星号太多的部分。我试图将熟悉 Knuth 的书作为招聘标准，但结果却大失所望，根本没几个人读过他的书。依我看来，任何自称为专业程序员的人都应该读过 Knuth 的书，至少也应该买过他的书。

Seibel：也就是说 Knuth 讲述的是如何实现最根本的东西，然后才有全景。即便清理了平台，但使用理性的方式构建大型系统和设计也是非常困难的。请问你是如何设计代码的？

Crockford：编写程序与对程序的生命周期进行迭代是不同的。通常，编写软件的原因在于我们知道将要修改它，而修改任何东西都不那么容易，因为很多时候修改意味着打破旧有的东西。

你不能期望使用这种方式完成所有事情，但还是应该尽力保证足够的灵活性，这样不管做什么都能适应。这就是我的观点。如何避免误入死胡同？如何保证灵活性？

这就是我喜欢 JavaScript 的原因之一。我发现 JavaScript 可以轻松实现重构，而重构一个继承层次很深的类实在太痛苦了。

Seibel：你觉得自己是个科学家、工程师、艺术家、工匠还是什么？

Crockford：我觉得自己是个作家。有时我使用英语写作，有时使用 JavaScript。归根结底，这完全取决于交流方式以及为了促进这种交流所采取的结构。人类语言与计算机语言在很多地方都是大相径庭的，我们需要阅读计算机语言，因此必须与之交流，我根据语言的这种交流能力判断计算机程序的优劣。在这个层次上，人类语言与计算机语言的差别不大。

Seibel：你对自学的程序员有什么建议呢？

Crockford：两个字：多读。现在有不少好书，去找些好书来看吧。如果从事Web开发工作，请找一些优秀的站点，看看他们的代码。话虽如此，其实我不太想这么建议。大多数Web开发者都是从“查看源代码”开始走上Web开发之路的，但直到现在，大多数源代码的质量都是非常低劣的。因此有一代程序员都被那些低劣的示例误导了，他们写的代码质量也不高。现在的情况已经有所改观，但依然有很多低质量的代码游离于你我之间，所以我是不太想给出这个建议的。■

专题：JavaScript第二季

Node上的JavaScript性能测试实践

作者/朴灵

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。

单元测试主要用于检测代码的行为是否符合预期。在完成代码的行为检测后，还需要对已有代码的性能作出评估，检测已有功能是否能满足生产环境的性能要求，能否承担实际业务带来的压力。换句话说，性能也是功能。

性能测试的范畴比较广泛，包括负载测试、压力测试和基准测试等。由于这部分内容并非Node特有，为了收敛范畴，这里将只会简单介绍下基准测试。

除了基准测试，这里还将介绍如何对Web应用进行网络层面的性能测试和业务指标的换算。

基准测试

基本上，每个开发者都具备为自己的代码写基准测试的能力。基准测试要统计的就是在多少时间内执行了多少次某个方法。为了增强可比性，一般会以次数作为参照物，然后比较时间，以此来判别性能的差距。

假如我们要测试ECMAScript5提供的Array.prototype.map和循环提取值两种方式，它们都是迭代一个数组，根据回调函数执行的返回值得到一个新的数组，相关代码如下：

```
var nativeMap = function (arr, callback) {
  return arr.map(callback);
};

var customMap = function (arr, callback) {
  var ret = [];
  for (var i = 0; i < arr.length; i++) {
    ret.push(callback(arr[i], i, arr));
  }
  return ret;
};
```

比较简单直接的方式就是构造相同的输入数据，然后执行相同的次数，最后比较时间。为此我们可以写一个方法来执行这个任务，具体如下所示：

```
var run = function (name, times, fn, arr, callback) {
  var start = (new Date()).getTime();
  for (var i = 0; i < times; i++) {
    fn(arr, callback);
  }
  var end = (new Date()).getTime();
  console.log('Running s d times cost d ms', name, times,
end  %% % - start);
};
```

最后，分别调用1 000 000次：

```
var callback = function (item) {
  return item;
};

run('nativeMap', 1000000, nativeMap, [0, 1, 2, 3, 5, 6],
callback);
run('customMap', 1000000, customMap, [0, 1, 2, 3, 5, 6],
callback);
```

得到的结果如下所示：

```
Running nativeMap 1000000 times cost 873 ms
Running customMap 1000000 times cost 122 ms
```

在我的机器上测试结果显示Array.prototype.map执行相同的任务，要花费for循环方式7倍左右的时间。

上面就是进行基准测试的基本方法。 为了得到更规范和更好的输出结果，这里介绍benchmark这个模块是如何组织基准测试的，相关代码如下：

```
var Benchmark = require('benchmark');

var suite = new Benchmark.Suite();

var arr = [0, 1, 2, 3, 5, 6];
suite.add('nativeMap', function () {
  return arr.map(callback);
}).add('customMap', function () {
  var ret = [];
  for (var i = 0; i < arr.length; i++) {
    ret.push(callback(arr[i]));
  }
  return ret;
}).on('cycle', function (event) {
  console.log(String(event.target));
}).on('complete', function() {
   console.log('Fastest is ' + this.filter('fastest').
pluck('name'));
}).run();
```

它通过suite来组织每组测试，在测试套件中调用add()来添加被测试的代码。

执行上述代码，得到的输出结果如下：

```
nativeMap x 1,227,341 ops/sec ±1.99 (83 runs sampled) %
customMap x 7,919,649 ops/sec ±0.57  % (96 runs sampled)
Fastest is customMap
```

benchmark模块输出的结果与我们用普通方式进行测试多出±1.99 (83 runs sampled) % 这么一段。事实上，benchmark模块并不是简单地统计执行多少次测试代码后对比时间，它对测试有着严密的抽样过程。执行多少次方法取决于采样到的数据能否完成统计。83 runs sampled表示对nativeMap测试的过程中，有83个样本，然后我们根据这些样本，可以推算出标准方差，即±1.99%这部分数据。

压力测试

除了可以对基本的方法进行基准测试外，通常还会对网络接口进行压力测试以判断网络接口的性能，这在6.4节演示过。对网络接口做压力测试需要考查的几个指标有吞吐率、响应时间和并发数，这些指标反映了服务器的并发处理能力。

最常用的工具是ab、siege、http_load等，下面我们通过ab工具来构造压力测试，相关代码如下：

```
$ ab -c 10 -t 3 http://localhost:8001/
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://
www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.
apache.org/

Benchmarking localhost (be patient)
Completed 5000 requests
```

```
Completed 10000 requests
Finished 11573 requests


Server Software:
Server Hostname:        localhost
Server Port:            8001

Document Path:          /
Document Length:        10240 bytes

Concurrency Level:      10
Time taken for tests:   3.000 seconds
Complete requests:      11573
Failed requests:        0
Write errors:           0
Total transferred:      119375495 bytes
HTML transferred:       118507520 bytes
Requests per second:    3857.60 [#/sec] (mean)
Time per request:       2.592 [ms] (mean)
Time per request:       0.259 [ms] (mean, across all
concurrent requests)
Transfer rate:          38858.59 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.3      0      31
Processing:     1    2   1.9      2      35
Waiting:        0    2   1.9      2      35
Total:          1    3   2.0      2      35

Percentage of the requests served within a certain time
(ms)
  50      2 %
  66      3 %
  75      3 %
  80      3 %
  90      3 %
  95      3 %
```

```
 98      5 %
 99      6 %
100     35 (longes % t request)
```

上述命令表示10个并发用户持续3秒向服务器端发出请求。下面简要介绍上述代码中各个参数的含义。

- Document Path：表示文档的路径，此处为/。
- Document Length：表示文档的长度，就是报文的大小，这里有10KB。
- Concurrency Level：并发级别，就是我们在命令中传入的c，此处为10，即10个并发。
- Time taken for tests：表示完成所有测试所花费的时间，它与命令行中传入的t选项有细微出入。
- Complete requests：表示在这次测试中一共完成多少次请求。
- Failed requests：表示其中产生失败的请求数，这次测试中没有失败的请求。
- Write errors：表示在写入过程中出现的错误次数（连接断开导致的）。
- Total transferred：表示所有的报文大小。
- HTML transferred：表示仅HTTP报文的正文大小，它比上一个值小。
- Requests per second：这是我们重点关注的一个值，它表示服务器每秒能处理多少请求，是重点反映服务器并发能力的指标。这个值又称RPS或QPS。
- 两个Time per request值：第一个代表的是用户平均等待时间，第二个代表的是服务器平均请求处理事件，前者除以并发数得到后者。
- Transfer rate：表示传输率，等于传输的大小除以传输时间，这个值受网卡的带宽限制。
- Connection Times：连接时间，它包括客户端向服务器端建立连接、服务器端处理请求、等待报文响应的过程。

最后的数据是请求的响应时间分布，这个数据是Time per

request的实际分布。可以看到，50%的请求都在2ms内完成，99%的请求都在6ms内返回。

另外，需要说明的是，上述测试是在我的笔记本上进行的，我的笔记本的相关配置如下：

处理器 2.4 GHz Intel Core i5
内存 8 GB 1333 MHz DDR3

基准测试驱动开发

Felix Geisendörfer是Node早期的一个代码贡献者，同时也是一些优秀模块的作者，其中最著名的为他的几个MySQL驱动，以追求性能著称。他在“Faster than C”幻灯片中提到了一种他所使用的开发模式，简称也是BDD，全称为Benchmark Driven Development，即基准测试驱动开发，其中主要分为如下几步其流程图如图10-9所示。

- (1) 写基准测试。
- (2) 写/改代码。
- (3) 收集数据。
- (4) 找出问题。
- (5) 回到第(2)步。

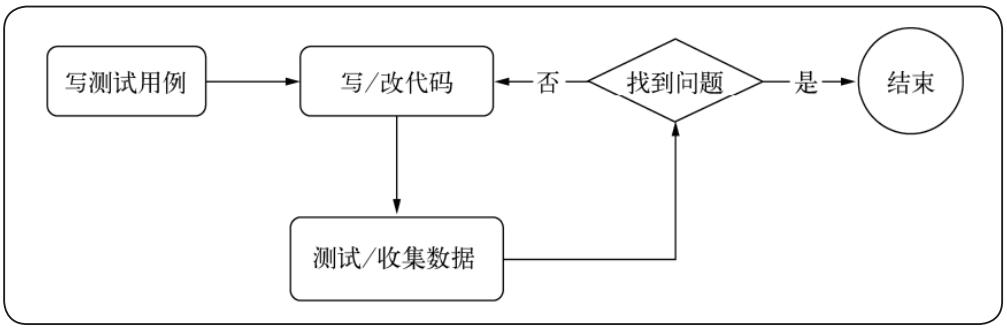


图1　基准测试驱动开发的流程图

之前测试的服务器端脚本运行在单个CPU上，为了验证cluster模块是否有效，我们可以参照Felix Geisendörfer的方法进行迭代。通过上面的测试，我们已经完成了一遍上述流程。接下来，我们回到第(2)步，看看是否有性能的提升。

原始代码无需任何更改，下面我们新增一个cluster.js文件，用于根据机器上的CPU数量启动多进程来进行服务，相关代码如下：

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

cluster.setupMaster({
  exec: "server.js"
});

var cpus = require('os').cpus();
for (var i = 0; i < cpus.length; i++) {
  cluster.fork();
}
console.log('start ' + cpus.length + ' workers.');
```

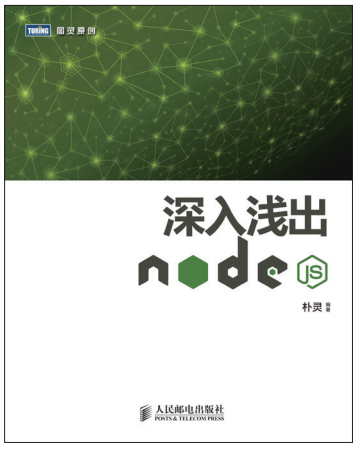
接着通过如下代码启动新的服务：

```
node cluster.js
start 4 workers.
```

然后用相同的参数测试，根据结果判断启动多个进程是否是行之有效的方法。 测试结果如下：

```
$ ab -c 10 -t 3 http://localhost:8001/
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://
www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.
apache.org/
```

```
Benchmarking localhost (be patient)
Completed 5000 requests
Completed 10000 requests
Finished 14145 requests

Server Software:
Server Hostname:        localhost
Server Port:            8001

Document Path:          /
Document Length:        10240 bytes

Concurrency Level:      10
Time taken for tests:   3.010 seconds
Complete requests:      14145
Failed requests:        0
Write errors:           0
Total transferred:      145905675 bytes
HTML transferred:       144844800 bytes
Requests per second:    4699.53 [#/sec] (mean)
Time per request:       2.128 [ms] (mean)
Time per request:       0.213 [ms] (mean, across all
concurrent requests)
Transfer rate:          47339.54 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.5      0      61
Processing:     0    2   5.8      1     215
Waiting:        0    2   5.8      1     215
Total:          1    2   5.8      2     215

Percentage of the requests served within a certain time
(ms)
  50      2 %
  66      2 %
  75      2 %
  80      2 %
  90      3 %
```



《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。

```
 95      3 %
 98      4 %
 99      5 %
100    215 (longest request) %
```

从测试结果可以看到，QPS从原来的3857.60变成了4699.53，这个结果显示性能并没有与CPU的数量成线性增长，这个问题我们暂不排查，但它已经验证了我们的改动确实是能够提升性能的。

测试数据与业务数据的转换

通常，在进行实际的功能开发之前，我们需要评估业务量，以便功能开发完成后能够胜任实际的在线业务量。如果用户量只有几个，每天的PV只有几十个，那么网站开发几乎不需要什么优化就能胜任。如果PV上10万甚至百万、千万，就需要运用性能测试来验证是否能够满足实际业务需求，如果不满足，就要运用各种优化手段提升服务能力。

假设某个页面每天的访问量为100万。根据实际业务情况，主要访问量大致集中在10个小时以内，那么换算公式就是：QPS = PV / 10h 100万的业务访问量换算为QPS，约等于27.7，即服务器需要每秒处理27.7个请求才能胜任业务量。

总结

测试是应用或者系统最重要的质量保证手段。有单元测试实践的项目，必然对代码的粒度和层次都掌握得较好。单元测试能够保证项目每个局部的正确性，也能够在项目迭代过程中很好地监督和反馈迭代质量。如果没有单元测试，就如同黑夜里没有秉烛的行走。

对于性能，在编码过程中一定存在部分感性认知，与实际情况有部分偏差，而性能测试则能很好地斧正这种差异。

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12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。

万涛：我用公益实现黑客信条

万涛被媒体称为“黑客教父”，在印尼排华、中美撞击等历史事件中，他曾组织发起跨国网络攻击。2001年，他成立中国鹰派联盟，宣告“我们要做民族的精英，我们会永远战斗不息”。经历了骄傲的黑客时代，他再次陷入迷茫期。很多人从此卷入了黑产，他承认自己有深深的“原罪感”。如今他用新的方法履行他的黑客信条：自由、平等、分享、互

兴趣的力量

“一个热爱生活的人是打不倒的，他总能找到乐趣。”

你从什么时候开始接触计算机？

我在大三时开始接触计算机。在上机时遇到病毒，我的反应是：“这个小程序不错啊，四两拨千斤。”从此泡机房成了我的主要活动。那时候比现在的码农们辛苦得多，MS-DOS时代初期全得靠自己。那时候互联网才刚开始，没有那么多现成东西可以用，就是写菜单也得自己琢磨，还得找书看别人的技巧。所以那时候基本上是从程序设计怎么封装UI，然后到代码编写调试到测试，基本都是一个人干。我现在还保留以前手写的代码。最早的时候去图书馆，因为买不起那么多书，复印也很贵，所以就得去抄。因为当时我们都是用汇编写的，所以就得抄汇编代码。书有时候印错了，代码也可能复印错，你得上机再去调，调不好，还得回去看书。现在可能因为太容易了，自主学习乐趣和驱动力可能就少了。

助。但是他的原则中从此多了一条：不要因为自己的理想而产生新的社会问题。从当初他不小心“撞上”的社会创业，到现在人所共知的互联网公益，他相信这里是一片蓝海。

后来毕业后为什么加入了铁路系统？

毕业后因为毕业分配和父母期望，我进了广州铁路局。到了单位我终于有了自己的计算机，大学毕业后上班第一个月什么事都没干，我每天主动以加班的名义留在办公室打游戏，我每天打到半夜三四点钟。那时大门都锁了，翻墙回宿舍，然后早上睡眼惺松再来上班。

后来实在无聊我就主动把工资管理程序给开发了。做完了主程序，我又自己写了两个游戏进去，一个彩票分析，还有一个音乐程序。然后还一直维护，不断丰富，完善它的逻辑结构，写很详细的文档，就是为了希望这个程序看起来是美的。我那时可以算是最懂电脑的会计了。

但是我想离开铁路系统，但是我自己走不了，因为我父母干涉力太强。所以最终我就唯有长期旷工，最后终于被开除了。然后我自己应聘去了普华永道。

当时有什么爱好？

我是这样认为，一个热爱生活的人是打不倒的，他总能找到乐趣。我刚离开国企去外企那段时间，有过一段比较窘迫的日子。因为没有积累，我不可能问家里要钱，要争一口气。我当时住在亲戚朋友一个房子下面的杂物间，反正不要钱，下面睡个大狼狗，我睡在阁楼上，等于就睡在狗窝里。周末是我最难受的时候，无事可干，刚去到广东朋友也少，然后我干什么呢？就天天看录像，跟民工一起看录像，看一些恶俗港产片什么的，我也看的挺来劲的。我吃饭跟出租车司机一起吃，出租车司机会有一个据点吃快餐比较方便，我也很喜欢跟他们聊点八卦。这样你也会了解很多行业生态，挺快乐的。

那你现在的爱好是什么？

我现在玩太阳能、玩业余无线电，自己还在重新学，对电力通讯也不熟，无线电里面还涉及到卫星通讯的知识，那里面编程量都是轻微的。其实玩太阳能发电站纯属为了体验，城市供电系统很发达，我只是不忍看到这么多屋顶是灰色的。既然不能逃离城市，就把城市变成森林。我去年完成了屋顶的太阳能电站和业余无线电基地台的架设。今年我要做智能屋顶，改造太阳能热水器，我要搞绿色灌溉，在屋顶上种菜。但是程序员是懒人，我肯定没有时间天天去伺候，我得要做自动灌溉，加上物联网检测土壤的酸碱度，或者尝试做无土栽培。

你为什么现在对电力通讯感兴趣了？

就是因为数字化在延展，过去我们的IT基本上只限定在信息化基础设施和流程应用领域。但是现在所有东西都在逐步数字化，电网也在走数字化。所以这里面都要延展，全社会的数字化延展的话IT也可以往这里面去延伸。

我职业做信息安全，但安全里面也有工业控制的安全。那些领域的安全可能就会比信息安全由于和大家的体验感受以及利益相关感更强而更得到大家的认同，因为很多人觉得信息安全不就是电脑不工作了吗。但是如果你们家没电了呢？地震的时候，通讯手机都可能没信号了，就像《灭世》里的谷歌工程师一样，顿时失业了。

红旗下的蛋

> “老外觉得我们特别组织化，怀疑我们跟政府有关系，你看到这里面都有组织部、宣传部、外联部，就跟搞社团似的。那个时代的黑客组织打上了那个时代的烙印。”

现在回头看90年代末的那些黑客运动，给人一种恍如隔世的感觉，您能帮我们重温一下当时的背景吗？

早期的互联网基本上是精英互联网，精英不是指社会顶层，而是指知识阶层。我们现在不都是叫草根互联网或者小白用户嘛。当时一个邮箱一个月两百块钱，一个Modem一两千块钱。我自己装第一台586花掉一万多，那时候一个月一千五的工资，一年的积蓄全花光了。所以那个时候相对家庭条件好一些，或者在高校才能够接触互联网。那个时候只有互联网泡沫没有互联网经济，就是说大家很强的经济目的，都很理想化，觉得互联网就是未来的共产主义。信息都是免费的，大家可以跟世界交流，你想在大学我们还在交笔友阶段，现在你可以直接跟世界各地联系，那时候还没有普及QQ吧，我们用IRC或者是用OICQ交流。

我原来小学同学就是印尼华侨，她是第一次印尼排华的时候被赶回来的，所以我对那个特敏感。然后发生了98年印尼排华事件，我就义无返顾的加入了绿色兵团。当时最初虽然只有三十多个人，但是里面大都是今天安全界的精英。Green Army是任天堂一款游戏的名字，我们倡导的是绿色的、和平的网络。这里面应该有正义，还有自由，其实可能就跟游戏玩家的心态差不多。早期的游戏玩家，不像现在打游戏还要打架或者拼装

备。那时候大家都是很纯粹的玩，因为传统社会还是塔形的社会，我们突然有了平行的世界，自我感觉都很好。

你们后来为什么要建设有中国特色的黑客文化？

这其实主要是因为绿色兵团分裂给我带来的思考。99年1月份我们在上海开了一个年会。开年会的核心问题是讨论商业化，就是要成立一个安全公司，然后做安全服务什么的。我不太认可。那时候我是自由程序员，后来业务多了就成立了一个公司。我从1998年开始跟公安系统打交道，可能还是对体制比较了解。

我在绿色兵团商业化过程中看到了一些关于利益的争吵，还有在上海、北京纷争中看到的商业谋略，我感觉有点寒心。我觉得个人可以搞商业化，但是组织这个东西不能商业化。用组织挣的钱再来支撑组织的发展，我觉得这个比较扯淡。

所以同样放在黑客这个话题，我觉得咱们没有自由主义的文化土壤，也没有风险精神。大家都想要别人的东西，这样的环境下成名比较容易。和国外的真正素质教育不同，国内大家出身都比较草根，hacking文化也有点山寨。我那个时候是一个愤青，从反印尼排华、反美这些事件过来。我觉得只有打上中国特色的烙印，才能凝聚更多人。如果你讲自由，那就是一盘散沙，是很难凝聚。所以我觉得在中国hacking里面，你可能要承担一定的社会使命。这个使命就是说政府是韬光养晦的，但是黑客可以跳出来。我们用网络的方式去表现我们年轻人的愤怒。

你在这几次黑客大战里面作用是什么？

煽风点火算吗？呵呵，早期我都是亲自参与的。像印尼、反日运动，反日应该是在抗战六十周年的时候。那个时候其实就是

拿电脑发邮件，最早用垃圾邮件生成器给日本首相一晚上发了一万多封邮件。

在中美黑客大战中我算是一个组织者。首先技术水平来讲当然是初级阶段，但在那个时候只能到那样了。导火索其实就跟马航找不到是一样，当时也是飞机找不到。中美军机在海南那边对撞。我们的飞机员找不到了，飞机掉下去了也没找着。如今十几年过去了咱们飞机还是找不着。当时美国只赔了点小钱我们就把他们的军机放回去了。我们想知道真相。当时先是《北京青年报》说美国一个黑客团体，因为中国扣了他们的飞行员和飞机，就对我们发起黑客攻击。那时候已经成立了鹰盟的前身鹰派俱乐部，所以我们也发起了反击。

后来为什么主动停止攻击了？

2002年的时候我看到有一家美国的研究机构IDEFENSE给我们出了一份研究报告。老美互联网威慑情报工作做的非常到家。从我们成立之初他们就开始盯上我们了，一直在我们里面潜伏。他们有懂中文的情报分析员。他就会在你BBS里面做长期分析，我们的聊天他都会去跟踪，然后再把活跃的ID拿去分析，出了一份60多页的评估报告。

当我把那份报告拿到手的时候，我就觉得跟人家比，我们真的有点像小孩子胡闹。当时还以为自己不错，但其实人家的顶层设计和各方面都相当齐全，我们当然宣布停止一切攻击。再攻击就相当于给别人做免费检测。

之后你们的行动宗旨是什么？又有什么样的活动？

然后那就要韬光养晦，要做强。在2001年时我们提出的口号就

是“刺刀上带着的思想”。这句话来自于对拿坡仑军队的评价。拿坡仑虽然称了帝，但是它代表的是新兴的资产阶级。而欧洲当时都属于封建王朝，所以他的部队就跟我们解放军似的，代表了先进的生产力。他的军队的士气都是来自于民众的支持，所以在战场上才能创新才能够横扫欧洲。我们觉得hacking是把双刃剑，用不好会伤到自己，所以我们应该有一个道德的度量。

当时我们提出自己的主张，你可以看到我们以前的章程。老外看不明白，觉得我们特别组织化，怀疑我们跟政府有关系。这就跟我们60后、70后受到的熏陶有关。崔健有一首歌叫《红旗下的蛋》，我们就是红旗下的蛋。我们习惯性把自己组织化，你看到这里面都有组织部、宣传部、外联部，就跟搞社团似的。相比之下老外的黑客组织都特别自由散漫。所以从这个层面来讲，我觉得那个时代的黑客组织是打上了那个时代的烙印的。

现在的黑客组织原来的成员还有多少？

绿色兵团散伙以后，有一些去搞安全公司，创业了，有一些被抓了。后面分裂出来的一些小孩随着我们进入迷茫期，有一些还进入到黑产。所以也有人把我封成“安全界十大恶人”之一。因为我们带了很多小朋友进入这个领域，但是又没把他们带好，所以我们是原罪人。

何德全院士在2002年的一个安全文化的演讲中提到黑客是国家宝贵的财富，但是黑客如果泛化的话会产生很大的危害。但是事实上后面的确走上泛化了，有很多年轻的组织或者论坛不错，保持了技术性的延续。也有一些一方面这样做，一方面自己去做一些黑产。这里面缺乏约束，跟体制之间也没有对话。

地下互联网的现状怎么样?

地下黑产我还是比较了解的。我们IDF实验室就是一直有做一些这样的分析和跟踪。今天的黑产相当复杂，早期是一些小黑，就是所谓的圈中人。到后来就是各种群体都进来了，比如原来是开黑网吧或者做传统产业的，发现这个东西来钱快、风险低。

现在的李俊（熊猫烧香作者，熊猫软件安全顾问、买软件网首席执行官），他自己其实没得到多少钱，他是下游链的，上游链和中间的桥梁可能挣到了更多的钱。现在黑产已经形成一套产业体系，有各种环节在里面。上游链就是投资人，有渠道商做分发的，有客户就是黑产需求方。另外有脚本小子，专门写一些小代码的。也有写木马工具的，也有专门教人的，什么人都有，这个产业生态链比较复杂。互联网圈关心的都是业务层面，网游、广告都是做这些的。现在电商盯的是钱，互联网安全圈很大程度上盯的也是钱。

但是还是有一些独立自主安全厂商，其实也就是我们那一代去形成的，他们是以独立安全价值为宗旨的，就是强调安全的独立性。今天安全圈也产生了博弈，这个博弈就是安全价值，是以360为代表的互联网安全企业和以传统安全厂商为代表的独立安全观的博弈。就是说360其实把安全做了对价，他不是靠安全挣钱，他是靠入口带来的流量控制权挣钱。那么独立安全观就是安全不能靠对价去挣钱，安全只能靠安全价值挣钱。比如你为一个保安付费，因为这是保安的专业，而不是因为这个保安提供了免费服务。这样会使你缺乏中立性。

你怎么评价匿名者（Anonymous）？你觉得他们算是公益组织吗？

匿名者的核心骨干大部分被抓了。现在他们的技术水平可能是口水更多吧，就是都是靠嘴说我要攻谁。匿名者一开始就不是一个特别严密的组织，当然特别紧凑的话也很难搞成这样，早被当成基地组织打了。 从法律角度来说匿名者肯定是违法的。但是它倡导：黑客是一个乌托邦。而整个儒家文化讲的是秩序，这种文化认为黑客这样的组织一旦大了，就意味着乱，意味着失去平衡。所以我自己认为hacking是一种平衡的艺术，它有点像侠客文化似的，侠客要靠自己的能力除暴安良。虽然其中有很多虚构的成分，但是我觉得在今天这个技术时代，的确有了这样一个可能，就是个人的力量或者小团体的力量，有可能带来这样的博弈。比如拿微软来说，微软发展到今天，一直有个体的力量来挑战他，挑战者不一定是商业竞争对手。这里面很大的力量就是黑客，因为黑客不断地在给微软找茬。

对于匿名者，我觉得首先要站在对于乌托邦的情节理解上去理解匿名者，理解这里面的年轻人。当然我觉得这个理解是要用实力去说话的。就跟当初从绿色兵团一直到鹰盟、红盟这些，后来就像小孩子的游戏一样，一旦泛化，总体的技术水平就被平均了。如果早期的匿名者属于70后，那现在就是一帮90后在这儿做。他们可能说的更多，但口气还要保持原来的。但是不要去嘲笑这些年轻人，虽然他们现在是junior，但是中间也会有出类拔萃的，也会成长。

你怎么看待网络自由？

这个也是我觉得困惑的地方，这个困惑从三点上来说。一是互

联网日益成为一个商业社会，而且这个商业里面有舆论自持的倾向。可能跟国外对比的话，大家会觉得这个互联网特别有中国特色，野蛮生长，没有规则。它产生了两个极端的面，一个是觉得要管，就像成龙说的“中国人一定要管”，这是因为我们受传统社会架构影响，认为一切是有序的。而互联网看起来像黑暗森林似的，用周鸿祎的话说“谁的屁股都不干净”。所以我一直有一个疑惑：互联网究竟使人变聪明还是变傻了。有一些老辈人也在问我，现在啥事都百度、谷歌，过去都得自己去找答案了，现在百度一下就知道了，这个真是我们需要的互联网吗？。互联网巨头掌握信息话语权，它说什么就是什么。所以现在才会有那么多互联网软文。

第二点就是用户，所谓互联网围绕着用户，用户是第一，用户是体验。用户这个群体由于泛化得特别宽，早期的用户是精英用户，所以不怎么在乎商业层面，他关心如何利用。今天用户的体验泛化了，比如说用户的隐私，还有用户自己的意识，用户的权益。每年搞3·15，大家也能看到了这不是所谓的公平。是用户变聪明还是变傻了，我们迎和哪一种用户，还是说应该去承担引导和培养用户这样的角色，或者我们要等用户去成长。 第三个问题是国家，其实国家还是人的集合，我的理解就是，所谓互联网的分歧就是50后、60后、70后、80后、90后的分歧。大家用不同的眼光去选择互联网、参与互联网。50后的体验少，但是他对互联网有强烈的责任意识，他觉得要管起来。70后是一个承上启下的群体，60、70后更倾向于智慧，倾向互联网里面要有理性的内容和更智慧的治理。可能对于80后、90后来说，更多的是希望看到互联网带来的机会，他们要用互联网颠覆传统给予他们的压力。所以也有很多互联网从业者都是顺着80后、90后的需求去颠覆。

益云

> “社会地图的作用就是数据化，比如儿童性侵，如果一张地图上一年有好几千起，你就会觉得很可怕。”

你从原来的知名黑客到现在开始做公益，你的理想有变化吗？

没变化。你经常会看到我们说的是自由、平等、分享、互助，这就是公益。这些理念能够通过公益的行为得到最广泛的普及，而不需要通过黑客大战的方法。因为你黑掉别人还是不会解决问题。黑掉老美又怎么样，你写个几个中文，你写 fuck USA，他们只会觉得中国人不可理喻。还不如我去帮助自己的国家，帮助自己的人民。公益代表了社会理想，这不是一个直接赚钱的产业，都是模式迄今还是需要别人捐款的，所以更需要可持续、可复制的模式。做了那么多的助学，每年要去招这么多支教，如果这些东西都能像 MOOC 教育一样，就能降低很多成本。

自己做公益遇到什么挫折？

我们很多人也做过希望工程和希望小学，结果没搞成，为啥？跟当地教委意见不一致，比如说我们关心软件，那边关心硬件。我也做过一种像免费午餐那样的项目，针对湖南一个镇中学的留守儿童或者比较困难的家庭，每天资助他们的午饭钱。我做了两年，但是后来我觉得这个事不是我擅长的。虽然我捐的钱不多，但是为了看看他们，我每次都要飞过去，花的钱可能比那个还多。那帮小孩初中毕业以后还是会去打工，因为初

中是义务教育，高中不是。结果就变成我给了他希望，却没有实现。我曾带他们来北京玩，原来的生活他们觉得挺快乐的，来这里一看城里人可以这样过日子，反而产生了落差。所以我觉得这个公益可能是我们城里人想象的公益，只是满足你自己而已，并没有真正促进社会公平。

找到了更好地参与公益的方式了吗？

可能我们需要做的是让信息对称。比如说他们要有好的老师，他们的老师视野也能放开，知识面不比城里老师差多少。在这个层面上我们搞技术的人不擅长，但是有职业的公益组织可以做这些事情。但是他们的技术是比较弱的，他们做的方式可能是比较原始的。我们能不能退而结网去支持这种有想法的组织，比如那些能够用互联技术去促进信息公平的公益组织。

这个方法上更科学，所以我觉得我应该退后，就是用技术去支持公益。这样反而更是真正在践行黑客文化和黑客理想，而且是落地的。如果是匿名者，最多跟公知一样，提出问题，但是没有做任何解决问题的事情。可能还不如盗版干的事，盗版起码普及了我们的IT使用者。过去我们穷，的确支付不起这个版权费。但是今天盗版威胁到了我们知识创新。

像你们这种用技术做公益的组织都有哪些？都有什么样的方式？

其实在国内现在开始慢慢多了一些，比如有专门做IT环保的，有一些做MOOC教育的。我们曾经帮助客户开发乡村教师培育，包括梦想图书馆，我们都是用信息技术的方式去做的。另外还有一些是用IT资源来做的，或者用上一些IT的思维，产品化的思维。

我们国家互联网门户做的比较多，国外像谷歌这些也会做，但是更多的不是这个大公司去主做。大公司做自己的东西，会开发一些资源给其他组织或者创新机构去做，这是个不同。第二，国外的创新点比较多，创新点在于它有一些互联网的模式，包括社会众筹这样的方式。

我们现在说的公益，就是慈善做好事，捐钱。其实这个边界正在模糊，很多里面可以有商业模式。有的时候我认为互联网公司很大层面都是公益，因为它的很多东西都是免费的。比如微博，大家在上面交流社会问题，有利于社会问题的解决，这难道不是公益吗?

只是在一些不能产生商业模式的地方，用技术来做还偏少，或者说我们公益组织在这些方面能力不足。在我看来很多问题完全可以用技术做很好的解决，但是公益组织在这方面的协作非常不好。

有哪些公益领域需要注入技术力量?

我基本上关心三个领域，第一是教育，第二是儿童，比如说拐卖、保护相关的问题，第三个是环境。 我觉得人出生是没有办法生而平等的，有地区差异、文化差异、经济差异、社会地位差异。人有很多基本权利，但是我觉得“信息权”也很重要。信息权应该比教育权延展一些，它包括知情权在内。教育其实是一个复制过程，言传身教，所以很大程度上是可以通过技术来弥补的。我们国家也做了很多，比如村村通，这个是基础设施层面。但是关键是利用，我们城市里头还有这样的问题，小升初，不就是所谓教育资源竞争吗? 今天我们有互联网教育，一个优秀老师为什么只能给这一个学校上课，如果变成Mooc的话不就能给这么多学校参与了吗。教育是第一个要被突破的东西，因为它影响到未来。只要产生文盲或者知识信息偏差就是在制造社会问题。

公益创业跟普通创业有什么区别吗?

一般创业更关心你的商业模式和用户。我认为公益创业就是需要你能守的久一点，而商业创业拼的是快。我们称公益创业为社会创业，社会创业一般立足于解决社会问题。它的周期可能会更长，从商业回报上来说，一定要找到可持续的模式。而且它要秉承一个原则，就是不要因为你的理想而产生新的社会问题。

比如我做手游，那我肯定要求快，因为手游的生命周期都不长。但是我可能不太会关心这个手游带来的其他问题，其他问题交给社会解决。比如小孩子玩游戏成瘾。我倒是反对网瘾这一说，因为我觉得打麻将也成瘾，抽烟也有瘾，这些都是一样的。很多时候也是因为现实里的问题造成网络成瘾，比如说教育很枯燥，上课不爱听。我大学也打游戏，那时候没有互联网我还打游戏，何况有互联网。

益云目前的核心产品是什么?

益云的核心产品是两个，一个是公益广告，一个是社会地图。这两个产品有什么关系？我们未来的商业广告都应该是软广告，一些广告成为很好的传播短片。比如汽车广告，他可能是强调汽车安全的，那个汽车很安全，所以狗狗穿马路的时候能刹住车。公益广告也可以用这种形式，所以未来越来越多的广告是软广告。我们就希望把这种广告对接，所以我们现在在聚合这个门类，然后逐步让自己成为渠道，这是它的商业模式。

我们的社会地图是什么？公益组织一方面产生内容上比较弱，另外一方面产生数据能力不足。它们很多没有自己的网站，或者有网站没有钱或者人去运营。其实公益应该让大家看到它的

数据。比如空气污染，PM2.5爆点了，到处检测，这个就是数据化。公益里面的儿童问题，比如儿童性侵，如果一张地图上一年有好几千起，你就会觉得很可怕，就可以促进社会去加速解决。

益云地图的作用就是支持社会问题数据化。环境上比如说水安全的数据，就是给他们提供一个平台，让他们把内容、数据呈现出来。过去很多公益项目就是靠讲故事，但是现在你还可以把你的数据拿出来。这样也提升了公益的质量，这里面逐步会筛分出一些，有一些靠故事，更多要有数据呈现。这是一个大数据的时代。

现在益云的主要收入来源是什么？

现在一部分是定制化的业务。我们在益云创立之初没有那么快做自己的产品，因为也怕走弯路。客户里面有一些机构基金会业务已经比较成熟了，但是他们需要从财务信息化开始，然后业务逐步信息化，自然就会产生共享。所以我们希望支撑里面的一些龙头。我们跟一般的IT公司相比，更专注这个领域，我们在行业里面还要做一些事情，所以我们将来可以促进他们一些信息的融合。这是我们的主要收入，未来可能就在益播、益图的互联网广告那一块。

现在互联网公益的发展情况如何？

三年前益云刚筹备的时候，互联网公益才刚开始。今天互联网公益大家都认同，其他网站门户都在做，因为他们看到了这里面的价值，会为他们带来流量。新浪微博公益是新浪主推的项

目，因为这个东西第一正能量，第二还有眼球，也有参与性。所以从这个层面上说我认为社会公益是一个蓝海，很多以后的商业模式都会发生转换。

公益生态圈不是我们一家能够创立的，是需要在发展过程中去壮大的。我们为什么叫益云？“益”大家都知道是公益，“云”是每一个公益组织，每个个体想做好事，都是一滴水珠，我们希望把它汇聚成云，成云以后才有可能产生生态效应。另外，云是没有边界的，没有形状的。

你们现在团队成员都是从哪里来的？

主要是两部分，一部分来自原来鹰盟的志愿者，加入鹰盟的时候他们是大学生，后来毕业工作了，有一定经验后加入了这里，成为我们专职工作人员，还有一部分来自社会招聘。我们现在大概是20个人，应该有三分之二都有技术背景。

什么样的程序员愿意加入益云？

我们这里的程序员最重要的是要有社会性的想法，有想法的话他就会想做一些不同的事情，而不是单纯想挣很多钱。跟所有创业公司是一样的，我不能现在承诺给你很高的收入，但是在益云有一个好处，就是视野。我们这方面资源太丰富了，比如我们与各种高大上的国际组织或者行协会参加很多活动，跟很多公益机构有深入接触，他们有很多关于环保、儿童的项目，你可以参与其中，每放一个地图，你都可以加深对社会问题的理解。另外，这是一个跨界圈，你会接触不同的人，比如名人、明星，或者说大牛，都有可能。所以可能会让你的事业得到较快成长。

你们有没有就是参与一些开源项目？

我们还有一个IDF实验室，安全我们保留了一块，现在处在孵化的阶段。我们主要侧重在做安全在线教育，这里面我们有一个开源项目就是做社工库API。现在很多网站的账号都被偷了，产生了很多撞库行为。现在我们把这个库整合起来，反过来做成一个开源API。当你的用户注册的时候，可以帮他检测账户是否已经泄露。然后我们可以提供一些防撞库的措施，它可以有一定简单的商业模式。也就是说让大家来把这个所谓的有害资源，变成一个有益资源。

最有趣的语言

作者 / Ola Bini

Ola Bini，他从7岁开始编程到现在，如果他不在电脑前，那他很有可能在研究箭术，武术，或者在理发店里。他一直关注与安全、AI，以及编程语言。他是JRuby开发团队的早期成员，让它从一个玩具变成一个工具，并顺便写了一本《JRuby实战》。他设计了两种编程语言，它们都是他的试验品。去年他一直都在学习关于基因和分子生物学的知识，因为结合大数据的信息，基因测序技术的发展已经让攻克癌症变成可能。

编程之道

道生机器语言，机器语言生汇编器，汇编器生编译器，于今，语言逾万。
语言纵微，自有目用，皆可道软件之阴阳，皆于道中占一席之地。
但若可能，远离COBOL。

一场语言的复兴正在酝酿之中，而且已经持续了很多年。对于语言极客来说，当下的生活可能是自20世纪70年代以来最好的年代了。我们亲历了很多新语言的诞生，也看到很多古老的语言在新的领域里重焕生机——或者像Erlang一样，它所能解决的问题在今天突然变得至关重要。

为什么我们今天会看到语言的复兴？很大一部分原因在于我们正在面对更艰难的问题。代码库一天比一天臃肿，传统的工作方法不再有效。我们在越来越紧迫的时间压力下工作——尤其是创业公司，其生死就取决于产品的发布速度。另外，我们面对的问题越来越依赖并发与并行能力。面对这些问题，传统方法早已过时，因此，诸多开发者转向不同的语言，希望新的语言能以更简单的方式解决问题。

一方面，我们对新方法的渴求正与日俱增，而另一方面，我们也拥有了更加丰富的资源，可以创建新的语言。创建语言所需的工具已经达到了全新的高度，甚至在几天之内就可以打造出一个可用的语言。一旦有了可运行的语言，我们就可以把它放

到任何一个成熟的平台（比如JVM、CLR或者LLVM）上运行。如果语言运行在这些平台上，它就可以访问平台上的库、框架和工具，也正是这些东西都让这些平台如此强大。这也意味着，语言的创造者不必重新发明“轮子”。

本文将讨论当下一些有趣的语言。我相信每个程序员都能从本文中列举的语言中有所斩获。不过这样的语言清单必然是主观的，且会随时间而改变。我希望在未来几年之内，这些语言都不会过时。

为什么语言很重要

构建计算机科学的基础之一是邱奇-图灵命题（Church-Turing Thesis）。该命题及其相关结论有效地证明了，不同的编程语言在基本层面上并没有区别。你用一种语言可以实现的事情，用另一种语言同样可以。

既然如此，我们又何必在意编程语言之间的区别呢？为什么你不继续用Java写程序呢？想想吧，如果语言真的无关紧要，为什么有人会发明Java，又为什么会有人用它呢？玩笑归玩笑，这里的重点是，我们很在乎选择编程语言，而理由很简单，就是这些语言各有所长。即使能够使用任何语言做任何事情，但很多情况下，在一种语言里做某事的最好方式却等价于创建了另一种语言的解释器。这有时会被称为格林斯潘编程第十定律（Greenspun's Tenth Rule of Programming）：

> 任何C或Fortran程序复杂到一定程度之后，都会包含一个临时的、只有一半功能的、不完全符合规格的、到处者是bug的、运行速度很慢的Common Lisp实现。

事实上大多数语言都可以完成大部分工作，区别只是在于实现起来的难易程度。因此，为某项任务选择一种正确的语言，就意味着接下来所有工作都会变得更为轻松。而了解多种语言，也意味着你在解决特定问题时能有更多的选择。

作为程序员，我认为使用哪种编程语言是最重要的选择。语言的选择需要慎重，因为其他一切工作都依赖这门语言，甚至可以说语言的选择是项目存亡的关键。

一些有趣的语言

我知道众口难调，也无法在一篇文章中满足所有人的喜好。如果你所钟爱的语言没有列出来，并不意味着我认为它无趣。我考察了许多语言，却由于篇幅限制，无法一一展示，最后只好甄选出其中几种列在本文中。它们最能体现出不同类型语言之间的差异。有兴趣的话，你也可以像我一样甄选出其他的语言。如果你没看到自己最爱的语言出现在这里，请不要失望，可以写邮件告诉我为什么你认为某种语言应该在本文列出。或者你也可以写一篇相同模式的博客，介绍你所喜欢的有趣的语言。

本文不会介绍如何下载和安装文中提到的语言。此类内容变化很快，因此你最好求助Google。我也不会为你展示某种语言的所有内容，只是介绍一些值得注意的特性，希望以此引起你对它们的兴趣。

Clojure

Rich Hickey于2007年发布了第一版的Clojure。从那时起，Clojure开始迅速流行起来。现在，Clojure背后已有商业化

运作，募集了大量开发资金，并且有几本非常不错的相关书籍。Clojure本身发展得也很快——自从第一版发布以来，已经有4个重要版本发布：1.0，1.1，1.2和1.3[1]。这4个版本给Clojure带来了极大的提高和改善。

① 截止翻译时，Clojure 1.5已经发布。——译者注

Clojure是一种Lisp方言，但却并不是Common Lisp或Scheme的某种实现。事实上，它从众多不同的语言中汲取了很多灵感，因此是一个全新版本的Lisp。它运行于JVM之上，可以轻松访问任何既有的Java程序库。

有Lisp编程经验的读者一定知道“列表”（list）是Lisp的核心。Clojure将此特点发扬光大，并在列表之上加入了额外的抽象。因此，数据结构成为了该语言的核心。不仅是列表，还有向量、集合、Map，所有这些数据结构都有相应的语法，Clojure程序的代码本质上即是由这些数据结构写就的，也最终会表现为这些数据结构。相比于其他语言，Clojure有一点不同，它的数据结构是不能修改的。如果要修改它们，首先要描述出这个修改是什么样的，然后就会返回一个新的数据结构，旧的那个依然存在并可以使用。这似乎过于浪费。的确如此，它确实不如直接摆弄字节码效率高。但它也不像你想象的那样慢，因为Clojure的数据结构实现是非常成熟和智能的。刚才提到的数据结构的不变性，使Clojure可以轻松完成很多其他语言看来非常困难的工作。不可变的数据结构有一个非常明显的好处：由于永远不能修改，因而肯定能保证它们满足线程安全的要求。

今天，我们选择Clojure的主要原因在于它有一套非常周密的模式，十分适合处理并行与并发执行。Clojure的基本特点是不可变性。如果你需要可变性，那么可以根据控制可变性的预期方式，创建一些特殊的数据结构来进行模拟。

比如说，你想要确保某3个变量能同时改变，那么在Clojure里做到这一点并不难，只要将这些变量存入几个引用ref，然后使用Clojure的软件事务性存储（STM）来协调变量访问即可。

总之，Clojure有很多不错的特性。它与Java的互操作性非常实用。我们可以完全控制程序中的并发行为，同时不必使用诸如锁、互斥量（mutex）等容易出错的方法。

接下来，我们看看Clojure的代码。第一个例子是简单的“Hello World”程序。和很多所谓的脚本语言一样，Clojure在顶层就可以执行任何东西。下述代码首先定义了一个函数，名为(hello)，然后传入两个不同的参数调用它。

MostInterestingLanguages/clojure/hello.clj

```
(defn hello [name]
   (println "Hello" name))

(hello "Ola")
(hello "Stella")
```

如果定义了clj命令，我们可以运行这个文件获得以下预期结果：

```
$ clj hello.clj
Hello Ola
Hello Stella
```

前文提到过，在Clojure中使用数据结构非常方便，而且它们也提供了很强大的操作。下面的例子示范了如何创建不同的数据结构，并获取它们的元素。

MostInterestingLanguages/clojure/data_structures.clj

```
(def a_value 42)

(def a_list '(55 24 10))

(def a_vector [1 1 2 3 5])

(def a_map {:one 1 :two 2 :three 3, :four 4})

(def a_set #{1 2 3})

(println (first a_list))
(println (nth a_vector 4))
(println (:three a_map))
(println (contains? a_set 3))

(let [[x y z] a_list]
   (println x)
   (println y)
   (println z))
```

这段代码的最后几行所做的事情是最有意思的。let语句可以把集合解构成几部分，本例只是拆分了一个含有3个元素的列表，然后分别赋给x、y和z。事实上，Clojure可以任意嵌套和解构这样的集合。

运行上面的代码，将得到如下输出：

```
$ clj data_structures.clj
55
5
3
true
55
24
10
```

使用Clojure的数据容器时，通常都是这样添加或移除元素的，继而可以创建新容器。无论使用什么容器，Clojure都会支持3个函数，满足你的大部分需要。这些函数是(count)、(conj)和(seq)。(count)函数不言自明。调用容器的(conj)函数可以向容器中添加新内容，不过，底插在哪取决于容器的类型，比如使用(conj)向List中加入元素，新元素会在List的头部。对Vector来说，新元素则会在尾部，而对Map而言，(conj)会添加一个键/值对。

为了支持数据容器上的一些更常用的操作，Clojure提供了一个名为Sequence的抽象。调用(seq)可以将任何一个集合转化为Sequence。一旦有了Sequence，就可以使用(first)和(rest)遍历它了。

那么，这在实践中怎么用呢?

MostInterestingLanguages/clojure/data_structures2.clj

```
(def a_list '(1 2 3 4))
(def a_map {:foo 42 :bar 12})

(println (first a_list))
(println (rest a_list))

(println (first a_map))
(println (rest a_map))

(def another_map (conj a_map [:quux 32]))

(println a_map)
(println another_map)
```

在这段代码中，先分别打印了一个list，map的第一个元素

以及剩余的部分。然后，向现有的map中添加了一个键/值对，从而创建新的map，而原来的那个map保持不变。执行这段代码，输出如下：

```
$ clj data_structures2.clj
1
(2 3 4)
[:foo 42]
([:bar 12])
{:foo 42, :bar 12}
{:foo 42, :quux 32, :bar 12}
```

Clojure非常容易和Java集成。事实上，有时候很难分辨出Clojure与Java代码的界线。比如，我们前面谈到的Sequence抽象机制。它其实就是一个Java接口。Clojure与Java库的互操作通常就是直接调用Java库。

MostInterestingLanguages/clojure/java_interop.clj

```
(def a_hash_map (new java.util.HashMap))
(def a_tree_map (java.util.TreeMap.))

(println a_hash_map)
(.put a_hash_map "foo" "42")
(.put a_hash_map "bar" "46")

(println a_hash_map)
(println (first a_hash_map))
(println (.toUpperCase "hello"))
```

任何在Classpath上的Java类都能很容易地实例化：可以调用(new)，并将class作为参数传入；还有一种特殊形式——在类名后面加一个点(.)，把它当作一个函数。获得了Java实例后，可以像使用任何Clojure对象一样使用Java对象。调用

对象上的Java方法需要特殊的语法，即在方法名前面加一个点。以这种方式调用Java方法并不局限于通过Java类创建的对象。实际上，Clojure的字符串只是普通的Java字符串，所以，我们可以直接对它调用toUpperCaes()。

执行上述代码，将得到下面的输出：

```
$ clj java_interop.clj
#<HashMap {}>
#<HashMap {foo=42, bar=46}>
#<Entry foo=42>
HELLO
```

前面提到了Clojure的并发问题，因此，我想演示一下STM的使用。虽然这听上去非常高深，但实际中用起来却相当简单。

MostInterestingLanguages/clojure/stm.clj

```
(defn transfer [from to amount]
  (dosync
   (alter from #(- % amount))
   (alter to #(+ % amount))
   )
  )

(def ola_balance (ref 42))
(def matt_balance (ref 4000))

(println @ola_balance @matt_balance)

(transfer matt_balance ola_balance 200)

(println @ola_balance @matt_balance)

(transfer ola_balance matt_balance 2)

(println @ola_balance @matt_balance)
```

这个例子做了很多事情，不过需要注意的是(ref)、(dosync)和(alter)。在代码中调用(ref)将创建一个引用，并赋给它初始值。@符号用来取出引用的当前值。(dosync)代码块中执行的任何操作都会在一个事务中完成，这就意味着没有任何代码可以看到(dosync)代码块处于不一致的状态。

然而，为了让这种情况成为可能，(dosync)可能会多次执行其代码。(alter)会真正改变引用的值。井号(#)语法在Clojure中用于创建匿名函数，这种语法非常独特。

运行这段代码将会得到预期的输出。这段代码其实没有使用任何线程，不过如果真有很多线程需要交错使用这些引用的话，也不必担心结果的正确性。

```
$ clj stm.clj
42 4000
242 3800
240 3802
```

我本想在这一节中向你展示更多的Clojure特性，不过此刻我们必须开始下一个语言了。如果想获得更多关于Clojure的信息，可以看一看下面的学习资源。我强烈建议你尝试下Clojure——那的确是一种令人愉悦的体验。

学习资源

关于Clojure的书已经有不少了。Stuart Halloway的*Programming Clojure* [Hal09]是第一本关于Clojure的书，而且至今仍然可以用这本书来了解Clojure。这本书的第二版刚刚发布，并且多了一位合著者——Aaron Bedra。

同时，我也是*The Joy of Clojure*[FH11]一书的粉丝，它会教你如何写出地道规范的Clojure代码。

对语言有了全面的了解后，我推荐你看看Clojure的主页（http://clojure.org）。其中有很多不错的资源，通过阅读该网站的文章，你能学到很多关于Clojure的知识。

最后，Clojure的邮件列表是学习过程中的重要辅助。这是个非常活跃的列表，你经常会看到Clojure的核心成员回答问题。这里也经常会探讨Clojure未来的新特性。

CoffeeScript

在过去几年里，JavaScript迅速流行起来。其中的主要原因在于，越来越多的公司选择HTML5作为应用程序的主要发布机制；此外，对于Web应用程序来说，创建更好的用户接口也越来越关键。基于上述原因，我们用JavaScript写的程序越来越多。但是这里有一个大问题，JavaScript有时很难写好。它有一个奇特的对象模型，而且工作方式表面上看来总是不那么合理，语法也非常笨重。

于是就有了CoffeeScript。

CoffeeScript是一种相对比较新的语言，不过，GitHub上的排名说明它已经是一个最有趣的项目了。在这个语言集合中，它还是一个“不合群的家伙”，因为它还算不上一门完整的语言。它更像JavaScript之上薄薄的一层——它能编译为可读性相当好的JavaScript代码。它从Ruby和Python中获得了大量灵感，如果用过其中任何一种语言，你都会觉得CoffeeScript非常顺手。

就和Python一样，Coffee使用缩进组织程序的结构。它的主要目标是要比JavaScript更具可读性和易用性，这在很大程度上是指语法层面上的。

尽管语法是很重要的因素，但CoffeeScript并不仅仅是语法不同。它也支持其他高级特性，比如推导（comprehension）和模式匹配。

CoffeeScript还很容易建立基于类的继承结构，它有特定的语法支持。JavaScript中更令人苦恼的一点，就是有了一个正确的继承结构后，如何把它们组织到一起？如果你刚刚从某个标准的基于类的面向对象的系统转过来，那么你就会发现CoffeeScript更容易接受。

到目前为止，CoffeeScript并没有任何主要的新功能，但是它能使应用程序的JavaScript部分更简单。它还能使JavaScript代码更一致，容易阅读和维护。

那我们就开始吧！看看“Hello World”的CoffeeScript代码：

MostInterestingLanguages/coffee_script/hello.coffee

```
greeting = "hello: "
hello = (name) =>
    console.log greeting + name

hello "Ola"
hello "Stella"
```

如果你安装了CoffeeScript和Node.js，可以这样运行：

```
$ coffee hello.coffee
hello: Ola
hello: Stella
```

通过这个简单例子可以看到，我们创建的方法是一个词法闭包，使用的变量是greeting。和Ruby一样，不需要括号。解析器尽量简化这部分工作。

在CoffeeScript中创建内嵌对象是非常容易的。既可以使用显式的分割符，也可以使用缩进表明对象的起始与终止。

MostInterestingLanguages/coffeescript/nestedobjects.coffee

```
words = ["foo", "bar", "quux"]
numbers = {One: 1, Three: 3, Four: 4}

sudoku = [
    4, 3, 5
    6, 8, 2
    1, 9, 7
]

languages =
    ruby:
        creator: "Matz"
        appeared: 1995

    clojure:
        creator: "Rich Hickey"
        appeared: 2007

console.log words
console.log numbers
console.log sudoku
console.log languages
```

运行这段代码，将得到如下输出：

```
$ coffee nested_objects.coffee
[ 'foo', 'bar', 'quux' ]
```

```
{ One: 1, Three: 3, Four: 4 }
[ 4, 3, 5, 6, 8, 2, 1, 9, 7 ]
{ ruby: { creator: 'Matz', appeared: 1995 }
, clojure: { creator: 'Rich Hickey', appeared: 2007 }
}
```

打印的输出有点儿乱，不像创建语句那么干净，对此我稍感遗憾。不过我想这正是CoffeeScript的一个优势——可以非常干净地创建内嵌对象。

CoffeeScript的另一个优势在于可以使用for关键字定义对象的推导：

MostInterestingLanguages/coffee_script/comprehensions.coffee

```
values =
    for x in [1..100] by 2 when 1000 < x*x*x < 10000
        [x, x*x*x]

console.log values
```

运行这段代码，可以得到所有在1至100之间，且立方值在1000至10000之间的奇数。

```
$ coffee comprehensions.coffee
[ [ 11, 1331 ]
, [ 13, 2197 ]
, [ 15, 3375 ]
, [ 17, 4913 ]
, [ 19, 6859 ]
, [ 21, 9261 ]
]
```

CoffeeScript的推导不仅可以结合list和range完成很多与数据容器相关的操作，也可以与object和dictionary很好地配合。

MostInterestingLanguages/coffee_script/classes.coffee

```
class Component
    constructor: (@name) ->

    print: ->
        console.log "component #{@name}"

class Label extends Component
    constructor: (@title) ->
        super "Label: #{@title}"
    print: ->
        console.log @title

class Composite extends Component
    constructor: (@objects...) ->
        super "composite"
    print: ->
        console.log "["
        object.print() for object in @objects
        console.log "]"

l1 = new Label "hello"
l2 = new Label "goodbye"
l3 = new Label "42"

new Composite(l1, l3, l2).print()
```

从上面这最后一个例子可以看出，如果我们的问题更适合用传统的面向对象的结构表达，CoffeeScript也是可以办到的。如果熟悉Java或者Ruby的规律，那么使用CoffeeScript中的构造函数和super就会非常顺手。这段程序的输出结果为：

```
$ coffee classes.coffee
[
hello
42
goodbye
]
```

如果你以前用过但不喜欢JavaScript，CoffeeScript则是个不错的代替品。它可以同时应用于服务器端和客户端。Rails现在已经绑定了CoffeeScript。你不应该错过！

学习资源

学习CoffeeScript的最佳起点是http://coffeescript.org。该站点有一份语言特性汇总，内容清晰明确。此外，这个站点还包含了一个可交互的控制台，可以在上面键入CoffeeScript代码，然后马上可以生成翻译好的JavaScript代码。

Trevor Burnham的著作《深入浅出CoffeeScript》（*CoffeeScript* [Bur11]）也是不错的资源。

如果你喜欢通过示例学习编程语言，CoffeeScript的主页上还有一些注释过的源码，供人了解其内部实现。这会让阅读和理解代码程序更加容易。

Erlang

Erlang是本文中最古老的语言，约诞生于20世纪80年代。不过，直到最近人们才开始真正关注它。

Erlang最初由Joe Armstrong开发，当时是为了编写可容错

的程序。Erlang主要应用的领域是长距离电话交换机以及其他相关领域。这些领域的最关键因素在于系统的正常运行时间。Erlang的大部分需求都来自于对代码的要求：健壮、容错、可以在运行时期更换。

今天，Erlang正越来越多地应用于其他领域，其原因在于它底层的Actor模型非常适合创建健壮的、可扩展的服务。

Erlang是函数式语言。函数是“一等公民”，可以在需要的时候创建，可作为参数传递，还可作为其他函数的返回值。Erlang只允许给一个变量名赋值一次——因此实现了不可变性。

Erlang的核心模型是Actor模型。其思想是我们可以拥有大量轻量的进程（称为Actor），它们可以通过发送消息来彼此通信。所以在Erlang中，要利用Actor来对行为建模或修改状态。如果你已经在其他语言中使用过进程和线程，一定要记住Erlang的进程是非常不同的：它们是轻量的，可以快速地创建，而且可以按照需要分布到不同的物理机器上。这样我们写出的同一份代码，既可以在一台机器上运行，也可以在上百台机器上运行。

与Erlang紧密相连的是开放电信平台（Open Telecom Platform，OTP），它是一些库的集合，可以用来创建非常健壮的服务。该平台给程序员提供了一个框架，程序员可以通过钩子使用一些高级的模式，创建可靠的Erlang服务——比如让特定Actor监控其他Actor的健康状况、Actor在运行期间可以实现代码的热替换等其他一些高级特性。

和前面看到过的语言不同，Erlang无法从脚本的顶层直接运行，因此在例子中，我们会从Erlang控制台中执行代码。但这样会

产生的一个副作用，即使最简单的程序也会比较长，因为我们必须将它作为一个Erlang模块导出。

MostInterestingLanguages/erlang/hello.erl

```
-module(hello).
-export([hello/1]).

hello(Name) ->
    io:format("Hello ~s~n", [Name]).
```

前两行代码是导出模块信息的指令。我们定义了一个hello()函数。变量名必须以大写字母开头，比如Name就是参数变量。format()是io模块中的函数。Erlang可以非常灵活地进行格式化。在这段程序中，我们只是把名字插入到字符串，然后打印出来。

在Erlang shell中执行这段代码，可以看到：

```
1> c(hello).
{ok,hello}
2> hello:hello("Ola").
Hello Ola
ok
3> hello:hello("Stella").
Hello Stella
ok
```

每个Erlang语句都以句点(.)结束，告诉解释器输入已完成。我们必须先编译模块，然后才能使用它。c()函数用来完成模块的编译。然后，我们可以调用模块。值ok是我们创建的函数的返回值。

虽然Erlang是函数式语言，不过它真正的强项是支持模式匹

配和递归算法。在看下一个例子之前，首先要知道，Erlang中的小写字母开头的标识符是符号，花括号括起来的是元组（tuple），方括号括起来的是列表。在Erlang中，这三者的组合可以表现不同类型的事物，通常是模式匹配的对象。

MostInterestingLanguages/erlang/patterns.erl

```
-module(patterns).
-export([run/0]).

run() ->
    io:format("- ~s~n", [pattern_in_func("something")]),
    io:format("- ~w~n", [pattern_in_func({foo, 43})]),
    io:format("- ~w~n", [pattern_in_func({foo, 42})]),
    io:format("- ~s~n", [pattern_in_func([])]),
    io:format("- ~s~n", [pattern_in_func(["foo"])]),
      io:format("- ~s~n", [pattern_in_func(["foo",
"bar"])]),
    io:format("- ~w~n", [pattern_in_case()]),
    io:format("- ~w~n", [reverse([1,2,3])]),
    io:format("- ~w~n", [reverse([])])
    .

pattern_in_func({foo, 43}) ->
    23;
pattern_in_func({foo, Value}) ->
    Value + 10;
pattern_in_func([]) ->
    "Empty list";
pattern_in_func([H|[]]) ->
    "List with one element";
pattern_in_func(X) ->
    "Something else".

pattern_in_case() ->
    case {42, [55, 60]} of
        {55, [42 | Rest]} -> {rest, Rest};
        {42, [55 | Rest]} -> {something, Rest}
```

```
        end.
reverse(L) ->
    reverse(L, []).

reverse([], Accum) ->
    Accum;

reverse([H|T], Accum) ->
    reverse(T, [H] ++ Accum).
```

这段代码首先创建了一个run()方法，它会执行定义在模块中的其他方法。Erlang的模式匹配用于三种场景，第一种是函数参数，第二种是case语句，第三种是用于处理消息传递。这段代码只示范了前两种情况。此外，代码还示范了如何使用Erlang的模式匹配机制，轻松实现尾递归算法。

```
18> c(patterns).
{ok,patterns}
19> patterns:run().
- Something else
- 23
- 52
- Empty list
- List with one element
- Something else
- {something,[60]}
- [3,2,1]
- []
ok
```

我们在列表中使用了管道(|)语法，把列表的头元素和剩余的元素分开。这种拆分数据集，然后分别对头、尾部分进行处理的模式是很多函数式语言中常见的模式。在reverse()函数的例子中，我只是简单地把头元素和尾部调换顺序，再重新组装在一起。

Erlang广为人知的一点是它支持Actor。在下一个例子中，我们会看到一个Actor，它将包含几种状态。某种程度上说，Actor和一个能永远保持内部一致性的同步内存区域相似。这里需要理解的主要语法是叹号(!)，它用于向Actor发送消息。我们可以把任何可序列化的Erlang表达式发给Actor，甚至发送一个函数。receive关键字此时的用处更像一个case语句，不同之处在于它会等待发送给当前正在运行的Actor的消息。

MostInterestingLanguages/erlang/actor.erl

```
-module(actor).
-export([run/0]).

run() ->
    State1 = spawn(fun() -> state(42) end),
    State2 = spawn(fun() -> state(2000) end),
    io:format("State1 ~w~n", [get_from(State1)]),
    io:format("State2 ~w~n", [get_from(State2)]),

    State1 ! {inc}, State1 ! {inc},
    State2 ! {inc}, State2 ! {inc}, State2 ! {inc},

    io:format("State1 ~w~n", [get_from(State1)]),
    io:format("State2 ~w~n", [get_from(State2)]),

    State1 ! {update, fun(Value) -> Value * 100 end},

    io:format("State1 ~w~n", [get_from(State1)]),
    io:format("State2 ~w~n", [get_from(State2)])
    .

get_from(State) ->
    State ! {self(), get},
    receive
        Value ->
            Value
    end.
```

```
state(Value) ->
    receive
        {From, get} ->
            From ! Value,
            state(Value);
        {inc} ->
            state(Value + 1);
        {From, cas, OldValue, NewValue} ->
            case Value of
                {OldValue} ->
                    From ! {set, NewValue},
                    state(NewValue);
                _ ->
                    From ! {notset, Value},
                    state(Value)
            end;
        {update, Func} ->
            state(Func(Value))
        end.
```

这段代码中定义了三个不同的函数。第一个用来运行整个示例。它调用了两次spawn，创建了两个不同的状态Actor。简单地说，一个Actor就是一个运行中的函数，所以代码中使用fun关键字来创建匿名函数，并且分别设置初始值42和2000。获得初始值后，首先打印它们，接着将第一个状态自增2次，第二个状态自增3次，然后再次打印它们，最后再向Actor发送一个函数，该函数可以用原始值乘以100后得出一个新值。结束前，再一次打印所有的值。第二个函数是get_from()，它是一个助手函数，可以简化从Actor获取值的过程。它会向Actor发送一个get消息(Actor来自于get_from的参数)，然后等待接收答案。

最后的函数是个真正的Actor。它会等待消息，然后根据收到的不同消息作出不同的响应。state执行结束后要递归地调用自身，以此保持状态。

```
32> c(actor).
{ok,actor}
33> actor:run().
State1 42
State2 2000
State1 44
State2 2003
State1 4400
State2 2003
ok
```

如果你可能需要花些时间才能完全理解最后一个例子，也不用太着急。因为Erlang处理状态的方式和大多数编程语言差别很大。总的来说，Erlang提供了非常强大的原语来处理并发，而且，巧妙地组合运用Actor可以写出非常漂亮的算法。

学习资源

学习Erlang的最佳起点就是Joe Armstrong的《Erlang程序设计》(*Programming Erlang* [Arm07])。该书对于Erlang的各个方面都有完整的介绍，对其中较为复杂的部分均有涉猎。此外，还有一本不错的书是Francesco Cesarini和Simon Thompson所著的*Erlang Programming* [CT09]。

你也可以网上获得一些不错的资源，比如http://learnyousomeerlang.com。

Haskell

就本文中的所有函数式语言来说，Haskell绝对可以说是把函数式范式发挥到最为极致的语言。Haskell是一门纯函数式编程语言，也就是说不支持任何形式的可变性或副作用。当然，这条真理还是有例外的，如果Haskell不支持任何副作用，你就无法执行打印操作，也无法从用户那里获得输入。Haskell可以执行I/O，也可以实现一些看上去像副作用的操作。但是在语言的模型中，其实并没有副作用出现。

Haskell是一门惰性语言，这一特点使它本质上有别于其他语言。这也就是说，函数的参数要到真正需要的时候才会计算。这一特性简化了很多工作，比如创建无限的流、递归函数定义以及很多其他有用的事情。由于没有副作用，因此除非刻意使用这一特性，否则我们通常不会注意到Haskell是一门惰性语言。

自ML以来，函数式编程语言开始分为两大不同家族——一族使用静态类型，另一族则不使用。有些函数式语言具有很先进的静态类型系统，Haskell便是其中之一。在某些语言的类型系统中很难表达的东西，Haskell的类型系统却可以很容易地表达出来。然而，虽然类型系统非常强大，但它并不会在写程序时造成太大的干扰。大多数情况下，我们不必为函数或者变量名指明类型；因为Haskell自身可以通过类型推演找到正确的类型。

Haskell没有基于继承的类型系统，但它大量使用了范型。Haskell的类型系统中的很大一部分都是**类型类**（type class）。这些类让我们可以向已有的类型中添加不同的行为。我们也可以把类型类看做一个接口，只不过在它定义好之后，包含了一

些添加到类中的实现。这是Haskell非常强大的特性，一旦用过这种类型类，在使用其他语言时就会非常想念它。

总的来说，Haskell是一门非常强大的语言。许多领域的研究者都能用Haskell进行进一步开发，因此，很多有趣的程序库最初都是用Haskell实现的。举例来说，Haskell支持很多不同的并发范式，其中包括软事务存储（STM）以及嵌套数据并行。

用Haskell编写“Hello World”程序，作为起始的例子可能有些怪异，这是因为在Haskell中进行I/O操作多少还是有点儿复杂。不过没关系，先来看一下程序代码：

MostInterestingLanguages/haskell/hello.hs

```
module Main where

main = do
  hello "Ola"
  hello "Stella"

hello name = putStrLn ("Hello " ++ name)
```

若想将这段程序做为单独的文件运行，必须在名为Main的模块中定义一个main()函数。do关键字让我们可以一次完成好几件事。最后，我们定义了一个hello()函数，它接收一个参数，将参数与Hello拼接，然后打印它。

编译并运行代码，可以得到如下输出：

```
$ ghc -o hello hello.hs
$ ./hello
Hello Ola
Hello Stella
```

与Erlang一样，Haskell也非常适合模式匹配。我还没有提到这一点，不过空格在Haskell中非常关键，也就是说，Haskell依靠空格确定代码结构，这一点和CoffeeScript与Python很像。应用模式匹配时，程序看上去会相当整洁。下面的代码创建了一个数据类型表示形状，然后使用模式匹配计算不同形状的面积。在这段代码中，我们又见到了使用递归和模式匹配实现反转队列的例子。

MostInterestingLanguages/haskell/patterns.hs

```
module Main where

type Radius = Double
type Side = Double

data Shape =
           Point
         | Circle Radius
         | Rectangle Side Side
         | Square Side
area Point = 0
area (Circle r) = pi * r * r
area (Rectangle w h) = w * h
area (Square s) = s * s
rev [] = []
rev (x:xs) = rev xs ++ [x]

main = do
  print (area Point)
  print (area (Circle 10))
  print (area (Rectangle 20 343535))
  print (area (Square 20))
  print (rev [42, 55, 10, 20])
```

上述代码输出如下：

```
$ ghc -o patterns patterns.hs
$ ./patterns
0.0
314.1592653589793
6870700.0
400.0
[20,10,55,42]
```

在Haskell中，我们可以看到大多数函数定义看上去很像代数公式。定义像Shape这类数据类型时，需要列出所有可能性，还要列出每种可能性需要的数据。然后，当我们根据运行时的不同数据派发到不同的area()方法时，也要挑出数据类型中包含的数据。

前文提到过Haskell是惰性语言。证明这一点很容易，例如定义一些会无限执行的操作：

MostInterestingLanguages/haskell/lazy.hs

```
module Main where

from n = n : (from (n + 1))

main = do
  print (take 10 (from 20))
```

这段代码看上去相当简单。take()函数是在Haskell核心类库中定义的。take()会从给定的列表中取出指定数目的元素(本例中是10个)。函数from()使用冒号构建新的列表，该列表定义成n的值，它后面跟着一个列表，通过n+1再次调用from()产生。在大多数语言中，一旦调用这个函数就会陷入无限递归，程序也就崩溃了。但是Haskell只会有限次地调用from()，直到取得足够用的值为止。这一点非常难懂，需要花些时间才能理解。不过请记住，这里的冒号没有什么特殊之处，它不过是Haskell的求值方式而已。

运行代码后的结果如下：

```
$ ghc -o lazy lazy.hs
$ ./lazy
[20,21,22,23,24,25,26,27,28,29]
```

关于Haskell，我想展示的最后一件事是关于类型类的。由于Haskell不是面向对象的，也没有继承，因此有些事做起来非常笨重，比如定义范型函数、执行打印、检查相等性或者其他类似的工作。类型类可以解决这个问题，基本上，它允许我们根据不同的Haskell类型变换不同的实现。类型类异常强大，并且与之前所见的传统的面向对象语言截然不同。让我们看一个例子：

MostInterestingLanguages/haskell/type_classes.hs

```
module Main where

type Name = String

data Platypus =
    Platypus Name
data Bird =
    Pochard Name
  | RingedTeal Name
  | WoodDuck Name

class Duck d where
  quack :: d -> IO ()
  walk :: d -> IO ()

instance Duck Platypus where
     quack (Platypus name) = putStrLn ("QUACK from Mr
Platypus " ++ name)
    walk (Platypus _) = putStrLn "*platypus waddle*"
instance Duck Bird where
```

```
     quack (Pochard name) = putStrLn ("(quack) says " ++
name)
    quack (RingedTeal name) = putStrLn ("QUACK!! says the
Ringed Teal " ++ name)
    quack (WoodDuck _) = putStrLn "silence... "
    walk _ = putStrLn "*WADDLE*"

main = do
  quack (Platypus "Arnold")
  walk (Platypus "Arnold")
  quack (Pochard "Donald")
  walk (Pochard "Donald")
  quack (WoodDuck "Pelle")
  walk (WoodDuck "Pelle")
```

这段代码包含了很多信息。首先，我们定义了两种数据类型：一种是Bird（鸟类），一种是Platypus（鸭嘴兽），它们都有一个名字(Name)。接下来我们创建一个类型类，名为Duck（鸭子）。我们知道如果某种动物叫声像鸭子，走路姿势也像鸭子，那么它就是鸭子。因此，类型类Duck定义了两个函数，quack()和walk()。这些声明只是规定了参数的类型和返回值的类型。这些类型签名表明，它们接受一个像鸭子的东西，然后打印一些输出。随后，我们定义了类型类Platpus的一个实例。我们只在实例中定义必要的函数，这些函数和Haskell的顶层函数并无区别。之后，我们对Bird做了同样的事情。最终，我们实际上是在不同的数据实例上调用了quack()和walk()。

运行此示例，可以看到和预期相符的输出：

```
$ ghc -o type_classes type_classes.hs
$ ./type_classes
QUACK from Mr Platypus Arnold
*platypus waddle*
(quack) says Donald
*WADDLE*
```

```
silence...
*WADDLE*
```

类型类相当强大，但是在这样一小段代码中只是管中窥豹。不过请放心，一旦彻底理解了类型类，你就迈入了精通Haskell的大门。

学习资源

学习Haskell的最佳起点是一本在线图书，名为*Learn You a Haskell for Great Good* http://learnyouahaskell.com。该书以简单生动的方式，带你循序渐进地学习Haskell。

除此以外还有一些涉及Haskell的书，不过却大同小异，大部分都注重从数学或者计算机科学的角度讲述如何使用Haskell。如果你想学习如何将Haskell用做一种通用目的编程语言，最好读一读Bryan O'Sullivan、Don Stewart和John Goerzen合著的Real World Haskell [OGS08] 。这本书也可以在线访问http://book.realworldhaskell.org/read。

Io

在本文介绍的所有语言中，我认为Io绝对是我的最爱。它虽然很小，但是很强大。虽然它的核心模型简单且中规中矩，但却诞生出很多新奇、精彩的特性。

Io是纯面向对象语言，“纯”，即Io中所有事物都是对象，没有例外。所有我们碰到的、用到的，或是实现用到的都是对象，我们可以到达它，持有它。相比于Java、C#，Smalltalk以及很多其他面向对象语言，Io并不使用类。Io使用基于原型的对

象系统取代类，其思想是根据已有的对象创建新对象。我们可以直接修改某个对象，然后将其当做新对象的基础。

传统的面向对象语言有两个不同的概念：类和对象。在更纯粹的面向对象语言中，类也算一种对象。不过这两者之间有着根本的差异，即类具有对象所没有的行为。在Io中，方法也是对象，就像其他东西一样。方法可以加到任何对象上。这种语言的编程模型与基于类的语言差别很大，因此，我们可以用截然不同的方式建模。基于原型的语言有一个优势，即它们能够很好地模拟基于类的语言。因此，如果想继续使用基于类的模型，Io也不会限制我们。

Io是一门小巧的语言，但却支持很多功能。它支持一些不错的基于协程（coroutine）的并发特性。利用Io的Actor，构建健壮的、可扩展的并发程序非常容易。

还有一个方面，Io也做到了极致：用于表示Io代码的元素都是一等对象。这意味着我们可以在运行时创建新代码，可以修改已有的代码，还可以自查已有的代码。利用这一特性可以创建出极其强大的元编程程序。

在Io中，定义方法就像普通的赋值一样——首先，创建一个方法，然后把它赋给某个名字。第一次赋予名字时，要使用:=，之后就可以直接用=。我们的“Hello, World”例子如下：

MostInterestingLanguages/io/hello.io

```
hello := method(n,
  ("Hello " .. n) println)

hello("Ola")
hello("Stella")
```

我们先用..操作符拼接字符串，然后要求字符串打印自身。输出应该很容易猜得到：

```
$ io hello.io
Hello Ola
Hello Stella
```

Io有一个使用Actor和Future的协同多任务机制。只要调用Actor的asyncSend()方法，并传入要调用的方法的名字，任何Io对象都可以用作Actor。我们必须显式地调用yield，以确保所有代码都已运行。

MostInterestingLanguages/io/actors.io

```
t1 := Object clone do(
  test := method(
    for(n, 1, 5,
      n print
      yield))
)

t2 := t1 clone
t1 asyncSend(test)
t2 asyncSend(test)

10 repeat(yield)
"" println

t3 := Object clone do(
  test := method(
    "called" println
    wait(1)
    "after" println
    42))
result := t3 futureSend(test)
"we want the result now" println
result println
```

上述代码首先创建了新对象t1，它有test()方法，能够打印数字1~5，每次打印之间调用yield。然后再克隆这个对象，依次调用这两个对象的asyncSend(test)()方法。最后在主线程中调用10次yield。

第二段代码又创建了一个新的对象，它也有一个test()方法，它会先打印一些东西，然后等1秒，再打印一些东西，最后返回一个值。我们调用这个对象的futureSend(test)()，就可以将它转变为一个透明的Future。这个调用的结果不会立即计算，而是等到真正需要其值的时候，也就是最后一行，此时我们要打印它的结果。该特性和Haskell处理惰性值的方式非常类似，不过在Io中，我们必须显式地创建Future才能获得同样的效果。

运行程序，得到如下输出：

```
$ io actors.io
1122334455
we want the result now
called
after
42
```

两个Actor彼此交替地打印输出，从中可以看出Actor协作的本质来。你可能也注意到了，通过Future调用的方法的输出并未立刻被打印出来，这说明它是到最后一刻才调用的。

Io另一个强大特性是它支持反射和元编程；基本上Io中的任何东西都可以被访问和修改。Io中的代码是可以在运行时访问的，它以一种消息的形式展现。我们可以利用此特性完成很多事情，比如创建高级的宏设施。尽管下面这个例子可能没有什么实际的用途，但它确实展示了这种方法的威力：

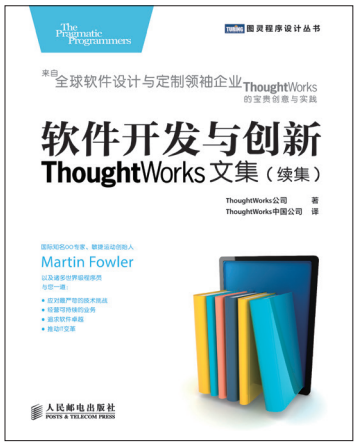
MostInterestingLanguages/io/meta.io

```
add := method(n,
  n + 10)

add(40) println

getSlot("add") println
getSlot("add") message println
getSlot("add") message next println
getSlot("add") message next setName("-")

add(40) println
```

首先，这段代码创建了add方法，可以给参数值加10。我们调用它，确保它是可用的。然后，我们使用getSlot()访问方法对象，但不真正对它求值。打印方法对象，然后得到它的message对象，之后再打印message对象。Message是对象链，因此，对一个message求值后，Io将根据next指针指向下一步操作。打印next指针的值，然后修改了next的这个message名字。最后，重新以40为参数调用add。基本上，这段代码就在运行时期动态地修改了add()方法的实现。

运行代码，可以看到相应的结果：

```
$ io meta.io
50

# meta.io:2
method(n,
    n + 10
)
n +(10)
+(10)
30
```



在软件开发中遇到困难时，如果得知前人也曾至此，便真是幸甚至哉。在《软件开发与创新：ThoughtWorks文集（续集）中》，ThoughtWorks的领域专家们分享自身所学，将他们在IT及软件开发领域中久经考验的最佳洞见结集成册。这些经验会让我们受益良多，从测试到信息可视化，从面向对象到函数式编程，从增量开发到在交付中持续创新，从改善敏捷方法学到顶尖的语言极客范儿。无论何时，当你需要专家建议时，都能从这些已成功解决的问题中汲取营养。

Io可塑性极强，几乎所有东西都是可访问的、可改变的。它是一门非常强大的语言，而它的强大之处却隐藏在小巧的外表之下。我第一次学习Io时就被它震撼到了，而且在之后的学习过程中，它也不断地改变着我的想法。

学习资源

目前还没有介绍Io的书。不过http://www.iolanguage.com/scm/io/docs/IoGuide.html上的入门教程是个不错的起点。看完这个教程后，你可以读一读参考文档，尝试理解Io的一些要点。同时，Io也非常易于理解，我们通常可以直接查看某个对象包含了什么功能。

Steve Dekorte做过一些关于Io的在线演讲和访谈。Bruce Tate的《七周七语言：理解多种编程范型》一书中也有关于Io的章节。

总结

如果你对编程语言感兴趣，那么这是属于你的时代。我已经展示了一些不同的语言，以及每种语言中有趣的特性。从现在开始，就轮到你了。找寻其他有趣的语言，看看用这些语言能做些什么。学习新语言的成效在于，它会改变我们使用主流编程语言的方式。如果尝试一种编程范式完全不同的语言，那么它对你的影响将尤为明显。

我还有很多有趣的语言想在这一章里来讨论。但是，这会让这一章变成一本书。这些语言（有新有旧）包括（没有特定顺序）：Frink、Ruby、Scala、Mirah、F#、Prolog、Go以及Self。

我写本文时正值新年前夕。在这一天有一个传统，就是要为来年做些打算。对于程序员来说，选择并学习一门新语言是个传统项目——我是从《程序员修炼之道》这本书开始的，建议你也这么做。这会让你成为一名更优秀的程序员。

九　卦

开火前进
——识破微软的龌龊伎俩

作者/ Joel Spolsky

Avram Joel Spolsky生于1965年，他是一位软件工程师和作家。他是“Joel on Software”博客的作者。他从1991年到1994年间担任Microsoft Excel团队的项目经理。在2000年，他创立了Fog Creek软件并开启了“Joel on Software”博客。2008年，他和Jeff Atwood一起启动了如今极为成功的Stack Overflow

有些时候，我会有一种力不从心的感觉。

进入办公室后，我当然可以闲逛，每隔一会查收下邮件，读读网页，然后做一些像是支付美国运通账单的蒜皮小事。但是想要敲打键盘写代码却比登天还难。

通常，这种没有工作效率的情况会持续一到两天。但是在我的软件开发的生涯里，我遇到过多次几周内无法完成任何任务的情况。也许像他人说的，我不在状态，我的心思不在工作上，或者我的脑子被僵尸吃了。 人人都会有情绪波动的时候，只是有的人不明显，而有的人更明显些，甚至情绪有时还会失去控制。并且低效的工作时段似乎和情绪波动有所关联。

这一点让我想起了一项有趣的调查。此调查中指出，人们基本上无法控制自己的食欲，任何节食的试图必定是短期的行为，而且最后人们都慢慢的恢复至原来的体重。作为软件开发者，也许无法控制何时效率才会高。所以我必须接受工作有时高效而有时低效的现实，并且希望综合起来能够开发出使我满意的代码量。

令人沮丧的是，从事软件开发的第一份工作开始就意识到，我平均每天高效编程只能持续两至三个小时。当我还在微软暑期

程序员问答网站。他们用Stack Exchange软件产品作为Stack Overflow的引擎。现如今Stack Exchange网络已经包含了91个站点。

实习时，一个同期实习生告诉我，他每天只有12点到5点才能进入工作状态。也就5个小时，再减去吃饭的时间……但是开发组的人还是很喜欢他，因为他平均完成任务比别人还多得多。我发现这种情况确实如此。每当我发现别人看起来整天很忙而我一天只能高效的工作2到3个小时时，些许的内疚感便油然而生。但我仍然是我们组最多产的组员之一。这也许就是Peopleware（人件编程）和XP（极限编程）的项目组拒绝加班并每周严格坚持工作40小时的原因。他们有把握这么做不会减少团队的产出。

当然，使我感到焦急地不是每天只能工作两个小时，而是那些我无法做任何事情的日子。

对于这一点，我思考过很多。我试图回忆在我事业中什么时候是完成工作最多并且是最高效的。也许是Microsoft把我调到一个漂亮、豪华的办公室时。这间办公室有一张大大的落地窗，从这扇窗向外望去，可以看见漂亮的用石头堆砌的庭院，庭院中开满了樱桃花。每件事情都很顺利。我花了好几个月的时间不停止的琢磨Excel Basic详细的说明书。这份说明书可以说是具有里程碑意义的文档。为了将其中复杂的对象模型和编程环境介绍的细致全面，我马不停蹄的足足耗费了可以垒一座山的纸张。这种日子一直持续到我不得不去波士顿参加苹果开发者大会。大会期间，我还舒适地坐在哈佛商学院的露天阳台，用随身的笔记本编写窗口类。

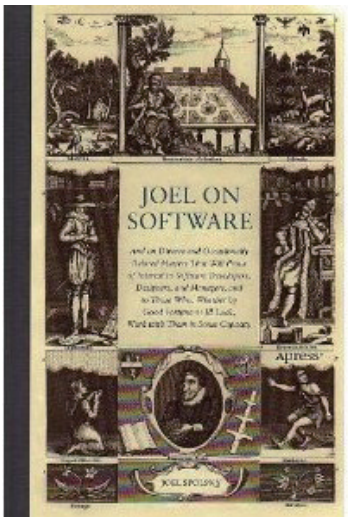
一旦你进入了状态，那么维持这种状态并不是很艰难。我很多天都是这样过来的：（1）进入办公室（2）查收邮件，看看网页等（3）决定不如在开始工作前吃午饭（4）午饭归来（5）查收邮件，看看网页等（6）最终决定我不得不开始工作了（7）查收邮件，看看网页等（8）再次地决定我不得不开始工作了（9）启动该死的编辑器（10）不停的编码直到我意识到已经下午7:30

第八步和第九步有时会出现一点小状况，因为我不能总是完成其间跳转。对于我来说，只有起步时是艰难的。因为静止的物体总是倾向于静止（也就是物理课上的惯性，译者注）。我的大脑里形成了深刻的意识，那就是一开始时加速很艰难，但是一旦全速运行，维持这个状态便会不费吹灰之力。就像骑车参加自助式越野环行--当你刚踏上自行车转动齿轮时，很难想象要用多大的力气才能使它们运转起来，但一旦运转起来，自行车便像没有齿轮一样易于驾驶。

也许这就是多产高效的金钥匙：只管开始就行！结对编程时也许更有效，因为当你和同伴安排结对编程会话时，可以敦促彼此开始编程。

当我还是以色列伞兵时，一位将军来给我们作了一次关于作战策略的简短训话。他告诉我们，在激烈的陆战战场上，只有一个策略：开枪冲锋。边开火，边向你的敌人移动。火力会迫使他低头，以便不被子弹击中。（这就是士兵大喊“掩护我”的意图：朝我们的敌人开火，以让他不能在我穿过街道时向我射击。这种策略很实用）。冲锋可以夺取敌人的阵地，还可以靠近你的敌人提高子弹命中的概率。如果你不移动位置，那么敌人就控制战局使你陷入被动。如果你不开火，那么敌人就会朝你射击阻止你前进。

我一直记着这一点，而且我发现各种各样的军事战略，从空军混战到大型的海军演习，都是基于射击前进。大约又经过了15年，我意识到射击前进同样适用于日常生活。即使代码烂到别人嗤之以鼻的程度，你每天也必须前进一点点。只要坚持不停的写代码和持续的修改错误，最终时间会站在你这边。当竞争对手朝你发飙时，你要格外的留心。是不是他们只是让你疲于应付而无法前进。



《软件随想录，卷1：乔尔博客文集》是一本介绍软件管理的小品文集。全书分为45章，内容十分丰富全面，小到项目负责人制订进度表，大到软件执行总裁提出富有竞争性的战略，都在本书的介绍之列。

回顾一下微软推出的数据存储策略。如ODBC，RDO，DAO，ADO，OLEDB和最近的ADO.NET，它们无疑采用了全新的策略。这些技术都是必须的吗？一个平庸的设计团队才会每年都费力不讨好的使用新的数据存储策略（也许他们真的很平庸）。但是结果是这些只是掩护火力。这样，竞争对手为了不被技术淘汰，不得不花费所有的时间来移植。但同时，也失去 了开发新特性的经历。仔细研究一下软件开发领域，那些运行良好的公司都很少依赖大型公司，而且不盲目花费所有时间跟风新技术，也不仅仅的修改只在Windows XP上出的的bug。那些陷入困境的公司都是花费太多时间在分析微软未来技术走向的企业。微软发现人们很关注.NET架构，于是决定重新实现整个.NET的架构，而这么做没有任何理由。注意！微软正在狙击你，重新实现架构只是些掩护火力，以便他们可以顺利前进而你不能。这就是游戏的规则！你的软件将会支持Hailstorm，SOAP和RDF吗？你的软件支持这些是因为用户真的需要吗，还是只是有人朝你开火你觉得不得不反击？大型公司的销售团队很懂得提供火力掩护。他们到客户那里，然后告诉客户：“好吧。你不是必须要从我们这里购买产品，你完全可以从最好的供应商那里买。但是你的产品一定要支持这些（XML / SOAP / CDE / J2EE），不然你就别想成功！”然后当一些小公司去推销同样的产品时，他们就会听到没有主见的CTO的鹦鹉学舌“你们支持J2EE吗？”再然后，即时还没有卖出一份产品也无法区分它们的区别，小公司不得不浪费所有的时间去支持J2EE。这是一项复选功能，添加它只是标示你有这个功能，而没有是使用它，甚至没有人需要他。可以看出，这只是火力掩护。

开火前进对于像我这种小型公司来说，有两种意义。必须有充分的时间，每天必须前进。你最终会取得胜利。昨天一整天，我全力去做的只是提升FogBUGZ中颜色方案一点点性能。我

译者/王辉

在美丽的海滨城市青岛工作，现从事android的framework和底层的移植工作。做事三分热度，就连打了5年的dota还在1200左右混。喜欢大河剧《风林山火》，图灵社区ID：风林火山

们的软件一天更比一天好，我们的客户也渐渐增多。这才是至关重要的！直到我们公司到了Oracle的规模，我们就可以不必去想使用什么宏伟的战略了。我们只是每天清晨进入办公室，在自己喜欢的时候，启动编辑器。

英文原文：Fire And Motion

我在中兴软创这9年

作者/徐飞

徐飞，05年至今就职于中兴软创，致力于企业软件前端工程化的发展。徐飞坚信：作为技术人员应当乐于助人，有好东西要主动拿出来分享，资产阶级知识分子垄断电子书的现象再也不能出现了！微博@民工精髓V，图灵社区ID:民工精髓。

一个月前离开呆了9年的中兴软创，有不少东西值得写下来，千头万绪，不知从何写起，自己留下了10多万字的回忆，但这里面涉及的东西太多，不便公开，还是把这些年对工作的感悟写一下吧，这9年的工作基本都是围绕前端框架的。

中兴软创的主要业务称为BOSS，也就是电信行业的运营支撑软件。早期的BOSS系统一般都不是Web化的，而是C/S架构，当时大家做所谓的“前端”，用的是Delphi，C++ Builder，或者Java Swing。后来B/S流行之后，大家就逐渐往浏览器上迁移。

那个时期的浏览器，不像现在这么多样化，一般指的都是IE，而且可以具体到三个版本，5.0，5.5，6.0。第一批迁移到B/S模式的系统，多半是那些简单表单的系统，界面只是填值，作个简单校验，然后提交给服务器。可以说，这个时候的Web前端是很乏味的，因为没什么可做的，用table布局，里面放些form，极少量的JavaScript代码，更谈不上用CSS。

不久，Web系统就复杂化了，在C/S里面，我们可能有大量的“控件”可用，基本的输入框这些不谈，在HTML里也有，时间日期这类，就要费些周折了，更复杂的，比如Tree，

DataGrid，甚至TreeGrid，就更折腾。当时写JavaScript的人还是有不少的，面对这种情况，也想出了一些办法。

这些办法的根本原理，都是用已有的HTML来拼凑出一个控件的样子，再加上事件，一直到现在也没有更好的办法。比如说，日历，就用table标签来生成一个表格，然后当前日期加个颜色。又比如DataGrid，也是一个表格，然后tr上面加点击事件。有的流派是跟现在一样，把控件的声明和初始化都放在JavaScript代码中，只在HTML里留一个容器标识，更主流的方式是用HTC来封装控件。

如果仔细看过早期ASP.net的代码，就会发现它带了几个htc文件，比如treelist，tabstrip，multiview等等，这些控件的功能已经基本能满足需要了，就是有些丑，有些人在此基础上作美化，04年的时候，中兴软创的多数基础控件就是这么来的。HTC提供了一种扩展HTML标签的机制，业务开发人员用起来很方便，所以很有效地降低了开发门槛。

再看另外一个方面，传输的问题。最开始大家都是把数据用submit按钮提交给服务端的，但是提交就会刷新整页，效果不好。我们知道，AJAX的概念是05年提出的，但是在此之前好几年，就有不少用XMLHTTP的人了，我自己入职之前，03年的时候，就用过这个，入职之后看公司的前端框架，也一眼发现了这些东西。中兴软创的这套传输机制是在微软顾问的帮助下创建的，传输的原理就是在前端把表单数据序列化成XML，通过XMLHTTP传给后端的Servlet，当然，那时候用的是同步传输，传的时候界面会卡一会。

05年我刚入职的时候，还没看过系统，老大问我对前端这块有什么看法，我说可以考虑做组件化，把更多的东西封装成HTC

那样的组件，然后组件内部通过XMLHTTP跟服务端通信，他听了之后说我们现在已经有一些HTC控件，通信也基本都是XMLHTTP了。回想起来，当时我的思路是纵向的组件，端到端，每个组件实际上只通过事件和方法与其他组件交互，各组件自身就可以独立运行，应该算是早期前端组件化的一种思路。

为了通用性，前端封装了一个方法叫callRemoteFunction，三个参数，分别是后端的Java类名，方法名和参数对象，用XMLHTTP发送到后端的Servlet之后，通过前两者反射得到对应的Java方法，执行结果再返回给前端。这样，在JavaScript里“调用”后端代码，就像调用普通的JS函数那么方便。也有这样调用动态SQL的，后来这两者统一成服务，只要传入唯一的服务名和参数，不用管是Java服务还是SQL服务。

有了这些东西做保障，业务系统的B/S化就容易多了。当时的开发模式是前后端分离，后端负责写服务，前端写界面和JavaScript，这种模式也带来很多好处，比如有的业务系统后端从.net迁移到Java，前端部分基本除了登录之类，都没什么要改动的，人员的协作也是很顺畅的。

在迁移系统的过程中，也有其他一些混杂技术，比如说，处理一些监控图形之类的，由于缺乏经验，加之为了重用之前的Swing代码，搞了一些Applet，虽然混搭的风格不太好看，但当时是没什么人讲究这个的，业务系统能用就行了。因为我入职之前搞过VML，所以极力鼓动把各种图形的东西搞成VML，这个东西在当时最大的优点是不需要给浏览器安装插件，其他任何方式都做不到。

后来就有了IOM系统那个很典型的自动布局流程建模界面，核心部分有3k多行JS，花了近2个月，期间还重构过一次，后来陆续改需求，到06年下半年才不太改动了。从此之后，公司的Web图形这块，基本都是用VML，不再有人提Applet的事了，而且几年内也没有用Flash做这类图形的，据我所知，业界当时用Flash做图形界面比用VML的还多些。

到了07年，Firefox就占不小的市场比例了，而且HTC这个东西，微软自己也不太看好，所以不得不未雨绸缪，考虑这些东西的替代方案。正好当时调动部门，新部门打算彻底翻新产品，所以有机会考虑前端的新方案。作为前端的整合框架，有两条道路可走，一条就是选择别人的方案，比如早一点的Bindows，还有当时比较火的ExtJS，另一条就是先引入一个JavaScript基础库，然后在上面自己做控件。经过慎重考虑，还是选了后者，因为我们的业务需求比较复杂，改控件的情况很多，要是用了ExtJS这类，虽然看起来什么都有，但是改东西估计就痛苦了。

接下来就是选基础库了，流行的有Prototype，Mootools，jQuery，甚至还有万常华的JSVM，在那个时候其实很难预料到后面jQuery这么火，就算到现在我也不能理解，所以我们的选择是Prototype，然后在它基础上构建外围库，主要是控件。

当时看过很多UI库的机制，比较来比较去，觉得最能接受的还是YUI的方式，所以大致按照这种方式做下去了。我们的控件体系是比较松散的，彼此之间无任何依赖关系，可以独立引用，控件的唯一参数就是父容器，然后传入初始化参数，加载数据之类。

这一代控件的DataGrid和TreeGrid是我做的，跟上一代最大的区别是简化了事件。比如说，之前的控件选中行用的是点击，但是键盘的方向键也可以改变选中行啊，这时候业务方需要监听控件的两种事件，在每种里面都做选中行变更的操作。这一代里面我只给业务方开放change事件，不管实际是从什么事件发起的，最终需要关注的只是这个change，在控件上，行的点击事件这种过于原始的事件是没有意义的，直接抛给业务方非常不合适。刚开始改成change的时候，有些业务开发人员不太习惯，不过很快就觉得这样方便了。

这一代的TreeGrid控件我作了懒加载，但实现的细节上有些考虑不周了，比如说，下层节点在未展开的时候，DOM不创建，这没有问题，但是我连节点对象都没创建，当业务方要访问未展开的节点数据时，只能从数据源上去获取，已展开的节点和未展开节点的访问方式不同，这算是一个败笔。

整体来说，这一代的框架运作还是很成功的，比较稳定，但整个版本关键的一点没有达到，就是跨浏览器，也就是说，即使把控件代码改成纯JS的，也没让整个版本跨浏览器，这很悲剧。一个关键问题是版本时间太紧，框架层从无到有三个月之后，业务侧就大量启动开发，有不少问题没有来得及解决，更本质的问题在于当时我们缺乏经验，没有对业务开发人员作约束，比如说，有些要避免的写法没有列出，对于跨浏览器怎样测试，也没有时间作考虑。等到打算解决这些问题的时候，面对海量的业务代码，已经无从下手了。

这个版本中，也遇到一些比较新的需求，比如说有的监控需求，要实时通信，那时候没有WebSocket可用，就用Flash的Socket，搞了一个不显示的Flash，专门用来连Socket，

然后再用JS跟它交互，效果还可以，只是因为Flash的跨域策略升级过几次，导致踩了一些坑。

说到这个Flash，又扯到另外一些话题，早期搞前端的人，多数都玩过它。Flash内置一些控件，比如基本表单输入，还有调用WSDL格式WebService的通信控件，整个体系其实成熟度不比HTML低，只是我一直对时间轴很痛恨，所以即使搞，也都倾向直接用AS写，很少用元件转MovieClip那些东西。后来2004年推出的Flex1.0，彻底不一样了，我研究过一阵，也想过如果在企业应用领域，全部用它来构建前端如何?

这个想法是有些激进，但对于企业应用而言并不过分，企业应用连Applet都能接受，机器上要装10多M的JRE，那用1M多的FlashPlayer不是更好嘛，而且当时很多开发人员写不好JS，尤其是代码规模较大的时候，但他们写Java都还凑合，如果用AS来写，代码效果应该好不少。

当时的Flex是要部署到应用服务器里的，运行机制大致就像JSP那样，文本代码经过一个预编译，然后发到浏览器端来执行。当时制约Flex发展的主要因素是客户端机器的配置，Flash体系的界面效果较好，但比较占资源，而且在开发阶段的优势也体现不出来。

但我一直认为，Flex体系在较大一个时间段中很适合企业应用体系，因为浏览器混战的时期很长，乱象环生，老的浏览器迟迟不去，多少年也抹不平兼容的坎。对企业应用而言，搞跨浏览器兼容这方面并非它的核心价值，如果有一种技术能暂时抹平这些浏览器的差异，优势会是很明显的。要说占资源大，难道ExtJS占资源就小了？企业应用连ExtJS都可以接受，当然更能接受Flex。

所以从09年开始，又逐步进行Flex的引进，当时的Flex发展到了3.0，整体算是比较成熟了，后来陆续花了两年时间支撑业务产品的开发，效果还可以，但从引入时机来说，还是略有些晚，如果再早两年引入，状况会更好一些。

另外一方面，BOSS领域的应用系统并不局限于企业应用类，也有一些是面向个人用户的，比如说自服务和网上商城，前者类似10086.cn那种模式，个人消费者可以登录办理一些简单业务，后者就是典型的网店，只是所卖的限于电信类的实体商品（手机、上网卡等）或者虚拟商品（套餐，流量）等。

这个场景跟之前的内网应用大有不同，算是真正的互联网模式了，所以它所用的前端框架就与其他的不同。由于精力所限，开始几年在这方面的投入很少，一般都是用jQuery外加一些开源的控件，这样整合起来用，页面不花哨也不复杂，基本功能也是能够满足的，做的效果只能算是凑合，主要是没有熟悉CSS的人。

在做电信业务运营支撑的这类公司，UI一直是薄弱环节，不可能得到本质上的重视。整个中兴的整个体系里，软件的重视程度并不如硬件，比如从手机上面就看得出来，卖了手机之后就不太重视后续软件升级了，还是卖老的功能机的思路。在软件体系里面，前端也处于相对弱势的地位，毕业生入职的时候，都会优先让编程水平较高的做后端，在前端里面，逻辑和业务的重视度又高于UI，所以UI保持能用就不错了，在关键的一些跨浏览器兼容，CSS规划方面，基本是没有什么进展的。好在近两年，由于有了BootStrap这样的东西，把很多原本要做的事情做掉了，所以只要对界面没有特别的需求，光会写JS也能把界面搞得像模像样。

这部分的前端框架，其实也不是这么搞就完事的，基于传统的思维，做这些界面的时候，开发人员仍然倾向于使用偏重量级的控件，而不是使用界面模板库等方式来做一些数据展示的效果，这一方面带来的是观感的不佳，另一方面，由于引用的一些控件库没有很精细地隔离，往往都是整套控件一起引入，甚至在一个界面里还出现同时引用多种界面库的恶劣情形，一个并不算复杂的界面，引用的压缩之后的文本代码就高达1-2M之多。

所以从这个方面讲，公司的多数前端人员并不专业，专业与不专业体现在什么地方？是要有一个整体的优化。前端与后端开发方式的一个本质差异是引入任何东西的代价都比较大，因为你的代码要先经过一次网络传输才能执行到，而且还要注意避免冲突。如果只要引用某个功能，就不应把其他不相关的东西也一起引入，所以那种一个大控件库整体打包的方式在这种面向互联网终端用户的模式下非常不合适。这个道理并不难理解，但为什么操作的时候很少有人注意避免呢？

因为两个原因：

- 精确控制的代价较大。这一点确实是个大问题，要做精确控制，最小依赖，需要把整个框架的依赖关系理清楚，在现有的开发体制下，谁为这个时间买单？既然没有，那基本上就没人管了。
- 加载的字节量未作为系统上线的考核指标。从反面说，如果这么做了，功能倒是能用，但系统加载慢了，有多慢，这个没有预设的性能底线，一般赶时间做的系统也都不会太纠结在这上面，能用了按时上线了就大家都谢天谢地。

从决策层的观念上，也有一个误区，比如认为自服务类系统不算核心系统，对开发技能的要求也不会多高，凑合能用就行

了，事实并非如此！企业应用型的系统，才是不特别考验开发技能的，考验的更多是架构水平，它在前端的坑并不多，所以完全可以由个别架构水平高的带着一群偏弱点的开发人员做，而网站类的对每个开发人员的前端技能水准要求都更高，如果不改变以往的思维方式，后续这类系统会经常收到投诉。

近两年，因为要考虑未来老旧浏览器淘汰之后的事情，所以我花了不少时间研究了一些懒加载框架，还有一些前端MV*框架，尤其在AngularJS上，花了很多精力，比如12年的时候打印了源码来看，也做了各种尝试。这些东西用在企业应用领域，是极好的。第一次看到AngularJS，是因为当时在寻找通过HTML属性实现数据绑定机制的方案，然后就看源码，看同类方案，一发不可收拾。

后来的规划，是用它来实现核心逻辑，而外围的directive层分为PC浏览器和移动终端两类，这样可以实现逻辑的共享。到了该考虑移动端的时候，又碰到了Ionic，真是想什么来什么，也说明我的这些路不孤单，还有一些人用同样的思路在走。

之前公司也搞过移动端的系统，用了响应式设计，也碰到一些坑，从我的角度看，公司用响应式设计还是要慎重，因为完全没有熟悉CSS的人，要用这个风险很大。

近两年考虑的另外一些事情是前端开发的工程化，这个路也不孤单，各大公司都或多或少的在做，比如前端组件的管理，自动化测试，发布等等，典型的有百度FIS。当系统规模扩大的时候，在代码管理和发布问题就特别多，前几天看到winter的微博，应该也是踩到不少坑。。。所以说，架构师要考虑的事情，一方面是系统自身的架构，另一方面要考虑团队在协同开发时候可能遇到的问题，从技术角度尽可能及早把这些东西化解。

这方面花费的精力很可能比真正在产品里花的还多，而且是很痛苦的，做了很多之后还不容易看出作用。

去年在上海一家公司面试的时候，跟面试官聊得非常投缘，他问一句我答一句，有时候他话没说话我就接着说下句，我话没说完他就接着说下去，最后两个人相对大笑，那是发自内心的苦笑，前端架构这个大坑啊。他说，对吧，架构这事，比的就是你踩过多少坑，我们这一路上踩过来的坑，都是血和泪。我俩笑得像《投名状》电影里，刘德华最后流着泪笑得样子。

碰到的另外一个聊得很投缘的人就是支付宝的玉伯，可能因为业务场景比较接近，而且大家的努力方向都在前端的工程化方面，所以很多东西都是所见略同。

早些年，公司的前后端分离开发，效率很高，问题也少，不知为什么做着做着就成了不分的模式，开发人员从HTML，JS，Java一直写到SQL，什么都搞，什么都不专业，很可怕，我提了不知多少次意见，从未得到回应。虽然最近业界很多鼓吹全栈工程师的，但这只能让那些个人能力较强的去做，作为补充，不能成为普遍做法，对于招聘人员水准比不上互联网公司的传统软件商，更是应当把人员分工搞好，这样才可能真正做好产品。

过去的事情都过去了，回头看看自己这些年，在工作上还是花了不少心思，每次有想法，都会说出来，哪里觉得不对，都会认真提出自己的理由。努力做一些与众不同的事情，会写一些工作方面的文章，会用业余时间组织培训交流，会自己出钱买书送给同事。从未提出过让自己团队任何人加班，研发过程奖也从未给过自己一分钱。有时候真不知道自己的坚持是为了什么，努力过之后，发现能改变的东西还是太少，很失落。

曾经是一个缺乏勇气的人，下棋或者打游戏碰到形势不好就立刻认输，后来看我同学阿龙打星际，屡屡被人打得只剩一个农民还到处逃窜开矿企图翻盘，看得多了，也就比以前肯坚持。人的一生，两件事最重要，一是努力，二是选择。这两者都不容易，这次狠心选择了新的道路，希望能坚持下去，不知道再有9年之后，会是什么样？

Nicholas Zakas：

LeanPub 自出版一年记

作者/ Nicholas C. Zakas

前端咨询师、畅销书作者、技术布道者，世界顶级 Web 技术专家，曾在雅虎工作近 5 年，离开该公司前负责 My Yahoo! 和雅虎首页等大访问量站点设计，担任界面呈现架构师。拥有丰富的 Web 开发和界面设计经验，曾经参与许多世界级大公司的 Web 解决方案开发，是《JavaScript 高级程序设计》、《高性能

去年年初，我自出版了The Principles of Object-Oriented Programming in JavaScript这本书。之所以想趟趟这条路子，主要有几方面原因。其中最重要的，就是我根本不知道什么时候能写完它，不希望有交搞时间卡着。我也不想找一家出版社天天督促自己，他们有出版计划，但我没有。

我对自出版革命也怀有好奇心。很多有自出版经验的人对此议论纷纷，说好的和说不好的都有。毕竟我借助传统出版方式出过不少书，稍微一想就知道会失去什么：文字编辑、技术编辑、插图设计、营销推广，还有实体书。但我认为读者反馈可以代替文字和技术编辑，简单的图自己画就行，至于营销推广我不是有Twitter和博客嘛。我知道自出版的销量可能比不上与出版社合作的销量，可我不用卖那么多啊，因为我从售价中拿到的分成比例要比跟出版社合作多多了。

转眼一年过去了，我写写自己的体会吧。这本电子书最终还是让No Starch Press看上，出了一本纸版书，叫The Principles of Object-Oriented JavaScript。虽然我的最终目的不是这个，但连带出了一本纸书还算个惊喜吧。

选择 Leanpub

从一开始，我就想让自己能随时修改或修正书的内容。因为看

JavaScript》作者。个人网站 www.nczonline.net，Twitter：@slicknet。

着纸书上的打字和其他错误而束手无策，那种感觉非常难受。如果有读者反馈了一个错误，我希望立即就改正。电子书就可以做到这一点，可我又希望找一个能支持这种随时修改工作流的平台。

我考虑的另一个问题是写作格式。我真心想用Markdown，这是我最常用的格式。我越来越不喜欢使用Word，选中文本，然后修改样式。Markdown可以让我更自由，不用太关注文字将来显示成什么样。

我还希望能生成所有流行的电子书格式：PDF、ePub和MOBI。我不想让人抱怨说在哪个设备里看不了，所以这个平台应该能自动生成这些格式。

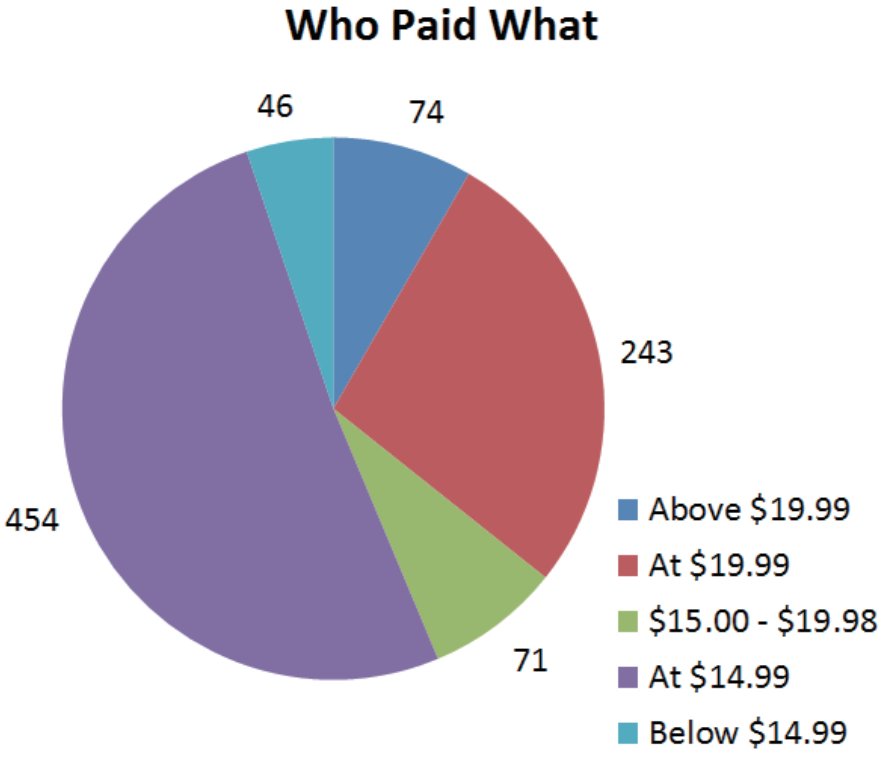
Leanpub完美满足这些需求。我就用Markdown写，随时可以修改，而它能自动生成所有格式。单凭这些我肯定就要选Leanpub了，可真正让我喜出望外的还有：

- 定价可调可以让我定一个可接受的低价和一个建议价。于是我就把低价定在$14.99，建议价定为$19.99。这样，所有买这本书的人都可以不用花$19.99，自己给自己打折。当然，如果愿意，他们也可以多花钱。我觉得这种给读者定价空间的方式太好了。
- 可定制的着陆页和URL都是在发布的时候自动生成的。那么我要做的，就是提供一些内容，然后页面就有了，就等着接单了。
- Leanpub处理订单和收益的方式好极了。我可不想自己手工来做这些。通过它，我可以在自己演讲的大会上发放一些优惠券，或者自己可以决定给谁打个折（或免费）。
- 每个月自动支付版税，存到PayPal上。作者的分成是每本售价的90%减50美分，这么好的买卖打着灯笼也难找啊。

任何平台都一样，都有一些古怪的情况要适应，比如默认编码和页面布局。而且图片也必须非常大，以便支持各种格式的缩放。不过总体上说，我的Leanpub自出版体验是非常愉快的。

成交价分析

我让读者自己定价的实验非常有意思。用我妈的话说："人家为什么要多给你钱呢？"这话说到了点子上。事实上，大多数读者都愿意乐享现成的折扣，只花$14.99，少数人会多选择多付一点。尽可能选择低价这不是问题，但我确实发现有些人会有意识地选择$19.99。先来看看基本销售情况（不包含捆绑销售的部分）：



- 总销售：$14 866
- 平均价：$16.74
- 无优惠平均价：$17.66
- 最高价：$78.62

大体上，在都剔除优惠的情况下，平均价比最低价略高出了$1.75。也就是说，人们平均来讲，还是愿意比最低价多出一点钱。最高价是$78.62，有点吓人，但也有出$30和$50的。事实上，支付价超过$19.99的很多，有的多几美分，有的多几美元。

在888个订单中，有74个高

出建议价，有243个按建议价支付。虽然多数还是按最低价付款的（使不使用优惠券的都算上），但不少人会选择多付这一点也不容忽视。甚至还有一位读者退了刚买的电子书，就想花高价再买一次。

我觉得这是一件不可思议的事：只要允许，人们实际上会多付钱的，并且不是有时候会这样，而是不断会这样。我是这样看的，让人们能够自己评估要购买的商品价值，这是一种权力。当然，我也设定了一个最低价，以保证最低每本收到$14.99，但让人们可以在此基础上随意定价，也让我摆脱了对最终能卖多少的内疚感。

虽然$14 000远比不上我出版的其他任何一本书的收入，但那些书的销售很大程度上取决于我在新书头一年都做了什么。所以，我最终卖的本数少，收入也大致相当。

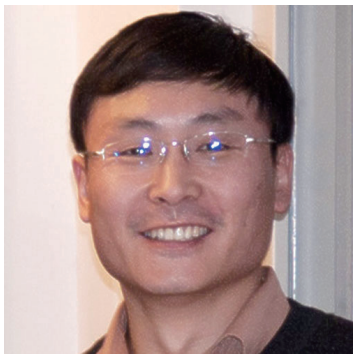
结论

我去年在Leanpub玩自出版的体验是正面的。通过这个平台写书和更新是一种享受。销售也令人满意，所以如果将来我再写一本书，还会继续用它。而且，对没有DRM限制的电子书我也有了自己足够的经验，今后我会考虑怎么把电子书做得更开放。没有时间卡在那的感觉太好了，我想什么时候写就什么时候写。这本书最终还是出了纸版，进一步坚定了我以后要写书就这么写的信念。

这并不是说与传统出版社合作没有意义。正因为跟Wiley和O'Reilly合作过，我很清楚自出版会让自己失去什么。我强烈建议有志著书立说的作者，至少要跟传统出版社合作一次，这样你才能了解整个出版流程，每个环节该怎么处置。完全靠自

己看似是最快的出版方式，但你必须能保证自己拿出来的东西合乎规范、质量过关。如果事先没有经过一两次出版社的指导，要做到这一点很难。

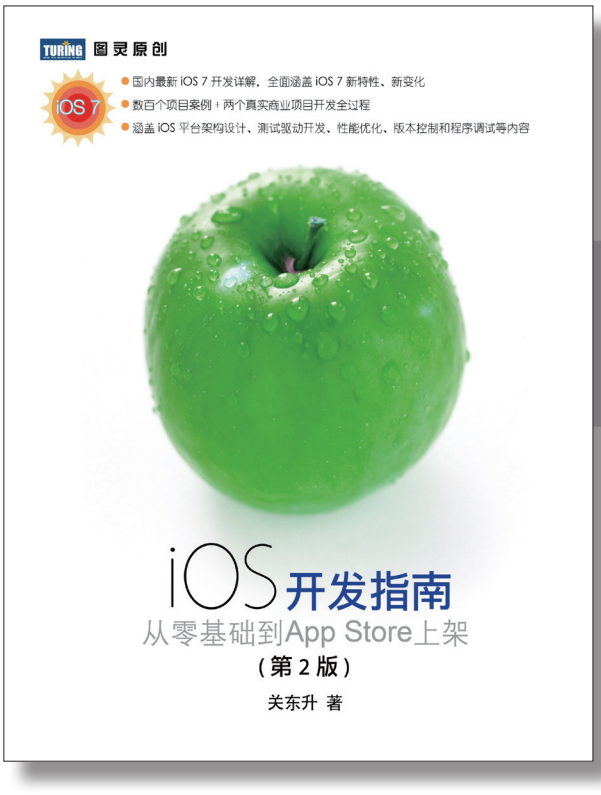
英文原文：http://www.nczonline.net/blog/2014/03/18/leanpub-one-year-later/



译者 / 李松峰

@ 李松峰，非计算机专业出身的技术爱好者，曾从事 5 年 Web 前后端开发工作，现为北京图灵文化发展有限公司 QA 部主任。2006 年开始涉足计算机相关图书翻译，至今翻译字数超过 458 万字，已出版译著 20 余部。图灵社区 ID: 李松峰

书　榜



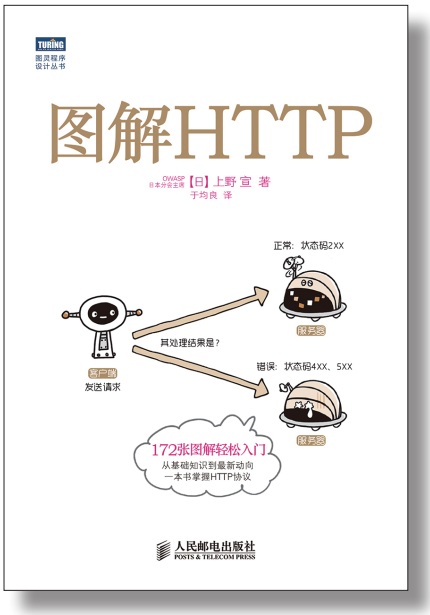
iOS开发指南：从零基础到App Store上架

作者：关东升

书号：978-7-115-34802-9

图灵社区推荐：7

本书采用全新的iOS 7 API，详细介绍了iOS 7相关的知识点。本书共分为4个部分：第一部分为基础篇，介绍了iOS的一些基础知识；第二部分为网络篇，介绍了iOS网络开发相关的知识；第三部分为进阶篇，介绍了iOS高级内容、商业思考等；第四部分为实战篇，从无到有地介绍了两个真实的iOS应用：MyNotes应用和2016里约热内卢奥运会应用。书中包括了100多个完整的案例项目源代码，大家可以到本书网站http://www.iosbook1.com下载。



2 图解HTTP

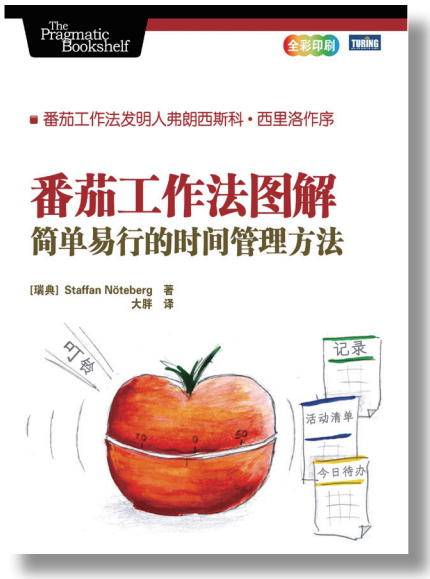
作者：上野 宣

译者：于均良

书号：978-7-115-35153-1

图灵社区推荐：12

本书的特色为在讲解的同时，辅以大量生动形象的通信图例，更好地帮助读者深刻理解HTTP通信过程中客户端与服务器之间的交互情况。



3 番茄工作法图解：简单易行的时间管理方法

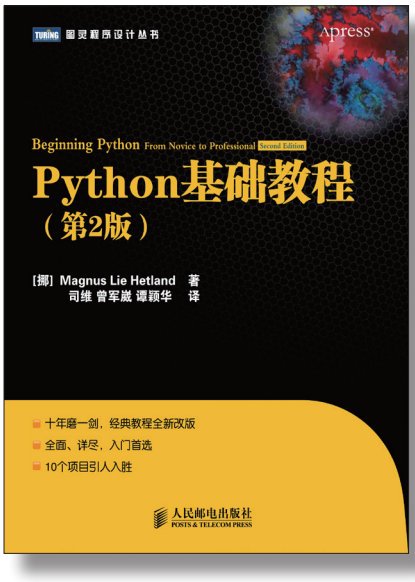
作者：Staffan Noteberg

译者：大胖

书号：978-7-115-24669-1

图灵社区推荐：13

作者根据亲身运用番茄工作法的经历，以生动的语言，传神的图画，将番茄工作法的具体理论和实践呈现在读者面前。番茄工作法简约而不简单，本书亦然。



4 Python基础教程（第2版）

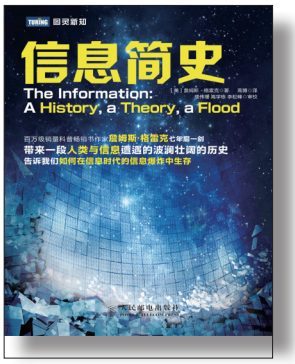
作者：Magnus Lie Hetland

译者：司维　曾军崴　谭颖华

书号：978-7-115-23027-0

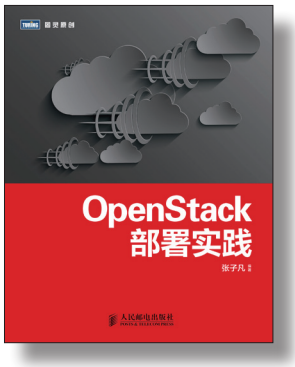
图灵社区推荐：7

本书从讲述Python的安装开始，一直到按照实际项目开发的步骤向读者介绍了几个具有实际意义的Python项目的开发过程，内容涵盖程序设计的方方面面。



5 信息简史

作者：［美］詹姆斯•格雷克
译者：高博
书号：978-7-115-33180-9
图灵社区推荐：23

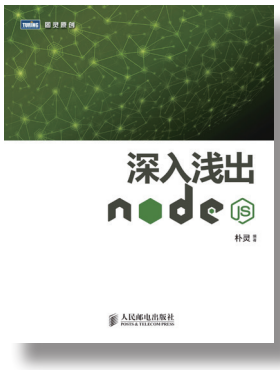


6 OpenStack部署实践

作者：张子凡
书号：978-7-115-34679-7
图灵社区推荐：5

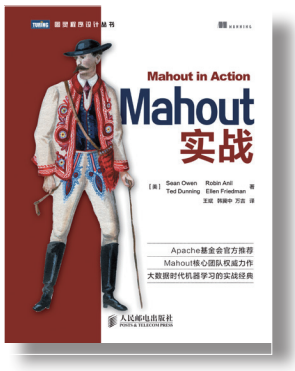
7 Android编程权威指南

作者：Brian Hardy　Bill Phillips
译者：王明发
书号：978-7-115-34643-8
图灵社区推荐：12



8 深入浅出Node.js

作者：朴灵
书号：978-7-115-33550-0
图灵社区推荐：13



9 Mahout实战

作者：Sean Owen　Robin Anil　Ted Dunning
　　Ellen Friedman
译者：王斌　韩冀中
书号：978-7-115-34722-0
图灵社区推荐：10

10 Google Glass开发指南

作者：BestApp工作室
译者：劳佳
书号：978-7-115-34947-7
图灵社区推荐：3

电子书榜

1.《关于mruby的一切》
——Ruby之父开拓嵌入式软件的广阔世界，关于mruby的一切！

2.TCP Sockets编程
——短小精悍的Socket编程手册，利用Ruby的语法糖实现高效构建

3.《程序员面试金典（第5版）》
——谷歌资深面试官的经验之作，紧扣程序员面试的每一个环节

4.《Ruby on Rails教程》
——如果想学习使用Ruby on Rails进行Web开发，你应该从这开始

5.《HTML5与CSS3基础教程（第8版）》
——全球最畅销Web开发入门书最新版

6.《嗨翻C语言》
——这里没有学究腔，让C语言快乐起来！

7.《Web性能权威指南》
——所有Web开发人员都应该知道的网络知识

8.《软件定义网络：SDN与OpenFlow解析》
——一线专家厚积薄发力作，网络可编程技术无可替代的权威解读

9.《Android编程权威指南》
——把Android开发所需的庞杂知识、行业实践、编程规范等融入一本书

10.《互联网思维的企业》
——互联企业不论在哪里，都可以快速获得消费者支持和竞争优势

“芯”照不宣:《CPU 自制入门》

作者 / 邱岳

二爷鉴书，第一时间向大家推荐好书和介绍，同时坚决揭露烂书，以IT、互联网行业为主。个人品味，仅供参考。微信号「Findbook」。

这本书的译者赵谦是我多年好友，我前些天跟他要了手稿，连夜读完，跟大家分享一些感受。另外由于他人在日本，便让图灵社把样书都寄到了我这里，除了他托我转赠朋友外还剩下几本。我便顺水推舟做个人情，送给二爷鉴书的读者们吧。

这篇东西可能有些令人不适的专业术语，文青慎入。

书是连滚带爬读完的，读完静思，心里突然充满了感动，似乎看到那些在周围人眼中毫无情趣的所谓「屌丝」们嘴角不经意的浪漫。他们彼此并不相识，但他们之间的信任，构成了整个当代电子文明的基石。

为啥这么说呢？就说我们手上的手机，我们使用的手机应用来自设计师们的设计，而设计师的草稿需要软件工程师的实现，软件工程师调用手机系统封装好的接口，接口依赖于操作系统的调度，操作系统建立在硬件驱动之上，驱动经过编译器变成芯片们可以理解的指令，指令通过电路板上纤细的金属进入芯片，在芯片中的逻辑门间穿梭，逻辑门等待场效应管的开启和关闭，而硅晶片上的刻蚀，形成了这些场效应管。

作为一个做产品设计的，向应用中添加一个按钮的时候，我从不担心芯片上发生的事情。我知道它们会精确工作。正是这样的信任成全了设计师，成全了用户。

《CPU自制入门》可以帮助软件工程师深入了解硬件与底层，开发出高效代码。硬件工程师可以在本书基础上设计定制硬件，开发高速计算机系统。相信所有读者都可以在本书的阅读过程中，体会到自制计算机系统的乐趣与热情。

我在高中毕业那一年买了人生第一块单片机开发板，当我按照手册上一步一步插线上电，只写了不到十行代码就点亮了板上的一个 LED 时，我震住了，我想向墨绿色的板子深深鞠躬，向芯片那边的可靠致意，把信任交给他们，与他们心照不宣。

大部分程序员对计算机系统都会有或模糊或清晰的概念模型，有本《深入理解计算机系统》把这个描述得出神入化（这是我给所有程序员荐书榜的第一名）。而《CPU 自制入门》则是对计算机系统实现模型的深入解释，一本看起来应该非常晦涩的书，竟然具备了「操作性」，确实挺不容易的。

再说说书的内容，书里基于 FPGA 实现了一个 CPU 原型，说起来也不能算是 CPU，因为 FPGA 里面还实现了总线、内存和其他模块，所以应该算是一个片上系统（SoC）。

第一章基本从0开始介绍了 CPU 的原理，这个0是真的很0，从场效应管开始到逻辑门到数字电路，然后简单介绍了硬件描述语言（HDL），又深入浅出地介绍了 CPU 的工作原理，然后就地用 Verilog 语言实现了一个简单的 CPU 原型。读这部分的时候我挺怀旧的，想起大学时光。这一章的占比最大，也算是最有意思的一部分。

即使你永远也不设计 CPU，从这一章中也能学到不少解决问题的思路和方法。比如锁存器，比如处理 CPU 内部的流水线冒险，比如总线裁决和控制等等，有不少让人拍案叫绝的智慧，很有借鉴意义。

第二章是讲如何制板，从原理图到封装到布板一直到最后的制板和焊接，日本人写书都娓娓道来，一帧一帧的，书里竟然还有买电子元器件的店的地址，还真是严谨得有点儿蠢，哈哈哈哈。

第三章是编程，算是操作手册吧，最终的样例程序是汇编，汇编器是现成的，其中说了交叉编译啦，中断啦，JTAG 调试之类的一些概念，还有实际的操作步骤。

总的说来，整本书是从底层到上层，从原理到工程，从构思到实现的过程。话说估计大部分人永远都不会真的「设计 CPU」，这本书的真正目的其实是通过一个最小系统来让大家手把手地理解 CPU。

FPGA 可以引出一个非常有意思的话题。硬件行业一直高举着摩尔定律的大旗蒸蒸日上，但市场对性能和存储的需求却翻得比摩尔定律还快，前几天跟朋友聊天，说起淘宝商品详情页的每日 PV 数量级，我差点儿吓休克。在这样的情景中，我们熟悉的 CPU 的运算控制架构在需求面前不经意流露出了一点点力不从心，虽然不知道大家在通用硬件架构上拼软件的日子还有多久，但我猜至少已经到了下半场了吧。

另一方面，从一九八几年普适计算提出，到现在穿戴式设备火爆。计算能力的场景化和个性化越来越重要。有的需要大量的数字信号处理，有的需要低功耗，有的需要高速数模转换或者海量存储。随着对更个性化异构运算的需求越来越细致以及硬件订制成本的下降，基于个性化硬件架构的生意可能会越来越多。原来这些是英伟达英特尔高通之类的大厂才有实力做的事情，或许未来在车库就能完成。

从这个意义上来看，iPhone 5s 其实是一部跨时代的手机，他的协处理器直接在硬件层面解决了「低功耗地完成追踪及数据记录」的问题。

赵谦在葡萄牙参加一个学术会议期间，曾经给我发来一个消息，告诉我会上有个人用 FPGA 加速了 MySQL 的 group 效率，「或

许很快就能走出实验室了」，他兴奋地跟我说。我顺手翻了几篇文献，有个挺感慨的事情是，在这个行业中谷歌微软等公司的硬件实验室频繁露面，国内的互联网厂商似乎还鲜有发声。

是啊，听说国内好多高科技企业的研究所还有「赢利压力」呢。

瞎扯·记与本书译者赵博士二三事

这本书的译者赵谦是我多年的老友，大约十年前我们相识在学校信息学奥林匹克竞赛集训队里，当时我们只有一本严蔚敏的绿皮《数据结构》和一堆竞赛题目。学校不太重视这个学科，也没什么培训，就把我们丢在机房自己折腾，非常自由。

我们刚认识的时候，他给我讲一道防空高射炮打飞机的算法题，「这是一个二叉树」，他用笔在纸上画起来，很笃定地告诉我，我很认真的记在心里。

后来我们都知道那是个典型的最长不下降子串的问题，但心照不宣地都没有再聊起这个事情。我读大学参加 ACM 邀请赛，有一道类似的题目，我一次编译测试提交通过，我的队友都震惊了。我望向窗外，天那么蓝，树叶那么绿，空气温热直插入肺，想起当时他给我讲「这是一个二叉树」时紧紧包裹手腕的袖口，我想我这辈子都很难再有这么洒脱这么酷毙又自信的时刻了。

他高中时和同学组过一个乐队，他是吉他手，乐队上过《快乐大本营》节目，成了我们学校的明星，收发室一度被写给他们的信件淹没。有一次我在他家玩儿，他指着吉他问我学不学，我说走着呗。可惜最后我还是没学好，没能成功沦为一个文青。

之后我们断断续续地写歌。我们像所有无病呻吟多愁善感的年轻人一样歌颂理想和心爱的姑娘，用的都是些俗套的和弦平淡

的旋律和毫无新意的韵脚。前些日子我们终于合作匆匆写了首歌，歌曲名字叫做《反正世界上已经有这么多烂歌也不差这一首》，这是后话了。

高中快毕业时老师来找我们，说你们俩会电脑，折腾个网站代表学校参加比赛。其实我们都不会这个，于是我去买了本21天精通Dreamweaver，扳着指头算了一下时间应该来得及，他说那咱来吧。

我们俩合作了两个作品。后来两个都得了市里的奖，被送到省里参加比赛，有一个得了一等奖。后来老师私下说你们有点儿不懂事儿，如果你们每个作品只署一个名字说不定能得两个一等奖，浪费了。我们大度地淡淡一笑，说卧槽你为什么不早点说。

得奖的作品来源于我当时看的一本关于视错觉的书，那本书给我的震动很大，赵谦选定了一个主题——「眼见未必为实」，他说这是包含朴素的哲学含义的。

当时我们都不会做动画、视频和音乐什么的。我们一商量，说去他妈的既然我们跟他们一样搞不过他们，干脆就做点儿不一样的。于是我们的所有图标、Banner 和导航都是用铅笔画好扫描进去的。为我们执笔的那个姑娘是赵谦的歌迷，她可能是我合作过最配合的设计师了，满足了我所有的苛刻的挑剔的反复的要求。我想她将来一定会是一个优秀的乙方。

当时的作品至今还在硬盘里保存着，作品叫做 Lying Eyes，若干年后我才知道这是老鹰乐队的一首歌名，我当时的第一个反应是，还好，没有语法错误。

「我们将向您证明，您的眼睛有时会说谎」我们在网站入口页上深情地写到。

后来我们从高中毕业上了大学，虽然不在同一个城市，但还是保持联系，扯各种层次的蛋。分享见闻，分享心得，分享种子，猜测未来的日子，总觉得未来遥远的就像一篇课堂作文。

上个月我和媳妇远赴东瀛去拜访他一家人，他可爱的女儿趴在我怀里呼呼大睡，我把她摇醒说来吧姑娘，邱爸爸给你讲讲人生的哲理和生活的真谛。宝贝嘴一撇要哭出来的样子，我心里暗喜，将来肯定不会被装逼的坏男孩儿骗。想到这里背后一凉，昨天自己还是那个装逼的坏男孩儿，今天已经是他们想诱骗的姑娘的父亲，心中一阵绞痛。快如迅雷的时间啊，请给我们掩耳盗铃的片刻喘息吧。

在我们的大学期间，他断断续续做了个化妆品评测网站，号称当时是国内最大的，成了一个 Web 程序员。我说你真他妈没出息，搞那些花里胡哨的有啥用，让你看看什么才是真正有价值的技术，于是一头扎进芯片和电烙铁之间开始搞嵌入式。

就在他眼看就要成为互联网行业一颗冉冉升起的新星，而我也即将在元件库中找到属于我的封装之际，突然峰回路转。他去日本深造研究起 IC 设计，我则彻底告别了底层和代码，一头扎进了互联网行业。

上周，他翻译的《CPU 自制入门》完稿了，我则即将磕磕绊绊地开启第七年 Web 产品经理的生涯。

命运真他妈是个特别神奇的东西。 ■

欢迎加入
图灵社区

最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

现在购买电子书，读者将获赠书款20%的社区银子，可用于兑换纸质样书。

最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

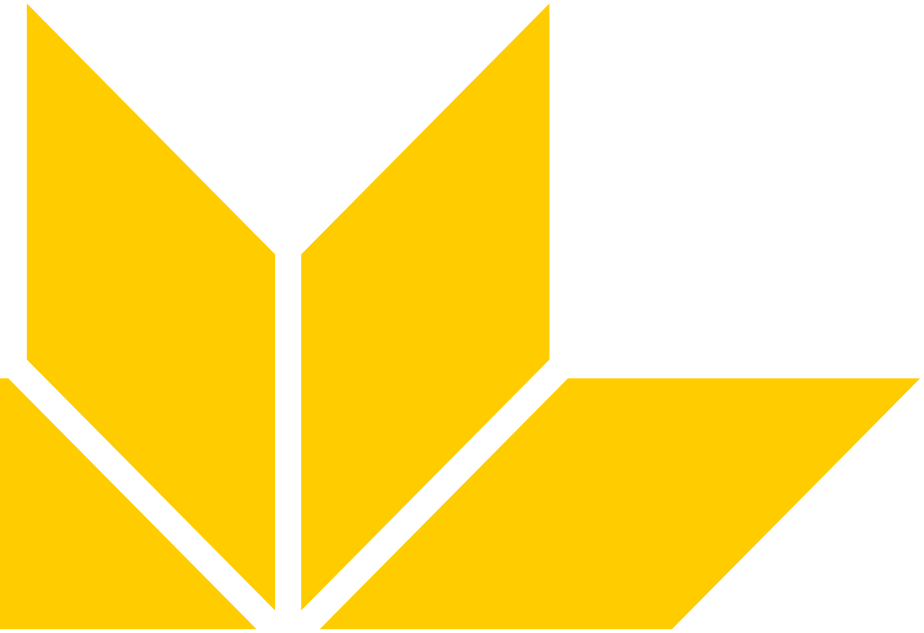
最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、乐译、评选等多种活动，赢取积分和银子，积累个人声望。

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