Aprendendo Padrões de Design JavaScript

Um livro de [Addy Osmani](https://twitter.com/addyosmani)

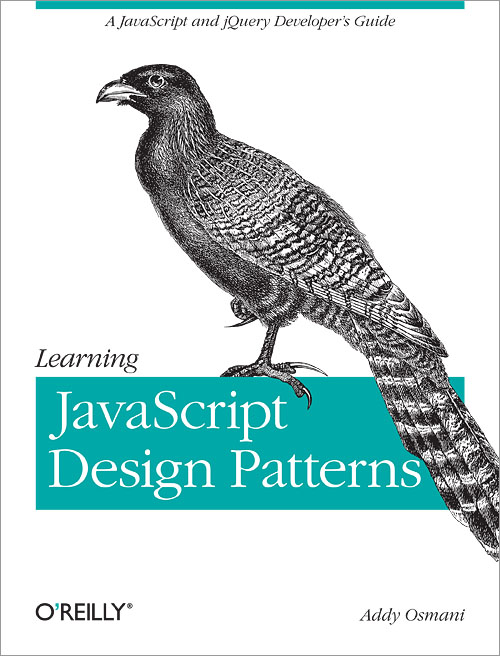
Volume 1.7.0

Parte superior do formulário

Parte inferior do formulário

Copyright © Addy Osmani 2012-2020.

*O Learning JavaScript Design Patterns* é lançado sob uma [licença Creative Commons Atribuição-Não-](https://creativecommons.org/licenses/by-nc-nd/3.0/) comercial [-Nenhum Trabalho Derivativo 3.0 não](https://creativecommons.org/licenses/by-nc-nd/3.0/) portado. Ele está disponível para compra através da [O'Reilly](http://shop.oreilly.com/product/0636920025832.do) Media, mas permanecerá disponível para compra online gratuita e física (ou e-book) para leitores que desejam apoiar o projeto.



Prefácio

Os padrões de design são soluções reutilizáveis ​​para problemas comuns no design de software. Eles são excitantes e um tópico fascinante para explorar em qualquer linguagem de programação.

Uma razão para isso é que eles nos ajudam a construir sobre a experiência combinada de muitos desenvolvedores que vieram antes de nós e garantem que estruturemos nosso código de uma maneira otimizada, atendendo às necessidades dos problemas que estamos tentando resolver.

Os padrões de design também nos fornecem um vocabulário comum para descrever soluções. Isso pode ser significativamente mais simples do que descrever sintaxe e semântica quando estamos tentando transmitir uma maneira de estruturar uma solução em forma de código para outras pessoas.

Neste livro, exploraremos a aplicação de padrões de design clássicos e modernos à linguagem de programação JavaScript.

Público-alvo

Este livro é direcionado a desenvolvedores profissionais que desejam aprimorar seus conhecimentos sobre padrões de design e como eles podem ser aplicados à linguagem de programação JavaScript.

Alguns dos conceitos cobertos (fechamentos, herança prototípica) irão assumir um nível de conhecimento prévio básico e compreensão. Se você precisar ler mais sobre esses tópicos, uma lista de títulos sugeridos é fornecida para sua conveniência.

Se você gostaria de aprender como escrever um código bonito, estruturado e organizado, acredito que este é o livro para você.

Agradecimentos

Sempre serei grato pelos talentosos revisores técnicos que ajudaram a revisar e melhorar este livro, incluindo os da comunidade em geral. O conhecimento e o entusiasmo que trouxeram para o projeto foram simplesmente incríveis. Os tweets e blogs oficiais dos revisores técnicos também são uma fonte regular de ideias e inspiração e recomendo sinceramente que os verifique.

* Nicholas Zakas ( [http://nczonline.net](https://www.nczonline.net/) , [@slicknet](https://twitter.com/slicknet) )
* Andrée Hansson ( [http://andreehansson.se](http://andreehansson.se/) , [@peolanha](https://twitter.com/peolanha) )
* Luke Smith ( [http://lucassmith.name](http://lucassmith.name/) , [@ls\_n](https://twitter.com/ls_n) )
* Eric Ferraiuolo ( <http://ericf.me/> , [@ericf](https://twitter.com/ericf) )
* Peter Michaux ( [http://michaux.ca](http://michaux.ca/) , [@petermichaux](https://twitter.com/petermichaux) )
* Alex Sexton ( [http://alexsexton.com](https://alexsexton.com/) , [@slexaxton](https://twitter.com/slexaxton) )

Também gostaria de agradecer a Rebecca Murphey ( [http://rmurphey.com](http://rmurphey.com/) , [@rmurphey](https://twitter.com/rmurphey) ) por fornecer a inspiração para escrever este livro e, mais importante, continuar a disponibilizá-lo no GitHub e via O'Reilly.

Por fim, gostaria de agradecer à minha maravilhosa esposa Ellie por todo o seu apoio enquanto eu estava preparando esta publicação.

Créditos

Embora alguns dos padrões abordados neste livro tenham sido implementados com base na experiência pessoal, muitos deles foram identificados anteriormente pela comunidade JavaScript. Este trabalho é, como tal, a produção da experiência combinada de vários desenvolvedores. Semelhante à abordagem lógica de Stoyan Stefanov para evitar a interrupção da narrativa com créditos (em *Padrões JavaScript* ), listei os créditos e sugeri a leitura de qualquer conteúdo coberto na seção de referências.

Se algum artigo ou link foi omitido na lista de referências, aceite minhas sinceras desculpas. Se você entrar em contato comigo, terei certeza de atualizá-los para incluí-lo na lista.

Lendo

Embora este livro seja direcionado a desenvolvedores iniciantes e intermediários, pressupõe-se um entendimento básico dos fundamentos do JavaScript. Se você deseja aprender mais sobre o idioma, tenho o prazer de recomendar os seguintes títulos:

* *JavaScript: The Definitive Guide* por David Flanagan
* *Eloquent JavaScript* por Marijn Haverbeke
* *Padrões de JavaScript* por Stoyan Stefanov
* *Escrevendo JavaScript* Sustentável por Nicholas Zakas
* *JavaScript: The Good Parts* de Douglas Crockford

*Índice*

* [Introdução](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#introduction)
* [O que é um padrão?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#whatisapattern)
* [Teste de qualidade de "padrão", protopadrões e a regra de três](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#patternity)
* [A estrutura de um padrão de projeto](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#designpatternstructure)
* [Escrevendo Padrões de Design](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#writingdesignpatterns)
* [Antipadrões](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#antipatterns)
* [Categorias de padrão de design](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#categoriesofdesignpatterns)
* [Tabela de resumo da categorização de padrões de design](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#summarytabledesignpatterns)
* [Padrões de Design JavaScript](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#designpatternsjavascript)
  + [Padrão de Construtor](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#constructorpatternjavascript)
  + [Padrão de Módulo](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#modulepatternjavascript)
  + [Padrão de Módulo Revelador](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#revealingmodulepatternjavascript)
  + [Padrão Singleton](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#singletonpatternjavascript)
  + [Padrão Observador](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#observerpatternjavascript)
  + [Padrão Mediador](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#mediatorpatternjavascript)
  + [Padrão de Protótipo](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#prototypepatternjavascript)
  + [Padrão de Comando](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#commandpatternjavascript)
  + [Padrão de fachada](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#facadepatternjavascript)
  + [Padrão de Fábrica](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#factorypatternjavascript)
  + [Padrão Mixin](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#mixinpatternjavascript)
  + [Padrão Decorator](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#decoratorpatternjavascript)
  + [Padrão Flyweight](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#detailflyweight)
* [Padrões JavaScript MV \*](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#detailmvcmvp)
  + [Padrão MVC](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#detailmvc)
  + [Padrão MVP](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#detailmvp)
  + [Padrão MVVM](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#detailmvvm)
* [Modern Modular JavaScript Design Patterns](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#modularjavascript)
  + [AMD](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#detailamd)
  + [CommonJS](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#detailcommonjs)
  + [ES Harmony](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#detailharmony)
* [Padrões de design em jQuery](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#designpatternsjquery)
  + [Padrão Composto](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#compositepatternjquery)
  + [Padrão Adaptador](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#wrapperpatternjquery)
  + [Padrão de fachada](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#facadepatternjquery)
  + [Padrão Observador](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#observerpatternjquery)
  + [Padrão do Iterador](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#iteratorpatternjquery)
  + [Padrão de inicialização lenta](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#lazyinitialisationjquery)
  + [Padrão Proxy](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#proxypatternjquery)
  + [Padrão de Construtor](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#builderpatternjquery)
* [jQuery Plugin Design Patterns](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#jquerypluginpatterns)
* [Padrões de namespacing de JavaScript](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#detailnamespacing)
* [Conclusões](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#conclusions)
* [Referências](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#references)

Introdução

Um dos aspectos mais importantes da escrita de código sustentável é ser capaz de perceber os temas recorrentes nesse código e otimizá-los. Esta é uma área em que o conhecimento dos padrões de projeto pode ser inestimável.

Na primeira parte deste livro, exploraremos a história e a importância dos padrões de projeto que podem realmente ser aplicados a qualquer linguagem de programação. Se você já vendeu ou está familiarizado com essa história, sinta-se à vontade para pular para o capítulo " [O que é um padrão?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#whatisapattern) " Para continuar lendo.

Os padrões de design podem ser rastreados até os primeiros trabalhos de um arquiteto chamado [Christopher Alexander](https://en.wikipedia.org/wiki/Christopher_Alexander) . Ele costumava escrever publicações sobre sua experiência na solução de problemas de design e como eles se relacionavam com edifícios e cidades. Um dia, ocorreu a Alexander que, quando usado repetidamente, certas construções de design levam a um efeito ideal desejado.

Em colaboração com Sara Ishikawa e Murray Silverstein, Alexander produziu uma linguagem de padrões que ajudaria a capacitar qualquer pessoa que desejasse projetar e construir em qualquer escala. Este foi publicado em 1977 em um artigo intitulado "A Pattern Language", que mais tarde foi lançado como um [livro de](https://www.amazon.co.uk/Pattern-Language-Buildings-Construction-Environmental/dp/0195019199/ref=sr_1_1?s=books&ie=UTF8&qid=1329440685&sr=1-1) capa dura completo .

Cerca de 30 anos atrás, os engenheiros de software começaram a incorporar os princípios sobre os quais Alexander havia escrito na primeira documentação sobre padrões de projeto, que deveria ser um guia para desenvolvedores novatos que buscavam melhorar suas habilidades de codificação. É importante observar que os conceitos por trás dos padrões de design existem na indústria de programação desde seu início, embora de uma forma menos formalizada.

Um dos primeiros e mais icônicos trabalhos formais publicados sobre padrões de projeto em engenharia de software foi um livro em 1995 chamado *Design Patterns: Elements Of Reusable Object-Oriented Software* . Este foi escrito por [Erich Gamma](https://en.wikipedia.org/wiki/Erich_Gamma) , [Richard Helm](https://en.wikipedia.org/w/index.php?title=Richard_Helm&action=edit&redlink=1) , [Ralph Johnson](https://en.wikipedia.org/wiki/Ralph_Johnson) e [John Vlissides](https://en.wikipedia.org/wiki/John_Vlissides) - um grupo que ficou conhecido como Gang of Four (ou GoF para breve).

A publicação do GoF é considerada bastante instrumental para levar o conceito de padrões de design ainda mais longe em nosso campo, pois descreve uma série de técnicas de desenvolvimento e armadilhas, bem como fornece vinte e três padrões de design orientados a objetos centrais freqüentemente usados ​​em todo o mundo hoje. Estaremos cobrindo esses padrões com mais detalhes na seção "Categorias de Padrões de Projeto".

Neste livro, daremos uma olhada em vários padrões de design JavaScript populares e exploraremos por que certos padrões podem ser mais adequados para seus projetos do que outros. Lembre-se de que os padrões podem ser aplicados não apenas ao JavaScript vanilla (ou seja, código JavaScript padrão), mas também a bibliotecas abstratas, como [jQuery](https://jquery.com/) ou [dojo](https://dojotoolkit.org/) . Antes de começar, vamos examinar a definição exata de um "padrão" no design de software.

O que é um padrão?

Um padrão é uma solução reutilizável que pode ser aplicada a problemas comuns em design de software - em nosso caso - ao escrever aplicativos da Web em JavaScript. Outra maneira de ver os padrões são como modelos de como resolvemos problemas - aqueles que podem ser usados ​​em várias situações diferentes.

Então, por que é importante entender os padrões e se familiarizar com eles? Os padrões de design têm três benefícios principais:

1. **Os padrões são soluções comprovadas:** eles fornecem abordagens sólidas para resolver problemas no desenvolvimento de software usando técnicas comprovadas que refletem a experiência e os insights que os desenvolvedores que ajudaram a defini-los trazem para o padrão.
2. **Os padrões podem ser facilmente reutilizados:** Um padrão geralmente reflete uma solução pronta para uso que pode ser adaptada para atender às nossas próprias necessidades. Esse recurso os torna bastante robustos.
3. **Os padrões podem ser expressivos:** quando olhamos para um padrão, geralmente há uma estrutura definida e um *vocabulário* para a solução apresentada que pode ajudar a expressar soluções bastante grandes com bastante elegância.

Os padrões **não** são uma solução exata. É importante lembrarmos que a função de um padrão é meramente nos fornecer um esquema de solução. Os padrões não resolvem todos os problemas de design nem substituem bons designers de software; no entanto, eles **os** suportam. A seguir, daremos uma olhada em algumas das outras vantagens que os padrões têm a oferecer.

* **A reutilização de padrões auxilia na prevenção de pequenos problemas que podem causar grandes problemas no processo de desenvolvimento de aplicativos.**Isso significa que, quando o código é construído em padrões comprovados, podemos perder menos tempo nos preocupando com a estrutura de nosso código e mais tempo focando na qualidade de nossa solução geral. Isso ocorre porque os padrões podem nos encorajar a codificar de uma forma mais estruturada e organizada, evitando a necessidade de refatorá-lo para fins de limpeza no futuro.
* **Os padrões podem fornecer soluções generalizadas que são documentadas de uma maneira que não exige que sejam vinculados a um problema específico.**Essa abordagem generalizada significa que, independentemente do aplicativo (e em muitos casos da linguagem de programação) com o qual estamos trabalhando, os padrões de design podem ser aplicados para melhorar a estrutura do nosso código.
* **Certos padrões podem realmente diminuir a pegada geral do tamanho do arquivo de nosso código, evitando a repetição.**Ao encorajar os desenvolvedores a olhar mais de perto suas soluções para áreas onde reduções instantâneas na repetição podem ser feitas, por exemplo, reduzindo o número de funções que executam processos semelhantes em favor de uma única função generalizada, o tamanho geral de nossa base de código pode ser diminuído. Isso também é conhecido como tornar o código mais *SECO* .
* **Os padrões são adicionados ao vocabulário do desenvolvedor, o que torna a comunicação mais rápida.**
* **Os padrões frequentemente usados ​​podem ser melhorados com o tempo, aproveitando as experiências coletivas de outros desenvolvedores que usam esses padrões e contribuem de volta para a comunidade de padrões de design.**Em alguns casos, isso leva à criação de padrões de design inteiramente novos, enquanto em outros pode levar ao fornecimento de diretrizes aprimoradas sobre como padrões específicos podem ser melhor usados. Isso pode garantir que as soluções baseadas em padrões continuem a se tornar mais robustas do que as soluções ad-hoc.

Já usamos padrões todos os dias

Para entender como os padrões podem ser úteis, vamos revisar um problema de seleção de elemento muito simples que a biblioteca jQuery resolve para nós.

Imagine que temos um script onde para cada elemento DOM encontrado em uma página com a classe "foo" desejamos incrementar um contador. Qual é a maneira mais eficiente de consultar essa coleção de elementos? Bem, existem algumas maneiras diferentes de lidar com esse problema:

1. Selecione todos os elementos na página e armazene as referências a eles. Em seguida, filtre essa coleção e use expressões regulares (ou outros meios) para armazenar apenas aquelas com a classe "foo".
2. Use um recurso de navegador nativo moderno querySelectorAll()para selecionar todos os elementos com a classe "foo".
3. Use um recurso nativo, como getElementsByClassName()para obter de volta a coleção desejada.

Então, qual dessas opções é a mais rápida? Na verdade, é a opção 3. por um fator de 8 a 10 vezes as [alternativas](https://jsperf.com/getelementsbyclassname-vs-queryselectorall/5) . Em um aplicativo do mundo real, no entanto, 3. não funcionará em versões do Internet Explorer abaixo de 9 e, portanto, é necessário usar 1. onde 2. e 3. não são suportados.

Os desenvolvedores que usam jQuery não precisam se preocupar com esse problema, pois felizmente ele foi abstraído para nós usando o padrão *Facade* . Como revisaremos com mais detalhes posteriormente, esse padrão fornece um conjunto simples de interfaces abstratas (por exemplo $el.css(), $el.animate()) para vários corpos de código subjacentes mais complexos. Como vimos, isso significa menos tempo para se preocupar com os detalhes do nível de implementação.

Nos bastidores, a biblioteca simplesmente opta pela abordagem mais ideal para selecionar elementos, dependendo do que nosso navegador atual suporta e nós apenas consumimos a camada de abstração.

Provavelmente, todos nós também estamos familiarizados com o jQuery $("selector"). Isto é significativamente mais fácil de usar para selecionar elementos HTML em uma página contra ter que opt manualmente para getElementById(), getElementsByClassName(), getElementsByTagName()e assim por diante.

Embora saibamos que é uma querySelectorAll()tentativa de resolver esse problema, compare o esforço envolvido no uso das interfaces Façade do jQuery com a seleção dos caminhos de seleção mais adequados. Não há competição! Abstrações usando padrões podem oferecer valor do mundo real.

Veremos isso e mais padrões de projeto posteriormente neste livro.

Teste de qualidade de "padrão", protopadrões e a regra de três

Lembre-se de que nem todo algoritmo, prática recomendada ou solução representa o que pode ser considerado um padrão completo. Pode haver alguns ingredientes-chave que estão faltando aqui e a comunidade de padrões geralmente tem medo de algo que afirma ser um, a menos que tenha sido fortemente controlado. Mesmo que algo seja apresentado a nós que **pareça** atender aos critérios para um padrão, não deve ser considerado um até que tenha passado por períodos adequados de escrutínio e teste por outros.

Revendo o trabalho de Alexander mais uma vez, ele afirma que um padrão deve ser um processo e uma "coisa". Esta definição é obtusa propositalmente, pois ele segue dizendo que é o processo que deve criar a "coisa". Esta é a razão pela qual os padrões geralmente se concentram em abordar uma estrutura visualmente identificável, ou seja, devemos ser capazes de representar visualmente (ou desenhar) uma imagem que representa a estrutura que resulta em colocar o padrão em prática.

Ao estudar padrões de projeto, não é irregular encontrar o termo "protopadrão". O que é isso? Bem, um padrão que ainda não foi conhecido por passar nos testes de "padrão" é geralmente referido como um protopadrão. Os protopadrões podem resultar do trabalho de alguém que estabeleceu uma solução particular que vale a pena compartilhar com a comunidade, mas pode ainda não ter tido a oportunidade de ter sido examinado pesadamente devido à sua tenra idade.

Alternativamente, o (s) indivíduo (s) que compartilham o padrão podem não ter tempo ou interesse de passar pelo processo de identidade do "padrão" e, em vez disso, podem lançar uma breve descrição de seu protopadrão. Breves descrições ou fragmentos desse tipo de padrão são conhecidos como patlets.

O trabalho envolvido na documentação completa de um padrão qualificado pode ser bastante assustador. Olhando para trás, para alguns dos primeiros trabalhos no campo de padrões de design, um padrão pode ser considerado "bom" se fizer o seguinte:

* **Resolve um problema específico** : os padrões não devem apenas capturar princípios ou estratégias. Eles precisam capturar soluções. Este é um dos ingredientes mais essenciais para um bom padrão.
* **A solução para esse problema não pode ser óbvia** : podemos descobrir que as técnicas de solução de problemas freqüentemente tentam derivar de princípios primeiros bem conhecidos. Os melhores padrões de projeto geralmente fornecem soluções para problemas indiretamente - esta é considerada uma abordagem necessária para os problemas mais desafiadores relacionados ao projeto.
* **O conceito descrito deve ter sido comprovado** : os padrões de projeto exigem prova de que funcionam conforme descrito e sem esta prova o design não pode ser considerado seriamente. Se um padrão é de natureza altamente especulativa, apenas os corajosos podem tentar usá-lo.
* **Deve descrever um relacionamento** : em alguns casos, pode parecer que um padrão descreve um tipo de módulo. Embora uma implementação possa aparecer dessa maneira, a descrição oficial do padrão deve descrever estruturas e mecanismos de sistema muito mais profundos que explicam sua relação com o código.

Seríamos perdoados se pensássemos que não vale a pena aprender com um protopadrão que não atende às diretrizes, no entanto, isso está longe da verdade. Muitos proto-padrões são realmente muito bons. Não estou dizendo que vale a pena examinar todos os protopadrões, mas existem alguns poucos úteis na natureza que podem nos ajudar em projetos futuros. Use o bom senso com a lista acima em mente e você estará bem em seu processo de seleção.

Um dos requisitos adicionais para que um padrão seja válido é que eles exibam algum fenômeno recorrente. Freqüentemente, isso é algo que pode ser qualificado em pelo menos três áreas principais, chamadas de *regra de três* . Para mostrar a recorrência usando esta regra, deve-se demonstrar:

1. **Adequação de propósito** - como o padrão é considerado bem-sucedido?
2. **Utilidade** - por que o padrão é considerado bem-sucedido?
3. **Aplicabilidade** - o design é digno de ser um padrão porque tem uma aplicabilidade mais ampla? Nesse caso, isso precisa ser explicado. Ao revisar ou definir um padrão, é importante manter o acima em mente.

A estrutura de um padrão de projeto

Você pode estar curioso sobre como um autor de padrão pode abordar a estrutura de esboço, implementação e propósito de um novo padrão. Um padrão é inicialmente apresentado na forma de uma **regra** que estabelece uma relação entre:

* Um **contexto**
* Um sistema de **forças** que surge nesse contexto e
* Uma **configuração** que permite que essas forças se resolvam no contexto

Com isso em mente, vamos agora dar uma olhada em um resumo dos elementos componentes de um padrão de design. Um padrão de design deve ter:

* **Nome do padrão** e uma **descrição**
* **Esboço de contexto** - os contextos nos quais o padrão é eficaz para responder às necessidades dos usuários.
* **Declaração do problema** - uma declaração do problema que está sendo tratado para que possamos compreender a intenção do padrão.
* **Solução** - uma descrição de como o problema do usuário está sendo resolvido em uma lista compreensível de etapas e percepções.
* **Design** - uma descrição do design do padrão e, em particular, o comportamento do usuário ao interagir com ele
* **Implementação** - um guia de como o padrão seria implementado
* **Ilustrações** - uma representação visual das classes no padrão (por exemplo, um diagrama)
* **Exemplos** - uma implementação do padrão em uma forma mínima
* **Co-requisitos** - que outros padrões podem ser necessários para apoiar o uso do padrão que está sendo descrito?
* **Relações** - com quais padrões esse padrão se assemelha? imita de perto qualquer outro?
* **Uso conhecido** - o padrão está sendo usado em estado *selvagem* ? Se sim, onde e como?
* **Discussões** - a equipe ou os pensamentos do autor sobre os benefícios emocionantes do padrão

Os padrões de design são uma abordagem bastante poderosa para colocar todos os desenvolvedores em uma organização ou equipe na mesma página ao criar ou manter soluções. Se estiver pensando em trabalhar em um padrão próprio, lembre-se de que, embora possam ter um alto custo inicial nas fases de planejamento e redação, o valor retornado desse investimento pode valer a pena. Sempre pesquise completamente antes de trabalhar em novos padrões, no entanto, pois você pode achar mais benéfico usar ou construir sobre os padrões comprovados existentes do que começar de novo.

Escrevendo Padrões de Design

Embora este livro seja voltado para os novos padrões de projeto, uma compreensão fundamental de como um padrão de projeto é escrito pode oferecer uma série de benefícios úteis. Para começar, podemos obter uma apreciação mais profunda do raciocínio por trás da necessidade de um padrão. Também podemos aprender como saber se um padrão (ou protopadrão) está à altura ao revisá-lo para nossas próprias necessidades.

Escrever bons padrões é uma tarefa desafiadora. Os padrões não precisam apenas (idealmente) fornecer uma quantidade substancial de material de referência para os usuários finais, mas também precisam ser capazes de defender por que são necessários.

Tendo lido a seção anterior sobre o *que é* um padrão, podemos pensar que isso por si só é o suficiente para nos ajudar a identificar os padrões que vemos na natureza. Na verdade, isso não é totalmente verdade. Nem sempre fica claro se um trecho de código que estamos examinando está seguindo um padrão definido ou apenas acidentalmente aparece como está.

Quando estamos olhando para um corpo de código que achamos que pode estar usando um padrão, devemos considerar escrever alguns dos aspectos do código que acreditamos se enquadram em um determinado padrão existente ou conjunto de padrões.

Em muitos casos de análise de padrões, podemos descobrir que estamos apenas olhando para o código que segue bons princípios e práticas de design que podem se sobrepor às regras de um padrão por acidente. Lembre-se - soluções nas quais nem interações nem regras definidas aparecem *não* são padrões.

Se estiver interessado em se aventurar no caminho de escrever seus próprios padrões de design, recomendo aprender com outras pessoas que já passaram pelo processo e o fizeram bem. Passe algum tempo absorvendo as informações de várias descrições de padrões de design diferentes e observe o que é significativo para você.

Explore a estrutura e a semântica - isso pode ser feito examinando as interações e o contexto dos padrões nos quais você está interessado, para que possa identificar os princípios que ajudam a organizar esses padrões juntos em configurações úteis.

Depois de nos expor a uma riqueza de informações sobre a literatura de padrões, podemos começar a escrever nosso padrão usando um formato *existente* e ver se podemos debater novas ideias para melhorá-lo ou integrar nossas ideias nele.

Um exemplo de desenvolvedor que fez isso nos últimos anos é Christian Heilmann, que pegou o padrão de *Módulo* existente e fez algumas alterações fundamentalmente úteis nele para criar o padrão de *Módulo Revelador* (este é um dos padrões abordados mais adiante neste livro).

A seguir estão algumas dicas que eu sugeriria se tivesse interesse em criar um novo padrão de design:

* **Quão prático é o padrão?**: Certifique-se de que o padrão descreve soluções comprovadas para problemas recorrentes, em vez de apenas soluções especulativas que não foram qualificadas.
* **Lembre-se das melhores práticas:** as decisões de design que tomamos devem ser baseadas em princípios que derivamos da compreensão das melhores práticas.
* **Nossos padrões de design devem ser transparentes para o usuário** : os padrões de design devem ser totalmente transparentes para qualquer tipo de experiência do usuário. Eles estão lá principalmente para servir aos desenvolvedores que os utilizam e não devem forçar mudanças no comportamento da experiência do usuário que não ocorreria sem o uso de um padrão.
* **Lembre-se de que a originalidade *não* é fundamental no design de padrões** : ao escrever um padrão, não precisamos ser os descobridores originais das soluções que estão sendo documentadas, nem você precisa se preocupar com a sobreposição de nosso design com pequenas peças de outros padrões. Se a abordagem for forte o suficiente para ter ampla aplicabilidade útil, ela tem uma chance de ser reconhecida como um padrão válido.
* **Os padrões precisam de um forte conjunto de exemplos:** uma boa descrição de padrão deve ser seguida por um conjunto igualmente forte de exemplos que demonstrem a aplicação bem-sucedida de nosso padrão. Para mostrar o uso amplo, os exemplos que exibem bons princípios de design são ideais.

A escrita de padrões é um equilíbrio cuidadoso entre a criação de um design geral, específico e, acima de tudo, útil. Tente se certificar de que, ao escrever um padrão, você cobrirá as áreas de aplicação mais amplas possíveis e estará bem. Espero que esta breve introdução aos padrões de escrita tenha dado a você alguns insights que ajudarão seu processo de aprendizado para as próximas seções deste livro.

Antipadrões

Se considerarmos que um padrão representa uma melhor prática, um antipadrão representa uma lição que foi aprendida. O termo antipadrões foi cunhado em 1995 por Andrew Koenig no relatório C ++ de novembro daquele ano, inspirado no livro *Design Patterns* do GoF . No relatório de Koenig, são apresentadas duas noções de antipadrões. Antipadrões:

* Descreva uma solução *ruim* para um problema específico que resultou na ocorrência de uma situação ruim
* Descreva *como* sair dessa situação e como ir a partir daí para uma boa solução

Sobre este tópico, Alexander escreve sobre as dificuldades em alcançar um bom equilíbrio entre uma boa estrutura de design e um bom contexto:

*“Essas notas são sobre o processo de design; o processo de inventar coisas físicas que exibem uma nova ordem física, organização, forma, em resposta à função ... todo problema de design começa com um esforço para alcançar a adequação entre duas entidades: a forma em questão e seu contexto. O formulário é a solução para o problema; o contexto define o problema ”.*

Embora seja muito importante estar ciente dos padrões de design, pode ser igualmente importante entender os antipadrões. Vamos qualificar a razão por trás disso. Ao criar um aplicativo, o ciclo de vida de um projeto começa com a construção, no entanto, uma vez que você tenha concluído o release inicial, ele precisa ser mantido. A qualidade de uma solução final será *boa* ou *ruim* , dependendo do nível de habilidade e do tempo que a equipe investiu nela. Aqui, o *bom* e o *mau* são considerados no contexto - um design 'perfeito' pode ser qualificado como um antipadrão se aplicado no contexto errado.

Os maiores desafios acontecem depois que um aplicativo atinge a produção e está pronto para entrar no modo de manutenção. Um desenvolvedor trabalhando em tal sistema que não trabalhou no aplicativo antes pode introduzir um design *incorreto* no projeto por acidente. Se essas *más* práticas forem criadas como antipadrões, eles permitem aos desenvolvedores um meio de reconhecê-los com antecedência, de modo que possam evitar erros comuns que possam ocorrer - isso é paralelo à maneira como os padrões de design nos fornecem uma maneira de reconhecer os comuns técnicas que são *úteis.*

Para resumir, um antipadrão é um projeto ruim que vale a pena documentar. Exemplos de antipadrões em JavaScript são os seguintes:

* Poluir o namespace global ao definir um grande número de variáveis ​​no contexto global
* Passar strings em vez de funções para setTimeout ou setInterval, pois isso aciona o uso de eval()internamente.
* Modificando o Objectprotótipo de classe (este é um antipadrão particularmente ruim)
* Usar JavaScript em um formulário embutido, pois isso é inflexível
* O uso de document.write onde alternativas DOM nativas, como document.createElement, são mais apropriadas. document.write foi grosseiramente mal utilizado ao longo dos anos e tem algumas desvantagens, incluindo que se for executado após a página ser carregada, ele pode realmente sobrescrever a página em que estamos, enquanto document.createElement não. Podemos ver [aqui](https://jsfiddle.net/addyosmani/6T9vX/) um exemplo ao vivo disso em ação. Ele também não funciona com XHTML, que é outra razão para optar por métodos mais amigáveis ​​ao DOM, como document.createElement.

O conhecimento dos antipadrões é fundamental para o sucesso. Assim que formos capazes de reconhecer esses antipadrões, podemos refatorar nosso código para negá-los, de modo que a qualidade geral de nossas soluções melhore instantaneamente.

Categorias de padrão de design

Um glossário do conhecido livro de design, *Domain-Driven Terms* , afirma corretamente que:

*“Um padrão de design nomeia, abstrai e identifica os principais aspectos de uma estrutura de design comum que o torna útil para a criação de um design orientado a objetos reutilizável. O padrão de design identifica as classes participantes e suas instâncias, suas funções e colaborações e a distribuição de responsabilidades.*

*Cada padrão de design se concentra em um problema ou questão de design orientado a objetos específico. Ele descreve quando se aplica, se pode ou não ser aplicado em vista de outras restrições de design e as consequências e compensações de seu uso. Uma vez que devemos eventualmente implementar nossos designs, um padrão de design também fornece código de amostra para ilustrar uma implementação.*

*Embora os padrões de projeto descrevam projetos orientados a objetos, eles são baseados em soluções práticas que foram implementadas nas principais linguagens de programação orientadas a objetos ... ”*

Os padrões de design podem ser divididos em várias categorias diferentes. Nesta seção, revisaremos três dessas categorias e mencionaremos brevemente alguns exemplos dos padrões que se enquadram nessas categorias antes de explorar alguns específicos em mais detalhes.

Padrões de Design Criativo

Os padrões de design criacionais se concentram em lidar com os mecanismos de criação de objetos em que os objetos são criados de maneira adequada para a situação em que estamos trabalhando. A abordagem básica para a criação de objetos pode, de outra forma, levar a uma complexidade adicional em um projeto, enquanto esses padrões visam resolver esse problema por *controlar* o processo de criação.

Alguns dos padrões que se enquadram nesta categoria são: Construtor, Fábrica, Abstrato, Protótipo, Singleton e Construtor.

Padrões de Projeto Estrutural

Os padrões estruturais se preocupam com a composição do objeto e normalmente identificam maneiras simples de realizar relacionamentos entre diferentes objetos. Eles ajudam a garantir que, quando uma parte de um sistema muda, toda a estrutura do sistema não precisa fazer o mesmo. Eles também auxiliam na reformulação de partes do sistema que não se enquadram em um propósito específico.

Os padrões que se enquadram nesta categoria incluem: Decorator, Facade, Flyweight, Adapter e Proxy.

Padrões de Design Comportamental

Os padrões de comportamento se concentram em melhorar ou otimizar a comunicação entre objetos distintos em um sistema.

Alguns padrões de comportamento incluem: Iterator, Mediator, Observer e Visitor.

Categorização do Design Pattern

Em minhas primeiras experiências de aprendizado sobre padrões de projeto, eu pessoalmente achei a tabela a seguir um lembrete muito útil do que vários padrões têm a oferecer - ela cobre os 23 padrões de projeto mencionados pelo GoF. A tabela original foi resumida por Elyse Nielsen em 2004 e eu a modifiquei quando necessário para se adequar à nossa discussão nesta seção do livro.

Recomendo usar esta tabela como referência, mas lembre-se de que há vários padrões adicionais que não são mencionados aqui, mas serão discutidos posteriormente neste livro.

Uma breve nota sobre as aulas

**Nota:** ES2015 introduziu suporte nativo para classes para JavaScript, no entanto, elas são principalmente uma adição sintática sobre o modelo de herança baseado em protótipo existente do JavaScript. Não cobriremos as classes ES2015 neste livro, mas [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes) tem uma excelente introdução a elas.

Lembre-se de que haverá padrões nesta tabela que fazem referência ao conceito de "classes". No ES5, JavaScript é uma linguagem sem classes, no entanto, as classes podem ser simuladas usando funções.

A abordagem mais comum para conseguir isso é definir uma função JavaScript onde criamos um objeto usando a newpalavra - chave. thispode ser usado para ajudar a definir novas propriedades e métodos para o objeto da seguinte maneira:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | // A car "class"  function Car( model ) {      this.model = model;    this.color = "silver";    this.year = "2012";      this.getInfo = function () {      return this.model + " " + this.year;    };    } |

Podemos então instanciar o objeto usando o construtor Car que definimos acima desta forma:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5 | var myCar = new Car("ford");    myCar.year = "2010";    console.log( myCar.getInfo() ); |

Para obter mais maneiras de definir "classes" usando JavaScript, consulte a [postagem](http://www.phpied.com/3-ways-to-define-a-javascript-class/) útil de Stoyan Stefanov sobre elas.

Vamos agora revisar a tabela.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Criativo** | | | | Baseado no conceito de criação de um objeto. | | | |
| ***Classe*** | | | | | | | |
| *Método de Fábrica* | | | | | Isso cria uma instância de várias classes derivadas com base em dados ou eventos de interface. | | |
| ***Objeto*** | | | | | | | |
| *Fábrica Abstrata* | | | | | Cria uma instância de várias famílias de classes sem detalhar classes concretas. | | |
| *Construtor* | | | | | Separa a construção do objeto de sua representação, sempre cria o mesmo tipo de objeto. | | |
| *Protótipo* | | | | | Uma instância totalmente inicializada usada para cópia ou clonagem. | | |
| *Singleton* | | | | | Uma classe com apenas uma única instância com pontos de acesso globais. | | |
|  |  |  |  |  |  |  |  |
| **Estrutural** | | | | Baseado na ideia de construção de blocos de objetos. | | | |
| ***Classe*** | | | | | | | |
| *Adaptador* | | | | | Corresponder interfaces de classes diferentes, portanto, as classes podem trabalhar juntas, apesar das interfaces incompatíveis. | | |
| ***Objeto*** | | | | | | | |
| *Adaptador* | | | | | Corresponder interfaces de classes diferentes, portanto, as classes podem trabalhar juntas, apesar das interfaces incompatíveis. | | |
| *Ponte* | | | | | Separa a interface de um objeto de sua implementação para que as duas possam variar independentemente. | | |
| *Composto* | | | | | Uma estrutura de objetos simples e compostos que torna o objeto total mais do que apenas a soma de suas partes. | | |
| *Decorador* | | | | | Adicione dinamicamente processamento alternativo aos objetos. | | |
| *Fachada* | | | | | Uma única classe que esconde a complexidade de um subsistema inteiro. | | |
| *Flyweight* | | | | | Uma instância refinada usada para compartilhamento eficiente de informações contidas em outro lugar. | | |
| *Proxy* | | | | | Um objeto de espaço reservado que representa o objeto verdadeiro. | | |
|  | | | | | | | |
| **Comportamental** | | | | Baseado na forma como os objetos brincam e funcionam juntos. | | | |
| ***Classe*** | | | | | | | |
| *Intérprete* | | | | | Uma maneira de incluir elementos de linguagem em um aplicativo para corresponder à gramática do idioma pretendido. | | |
| *Método de       Template* | | | | | Cria o shell de um algoritmo em um método e, a seguir, transfere as etapas exatas para uma subclasse. | | |
| ***Objeto*** | | | | | | | |
| *Cadeia de       Responsabilidade* | | | | | Uma maneira de passar uma solicitação entre uma cadeia de objetos para localizar o objeto que pode lidar com a solicitação. | | |
| *Comando* | | | | | Encapsula uma solicitação de comando como um objeto para habilitar, registrar e / ou enfileirar solicitações e fornece tratamento de erros para solicitações não tratadas. | | |
| *Iterator* | | | | | Acesse sequencialmente os elementos de uma coleção sem conhecer o funcionamento interno da coleção. | | |
| *Mediador* | | | | | Define a comunicação simplificada entre classes para evitar que um grupo de classes se refira explicitamente umas às outras. | | |
| *Lembrança* | | | | | Capture o estado interno de um objeto para poder restaurá-lo posteriormente. | | |
| *Observador* | | | | | Uma forma de notificar mudanças em várias classes para garantir a consistência entre as classes. | | |
| *Estado* | | | | | Altera o comportamento de um objeto quando seu estado muda. | | |
| *Estratégia* | | | | | Encapsula um algoritmo dentro de uma classe que separa a seleção da implementação. | | |
| *Visitante* | | | | | Adiciona uma nova operação a uma classe sem alterar a classe. | | |

Padrões de Design JavaScript

Nesta seção, exploraremos as implementações de JavaScript de vários padrões de design clássicos e modernos.

Os desenvolvedores geralmente se perguntam se há um padrão *ideal* ou conjunto de padrões que deveriam ser usados ​​em seu fluxo de trabalho. Não há uma única resposta verdadeira para essa pergunta; cada script e aplicativo da web em que trabalhamos provavelmente tem suas próprias necessidades individuais e precisamos pensar sobre onde sentimos que um padrão pode oferecer valor real para uma implementação.

Por exemplo, alguns projetos podem se beneficiar dos benefícios de desacoplamento oferecidos pelo padrão Observer (que reduz o quanto as partes dependentes de um aplicativo são umas das outras), enquanto outros podem simplesmente ser pequenos demais para que o desacoplamento seja uma preocupação.

Dito isso, uma vez que temos um domínio firme dos padrões de projeto e dos problemas específicos para os quais eles são mais adequados, fica muito mais fácil integrá-los em nossas arquiteturas de aplicativos.

**Os padrões que exploraremos nesta seção são:**

* [Padrão de Construtor](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#constructorpatternjavascript)
* [Padrão de Módulo](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#modulepatternjavascript)
* [Padrão de Módulo Revelador](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#revealingmodulepatternjavascript)
* [Padrão Singleton](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#singletonpatternjavascript)
* [Padrão Observador](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#observerpatternjavascript)
* [Padrão Mediador](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#mediatorpatternjavascript)
* [Padrão de Protótipo](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#prototypepatternjavascript)
* [Padrão de Comando](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#commandpatternjavascript)
* [Padrão de fachada](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#facadepatternjavascript)
* [Padrão de Fábrica](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#factorypatternjavascript)
* [Padrão Mixin](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#mixinpatternjavascript)
* [Padrão Decorator](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#decoratorpatternjavascript)
* [Padrão Flyweight](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html#detailflyweight)

O Padrão do Construtor

Em linguagens de programação orientadas a objetos clássicas, um construtor é um método especial usado para inicializar um objeto recém-criado, uma vez que a memória foi alocada para ele. Em JavaScript, como quase tudo é um objeto, geralmente estamos interessados ​​em construtores de *objetos* .

Os construtores de objetos são usados ​​para criar tipos específicos de objetos - preparando o objeto para uso e aceitando argumentos que um construtor pode usar para definir os valores das propriedades e métodos do membro quando o objeto é criado pela primeira vez.

Criação de Objetos

As três maneiras comuns de criar novos objetos em JavaScript são as seguintes:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // Each of the following options will create a new empty object:    var newObject = {};    // or  var newObject = Object.create( Object.prototype );    // or  var newObject = new Object(); |

Onde o construtor "Object" no exemplo final cria um wrapper de objeto para um valor específico, ou onde nenhum valor é passado, ele criará um objeto vazio e o retornará.

Existem, então, quatro maneiras pelas quais chaves e valores podem ser atribuídos a um objeto:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79 | // ECMAScript 3 compatible approaches    // 1. Dot syntax    // Set properties  newObject.someKey = "Hello World";    // Get properties  var value = newObject.someKey;        // 2. Square bracket syntax    // Set properties  newObject["someKey"] = "Hello World";    // Get properties  var value = newObject["someKey"];        // ECMAScript 5 only compatible approaches  // For more information see: <http://kangax.github.com/es5-compat-table/>    // 3. Object.defineProperty    // Set properties  Object.defineProperty( newObject, "someKey", {      value: "for more control of the property's behavior",      writable: true,      enumerable: true,      configurable: true  });    // If the above feels a little difficult to read, a short-hand could  // be written as follows:    var defineProp = function ( obj, key, value ){    var config = {      value: value,      writable: true,      enumerable: true,      configurable: true    };    Object.defineProperty( obj, key, config );  };    // To use, we then create a new empty "person" object  var person = Object.create( Object.prototype );    // Populate the object with properties  defineProp( person, "car", "Delorean" );  defineProp( person, "dateOfBirth", "1981" );  defineProp( person, "hasBeard", false );    console.log(person);  // Outputs: Object {car: "Delorean", dateOfBirth: "1981", hasBeard: false}      // 4. Object.defineProperties    // Set properties  Object.defineProperties( newObject, {      "someKey": {      value: "Hello World",      writable: true    },      "anotherKey": {      value: "Foo bar",      writable: false    }    });    // Getting properties for 3. and 4. can be done using any of the  // options in 1. and 2. |

Como veremos um pouco mais adiante no livro, esses métodos podem até ser usados ​​para herança, da seguinte maneira:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | // Usage:    // Create a race car driver that inherits from the person object  var driver = Object.create( person );    // Set some properties for the driver  defineProp(driver, "topSpeed", "100mph");    // Get an inherited property (1981)  console.log( driver.dateOfBirth );    // Get the property we set (100mph)  console.log( driver.topSpeed ); |

Construtores Básicos

Como vimos antes, o JavaScript não oferece suporte ao conceito de classes, mas oferece suporte a funções construtoras especiais que funcionam com objetos. Simplesmente prefixando uma chamada a uma função construtora com a palavra-chave "novo", podemos dizer ao JavaScript que gostaríamos que a função se comportasse como um construtor e instanciar um novo objeto com os membros definidos por essa função.

Dentro de um construtor, a palavra *-* chave *this faz* referência ao novo objeto que está sendo criado. Revisitando a criação do objeto, um construtor básico pode ter a seguinte aparência:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | 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";    };  }    // Usage:    // We can create new instances of the car  var civic = new Car( "Honda Civic", 2009, 20000 );  var mondeo = new Car( "Ford Mondeo", 2010, 5000 );    // and then open our browser console to view the  // output of the toString() method being called on  // these objects  console.log( civic.toString() );  console.log( mondeo.toString() ); |

O exemplo acima é uma versão simples do padrão do construtor, mas apresenta alguns problemas. Uma é que torna a herança difícil e a outra é que funções como toString()são redefinidas para cada um dos novos objetos criados com o construtor Car. Isso não é muito ideal, pois a função deve ser idealmente compartilhada entre todas as instâncias do tipo Carro.

Felizmente, como existem várias alternativas compatíveis com ES3 e ES5 para a construção de objetos, é trivial contornar essa limitação.

Construtores com protótipos

As funções, como quase todos os objetos em JavaScript, contêm um objeto "protótipo". Quando chamamos um construtor JavaScript para criar um objeto, todas as propriedades do protótipo do construtor são disponibilizadas para o novo objeto. Desta forma, vários objetos Car podem ser criados que acessam o mesmo protótipo. Podemos, portanto, estender o exemplo original da seguinte maneira:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | function Car( model, year, miles ) {      this.model = model;    this.year = year;    this.miles = miles;    }      // Note here that we are using Object.prototype.newMethod rather than  // Object.prototype so as to avoid redefining the prototype object  Car.prototype.toString = function () {    return this.model + " has done " + this.miles + " miles";  };    // Usage:    var civic = new Car( "Honda Civic", 2009, 20000 );  var mondeo = new Car( "Ford Mondeo", 2010, 5000 );    console.log( civic.toString() );  console.log( mondeo.toString() ); |

Acima, uma única instância de toString () agora será compartilhada entre todos os objetos Car.

O Padrão do Módulo

Módulos

Os módulos são uma parte integrante da arquitetura de qualquer aplicativo robusto e normalmente ajudam a manter as unidades de código de um projeto separadas e organizadas de maneira limpa.

Em JavaScript, existem várias opções para implementar módulos. Esses incluem:

* O padrão do módulo
* Notação literal de objeto
* Módulos AMD
* Módulos CommonJS
* Módulos ECMAScript Harmony

Exploraremos as últimas três opções mais adiante no livro, na seção *Modern Modular JavaScript Design Patterns* .

O padrão de módulo é baseado em parte em literais de objeto e, portanto, faz sentido atualizar nosso conhecimento sobre eles primeiro.

Literais de objeto

Na notação literal de objeto, um objeto é descrito como um conjunto de pares de nome / valor separados por vírgula entre chaves ( {}). Os nomes dentro do objeto podem ser strings ou identificadores que são seguidos por dois pontos. Não deve haver vírgula após o par nome / valor final no objeto, pois isso pode resultar em erros.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | var myObjectLiteral = {        variableKey: variableValue,        functionKey: function () {        // ...      }  }; |

Literais de objeto não requerem instanciação usando o newoperador, mas não devem ser usados ​​no início de uma instrução, pois a abertura {pode ser interpretada como o início de um bloco. Fora de um objeto, novos membros podem ser adicionados a ele usando as designações a seguirmyModule.property = "someValue";

Abaixo, podemos ver um exemplo mais completo de um módulo definido usando a notação literal de objeto:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | var myModule = {      myProperty: "someValue",      // object literals can contain properties and methods.    // e.g we can define a further object for module configuration:    myConfig: {      useCaching: true,      language: "en"    },      // a very basic method    saySomething: function () {      console.log( "Where in the world is Paul Irish today?" );    },      // output a value based on the current configuration    reportMyConfig: function () {      console.log( "Caching is: " + ( this.myConfig.useCaching ? "enabled" : "disabled") );    },      // override the current configuration    updateMyConfig: function( newConfig ) {        if ( typeof newConfig === "object" ) {        this.myConfig = newConfig;        console.log( this.myConfig.language );      }    }  };    // Outputs: Where in the world is Paul Irish today?  myModule.saySomething();    // Outputs: Caching is: enabled  myModule.reportMyConfig();    // Outputs: fr  myModule.updateMyConfig({    language: "fr",    useCaching: false  });    // Outputs: Caching is: disabled  myModule.reportMyConfig(); |

O uso de literais de objeto pode ajudar a encapsular e organizar seu código, e Rebecca Murphey já escreveu sobre esse tópico em [profundidade,](http://rmurphey.com/blog/2009/10/15/using-objects-to-organize-your-code/) caso você queira ler mais literais de objeto.

Dito isso, se estivermos optando por essa técnica, podemos estar igualmente interessados ​​no padrão de Módulo. Ele ainda usa literais de objeto, mas apenas como o valor de retorno de uma função de escopo.

O Padrão do Módulo

O padrão de Módulo foi originalmente definido como uma forma de fornecer encapsulamento público e privado para aulas de engenharia de software convencional.

Em JavaScript, o padrão de módulo é usado para *emular* ainda mais o conceito de classes de forma que possamos incluir métodos públicos / privados e variáveis ​​dentro de um único objeto, protegendo assim partes específicas do escopo global. Isso resulta em uma redução na probabilidade de nossos nomes de função conflitarem com outras funções definidas em scripts adicionais na página.

Privacidade

O padrão de Módulo encapsula "privacidade", estado e organização usando fechamentos. Ele fornece uma maneira de agrupar uma mistura de métodos e variáveis ​​públicos e privados, protegendo as peças contra vazamento no escopo global e colisão acidental com a interface de outro desenvolvedor. Com esse padrão, apenas uma API pública é retornada, mantendo todo o resto dentro do encerramento privado.

Isso nos dá uma solução limpa para blindar a lógica que faz o trabalho pesado, enquanto apenas expõe uma interface que desejamos que outras partes de nosso aplicativo usem. O padrão utiliza uma expressão de função chamada imediatamente ( [IIFE](http://benalman.com/news/2010/11/immediately-invoked-function-expression/) - consulte a seção sobre padrões de namespacing para obter mais informações), onde um objeto é retornado.

Deve-se notar que não há realmente um sentido explicitamente verdadeiro de "privacidade" dentro do JavaScript porque, ao contrário de algumas linguagens tradicionais, ele não tem modificadores de acesso. Variáveis ​​não podem ser declaradas tecnicamente como sendo públicas ou privadas, então usamos o escopo da função para simular este conceito. Dentro do padrão do Módulo, as variáveis ​​ou métodos declarados só estão disponíveis dentro do próprio módulo graças ao fechamento. Variáveis ​​ou métodos definidos no objeto de retorno, entretanto, estão disponíveis para todos.

História

De uma perspectiva histórica, o padrão do Módulo foi originalmente desenvolvido por várias pessoas, incluindo [Richard Cornford](https://groups.google.com/group/comp.lang.javascript/msg/9f58bd11bd67d937) em 2003. Mais tarde, foi popularizado por Douglas Crockford em suas palestras. Outra curiosidade é que, se você já brincou com a biblioteca YUI do Yahoo, alguns de seus recursos podem parecer bastante familiares e a razão para isso é que o padrão do Módulo foi uma forte influência para o YUI ao criar seus componentes.

Exemplos

Vamos começar examinando uma implementação do padrão de Módulo criando um módulo que é independente.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | 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: counter value prior to reset: 1  testModule.resetCounter(); |

Aqui, outras partes do código não conseguem ler diretamente o valor de nosso incrementCounter()ou resetCounter(). A variável de contador é, na verdade, totalmente protegida de nosso escopo global, de modo que age exatamente como uma variável privada faria - sua existência é limitada ao fechamento do módulo, de modo que o único código capaz de acessar seu escopo são nossas duas funções. Nossos métodos têm um namespace eficaz, portanto, na seção de teste de nosso código, precisamos prefixar todas as chamadas com o nome do módulo (por exemplo, "testModule").

Ao trabalhar com o padrão de Módulo, podemos achar útil definir um modelo simples que usamos para começar com ele. Aqui está um que cobre namespacing, variáveis ​​públicas e privadas:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | 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 );        }    };    })(); |

Olhando outro exemplo, abaixo podemos ver uma cesta de compras implementada usando este padrão. O próprio módulo é totalmente autocontido em uma variável global chamada basketModule. A basketmatriz no módulo é mantida privada e, portanto, outras partes de nosso aplicativo não podem lê-la diretamente. Ele só existe com o fechamento do módulo e, portanto, os únicos métodos capazes de acessá-lo são aqueles com acesso ao seu escopo (ou seja addItem(), getItemCount()etc).

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | 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;      }    };  })(); |

Dentro do módulo, você deve ter notado que retornamos um object. Isso é atribuído automaticamente para basketModuleque possamos interagir com ele da seguinte maneira:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | // 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  // public API  console.log( basketModule.basket );    // This also won't work as it only exists within the scope of our  // basketModule closure, but not in the returned public object  console.log( basket ); |

Os métodos acima são efetivamente com namespaces internos basketModule.

Observe como a função de escopo no módulo basket acima envolve todas as nossas funções, as quais chamamos e armazenamos imediatamente o valor de retorno. Isso tem uma série de vantagens, incluindo:

* A liberdade de ter funções privadas e membros privados que só podem ser consumidos pelo nosso módulo. Como eles não são expostos ao resto da página (apenas nossa API exportada é), eles são considerados realmente privados.
* Dado que as funções são declaradas normalmente e são nomeadas, pode ser mais fácil mostrar as pilhas de chamadas em um depurador quando estamos tentando descobrir quais funções lançaram uma exceção.
* Como TJ Crowder apontou no passado, ele também nos permite retornar funções diferentes dependendo do ambiente. No passado, vi desenvolvedores usarem isso para realizar testes de UA a fim de fornecer um caminho de código em seu módulo específico para o IE, mas podemos facilmente optar pela detecção de recursos hoje em dia para atingir um objetivo semelhante.

Variações de padrão de módulo

**Importar mixins**

Esta variação do padrão demonstra como os globais (por exemplo, jQuery, Underscore) podem ser passados ​​como argumentos para a função anônima do nosso módulo. Isso efetivamente nos permite *importá-* los e criar um alias local para eles, conforme desejarmos.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | // 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(); |

**Exportações**

Esta próxima variação nos permite declarar globais sem consumi-los e poderia da mesma forma apoiar o conceito de importações globais visto no último exemplo.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | // 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;    })(); |

Implementações de padrão de módulo específicas para kit de ferramentas e estrutura

**Dojo**

O Dojo fornece um método conveniente para trabalhar com objetos chamados dojo.setObject(). Isso leva como seu primeiro argumento uma string separada por pontos, como a myObj.parent.childque se refere a uma propriedade chamada "filho" dentro de um objeto "pai" definido dentro de "meuObj". Usar setObject()nos permite definir o valor dos filhos, criando qualquer um dos objetos intermediários no resto do caminho passado, se ainda não existirem.

Por exemplo, se quisermos declarar basket.corecomo um objeto do storenamespace, isso pode ser feito da seguinte maneira, usando a maneira tradicional:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | var store = window.store || {};    if ( !store["basket"] ) {    store.basket = {};  }    if ( !store.basket["core"] ) {    store.basket.core = {};  }    store.basket.core = {    // ...rest of our logic  }; |

Ou da seguinte forma, usando Dojo 1.7 (versão compatível com AMD) e superior:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | 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();            }        };      })());    }); |

Para obter mais informações sobre dojo.setObject(), consulte a [documentação](https://dojotoolkit.org/reference-guide/1.7/dojo/setObject.html) oficial .

**ExtJS**

Para aqueles que usam ExtJS do Sencha, um exemplo que demonstra como usar corretamente o padrão Módulo com a estrutura pode ser encontrado abaixo.

Aqui, vemos um exemplo de como definir um namespace que pode ser preenchido com um módulo contendo uma API privada e uma API pública. Com exceção de algumas diferenças semânticas, é muito próximo de como o padrão de Módulo é implementado em JavaScript vanilla:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | // 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**

Da mesma forma, também podemos implementar o padrão de módulo ao construir aplicativos usando YUI3. O exemplo a seguir é fortemente baseado na implementação original do padrão do Módulo YUI por Eric Miraglia, mas, novamente, não é muito diferente da versão do JavaScript básico:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | Y.namespace( "store.basket" ) ;  Y.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**

Existem várias maneiras pelas quais o código jQuery não específico para plug-ins pode ser empacotado dentro do padrão de módulo. Ben Cherry sugeriu anteriormente uma implementação onde um wrapper de função é usado em torno de definições de módulo no caso de haver uma série de pontos em comum entre os módulos.

No exemplo a seguir, libraryé definida uma função que declara uma nova biblioteca e automaticamente vincula a initfunção document.readyquando novas bibliotecas (isto é, módulos) são criadas.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | function library( module ) {      $( function() {      if ( module.init ) {        module.init();      }    });      return module;  }    var myLibrary = library(function () {      return {      init: function () {        // module implementation      }    };  }()); |

Vantagens

Vimos por que o padrão Construtor pode ser útil, mas por que o padrão Módulo é uma boa escolha? Para começar, é muito mais claro para desenvolvedores que vêm de um background orientado a objetos do que a ideia de encapsulamento verdadeiro, pelo menos de uma perspectiva JavaScript.

Em segundo lugar, ele suporta dados privados - então, no padrão de Módulo, as partes públicas do nosso código são capazes de tocar as partes privadas, no entanto, o mundo externo é incapaz de tocar as partes privadas da classe (sem risos! Oh, e graças a David Engfer para a piada).

Desvantagens

As desvantagens do padrão Módulo são que, como acessamos membros públicos e privados de maneira diferente, quando desejamos alterar a visibilidade, na verdade temos que fazer alterações em cada local em que o membro foi usado.

Também não podemos acessar membros privados em métodos que são adicionados ao objeto posteriormente. Dito isso, em muitos casos o padrão Module ainda é bastante útil e quando usado corretamente, certamente tem o potencial de melhorar a estrutura de nossa aplicação.

Outras desvantagens incluem a incapacidade de criar testes de unidade automatizados para membros privados e complexidade adicional quando os bugs requerem hot fixes. Simplesmente não é possível corrigir partes privadas. Em vez disso, deve-se substituir todos os métodos públicos que interagem com os privates com erros. Os desenvolvedores também não podem estender os privates com facilidade, então vale a pena lembrar que os privates não são tão flexíveis quanto podem parecer inicialmente.

Para ler mais sobre o padrão de módulo, consulte o excelente [artigo](http://www.adequatelygood.com/2010/3/JavaScript-Module-Pattern-In-Depth) detalhado de Ben Cherry sobre ele.

O padrão de módulo revelador

Agora que estamos um pouco mais familiarizados com o padrão de módulo, vamos dar uma olhada em uma versão ligeiramente melhorada - o padrão Revealing Module de Christian Heilmann.

O padrão Revealing Module surgiu quando Heilmann estava frustrado com o fato de que ele tinha que repetir o nome do objeto principal quando queríamos chamar um método público de outro ou acessar variáveis ​​públicas. Ele também não gostou do requisito do padrão de Módulo de ter que mudar para a notação literal de objeto para as coisas que ele desejava tornar públicas.

O resultado de seus esforços foi um padrão atualizado onde simplesmente definiríamos todas as nossas funções e variáveis ​​no escopo privado e retornaríamos um objeto anônimo com ponteiros para a funcionalidade privada que desejávamos revelar como pública.

Um exemplo de como usar o padrão do Módulo Revelador pode ser encontrado abaixo:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | 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" ); |

O padrão também pode ser usado para revelar funções e propriedades privadas com um esquema de nomenclatura mais específico, se preferirmos:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | 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(); |

**Vantagens**

Esse padrão permite que a sintaxe de nossos scripts seja mais consistente. Também torna mais claro no final do módulo quais das nossas funções e variáveis ​​podem ser acessadas publicamente, o que facilita a legibilidade.

**Desvantagens**

Uma desvantagem desse padrão é que se uma função privada se refere a uma função pública, essa função pública não pode ser substituída se um patch for necessário. Isso ocorre porque a função privada continuará a se referir à implementação privada e o padrão não se aplica a membros públicos, apenas a funções.

Membros de objetos públicos que se referem a variáveis ​​privadas também estão sujeitos às notas de regra sem patch acima.

Como resultado disso, os módulos criados com o padrão Revealing Module podem ser mais frágeis do que aqueles criados com o padrão Module original, portanto, deve-se tomar cuidado durante o uso.

O Padrão Singleton

O padrão Singleton é, portanto, conhecido porque restringe a instanciação de uma classe a um único objeto. Classicamente, o padrão Singleton pode ser implementado criando uma classe com um método que cria uma nova instância da classe, se ainda não houver uma. No caso de uma instância já existir, ele simplesmente retorna uma referência a esse objeto.

Singletons diferem de *classes* (ou objetos) estáticos *,* pois podemos atrasar sua inicialização, geralmente porque eles requerem algumas informações que podem não estar disponíveis durante o tempo de inicialização. Eles não fornecem uma maneira para o código que não conhece uma referência anterior a eles recuperá-los facilmente. Isso ocorre porque não é o objeto ou "classe" que é retornado por um Singleton, é uma estrutura. Pense em como as variáveis ​​fechadas não são realmente fechamentos - o escopo da função que fornece o fechamento é o fechamento.

Em JavaScript, os singletons servem como um namespace de recurso compartilhado que isola o código de implementação do namespace global de modo a fornecer um único ponto de acesso para funções.

Podemos implementar um Singleton da seguinte maneira:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102 | var mySingleton = (function () {      // Instance stores a reference to the Singleton    var instance;      function init() {        // Singleton        // Private methods and variables      function privateMethod(){          console.log( "I am private" );      }        var privateVariable = "Im also private";        var privateRandomNumber = Math.random();        return {          // Public methods and variables        publicMethod: function () {          console.log( "The public can see me!" );        },          publicProperty: "I am also public",          getRandomNumber: function() {          return privateRandomNumber;        }        };      };      return {        // Get the Singleton instance if one exists      // or create one if it doesn't      getInstance: function () {          if ( !instance ) {          instance = init();        }          return instance;      }      };    })();    var myBadSingleton = (function () {      // Instance stores a reference to the Singleton    var instance;      function init() {        // Singleton        var privateRandomNumber = Math.random();        return {          getRandomNumber: function() {          return privateRandomNumber;        }        };      };      return {        // Always create a new Singleton instance      getInstance: function () {          instance = init();          return instance;      }      };    })();      // Usage:    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    // Note: as we are working with random numbers, there is a  // mathematical possibility both numbers will be the same,  // however unlikely. The above example should otherwise still  // be valid. |

O que torna o Singleton é o acesso global à instância (geralmente por meio MySingleton.getInstance()), uma vez que não chamamos (pelo menos em linguagens estáticas) new MySingleton()diretamente. No entanto, isso é possível em JavaScript.

No livro GoF, a *aplicabilidade* do padrão Singleton é descrita da seguinte forma:

* Deve haver exatamente uma instância de uma classe e deve estar acessível aos clientes de um ponto de acesso conhecido.
* Quando a única instância deve ser extensível por subclasse, e os clientes devem ser capazes de usar uma instância estendida sem modificar seu código.

O segundo desses pontos refere-se a um caso em que podemos precisar de códigos como:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | mySingleton.getInstance = function(){    if ( this.\_instance == null ) {      if ( isFoo() ) {         this.\_instance = new FooSingleton();      } else {         this.\_instance = new BasicSingleton();      }    }    return this.\_instance;  }; |

Aqui, getInstancetorna-se um pouco como um método Factory e não precisamos atualizar cada ponto de nosso código para acessá-lo. FooSingletonacima seria uma subclasse de BasicSingletone implementaria a mesma interface.

Por que adiar a execução é considerado importante para um Singleton ?:

Em C ++ serve para isolar da imprevisibilidade da ordem de inicialização dinâmica, devolvendo o controle ao programador.

É importante notar a diferença entre uma instância estática de uma classe (objeto) e um Singleton: embora um Singleton possa ser implementado como uma instância estática, ele também pode ser construído lentamente, sem a necessidade de recursos ou memória até que seja realmente necessário.

Se tivermos um objeto estático que pode ser inicializado diretamente, precisamos garantir que o código seja sempre executado na mesma ordem (por exemplo, no caso de ser objCarnecessário objWheeldurante sua inicialização) e isso não pode ser escalado quando você tem um grande número de arquivos de origem.

Tanto Singletons quanto objetos estáticos são úteis, mas não devem ser usados ​​demais - da mesma forma que não devemos usar demais outros padrões.

Na prática, o padrão Singleton é útil quando exatamente um objeto é necessário para coordenar outros em um sistema. Aqui está um exemplo com o padrão sendo usado neste contexto:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | 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 ); |

Embora o Singleton tenha usos válidos, geralmente quando nos encontramos precisando dele em JavaScript, é um sinal de que podemos precisar reavaliar nosso design.

Freqüentemente, são uma indicação de que os módulos em um sistema estão fortemente acoplados ou que a lógica está excessivamente espalhada por várias partes de uma base de código. Singletons podem ser mais difíceis de testar devido a problemas que variam de dependências ocultas, dificuldade em criar várias instâncias, dificuldade em criar stub para dependências e assim por diante.

Miller Medeiros recomendou anteriormente [este](https://www.ibm.com/developerworks/webservices/library/co-single/index.html) excelente artigo sobre o Singleton e seus vários números para leitura posterior, bem como os comentários a [este](http://misko.hevery.com/2008/10/21/dependency-injection-myth-reference-passing/) artigo, discutindo como os Singletons podem aumentar o acoplamento estreito. Fico feliz em apoiar essas recomendações, pois ambas as peças levantam muitos pontos importantes sobre esse padrão que também merecem ser observados.

O Padrão Observer

O Observer é um padrão de design em que um objeto (conhecido como sujeito) mantém uma lista de objetos que dependem dele (observadores), notificando-os automaticamente sobre qualquer mudança de estado.

Quando um assunto precisa notificar os observadores sobre algo interessante acontecendo, ele transmite uma notificação aos observadores (que pode incluir dados específicos relacionados ao assunto da notificação).

Quando não desejamos mais que um determinado observador seja notificado de mudanças pelo sujeito com o qual está registrado, o sujeito pode removê-lo da lista de observadores.

Muitas vezes é útil consultar as definições publicadas de padrões de design que são agnósticos de linguagem para obter um sentido mais amplo de seu uso e vantagens ao longo do tempo. A definição do padrão Observer fornecida no livro GoF, *Design Patterns: Elements of Reusable Object-Oriented Software* , é:

*“Um ou mais observadores estão interessados ​​no estado de um assunto e registram seu interesse com o assunto anexando-se. Quando algo muda em nosso assunto que o observador possa estar interessado, é enviada uma mensagem de notificação que chama o método de atualização em cada observador. Quando o observador não está mais interessado no estado do sujeito, ele pode simplesmente se destacar. "*

Agora podemos expandir o que aprendemos para implementar o padrão Observer com os seguintes componentes:

* **Assunto** : mantém uma lista de observadores, facilita a adição ou remoção de observadores
* **Observer** : fornece uma interface de atualização para objetos que precisam ser notificados sobre as mudanças de estado de um Assunto
* **ConcreteSubject** : transmite notificações para observadores sobre mudanças de estado, armazena o estado de ConcreteObservers
* **ConcreteObserver** : armazena uma referência para o ConcreteSubject, implementa uma interface de atualização para o Observer para garantir que o estado seja consistente com o do Assunto

Primeiro, vamos modelar a lista de Observadores dependentes que um sujeito pode ter:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | function ObserverList(){    this.observerList = [];  }    ObserverList.prototype.add = function( obj ){    return this.observerList.push( obj );  };    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.indexOf = function( obj, startIndex ){    var i = startIndex;      while( i < this.observerList.length ){      if( this.observerList[i] === obj ){        return i;      }      i++;    }      return -1;  };    ObserverList.prototype.removeAt = function( index ){    this.observerList.splice( index, 1 );  }; |

A seguir, vamos modelar o Assunto e a capacidade de adicionar, remover ou notificar observadores na lista de observadores.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | 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 );    }  }; |

Em seguida, definimos um esqueleto para a criação de novos Observadores. A updatefuncionalidade aqui será substituída posteriormente pelo comportamento personalizado.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // The Observer  function Observer(){    this.update = function(){      // ...    };  } |

Em nosso aplicativo de amostra usando os componentes do Observer acima, agora definimos:

* Um botão para adicionar novas caixas de seleção observáveis ​​à página
* Uma caixa de seleção de controle que atuará como um assunto, notificando outras caixas de seleção que devem ser marcadas
* Um contêiner para as novas caixas de seleção adicionadas

Em seguida, definimos os manipuladores ConcreteSubject e ConcreteObserver para adicionar novos observadores à página e implementar a interface de atualização. Veja abaixo os comentários em linha sobre o que esses componentes fazem no contexto do nosso exemplo.

**HTML:**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3 | <button id="addNewObserver">Add New Observer checkbox</button>  <input id="mainCheckbox" type="checkbox"/>  <div id="observersContainer"></div> |

**Script de amostra:**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | // Extend an object with an extension  function extend( obj, extension ){    for ( var key in extension ){      obj[key] = extension[key];    }  }    // References to our DOM elements    var controlCheckbox = document.getElementById( "mainCheckbox" ),    addBtn = document.getElementById( "addNewObserver" ),    container = document.getElementById( "observersContainer" );      // Concrete Subject    // Extend the controlling checkbox with the Subject class  extend( controlCheckbox, new Subject() );    // Clicking the checkbox will trigger notifications to its observers  controlCheckbox.onclick = function(){    controlCheckbox.notify( controlCheckbox.checked );  };    addBtn.onclick = addNewObserver;    // Concrete Observer    function addNewObserver(){      // Create a new checkbox to be added    var check = document.createElement( "input" );    check.type = "checkbox";      // Extend the checkbox with the Observer class    extend( check, new Observer() );      // Override with custom update behaviour    check.update = function( value ){      this.checked = value;    };      // Add the new observer to our list of observers    // for our main subject    controlCheckbox.addObserver( check );      // Append the item to the container    container.appendChild( check );  } |

Neste exemplo, vimos como implementar e utilizar o padrão Observer, abrangendo os conceitos de Subject, Observer, ConcreteSubject e ConcreteObserver.

Diferenças entre o observador e o padrão de publicação / assinatura

Embora seja útil estar ciente do padrão Observer, muitas vezes no mundo do JavaScript, vamos descobri-lo comumente implementado usando uma variação conhecida como o padrão Publicar / Assinar. Embora muito semelhantes, existem diferenças entre esses padrões dignos de nota.

O padrão Observer requer que o observador (ou objeto) que deseja receber notificações de tópico deve assinar esse interesse para o objeto que dispara o evento (o assunto).

O padrão Publicar / Assinar, entretanto, usa um canal de tópico / evento que fica entre os objetos que desejam receber notificações (assinantes) e o objeto que dispara o evento (o editor). Este sistema de eventos permite que o código defina eventos específicos do aplicativo que podem passar argumentos personalizados contendo os valores necessários ao assinante. A ideia aqui é evitar dependências entre o assinante e o editor.

Isso difere do padrão Observer, pois permite que qualquer assinante que implementa um manipulador de eventos apropriado se registre e receba notificações de tópico transmitidas pelo editor.

Aqui está um exemplo de como alguém pode usar o Publicar / Assinar, se fornecido com uma implementação funcional publish(), subscribe()e unsubscribe()nos bastidores:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | // A very simple new mail handler    // A count of the number of messages received  var mailCounter = 0;    // Initialize subscribers that will listen out for a topic  // with the name "inbox/newMessage".    // Render a preview of new messages  var subscriber1 = subscribe( "inbox/newMessage", function( topic, data ) {      // Log the topic for debugging purposes    console.log( "A new message was received: ", topic );      // Use the data that was passed from our subject    // to display a message preview to the user    $( ".messageSender" ).html( data.sender );    $( ".messagePreview" ).html( data.body );    });    // Here's another subscriber using the same data to perform  // a different task.    // Update the counter displaying the number of new  // messages received via the publisher    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?"  }]);    // We could then at a later point unsubscribe our subscribers  // from receiving any new topic notifications as follows:  // unsubscribe( subscriber1 );  // unsubscribe( subscriber2 ); |

A ideia geral aqui é a promoção de acoplamento fraco. Em vez de objetos únicos chamando os métodos de outros objetos diretamente, eles se inscrevem em uma tarefa ou atividade específica de outro objeto e são notificados quando ela ocorre.

Vantagens

Os padrões Observer e Publicar / Assinar nos incentivam a pensar muito sobre os relacionamentos entre as diferentes partes de nosso aplicativo. Eles também nos ajudam a identificar quais camadas contêm relações diretas que poderiam ser substituídas por conjuntos de sujeitos e observadores. Isso efetivamente poderia ser usado para dividir um aplicativo em blocos menores e mais fracamente acoplados para melhorar o gerenciamento de código e os potenciais de reutilização.

Uma motivação adicional por trás do uso do padrão Observer é onde precisamos manter a consistência entre os objetos relacionados sem fazer classes fortemente acopladas. Por exemplo, quando um objeto precisa ser capaz de notificar outros objetos sem fazer suposições sobre esses objetos.

Podem existir relações dinâmicas entre observadores e assuntos ao usar qualquer um dos padrões. Isso fornece uma grande flexibilidade que pode não ser tão fácil de implementar quando partes distintas de nosso aplicativo estão fortemente acopladas.

Embora nem sempre seja a melhor solução para todos os problemas, esses padrões continuam sendo uma das melhores ferramentas para projetar sistemas desacoplados e devem ser considerados uma ferramenta importante no cinturão de utilitários de qualquer desenvolvedor de JavaScript.

Desvantagens

Conseqüentemente, alguns dos problemas com esses padrões decorrem, na verdade, de seus principais benefícios. Em Publicar / Assinar, ao separar editores de assinantes, às vezes pode se tornar difícil obter garantias de que partes específicas de nossos aplicativos estão funcionando conforme esperamos.

Por exemplo, os editores podem presumir que um ou mais assinantes os estão ouvindo. Digamos que estejamos usando tal suposição para registrar ou gerar erros relacionados a algum processo de aplicativo. Se o assinante que executa o log travar (ou por algum motivo deixar de funcionar), o editor não terá uma maneira de ver isso devido à natureza desacoplada do sistema.

Outra desvantagem do padrão é que os assinantes ignoram completamente a existência uns dos outros e não sabem o custo de mudar de editor. Devido ao relacionamento dinâmico entre assinantes e editores, a dependência de atualização pode ser difícil de rastrear.

Publicar / Assinar Implementações

Publicar / Assinar se encaixa muito bem nos ecossistemas JavaScript, principalmente porque, no núcleo, as implementações ECMAScript são orientadas a eventos. Isso é particularmente verdadeiro em ambientes de navegador, pois o DOM usa eventos como sua API de interação principal para scripts.

Dito isso, nem o ECMAScript nem o DOM fornecem objetos ou métodos básicos para a criação de sistemas de eventos personalizados no código de implementação (com exceção, talvez, do DOM3 CustomEvent, que é vinculado ao DOM e, portanto, não é genericamente útil).

Felizmente, bibliotecas JavaScript populares como dojo, jQuery (eventos personalizados) e YUI já têm utilitários que podem ajudar na implementação fácil de um sistema Publicar / Assinar com muito pouco esforço. Abaixo, podemos ver alguns exemplos disso:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | // Publish    // 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"} );      // Subscribe    // 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 ){...} );      // Unsubscribe    // jQuery: $(obj).off( "channel" );  $( el ).off( "/login" );    // Dojo: dojo.unsubscribe( handle );  dojo.unsubscribe( handle );    // YUI: el.detach("channel");  el.detach( "/login" ); |

Para aqueles que desejam usar o padrão Publicar / Assinar com o vanilla JavaScript (ou outra biblioteca), [AmplifyJS](http://amplifyjs.com/) inclui uma implementação limpa e [independente de](http://amplifyjs.com/) biblioteca que pode ser usada com qualquer biblioteca ou kit de ferramentas. Radio.js ( <http://radio.uxder.com/> ), PubSubJS ( <https://github.com/mroderick/PubSubJS> ) ou Pure JS PubSub de Peter Higgins ( [https://github.com/phiggins42/bloody- jquery-plugins / blob / 55e41df9bf08f42378bb08b93efcb28555b61aeb / pubsub.js](https://github.com/phiggins42/bloody-jquery-plugins/blob/55e41df9bf08f42378bb08b93efcb28555b61aeb/pubsub.js) ) também são alternativas semelhantes que vale a pena conferir.

Os desenvolvedores de jQuery em particular têm algumas outras opções e podem optar por usar uma das muitas implementações bem desenvolvidas, desde o plug-in jQuery de Peter Higgins até o pub / Sub jQuery gist (otimizado) de Ben Alman no GitHub. Links para apenas alguns deles podem ser encontrados abaixo.

* Gist Pub / Sub de Ben Alman <https://gist.github.com/661855> (recomendado)
* O estilo jQuery-core de Rick Waldron sobre o <https://gist.github.com/705311> acima
* Plugin de Peter Higgins " [http://github.com/phiggins42/bloody-jquery-plugins/blob/master/pubsub.js](https://github.com/phiggins42/bloody-jquery-plugins/blob/master/pubsub.js) .
* Pub / Sub de AppendTo em AmplifyJS [http://amplifyjs.com](http://amplifyjs.com/)
* A essência de Ben Truyman <https://gist.github.com/826794>

Para que possamos obter uma avaliação de quantas das implementações de JavaScript vanilla do padrão Observer podem funcionar, vamos dar uma [olhada em](https://github.com/addyosmani/pubsubz) uma versão minimalista de Publicar / Assinar que [lancei](https://github.com/addyosmani/pubsubz) no GitHub em um projeto chamado [pubsubz](https://github.com/addyosmani/pubsubz) . Isso demonstra os principais conceitos de assinatura e publicação, bem como o conceito de cancelamento de assinatura.

Optei por basear nossos exemplos neste código, pois ele se apega às assinaturas do método e à abordagem de implementação que eu esperaria ver em uma versão JavaScript do padrão clássico Observer.

Uma implementação de publicação / assinatura

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65 | var pubsub = {};    (function(myObject) {        // Storage for topics that can be broadcast      // or listened to      var topics = {};        // A topic identifier      var subUid = -1;        // Publish or broadcast events of interest      // with a specific topic name and arguments      // such as the data to pass along      myObject.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      myObject.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      myObject.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 )); |

Exemplo: usando nossa implementação

Agora podemos usar a implementação para publicar e assinar eventos de interesse da seguinte maneira:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | // 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 can 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?" ); |

Exemplo: notificações da interface do usuário

A seguir, vamos imaginar que temos um aplicativo da web responsável por exibir informações do estoque em tempo real.

O aplicativo pode ter uma grade para exibir as estatísticas do estoque e um contador para exibir o último ponto de atualização. Quando o modelo de dados muda, o aplicativo precisará atualizar a grade e o contador. Nesse cenário, nosso assunto (que publicará tópicos / notificações) é o modelo de dados e nossos assinantes são a grade e o contador.

Quando nossos assinantes recebem uma notificação de que o próprio modelo mudou, eles podem se atualizar de acordo.

Em nossa implementação, nosso assinante ouvirá o tópico "newDataAvailable" para descobrir se há novas informações de estoque disponíveis. Se uma nova notificação for publicada neste tópico, será acionada gridUpdatepara adicionar uma nova linha à nossa grade contendo essas informações. Ele também atualizará um contador *atualizado* pela última vez para registrar a última vez que os dados foram adicionados

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58 | // 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  }); |

Exemplo: desacoplamento de aplicativos usando a implementação Pub / Sub de Ben Alman

No seguinte exemplo de classificação de filmes, estaremos usando a implementação jQuery de Ben Alman de Publicar / Assinar para demonstrar como podemos desacoplar uma interface de usuário. Observe como o envio de uma classificação só tem o efeito de publicar o fato de que novos usuários e dados de classificação estão disponíveis.

É deixado para os assinantes desses tópicos delegar o que acontece com esses dados. Em nosso caso, estamos empurrando esses novos dados em matrizes existentes e, em seguida, renderizando-os usando o .template()método da biblioteca Underscore para modelagem.

**HTML / Templates**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | <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**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | ;(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 ){        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 ); |

Exemplo: desacoplamento de um aplicativo jQuery baseado em Ajax

Em nosso exemplo final, vamos dar uma olhada prática em como desacoplar nosso código usando o Pub / Sub no início do processo de desenvolvimento pode nos poupar algumas refatorações potencialmente dolorosas mais tarde.

Muitas vezes, em aplicativos pesados ​​Ajax, uma vez que recebemos uma resposta a uma solicitação, queremos realizar mais do que apenas uma ação exclusiva. Pode-se simplesmente adicionar toda a lógica pós-solicitação em um retorno de chamada bem-sucedido, mas há desvantagens nessa abordagem.

Aplicativos altamente acoplados às vezes aumentam o esforço necessário para reutilizar a funcionalidade devido à maior dependência entre funções / código. O que isso significa é que, embora manter nossa lógica pós-solicitação codificada em um retorno de chamada possa ser bom se estivermos apenas tentando obter um conjunto de resultados uma vez, não é apropriado quando queremos fazer chamadas Ajax adicionais para a mesma fonte de dados (e comportamento final diferente) sem reescrever partes do código várias vezes. Em vez de ter que voltar por cada camada que chama a mesma fonte de dados e generalizá-los mais tarde, podemos usar pub / sub desde o início e economizar tempo.

Usando Observadores, também podemos separar facilmente notificações de todo o aplicativo sobre diferentes eventos para qualquer nível de granularidade com o qual estamos confortáveis ​​- algo que pode ser feito de forma menos elegante usando outros padrões.

Observe como, em nosso exemplo abaixo, uma notificação de tópico é feita quando um usuário indica que deseja fazer uma consulta de pesquisa e outra é feita quando a solicitação retorna e os dados reais estão disponíveis para consumo. É deixado para os assinantes decidirem como usar o conhecimento desses eventos (ou os dados retornados). Os benefícios disso são que, se quiséssemos, poderíamos ter 10 assinantes diferentes utilizando os dados retornados de maneiras diferentes, mas no que diz respeito à camada Ajax, isso não importa. Sua única função é solicitar e devolver os dados e, em seguida, repassá-los a quem quiser utilizá-los. Essa separação de interesses pode tornar o design geral de nosso código um pouco mais limpo.

**HTML / modelos** :

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | <form id="flickrSearch">       <input type="text" name="tag" id="query"/>       <input type="submit" name="submit" value="submit"/>    </form>        <div id="lastQuery"></div>    <ol id="searchResults"></ol>        <script id="resultTemplate" type="text/html">      <% \_.each(items, function( item ){ %>          <li><img src="<%= item.media.m %>"/></li>      <% });%>  </script> |

**JavaScript** :

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59 | ;(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( e, tags ) {         $( "#lastQuery" )                  .html("<p>Searched for:<strong>" + tags + "</strong></p>");     });       // Subscribe to the new results topic     $.subscribe( "/search/resultSet", function( e, results ){           $( "#searchResults" ).empty().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( e, 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", { items: data.items } );         });       });      })( jQuery ); |

O padrão Observer é útil para desacoplar uma série de cenários diferentes no design do aplicativo e, se você não o estiver usando, recomendo pegar uma das implementações pré-escritas mencionadas hoje e apenas testá-la. É um dos padrões de design mais fáceis de começar, mas também um dos mais poderosos.

O Padrão Mediador

Na seção sobre o padrão Observer, fomos apresentados a uma maneira de canalizar várias fontes de eventos por meio de um único objeto. Isso também é conhecido como Publicar / Assinar ou Agregação de evento. É comum que os desenvolvedores pensem em Mediadores quando confrontados com esse problema, então vamos explorar como eles diferem.

O dicionário se refere a um mediador como *uma parte neutra que auxilia nas negociações e resolução de conflitos* . Em nosso mundo, um mediador é um padrão de design comportamental que nos permite expor uma interface unificada por meio da qual as diferentes partes de um sistema podem se comunicar.

Se parecer que um sistema tem muitos relacionamentos diretos entre os componentes, pode ser hora de ter um ponto central de controle através do qual os componentes se comunicam. O Mediador promove o acoplamento fraco, garantindo que, em vez de componentes referindo-se um ao outro explicitamente, sua interação seja tratada por meio desse ponto central. Isso pode nos ajudar a desacoplar sistemas e melhorar o potencial de reutilização de componentes.

Uma analogia do mundo real poderia ser um sistema de controle de tráfego de aeroporto típico. Uma torre (mediador) controla quais aviões podem decolar e pousar porque todas as comunicações (notificações sendo ouvidas ou transmitidas) são feitas dos aviões para a torre de controle, ao invés de um plano para outro. Um controlador centralizado é a chave para o sucesso deste sistema e esse é realmente o papel que um mediador desempenha no design de software.

Outra analogia seria o bubbling de eventos DOM e a delegação de eventos. Se todas as assinaturas em um sistema forem feitas no documento em vez de em nós individuais, o documento servirá efetivamente como um mediador. Em vez de se vincular aos eventos dos nós individuais, um objeto de nível superior recebe a responsabilidade de notificar os assinantes sobre os eventos de interação.

Quando se trata dos padrões Mediator e Event Aggregator, há momentos em que pode parecer que os padrões são intercambiáveis ​​devido a semelhanças de implementação. No entanto, a semântica e a intenção desses padrões são muito diferentes.

E mesmo que as implementações usem algumas das mesmas construções principais, acredito que haja uma diferença distinta entre elas. Eu também acredito que eles não devem ser trocados ou confundidos na comunicação por causa das diferenças.

**Um mediador simples**

Um mediador é um objeto que coordena as interações (lógica e comportamento) entre vários objetos. Ele toma decisões sobre quando chamar quais objetos, com base nas ações (ou inação) de outros objetos e na entrada.

Você pode escrever um mediador usando uma única linha de código:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1 | var mediator = {}; |

Sim, é claro que este é apenas um objeto literal em JavaScript. Mais uma vez, estamos falando sobre semântica aqui. O objetivo do mediador é controlar o fluxo de trabalho entre os objetos e realmente não precisamos de nada mais do que um literal de objeto para fazer isso.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | var orgChart = {      addNewEmployee: function(){        // getEmployeeDetail provides a view that users interact with      var employeeDetail = this.getEmployeeDetail();        // when the employee detail is complete, the mediator (the 'orgchart' object)      // decides what should happen next      employeeDetail.on("complete", function(employee){          // set up additional objects that have additional events, which are used        // by the mediator to do additional things        var managerSelector = this.selectManager(employee);        managerSelector.on("save", function(employee){          employee.save();        });        });    },      // ...  } |

Este exemplo mostra uma implementação muito básica de um objeto mediador com alguns métodos utilitários que podem disparar e assinar eventos.

Muitas vezes me referi a esse tipo de objeto como um objeto de “fluxo de trabalho” no passado, mas a verdade é que é um mediador. É um objeto que controla o fluxo de trabalho entre muitos outros objetos, agregando a responsabilidade desse conhecimento de fluxo de trabalho em um único objeto. O resultado é um fluxo de trabalho mais fácil de entender e manter.

**Semelhanças e diferenças**

Existem, sem dúvida, semelhanças entre os exemplos de agregador de eventos e mediador que mostrei aqui. As semelhanças se resumem a dois itens principais: eventos e objetos de terceiros. Essas diferenças são, na melhor das hipóteses, superficiais. Quando investigamos a intenção do padrão e vemos que as implementações podem ser dramaticamente diferentes, a natureza dos padrões se torna mais aparente.

**Eventos**

Tanto o agregador de eventos quanto o mediador usam eventos, nos exemplos acima. Um agregador de eventos obviamente lida com eventos - afinal, está no nome. O mediador só usa eventos porque torna a vida mais fácil ao lidar com estruturas de webapp JavaScript modernas. Não há nada que diga que um mediador deve ser construído com eventos. Você pode construir um mediador com métodos de retorno de chamada, entregando a referência do mediador ao objeto filho ou por qualquer um de vários outros meios.

A diferença, então, é por que esses dois padrões estão usando eventos. O agregador de eventos, como um padrão, é projetado para lidar com eventos. O mediador, porém, só os usa porque é conveniente.

**Objetos de Terceiros**

Tanto o agregador de eventos quanto o mediador, por design, usam um objeto de terceiros para facilitar as coisas. O próprio agregador de eventos é um terceiro para o editor e assinante do evento. Ele atua como um hub central para a passagem de eventos. O mediador também é um terceiro para outros objetos, no entanto. Então, onde está a diferença? Por que não chamamos um agregador de eventos de mediador? A resposta em grande parte se resume a onde a lógica do aplicativo e o fluxo de trabalho são codificados.

No caso de um agregador de eventos, o objeto de terceiros está lá apenas para facilitar a passagem de eventos de um número desconhecido de fontes para um número desconhecido de manipuladores. Todo fluxo de trabalho e lógica de negócios que precisam ser iniciados são colocados diretamente no objeto que aciona os eventos e nos objetos que tratam dos eventos.

No caso do mediador, porém, a lógica de negócios e o fluxo de trabalho são agregados ao próprio mediador. O mediador decide quando um objeto deve ter seus métodos chamados e atributos atualizados com base em fatores que o mediador conhece. Ele encapsula o fluxo de trabalho e o processo, coordenando vários objetos para produzir o comportamento de sistema desejado. Cada um dos objetos individuais envolvidos neste fluxo de trabalho sabe como executar sua própria tarefa. Mas é o mediador que diz aos objetos quando executar as tarefas, tomando decisões em um nível superior ao dos objetos individuais.

Um agregador de eventos facilita um modelo de comunicação do tipo “dispare e esqueça”. O objeto que dispara o evento não se importa se há algum assinante. Ele apenas dispara o evento e segue em frente. Um mediador, entretanto, pode usar eventos para tomar decisões, mas definitivamente não é “disparar e esquecer”. Um mediador presta atenção a um conjunto conhecido de entradas ou atividades para que possa facilitar e coordenar o comportamento adicional com um conjunto conhecido de atores (objetos).

**Relacionamentos: quando usar quais**

Compreender as semelhanças e diferenças entre um agregador de eventos e um mediador é importante por razões semânticas. No entanto, é igualmente importante entender quando usar qual padrão. A semântica básica e a intenção dos padrões informam a questão de quando, mas a experiência real no uso dos padrões o ajudará a entender os pontos mais sutis e as decisões diferenciadas que precisam ser tomadas.

**Uso de agregador de evento**

Em geral, um agregador de eventos é usado quando você tem muitos objetos para ouvir diretamente ou objetos que não estão relacionados.

Quando dois objetos já têm um relacionamento direto - digamos, uma visualização pai e uma visualização filho - pode haver benefícios em usar um agregador de eventos. Faça com que a visão filha acione um evento e a visão pai possa lidar com o evento. Em termos de estrutura JavaScript, isso é mais comumente visto na Coleção e no Modelo do Backbone, onde todos os eventos do Modelo são enviados para e por meio da Coleção pai. Uma coleção geralmente usa eventos de modelo para modificar o estado de si mesma ou de outros modelos. O manuseio de itens “selecionados” em uma coleção é um bom exemplo disso.

O método on da jQuery como um agregador de eventos é um ótimo exemplo de muitos objetos para ouvir. Se você tiver 10, 20 ou 200 elementos DOM que podem acionar um evento de “clique”, pode ser uma má ideia configurar um ouvinte em todos eles individualmente. Isso pode deteriorar rapidamente o desempenho do aplicativo e a experiência do usuário. Em vez disso, usar o método on do jQuery nos permite agregar todos os eventos e reduzir a sobrecarga de 10, 20 ou 200 manipuladores de eventos para 1.

Os relacionamentos indiretos também são um ótimo momento para usar agregadores de eventos. Em aplicativos modernos, é muito comum ter vários objetos de visualização que precisam se comunicar, mas não têm relacionamento direto. Por exemplo, um sistema de menu pode ter uma visão que lida com os cliques do item de menu. Mas não queremos que o menu seja diretamente vinculado às visualizações de conteúdo que mostram todos os detalhes e informações quando um item de menu é clicado. Ter o conteúdo e o menu acoplados tornaria o código muito difícil de manter, a longo prazo. Em vez disso, podemos usar um agregador de eventos para acionar os eventos “menu: click: foo” e ter um objeto “foo” que manipule o evento click para mostrar seu conteúdo na tela.

**Uso do mediador**

Um mediador é mais bem aplicado quando dois ou mais objetos têm uma relação de trabalho indireta e a lógica de negócios ou fluxo de trabalho precisa ditar as interações e coordenação desses objetos.

Uma interface de assistente é um bom exemplo disso, conforme mostrado com o exemplo “orgChart” acima. Existem várias visualizações que facilitam todo o fluxo de trabalho do assistente. Em vez de acoplar fortemente a visualização, fazendo-os referenciar um ao outro diretamente, podemos separá-los e modelar mais explicitamente o fluxo de trabalho entre eles, introduzindo um mediador.

O mediador extrai o fluxo de trabalho dos detalhes de implementação e cria uma abstração mais natural em um nível superior, mostrando-nos rapidamente o que é esse fluxo de trabalho. Não precisamos mais nos aprofundar nos detalhes de cada visualização no fluxo de trabalho para ver o que o fluxo de trabalho realmente é.

**Agregador de eventos (Pub / Sub) e mediador juntos**

O ponto crucial da diferença entre um agregador de evento e um mediador, e por que esses nomes de padrão não devem ser trocados entre si, é ilustrado melhor mostrando como eles podem ser usados ​​juntos. O exemplo de menu para um agregador de eventos é o lugar perfeito para apresentar também um mediador.

Clicar em um item de menu pode disparar uma série de alterações em todo o aplicativo. Algumas dessas mudanças serão independentes de outras, e o uso de um agregador de eventos para isso faz sentido. No entanto, algumas dessas mudanças podem estar relacionadas internamente entre si e podem usar um mediador para fazer essas mudanças.

Um mediador, então, pode ser configurado para ouvir o agregador de eventos. Ele poderia executar sua lógica e processo para facilitar e coordenar muitos objetos que estão relacionados entre si, mas não relacionados à fonte de evento original.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | var MenuItem = MyFrameworkView.extend({      events: {      "click .thatThing": "clickedIt"    },      clickedIt: function(e){      e.preventDefault();        // assume this triggers "menu:click:foo"      MyFramework.trigger("menu:click:" + this.model.get("name"));    }    });    // ... somewhere else in the app    var MyWorkflow = function(){    MyFramework.on("menu:click:foo", this.doStuff, this);  };    MyWorkflow.prototype.doStuff = function(){    // instantiate multiple objects here.    // set up event handlers for those objects.    // coordinate all of the objects into a meaningful workflow.  }; |

Neste exemplo, quando o MenuItem com o modelo correto for clicado, o “menu:click:foo”evento será disparado. Uma instância do objeto “MyWorkflow”, supondo que um já esteja instanciado, tratará esse evento específico e coordenará todos os objetos que conhece, para criar a experiência do usuário e o fluxo de trabalho desejados.

Um agregador de eventos e um mediador foram combinados para criar uma experiência muito mais significativa no código e no próprio aplicativo. Agora temos uma separação clara entre o menu e o fluxo de trabalho por meio de um agregador de eventos e ainda mantemos o próprio fluxo de trabalho limpo e sustentável por meio do uso de um mediador.

Vantagens desvantagens

O maior benefício do padrão Mediator é que ele reduz os canais de comunicação necessários entre objetos ou componentes em um sistema de muitos para muitos para apenas muitos para um. Adicionar novos editores e assinantes é relativamente fácil devido ao nível de desacoplamento presente.

Talvez a maior desvantagem de usar o padrão seja que ele pode apresentar um único ponto de falha. Colocar um mediador entre os módulos também pode causar um impacto no desempenho, pois eles estão sempre se comunicando indiretamente. Devido à natureza do acoplamento fraco, é difícil estabelecer como um sistema pode reagir olhando apenas para as transmissões.

Dito isso, é útil nos lembrarmos de que os sistemas desacoplados têm uma série de outros benefícios - se nossos módulos se comunicassem diretamente uns com os outros, as alterações nos módulos (por exemplo, outro módulo gerando uma exceção) poderiam facilmente ter um efeito dominó no resto do nosso aplicativo . Este problema é menos preocupante com sistemas desacoplados.

No final do dia, o acoplamento forte causa todos os tipos de dores de cabeça e esta é apenas outra solução alternativa, mas que pode funcionar muito bem se implementada corretamente.

Mediador vs. Fachada

Estaremos cobrindo o padrão Façade em breve, mas para fins de referência, alguns desenvolvedores também podem se perguntar se há semelhanças entre os padrões Mediator e Façade. Ambos abstraem a funcionalidade dos módulos existentes, mas existem algumas diferenças sutis.

O Mediator centraliza a comunicação entre os módulos onde é explicitamente referenciado por esses módulos. Em certo sentido, isso é multidirecional. A fachada, entretanto, apenas define uma interface mais simples para um módulo ou sistema, mas não adiciona nenhuma funcionalidade adicional. Outros módulos do sistema não estão diretamente cientes do conceito de fachada e podem ser considerados unidirecionais.

O Padrão de Protótipo

O GoF se refere ao padrão de protótipo como aquele que cria objetos com base em um modelo de um objeto existente por meio de clonagem.

Podemos pensar no padrão de protótipo como sendo baseado na herança prototípica, onde criamos objetos que atuam como protótipos para outros objetos. O próprio objeto de protótipo é efetivamente usado como um projeto para cada objeto que o construtor cria. Se o protótipo da função construtora usada contiver uma propriedade chamada namepor exemplo (conforme o exemplo de código mais abaixo), então cada objeto criado por esse mesmo construtor também terá essa mesma propriedade.

Rever as definições para esse padrão na literatura existente (não-JavaScript), que **podem** encontrar referências a classes mais uma vez. A realidade é que a herança prototípica evita o uso de classes. Não existe um objeto de "definição" nem um objeto central em teoria. Estamos simplesmente criando cópias de objetos funcionais existentes.

Um dos benefícios de usar o padrão de protótipo é que estamos trabalhando com os pontos fortes do protótipo que o JavaScript tem a oferecer nativamente, em vez de tentar imitar recursos de outras linguagens. Com outros padrões de design, nem sempre é esse o caso.

O padrão não é apenas uma maneira fácil de implementar herança, mas também pode vir com um aumento de desempenho: ao definir uma função em um objeto, eles são todos criados por referência (portanto, todos os objetos filhos apontam para a mesma função) em vez de criar suas próprias cópias individuais.

Para os interessados, a herança prototípica real, conforme definida no padrão ECMAScript 5, requer o uso de Object.create(que examinamos anteriormente nesta seção). Para nos lembrar, Object.createcria um objeto que possui um protótipo especificado e, opcionalmente, contém propriedades especificadas também (por exemplo Object.create( prototype, optionalDescriptorObjects )).

Podemos ver isso demonstrado no exemplo abaixo:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | 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.createtambém nos permite implementar facilmente conceitos avançados, como herança diferencial, em que os objetos podem herdar diretamente de outros objetos. Vimos anteriormente que Object.createnos permite inicializar as propriedades do objeto usando o segundo argumento fornecido. Por exemplo:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | 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    }    }); |

Aqui, as propriedades podem ser inicializadas no segundo argumento de Object.createusar um literal de objeto com uma sintaxe semelhante à usada pelos métodos Object.definePropertiese Object.definePropertyque examinamos anteriormente.

É importante notar que os relacionamentos prototípicos podem causar problemas ao enumerar propriedades de objetos e (como Crockford recomenda) envolver o conteúdo do loop em uma hasOwnProperty()verificação.

Se quisermos implementar o padrão de protótipo sem usar diretamente Object.create, podemos simular o padrão de acordo com o exemplo acima da seguinte maneira:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | 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(); |

**Nota:** Esta alternativa não permite que o usuário defina propriedades somente leitura da mesma maneira (já que o VehiclePrototype pode ser alterado se não for cuidadoso).

Uma implementação alternativa final do padrão de protótipo pode ser a seguinte:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | var beget = (function () {        function F() {}        return function ( proto ) {          F.prototype = proto;          return new F();      };  })(); |

Pode-se fazer referência a esse método a partir da vehiclefunção. Observe, entretanto, que vehicleaqui está emulando um construtor, uma vez que o padrão de protótipo não inclui nenhuma noção de inicialização além de vincular um objeto a um protótipo.

O Padrão de Comando

O padrão Command visa encapsular a invocação de método, solicitações ou operações em um único objeto e nos dá a capacidade de parametrizar e passar chamadas de método que podem ser executadas a nosso critério. Além disso, ele nos permite desacoplar objetos que invocam a ação dos objetos que os implementam, dando-nos um maior grau de flexibilidade geral na troca de *classes* concretas (objetos).

*As* classes *concretas* são melhor explicadas em termos de linguagens de programação baseadas em classes e estão relacionadas à ideia de classes abstratas. Uma classe *abstrata* define uma interface, mas não necessariamente fornece implementações para todas as suas funções-membro. Ele atua como uma classe base da qual outras são derivadas. Uma classe derivada que implementa a funcionalidade ausente é chamada de classe *concreta* .

A ideia geral por trás do padrão Command é que ele nos fornece um meio de separar as responsabilidades de emitir comandos de qualquer coisa que os execute, delegando essa responsabilidade a diferentes objetos.

Em termos de implementação, os objetos de comando simples unem uma ação e o objeto que deseja invocar a ação. Eles consistentemente incluem uma operação de execução (como run()ou execute()). Todos os objetos Command com a mesma interface podem ser facilmente trocados conforme necessário e isso é considerado um dos maiores benefícios do padrão.

Para demonstrar o padrão Command, vamos criar um serviço simples de compra de automóveis.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | (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 + " ) ";      }      };    })(); |

Dando uma olhada no código acima, seria trivial invocar nossos carManagermétodos acessando diretamente o objeto. Todos nós seríamos perdoados por pensar que não há nada de errado com isso - tecnicamente, é um JavaScript totalmente válido. No entanto, existem cenários em que isso pode ser desvantajoso.

Por exemplo, imagine se a API principal por trás da carManagermudança. Isso exigiria que todos os objetos que acessam diretamente esses métodos em nosso aplicativo também sejam modificados. Isso pode ser visto como uma camada de acoplamento que efetivamente vai contra a metodologia OOP de objetos fracamente acoplados, tanto quanto possível. Em vez disso, poderíamos resolver esse problema abstraindo ainda mais a API.

Vamos agora expandir nosso carManagerpara que nossa aplicação do padrão Command resulte no seguinte: aceite quaisquer métodos nomeados que possam ser executados no carManagerobjeto, passando adiante quaisquer dados que possam ser usados, como o modelo do carro e ID.

Aqui está o que gostaríamos de conseguir:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1 | carManager.execute( "buyVehicle", "Ford Escort", "453543" ); |

De acordo com esta estrutura, devemos agora adicionar uma definição para o carManager.executemétodo da seguinte forma:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3 | carManager.execute = function ( name ) {      return carManager[name] && carManager[name].apply( carManager, [].slice.call(arguments, 1) );  }; |

Nossos exemplos finais de chamadas seriam assim:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4 | carManager.execute( "arrangeViewing", "Ferrari", "14523" );  carManager.execute( "requestInfo", "Ford Mondeo", "54323" );  carManager.execute( "requestInfo", "Ford Escort", "34232" );  carManager.execute( "buyVehicle", "Ford Escort", "34232" ); |

O Padrão da Fachada

Quando colocamos uma fachada, apresentamos ao mundo uma aparência externa que pode ocultar uma realidade muito diferente. Essa foi a inspiração para o nome por trás do próximo padrão que iremos revisar - o padrão Facade. Esse padrão fornece uma interface conveniente de nível superior para um corpo maior de código, ocultando sua verdadeira complexidade subjacente. Pense nisso como uma simplificação da API apresentada a outros desenvolvedores, algo que quase sempre melhora a usabilidade.

Fachadas são um padrão estrutural que muitas vezes pode ser visto em bibliotecas JavaScript como jQuery onde, embora uma implementação possa suportar métodos com uma ampla gama de comportamentos, apenas uma "fachada" ou abstração limitada desses métodos é apresentada ao público para uso.

Isso nos permite interagir com a fachada diretamente, em vez de com o subsistema nos bastidores. Sempre que usamos métodos $(el).css()ou jQuery $(el).animate(), estamos na verdade usando uma fachada - a interface pública mais simples que nos evita ter que chamar manualmente os muitos métodos internos no núcleo do jQuery necessários para fazer algum comportamento funcionar. Isso também evita a necessidade de interagir manualmente com APIs DOM e manter variáveis ​​de estado.

Os métodos básicos do jQuery devem ser considerados abstrações intermediárias. A carga mais imediata para os desenvolvedores é a API DOM e as fachadas são o que tornam a biblioteca jQuery tão fácil de usar.

Para desenvolver o que aprendemos, o padrão Facade simplifica a interface de uma classe e também desacopla a classe do código que a utiliza. Isso nos dá a capacidade de interagir indiretamente com os subsistemas de uma maneira que às vezes pode ser menos sujeita a erros do que acessar o subsistema diretamente. As vantagens de uma fachada incluem facilidade de uso e geralmente um tamanho pequeno na implementação do padrão.

Vamos dar uma olhada no padrão em ação. Este é um exemplo de código não otimizado, mas aqui estamos utilizando um Façade para simplificar uma interface para ouvir eventos em vários navegadores. Fazemos isso criando um método comum que pode ser usado no código de alguém que faz a tarefa de verificar a existência de recursos para que possa fornecer uma solução segura e compatível com vários navegadores.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | 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;      }    }; |

De maneira semelhante, estamos todos familiarizados com o jQuery $(document).ready(..). Internamente, isso está sendo alimentado por um método chamado bindReady(), que faz o seguinte:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | 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 );                 ... |

Este é outro exemplo de uma fachada, onde o resto do mundo simplesmente usa a interface limitada exposta $(document).ready(..)e a implementação mais complexa que a alimenta é mantida oculta.

No entanto, as fachadas não precisam ser usadas sozinhas. Eles também podem ser integrados a outros padrões, como o padrão de Módulo. Como podemos ver abaixo, nossa instância dos padrões de módulo contém uma série de métodos que foram definidos de forma privada. Uma fachada é então usada para fornecer uma API muito mais simples para acessar esses métodos:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | 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} ); |

Neste exemplo, a chamada module.facade()irá realmente disparar um conjunto de comportamentos privados dentro do módulo, mas, novamente, o usuário não está preocupado com isso. tornamos muito mais fácil para eles consumir um recurso sem a necessidade de se preocupar com detalhes no nível de implementação.

Notas sobre abstração

As fachadas geralmente têm poucas desvantagens, mas uma preocupação que vale a pena observar é o desempenho. Ou seja, deve-se determinar se há um custo implícito para a abstração que um Façade oferece para nossa implementação e, se houver, se esse custo é justificável. Voltando à biblioteca jQuery, a maioria de nós está ciente de que ambos getElementById("identifier")e $("#identifier")podem ser usados ​​para consultar um elemento em uma página por seu ID.

Você sabia, entretanto, que getElementById()por si só é significativamente mais rápido em uma alta ordem de magnitude? Dê uma olhada neste teste jsPerf para ver os resultados em um nível por navegador: [http://jsperf.com/getelementbyid-vs-jquery-id](https://jsperf.com/getelementbyid-vs-jquery-id) . Agora, é claro, temos que ter em mente que o jQuery (e o Sizzle - seu mecanismo seletor) estão fazendo muito mais nos bastidores para otimizar nossa consulta (e que um objeto jQuery, não apenas um nó DOM, é retornado).

O desafio com esta fachada em particular é que, para fornecer uma função de seletor elegante capaz de aceitar e analisar vários tipos de consultas, há um custo implícito de abstração. O usuário não é obrigado a acessar jQuery.getById("identifier")ou jQuery.getByClass("identifier")e assim por diante. Dito isso, a compensação de desempenho foi testada na prática ao longo dos anos e, dado o sucesso do jQuery, uma fachada simples funcionou muito bem para a equipe.

Ao usar o padrão, tente estar ciente dos custos de desempenho envolvidos e verifique se eles valem o nível de abstração oferecido.

O Padrão de Fábrica

O padrão de fábrica é outro padrão de criação preocupado com a noção de criação de objetos. O que difere dos outros padrões em sua categoria é que não exige explicitamente que usemos um construtor. Em vez disso, um Factory pode fornecer uma interface genérica para a criação de objetos, onde podemos especificar o tipo de objeto de fábrica que desejamos criar.

Imagine que temos uma fábrica de UI onde somos solicitados a criar um tipo de componente de UI. Em vez de criar esse componente diretamente usando o newoperador ou por meio de outro construtor criativo, pedimos a um objeto Factory um novo componente. Informamos à Fábrica o tipo de objeto necessário (ex: "Botão", "Painel") e ela instancia, devolvendo-o para uso.

Isso é particularmente útil se o processo de criação do objeto for relativamente complexo, por exemplo, se depender fortemente de fatores dinâmicos ou da configuração do aplicativo.

Exemplos desse padrão podem ser encontrados em bibliotecas de IU, como ExtJS, onde os métodos para criar objetos ou componentes podem ter subclasses adicionais.

A seguir está um exemplo que se baseia em nossos snippets anteriores usando a lógica do padrão Construtor para definir carros. Ele demonstra como uma *fábrica de* veículos pode ser implementada usando o padrão de fábrica:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62 | // 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 ) {      switch(options.vehicleType){      case "car":        this.vehicleClass = Car;        break;      case "truck":        this.vehicleClass = Truck;        break;      //defaults to VehicleFactory.prototype.vehicleClass (Car)    }      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 ); |

**Abordagem nº 1: modificar uma instância VehicleFactory para usar a classe Truck**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | 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 ); |

**Abordagem # 2: Subclasse VehicleFactory para criar uma classe de fábrica que constrói caminhões**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | 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 ); |

Quando usar o padrão de fábrica

O padrão de fábrica pode ser especialmente útil quando aplicado às seguintes situações:

* Quando nosso objeto ou configuração de componente envolve um alto nível de complexidade
* Quando precisamos gerar facilmente diferentes instâncias de objetos, dependendo do ambiente em que estamos
* Quando estamos trabalhando com muitos pequenos objetos ou componentes que compartilham as mesmas propriedades
* Ao compor objetos com instâncias de outros objetos que precisam apenas satisfazer um contrato de API (também conhecido como, digitação de pato) para funcionar. Isso é útil para desacoplamento.

Quando não usar o padrão de fábrica

Quando aplicado ao tipo errado de problema, esse padrão pode introduzir uma complexidade desnecessariamente grande em um aplicativo. A menos que fornecer uma interface para a criação de objetos seja um objetivo de design para a biblioteca ou estrutura que estamos escrevendo, eu sugeriria que nos limitemos a construtores explícitos para evitar sobrecarga desnecessária.

Devido ao fato de que o processo de criação de objeto é efetivamente abstraído por trás de uma interface, isso também pode introduzir problemas com o teste de unidade, dependendo de quão complexo esse processo pode ser.

Fábricas Abstratas

Também é útil estar ciente do padrão **Abstract Factory** , que visa encapsular um grupo de fábricas individuais com um objetivo comum. Ele separa os detalhes de implementação de um conjunto de objetos de seu uso geral.

Um Abstract Factory deve ser usado onde um sistema deve ser independente da maneira como os objetos que ele cria são gerados ou precisa trabalhar com vários tipos de objetos.

Um exemplo simples e fácil de entender é uma fábrica de veículos, que define formas de obter ou registrar tipos de veículos. A fábrica abstrata pode ser denominada abstractVehicleFactory. A fábrica de abstratos permitirá a definição de tipos de veículos como "carro" ou "caminhão" e as fábricas de concreto irão implementar apenas classes que cumpram o contrato do veículo (ex. Vehicle.prototype.driveE Vehicle.prototype.breakDown).

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | 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" } ); |

O padrão Mixin

Em linguagens de programação tradicionais, como C ++ e Lisp, Mixins são classes que oferecem funcionalidades que podem ser facilmente herdadas por uma subclasse ou grupo de subclasses para fins de reutilização de função.

Subclassificação

Para os desenvolvedores não familiarizados com a subclasse, vamos passar por uma breve cartilha para iniciantes antes de nos aprofundarmos em Mixins e Decoradores.

*Subclassificação* é um termo que se refere à herança de propriedades para um novo objeto de um objeto base ou *superclasse* . Na programação tradicional orientada a objetos, uma classe Bé capaz de estender outra classe A. Aqui, consideramos Auma superclasse e Buma subclasse de A. Como tal, todas as instâncias de Bherdam os métodos de A. Bno entanto, ainda é capaz de definir seus próprios métodos, incluindo aqueles que substituem os métodos originalmente definidos por A.

Caso seja Bnecessário invocar um método Aque foi sobrescrito, nos referimos a isso como encadeamento de método. Deve ser Bnecessário invocar o construtor A(a superclasse), chamamos esse construtor de encadeamento.

Para demonstrar a subclasse, primeiro precisamos de um objeto base que possa ter novas instâncias criadas. vamos modelar isso em torno do conceito de pessoa.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var Person = function( firstName, lastName ){      this.firstName = firstName;    this.lastName = lastName;    this.gender = "male";    }; |

A seguir, queremos especificar uma nova classe (objeto) que é uma subclasse do Personobjeto existente . Vamos imaginar que queremos adicionar propriedades distintas para distinguir a Personde a Superheroenquanto herdamos as propriedades da Person"superclasse". Como os super-heróis compartilham muitos traços comuns com pessoas normais (por exemplo, nome, gênero), isso deve ilustrar como a subclasse funciona adequadamente.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | // 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 |

O Superheroconstrutor cria um objeto que desce de Person. Objetos desse tipo têm atributos dos objetos que estão acima dele na cadeia e se tivéssemos definido os valores padrão no Personobjeto, ele Superheroé capaz de substituir quaisquer valores herdados por valores específicos para seu objeto.

Mixins

Em JavaScript, podemos ver a herança de Mixins como um meio de coletar funcionalidade por meio de extensão. Cada novo objeto que definimos tem um protótipo do qual pode herdar outras propriedades. Os protótipos podem herdar de outros protótipos de objeto, mas, ainda mais importante, podem definir propriedades para qualquer número de instâncias de objeto. Podemos aproveitar esse fato para promover a reutilização de funções.

Mixins permitem que os objetos tomem emprestado (ou herdem) a funcionalidade deles com um mínimo de complexidade. Como o padrão funciona bem com protótipos de objeto JavaScripts, ele nos oferece uma maneira bastante flexível de compartilhar funcionalidade não apenas de um Mixin, mas efetivamente de muitos por meio de herança múltipla.

Eles podem ser vistos como objetos com atributos e métodos que podem ser facilmente compartilhados entre vários outros protótipos de objetos. Imagine que definimos um Mixin contendo funções de utilidade em um literal de objeto padrão da seguinte maneira:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | var myMixins = {      moveUp: function(){      console.log( "move up" );    },      moveDown: function(){      console.log( "move down" );    },      stop: function(){      console.log( "stop! in the name of love!" );    }    }; |

Podemos então estender facilmente o protótipo das funções construtoras existentes para incluir esse comportamento usando um auxiliar, como o \_.extend()método Underscore.js :

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | // 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! |

Como podemos ver, isso nos permite "misturar" facilmente o comportamento comum em construtores de objetos de maneira bastante trivial.

No próximo exemplo, temos dois construtores: um Car e um Mixin. O que vamos fazer é aumentar (outra forma de dizer estender) o Carro para que ele possa herdar métodos específicos definidos no Mixin, a saber driveForward()e driveBackward(). Desta vez, não usaremos Underscore.js.

Em vez disso, este exemplo demonstrará como aumentar um construtor para incluir funcionalidade sem a necessidade de duplicar esse processo para cada função de construtor que possamos ter.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87 | // 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.call(receivingClass.prototype, methodName) ) {                  receivingClass.prototype[methodName] = givingClass.prototype[methodName];              }                // Alternatively (check prototype chain as well):              // 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 |

Vantagens desvantagens

Mixins ajudam a diminuir a repetição funcional e aumentar a reutilização de funções em um sistema. Quando um aplicativo provavelmente requer comportamento compartilhado entre instâncias de objeto, podemos facilmente evitar qualquer duplicação mantendo essa funcionalidade compartilhada em um Mixin e, assim, focando na implementação apenas da funcionalidade em nosso sistema que seja verdadeiramente distinta.

Dito isso, as desvantagens dos Mixins são um pouco mais discutíveis. Alguns desenvolvedores acham que injetar funcionalidade em um protótipo de objeto é uma má ideia, pois leva à poluição do protótipo e a um nível de incerteza em relação à origem de nossas funções. Em grandes sistemas, pode muito bem ser o caso.

Eu diria que uma documentação forte pode ajudar a minimizar a quantidade de confusão em relação à origem das funções mistas, mas como com todo padrão, se for tomado cuidado durante a implementação, estaremos bem.

O padrão do decorador

Decoradores são um padrão de design estrutural que visa promover a reutilização de código. Semelhante aos Mixins, eles podem ser considerados outra alternativa viável para a subclassificação de objetos.

Classicamente, os decoradores oferecem a capacidade de adicionar comportamento às classes existentes em um sistema de forma dinâmica. A ideia era que a *decoração em* si não era essencial para a funcionalidade básica da classe, caso contrário, seria embutida na própria *superclasse* .

Eles podem ser usados ​​para modificar sistemas existentes onde desejamos adicionar recursos adicionais aos objetos sem a necessidade de modificar pesadamente o código subjacente que os usa. Uma razão comum pela qual os desenvolvedores os usam é que seus aplicativos podem conter recursos que requerem uma grande quantidade de tipos distintos de objetos. Imagine ter que definir centenas de construtores de objetos diferentes para, digamos, um jogo JavaScript.

Os construtores de objetos podem representar tipos de jogadores distintos, cada um com recursos diferentes. Um *Senhor dos Anéis* jogo poderia exigir construtores para Hobbit, Elf, Orc, Wizard, Mountain Giant, Stone Giante assim por diante, mas não poderia ser facilmente centenas destes. Se, então, consignado em capacidades, imagine ter que criar sub-classes para cada combinação de tipo de capacidade, por exemplo HobbitWithRing, HobbitWithSword, HobbitWithRingAndSworde assim on.This não é muito prático e certamente não é controlável quando fator em um número crescente de diferentes habilidades.

O padrão Decorator não está fortemente ligado a como os objetos são criados, mas ao invés disso, foca no problema de estender sua funcionalidade. Em vez de depender apenas da herança prototípica, trabalhamos com um único objeto base e adicionamos progressivamente objetos decoradores que fornecem recursos adicionais. A ideia é que, em vez de subclassificar, adicionamos (decoramos) propriedades ou métodos a um objeto base para que seja um pouco mais simplificado.

Adicionar novos atributos a objetos em JavaScript é um processo muito simples, portanto, com isso em mente, um decorador muito simplista pode ser implementado da seguinte maneira:

Exemplo 1: Decorando Construtores com Nova Funcionalidade

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | // 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 |

Esse tipo de implementação simplista é funcional, mas realmente não demonstra todos os pontos fortes que os Decoradores têm a oferecer. Para isso, primeiro vamos examinar minha variação do exemplo Coffee de um excelente livro chamado *Head First Design Patterns* de Freeman, Sierra e Bates, que foi modelado em torno de uma compra de Macbook.

Exemplo 2: Decoração de objetos com vários decoradores

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | // 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() ); |

No exemplo acima, nossos decoradores estão substituindo a função de MacBook()objetos da superclasse .cost()para retornar o preço atual do Macbookmais o custo da atualização que está sendo especificada.

É considerado uma decoração como os Macbookmétodos do construtor de objetos originais que não são substituídos (por exemplo screenSize()), bem como quaisquer outras propriedades que possamos definir como uma parte do Macbookpermanecer inalterado e intacto.

Não há realmente uma *interface* definida no exemplo acima e estamos mudando a responsabilidade de garantir que um objeto encontre uma interface ao passar do criador para o receptor.

Decoradores pseudo-clássicos

Agora vamos examinar uma variação do Decorator apresentado pela primeira vez em uma forma JavaScript no *Pro JavaScript Design Patterns* (PJDP) por Dustin Diaz e Ross Harmes.

Ao contrário de alguns dos exemplos anteriores, Diaz e Harmes se concentram mais em como os decoradores são implementados em outras linguagens de programação (como Java ou C ++) usando o conceito de "interface", que definiremos em mais detalhes em breve.

**Nota:** Esta variação particular do padrão Decorator é fornecida para fins de referência. Se achar que é muito complexo, recomendo optar por uma das implementações mais simples abordadas anteriormente.

Interfaces

O PJDP descreve o Decorator como um padrão que é usado para envolver objetos de forma transparente dentro de outros objetos da mesma interface. Uma interface é uma forma de definir os métodos que um objeto **deve** ter; no entanto, ela não especifica diretamente como esses métodos devem ser implementados.

Eles também podem indicar quais parâmetros os métodos usam, mas isso é considerado opcional.

Então, por que usaríamos uma interface em JavaScript? A ideia é que eles se autodocumentem e promovam a reutilização. Em teoria, as interfaces também tornam o código mais estável, garantindo que as alterações nelas também sejam feitas nos objetos que as implementam.

Abaixo está um exemplo de uma implementação de interfaces em JavaScript usando duck-typing - uma abordagem que ajuda a determinar se um objeto é uma instância de construtor / objeto com base nos métodos que implementa.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | // 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 = new 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 |

Acima, Interface.ensureImplementsfornece verificação de funcionalidade estrita e o código para este e o Interfaceconstrutor pode ser encontrado [aqui](https://gist.github.com/1057989) .

O maior problema com as interfaces é que, como não há suporte integrado para elas em JavaScript, há o perigo de tentarmos emular um recurso de outra linguagem que pode não ser o ajuste ideal. Interfaces leves podem ser usadas sem um grande custo de desempenho, no entanto, veremos a seguir Decoradores *abstratos* usando este mesmo conceito.

Decoradores abstratos

Para demonstrar a estrutura desta versão do padrão Decorator, vamos imaginar que temos uma superclasse que modela uma Macbookvez mais e uma loja que nos permite "decorar" nosso Macbook com uma série de melhorias por uma taxa adicional.

As melhorias podem incluir atualizações para 4GB ou 8GB Ram, gravura, Parallels ou um case. Agora, se tivéssemos que modelar isso usando uma subclasse individual para cada combinação de opções de melhoria, poderia ser algo assim:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | var Macbook = function(){          //...  };    var  MacbookWith4GBRam = function(){},       MacbookWith8GBRam = function(){},       MacbookWith4GBRamAndEngraving = function(){},       MacbookWith8GBRamAndEngraving = function(){},       MacbookWith8GBRamAndParallels = function(){},       MacbookWith4GBRamAndParallels = function(){},       MacbookWith8GBRamAndParallelsAndCase = function(){},       MacbookWith4GBRamAndParallelsAndCase = function(){},       MacbookWith8GBRamAndParallelsAndCaseAndInsurance = function(){},       MacbookWith4GBRamAndParallelsAndCaseAndInsurance = function(){}; |

e assim por diante.

Essa seria uma solução impraticável, pois uma nova subclasse seria necessária para cada combinação possível de aprimoramentos que estão disponíveis. Como preferiríamos manter as coisas simples sem manter um grande conjunto de subclasses, vamos ver como os decoradores podem ser usados ​​para resolver melhor esse problema.

Em vez de exigir todas as combinações que vimos anteriormente, deveríamos simplesmente ter que criar cinco novas classes de decorador. Os métodos que são chamados nessas classes de aprimoramento seriam passados ​​para nossa Macbookclasse.

Em nosso próximo exemplo, os decoradores envolvem seus componentes de maneira transparente e podem ser trocados de maneira interessante, pois usam a mesma interface.

Esta é a interface que vamos definir para o Macbook:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | 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;      }  }; |

Para tornar mais fácil para nós adicionarmos quantas opções forem necessárias mais tarde, uma classe Abstract Decorator é definida com métodos padrão necessários para implementar a Macbookinterface, que o resto das opções irão subclassificar. Decoradores abstratos garantem que possamos decorar uma classe base independentemente com quantos decoradores forem necessários em diferentes combinações (lembra do exemplo anterior?), Sem precisar derivar uma classe para cada combinação possível.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | // 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();      }  }; |

O que está acontecendo no exemplo acima é que o MacbookDecorator aceita um objeto (um Macbook) para usar como nosso componente base. Ele está usando a Macbookinterface que definimos anteriormente e para cada método está apenas chamando o mesmo método no componente. Agora podemos criar nossas classes de opções para o que pode ser adicionado, apenas usando o MacbookDecorator.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | // First, define a way to extend an object a  // with the properties in object b. We'll use  // this shortly!  function extend( a, b ){      for( var key in b )          if( b.hasOwnProperty(key) )              a[key] = b[key];      return a;  }    var CaseDecorator = function( macbook ){     this.macbook = macbook;  };    // Let's now extend (decorate) the CaseDecorator  // with a MacbookDecorator  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;  }; |

O que estamos fazendo aqui é substituir os métodos addCase()e getPrice()que precisam ser decorados e estamos conseguindo isso chamando primeiro esses métodos no original macbooke, em seguida, simplesmente anexando uma string ou valor numérico (por exemplo, 45,00) a eles de acordo.

Como muitas informações foram apresentadas nesta seção até agora, vamos tentar reunir tudo em um único exemplo que, com sorte, destacará o que aprendemos.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | // Instantiation of the macbook  var myMacbookPro = new MacbookPro();    // Outputs: 900.00  console.log( myMacbookPro.getPrice() );    // Decorate the macbook  var decoratedMacbookPro = new CaseDecorator( myMacbookPro );    // This will return 945.00  console.log( decoratedMacbookPro.getPrice() ); |

Como os decoradores são capazes de modificar objetos dinamicamente, eles são um padrão perfeito para alterar sistemas existentes. Ocasionalmente, é apenas mais simples criar decoradores em torno de um objeto em vez do problema de manter subclasses individuais para cada tipo de objeto. Isso torna a manutenção de aplicativos que podem exigir um grande número de objetos de subclasse significativamente mais simples.

Uma versão funcional deste exemplo pode ser encontrada em [JSBin](http://jsbin.com/UMEJaXu/1/edit) .

Decoradores com jQuery

Tal como acontece com outros padrões que abordamos, também existem exemplos do padrão Decorator que pode ser implementado com jQuery. jQuery.extend()nos permite estender (ou mesclar) dois ou mais objetos (e suas propriedades) juntos em um único objeto em tempo de execução.

Nesse cenário, um objeto de destino pode ser decorado com uma nova funcionalidade sem necessariamente quebrar ou substituir os métodos existentes no objeto de origem / superclasse (embora isso possa ser feito).

No exemplo a seguir, definimos três objetos: padrões, opções e configurações. O objetivo da tarefa é decorar o defaultsobjeto com funcionalidades adicionais encontradas em optionssettings. Nós devemos:

(a) Deixe os "padrões" em um estado inalterado onde não perdemos a capacidade de acessar as propriedades ou funções encontradas neles posteriormente (b) Ganhe a capacidade de usar as propriedades decoradas e funções encontradas em "opções"

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54 | 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 world" ); } }  // options -- { validate: true, name: bar, helloWorld: function (){ console.log( "hello world" ); } }  // defaults -- { validate: false, limit: 5, name: foo, welcome: function (){ console.log("welcome!"); } } |

Vantagens desvantagens

Os desenvolvedores gostam de usar esse padrão, pois ele pode ser usado de forma transparente e também é bastante flexível - como vimos, os objetos podem ser embrulhados ou "decorados" com um novo comportamento e então continuar a ser usados ​​sem a necessidade de se preocupar com o objeto base sendo modificado . Em um contexto mais amplo, esse padrão também evita que precisemos contar com um grande número de subclasses para obter os mesmos benefícios.

No entanto, existem desvantagens das quais devemos estar cientes ao implementar o padrão. Se mal gerenciado, pode complicar significativamente nossa arquitetura de aplicativo, pois introduz muitos objetos pequenos, mas semelhantes em nosso namespace. A preocupação aqui é que, além de se tornar difícil de gerenciar, outros desenvolvedores não familiarizados com o padrão podem ter dificuldade em entender por que ele está sendo usado.

Comentários suficientes ou pesquisa de padrões devem ajudar com o último, no entanto, enquanto mantivermos o controle sobre o quão difundido usamos o decorador em nossos aplicativos, estaremos bem em ambos os casos.

Flyweight

O padrão Flyweight é uma solução estrutural clássica para otimizar código que é repetitivo, lento e compartilha dados de forma ineficiente. Tem como objetivo minimizar o uso de memória em um aplicativo, compartilhando o máximo de dados possível com objetos relacionados (por exemplo, configuração do aplicativo, estado e assim por diante).

O padrão foi concebido por Paul Calder e Mark Linton em 1990 e foi nomeado após a classe de peso do boxe que inclui lutadores com peso inferior a 112 lb. O próprio nome Flyweight é derivado dessa classificação de peso, pois se refere ao pequeno peso (pegada de memória) que o padrão visa nos ajudar a alcançar.

Na prática, o compartilhamento de dados Flyweight pode envolver vários objetos semelhantes ou construções de dados usados ​​por vários objetos e colocar esses dados em um único objeto externo. Podemos passar esse objeto para aqueles que dependem desses dados, em vez de armazenar dados idênticos em cada um.

Usando Flyweights

Existem duas maneiras de aplicar o padrão Flyweight. O primeiro é na camada de dados, onde lidamos com o conceito de compartilhamento de dados entre grandes quantidades de objetos semelhantes armazenados na memória.

O segundo é na camada DOM onde o Flyweight pode ser usado como um gerenciador de eventos central para evitar anexar manipuladores de eventos a cada elemento filho em um contêiner pai que desejamos ter algum comportamento semelhante.

Como a camada de dados é onde o padrão flyweight é mais usado tradicionalmente, vamos dar uma olhada nisso primeiro.

Flyweights e compartilhamento de dados

Para esta aplicação, existem mais alguns conceitos em torno do padrão Flyweight clássico que precisamos conhecer. No padrão Flyweight, há um conceito de dois estados - intrínseco e extrínseco. Informações intrínsecas podem ser exigidas por métodos internos em nossos objetos, sem os quais eles absolutamente não podem funcionar. No entanto, as informações extrínsecas podem ser removidas e armazenadas externamente.

Objetos com os mesmos dados intrínsecos podem ser substituídos por um único objeto compartilhado, criado por um método de fábrica. Isso nos permite reduzir a quantidade geral de dados implícitos armazenados de forma bastante significativa.

A vantagem disso é que podemos ficar de olho nos objetos que já foram instanciados, de modo que novas cópias só sejam criadas caso o estado intrínseco seja diferente do objeto que já temos.

Usamos um gerenciador para lidar com os estados extrínsecos. A forma como isso é implementado pode variar, mas uma abordagem para isso é fazer com que o objeto gerenciador contenha um banco de dados central dos estados extrínsecos e dos objetos flyweight aos quais eles pertencem.

Implementando Flyweights Clássicos

Como o padrão Flyweight não tem sido muito usado em JavaScript nos últimos anos, muitas das implementações que podemos usar como inspiração vêm dos mundos Java e C ++.

Nossa primeira olhada no Flyweights em código é minha implementação em JavaScript do exemplo Java do padrão Flyweight da Wikipedia ( [http://en.wikipedia.org/wiki/Flyweight\_pattern](https://en.wikipedia.org/wiki/Flyweight_pattern) ).

Estaremos fazendo uso de três tipos de componentes Flyweight nesta implementação, que estão listados abaixo:

* **Flyweight** corresponde a uma interface através da qual flyweights são capazes de receber e agir em estados extrínsecos
* **Concreto Flyweight** realmente implementa a interface Flyweight e armazena o estado intrínseco. Os flyweights de concreto precisam ser compartilháveis ​​e capazes de manipular o estado que é extrínseco
* **Flyweight Factory** gerencia objetos flyweight e os cria também. Ele garante que nossos pesos-mosca sejam compartilhados e os gerencia como um grupo de objetos que podem ser consultados se exigirmos instâncias individuais. Se um objeto já foi criado no grupo, ele o retorna, caso contrário, adiciona um novo objeto ao pool e o retorna.

Eles correspondem às seguintes definições em nossa implementação:

* CoffeeOrder: Flyweight
* CoffeeFlavor: Concreto Flyweight
* CoffeeOrderContext: Helper
* CoffeeFlavorFactory: Flyweight Factory
* testFlyweight: Utilização de nossos Flyweights

"Implementos" de punção de pato

A punção duck nos permite estender os recursos de uma linguagem ou solução sem necessariamente precisar modificar a fonte de tempo de execução. Como a próxima solução requer o uso de uma palavra-chave Java ( implements) para implementar interfaces e não é encontrada em JavaScript nativamente, vamos primeiro dar um soco nisso.

Function.prototype.implementsFor funciona em um construtor de objeto e aceitará uma classe pai (função) ou objeto e herdará disso usando herança normal (para funções) ou herança virtual (para objetos).

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | // Simulate pure virtual inheritance/"implement" keyword for JS  Function.prototype.implementsFor = function( parentClassOrObject ){      if ( parentClassOrObject.constructor === Function )      {          // Normal Inheritance          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;  }; |

Podemos usar isso para corrigir a falta de uma implementspalavra - chave, fazendo com que uma função herde uma interface explicitamente. Abaixo, CoffeeFlavorimplementa a CoffeeOrderinterface e deve conter seus métodos de interface para que possamos atribuir a funcionalidade que alimenta essas implementações a um objeto.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118 | // 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 (typeof 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 = [],      // The tables for the orders.      tables = [],      // Number of orders made      ordersMade = 0,      // The CoffeeFlavorFactory instance      flavorFactory = new CoffeeFlavorFactory();      function takeOrders( flavorIn, table) {       flavors.push( flavorFactory.getCoffeeFlavor( flavorIn ) );       tables.push( new CoffeeOrderContext( table ) );       ordersMade++;    }       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());  } |

Converter o código para usar o padrão Flyweight

A seguir, vamos continuar nossa análise do Flyweights implementando um sistema para gerenciar todos os livros em uma biblioteca. Os metadados importantes para cada livro provavelmente poderiam ser divididos da seguinte forma:

* EU IRIA
* Título
* Autor
* Gênero
* Contagem de páginas
* ID do editor
* ISBN

Também exigiremos as seguintes propriedades para rastrear qual membro retirou um livro específico, a data em que o fez o check-out, bem como a data prevista de devolução.

* checkoutDate
* checkoutMember
* dueReturnDate
* disponibilidade

Cada livro seria, portanto, representado da seguinte forma, antes de qualquer otimização usando o padrão Flyweight:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | var Book = function( id, title, author, genre, pageCount,publisherID, ISBN, checkoutDate, checkoutMember, dueReturnDate,availability ){       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 );       }  }; |

Isso provavelmente funciona bem inicialmente para pequenas coleções de livros, no entanto, à medida que a biblioteca se expande para incluir um estoque maior com várias versões e cópias de cada livro disponíveis, podemos descobrir que o sistema de gerenciamento está cada vez mais lento com o tempo. Usar milhares de objetos de livro pode sobrecarregar a memória disponível, mas podemos otimizar nosso sistema usando o padrão Flyweight para melhorar isso.

Agora podemos separar nossos dados em estados intrínsecos e extrínsecos da seguinte maneira: os dados relevantes para o objeto do livro ( title, authoretc) são intrínsecos, enquanto os dados de checkout ( checkoutMember, dueReturnDateetc) são considerados extrínsecos. Efetivamente, isso significa que apenas um objeto Livro é necessário para cada combinação de propriedades do livro. ainda é uma quantidade considerável de objetos, mas significativamente menos do que tínhamos anteriormente.

A seguinte instância única de nossas combinações de metadados de livro será compartilhada entre todas as cópias de um livro com um título específico.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | // 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;    }; |

Como podemos ver, os estados extrínsecos foram removidos. Tudo o que está relacionado com check-outs de biblioteca será movido para um gerente e como os dados do objeto agora estão segmentados, uma fábrica pode ser usada para instanciação.

Uma Fábrica Básica

Vamos agora definir uma fábrica muito básica. O que vamos fazer é fazer uma verificação para ver se um livro com um título específico foi criado anteriormente dentro do sistema; se tiver, iremos devolvê-lo - caso contrário, um novo livro será criado e armazenado para que possa ser acessado posteriormente. Isso garante que criamos apenas uma única cópia de cada dado intrínseco exclusivo:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | // 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;          }      }    };    })(); |

Gerenciando os estados extrínsecos

Em seguida, precisamos armazenar os estados que foram removidos dos objetos Book em algum lugar - felizmente, um gerenciador (que definiremos como um Singleton) pode ser usado para encapsulá-los. As combinações de um objeto Livro e o membro da biblioteca que fez o check-out serão chamadas de registros do livro. Nosso gerente armazenará ambos e também incluirá a lógica relacionada à verificação que eliminamos durante a otimização do flyweight da classe Book.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | // 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, dueReturnDate, availability ) {          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 );      }    };    })(); |

O resultado dessas mudanças é que todos os dados extraídos da *classe* Book agora estão sendo armazenados em um atributo do singleton BookManager (BookDatabase) - algo consideravelmente mais eficiente do que o grande número de objetos que estávamos usando anteriormente. Métodos relacionados a checkouts de livros também são baseados aqui, pois lidam com dados que são extrínsecos em vez de intrínsecos.

Esse processo adiciona um pouco de complexidade à nossa solução final; no entanto, é uma pequena preocupação quando comparada aos problemas de desempenho que foram resolvidos. Em termos de dados, se temos 30 cópias do mesmo livro, agora o estamos armazenando apenas uma vez. Além disso, todas as funções ocupam memória. Com o padrão flyweight, essas funções existem em um lugar (no gerenciador) e não em todos os objetos, economizando assim o uso de memória. Para a versão não otimizada do flyweight mencionada acima, armazenamos apenas um link para o objeto de função, pois usamos o protótipo do construtor Book, mas se fosse implementado de outra forma, as funções seriam criadas para cada instância do livro.

O padrão Flyweight e o DOM

O DOM (Document Object Model) suporta duas abordagens que permitem que os objetos detectem eventos - de cima para baixo (captura de evento) ou de baixo para cima (bolha de evento).

Na captura de evento, o evento é primeiro capturado pelo elemento mais externo e propagado para o elemento mais interno. No borbulhamento de evento, o evento é capturado e fornecido ao elemento mais interno e, em seguida, propagado para os elementos externos.

Uma das melhores metáforas para descrever Flyweights neste contexto foi escrita por Gary Chisholm e é mais ou menos assim:

Tente pensar no peso-mosca em termos de um lago. Um peixe abre a boca (o evento), as bolhas sobem à superfície (o borbulhar) uma mosca sentada no topo voa para longe quando a bolha atinge a superfície (a ação). Neste exemplo podemos facilmente transpor o peixe abrindo sua boca para um botão sendo clicado, as bolhas como o efeito borbulhante e a mosca voando para alguma função sendo executada

Bubbling foi introduzido para lidar com situações em que um único evento (por exemplo, um clique) pode ser manipulado por vários manipuladores de eventos definidos em diferentes níveis da hierarquia DOM. Onde isso acontece, o borbulhamento de eventos executa manipuladores de eventos definidos para elementos específicos no nível mais baixo possível. A partir daí, o evento borbulha para conter elementos antes de ir para aqueles ainda mais altos.

Flyweights podem ser usados ​​para ajustar o processo de bolha de eventos ainda mais, como veremos em breve.

Exemplo 1: tratamento centralizado de eventos

Para nosso primeiro exemplo prático, imagine que temos vários elementos semelhantes em um documento com comportamento semelhante executado quando uma ação do usuário (por exemplo, clicar, passar o mouse) é executada contra eles.

Normalmente, o que fazemos ao construir nosso próprio componente de acordeão, menu ou outro widget baseado em lista é vincular um evento de clique a cada elemento de link no contêiner pai (por exemplo, em $('ul li a').on(..)vez de vincular o clique a vários elementos, podemos facilmente anexar um Flyweight a a parte superior do nosso contêiner, que pode escutar eventos vindos de baixo. Eles podem então ser tratados usando uma lógica tão simples ou complexa quanto necessário.

Como os tipos de componentes mencionados geralmente têm a mesma marcação de repetição para cada seção (por exemplo, cada seção de um acordeão), há uma boa chance de que o comportamento de cada elemento que pode ser clicado seja bastante semelhante e relativo a classes semelhantes nas proximidades. Usaremos essas informações para construir um acordeão muito básico usando o Flyweight abaixo.

Um namespace stateManager é usado aqui para encapsular nossa lógica flyweight enquanto jQuery é usado para vincular o clique inicial a um div de contêiner. Para garantir que nenhuma outra lógica na página esteja anexando identificadores semelhantes ao contêiner, um evento unbind é aplicado primeiro.

Agora, para estabelecer exatamente qual elemento filho no contêiner é clicado, usamos uma targetverificação que fornece uma referência ao elemento que foi clicado, independentemente de seu pai. Em seguida, usamos essas informações para manipular o evento de clique sem realmente precisar vincular o evento a filhos específicos quando nossa página é carregada.

**HTML**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <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&amp;width=500...gt;>"</iframe>         </span>     </div>  </div> |

**JavaScript**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | var stateManager = {      fly: function () {        var self = this;        $( "#container" )            .unbind()            .on( "click", "div.toggle", function ( e ) {              self.handleClick( e.target );            });    },      handleClick: function ( elem ) {      $( elem ).find( "span" ).toggle( "slow" );    }  }; |

O benefício aqui é que estamos convertendo muitas ações independentes em ações compartilhadas (potencialmente economizando memória).

Exemplo 2: Usando o Flyweight para otimização de desempenho

Em nosso segundo exemplo, faremos referência a alguns ganhos de desempenho adicionais que podem ser alcançados usando Flyweights com jQuery.

James Padolsey escreveu anteriormente um artigo chamado *76 bytes para jQuery mais rápido,* onde nos lembrou que cada vez que o jQuery dispara um retorno de chamada, independentemente do tipo (filtro, cada, manipulador de eventos), somos capazes de acessar o contexto da função (o elemento DOM relacionado a ele) por meio da thispalavra - chave.

Infelizmente, muitos de nós nos acostumamos com a idéia de envolvimento thisem $()ou jQuery(), o que significa que uma nova instância do jQuery é desnecessariamente construídos cada vez, ao invés de simplesmente fazer isso:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | $("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 queria usar jQuery jQuery.textno seguinte contexto, no entanto, ele discordou da noção de que um novo objeto jQuery tinha que ser criado em cada iteração:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3 | $( "a" ).map( function () {    return $( this ).text();  }); |

Agora, com relação ao empacotamento redundante, onde possível com os métodos utilitários do jQuery, é melhor usar jQuery.methodName(por exemplo jQuery.text) ao invés de jQuery.fn.methodName(por exemplo jQuery.fn.text) onde o methodName representa um utilitário como each()ou text. Isso evita a necessidade de chamar um nível adicional de abstração ou construir um novo objeto jQuery cada vez que nossa função é chamada da maneira jQuery.methodNameque a própria biblioteca usa em um nível inferior para poder jQuery.fn.methodName.

No entanto, porque nem todos os métodos jQuery têm funções de nó único correspondentes, Padolsey concebeu a ideia de um utilitário jQuery.single.

A ideia aqui é que um único objeto jQuery seja criado e usado para cada chamada para jQuery.single (efetivamente significando que apenas um objeto jQuery é criado). A implementação para isso pode ser encontrada abaixo e, como estamos consolidando dados para vários objetos possíveis em uma estrutura singular mais central, é tecnicamente também um Flyweight.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | jQuery.single = (function( o ){       var collection = jQuery([1]);     return function( element ) {           // Give collection the element:         collection[0] = element;            // Return the collection:         return collection;       };  })(); |

Um exemplo disso em ação com encadeamento é:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | $( "div" ).on( "click", function () {       var html = jQuery.single( this ).next().html();     console.log( html );    }); |

Nota: Embora possamos acreditar que simplesmente armazenar em cache nosso código jQuery pode oferecer ganhos de desempenho equivalentes, Padolsey afirma que $ .single () ainda vale a pena usar e pode ter um desempenho melhor. Isso não quer dizer que você não aplique nenhum armazenamento em cache, apenas esteja ciente de que essa abordagem pode ajudar. Para mais detalhes sobre $ .único, recomendo a leitura da postagem completa de Padolsey.

Padrões JavaScript MV \*

Nesta seção, vamos revisar três padrões de arquitetura muito importantes - MVC (Model-View-Controller), MVP (Model-View-Presenter) e MVVM (Model-View-ViewModel). No passado, esses padrões foram muito usados ​​para estruturar aplicativos de desktop e do lado do servidor, mas só nos últimos anos começaram a ser aplicados ao JavaScript.

Como a maioria dos desenvolvedores de JavaScript atualmente usando esses padrões optam por utilizar bibliotecas como Backbone.js para implementar uma estrutura semelhante a MVC / MV \*, compararemos como as soluções modernas como essa diferem em sua interpretação de MVC em comparação com abordagens clássicas esses padrões.

Vamos primeiro cobrir o básico.

MVC

MVC é um padrão de design de arquitetura que incentiva a organização aprimorada do aplicativo por meio de uma separação de interesses. Ele reforça o isolamento dos dados de negócios (Modelos) das interfaces do usuário (Visualizações), com um terceiro componente (Controladores) gerenciando tradicionalmente a lógica e a entrada do usuário. O padrão foi originalmente projetado por [Trygve Reenskaug](https://en.wikipedia.org/wiki/Trygve_Reenskaug) durante seu tempo trabalhando em Smalltalk-80 (1979), onde foi inicialmente chamado de Model-View-Controller-Editor. MVC passou a ser descrito em profundidade em 1995, [“Design Patterns: Elements of Reusable Object-Oriented Software”](https://www.amazon.co.uk/Design-patterns-elements-reusable-object-oriented/dp/0201633612) (o livro "GoF"), que desempenhou um papel na popularização de seu uso.

Smalltalk-80 MVC

É importante entender o que o padrão MVC original pretendia resolver, já que sofreu uma forte mutação desde os dias de sua origem. Na década de 70, as interfaces gráficas com o usuário eram poucas e distantes entre si e um conceito conhecido como [Apresentação Separada](http://martinfowler.com/eaaDev/uiArchs.html) começou a ser usado como um meio para fazer uma divisão clara entre objetos de domínio que modelavam conceitos no mundo real (por exemplo, uma foto, uma pessoa ) e os objetos de apresentação que foram renderizados na tela do usuário.

A implementação do MVC em Smalltalk-80 levou esse conceito adiante e tinha o objetivo de separar a lógica do aplicativo da interface do usuário. A ideia era que desacoplar essas partes da aplicação também permitiria a reutilização de modelos para outras interfaces da aplicação. Existem alguns pontos interessantes a serem observados sobre a arquitetura MVC do Smalltalk-80:

* Um modelo representava dados específicos do domínio e não conhecia a interface do usuário (visualizações e controladores). Quando um modelo mudasse, ele informaria seus observadores.
* Uma vista representava o estado atual de um modelo. O padrão Observer foi usado para permitir que a View saiba sempre que o Model foi atualizado ou modificado.
* A apresentação era cuidada pelo View, mas não havia apenas um View e Controller - um par View-Controller era necessário para cada seção ou elemento sendo exibido na tela.
* A função dos Controladores neste par era lidar com a interação do usuário (como pressionamentos de teclas e ações, por exemplo, cliques), tomando decisões para a Visualização.

Os desenvolvedores às vezes ficam surpresos quando descobrem que o padrão Observer (atualmente comumente implementado como a variação Publicar / Assinar) foi incluído como parte da arquitetura do MVC há muitas décadas. No MVC do Smalltalk-80, a vista observa o modelo. Conforme mencionado no ponto acima, sempre que o modelo muda, as visualizações reagem. Um exemplo simples disso é um aplicativo apoiado por dados do mercado de ações - para que o aplicativo seja útil, qualquer alteração nos dados em nossos Modelos deve resultar na atualização instantânea da Visualização.

Martin Fowler fez um excelente trabalho escrevendo sobre as [origens](http://martinfowler.com/eaaDev/uiArchs.html) do MVC ao longo dos anos e, se estiver interessado em mais informações históricas sobre o MVC do Smalltalk-80, recomendo a leitura de seu trabalho.

MVC para desenvolvedores de JavaScript

Revisamos os anos 70, mas vamos voltar ao aqui e agora. Nos tempos modernos, o padrão MVC foi aplicado a uma ampla gama de linguagens de programação, incluindo a mais relevante para nós: JavaScript. JavaScript agora tem uma série de frameworks com suporte para MVC (ou variações dele, que chamamos de família MV \*), permitindo que os desenvolvedores adicionem estrutura facilmente a seus aplicativos sem grande esforço.

Essas estruturas incluem Backbone, Ember.js e AngularJS. Dada a importância de evitar o código "espaguete", um termo que descreve o código que é muito difícil de ler ou manter devido à sua falta de estrutura, é imperativo que o desenvolvedor de JavaScript moderno entenda o que esse padrão oferece. Isso nos permite avaliar com eficácia o que essas estruturas nos permitem fazer de maneira diferente.

Sabemos que o MVC é composto por três componentes principais:

Modelos

Os modelos gerenciam os dados de um aplicativo. Eles não se preocupam nem com a interface do usuário nem com as camadas de apresentação, mas representam formas exclusivas de dados que um aplicativo pode exigir. Quando um modelo muda (por exemplo, quando é atualizado), ele normalmente notifica seus observadores (por exemplo, visualizações, um conceito que abordaremos em breve) que uma mudança ocorreu para que eles possam reagir de acordo.

Para entender melhor os modelos, vamos imaginar que temos um aplicativo de galeria de fotos em JavaScript. Em uma galeria de fotos, o conceito de uma foto merece seu próprio modelo, pois representa um tipo único de dados específicos de domínio. Tal modelo pode conter atributos relacionados, como legenda, fonte de imagem e metadados adicionais. Uma foto específica seria armazenada em uma instância de um modelo e um modelo também pode ser reutilizável. Abaixo podemos ver um exemplo de um modelo muito simplista implementado usando o Backbone.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | var Photo = Backbone.Model.extend({        // Default attributes for the photo      defaults: {        src: "placeholder.jpg",        caption: "A default image",        viewed: false      },        // Ensure that each photo created has an `src`.      initialize: function() {         this.set( { "src": this.defaults.src} );      }    }); |

Os recursos integrados dos modelos variam entre as estruturas, no entanto, é bastante comum que eles suportem a validação de atributos, onde os atributos representam as propriedades do modelo, como um identificador de modelo. Ao usar modelos em aplicativos do mundo real, geralmente também desejamos a persistência do modelo. A persistência nos permite editar e atualizar modelos com o conhecimento de que seu estado mais recente será salvo em: memória, no armazenamento de dados local do usuário ou sincronizado com um banco de dados.

Além disso, um modelo também pode ter múltiplas visualizações observando-o. Se, digamos, nosso modelo de foto contivesse metadados, como sua localização (longitude e latitude), amigos que estavam presentes na foto (uma lista de identificadores) e uma lista de tags, um desenvolvedor pode decidir fornecer uma única visualização para exibir cada uma dessas três facetas.

Não é incomum que os frameworks MVC / MV \* modernos forneçam um meio de agrupar modelos (por exemplo, no Backbone, esses grupos são chamados de "coleções"). Gerenciar modelos em grupos nos permite escrever a lógica do aplicativo com base nas notificações do grupo, caso qualquer modelo que ele contenha seja alterado. Isso evita a necessidade de observar manualmente as instâncias individuais do modelo.

Um exemplo de agrupamento de modelos em uma coleção simplificada de Backbone pode ser visto abaixo.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | 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() );      }  }); |

Textos mais antigos no MVC também podem conter referência a uma noção de modelos que gerenciam o *estado do* aplicativo . Em aplicativos JavaScript, o *estado* tem uma conotação diferente, normalmente se referindo ao "estado" atual, ou seja, visualização ou subvisualização (com dados específicos) na tela de um usuário em um ponto fixo. Estado é um tópico regularmente discutido quando se olha para aplicativos de página única, onde o conceito de estado precisa ser simulado.

Para resumir, os modelos preocupam-se principalmente com os dados de negócios.

Visualizações

As visualizações são uma representação visual de modelos que apresentam uma visualização filtrada de seu estado atual. Enquanto as visualizações do Smalltalk tratam de pintar e manter um bitmap, as visualizações do JavaScript tratam da construção e manutenção de um elemento DOM.

Uma visão normalmente observa um modelo e é notificada quando o modelo muda, permitindo que a visão se atualize de acordo. A literatura sobre padrões de projeto comumente se refere a visualizações como "burras", uma vez que seu conhecimento de modelos e controladores em uma aplicação é limitado.

Os usuários podem interagir com as visualizações e isso inclui a capacidade de ler e editar (ou seja, obter ou definir os valores de atributo em) modelos. Como a visualização é a camada de apresentação, geralmente apresentamos a capacidade de editar e atualizar de uma forma amigável. Por exemplo, no antigo aplicativo de galeria de fotos que discutimos anteriormente, a edição do modelo pode ser facilitada por meio de uma visualização de "edição", onde um usuário que selecionou uma foto específica pode editar seus metadados.

A tarefa real de atualizar o modelo recai sobre os controladores (que abordaremos em breve).

Vamos explorar as visualizações um pouco mais, usando uma implementação de exemplo de JavaScript vanilla. Abaixo, podemos ver uma função que cria uma única visualização de fotos, consumindo uma instância de modelo e uma instância de controlador.

Definimos um render()utilitário dentro de nossa visão que é responsável por renderizar o conteúdo do photoModelusando um mecanismo de modelagem JavaScript (modelagem sublinhada) e atualizar o conteúdo de nossa visão, referenciado por photoEl.

O photoModelentão adiciona nosso render()retorno de chamada como um de seus assinantes para que, por meio do padrão Observer, possamos acionar a visualização para atualizar quando o modelo mudar.

Alguém pode se perguntar onde a interação do usuário entra em jogo aqui. Quando os usuários clicam em qualquer elemento da visualização, não é responsabilidade da visualização saber o que fazer a seguir. Ele depende de um controlador para tomar essa decisão. Em nossa implementação de exemplo, isso é obtido adicionando um ouvinte de evento ao photoElqual delegará o tratamento do comportamento de clique de volta ao controlador, passando as informações do modelo junto com ele, caso seja necessário.

O benefício dessa arquitetura é que cada componente desempenha sua própria função separada ao fazer o aplicativo funcionar conforme necessário.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | 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    };    }; |

**Templating**

No contexto de estruturas JavaScript que suportam MVC / MV \*, vale a pena discutir brevemente a modelagem JavaScript e sua relação com as visualizações, conforme abordamos brevemente na última seção.

Há muito tempo é considerado (e comprovado) uma má prática de desempenho criar manualmente grandes blocos de marcação HTML na memória por meio da concatenação de strings. Os desenvolvedores que fazem isso são vítimas de iteração informante por meio de seus dados, envolvendo-os em divs aninhados e usando técnicas desatualizadas, como document.writeinjetar o "modelo" no DOM. Como isso normalmente significa manter a marcação de script em linha com nossa marcação padrão, pode rapidamente se tornar difícil de ler e, mais importante, manter esses desastres, especialmente ao construir aplicativos de tamanho não trivial.

Soluções de modelagem JavaScript (como Handlebars.js e Mustache) são frequentemente usadas para definir modelos para visualizações como marcação (armazenadas externamente ou dentro de tags de script com um tipo personalizado - por exemplo, texto / modelo) contendo variáveis ​​de modelo. As variáveis ​​podem ser delimitadas usando uma sintaxe de variável (por exemplo, {{nome}}) e os frameworks são normalmente inteligentes o suficiente para aceitar dados em um formato JSON (para o qual as instâncias do modelo podem ser convertidas), de modo que precisamos apenas nos preocupar em manter modelos limpos e modelos limpos. A maior parte do trabalho pesado a ser feito com a população é realizada pela própria estrutura. Isso tem um grande número de benefícios, especialmente ao optar por armazenar modelos externamente, pois isso pode permitir que os modelos sejam carregados dinamicamente conforme a necessidade, quando se trata de criar aplicativos maiores.

Abaixo, podemos ver dois exemplos de modelos HTML. Um implementado usando o popular framework Handlebars.js e outro usando os modelos do Underscore.

**Handlebars.js:**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <li class="photo">    <h2>{{caption}}</h2>    <img class="source" src="{{src}}"/>    <div class="meta-data">      {{metadata}}    </div>  </li> |

**Microtemplates Underscore.js:**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <li class="photo">    <h2><%= caption %></h2>    <img class="source" src="<%= src %>"/>    <div class="meta-data">      <%= metadata %>    </div>  </li> |

Observe que os modelos não são visualizações em si. Desenvolvedores vindos de uma arquitetura Struts Model 2 podem sentir que um template \* é \* uma visão, mas não é. Uma vista é um objeto que observa um modelo e mantém a representação visual atualizada. Um modelo \* pode \* ser uma forma declarativa de especificar parte ou até mesmo todo um objeto de visualização para que possa ser gerado a partir da especificação do modelo.

Também é importante notar que no desenvolvimento clássico da web, navegar entre visualizações independentes exigia o uso de uma atualização de página. Em aplicativos JavaScript de página única, no entanto, uma vez que os dados são buscados em um servidor via Ajax, eles podem simplesmente ser renderizados dinamicamente em uma nova visualização na mesma página sem que nenhuma atualização seja necessária.

A função da navegação, portanto, recai sobre um "roteador", que auxilia no gerenciamento do estado do aplicativo (por exemplo, permitindo que os usuários marquem uma visualização específica para a qual navegaram). Como os roteadores, entretanto, não fazem parte do MVC nem estão presentes em todas as estruturas do tipo MVC, não irei entrar em detalhes sobre eles nesta seção.

Para resumir, as visualizações são uma representação visual dos dados do nosso aplicativo.

Controladores

Os controladores são intermediários entre os modelos e as visualizações, classicamente responsáveis ​​por atualizar o modelo quando o usuário manipula a visualização.

Em nosso aplicativo de galeria de fotos, um controlador seria responsável por lidar com as alterações feitas pelo usuário na visualização de edição de uma foto específica, atualizando um modelo de foto específico quando um usuário terminar a edição.

Lembre-se de que os controladores cumprem uma função no MVC: a facilitação do padrão Strategy para a visualização. No que diz respeito ao padrão de estratégia, a visão delega ao controlador a seu critério. Então, é assim que funciona o padrão de estratégia. A visão pode delegar o tratamento de eventos do usuário ao controlador quando a visão achar conveniente. A visão \* poderia \* delegar o tratamento de eventos de mudança de modelo ao controlador se a visão achar adequado, mas esta não é a função tradicional do controlador.

Em termos de onde a maioria dos frameworks JavaScript MVC prejudica o que é convencionalmente considerado "MVC", no entanto, é com os controladores. As razões para isso variam, mas na minha opinião honesta, é que os autores do framework inicialmente olham para a interpretação do lado do servidor do MVC, percebem que ela não traduz 1: 1 no lado do cliente e reinterpretam o C em MVC significa algo que eles sentem que faz mais sentido. O problema com isso, no entanto, é que é subjetivo, aumenta a complexidade tanto na compreensão do padrão MVC clássico quanto, claro, do papel dos controladores em estruturas modernas.

Como exemplo, vamos revisar brevemente a arquitetura do popular framework arquitetônico Backbone.js. O backbone contém modelos e visualizações (algo semelhante ao que analisamos anteriormente), no entanto, não possui controladores verdadeiros. Suas visualizações e roteadores agem de maneira um pouco semelhante a um controlador, mas nenhum dos dois são controladores por conta própria.

A este respeito, ao contrário do que pode ser mencionado na documentação oficial ou em posts de blog, o Backbone não é um framework verdadeiramente MVC / MVP nem MVVM. Na verdade, é melhor considerá-lo um membro da família MV \* que aborda a arquitetura à sua maneira. É claro que não há nada de errado com isso, mas é importante distinguir entre MVC clássico e MV \* se começarmos a confiar nos conselhos da literatura clássica sobre o primeiro para ajudar com o último.

Controladores em outra biblioteca (Spine.js) vs Backbone.js

**Spine.js**

Agora sabemos que os controladores são tradicionalmente responsáveis ​​por atualizar o modelo quando o usuário atualiza a visualização. É interessante notar que o framework JavaScript MVC / MV \* mais popular no momento da escrita (Backbone) não tem seu **próprio** conceito explícito de controladores.

Portanto, pode ser útil revisar o controlador de outro framework MVC para avaliar a diferença nas implementações e demonstrar ainda como os frameworks não tradicionais abordam a função do controlador. Para isso, vamos dar uma olhada em um controlador de amostra do Spine.js:

Neste exemplo, teremos um controlador chamado PhotosControllerque ficará encarregado das fotos individuais no aplicativo. Isso garantirá que, quando a visualização for atualizada (por exemplo, um usuário editou os metadados da foto), o modelo correspondente também será.

Observação: não iremos nos aprofundar muito em Spine.js, mas apenas teremos uma visão de três metros do que seus controladores podem fazer:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | // 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();    }  }); |

No Spine, os controladores são considerados a cola de um aplicativo, adicionando e respondendo a eventos DOM, renderizando modelos e garantindo que as visualizações e modelos sejam mantidos em sincronia (o que faz sentido no contexto do que sabemos ser um controlador).

O que estamos fazendo no exemplo acima é a criação de ouvintes na updatee destroyeventos usando render()e remove(). Quando uma entrada de foto é atualizada, nós re-renderizamos a vista para refletir as mudanças nos metadados. Da mesma forma, se a foto for excluída da galeria, nós a removeremos da visualização. Na render()função, estamos usando o micro-modelo Underscore (via \_.template()) para renderizar um modelo JavaScript com o ID #photoTemplate. Isso simplesmente retorna uma string HTML compilada usada para preencher o conteúdo de photoEl.

O que isso nos fornece é uma maneira muito leve e simples de gerenciar as alterações entre o modelo e a visualização.

**Backbone.js**

Mais adiante nesta seção, vamos revisitar as diferenças entre o Backbone e o MVC tradicional, mas por enquanto vamos nos concentrar nos controladores.

No Backbone, compartilha-se a responsabilidade de um controlador com Backbone.Viewe Backbone.Router. Algum tempo atrás, o Backbone veio com o seu próprio Backbone.Controller, mas como a nomenclatura desse componente não fazia sentido para o contexto em que estava sendo usado, ele foi renomeado posteriormente para Roteador.

Os roteadores lidam um pouco mais com a responsabilidade do controlador, pois é possível vincular os eventos lá para os modelos e fazer com que nossa visualização responda aos eventos DOM e renderização. Como Tim Branyen (outro contribuidor do Backbone baseado no Bocoup) também apontou anteriormente, é possível escapar impunes de não precisar Backbone.Routerdisso, então uma maneira de pensar sobre isso usando o paradigma do Roteador é provavelmente:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | 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 );    }  }); |

Para resumir, a conclusão desta seção é que os controladores gerenciam a lógica e a coordenação entre modelos e visualizações em um aplicativo.

O que o MVC nos dá?

Esta separação de interesses em MVC facilita a modularização mais simples da funcionalidade de um aplicativo e permite:

* Manutenção geral mais fácil. Quando atualizações precisam ser feitas no aplicativo, é muito claro se as mudanças são centradas nos dados, significando mudanças nos modelos e possivelmente nos controladores, ou apenas visuais, significando mudanças nas visualizações.
* A dissociação de modelos e visualizações significa que é significativamente mais simples escrever testes de unidade para lógica de negócios
* A duplicação do modelo de baixo nível e do código do controlador (ou seja, o que podemos estar usando) é eliminada em todo o aplicativo
* Dependendo do tamanho do aplicativo e da separação de funções, esta modularidade permite que os desenvolvedores responsáveis ​​pela lógica principal e os desenvolvedores que trabalham nas interfaces do usuário trabalhem simultaneamente

Smalltalk-80 MVC em JavaScript

Embora a maioria dos frameworks JavaScript modernos tentem evoluir o paradigma MVC para melhor atender às diferentes necessidades de desenvolvimento de aplicativos da web, há um framework que tenta aderir à forma pura do padrão encontrado em Smalltalk-80. Maria.js ( <https://github.com/petermichaux/maria> ) de Peter Michaux oferece uma implementação que é fiel às origens dos MVCs - Modelos são modelos, Visualizações são visualizações e Controladores nada mais são do que controladores. Embora alguns desenvolvedores possam achar que um framework MV \* deva abordar mais questões, esta é uma referência útil a se estar ciente caso você queira uma implementação em JavaScript do MVC original.

Aprofundando

Neste ponto do livro, devemos ter um entendimento básico do que o padrão MVC oferece, mas ainda há algumas informações fascinantes sobre ele que vale a pena observar.

O GoF não se refere ao MVC como um padrão de design, mas sim um *conjunto de classes para construir uma interface de usuário* . Na visão deles, é na verdade uma variação de três padrões de design clássicos: os padrões Observer, Strategy e Composite. Dependendo de como o MVC foi implementado em uma estrutura, ele também pode usar os padrões Factory e Template. O livro GoF menciona esses padrões como extras úteis ao trabalhar com MVC.

Conforme discutimos, os modelos representam os dados do aplicativo, enquanto as visualizações são o que o usuário é apresentado na tela. Como tal, o MVC depende do padrão Observer para algumas de suas comunicações principais (algo que surpreendentemente não é abordado em muitos artigos sobre o padrão MVC). Quando um modelo é alterado, ele notifica seus observadores (Visualizações) de que algo foi atualizado - este é talvez o relacionamento mais importante no MVC. A natureza do observador desse relacionamento também é o que facilita a anexação de várias visualizações ao mesmo modelo.

Para desenvolvedores interessados ​​em saber mais sobre a natureza desacoplada do MVC (mais uma vez, dependendo da implementação), um dos objetivos do padrão é ajudar a definir relacionamentos um-para-muitos entre um tópico (objeto de dados) e seus observadores. Quando um tópico muda, seus observadores são atualizados. Visualizações e controladores têm um relacionamento ligeiramente diferente. Os controladores facilitam as visualizações para responder a diferentes entradas do usuário e são um exemplo do padrão Strategy.

Resumo

Tendo revisado o padrão MVC clássico, devemos agora entender como ele nos permite separar claramente os interesses em um aplicativo. Devemos agora apreciar como os frameworks JavaScript MVC podem diferir em sua interpretação do padrão MVC, que embora bastante aberto a variações, ainda compartilha alguns dos conceitos fundamentais que o padrão original tem a oferecer.

Ao revisar um novo framework JavaScript MVC / MV \*, lembre-se - pode ser útil dar um passo atrás e revisar como ele optou por abordar a arquitetura (especificamente, como ele suporta a implementação de modelos, visualizações, controladores ou outras alternativas), pois isso pode nos ajudar melhor grok como o framework espera ser usado.

MVP

Model-view-presenter (MVP) é um derivado do padrão de design MVC que se concentra em melhorar a lógica de apresentação. Ele se originou em uma empresa chamada [Taligent](https://en.wikipedia.org/wiki/Taligent) no início de 1990, enquanto eles trabalhavam em um modelo para um ambiente C ++ CommonPoint. Embora MVC e MVP tenham como objetivo a separação de interesses em vários componentes, existem algumas diferenças fundamentais entre eles.

Para os fins deste resumo, vamos nos concentrar na versão do MVP mais adequada para arquiteturas baseadas na web.

Modelos, exibições e apresentadores

O P em MVP significa apresentador. É um componente que contém a lógica de negócios da interface do usuário para a visualização. Ao contrário do MVC, as invocações da visualização são delegadas ao apresentador, que é desacoplado da visualização e, em vez disso, conversam com ele por meio de uma interface. Isso permite todos os tipos de coisas úteis, como ser capaz de simular visualizações em testes de unidade.

A implementação mais comum do MVP é aquela que usa uma visão passiva (uma visão que é para todos os efeitos "burra"), contendo pouca ou nenhuma lógica. Se MVC e MVP são diferentes, é porque C e P fazem coisas diferentes. No MVP, o P observa os modelos e atualiza as visualizações quando os modelos mudam. OP efetivamente vincula modelos a visualizações, uma responsabilidade que era anteriormente mantida pelos controladores em MVC.

Solicitados por uma visualização, os apresentadores realizam qualquer trabalho relacionado às solicitações do usuário e passam os dados de volta para eles. Nesse sentido, eles recuperam dados, os manipulam e determinam como os dados devem ser exibidos na visualização. Em algumas implementações, o apresentador também interage com uma camada de serviço para persistir os dados (modelos). Os modelos podem disparar eventos, mas é função do apresentador se inscrever neles para que possa atualizar a visualização. Nessa arquitetura passiva, não temos o conceito de vinculação direta de dados. As visualizações expõem setters que os apresentadores podem usar para definir dados.

O benefício dessa mudança do MVC é que ela aumenta a testabilidade de nosso aplicativo e fornece uma separação mais limpa entre a visualização e o modelo. No entanto, isso não é isento de custos, pois a falta de suporte de vinculação de dados no padrão pode frequentemente significar ter que cuidar dessa tarefa separadamente.

Embora uma implementação comum de uma [Visão Passiva](http://martinfowler.com/eaaDev/PassiveScreen.html) seja a visão implementar uma interface, existem variações nela, incluindo o uso de eventos que podem desacoplar a Visão do Apresentador um pouco mais. Como não temos a construção de interface em JavaScript, estamos usando mais um protocolo do que uma interface explícita aqui. Tecnicamente, ainda é uma API e provavelmente é justo que nos referamos a ela como uma interface dessa perspectiva.

Há também uma variação de [Supervising Controller](http://martinfowler.com/eaaDev/SupervisingPresenter.html) do MVP, que está mais próxima dos padrões MVC e [MVVM](https://en.wikipedia.org/wiki/Model_View_ViewModel) , pois fornece vinculação de dados do modelo diretamente da visualização. Os plug-ins de observação de valor-chave (KVO) (como o plug-in Backbone.ModelBinding de Derick Bailey) tendem a trazer o Backbone da Visualização Passiva e mais para o Controlador de Supervisão ou variações MVVM.

MVP ou MVC?

O MVP geralmente é usado com mais frequência em aplicativos de nível corporativo, onde é necessário reutilizar o máximo possível de lógica de apresentação. Aplicativos com visualizações muito complexas e uma grande quantidade de interação do usuário podem descobrir que o MVC não se encaixa muito bem aqui, já que resolver esse problema pode significar depender muito de vários controladores. No MVP, toda essa lógica complexa pode ser encapsulada em um apresentador, o que pode simplificar muito a manutenção.

Como as visualizações MVP são definidas por meio de uma interface e a interface é tecnicamente o único ponto de contato entre o sistema e a visualização (além de um apresentador), este padrão também permite que os desenvolvedores escrevam lógica de apresentação sem precisar esperar que os designers produzam layouts e gráficos para o aplicativo.

Dependendo da implementação, o MVP pode ser mais fácil de testar a unidade automaticamente do que o MVC. O motivo frequentemente citado para isso é que o apresentador pode ser usado como uma simulação completa da interface do usuário e, portanto, pode ser testado na unidade independentemente de outros componentes. Na minha experiência, isso realmente depende das linguagens em que estamos implementando o MVP (há uma grande diferença entre optar pelo MVP para um projeto JavaScript em vez de, digamos, ASP.net).

No final do dia, as preocupações subjacentes que podemos ter com o MVC provavelmente serão verdadeiras para o MVP, dado que as diferenças entre eles são principalmente semânticas. Enquanto estivermos separando claramente as preocupações em modelos, visualizações e controladores (ou apresentadores), devemos obter a maioria dos mesmos benefícios, independentemente da variação que escolhermos.

MVC, MVP e Backbone.js

Existem muito poucos, se houver, frameworks de arquitetura JavaScript que afirmam implementar os padrões MVC ou MVP em sua forma clássica, já que muitos desenvolvedores de JavaScript não veem MVC e MVP como sendo mutuamente exclusivos (na verdade, é mais provável que vejamos MVP estritamente implementado quando olhando para estruturas da web, como ASP.net ou GWT). Isso ocorre porque é possível ter lógica de apresentador / visualização adicional em nosso aplicativo e ainda considerá-la uma espécie de MVC.

A colaboradora do backbone [Irene Ros](http://ireneros.com/) (da Bocoup, sediada em Boston) concorda com essa forma de pensar, pois quando separa as visualizações em seus próprios componentes distintos, ela precisa de algo para realmente montá-los para ela. Isso pode ser uma rota de controlador (como uma Backbone.Router, abordada posteriormente no livro) ou um retorno de chamada em resposta aos dados sendo buscados.

Dito isso, alguns desenvolvedores acham que Backbone.js se encaixa melhor na descrição de MVP do que MVC. A visão deles é que:

* O apresentador no MVP descreve melhor o Backbone.View(a camada entre os modelos de visualização e os dados vinculados a ele) do que um controlador.
* O modelo se encaixa Backbone.Model(não é muito diferente dos modelos em MVC)
* As visualizações representam melhor os modelos (por exemplo, modelos de marcação de barras de guia / bigode)

Uma resposta a isso pode ser que a visualização também pode ser apenas uma visualização (de acordo com MVC) porque o Backbone é flexível o suficiente para permitir que seja usado para vários fins. O V no MVC e o P no MVP podem ser realizados Backbone.Viewporque eles são capazes de atingir dois objetivos: renderizar componentes atômicos e montar esses componentes renderizados por outras visualizações.

Também vimos que no Backbone a responsabilidade de um controlador é compartilhada com o Backbone.View e o Backbone.Router e, no exemplo a seguir, podemos realmente ver que aspectos disso são certamente verdadeiros.

Nosso Backbone PhotoViewusa o padrão Observer para "inscrever-se" nas alterações do modelo de uma View na linha this.model.bind("change",...). Ele também lida com modelos no render()método, mas ao contrário de algumas outras implementações, a interação do usuário também é tratada na Visualização (consulte Recursos events).

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | 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();      }    }); |

Outra opinião (bem diferente) é que o Backbone se assemelha mais ao [Smalltalk-80 MVC](http://martinfowler.com/eaaDev/uiArchs.html#ModelViewController) , que examinamos anteriormente.

Como o blogueiro regular do Backbone, Derick Bailey, [já](http://lostechies.com/derickbailey/2011/12/23/backbone-js-is-not-an-mvc-framework/) disse, é melhor não forçar o Backbone a se ajustar a nenhum padrão de design específico. Os padrões de projeto devem ser considerados guias flexíveis de como os aplicativos podem ser estruturados e, a esse respeito, o Backbone não se adapta nem ao MVC nem ao MVP. Em vez disso, ele pega emprestado alguns dos melhores conceitos de vários padrões arquitetônicos e cria uma estrutura flexível que funciona bem.

Ele *é* no entanto vale a pena entender onde e por que esses conceitos se originou, por isso, espero que minhas declarações de MVC e MVP ter sido de ajuda. Chame- **o de Backbone way** , MV \* ou o que quer que ajude a referenciar seu sabor de arquitetura de aplicativo. A maioria dos frameworks JavaScript estruturais adotarão sua própria visão dos padrões clássicos, intencionalmente ou por acidente, mas o importante é que eles nos ajudam a desenvolver aplicativos que são organizados, limpos e podem ser facilmente mantidos.

MVVM

MVVM (Model View ViewModel) é um padrão de arquitetura baseado em MVC e MVP, que tenta separar mais claramente o desenvolvimento de interfaces de usuário (UI) daquele da lógica de negócios e comportamento em um aplicativo. Para esse fim, muitas implementações desse padrão fazem uso de ligações de dados declarativas para permitir uma separação do trabalho em Visualizações de outras camadas.

Isso facilita a IU e o trabalho de desenvolvimento que ocorrem quase simultaneamente na mesma base de código. Os desenvolvedores de IU gravam ligações para o ViewModel dentro de sua marcação de documento (HTML), onde o Model e o ViewModel são mantidos por desenvolvedores que trabalham na lógica do aplicativo.

História

MVVM (pelo nome) foi originalmente definido pela Microsoft para uso com Windows Presentation Foundation ( [WPF](https://en.wikipedia.org/wiki/Windows_Presentation_Foundation) ) e [Silverlight](https://www.microsoft.com/silverlight/) , tendo sido anunciado oficialmente em 2005 por [John Grossman](https://blogs.msdn.com/b/johngossman/) em uma postagem de blog sobre Avalon (o codinome do WPF). Ele também encontrou alguma popularidade na comunidade Adobe Flex como uma alternativa para simplesmente usar MVC.

Antes da Microsoft adotar o nome MVVM, havia, no entanto, um movimento na comunidade para ir de MVP para MVPM: Model View [PresentationModel](https://blogs.adobe.com/paulw/archives/2007/10/presentation_pa_3.html) . Martin Fowler escreveu um [artigo](http://martinfowler.com/eaaDev/PresentationModel.html) sobre PresentationModels em 2004 para aqueles interessados ​​em ler mais sobre ele. A ideia de um modelo de [apresentação](http://blogs.infragistics.com/blogs/craig_shoemaker/archive/2009/11/03/learning-model-view-viewmodel-and-presentation-model.aspx) já existia há muito mais tempo do que este artigo, no entanto, foi considerada a grande inovação da ideia e ajudou muito a popularizá-la.

Houve um grande alvoroço nos círculos "alt.net" depois que a Microsoft anunciou o MVVM como uma alternativa ao MVPM. Muitos alegaram que o domínio da empresa no mundo da GUI estava lhes dando a oportunidade de dominar a comunidade como um todo, renomeando conceitos existentes como bem lhes aprouver para fins de marketing. Uma multidão progressiva reconheceu que, embora MVVM e MVPM fossem efetivamente a mesma ideia, eles vieram em pacotes ligeiramente diferentes.

Nos últimos anos, o MVVM foi implementado em JavaScript na forma de frameworks estruturais como [KnockoutJS](http://knockoutjs.com/) , [Kendo MVVM](http://www.kendoui.com/web/roadmap.aspx) e [Knockback.js](https://github.com/kmalakoff/knockback) , com uma resposta geral positiva da comunidade.

Vamos agora revisar os três componentes que compõem o MVVM.

Modelo

Como acontece com outros membros da família MV \*, o Modelo em MVVM representa dados ou informações específicas do domínio com as quais nosso aplicativo trabalhará. Um exemplo típico de dados específicos de domínio pode ser uma conta de usuário (por exemplo, nome, avatar, e-mail) ou uma faixa de música (por exemplo, título, ano, álbum).

Os modelos contêm informações, mas normalmente não lidam com o comportamento. Eles não formatam informações ou influenciam como os dados aparecem no navegador, pois isso não é responsabilidade deles. Em vez disso, a formatação dos dados é tratada pelo View, enquanto o comportamento é considerado uma lógica de negócios que deve ser encapsulada em outra camada que interage com o Model - o ViewModel.

A única exceção a esta regra tende a ser a validação e é considerada aceitável para Modelos validar dados sendo usados ​​para definir ou atualizar modelos existentes (por exemplo, um endereço de e-mail sendo inserido atende aos requisitos de uma expressão regular particular?).

No KnockoutJS, os Modelos se enquadram na definição acima, mas geralmente fazem chamadas Ajax para um serviço do lado do servidor para ler e gravar dados do Modelo.

Se estivéssemos construindo um aplicativo Todo simples, um modelo KnockoutJS representando um único item Todo poderia ter a seguinte aparência:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5 | var Todo = function ( content, done ) {      this.content = ko.observable(content);      this.done = ko.observable(done);      this.editing = ko.observable(false);  }; |

Nota: Pode-se notar no snippet acima que estamos chamando o método observable()no namespace KnockoutJS ko. No KnockoutJS, os observáveis ​​são objetos JavaScript especiais que podem notificar os assinantes sobre as alterações e detectar dependências automaticamente. Isso nos permite sincronizar Models e ViewModels quando o valor de um atributo Model é modificado.

Visão

Assim como acontece com o MVC, a Visualização é a única parte do aplicativo com a qual os usuários realmente interagem. Eles são uma IU interativa que representa o estado de um ViewModel. Nesse sentido, a visualização é considerada ativa em vez de passiva, mas isso também é verdadeiro para visualizações em MVC e MVP. Em MVC, MVP e MVVM, uma visualização também pode ser passiva, mas o que isso significa?

Uma visualização passiva apenas exibe uma exibição e não aceita nenhuma entrada do usuário.

Essa visualização também pode não ter nenhum conhecimento real dos modelos em nosso aplicativo e pode ser manipulada por um apresentador. A visão ativa do MVVM contém as ligações de dados, eventos e comportamentos que requerem uma compreensão do ViewModel. Embora esses comportamentos possam ser mapeados para propriedades, o View ainda é responsável por tratar os eventos do ViewModel.

É importante lembrar que o View não é responsável aqui por manipular o estado - ele o mantém em sincronia com o ViewModel.

Um KnockoutJS View é simplesmente um documento HTML com ligações declarativas para vinculá-lo ao ViewModel. O KnockoutJS Views exibe informações do ViewModel, passa comandos para ele (por exemplo, um usuário clicando em um elemento) e atualiza conforme o estado do ViewModel muda. Os modelos que geram marcação usando dados do ViewModel também podem ser usados ​​para essa finalidade.

Para dar um breve exemplo inicial, podemos olhar para a estrutura JavaScript MVVM KnockoutJS para saber como ela permite a definição de um ViewModel e suas ligações relacionadas na marcação:

ViewModel:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4 | var aViewModel = {      contactName: ko.observable("John")  };  ko.applyBindings(aViewModel); |

Visão:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5 | <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> |

Nossa caixa de texto de entrada (fonte) obtém seu valor inicial de contactName, atualizando automaticamente este valor sempre que contactName muda. Como a vinculação de dados é bidirecional, digitar na caixa de texto será atualizado de contactNameacordo, de modo que os valores estão sempre sincronizados.

Embora a implementação seja específica para KnockoutJS, o <div>contendo a mensagem "Você tem um nome muito longo!" o texto também contém validação simples (mais uma vez na forma de vinculações de dados). Se a entrada exceder 10 caracteres, ele será exibido, caso contrário, permanecerá oculto.

Passando para um exemplo mais avançado, podemos retornar ao nosso aplicativo Todo. Uma visualização KnockoutJS reduzida para isso, incluindo todas as ligações de dados necessárias, pode ter a seguinte aparência.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | <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: editing, event: { blur: $root.stopEditing }"/>              </li>            </ul>        </section>  </div> |

Observe que o layout básico da marcação é relativamente simples, contendo uma caixa de texto de entrada ( new-todo) para adicionar novos itens, alternadores para marcar itens como completos e uma lista ( todo-list) com um modelo para um item Todo na forma de um li.

As ligações de dados na marcação acima podem ser divididas da seguinte forma:

* A caixa de texto de entrada new-todotem uma vinculação de dados para a currentpropriedade, que é onde o valor do item atual sendo adicionado é armazenado. Nosso ViewModel (mostrado em breve) observa a currentpropriedade e também tem um vínculo com o addevento. Quando a tecla Enter é pressionada, o addevento é acionado e nosso ViewModel pode cortar o valor de currente adicioná-lo à lista de tarefas conforme necessário
* A caixa de seleção de entrada toggle-allpode marcar todos os itens atuais como concluídos, se clicada. Se marcado, ele dispara o allCompletedevento, que pode ser visto em nosso ViewModel
* O item litem a classe done. Quando uma tarefa é marcada como concluída, a classe CSS editingé marcada de acordo. Se clicar duas vezes no item, o $root.editItemretorno de chamada será executado
* A caixa de seleção com a classe togglemostra o estado da donepropriedade
* Um rótulo contém o valor do texto do item Todo ( content)
* Há também um botão de remoção que chamará o $root.removeretorno de chamada quando clicado.
* Uma caixa de texto de entrada usada para o modo de edição também contém o valor do item Todo content. O enterKeyevento irá definir a editingpropriedade como verdadeira ou falsa

ViewModel

O ViewModel pode ser considerado um controlador especializado que atua como um conversor de dados. Ele muda as informações do modelo em informações da vista, passando comandos da vista para o modelo.

Por exemplo, vamos imaginar que temos um modelo contendo um atributo de data em formato unix (por exemplo, 1333832407). Em vez de nossos modelos estarem cientes da visão de um usuário da data (por exemplo, 04/07/2012 às 17:00), onde seria necessário converter o atributo para seu formato de exibição, nosso modelo simplesmente mantém o formato bruto dos dados . Nosso View contém a data formatada e nosso ViewModel atua como um intermediário entre os dois.

Nesse sentido, o ViewModel pode ser visto mais como um Model do que como uma View, mas ele lida com a maior parte da lógica de exibição da View. O ViewModel também pode expor métodos para ajudar a manter o estado da View, atualizar o modelo com base nas ações em uma View e acionar eventos na View.

Em resumo, o ViewModel fica por trás de nossa camada de IU. Ele expõe os dados necessários para uma Visualização (de um Modelo) e pode ser visto como a fonte para a qual nossas Visualizações vão para dados e ações.

KnockoutJS interpreta o ViewModel como a representação de dados e operações que podem ser realizadas em uma IU. Esta não é a própria IU nem o modelo de dados que persiste, mas sim uma camada que também pode conter os dados que ainda não foram salvos com os quais um usuário está trabalhando. Os ViewModels do Knockout são objetos JavaScript implementados sem nenhum conhecimento de marcação HTML. Essa abordagem abstrata para sua implementação permite que eles permaneçam simples, o que significa que comportamentos mais complexos podem ser gerenciados mais facilmente no topo, conforme necessário.

Um KnockoutJS ViewModel parcial para nosso aplicativo Todo poderia ter a seguinte aparência:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57 | // 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 );      };   .. |

Acima, estamos basicamente fornecendo os métodos necessários para adicionar, editar ou remover itens, bem como a lógica para marcar todos os itens restantes como concluídos. Nota: A única diferença real digna de nota dos exemplos anteriores em nosso ViewModel são os arrays observáveis. Em KnockoutJS, se quisermos detectar e responder às mudanças em um único objeto, usaríamos observables. Se, no entanto, quisermos detectar e responder às mudanças de uma coleção de coisas, podemos usar um observableArray. Um exemplo mais simples de como usar matrizes observáveis ​​pode ser o seguinte:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5 | // 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' ); |

Nota: O aplicativo Knockout.js Todo completo que analisamos acima pode ser [obtido](http://todomvc.com/) em [TodoMVC](http://todomvc.com/) se estiver interessado.

Recapitulação: The View e ViewModel

Views e ViewModels se comunicam usando data-bindings e eventos. Como vimos em nosso exemplo inicial de ViewModel, o ViewModel não apenas expõe atributos de Model, mas também acessa outros métodos e recursos, como validação.

Nossos Views tratam de seus próprios eventos de interface de usuário, mapeando-os para o ViewModel conforme necessário. Modelos e atributos no ViewModel são sincronizados e atualizados por meio de vinculação de dados bidirecional.

Triggers (data-triggers) também nos permitem reagir melhor às mudanças no estado de nossos atributos de Model.

Recapitulação: o ViewModel e o modelo

Embora possa parecer que ViewModel é totalmente responsável pelo Modelo no MVVM, existem algumas sutilezas com esse relacionamento que vale a pena observar. O ViewModel pode expor um Model ou atributos de Model para fins de vinculação de dados e também pode conter interfaces para buscar e manipular propriedades expostas na visualização.

Prós e contras

Esperamos agora ter uma avaliação melhor do que é MVVM e como funciona. Vamos agora revisar as vantagens e desvantagens de empregar o padrão:

Vantagens

* MVVM Facilita o desenvolvimento paralelo mais fácil de uma IU e os blocos de construção que a alimentam
* Abstrai a visão e, portanto, reduz a quantidade de lógica de negócios (ou cola) necessária no código por trás dela
* O ViewModel pode ser mais fácil de testar a unidade do que o código orientado a eventos
* O ViewModel (sendo mais Model do que View) pode ser testado sem preocupações de automação e interação da IU

Desvantagens

* Para interfaces de usuário mais simples, o MVVM pode ser um exagero
* Embora as vinculações de dados possam ser declarativas e agradáveis ​​de se trabalhar, elas podem ser mais difíceis de depurar do que o código imperativo, onde simplesmente definimos pontos de interrupção
* As vinculações de dados em aplicativos não triviais podem criar muitos registros contábeis. Também não queremos acabar em uma situação em que as ligações são mais pesadas do que os objetos que estão sendo ligados
* Em aplicativos maiores, pode ser mais difícil projetar o ViewModel antecipadamente para obter a quantidade necessária de generalização

MVVM com ligações de dados mais flexíveis

Não é incomum para desenvolvedores JavaScript com experiência em MVC ou MVP revisar o MVVM e reclamar de sua verdadeira separação de interesses. Ou seja, a quantidade de vinculações de dados embutidas mantidas na marcação HTML de uma Visualização.

Devo admitir que, quando analisei pela primeira vez as implementações do MVVM (por exemplo, KnockoutJS, Knockback), fiquei surpreso que qualquer desenvolvedor gostaria de voltar aos dias de antigamente, onde misturávamos lógica (JavaScript) com nossa marcação e descobrimos que era rapidamente impossível de manter. A realidade, entretanto, é que o MVVM faz isso por uma série de boas razões (que cobrimos), incluindo facilitar os designers a se vincularem mais facilmente à lógica de sua marcação.

Para os puristas entre nós, você ficará feliz em saber que agora também podemos reduzir muito a nossa dependência de vinculações de dados graças a um recurso conhecido como provedores de vinculação customizada, introduzido no KnockoutJS 1.3 e disponível em todas as versões desde então.

KnockoutJS por padrão tem um provedor de vinculação de dados que procura por quaisquer elementos com data-bindatributos como no exemplo abaixo.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1 | <input id="new-todo" type="text" data-bind="value: current, valueUpdate: 'afterkeydown', enterKey: add" placeholder="What needs to be done?"/> |

Quando o provedor localiza um elemento com este atributo, ele o analisa e o transforma em um objeto de ligação usando o contexto de dados atual. Esta é a maneira que KnockoutJS sempre funcionou, permitindo-nos adicionar associações declarativamente a elementos que KnockoutJS vincula aos dados nessa camada.

Uma vez que começamos a construir visualizações que não são mais triviais, podemos acabar com um grande número de elementos e atributos cujas ligações na marcação podem se tornar difíceis de gerenciar. Com os provedores de ligação customizada, no entanto, isso não é mais um problema.

Um provedor de vinculação está interessado principalmente em duas coisas:

* Quando dado um nó DOM, ele contém alguma vinculação de dados?
* Se o nó passou nesta primeira pergunta, como o objeto de ligação se parece no contexto de dados atual?

Provedores de vinculação implementam duas funções:

* nodeHasBindings: isso leva em um nó DOM que não necessariamente tem que ser um elemento
* getBindings: retorna um objeto que representa as ligações aplicadas ao contexto de dados atual

Um provedor de vinculação de esqueleto pode, portanto, ter a seguinte aparência:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | var ourBindingProvider = {    nodeHasBindings: function( node ) {        // returns true/false    },      getBindings: function( node, bindingContext ) {        // returns a binding object    }  }; |

Antes de detalharmos esse provedor, vamos discutir brevemente a lógica nos atributos de vinculação de dados.

Se, ao usar o MVVM do Knockout, ficarmos insatisfeitos com a ideia de a lógica do aplicativo estar excessivamente vinculada à sua visualização, podemos mudar isso. Poderíamos implementar algo um pouco como classes CSS para atribuir ligações por nome a elementos. Ryan Niemeyer (de knockmeout.net) sugeriu anteriormente usar data-classpara isso para evitar confundir classes de apresentação com classes de dados, então vamos colocar nossa nodeHasBindingsfunção de suporte para isso:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4 | // does an element have any bindings?  function nodeHasBindings( node ) {      return node.getAttribute ? node.getAttribute("data-class") : false;  }; |

Em seguida, precisamos de uma getBindings()função sensata . Como estamos aderindo à ideia de classes CSS, por que não considerar também o suporte a classes separadas por espaço para nos permitir compartilhar especificações de vinculação entre diferentes elementos?

Vamos primeiro revisar como serão as nossas ligações. Criamos um objeto para mantê-los onde nossos nomes de propriedade precisam corresponder às chaves que desejamos usar em nossas classes de dados.

Nota: Não é necessário muito trabalho para converter um aplicativo KnockoutJS do uso de vinculações de dados tradicionais em vinculações discretas com provedores de vinculação personalizados. Simplesmente extraímos todos os nossos atributos de vinculação de dados, os substituímos por atributos de classe de dados e colocamos nossas vinculações em um objeto de vinculação conforme a seguir:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97 | 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          }      };        .... |

No entanto, há duas linhas faltando no trecho acima - ainda precisamos de nossa getBindingsfunção, que fará um loop por cada uma das chaves em nossos atributos de classe de dados e construirá o objeto resultante de cada um deles. Se detectarmos que o objeto de ligação é uma função, o chamaremos com nossos dados atuais usando o contexto this. Nosso provedor completo de ligação personalizada seria o seguinte:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | // 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) : bindingAccessor;                     ko.utils.extend(result, binding);                 }                }          }            return result;      };  }; |

Assim, as poucas linhas finais do nosso bindingsobjeto podem ser definidas da seguinte forma:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | // 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 );    })(); |

O que estamos fazendo aqui é definir efetivamente o construtor para nosso manipulador de vinculação que aceita um objeto (vinculações) que usamos para pesquisar nossas vinculações. Poderíamos então reescrever a marcação para nosso aplicativo View usando classes de dados da seguinte maneira:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | <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</span>              </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> |

Neil Kerkin montou um aplicativo de demonstração TodoMVC completo usando o acima, que pode ser acessado e usado [aqui](https://jsfiddle.net/nkerkin/hmq7D/light/) .

Embora possa parecer muito trabalhoso na explicação acima, agora que temos um getBindingsmétodo genérico escrito, é muito mais trivial simplesmente reutilizá-lo e usar classes de dados em vez de vinculações de dados estritas para escrever nosso KnockoutJS aplicativos em vez disso. O resultado líquido é, esperançosamente, uma marcação mais limpa com nossas vinculações de dados sendo deslocadas da Visualização para um objeto de vinculações.

MVC vs. MVP vs. MVVM

Tanto o MVP quanto o MVVM são derivados do MVC. A principal diferença entre ele e seus derivados é a dependência que cada camada tem das outras camadas, bem como o grau de vinculação entre elas.

No MVC, o View fica no topo de nossa arquitetura com o controlador ao lado dele. Os modelos ficam abaixo do controlador e, portanto, nossas visualizações sabem sobre nossos controladores e os controladores sabem sobre os modelos. Aqui, nossas Visualizações têm acesso direto aos Modelos. Expor o Modelo completo à Visualização, entretanto, pode ter custos de segurança e desempenho, dependendo da complexidade de nossa aplicação. O MVVM tenta evitar esses problemas.

No MVP, a função do controlador é substituída por um apresentador. Os apresentadores se sentam no mesmo nível que as visualizações, ouvindo os eventos da visualização e do modelo e mediando as ações entre eles. Ao contrário do MVVM, não há um mecanismo para vincular Views a ViewModels, então, em vez disso, contamos com cada View implementando uma interface permitindo que o Presenter interaja com a View.

Consequentemente, o MVVM nos permite criar subconjuntos específicos de Visualização de um Modelo que podem conter informações de estado e lógica, evitando a necessidade de expor todo o Modelo a uma Visualização. Ao contrário do Presenter do MVP, um ViewModel não é necessário para fazer referência a uma View. O View pode se ligar a propriedades no ViewModel que por sua vez expõe os dados contidos em Models para o View. Como mencionamos, a abstração da View significa que há menos lógica necessária no código por trás dela.

No entanto, uma das desvantagens disso é que um nível de interpretação é necessário entre ViewModel e View e isso pode ter custos de desempenho. A complexidade dessa interpretação também pode variar - pode ser tão simples quanto copiar dados ou tão complexa quanto manipulá-los em uma forma que gostaríamos que a Visualização visse. O MVC não tem esse problema, pois todo o modelo está prontamente disponível e tal manipulação pode ser evitada.

Backbone.js vs. KnockoutJS

Entender as diferenças sutis entre MVC, MVP e MVVM é importante, mas os desenvolvedores perguntarão se devem considerar o uso de KnockoutJS em vez de Backbone com base no que aprendemos. As notas a seguir podem ser úteis aqui:

* Ambas as bibliotecas são projetadas com objetivos diferentes em mente e muitas vezes não é tão simples quanto escolher MVC ou MVVM
* Se a vinculação de dados e a comunicação bidirecional são suas principais preocupações, KnockoutJS é definitivamente o caminho a seguir. Praticamente qualquer atributo ou valor armazenado em nós DOM pode ser mapeado para objetos JavaScript com essa abordagem.
* O Backbone se destaca pela facilidade de integração com os serviços RESTful, enquanto os modelos KnockoutJS são simplesmente objetos JavaScript e o código necessário para atualizar o modelo deve ser escrito pelo desenvolvedor.
* KnockoutJS tem como foco a automação de vinculações de IU, o que requer um código customizado muito mais detalhado se tentar fazer isso com o Backbone. Isso não é um problema com o Backbone em si, pois ele tenta propositalmente ficar fora da IU. No entanto, o Knockback tenta ajudar com esse problema.
* Com KnockoutJS, podemos vincular nossas próprias funções aos observáveis ​​do ViewModel, que são executados a qualquer momento nas mudanças observáveis. Isso nos permite o mesmo nível de flexibilidade que pode ser encontrado no Backbone
* O Backbone possui uma solução de roteamento sólida integrada, enquanto o KnockoutJS não oferece opções de roteamento prontas para o uso. No entanto, pode-se facilmente preencher esse comportamento, se necessário, usando o [plugin BBQ de](http://benalman.com/projects/jquery-bbq-plugin/) Ben Alman ou um sistema de roteamento autônomo como o excelente [Crossroads de](https://millermedeiros.github.com/crossroads.js/) Miller Medeiros  .

Para concluir, eu pessoalmente acho o KnockoutJS mais adequado para aplicativos menores, enquanto o conjunto de recursos do Backbone realmente brilha ao construir qualquer coisa não trivial. Dito isso, muitos desenvolvedores têm usado ambas as estruturas para escrever aplicativos de complexidade variável e eu recomendo tentar ambos em uma escala menor antes de tomar uma decisão sobre qual pode funcionar melhor para seu projeto.

**Para ler mais sobre MVVM ou Knockout, recomendo os seguintes artigos:**

* [As vantagens do MVVM](http://www.silverlightshow.net/news/The-Advantages-of-MVVM.aspx)
* [SO: Quais são os problemas com o MVVM?](https://stackoverflow.com/questions/883895/what-are-the-problems-of-the-mvvm-pattern)
* [MVVM explicado](http://www.codeproject.com/Articles/100175/Model-View-ViewModel-MVVM-Explained)
* [Como o MVVM se compara ao MVC?](https://www.quora.com/Pros-and-cons-of-MVVM-framework-and-how-I-can-campare-it-with-MVC)
* [Vinculações personalizadas em KnockoutJS](http://www.knockmeout.net/2011/09/ko-13-preview-part-2-custom-binding.html)
* [Explorando Knockout com TodoMVC](https://gratdevel.blogspot.co.uk/2012/02/exploring-todomvc-and-knockoutjs-with.html)

Modern Modular JavaScript Design Patterns

A importância dos aplicativos de desacoplamento

No mundo do JavaScript escalável, quando dizemos que um aplicativo é **modular** , geralmente queremos dizer que ele é composto de um conjunto de peças de funcionalidade distintas e altamente desacopladas armazenadas em módulos. [O acoplamento fraco](http://arguments.callee.info/2009/05/18/javascript-design-patterns--mediator/) facilita a manutenção de aplicativos, removendo *dependências* onde possível. Quando isso é implementado de forma eficiente, é muito fácil ver como as alterações em uma parte do sistema podem afetar outra.

Ao contrário de algumas linguagens de programação mais tradicionais, no entanto, a iteração atual do JavaScript ( [ECMA-262](https://www.ecma-international.org/publications/standards/Ecma-262.htm) ) não fornece aos desenvolvedores os meios para importar esses módulos de código de maneira limpa e organizada. É uma das preocupações com as especificações que não exigiram muita reflexão até os anos mais recentes, quando a necessidade de aplicativos JavaScript mais organizados se tornou aparente.

Em vez disso, os desenvolvedores no momento precisam recorrer a variações dos padrões [literais de](http://rmurphey.com/blog/2009/10/15/using-objects-to-organize-your-code/)[módulo](http://www.adequatelygood.com/2010/3/JavaScript-Module-Pattern-In-Depth) ou [objeto](http://rmurphey.com/blog/2009/10/15/using-objects-to-organize-your-code/) , que abordamos anteriormente neste livro. Com muitos deles, os scripts de módulo são agrupados no DOM com namespaces sendo descritos por um único objeto global onde ainda é possível incorrer em colisões de nomenclatura em nossa arquitetura. Também não há uma maneira limpa de lidar com o gerenciamento de dependências sem algum esforço manual ou ferramentas de terceiros.

Embora soluções nativas para esses problemas cheguem ao [ES Harmony](http://wiki.ecmascript.org/doku.php?id=harmony:modules) (provavelmente a próxima versão do JavaScript), a boa notícia é que escrever JavaScript modular nunca foi tão fácil e podemos começar a fazê-lo hoje.

Nesta seção, veremos três formatos para escrever JavaScript modular: **AMD** , **CommonJS** e propostas para a próxima versão de JavaScript, **Harmony** .

Uma nota sobre carregadores de script

É difícil discutir os módulos AMD e CommonJS sem falar sobre o elefante na sala - [carregadores de script](https://msdn.microsoft.com/en-us/scriptjunkie/hh227261) . No momento em que escrevo este livro, o carregamento de script é um meio para atingir um objetivo, esse objetivo é o JavaScript modular que pode ser usado em aplicativos hoje - para isso, infelizmente, o uso de um carregador de script compatível é necessário. Para obter o máximo desta seção, recomendo obter uma **compreensão básica** de como as ferramentas populares de carregamento de script funcionam para que as explicações dos formatos de módulo façam sentido no contexto.

Há um grande número de carregadores para módulo de manipulação de carga nos formatos AMD e commonjs, mas as minhas preferências pessoais são [RequireJS](http://requirejs.org/) e [curl.js](https://github.com/unscriptable/curl) . Tutoriais completos sobre essas ferramentas estão fora do escopo deste livro, mas posso recomendar a leitura do artigo de John Hann sobre [curl.js](http://unscriptable.com/index.php/2011/03/30/curl-js-yet-another-amd-loader/) e da documentação da API [RequireJS de](http://requirejs.org/docs/api.html) James Burke para mais informações.

De uma perspectiva de produção, o uso de ferramentas de otimização (como o otimizador RequireJS) para concatenar scripts é recomendado para implantação ao trabalhar com esses módulos. Curiosamente, com o shim [Almond](https://github.com/jrburke/almond) AMD, o RequireJS não precisa ser implementado no site implantado e o que se pode considerar um carregador de script pode ser facilmente deslocado para fora do desenvolvimento.

Dito isso, James Burke provavelmente diria que a capacidade de carregar scripts dinamicamente após o carregamento da página ainda tem seus casos de uso e o RequireJS também pode ajudar nisso. Com essas notas em mente, vamos começar.

AMD

Um formato para escrever JavaScript modular no navegador

A meta geral para o formato AMD (Asynchronous Module Definition) é fornecer uma solução para JavaScript modular que os desenvolvedores possam usar hoje. Ele nasceu da experiência do mundo real do Dojo usando XHR + eval e os proponentes desse formato queriam evitar que qualquer solução futura sofresse com as fraquezas do passado.

O próprio formato do módulo AMD é uma proposta para definir módulos onde o módulo e as dependências podem ser carregados de [forma assíncrona](http://dictionary.reference.com/browse/asynchronous) . Ele tem uma série de vantagens distintas, incluindo ser assíncrono e altamente flexível por natureza, o que remove o forte acoplamento que pode ser comumente encontrado entre o código e a identidade do módulo. Muitos desenvolvedores gostam de usá-lo e pode-se considerá-lo um trampolim confiável para o [sistema de módulos](http://wiki.ecmascript.org/doku.php?id=harmony:modules) proposto para o ES Harmony.

A AMD começou como um rascunho de especificação para um formato de módulo na lista CommonJS, mas como não foi capaz de chegar a um consenso completo, o desenvolvimento do formato foi transferido para o grupo [amdjs](https://github.com/amdjs) .

Hoje ele é adotado por projetos como Dojo, MooTools, Firebug e até mesmo jQuery. Embora o termo *formato CommonJS AMD* tenha sido visto à solta na ocasião, é melhor referir-se a ele apenas como suporte a AMD ou Módulo Async, já que nem todos os participantes da lista CommonJS desejaram segui-lo.

**Nota:** Houve um tempo em que a proposta era chamada de Modules Transport / C, no entanto, como a especificação não era voltada para o transporte de módulos CommonJS existentes, mas - para definir módulos - fazia mais sentido optar pela convenção de nomenclatura AMD .

Introdução aos módulos

Os dois primeiros conceitos dignos de nota sobre o AMD são a ideia de um definemétodo para facilitar a definição do módulo e um requiremétodo para lidar com o carregamento de dependência. *define* é usado para definir módulos nomeados ou não com base na seguinte assinatura:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5 | define(      module\_id /\*optional\*/,      [dependencies] /\*optional\*/,      definition function /\*function for instantiating the module or object\*/  ); |

Como podemos dizer pelos comentários inline, o module\_idé um argumento opcional que normalmente só é necessário quando ferramentas de concatenação não-AMD estão sendo usadas (pode haver alguns outros casos extremos em que também é útil). Quando esse argumento é omitido, nos referimos ao módulo como *anônimo* .

Ao trabalhar com módulos anônimos, a ideia da identidade de um módulo é DRY, tornando-se trivial para evitar a duplicação de nomes de arquivos e código. Como o código é mais portátil, ele pode ser facilmente movido para outros locais (ou ao redor do sistema de arquivos) sem a necessidade de alterar o próprio código ou alterar seu ID de módulo. Considere o module\_idsemelhante ao conceito de caminhos de pasta.

Nota: Os desenvolvedores podem executar este mesmo código em vários ambientes usando apenas um AMD otimizador que funciona com um ambiente commonjs como [r.js](https://github.com/jrburke/r.js/) .

De volta à defineassinatura, o argumento dependencies representa um array de dependências que são exigidas pelo módulo que estamos definindo e o terceiro argumento ("função de definição" ou "função de fábrica") é uma função executada para instanciar nosso módulo. Um módulo de osso descoberto pode ser definido da seguinte forma:

Compreendendo a AMD: define ()

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | // 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 ) );              }          };  }); |

por outro lado, o *require* é normalmente usado para carregar o código em um arquivo JavaScript de nível superior ou dentro de um módulo, caso desejemos buscar dependências dinamicamente. Um exemplo de seu uso é:

Compreendendo a AMD: require ()

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // 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();  }); |

Dependências carregadas dinamicamente

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | 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      };  }); |

Compreendendo a AMD: plug-ins

A seguir está um exemplo de definição de um plug-in compatível com AMD:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | // 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      }    }); |

**Nota:** embora css! está incluído para carregar dependências CSS no exemplo acima, é importante lembrar que essa abordagem tem algumas ressalvas, como não ser totalmente possível estabelecer quando o CSS está totalmente carregado. Dependendo de como abordamos nosso processo de construção, isso também pode resultar na inclusão de CSS como uma dependência no arquivo otimizado, portanto, use CSS como uma dependência carregada em tais casos com cuidado. Se estiver interessado em fazer o acima, podemos também explorar @ RequireJS CSS do VIISON plugins mais [aqui](https://github.com/VIISON/RequireCSS) .

Carregando módulos AMD usando RequireJS

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | require(["app/myModule"],        function( myModule ){          // start the main module which in-turn          // loads other modules          var module = new myModule();          module.doStuff();  }); |

Este exemplo poderia simplesmente ser visto como o requirejs(["app/myModule"], function(){})que indica que os globais de nível superior do carregador estão sendo usados. É assim que inicia o carregamento de nível superior de módulos com diferentes carregadores AMD, porém com uma define()função, se for passado um require([])requerimento local, todos os exemplos se aplicam a ambos os tipos de carregadores (curl.js e RequireJS).

Carregando módulos AMD usando curl.js

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | curl(["app/myModule.js"],        function( myModule ){          // start the main module which in-turn          // loads other modules          var module = new myModule();          module.doStuff();    }); |

Módulos com dependências diferidas

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | // 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();  }); |

Módulos AMD com Dojo

Definir módulos compatíveis com AMD usando Dojo é bastante simples. Conforme acima, defina quaisquer dependências de módulo em uma matriz como o primeiro argumento e forneça um retorno de chamada (fábrica) que executará o módulo assim que as dependências forem carregadas. por exemplo:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | define(["dijit/Tooltip"], function( Tooltip ){        //Our dijit tooltip is now available for local use      new Tooltip(...);    }); |

Observe a natureza anônima do módulo, que agora pode ser consumido por um carregador assíncrono Dojo, RequireJS ou o carregador de módulo [dojo.require ()](http://livedocs.dojotoolkit.org/dojo/require) padrão.

Existem algumas dicas interessantes com referência a módulos que são úteis saber aqui. Embora a maneira defendida pela AMD de referenciar módulos os declare na lista de dependências com um conjunto de argumentos correspondentes, isso não é suportado pelo sistema de compilação Dojo 1.6 mais antigo - ele realmente só funciona para carregadores compatíveis com AMD. por exemplo:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | define(["dojo/cookie", "dijit/Tooltip"], function( cookie, Tooltip ){        var cookieValue = cookie( "cookieName" );      new Tooltip(...);    }); |

Isso tem muitas vantagens sobre o namespacing aninhado, pois os módulos não precisam mais fazer referência direta a namespaces completos todas as vezes - tudo o que precisamos é o caminho "dojo / cookie" nas dependências, que uma vez com alias para um argumento, pode ser referenciado por essa variável. Isso elimina a necessidade de digitar repetidamente "dojo". em nossas aplicações.

A pegadinha final a ter em conta é que se desejarmos continuar usando o sistema de compilação Dojo mais antigo ou se desejarmos migrar módulos mais antigos para este estilo AMD mais recente, a versão mais detalhada a seguir permite uma migração mais fácil. Observe que dojo e dijit e também referenciados como dependências:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4 | define(["dojo", "dijit', "dojo/cookie", "dijit/Tooltip"], function( dojo, dijit ){      var cookieValue = dojo.cookie( "cookieName" );      new dijit.Tooltip(...);  }); |

AMD Module Design Patterns (Dojo)

Como vimos nas seções anteriores, os padrões de design podem ser altamente eficazes para melhorar a forma como abordamos soluções de estruturação para problemas comuns de desenvolvimento. [John Hann](https://twitter.com/unscriptable) deu algumas apresentações excelentes sobre os padrões de design do módulo AMD cobrindo o Singleton, Decorator, Mediator e outros e eu recomendo fortemente verificar seus [slides](http://unscriptable.com/code/AMD-module-patterns/) se tivermos uma chance.

Uma seleção de padrões de design AMD pode ser encontrada abaixo.

**Padrão do decorador:**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | // mylib/UpdatableObservable: a Decorator for dojo/store/Observable  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;      };  });      // Decorator consumer  // a consumer for mylib/UpdatableObservable    define(["mylib/UpdatableObservable"], function ( makeUpdatable ) {      var observable,          updatable,          someItem;        // make the observable store updatable      updatable = makeUpdatable( observable ); // `new` is optional!        // we can then call .updated() later on if we wish to pass      // on data that has changed      //updatable.updated( updatedItem );  }); |

**Padrão do adaptador**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | // "mylib/Array" adapts `each` function to mimic 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              });          }      });  });    // Adapter consumer  // "myapp/my-module":  define(["mylib/Array"], function ( array ) {      array.each( ["uno", "dos", "tres"], function ( i, esp ) {          // here, `this` == item      });  }); |

Módulos AMD com jQuery

Ao contrário do Dojo, o jQuery realmente vem com apenas um arquivo, no entanto, dada a natureza baseada em plug-in da biblioteca, podemos demonstrar como é simples definir um módulo AMD que o usa abaixo.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | define(["js/jquery.js","js/jquery.color.js","js/underscore.js"],        function( $, colorPlugin, \_ ){          // Here we've passed in jQuery, the color plugin and Underscore          // None of these will be accessible in the global scope, but we          // can easily reference them below.            // Pseudo-randomize an array of colors, selecting the first          // item in the shuffled array          var shuffleColor = \_.first( \_.shuffle( ["#666","#333","#111"] ) );            // Animate the background-color of any elements with the class          // "item" on the page using the shuffled color          $( ".item" ).animate( {"backgroundColor": shuffleColor } );            // What we return can be used by other modules          return {};      }); |

No entanto, há algo faltando neste exemplo e é o conceito de registro.

Registrando jQuery como um módulo compatível com Async

Um dos principais recursos que surgiram no jQuery 1.7 foi o suporte para registrar o jQuery como um módulo assíncrono. Existem vários carregadores de script compatíveis (incluindo RequireJS e curl) que são capazes de carregar módulos usando um formato de módulo assíncrono e isso significa que menos hacks são necessários para fazer as coisas funcionarem.

Se um desenvolvedor quiser usar AMD e não quiser que sua versão do jQuery vaze para o espaço global, ele deve chamar noConflictseu módulo de nível superior que usa jQuery. Além disso, como várias versões do jQuery podem estar em uma página, há considerações especiais que um carregador AMD deve considerar e, portanto, o jQuery só se registra com carregadores AMD que reconheceram essas preocupações, que são indicadas pela especificação do carregador define.amd.jQuery. RequireJS e curl são dois carregadores que fazem isso

O AMD nomeado fornece uma cobertura de segurança de ser robusto e seguro para a maioria dos casos de uso.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | // 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 () {};  }); |

Por que a AMD é a melhor opção para escrever JavaScript modular?

* Fornece uma proposta clara de como abordar a definição de módulos flexíveis.
* Significativamente mais limpo do que o atual namespace global e <script>soluções de tag em que muitos de nós confiamos. Existe uma maneira limpa de declarar módulos autônomos e dependências que eles possam ter.
* As definições dos módulos são encapsuladas, ajudando-nos a evitar a poluição do namespace global.
* Provavelmente funciona melhor do que algumas soluções alternativas (por exemplo, CommonJS, que veremos em breve). Ele não tem problemas com domínio cruzado, local ou depuração e não depende de ferramentas do lado do servidor para serem usadas. A maioria dos carregadores AMD oferece suporte a módulos de carregamento no navegador sem um processo de construção.
* Fornece uma abordagem de "transporte" para incluir vários módulos em um único arquivo. Outras abordagens, como o CommonJS, ainda não concordaram com um formato de transporte.
* É possível fazer o carregamento lento de scripts, se necessário.

**Nota:** Muitos dos itens acima podem ser ditos sobre a estratégia de carregamento de módulos da YUI também.

**Leitura Relacionada**

[O Guia RequireJS para AMD](http://requirejs.org/docs/whyamd.html)

[Qual é a maneira mais rápida de carregar módulos AMD?](http://unscriptable.com/index.php/2011/09/21/what-is-the-fastest-way-to-load-amd-modules/)

[AMD vs. CommonJS, qual é o melhor formato?](http://unscriptable.com/index.php/2011/09/30/amd-versus-cjs-whats-the-best-format/)

[AMD é melhor para a Web do que módulos CommonJS](http://blog.millermedeiros.com/2011/09/amd-is-better-for-the-web-than-commonjs-modules/)

[O futuro são módulos, não frameworks](http://unscriptable.com/code/Modules-Frameworks/)

[AMD não é mais uma especificação CommonJS](https://groups.google.com/group/commonjs/browse_thread/thread/96a0963bcb4ca78f/cf73db49ce267ce1?lnk=gst)

[Sobre a invenção de formatos de módulo JavaScript e carregadores de script](https://tagneto.blogspot.com/2011/04/on-inventing-js-module-formats-and.html)

[The AMD Mailing List](https://groups.google.com/group/amd-implement)

Quais carregadores de script e frameworks suportam AMD?

**No navegador:**

* RequireJS [http://requirejs.org](http://requirejs.org/)
* curl.js [http://github.com/unscriptable/curl](https://github.com/unscriptable/curl)
* bdLoad <http://bdframework.com/bdLoad>
* Yabble [http://github.com/jbrantly/yabble](https://github.com/jbrantly/yabble)
* PINF [http://github.com/pinf/loader-js](https://github.com/pinf/loader-js)
* (e mais)

**Lado do servidor:**

* RequireJS [http://requirejs.org](http://requirejs.org/)
* PINF [http://github.com/pinf/loader-js](https://github.com/pinf/loader-js)

Conclusões da AMD

Tendo usado a AMD para uma série de projetos, minhas conclusões são que ela preenche muitas das caixas de seleção que os desenvolvedores que criam aplicativos sérios podem desejar a partir de um formato de módulo melhor. Ele evita a necessidade de se preocupar com globais, suporta módulos nomeados, não requer transformação de servidor para funcionar e é um prazer de usar para gerenciamento de dependências.

É também uma excelente adição para o desenvolvimento modular usando Backbone.js, ember.js ou qualquer outra estrutura estrutural para manter os aplicativos organizados.

Como a AMD foi amplamente discutida por quase dois anos nos mundos Dojo e CommonJS, sabemos que teve tempo de amadurecer e evoluir. Também sabemos que foi testado em batalha por uma série de grandes empresas para construir aplicativos não triviais (IBM, BBC iPlayer) e, se não funcionasse, provavelmente já o teriam abandonado, mas não tenho.

Dito isso, ainda existem áreas onde a AMD poderia ser melhorada. Os desenvolvedores que usam o formato há algum tempo podem achar que o código padrão / wrapper da AMD é uma sobrecarga irritante. Embora eu compartilhe essa preocupação, existem ferramentas como o [Volo](https://github.com/volojs/volo) que podem ajudar a contornar esses problemas e eu diria que, no geral, os prós de usar a AMD superam os contras.

CommonJS

Um formato de módulo otimizado para o servidor

A proposta do módulo CommonJS especifica uma API simples para declarar módulos do lado do servidor e, ao contrário da AMD, tenta cobrir um conjunto mais amplo de questões, como io, sistema de arquivos, promessas e muito mais.

O formato foi proposto por [CommonJS](http://www.commonjs.org/) - um grupo de trabalho voluntário que visa projetar, prototipar e padronizar APIs JavaScript. Até o momento, eles tentaram ratificar padrões para [módulos](http://www.commonjs.org/specs/modules/1.0/) e [pacotes](http://wiki.commonjs.org/wiki/Packages/1.0) .

Começando

De uma perspectiva de estrutura, um módulo CommonJS é uma parte reutilizável de JavaScript que exporta objetos específicos disponibilizados para qualquer código dependente. Ao contrário da AMD, normalmente não há wrappers de função em torno de tais módulos (portanto, não veremos defineaqui, por exemplo).

Módulos CommonJS basicamente contêm duas partes primárias: uma variável livre chamada exportsque contém os objetos que um módulo deseja disponibilizar para outros módulos e uma requirefunção que os módulos podem usar para importar as exportações de outros módulos.

Noções básicas sobre CommonJS: require () e exportações

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | // package/lib is a dependency we require  var lib = require( "package/lib" );    // behaviour for our module  function foo(){      lib.log( "hello world!" );  }    // export (expose) foo to other modules  exports.foo = foo; |

Consumo básico das exportações

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | // define more behaviour we would like to expose  function foobar(){    this.foo = function(){      console.log( "Hello foo" );    }      this.bar = function(){      console.log( "Hello bar" );    }  }    // expose foobar to other modules  exports.foobar = foobar;    // an application consuming "foobar"    // access the module relative to the path  // where both usage and module files exist  // in the same directory    var foobar = require("./foobar").foobar,      test   = new foobar();    // Outputs: "Hello bar"  test.bar(); |

Equivalente em AMD do primeiro exemplo CommonJS

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | define(function(require){     var lib = require( "package/lib" );        // some behaviour for our module      function foo(){          lib.log( "hello world!" );      }        // export (expose) foo for other modules      return {          foobar: foo      };  }); |

Isso pode ser feito porque a AMD oferece suporte a um recurso de [empacotamento CommonJS simplificado](http://requirejs.org/docs/whyamd.html#sugar) .

Consumindo múltiplas dependências

**app.js**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | var modA = require( "./foo" );  var modB = require( "./bar" );    exports.app = function(){      console.log( "Im an application!" );  }    exports.foo = function(){      return modA.helloWorld();  } |

**bar.js**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1 | exports.name = "bar"; |

**foo.js**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4 | require( "./bar" );  exports.helloWorld = function(){      return "Hello World!!"  } |

Quais carregadores e frameworks suportam CommonJS?

**No navegador:**

* curl.js [http://github.com/unscriptable/curl](https://github.com/unscriptable/curl)
* SproutCore 1.1 [http://sproutcore.com](http://sproutcore.com/)
* PINF [http://github.com/pinf/loader-js](https://github.com/pinf/loader-js)

**Lado do servidor:**

* Node [http://nodejs.org](https://nodejs.org/)
* Narwhal <https://github.com/tlrobinson/narwhal>
* Persevere <http://www.persvr.org/>
* Wakanda <http://www.wakandasoft.com/>

O CommonJS é adequado para o navegador?

Há desenvolvedores que acham que o CommonJS é mais adequado para o desenvolvimento do lado do servidor, o que é uma razão pela qual há atualmente um nível de **desacordo** sobre qual formato deve e será usado como o padrão de fato na era pré-Harmony. Alguns dos argumentos contra CommonJS incluem uma observação de que muitas APIs CommonJS abordam recursos orientados para o servidor que alguém simplesmente não seria capaz de implementar em um nível de navegador em JavaScript - por exemplo, *io* , *system* e *js* podem ser considerados não implementáveis ​​por natureza de sua funcionalidade.

Dito isso, é útil saber como estruturar módulos CommonJS independentemente, para que possamos avaliar melhor como eles se encaixam ao definir módulos que podem ser usados ​​em qualquer lugar. Os módulos que possuem aplicativos tanto no cliente quanto no servidor incluem mecanismos de validação, conversão e modelagem. A forma como alguns desenvolvedores estão escolhendo qual formato usar é optando por CommonJS quando um módulo pode ser usado em um ambiente do lado do servidor e usando AMD se este não for o caso.

Como os módulos AMD são capazes de usar plug-ins e definir coisas mais granulares, como construtores e funções, isso faz sentido. Módulos CommonJS só são capazes de definir objetos que podem ser entediantes de se trabalhar se estivermos tentando obter construtores a partir deles.

Embora esteja além do escopo desta seção, também se pode ter notado que existem diferentes tipos de métodos "requeridos" mencionados ao discutirmos AMD e CommonJS. A preocupação com uma convenção de nomenclatura semelhante é, obviamente, confusão e a comunidade está atualmente dividida quanto aos méritos de uma função de requerimento global. A sugestão de John Hann aqui é que em vez de chamá-lo de "requer", o que provavelmente não alcançaria o objetivo de informar os usuários sobre a diferença entre um requerimento global e interno, pode fazer mais sentido renomear o método do carregador global para outra coisa (por exemplo o nome da biblioteca). É por esta razão que um carregador como curl.js usa curl()ao invés de require.

**Leitura Relacionada**

[Desmistificando Módulos CommonJS](http://dailyjs.com/2010/10/18/modules/)

[JavaScript crescendo](http://www.slideshare.net/davidpadbury/javascript-growing-up)

[As notas do RequireJS no CommonJS](http://requirejs.org/docs/commonjs.html)

[Dando passos de bebê com Node.js e CommonJS - Criando módulos personalizados](http://elegantcode.com/2011/02/04/taking-baby-steps-with-node-js-commonjs-and-creating-custom-modules/)

[Módulos CommonJS assíncronos para o navegador](https://www.sitepen.com/blog/2010/07/16/asynchronous-commonjs-modules-for-the-browser-and-introducing-transporter/)

[A Lista de Correio CommonJS](https://groups.google.com/group/commonjs)

AMD && CommonJS concorrentes, mas igualmente válidos padrões

Ambos AMD e CommonJS são formatos de módulo válidos com objetivos finais diferentes.

A AMD adota uma abordagem de desenvolvimento que prioriza o navegador, optando por comportamento assíncrono e compatibilidade simplificada com versões anteriores, mas não tem nenhum conceito de E / S de arquivo. Suporta objetos, funções, construtores, strings, JSON e muitos outros tipos de módulos, rodando nativamente no navegador. É incrivelmente flexível.

O CommonJS, por outro lado, adota uma abordagem que prioriza o servidor, assumindo um comportamento síncrono, sem *bagagem* global e tenta atender ao futuro (no servidor). O que queremos dizer com isso é que, como o CommonJS oferece suporte a módulos desembrulhados, pode parecer um pouco mais próximo das especificações ES.next/Harmony, nos livrando do define()invólucro que a AMD impõe. Módulos CommonJS, entretanto, só suportam objetos como módulos.

UMD: Módulos compatíveis com AMD e CommonJS para plug-ins

Para os desenvolvedores que desejam criar módulos que podem funcionar em ambientes de navegador e servidor, as soluções existentes podem ser consideradas com pouca falta. Para ajudar a aliviar isso, James Burke, eu e vários outros desenvolvedores criamos UMD (Universal Module Definition) <https://github.com/umdjs/umd> .

UMD é um formato de módulo experimental que permite a definição de módulos que funcionam em ambientes de cliente e servidor com todas ou a maioria das técnicas populares de carregamento de script disponíveis no momento da escrita. Embora a ideia de (ainda) outro formato de módulo possa ser assustadora, cobriremos o UMD brevemente para fins de detalhamento.

Originalmente, começamos a definir o UMD dando uma olhada no wrapper CommonJS simplificado com suporte na especificação AMD. Para desenvolvedores que desejam escrever módulos como se fossem módulos CommonJS, o seguinte formato compatível com CommonJS pode ser usado:

Formato Híbrido AMD Básico

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | define( function ( require, exports, module ){        var shuffler = require( "lib/shuffle" );        exports.randomize = function( input ){          return shuffler.shuffle( input );      }  }); |

No entanto, é importante notar que um módulo só é realmente tratado como um módulo CommonJS se não contiver uma matriz de dependência e a função de definição contiver um parâmetro no mínimo. Isso também não funcionará corretamente em alguns dispositivos (por exemplo, o PS3). Para obter mais informações sobre o wrapper acima, consulte <http://requirejs.org/docs/api.html#cjsmodule> .

Levando isso adiante, queríamos fornecer vários padrões diferentes que não funcionassem apenas com AMD e CommonJS, mas também resolvessem problemas de compatibilidade comuns que os desenvolvedores que desejavam desenvolver tais módulos tinham com outros ambientes.

Uma dessas variações que podemos ver abaixo nos permite usar CommonJS, AMD ou navegadores globais para criar um módulo.

Usando CommonJS, AMD ou navegadores globais para criar um módulo

Defina um módulo commonJsStrict, que depende de outro módulo chamado b. O nome do módulo está implícito no nome do arquivo e sua melhor prática para que o nome do arquivo e o global exportado tenham o mesmo nome.

Se o módulo btambém usar o mesmo tipo de clichê no navegador, ele criará um global .bque será usado. Se não quisermos oferecer suporte ao patch global do navegador, podemos remover o roote o passando thiscomo o primeiro argumento para a função superior.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | (function ( root, factory ) {      if ( typeof exports === 'object' ) {          // CommonJS          factory( exports, require('b') );      } else if ( typeof define === 'function' && define.amd ) {          // AMD. Register as an anonymous module.          define( ['exports', 'b'], factory);      } else {          // Browser globals          factory( (root.commonJsStrict = {}), root.b );      }  }(this, function ( exports, b ) {      //use b in some fashion.        // attach properties to the exports object to define      // the exported module properties.      exports.action = function () {};  })); |

O repositório UMD contém variações que abrangem módulos que funcionam perfeitamente no navegador, aqueles melhores para fornecer exportações, aqueles ideais para tempos de execução CommonJS e até mesmo aqueles que funcionam melhor para definir plug-ins jQuery, que veremos a seguir.

Plug-ins jQuery que funcionam em todos os ambientes

O UMD fornece dois padrões para trabalhar com plug-ins jQuery - um que define plug-ins que funcionam bem com AMD e navegadores globais e outro que também pode funcionar em ambientes CommonJS. O jQuery provavelmente não será usado na maioria dos ambientes CommonJS, portanto, mantenha isso em mente, a menos que estejamos trabalhando com um ambiente que funcione bem com ele.

Vamos agora definir um plugin composto de um núcleo e uma extensão para esse núcleo. O plug-in principal é carregado em um $.corenamespace, que pode então ser facilmente estendido usando extensões de plug-in por meio do padrão de namespacing. Plug-ins carregados por meio de tags de script preenchem automaticamente um pluginnamespace sob core(ou seja $.core.plugin.methodName()).

O padrão pode ser muito bom de trabalhar porque as extensões de plug-in podem acessar propriedades e métodos definidos na base ou, com um pequeno ajuste, sobrescrever o comportamento padrão para que possa ser estendido para fazer mais. Um carregador também não é necessário para tornar nada disso totalmente funcional.

Para obter mais detalhes sobre o que está sendo feito, consulte os comentários embutidos nos exemplos de código abaixo.

**uso.html**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | <script type="text/javascript" src="jquery-1.7.2.min.js"></script>  <script type="text/javascript" src="pluginCore.js"></script>  <script type="text/javascript" src="pluginExtension.js"></script>    <script type="text/javascript">    $(function(){        // Our plugin "core" is exposed under a core namespace in      // this example, which we first cache      var core = $.core;        // Then use use some of the built-in core functionality to      // highlight all divs in the page yellow      core.highlightAll();        // Access the plugins (extensions) loaded into the "plugin"      // namespace of our core module:        // Set the first div in the page to have a green background.      core.plugin.setGreen( "div:first");      // Here we're making use of the core's "highlight" method      // under the hood from a plugin loaded in after it        // Set the last div to the "errorColor" property defined in      // our core module/plugin. If we review the code further down,      // we can see how easy it is to consume properties and methods      // between the core and other plugins      core.plugin.setRed("div:last");  });    </script> |

**pluginCore.js**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | // Module/Plugin core  // Note: the wrapper code we see around the module is what enables  // us to support multiple module formats and specifications by  // mapping the arguments defined to what a specific format expects  // to be present. Our actual module functionality is defined lower  // down, where a named module and exports are demonstrated.  //  // Note that dependencies can just as easily be declared if required  // and should work as demonstrated earlier with the AMD module examples.    (function ( name, definition ){    var theModule = definition(),        // this is considered "safe":        hasDefine = typeof define === "function" && define.amd,        // hasDefine = typeof define === "function",        hasExports = typeof module !== "undefined" && module.exports;      if ( hasDefine ){ // AMD Module      define(theModule);    } else if ( hasExports ) { // Node.js Module      module.exports = theModule;    } else { // Assign to common namespaces or simply the global object (window)      ( this.jQuery || this.ender || this.$ || this)[name] = theModule;    }  })( "core", function () {      var module = this;      module.plugins = [];      module.highlightColor = "yellow";      module.errorColor = "red";      // define the core module here and return the public API      // This is the highlight method used by the core highlightAll()    // method and all of the plugins highlighting elements different    // colors    module.highlight = function( el,strColor ){      if( this.jQuery ){        jQuery(el).css( "background", strColor );      }    }    return {        highlightAll:function(){          module.highlight("div", module.highlightColor);        }    };    }); |

**pluginExtension.js**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | // Extension to module core    (function ( name, definition ) {      var theModule = definition(),          hasDefine = typeof define === "function",          hasExports = typeof module !== "undefined" && module.exports;        if ( hasDefine ) { // AMD Module          define(theModule);      } else if ( hasExports ) { // Node.js Module          module.exports = theModule;      } else {            // Assign to common namespaces or simply the global object (window)          // account for for flat-file/global module extensions          var obj = null,              namespaces,              scope;            obj = null;          namespaces = name.split(".");          scope = ( this.jQuery || this.ender || this.$ || this );            for ( var i = 0; i < namespaces.length; i++ ) {              var packageName = namespaces[i];              if ( obj && i == namespaces.length - 1 ) {                  obj[packageName] = theModule;              } else if ( typeof scope[packageName] === "undefined" ) {                  scope[packageName] = {};              }              obj = scope[packageName];          }        }  })( "core.plugin", function () {        // Define our module here and return the public API.      // This code could be easily adapted with the core to      // allow for methods that overwrite and extend core functionality      // in order to expand the highlight method to do more if we wish.      return {          setGreen: function ( el ) {              highlight(el, "green");          },          setRed: function ( el ) {              highlight(el, errorColor);          }      };    }); |

O UMD não tem como objetivo substituir o AMD nem o CommonJS, mas apenas oferece alguma assistência suplementar para desenvolvedores que desejam fazer seu código funcionar em mais ambientes hoje. Para mais informações ou para contribuir com sugestões para este formato experimental, consulte <https://github.com/umdjs/umd> .

Leitura Adicional

* “ [Using AMD Loaders to Write and Manage Modular JavaScript](http://unscriptable.com/code/Using-AMD-loaders/#0) ,” John Hann
* “ [Demystifying CommonJS Modules](http://dailyjs.com/2010/10/18/modules/) ,” Alex Young
* “ [AMD Module Patterns: Singleton](http://unscriptable.com/index.php/2011/09/22/amd-module-patterns-singleton/) ,” John Hann
* “ [Código padrão de módulos JavaScript Run-Anywhere](https://www.sitepen.com/blog/2010/09/30/run-anywhere-javascript-modules-boilerplate-code/) ,” Kris Zyp
* “ [Padrões e propostas para módulos JavaScript e jQuery](https://tagneto.blogspot.com/2010/12/standards-and-proposals-for-javascript.html) ”, James Burke

ES Harmony

Módulos do futuro

[TC39](https://www.ecma-international.org/memento/TC39.htm) , o corpo de padrões encarregado de definir a sintaxe e a semântica do ECMAScript e suas futuras iterações é composto por vários desenvolvedores muito inteligentes. Alguns desses desenvolvedores (como [Alex Russell](https://twitter.com/slightlylate) ) têm observado de perto a evolução do uso de JavaScript para desenvolvimento em grande escala nos últimos anos e estão bem cientes da necessidade de melhores recursos de linguagem para escrever JS mais modular.

Por esse motivo, existem atualmente propostas para uma série de adições interessantes à linguagem, incluindo [módulos](http://wiki.ecmascript.org/doku.php?id=harmony:modules) flexíveis que podem funcionar tanto no cliente quanto no servidor, um [carregador de módulo](http://wiki.ecmascript.org/doku.php?id=harmony:module_loaders) e muito [mais](http://wiki.ecmascript.org/doku.php?id=harmony:proposals) . Nesta seção, exploraremos exemplos de código usando a sintaxe proposta para módulos em ES.next para que possamos ter uma ideia do que está por vir.

**Observação:** embora o Harmony ainda esteja nas fases de proposta, já podemos experimentar recursos (parciais) do ES.next que abordam o suporte nativo para escrever JavaScript modular graças ao compilador [Traceur](https://code.google.com/p/traceur-compiler/) do Google . Para se levantar e correr com Traceur em menos de um minuto, leia este [começou a receber](https://code.google.com/p/traceur-compiler/wiki/GettingStarted) guia. Há também uma [apresentação do](https://traceur-compiler.googlecode.com/svn/branches/v0.10/presentation/index.html) JSConf sobre ele que vale a pena conferir se estiver interessado em aprender mais sobre o projeto.

Módulos com importação e exportação

Depois de ler as seções sobre módulos AMD e CommonJS, você pode estar familiarizado com o conceito de dependências de módulo (importações) e exportações de módulo (ou a API / variáveis ​​públicas que permitimos que outros módulos consumam). Em ES.next, esses conceitos foram propostos de uma maneira um pouco mais sucinta com as dependências sendo especificadas usando uma importpalavra - chave. exportnão é muito diferente do que poderíamos esperar e muitos desenvolvedores olharão os exemplos de código mais abaixo e os agarrarão instantaneamente.

* **As** declarações de **importação** vinculam as exportações de módulos como variáveis ​​locais e podem ser renomeadas para evitar colisões / conflitos de nomes.
* **as** declarações de **exportação** declaram que uma ligação local de um módulo é externamente visível, de forma que outros módulos podem ler as exportações, mas não podem modificá-las. Curiosamente, os módulos podem exportar módulos filhos, mas não podem exportar módulos que foram definidos em outro lugar. Também podemos renomear exportações para que seus nomes externos sejam diferentes dos nomes locais.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | module staff{      // specify (public) exports that can be consumed by      // other modules      export var baker = {          bake: function( item ){              console.log( "Woo! I just baked " + item );          }      }  }    module skills{      export var specialty = "baking";      export var experience = "5 years";  }    module cakeFactory{        // specify dependencies      import baker from staff;        // import everything with wildcards      import \* from skills;        export var oven = {          makeCupcake: function( toppings ){              baker.bake( "cupcake", toppings );          },          makeMuffin: function( mSize ){              baker.bake( "muffin", size );          }      }  } |

Módulos carregados de fontes remotas

As propostas de módulos também atendem a módulos que são baseados remotamente (por exemplo, bibliotecas de terceiros), tornando mais simples carregar módulos de locais externos. Aqui está um exemplo de puxar o módulo que definimos acima e utilizá-lo:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3 | module cakeFactory from "<http://addyosmani.com/factory/cakes.js>";  cakeFactory.oven.makeCupcake( "sprinkles" );  cakeFactory.oven.makeMuffin( "large" ); |

API Module Loader

O carregador de módulo proposto descreve uma API dinâmica para carregar módulos em contextos altamente controlados. As assinaturas com suporte no carregador incluem load(url, moduleInstance, error) módulos de carregamento createModule(object, globalModuleReferences)e [outros](http://wiki.ecmascript.org/doku.php?id=harmony:module_loaders) .

Aqui está outro exemplo de carregamento dinâmico no módulo que definimos inicialmente. Observe que, ao contrário do último exemplo em que extraímos um módulo de uma fonte remota, a API do carregador de módulo é mais adequada para contextos dinâmicos.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4 | Loader.load( "<http://addyosmani.com/factory/cakes.js>",      function( cakeFactory ){          cakeFactory.oven.makeCupcake( "chocolate" );      }); |

Módulos do tipo CommonJS para o servidor

Para desenvolvedores que estão mais interessados ​​em ambientes de servidor, o sistema de módulos proposto para ES.next não se limita apenas a olhar para os módulos no navegador. Abaixo, por exemplo, podemos ver um módulo do tipo CommonJS proposto para uso no servidor:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3 | // io/File.js  export function open( path ) { ... };  export function close( hnd ) { ... }; |

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | // compiler/LexicalHandler.js  module file from "io/File";    import { open, close } from file;  export function scan( in ) {      try {          var h = open( in ) ...      }      finally { close( h ) }  } |

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4 | module lexer from "compiler/LexicalHandler";  module stdlib from "@std";    //... scan(cmdline[0]) ... |

Aulas com construtores, getters e setters

A noção de uma classe sempre foi uma questão polêmica com os puristas e até agora nos demos bem em recorrer à natureza [prototípica](http://javascript.crockford.com/prototypal.html) do JavaScript ou ao usar frameworks ou abstrações que oferecem a capacidade de usar definições de *classe de* uma forma que desautoriza para o mesmo comportamento prototípico.

No Harmony, classes foram propostas para a linguagem junto com construtores e (finalmente) algum senso de verdadeira privacidade. Nos exemplos a seguir, comentários sequenciais são fornecidos para ajudar a explicar como as classes são estruturadas.

Ao ler, também se pode notar a falta da palavra "função" aqui. Este não é um erro de digitação: TC39 tem feito um esforço consciente para diminuir nosso abuso da functionpalavra - chave para tudo e a esperança é que isso ajude a simplificar como escrevemos o código.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | class Cake{        // We can define the body of a class" constructor      // function by using the keyword "constructor" followed      // by an argument list of public and private declarations.      constructor( name, toppings, price, cakeSize ){          public name = name;          public cakeSize = cakeSize;          public toppings = toppings;          private price = price;        }        // As a part of ES.next's efforts to decrease the unnecessary      // use of "function" for everything, you'll notice that it's      // dropped for cases such as the following. Here an identifier      // followed by an argument list and a body defines a new method        addTopping( topping ){          public( this ).toppings.push( topping );      }        // Getters can be defined by declaring get before      // an identifier/method name and a curly body.      get allToppings(){          return public( this ).toppings;      }        get qualifiesForDiscount(){          return private( this ).price > 5;      }        // Similar to getters, setters can be defined by using      // the "set" keyword before an identifier      set cakeSize( cSize ){          if( cSize < 0 ){              throw new Error( "Cake must be a valid size -              either small, medium or large" );          }          public( this ).cakeSize = cSize;      }      } |

Conclusões do ES Harmony

Como vimos, o Harmony pode vir com algumas novas adições interessantes que irão facilitar o desenvolvimento de aplicativos modulares e lidar com questões como gerenciamento de dependências.

No momento, nossas melhores opções para usar a sintaxe Harmony nos navegadores de hoje é por meio de um transpiler como [Google Traceur](https://code.google.com/p/traceur-compiler/) ou [Esprima](http://esprima.googlecode.com/) . Existem também projetos como [Require HM,](https://github.com/addyosmani/require-hm) que nos permite usar módulos Harmony com AMD. Nossas melhores apostas, entretanto, até que tenhamos a finalização das especificações, são AMD (para módulos no navegador) e CommonJS (para aqueles no servidor).

**Leitura Relacionada**

[Uma primeira olhada nos próximos módulos JavaScript](http://www.2ality.com/2011/03/first-look-at-upcoming-javascript.html)

[David Herman em JavaScript / ES.Next (Vídeo)](https://blog.mozilla.com/dherman/2011/02/23/my-js-meetup-talk/)

[Propostas do Módulo ES Harmony](http://wiki.ecmascript.org/doku.php?id=harmony:modules)

[Semântica / Estrutura lógica do ES Harmony Module](http://wiki.ecmascript.org/doku.php?id=harmony:modules_rationale)

[Propostas de classe ES Harmony](http://wiki.ecmascript.org/doku.php?id=harmony:classes)

Conclusões

Nesta seção, revisamos várias das opções disponíveis para escrever JavaScript modular usando formatos de módulo modernos.

Esses formatos têm uma série de vantagens em relação ao uso do padrão de módulo sozinho, incluindo: evitar a necessidade de gerenciar variáveis ​​globais, melhor suporte para gerenciamento de dependências estáticas e dinâmicas, compatibilidade aprimorada com carregadores de script, melhor compatibilidade para módulos no servidor e muito mais.

Resumindo, recomendo experimentar o que foi sugerido neste capítulo, pois esses formatos oferecem muito poder e flexibilidade que podem ajudar significativamente a organizar melhor nossos aplicativos.

Padrões de design em jQuery

jQuery é atualmente a biblioteca de manipulação de DOM de JavaScript mais popular e fornece uma camada abstrata para interagir com o DOM de maneira segura e entre navegadores. Curiosamente, a biblioteca também serve como um exemplo de como os padrões de projeto podem ser usados ​​de forma eficaz para criar uma API que seja legível e fácil de usar.

Embora em muitos casos os principais contribuidores que escreveram o jQuery não se propusessem a usar padrões específicos, eles existem independentemente e são úteis para aprender. Vamos dar uma olhada no que são alguns desses padrões e como eles são usados ​​na API.

O Padrão Composto

**O Padrão Composto** descreve um grupo de objetos que podem ser tratados da mesma forma que uma única instância de um objeto.

Isso nos permite tratar objetos individuais e composições de maneira uniforme, o que significa que o mesmo comportamento será aplicado independentemente de estarmos trabalhando com um ou mil itens.

Em jQuery, quando aplicamos métodos a um elemento ou coleção de elementos, podemos tratar os dois conjuntos de maneira uniforme, pois ambas as seleções retornam um objeto jQuery.

Isso é demonstrado pelo exemplo de código usando o seletor jQuery abaixo. Aqui é possível adicionar uma activeclasse a ambas as seleções para um único elemento (por exemplo, um elemento com um ID único) ou um grupo de elementos com o mesmo nome de tag ou classe, sem esforço adicional:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // Single elements  $( "#singleItem" ).addClass( "active" );  $( "#container" ).addClass( "active" );    // Collections of elements  $( "div" ).addClass( "active" );  $( ".item" ).addClass( "active" );  $( "input" ).addClass( "active" ); |

A addClass()implementação do jQuery pode usar diretamente os loops *for* nativos (ou jQuery.each()/ do jQuery jQuery.fn.each()) para iterar por meio de uma coleção a fim de aplicar o método a itens individuais ou grupos. Olhando através da fonte, podemos ver que este é realmente o caso:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | addClass: function( value ) {    var classNames, i, l, elem,      setClass, c, cl;      if ( jQuery.isFunction( value ) ) {      return this.each(function( j ) {        jQuery( this ).addClass( value.call(this, j, this.className) );      });    }      if ( value && typeof value === "string" ) {      classNames = value.split( rspace );        for ( i = 0, l = this.length; i < l; i++ ) {        elem = this[ i ];          if ( elem.nodeType === 1 ) {          if ( !elem.className && classNames.length === 1 ) {            elem.className = value;            } else {            setClass = " " + elem.className + " ";              for ( c = 0, cl = classNames.length; c < cl; c++ ) {              if ( !~setClass.indexOf( " " + classNames[ c ] + " " ) ) {                setClass += classNames[ c ] + " ";              }            }            elem.className = jQuery.trim( setClass );          }        }      }    }      return this;  } |

**O padrão do adaptador**

**O Adapter Pattern** traduz uma *interface* para um objeto ou classe em uma interface compatível com um sistema específico.

Adaptadores basicamente permitem que objetos ou classes funcionem juntos, o que normalmente não poderia devido a suas interfaces incompatíveis. O adaptador converte as chamadas para sua interface em chamadas para a interface original e o código necessário para fazer isso geralmente é mínimo.

Um exemplo de adaptador que podemos ter usado é o jQuery.fn.css()método jQuery . Ajuda a normalizar as interfaces de como os estilos podem ser aplicados em vários navegadores, tornando trivial para nós o uso de uma sintaxe simples que é adaptada para usar o que o navegador realmente suporta nos bastidores:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // Cross browser opacity:  // opacity: 0.9; Chrome 4+, FF2+, Saf3.1+, Opera 9+, IE9, iOS 3.2+, Android 2.1+  // filter: alpha(opacity=90); IE6-IE8    // Setting opacity  $( ".container" ).css( { opacity: .5 } );    // Getting opacity  var currentOpacity = $( ".container" ).css('opacity'); |

O cssHook do núcleo jQuery correspondente que torna o acima possível pode ser visto abaixo:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | get: function( elem, computed ) {    // IE uses filters for opacity    return ropacity.test( (          computed && elem.currentStyle ?              elem.currentStyle.filter : elem.style.filter) || "" ) ?      ( parseFloat( RegExp.$1 ) / 100 ) + "" :      computed ? "1" : "";  },    set: function( elem, value ) {    var style = elem.style,      currentStyle = elem.currentStyle,      opacity = jQuery.isNumeric( value ) ?            "alpha(opacity=" + value \* 100 + ")" : "",      filter = currentStyle && currentStyle.filter || style.filter || "";      // IE has trouble with opacity if it does not have layout    // Force it by setting the zoom level    style.zoom = 1;      // if setting opacity to 1, and no other filters    //exist - attempt to remove filter attribute #6652    if ( value >= 1 && jQuery.trim( filter.replace( ralpha, "" ) ) === "" ) {        // Setting style.filter to null, "" & " " still leave      // "filter:" in the cssText if "filter:" is present at all,      // clearType is disabled, we want to avoid this style.removeAttribute      // is IE Only, but so apparently is this code path...      style.removeAttribute( "filter" );        // if there there is no filter style applied in a css rule, we are done      if ( currentStyle && !currentStyle.filter ) {        return;      }    }      // otherwise, set new filter values    style.filter = ralpha.test( filter ) ?      filter.replace( ralpha, opacity ) :      filter + " " + opacity;  }  }; |

O Padrão da Fachada

Conforme revisamos anteriormente no livro, o **Padrão de Fachada** fornece uma interface abstrata mais simples para um corpo de código maior (potencialmente mais complexo).

As fachadas podem ser freqüentemente encontradas na biblioteca jQuery e fornecem aos desenvolvedores acesso fácil a implementações para manipulação de DOM, animação e, de particular interesse, Ajax cross-browser.

A seguir estão as fachadas para jQuery's $.ajax():

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4 | $.get( url, data, callback, dataType );  $.post( url, data, callback, dataType );  $.getJSON( url, data, callback );  $.getScript( url, callback ); |

Estes são traduzidos nos bastidores para:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | // $.get()  $.ajax({    url: url,    data: data,    dataType: dataType  }).done( callback );    // $.post  $.ajax({    type: "POST",    url: url,    data: data,    dataType: dataType  }).done( callback );    // $.getJSON()  $.ajax({    url: url,    dataType: "json",    data: data,  }).done( callback );    // $.getScript()  $.ajax({    url: url,    dataType: "script",  }).done( callback ); |

O que é ainda mais interessante é que as fachadas acima são na verdade fachadas por direito próprio, escondendo uma grande complexidade nos bastidores.

Isso ocorre porque a jQuery.ajax()implementação no núcleo do jQuery é um pedaço de código não trivial, para dizer o mínimo. No mínimo, ele normaliza as diferenças entre navegadores XHR (XMLHttpRequest) e torna trivial para nós realizarmos ações HTTP comuns (por exemplo get, postetc), trabalhar com Deferreds e assim por diante.

Como levaria um capítulo inteiro para mostrar todo o código relacionado às fachadas acima, aqui está o código no núcleo do jQuery normalizando XHR:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | // Functions to create xhrs  function createStandardXHR() {    try {      return new window.XMLHttpRequest();    } catch( e ) {}  }    function createActiveXHR() {    try {      return new window.ActiveXObject( "Microsoft.XMLHTTP" );    } catch( e ) {}  }    // Create the request object  jQuery.ajaxSettings.xhr = window.ActiveXObject ?    /\* Microsoft failed to properly     \* implement the XMLHttpRequest in IE7 (can't request local files),     \* so we use the ActiveXObject when it is available     \* Additionally XMLHttpRequest can be disabled in IE7/IE8 so     \* we need a fallback.     \*/    function() {      return !this.isLocal && createStandardXHR() || createActiveXHR();    } :    // For all other browsers, use the standard XMLHttpRequest object    createStandardXHR;    ... |

Embora o bloco de código a seguir também esteja um nível acima da jqXHRimplementação real do jQuery XHR ( ), é a fachada de conveniência com a qual realmente interagimos:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | // Request the remote document      jQuery.ajax({        url: url,        type: type,        dataType: "html",        data: params,        // Complete callback (responseText is used internally)        complete: function( jqXHR, status, responseText ) {          // Store the response as specified by the jqXHR object          responseText = jqXHR.responseText;          // If successful, inject the HTML into all the matched elements          if ( jqXHR.isResolved() ) {            // Get the actual response in case            // a dataFilter is present in ajaxSettings            jqXHR.done(function( r ) {              responseText = r;            });            // See if a selector was specified            self.html( selector ?              // Create a dummy div to hold the results              jQuery("<div>")                // inject the contents of the document in, removing the scripts                // to avoid any 'Permission Denied' errors in IE                .append(responseText.replace(rscript, ""))                  // Locate the specified elements                .find(selector) :                // If not, just inject the full result              responseText );          }            if ( callback ) {            self.each( callback, [ responseText, status, jqXHR ] );          }        }      });        return this;    }  </div> |

O Padrão Observer

Outro padrão que revisamos anteriormente é o padrão Observer (Publicar / Assinar). É aqui que os objetos em um sistema podem se inscrever em outros objetos e ser notificados por eles quando um evento de interesse ocorre.

O jQuery core vem com suporte integrado para um sistema semelhante a publicar / assinar há alguns anos, ao qual se refere como *eventos personalizados* .

Em versões anteriores da biblioteca, o acesso a esses eventos personalizados era possível usando jQuery.bind()(assinar), jQuery.trigger()(publicar) e jQuery.unbind()(cancelar assinatura), mas em versões recentes isso pode ser feito usando jQuery.on(), jQuery.trigger()e jQuery.off().

Abaixo, podemos ver um exemplo disso sendo usado na prática:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | // Equivalent to subscribe(topicName, callback)  $( document ).on( "topicName", function () {      //..perform some behaviour  });    // Equivalent to publish(topicName)  $( document ).trigger( "topicName" );    // Equivalent to unsubscribe(topicName)  $( document ).off( "topicName" ); |

As chamadas para jQuery.on()e jQuery.off()eventualmente passam pelo sistema de eventos jQuery. Semelhante ao Ajax, como a implementação para isso é relativamente longa, podemos, em vez disso, observar onde e como os manipuladores de eventos reais para eventos personalizados são anexados:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | jQuery.event = {      add: function( elem, types, handler, data, selector ) {        var elemData, eventHandle, events,        t, tns, type, namespaces, handleObj,        handleObjIn, quick, handlers, special;        ...        // Init the element's event structure and main handler,      //if this is the first      events = elemData.events;      if ( !events ) {        elemData.events = events = {};      }      ...        // Handle multiple events separated by a space      // jQuery(...).bind("mouseover mouseout", fn);      types = jQuery.trim( hoverHack(types) ).split( " " );      for ( t = 0; t < types.length; t++ ) {          ...          // Init the event handler queue if we're the first        handlers = events[ type ];        if ( !handlers ) {          handlers = events[ type ] = [];          handlers.delegateCount = 0;            // Only use addEventListener/attachEvent if the special          // events handler returns false          if ( !special.setup || special.setup.call( elem, data,          //namespaces, eventHandle ) === false ) {            // Bind the global event handler to the element            if ( elem.addEventListener ) {              elem.addEventListener( type, eventHandle, false );              } else if ( elem.attachEvent ) {              elem.attachEvent( "on" + type, eventHandle );            }          }        } |

Para aqueles que preferem usar o esquema de nomenclatura convencional para o padrão Observer, [Ben Alman](https://gist.github.com/661855) criou um invólucro simples em torno dos métodos acima que nos permite o acesso a jQuery.publish(), jQuery.subscribee jQuery.unsubscribemétodos. Eu criei um link para eles anteriormente neste livro, mas podemos ver o wrapper completo abaixo.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | (function( $ ) {      var o = $({});      $.subscribe = function() {      o.on.apply(o, arguments);    };      $.unsubscribe = function() {      o.off.apply(o, arguments);    };      $.publish = function() {      o.trigger.apply(o, arguments);    };    }( jQuery )); |

Em versões recentes do jQuery, um objeto de callbacks multiuso ( jQuery.Callbacks) foi disponibilizado para permitir aos usuários escrever novas soluções com base em listas de callback. Uma solução para escrever usando esse recurso é outro sistema de Publicação / Assinatura. Uma implementação disso é a seguinte:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | var topics = {};    jQuery.Topic = function( id ) {      var callbacks,          topic = id && topics[ id ];      if ( !topic ) {          callbacks = jQuery.Callbacks();          topic = {              publish: callbacks.fire,              subscribe: callbacks.add,              unsubscribe: callbacks.remove          };          if ( id ) {              topics[ id ] = topic;          }      }      return topic;  }; |

que pode então ser usado da seguinte maneira:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | // Subscribers  $.Topic( "mailArrived" ).subscribe( fn1 );  $.Topic( "mailArrived" ).subscribe( fn2 );  $.Topic( "mailSent" ).subscribe( fn1 );    // Publisher  $.Topic( "mailArrived" ).publish( "hello world!" );  $.Topic( "mailSent" ).publish( "woo! mail!" );    // Here, "hello world!" gets pushed to fn1 and fn2  // when the "mailArrived" notification is published  // with "woo! mail!" also being pushed to fn1 when  // the "mailSent" notification is published.    // Outputs:  // hello world!  // fn2 says: hello world!  // woo! mail! |

O Padrão Iterator

O Iterator é um padrão de design em que os iteradores (objetos que nos permitem percorrer todos os elementos de uma coleção) acessam os elementos de um objeto agregado sequencialmente, sem precisar expor sua forma subjacente.

Os iteradores encapsulam a estrutura interna de como essa iteração específica ocorre. No caso do jQuery.fn.each()iterador do jQuery , somos realmente capazes de usar o code behind subjacente jQuery.each()para iterar por meio de uma coleção, sem precisar ver ou entender o código que funciona nos bastidores fornecendo esse recurso.

Esse é um padrão que pode ser considerado um caso especial de fachada, onde lidamos explicitamente com problemas relacionados à iteração.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | $.each( ["john","dave","rick","julian"], function( index, value ) {    console.log( index + ": " + value);  });    $( "li" ).each( function ( index ) {    console.log( index + ": " + $( this ).text());  }); |

Aqui podemos ver o código para jQuery.fn.each():

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4 | // Execute a callback for every element in the matched set.  each: function( callback, args ) {    return jQuery.each( this, callback, args );  } |

Seguido pelo código por trás do jQuery.each()qual lida com duas maneiras de iterar por meio de objetos:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | each: function( object, callback, args ) {    var name, i = 0,      length = object.length,      isObj = length === undefined || jQuery.isFunction( object );      if ( args ) {      if ( isObj ) {        for ( name in object ) {          if ( callback.apply( object[ name ], args ) === false ) {            break;          }        }      } else {        for ( ; i < length; ) {          if ( callback.apply( object[ i++ ], args ) === false ) {            break;          }        }      }      // A special, fast, case for the most common use of each    } else {      if ( isObj ) {        for ( name in object ) {          if ( callback.call( object[ name ], name, object[ name ] ) === false ) {            break;          }        }      } else {        for ( ; i < length; ) {          if ( callback.call( object[ i ], i, object[ i++ ] ) === false ) {            break;          }        }      }    }      return object;  }; |

O padrão de inicialização lenta

**A inicialização lenta** é um padrão de design que nos permite atrasar processos caros até a primeira instância em que são necessários. Um exemplo disso é a .ready()função em jQuery que só executa um retorno de chamada quando o DOM está pronto.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | $( document ).ready( function () {        // The ajax request won't attempt to execute until      // the DOM is ready        var jqxhr = $.ajax({        url: "<http://domain.com/api/>",        data: "display=latest&order=ascending"      })      .done( function( data ) ){          $(".status").html( "content loaded" );          console.log( "Data output:" + data );      });    }); |

jQuery.fn.ready()é alimentado por jQuery.bindReady(), visto abaixo:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | bindReady: function() {    if ( readyList ) {      return;    }      readyList = jQuery.Callbacks( "once memory" );      // Catch cases where $(document).ready() is called after the    // browser event has already occurred.    if ( document.readyState === "complete" ) {      // Handle it asynchronously to allow scripts the opportunity to delay ready      return setTimeout( jQuery.ready, 1 );    }      // Mozilla, Opera and webkit support this event    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 ) {      // ensure firing before onload,      // maybe late but safe also for iframes      document.attachEvent( "onreadystatechange", DOMContentLoaded );        // A fallback to window.onload, that will always work      window.attachEvent( "onload", jQuery.ready );        // If IE and not a frame      // continually check to see if the document is ready      var toplevel = false;        try {        toplevel = window.frameElement == null;      } catch(e) {}        if ( document.documentElement.doScroll && toplevel ) {        doScrollCheck();      }    }  }, |

Embora não seja usado diretamente no núcleo do jQuery, alguns desenvolvedores também podem estar familiarizados com o conceito de LazyLoading por meio de plug-ins como [este](http://www.appelsiini.net/projects/lazyload) .

LazyLoading é efetivamente o mesmo que inicialização Lazy e é uma técnica pela qual dados adicionais em uma página são carregados quando necessário (por exemplo, quando um usuário rola até o final da página). Nos últimos anos, esse padrão se tornou bastante proeminente e pode ser encontrado nas interfaces do Twitter e do Facebook.

O padrão proxy

Há momentos em que é necessário controlarmos o acesso e o contexto por trás de um objeto e é aqui que o padrão Proxy pode ser útil.

Pode nos ajudar a controlar quando um objeto caro deve ser instanciado, fornecer maneiras avançadas de referenciar o objeto ou modificar o objeto para funcionar de uma maneira particular em contextos específicos.

No núcleo jQuery, jQuery.proxy()existe um método que aceita como entrada uma função e retorna uma nova que sempre terá um contexto específico. Isso garante que o valor de thisdentro de uma função seja o valor que esperamos.

Um exemplo de onde isso é útil é quando estamos fazendo uso de timers em um clickmanipulador de eventos. Imagine que temos o seguinte manipulador antes de adicionar qualquer temporizador:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4 | $( "button" ).on( "click", function () {    // Within this function, "this" refers to the element that was clicked    $( this ).addClass( "active" );  }); |

Se desejarmos adicionar um atraso significativo antes da activeclasse ser adicionada, *poderíamos* usar setTimeout()para fazer isso. Infelizmente, há um pequeno problema com essa solução: qualquer função que for passada setTimeout()terá um valor diferente para thisdentro dessa função. Em vez disso, vai se referir ao windowobjeto, que não é o que desejamos.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | $( "button" ).on( "click", function () {    setTimeout(function () {      // "this" doesn't refer to our element!      // It refers to window      $( this ).addClass( "active" );    });  }); |

Para contornar esse problema, podemos usar jQuery.proxy()para implementar um tipo de padrão de proxy. Ao chamá-lo com a função e o valor que gostaríamos de atribuir a thisele, na verdade, retornará uma função que retém o valor que desejamos dentro do contexto correto. Veja como isso ficaria:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | $( "button" ).on( "click", function () {        setTimeout( $.proxy( function () {          // "this" now refers to our element as we wanted          $( this ).addClass( "active" );      }, this), 500);        // the last "this" we're passing tells $.proxy() that our DOM element      // is the value we want "this" to refer to.  }); |

A implementação do jQuery jQuery.proxy()pode ser encontrada abaixo:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | // Bind a function to a context, optionally partially applying any  // arguments.  proxy: function( fn, context ) {    if ( typeof context === "string" ) {      var tmp = fn[ context ];      context = fn;      fn = tmp;    }      // Quick check to determine if target is callable, in the spec    // this throws a TypeError, but we will just return undefined.    if ( !jQuery.isFunction( fn ) ) {      return undefined;    }      // Simulated bind    var args = slice.call( arguments, 2 ),      proxy = function() {        return fn.apply( context, args.concat( slice.call( arguments ) ) );      };      // Set the guid of unique handler to the same of original handler, so it can be removed    proxy.guid = fn.guid = fn.guid || proxy.guid || jQuery.guid++;      return proxy;  } |

O Padrão Builder

Ao trabalhar com o DOM, muitas vezes queremos construir novos elementos dinamicamente - um processo que pode aumentar em complexidade dependendo da marcação final, atributos e propriedades que desejamos que nossos elementos construídos contenham.

Elementos complexos requerem cuidado especial ao serem definidos, especialmente se quisermos a flexibilidade para definir literalmente a marcação final para nossos elementos (o que pode ficar confuso) ou tomar uma rota orientada a objetos mais legível. Ter um mecanismo para construir nossos objetos DOM complexos que é independente dos próprios objetos nos dá essa flexibilidade e é exatamente isso que o padrão Builder fornece.

Os construtores nos permitem construir objetos complexos especificando apenas o tipo e o conteúdo do objeto, protegendo-nos do processo de criação ou representação explícita do objeto.

O cifrão jQuery nos permite fazer exatamente isso, pois fornece vários meios diferentes para construir dinamicamente novos objetos jQuery (e DOM), passando a marcação completa para um elemento, marcação parcial e conteúdo ou usando o jQuery para construção :

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | $( '<div class="foo">bar</div>' );    $( '<p id="test">foo <em>bar</em></p>').appendTo("body");    var newParagraph = $( "<p />" ).text( "Hello world" );    $( "<input />" )        .attr({ "type": "text", "id":"sample"})        .appendTo("#container"); |

Abaixo está um trecho do jQuery.prototypemétodo interno do jQuery core que auxilia na construção de objetos jQuery a partir da marcação passada para o jQuery()seletor. Independentemente de ser ou não document.createElementusado para criar um novo elemento, uma referência ao elemento (encontrado ou criado) é injetada no objeto retornado para que métodos adicionais, como os que .attr()possam ser facilmente usados ​​nele, logo em seguida.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | // HANDLE: $(html) -> $(array)    if ( match[1] ) {      context = context instanceof jQuery ? context[0] : context;      doc = ( context ? context.ownerDocument || context : document );        // If a single string is passed in and it's a single tag      // just do a createElement and skip the rest      ret = rsingleTag.exec( selector );        if ( ret ) {        if ( jQuery.isPlainObject( context ) ) {          selector = [ document.createElement( ret[1] ) ];          jQuery.fn.attr.call( selector, context, true );          } else {          selector = [ doc.createElement( ret[1] ) ];        }        } else {        ret = jQuery.buildFragment( [ match[1] ], [ doc ] );        selector = ( ret.cacheable ? jQuery.clone(ret.fragment) : ret.fragment ).childNodes;      }        return jQuery.merge( this, selector ); |

jQuery Plugin Design Patterns

O desenvolvimento do plugin jQuery evoluiu nos últimos anos. Não temos mais apenas uma maneira de escrever plug-ins, mas muitas. Na realidade, certos padrões de design de plug-ins podem funcionar melhor para um determinado problema ou componente do que outros.

Alguns desenvolvedores podem desejar usar a [fábrica de widgets de](http://ajpiano.com/widgetfactory/#slide1) UI do jQuery ; é ótimo para componentes de IU complexos e flexíveis. Alguns podem não.

Alguns podem querer estruturar seus plug-ins mais como módulos (semelhante ao padrão de módulo) ou usar um formato de módulo mais moderno, como AMD.

Alguns podem querer que seus plug-ins aproveitem o poder da herança prototípica. Outros podem desejar usar eventos personalizados ou Publicar / Assinar para se comunicar de plug-ins com o restante do aplicativo. E assim por diante.

Comecei a pensar sobre os padrões de plug-in depois de notar uma série de esforços para criar um padrão de plug-in jQuery de tamanho único. Embora tal clichê seja uma ótima ideia em teoria, a realidade é que raramente escrevemos plug-ins de uma maneira fixa, usando um único padrão o tempo todo.

Vamos supor que já tentamos escrever nossos próprios plug-ins jQuery em algum momento e estamos confortáveis ​​em montar algo que funcione. É funcional. Ele faz o que precisa fazer, mas talvez pensemos que poderia ser melhor estruturado. Talvez pudesse ser mais flexível ou projetado para lidar com mais problemas que os desenvolvedores normalmente enfrentam. Se isso lhe parece familiar, você pode achar este capítulo útil. Nele, vamos explorar uma série de padrões de plug-ins jQuery que funcionaram bem para outros desenvolvedores.

**Nota:** Este capítulo é direcionado a desenvolvedores intermediários a avançados, embora iremos revisar brevemente alguns fundamentos do plugin jQuery para começar.

Se você ainda não se sente preparado para isso, tenho o prazer de recomendar o guia oficial de [plug-ins / autoria](http://docs.jquery.com/Plugins/Authoring) jQuery , o [guia de estilo de plug-in de](https://msdn.microsoft.com/en-us/scriptjunkie/ff696759) Ben Alman e " [Sinais de um plug-in jQuery mal escrito](http://remysharp.com/2010/06/03/signs-of-a-poorly-written-jquery-plugin/) " de Remy Sharp . como material de leitura antes de iniciar esta seção.

Padrões

Os plug-ins jQuery têm poucas regras concretas, o que é uma das razões para a incrível diversidade de como eles são implementados em toda a comunidade. No nível mais básico, podemos escrever um plugin simplesmente adicionando uma nova propriedade de função ao jQuery.fnobjeto jQuery , como segue:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3 | $.fn.myPluginName = function () {      // our plugin logic  }; |

Isso é ótimo para compactação, mas o seguinte seria uma base melhor para construir:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5 | (function( $ ){    $.fn.myPluginName = function () {      // our plugin logic    };  })( jQuery ); |

Aqui, envolvemos nossa lógica de plugin em uma função anônima. Para garantir que nosso uso do $sinal como uma abreviação não crie conflitos entre o jQuery e outras bibliotecas JavaScript, simplesmente o passamos para este encerramento, que o mapeia para o cifrão. Isso garante que ele não seja afetado por nada fora de seu escopo de execução.

Uma maneira alternativa de escrever esse padrão seria usar jQuery.extend(), o que nos permite definir várias funções de uma vez e que às vezes faz mais sentido semanticamente:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | (function( $ ){      $.extend($.fn, {          myplugin: function(){              // your plugin logic          }      });  })( jQuery ); |

Nós agora revisamos alguns fundamentos do plugin jQuery, mas muito mais poderia ser feito para levar isso adiante. *A Lightweight Start* é o primeiro padrão de design de plug-in completo que exploraremos e cobre algumas das melhores práticas que podemos usar para o desenvolvimento básico de plug-ins do dia-a-dia, levando em consideração problemas comuns que valem a pena aplicar.

Nota

Embora a maioria dos padrões abaixo seja explicada, recomendo a leitura dos comentários no código, porque eles oferecem mais informações sobre por que certas práticas recomendadas são aplicadas.

Devo também mencionar que nada disso seria possível sem o trabalho anterior, contribuições e conselhos de outros membros da comunidade jQuery. Listei-os inline com cada padrão para que se possa ler sobre seu trabalho individual, se estiver interessado.

Padrão 'Um começo leve'

Vamos começar nossa análise mais aprofundada dos padrões de plug-in com algo básico que segue as melhores práticas (incluindo aquelas no guia de criação de plug-in jQuery). Esse padrão é ideal para desenvolvedores que são novos no desenvolvimento de plug-ins ou que apenas desejam alcançar algo simples (como um plug-in utilitário). *Um Lightweight Start* usa o seguinte:

* Práticas recomendadas comuns, como ponto-e-vírgula colocado antes da invocação das funções (veremos o porquê nos comentários abaixo)
* window, document, undefined transmitidos como argumentos.
* Um objeto de padrões básicos.
* Um construtor de plug-in simples para lógica relacionada à criação inicial e à atribuição do elemento com o qual trabalhar.
* Estendendo as opções com padrões.
* Um invólucro leve em torno do construtor, que ajuda a evitar problemas como várias instanciações.
* Aderência às diretrizes de estilo do núcleo do jQuery para maximizar a legibilidade.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68 | /\*!   \* jQuery lightweight plugin boilerplate   \* Original author: @ajpiano   \* Further changes, comments: @addyosmani   \* Licensed under the MIT license   \*/      // the semi-colon before the function invocation is a safety  // net against concatenated scripts and/or other plugins  // that are not closed properly.  ;(function ( $, window, document, undefined ) {        // undefined is used here as the undefined global      // variable in ECMAScript 3 and is mutable (i.e. it can      // be changed by someone else). undefined isn't really      // being passed in so we can ensure that its value is      // truly undefined. In ES5, undefined can no longer be      // modified.        // window and document are passed through as local      // variables rather than as globals, because this (slightly)      // quickens the resolution process and can be more      // efficiently minified (especially when both are      // regularly referenced in our plugin).        // Create the defaults once      var pluginName = "defaultPluginName",          defaults = {              propertyName: "value"          };        // The actual plugin constructor      function Plugin( element, options ) {          this.element = element;            // jQuery has an extend method that merges the          // contents of two or more objects, storing the          // result in the first object. The first object          // is generally empty because we don't want to alter          // the default options for future instances of the plugin          this.options = $.extend( {}, defaults, options) ;            this.\_defaults = defaults;          this.\_name = pluginName;            this.init();      }        Plugin.prototype.init = function () {          // Place initialization logic here          // We already have access to the DOM element and          // the options via the instance, e.g. this.element          // and this.options      };        // A really lightweight plugin wrapper around the constructor,      // preventing against multiple instantiations      $.fn[pluginName] = function ( options ) {          return this.each(function () {              if ( !$.data(this, "plugin\_" + pluginName )) {                  $.data( this, "plugin\_" + pluginName,                  new Plugin( this, options ));              }          });      }    })( jQuery, window, document ); |

Uso:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3 | $("#elem").defaultPluginName({    propertyName: "a custom value"  }); |

Leitura Adicional

* [Plugins / autoria](http://docs.jquery.com/Plugins/Authoring) , jQuery
* “ [Signs of a Poorly Written jQuery Plugin](http://remysharp.com/2010/06/03/signs-of-a-poorly-written-jquery-plugin/) ,” Remy Sharp
* “ [How to Create Your Own jQuery Plugin](https://msdn.microsoft.com/en-us/scriptjunkie/ff608209) ”, Elijah Manor
* “ [Style in jQuery Plugins and Why It Matters](https://msdn.microsoft.com/en-us/scriptjunkie/ff696759) ,” Ben Almon
* “ [Create Your First jQuery Plugin, Part 2](http://enterprisejquery.com/2010/07/create-your-first-jquery-plugin-part-2-revising-your-plugin/) ,” Andrew Wirick

Padrão de fábrica de widget “completo”

Embora o guia de criação de plug-ins jQuery seja uma ótima introdução ao desenvolvimento de plug-ins, ele não ajuda a ocultar tarefas comuns de encanamento de plug-in que temos que lidar regularmente.

O jQuery UI Widget Factory é uma solução para esse problema que nos ajuda a construir plug-ins complexos e com estado baseados em princípios orientados a objetos. Também facilita a comunicação com nossa instância de plug-ins, ofuscando uma série de tarefas repetitivas que teríamos que codificar ao trabalhar com plug-ins básicos.

Plug-ins com estado nos ajudam a acompanhar seu estado atual, também nos permitindo alterar as propriedades do plug-in após sua inicialização.

Uma das melhores coisas sobre o Widget Factory é que a maioria da biblioteca de UI do jQuery o usa como base para seus componentes. Isso significa que, se estamos procurando mais orientações sobre a estrutura além desse padrão, não precisaremos olhar além do repositório jQuery UI no GitHub ( <https://github.com/jquery/jquery-ui> ).

Este padrão jQuery UI Widget Factory cobre quase todos os métodos de fábrica padrão suportados, incluindo eventos de disparo. De acordo com o último padrão, comentários são incluídos para todos os métodos usados ​​e mais orientações são fornecidas nos comentários embutidos.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87 | /\*!   \* jQuery UI Widget-factory plugin boilerplate (for 1.8/9+)   \* Author: @addyosmani   \* Further changes: @peolanha   \* Licensed under the MIT license   \*/      ;(function ( $, window, document, undefined ) {        // define our widget under a namespace of your choice      // with additional parameters e.g.      // $.widget( "namespace.widgetname", (optional) - an      // existing widget prototype to inherit from, an object      // literal to become the widget's prototype );        $.widget( "namespace.widgetname", {            //Options to be used as defaults          options: {              someValue: null          },            //Setup widget (e.g. element creation, apply theming          //, bind events etc.)          \_create: function () {                // \_create will automatically run the first time              // this widget is called. Put the initial widget              // setup code here, then we can access the element              // on which the widget was called via this.element.              // The options defined above can be accessed              // via this.options this.element.addStuff();          },            // Destroy an instantiated plugin and clean up          // modifications the widget has made to the DOM          destroy: function () {                // this.element.removeStuff();              // For UI 1.8, destroy must be invoked from the              // base widget              $.Widget.prototype.destroy.call( this );              // For UI 1.9, define \_destroy instead and don't              // worry about              // calling the base widget          },            methodB: function ( event ) {              //\_trigger dispatches callbacks the plugin user              // can subscribe to              // signature: \_trigger( "callbackName", [eventObject],              // [uiObject] )              // e.g. this.\_trigger( "hover", e /\*where e.type ==              // "mouseenter"\*/, { hovered: $(e.target)});              this.\_trigger( "methodA", event, {                  key: value              });          },            methodA: function ( event ) {              this.\_trigger( "dataChanged", event, {                  key: value              });          },            // Respond to any changes the user makes to the          // option method          \_setOption: function ( key, value ) {              switch ( key ) {              case "someValue":                  // this.options.someValue = doSomethingWith( value );                  break;              default:                  // this.options[ key ] = value;                  break;              }                // For UI 1.8, \_setOption must be manually invoked              // from the base widget              $.Widget.prototype.\_setOption.apply( this, arguments );              // For UI 1.9 the \_super method can be used instead              // this.\_super( "\_setOption", key, value );          }      });    })( jQuery, window, document ); |

Uso:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5 | var collection = $("#elem").widgetName({    foo: false  });    collection.widgetName("methodB"); |

Leitura Adicional

* [O jQuery UI Widget Factory](http://ajpiano.com/widgetfactory/#slide1)
* “ [Introduction to Stateful Plugins and the Widget Factory](https://msdn.microsoft.com/en-us/scriptjunkie/ff706600) ,” Doug Neiner
* “ [Widget Factory](http://wiki.jqueryui.com/w/page/12138135/Widget%20factory) ” (explicado), Scott Gonzalez
* “ [Understanding jQuery UI Widgets: A Tutorial](http://bililite.com/blog/understanding-jquery-ui-widgets-a-tutorial/) ,” Hacking at 0300

Padrão de plug-in de namespacing aninhado

Como abordamos anteriormente no livro, o namespace de nosso código é uma maneira de evitar colisões com outros objetos e variáveis ​​no namespace global. Eles são importantes porque queremos proteger nosso plug-in contra falhas no caso de outro script na página usar a mesma variável ou nomes de plug-in que os nossos. Como um bom cidadão do namespace global, também devemos fazer o nosso melhor para não impedir que scripts de outros desenvolvedores sejam executados devido aos mesmos problemas.

JavaScript realmente não tem suporte integrado para namespaces como outras linguagens, mas tem objetos que podem ser usados ​​para obter um efeito semelhante. Empregando um objeto de nível superior como o nome de nosso namespace, podemos facilmente verificar a existência de outro objeto na página com o mesmo nome. Se tal objeto não existe, então o definimos; se existir, simplesmente o estendemos com nosso plugin.

Objetos (ou, melhor dizendo, literais de objeto) podem ser usados ​​para criar namespaces aninhados, como namespace.subnamespace.pluginNamee assim por diante. Mas para manter as coisas simples, o padrão de namespace abaixo deve nos mostrar tudo o que precisamos para começar com esses conceitos.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54 | /\*!   \* jQuery namespaced "Starter" plugin boilerplate   \* Author: @dougneiner   \* Further changes: @addyosmani   \* Licensed under the MIT license   \*/    ;(function ( $ ) {      if (!$.myNamespace) {          $.myNamespace = {};      };        $.myNamespace.myPluginName = function ( el, myFunctionParam, options ) {          // To avoid scope issues, use "base" instead of "this"          // to reference this class from internal events and functions.          var base = this;            // Access to jQuery and DOM versions of element          base.$el = $( el );          base.el = el;            // Add a reverse reference to the DOM object          base.$el.data( "myNamespace.myPluginName", base );            base.init = function () {              base.myFunctionParam = myFunctionParam;                base.options = $.extend({},              $.myNamespace.myPluginName.defaultOptions, options);                // Put our initialization code here          };            // Sample Function, Uncomment to use          // base.functionName = function( parameters ){          //          // };          // Run initializer          base.init();      };        $.myNamespace.myPluginName.defaultOptions = {          myDefaultValue: ""      };        $.fn.mynamespace\_myPluginName = function          ( myFunctionParam, options ) {          return this.each(function () {              (new $.myNamespace.myPluginName( this,              myFunctionParam, options ));          });      };    })( jQuery ); |

Uso:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3 | $("#elem").mynamespace\_myPluginName({    myDefaultValue: "foobar"  }); |

Leitura Adicional

* “ [Namespacing in JavaScript](https://javascriptweblog.wordpress.com/2010/12/07/namespacing-in-javascript/) ,” Angus Croll
* “ [Use Your $ .fn jQuery Namespace](http://ryanflorence.com/use-your-fn-jquery-namespace/) ,” Ryan Florence
* “ [JavaScript Namespacing](http://michaux.ca/articles/javascript-namespacing) ,” Peter Michaux
* “ [Módulos e namespaces em JavaScript](http://www.2ality.com/2011/04/modules-and-namespaces-in-javascript.html) ,” Axel Rauschmayer

Padrão de plug-in de eventos personalizados (com a fábrica de widgets)

Na seção *JavaScript Design Patterns* do livro, discutimos o padrão Observer e, mais tarde, passamos a abordar o suporte da jQuery para eventos personalizados, que oferecem uma solução semelhante para implementar Publicar / Assinar. Este mesmo padrão pode ser usado ao escrever plug-ins jQuery.

A ideia básica aqui é que os objetos em uma página podem publicar notificações de eventos quando algo interessante ocorre em nosso aplicativo. Outros objetos então assinam (ou ouvem) esses eventos e respondem de acordo. Isso resulta na lógica de nosso aplicativo ser significativamente mais desacoplada, pois cada objeto não precisa mais se comunicar diretamente com outro.

No seguinte padrão de fábrica de widget de IU da jQuery, implementaremos um sistema de Publicação / Assinatura baseado em eventos customizado básico que permite que nosso plug-in se inscreva para notificações de eventos do restante de nosso aplicativo, que será responsável por publicá-las.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | /\*!   \* jQuery custom-events plugin boilerplate   \* Author: DevPatch   \* Further changes: @addyosmani   \* Licensed under the MIT license   \*/    // In this pattern, we use jQuery's custom events to add  // pub/sub (publish/subscribe) capabilities to widgets.  // Each widget would publish certain events and subscribe  // to others. This approach effectively helps to decouple  // the widgets and enables them to function independently.    ;(function ( $, window, document, undefined ) {      $.widget( "ao.eventStatus", {          options: {            },            \_create: function() {              var self = this;                //self.element.addClass( "my-widget" );                //subscribe to "myEventStart"              self.element.on( "myEventStart", function( e ) {                  console.log( "event start" );              });                //subscribe to "myEventEnd"              self.element.on( "myEventEnd", function( e ) {                  console.log( "event end" );              });                //unsubscribe to "myEventStart"              //self.element.off( "myEventStart", function(e){                  ///console.log( "unsubscribed to this event" );              //});          },            destroy: function(){              $.Widget.prototype.destroy.apply( this, arguments );          },      });  })( jQuery, window, document );    // Publishing event notifications  // $( ".my-widget" ).trigger( "myEventStart");  // $( ".my-widget" ).trigger( "myEventEnd" ); |

Uso:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3 | var el = $( "#elem" );  el.eventStatus();  el.eventStatus().trigger( "myEventStart" ); |

Leitura Adicional

* “ [Communication Between jQuery UI Widgets](http://www.devpatch.com/articles/2010-03-22-communication-between-jquery-ui-widgets/) ,” Benjamin Sternthal

Herança prototípica com o padrão de ponte DOM-to-Object

Conforme abordado anteriormente, em JavaScript, não temos a noção tradicional de classes que encontraríamos em outras linguagens de programação clássicas, mas temos herança prototípica. Com a herança prototípica, um objeto herda de outro objeto. Podemos aplicar este conceito ao desenvolvimento do plugin jQuery.

[Alex Sexton,](https://alexsexton.com/) autor de Yepnope.js, e [Scott Gonzalez,](http://scottgonzalez.com/) membro da equipe jQuery, examinaram esse tópico em detalhes. Em suma, eles descobriram que, para o desenvolvimento modular organizado, separar claramente o objeto que define a lógica de um plug-in do próprio processo de geração de plug-ins pode ser benéfico.

O benefício é que testar nosso código de plug-ins se torna significativamente mais fácil e também podemos ajustar a maneira como as coisas funcionam nos bastidores, sem alterar a maneira como as APIs de objeto que implementamos são usadas.

No artigo de Sexton sobre esse tópico, ele implementou uma ponte que nos permite anexar nossa lógica geral a um plugin específico, que implementamos no padrão abaixo.

Uma das outras vantagens deste padrão é que não precisamos repetir constantemente o mesmo código de inicialização do plugin, garantindo assim que os conceitos por trás do desenvolvimento DRY sejam mantidos. Alguns desenvolvedores também podem achar esse padrão mais fácil de ler do que outros.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61 | /\*!   \* jQuery prototypal inheritance plugin boilerplate   \* Author: Alex Sexton, Scott Gonzalez   \* Further changes: @addyosmani   \* Licensed under the MIT license   \*/      // myObject - an object representing a concept we wish to model  // (e.g. a car)  var myObject = {    init: function( options, elem ) {      // Mix in the passed-in options with the default options      this.options = $.extend( {}, this.options, options );        // Save the element reference, both as a jQuery      // reference and a normal reference      this.elem = elem;      this.$elem = $( elem );        // Build the DOM's initial structure      this.\_build();        // return this so that we can chain and use the bridge with less code.      return this;    },    options: {      name: "No name"    },    \_build: function(){      //this.$elem.html( "<h1>"+this.options.name+"</h1>" );    },    myMethod: function( msg ){      // We have direct access to the associated and cached      // jQuery element      // this.$elem.append( "<p>"+msg+"</p>" );    }  };      // Object.create support test, and fallback for browsers without it  if ( typeof Object.create !== "function" ) {      Object.create = function (o) {          function F() {}          F.prototype = o;          return new F();      };  }      // Create a plugin based on a defined object  $.plugin = function( name, object ) {    $.fn[name] = function( options ) {      return this.each(function() {        if ( ! $.data( this, name ) ) {          $.data( this, name, Object.create( object ).init(          options, this ) );        }      });    };  }; |

Uso:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | $.plugin( "myobj", myObject );    $("#elem").myobj( {name: "John"} );    var collection = $( "#elem" ).data( "myobj" );  collection.myMethod( "I am a method"); |

Leitura Adicional

* “ [Using Inheritance Patterns To Organize Large jQuery Applications](https://alexsexton.com/?p=51) ,” Alex Sexton
* “ [Como gerenciar aplicativos grandes com jQuery ou qualquer coisa](http://www.slideshare.net/SlexAxton/how-to-manage-large-jquery-apps) ” (discussão mais detalhada), Alex Sexton
* “ [Exemplo prático da necessidade de herança](http://blog.bigbinary.com/2010/03/12/pratical-example-of-need-for-prototypal-inheritance.html) prototípica”, Neeraj Singh
* “ [Prototypal Inheritance in JavaScript](http://javascript.crockford.com/prototypal.html) ,” Douglas Crockford

JQuery UI Widget Factory Bridge Pattern

Se você gostou da ideia de gerar plug-ins com base em objetos no último padrão de design, pode estar interessado em um método chamado jQuery UI Widget Factory $.widget.bridge.

Essa ponte serve basicamente como uma camada intermediária entre um objeto JavaScript que é criado usando $.widgete a API principal do jQuery, fornecendo uma solução mais integrada para alcançar a definição de plug-in baseada em objeto. Efetivamente, somos capazes de criar plug-ins com estado usando um construtor personalizado.

Além disso, $.widget.bridgefornece acesso a uma série de outros recursos, incluindo o seguinte:

* Os métodos públicos e privados são tratados como seria de se esperar na OOP clássica (ou seja, os métodos públicos são expostos, enquanto chamadas para métodos privados não são possíveis).
* Proteção automática contra várias inicializações.
* Geração automática de instâncias de um objeto passado e armazenamento delas no $.datacache interno da seleção .
* As opções podem ser alteradas após a inicialização.

Para obter mais informações sobre como usar esse padrão, consulte os comentários embutidos abaixo:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73 | /\*!   \* jQuery UI Widget factory "bridge" plugin boilerplate   \* Author: @erichynds   \* Further changes, additional comments: @addyosmani   \* Licensed under the MIT license   \*/      // a "widgetName" object constructor  // required: this must accept two arguments,  // options: an object of configuration options  // element: the DOM element the instance was created on  var widgetName = function( options, element ){    this.name = "myWidgetName";    this.options = options;    this.element = element;    this.\_init();  }    // the "widgetName" prototype  widgetName.prototype = {        // \_create will automatically run the first time this      // widget is called      \_create: function(){          // creation code      },        // required: initialization logic for the plugin goes into \_init      // This fires when our instance is first created and when      // attempting to initialize the widget again (by the bridge)      // after it has already been initialized.      \_init: function(){          // init code      },        // required: objects to be used with the bridge must contain an      // "option". Post-initialization, the logic for changing options      // goes here.      option: function( key, value ){            // optional: get/change options post initialization          // ignore if you don't require them.            // signature: $("#foo").bar({ cool:false });          if( $.isPlainObject( key ) ){              this.options = $.extend( true, this.options, key );            // signature: $( "#foo" ).option( "cool" ); - getter          } else if ( key && typeof value === "undefined" ){              return this.options[ key ];            // signature: $( "#foo" ).bar("option", "baz", false );          } else {              this.options[ key ] = value;          }            // required: option must return the current instance.          // When re-initializing an instance on elements, option          // is called first and is then chained to the \_init method.          return this;      },        // notice no underscore is used for public methods      publicFunction: function(){          console.log( "public function" );      },        // underscores are used for private methods      \_privateFunction: function(){          console.log( "private function" );      }  }; |

Uso:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | // connect the widget obj to jQuery's API under the "foo" namespace  $.widget.bridge( "foo", widgetName );    // create an instance of the widget for use  var instance = $( "#foo" ).foo({     baz: true  });    // our widget instance exists in the elem's data  // Outputs: #elem  console.log(instance.data( "foo" ).element);    // bridge allows us to call public methods  // Outputs: "public method"  instance.foo("publicFunction");    // bridge prevents calls to internal methods  instance.foo("\_privateFunction"); |

Leitura Adicional

* “ [Using $ .widget.bridge Outside of the Widget Factory](http://erichynds.com/jquery/using-jquery-ui-widget-factory-bridge/) ,” Eric Hynds

Widgets jQuery Mobile com a fábrica de widgets

jQuery mobile é uma estrutura de projeto jQuery que incentiva o design de aplicativos da web onipresentes que funcionam em dispositivos e plataformas móveis populares e no desktop. Em vez de escrever aplicativos exclusivos para cada dispositivo ou sistema operacional, simplesmente escrevemos o código uma vez e, idealmente, ele deve ser executado em muitos dos navegadores de grau A, B e C disponíveis no momento.

Os fundamentos por trás do jQuery mobile também podem ser aplicados ao desenvolvimento de plug-ins e widgets.

O que é interessante neste próximo padrão é que embora existam pequenas e sutis diferenças na escrita de um widget otimizado para “móvel”, aqueles familiarizados com o uso do padrão JQuery UI Widget Factory anterior devem ser capazes de entender isso em quase nenhum momento.

O widget otimizado para celular abaixo tem uma série de diferenças interessantes em relação ao padrão de widget de IU padrão que vimos anteriormente:

* $.mobile.widgeté referenciado como um protótipo de widget existente do qual herdar. Para widgets padrão, passar por qualquer protótipo é desnecessário para o desenvolvimento básico, mas usar este protótipo de widget específico do jQuery-mobile fornece acesso interno a outras "opções" de formatação.
* Em \_create(), é fornecido um guia sobre como os widgets móveis jQuery oficiais lidam com a seleção de elementos, optando por uma abordagem baseada em função que melhor se ajusta à marcação jQM. Isso não quer dizer que a seleção padrão não seja recomendada, apenas que essa abordagem pode fazer mais sentido, dada a estrutura das páginas do jQuery Mobile.
* Diretrizes também são fornecidas no formulário de comentário para aplicar nossos métodos de plug-in pagecreate, bem como para selecionar o aplicativo de plug-in por meio de funções de dados e atributos de dados.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90 | /\*!   \* (jQuery mobile) jQuery UI Widget-factory plugin boilerplate (for 1.8/9+)   \* Author: @scottjehl   \* Further changes: @addyosmani   \* Licensed under the MIT license   \*/    ;(function ( $, window, document, undefined ) {        // define a widget under a namespace of our choice      // here "mobile" has been used in the first argument      $.widget( "mobile.widgetName", $.mobile.widget, {            // Options to be used as defaults          options: {              foo: true,              bar: false          },            \_create: function() {              // \_create will automatically run the first time this              // widget is called. Put the initial widget set-up code              // here, then we can access the element on which              // the widget was called via this.element              // The options defined above can be accessed via              // this.options                // var m = this.element,              // p = m.parents( ":jqmData(role="page")" ),              // c = p.find( ":jqmData(role="content")" )          },            // Private methods/props start with underscores          \_dosomething: function(){ ... },            // Public methods like these below can can be called          // externally:          // $("#myelem").foo( "enable", arguments );            enable: function() { ... },            // Destroy an instantiated plugin and clean up modifications          // the widget has made to the DOM          destroy: function () {              // this.element.removeStuff();              // For UI 1.8, destroy must be invoked from the              // base widget              $.Widget.prototype.destroy.call( this );              // For UI 1.9, define \_destroy instead and don't              // worry about calling the base widget          },            methodB: function ( event ) {              //\_trigger dispatches callbacks the plugin user can              // subscribe to              // signature: \_trigger( "callbackName", [eventObject],              // [uiObject] )              // e.g. this.\_trigger( "hover", e /\*where e.type ==              // "mouseenter"\*/, { hovered: $(e.target)});              this.\_trigger( "methodA", event, {                  key: value              });          },            methodA: function ( event ) {              this.\_trigger( "dataChanged", event, {                  key: value              });          },            // Respond to any changes the user makes to the option method          \_setOption: function ( key, value ) {              switch ( key ) {              case "someValue":                  // this.options.someValue = doSomethingWith( value );                  break;              default:                  // this.options[ key ] = value;                  break;              }                // For UI 1.8, \_setOption must be manually invoked from              // the base widget              $.Widget.prototype.\_setOption.apply(this, arguments);              // For UI 1.9 the \_super method can be used instead              // this.\_super( "\_setOption", key, value );          }      });    })( jQuery, window, document ); |

Uso:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5 | var instance = $( "#foo" ).widgetName({    foo: false  });    instance.widgetName( "methodB" ); |

Também podemos auto-inicializar esse widget sempre que uma nova página no jQuery Mobile for criada. O plugin de *página* do jQuery Mobile despacha um evento de *criação* quando uma página do jQuery Mobile (encontrada por meio do data-role="page"atributo) é inicializada pela primeira vez. Podemos ouvir esse evento (chamado "pagecreate") e executar nosso plugin automaticamente sempre que uma nova página for criada.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | $(document).on("pagecreate", function ( e ) {      // In here, e.target refers to the page that was created      // (it's the target of the pagecreate event)      // So, we can simply find elements on this page that match a      // selector of our choosing, and call our plugin on them.      // Here's how we'd call our "foo" plugin on any element with a      // data-role attribute of "foo":      $(e.target).find( "[data-role="foo"]" ).foo( options );        // Or, better yet, let's write the selector accounting for the configurable      // data-attribute namespace      $( e.target ).find( ":jqmData(role="foo")" ).foo( options );  }); |

Agora podemos simplesmente fazer referência ao script que contém nosso widget e pagecreatevinculação em uma página que executa o site do jQuery Mobile e ele será executado automaticamente como qualquer outro plugin do jQuery Mobile.

RequireJS e jQuery UI Widget Factory

Conforme abordamos na seção sobre *Padrões de design de módulo moderno* , RequireJS é um carregador de script compatível com AMD que fornece uma solução limpa para encapsular a lógica do aplicativo dentro de módulos gerenciáveis.

É capaz de carregar módulos na ordem correta (por meio de seu plugin de pedidos), simplifica o processo de combinação de scripts por meio de seu excelente otimizador r.js e fornece os meios para definir dependências dinâmicas por módulo.

No padrão clichê abaixo, demonstramos como um widget de UI jQuery compatível com AMD (e, portanto, RequireJS) pode ser definido para fazer o seguinte:

* Permite a definição de dependências do módulo de widget, construindo sobre o padrão jQuery UI Widget Factory anterior apresentado anteriormente.
* Demonstra uma abordagem para passar ativos de modelo HTML para a criação de widgets com modelo (usando micro-modelos Underscore.js).
* Inclui uma dica rápida sobre os ajustes que podemos fazer em nosso módulo de widget se desejarmos passá-lo posteriormente para o otimizador RequireJS.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96 | /\*!   \* jQuery UI Widget + RequireJS module boilerplate (for 1.8/9+)   \* Authors: @jrburke, @addyosmani   \* Licensed under the MIT license   \*/    // Note from James:  //  // This assumes we are using the RequireJS+jQuery file, and  // that the following files are all in the same directory:  //  // - require-jquery.js  // - jquery-ui.custom.min.js (custom jQuery UI build with widget factory)  // - templates/  // - asset.html  // - ao.myWidget.js    // Then we can construct the widget as follows:    // ao.myWidget.js file:  define( "ao.myWidget", ["jquery", "text!templates/asset.html", "underscore", "jquery-ui.custom.min"], function ( $, assetHtml, \_ ) {        // define our widget under a namespace of our choice      // "ao" is used here as a demonstration      $.widget( "ao.myWidget", {            // Options to be used as defaults          options: {},            // Set up widget (e.g. create element, apply theming,          // bind events, etc.)          \_create: function () {                // \_create will automatically run the first time              // this widget is called. Put the initial widget              // set-up code here, then we can access the element              // on which the widget was called via this.element.              // The options defined above can be accessed via              // this.options                // this.element.addStuff();              // this.element.addStuff();                // We can then use Underscore templating with              // with the assetHtml that has been pulled in              // var template = \_.template( assetHtml );              // this.content.append( template({}) );          },            // Destroy an instantiated plugin and clean up modifications          // that the widget has made to the DOM          destroy: function () {              // this.element.removeStuff();              // For UI 1.8, destroy must be invoked from the base              // widget              $.Widget.prototype.destroy.call( this );              // For UI 1.9, define \_destroy instead and don't worry              // about calling the base widget          },            methodB: function ( event ) {              // \_trigger dispatches callbacks the plugin user can              // subscribe to              // signature: \_trigger( "callbackName", [eventObject],              // [uiObject] )              this.\_trigger( "methodA", event, {                  key: value              });          },            methodA: function ( event ) {              this.\_trigger("dataChanged", event, {                  key: value              });          },            // Respond to any changes the user makes to the option method          \_setOption: function ( key, value ) {              switch (key) {              case "someValue":                  // this.options.someValue = doSomethingWith( value );                  break;              default:                  // this.options[ key ] = value;                  break;              }                // For UI 1.8, \_setOption must be manually invoked from              // the base widget              $.Widget.prototype.\_setOption.apply( this, arguments );              // For UI 1.9 the \_super method can be used instead              // this.\_super( "\_setOption", key, value );          }        });  }); |

Uso:

index.html:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1 | <script data-main="scripts/main" src="<http://requirejs.org/docs/release/1.0.1/minified/require.js>"></script> |

main.js

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | require({        paths: {          "jquery": "<https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min>",          "jqueryui": "<https://ajax.googleapis.com/ajax/libs/jqueryui/1.8.18/jquery-ui.min>",          "boilerplate": "../patterns/jquery.widget-factory.requirejs.boilerplate"      }  }, ["require", "jquery", "jqueryui", "boilerplate"],  function (req, $) {        $(function () {            var instance = $("#elem").myWidget();          instance.myWidget("methodB");        });  }); |

Leitura Adicional

* “ [Fast Modular Code With jQuery and RequireJS](http://speakerrate.com/talks/2983-fast-modular-code-with-jquery-and-requirejs) ,” James Burke
* “ [Melhores amigos da jQuery](http://jquerysbestfriends.com/#slide1) ” , Alex Sexton
* “ [Gerenciando dependências com RequireJS](http://www.angrycoding.com/2011/09/managing-dependencies-with-requirejs.html) ,” Ruslan Matveev

Opções substituíveis globalmente e por chamada (padrão de melhores opções)

Para nosso próximo padrão, veremos uma abordagem ideal para configurar opções e padrões para um plugin. A forma como a maioria de nós provavelmente está familiarizada com a definição de opções de plug-in é passar por um objeto literal de padrões $.extend(), conforme demonstrado em nosso texto padrão de plug-in básico.

Se, no entanto, estivermos trabalhando com um plug-in com muitas opções personalizáveis ​​que gostaríamos que os usuários pudessem substituir globalmente ou por chamada, então podemos estruturar as coisas de uma forma um pouco mais otimizada.

Em vez disso, referindo-se a um objeto de opções definido no namespace do plug-in explicitamente (por exemplo, $fn.pluginName.options) e mesclando-o com quaisquer opções passadas para o plug-in quando ele é inicialmente invocado, os usuários têm a opção de passar opções durante a inicialização do plug-in ou substituir opções fora do plugin (conforme demonstrado aqui).

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | /\*!   \* jQuery "best options" plugin boilerplate   \* Author: @cowboy   \* Further changes: @addyosmani   \* Licensed under the MIT license   \*/      ;(function ( $, window, document, undefined ) {        $.fn.pluginName = function ( options ) {            // Here's a best practice for overriding "defaults"          // with specified options. Note how, rather than a          // regular defaults object being passed as the second          // parameter, we instead refer to $.fn.pluginName.options          // explicitly, merging it with the options passed directly          // to the plugin. This allows us to override options both          // globally and on a per-call level.            options = $.extend( {}, $.fn.pluginName.options, options );            return this.each(function () {                var elem = $(this);            });      };        // Globally overriding options      // Here are our publicly accessible default plugin options      // that are available in case the user doesn't pass in all      // of the values expected. The user is given a default      // experience but can also override the values as necessary.      // e.g. $fn.pluginName.key ="otherval";        $.fn.pluginName.options = {            key: "value",          myMethod: function ( elem, param ) {            }      };    })( jQuery, window, document ); |

Uso:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3 | $("#elem").pluginName({    key: "foobar"  }); |

Leitura Adicional

* [Pluginização jQuery](http://benalman.com/talks/jquery-pluginization.html) e a [essência que](https://gist.github.com/472783/e8bf47340413129a8abe5fac55c83336efb5d4e1) o [acompanha](https://gist.github.com/472783/e8bf47340413129a8abe5fac55c83336efb5d4e1) , Ben Alman

Um padrão de plug-in altamente configurável e mutável

Nesse padrão, semelhante ao padrão de plug-in de herança prototípico de Alex Sexton, a lógica do nosso plug-in não está aninhada em um plug-in jQuery em si. Em vez disso, definimos nossa lógica de plugin usando um construtor e um literal de objeto definido em seu protótipo. O jQuery é então usado para a instanciação real do objeto de plugin.

A personalização é levada para o próximo nível, empregando dois pequenos truques, um dos quais vimos nos padrões anteriores:

* As opções podem ser substituídas globalmente e por coleção de elementos /
* As opções podem ser personalizadas em um nível por **elemento** por meio de atributos de dados HTML5 (conforme mostrado abaixo). Isso facilita o comportamento do plug-in que pode ser aplicado a uma coleção de elementos, mas depois customizado em linha, sem a necessidade de instanciar cada elemento com um valor padrão diferente.

Não vemos a última opção na natureza com muita frequência, mas pode ser uma solução significativamente mais limpa (contanto que não nos importemos com a abordagem em linha). Se você está se perguntando onde isso pode ser útil, imagine escrever um plug-in arrastável para um grande conjunto de elementos. Poderíamos personalizar suas opções da seguinte maneira:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4 | $( ".item-a" ).draggable( {"defaultPosition":"top-left"} );  $( ".item-b" ).draggable( {"defaultPosition":"bottom-right"} );  $( ".item-c" ).draggable( {"defaultPosition":"bottom-left"} );  //etc |

Mas usando nossa abordagem de padrões embutidos, o seguinte seria possível:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1 | $( ".items" ).draggable(); |

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3 | html  <li class="item" data-plugin-options="{"defaultPosition":"top-left"}"></div>  <li class="item" data-plugin-options="{"defaultPosition":"bottom-left"}"></div> |

E assim por diante. Podemos muito bem ter uma preferência por uma dessas abordagens, mas é apenas outra variação da qual vale a pena estar atento.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69 | /\*   \* "Highly configurable" mutable plugin boilerplate   \* Author: @markdalgleish   \* Further changes, comments: @addyosmani   \* Licensed under the MIT license   \*/      // Note that with this pattern, as per Alex Sexton's, the plugin logic  // hasn't been nested in a jQuery plugin. Instead, we just use  // jQuery for its instantiation.    ;(function( $, window, document, undefined ){      // our plugin constructor    var Plugin = function( elem, options ){        this.elem = elem;        this.$elem = $(elem);        this.options = options;          // This next line takes advantage of HTML5 data attributes        // to support customization of the plugin on a per-element        // basis. For example,        // <div class="item" data-plugin-options="{'message':'Goodbye World!'}"></div>        this.metadata = this.$elem.data( "plugin-options" );      };      // the plugin prototype    Plugin.prototype = {      defaults: {        message: "Hello world!"      },        init: function() {        // Introduce defaults that can be extended either        // globally or using an object literal.        this.config = $.extend( {}, this.defaults, this.options,        this.metadata );          // Sample usage:        // Set the message per instance:        // $( "#elem" ).plugin( { message: "Goodbye World!"} );        // or        // var p = new Plugin( document.getElementById( "elem" ),        // { message: "Goodbye World!"}).init()        // or, set the global default message:        // Plugin.defaults.message = "Goodbye World!"          this.sampleMethod();        return this;      },        sampleMethod: function() {        // e.g. show the currently configured message        // console.log(this.config.message);      }    }      Plugin.defaults = Plugin.prototype.defaults;      $.fn.plugin = function( options ) {      return this.each(function() {        new Plugin( this, options ).init();      });    };      // optional: window.Plugin = Plugin;    })( jQuery, window, document ); |

Uso:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3 | $("#elem").plugin({    message: "foobar"  }); |

Leitura Adicional

* “ [Criando Plug-ins jQuery Altamente Configuráveis](http://markdalgleish.com/2011/05/creating-highly-configurable-jquery-plugins/) ,” Mark Dalgleish
* “ [Escrevendo Plug-ins jQuery Altamente Configuráveis, Parte 2](http://markdalgleish.com/2011/09/html5data-creating-highly-configurable-jquery-plugins-part-2/) ,” Mark Dalgleish

O que torna um bom plug-in além dos padrões?

No final do dia, os padrões de projeto são apenas uma faceta para escrever plug-ins jQuery sustentáveis. Há uma série de outros fatores que vale a pena considerar e eu gostaria de compartilhar meus próprios critérios para selecionar plug-ins de terceiros para abordar algumas das outras preocupações. Espero que isso ajude a aumentar a qualidade geral de seus projetos de plug-in:

**Qualidade**

Siga as melhores práticas com relação ao JavaScript e jQuery que você escreve. Estão sendo feitos esforços para lintar o código do plugin usando jsHint ou jsLint? O plugin foi escrito de forma otimizada?

**Estilo de Código**

O plugin segue um guia de estilo de código consistente, como as [diretrizes de estilo do núcleo jQuery](http://docs.jquery.com/JQuery_Core_Style_Guidelines) ? Se não, seu código está pelo menos relativamente limpo e legível?

**Compatibilidade**

Com quais versões do jQuery o plugin é compatível? Foi testado com as compilações jQuery-git mais recentes ou com o estável mais recente? Se o plugin foi escrito antes do jQuery 1.6, então ele pode ter problemas com atributos e propriedades, porque a forma como eles foram abordados mudou naquela versão.

Novas versões do jQuery oferecem melhorias e oportunidades para o projeto jQuery melhorar o que a biblioteca principal oferece. Com isso, vêm as quebras ocasionais (principalmente em lançamentos importantes) à medida que avançamos em direção a uma maneira melhor de fazer as coisas. Gostaria de ver os autores de plug-ins atualizando seus códigos quando necessário ou, no mínimo, testando seus plug-ins com novas versões para ter certeza de que tudo funciona como esperado.

**Confiabilidade**

O plugin deve vir com seu próprio conjunto de testes de unidade. Isso não apenas prova que ele realmente funciona conforme o esperado, mas também pode melhorar o design sem prejudicá-lo para os usuários finais. Eu considero os testes de unidade essenciais para qualquer plugin jQuery sério que se destina a um ambiente de produção, e eles não são tão difíceis de escrever. Para obter um guia excelente para testes automatizados de JavaScript com QUnit, você pode estar interessado em “ [Automatizar testes de JavaScript com QUnit](https://msdn.microsoft.com/en-us/scriptjunkie/gg749824) ”, de [Jörn Zaefferer](http://bassistance.de/) .

**atuação**

Se o plugin precisa realizar tarefas que requerem processamento extensivo ou manipulação pesada do DOM, deve-se seguir as melhores práticas de benchmarking para ajudar a minimizar isso. Use [jsPerf.com](https://jsperf.com/) para testar segmentos do código para a) quão bem ele executa em navegadores diferentes eb) descobrir o que, se houver algo, pode ser otimizado ainda mais.

**Documentação**

Se a intenção é que outros desenvolvedores usem o plugin, certifique-se de que ele esteja bem documentado. Documente a API e como o plug-in deve ser usado. Quais métodos e opções o plugin suporta? Ele tem alguma pegadinha que os usuários precisam saber? Se os usuários não conseguirem descobrir como usar o plug-in, provavelmente procurarão uma alternativa. Também é de grande ajuda comentar o código do plugin. Este é de longe o melhor presente que você pode oferecer a outros desenvolvedores. Se alguém acha que pode navegar em sua base de código bem o suficiente para usá-lo ou melhorá-lo, então você fez um bom trabalho.

**Probabilidade de manutenção**

Ao lançar um plugin, estime quanto tempo pode ser necessário para manutenção e suporte. Todos nós amamos compartilhar nossos plug-ins com a comunidade, mas é preciso definir expectativas quanto à capacidade de responder perguntas, resolver problemas e fazer melhorias contínuas. Isso pode ser feito simplesmente declarando as intenções do projeto para suporte de manutenção antecipadamente no arquivo *README* .

Conclusões

Neste capítulo, exploramos vários padrões de design que economizam tempo e práticas recomendadas que podem ser empregados para melhorar a forma como os plug-ins jQuery podem ser escritos. Alguns são mais adequados a determinados casos de uso do que outros, mas espero que, de modo geral, esses padrões sejam úteis.

Lembre-se de que, ao selecionar um padrão, é importante ser prático. Não use um padrão de plug-in apenas por causa dele, em vez disso, invista tempo no entendimento da estrutura subjacente e estabeleça quão bem ele resolve seu problema ou se ajusta ao componente que você está tentando construir.

Padrões de namespacing

Nesta seção, vamos explorar os padrões de namespacing em JavaScript. Os namespaces podem ser considerados um agrupamento lógico de unidades de código sob um identificador único. O identificador pode ser referenciado em muitos namespaces e cada identificador pode conter uma hierarquia de seus próprios namespaces aninhados (ou sub).

No desenvolvimento de aplicativos, empregamos namespaces por uma série de razões importantes. Em JavaScript, eles nos ajudam a evitar **colisões** com outros objetos ou variáveis ​​no namespace global. Eles também são extremamente úteis para ajudar a organizar blocos de funcionalidade em uma base de código para que possam ser referenciados e usados ​​com mais facilidade.

O namespacing de qualquer script ou aplicativo sério é crítico, pois é importante proteger nosso código contra falhas no caso de outro script na página usando os **mesmos** nomes de variáveis ​​ou métodos que nós. Com o número de tags de **terceiros** injetadas regularmente nas páginas atualmente, esse pode ser um problema comum que todos precisamos enfrentar em algum momento de nossas carreiras. Como um "cidadão" bem-comportado do namespace global, também é fundamental que façamos o nosso melhor para não impedir a execução de scripts de outros desenvolvedores devido aos mesmos problemas.

Embora o JavaScript realmente não tenha suporte embutido para namespaces como outras linguagens, ele possui objetos e fechamentos que podem ser usados ​​para obter um efeito semelhante.

Fundamentos de namespacing

Os namespaces podem ser encontrados em quase qualquer aplicativo JavaScript sério. A menos que estejamos trabalhando com um trecho de código simples, é fundamental que façamos o nosso melhor para garantir que estamos implementando o namespacing corretamente, pois não é apenas simples de selecionar, mas também evitará que o código de terceiros destrua o nosso. Os padrões que examinaremos nesta seção são:

1. Variáveis ​​globais únicas
2. Espaço de nomes de prefixo
3. Notação literal de objeto
4. Namespacing aninhado
5. Expressões de função invocadas imediatamente
6. Injeção de namespace

1. Variáveis ​​globais únicas

Um padrão popular para namespacing em JavaScript é optar por uma única variável global como nosso principal objeto de referência. Um esqueleto de implementação onde retornamos um objeto com funções e propriedades pode ser encontrado abaixo:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | var myApplication = (function () {    function() {      //...    },    return {      //...    }  })(); |

Embora isso funcione para certas situações, o maior desafio com o padrão de variável global única é garantir que ninguém mais use o mesmo nome de variável global que temos na página.

2. Espaço de nomes de prefixo

Uma solução para o problema acima, conforme mencionado por [Peter Michaux](http://michaux.ca/articles/javascript-namespacing) , é usar namespacing de prefixo. É um conceito simples no fundo, mas a ideia é que selecionamos um namespace de prefixo exclusivo que desejamos usar (neste exemplo myApplication\_) e, em seguida, definimos quaisquer métodos, variáveis ​​ou outros objetos após o prefixo da seguinte maneira:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5 | var myApplication\_propertyA = {};  var myApplication\_propertyB = {};  function myApplication\_myMethod() {    //...  } |

Isso é eficaz da perspectiva de diminuir as chances de uma determinada variável existir no escopo global, mas lembre-se de que um objeto com nome exclusivo pode ter o mesmo efeito.

Deixando isso de lado, o maior problema com o padrão é que ele pode resultar em um grande número de objetos globais assim que nosso aplicativo começar a crescer. Também existe uma grande dependência de nosso prefixo não ser usado por nenhum outro desenvolvedor no namespace global, portanto, tome cuidado ao optar por usá-lo.

Para obter mais informações sobre as opiniões de Peter sobre o padrão de variável global única, leia sua excelente postagem sobre eles <http://peter.michaux.ca/articles/javascript-namespacing> .

3. Notação literal do objeto

A notação literal de objeto (que também abordamos na seção de padrão de módulo do livro) pode ser considerada um objeto que contém uma coleção de pares de chave: valor com dois pontos separando cada par de chaves e valores onde as chaves também podem representar novos namespaces.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | var myApplication = {      // As we've seen, we can easily define functionality for    // this object literal..    getInfo: function() {      //...    },      // but we can also populate it to support    // further object namespaces containing anything    // anything we wish:    models: {},    views: {        pages: {}    },    collections: {}  }; |

Também se pode optar por adicionar propriedades diretamente ao namespace:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | myApplication.foo = function() {    return "bar";  }    myApplication.utils = {    toString: function() {        //...    },    export: function() {        //...    }  } |

Literais de objeto têm a vantagem de não poluir o namespace global, mas ajudam a organizar o código e os parâmetros de maneira lógica. Eles são realmente benéficos se desejarmos criar estruturas de fácil leitura que possam ser expandidas para suportar aninhamento profundo. Ao contrário das variáveis ​​globais simples, os literais de objeto frequentemente também levam em consideração os testes para a existência de uma variável com o mesmo nome, de modo que as chances de ocorrência de colisão são significativamente reduzidas.

No próximo exemplo, demonstramos uma série de maneiras diferentes nas quais podemos verificar se uma variável (namespace de objeto ou plugin) já existe, definindo-a caso não exista.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | // This doesn't check for existence of "myApplication" in  // the global namespace. Bad practice as we can easily  // clobber an existing variable/namespace with the same name  var myApplication = {};    // The following options \*do\* check for variable/namespace existence.  // If already defined, we use that instance, otherwise we assign a new  // object literal to myApplication.  //  // Option 1: var myApplication = myApplication || {};  // Option 2: if( !MyApplication ){ MyApplication = {} };  // Option 3: window.myApplication || ( window.myApplication = {} );  // Option 4: var myApplication = $.fn.myApplication = function() {};  // Option 5: var myApplication = myApplication === undefined ? {} : myApplication; |

Freqüentemente, veremos os desenvolvedores optando pela Opção 1 ou Opção 2 - ambas são fáceis de entender e são equivalentes em termos de resultado final.

A opção 3 pressupõe que estamos trabalhando no namespace global, mas também pode ser escrita como:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1 | myApplication || (myApplication = {}); |

Esta variação assume que myApplicationjá foi inicializado e, portanto, só é realmente útil para um cenário de parâmetro / argumento como no exemplo a seguir:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | function foo() {    myApplication || ( myApplication = {} );  }    // myApplication hasn't been initialized,  // so foo() throws a ReferenceError    foo();    // However accepting myApplication as an  // argument    function foo( myApplication ) {    myApplication || ( myApplication = {} );  }    foo();    // Even if myApplication === undefined, there is no error  // and myApplication gets set to {} correctly |

As opções 4 podem ser úteis para escrever plug-ins jQuery onde:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // If we were to define a new plugin..  var myPlugin = $.fn.myPlugin = function() { ... };    // Then later rather than having to type:  $.fn.myPlugin.defaults = {};    // We can do:  myPlugin.defaults = {}; |

Isso resulta em uma melhor compactação (minificação) e pode economizar nas pesquisas de escopo.

A opção 5 é um pouco semelhante à opção 4, mas é um formato longo que avalia se myApplicationestá undefinedembutido de forma que seja definido como um objeto se não, caso contrário, é definido como um valor existente para myApplicationse assim for.

É mostrado apenas para ser completo, mas na maioria das situações, as opções 1-4 serão mais do que suficientes para a maioria das necessidades.

É claro que há uma grande variação em como e onde os literais de objeto são usados ​​para organizar e estruturar o código. Para aplicativos menores que desejam expor uma API aninhada para um determinado módulo auto-fechado, podemos apenas nos encontrar usando o Padrão de Módulo Revelador, que abordamos anteriormente no livro:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | var namespace = (function () {        // defined within the local scope      var privateMethod1 = function () { /\* ... \*/ },          privateMethod2 = function () { /\* ... \*/ },          privateProperty1 = "foobar";      return {        // the object literal returned here can have as many      // nested depths as we wish, however as mentioned,      // this way of doing things works best for smaller,      // limited-scope applications in my personal opinion      publicMethod1: privateMethod1,        // nested namespace with public properties      properties: {          publicProperty1: privateProperty1      },        // another tested namespace      utils: {          publicMethod2: privateMethod2      }      ...    }  })(); |

O benefício dos literais de objeto é que eles nos oferecem uma sintaxe de chave / valor muito elegante para trabalhar; aquele em que somos capazes de encapsular facilmente qualquer lógica ou funcionalidade distinta para nosso aplicativo de uma forma que o separe claramente dos outros e forneça uma base sólida para estender nosso código.

Uma possível desvantagem, entretanto, é que os literais de objeto têm o potencial de crescer em construções sintáticas longas. Optar por tirar proveito do padrão de namespace aninhado (que também usa o mesmo padrão de sua base)

Esse padrão também tem várias outras aplicações úteis. Além do namespacing, geralmente é benéfico desacoplar a configuração padrão de nosso aplicativo em uma única área que pode ser facilmente modificada sem a necessidade de pesquisar em toda a nossa base de código apenas para alterá-los - literais de objeto funcionam muito bem para esse propósito. Aqui está um exemplo de um literal de objeto hipotético para configuração:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | var myConfig = {      language: "english",      defaults: {      enableGeolocation: true,      enableSharing: false,      maxPhotos: 20    },      theme: {      skin: "a",      toolbars: {        index: "ui-navigation-toolbar",        pages: "ui-custom-toolbar"      }    }  } |

Observe que JSON é um subconjunto da notação literal de objeto e, na verdade, existem apenas pequenas diferenças sintáticas entre ele e o anterior (por exemplo, as chaves JSON devem ser strings). Se, por algum motivo, você desejar usar JSON para armazenar dados de configuração (por exemplo, para armazenamento mais simples ao enviar para o back-end), sinta-se à vontade para. Para saber mais sobre o padrão literal do objeto, recomendo a leitura do excelente [artigo de](http://rmurphey.com/blog/2009/10/15/using-objects-to-organize-your-code/) Rebecca Murphey sobre o assunto, pois ela cobre algumas áreas que não abordamos.

4. Namespacing aninhado

Uma extensão do padrão literal do objeto é o namespacing aninhado. É outro padrão comum usado que oferece um risco menor de colisão devido ao fato de que, mesmo que um namespace já exista, é improvável que os mesmos filhos aninhados existam.

Isso parece familiar?

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1 | YAHOO.util.Dom.getElementsByClassName("test"); |

Versões mais antigas da biblioteca YUI do Yahoo! Usam o padrão de namespacing de objeto aninhado regularmente. Durante meu tempo como engenheiro na AOL, também usamos esse padrão em muitas de nossas aplicações maiores. Um exemplo de implementação de namespacing aninhado pode ter a seguinte aparência:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | var myApp = myApp || {};    // perform a similar existence check when defining nested  // children  myApp.routers = myApp.routers || {};  myApp.model = myApp.model || {};  myApp.model.special = myApp.model.special || {};    // nested namespaces can be as complex as required:  // myApp.utilities.charting.html5.plotGraph(/\*..\*/);  // myApp.modules.financePlanner.getSummary();  // myApp.services.social.facebook.realtimeStream.getLatest(); |

**Nota: O acima é diferente de como o YUI3 aborda o namespacing, pois os módulos usam um objeto de host de API em sandbox com um namespacing muito menor e mais superficial.**

Também podemos optar por declarar novos namespaces / propriedades aninhados como propriedades indexadas da seguinte maneira:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3 | myApp["routers"] = myApp["routers"] || {};  myApp["models"] = myApp["models"] || {};  myApp["controllers"] = myApp["controllers"] || {}; |

Ambas as opções são legíveis, organizadas e oferecem uma maneira relativamente segura de namespacing nosso aplicativo de uma maneira semelhante ao que estamos acostumados em outras linguagens. A única ressalva real, no entanto, é que ele exige que o mecanismo JavaScript de nosso navegador primeiro localize o objeto myApp e, em seguida, investigue até chegar à função que realmente queremos usar.

Isso pode significar uma quantidade maior de trabalho para realizar pesquisas, no entanto, desenvolvedores como [Juriy Zaytsev](https://twitter.com/kangax) testaram anteriormente e descobriram que as diferenças de desempenho entre o namespacing de um único objeto e a abordagem "aninhada" são bastante insignificantes.

5. Expressões de função imediatamente invocadas (IIFE) s

No início do livro, cobrimos resumidamente o conceito de um [IIFE](http://benalman.com/news/2010/11/immediately-invoked-function-expression/) (expressão de função imediatamente invocada) que é efetivamente uma função sem nome, invocada imediatamente após ter sido definida. Se parece familiar, é porque você pode ter visto anteriormente como uma função anônima autoexecutável (ou auto-invocada), no entanto, eu pessoalmente acho que a nomenclatura IIFE de Ben Alman é mais precisa. Em JavaScript, como as variáveis ​​e funções explicitamente definidas em tal contexto só podem ser acessadas dentro dele, a invocação de função fornece um meio fácil de obter privacidade.

IIFEs são uma abordagem popular para encapsular a lógica do aplicativo para protegê-lo do namespace global, mas também têm seu uso no mundo do namespacing.

Exemplos de IIFEs podem ser encontrados abaixo:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5 | // an (anonymous) immediately-invoked function expression  (function () { /\*...\*/ })();    // a named immediately-invoked function expression  (function foobar () { /\*..\*/ })(); |

Exemplos de funções autoexecutáveis, que são bastante diferentes dos IIFEs, podem ser encontrados abaixo:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5 | // named self-executing function  function foobar () { foobar(); }    // anonymous self-executing function  var foobar = function () { arguments.callee(); } |

De volta aos IIFEs, uma versão um pouco mais expandida do primeiro exemplo do IIFE pode ser semelhante a:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | var namespace = namespace || {};    // here a namespace object is passed as a function  // parameter, where we assign public methods and  // properties to it  (function( o ) {    o.foo = "foo";    o.bar = function() {      return "bar";    };  })( namespace );    console.log( namespace ); |

Embora legível, este exemplo pode ser significativamente expandido para abordar questões comuns de desenvolvimento, como níveis definidos de privacidade (funções e variáveis ​​públicas / privadas), bem como extensão de espaço de nomes conveniente. Vamos examinar mais alguns códigos:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | // namespace (our namespace name) and undefined are passed here  // to ensure 1. namespace can be modified locally and isn't  // overwritten outside of our function context  // 2. the value of undefined is guaranteed as being truly  // undefined. This is to avoid issues with undefined being  // mutable pre-ES5.    ;(function ( namespace, undefined ) {      // private properties    var foo = "foo",        bar = "bar";      // public methods and properties    namespace.foobar = "foobar";      namespace.say = function ( msg ) {      speak( msg );    };      namespace.sayHello = function () {      namespace.say( "hello world" );    };      // private method    function speak(msg) {      console.log( "You said: " + msg );    };      // check to evaluate whether "namespace" exists in the    // global namespace - if not, assign window.namespace an    // object literal    })( window.namespace = window.namespace || {} );      // we can then test our properties and methods as follows    // public    // Outputs: foobar  console.log( namespace.foobar );    // Outputs: You said: hello world  namespace.sayHello();    // assigning new properties  namespace.foobar2 = "foobar";    // Outputs: foobar  console.log( namespace.foobar2 ); |

A extensibilidade é, obviamente, a chave para qualquer padrão de namespacing escalável e os IIFEs podem ser usados ​​para conseguir isso com bastante facilidade. No exemplo abaixo, nosso "namespace" é mais uma vez passado como um argumento para nossa função anônima e é então estendido (ou decorado) com outras funcionalidades:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | // let's extend the namespace with new functionality  (function( namespace, undefined ) {      // public method    namespace.sayGoodbye = function () {      namespace.say( "goodbye" );    }  })( window.namespace = window.namespace || {});    // Outputs: goodbye  namespace.sayGoodbye(); |

Se você quiser saber mais sobre esse padrão, recomendo a leitura do [post IIFE](http://benalman.com/news/2010/11/immediately-invoked-function-expression/) de Ben para obter mais informações.

6. Injeção de namespace

A injeção de namespace é outra variação do IIFE, onde "injetamos" os métodos e propriedades de um namespace específico de dentro de um wrapper de função usando *-o* como um proxy de namespace. O benefício que esse padrão oferece é a fácil aplicação do comportamento funcional a vários objetos ou namespaces e pode ser útil ao aplicar um conjunto de métodos básicos a serem construídos posteriormente (por exemplo, getters e setters).

As desvantagens desse padrão são que pode haver abordagens mais fáceis ou mais ideais para atingir esse objetivo (por exemplo, extensão / fusão de objetos profundos), que abordarei anteriormente nesta seção.

Abaixo, podemos ver um exemplo desse padrão em ação, onde o usamos para preencher o comportamento de dois namespaces: um definido inicialmente (utils) e outro que criamos dinamicamente como parte da atribuição de funcionalidade para utils (um novo namespace chamado *ferramentas* ).

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | var myApp = myApp || {};  myApp.utils = {};    (function () {    var val = 5;      this.getValue = function () {      return val;    };      this.setValue = function( newVal ) {      val = newVal;    }      // also introduce a new sub-namespace    this.tools = {};    }).apply( myApp.utils );    // inject new behaviour into the tools namespace  // which we defined via the utilities module    (function () {      this.diagnose = function() {        return "diagnosis";      }  }).apply( myApp.utils.tools );    // note, this same approach to extension could be applied  // to a regular IIFE, by just passing in the context as  // an argument and modifying the context rather than just  // "this"    // Usage:    // Outputs our populated namespace  console.log( myApp );    // Outputs: 5  console.log( myApp.utils.getValue() );    // Sets the value of `val` and returns it  myApp.utils.setValue( 25 );  console.log( myApp.utils.getValue() );    // Testing another level down  console.log( myApp.utils.tools.diagnose() ); |

Angus Croll também sugeriu [anteriormente](https://msdn.microsoft.com/en-us/scriptjunkie/gg578608) a ideia de usar a API de chamada para fornecer uma separação natural entre contextos e argumentos. Esse padrão pode parecer muito mais com um criador de módulo, mas como os módulos ainda oferecem uma solução de encapsulamento, vamos abordá-la brevemente para fins de detalhamento:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | // define a namespace we can use later  var ns = ns || {},      ns2 = ns2 || {};    // the module/namespace creator  var creator = function( val ) {      var val = val || 0;      this.next = function () {      return val++    };      this.reset = function () {      val = 0;    };  };    creator.call( ns );    // ns.next, ns.reset now exist  creator.call( ns2, 5000 );    // ns2 contains the same methods  // but has an overridden value for val  // of 5000 |

Conforme mencionado, esse tipo de padrão é útil para atribuir um conjunto básico de funcionalidade semelhante a vários módulos ou namespaces. No entanto, gostaria apenas de sugerir usá-lo onde declarar explicitamente a funcionalidade dentro de um objeto / encerramento para acesso direto não faz sentido.

Padrões de namespacing avançados

Exploraremos agora alguns padrões e utilitários avançados que considero inestimáveis ​​ao trabalhar em aplicativos maiores - alguns dos quais exigiram uma revisão das abordagens tradicionais de namespacing de aplicativos. Observarei que não estou defendendo nenhum dos itens a seguir como *o* caminho para o namespace, mas sim maneiras que descobri que funcionam na prática.

Automatizando namespacing aninhado

Como revisamos, os namespaces aninhados podem fornecer uma hierarquia organizada de estrutura para uma unidade de código. Um exemplo de tal namespace poderia ser o seguinte: *application.utilities.drawing.canvas.2d* . Isso também pode ser expandido usando o padrão literal do objeto para ser:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | var application = {    utilities: {      drawing: {        canvas: {          2d:{            //...          }        }      }    }  }; |

Um dos desafios óbvios com esse padrão é que cada camada adicional que desejamos criar requer ainda outro objeto a ser definido como filho de algum pai em nosso namespace de nível superior. Isso pode se tornar particularmente trabalhoso quando várias profundidades são necessárias conforme nossa aplicação aumenta em complexidade.

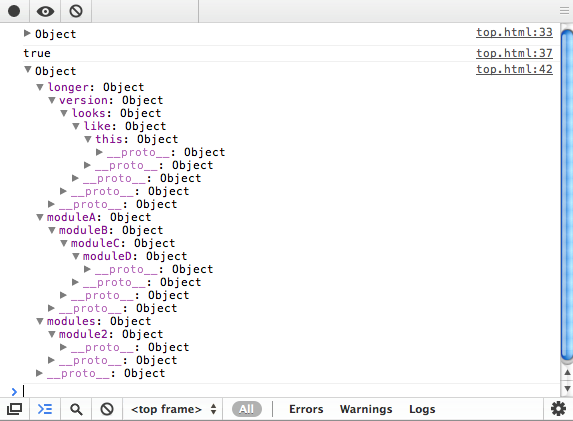
Como esse problema pode ser melhor resolvido? Em [JavaScript Patterns](https://www.amazon.com/JavaScript-Patterns-Stoyan-Stefanov/dp/0596806752) , [Stoyan Stefanov](http://jspatterns.com/) apresenta uma abordagem muito inteligente para definir automaticamente namespaces aninhados em uma variável global existente. Ele sugere um método de conveniência que usa um único argumento de string para um ninho, analisa isso e preenche automaticamente nosso namespace base com os objetos necessários.

O método que ele sugere usar é o seguinte, que eu atualizei para ser uma função genérica para facilitar a reutilização com vários namespaces:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | // top-level namespace being assigned an object literal  var myApp = myApp || {};    // a convenience function for parsing string namespaces and  // automatically generating nested namespaces  function extend( ns, ns\_string ) {    var parts = ns\_string.split("."),        parent = ns,        pl;      pl = parts.length;      for ( var i = 0; i < pl; i++ ) {      // create a property if it doesn't exist      if ( typeof parent[parts[i]] === "undefined" ) {        parent[parts[i]] = {};      }        parent = parent[parts[i]];    }      return parent;  }    // Usage:  // extend myApp with a deeply nested namespace  var mod = extend(myApp, "modules.module2");    // the correct object with nested depths is output  console.log(mod);    // minor test to check the instance of mod can also  // be used outside of the myApp namesapce as a clone  // that includes the extensions    // Outputs: true  console.log(mod == myApp.modules.module2);    // further demonstration of easier nested namespace  // assignment using extend  extend(myApp, "moduleA.moduleB.moduleC.moduleD");  extend(myApp, "longer.version.looks.like.this");  console.log(myApp); |

Resultado do inspetor da web:



Onde antes era necessário declarar explicitamente os vários ninhos de seu namespace como objetos, isso agora pode ser feito facilmente usando uma única linha de código mais limpa.

Padrão de declaração de dependência

Agora vamos explorar um pequeno aumento no padrão de namespacing aninhado, ao qual nos referiremos como padrão de declaração de dependência. Todos nós sabemos que referências locais a objetos podem diminuir o tempo geral de pesquisa, mas vamos aplicar isso ao namespacing para ver como ficaria na prática:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | // common approach to accessing nested namespaces  myApp.utilities.math.fibonacci( 25 );  myApp.utilities.math.sin( 56 );  myApp.utilities.drawing.plot( 98,50,60 );    // with local/cached references  var utils = myApp.utilities,  maths = utils.math,  drawing = utils.drawing;    // easier to access the namespace  maths.fibonacci( 25 );  maths.sin( 56 );  drawing.plot( 98, 50,60 );    // note that the above is particularly performant when  // compared to hundreds or thousands of calls to nested  // namespaces vs. a local reference to the namespace |

Trabalhar com uma variável local aqui é quase sempre mais rápido do que trabalhar com um global de nível superior (por exemplo, myApp). Também é mais conveniente e tem mais desempenho do que acessar propriedades / subnomespaces aninhados em cada linha subsequente e pode melhorar a legibilidade em aplicativos mais complexos.

Stoyan recomenda declarar namespaces localizados exigidos por uma função ou módulo no topo de nosso escopo de função (usando o padrão de variável única) e chama isso de padrão de declaração de dependência. Um dos benefícios que isso oferece é uma redução na localização de dependências e resolvê-las, caso tenhamos uma arquitetura extensível que carregue módulos dinamicamente em nosso namespace quando necessário.

Na minha opinião, esse padrão funciona melhor ao trabalhar em um nível modular, localizando um namespace a ser usado por um grupo de métodos. A localização de namespaces em um nível por função, especialmente onde há sobreposição significativa entre as dependências de namespace, seria algo que eu recomendaria evitar sempre que possível. Em vez disso, defina-o mais acima e faça com que todos acessem a mesma referência.

Extensão de objeto profundo

Uma abordagem alternativa ao namespacing automático é a extensão de objeto profundo. Os namespaces definidos usando a notação literal de objeto podem ser facilmente estendidos (ou mesclados) com outros objetos (ou namespaces) de forma que as propriedades e funções de ambos os namespaces possam ser acessíveis no mesmo namespace pós-mesclagem.

Isso é algo que se tornou bastante fácil de realizar com estruturas JavaScript modernas (por exemplo, consulte [$ .extend](https://api.jquery.com/jQuery.extend/) da jQuery ), no entanto, se estiver procurando estender o objeto (namespaces) usando o JS vanilla, a rotina a seguir pode ser útil.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83 | // extend.js  // Written by Andrew Dupont, optimized by Addy Osmani    function extend( destination, source ) {      var toString = Object.prototype.toString,        objTest = toString.call({});      for ( var property in source ) {      if ( source[property] && objTest === toString.call(source[property]) ) {        destination[property] = destination[property] || {};        extend(destination[property], source[property]);      } else {        destination[property] = source[property];      }    }      return destination;  };    console.group( "objExtend namespacing tests" );    // define a top-level namespace for usage  var myNS = myNS || {};    // 1. extend namespace with a "utils" object  extend(myNS, {    utils:{      //...    }  });    console.log( "test 1", myNS);  // myNS.utils now exists    // 2. extend with multiple depths (namespace.hello.world.wave)  extend(myNS, {    hello: {      world: {        wave: {          test: function() {            //...          }        }      }    }  });    // test direct assignment works as expected  myNS.hello.test1 = "this is a test";  myNS.hello.world.test2 = "this is another test";  console.log( "test 2", myNS );    // 3. what if myNS already contains the namespace being added  // (e.g. "library")? we want to ensure no namespaces are being  // overwritten during extension    myNS.library = {    foo:function () {}  };    extend( myNS, {    library: {      bar: function() {        //...      }    }  });    // confirmed that extend is operating safely (as expected)  // myNS now also contains library.foo, library.bar  console.log( "test 3", myNS );    // 4. what if we wanted easier access to a specific namespace without having  // to type the whole namespace out each time?    var shorterAccess1 = myNS.hello.world;  shorterAccess1.test3 = "hello again";  console.log( "test 4", myNS);    //success, myApp.hello.world.test3 is now "hello again"    console.groupEnd(); |

**Nota:** A implementação acima não é compatível com navegadores cruzados para todos os objetos e deve ser considerada apenas uma prova de conceito. Pode-se achar o extend()método Underscore.js uma implementação mais simples e cruzada para navegadores para começar com [http://documentcloud.github.io/underscore/docs/underscore.html#section-82](https://documentcloud.github.io/underscore/docs/underscore.html#section-82) .

Para desenvolvedores que vão usar jQuery em seus aplicativos, pode-se obter exatamente a mesma extensibilidade de namespace de objeto com $.extendo seguinte:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | // top-level namespace  var myApp = myApp || {};    // directly assign a nested namespace  myApp.library = {    foo: function() {      //...    }  };    // deep extend/merge this namespace with another  // to make things interesting, let's say it's a namespace  // with the same name but with a different function  // signature: $.extend( deep, target, object1, object2 )  $.extend( true, myApp, {    library:{      bar: function(){        //...      }    }  });    console.log("test", myApp);  // myApp now contains both library.foo() and library.bar() methods  // nothing has been overwritten which is what we're hoping for. |

Para fins de detalhamento, consulte [aqui](https://gist.github.com/1221980) jQuery $ .extend equivalentes ao restante dos experimentos de namespacing encontrados nesta seção.

Recomendação

Revendo os padrões de namespace que exploramos nesta seção, a opção que eu usaria pessoalmente para a maioria dos aplicativos maiores é o namespace de objeto aninhado com o padrão literal de objeto. Sempre que possível, eu implementaria isso usando namespacing automatizado aninhado, no entanto, essa é apenas uma preferência pessoal.

IIFEs e variáveis ​​globais únicas podem funcionar bem para aplicativos no intervalo pequeno a médio, no entanto, bases de código maiores que requerem namespaces e sub-namespaces profundos requerem uma solução sucinta que promova legibilidade e escalabilidade. Acho que esse padrão atinge bem todos esses objetivos.

Eu também recomendaria experimentar alguns dos métodos de utilitário avançado sugeridos para extensão de namespace, pois eles realmente podem nos poupar tempo a longo prazo.

Conclusões

É isso para esta aventura introdutória ao mundo dos padrões de design em JavaScript e jQuery - espero que você tenha achado benéfico.

Os padrões de design tornam mais fácil construir sobre os ombros de desenvolvedores que definiram soluções para problemas e arquiteturas desafiadoras ao longo de várias décadas. O conteúdo deste livro deve fornecer informações suficientes para começar a usar os padrões que cobrimos em seus próprios scripts, plug-ins e aplicativos da web.

É importante que estejamos cientes desses padrões, mas também é essencial saber como e quando usá-los. Estude os prós e os contras de cada padrão antes de empregá-los. Reserve um tempo para experimentar os padrões para apreciar totalmente o que eles oferecem e fazer julgamentos de uso com base no verdadeiro valor de um padrão para sua aplicação.

Se eu encorajei ainda mais seu interesse nesta área e você gostaria de aprender mais sobre padrões de projeto, há uma série de títulos excelentes nesta área disponíveis para desenvolvimento de software genérico e, claro, JavaScript.

Fico feliz em recomendar:

1. " [Patterns Of Enterprise Application Architecture](https://www.amazon.com/Patterns-Enterprise-Application-Architecture-Martin/dp/0321127420) ", de Martin Fowler
2. " [Padrões JavaScript](https://www.amazon.com/JavaScript-Patterns-Stoyan-Stefanov/dp/0596806752/ref=sr_1_1?ie=UTF8&s=books&qid=1289759956&sr=1-1) " por Stoyan Stefanov

Obrigado por ler *Learning JavaScript Design Patterns* . Para obter mais material educacional sobre como aprender JavaScript, sinta-se à vontade para ler mais sobre mim em meu blog em [http://addyosmani.com](https://addyosmani.com/) ou no Twitter [@addyosmani](https://twitter.com/addyosmani) .

Até a próxima, muita sorte em suas aventuras em JavaScript!

Referências

1. Princípios e padrões de design - Robert C Martin <http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf>
2. Ralph Johnson - edição especial da ACM sobre padrões e linguagens de padrões - <http://www.cs.wustl.edu/~schmidt/CACM-editorial.html>
3. Biblioteca Hillside Engineering Design Patterns - <http://hillside.net/patterns/>
4. Padrões de design JavaScript profissionais - Ross Harmes e Dustin Diaz <http://jsdesignpatterns.com/>
5. Definições de padrões de design - [http://en.wikipedia.org/wiki/Design\_Patterns](https://en.wikipedia.org/wiki/Design_Patterns)
6. Padrões e terminologia de software <http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html>
7. Aproveite os benefícios dos Design Patterns - Jeff Juday [http://articles.techrepublic.com.com/5100-10878\_11-5173591.html](https://www.techrepublic.com/article/reap-the-benefits-of-design-patterns-in-software-development/)
8. Padrões de design JavaScript - Subramanyan Guhan <http://www.slideshare.net/rmsguhan/javascript-design-patterns>
9. O que são padrões de design e eu preciso deles? - James Moaoriello <http://www.developer.com/design/article.php/1474561>
10. Padrões de design de software - Alex Barnett <http://alexbarnett.net/blog/archive/2007/07/20/software-design-patterns.aspx>
11. Avaliando Padrões de Projeto de Software - Gunni Rode <http://www.rode.dk/thesis/>
12. SourceMaking Design Patterns [http://sourcemaking.com/design\_patterns](https://sourcemaking.com/design_patterns)
13. Padrões de JavaScript - Stoyan Stevanov - <http://www.slideshare.net/stoyan/javascript-patterns>
14. Stack Overflow - Implementações de padrões de design em JavaScript (discussão) [http://stackoverflow.com/questions/24642/what-are-some-examples-of-design-pattern-implementations-using-javascript](https://stackoverflow.com/questions/24642/what-are-some-examples-of-design-pattern-implementations-using-javascript)
15. Os elementos de um padrão de projeto - Jared Spool [http://www.uie.com/articles/elements\_of\_a\_design\_pattern/](https://www.uie.com/articles/elements_of_a_design_pattern/)
16. Stack Overflow - Exemplos de padrões de design JS práticos (discussão) [http://stackoverflow.com/questions/3722820/examples-of-practical-javascript-object-oriented-design-patterns](https://stackoverflow.com/questions/3722820/examples-of-practical-javascript-object-oriented-design-patterns)
17. Padrões de projeto em JavaScript Parte 1 - Nicholas Zakas <http://www.webreference.com/programming/javascript/ncz/column5/>
18. Stack Overflow - Padrões de design em jQuery [http://stackoverflow.com/questions/3631039/design-patterns-used-in-the-jquery-library](https://stackoverflow.com/questions/3631039/design-patterns-used-in-the-jquery-library)
19. Classificando padrões de projeto por AntiClue - Elyse Neilson <http://www.anticlue.net/archives/000198.htm>
20. Padrões de projeto, linguagens de padrões e estruturas - Douglas Schmidt <http://www.cs.wustl.edu/~schmidt/patterns.html>
21. Mostre amor ao padrão do módulo - Christian Heilmann <http://www.wait-till-i.com/2007/07/24/show-love-to-the-module-pattern/>
22. Projetos de software simplificados - Anoop Mashudanan <http://www.scribd.com/doc/16352479/Software-Design-Patterns-Made-Simple>
23. Introdução ao padrão de módulo JavaScript <https://www.unleashed-technologies.com/blog/2010/12/09/introduction-javascript-module-design-pattern>
24. Padrões de design explicados - <http://c2.com/cgi/wiki?DesignPatterns>
25. Mixins explicado [http://en.wikipedia.org/wiki/Mixin](https://en.wikipedia.org/wiki/Mixin)
26. Trabalhando com os padrões de design do GoF em JavaScript <http://aspalliance.com/1782_Working_with_GoFs_Design_Patterns_in_JavaScript_Programming.all>
27. Usando Object.create [http://stackoverflow.com/questions/2709612/using-object-create-instead-of-new](https://stackoverflow.com/questions/2709612/using-object-create-instead-of-new)
28. Padrões de design JavaScript de t3knomanster - [http://t3knomanser.livejournal.com/922171.html](https://t3knomanser.livejournal.com/922171.html)
29. Vantagens do JavaScript - literais de objeto [http://stackoverflow.com/questions/1600130/javascript-advantages-of-object-literal](https://stackoverflow.com/questions/1600130/javascript-advantages-of-object-literal)
30. Padrões de classe JavaScript - Liam McLennan <http://geekswithblogs.net/liammclennan/archive/2011/02/06/143842.aspx>
31. Entendendo proxies em jQuery - [http://stackoverflow.com/questions/4986329/understanding-proxy-in-jquery](https://stackoverflow.com/questions/4986329/understanding-proxy-in-jquery)
32. Observer Pattern Using JavaScript - <http://www.codeproject.com/Articles/13914/Observer-Design-Pattern-Using-JavaScript>
33. Falando no padrão Observer - <http://www.javaworld.com/javaworld/javaqa/2001-05/04-qa-0525-observer.html>

Aprendendo Padrões de Design JavaScript. © Addy Osmani 2017.

[volte para o código.Fechar.](https://addyosmani.com/resources/essentialjsdesignpatterns/book/index.html)