学用 JavaScript 设计模式

阐述 JavaScript 编程语言中经典和现代的设计模式

极客学院出版

前言

设计模式是可重用的用于解决软件设计中一般问题的方案。设计模式如此让人着迷，以至在任何编程语言中都有对其进行的探索。

其中一个原因是它可以让我们站在巨人的肩膀上，获得前人所有的经验，保证我们以优雅的方式组织我们的代码，满足我们解决问题所需要的条件。

设计模式同样也为我们描述问题提供了通用的词汇。这比我们通过代码来向别人传达语法和语义性的描述更为方便。

本文我们将阐述 JavaScript 编程语言中经典和现代的设计模式。

适用人群

本书适用那些期望提高自己在设计模式方面的知识并将它们应用到 JavaScript 编程语言中的专业开发者。

学习前提

本书同时面向初学者和中级开发者，因此假设读者已掌握 Javascript 的基本知识，对闭包，原型继承等思想有一定的了解。

鸣谢：作者：史涛 (http://www.oschina.net/translate/learning-javascript-design-patterns)

目录

1 设计模式

简介

写出可维护的代码的一个最重要的方面就是在代码中能够注意到重复出现的主题并对其进行优化。设计模式的知识领域是无价的。

在本书的第一部分，我们将探索那些真正可以应用于任何编程语言的设计模式的历史和重要性。如果你已经熟悉这段历史，可以直接跳过"什么是模式？" (what.md) 这一章继续阅读。

设计模式可以追溯到早期的一名叫Christopher Alexander的建筑师。他经常会发表一些他在处理设计问题时的经验和如何与建筑和城镇相联系的。有一天，当Alexander使用了一次又一次后，他发现某些设计结构会导致做出的效果是最好的。

在Sara Ishikawa和Murray Silverstein的协作下，Alexander发明了一种可以帮助授权任何人去设计和构建希望的任何规模的模式语言。这在1977年的一篇名为"A Pattern Language"的论文中发表，在后来作为一本完整的精装书发表。

大约30年前，软件工程师开始将Alexander曾写过的原理并入第一版的设计模式，这是一个用来对那些想要改善他们编码技巧的新手开发者的一个指南。要注意，这时设计模式背后的概念实际上已经在编程行业成立以来就有了，虽然不是那么正式的形式。

第一个也是最标志性的关于软件工程的设计模式的正式作品是在1995年一本叫Design Patterns: Elements Of Reusable Object-Oriented Software的书中发表，这是Erich Gamma, Richard Helm, Ralph Johnson和 John Vlissides – 一群被称为Gang of Four(简称GoF)的人写的。

GoF的出版物被认为是非常有助于推动设计模式的概念在我们的领域发展的，因为它描述了大量的开发技术和缺陷，而且还有在今天的世界中大量使用的23个核心的面向对象的设计模式。我们将详细地在"设计模式分类"这一章中介绍这些模式。

在本书中，我们将看到一些流行的JavaScript设计模式，并探索为什么一些特定的模式比其他的更适合你的项目。但请记住模式不仅仅可以应用在单纯的JavaScript (例如：标准JavaScript代码)里，也可以在一些像jQuery或dojo的抽象库里使用。在我们开始之前，让我们看看模式在软件设计中的确切定义。

什么是设计模式

一个模式就是一个可重用的方案，可应用于在软件设计中的常见问题 – 在我们的例子里 – 就是编写JavaScript的web应用程序。模式的另一种解释就是一个我们如何解决问题的模板 – 那些可以在许多不同的情况里使用的模板。 那么理解和熟悉模式为什么是如此的重要？设计模式有以下三点好处：

- 模式是行之有效的解决方法：他们提供固定的解决方法来解决在软件开发中出现的问题，这些都是久经考验的反应了开发者的经验和见解的使用模式来定义的技术。
- 模式可以很容易地重用：一个模式通常反映了一个可以适应自己需要的开箱即用的解决方案。这个特性让它们很健壮。
- 模式善于表达：当我们看到一个提供某种解决方案的模式时，一般有一组结构和词汇可以非常优雅地帮助表达相当大的解决方案。

模式不是一个确切的解决方案。我们要记住模式的角色仅仅是给我们提供一个解决方案。模式不能解决所有的设计问题，也不能代替优秀的软件设计师。然而，它们在帮助我们。接下来我们将看看模式必须提供的其他的一些优势。

- **模式的重用可以帮助防止在应用程序开发过程中出现的一些可能导致重大问题的小问题。**这意味着当代码是建立在行之有效的模式上时，我们可以花更少的时间去关心我们的代码结构，从而能花更多的时间关注我们的解决方案的整体质量。这是因为模式可以鼓励我们在更好的结构化和有组织的方式下编码，这将避免在未来由于清洁的目的而去重构它。
- **模式可以提供一个不需要绑定到一个特定问题的书面的概括性的解决方案。**这个广义的方法意味着不用管我们正在处理的应用程序 (许多情况下的编程语言) 设计模式的应用可以提高我们的代码的结构。
- **某些模式可以通过避免重复来减小我们代码的文件大小。**通过鼓励开发者更仔细地看待他们的解决方案来减少重复的地方，如通过将类似的执行流程作为一个一般性的函数来减少函数的数量，这样我们就可以减小代码库的总体大小，这也成为使代码更DRY。
- **模式增加了开发者的词汇，这使得交流更快速。**
- **经常使用的模式可通过收集其他使用这些模式的开发人员贡献给设计模式社区的经验来改进。**在某些情况下，这将导致全新模式的创建，同时也可以提供改进的指导大家如何使用特定的模式才是最好的。这可以确保基于模式的解决方案继续变得比特别的解决方案更健壮。

我们已经每天都在使用模式

为了了解模式有多有用，让我们看看jQuery提供给我们的一个很简单的元素选择问题。 假设我们有一个为页面上每一个class为"foo"的DOM元素添加一个计数器的脚本，什么才是查询这个元素的集合的最有效的方法呢？ 有几种不同的方法可以解决这个问题：

选择页面上所有的元素并存储它们的引用，然后使用正则表达式 (或其他方式) 来过滤这个集合中那些class为"foo"的元素的引用。

- 使用像asquerySelectorAll()的现代原生浏览器的特性，来选择所有的class为"foo"的元素。
- 使用像asgetElementsByClassName()的原生特性同样可以获取期望的集合。

那些，这些选择哪个是最快的呢？ 实际上第三个，比其他的替代选择快 8–10倍。但在实际的应用程序中，第三个选择无法在Internet Explorer 9以下的版本中使用，从而只能使用第一个，第二个和第三个都不支持。

使用jQuery的开发人员就不必担心这个问题，因为很幸运的是它使用Facade模式把这个问题抽象了出来。正如我们即将在后面更详细的介绍的那样，这种模式提供了一组简单的对更复杂的底层代码的抽象接口 (例如$el.css(),$el.animate()) 。正如我们所看到的，这意味着我们只会对实现级别的细节花费更少的时间。

在其后，库会根据我们当前浏览器的支持自动选择最优的方法来选择元素，我们只使用抽象层。

我们可能都熟悉jQuery的$("selector")，这是更容易使用的在一个页面选择HTML元素的方法，这样我们就不必手动来选择getElementById(),getElementsByClassName(),getElementByTagName()等方法。

虽然我们知道querySelectorAll()试图解决这个问题，但比比使用jQuery的Facade接口和自己来选择最优的方式时花费的精力，毫无疑问，使用模式可以提供真实世界的抽象价值。

我们将在本书的后面看到更多的设计模式。

“模式特性”测试，模式原型和三条规则

记住并不是每个算法、每个最佳实践和每个解决方案都可能被认为是一个完整的模式。这儿可能缺少了几个关键因素，而且模式社团除非经过严格的审查才谨慎地声明某东西为模式的。即使某东西对我们来说似乎满足了模式标准，它都不应该被当作模式，直到它由他人经过适当时间的周密调查和测试后才可能当作模式。

回头看看Alexander曾经做过的工作，他声明模式应当既是过程也是“事物”。这个定义故意不明确，因为他紧跟着说模式应该是创建“事物”的过程。这就是为什么模式通常集中定位在表面上可识别的结构的原因。例如，我们应当能够可视化地描绘（或者绘制）图片来展示把模式应用到实践中的结构。

在研究设计模式的时候，无意间碰到术语“模式原型”是很正常的。那么什么是模式原型呢？ 好，仍然没有通过"模式特性”测试的模式通常认为是模式原型。模式原型也许源自于某人已经确定的值得与社团共享的特定解决方案的工作，然而由于它提出时间短，所以可能仍然没有机会接受严格的审查。

另外，个人共享的模式也许没有时间或者没有兴趣通过“模式特性”测试这个过程，不过可能发布了这些模式原型的简短说明。这种类型模式的简要描述或者片段就是众所周知的小模式。

全面文档化具有资格的模式这样的工作是非常令人气馁的。回头看看设计模式领域最早期的某些工作，如果一个模式能做到以下事情，那么这个模式就可以认为是“好的”模式：

- 解决一类特定的问题：模式不能假设仅仅关注原理或者策略。它们需要关注解决方案。这是好的模式最重要的因素之一。
- 问题的解决方案不是表面上的：我们发现解决问题的技术常常首先试图源自于某个众所周知的原理。最佳的设计模式通常间接地提供问题的解决方案–认为模式是与设计相关的最具有挑战性的问题必然的解决方法。
- 所描述的想法一定得到了证明：设计模式需要提供所描述的它们运行的证据，如果没有这些证据，就不会认真的考虑这个设计。如果模式事实上是高度理论性的话，那么只有冒险者才可能试着用它。
- 它必须说明与代码之间关系：在某些情况下，模式似乎说明了一种类型的模块。虽然实现的可能就是这个方法，但是官方的模式说明一定要更深入的描述系统结构和机制，以解释它与代码之间的关系。

我们认为不满足准则的模式原型不值得学习，这可以得到谅解，然而，事实远不是这样的。许多模式原型确实非常的好。我不是说所有的模式原型都值得看，不过总有几个在自然环境下成长的有用的模式原型可以在未来的项目中帮到我们。从心底里使用上面列表来做最佳评判的话，你在选择哪个是模式的过程中将感觉非常愉快。

模式是否有效的附加要求之一是模式要展示某些重现现象。这个就是至少在三个关键方面 ，也就是三条规则验证是否取得资格经常要做的事情。为了展示使用这个规则后的重现，模式必须证明其：

- **适用性**–模式怎样才能被认为是成功的。
- **有用性**–为什么认为这个模式是成功的?
- **可用性** –因为设计得到广泛的应用，所以认为这个设计就是模式吗？如果是这样的话，那么需要说明。重新审核或者定义模式的时候，牢记以上规则非常重要。

设计模式的结构

你可能会对设计模式的作者如何接近勾勒出概念轮廓，实施和新模式的目的。模式是最初提出的一种在两者之间建立关系的规则：

- 上下文环境。
- 在这种环境下产生的系统的力量。
- 一类配置，考虑到允许这种力量在自己的上下文环境中解决这一点，现在让我们对一种设计模式的组件元素，一探究竟。一种设计模式应该具有。
- 模式名称和相应的描述。
- 上下文概述–在设计模式中的上下文对响应用户需求是很有效的。
- 问题声明–一类问题的声明，能让我们理解模式的意图。
- 解决方案–在可理解的列表和看法上，对用户的问题如何被解决的一种描述。
- 设计–模式设计，特别是与之交互的用户行为的描述。
- 实现–对模式如何被实现的一种指引。
- 例证–在模式中的一种类的虚拟化表示。
- 例子–模式实现的一种最下的形式。
- 共同条件–可能会有其他的什么模式会被用到，以对被描述的模式进行支持?
- 关系–与该模式相似的模式有哪些？是最相似的吗?
- 已知的使用–模式没有被正常使用？如果是，在哪，怎样做到的?
- 讨论–有激动人心的获利模式想法的团队或者是作者。

在一个组织或团队中，当在同一页面上创建和维护的解决方案时，对所有涉及到的开发者来说，设计模式能帮上大忙。如果考虑到你自己的工作模式，记住，虽然他们可能在制定计划和编写阶段，有一个较大的初期成本投入，但从投资方返回的值是值得的。然而，新的模式工作前，务必深入研究，你会发现它比起重新开始，更有利于使用或建立比现有的行之有效的模式之上。

编写设计模式

虽然本书的目标，针对的是新的设计模式，但对设计模式是怎样编写的有一个根本的理解后，会让我们受益匪浅。对于初学者来说，对于为什么需要一个模式背后的推理，我们可以得到更深的理解。我们同时也会学习到当我们在重视我们自己的需求的时候，如何区分一种模式（或原模式）。

要编写好的模式，是一种极具挑战性的任务。模式不仅仅需要对终端用户提供数量可观的材料，还要能够说明为什么需要这种模式。

在读过前续章节–什么是模式以后，我们可能会认为足够帮助我们去辨别我们在非标准条件下看到的模式。事实上这并非完全正确。这并不总是很清楚，如果我们正在寻找的一段代码，出现像它一样符合的一组模式，或只是偶然发生。

当我们在寻找认为可能使用某种设计模式的代码的时候，应该考虑写下的代码的一些方面，我们相信属于一个特定的现有格局或一组模式。

在很多模式分析的案例中，我们会发现，正巧看到了那些具有良好的原则和设计实践，而这些可能突然引起对模式的覆盖规则。记住–既不相互作用，也没有定义规则的解决方案模式。

如果敢于尝试编写自己的设计模式的道路，我推荐从其他那些已经过来之人学习，学习他们好的方面。花时间从大量不同的设计模式描述中吸取信息，并找到对你有意义的。

探索结构和语义–可以通过检查交互和你感兴趣的模式的上下文，因此你可以标示出运用有用的配置，将模式组织在一起的原则。

一旦我们暴露了自己丰富的模式文献资料，我们不妨使用现有的格式，开始写我们的模式，并看看我们是否能集思广益，打开新思路，对它进行改进或把我们的想法进行整合。

一个开发者的例子，该例子的作者是近几年的Christian Heilmann，他在对已存在的模式的基础上做了一些基本的改变，以此创建了暴露模块模式（该模式在本书后续部分会讲到）。

对于那些对创建新设计模式的人，我对他们有如下的建议：

- **模式是否实用?**：确保这个模式能够对一些常见的问题有明确的解决方案，而不是临时的解决方案。
- **保持最佳实践**: 我们的设计需要以最佳实践中所获得的理解作为基础。
- **设计模式对用户来说应该是清晰的**: 设计模式必须对任何形式的用户体验都是清晰的。 因为设计模式主要服务于开发者们，所以不能强迫他们去改变原来的行为，那样开发者们才会去使用这个模式。

- **独创力不是设计模式的关键**：当我们在设计一个模式的时候，我们既不需要是发明者，也不需要去担心是否是其他模式的子集。如果某个想法有很强的实用性，那么这就是一个创造新模式的机会。
- **需要有几个有说服力的例子**：一个好的设计模式需要有一个有说服力的例子来展示这个模式是成功的。为了广泛使用这个设计模式，这些例子需要展示良好的设计原则。

在创造一个新的设计模式的时候，在通用性，特殊性和可用性之间有一个微妙的平衡点。如果新的模式覆盖了应用中最多的可能情况，那么这个模式应该是良好的。我希望通过这段简介能够对下个章节内容的学习有所帮助。

反模式

如果我们认为模式代表一个最佳的实践，那么反模式将代表我们已经学到一个教训。受启发于Gof的《设计模式》，Andrew Koeing在1995年的11月的C++报告大会上首次提出反模式。在Koeing的报告中，反模式有着两种观念:

- 描述对于一个特殊的问题，提出了一个糟糕的解决方案，最终导致一个坏结果发生
- 描述如何摆脱上述解决方案并能提出一个好的解决方案

关于这个话题，Alexander写过要在好的设计结构和好的上下文中找到平衡是困难的:

> 这些笔记是关于设计的过程，这个过程发明显示一个新的物理顺序响应功能，组织形式，物质的东西.....每一个设计问题开始于努力实现两个实体之间的形式：问题中的形式和它的上下文。此形式是解决问题的方法，而上下文定义了该问题。

虽然理解设计模式很重要，但对于理解反模式也是同等重要。我们有资格知道这背后的原因。当我们开发一个应用，这个工程的生命周期开始建设一直至项目完成，但一旦完成后，就进入维护阶段。判断一个解决方案的好坏要看这个团队在这个项目上投资的技术和花费的时间。这里被认为是好的和坏的情况下-如果应用在错误的情况下，一个“完美”的设计可能有资格作为一个反模式。

最大的挑战发生于应用进入生产和维护阶段。一个之前没有开发过这个应用的开发者来维护一个系统可能会引进糟糕的设计。如果说糟糕的设计是因为反模式，那么将允许开发者提前找到一种认识到时这样的手段，这样就能避免一些普通错误的发生，与此同时这也是设计模式给我们提供一种认识到普通技术也是有用的方式。

反模式是一个值得为此专门编写编写总结文档的糟糕设计。Javascript的反模式例子如下:

- 在全局上面文中定义大量污染全局命令空间的变量
- 在调用setTimeout和setInterval时传递字符串(会用eval来执行)而不是函数。
- 修改Object的原型 (这是最糟糕的反模式)
- 使用内联Javascript

在本应使用document.createElement的地方使用document.write。document.write被错误的用了相当多的年头，它有相当多的缺点，包括如果在页面加载后执行它可能会覆盖我们的页面。再有它不能工作在XHTML下，这也是另外一个我们使用像document.createElement这种对DOM友好方法的原因。

知道反模式对成功来说很关键。一旦我们能识别这些反模式，我们就能够重构我们的代码使项目的整体质量立马提升。

设计模式的分类

设计模式的种类

在众所周知的设计书《 Domain-Driven Terms 》中,它被描述为:

“设计模式是命名、抽象和识别对可重用的面向对象设计有用的的通用设计结构。设计模式确定类和他们的实体、他们的角色和协作、还有他们的责任分配。

每一个设计模式都聚焦于一个面向对象的设计难题或问题。它描述了在其它设计的约束下它能否使用，使用它后的后果和得失。因为我们必须最终实现我们的设计模式，所以每个设计模式都提供了例子..代码来对实现进行阐释。

虽然设计模式被描述为面向对象的设计，它们基于那些已经被主流面向对象语言实现过的解决方案...”

设计模式可以被分成几个不同的种类。在这个部分我们将复习三个分类，并且在我们进入特定的设计模式详情之前我们提到该分组下的模式的几个示例。

创建型设计模式

创建型设计模式关注于对象创建的机制方法，通过该方法,对象以适应工作环境的方式被创建。基本的对象创建方法可能会给项目增加额外的复杂性，而这些模式的目的就是为了通过控制创建过程解决这个问题。

属于这一类的一些模式是：构造器模式（ Constructor ）,工厂模式（ Factory ）,抽象工厂模式 （ Abstract ）,原型模式 （ Prototype ）,单例模式 （ Singleton ）以及 建造者模式（ Builder ）。

结构设计模式

结构模式关注于对象组成和通常识别的方式实现不同对象之间的关系。该模式有助于在系统的某一部分发生改变的时候，整个系统结构不需要改变。该模式同样有助于对系统中某部分没有达到某一目的的部分进行重组。

在该分类下的模式有：装饰模式，外观模式，享元模式，适配器模式和代理模式。

行为设计模式

行为模式关注改善或精简在系统中不同对象间通信。

行为模式包括：迭代模式，中介者模式，观察者模式和访问者模式。

设计模式的分类

在我早起学习设计模式的经验中，我个人发现，下面的表格是一个非常有用的提醒，大多数模式所提供-它覆盖了由GOF提出的23种模式。最早的表格由 Elyse Nielsen 在2004年汇总，我已经做了部分修改以适应我们的讨论。 我推荐使用该表格作为参考，但要记住大量额外的模式在这里么有提及，但在本书的后续的章节中会提到。

关于类的简单说明

要记住这张表中会有模式引用“类”的概念。JavaScript是一种弱类型语言，不过类可以通过函数模拟出来。 最常见的实现这一点的方法，是先定义一个JavaScript函数，然后再使用这个新的关键字创建一个对象。可以通过这种方法像下面这样给类定义新的属性与方法。

```
// A car "class"
function Car( model ) {

  this.model = model;
  this.color = "silver";
  this.year  = "2012";

  this.getInfo = function () {
    return this.model + " " + this.year;
  };

}
```

接着我们可以使用上面定义的Car构造函数实例化对象，就像这样：

```
var myCar = new Car("ford");

myCar.year = "2010";

console.log( myCar.getInfo() );
```

更多使用JavaScript定义“类”的方法，参见Stoyan Stefanov的关于这些的有用帖子 (http://www.phpied.com/3-ways-to-define-a-javascript-class/) 。

设计模式分类概览表

现在让我们看看这个表格。

SN	描述
Creational	根据创建对象的概念分成下面几类。
Class	
Factory Method(工厂方法)	通过将数据和事件接口化来构建若干个子类。
Object	
Abstract Factory(抽象工厂)	建立若干族类的一个实例，这个实例不需要具体类的细节信息。（ 抽象类 ）
Builder (建造者)	将对象的构建方法和其表现形式分离开来，总是构建相同类型的对象。
Prototype(原型)	一个完全初始化的实例，用于拷贝或者克隆。
Singleton(单例)	一个类只有唯一的一个实例，这个实例在整个程序中有一个全局的访问点。
Structural	根据构建对象块的方法分成下面几类。
Class	
Adapter(适配器)	将不同类的接口进行匹配，调整，这样尽管内部接口不兼容但是不同的类还是可以协同工作的。
Bridge(桥接模式)	将对象的接口从其实现中分离出来，这样对象的实现和接口可以独立的变化。
Composite(组合模式)	通过将简单可组合的对象组合起来，构成一个完整的对象，这个对象的能力将会超过这些组成部分的能力的总和，即会有新的能力产生。
Decorator(装饰器)	动态给对象增加一些可替换的处理流程。
Facada(外观模式)	一个类隐藏了内部子系统的复杂度，只暴露出一些简单的接口。
Flyweight(享元模式)	一个细粒度对象，用于将包含在其它地方的信息 在不同对象之间高效地共享。
Proxy(代理模式)	一个充当占位符的对象用来代表一个真实的对象。
Behavioral	基于对象间作用方式来分类。
Class	
Interpreter(解释器)	将语言元素包含在一个应用中的一种方式，用于匹配目标语言的语法。
Template Method(模板方法)	在一个方法中为某个算法建立一层外壳，将算法的具体步骤交付给子类去做。
Object	
Chain of Responsibility(响应链)	一种将请求在一串对象中传递的方式，寻找可以处理这个请求的对象。
Command(命令)	封装命令请求为一个对象，从而使记录日志，队列缓存请求，未处理请求进行错误处理这些功能称为可能。
Iterator(迭代器)	在不需要直到集合内部工作原理的情况下，顺序访问一个集合里面的元素。
Mediator(中介者模式)	在类之间定义简化的通信方式，用于避免类之间显式的持有彼此的引用。

SN	描述
Observer(观察者模式)	用于将变化通知给多个类的方式，可以保证类之间的一致性。
State(状态)	当对象状态改变时，改变对象的行为。
Strategy(策略)	将算法封装到类中，将选择和实现分离开来。
Visitor(访问者)	为类增加新的操作而不改变类本身。

2

JavaScript 设计模式

构造器模式

在面向对象编程中，构造器是一个当新建对象的内存被分配后，用来初始化该对象的一个特殊函数。在JavaScript中几乎所有的东西都是对象，我们经常会对对象的构造器十分感兴趣。

对象构造器是被用来创建特殊类型的对象的，首先它要准备使用的对象，其次在对象初次被创建时，通过接收参数，构造器要用来对成员的属性和方法进行赋值。

对象创建

下面是我们创建对象的三种基本方式:

下面的每一种都会创建一个新的对象:

```
var newObject = {};

// or
var newObject = Object.create( null );

// or
var newObject = new Object();
```

最后一个例子中"Object"构造器创建了一个针对特殊值的对象包装，只不过这里没有传值给它，所以它将会返回一个空对象。

有四种方式可以将一个键值对赋给一个对象:

```
// ECMAScript 3 兼容形式

// 1. “点号”法

// 设置属性
newObject.someKey = "Hello World";

// 获取属性
var key = newObject.someKey;



// 2. “方括号”法
```

```
// 设置属性
newObject["someKey"] = "Hello World";

// 获取属性
var key = newObject["someKey"];



// ECMAScript 5 仅兼容性形式
// For more information see: http://kangax.github.com/es5-compat-table/

// 3. Object.defineProperty方式

// 设置属性
Object.defineProperty( newObject, "someKey", {
    value: "for more control of the property's behavior",
    writable: true,
    enumerable: true,
    configurable: true
});

// 如果上面的方式你感到难以阅读，可以简短的写成下面这样：

var defineProp = function ( obj, key, value ){
  config.value = value;
  Object.defineProperty( obj, key, config );
};

// 为了使用它，我们要创建一个“person”对象
var person = Object.create( null );

// 用属性构造对象
defineProp( person, "car",  "Delorean" );
defineProp( person, "dateOfBirth", "1981" );
defineProp( person, "hasBeard", false );



// 4. Object.defineProperties方式

// 设置属性
Object.defineProperties( newObject, {

  "someKey": {
    value: "Hello World",
    writable: true
```

```
  },

  "anotherKey": {
    value: "Foo bar",
    writable: false
  }

});

// 3和4中的读取属行可用1和2中的任意一种
```

在这本书的后面一点，这些方法会被用于继承，如下：

```
// 使用:

// 创建一个继承与Person的赛车司机
var driver = Object.create( person );

// 设置司机的属性
defineProp(driver, "topSpeed", "100mph");

// 获取继承的属性 (1981)
console.log( driver.dateOfBirth );

// 获取我们设置的属性 (100mph)
console.log( driver.topSpeed );
```

基础构造器

正如我们先前所看到的，Javascript不支持类的概念，但它有一种与对象一起工作的构造器函数。使用new关键字来调用该函数，我们可以告诉Javascript把这个函数当做一个构造器来用,它可以用自己所定义的成员来初始化一个对象。

在这个构造器内部，关键字this引用到刚被创建的对象。回到对象创建，一个基本的构造函数看起来像这样:

```
function Car( model, year, miles ) {

  this.model = model;
  this.year = year;
  this.miles = miles;

  this.toString = function () {
    return this.model + " has done " + this.miles + " miles";
  };
```

```
}

// 使用:

// 我们可以示例化一个Car
var civic = new Car( "Honda Civic", 2009, 20000 );
var mondeo = new Car( "Ford Mondeo", 2010, 5000 );

// 打开浏览器控制台查看这些对象toString()方法的输出值
// output of the toString() method being called on
// these objects
console.log( civic.toString() );
console.log( mondeo.toString() );
```

上面这是个简单版本的构造器模式，但它还是有些问题。一个是难以继承，另一个是每个Car构造函数创建的对象中，toString()之类的函数都被重新定义。这不是非常好，理想的情况是所有Car类型的对象都应该引用同一个函数。 这要谢谢 ECMAScript3和ECMAScript5-兼容版，对于构造对象他们提供了另外一些选择，解决限制小菜一碟。

使用“原型”的构造器

在Javascript中函数有一个prototype的属性。当我们调用Javascript的构造器创建一个对象时，构造函数prototype上的属性对于所创建的对象来说都看见。照这样，就可以创建多个访问相同prototype的Car对象了。下面，我们来扩展一下原来的例子：

```
function Car( model, year, miles ) {

  this.model = model;
  this.year = year;
  this.miles = miles;

}


// 注意这里我们使用Note here that we are using Object.prototype.newMethod 而不是
// Object.prototype ，以避免我们重新定义原型对象
Car.prototype.toString = function () {
  return this.model + " has done " + this.miles + " miles";
};

// 使用:

var civic = new Car( "Honda Civic", 2009, 20000 );
```

```
var mondeo = new Car( "Ford Mondeo", 2010, 5000 );

console.log( civic.toString() );
console.log( mondeo.toString() );
```

通过上面代码，单个toString()实例被所有的Car对象所共享了。

模块化模式

模块

模块是任何健壮的应用程序体系结构不可或缺的一部分，特点是有助于保持应用项目的代码单元既能清晰地分离又有组织。

在JavaScript中，实现模块有几个选项，他们包括：

- 模块化模式
- 对象表示法
- AMD模块
- CommonJS 模块
- ECMAScript Harmony 模块

我们在书中后面的现代模块化JavaScript设计模式章节中将探讨这些选项中的最后三个。

模块化模式是基于对象的文字部分，所以首先对于更新我们对它们的知识是很有意义的。

对象字面值

在对象字面值的标记里，一个对象被描述为一组以逗号分隔的名称/值对括在大括号（{}）的集合。对象内部的名称可以是字符串或是标记符后跟着一个冒号":"。在对象里最后一个名称/值对不应该以","为结束符，因为这样会导致错误。

```
var myObjectLiteral = {

    variableKey: variableValue,

    functionKey: function () {
      // ...
    };
};
```

对象字面值不要求使用新的操作实例，但是不能够在结构体开始使用，因为打开"{"可能被解释为一个块的开始。在对象外新的成员会被加载，使用分配如下：smyModule.property = "someValue"; 下面我们可以看到一个更完整的使用对象字面值定义一个模块的例子：

```
var myModule = {

  myProperty: "someValue",

  // 对象字面值包含了属性和方法（properties and methods）.
  // 例如，我们可以定义一个模块配置进对象：
  myConfig: {
    useCaching: true,
    language: "en"
  },

  // 非常基本的方法
  myMethod: function () {
    console.log( "Where in the world is Paul Irish today?" );
  },

  // 输出基于当前配置（<span>configuration</span>）的一个值
  myMethod2: function () {
    console.log( "Caching is:" + ( this.myConfig.useCaching ) ? "enabled" : "disabled" );
  },

  // 重写当前的配置（configuration）
  myMethod3: function( newConfig ) {

    if ( typeof newConfig === "object" ) {
      this.myConfig = newConfig;
      console.log( this.myConfig.language );
    }
  }
};

// 输出: Where in the world is Paul Irish today?
myModule.myMethod();

// 输出: enabled
myModule.myMethod2();

// 输出: fr
myModule.myMethod3({
  language: "fr",
  useCaching: false
});
```

使用对象字面值可以协助封装和组织你的代码。如果你想近一步了解对象字面值可以阅读 Rebecca Murphey 写过的关于此类话题的更深入的文章(depth) (http://rmurphey.com/blog/2009/10/15/using-objects-to-organize-your-code/) 。

也就是说，如果我们选择了这种技术，我们可能对模块模式有同样的兴趣。即使使用对象字面值，但也只有一个函数的返回值。

模块化模式

模块化模式最初被定义为一种对传统软件工程中的类提供私有和公共封装的方法。

在JavaScript中，模块化模式用来进一步模拟类的概念，通过这样一种方式：我们可以在一个单一的对象中包含公共/私有的方法和变量，从而从全局范围中屏蔽特定的部分。这个结果是可以减少我们的函数名称与在页面中其他脚本区域定义的函数名称冲突的可能性。

私有信息

模块模式使用闭包的方式来将"私有信息",状态和组织结构封装起来。提供了一种将公有和私有方法，变量封装混合在一起的方式，这种方式防止内部信息泄露到全局中，从而避免了和其它开发者接口发生冲图的可能性。在这种模式下只有公有的API 会返回，其它将全部保留在闭包的私有空间中。

这种方法提供了一个比较清晰的解决方案，在只暴露一个接口供其它部分使用的情况下，将执行繁重任务的逻辑保护起来。这个模式非常类似于立即调用函数式表达式(IIFE-查看命名空间相关章节获取更多信息)，但是这种模式返回的是对象，而立即调用函数表达式返回的是一个函数。

需要注意的是，在javascript事实上没有一个显式的真正意义上的"私有性"概念，因为与传统语言不同，javascript没有访问修饰符。从技术上讲，变量不能被声明为公有的或者私有的，因此我们使用函数域的方式去模拟这个概念。在模块模式中，因为闭包的缘故，声明的变量或者方法只在模块内部有效。在返回对象中定义的变量或者方法可以供任何人使用。

历史

从历史角度来看，模块模式最初是在2003年由一群人共同发展出来的，这其中包括Richard Cornford。后来通过Douglas Crockford的演讲，逐渐变得流行起来。另外一件事情是，如果你曾经用过雅虎的YUI库，你会看到其中的一些特性和模块模式非常类似，而这种情况的原因是在创建YUI框架的时候，模块模式极大的影响了YUI的设计。

例子

下面这个例子通过创建一个自包含的模块实现了模块模式。

```
var testModule = (function () {

  var counter = 0;

  return {

    incrementCounter: function () {
      return counter++;
    },

    resetCounter: function () {
      console.log( "counter value prior to reset: " + counter );
      counter = 0;
    }
  };

})();

// Usage:

// Increment our counter
testModule.incrementCounter();

// Check the counter value and reset
// Outputs: 1
testModule.resetCounter();
```

在这里我们看到，其它部分的代码不能直接访问我们的incrementCounter() 或者 resetCounter()的值。counter变量被完全从全局域中隔离起来了，因此其表现的就像一个私有变量一样，它的存在只局限于模块的闭包内部，因此只有两个函数可以访问counter。我们的方法是有名字空间限制的，因此在我们代码的测试部分，我们需要给所有函数调用前面加上模块的名字(例如"testModule")。

当使用模块模式时，我们会发现通过使用简单的模板，对于开始使用模块模式非常有用。下面是一个模板包含了命名空间，公共变量和私有变量。

```
var myNamespace = (function () {

  var myPrivateVar, myPrivateMethod;

  // A private counter variable
  myPrivateVar = 0;

  // A private function which logs any arguments
  myPrivateMethod = function( foo ) {
      console.log( foo );
```

```
  };

  return {

    // A public variable
    myPublicVar: "foo",

    // A public function utilizing privates
    myPublicFunction: function( bar ) {

      // Increment our private counter
      myPrivateVar++;

      // Call our private method using bar
      myPrivateMethod( bar );

    }
  };

})();
```

看一下另外一个例子，下面我们看到一个使用这种模式实现的购物车。这个模块完全自包含在一个叫做basketModule 全局变量中。模块中的购物车数组是私有的，应用的其它部分不能直接读取。只存在与模块的闭包中，因此只有可以访问其域的方法可以访问这个变量。

```
var basketModule = (function () {

  // privates

  var basket = [];

  function doSomethingPrivate() {
    //...
  }

  function doSomethingElsePrivate() {
    //...
  }

  // Return an object exposed to the public
  return {

    // Add items to our basket
    addItem: function( values ) {
      basket.push(values);
```

```
  },

  // Get the count of items in the basket
  getItemCount: function () {
    return basket.length;
  },

  // Public alias to a  private function
  doSomething: doSomethingPrivate,

  // Get the total value of items in the basket
  getTotal: function () {

    var q = this.getItemCount(),
        p = 0;

    while (q--) {
      p += basket[q].price;
    }

    return p;
  }
 };
}());
```

在模块内部，你可能注意到我们返回了应外一个对象。这个自动赋值给了basketModule 因此我们可以这样和这个对象交互。

```
// basketModule returns an object with a public API we can use

basketModule.addItem({
 item: "bread",
 price: 0.5
});

basketModule.addItem({
 item: "butter",
 price: 0.3
});

// Outputs: 2
console.log( basketModule.getItemCount() );

// Outputs: 0.8
console.log( basketModule.getTotal() );
```

```
// However, the following will not work:

// Outputs: undefined
// This is because the basket itself is not exposed as a part of our
// the public API
console.log( basketModule.basket );

// This also won't work as it only exists within the scope of our
// basketModule closure, but not the returned public object
console.log( basket );
```

上面的方法都处于basketModule 的名字空间中。

请注意在上面的basket模块中 域函数是如何在我们所有的函数中被封装起来的，以及我们如何立即调用这个域函数，并且将返回值保存下来。这种方式有以下的优势：

- 可以创建只能被我们模块访问的私有函数。这些函数没有暴露出来（只有一些API是暴露出来的），它们被认为是完全私有的。
- 当我们在一个调试器中，需要发现哪个函数抛出异常的时候，可以很容易的看到调用栈，因为这些函数是正常声明的并且是命名的函数。
- 正如过去 T.J Crowder 指出的，这种模式同样可以让我们在不同的情况下返回不同的函数。我见过有开发者使用这种技巧用于执行UA（尿检，抽样检查）测试，目的是为了在他们的模块里面针对IE专门提供一条代码路径，但是现在我们也可以简单的使用特征检测达到相同的目的。

模块模式的变体

Import mixins(导入混合)

这个变体展示了如何将全局（例如 jQuery, Underscore）作为一个参数传入模块的匿名函数。这种方式允许我们导入全局，并且按照我们的想法在本地为这些全局起一个别名。

```
// Global module
var myModule = (function ( jQ, _ ) {

  function privateMethod1(){
    jQ(".container").html("test");
  }

  function privateMethod2(){
   console.log( _.min([10, 5, 100, 2, 1000]) );
```

```
    }

    return{
        publicMethod: function(){
            privateMethod1();
        }
    };

// Pull in jQuery and Underscore
}( jQuery, _ ));

myModule.publicMethod();
```

Exports（导出）

这个变体允许我们声明全局对象而不用使用它们，同样也支持在下一个例子中我们将会看到的全局导入的概念。

```
// Global module
var myModule = (function () {

    // Module object
  var module = {},
    privateVariable = "Hello World";

  function privateMethod() {
    // ...
  }

  module.publicProperty = "Foobar";
  module.publicMethod = function () {
    console.log( privateVariable );
  };

  return module;

}());
```

工具箱和框架特定的模块模式实现。

Dojo

Dojo提供了一个方便的方法 dojo.setObject() 来设置对象。这需要将以"."符号为第一个参数的分隔符，如：myObj.parent.child 是指定义在"myOjb"内部的一个对象“parent”，它的一个属性为"child"。使用setObject()方法允许我们设置children 的值，可以创建路径传递过程中的任何对象即使这些它们根本不存在。

例如，如果我们声明商店命名空间的对象basket.coreas，可以实现使用传统的方式如下：

```
var store = window.store || {};

if ( !store["basket"] ) {
  store.basket = {};
}

if ( !store.basket["core"] ) {
  store.basket.core = {};
}

store.basket.core = {
  // ...rest of our logic
};
```

或使用Dojo1.7（AMD兼容的版本）及以上如下：

```
require(["dojo/_base/customStore"], function( store ){

  // using dojo.setObject()
  store.setObject( "basket.core", (function() {

     var basket = [];

     function privateMethod() {
        console.log(basket);
     }

     return {
        publicMethod: function(){
              privateMethod();
        }
     };

  }()));
```

```
});
```

欲了解更多关于dojo.setObject（ ）方法的信息，请参阅官方文档 documentation (http://dojotoolkit.org/reference-guide/1.7/dojo/setObject.html)

ExtJS

对于这些使用Sencha的ExtJS的人们，你们很幸运，因为官方文档包含一些例子，用于展示如何正确地在框架里面使用模块模式。

下面我们可以看到一个例子关于如何定义一个名字空间，然后填入一个包含有私有和公有API的模块。除了一些语义上的不同之外，这个例子和使用vanilla javascript 实现的模块模式非常相似。

```
// create namespace
Ext.namespace("myNameSpace");

// create application
myNameSpace.app = function () {

  // do NOT access DOM from here; elements don't exist yet
  // private variables

  var btn1,
     privVar1 = 11;

  // private functions
  var btn1Handler = function ( button, event ) {
     console.log( "privVar1=" + privVar1 );
     console.log( "this.btn1Text=" + this.btn1Text );
   };

  // public space
  return {
   // public properties, e.g. strings to translate
   btn1Text: "Button 1",

   // public methods
   init: function () {

     if ( Ext.Ext2 ) {

       btn1 = new Ext.Button({
         renderTo: "btn1-ct",
```

```
        text: this.btn1Text,
        handler: btn1Handler
      });

    } else {

      btn1 = new Ext.Button( "btn1-ct", {
        text: this.btn1Text,
        handler: btn1Handler
      });

    }
  }
 };
}();
```

YUI

类似地，我们也可以使用YUI3来实现模块模式。下面的例子很大程度上是基于原始由Eric Miraglia实现的YUI本身的模块模式，但是和vanillla Javascript 实现的版本比较起来差异不是很大。

```
Y.namespace( "store.basket" ) = (function () {

  var myPrivateVar, myPrivateMethod;

  // private variables:
  myPrivateVar = "I can be accessed only within Y.store.basket.";

  // private method:
  myPrivateMethod = function () {
    Y.log( "I can be accessed only from within YAHOO.store.basket" );
  }

  return {
    myPublicProperty: "I'm a public property.",

    myPublicMethod: function () {
      Y.log( "I'm a public method." );

      // Within basket, I can access "private" vars and methods:
      Y.log( myPrivateVar );
      Y.log( myPrivateMethod() );

      // The native scope of myPublicMethod is store so we can
```

```
        // access public members using "this":
        Y.log( this.myPublicProperty );
      }
    };

})();
```

jQuery

因为jQuery编码规范没有规定插件如何实现模块模式，因此有很多种方式可以实现模块模式。Ben Cherry 之间提供一种方案，因为模块之间可能存在大量的共性，因此通过使用函数包装器封装模块的定义。

在下面的例子中，定义了一个library 函数，这个函数声明了一个新的库，并且在新的库（ 例如 模块 ）创建的时候，自动将初始化函数绑定到document的ready上。

```
function library( module ) {

  $( function() {
    if ( module.init ) {
      module.init();
    }
  });

  return module;
}

var myLibrary = library(function () {

  return {
    init: function () {
      // module implementation
    }
  };
}());
```

优势

既然我们已经看到单例模式很有用，为什么还是使用模块模式呢？首先，对于有面向对象背景的开发者来讲，至少从javascript语言上来讲，模块模式相对于真正的封装概念更清晰。

其次，模块模式支持私有数据–因此，在模块模式中，公共部分代码可以访问私有数据，但是在模块外部，不能访问类的私有部分（ 没开玩笑！感谢David Engfer 的玩笑 ）。

缺点

模块模式的缺点是因为我们采用不同的方式访问公有和私有成员，因此当我们想要改变这些成员的可见性的时候，我们不得不在所有使用这些成员的地方修改代码。

我们也不能在对象之后添加的方法里面访问这些私有变量。也就是说，很多情况下，模块模式很有用，并且当使用正确的时候，潜在地可以改善我们代码的结构。

其它缺点包括不能为私有成员创建自动化的单元测试，以及在紧急修复bug时所带来的额外的复杂性。根本没有可能可以对私有成员打补丁。相反地，我们必须覆盖所有的使用存在bug私有成员的公共方法。开发者不能简单的扩展私有成员，因此我们需要记得，私有成员并非它们表面上看上去那么具有扩展性。

想要了解更深入的信息，可以阅读 Ben Cherry (http://www.adequatelygood.com/JavaScript-Module-Pattern-In-Depth.html) 这篇精彩的文章。

暴露模块模式

既然我们对模块模式已经有一些了解了，让我们看一下改进版本 – Christian Heilmann 的启发式模块模式。 启发式模块模式来自于，当Heilmann对这样一个现状的不满，即当我们想要在一个公有方法中调用另外一个公有方法，或者访问公有变量的时候，我们不得不重复主对象的名称。他也不喜欢模块模式中，当想要将某个成员变成公共成员时，修改文字标记的做法。

因此他工作的结果就是一个更新的模式，在这个模式中，我们可以简单地在私有域中定义我们所有的函数和变量，并且返回一个匿名对象，这个对象包含有一些指针，这些指针指向我们想要暴露出来的私有成员，使这些私有成员公有化。

下面给出一个如何使用暴露式模块模式的例子:

```
var myRevealingModule = function () {

    var privateVar = "Ben Cherry",
        publicVar  = "Hey there!";

    function privateFunction() {
        console.log( "Name:" + privateVar );
    }

    function publicSetName( strName ) {
        privateVar = strName;
    }

    function publicGetName() {
        privateFunction();
    }


    // Reveal public pointers to
    // private functions and properties

    return {
        setName: publicSetName,
        greeting: publicVar,
        getName: publicGetName
    };

}();
```

```
myRevealingModule.setName( "Paul Kinlan" );
```

这个模式可以用于将私有函数和属性以更加规范的命名方式展现出来。

```
var myRevealingModule = function () {

       var privateCounter = 0;

       function privateFunction() {
          privateCounter++;
       }

       function publicFunction() {
          publicIncrement();
       }

       function publicIncrement() {
          privateFunction();
       }

       function publicGetCount(){
         return privateCounter;
       }

       // Reveal public pointers to
       // private functions and properties

      return {
          start: publicFunction,
          increment: publicIncrement,
          count: publicGetCount
       };

   }();

myRevealingModule.start();
```

优势

这个模式是我们脚本的语法更加一致。同样在模块的最后关于那些函数和变量可以被公共访问也变得更加清晰，增强了可读性。

缺点

这个模式的一个缺点是如果私有函数需要使用公有函数，那么这个公有函数在需要打补丁的时候就不能被重载。因为私有函数仍然使用的是私有的实现，并且这个模式不能用于公有成员，只用于函数。

公有成员使用私有成员也遵循上面不能打补丁的规则。

因为上面的原因，使用暴露式模块模式创建的模块相对于原始的模块模式更容易出问题，因此在使用的时候需要小心。

单例模式

单例模式之所以这么叫，是因为它限制一个类只能有一个实例化对象。经典的实现方式是，创建一个类，这个类包含一个方法，这个方法在没有对象存在的情况下，将会创建一个新的实例对象。如果对象存在，这个方法只是返回这个对象的引用。

单例和静态类不同，因为我们可以退出单例的初始化时间。通常这样做是因为，在初始化的时候需要一些额外的信息，而这些信息在声明的时候无法得知。对于并不知晓对单例模式引用的代码来讲，单例模式没有为它们提供一种方式可以简单的获取单例模式。这是因为，单例模式既不返回对象也不返回类，它只返回一种结构。可以类比闭包中的变量不是闭包–提供闭包的函数域是闭包（绕进去了）。

在JavaScript语言中, 单例服务作为一个从全局空间的代码实现中隔离出来共享的资源空间是为了提供一个单独的函数访问指针。

我们能像这样实现一个单例:

```
var mySingleton = (function () {

 // Instance stores a reference to the Singleton
 var instance;

 function init() {

  // 单例

  // 私有方法和变量
  function privateMethod(){
      console.log( "I am private" );
  }

  var privateVariable = "Im also private";

  var privateRandomNumber = Math.random();

  return {

   // 共有方法和变量
   publicMethod: function () {
     console.log( "The public can see me!" );
   },
```

```
    publicProperty: "I am also public",

    getRandomNumber: function() {
      return privateRandomNumber;
    }

  };

};

 return {

  // 如果存在获取此单例实例，如果不存在创建一个单例实例
  getInstance: function () {

    if ( !instance ) {
      instance = init();
    }

    return instance;
  }

 };

})();

var myBadSingleton = (function () {

 // 存储单例实例的引用
 var instance;

 function init() {

  // 单例

  var privateRandomNumber = Math.random();

  return {

    getRandomNumber: function() {
      return privateRandomNumber;
    }

  };
```

```
  };

  return {

    // 总是创建一个新的实例
    getInstance: function () {

      instance = init();

      return instance;
    }

  };

})();


// 使用:

var singleA = mySingleton.getInstance();
var singleB = mySingleton.getInstance();
console.log( singleA.getRandomNumber() === singleB.getRandomNumber() ); // true

var badSingleA = myBadSingleton.getInstance();
var badSingleB = myBadSingleton.getInstance();
console.log( badSingleA.getRandomNumber() !== badSingleB.getRandomNumber() ); // true
```

创建一个全局访问的单例实例 (通常通过 MySingleton.getInstance()) 因为我们不能(至少在静态语言中) 直接调用 new MySingleton() 创建实例. 这在JavaScript语言中是不可能的。

在四人帮(GoF)的书里面，单例模式的应用描述如下：

- 每个类只有一个实例，这个实例必须通过一个广为人知的接口，来被客户访问。
- 子类如果要扩展这个唯一的实例，客户可以不用修改代码就能使用这个扩展后的实例。

关于第二点，可以参考如下的实例，我们需要这样编码:

```
mySingleton.getInstance = function(){
  if ( this._instance == null ) {
    if ( isFoo() ) {
      this._instance = new FooSingleton();
    } else {
      this._instance = new BasicSingleton();
    }
  }
```

```
  return this._instance;
};
```

在这里，getInstance 有点类似于工厂方法，我们不需要去更新每个访问单例的代码。FooSingleton可以是BasicSinglton的子类，并且实现了相同的接口。

为什么对于单例模式来讲，延迟执行执行这么重要?

在c++代码中，单例模式将不可预知的动态初始化顺序问题隔离掉，将控制权返回给程序员。

区分类的静态实例和单例模式很重要：尽管单例模式可以被实现成一个静态实例，但是单例可以懒构造，在真正用到之前，单例模式不需要分配资源或者内存。

如果我们有个静态对象可以被直接初始化，我们需要保证代码总是以同样的顺序执行（例如 汽车需要轮胎先初始化）当你有很多源文件的时候，这种方式没有可扩展性。

单例模式和静态对象都很有用，但是不能滥用-同样的我们也不能滥用其它模式。

在实践中，当一个对象需要和另外的对象进行跨系统协作的时候，单例模式很有用。下面是一个单例模式在这种情况下使用的例子：

```
var SingletonTester = (function () {

  // options: an object containing configuration options for the singleton
  // e.g var options = { name: "test", pointX: 5};
  function Singleton( options ) {

    // set options to the options supplied
    // or an empty object if none are provided
    options = options || {};

    // set some properties for our singleton
    this.name = "SingletonTester";

    this.pointX = options.pointX || 6;

    this.pointY = options.pointY || 10;

  }

  // our instance holder
  var instance;

  // an emulation of static variables and methods
  var _static  = {
```

```
  name:  "SingletonTester",

  // Method for getting an instance. It returns
  // a singleton instance of a singleton object
  getInstance:  function( options ) {
    if( instance  ===  undefined )  {
      instance = new Singleton( options );
    }

    return  instance;

  }
 };

 return  _static;

})();

var singletonTest  =  SingletonTester.getInstance({
 pointX:  5
});

// Log the output of pointX just to verify it is correct
// Outputs: 5
console.log( singletonTest.pointX );
```

尽管单例模式有着合理的使用需求，但是通常当我们发现自己需要在javascript使用它的时候，这是一种信号，表明我们可能需要去重新评估自己的设计。

这通常表明系统中的模块要么紧耦合要么逻辑过于分散在代码库的多个部分。单例模式更难测试，因为可能有多种多样的问题出现，例如隐藏的依赖关系，很难去创建多个实例，很难清理依赖关系，等等。

要想进一步了解关于单例的信息，可以读读 Miller Medeiros 推荐的这篇非常棒的关于单例模式以及单例模式各种各样问题的文章 (http://www.ibm.com/developerworks/webservices/library/co-single/index.html)，也可以看看这篇文章的评论，这些评论讨论了单例模式是怎样增加了模块间的紧耦合。我很乐意去支持这些推荐，因为这两篇文章提出了很多关于单例模式重要的观点，而这些观点是很值得重视的。

观察者模式

观察者模式是这样一种设计模式。一个被称作被观察者的对象，维护一组被称为观察者的对象，这些对象依赖于被观察者，被观察者自动将自身的状态的任何变化通知给它们。

当一个被观察者需要将一些变化通知给观察者的时候，它将采用广播的方式，这条广播可能包含特定于这条通知的一些数据。

当特定的观察者不再需要接受来自于它所注册的被观察者的通知的时候，被观察者可以将其从所维护的组中删除。 在这里提及一下设计模式现有的定义很有必要。这个定义是与所使用的语言无关的。通过这个定义，最终我们可以更深层次地了解到设计模式如何使用以及其优势。在四人帮的《 设计模式:可重用的面向对象软件的元素 》这本书中，是这样定义观察者模式的:

一个或者更多的观察者对一个被观察者的状态感兴趣，将自身的这种兴趣通过附着自身的方式注册在被观察者身上。当被观察者发生变化，而这种便可也是观察者所关心的，就会产生一个通知，这个通知将会被送出去，最后将会调用每个观察者的更新方法。当观察者不在对被观察者的状态感兴趣的时候，它们只需要简单的将自身剥离即可。

我们现在可以通过实现一个观察者模式来进一步扩展我们刚才所学到的东西。这个实现包含一下组件:

- 被观察者：维护一组观察者， 提供用于增加和移除观察者的方法。
- 观察者：提供一个更新接口，用于当被观察者状态变化时，得到通知。
- 具体的被观察者：状态变化时广播通知给观察者，保持具体的观察者的信息。
- 具体的观察者：保持一个指向具体被观察者的引用，实现一个更新接口，用于观察，以便保证自身状态总是和被观察者状态一致的。

首先，让我们对被观察者可能有的一组依赖其的观察者进行建模:

```
function ObserverList(){
  this.observerList = [];
}

ObserverList.prototype.Add = function( obj ){
  return this.observerList.push( obj );
};

ObserverList.prototype.Empty = function(){
  this.observerList = [];
};
```

```
ObserverList.prototype.Count = function(){
  return this.observerList.length;
};


ObserverList.prototype.Get = function( index ){
  if( index > -1 && index < this.observerList.length ){
    return this.observerList[ index ];
  }
};

ObserverList.prototype.Insert = function( obj, index ){
  var pointer = -1;

  if( index === 0 ){
    this.observerList.unshift( obj );
    pointer = index;
  }else if( index === this.observerList.length ){
    this.observerList.push( obj );
    pointer = index;
  }

  return pointer;
};

ObserverList.prototype.IndexOf = function( obj, startIndex ){
  var i = startIndex, pointer = -1;

  while( i < this.observerList.length ){
    if( this.observerList[i] === obj ){
      pointer = i;
    }
    i++;
  }

  return pointer;
};


ObserverList.prototype.RemoveAt = function( index ){
  if( index === 0 ){
    this.observerList.shift();
  }else if( index === this.observerList.length -1 ){
    this.observerList.pop();
```

```
  }
};


// Extend an object with an extension
function extend( extension, obj ){
  for ( var key in extension ){
    obj[key] = extension[key];
  }
}
```

接着，我们对被观察者以及其增加，删除，通知在观察者列表中的观察者的能力进行建模：

```
function Subject(){
  this.observers = new ObserverList();
}

Subject.prototype.AddObserver = function( observer ){
  this.observers.Add( observer );
};

Subject.prototype.RemoveObserver = function( observer ){
  this.observers.RemoveAt( this.observers.IndexOf( observer, 0 ) );
};

Subject.prototype.Notify = function( context ){
  var observerCount = this.observers.Count();
  for(var i=0; i < observerCount; i++){
    this.observers.Get(i).Update( context );
  }
};
```

我们接着定义建立新的观察者的一个框架。这里的update 函数之后会被具体的行为覆盖。

```
// The Observer
function Observer(){
  this.Update = function(){
    // ...
  };
}
```

在我们的样例应用里面，我们使用上面的观察者组件，现在我们定义：

- 一个按钮，这个按钮用于增加新的充当观察者的选择框到页面上
- 一个控制用的选择框 , 充当一个被观察者，通知其它选择框是否应该被选中

- 一个容器，用于放置新的选择框

我们接着定义具体被观察者和具体观察者，用于给页面增加新的观察者，以及实现更新接口。通过查看下面的内联的注释，搞清楚在我们样例中的这些组件是如何工作的。

HTML

```
<button id="addNewObserver">Add New Observer checkbox</button>
<input id="mainCheckbox" type="checkbox"/>
<div id="observersContainer"></div>
```

Sample script

```
// 我们DOM 元素的引用

var controlCheckbox = document.getElementById( "mainCheckbox" ),
  addBtn = document.getElementById( "addNewObserver" ),
  container = document.getElementById( "observersContainer" );


// 具体的被观察者

//Subject 类扩展controlCheckbox 类
extend( new Subject(), controlCheckbox );

//点击checkbox 将会触发对观察者的通知
controlCheckbox["onclick"] = new Function( "controlCheckbox.Notify(controlCheckbox.checked)" );


addBtn["onclick"] = AddNewObserver;

// 具体的观察者

function AddNewObserver(){

  //建立一个新的用于增加的checkbox
  var check  = document.createElement( "input" );
  check.type = "checkbox";

  // 使用Observer 类扩展checkbox
  extend( new Observer(), check );
```

```
  // 使用定制的Update函数重载
  check.Update = function( value ){
    this.checked = value;
  };

  // 增加新的观察者到我们主要的被观察者的观察者列表中
  controlCheckbox.AddObserver( check );

  // 将元素添加到容器的最后
  container.appendChild( check );
}
```

在这个例子里面，我们看到了如何实现和配置观察者模式，了解了被观察者，观察者，具体被观察者，具体观察者的概念。

观察者模式和发布/订阅模式的不同

观察者模式确实很有用，但是在javascript时间里面，通常我们使用一种叫做发布/订阅模式的变体来实现观察者模式。这两种模式很相似，但是也有一些值得注意的不同。

观察者模式要求想要接受相关通知的观察者必须到发起这个事件的被观察者上注册这个事件。

发布/订阅模式使用一个主题/事件频道，这个频道处于想要获取通知的订阅者和发起事件的发布者之间。这个事件系统允许代码定义应用相关的事件，这个事件可以传递特殊的参数，参数中包含有订阅者所需要的值。这种想法是为了避免订阅者和发布者之间的依赖性。

这种和观察者模式之间的不同，使订阅者可以实现一个合适的事件处理函数，用于注册和接受由发布者广播的相关通知。

这里给出一个关于如何使用发布者/订阅者模式的例子，这个例子中完整地实现了功能强大的publish(), subscribe() 和 unsubscribe()。

```
// 一个非常简单的邮件处理器

// 接受的消息的计数器
var mailCounter = 0;

// 初始化一个订阅者，这个订阅者监听名叫"inbox/newMessage" 的频道

// 渲染新消息的粗略信息
var subscriber1 = subscribe( "inbox/newMessage", function( topic, data ) {

  // 日志记录主题，用于调试
```

```
  console.log( "A new message was received: ", topic );

  // 使用来自于被观察者的数据，用于给用户展示一个消息的粗略信息
  $( ".messageSender" ).html( data.sender );
  $( ".messagePreview" ).html( data.body );

});

// 这是另外一个订阅者，使用相同的数据执行不同的任务

// 更细计数器，显示当前来自于发布者的新信息的数量
var subscriber2 = subscribe( "inbox/newMessage", function( topic, data ) {

  $('.newMessageCounter').html( mailCounter++ );

});

publish( "inbox/newMessage", [{
  sender:"hello@google.com",
  body: "Hey there! How are you doing today?"
}]);

// 在之后，我们可以让我们的订阅者通过下面的方式取消订阅来自于新主题的通知
// unsubscribe( subscriber1，);
// unsubscribe( subscriber2 );
```

这个例子的更广的意义是对松耦合的原则的一种推崇。不是一个对象直接调用另外一个对象的方法，而是通过订阅另外一个对象的一个特定的任务或者活动，从而在这个任务或者活动出现的时候的得到通知。

优势

观察者和发布/订阅模式鼓励人们认真考虑应用不同部分之间的关系，同时帮助我们找出这样的层，该层中包含有直接的关系，这些关系可以通过一些列的观察者和被观察者来替换掉。这中方式可以有效地将一个应用程序切割成小块，这些小块耦合度低，从而改善代码的管理，以及用于潜在的代码复用。

使用观察者模式更深层次的动机是，当我们需要维护相关对象的一致性的时候，我们可以避免对象之间的紧密耦合。例如，一个对象可以通知另外一个对象，而不需要知道这个对象的信息。

两种模式下，观察者和被观察者之间都可以存在动态关系。这提供很好的灵活性，而当我们的应用中不同的部分之间紧密耦合的时候，是很难实现这种灵活性的。

尽管这些模式并不是万能的灵丹妙药，这些模式仍然是作为最好的设计松耦合系统的工具之一，因此在任何的JavaScript 开发者的工具箱里面，都应该有这样一个重要的工具。

缺点

事实上，这些模式的一些问题实际上正是来自于它们所带来的一些好处。在发布/订阅模式中，将发布者共订阅者上解耦，将会在一些情况下，导致很难确保我们应用中的特定部分按照我们预期的那样正常工作。

例如，发布者可以假设有一个或者多个订阅者正在监听它们。比如我们基于这样的假设，在某些应用处理过程中来记录或者输出错误日志。如果订阅者执行日志功能崩溃了（或者因为某些原因不能正常工作），因为系统本身的解耦本质，发布者没有办法感知到这些事情。

另外一个这种模式的缺点是，订阅者对彼此之间存在没有感知，对切换发布者的代价无从得知。因为订阅者和发布者之间的动态关系，更新依赖也很能去追踪。

发布/订阅实现

发布/订阅在JavaScript的生态系统中非常合适，主要是因为作为核心的ECMAScript 实现是事件驱动的。尤其是在浏览器环境下更是如此，因为DOM使用事件作为其主要的用于脚本的交互API。

也就是说，无论是ECMAScript 还是DOM都没有在实现代码中提供核心对象或者方法用于创建定制的事件系统（DOM3 的CustomEvent是一个例外，这个事件绑定在DOM上，因此通常用处不大）。

幸运的是，流行的JavaScript库例如dojo, jQuery(定制事件)以及YUI已经有相关的工具，可以帮助我们方便的实现一个发布/订阅者系统。下面我们看一些例子。

```
// 发布

// jQuery: $(obj).trigger("channel", [arg1, arg2, arg3]);
$( el ).trigger( "/login", [{username:"test", userData:"test"}] );

// Dojo: dojo.publish("channel", [arg1, arg2, arg3] );
dojo.publish( "/login", [{username:"test", userData:"test"}] );

// YUI: el.publish("channel", [arg1, arg2, arg3]);
el.publish( "/login", {username:"test", userData:"test"} );


// 订阅

// jQuery: $(obj).on( "channel", [data], fn );
$( el ).on( "/login", function( event ){...} );

// Dojo: dojo.subscribe( "channel", fn);
```

```
var handle = dojo.subscribe( "/login", function(data){..} );

// YUI: el.on("channel", handler);
el.on( "/login", function( data ){...} );


// 取消订阅

// jQuery: $(obj).off( "channel" );
$( el ).off( "/login" );

// Dojo: dojo.unsubscribe( handle );
dojo.unsubscribe( handle );

// YUI: el.detach("channel");
el.detach( "/login" );
```

对于想要在vanilla Javascript(或者其它库)中使用发布/订阅模式的人来讲， AmplifyJS 包含了一个干净的，库无关的实现，可以和任何库或者工具箱一起使用。Radio.js (http://radio.uxder.com/) , PubSubJS (https://github.com/mroderick/PubSubJS) 或者 Pure JS PubSub 来自于 Peter Higgins (https://github.com/phiggins42/bloody-jquery-plugins/blob/55e41df9bf08f42378bb08b93efcb28555b61aeb/pubsub.js) 都有类似的替代品值得研究。

尤其对于jQuery 开发者来讲，他们拥有很多其它的选择，可以选择大量的良好实现的代码，从Peter Higgins 的jQuery插件到Ben Alman 在GitHub 上的（优化的）发布/订阅 jQuery gist。下面给出了这些代码的链接。

- Ben Alman的发布/订阅 gist (https://gist.github.com/661855) (推荐)
- Rick Waldron 在上面基础上修改的 jQuery-core 风格的实现 (https://gist.github.com/705311)
- Peter Higgins 的插件 (http://github.com/phiggins42/bloody-jquery-plugins/blob/master/pubsub.js)
- AppendTo 在AmplifyJS中的 发布/订阅实现 (http://amplifyjs.com)
- Ben Truyman的 gist (https://gist.github.com/826794)

从上面我们可以看到在javascript中有这么多种观察者模式的实现，让我们看一下最小的一个版本的发布/订阅模式实现，这个实现我放在github 上，叫做pubsubz。这个实现展示了发布，订阅的核心概念，以及如何取消订阅。

我之所以选择这个代码作为我们例子的基础，是因为这个代码紧密贴合了方法签名和实现方式，这种实现方式正是我想看到的javascript版本的经典的观察者模式所应该有的样子。

发布/订阅实例

```
var pubsub = {};

(function(q) {

    var topics = {},
        subUid = -1;

    // Publish or broadcast events of interest
    // with a specific topic name and arguments
    // such as the data to pass along
    q.publish = function( topic, args ) {

        if ( !topics[topic] ) {
            return false;
        }

        var subscribers = topics[topic],
            len = subscribers ? subscribers.length : 0;

        while (len--) {
            subscribers[len].func( topic, args );
        }

        return this;
    };

    // Subscribe to events of interest
    // with a specific topic name and a
    // callback function, to be executed
    // when the topic/event is observed
    q.subscribe = function( topic, func ) {

        if (!topics[topic]) {
            topics[topic] = [];
        }

        var token = ( ++subUid ).toString();
        topics[topic].push({
            token: token,
            func: func
        });
        return token;
```

```
    };

    // Unsubscribe from a specific
    // topic, based on a tokenized reference
    // to the subscription
    q.unsubscribe = function( token ) {
        for ( var m in topics ) {
            if ( topics[m] ) {
                for ( var i = 0, j = topics[m].length; i < j; i++ ) {
                    if ( topics[m][i].token === token) {
                        topics[m].splice( i, 1 );
                        return token;
                    }
                }
            }
        }
        return this;
    };
}( pubsub ));
```

示例:使用我们的实现

我们现在可以使用发布实例和订阅感兴趣的事件，例如:

```
// Another simple message handler

// A simple message logger that logs any topics and data received through our
// subscriber
var messageLogger = function ( topics, data ) {
    console.log( "Logging: " + topics + ": " + data );
};

// Subscribers listen for topics they have subscribed to and
// invoke a callback function (e.g messageLogger) once a new
// notification is broadcast on that topic
var subscription = pubsub.subscribe( "inbox/newMessage", messageLogger );

// Publishers are in charge of publishing topics or notifications of
// interest to the application. e.g:

pubsub.publish( "inbox/newMessage", "hello world!" );

// or
pubsub.publish( "inbox/newMessage", ["test", "a", "b", "c"] );
```

```
// or
pubsub.publish( "inbox/newMessage", {
  sender: "hello@google.com",
  body: "Hey again!"
});

// We cab also unsubscribe if we no longer wish for our subscribers
// to be notified
// pubsub.unsubscribe( subscription );

// Once unsubscribed, this for example won't result in our
// messageLogger being executed as the subscriber is
// no longer listening
pubsub.publish( "inbox/newMessage", "Hello! are you still there?" );
```

例如：用户界面通知

接下来，让我们想象一下，我们有一个Web应用程序，负责显示实时股票信息。

应用程序可能有一个表格显示股票统计数据和一个计数器显示的最后更新点。当数据模型发生变化时，应用程序将需要更新表格和计数器。在这种情况下，我们的主题（这将发布主题/通知）是数据模型以及我们的订阅者是表格和计数器。

当我们的订阅者收到通知：该模型本身已经改变，他们自己可以进行相应的更新。

在我们的实现中，如果发现新的股票信息是可用的，我们的订阅者将收听到的主题“新数据可用”。如果一个新的通知发布到该主题，那将触发表格去添加一个包含此信息的新行。它也将更新最后更新计数器，记录最后一次添加的数据

```
// Return the current local time to be used in our UI later
getCurrentTime = function (){

   var date = new Date(),
        m = date.getMonth() + 1,
        d = date.getDate(),
        y = date.getFullYear(),
        t = date.toLocaleTimeString().toLowerCase();

       return (m + "/" + d + "/" + y + " " + t);
};

// Add a new row of data to our fictional grid component
function addGridRow( data ) {
```

```
  // ui.grid.addRow( data );
  console.log( "updated grid component with:" + data );

}

// Update our fictional grid to show the time it was last
// updated
function updateCounter( data ) {

  // ui.grid.updateLastChanged( getCurrentTime() );
  console.log( "data last updated at: " + getCurrentTime() + " with " + data);

}

// Update the grid using the data passed to our subscribers
gridUpdate = function( topic, data ){

 if ( data !== "undefined" ) {
   addGridRow( data );
   updateCounter( data );
  }

};

// Create a subscription to the newDataAvailable topic
var subscriber = pubsub.subscribe( "newDataAvailable", gridUpdate );

// The following represents updates to our data layer. This could be
// powered by ajax requests which broadcast that new data is available
// to the rest of the application.

// Publish changes to the gridUpdated topic representing new entries
pubsub.publish( "newDataAvailable", {
 summary: "Apple made $5 billion",
 identifier: "APPL",
 stockPrice: 570.91
});

pubsub.publish( "newDataAvailable", {
 summary: "Microsoft made $20 million",
 identifier: "MSFT",
 stockPrice: 30.85
});
```

样例：在下面这个电影评分的例子里面，我们使用Ben Alman的发布/订阅实现来解耦应用程序。我们使用Ben Alman的jQuery实现，来展示如何解耦用户界面。请注意，我们如何做到提交一个评分，来产生一个发布信息，这个信息表明了当前新的用户和评分数据可用。

剩余的工作留给订阅者，由订阅者来代理这些主题中的数据发生的变化。在我们的例子中，我们将新的数据压入到现存的数组中，接着使用Underscore库的template()方法来渲染模板。

HTML/模板

```
<script id="userTemplate" type="text/html">
  <li><%= name %></li>
</script>


<script id="ratingsTemplate" type="text/html">
  <li><strong><%= title %></strong> was rated <%= rating %>/5</li>
</script>


<div id="container">

  <div class="sampleForm">
    <p>
      <label for="twitter_handle">Twitter handle:</label>
      <input type="text" id="twitter_handle" />
    </p>
    <p>
      <label for="movie_seen">Name a movie you've seen this year:</label>
      <input type="text" id="movie_seen" />
    </p>
    <p>

      <label for="movie_rating">Rate the movie you saw:</label>
      <select id="movie_rating">
          <option value="1">1</option>
          <option value="2">2</option>
          <option value="3">3</option>
          <option value="4">4</option>
          <option value="5" selected>5</option>

      </select>
    </p>
    <p>
```

```
        <button id="add">Submit rating</button>
      </p>
    </div>



    <div class="summaryTable">
      <div id="users"><h3>Recent users</h3></div>
      <div id="ratings"><h3>Recent movies rated</h3></div>
    </div>

  </div>
```

JavaScript

```
;(function( $ ) {

  // Pre-compile templates and "cache" them using closure
  var
    userTemplate = _.template($( "#userTemplate" ).html()),
    ratingsTemplate = _.template($( "#ratingsTemplate" ).html());

  // Subscribe to the new user topic, which adds a user
  // to a list of users who have submitted reviews
  $.subscribe( "/new/user", function( e, data ){

    if( data ){

      $('#users').append( userTemplate( data ));

    }

  });

  // Subscribe to the new rating topic. This is composed of a title and
  // rating. New ratings are appended to a running list of added user
  // ratings.
  $.subscribe( "/new/rating", function( e, data ){

    var compiledTemplate;

    if( data ){
```

```
    $( "#ratings" ).append( ratingsTemplate( data );

  }

});

// Handler for adding a new user
$("#add").on("click", function( e ) {

  e.preventDefault();

  var strUser = $("#twitter_handle").val(),
    strMovie = $("#movie_seen").val(),
    strRating = $("#movie_rating").val();

  // Inform the application a new user is available
  $.publish( "/new/user", { name: strUser } );

  // Inform the app a new rating is available
  $.publish( "/new/rating", { title: strMovie, rating: strRating} );

  });

})( jQuery );
```

样例：解耦一个基于Ajax的jQuery应用。

在我们最后的例子中，我们将从实用的角度来看一下如何在开发早起使用发布/订阅模式来解耦代码，这样可以帮助我们避免之后痛苦的重构过程。

在Ajax重度依赖的应用里面，我们常会见到这种情况，当我们收到一个请求的响应之后，我们希望能够完成不仅仅一个特定的操作。我们可以简单的将所有请求后的逻辑加入到成功的回调函数里面，但是这样做有一些问题。

高度耦合的应用优势会增加重用功能的代价，因为高度耦合增加了内部函数/代码的依赖性。这意味着如果我们只是希望获取一次性获取结果集，可以将请求后 的逻辑代码 硬编码在回调函数里面，这种方式可以正常工作，但是当我们想要对相同的数据源(不同的最终行为)做更多的Ajax调用的时候，这种方式就不适合了，我们必须要多次重写部分代码。与其回溯调用相同数据源的每一层，然后在将它们泛化，不如一开始就使用发布/订阅模式来节约时间。

使用观察者，我们可以简单的将整个应用范围的通知进行隔离，针对不同的事件，我们可以把这种隔离做到我们想要的粒度上，如果使用其它模式，则可能不会有这么优雅的实现。

注意我们下面的例子中，当用户表明他们想要做一次搜索查询的时候，一个话题通知就会生成，而当请求返回，并且实际的数据可用的时候，又会生成另外一个通知。而如何使用这些事件（或者返回的数据），都是由订

阅者自己决定的。这样做的好处是，如果我们想要，我们可以有10个不同的订阅者，以不同的方式使用返回的数据，而对于Ajax层来讲，它不会关心你如何处理数据。它唯一的责任就是请求和返回数据，接着将数据发送给所有想要使用数据的地方。这种相关性上的隔离可以是我们整个代码设计更为清晰。

HTML/Templates

```
<form id="flickrSearch">

  <input type="text" name="tag" id="query"/>

  <input type="submit" name="submit" value="submit"/>

</form>



<div id="lastQuery"></div>

<div id="searchResults"></div>



<script id="resultTemplate" type="text/html">
  <% _.each(items, function( item ){  %>
       <li><p><img src="<%= item.media.m %>"/></p></li>
  <% });%>
</script>
```

JavaScript

```
;(function( $ ) {

  // Pre-compile template and "cache" it using closure
  var resultTemplate = _.template($( "#resultTemplate" ).html());

  // Subscribe to the new search tags topic
  $.subscribe( "/search/tags" , function( tags ) {
    $( "#searchResults" )
          .html("
<p>
  Searched for:<strong>" + tags + "</strong>
</p>
```

```
");
  });

  // Subscribe to the new results topic
  $.subscribe( "/search/resultSet" , function( results ){

     $( "#searchResults" ).append(resultTemplate( results ));

  });

  // Submit a search query and publish tags on the /search/tags topic
  $( "#flickrSearch" ).submit( function( e ) {

     e.preventDefault();
     var tags = $(this).find( "#query").val();

     if ( !tags ){
      return;
     }

     $.publish( "/search/tags" , [ $.trim(tags) ]);

  });


  // Subscribe to new tags being published and perform
  // a search query using them. Once data has returned
  // publish this data for the rest of the application
  // to consume

  $.subscribe("/search/tags", function( tags ) {

     $.getJSON( "http://api.flickr.com/services/feeds/photos_public.gne?jsoncallback=?" ,{
           tags: tags,
           tagmode: "any",
           format: "json"
         },

        function( data ){

           if( !data.items.length ) {
             return;
           }

           $.publish( "/search/resultSet" , data.items  );
```

```
    });

  });


})();
```

观察者模式在应用设计中，解耦一系列不同的场景上非常有用，如果你没有用过它，我推荐你尝试一下今天提到的之前写到的某个实现。这个模式是一个易于学习的模式，同时也是一个威力巨大的模式。

中介者模式

字典中中介者的定义是，一个中立方，在谈判和冲突解决过程中起辅助作用。在我们的世界，一个中介者是一个行为设计模式，使我们可以导出统一的接口，这样系统不同部分就可以彼此通信。

如果系统组件之间存在大量的直接关系，就可能是时候，使用一个中心的控制点，来让不同的组件通过它来通信。中介者通过将组件之间显式的直接的引用替换成通过中心点来交互的方式，来做到松耦合。这样可以帮助我们解耦，和改善组件的重用性。

在现实世界中，类似的系统就是，飞行控制系统。一个航站塔（中介者）处理哪个飞机可以起飞，哪个可以着陆，因为所有的通信（监听的通知或者广播的通知）都是飞机和控制塔之间进行的，而不是飞机和飞机之间进行的。一个中央集权的控制中心是这个系统成功的关键，也正是中介者在软件设计领域中所扮演的角色。

从实现角度来讲，中介者模式是观察者模式中的共享被观察者对象。在这个系统中的对象之间直接的发布/订阅关系被牺牲掉了，取而代之的是维护一个通信的中心节点。

也可以认为是一种补充–用于应用级别的通知，例如不同子系统之间的通信，子系统本身很复杂，可能需要使用发布/订阅模式来做内部组件之间的解耦。

另外一个类似的例子是DOM的事件冒泡机制，以及事件代理机制。如果系统中所有的订阅者都是对文档订阅，而不是对独立的节点订阅，那么文档就充当一个中介者的角色。DOM的这种做法，不是将事件绑定到独立节点上，而是用一个更高级别的对象负责通知订阅者关于交互事件的信息。

基础的实现

中间人模式的一种简单的实现可以在下面找到,publish()和subscribe()方法都被暴露出来使用:

```
var mediator = (function(){

  // Storage for topics that can be broadcast or listened to
  var topics = {};

  // Subscribe to a topic, supply a callback to be executed
  // when that topic is broadcast to
  var subscribe = function( topic, fn ){

    if ( !topics[topic] ){
      topics[topic] = [];
    }
```

```
        topics[topic].push( { context: this, callback: fn } );

        return this;
    };

    // Publish/broadcast an event to the rest of the application
    var publish = function( topic ){

        var args;

        if ( !topics[topic] ){
          return false;
        }

        args = Array.prototype.slice.call( arguments, 1 );
        for ( var i = 0, l = topics[topic].length; i < l; i++ ) {

            var subscription = topics[topic][i];
            subscription.callback.apply( subscription.context, args );
        }
        return this;
    };

    return {
        publish: publish,
        subscribe: subscribe,
        installTo: function( obj ){
            obj.subscribe = subscribe;
            obj.publish = publish;
        }
    };

}());
```

高级的实现

对于那些对更加高级实现感兴趣的人,以走读的方式看一看以下我对Jack Lawson优秀的Mediator.js重写的一个缩略版本.在其它方面的改进当中,为我们的中间人支持主题命名空间,用户拆卸和一个更加稳定的发布/订阅系统。但是如果你想跳过这个走读，你可以直接进入到下一个例子继续阅读。

得感谢Jack优秀的代码注释对这部分内容的协助。

首先，让我们实现认购的概念，我们可以考虑一个中间人主题的注册。

通过生成对象实体,我们稍后能够简单的更新认购,而不需要去取消注册然后重新注册它们.认购可以写成一个使用被称作一个选项对象或者一个上下文环境的函数

```
// Pass in a context to attach our Mediator to.
// By default this will be the window object
(function( root ){

  function guidGenerator() { /*..*/}

  // Our Subscriber constructor
  function Subscriber( fn, options, context ){

    if ( !(this instanceof Subscriber) ) {

      return new Subscriber( fn, context, options );

    }else{

      // guidGenerator() is a function that generates
      // GUIDs for instances of our Mediators Subscribers so
      // we can easily reference them later on. We're going
      // to skip its implementation for brevity

      this.id = guidGenerator();
      this.fn = fn;
      this.options = options;
      this.context = context;
      this.topic = null;

    }
  }
})();
```

在我们的中间人主题中包涵了一长串的回调和子主题,当中间人发布在我们中间人实体上被调用的时候被启动.它也包含操作数据列表的方法

```
// Let's model the Topic.
// JavaScript lets us use a Function object as a
// conjunction of a prototype for use with the new
// object and a constructor function to be invoked.
function Topic( namespace ){

  if ( !(this instanceof Topic) ) {
    return new Topic( namespace );
  }else{
```

```
    this.namespace = namespace || "";
    this._callbacks = [];
    this._topics = [];
    this.stopped = false;

  }
}


// Define the prototype for our topic, including ways to
// add new subscribers or retrieve existing ones.
Topic.prototype = {

  // Add a new subscriber
  AddSubscriber: function( fn, options, context ){

    var callback = new Subscriber( fn, options, context );

    this._callbacks.push( callback );

    callback.topic = this;

    return callback;
  },
...
```

我们的主题实体被当做中间人调用的一个参数被传递.使用一个方便实用的calledStopPropagation()方法,回调就可以进一步被传播开来:

```
StopPropagation: function(){
  this.stopped = true;
},
```

我们也能够使得当提供一个GUID的标识符的时候检索订购用户更加容易:

```
GetSubscriber: function( identifier ){

  for(var x = 0, y = this._callbacks.length; x < y; x++ ){
    if( this._callbacks[x].id == identifier || this._callbacks[x].fn == identifier ){
      return this._callbacks[x];
    }
  }

  for( var z in this._topics ){
    if( this._topics.hasOwnProperty( z ) ){
```

```
    var sub = this._topics[z].GetSubscriber( identifier );
    if( sub !== undefined ){
      return sub;
    }
   }
  }

},
```

接着,在我们需要它们的情况下,我们也能够提供添加新主题,检查现有的主题或者检索主题的简单方法:

```
AddTopic: function( topic ){
  this._topics[topic] = new Topic( (this.namespace ? this.namespace + ":" : "") + topic );
},

HasTopic: function( topic ){
  return this._topics.hasOwnProperty( topic );
},

ReturnTopic: function( topic ){
  return this._topics[topic];
},
```

如果我们觉得不再需要它们了,我们也可以明确的删除这些订购用户.下面就是通过它的其子主题递归删除订购用户的代码:

```
RemoveSubscriber: function( identifier ){

  if( !identifier ){
    this._callbacks = [];

    for( var z in this._topics ){
      if( this._topics.hasOwnProperty(z) ){
        this._topics[z].RemoveSubscriber( identifier );
      }
    }
  }

  for( var y = 0, x = this._callbacks.length; y < x; y++ ) {
    if( this._callbacks[y].fn == identifier || this._callbacks[y].id == identifier ){
      this._callbacks[y].topic = null;
      this._callbacks.splice( y,1 );
      x--; y--;
    }
  }
```

```
},
```

接着我们通过递归子主题将发布任意参数的能够包含到订购服务对象中:

```
Publish: function( data ){

    for( var y = 0, x = this._callbacks.length; y < x; y++ ) {

        var callback = this._callbacks[y], l;
          callback.fn.apply( callback.context, data );

      l = this._callbacks.length;

      if( l < x ){
        y--;
        x = l;
      }
    }

    for( var x in this._topics ){
      if( !this.stopped ){
        if( this._topics.hasOwnProperty( x ) ){
          this._topics[x].Publish( data );
        }
      }
    }

    this.stopped = false;
  }
};
```

接着我们暴露我们将主要交互的调节实体.这里它是通过注册的并且从主题中删除的事件来实现的

```
function Mediator() {

  if ( !(this instanceof Mediator) ) {
    return new Mediator();
  }else{
    this._topics = new Topic( "" );
  }

};
```

想要更多先进的用例,我们可以看看调解支持的主题命名空间,下面这样的asinbox:messages:new:read.GetTopic 返回基于一个命名空间的主题实体。

```
Mediator.prototype = {

  GetTopic: function( namespace ){
    var topic = this._topics,
        namespaceHierarchy = namespace.split( ":" );

    if( namespace === "" ){
      return topic;
    }

    if( namespaceHierarchy.length > 0 ){
      for( var i = 0, j = namespaceHierarchy.length; i < j; i++ ){

        if( !topic.HasTopic( namespaceHierarchy[i]) ){
          topic.AddTopic( namespaceHierarchy[i] );
        }

        topic = topic.ReturnTopic( namespaceHierarchy[i] );
      }
    }

    return topic;
  },
```

这一节我们定义了一个Mediator.Subscribe方法，它接受一个主题命名空间,一个将要被执行的函数,选项和又一个在订阅中调用函数的上下文环境.这样就创建了一个主题,如果这样的一个主题存在的话

```
Subscribe: function( topiclName, fn, options, context ){
  var options = options || {},
      context = context || {},
      topic = this.GetTopic( topicName ),
      sub = topic.AddSubscriber( fn, options, context );

  return sub;
},
```

根据这一点,我们可以进一步定义能够访问特定订阅用户,或者将他们从主题中递归删除的工具

```
// Returns a subscriber for a given subscriber id / named function and topic namespace

GetSubscriber: function( identifier, topic ){
  return this.GetTopic( topic || "" ).GetSubscriber( identifier );
},

// Remove a subscriber from a given topic namespace recursively based on
// a provided subscriber id or named function.
```

```
Remove: function( topicName, identifier ){
  this.GetTopic( topicName ).RemoveSubscriber( identifier );
},
```

我们主要的发布方式可以让我们随意发布数据到选定的主题命名空间，这可以在下面的代码中看到。

主题可以被向下递归.例如,一条对inbox:message的post将发送到inbox:message:new和inbox:message:new:read.它将像接下来这样被使用:Mediator.Publish("inbox:messages:new", [args]);

```
Publish: function( topicName ){
    var args = Array.prototype.slice.call( arguments, 1),
        topic = this.GetTopic( topicName );

    args.push( topic );

    this.GetTopic( topicName ).Publish( args );
  }
};
```

最后，我们可以很容易的暴露我们的中间人，将它附着在传递到根中的对象上：

```
 root.Mediator = Mediator;
  Mediator.Topic = Topic;
  Mediator.Subscriber = Subscriber;

// Remember we can pass anything in here. I've passed inwindowto
// attach the Mediator to, but we can just as easily attach it to another
// object if desired.
})( window );
```

示例

无论是使用来自上面的实现（简单的选项和更加先进的选项都是），我们能够像下面这样将一个简单的聊天记录系统整到一起:

HTML

```
<h1>Chat</h1>
<form id="chatForm">
    <label for="fromBox">Your Name:</label>
    <input id="fromBox" type="text"/>
    <br />
```

```
    <label for="toBox">Send to:</label>
    <input id="toBox" type="text"/>
    <br />
    <label for="chatBox">Message:</label>
    <input id="chatBox" type="text"/>
    <button type="submit">Chat</button>
</form>

<div id="chatResult"></div>
```

Javascript

```
$( "#chatForm" ).on( "submit", function(e) {
    e.preventDefault();

    // Collect the details of the chat from our UI
    var text = $( "#chatBox" ).val(),
        from = $( "#fromBox" ).val(),
        to = $( "#toBox" ).val();

    // Publish data from the chat to the newMessage topic
    mediator.publish( "newMessage" , { message: text, from: from, to: to } );
});

// Append new messages as they come through
function displayChat( data ) {
    var date = new Date(),
        msg = data.from + " said \"" + data.message + "\" to " + data.to;

    $( "#chatResult" )
        .prepend("
<p>
    " + msg + " (" + date.toLocaleTimeString() + ")
</p>
");
}

// Log messages
function logChat( data ) {
    if ( window.console ) {
        console.log( data );
    }
}
```

```
// Subscribe to new chat messages being submitted
// via the mediator
mediator.subscribe( "newMessage", displayChat );
mediator.subscribe( "newMessage", logChat );


// The following will however only work with the more advanced implementation:

function amITalkingToMyself( data ) {
   return data.from === data.to;
}

function iAmClearlyCrazy( data ) {
   $( "#chatResult" ).prepend("
<p>
   " + data.from + " is talking to himself.
</p>
");
}

 mediator.Subscribe( amITalkingToMyself, iAmClearlyCrazy );
```

优点&缺点

中间人模式最大的好处就是，它节约了对象或者组件之间的通信信道，这些对象或者组件存在于从多对多到多对一的系统之中。由于解耦合水平的因素，添加新的发布或者订阅者是相对容易的。

也许使用这个模式最大的缺点是它可以引入一个单点故障。在模块之间放置一个中间人也可能会造成性能损失，因为它们经常是间接地的进行通信的。由于松耦合的特性，仅仅盯着广播很难去确认系统是如何做出反应的。

这就是说，提醒我们自己解耦合的系统拥有许多其它的好处，是很有用的——如果我们的模块互相之间直接的进行通信，对于模块的改变（例如：另一个模块抛出了异常）可以很容易的对我们系统的其它部分产生多米诺连锁效应。这个问题在解耦合的系统中很少需要被考虑到。

在一天结束的时候，紧耦合会导致各种头痛，这仅仅只是另外一种可选的解决方案，但是如果得到正确实现的话也能够工作得很好。

中间人VS观察者

开发人员往往不知道中间人模式和观察者模式之间的区别。不可否认，这两种模式之间有一点点重叠，但让我们回过头来重新寻求GoF的一种解释：

“在观察者模式中，没有封装约束的单一对象”。取而代之，观察者和主题必须合作来维护约束。通信的模式决定于观察者和主题相互关联的方式：一个单独的主题经常有许多的观察者，而有时候一个主题的观察者是另外一个观察者的主题。“

中间人和观察者都提倡松耦合，然而，中间人默认使用让对象严格通过中间人进行通信的方式实现松耦合。观察者模式则创建了观察者对象，这些观察者对象会发布触发对象认购的感兴趣的事件。

中间人VS门面

不久我们的描述就将涵盖门面模式，但作为参考之用，一些开发者也想知道中间人和门面模式之间有哪些相似之处。它们都对模块的功能进行抽象，但有一些细微的差别。

中间人模式让模块之间集中进行通信，它会被这些模块明确的引用。门面模式却只是为模块或者系统定义一个更加简单的接口，但不添加任何额外的功能。系统中其他的模块并不直接意识到门面的概念，而可以被认为是单向的。

原型模式

GoF将原型模式引用为通过克隆的方式基于一个现有对象的模板创建对象的模式。

我们能够将原型模式认作是基于原型的继承中,我们创建作为其它对象原型的对象.原型对象自身被当做构造器创建的每一个对象的蓝本高效的使用着.如果构造器函数使用的原型包含例如叫做name的属性,那么每一个通过同一个构造器创建的对象都将拥有这个相同的属性。

在现存的(非Javascript的)语法中重新看一看对这个模式的定义,我们也许可以再一次发现对类的引用.真实的情况是那种原型继承避免了完全使用类.理论上既不是一个"定义的“对象，也不是一个核心对象。我们可以简单的创建现存函数型对象的拷贝。

使用原型模式的好处之一就是,我们在JavaScript提供的原生能力之上工作的,而不是JavaScript试图模仿的其它语言的特性.而对于其它的模式来说,情况并非如此。

这一模式不仅仅是实现继承的一种简单方式,它顺便还能够带来一点性能上的提升:当定义对象的一个方法时,它们都是使用引用创建的(因此所有的子对象都指向同一个函数),而不是创建属于它们的单独的拷贝。

对于那些有趣的,真正原型的集成,像ECMAScript 5标准中所定义的那样,需要使用 Object.create(如我们在本节的前面部分所见到的).为了提醒我们自己,Object.create创建了一个拥有特定原型的对象,并且也包含选项式的特定属性.(例如,Object.create(prototype,optionalDescriptorObject))。

我们可以在下面的示例中看到对这个的展示:

```
var myCar = {

  name: "Ford Escort",

  drive: function () {
    console.log( "Weeee. I'm driving!" );
  },

  panic: function () {
    console.log( "Wait. How do you stop this thing?" );
  }

};

// Use Object.create to instantiate a new car
var yourCar = Object.create( myCar );
```

```
// Now we can see that one is a prototype of the other
console.log( yourCar.name );
```

Object.create也允许我们简单的继承先进的概念,比如对象能够直接继承自其它对象,这种不同的继承.我们早先也看到Object.create允许我们使用 供应的第二个参数来初始化对象属性。例如:

```
var vehicle = {
  getModel: function () {
    console.log( "The model of this vehicle is.." + this.model );
  }
};

var car = Object.create(vehicle, {

  "id": {
    value: MY_GLOBAL.nextId(),
    // writable:false, configurable:false by default
    enumerable: true
  },

  "model": {
    value: "Ford",
    enumerable: true
  }

});
```

这里的属性可以被Object.create的第二个参数来初始化,使用一种类似于我们前面看到的Object.defineProperties和Object.defineProperties方法所使用语法的对象字面值。

在枚举对象的属性,和(如Crockford所提醒的那样)在一个hasOwnProperty()检查中封装循环的内容时,原型关系会造成麻烦,这一事实是值得我们关注的。

如果我们希望在不直接使用Object.create的前提下实现原型模式,我们可以像下面这样,按照上面的示例,模拟这一模式:

```
var vehiclePrototype = {

  init: function ( carModel ) {
    this.model = carModel;
  },

  getModel: function () {
    console.log( "The model of this vehicle is.." + this.model);
  }
```

```
};


function vehicle( model ) {

  function F() {};
  F.prototype = vehiclePrototype;

  var f = new F();

  f.init( model );
  return f;

}

var car = vehicle( "Ford Escort" );
car.getModel();
```

> 注意:这种可选的方式不允许用户使用相同的方式定义只读的属性(因为如果不小心的话vehicle原型可能会被改变)。

原型模式的最后一种可选实现可以像下面这样:

```
var beget = (function () {

    function F() {}

    return function ( proto ) {
        F.prototype = proto;
        return new F();
    };
})();
```

一个人可以从vehicle函数引用这个方法,注意,这里的那个vehicle正是在模拟着构造器,因为原型模式在将一个对象链接到一个原型之外没有任何初始化的概念。

命令模式

命名模式的目标是将方法的调用,请求或者操作封装到一个单独的对象中,给我们酌情执行同时参数化和传递方法调用的能力.另外,它使得我们能将对象从实现了行为的对象对这些行为的调用进行解耦,为我们带来了换出具体的对象这一更深程度的整体灵活性。

具体类是对基于类的编程语言的最好解释,并且同抽象类的理念联系紧密.抽象类定义了一个接口,但并不需要提供对它的所有成员函数的实现.它扮演着驱动其它类的基类角色.被驱动类实现了缺失的函数而被称为具体类. 命令模式背后的一般理念是为我们提供了从任何执行中的命令中分离出发出命令的责任,取而代之将这一责任委托给其它的对象。

实现明智简单的命令对象,将一个行为和对象对调用这个行为的需求都绑定到了一起.它们始终都包含一个执行操作(比如run()或者execute()).所有带有相同接口的命令对象能够被简单地根据需要调换,这被认为是命令模式的更大的好处之一。

为了展示命令模式,我们创建一个简单的汽车购买服务:

```
(function(){

  var CarManager = {

    // request information
    requestInfo: function( model, id ){
      return "The information for " + model + " with ID " + id + " is foobar";
    },

    // purchase the car
    buyVehicle: function( model, id ){
      return "You have successfully purchased Item " + id + ", a " + model;
    },

    // arrange a viewing
    arrangeViewing: function( model, id ){
      return "You have successfully booked a viewing of " + model + " ( " + id + " ) ";
    }

   };

})();
```

看一看上面的这段代码,它也许是通过直接访问对象来琐碎的调用我们CarManager的方法。在技术上我们也许都会都会对这个没有任何失误达成谅解.它是完全有效的Javascript然而也会有情况不利的情况。

例如,想象如果CarManager的核心API会发生改变的这种情况.这可能需要所有直接访问这些方法的对象也跟着被修改.这可以被看成是一种耦合,明显违背了OOP方法学尽量实现松耦合的理念.取而代之,我们可以通过更深入的抽象这些API来解决这个问题。

现在让我们来扩展我们的CarManager,以便我们这个命令模式的应用程序得到接下来的这种效果:接受任何可以在CarManager对象上面执行的方法,传送任何可以被使用到的数据,如Car模型和ID。

这里是我们希望能够实现的样子:

```
CarManager.execute( "buyVehicle", "Ford Escort", "453543" );
```

按照这种结构,我们现在应该像下面这样,添加一个对于"CarManager.execute()"方法的定义:

```
CarManager.execute = function ( name ) {
   return CarManager[name] && CarManager[name].apply( CarManager, [].slice.call(arguments, 1) );
};
```

最终我们的调用如下所示:

```
CarManager.execute( "arrangeViewing", "Ferrari", "14523" );
CarManager.execute( "requestInfo", "Ford Mondeo", "54323" );
CarManager.execute( "requestInfo", "Ford Escort", "34232" );
CarManager.execute( "buyVehicle", "Ford Escort", "34232" );
```

外观模式

当我们提出一个门面，我们要向这个世界展现的是一个外观，这一外观可能藏匿着一种非常与众不同的真实。这就是我们即将要回顾的模式背后的灵感——门面模式。这一模式提供了面向一种更大型的代码体提供了一个的更高级别的舒适的接口，隐藏了其真正的潜在复杂性。把这一模式想象成要是呈现给开发者简化的API，一些总是会提升使用性能的东西。

门面是一种经常可以在Javascript库中看到的结构性模式，像在jQuery中，尽管一种实现可能支持带有广泛行为的方法，但仅仅只有这些方法的“门面”或者说被限制住的抽象才会公开展现出来供人们所使用。

这允许我们直接同门面，而不是同幕后的子系统交互。不论何时我们使用jQuery的$(el).css或者$(el).animate()方法，我们实际上都是在使用一个门面——更加简单的公共接口让我们避免为了使得行为工作起来而不得不去手动调用jQuery核心的内置方法。这也避免了手动同DOM API交互和维护状态变量的需要。

应该考虑对jQuery的核心方法做一层中间抽象。对于开发者来说更直接的负担是DOM API，而门面使得jQuery使用起来如此的容易。

为了在我们所学的基础上进行构建，门面模式同时需要简化一个类的接口，和把类同使用它的代码解耦。这给予了我们使用一种方式直接同子系统交互的能力，这一方式有时候会比直接访问子系统更加不容易出错。门面的优势包括易用，还有常常实现起这个模式来只是一小段路，不费力。

让我们通过实践来看看这个模式。这是一个没有经过优化的代码示例，但是这里我们使用了一个门面来简化跨浏览器事件监听的接口。我们创建了一个公共的方法来实现，此方法 能够被用在检查特性的存在的代码中，以便这段代码能够提供一种安全和跨浏览器兼容方案。

```
var addMyEvent = function( el,ev,fn ){

  if( el.addEventListener ){
        el.addEventListener( ev,fn, false );
    }else if(el.attachEvent){
        el.attachEvent( "on" + ev, fn );
    } else{
        el["on" + ev] = fn;
  }

};
```

我们都熟知jQuery的$(document).ready(..)，使 用了一种类似的方式。在内部，这实际上是考一个叫做bindReady()的方法来驱动的，它做了一些这样的事：

```
bindReady: function() {
    ...
    if ( document.addEventListener ) {
      // Use the handy event callback
      document.addEventListener( "DOMContentLoaded", DOMContentLoaded, false );

      // A fallback to window.onload, that will always work
      window.addEventListener( "load", jQuery.ready, false );

    // If IE event model is used
    } else if ( document.attachEvent ) {

      document.attachEvent( "onreadystatechange", DOMContentLoaded );

      // A fallback to window.onload, that will always work
      window.attachEvent( "onload", jQuery.ready );
```

这是门面的另外一个例子，其它人只需要使用被$(document).ready(...)有限暴露的简单接口，而更加复杂的实现被从视野中隐藏了。

门面不仅仅只被用在它们自己身上，它们也能够被用来同其它的模式诸如模块模式进行集成。如我们在下面所看到的，我们模块模式的实体包含许多被定义为私有的方法。门面则被用来提供访问这些方法的更加简单的API：

```
var module = (function() {

    var _private = {
        i:5,
        get : function() {
            console.log( "current value:" + this.i);
        },
        set : function( val ) {
            this.i = val;
        },
        run : function() {
            console.log( "running" );
        },
        jump: function(){
            console.log( "jumping" );
        }
    };

    return {

        facade : function( args ) {
            _private.set(args.val);
```

```
            _private.get();
            if ( args.run ) {
                _private.run();
            }
        }
    };
}());


// Outputs: "current value: 10" and "running"
module.facade( {run: true, val:10} );
```

在这个示例中，调用module.facade()将会触发一堆模块中的私有方法。但再一次，用户并不需要关心这些。我们已经使得对用户而言不需要担心实现级别的细节就能消受一种特性。

关于抽象的注意事项

门面一般没有多少缺陷，但是性能是值得注意的问题。也就是说，需要确定门面在为我们提供实现的同时是否为我们带来了隐性的消耗，如果是这样的话，那么这种消耗是否合理。回到jQuery库，我们都知道getElementBy Id（’identifier‘）和$（“#identifier”）都能够被用来借助ID查找页面上的一个元素。

然而你是否知道getElementById（）拥有更高数量级的速度呢？来瞧瞧这个jsPerf的测试，看一看在每一个浏览器级别的结果：http://jsperf.com/getelementbyid-vs-jquery-id。当然现在，我们应该牢记在心的是jQuer y（和Sizzle-它的的选择器引擎）在幕后对我们的查询（而这返回的是一个jQuery对象，并不是一个DOM节点）做了更大量的优化。

这个特定的门面模式所面临的挑战就是，为了提供一种优雅的接受和转换多种查询类型的选择器功能，就会有在抽象上的隐性成本。用户并不需要访问jQuery.getById("identifier")或者jQuery.getbyClass("identifier")等等方法。那就是说，在性能上权衡已经通过了多年的实践考量，并且带了jQuery的成功，一个实际上为团队工作得很好的门面。

当使用这个模式的时候，尝试了解任何有关性能上面的消耗，要知道它们是否值得以抽象的级别被提供出来调用。

工厂模式

工厂模式是另外一种关注对象创建概念的创建模式。它的领域中同其它模式的不同之处在于它并没有明确要求我们使用一个构造器。取而代之，一个工厂能提供一个创建对象的公共接口，我们可以在其中指定我们希望被创建的工厂对象的类型。

试想一下，在我们被要求创建一种类型的UI组件时，我们就有一个UI工厂。并不是通过直接使用new操作符或者通过另外一个构造器来创建这个组件，我们取而代之的向一个工厂对象索要一个新的组件。我们告知工厂我们需要什么类型的组件（例如：“按钮”，“面板”），而它会将其初始化，然后返回供我们使用。

如果创建过程相当复杂的话，那这会特别的有用，例如：如果它强烈依赖于动态因素或者应用程序配置的话。

这个模式的一些例子可以在UI库里面找到，例如ExtJS, 用于创建对象或者组件的方法可以被做更深层次的子类。下面使用用我们之前的那些代码来做的一个例子，通过使用构造器模式逻辑来定义汽车。这个例子展示了Vehicle工厂可以使用工厂模式来实现。

```
// Types.js - Constructors used behind the scenes

// A constructor for defining new cars
function Car( options ) {

  // some defaults
  this.doors = options.doors || 4;
  this.state = options.state || "brand new";
  this.color = options.color || "silver";

}

// A constructor for defining new trucks
function Truck( options){

  this.state = options.state || "used";
  this.wheelSize = options.wheelSize || "large";
  this.color = options.color || "blue";
}


// FactoryExample.js

// Define a skeleton vehicle factory
function VehicleFactory() {}
```

```
// Define the prototypes and utilities for this factory

// Our default vehicleClass is Car
VehicleFactory.prototype.vehicleClass = Car;

// Our Factory method for creating new Vehicle instances
VehicleFactory.prototype.createVehicle = function ( options ) {

  if( options.vehicleType === "car" ){
    this.vehicleClass = Car;
  }else{
    this.vehicleClass = Truck;
  }

  return new this.vehicleClass( options );

};

// Create an instance of our factory that makes cars
var carFactory = new VehicleFactory();
var car = carFactory.createVehicle( {
            vehicleType: "car",
            color: "yellow",
            doors: 6 } );

// Test to confirm our car was created using the vehicleClass/prototype Car

// Outputs: true
console.log( car instanceof Car );

// Outputs: Car object of color "yellow", doors: 6 in a "brand new" state
console.log( car );
```

方法1: 修改 VehicleFactory 实例使用 Truck 类

```
var movingTruck = carFactory.createVehicle( {
                    vehicleType: "truck",
                    state: "like new",
                    color: "red",
                    wheelSize: "small" } );

// Test to confirm our truck was created with the vehicleClass/prototype Truck
```

```
// Outputs: true
console.log( movingTruck instanceof Truck );

// Outputs: Truck object of color "red", a "like new" state
// and a "small" wheelSize
console.log( movingTruck );
```

方法2: 做 VehicleFactory 的子类用于创建一个工厂类生产 Trucks

```
function TruckFactory () {}
TruckFactory.prototype = new VehicleFactory();
TruckFactory.prototype.vehicleClass = Truck;

var truckFactory = new TruckFactory();
var myBigTruck = truckFactory.createVehicle( {
                state: "omg..so bad.",
                color: "pink",
                wheelSize: "so big" } );

// Confirms that myBigTruck was created with the prototype Truck
// Outputs: true
console.log( myBigTruck instanceof Truck );

// Outputs: Truck object with the color "pink", wheelSize "so big"
// and state "omg. so bad"
console.log( myBigTruck );
```

何时使用工厂模式

当被应用到下面的场景中时，工厂模式特别有用：

- 当我们的对象或者组件设置涉及到高程度级别的复杂度时。
- 当我们需要根据我们所在的环境方便的生成不同对象的实体时。
- 当我们在许多共享同一个属性的许多小型对象或组件上工作时。
- 当带有其它仅仅需要满足一种API约定(又名鸭式类型)的对象的组合对象工作时.这对于解耦来说是有用的。

何时不要去使用工厂模式

当被应用到错误的问题类型上时,这一模式会给应用程序引入大量不必要的复杂性.除非为创建对象提供一个接口是我们编写的库或者框架的一个设计上目标,否则我会建议使用明确的构造器,以避免不必要的开销。

由于对象的创建过程被高效的抽象在一个接口后面的事实,这也会给依赖于这个过程可能会有多复杂的单元测试带来问题。

抽象工厂

了解抽象工厂模式也是非常实用的,它的目标是以一个通用的目标将一组独立的工厂进行封装.它将一堆对象的实现细节从它们的一般用例中分离。

抽象工厂应该被用在一种必须从其创建或生成对象的方式处独立,或者需要同多种类型的对象一起工作,这样的系统中。

简单且容易理解的例子就是一个发动机工厂,它定义了获取或者注册发动机类型的方式.抽象工厂会被命名为AbstractVehicleFactory.抽象工厂将允许像"car"或者"truck"的发动机类型的定义,并且构造工厂将仅实现满足发动机合同的类.(例如:Vehicle.prototype.driven和Vehicle.prototype.breakDown)。

```
var AbstractVehicleFactory = (function () {

  // Storage for our vehicle types
  var types = {};

  return {
    getVehicle: function ( type, customizations ) {
      var Vehicle = types[type];

      return (Vehicle ? new Vehicle(customizations) : null);
    },

    registerVehicle: function ( type, Vehicle ) {
      var proto = Vehicle.prototype;

      // only register classes that fulfill the vehicle contract
      if ( proto.drive && proto.breakDown ) {
        types[type] = Vehicle;
      }
```

```
        return AbstractVehicleFactory;
    }
  };
})();


// Usage:

AbstractVehicleFactory.registerVehicle( "car", Car );
AbstractVehicleFactory.registerVehicle( "truck", Truck );

// Instantiate a new car based on the abstract vehicle type
var car = AbstractVehicleFactory.getVehicle( "car" , {
        color: "lime green",
        state: "like new" } );

// Instantiate a new truck in a similar manner
var truck = AbstractVehicleFactory.getVehicle( "truck" , {
        wheelSize: "medium",
        color: "neon yellow" } );
```

Mixin 模式

在诸如C++或者List着这样的传统语言中,织入模式就是一些提供能够被一个或者一组子类简单继承功能的类,意在重用其功能。

子类划分

对于不熟悉子类划分的开发者,在深入织入模式和装饰器模式之前,我们将对他们进行一个简短的初学者入门指引。

子类划分是一个参考了为一个新对象继承来自一个基类或者超类对象的属性的术语.在传统的面向对象编程中,类B能够从另外一个类A处扩展.这里我们将A看做是超类,而将B看做是A的子类.如此,所有B的实体都从A处继承了其A的方法.然而B仍然能够定义它自己的方法,包括那些重载的原本在A中的定义的方法。

B是否应该调用已经被重载的A中的方法,我们将这个引述为方法链.B是否应该调用A(超类)的构造器,我们将这称为构造器链。

为了演示子类划分,首先我们需要一个能够创建自身新实体的基对象。

```
var Person =  function( firstName , lastName ){

  this.firstName = firstName;
  this.lastName =  lastName;
  this.gender = "male";

};
```

接下来,我们将制定一个新的类(对象),它是一个现有的Person对象的子类.让我们想象我们想要加入一个不同属性用来分辨一个Person和一个继承了Person"超类"属性的Superhero.由于超级英雄分享了一般人类许多共有的特征(例如:name,gender),因此这应该很有希望充分展示出子类划分是如何工作的。

```
// a new instance of Person can then easily be created as follows:
var clark = new Person( "Clark" , "Kent" );

// Define a subclass constructor for for "Superhero":
var Superhero = function( firstName, lastName , powers ){

   // Invoke the superclass constructor on the new object
   // then use .call() to invoke the constructor as a method of
   // the object to be initialized.
```

```
    Person.call( this, firstName, lastName );

    // Finally, store their powers, a new array of traits not found in a normal "Person"
    this.powers = powers;
};

SuperHero.prototype = Object.create( Person.prototype );
var superman = new Superhero( "Clark" ,"Kent" , ["flight","heat-vision"] );
console.log( superman );

// Outputs Person attributes as well as powers
```

Superhero构造器创建了一个自Peroson下降的对象。这种类型的对象拥有链中位于它之上的对象的属性,而且如果我们在Person对象中设置了默认的值,Superhero能够使用特定于它的对象的值覆盖任何继承的值。

Mixin(织入目标类)

在Javascript中,我们会将从Mixin继承看作是通过扩展收集功能的一种途径.我们定义的每一个新的对象都有一个原型,从其中它可以继承更多的属性.原型可以从其他对象继承而来,但是更重要的是,能够为任意数量的对象定义属性.我们可以利用这一事实来促进功能重用。

Mix允许对象以最小量的复杂性从它们那里借用(或者说继承)功能.作为一种利用Javascript对象原型工作得很好的模式,它为我们提供了从不止一个Mix处分享功能的相当灵活,但比多继承有效得多得多的方式。

它们可以被看做是其属性和方法可以很容易的在其它大量对象原型共享的对象.想象一下我们定义了一个在一个标准对象字面量中含有实用功能的Mixin,如下所示:

```
var myMixins = {

  moveUp: function(){
    console.log( "move up" );
  },

  moveDown: function(){
    console.log( "move down" );
  },

  stop: function(){
    console.log( "stop! in the name of love!" );
  }

};
```

然后我们可以方便的扩展现有构造器功能的原型,使其包含这种使用一个 如下面的score.js_.extends()方法辅助器的行为:

```
// A skeleton carAnimator constructor
function carAnimator(){
  this.moveLeft = function(){
    console.log( "move left" );
  };
}

// A skeleton personAnimator constructor
function personAnimator(){
  this.moveRandomly = function(){ /*..*/ };
}

// Extend both constructors with our Mixin
_.extend( carAnimator.prototype, myMixins );
_.extend( personAnimator.prototype, myMixins );

// Create a new instance of carAnimator
var myAnimator = new carAnimator();
myAnimator.moveLeft();
myAnimator.moveDown();
myAnimator.stop();

// Outputs:
// move left
// move down
// stop! in the name of love!
```

如我们所见,这允许我们将通用的行为轻易的"混"入相当普通对象构造器中。

在接下来的示例中,我们有两个构造器:一个Car和一个Mixin.我们将要做的是静Car参数化(另外一种说法是扩展),以便它能够继承Mixin中的特定方法,名叫driveForwar()和driveBackward().这一次我们不会使用Underscore.js。

取而代之，这个示例将演示如何将一个构造器参数化，以便在无需重复每一个构造器函数过程的前提下包含其功能。

```
// Define a simple Car constructor
var Car = function ( settings ) {

      this.model = settings.model || "no model provided";
      this.color = settings.color || "no colour provided";
```

```
    };

// Mixin
var Mixin = function () {};

Mixin.prototype = {

    driveForward: function () {
        console.log( "drive forward" );
    },

    driveBackward: function () {
        console.log( "drive backward" );
    },

    driveSideways: function () {
        console.log( "drive sideways" );
    }

};


// Extend an existing object with a method from another
function augment( receivingClass, givingClass ) {

    // only provide certain methods
    if ( arguments[2] ) {
        for ( var i = 2, len = arguments.length; i < len; i++ ) {
            receivingClass.prototype[arguments[i]] = givingClass.prototype[arguments[i]];
        }
    }
    // provide all methods
    else {
        for ( var methodName in givingClass.prototype ) {

            // check to make sure the receiving class doesn't
            // have a method of the same name as the one currently
            // being processed
            if ( !Object.hasOwnProperty(receivingClass.prototype, methodName) ) {
                receivingClass.prototype[methodName] = givingClass.prototype[methodName];
            }

            // Alternatively:
            // if ( !receivingClass.prototype[methodName] ) {
```

```
            //  receivingClass.prototype[methodName] = givingClass.prototype[methodName];
            // }
        }
    }
}


// Augment the Car constructor to include "driveForward" and "driveBackward"
augment( Car, Mixin, "driveForward", "driveBackward" );

// Create a new Car
var myCar = new Car({
    model: "Ford Escort",
    color: "blue"
});

// Test to make sure we now have access to the methods
myCar.driveForward();
myCar.driveBackward();

// Outputs:
// drive forward
// drive backward

// We can also augment Car to include all functions from our mixin
// by not explicitly listing a selection of them
augment( Car, Mixin );

var mySportsCar = new Car({
    model: "Porsche",
    color: "red"
});

mySportsCar.driveSideways();

// Outputs:
// drive sideways
```

优点&缺点

Mixin支持在一个系统中降解功能的重复性,增加功能的重用性.在一些应用程序也许需要在所有的对象实体共享行为的地方,我们能够通过在一个Mixin中维护这个共享的功能,来很容易的避免任何重复,而因此专注于只实现我们系统中真正彼此不同的功能。

也就是说,对Mixin的副作用是值得商榷的.一些开发者感觉将功能注入到对象的原型中是一个坏点子,因为它会同时导致原型污染和一定程度上的对我们原有功能的不确定性.在大型的系统中,很可能是有这种情况的。

我认为,强大的文档对最大限度的减少对待功能中的混入源的迷惑是有帮助的,而且对于每一种模式而言,如果在实现过程中小心行事,我们应该是没多大问题的。

装饰模式

装饰器是旨在提升重用性能的一种结构性设计模式。同Mixin类似，它可以被看作是应用子类划分的另外一种有价值的可选方案。

典型的装饰器提供了向一个系统中现有的类动态添加行为的能力。其创意是装饰本身并不关心类的基础功能，而只是将它自身拷贝到超类之中。

它们能够被用来在不需要深度改变使用它们的对象的依赖代码的前提下，变更我们希望向其中附加功能的现有系统之中。开发者使用它们的一个通常的理由是，它们的应用程序也许包含了需要大量彼此不相干类型对象的特性。想象一下不得不要去定义上百个不同对象的构造器，比方说，一个Javascript游戏。

对象构造器可以代表不同播放器类型，每一种类型具有不同的功能。一种叫做领主戒指的游戏会需要霍比特人、巫术师，兽人，巨兽，精灵，山岭巨人，乱世陆地等对象的构造器，而这些的数量很容易过百。而我们还要考虑为每一个类型的能力组合创建子类。

例如，带指环的霍比特人，带剑的霍比特人和插满宝剑的陆地等等。这并不是非常的实用，当我们考虑到不同能力的数量在不断增长这一因素时，最后肯定是不可控的。

装饰器模式并不去深入依赖于对象是如何创建的，而是专注于扩展它们的功能这一问题上。不同于只依赖于原型继承，我们在一个简单的基础对象上面逐步添加能够提供附加功能的装饰对象。它的想法是，不同于子类划分，我们向一个基础对象添加（装饰）属性或者方法，因此它会是更加轻巧的。

向Javascript中的对象添加新的属性是一个非常直接了当的过程，因此将这一特定牢记于心，一个非常简单的装饰器可以实现如下：

示例1：带有新功能的装饰构造器

```
// A vehicle constructor
function vehicle( vehicleType ){

    // some sane defaults
    this.vehicleType = vehicleType || "car";
    this.model = "default";
    this.license = "00000-000";

}

// Test instance for a basic vehicle
```

```
var testInstance = new vehicle( "car" );
console.log( testInstance );

// Outputs:
// vehicle: car, model:default, license: 00000-000

// Lets create a new instance of vehicle, to be decorated
var truck = new vehicle( "truck" );

// New functionality we're decorating vehicle with
truck.setModel = function( modelName ){
    this.model = modelName;
};

truck.setColor = function( color ){
    this.color = color;
};

// Test the value setters and value assignment works correctly
truck.setModel( "CAT" );
truck.setColor( "blue" );

console.log( truck );

// Outputs:
// vehicle:truck, model:CAT, color: blue

// Demonstrate "vehicle" is still unaltered
var secondInstance = new vehicle( "car" );
console.log( secondInstance );

// Outputs:
// vehicle: car, model:default, license: 00000-000
```

这种类型的简单实现是实用的，但它没有真正展示出装饰能够贡献出来的全部潜能。为这个，我们首先区分一下我的Coffee示例和Freeman，Sierra和Bates所著Head First Design Patterns这一本优秀的书中围绕Mackbook商店建立的模型，这两个之间的不同。

示例2：带有多个装饰器的装饰对象

```
// The constructor to decorate
function MacBook() {

  this.cost = function () { return 997; };
```

```
  this.screenSize = function () { return 11.6; };

}

// Decorator 1
function Memory( macbook ) {

  var v = macbook.cost();
  macbook.cost = function() {
    return v + 75;
  };

}

// Decorator 2
function Engraving( macbook ){

  var v = macbook.cost();
  macbook.cost = function(){
    return  v + 200;
  };

}

// Decorator 3
function Insurance( macbook ){

  var v = macbook.cost();
  macbook.cost = function(){
     return  v + 250;
  };

}

var mb = new MacBook();
Memory( mb );
Engraving( mb );
Insurance( mb );

// Outputs: 1522
console.log( mb.cost() );

// Outputs: 11.6
console.log( mb.screenSize() );
```

在上面的示例中，我们的装饰器重载了超类对象MacBook()的 object.cost()函数，使其返回的Macbook的当前价格加上了被定制后升级的价格。

这被看做是对原来的Macbook对象构造器方法的装饰，它并没有将其重写（例如，screenSize()），我们所定义的Macbook的其它属性也保持不变，完好无缺。

上面的示例并没有真正定义什么接口，而且我们也转移了从创造者到接受者移动时确保一个对象对应一个接口的责任。

伪古典装饰器

我们现在要来试试首见于Dustin Diaz与Ross Harmes合著的Pro Javascript Design Patterns（PJDP）中一种装饰器的变体。

不像早些时候的一些实例，Diaz和Harms坚持更加近似于其他编程语言（如Java或者C++)如何使用一种“接口”的概念来实现装饰器，我们不久就将对此进行详细的定义。

> 注意：装饰模式的这一特殊变体是提供出来做参考用的。如果发现它过于复杂，建议你选择前面更加简单的实现。

接口

PJDP所描述的装饰器是一种被用于将具备相同接口的对象进行透明封装的对象，这样一种模式。接口是一种定义一个对象应该具有哪些方法的途径，然而，它实际上并不指定那些方法应该如何实现。

它们也可以声明方法应该有些什么参数，但这被看做是可选项。

因此，为什么我们要在Javascript中使用接口呢？这个想法意在让它们具有自说明文档特性，并促进其重用性。在理论上，接口通过确保了其被改变的同时也要让其对象实现这些改变，从而使得代码更加的稳定。

下面是一个在Javascript中使用鸭式类型来实现接口的示例，鸭式类型是一种基于所实现的方法来帮助判定一个对象是否是一种构造器/对象的实体的方法。

```
// Create interfaces using a pre-defined Interface
// constructor that accepts an interface name and
// skeleton methods to expose.

// In our reminder example summary() and placeOrder()
// represent functionality the interface should
// support
```

```
var reminder = new Interface( "List", ["summary", "placeOrder"] );

var properties = {
  name: "Remember to buy the milk",
  date: "05/06/2016",
  actions:{
    summary: function (){
      return "Remember to buy the milk, we are almost out!";
   },
    placeOrder: function (){
      return "Ordering milk from your local grocery store";
    }
  }
};

// Now create a constructor implementing the above properties
// and methods

function Todo( config ){

  // State the methods we expect to be supported
  // as well as the Interface instance being checked
  // against

  Interface.ensureImplements( config.actions, reminder );

  this.name = config.name;
  this.methods = config.actions;

}

// Create a new instance of our Todo constructor

var todoItem = Todo( properties );

// Finally test to make sure these function correctly

console.log( todoItem.methods.summary() );
console.log( todoItem.methods.placeOrder() );

// Outputs:
// Remember to buy the milk, we are almost out!
// Ordering milk from your local grocery store
```

在上面的代码中，接口确保了实现提供严格的功能检查，而这个和接口构造器的接口代码能在这里找到。

使用接口最大的问题是，由于这并不是Javascript内置的对它们的支持，对我们而言就会存在尝试去模仿另外一种语言的特性，但看着并不完全合适，这样一种风险。然而对于没有太大性能消耗的轻量级接口是可以被使用的，并且下面我们将要看到的抽象装饰器同样使用了这个概念。

抽象装饰者

为了阐明这个版本的装饰者模式的结构，我们想象有一个超级类，还是一个Macbook模型，以及一个store，使我们可以用耗费额外费用的许多种增强来“装饰”Macbook。

增强可以包括升级到4GB或8GB的Ram，雕刻，或相似案例。如果现在我们要针对每一种增强选项的组合，使用单独的子类进行建模，可能看起来是这样的：

```
var Macbook = function(){
      //...
};

var  MacbookWith4GBRam =  function(){},
    MacbookWith8GBRam = function(){},
    MacbookWith4GBRamAndEngraving = function(){},
    MacbookWith8GBRamAndEngraving = function(){},
    MacbookWith8GBRamAndParallels = function(){},
    MacbookWith4GBRamAndParallels = function(){},
    MacbookWith8GBRamAndParallelsAndCase = function(){},
    MacbookWith4GBRamAndParallelsAndCase = function(){},
    MacbookWith8GBRamAndParallelsAndCaseAndInsurance = function(){},
    MacbookWith4GBRamAndParallelsAndCaseAndInsurance = function(){};
```

等等。

这不是一个实际的解决方案，因为一个新的子类可能需要具有每一种可能的增强组合。由于我们倾向于保持事物简单，不想维持一个巨大的子类集合，我们来看看怎样用装饰者更好的解决这个问题。

不需要我们前面看到的所有组合，我们只需要简单的创建五个新的装饰者类。对这些增强类的方法调用，将会传递给Macbook类。

在我们下一个例子中，装饰者透明的包装了它们的组件，而且有趣的是，可以在相同的接口互换。

这里是我们给Macbook定义的接口：

```
var Macbook = new Interface( "Macbook",
 ["addEngraving",
 "addParallels",
```

```
  "add4GBRam",
  "add8GBRam",
  "addCase"]);

// A Macbook Pro might thus be represented as follows:
var MacbookPro = function(){
    // implements Macbook
};

MacbookPro.prototype = {
    addEngraving: function(){
    },
    addParallels: function(){
    },
    add4GBRam: function(){
    },
    add8GBRam:function(){
    },
    addCase: function(){
    },
    getPrice: function(){
      // Base price
      return 900.00;
    }
};
```

为了使得我们稍后更加容易的添加所需的更多选项，一种带有被用来实现Mackbook接口的默认方法的抽象装饰器方法被定义了出来，其剩余的选项将会进行子类划分。抽象装饰器确保了我们能够独立于尽可能多的在不同的组合中所需的装饰器，去装饰一个基础类（记得早先的那个示例么？），而不需要去为了每一种可能的组合而去驱动一个类。

```
// Macbook decorator abstract decorator class

var MacbookDecorator = function( macbook ){

    Interface.ensureImplements( macbook, Macbook );
    this.macbook = macbook;

};

MacbookDecorator.prototype = {
    addEngraving: function(){
        return this.macbook.addEngraving();
    },
    addParallels: function(){
```

```
        return this.macbook.addParallels();
    },
    add4GBRam: function(){
        return this.macbook.add4GBRam();
    },
    add8GBRam:function(){
        return this.macbook.add8GBRam();
    },
    addCase: function(){
        return this.macbook.addCase();
    },
    getPrice: function(){
        return this.macbook.getPrice();
    }
};
```

上述示例中所发生的是Macbook装饰器在像组件一样的使用一个对象。它使用了我们早先定义的Macbook接口，对于每一个方法都调用了组件上相同的方法。我们现在就能够只使用Macbook装饰器来创建我们的选项类了——通过简单调用超类的构造器和根据需要可以被重载的方法。

```
var CaseDecorator = function( macbook ){

  // call the superclass's constructor next
  this.superclass.constructor( macbook );

};

// Let's now extend the superclass
extend( CaseDecorator, MacbookDecorator );

CaseDecorator.prototype.addCase = function(){
  return this.macbook.addCase() + "Adding case to macbook";
};

CaseDecorator.prototype.getPrice = function(){
  return this.macbook.getPrice() + 45.00;
};
```

如我们所见，大多数都是相对应的直接实现。我们所做的是重载需要被装饰的addCase()和getPrise()方法，而我们通过首先执行组件的方法然后将其添加到它里面，来达到目的。

鉴于到目前为止本节所介绍的信息一斤相当的多了，让我们试试将其全部放到一个单独的实例中，以期突出我们所学。

```
// Instantiation of the macbook
var myMacbookPro = new MacbookPro();

// Outputs: 900.00
console.log( myMacbookPro.getPrice() );

// Decorate the macbook
myMacbookPro = new CaseDecorator( myMacbookPro );

// This will return 945.00
console.log( myMacbookPro.getPrice() );
```

由于装饰器能够动态的修改对象，它们就是改变现有系统的理想模式。有时候，它只是简单的围绕一个对象及其维护针对每一个对象类型单独的子类划分所产生的麻烦，来创建装饰器的。这使得维护起可能需要大量子类划分对象的应用程序来更加显著的直接。

装饰器和jQuery

同我们所涵盖的其它模式一起，也有许多装饰器模式的示例能够使用jQuery来实现。jQuery.extend()允许我们将两个或者更多个对象（以及它们的属性）扩展（或者混合）到一个对象中，不论是在运行时或者动态的在一个稍后的时点上。

在这一场景中，目标对象没必要打断或者重载源/超类中现有的方法（尽管这可以被做到）就能够使用新的功能装饰起来。 在接下来的示例中，我们定义了三个对象：默认，选项和设置。任务的目标是用在选项中找到的附加功能来装饰默认对象。

- 将“默认”放置在一个不可触及的状态之中，在这里我们不会失去访问稍后会在其中发现的属性和方法的能力
- 赢得了使用在“选项”中找到被装饰起来的属性和函数的能力。

```
var decoratorApp = decoratorApp || {};

// define the objects we're going to use
decoratorApp = {

   defaults: {
      validate: false,
      limit: 5,
      name: "foo",
      welcome: function () {
         console.log( "welcome!" );
      }
```

```
    },

    options: {
        validate: true,
        name: "bar",
        helloWorld: function () {
            console.log( "hello world" );
        }
    },

    settings: {},

    printObj: function ( obj ) {
        var arr = [],
            next;
        $.each( obj, function ( key, val ) {
            next = key + ": ";
            next += $.isPlainObject(val) ? printObj( val ) : val;
            arr.push( next );
        } );

        return "{ " + arr.join(", ") + " }";
    }

};

// merge defaults and options, without modifying defaults explicitly
decoratorApp.settings = $.extend({}, decoratorApp.defaults, decoratorApp.options);

// what we have done here is decorated defaults in a way that provides
// access to the properties and functionality it has to offer (as well as
// that of the decorator "options"). defaults itself is left unchanged

$("#log")
    .append( decoratorApp.printObj(decoratorApp.settings) +
          + decoratorApp.printObj(decoratorApp.options) +
          + decoratorApp.printObj(decoratorApp.defaults));

// settings -- { validate: true, limit: 5, name: bar, welcome: function (){ console.log( "welcome!" ); },
// helloWorld: function (){ console.log("hello!"); } }
// options -- { validate: true, name: bar, helloWorld: function (){ console.log("hello!"); } }
// defaults -- { validate: false, limit: 5, name: foo, welcome: function (){ console.log("welcome!"); } }
```

优点 & 缺点

因为它可以被透明的使用，并且也相当的灵活，因此开发者都挺乐意去使用这个模式——如我们所见，对象可以用新的行为封装或者“装饰”起来，而后继续使用，并不用去担心基础的对象被改变。在一个更加广泛的范围内，这一模式也避免了我们去依赖大量子类来实现同样的效果。

然而在实现这个模式时，也存在我们应该意识到的缺点。如果穷于管理，它也会由于引入了许多微小但是相似的对象到我们的命名空间中，从而显著的使得我们的应用程序架构变得复杂起来。这里所担忧的是，除了渐渐变得难于管理，其他不能熟练使用这个模式的开发者也可能会有一段要掌握它被使用的理由的艰难时期。

足够的注释或者对模式的研究，对此应该有助益，而只要我们对在我们的应程序中的多大范围内使用这一模式有所掌控的话，我们就能让两方面都得到改善。

享元模式

享元模式是一个优化重复、缓慢和低效数据共享代码的经典结构化解决方案。它的目标是以相关对象尽可能多的共享数据，来减少应用程序中内存的使用(例如：应用程序的配置、状态等)。

此模式最先由Paul Calder 和 Mark Linton在1990提出，并用拳击等级中少于112磅体重的等级名称来命名。享元(“Flyweight”英语中的轻量级)的名称本身是从以帮以助我们完成减少重量(内存标记)为目标的重量等级推导出的。

实际应用中，轻量级的数据共享采集被多个对象使用的相似对象或数据结构，并将这些数据放置于单个的扩展对象中。我们可以把它传递给依靠这些数据的对象，而不是在他们每个上面都存储一次。

使用享元

有两种方法来使用享元。第一种是数据层，基于存储在内存中的大量相同对象的数据共享的概念。第二种是DOM层，享元模式被作为事件管理中心，以避免将事件处理程序关联到我们需要相同行为父容器的所有子节点上。 享元模式通常被更多的用于数据层，我们先来看看它。

享元和数据共享

对于这个应用程序而言，围绕经典的享元模式有更多需要我们意识到的概念。享元模式中有一个两种状态的概念——内在和外在。内在信息可能会被我们的对象中的内部方法所需要，它们绝对不可以作为功能被带出。外在信息则可以被移除或者放在外部存储。

带有相同内在数据的对象可以被一个单独的共享对象所代替，它通过一个工厂方法被创建出来。这允许我们去显著降低隐式数据的存储数量。

个中的好处是我们能够留心于已经被初始化的对象，让只有不同于我们已经拥有的对象的内在状态时，新的拷贝才会被创建。

我们使用一个管理器来处理外在状态。如何实现可以有所不同，但针对此的一种方法就是让管理器对象包含一个存储外在状态以及它们所属的享元对象的中心数据库。

经典的享元实现

近几年享元模式已经在Javascript中得到了深入的应用，我们会用到的许多实现方式其灵感来自于Java和C++的世界。

我们第一个要来看的关于享元模式的代码是我的对来自维基百科 (http://en.wikipedia.org/wiki/Flyweight_pattern) 的针对享元模式的 Java 示例的 Javascript 实现。

在这个实现中我们将要使用如下所列的三种类型的享元组件：

- 享元对应的是一个接口，通过此接口能够接受和控制外在状态。
- 构造享元来实际的实际的实现接口，并存储内在状态。构造享元须是能够被共享的，并且具有操作外在状态的能力。
- 享元工厂负责管理享元对象，并且也创建它们。它确保了我们的享元对象是共享的，并且可以对其作为一组对象进行管理，这一组对象可以在我们需要的时候查询其中的单个实体。如果一个对象已经在一个组里面创建好了，那它就会返回该对象，否则它会在对象池中新创建一个，并且返回之。

这些对应于我们实现中的如下定义：

- CoffeeOrder：享元
- CoffeeFlavor：构造享元
- CoffeeOrderContext：辅助器
- CoffeeFlavorFactory：享元工厂
- testFlyweight：对我们享元的使用

鸭式冲减的 “implements”

鸭式冲减允许我们扩展一种语言或者解决方法的能力，而不需要变更运行时的源。由于接下的方案需要使用一个Java关键字“implements”来实现接口，而在Javascript本地看不到这种方案，那就让我们首先来对它进行鸭式冲减。

Function.prototype.implementsFor 在一个对象构造器上面起作用，并且将接受一个父类（函数一）或者对象，而从继承于普通的继承（对于函数而言）或者虚拟继承（对于对象而言）都可以。

// Simulate pure virtual inheritance/"implement" keyword for JS Function.prototype.implementsFor = function(parentClassOrObject){ if (parentClassOrObject.constructor === Function) { // Normal Inherit

ance this.prototype = new parentClassOrObject(); this.prototype.constructor = this; this.prototype.parent = parentClassOrObject.prototype; } else { // Pure Virtual Inheritance this.prototype = parentClassOrObject; this.prototype.constructor = this; this.prototype.parent = parentClassOrObject; } return this; };

我们可以通过让一个函数明确的继承自一个接口来弥补implements关键字的缺失。下面，为了使我们得以去分配支持一个对象的这些实现的功能，CoffeeFlavor实现了CoffeeOrder接口，并且必须包含其接口的方法。

```
// Flyweight object
var CoffeeOrder = {

  // Interfaces
  serveCoffee:function(context){},
    getFlavor:function(){}

};


// ConcreteFlyweight object that creates ConcreteFlyweight
// Implements CoffeeOrder
function CoffeeFlavor( newFlavor ){

    var flavor = newFlavor;

    // If an interface has been defined for a feature
    // implement the feature
    if( typeof this.getFlavor === "function" ){
      this.getFlavor = function() {
          return flavor;
      };
    }

    if( typeof this.serveCoffee === "function" ){
      this.serveCoffee = function( context ) {
        console.log("Serving Coffee flavor "
          + flavor
          + " to table number "
          + context.getTable());
    };
    }

}
```

```
// Implement interface for CoffeeOrder
CoffeeFlavor.implementsFor( CoffeeOrder );


// Handle table numbers for a coffee order
function CoffeeOrderContext( tableNumber ) {
  return{
    getTable: function() {
      return tableNumber;
    }
  };
}


function CoffeeFlavorFactory() {
   var flavors = {},
   length = 0;

   return {
      getCoffeeFlavor: function (flavorName) {

         var flavor = flavors[flavorName];
         if (flavor === undefined) {
            flavor = new CoffeeFlavor(flavorName);
            flavors[flavorName] = flavor;
            length++;
         }
         return flavor;
      },

      getTotalCoffeeFlavorsMade: function () {
         return length;
      }
   };
}

// Sample usage:
// testFlyweight()

function testFlyweight(){


  // The flavors ordered.
  var flavors = new CoffeeFlavor(),
```

```
 // The tables for the orders.
  tables = new CoffeeOrderContext(),

 // Number of orders made
  ordersMade = 0,

 // The CoffeeFlavorFactory instance
  flavorFactory;

 function takeOrders( flavorIn, table) {
   flavors[ordersMade] = flavorFactory.getCoffeeFlavor( flavorIn );
   tables[ordersMade++] = new CoffeeOrderContext( table );
 }

  flavorFactory = new CoffeeFlavorFactory();

  takeOrders("Cappuccino", 2);
  takeOrders("Cappuccino", 2);
  takeOrders("Frappe", 1);
  takeOrders("Frappe", 1);
  takeOrders("Xpresso", 1);
  takeOrders("Frappe", 897);
  takeOrders("Cappuccino", 97);
  takeOrders("Cappuccino", 97);
  takeOrders("Frappe", 3);
  takeOrders("Xpresso", 3);
  takeOrders("Cappuccino", 3);
  takeOrders("Xpresso", 96);
  takeOrders("Frappe", 552);
  takeOrders("Cappuccino", 121);
  takeOrders("Xpresso", 121);

  for (var i = 0; i < ordersMade; ++i) {
     flavors[i].serveCoffee(tables[i]);
  }
  console.log(" ");
  console.log("total CoffeeFlavor objects made: " +  flavorFactory.getTotalCoffeeFlavorsMade());
} <span style="line-height:1.5;font-family:'sans serif', tahoma, verdana, helvetica;font-size:10pt;"></span>
```

转换代码为使用享元模式

接下来，让我们通过实现一个管理一个图书馆中所有书籍的系统来继续观察享元。分析得知每一本书的重要元数据如下：

- ID
- 标题
- 作者
- 类型
- 总页数
- 出版商ID
- ISBN

我们也将需要下面一些属性，来跟踪哪一个成员是被借出的一本特定的书，借出它们的日期，还有预计的归还日期。

- 借出日期
- 借出的成员
- 规定归还时间
- 可用性

```
var Book = function( id, title, author, genre, pageCount,publisherID, ISBN, checkoutDate, checkoutMember, dueReturnDa

  this.id = id;
  this.title = title;
  this.author = author;
  this.genre = genre;
  this.pageCount = pageCount;
  this.publisherID = publisherID;
  this.ISBN = ISBN;
  this.checkoutDate = checkoutDate;
  this.checkoutMember = checkoutMember;
  this.dueReturnDate = dueReturnDate;
  this.availability = availability;

};

Book.prototype = {

 getTitle: function () {
   return this.title;
 },

 getAuthor: function () {
```

```
    return this.author;
  },

  getISBN: function (){
    return this.ISBN;
  },

  // For brevity, other getters are not shown
  updateCheckoutStatus: function( bookID, newStatus, checkoutDate , checkoutMember, newReturnDate ){

    this.id  = bookID;
    this.availability = newStatus;
    this.checkoutDate = checkoutDate;
    this.checkoutMember = checkoutMember;
    this.dueReturnDate = newReturnDate;

  },

  extendCheckoutPeriod: function( bookID, newReturnDate ){

     this.id =  bookID;
     this.dueReturnDate = newReturnDate;

  },

  isPastDue: function(bookID){

    var currentDate = new Date();
    return currentDate.getTime() > Date.parse( this.dueReturnDate );

   }
};
```

这对于最初小规模的藏书可能工作得还好，然而当图书馆扩充至每一本书的多个版本和可用的备份，这样一个大型的库存，我们会发现管理系统的运行随着时间的推移会越来越慢。使用成千上万的书籍对象可能会压倒内存，而我们可以通过享元模式的提升来优化我们的系统。

现在我们可以像下面这样将我们的数据分离成为内在和外在的状态：同书籍对象（标题，版权归属）相关的数据是内在的，而借出数据（借出成员，规定归还日期）则被看做是外在的。这实际上意味着对于每一种书籍属性的组合仅需要一个书籍对象。这仍然具有相当大的数量，但相比之前已经得到大大的缩减了。

下面的书籍元数据组合的单一实体将在所有带有一个特定标题的书籍拷贝中共享。

```
// Flyweight optimized version
var Book = function ( title, author, genre, pageCount, publisherID, ISBN ) {
```

```
    this.title = title;
    this.author = author;
    this.genre = genre;
    this.pageCount = pageCount;
    this.publisherID = publisherID;
    this.ISBN = ISBN;

};
```

如我们所见，外在状态已经被移除了。从图书馆借出所要做的一切都被转移到一个管理器中，由于对象数据现在是分段的，工厂可以被用来做实例化。

一个基本工厂

现在让我们定义一个非常基本的工厂。我们用它做的工作是，执行一个检查来看看一本给定标题的书是不是之前已经在系统内创建过了；如果创建过了，我们就返回它 – 如果没有，一本新书就会被创建并保存，使得以后可以访问它。这确保了为每一条本质上唯一的数据，我们只创建了一份单一的拷贝：

```
// Book Factory singleton
var BookFactory = (function () {
  var existingBooks = {}, existingBook;

  return {
    createBook: function ( title, author, genre, pageCount, publisherID, ISBN ) {

      // Find out if a particular book meta-data combination has been created before
      // !! or (bang bang) forces a boolean to be returned
      existingBook = existingBooks[ISBN];
      if ( !!existingBook ) {
        return existingBook;
      } else {

        // if not, let's create a new instance of the book and store it
        var book = new Book( title, author, genre, pageCount, publisherID, ISBN );
        existingBooks[ISBN] = book;
        return book;

      }
    }
  };

});
```

管理外在状态

下一步，我们需要将那些从Book对象中移除的状态存储到某一个地方——幸运的是一个管理器（我们会将其定义成一个单例）可以被用来封装它们。书籍对象和借出这些书籍的图书馆成员的组合将被称作书籍借出记录。这些我们的管理器都将会存储，并且也包含我们在对Book类进行享元优化期间剥离的同借出相关的逻辑。

```
// BookRecordManager singleton
var BookRecordManager = (function () {

  var bookRecordDatabase = {};

  return {
    // add a new book into the library system
    addBookRecord: function ( id, title, author, genre, pageCount, publisherID, ISBN, checkoutDate, checkoutMember, due

      var book = bookFactory.createBook( title, author, genre, pageCount, publisherID, ISBN );

      bookRecordDatabase[id] = {
        checkoutMember: checkoutMember,
        checkoutDate: checkoutDate,
        dueReturnDate: dueReturnDate,
        availability: availability,
        book: book
      };
    },
    updateCheckoutStatus: function ( bookID, newStatus, checkoutDate, checkoutMember, newReturnDate ) {

      var record = bookRecordDatabase[bookID];
      record.availability = newStatus;
      record.checkoutDate = checkoutDate;
      record.checkoutMember = checkoutMember;
      record.dueReturnDate = newReturnDate;
    },

    extendCheckoutPeriod: function ( bookID, newReturnDate ) {
      bookRecordDatabase[bookID].dueReturnDate = newReturnDate;
    },

    isPastDue: function ( bookID ) {
      var currentDate = new Date();
      return currentDate.getTime() > Date.parse( bookRecordDatabase[bookID].dueReturnDate );
    }
  };
```

```
});
```

这些改变的结果是所有从Book类中撷取的数据现在被存储到了BookManager单例（BookDatabase）的一个属性之中——与我们以前使用大量对象相比可以被认为是更加高效的东西。同书籍借出相关的方法也被设置在这里，因为它们处理的数据是外在的而不内在的。

这个过程确实给我们最终的解决方法增加了一点点复杂性，然而同已经明智解决的数据性能问题相比，这只是一个小担忧，如果我们有同一本书的30份拷贝，现在我们只需要存储它一次就够了。每一个函数也会占用内存。使用享元模式这些函数只在一个地方存在（就是在管理器上），并且不是在每一个对象上面，这节约了内存上的使用。

享元模式和DOM

DOM（文档对象模型）支持两种允许对象侦听事件的方法——自顶向下（事件捕获）或者自底向下（时间冒泡）。 在事件捕获中，事件一开始会被最外面的元素捕获，并且传播到最里面的元素。在事件冒泡中，事件被捕获并且被赋给了最里面的元素，然后传播到最外面的元素。

在此背景下描述享元模式的最好隐喻来自Gary Chisholm写的文章，这里摘录了一点点：

> 尝试用一种池塘的方式思考享元模式。一只鱼张开了它的嘴巴（事件发生了），泡泡一直要上升到表面（冒泡），当泡泡到达表面时，停泊在顶部的一直苍蝇飞走了（动作执行）。在这个示例中我们能够很容易的将鱼张开嘴巴转换为按钮被点击了一下，将泡泡转换为冒泡效果，而苍蝇飞走了表示一些需要运行的函数。

冒泡被引入用来处理单个事件（比如：一次点击）可能会由在DOM层级中的不同级别的多个事件处理器处理，这样的场景。这在哪里发生了，事件冒泡就会为在尽可能最低的级别定义的事件处理器执行。从那里开始，事件向上冒泡，一直到包含比应该包含的更高层级的元素。

享元模式可用来进一步调整事件冒泡过程，这我们很快就将会看到。

例子1：集中式事件处理

一起来看看我们第一例子，当用户有个动作(如点击或是鼠标移动)时我们将有很多相似的文档对象以及相似的行为要处理。一般情况下，当我们构建手风琴式控件，菜单以及其它列表控件时，就会在每一个超链接元素父容器里绑定点击事件(如，$('ul li a').on(..)(jQuery代码,译者注))。我们可以方便的在可以监听事件容器里添加Flyweight，而不是在很多元素里绑定点击事件。这样就可处理或是简单或是复杂的需求。

提到组件的类型，经常会涉及到很多部分都有同样重复的标签(如，手风琴式控件)，这是个好机会，每个元素都有可能被点击的行为，而且基本上用相同的类。我们可以用Flyweight来构建一个基本的手风琴控件。

这里我们使用一个stateManager命名空间来封装我们的享元逻辑，同时使用jQuery来把初始点击事件绑定到一个div容器上。为了确保页面上没有其他程序逻辑把类似的处理器绑定到该容器上，首先使用了一个unbind事件。

现在明确的确立一下容器中的那个子元素会被点击，我们使用一次对target的检查来提供对被点击元素的引用，而不管它的父元素是谁。然后我们利用该信息来处理点击事件，而实际上不需要在页面装载时把该事件绑定到具体的子元素上。

HTML

```
<div id="container">
  <div class="toggle" href="#">More Info (Address)
    <span class="info">
      This is more information
    </span></div>
  <div class="toggle" href="#">Even More Info (Map)
    <span class="info">
      <iframe src="http://www.map-generator.net/extmap.php?name=London&amp;address=london%2C%20england&a
    </span>
  </div>
</div>
```

JavaScript

```
var stateManager = {

  fly: function () {

    var self = this;

    $( "#container" ).unbind().on( "click" , function ( e ) {
      var target = $( e.originalTarget || e.srcElement );
      if ( target.is( "div.toggle") ) {
        self.handleClick( target );
      }
    });
  },

  handleClick: function ( elem ) {
    elem.find( "span" ).toggle( "slow" );
  }
};
```

这样做的好处是，我们把许多不相关的动作转换为一个可以共享的动作(也许会保存在内存中)。

示例2：使用享元进行性能优化

在我们的第二个示例中，我们将会引述通过使用jQuery的享元可以获得的一些更多的性能上的收获。

Jame Padolsey 以前写过一篇叫做76比特的文章，讲述更快的jQuery，在其中他提醒我们每一次jQuery触发了一个回调，不管是什么类型（过滤器，每一个，事件处理器），我们都能够通过this关键字访问函数的上下文（与它相关的DOM元素）。

不幸的是，我们中的许多人已经习惯将this封装到$()或者jQuery()中的想法，这意味着新的jQuery实体没必要每次都被构造出来，而是简单的这样做：

```
$("div").on( "click", function () {
  console.log( "You clicked: " + $( this ).attr( "id" ));
});

// we should avoid using the DOM element to create a
// jQuery object (with the overhead that comes with it)
// and just use the DOM element itself like this:

$( "div" ).on( "click", function () {
  console.log( "You clicked:"  + this.id );
});
```

James想要下面的场景中使用jQuery的jQuery.text，然而他不能苟同一个新的jQuery对象必须在每次迭代中创建的概念。

```
$( "a" ).map( function () {
  return $( this ).text();
});
```

现在就使用jQuery的工具方法进行多余的包装而言，使用jQuery.methodName(如，jQuery.text）比jQuery.fn.methodName（如，jQuery.fn.text）更好，这里methodName代表了一种使用的工具，如each()或者text。这避免了调用更深远级别的抽象，或者每一次当我们的函数被调用时就构造一个新的jQuery对象，因为定义了jQuery.methodName的库本身在更底层使用jQuery.fn.methodName驱动的。

然而由于并不是所有jQuery的方法都有相应的单节点功能，Padolsey根据这个创意设计了jQuery.single工具。这里的创意是一个单独的jQuery对象会被被创建出来并且用于每一次对jQuery.single的调用（有意义的是仅有一个jQuery对象会被创建出来）。对于此的实现可以在下面看到，而且由于我们将来自多个可能的对象的数据整合到一个更加集中的单一结构中，技术上讲，它也是一个享元。

```
jQuery.single = (function( o ){
```

```
  var collection = jQuery([1]);
  return function( element ) {

    // Give collection the element:
    collection[0] = element;

     // Return the collection:
    return collection;

  };
});
```

对于这个的带有调用链的动作的示例如下：

```
$( "div" ).on( "click", function () {

  var html = jQuery.single( this ).next().html();
  console.log( html );

});
```

> 注意：尽管我们可能相信通过简单的缓存我们的jQuery代码会提供出同等良好的性能收获，但Padolsey声称$.single()仍然值得使用，且表现更好。那并不是说不使用任何的缓存，只要对这种方法的助益做到心里有数就行。想要对$.single有更加详细的了解，建议你却读一读Padolsey完整的文章。

3

JavaScript MV* 模式

MVC

MVC是一个架构设计模式，它通过分离关注点的方式来支持改进应用组织方式。它促成了业务数据(Models)从用户界面(Views)中分离出来，还有第三个组成部分(Controllers)负责管理传统意义上的业务逻辑和用户输入。该模式最初是由Trygve Reenskaug (http://en.wikipedia.org/wiki/Trygve_Reenskaug) 在研发Smalltalk-80 (1979)期间设计的，当时它起初被称作Model-View-Controller-Editor。在1995年的“设计模式: 面向对象软件中的可复用元素” (http://www.amazon.co.uk/Design-patterns-elements-reusable-object-oriented/dp/0201633612) (著名的"GoF"的书)中，MVC被进一步深入的描述，该书对MVC的流行使用起到了关键作用。

Smalltalk-80 MVC

了解一下最初的MVC模式打算解决什么问题是很重要的，因为自从诞生之日起它已经发生了很大的改变。回到70年代，图形用户界面还很稀少，一个被称为分离展示的概念开始被用来清晰的划分下面两种对象：领域对象，它对现实世界里的概念进行建模(比如一张照片，一个人)， 还有展示对象，它被渲染到用户屏幕上进行展示。

Smalltalk-80作为MVC的实现，把这一概念进一步发展，产生这样一个观点，即把应用逻辑从用户界面中分离开来。这种想法使得应用的各个部分之间得以解耦，也允许该应用中的其它界面对模型进行复用。关于Smalltalk-80的MVC架构，有几点很有趣，值得注意一下:

- 模型表现了领域特定的数据,并且不用考虑用户界面(视图和控制器).当一个模型有所改变的时候,它会通知它的观察者。
- 视图表现了一个模型的当前状态.观察者模式被用来让视图在任何时候都知晓模型已经被更新了或者被改变了。
- 展现受到视图的照管,但是不仅仅只有一个单独的视图或者控制器——每一个在屏幕上展现的部分或者元素都需要一个视图-控制器对。
- 控制器在这个视图-控制器对中扮演着处理用户交互的角色（比如按键或者点击动作），做出对视图的选择。

开发者有时候会惊奇于他们了解到的观察者模式（如今已经被普遍的作为发布/订阅的变异实现了）已经在几十年以前被作为MVC架构的一部分包含进去了.在Smalltalk-80的 MVC中,视图观察着模型.如上面要点中所提到的,模型在任何时候发生了改变,视图就会做出响应.一个简单的示例就是一个由股票市场数据支撑的应用程序——为了应用程序的实用性，任何对于我们模型中数据的改变都应该导致视图中的结果实时的刷新。

Martin Fowler在过去数年完成了对原生MVC有关问题进行写作的优秀工作，如果对关于Smalltalk-80的MVC的更深入的历史信息感兴趣的话，我建议您读一读他的作品。

JavaScript 开发者可以使用的 MVC

我们已经回顾了70年代，让我们回到当下回到眼前。现在，MVC模式已经被应用到大范围的编程语言当中，包括与我们关系最近的JavaScript。JavaScript领域现在有一些鼓励支持MVC (或者是它的变种，我们称之为 MV* 家族)的框架，允许开发者不用付出太多的努力就可以往他们的应用中添加新的结构。

这些框架包括诸如Backbone, Ember.js和AngularJS。考虑到避免出现“意大利面条”式的代码的重要性，该词是指那些由于缺乏结构设计而导致难于阅读和维护的代码，对现代JavaScript开发者来说，了解该模式能够提供什么已经是势在必行。这使得我们可以有效的领会到，这些框架能让我们以不同的方式做哪些事情。

我们知道MVC由三个核心部分组成:

Models

Models管理一个业务应用的数据。它们既与用户界面无关也与表现层无关，相反的它们代表了一个业务应用所需要的形式唯一的数据。当一个model改变时(比如当它被更新时)，它通常会通知它的观察者(比如我们很快会介绍的views)一个改变已经发生了，以便观察者采取相应的反应。

为了更深的理解models，让我们假设我们有一个JavaScript的相册应用。在一个相册中，照片这个概念配得上拥有一个自己的model， 因为它代表了特定领域数据的一个独特类型。这样一个model可以包含一些相关的属性，比如标题，图片来源和额外的元数据。一张特定的照片可以存储到model的一个实例中，而且一个model也可以被复用。下面我们可以看到一个用Backbone实现的被简化的model例子。

```
var Photo = Backbone.Model.extend({

  // 照片的默认属性
  defaults: {
    src: "placeholder.jpg",
    caption: "A default image",
    viewed: false
  },

  // 确保每一个被创建的照片都有一个`src`.
  initialize: function() {
     this.set( { "src": this.defaults.src} );
  }

});
```

不同的框架其内置的模型的能力有所不同,然而他们对于属性验证的支持还是相当普遍的,属性展现了模型的特征,比如一个模型标识符.当在一个真实的世界使用模型的时候,我们一般也希望模型能够持久.持久化允许我们用最近的状态对模型进行编辑和更新,这一状态会存储在内存、用户的本地数据存储区或者一个同步的数据库中。

另外，模型可能也会被多个视图观察着。如果说，我们的照片模型包含了一些元数据，比如它的位置（经纬度），照片中所展现的好友（一个标识符的列表）和一个标签的列表，开发者也许会选择为这三个方面的每一个提供一个单独的视图。

为现代 MVC/MV* 框架提供一种将模型组合到一起的方法（例如，在Backbone中，这些分组作为“集合”被引用）并不常见。管理分组中的模型允许我们基于来自分组中所包含的模型发生改变的通知,来编写应用程序逻辑.这避免了手动设置去观察每一个单独的模型实体的必要。

如下是一个将模型分组成一个简化的Backbone集合的示例:

```
var PhotoGallery = Backbone.Collection.extend({

  // Reference to this collection's model.
  model: Photo,

  // Filter down the list of all photos
  // that have been viewed
  viewed: function() {
    return this.filter(function( photo ){
      return photo.get( "viewed" );
    });
  },

  // Filter down the list to only photos that
  // have not yet been viewed
  unviewed: function() {
   return this.without.apply( this, this.viewed() );
  }
});
```

MVC上旧的文本可能也包含了模型管理着应用程序状态的一种概念的引述.Javascript中的应用程序状态有一种不同的意义,通常指的是当前的"状态",即在一个固定点上的用户屏幕上的视图或者子视图(带有特定的数据).状态是一个经常被谈论到的话题,看一看单页面应用程序,其中的状态的概念需要被模拟。

总而言之,模型主要关注的是业务数据。

视图

视图是模型的可视化表示，提供了一个当前状态的经过过滤的视图。Smaltalk的视图是关于绘制和操作位图的，而JavaScript的视图是关于构建和操作DOM元素的。

一个视图通常是模型的观察者，当模型改变的时候，视图得到通知，因此使得视图可以更新自身。用设计模式的语言可以称视图为“哑巴”，因为在应用程序中是它们关于模型和控制器的了解是受到限制的。

用户可以和视图进行交互，包括读和编辑模型的能力（例如，获取或者设置模型的属性值）。因为视图是表示层，我们通常以用户友好的方式提供编辑和更新的能力。例如，在之前我们讨论的照片库应用中，模型编辑可以通过“编辑”视图来进行，这个视图里面，用户可以选择一个特定的图片，接着编辑它的元数据。

而实际更新模型的任务落到了控制器上面（我们很快就会讲这个东西）。

让我们使用vanilla JavaScript 实现的例子来更深入的探索一下视图。下面我们可以看到一个函数创建了一个照片视图，使用了模型实例和控制器实例。

我们在视图里定义了一个render()工具，使用一个JavaScript模板引擎来用于渲染照片模型的内容（Underscore的模板），并且更新了我们视图的内容，供照片El来参考。

照片模型接着将我们的render()函数作为一个其一个订阅者的回调函数，这样通过观察者模式，当模型发生改变的时候，我们就能触发视图的更新。

人们可能会问用户交互如何在这里起作用的。当用户点击视图中的任何元素，不是由视图决定接下来怎么做。而是由控制器为视图做决定。在我们的例子中，通过为photoEl增加一个事件监听器，来达到这个目的，photoEl将会代理处理送往控制器的点击行为，在需要的时候将模型信息和事件一并传递。

这个架构的好处是每个组件在应用工作的时候都扮演着必要的独立的角色。

```
var buildPhotoView = function ( photoModel, photoController ) {

  var base = document.createElement( "div" ),
     photoEl = document.createElement( "div" );

  base.appendChild(photoEl);

  var render = function () {
        // We use a templating library such as Underscore
        // templating which generates the HTML for our
        // photo entry
        photoEl.innerHTML = _.template( "#photoTemplate" , {
```

```
            src: photoModel.getSrc()
        });
    };

  photoModel.addSubscriber( render );

  photoEl.addEventListener( "click", function () {
    photoController.handleEvent( "click", photoModel );
  });

  var show = function () {
    photoEl.style.display = "";
  };

  var hide = function () {
    photoEl.style.display = "none";
  };

  return {
    showView: show,
    hideView: hide
  };

};
```

模板

在支持 MVC/MV* 的JavaScript框架的下，有必要简略的讨论一下JavaScript的模板以及它们与视图之间的关系，在上一小节，我们已经接触到这种关系了。

历史已经证明在内存中通过字符串拼接来构建大块的HTML标记是一种糟糕的性能实践。开发者这样做，就会深受其害。遍历数据，将其封装成嵌套的div，使用例如document.writeto 这样过时的技术将"模板"注入到DOM中。这样通常意味着校本化的标记将会嵌套在我们标准的标记中，很快就变得很难阅读了，更重要的是，维护这样的代码将是一场灾难，尤其是在构建大型应用的时候。

JavaScript 模板解决方案（例如Handlebars.js 和Mustache）通常用于为视图定义模板作为标记（要么存储在外部，要么存储在脚本标签里面，使用自定义的类型例如text/template），标记中包含有模板变量。变量可以使用变化的语法来分割（例如{{name}}），框架通常也足够只能接受JSON格式的数据（模型可以转化成JSOn格式），这样我们只需要关心如何维护干净的模型和干净的模板。人们遭遇的绝大多数的苦差事都被框架本身所处理了。这样做有大量的好处，尤其选择是将模板存储在外部的时候，这样在构建大型引应用的时候可以是模板按照需要动态加载。

下面我们可以看到两个HTMP模板的例子。一个使用流行的Handlebar.js框架实现，一个使用Underscore模板实现。

Handlebars.js

```
<li class="photo">
  <h2>{{caption}}</h2>
  <img class="source" src="{{src}}"/>
  <div class="meta-data">
    {{metadata}}
  </div>
</li>
```

Underscore.js Microtemplates

```
<li class="photo">
  <h2><%= caption %></h2>
  <img class="source" src="<%= src %>"/>
  <div class="meta-data">
    <%= metadata %>
  </div>
</li><span style="line-height:1.5;font-family:'sans serif', tahoma, verdana, helvetica;font-size:10pt;"></span>
```

请注意模板并不是它们自身的视图，来自于Struts Model 2 架构的开发者可能会感觉模板就是一个视图，但并不是这样的。视图是一个观察着模型的对象，并且让可视的展现保持最新。模板也许是用一种声明的方式指定部分甚至所有的视图对象，因此它可能是从模板定制文档生成的。

在经典的web开发中，在单独的视图之间进行导航需要利用到页面刷新，然而也并不值得这样做。而在单页面Javascript应用程序中，一旦数据通过ajax从服务器端获取到了，并不需要任何这样必要的刷新，就可以简单的在同一个页面渲染出一个新的视图。

这里导航就降级为了“路由”的角色，用来辅助管理应用程序状态（例如，允许用户用书签标记它们已经浏览到的视图）。然而，路由既不是MVC的一部分，也不在每一个类MVC框架中展现出来，在这一节中我将不深入详细的讨论它们。

总而言之，视图是对我们的数据的一种可视化展现。

控制器

控制器是模型和视图之间的中介，典型的职责是当用户操作视图的时候同步更新模型。

在我们的照片廊应用程序中，控制器会负责处理用户通过对一个特定照片的视图进行编辑所造成改变，当用户完成编辑后，就更新一个特定的照片模型。

请记住满足了MVC中的一种角色：针对视图的策略模式的基础设施。在策略模式方面，视图在视图的自由载量权方面代表了控制器。因此，那就是测试模式是如何工作的，视图可以代表针对控制器的用户事件，当视图看起来合适的时候。视图也可以代表针对控制器的模型变更事件处理，当视图看起来合适的时候，但这并不是控制器的传统角色。

大多数的Javascript MVC框架都受到了对"MVC"通常认知的影响,而这种认知是和控制器绑定在一起的.出现这种情况的原因各异,但在我的真实想法中,那是由于框架的作者一开始就将从服务器端的角度看待MVC,意识到它并不在客户端进行1:1的翻译,而对MVC中的C进行重新诠释意在他们感觉更加有意义的事情.与此同在的问题在于它是主观的,增加了理解经典MVC模式的复杂度,当然还有控制器在现代框架中的角色。

作为示例,让我们来简要回顾一下当前流行的一种构造框架Backbone.js其架构.Backbone包含了模型和视图(某些东西同我们前面看到的类似),然而它实际上并没有真正的控制器.它的视图和路由行为同控制器有一点点类似,但它们自身实际上都不是控制器。

在这一方面,同官方文档或者博客文章中可能提到的相左,Backbone既不是一个真正的MVC/MVP框架,也不是一个MVVM框架.事实上把它看做是用它自身的方式架构方法的 `MV*` 家族中的一员,更加合适.当然这没有任何错误的地方,但区分经典MVC和 `MV*` 是重要的,我们应该依靠前者的经典语法来帮助理解后者。

Spine.js VS Backbone.js

Spine.js

我们现在知道传统的控制器负责当用户更新视图是同步更新模型.值得注意的一个有趣的地方是大多数时下流行的Javascript MVC/MV*框架在编写的时候(Backbone)都没有属于它们自己的明确的控制器的概念。

因此,这对于我们从另一个MVC框架中体会到控制器实现的差异,并更进一步的展现出控制如何扮演着非传统的角色是很有用处的.对于这一点,让我们来看看来自于Spine.js的示例控制器。

在这个示例中,我们会有一个叫做PhotosController的控制器，用来管理应用程序中的个人照片.它将确保当视图更新(例如,一个用户编辑了照片的元数据)时,对应的模型也会更新。

注意:我们并不会花大力气研究Spine.js,而只是对它的控制器能做什么进行一定程度的了解:

```
// Controllers in Spine are created by inheriting from Spine.Controller

var PhotosController = Spine.Controller.sub({

  init: function () {
    this.item.bind( "update" , this.proxy( this.render ));
    this.item.bind( "destroy", this.proxy( this.remove ));
  },

  render: function () {
    // Handle templating
    this.replace( $( "#photoTemplate" ).tmpl( this.item ) );
    return this;
  },

  remove: function () {
    this.el.remove();
    this.release();
  }
});
```

在Spine中,控制器被认为是一个应用程序的粘合剂,对DOM事件进行添加和响应,渲染模板,还有确保视图和模型保持同步(这在我们所知的控制器的上下文中起作用)。

我们在上面的example.js示例中所做的,是使用render()和remove()方法在更新和销毁事件中设置侦听器。当一个照片条目获得更新的时候，我们对视图进行重新渲染，以此反映对元数据的修改。类似的，如果照片从照片集中被删除了，我们也会把它从视图中移除。在render()函数中，我们使用Underscore微模板（通过_.template())来用ID #photoTemplate对一个Javascript模板进行渲染。这样会简单的返回一个编辑了的HTML字符串用来填充photoEL的内容。

这为我们提供了一个非常轻量级的，简单的管理模型和视图之间的变更的方法。

Backbone.js

后面的章节我们将会对Backbone和传统MVC之间的区别进行一下重新审视,但现在还是让我们专注于控制器吧。

在Backbone中,控制器的责任一分为二,由Backbone.View和Backbone.Router共享.前段时间Backbone确曾有其属于自己的Backbone.Controller,但是对这一组件的命名对于它所被使用的上下文环境中并没有什么意义,后来它就被重新命名为Router了。

Router比控制器要负担处理着更多一点点的责任,因为它使得为模型绑定事件,以及让我们的视图对DOM事件和渲染产生响应,成为可能.如Tim Branyen(另外一名基于Bocoup的Backbone贡献者)在以前所指出的，为此完全摆脱不使用Backbone.Router是有可能的，因此一种考虑让它使用Router范式的做法可能像下面这样：

```
var PhotoRouter = Backbone.Router.extend({
  routes: { "photos/:id": "route" },

  route: function( id ) {
    var item = photoCollection.get( id );
    var view = new PhotoView( { model: item } );

    $('.content').html( view.render().el );
  }
});
```

总之,本节的重点是控制器管理着应用程序中模型和视图之间的逻辑和协作。

MVC给了我们什么?

MVC中关注分离的思想有利于对应用程序中功能进行更加简单的模块化,并且使得:

- 整体的维护更加便利.当需要对应用程序进行更新时,到底这些改变是否是以数据为中心的,意味着对模型的修改还-有可能是控制器,或者仅仅是视觉的,意味着对视图的修改,这一区分是非常清楚的。
- 对模型和视图的解耦意味着为业务逻辑编写单元测试将会是更加直截了当的。
- 对底层模型和控制器的代码解耦(即我们可能会取代使用的)在整个应用程序中被淘汰了。
- 依赖于应用程序的体积和角色的分离,这种模块化允许负责核心逻辑的开发者和工作于用户界面的开发者同时进行工作。

JavaScript中的Smalltalk-80 MVC

尽管当今主流的JavaScript框架都尝试引入MVC的模式，来更好地面对web应用的开发。由Peter Michaux编写的Maria.js (https://github.com/petermichaux/maria) ，是一个尝试纯正的Smalltalk-80的框架。其中，Model只是Model，View也只完成View应该做的，controller则只负责控制。然后，一些开发人员认为，MV*架构更值得关注，如果你对纯正的MVC架构的JavaScript实现感兴趣，这将是很好的参考。

更加深入的钻研

在这本书的这一点上,我们应该对MVC模式提供了些什么有了一个基础的了解,然而仍然有一些值得去关注的非常美妙的信息。

GoF并不将MVC引述为一种设计模式,而是把它看做是构建一个用户界面的类的集合.按照他们的观点,它实际上是三种经典设计模式的变异组合:观察者模式,策略模式和组件模式.依赖于框架中的MVC如何实现,它也可能会使用工厂和模板模式.GoF Book提到这些模式在使用MVC工作时是非常有用的附加功能。

如我们所讨论的,模型代表应用程序的数据,而视图则是用户在屏幕上看到的被展现出来的东西.如此,MVC它的一些核心的通讯就要依赖于观察者模式(令人惊奇的是,一些相关的内容在许多关于MVC模式的书籍并没有被涵盖到).当模型被改变时,它会通知观察者(视图)一些东西已经被更新了——这也许是MVC中最重要的关系。观察者的这一特性也是实现将多个视图连结到同一个模型的基础。

对于那些对MVC解耦特性想了解更多的开发者（这再一次依赖于特定的实现），这一模式的目标之一就是帮助去实现一个主体（数据对象）和它的观察者之间的一对多关系的定义。当一个主体发生改变的时候，它的观察者也会被更新。视图和控制器有一种稍微不同的关系.控制器协助视图对不同的用户输入做出响应,这也是一个策略模式的例子。

总结

回顾完经典的MVC模式以后,我们现在应该理解了它是如何允许我们对一个应用程序中的各个关注点进行清晰地的区分.我们现在也应该感恩于Javascript MVC框架在它们对MVC模式的诠释中是如何的不同,而其对变异也是相当开放的,仍然分享着其原生模式已经提供的其中一些基础概念。

当审视一个新的Javas MVC/MV*框架时,请记住——回过头去考察考察它如何选择相近的架构(特别的,它支持实现了模型,视图,控制器或者其它的一些可选特性)可能会有些用处,因为这样能够更好的帮助我们深入了解这一框架预计需要被如何拿来使用。

MVP

模型-视图-展示器(MVP)是MVC设计模式的一个衍生模式,它专注于提升展现逻辑.它来自于上个世纪九十年代早期的一个叫做Taligent的公司,当时他们正工作于一个基于C++ CommonPoint环境的模型.而MVC和MVP的目标都直指对整个多组件关注点的分离,它们之间有一些基础上的不同。

为了要做出总结的目的,我们将专注于最适合于基于Web架构的MVP版本。

模型,视图&展示器

MVP中的P代表展示器.它是一个包含视图的用户界面逻辑的组件.不像MVC,来自视图的调用被委派给了展示器,它是从视图中解耦出来的,并且转而通过一个接口来同它进行对话.这允许所有类型的有用的东西,比如在单元测试中模拟视图的调用。

对MVP最通常的实现是使用一个被动视图(Passive View 一种对所有动机和目的保持静默的视图),包含很少甚至与没有任何逻辑.如果MVC和MVP是不同的,那是因为其C和P干了不同的事情.在MVP中,P观察着模型并且当模型发生改变的时候对视图进行更新.P切实的将模型绑定到了视图,这一责任在MVC中被控制器提前持有了。

通过视图发送请求，展示者执行所有和用户请求相关的工作，并且把数据返回给视图。从这个方面来讲，它们获取数据，操作数据，然后决定数据如何在视图上面展示。在一些实现当中，展示者同时和一个服务层交互，用于持久化数据（模型）。模型可以触发事件，但是是由展示者扮演这个角色，用于订阅这些事件，从而来更新视图。在这个被动体系架构下，我们没有直接数据绑定的概念。视图暴露setter ，而展示者使用这些setter 来设置数据。

相较于MVC模式的这个改变所带来的好处是，增强了我们应用的可测试性，并且提供了一个更加干净的视图和模型之间的隔离。但是在这个模式里面伴随着缺乏数据绑定支持的缺陷，这就意味着必须对这个任务做另外的处理。

尽管被动视图实现起来普遍都是为视图和实现一个接口，但在它之上还是有差异的，包括可以更多的把视图从展示器解耦的事件的使用。由于在Javascript中我们并没有接口的构造，我们这里更多的是使用一种约定而不是一个明确的接口。技术上看它仍然是一个接口，而从那个角度对于我们而言把它作为一个接口引用可能更加说得过去一些。

也有一种叫做监督控制器的MVP的变种，它更加接近于MVC和MVVM模式，因为它提供了来自于直接来源于视图的模型的数据绑定。键值观察（KVO）插件（比如Derick Bailey的Backbone.ModelBingding插件）趋向于吧Backbone带出被动视图的范畴，而更多的带入监督控制器和MVVM变异中。

MVP还是MVC?

MVP一般最常使用在企业级应用程序中，这样的程序中有必要对展现逻辑尽可能的重用。带有非常复杂的逻辑和大量用户交互的应用程序中，我们也许会发现MVC相对来说并不怎么满足需求，因为要解决这个问题可能意味着对多重控制器的重度依赖。在MVP中，所有这些复杂的逻辑能够被封装到一个展示器中，它可以显著的简化维护工作量。

由于MVP的视图是通过一个接口来被定义的，而这个接口在技术上唯一的要点只是系统和视图（展示器除外）之间接触，这一模式也允许开发者不需要等待设计师为应用程序制作出布局和图形，就可以开始编写展现逻辑。

根据其实现，MVP也许MVC更加容易进行自动的单元测试。为此常常被提及的理由是展示器可以被当做用户接口的完全模拟来使用，而因此它能够独立于其它组件接受单元测试。在我的经验中这取决于我们正在实现的MVP所使用的语言（超过一种取代Javascript来实现MVP的可选语言，同Javascript有着相当大的不同，比如说ASP.net）。

在一天的终点，我们对MVC可能会有的底层关注，可能将是保持对MVP的认可，因为它们之间的不同主要是在语义上的。一旦我们对清晰分离的关注被纳入到模型、视图和控制器（或者展示器）中，我们也许会获得大部分同样的好处，而不用去管我们所作出的选择的差异。

MVC, MVP 和 Backbone.js

很少有，但是如果有任何架构性质的Javascript框架声称用其经典形式实现了MVC或者MVP模式的话，那是因为许多开发者并不认为MVC和MVP是相互冲突的（看到诸如ASP.net或者GWT这样的web框架，我们实际上更加可能会认为MVP被严格的实现了）。这是因为让我们的应用程序有一个附加的展示器/视图逻辑，同时也仍然当其是一种MVC的意味，是有可能的。

Backbone贡献者Irene Ros（位于波士顿的Bocoup）赞同这种想法，当她将视图分离到属于它们自己的单独组件中时，她需要某些东西来实际为她组装它们。这可以是一个控制器路由（比如Backbone.Router，在本书的后面会提到）或者一个对被获取数据做出响应的回调。

这就是说，一些开发者确实感觉Backbone.js更加适合于MVP的描述，相比于MVC。他们的观点是：

- 相比于控制器，MVP中的展示器更好的描述了Backbone.View（视图模板和绑定在视图模板之上的数据之间的中间层）。
- 模型适合Backbone.Model（相较于MVC中的模型并没有很大的不同）。
- 视图最能代表模板（比如 Handlebars/Mustache标记模板）

对此的回应会是视图也可以是一个View（如MVC），因为Backbone对于让它用于多用途有足够的弹性。MVC中的V和MVP中的P都能够通过Backbone.View来完成，因为它们能够达成两个目标：都用来渲染原子组件，还有将那个组件组装起来让其它视图来渲染。

我们也已经看到Backbone中控制器的责任Backbone.View和Backbone.Router都有分享，而在下面的示例中我们能够实际看到那方面实际上都是千真万确的。

在this.model.bind("change",..)一行中，我们的BackbonePhotoView使用了观察者模式来对视图的改变进行“订阅”。它也处理render()方法中的模板，但是并不像一些其它的实现，用户交互也在视图中处理（见events参数）。

```
var PhotoView = Backbone.View.extend({

  //... is a list tag.
  tagName:  "li",

  // Pass the contents of the photo template through a templating
  // function, cache it for a single photo
  template: _.template( $("#photo-template").html() ),

  // The DOM events specific to an item.
  events: {
    "click img" : "toggleViewed"
  },

  // The PhotoView listens for changes to
  // its model, re-rendering. Since tHere's
  // a one-to-one correspondence between a
  // **Photo** and a **PhotoView** in this
  // app, we set a direct reference on the model for convenience.

  initialize: function() {
    this.model.on( "change", this.render, this );
    this.model.on( "destroy", this.remove, this );
  },

  // Re-render the photo entry
  render: function() {
    $( this.el ).html( this.template(this.model.toJSON() ));
    return this;
  },

  // Toggle the `"viewed"` state of the model.
  toggleViewed: function() {
```

```
    this.model.viewed();
  }

});
```

另一种（完全不同的）看法是Backbone更加向我们前面考察过的Smalltalk-80MVC靠拢。

定期为Backbone写博客的Derick Bailey之前已经提到过，最终最好不要去强迫Backbone让其适应任何特定的设计模式。设计模式应该考虑指导可能如何被构建的灵活性，而在这一方面，Backbone既不适应MVC，也不适应MVP。相反，它从多个架构模式中借用了一些最好的经验而创造出了一个灵活的框架，并且工作得很好。

而理解这些这些概念源自哪里和为什么源自那里是值得去做的，因此我希望我对于MVC和MVP的阐述对此已经有所帮助。就叫它Backbone方法吧，MV*或带有的其它应用程序架构的意味。大多数结构性Javascript框架自主决定自身采用的经典模式，不管是有意还是无意为之的，最重要的是它们帮助了我们有组织，干净的来开发方便维护的应用程序。

MVVM

MVVM(Model View ViewModel)是一种基于MVC和MVP的架构模式，它试图将用户界面（UI）从业务逻辑和行为中更加清晰地分离出来。为了这个目的，很多例子使用声明变量绑定来把View层的工作从其他层分离出来。

这促进了UI和开发工作在同一代码库中的同步进行。UI开发者用他们的文档标记（HTML）绑定到ViewModel，在这个地方Model和ViewModel由负责逻辑的开发人员维护。

历史

MVVM（如其大名）最初是由微软定义，用于Windows Presentation Foundation(WPF)和Silverlight，在John Grossman2005年的一篇关于Avalon（WPF的代号）的博文中被官方推出。它也作为方便使用MVC的一种可选方案，为Adobe Flex社区积累了一些用户量。

先于微软采用的MVVM名称，在社区中已经有了一场由MVC像MVPM迁移的运动：模型-视图-展现模型。Marton Fowler在2004年为那些对此感兴趣的人写了一篇关于展现模型的文章。展现模型的理念的内容要远远长于这篇文章，然而这篇文章被认为是这一理念的重大突破，并且极大的捧红了它。

在微软推出作为MVPM的可选方案的MVVM后，就出现了许多沸沸扬扬的“alt.net”圈子。其中许多声称这个公司在GUI世界的霸主地位给与了它们将社区统一为整体的机会，出于市场营销的目的，按照它们所高兴的方式对已有的概念重新命名。一个进步的群体也承认MVVM和MVPM其实实在是同样的概念，只是展现出来的是不同的包而已。

在近几年，MVVM已经在Javascript中得到了实现，其构造框架的形式诸如KnockoutJS，Kendo MVVM和Knockback.js，获得了整个社区的积极响应。

现在就让我们来看看组成了MVVM的这三个组件。

模型

和其它MV*家族成员一样，MVVM中的模型代表我们的应用用到的领域相关的数据或者信息。一个领域相关的数据的典型例子是用户账号（例如名字，头像，电子邮件）或者音乐唱片（例如唱片名，年代，专辑）。

模型持有信息，但是通常没有操作行为。它们不会格式化信息，也不会影响数据在浏览器中的表现，因为这些不是模型的责任。相反，数据格式化是由视图层处理的，尽管这种行为被认为是业务逻辑，这个逻辑应该被另外一个层封装，这个层和模型交互，这个曾就是视图模型。

这个规则唯一的例外是验证，由模型进行数据验证是被认为可以接受的，这些数据用于定义或者更新现存的模型（例如输入的电子邮件地址是否满足特定的正则表达式要求？）。

在KnockoutJS中，模型遵从上面的定义，但是通常对服务端服务的Ajax调用被做成即可以读取也可以写入模型数据。

如果我们正在构建一个简单的Todo应用，使用KnockoutJS模型来表示一个Todo条目，看起来像下面这个样子：

```
var Todo = function ( content, done ) {
    this.content = ko.observable(content);
    this.done = ko.observable(done);
    this.editing = ko.observable(false);
};
```

> 注意:在上面小段代码里面，你可能发现了，我们在KnockoutJS的名字空间里面调用observable()方法。在KnockoutJS中，观察者是一类特殊的JavaScript对象，可以将变化通知给订阅者，并且自动检测依赖关系。这个特性使我们在模型值修改之后，可以同步模型和视图模型。

视图

使用MVC,视图是应用程序中用户真正与之打交道的唯一一个部分.它们是展现一个视图模型状态的一个可交互UI.此种意义而言,视图是主动的而不是被动的,而这也是真正的MVC和MVP的观点.在MVC,MVP和MVVM中视图也可以是被动的,而这又是什么意思呢?

被动视图仅仅只输出要展示的东西,而不去接受任何用户的输入。

这样一个视图在我们的应用程序中可能也没有真正的模型的概念,而可以被一个代理控制.MVVM的主动视图包含数据绑定,事件和需要能够理解视图模型的行为.尽管这些行为能够被映射到属性,视图仍然处理这来自视图模型的事件。

记住视图在这里并不负责处理状态时很重要的——它使得其与视图模型得以同步。

KnockoutJS视图是简单的一个带有声明链接到视图模型的HTML文档。KnockoutJS视图展示来自视图模型的信息，并且传递命令给他（比如，用户在一个元素上面点击），并且针对视图模型的变化更新状态。而使用来自视图模型的数据来生成标记的模板也能够被用在这个目的上。

未来给出一个简单的初始示例，我们可以看看Javascritpt的MVVM框架KnockoutJS，看它如何允许一个视图模型的定义，还有它在标记中的相关绑定。

视图模型：

```
var aViewModel = {
   contactName: ko.observable("John")
};
ko.applyBindings(aViewModel);
```

视图：

```
<p><input id="source" data-bind="value: contactName, valueUpdate: 'keyup'" /></p>
<div data-bind="visible: contactName().length > 10">
   You have a really long name!
</div>
<p>Contact name: <strong data-bind="text: contactName"></strong></p>
```

我们的text-box输入(源)从contactName获取它的初始值,无论何时contactName发生了改变都会自动更新这个值.由于数据绑定是双向的,像text-box中输入也将据此更新contactName,以此保持值总是同步的。

尽管这个实现特定于KnockoutJS，但是包含着"You have a really long name!"文本的

<

div>标签包含有简单的验证(同样是以数据绑定的形式呈现)。如果输入超过10个字符，这个标签就会显示，否则保持隐藏。

让我们看看一个更高级的例子，我们可以看看我们的Todo应用。一个用于这个应用的裁剪后的KnockoutJS的视图，包含有所有必要的数据绑定，这个视图看起来是下面这个样子。

```
<div id="todoapp">
   <header>
      <h1>Todos</h1>
      <input id="new-todo" type="text" data-bind="value: current, valueUpdate: 'afterkeydown', enterKey: add"
            placeholder="What needs to be done?"/>
   </header>
   <section id="main" data-bind="block: todos().length">

      <input id="toggle-all" type="checkbox" data-bind="checked: allCompleted">
      <label for="toggle-all">Mark all as complete</label>

      <ul id="todo-list" data-bind="foreach: todos">

         <!-- item -->
         <li data-bind="css: { done: done, editing: editing }">
            <div class="view" data-bind="event: { dblclick: $root.editItem }">
               <input class="toggle" type="checkbox" data-bind="checked: done">
               <label data-bind="text: content"></label>
               <a class="destroy" href="#" data-bind="click: $root.remove"></a>
```

```
            </div>
            <input class="edit' type="text"
                data-bind="value: content, valueUpdate: 'afterkeydown', enterKey: $root.stopEditing, selectAndFocus: editin
        </li>

    </ul>

  </section>
</div>
```

请注意，这个标记的基本布局是相对直观的，包含有一个输入文本框(新的todo)用于增加新条目，用于标记条目完成的开关，以及一个拥有模板的列表（todo列表），这个模板以anli的形式展现Todo条目。

上面标记中绑定的数据可以分成下面几块：

- 输入的文本框new-todo 有一个当前属性的数据绑定，当前要增加的条目的值存储在这里。我们的视图模型(后面就会看到)观察当前属性，并且绑定在添加事件上。当回车键按下的时候，添加事件就被出发了，我们的视图模型就可以处理当前的值按照需要并且将其加入到Todo列表中。
- 输入勾选框可以通过点击标示所有当前条目为完成状态。如果勾选了，触发完成事件，这个事件可以被模型视图观察到。
- 有一类条目是进行中状态。当一个任务被标记为进行中，CSS类也会根据这个状态进行标识。如果双击条目，$root.editItem 回调就会被执行。
- toggle类的勾选框表明当前的进行状态。
- 一个文本标签包含有Todo条目的内容
- 当点击一个移除按钮时可以调用$root.remove 回调函数。
- 编辑模式下的一个输入文本框可以保存Todo条目的内容。回车键事件将会设定编辑属性为真或者假。

视图模型

视图模型被认为是一个专门进行数据转换的控制器。它可以把对象信息转换到视图信息，将命令从视图携带到对象。

例如，我们想象我们有一个对象的日期属性是unix格式的（e.g 1333832407），而不是用户视图的所需要的日期格式（e.g 04/07/2012 @ 5:00pm），这时就有必要把unix的日期格式转换为视图需要的格式。我们的对象只简单保存原始的unix数据格式日期，视图模型作为一个中间人角色会格式化原始的unix数据格式转换为视图需要的日期格式。

在这个场景下，视图模型可以被看做一个对象，它处理很多视图显示逻辑。视图模型也对外提供更新视图状态的方法，并通过视图方法和触发事件更新对象。

简单来说，视图模型位于我们UI层后面层。它通过视图发布对象的公共数据，同时它作为视图源提供数据和方法。

KnockoutJS描述视图模型作为数据的表现和操作可以在UI上访问和执行。视图模型并不是一个UI对象，也不是数据持久化对象，而是一个能够为用户提供储存状态及操作的层次对象。Knockout的视图模型实现了JavaScript对象与HTML语言无关性。通过这个实现使开发保持了简单，意味着我们可以在视图层更加简单的管理更多的组合方法。

对于我们的ToDo应用程序的一部分KnockoutJS视图模型可以是像下面这样:

```
// our main ViewModel
    var ViewModel = function ( todos ) {
        var self = this;

    // map array of passed in todos to an observableArray of Todo objects
    self.todos = ko.observableArray(
    ko.utils.arrayMap( todos, function ( todo ) {
        return new Todo( todo.content, todo.done );
    }));

    // store the new todo value being entered
    self.current = ko.observable();

    // add a new todo, when enter key is pressed
    self.add = function ( data, event ) {
        var newTodo, current = self.current().trim();
        if ( current ) {
            newTodo = new Todo( current );
            self.todos.push( newTodo );
            self.current("");
        }
    };

    // remove a single todo
    self.remove = function ( todo ) {
        self.todos.remove( todo );
    };

    // remove all completed todos
    self.removeCompleted = function () {
        self.todos.remove(function (todo) {
```

```
            return todo.done();
        });
    };

    // writeable computed observable to handle marking all complete/incomplete
    self.allCompleted = ko.computed({

        // always return true/false based on the done flag of all todos
        read:function () {
            return !self.remainingCount();
        },

        // set all todos to the written value (true/false)
        write:function ( newValue ) {
            ko.utils.arrayForEach( self.todos(), function ( todo ) {
                //set even if value is the same, as subscribers are not notified in that case
                todo.done( newValue );
            });
        }
    });

    // edit an item
    self.editItem = function( item ) {
        item.editing( true );
    };
..
```

上面我们基本上提供了必需的加入、编辑或者移除记录的方法，还有标记所有现存的记录已经被完成的逻辑。注意：唯一真正需要关注的同前面我们的视图模型的示例的不同之处就是观察数组.在KnockoutJS中,如果我们希望监测到并且去回应一个单独的对象发生的改变,我们可以使用观察.然而如果我们希望检测并且去回应一个集合的事物所发生的改变,我们可以换用一个观察数组.如何使用观察数组的一个简单示例就像下面这样:

```
// Define an initially an empty array
var myObservableArray = ko.observableArray();

// Add a value to the array and notify our observers
myObservableArray.push( 'A new todo item' );
```

> 注意:感兴趣的话,我们在前面所提到的完整的KnockoutJS Todo应用程序可以从 TodoMVC 获取到。

扼要重述: 视图和视图模型

视图和视图模型使用数据绑定和事件进行通信。正如我们之前的视图模型例子所见，视图模型不仅仅发布对象属性，它还提供其他的方法和特性，诸如验证。

我们的视图处理自己的用户接口事件，并会把相关事件映射到视图模型。对象和它属性与视图模型是同步的，且通过双向数据绑定进行更新。

触发器（数据触发器）允许我们进一步在视图状态变化后改变我们的对象属性。

小结:视图模型和模型

虽然可能会出现在MVVM中视图模型完全对模型负责的情况,这些关系确实有一些值得关注的微妙之处.处于数据绑定的目的,视图模型可以暴露出来一个模型或者模型属性,而且也能够包含获取和操作视图中暴露出来的属性。

优点和缺点

现在,我们完全对MVVM是什么,以及它是如何工作的,有了一个更好的了解.现在就让我们来看看使用这种模式的优点和缺点吧:

优点:

- MVVM更加便于UI和驱动UI的构造块,这两部分的并行开发
- 抽象视图使得背后所需要的业务逻辑(或者粘合剂)的代码数量得以减少
- 视图模型比事件驱动代码更加容易进行单元测试
- 视图模型(比视图更加像是模型)能够在不用担心UI自动化和交互的前提下被测试

缺点:

- 对于更简单的UI而言,MVVM可能矫枉过正了
- 虽然数据绑定可以是声明性质的并且工作得很好,但在我们简单设置断点的地方,它们比当务之急的代码更加难于调试
- 在非凡的应用程序中的数据绑定能够创造许多的账簿.我们也并不希望以绑定比被绑定目标对象更加重量级,这样的境地告终
- 在大型的应用程序中,将视图模型的设计提升到获取足够所需数量的泛化,会变得更加的困难

MVVM 的低耦合数据绑定

常见到有着MVC或者MVP开发经验的JavaScript程序员评论MVVM的时候在抱怨它会分散他们的关注点。也就是说，他们习惯在一个视图中有相当数量的数据被耦合在了HTML标签中。

我必须承认当我第一次体验实现了MVVM的JavaScript框架后(例如 KnockoutJS, Knockback)，我很惊讶很多程序员都想要回到一个难以维护的混淆了逻辑(JavaScript代码)和HTML标签做法的过去。然而现实是使用MVVM会有很多好处（我们之前说过），包括设计师能更容易的通过他们的标记去绑定相关逻辑。

在我们中间的传统程序员，你会很开心知道现在我们能够通过数据绑定这个特性大量减少程序代码的耦合程度，且KnockoutJS从1.3这个版本就开始提供自定义绑定功能。

KnockoutJS 默认有一个数据绑定提供者，这个提供者搜索所有的附属有数据绑定属性的元素，如下面的例子：

```
<input id="new-todo" type="text" data-bind="value: current, valueUpdate: 'afterkeydown', enterKey: add" placeholder="W
```

当这个提供者定位一个到包含有该属性的元素时，这个工具将会分析该元素，使用当前的数据上下文来将其转化成一个绑定对象。这种方式是 KnockoutJS百分百可以工作的方式，通过这种方式，我们可以采用声明式的方法对元素增加绑定，KnockoutJS之后会在该层上将数据绑定到元素上。

当我们开始构建复杂的视图的时候，我们最终就可能得到大量的元素和属性在标记中绑定数据，这种方式将会变得很难管理。通过自定义的绑定提供者，这就不算个问题。

一个绑定提供者主要关心两件事：

- 给定一个DOM节点，这个节点是否包含任何数据绑定?
- 如果节点回答是YES，那么这个绑定的对象在当前数据上下文中，看起来是什么样的?

绑定提供者实现了两个功能：

- nodeHasBindings:这个有一个DOM的节点参数，这个参数不一定是一个元素
- getBindings:返回一个对象代表当前数据上下文下的要使用的绑定

一个框架绑定提供者看起来如下：

```
var ourBindingProvider = {
  nodeHasBindings: function( node ) {
     // returns true/false
  },

  getBindings: function( node, bindingContext ) {
     // returns a binding object
  }
};
```

在我们充实这个提供者之前，让我们先简要的讨论一下数据绑定属性中的逻辑。

当使用Knockout的MVVM，我们会对将应用逻辑过度绑定到视图上的这种方法不满。我们可以实现像CSS类一样的东西，将绑定根据名字赋值给元素。Ryan Niemeyer(knockmeout.net上的)之前提出使用数据类用于这个目的，来避免将展示类和数据类混淆，让我们改造我们的nodeHasBindings 函数，来支持这个概念：

```
// does an element have any bindings?
function nodeHasBindings( node ) {
   return node.getAttribute ? node.getAttribute("data-class") : false;
};
```

接下来，我们需要一个敏感的getBindings()函数。既然我们坚持使用CSS类的概念，为什么不考虑一下支持空格分割类呢，这样可以使我们在不同元素之间共享绑定标准。

让我们首先看一下我们的绑定长什么样子。我们建立一个对象用于持有它们，在这些绑定处，我们的属性名需要和我们数据类中使用的关键字相匹配。

> 注意：对于将使用传统数据绑定方式的KnockoutJS应用转化成一个使用自定义绑定提供者的不引人瞩目的绑定方式。我们简单的拉取我们所有的数据绑定属性，使用数据类属性来替换它们，并且像之前做的一样，将我们的绑定放到绑定对象中去。

```
var viewModel = new ViewModel( todos || [] ),
   bindings = {

      newTodo:  {
         value: viewModel.current,
         valueUpdate: "afterkeydown",
         enterKey: viewModel.add
      },

      taskTooltip : {
         visible: viewModel.showTooltip
      },
      checkAllContainer : {
         visible: viewModel.todos().length
      },
      checkAll: {
         checked: viewModel.allCompleted
      },

      todos: {
         foreach: viewModel.todos
      },
      todoListItem: function() {
         return {
            css: {
```

```
            editing: this.editing
        }
    };
},
todoListItemWrapper: function() {
    return {
        css: {
            done: this.done
        }
    };
},
todoCheckBox: function() {
    return {
        checked: this.done
    };
},
todoContent: function() {
    return {
        text: this.content,
        event: {
            dblclick: this.edit
        }
    };
},
todoDestroy: function() {
    return {
        click: viewModel.remove
    };
},

todoEdit: function() {
    return {
        value: this.content,
        valueUpdate: "afterkeydown",
        enterKey: this.stopEditing,
        event: {
            blur: this.stopEditing
        }
    };
},

todoCount: {
    visible: viewModel.remainingCount
},
remainingCount: {
```

```
            text: viewModel.remainingCount
        },
        remainingCountWord: function() {
            return {
                text: viewModel.getLabel(viewModel.remainingCount)
            };
        },
        todoClear: {
            visible: viewModel.completedCount
        },
        todoClearAll: {
            click: viewModel.removeCompleted
        },
        completedCount: {
            text: viewModel.completedCount
        },
        completedCountWord: function() {
            return {
                text: viewModel.getLabel(viewModel.completedCount)
            };
        },
        todoInstructions: {
            visible: viewModel.todos().length
        }
    };

    ....
```

上面代码中，我们丢掉了两行，我们仍然需要getBindings函数，这个函数遍历数据类属性中每一个关键字，并从中构建最终对象。如果我们检测到绑定对象是个函数，我们使用当前的数据调用它。我们的完成版自定义绑定提供中，如下：

```
// We can now create a bindingProvider that uses
    // something different than data-bind attributes
    ko.customBindingProvider = function( bindingObject ) {
        this.bindingObject = bindingObject;

        // determine if an element has any bindings
        this.nodeHasBindings = function( node ) {
            return node.getAttribute ? node.getAttribute( "data-class" ) : false;
        };
     };

    // return the bindings given a node and the bindingContext
    this.getBindings = function( node, bindingContext ) {
```

```
        var result = {},
            classes = node.getAttribute( "data-class" );

        if ( classes ) {
            classes = classes.split( "" );

            //evaluate each class, build a single object to return
            for ( var i = 0, j = classes.length; i < j; i++ ) {

                var bindingAccessor = this.bindingObject[classes[i]];
                if ( bindingAccessor ) {
                    var binding = typeof bindingAccessor === "function" ? bindingAccessor.call(bindingContext.$data) : bindingAcc
                    ko.utils.extend(result, binding);
                }

            }
        }

        return result;
    };
};
```

我们绑定对象最后的几行，定义如下：

```
// set ko's current bindingProvider equal to our new binding provider
ko.bindingProvider.instance = new ko.customBindingProvider( bindings );

// bind a new instance of our ViewModel to the page
ko.applyBindings( viewModel );

})();
```

我们在这里所做的是为我们的绑定处理器有效的定义构造器，绑定处理器接受一个我们用来查找绑定的对象（绑定）。然后我们可以使用数据类为我们应用程序视图的重写标记，像下面这样做：

```
<div id="create-todo">
            <input id="new-todo" data-class="newTodo" placeholder="What needs to be done?" />
            <span class="ui-tooltip-top" data-class="taskTooltip" style="display: none;">Press Enter to save this task</spa
        </div>
        <div id="todos">
            <div data-class="checkAllContainer" >
                <input id="check-all" class="check" type="checkbox" data-class="checkAll" />
                <label for="check-all">Mark all as complete</label>
            </div>
            <ul id="todo-list" data-class="todos" >
```

```
        <li data-class="todoListItem" >
          <div class="todo" data-class="todoListItemWrapper" >
            <div class="display">
              <input class="check" type="checkbox" data-class="todoCheckBox" />
              <div class="todo-content" data-class="todoContent" style="cursor: pointer;"></div>
              <span class="todo-destroy" data-class="todoDestroy"></span>
            </div>
            <div class="edit'>
              <input class="todo-input" data-class="todoEdit'/>
            </div>
          </div>
        </li>
      </ul>
    </div>
```

Nei Kerkin 已经使用上面的方式组合成了一个完整的TodoMVC示例，它可以 从 这里获取到。 虽然上面的解释看起来像是有许多的工作要做，现在我们就有一个一般的getBindingmethod方法要写。比起为了编写我们的KnockoutJS应用程序而严格使用数据绑定，简单的重用和使用数据类更加的琐碎。最终的结果是希望得到一个干净的标记，其中我们的数据绑定会从视图切换到一个绑定对象。

MVC VS MVP VS MVVM

MVP和MVVM都是MVC的衍生物。它和它的衍生物之间关键的不同之处在于每一层对于其它层的依赖，以及它们相互之间是如何紧密结合在一起的。

在MVC中，视图位于我们架构的顶部，其背后是控制器。模型在控制器后面，而因此我们的视图了解得到我们的控制器，而控制器了解得到模型。这里，我们的视图有对模型的直接访问。然而将整个模型完全暴露给视图可能会有安全和性能损失，这取决于我们应用程序的复杂性。MVVM则尝试去避免这些问题。

在MVP中，控制器的角色被代理器所取代，代理器和视图处于同样的地位，视图和模型的事件都被它侦听着并且接受它的调解。不同于MVVM，没有一个将视图绑定到视图模型的机制，因此我们转而依赖于每一个视图都实现一个允许代理器同视图去交互的接口。

MVVM进一步允许我们创建一个模型的特定视图子集，包含了状态和逻辑信息，避免了将模型完全暴露给视图的必要。不同于MVP的代理器，视图模型并不需要去引用一个视图。视图可以绑定到视图模型的属性上面，视图模型则去将包含在模型中的数据暴露给视图。像我们所提到过的，对视图的抽象意味着其背后的代码需要较少的逻辑。

对此的副作用之一就是视图模型和视图层之间新增的的用于翻译解释的一层会有性能损失。这种解释层的复杂度根据情况也会有所差异——它可能像复制数据一样简单，也可能会像我们希望用视图理解的一种形式去操作它们，那样复杂。由于整个模型是现成可用的，从而这种操作可以被避免掉，所以MVC没有这种问题。

Backbone.js Vs KnockoutJS

了解MVC，MVP和MVVM之间的细微差别是很重要的，然而基于我们已经了解到的东西，开发者最终会问到是否它们应该考虑使用KnockoutJS而不是Backbone这个问题。下面的一些相关事项对此可能有些帮助：

- 两个库都设计用于不同的目标，它常常不仅仅简单的知识选择MVC或者MVVM的问题。
- 如果数据绑定和双向通信是你主要关注的问题，KnockoutJS绝对是应该选择的方式。实践中任何存储在DOM节点中的值或者属性都能够使用此方法映射到Javascript对象上面。
- Backbone在同RESTful服务的易于整合方面有其过人之处，而KnockoutJS模型就是简单的Javascript对象，而更新模型所需要的代码也必须要由开发者自己来写。
- KnockoutJS专注于自动的UI绑定，如果尝试使用Backbone来做的话则会要求更加细节的自定义代码。由于Backbone自身就意在独立于UI而存在，所以这并不是它的问题。然而Knockback也确实能协助并解决此问题。 使用KnockoutJS，我们能够将我们自己拥有的函数绑定到视图模型观察者上面，任何时候观察者一旦发生了变化，它都会执行。这允许我们能够拥有同在Backbone中发现的一样级别的灵活性。
- Backbone内置有一个坚实的路由解决方案，而KnockoutJS则没有提供路由供我们选择。然而人们如果需要的话，可以很容易的加入这个行为，使用Ben Alman的BBQ插件或者一个像Miller Medeiros优秀的Crossroads就行了。

总结下来，我个人发觉KnockoutJS更适合于小型的应用，而Backbone的特性在任何东西都是无序的场景下面才会是亮点。那就是说，许多开发者两个框架都已经使用过来编写不同复杂度的应用程序，而我建议在一个小范围内两种都尝试一下，在你决定哪一种能更好的为你工作之前。

4

模块化 JavaScript 设计模式

AMD

在浏览器中编写模块化Javascript的格式

AMD (异步模块定义Asynchronous Module Definition)格式的最终目的是提供一个当前开发者能使用的模块化Javascript方案。它出自于Dojo用XHR+eval的实践经验，这种格式的支持者想在以后的项目中避免忍受过去的这些弱点。

AMD模块格式本身是模块定义的一个建议，通过它模块本身和模块之间的引用可以被异步的加载。它有几个明显的优点，包括异步的调用和本身的高扩展性，它实现了解耦，模块在代码中也可通过识别号进行查找。当前许多开发者都喜欢使用它，并且认为它朝ES Harmony提出模块化系统 迈出了坚实的一步。

最开始AMD在CommonJs的列表中是作为模块化格式的一个草案，但是由于它不能达到与模块化完全一致，更进一步的开发被移到了在amdjs组中。

现在，它包含工程Dojo、MooTools、Firebug以及jQuery。尽管有时你会看见CommonJS AMD 格式化术语，但最好的和它相关的是AMD或者是异步模块支持，同样不是所有参与到CommonJS列表的成员都希望与它产生关系。

> 注意：曾有一段时间涉及Transport/C模块的提议规划没有面向已经存在的CommonJS模块，但是对于定义模块来说，它对选择AMD命名空间约定产生了影响。

从模块开始

关于AMD值得特别注意的两个概念就是：一个帮助定义模块的define方法和一个处理依赖加载的require方法。define被用来通过下面的方式定义命名的或者未命名的模块：

```
define(
    module_id /*可选的*/,
    [dependencies] /*可选的*/,
    definition function /*用来实例化模块或者对象的方法*/
);
```

通过代码中的注释我们可以发现，module_id 是可选的，它通常只有在使用非AMD连接工具的时候才是必须的（可能在其它不是特别常见的情况下，它也是有用的）。当不存在module_id参数的时候，我们称这个模块为匿名模块。

当使用匿名模块的时候，模块认定的概念是DRY的，这样使它在避免文件名和代码重复的时候显得很微不足道。因为这样一来代码方便切换，你可以很容易地把它移动到其它地方（或者文件系统的其他位置），而不需要更改代码内容或者它的模块ID。你可以认为模块id跟文件路径的概念是相似的。

注意：开发者们可以将同样的代码放到不同的环境中运行，只要他们使用一个在CommonJS环境下工作的AMD优化器（比如r.js）就可以了。

在回来看define方法签名, dependencies参数代表了我们正在定义的模块需要的dependency数组，第三个参数("definition function" or "factory function") 是用来执行的初始化模块的方法。 一个正常的模块可以像下面那样定义：

Understanding AMD: define()

```
// A module_id (myModule) is used here for demonstration purposes only
define( "myModule",

    ["foo", "bar"],

    // module definition function
    // dependencies (foo and bar) are mapped to function parameters
    function ( foo, bar ) {
        // return a value that defines the module export
        // (i.e the functionality we want to expose for consumption)

        // create your module here
        var myModule = {
            doStuff:function () {
                console.log( "Yay! Stuff" );
            }
        };

    return myModule;
});

// An alternative version could be..
define( "myModule",

    ["math", "graph"],

    function ( math, graph ) {

        // Note that this is a slightly different pattern
        // With AMD, it's possible to define modules in a few
        // different ways due to it's flexibility with
```

```
    // certain aspects of the syntax
    return {
        plot: function( x, y ){
            return graph.drawPie( math.randomGrid( x, y ) );
        }
    };
});
```

另一方面，require被用来从一个顶级文件或者模块里加载代码，而这是我们原本就希望的动态加载依赖的位置。它的一个用法如下：

理解AMD: require()

```
// Consider "foo" and "bar" are two external modules
// In this example, the "exports" from the two modules
// loaded are passed as function arguments to the
// callback (foo and bar) so that they can similarly be accessed

require(["foo", "bar"], function ( foo, bar ) {
    // rest of your code here
    foo.doSomething();
});
```

动态加载依赖

```
define(function ( require ) {
    var isReady = false, foobar;

    // note the inline require within our module definition
    require(["foo", "bar"], function ( foo, bar ) {
        isReady = true;
        foobar = foo() + bar();
    });

    // we can still return a module
    return {
        isReady: isReady,
        foobar: foobar
    };
});
```

理解 AMD: 插件

下面是定义一个兼容AMD插件的例子：

```
// With AMD, it's possible to load in assets of almost any kind
// including text-files and HTML. This enables us to have template
// dependencies which can be used to skin components either on
// page-load or dynamically.

define( ["./templates", "text!./template.md","css!./template.css" ],

  function( templates, template ){
    console.log( templates );
    // do something with our templates here
  }

});
```

> 注意:尽管上面的例子中css!被包含在在加载CSS依赖的过程中，要记住，这种方式有一些问题，比如它不完全可能在CSS完全加载的时候建立模块. 取决于我们如何实现创建过程，这也可能导致CSS被作为优化文件中的依赖被包含进来，所以在这些情况下把CSS作为已加载的依赖应该多加小心。如果你对上面的做法感兴趣，我们也可以从这里查看更多@VIISON的RequireJS CSS 插件：https://github.com/VIISON/RequireCSS

使用RequireJS加载AMD模块

```
require(["app/myModule"],

  function( myModule ){
    // start the main module which in-turn
    // loads other modules
    var module = new myModule();
    module.doStuff();
});
```

这个例子可以简单地看出asrequirejs（“app/myModule”,function(){}）已被加载到顶层使用。这就展示了通过AMD的define（）函数加载到顶层模块的不同，下面通过一个本地请求allrequire（[]）示例两种类型的装载机（curl.js和RequireJS）。

使用curl.js加载AMD模块

```
curl(["app/myModule.js"],

  function( myModule ){
    // start the main module which in-turn
    // loads other modules
    var module = new myModule();
    module.doStuff();

});
```

延迟依赖模块

```
// This could be compatible with jQuery's Deferred implementation,
// futures.js (slightly different syntax) or any one of a number
// of other implementations

define(["lib/Deferred"], function( Deferred ){
    var defer = new Deferred();

    require(["lib/templates/?index.html","lib/data/?stats"],
        function( template, data ){
            defer.resolve( { template: template, data:data } );
        }
    );
    return defer.promise();
});
```

使用Dojo的AMD模块

使用Dojo定义AMD兼容的模块是相当直接的.如上所述,就是在一个数组中定义任何的模块依赖作为第一个参数,并且提供回调函数来执行一次依赖已经被加载进来的模块.例如:

```
define(["dijit/Tooltip"], function( Tooltip ){

    //Our dijit tooltip is now available for local use
    new Tooltip(...);

});
```

请注意模块的匿名特性,现在它可以在一个Dojo匿名装载装置中的被处理,RequireJS或者标准的dojo.require()模块装载器。

了解一些有趣的关于模块引用的陷阱是非常有用的.虽然AMD倡导的引用模块的方式宣称它们在一组带有一些匹配参数的依赖列表里面,这在版本更老的Dojo 1.6构建系统中并不被支持--它真的仅仅对AMD兼容的装载器才起作用.例如:

```
define(["dojo/cookie", "dijit/Tooltip"], function( cookie, Tooltip ){

    var cookieValue = cookie( "cookieName" );
    new Tooltip(...);

});
```

越过嵌套的命名空间定义方式有许多好处,模块不再需要每一次都直接引用完整的命名空间了--所有我们所需要的是依赖中的"dojo/cookie"路径,它一旦赋给一个作为别名的参数,就可以用变量来引用了.这移除了在我们的应用程序中重复打出"dojo."的必要。

最后需要注意到的难点是,如果我们希望继续使用更老的Dojo构建系统,或者希望将老版本的模块迁移到更新的AMD形式,接下来更详细的版本会使得迁移更加容易.注意dojo和dijit也是作为依赖被引用的:

```
define(["dojo", "dijit', "dojo/cookie", "dijit/Tooltip"], function( dojo, dijit ){
    var cookieValue = dojo.cookie( "cookieName" );
    new dijit.Tooltip(...);
});
```

AMD 模块设计模式 (Dojo)

正如在前面的章节中，设计模式在提高我们的结构化构建的共同开发问题非常有效。 John Hann已经给AMD模块设计模式，涵盖单例，装饰，调解和其他一些优秀的设计模式，如果有机会，我强烈建议参考一下他的 幻灯片。 AMD设计模式的选择可以在下面找到。

一段AMD设计模式可以在下面找到。

修饰设计模式

```
// mylib/UpdatableObservable: dojo/store/Observable的一个修饰器
define(["dojo", "dojo/store/Observable"], function ( dojo, Observable ) {
    return function UpdatableObservable ( store ) {

        var observable = dojo.isFunction( store.notify ) ? store :
                new Observable(store);

        observable.updated = function( object ) {
            dojo.when( object, function ( itemOrArray) {
                dojo.forEach( [].concat(itemOrArray), this.notify, this );
            });
        };

        return observable;
    };
});


// 修饰器消费者
// mylib/UpdatableObservable的消费者
```

```
define(["mylib/UpdatableObservable"], function ( makeUpdatable ) {
    var observable,
        updatable,
        someItem;

    // 让observable 储存 updatable
    updatable = makeUpdatable( observable ); // `new` 关键字是可选的!

    // 如果我们想传递修改过的data，我们要调用.update()
    //updatable.updated( updatedItem );
});
```

适配器设计模式

```
// "mylib/Array" 适配`each`方法来模仿 jQuerys:
define(["dojo/_base/lang", "dojo/_base/array"], function ( lang, array ) {
    return lang.delegate( array, {
        each: function ( arr, lambda ) {
            array.forEach( arr, function ( item, i ) {
                lambda.call( item, i, item ); // like jQuery's each
            });
        }
    });
});

// 适配器消费者
// "myapp/my-module":
define(["mylib/Array"], function ( array ) {
    array.each( ["uno", "dos", "tres"], function ( i, esp ) {
        // here, `this` == item
    });
});
```

使用jQuery的AMD模块

不像Dojo，jQuery真的存在于一个文件中，而是基于插件机制的库，我们可以在下面代码中证明AMD模块是如何直线前进的。

```
define(["js/jquery.js","js/jquery.color.js","js/underscore.js"],

    function( $, colorPlugin, _ ){
        // <span></span>这里，我们通过jQuery中，颜色的插件，并强调没有这些将可在全局范围内访问，但我们可以很容易地在下面
        // 伪随机一系列的颜色，在改组后的数组中选择的第一个项目
```

```
<div>




</div>
    var shuffleColor = _.first( _.shuffle( "#666","#333","#111"] ) );

    // 在页面上有class为"item" 的元素随机动画改变背景色
    $( ".item" ).animate( {"backgroundColor": shuffleColor } );

    // 我们的返回可以被其他模块使用
    return {};
  });
```

然而，这个例子中缺失了一些东西，它只是注册的概念。

将jQuery当做一个异步兼容的模块注册

jQuery1.7中落实的一个关键特性是支持将jQuery当做一个异步兼容的模块注册。有很多兼容的脚本加载器(包括 RequireJS 和 curl ）可以使用异步模块形式加载模块,而这意味着在让事物起作用的时候,更少的需要使用取巧的特殊方法。

如果开发者想要使用AMD,并且不想将他们的jQuery的版本泄露到全局空间中,他们就应该在使用了jQuery的顶层模块中调用noConflict方法.另外,由于多个版本的jQuery可能在一个页面上,AMD加载器就必须作出特殊的考虑,以便jQuery只使用那些认识到这些问题的AMD加载器来进行注册,这是使用加载器特殊的define.amd.jQuery来表示的。RequireJS和curl是两个这样做了的加载器。

这个叫做AMD的家伙提供了一种安全的鲁棒的封包，这个封包可以用于绝大多数情况。

```
// Account for the existence of more than one global
// instances of jQuery in the document, cater for testing
// .noConflict()

var jQuery = this.jQuery || "jQuery",
$ = this.$ || "$",
originaljQuery = jQuery,
original$ = $;
```

```
define(["jquery"] , function ( $ ) {
    $( ".items" ).css( "background","green" );
    return function () {};
});
```

为什么AMD是写模块化Javascript代码的好帮手呢?

- 提供了一个清晰的方案，告诉我们如何定义一个可扩展的模块。
- 和我们常用的前面的全局命名空间以及