



IOC + JavaScript

Jeremy Grelle, SpringSource 主管工程师

Github/Twitter : @jeremyg484



內容？

cujo.js

- 类似于 Spring 的概念，但是并非 Spring 到 Javascript 的端口
- 包含 Javascript 的函数和原型根
- 为下一代 JavaScript 应用程序提供体系结构工具

<http://cujojs.com/>

方式？

代码演示：Monty Hall UI && TodoMVC

最近的项目统计

- 6 个“HTML 页面”
- 300 多个 Javascript 模块
- 100 多个“视图模块”，每个都具有：
 - HTML 模板
 - CSS 文件
 - i18n 捆绑
 - 测试工具

不包含第三方模块！

最近的项目统计

手动依赖项管理在这种规模下不可行

帮助！

更大型、更复杂的应用程序需要精心构建的严密体系结构、模式和代码组织。

-- Brian Cavalier

IOC

- 从 Spring 中，我们可以得知良好的体系结构架设有助于管理复杂性
- Javascript 也不例外
- 在浏览器中和服务器上

我们是否能够 ... ?

- 构建更易于维护和测试的更小模块
- 将配置和连接从应用程序逻辑中分离
- 将所有内容重新组合在一起

在 Javascript 中应用 IOC 概念

- 公布的组件创建
- 生命周期管理
- 配置
- 依赖项注入
- AOP

前端 Javascript 的 IOC

- 可能的情况是什么？
 - XML？风格并非十分类似 Javascript
 - 注释？不存在已有基础架构，需要购买源
- Javascript 很灵活：我们是否可以使用该语言？

组件

我们首先需要的是构建组件的方法

AMD

异步 (Asynchronous)

模块 (Module)

定义 (Definition)

三个主要部分

- 模块格式
- 运行时加载器
- 构建时编译器（推荐用于生产）

AMD

设计过程中考虑了浏览器环境

- 异步加载
- 无需分析或转编译
- 内置收尾
- 通过插件加载其他资源类型

由什么提供支持？

- dojo 1.7+
- cujo.js
- jQuery 1.7+
- MooTools 2+
- Lodash 等许多其他内容

define()

AMD 授权单个标准化的全局函数。

```
define();
```

define()

define(factory);

define(dependencyList, factory);

AMD 模块变体

“标准”AMD

```
define(['when', 'pkg/mod'], function (when, mod) {  
    // use the dependencies, return your module:  
    return {};  
});
```

AMD 模块变体

打包 AMD 的 CommonJS

```
/* no deps, factory params != 0 */
define(function (require, exports, module) {
// sync, r-value require
var when = require('when');
// decorate your exports object
exports.bestUtilEver = function (stuff) {
    return when(stuff);
};
});
```

AMD 模块变体

打包 AMD 的 Node

```
/* no deps, factory params != 0 */
define(function (require, exports, module) {
// sync, r-value require
var when = require('when');
// declare your exports
module.exports = function bestUtilEver (stuff) {
    return when(stuff);
};
});
```

UMD

通用 (Universal)

模块 (Module)

定义 (Definition)

UMD

用于简要说明环境并正确导出模块的样板

- AMD + 旧的全局函数
- AMD + CommonJS
- AMD + Node
- AMD + Node + 旧的全局函数等...

UMD 模块变体

AMD + Node (我们的最爱)

```
(function (define) {  
  
    define(function (require) {  
  
        // module code goes here  
  
    } );  
  
})(typeof define === 'function' && define.amd  
    ? define  
    : function (factory) {  
        module.exports = factory(require);  
    } );
```

UMD : AMD + Node

应用程序/游戏/控制器

(代码演示)

引导 AMD 应用程序

“其他全局函数”

- `curl()`;
- `requirejs()`;
- `require()`; (全局函数 `require` 出现问题!)

引导 AMD 应用程序

run.js

([代码演示](#))

AMD 插件

- 相同的依赖项机制
- 非 AMD 资源
- text! - HTML 模板和其他文本
- css! 和 link! – 样式表
- i18n! – 本地化 Google 地图
- JSON 数据等

AMD 插件

可以实现更强大的功能

- wire! - wire.js IOC 容器集成
- has! - has.js 功能检测和条件模块加载
- cs! – 加载并转编译 Coffeescript

插件

应用程序/主要
(代码演示)

CommonJS 模块格式

- 每个文件都是自带范围的模块
- 无收尾、无工厂、无 `define()`
- `Require`、`exports` 和 `module` 为“自由变量”

curl.js <3 CJS!

“编译到 AMD”

Node != CommonJS

- exports === this
- exports === module.exports

WTF

我知道您在思考什么

哪一个？！？

为什么存在 2（3？）个模块格式，我如何知道应选择哪个？！？！！

情况变得更糟

ES Harmony 模块即将推出

问题：Harmony 是一种创作格式。它无法处理：

- 依赖项管理
- 打包、版本管理
- 编译、串联
- 非 Harmony 资源（CSS、HTML、JSONP 等）

放轻松



放轻松

您的代码是安全的！

AMD 使用 CJS 和（即将推出的）Harmony 模块

演变

```
// curl.js config (coming soon!)
packages: [
  { name: 'node-thing', location: 'lib/node/thing', main: './main',
    transform: ['curl/transform/cjsm11'] },
  { name: 'cs-thing', location: 'lib/cs/thing', main: './init',
    transform: ['curl/transform/coffee'] },
  { name: 'future-thing', location: 'lib/harmony/stuff', main: './main',
    transform: ['curl/transform/es7'] }, ... ] ,
```

如今的 CommonJS 模块

monty-hall-ui/cjsm (分支)

相同的模块编写为 CommonJS Modules/1.1, 但是未打包 !

(代码演示)

微模块

越小越好！

微模块

单函数模块

- 更易于重用
- 更易于测试
- 更易于发现

危险 !

<https://github.com/dojo/dijit/blob/ef9e7bf5df60a8a74f7e7a7eeaf859b9df3b0>

```
1 define([
2   "dojo/_base/array", // array.forEach
3   "dojo/_base/declare", // declare
4   "dojo/_base/Deferred", // Deferred
5   "dojo/i18n", // i18n.getLocalization
6   "dojo/dom-attr", // domAttr.set
7   "dojo/dom-class", // domClass.add
8   "dojo/dom-geometry",
9   "dojo/dom-style", // domStyle.set, get
10  "dojo/_base/event", // event.stop
11  "dojo/keys", // keys.F1 keys.F15 keys.TAB
12  "dojo/_base/lang", // lang.getObject lang.hitch
13  "dojo/sniff", // has("ie") has("mac") has("webkit")
14  "dojo/string", // string.substitute
15  "dojo/topic", // topic.publish()
16  "dojo/_base/window", // win.withGlobal
17  "./_base/focus", // dijit.getBookmark()
18  "./Container",
19  "./Toolbar",
20  "./ToolbarSeparator",
21  "./layout/_LayoutWidget",
22  "./form/ToggleButton",
23  "./_editor/_Plugin",
24  "./_editor/plugins/EnterKeyHandling",
25  "./_editor/html",
26  "./_editor/range",
27  "./_editor/RichText",
28  "./main", // dijit._scopeName
29  "dojo/i18n!./_editor/nls/commands"
30 ], function(array, declare, Deferred, i18n, domAttr, domClass, domGeometry, domStyle,
31           event, keys, lang, has, string, topic, win,
32           focusBase, _Container, Toolbar, ToolbarSeparator, _LayoutWidget, ToggleButton,
33           _Plugin, EnterKeyHandling, html, rangeapi, RichText, dijit){
34 }
```

危险 !

在使用微模块时，我们如何避免依赖性困境（Dependency Hell）？

连接

- 框线图中的线
- 与您放入框中的内容同样重要
- 很遗憾，我们最终将线放置在框内

AMD

- 保持良好的关注点分离状态
- 但是更类似于 Java import，并非一定要适用于所有情况。

示例

```
define(['dojo/store/JsonRest'], function(JsonRest) {
    function Controller() {
        this.datastore = new JsonRest({ target: "mycart/items/" });
    }

    Controller.prototype = {
        addItem: function(thing) {
            return this.datastore.put(thing);
        },
        // ...
    }

    return Controller;
});
```

这听上去如何？

- `this.datastore = new JsonRest(...)` 实质上是我们 Controller 框中的一条线
- 您将如何对其进行单元测试？
- 您是否可以将其与其他数据存储类型一同使用？
- 多个实例，每个均具有不同的存储类型？
 - 不同的目标 URL？

重构

```
define(function() { // No AMD deps!

function Controller(datastore) {
    this.datastore = datastore;
}

Controller.prototype = {
    addItem: function(thing) {
        return this.datastore.put(thing);
    },
    // ...
}

return Controller;
}());
```

或者类似地

```
define(function() { // No AMD deps!  
  
    // Rely on the IOC Container to beget new instances  
  
    return {  
        datastore: null,  
        addItem: function(thing) {  
            return this.datastore.put(thing);  
        },  
        // ...  
    };  
});
```

我们做了什么？

- 解耦具体的 JsonRest 实施
- 重构以依赖数据存储接口
 - 即使接口是隐式的

我们完成了什么？

- 将绘制线的职责移出控制器。
- 使控制器更灵活且更易于测试

但是我们造成了一个问题

谁来提供数据存储？

我们知道如何解决

在应用程序复合层中注入依赖项

DI 与应用程序复合

```
define({
controller: {
  create: 'myApp/controller',
  properties: {
    datastore: { $ref: 'datastore' }
  }
},
datastore: {
  create: 'dojo/store/JsonRest',
  properties: {
    target: 'things/'
  }
}
});
```

DOM

- 很显然，在前端 Javascript 中必须使用 DOM
- 类似问题：框中的线

示例

```
define(['some/domLib'], function(domLib) {  
  
  function ItemView() {  
    this.domNode = domLib.byId('item-list');  
  }  
  
  ItemView.prototype = {  
    render: function() {  
      // Render into this.domNode  
    }  
  }  
  
  return ItemView;  
});
```

听上去一样

- 取决于 HTML ID 和 DOM 选择器库
- 更改 HTML 可能会损坏 JS
- 必须模拟 DOM 选择器库

重构

```
define(function() { // No AMD deps!

function ItemView(domNode) {
  this.domNode = domNode;
}

ItemView.prototype = {
  render: function() {
    // Render into this.domNode
  }
}

return ItemView;
});
```

更出色

- 解耦 DOM 选择机制
- 与 HTML：无需更改 ItemView 的源，便可注入不同的 DOM 节点。

DOM 与应用程序复合

```
define({
  itemView: {
    create: {
      module: 'myApp/ItemView',
      args: { $ref: 'dom!item-list' }
    }
  },
  plugins: [
    { module: 'wire/dom' }
    // or { module: 'wire/sizzle' }
    // or { module: 'wire/dojo/dom' }
    // or { module: 'wire/jquery/dom' }
  ]
});
```

DOM 事件

```
define(['some/domLib', 'some/domEventsLib'], function(domLib, domEventsLib) {  
  
  function Controller() {  
    domEventsLib.on('click', domLib.byId('the-button'), this.addItem.bind(this));  
  }  
  
  Controller.prototype = {  
    addItem: function(domEvent) {  
      // Add the item to the cart  
    }  
  }  
  
  return Controller;  
});
```

听上去一样，只是更加糟糕！

- 取决于：
 - 硬编码的事件类型、
 - HTML ID、
 - DOM 选择库、
 - DOM 事件库
- 更多模拟

重构

```
define(function() { // No AMD deps!

    function Controller() {}

    Controller.prototype = {
        addItem: function(domEvent) {
            // Update the thing
        }
    }

    return Controller;
} );
```

更出色

- 只关心常规事件：“现在是时候将商品添加到购物车了”
- 多个 DOM 节点上的不同/多个事件类型
- 无硬编码的 DOM 选择器：多个
- 控制器实例仅需模拟 `domEvent`, 然后调用 `addItem`

DOM 事件与应用程序复合

```
itemViewRoot: { $ref: 'dom.first!.item-view' },  
  
controller: {  
  create: 'myApp/Controller',  
  on: {  
    itemViewRoot: {  
      'click:button.add': 'addItem'  
    }  
  }  
},  
  
plugins: [  
  { module: 'wire/on' }  
  // or { module: 'wire/dojo/on' }  
  // or { module: 'wire/jquery/on' },  
  
  { module: 'wire/dom' }  
]
```

JS 到 JS 的连接

各个组件能否以比 DI 更松散的耦合方式进行协作？

合成事件

- Javascript 方法的作用类似于事件
- 将各个方法“连接”起来
- 任何一个组件都无法感知另一个组件

示例

```
Controller.prototype.addItem = function(domEvent) {...}  
CartCountView.prototype.incrementCount = function() {...}
```

使用 DI

```
controller: {
  create: 'myApp/cart/Controller',
  properties: {
    cartCountView: { $ref: 'cartCountView' }
  }
},
cartCountView: {
  create: 'myApp/cart/CartCountView'
}
```

我们可以改进的地方

- 控制器现在依赖于 CartCountView 接口
- 要对控制器进行单元测试，必须模拟 CartCountView
- 如果我们想在其他时候更新购物车计数，应该怎么办？

合成事件连接

```
controller: {
  create: 'myApp/cart/Controller' ,
  cartCountView: {
    create: 'myApp/cart/CartCountView',
    connect: {
      'controller.addItem': 'incrementCount'
    }
}
```

更出色

- 应用程序复合层进行连接
- 控制器不再依赖于 CartCountView
- 进行此连接时，无需对任何组件重新进行单元测试
 - 之后连接被断开时也无需这样做
 - 仅需重新运行功能测试
- 通过直接从应用程序复合规范中删除，即可完全移除 CartCountView

仍然不完美

如果 `addItem` 以某种方式抛出或失败，应该怎么办？

AOP 连接

```
controller: {
  create: 'myApp/cart/Controller' ,

  cartCountView: {
    create: 'myApp/cart/CartCountView',
    afterReturning: {
      'controller.addItem': 'incrementCount'
    }
}
```

进一步探讨

- 只对成功进行增量计数
- 如何处理失败？

AOP 连接

```
controller: {
  create: 'myApp/cart/Controller',
  afterReturning: {
    'addItem': 'cartCountView.incrementCount'
  },
  afterThrowing: {
    'addItem': 'someOtherComponent.showError'
  }
},
cartCountView: {
  create: 'myApp/cart/CartCountview'
},
someOtherComponent: // ...
```

更出色！但还不完美

- 解耦更彻底、可测试性和可重构性更好
- 仍然有一层我们可以移除的耦合

耦合的参数

```
function Controller() {}

Controller.prototype = {
  addItem: function(domEvent) {
    // How to find the item data, in order to add it?
  }
}
```

耦合的参数

- 控制器接收到 `domEvent`, 但必须找到关联的数据, 才能更新
- 需要 DOM 遍历, 并理解 DOM 结构
 - 数据 ID 或散列密钥隐藏在 DOM 属性中 ?
- 为进行单元测试, 必须进行模拟

耦合的参数

控制器确实只关注这个 domEvent

重构

```
function Controller() {}

Controller.prototype = {
  addItem: function(item) {

    // Just add it
  }
}
```

转换连接

可以转换数据的连接！

转换函数

```
define(function() {  
  
    // Encapsulate the work of finding the item  
    return function findItemFromEvent(domEvent) {  
        // Find the item, then  
        return item;  
    }  
  
});
```

应用程序复合

```
itemList: { $ref: 'dom.first!.item-list' },  
  
findItem: { module: 'myApp/data/findItemFromEvent' }  
  
controller: {  
  create: 'myApp/Controller',  
  on: {  
    itemList: {  
      'click:button.add': 'findItem | addItem'  
    }  
  }  
}
```

啊，最后

- 更容易地对控制器进行单元测试
- 用于找到对象的算法
 - 可以更容易的单独进行单元测试
 - 可独立于控制器进行更改
 - 可在应用程序的其他部分中进行重用

棒极了，大功告成，是吗？

还没有...

异步如何呢？

- 最常发生于组件与系统边界
- 因此，连接经常需要为异步
 - 典型示例：XHR

示例

```
Controller.prototype.addItem = function(item, callback) {...}  
CartCountView.prototype.incrementCount = function() {...}
```

示例

```
controller: {
    create: 'myApp/cart/Controller',
    afterReturning: {
        'addItem': 'cartCountView.incrementCount'
    },
    afterThrowing: {
        'addItem': 'someOtherComponent.showError'
    }
},
cartCountView: {
    create: 'myApp/cart/CartCountView'
},
someOtherComponent: // ...
```

噢哦

- 将函数结果从返回值移动到了参数列表
- 由于 addItem 无法返回任何结果， afterReturning 不起作用！
- 那么我们如何提供此回调呢？

简短异步绕行

- Javascript 围绕单线程事件循环而设计
- 浏览器 DOM 事件和网络 I/O 是异步的
- SSJS 平台（节点、RingoJS 等）围绕异步 I/O 而构建
- AMD 模块加载是异步的 -- AMD 中的 A !

回调

典型的解决方案是回调，也称作“持续传递”

示例

```
// You wish!  
var content = xhr('GET', '/stuff');
```

添加回调和错误处理程序

```
xhr('GET', '/stuff',
    function(content) {
        // do stuff
    },
    function(error) {
        // handle error
    }
);
```

到处都是回调

```
// It's turtles all the way *up*
function getStuff(handleContent, handleError) {
    xhr('GET', '/stuff',
        function(content) {
            // transform content somehow, then
            // (what happens if this throws?)
            handleContent(content);
        },
        function(error) {
            // Maybe parse error, then
            // (what happens if THIS throws?!?)
            handleError(error);
        }
    );
}
```

异步一团糟

- 代码快速变为深层嵌套，更难以推算
- 熟悉的编程惯用语法不起作用
 - 上下颠倒：值和错误从堆栈向下流动，而不是向上。
 - 函数不再可以轻易编写： $g(f(x))$ 不再起作用
 - 尝试/捕获/最后，或者推理上相似的事项都是不可能的
- Callback 和 errback 必须添加到可能最终导致异步操作的每个函数特征码
- 协调多个异步任务让人痛苦

Promise

- 同步构造
- 不是新创意
- 类似于 `java.util.concurrent.Future`
- 结果或错误的占位符将在稍后具体化。

示例

返回一个内容或错误将在其中具体化的 Promise。

```
function getStuff() {  
  var promise = xhr('GET', '/stuff');  
  return promise;  
}
```

Promise

- 恢复调用并返回语义
 - 将函数结果移动回到返回值
 - 移除回调函数特征码污染
- 向异常传播提供异步模拟，这是正常顺序

有关 Promise 的更多信息

http://en.wikipedia.org/wiki/Futures_and_promises

<http://blog.briancavalier.com/async-programming-part-1-its-messy>

<http://blog.briancavalier.com/async-programming-part-2-promises>

<http://github.com/cujojs/when/wiki>

<http://wiki.commonjs.org/wiki/Promises/A>

Promise

- 多种提议的标准
- Promises/A 实际标准
 - cujo.js : when.js
 - Dojo : dojo/Deferred
 - jQuery : \$.Deferred (well, close enough)
 - Q
 - soon YUI、Ember

IOC + Promise

- Promises/A 是一种异步的集成标准
- IOC 旨在将各个组件粘合起来，使其能够互相协作
- 听起来就像是匹配！

用于返回 Promise 的重构

```
Controller.prototype.addItem = function(item) {  
  // Asynchronously add the item, then  
  return promise;  
}
```

可感知 Promise 的 AOP

```
controller: {
    create: 'myApp/cart/Controller',
    afterResolving: {
        'addItem': 'cartCountView.incrementCount'
    },
    afterRejecting: {
        'addItem': 'someOtherComponent.showError'
    }
},
cartCountView: {
    create: 'myApp/cart/CartCountView'
},
someOtherComponent: // ...
```

成功

- 只有在成功添加商品后，计数才会增加！
- 如果添加失败，就会显示错误

无需异步的异步

- 用于异步连接的可感知 Promise 的 AOP
- AMD 加载器管理异步模块加载和依赖关系图形解决方案。
- 可感知 Promise 的 IOC 容器：
 - 与 AMD 加载器集成，以加载在应用程序复合规范中使用的模块。
 - 异步组件创建：构造函数或纯文本函数可以返回一个 Promise
 - 异步 DI：组件引用可为了组件解析而作为 Promise 注入
 - 组件启动/关闭方法可以返回一个 Promise

连接

- 在组件中实施应用程序逻辑
- 通过应用程序复合以非侵入方式将各个组件连接起来
 - DI、事件（DOM 和 JS）、AOP、可感知 Promise 的 AOP
- 通过沿着连接转换数据，对 API 进行调适。实现更轻松的测试和重构 :)

组织！

组件、组件、组件

“左边的文件树”派上实际用场！

-- Brian Cavalier

组织！

将应用程序分割为多个功能区

谈论应用程序时，您讨论的是哪些方面？

组织

应用程序/

(代码演示)

案例：视图组件

视图组件包含

- HTML(5) 模板（保持简单！）
- CSS 文件（OOCSS/SMACCS 的结构位）
- i18n 文件
- javascript 控制器（可选）
- 测试工具（也用于设计）
- 进行呈现所必需的任何资产
- 任何视图特定的数据转换、验证等
- 线路规范（可选）

案例：视图组件

应用程序/简介

(代码演示)

测试可视化组件

如何测试？？？？

测试可视化组件

双重负荷测试工具

- 创建 HTML 和设计 CSS 的定位工具
- 用户驱动的测试工具
- 单元测试工具

测试可视化组件

双重负荷测试工具

(代码演示)

单元测试

- 越小越好
- 更少的依赖项意味着更少的模拟！

单元测试

(代码演示)

cujo.js

- AMD 模块 - curl 和 cram
- IOC 与应用程序复合 – 线路
- Promises/A – 何时
- AOP – 合并
- ES5 - poly
- 数据绑定 - cola (alpha)

替代

- AMD 模块 – RequireJS、Dojo、Isjs、BravoJS
- IOC - AngularJS
- Promise – Q、Dojo
- AOP - Dojo
- ES5 - es5shim
- 数据绑定 – 骨干以及所有其他人

cujo.js

从 <http://cujojs.com> 获取

讨论、公告、问题等

<https://groups.google.com/d/forum/cujojs>

问题？

Cloud Foundry 启动营

在www.cloudfoundry.com注册账号并成功上传应用程序，
即可于12月8日中午后凭账号ID和应用URL到签到处换取Cloud Foundry主题卫衣一件。



iPhone5 等你拿

第二天大会结束前, 请不要提前离开, 将填写完整的意见反馈表投到签到处的抽奖箱内,
即可参与“iPhone5”抽奖活动。



Birds of a Feather 专家面对面

所有讲师都会在课程结束后，到紫兰厅与来宾讨论课程上的问题

