



JavaScript at Scale

大型JavaScript应用 最佳实践指南

以一线前沿JavaScript开发者对可扩展性深刻的洞悉力，构建历久弥新的JavaScript应用

[加] Adam Boduch 著
奇舞团 译



中国工信出版集团



电子工业出版社
Publishing House of Electronics Industry
www.eip.com.cn

JavaScript at Scale

大型JavaScript应用 最佳实践指南

[加] Adam Boduch 著
奇舞团 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书以介绍扩展 JavaScript 的特殊性，及影响其可扩展性的因素作为开头，逐步深入地介绍了组件的复合与通信、寻址与导航、用户偏好与默认设置、加载时间和响应速度、可移植性和测试、缩小规模、错误处理等大型 JavaScript 应用中的实践经验。本书将教会你如何在真实项目中扩展 JavaScript 应用，设计出灵活的架构。书中的每个主题都涵盖了实践指导，帮助你将在知识运用到实际项目中。

Copyright © 2016 Packt Publishing. First published in the English language under the title ‘JavaScript at Scale’.

本书简体中文版专有出版权由 Packt Publishing 授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号 图字：01-2015-7447

图书在版编目（CIP）数据

大型 JavaScript 应用最佳实践指南 / (加) 亚当·博达哈 (Adam Boduch) 著；奇舞团译. —北京：电子工业出版社，2017.2

书名原文：JavaScript at Scale

ISBN 978-7-121-30706-5

I. ①大… II. ①亚… ②奇… III. ①JAVA 语言—程序设计 IV. ①TP312.8

中国版本图书馆 CIP 数据核字（2016）第 311425 号

策划编辑：张春雨

责任编辑：徐津平

印 刷：三河市鑫金马印装有限公司

装 订：三河市鑫金马印装有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱

邮编：100036

开 本：787×980 1/16 印张：14.75 字数：285 千字

版 次：2017 年 2 月第 1 版

印 次：2017 年 2 月第 1 次印刷

定 价：65.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888, 88258888。

质量投诉请发邮件至 zlt@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819, faq@phei.com.cn。

关于作者

Adam Boduch 在开发大型 JavaScript 应用方面有近 10 年的工作经验。在转型为前端工程师之前，他曾使用 Python 与 Linux 参与了许多大型云计算产品的构建。Adam 拥有非常丰富的开发经验，擅长处理复杂的场景，提高软件的可扩展性。他编写了很多 JavaScript 方面的书籍，其中包括 *Lo-Dash Essentials*，并且，他还热衷于优化用户体验和性能。

Adam 现居住于多伦多，是 Virtustream 的一名高级软件工程师。

我想在此感谢我的妈妈和爸爸。

关于审校者

August N. Marcello III 是一位充满激情的软件工程师，在客户端的 Web 应用架构相关的设计、实现、部署方面，有着近 20 年的工作经验。他专注于基于 SaaS 创造良好的用户体验，并将其传播到 Web 生态系统，这无论从个人还是从专业角度来说都极具价值。对新兴通用技术的热爱以及对先进的 JavaScript 平台的专注，驱动着他在技术上精益求精。在工作之余，他会参加越野跑、山地自行车骑行，或者陪伴家人和朋友。他的个人网站为：
www.augustmarcello.com。

非常感谢 Chuck、Mark、Eric 和 Adam，很荣幸能够跟他们一起工作和学习。
谢谢我的家人、朋友，还有我所经历的一切。

Yogesh Singh 毕业于印度 JSS 技术教育学院。他是一位全栈 Web 开发者，在服务端 Web 开发栈方面（ASP.NET 以及 Node.js）很有经验，而且熟练掌握 HTML、CSS 以及 JavaScript。

Yogesh 热爱 JavaScript 以及相关的库和框架（Backbone、AngularJS、jQuery 和 Underscore）。

他最开始从事的是数据挖掘和数据仓库方面的工作，在数据库开发方面十分专业。他是 MSSQL 的微软认证解决方案成员（MCSA）。

Yogesh 自学能力很强，喜欢学习算法和数据结构，并在斯坦福大学 Coursera 上获得了算法课的结业证明。

他曾就职于 OLX India 和 MAQ Software，目前为 Gainsight 公司的全栈开发者。

业余时间，他喜欢在 <http://mylearning.in> 上写博客。他的 LinkedIn 简历地址为 <https://www.linkedin.com/in/yogesh21>。

感谢我的家人、朋友以及同事的支持。

Nikolay Sokolov 是一名软件工程师，他在云计算、自动化部署和企业软件开发方面有着丰富的经验。现在就职于 Tonomi (<http://tonomi.com/>)，负责基于弹性组件模型分发云应用的自动管理包。

可通过 <https://twitter.com/chemikadze> 随时联系他。

Serkan Yersen 是一名洛杉矶的软件开发者。他是一些开源库的作者，例如：[ifvisible.js](#)、[underscore.py](#) 以及 [kwargs.js](#)。Serkan 专门从事构建大型 JavaScript 应用，以及为用户广泛的应用创建 UI。2006 年至 2012 年，就职于 <http://www.jotform.com/> 期间，他开发了一个复杂的表单生成器，供上百万用户使用。现在，他就职于 Home Depot 和 Redbeacon (<http://www.redbeacon.com/>)，负责 Web 应用开发。你可以访问他的个人网站：<http://serkan.io/>。

关于译者

本书翻译工作由月影领衔的奇舞团翻译小组承担,由王伟华、黄小璐、黄薇负责翻译。

王伟华

网名 Aztack, 前端技术专家。曾就职百度、奇虎 360 等国内知名互联网公司。拥有丰富的 Web 前端开发经验, 擅长 JavaScript、Ruby、Java、C++ 等语言。

个人博客: <https://aztack.wang>

黄小璐

毕业于华中科技大学计算机学院。现为奇虎 360 软件开发工程师。曾参与开源项目 [stcjs](<https://github.com/stcjs/stc>) (高性能前端工作流系统)。参与翻译了《高性能 HTML5》等书。

黄薇

毕业于中山大学, 于 2013 年加入奇舞团, 近期参与了 Nova.js (Web Component 框架)、声享 (在线制作 PPT) 等项目, 对大型 JavaScript 应用有浓厚的兴趣和丰富的开发经验。

以上三位译者曾共同参与《移动 Web 手册》一书的翻译工作。

读者可扫描以下二维码关注奇舞团周刊。



前言

能够一直正常运行的应用只是特例，大部分的 JavaScript 应用多多少少都有些问题。而这些问题产生的原因是由于我们总是习惯性地忽略可扩展性。这本书介绍了如何通过扩展前端架构来提高软件质量。扩展 JavaScript 应用是一件有趣的事情，需要考虑很多因素：用户、开发者、开发环境、浏览器环境等。我们的任务就是全面考虑这些因素，从而提供最佳的用户体验。我们要扩展什么？为何要扩展？本书将为大家解答这些问题。

本书内容

第 1 章，扩展 JavaScript 应用，介绍了何为可扩展的 JavaScript 应用，以及扩展 JavaScript 应用与扩展其他应用的区别。

第 2 章，可扩展性的影响因素，介绍了如何理解可扩展的需求，设计出更好的架构。

第 3 章，组件组合，介绍了构成架构核心的模式，以及如何以之为蓝图组合组件。

第 4 章，组件的通信与职责，介绍了组件之间的通信是如何制约扩展的。组件的通信模式功能有决定性作用。

第 5 章，寻址和导航，详细介绍了拥有指向不同资源的 URI 的大型 Web 应用，以及如何设计才能应对不断增长的 URI 数量。

第 6 章，用户偏好和默认设置，介绍了设置用户偏好的必要性，以及可配置的组件对扩展应用的重要性。

第 7 章，加载时间和响应速度，介绍了文件数量的增加是如何降低应用效率的。在添加新功能时，要有所舍弃，才能保证 UI 的响应速度。

第 8 章，可移植性和测试，介绍了如何编写不依赖于特定环境的 JavaScript 代码，包

括创建可移植的模拟数据和测试代码。

第 9 章，缩小规模，介绍了移除无用或错误组件对扩展系统其他部分的重要性。

第 10 章，处理错误，介绍了优秀的 JavaScript 架构不会因为某个组件的错误而崩溃。许多时候，在设计时充分考虑对错误的处理是成功扩展的关键。

阅读本书的条件

- NodeJS
- 代码编辑器/集成开发环境
- 一个现代 Web 浏览器

本书读者

本书的目标读者是对前端架构感兴趣的高级 JavaScript 工程师。阅读本书无须预备框架知识，但本书介绍的大部分概念都来自于框架，例如 Backbone、Angular、Ember。阅读本书需要扎实的 JavaScript 语言知识基础，本书中所有的示例代码都使用 ECMAScript 6 语法编写。

约定


在阅读本书时，你会发现许多代表不同类型信息的不同文本样式。这里展示了一些例子及其含义详解。


正文中的代码、数据库表名、文件夹名、文件名、文件后缀、路径、URI 示例、用户输入，以及 Twitter 用户名等均按此格式进行展示：“以 `users/31729` 为例。路由器应该找到一个模式，能够匹配该字符串，并能够提取出变量 `31729`”

代码块按以下格式展示：

```
// 渲染试图的各个部分。
```

```
// 各部分可能有renderer，也可能没有。  
// 但不管有没有renderer，内容都会被返回。
```

 警告和重要提示都会按此格式展示。

 提示和技巧都会按此格式展示。

下载示例代码

你可以从 <http://www.broadview.com.cn> 下载所有已购买的博文视点书籍的示例代码文件。

勘误表

虽然我们已经尽力确保内容的准确性，但错误仍然可能存在。如发现任何错误，可登录博文视点官网 <http://www.broadview.com.cn> 提交勘误信息。一旦勘误信息被本书作者或编辑确认，即可获得博文视点奖励积分，可用于兑换电子书。读者可以随时浏览图书页面，查看已发布的勘误信息。

目录

- 1 扩展 JavaScript 应用 1
 - 影响扩展的因素 2
 - 对可扩展的需要 2
 - 不断增长的用户 3
 - 添加新功能 3
 - 雇佣更多的开发者 4
 - 架构角度 5
 - 浏览器是一个独特环境 5
 - 组件设计 7
 - 组件通信 7
 - 加载时间 8
 - 响应性 9
 - 可寻址性 9
 - 可配置性 10
 - 架构性取舍 11
 - 确定不可变内容 11
 - 从开发的便捷性考虑性能 11
 - 性能的可配置性 12
 - 从可替换性考虑性能 13
 - 可寻址性的开发便捷性 13
 - 性能的可维护性 13
 - 减少功能以提高可维护性 14

利用框架.....	15
框架与类库.....	16
一致地实现模式.....	16
内建的性能.....	16
利用社区智慧.....	16
框架并非天生支持扩展.....	17
小结.....	17
2 可扩展性的影响因素.....	19
扩展用户.....	20
许可证费用.....	20
订阅费用.....	21
消耗费用.....	21
广告支持.....	21
开源.....	22
与用户沟通.....	23
支持机制.....	24
反馈机制.....	25
提示用户.....	26
用户维度.....	26
扩展用户示例.....	27
扩展功能.....	28
应用价值.....	28
“杀手级”功能与“杀死”应用的功能.....	29
数据驱动的功能.....	30
与竞品比较.....	30
修改已有的功能.....	31
支持用户分组和角色.....	32
增加新服务.....	32

扩展功能示例	34
开发的可扩展性	34
寻找开发资源	35
开发职责	36
资源过多	36
扩展开发示例	37
影响因素检查表	37
用户检查清单	38
功能清单	39
开发者清单	41
小结	41
3 组件组合	43
通用组件	44
模块	44
路由器	46
模型/集合	50
控制器/视图	53
模板	55
应用特定的组件	56
扩展通用组件	56
识别公用数据、功能	56
扩展路由器组件	57
扩展模型/集合	58
扩展控制器/视图	59
将功能映射到组件	60
通用功能	61
特定功能	61
解构组件	62

维护和调试组件	62
重构复杂组件	64
可插拔的业务逻辑	64
扩展与配置	65
无状态的业务逻辑	65
组织组件代码	66
小结	67
4 组件的通信与职责	69
通信模型	69
消息传递模型	70
事件模型	70
通信数据结构	71
命名约定	71
数据格式	72
公共数据	73
可追踪的组件通信	74
订阅事件	74
全局事件日志	74
事件的生命周期	77
通信的开销	77
事件的频率	78
回调函数执行时间	80
事件复杂度	81
通信责任区	82
后端 API	82
Web Socket 用于更新状态	83
DOM 更新	85
松耦合的通信	86

替换组件	86
应对意外事件	87
组件分层	90
事件流向	90
开发者的职责	91
构建代码思维导图	91
小结	92
5 寻址和导航	93
实现路由的方法	93
Hash URI	94
传统 URI	94
路由是如何工作的	95
路由的职责	95
路由事件	96
URI 的结构和模式	96
编码信息	97
设计 URI	97
将资源映射到 URI	99
手动创建 URI	99
自动生成资源 URI	99
触发路由	103
用户行为	103
重定向用户	104
路由配置	104
静态路由声明	105
注册事件	105
禁用路由	105
故障排查	108

路由器冲突	108
记录初始配置	110
记录路由事件	110
处理非法资源的状态	110
小结	111
6 用户偏好和默认设置	113
偏好类型	113
地区	113
行为	114
外观	115
支持地区	115
决定支持哪些地区	115
维护地区	116
设置地区	116
选择地区	117
存储地区偏好	117
URI 中的地区	118
通用组件配置	118
选择配置的值	119
存储和硬编码默认值	119
对后端的影响	120
加载配置值	121
配置行为	122
启用和禁用组件	122
改变数量	123
改变顺序	124
配置通知	126
行内选项	126

改变外观.....	127
主题工具.....	127
选择一个主题.....	128
单独的样式偏好.....	128
性能影响.....	128
可配置地区的性能.....	129
可配置行为的性能.....	129
可配置主题的性能.....	132
小结.....	132
7 加载时间和响应速度	135
组件构件.....	135
组件依赖.....	135
构建组件.....	136
加载组件.....	137
加载模块.....	137
懒惰的模块加载.....	138
模块加载的延迟.....	139
通信瓶颈.....	141
减少间接引用.....	141
分析代码.....	143
组件优化.....	145
维护状态的组件.....	145
处理副作用.....	146
DOM 渲染技术.....	148
API 数据	150
加载延迟.....	150
处理大数据集.....	151
优化运行时组件.....	152
小结.....	153

8 可移植性和测试	155
与后端解耦	155
模拟后端 API	155
前端入口	156
模拟工具	157
生成模拟数据集	158
执行操作	159
功能设计过程	159
设计 API	160
实现模拟数据	160
实现功能	161
协调模拟数据与真实数据	162
单元测试工具	163
框架自带的工具	163
独立的单元测试工具	164
工具链和自动化	165
测试模拟场景	166
模拟 API 和测试固件	166
场景生成工具	167
端到端测试和持续集成	168
小结	169
9 缩小规模	171
扩展限制	171
JavaScript 文件体积	172
网络带宽	173
内存消耗	175
CPU 消耗	177
后端能力	179

互相矛盾的功能.....	180
重叠的功能.....	181
冗余的功能.....	182
用户需求.....	182
设计失效.....	183
多余的组件.....	184
低效的数据处理.....	186
过度创建标记.....	190
应用组合.....	191
功能的启动.....	191
新功能的影响.....	192
重要的库.....	192
小结.....	193
10 处理错误.....	195
快速失效.....	195
使用质量约束.....	196
提供有意义的反馈.....	196
当无法快速失效时.....	197
容错.....	198
区分关键行为.....	198
探测和控制错误行为.....	199
禁用出错组件.....	202
优雅地降级功能.....	203
故障恢复.....	204
重试失败操作.....	204
重启组件.....	207
用户手动干涉.....	208
当我们无法从故障中恢复.....	209

- 性能和复杂度.....210
 - 异常处理.....210
 - 状态检查.....211
 - 通知其他组件.....211
- 记录日志和调试.....212
 - 有意义的错误日志.....212
 - 为潜在故障发出警告.....213
 - 通知和指导用户.....214
- 改进架构.....214
 - 记录错误场景.....215
 - 改进组件分类.....215
 - 复杂导致出错.....216
- 小结.....216

1

扩展 JavaScript 应用

JavaScript 应用变得越来越庞大。这是因为使用 JavaScript 能做的事情远比我们大多数人认为的要多得多。JavaScript 一开始被设计成一门可以赋予静态页面更多功能、填补 HTML 不足的语言。年复一年，越来越多的网站开始使用 JavaScript 完善其页面功能。

尽管 JavaScript 有一些怪异的语言特性，但这并不妨碍它的流行。如今，它已经是 GitHub (<http://github.info/>) 上最流行的语言。如今网站越来越像桌面应用，类库和框架百花齐放。究其原因，是由于前端 JavaScript 应用已经变得庞大而复杂。

如今前端人员的工具箱里有很多工具。JavaScript 也已经进化，最基本的操作很少依赖类库。下一代 ECMAScript 语言规范更是如此：新加入的语言特性至少部分解决了困扰开发人员多年的问题。当然，应用程序框架仍然必不可少。虽然一直在不断改善之中，但前端开发环境和支撑它的 Web 标准远没有达到完善的程度。

“架构 (Architecture)”是前端开发中一直以来都缺少的。由于近几年 Web 应用日趋复杂，前端架构开始流行起来。成熟的工具使得开发人员可以针对要解决的问题设计出可扩展的架构。而这恰恰是本书的核心所在，即“可扩展的 JavaScript 架构”。那么问题来了，根据什么进行“扩展”呢？这不同于传统计算中的可扩展问题，因为传统情况下你需要在分布式服务器环境下处理更多的负载。前端的可扩展问题有它特有的挑战和限制。本章将会明确前端架构可扩展性面临的一些问题。

影响扩展的因素

我们不会仅仅因为技术上可行，就去考虑软件系统的扩展问题。虽然经常听到对可扩展性的吹捧，但是可扩展性的需求必须与实际相结合。所以，为软件增加可扩展性必须有合理的理由。如果没有扩展的必要，则根本不需要考虑是否划算，构建一个无扩展性的系统就可以了。为一个不需要扩展的系统增加扩展性是不值得的，尤其对最终用户来说，这只会使系统显得更加笨重。

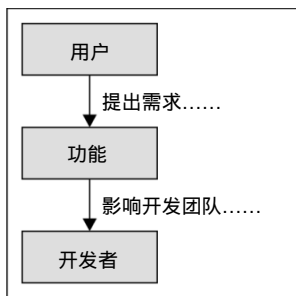
作为 JavaScript 开发者和架构师，必须承认并了解影响扩展性的因素。虽然不是所有 JavaScript 应用都需要扩展，但总有一部分是需要的。比如，我们很难确认某个系统不需要扩展，不需要为它的可扩展性花费时间和精力。除非我们开发的系统不需要后期维护，否则总会有对增长和成功的预期。

从另一方面讲，JavaScript 应用并非天生成熟的可扩展应用，而是逐步积累、进化成的可扩展应用。对于 JavaScript 开发人员来说，后面会讲到的“可扩展性的影响因素”是一个有效的工具。我们不希望一开始就过度设计，更不希望被早期设计绑住手脚，限制了可扩展性。

对可扩展的需要

扩展软件是一种基于反应的活动。考虑可扩展性的影响因素可以帮助我们积极地做出准备。在应用后端等系统中，这种“扩展活动”通常是被自动处理的，可能是短暂的访问高峰。例如，激增的用户请求导致负载骤增，这时负载均衡器介入，将负载均匀地分派到后端服务器。在某些极端情况下，系统可能会在需要时自动准备新的后端资源来应对变化，当不再需要时将这些资源自动销毁。

但是前端不一样，前端的扩展活动通常发生的时间周期较长，而且更加复杂。JavaScript 应用的独特一面在于，浏览器能获得的硬件资源就是它能使用的全部硬件资源，它从后端获取的数据可以很好地按比例增长，但这不是我们需要考虑的。随着软件的不断演进，我们要想成功做点什么，就必须关注“可扩展性的影响因素”。



上图自上而下地展示了可扩展性的影响因素。首先是用户提出软件需要实现的功能，接着功能尺寸、与其他功能的关系等因素会直接影响开发团队的构成，沿着箭头自上而下影响相应地增长。

不断增长的用户

如果构建的应用只服务于一个用户，就没有必要这么大费周章了。基于典型用户的需求来构建的应用将会为更多用户提供服务。所以在应用进化过程中，应该预见到用户的增长。尽管并没有确切的目标用户数量，不过，基于应用自身的特点，仍然可以使用 <http://www.alexa.com/> 这类工具作为基准，设定活跃用户数量的目标值。比如，如果我们的应用是任何人都可以访问的，就会希望有大量的注册用户；但如果仅针对个人安装，那么加入系统的用户数量的增长就会比较缓慢。但即使如此，我们还是希望部署数量不断增加，以提升软件的用户总量。

与前端界面交互的用户数量是扩展应用最大的影响因素。每增加一个用户都伴随着各种架构层面上指数级的增长。如果自上而下地看，用户决定一切。应用的存在终归是为了服务用户。JavaScript 代码越易于扩展，就越能取悦用户。

添加新功能

也许能够取悦用户的功能就是用户基数庞大的成功软件最显而易见的附带产物。软件的功能会随着用户数不断增长，尽管新功能显而易见，但还是经常被忽视。明明知道增加新功能不可避免，但我们还是很少思考如何合理地在代码中实现源源不断的新需求。正是缺少这样的思考，阻碍了我们继续发展。

这在软件交付初期非常棘手。软件开发商会竭尽全力吸引新的用户，但由于初期阶段

能够吸引用户的功能有限,导致收效甚微。没有足够多的成熟特性,没有庞大的开发团队,也没有机会去打破用户习惯。当没有这些限制条件时,比较容易能够实现一些功能让已有或潜在用户感到眼花缭乱。但是我们如何才能早期决策时迫使自己考虑周全?如何才能提供更多功能的前提下确保没有限制我们扩展软件的能力?

本书中你会发现,不管是开发新功能还是增强已有的功能,都是可扩展 JavaScript 架构始终需要考虑的问题。我们需要考虑的不仅仅是软件推广文案中罗列的各种功能,还要考虑这些功能的复杂度、各个功能之间的共性以及各个功能有多少“移动部件(Moving Parts)”。当自上而下审视 JavaScript 架构时,如果用户是第一层级,那么各个功能就是下一个层级。从这个层级开始扩展变得纷繁复杂。

使功能变复杂的,并不是某一个单独用户,而是一群需要这个功能的用户。从这个角度讲,我们不得不思考使用软件的用户特征或者角色,以及哪些功能提供给哪些角色。对这种组织结构的需求一开始并不明显。直到后期,我们先前的决策使得引入基于角色的特性难以实施时,它才会显现出来。并且,这还取决于我们的软件是如何部署的,有时可能需要支持多种不同的用例。比如,可能几个大机构用户,都有各自的部署方案,并且很可能有各自独特的用户结构上的限制。这是十分具有挑战性的,如果希望做到可扩展,架构就需要支持这些组织结构迥然不同的需求。

雇佣更多的开发者

实现软件的各种功能需要可靠的 JavaScript 开发人员,并且他们应该知道自己在做什么。能有一个这样的开发者团队是非常幸运的事情。团队组建不是自发的,在团队可以开发出优秀代码之前,需要在某种程度上建立起彼此之间的信任和尊重。一旦开始,我们就处于一个良好的状态。再看一下前面提到的自上而下的可扩展性影响因素,我们要开发的功能会直接影响团队的健康。这之间的平衡基本上是无法维持的,但是可以尽量接近。缺少人手但又有太多的功能要实现,这会让团队成员倍感压力。当如期交付毫无希望时,大家就不会努力尝试了。另一方面,如果开发人员过多,要开发的功能有限,就会带来更多的沟通负担,而定义职责又很困难,所以当大家对职责没有共识时,离失败就不远了。

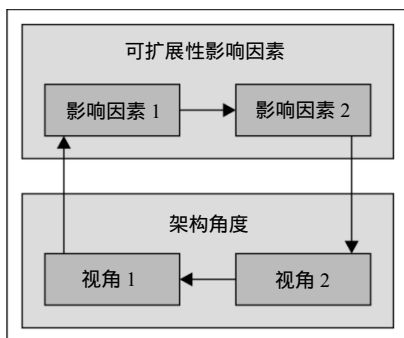
相对于拥有太多的开发人员,开发人员不足反而更易于功能的开发。当面临巨大的功能开发压力时,是一个很好的时机来退后想一想:“如果我们有更多的开发者,会与现在有哪些不同呢?”这个问题经常被忽略掉,直接去招更多的开发者。而让大家惊讶的是,招

聘到新人后功能的产出并没有立竿见影的效果。这就是为什么我们需要一个没有愚蠢问题、责任分配明确的研发文化。

团队组织结构和开发方法并没有定式，开发团队需要有针对性地处理开发中的情况，最大的问题无疑就是功能的数量、规模和复杂度。所以，这些才是我们在建立团队之初，以及团队成长过程用应该考虑的。后一点尤为重要，因为当功能大量增加后，初期的团队结构是无法适应的。

架构角度

前一节举了一些 JavaScript 应用中可扩展性的影响因素。从上到下每个因素都会影响到下面的因素。用户的数量和特点是第一个也是首要的影响因素，它直接影响了我们要开发功能的数量和特点。再进一步，应用的功能又影响了开发团队的结构和规模，我们的任务就是将这些影响因素转化为从架构角度可以调整的参数。



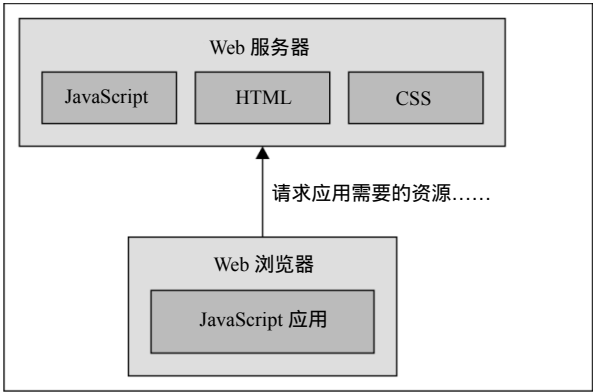
可扩展性影响着我们思考架构的角度，反过来，架构也决定了应用如何响应可扩展性影响因素。这个过程贯彻软件生命周期始终，不断重复、永不终结。

浏览器是一个独特环境

传统意义上的扩展策略在前端环境下无法工作，当后端遇到请求激增的时候，通常会通过直接增加更多硬件来解决。当然说比做简单，但是比起 20 年前，扩大数据服务已经容易多了，如今的软件系统在设计之初就已经考虑到了可扩展性。对我们的前端应用来说，后台服务总是可用且有响应是很有帮助的，但这仅仅是我们面临的众多问题中的一小部分。

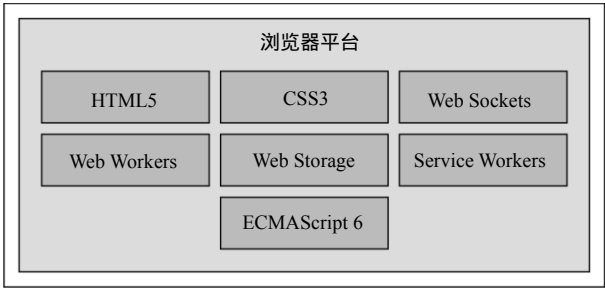
我们不能为浏览器投入更多的硬件来运行代码。正因如此我们算法的时间复杂度和空间复杂度变得更为重要。桌面应用通常对系统有一定的要求，比如操作系统版本、最小内存、最低 CPU 占用等。如果我们的 JavaScript 应用也有类似要求，用户量必然会骤减，并且很有可能会收到投诉邮件。

人们期望基于网页的应用是又轻又快的，也许一部分是我们面临的竞争所导致的。世界上有很多臃肿的应用，不管是基于浏览器的应用还是桌面应用，只要用户感到臃肿，他们就会换用其他竞品。



图注：JavaScript应用需要很多不同类型的资源，这些资源都由浏览器为应用获取。

雪上加霜的是我们使用的浏览器被设计为下载文件，显示超文本链接的一种手段。我们现在做的也是这些事情，只不过是在一个完整的应用中。多页面应用逐步被单页面应用所取代，虽然如此，应用还是被视为一个页面。尽管如此我们仍然处于巨变之中：浏览器仍然是一个充满活力的 Web 平台、JavaScript 语言正在逐步完善、大量 W3C 标准正在编写之中，这些都使得我们的页面更像一个应用而不是一个文档，如下图所示。



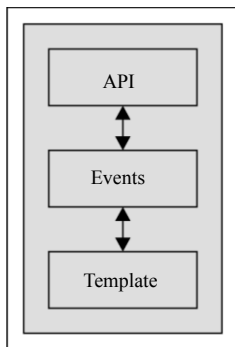
图注：不断发展的Web平台中使用到的一些技术。

我们要以架构的角度审视架构设计。以不同视角来检查设计是一种强大的技术，JavaScript 架构也不例外，尤其是对于可扩展的 JavaScript。JavaScript 的架构和其他架构的不同之处在于，前者有着独一无二的视角。浏览器环境要求我们在应用的设计、构建和部署方面有不同的思考。本质上浏览器中运行的任何东西都是稍纵即逝的，这一点要求我们改变多年以来软件的设计方法。另外，我们花在编写架构上的时间要多于绘制架构图，因为当画完架构时我们的设计可能已经被另一个标准或工具所取代。

组件设计

在架构级别，我们主要打交道的建构基础就是组件。这些组件可能是高阶的经过多层抽象的，或者是使用的框架暴露出来的，这些框架或工具都提供了自己定义的组件。本书中提到的“组件”是介于抽象和具体之间的。对应用程序的组成我们必须要考虑周全，而不必过多关注细节。

当我们第一次考虑可扩展性构建应用时，组件之间的组合关系就开始逐步成型。组件之间的组合方式很大程度上限制了可扩展的执行，因为组合方式建立了标准，组件为了保持一致，会去实现各种模式，而确保模式的正确非常重要。



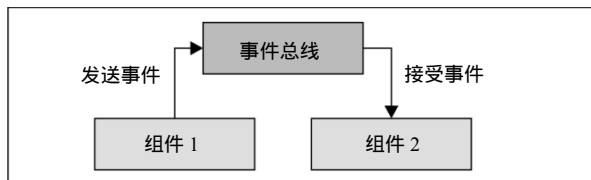
图注：组件的内部结构，内部结构的复杂度取决于组件的类型。

各种组件的设计与我们在其他视角做出的取舍紧密相连。这是件好事，因为这就意味着如果我们一直关注可扩展的软件质量，就可以回头通过调整组件设计来满足质量要求。

组件通信

组件并不是孤立地存在于浏览器中，它们之间在无时无刻都在互相通信。组件间通信

的方法有很多种。比如最简单的方法调用，较复杂的有异步的“发布/订阅”(PubSub)事件系统。使用哪种通信机制，取决于具体需求。其中的挑战在于：只有开始着手实现应用之后，才能逐渐确定需要的最理想的通信机制，同时我们还必须确保后期可以调整通信路径。

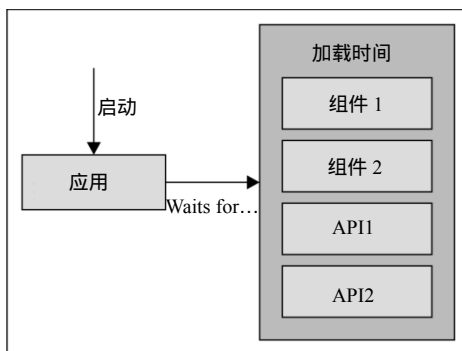


图注：组件间通信机制将组件解耦，使软件结构可扩展

现有工具已经可以帮我们解决一部分问题，在这种情况下不太会再去实现自己的组件通信机制。最有可能的是，最终会在已有工具的基础上，通过调整修改实现自己的通信规范。重要的是组件通信机制有自己独特的设计角度，可以完全独立于组件进行设计。

加载时间

JavaScript 应用总是在加载各种资源。最大的挑战其实就是应用本身。在用户可以与应用进行交互之前，需要加载所有需要的资源，然后是各种按需加载的数据。这些加载产生的延时都会被用户感知。所以加载时间是一个很重要的方面，因为它很大程度上体现了我们的产品质量。



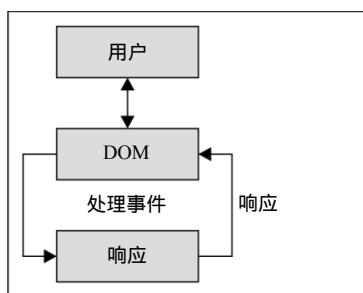
图注：首次加载决定了用户对应用程序的第一印象，而组件初始化正在此时。在其他领域，加快初始化而不牺牲性能是非常困难的。

可以通过很多途径抵消用户由于加载延时产生的负面影响。包括利用那些允许将浏览

器中的应用和服务视为可安装组件的 Web 规范。当然，这都是一些新的想法。但是随着应用和这些规范的逐渐成熟，还是值得考虑的。

响应性

在架构的性能方面，第二部分是与响应性相关的。所谓的响应性就是，在所有资源加载完毕后，应用程序需要多久才能响应用户的输入。虽然这与从后端加载资源是两个独立的问题，但是二者关系密切。通常，用户的行为触发 API 调用，而我们处理这些行为所采用的技术会影响用户感受到的响应性。

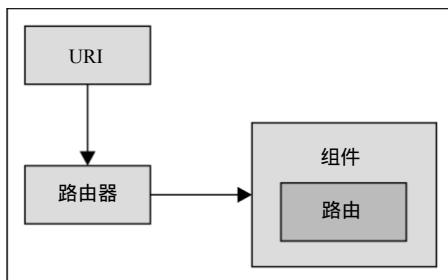


图注：我们的组件对DOM事件的响应速度会影响用户对响应性的感知，在用户行为触发DOM事件和我们更新DOM并提示用户的瞬间可能会发生很多其他事情。

由于这些必要的 API 交互，用户感知到的响应性变得十分重要。虽然我们无法让 API 运行得更快，但可以从其他方面采取措施，确保用户始终可以从界面上得到立即的反馈。比如，通过利用已经加载的缓存数据提升界面切换的响应性。架构的其他方面都与 JavaScript 的性能紧密相关，并最终影响用户对响应性的感知。从这个角度可以巧妙地组件设计和组件通信路径的完整性进行检查。

可寻址性

尽管我们正在建设一个单页面应用，但并不意味着我们不再关心 URI 的可寻址性（Addressability）。作为指向我们想要资源的唯一标识符，URI 也许是 Web 最大的成就了。将它粘贴到浏览器的地址栏，就可以等待奇妙的事情发生。我们的应用中当然也有可寻址的资源，不同的是我们通过不同的方式指向这些资源。URI 是被后端 Web 服务器解析的，后端服务器生成页面并发回给浏览器。与此不同，在单页面应用中是由 JavaScript 代码来解析 URI。



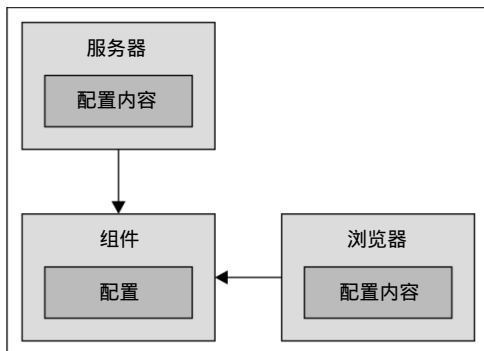
图注：组件监听、响应路由事件，浏览器改变URI就会触发这些事件。

通常情况下，这些 URI 被映射到 API 资源。当用户在应用中触发其中某个 URI 时，会把这个 URI 转换成另一个用来请求后端数据的 URI。我们将管理这些 URI 的组件叫作路由器（Router）。大量的框架和库实现了基本的路由功能，我们可以从中选择一个。

寻址性在架构中扮演了一个主要的角色，因为如果确保应用在各个方面都是可寻址的，会使设计变得十分复杂。但是如果设计合理，也会让很多事情变得更容易。我们可以让组件像用户利用链接那样来利用 URI。

可配置性

软件很少能恰好满足你的需要，做到开箱即用。所以，高度可配置的软件系统备受推崇。但在前端，可配置性是有挑战性的，因为前端的配置可以有很多维度，更不用说把这些配置选项存放在哪儿了。可配置组件的默认值也是个问题：默认值从哪里读取？例如，用户做出更改之前是否有默认的语言设置？多数情况下，不同的前端部署需要不同的默认设置。



图注：组件配置可以来自后端服务器，或者浏览器，默认值必须存放在某个地方。

软件的每个可配置的部分都会使设计变得复杂。更不用提可配置性会给性能带来的额外开销和潜在 bug。可配置性是个大问题，值得我们花时间与利益相关方针对可配置性的价值进行前期讨论。根据部署的性质，用户可能希望他们的配置是可移植的。这就意味着配置需要存储在后端用户账号中，显然这种需求会对后端设计有所影响，但有时还是希望最好可以不修改后端服务。

架构性取舍

如果想要构建可扩展的软件，可以从很多角度来思考软件架构。但是如果每个角度都去考虑，根本不可能做出想要的软件。这就是为什么需要从架构的角度对设计进行取舍：取我们最需要的，舍次要的。

确定不可变内容

在做出取舍之前有一点很重要：列出那些不能舍弃的需求——我们的设计的哪些方面对实现可扩展是至关重要的、不能改变的。比如，被渲染页面中的实体个数或者函数间接调用的最大深度就不能改变。虽然不可变的内容不会太多，但是它们确实存在。最好的办法是缩小这些内容的作用范围，减少它们的数量。如果有太多严格的设计原则不能被打破或改变以迎合需求，就不能更好地适应不断变化的可扩展性影响因素。

考虑到可扩展性影响因素的不可预测性，无法改变的设计原则是否还有意义？答案是肯定的，但只有在它们浮现出来，并且显而易见时才成立。所以这可能不是前期的原则，虽然我们经常要遵守至少一两个前期原则。这些原则可能是从早期代码重构或后期软件的成功中总结出来的。在任何情况下，这些不可改变的内容必须明确下来，并和所有相关人员达成一致。

从开发的便捷性考虑性能

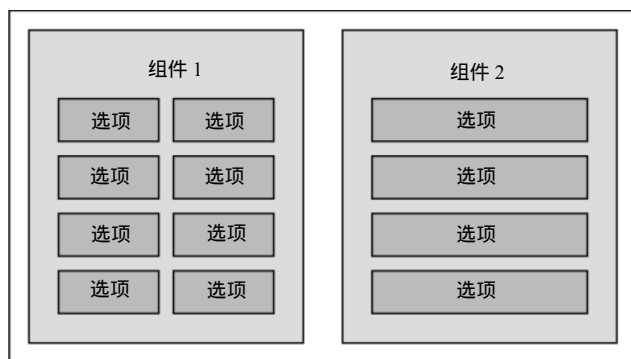
性能上的瓶颈需要在一开始就被修复或避免。一些性能瓶颈是显而易见的，会对用户体验造成明显的影响，这些就需要立即修复。因为这些瓶颈意味着我们的代码由于某些原因无法扩展，并且可能引出设计上更大的问题。

其他性能问题相对较小，通常是开发者为了通过各种手段提高性能，对代码进行基准测试时发现的。这些无法很好地扩展，因为这些小的性能瓶颈无法被用户察觉，但是修复起来非常耗时。即使应用程序大小合理，又有不少开发人员，但是每个开发人员都在修复这类小问题，就无法继续开发新的功能。

一方面，这种细微的优化会引入对特殊情况进行处理的代码，而这类代码对于其他开发者来说就没那么容易理解了。另一方面，如果不进行这种细微的优化，代码就会相对简洁，容易维护。在必要时，需舍弃性能优化来保证更好的代码质量。这样才能增强我们在其他方面提高可扩展性的能力。

性能的可配置性

如果有几乎每个方面都可配置的通用组件自然是极好的。然而，设计通用组件的代价需要牺牲性能。这在一开始只有少量组件时是无法察觉的，但是当软件功能、组件数量、组件配置项开始增加时，问题就显现出来了。随着每个组件尺寸（复杂度、可配置项的数量等）的增长，组件的性能就会呈指数递减，如下图所示。



图注：左侧组件的可配置项是右侧的两倍，使用、维护的难度也是两倍。

只要性能问题没有影响到用户，就可以保留配置选项。不过需要注意的是，可能会在消除某个性能瓶颈时不得不删除某些可配置项，不过可配置项不太可能成为性能瓶颈的主要来源。在不断扩大和增加功能的过程中还容易过于投入，回顾起来你会发现，在设计时创建的自认为有用的可配置项，最终并没有什么用，反而加大了开销。所以，当配置项没有带来确凿的好处时，为了保证性能，应该果断舍弃。

从可替换性考虑性能

一个与可配置性相关的问题是可替换性。现在的用户界面运行良好，但是随着用户数量和功能的增加，我们发现某些组件无法轻易地被另一个组件替换。这可能是一个软件成长问题：想设计一个新的组件用来替换已有组件，或者可能需要在运行时替换某些组件。

替换组件的能力基本上由组件通信模型来决定。如果新的组件可以像已有组件那样发送/接收消息/事件，那么替换起来就相当简单了。但是并不是软件的所有方面都需要可以替换，为了保障性能，可能根本没有可替换的组件。

但当扩展应用时，可能需要考虑将大组件重构为较小的可替换组件。但是这样做会引入新的间接层，从而影响性能。不过牺牲一点点性能换来可替换性，可以帮助我们在其他方面获得架构的可扩展性。

可寻址性的开发便捷性

为应用程序中的资源分配可寻址的 URI 必然会增加功能实现的难度。真的需要为应用暴露的每个资源分配 URI 吗？也许不需要。从保持一致的角度看，为每个资源分配 URI 是合理的。但是如果针对某些资源，没有路由以及统一、易于遵循的 URI 生成方案，那么更倾向于不为这种资源分配 URI。

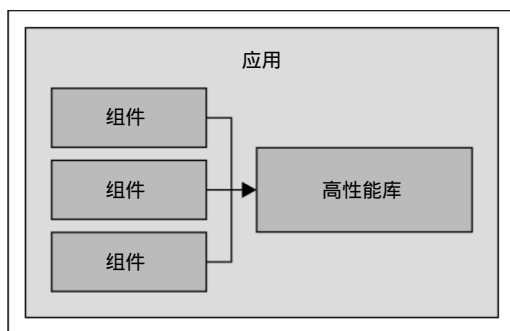
为应用中的每个资源分配 URI 带来的负担是值得的，因为不支持可寻址资源更糟糕。URI 让我们的应用和用户熟悉的其他页面表现一致。也许 URI 的生成和路由在应用中是不可改变的、不可舍弃的，所以几乎总是牺牲开发的便捷性来换取可寻址性。相较于 URI，开发的便捷性可以随着软件的成熟更深入地解决。

性能的可维护性

功能开发的难易程度最终取决于开发团队和可扩展性的影响因素。例如，可能出于预算的压力不得不招聘初级开发人员。这样是否能适应后期要求，则取决于代码。当需要考虑代码性能时，很可能会引入一些对于缺乏经验的开发者难以理解的代码。很显然，这会阻碍新功能的开发。如果新功能开发难度比较大，那就会花费更长的时间，这显然不能适应用户的需求变化。

开发人员并不是总需要费力理解这些用来解决特定领域性能瓶颈的晦涩代码。当然可以通过编写高质量、易于理解的代码，甚至编写文档来缓解这个问题。但这一切都是有代价的，随着团队的壮大，需要短期内完成对开发者的培训、指导，这些都是在生产效率方面需要付出的代价。

在关键代码上经常是优先考虑性能而不是开发的便捷性。不能总是逃避代码性能带来的代码丑陋，但是如果这些丑陋的代码能很好的被隐藏，就可以得到更易理解和自解释的代码。比如，底层 JavaScript 库性能良好，API 紧凑易用，但是如果你看一下底层的源码，就会发现并不是那么优美。这就是我们的收获——让别人维护出于性能原因而看起来丑陋的代码。



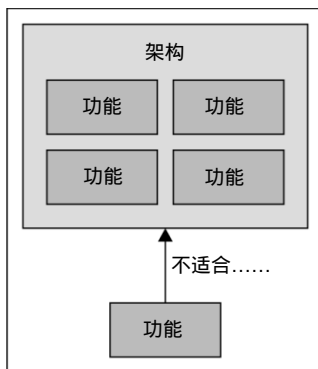
图注：图左的组件代码风格一致且易读，它们都使用了图右的高性能库。这使得应用在性能良好的同时，优化屏蔽了难以阅读、理解的代码。

减少功能以提高可维护性

当所有其他手段都失败时，需要退后一步，全面审视应用中的所有功能。架构是否能够支持所有的这些功能？把投入了大量时间来开发的架构废弃掉是毫无道理的，但也确实会发生。大多数时候，会被要求实现一组颇具挑战的与我们结构相悖的功能。

这种情况发生时，实际上是在打乱现有的稳定功能，或者在应用中加入一些低质量的东西。两种情况皆无益处，但值得投入时间，花费精力，与投资人沟通以决定哪些必须被删除，尽管这个过程并不愉快。

如果已经做出折中的选择并且花时间理清了我们的架构，那么应该有一个合理的理由来解释为什么我们的软件不能支持上百个功能。



图注：当一个架构已经饱和，我们就不能继续扩展。关键要知道临界点在哪里，以便我们能够更好地了解并向利益相关方解释。

利用框架

框架的存在是为了通过采用一套紧密聚合的模式，帮助我们实现架构。框架具有极大的多样性，选择哪个框架是由个人偏好以及与设计的契合度来决定的。例如，一个 JavaScript 应用框架可以实现更多创造性，而另一个框架拥有更多功能，但其中大部分功能是我们不需要的。

JavaScript 应用框架有不同的大小和成熟度。有些内置了很多套件，有些则倾向于机制而非政策。没有一个框架是专为我们的应用设计的，对于每个框架所声称的功能要持有存疑态度。框架标榜的功能只适用于简单的一般情况，而在我们架构中的应用则完全是另外一回事。

尽管如此，当然可以用一个自己喜欢的框架作为设计过程的输入。如果我们真的喜欢这个工具，团队也有使用经验，可以让它来影响决策者。只要我们明白，框架不会自动地响应扩展影响因素，因为这个部分是由我们来负责的。



选择错误框架的代价甚高，因此值得花时间来为我们的项目做调查并甄选出合适的框架。我们通常是在已经实现了很多功能之后，才能获得这样的领悟。结果就是需要进行大量的重新编写、重新计划、重新培训和重新记录，更不用提在第一次实现时花费的时间了。因此，我们需要明智地选择框架，对框架耦合持谨慎态度。

框架与类库

既然有一个整体架构提供了一切所需，为什么还要使用一个混杂的小型类库？类库是我们的工具，如果它们能够在架构中满足一个需求，那么就只管用吧。有些开发人员因为担心接踵而来的依赖关系混乱，而选择避开低阶工具。但实际上，即使使用一个大而全的框架，依赖关系混乱还是会发生的。

归根结底，框架和类库之间的区别对我们来说并不重要。创建一个噩梦般的第三方依赖关系，或者坚持只使用一个工具并自己维护大量代码都不会实现很好的扩展。关键是，我们要在严重依赖其他项目和自己全盘重来之间，找到一个正确的契合点。

一致地实现模式

我们用来实现架构的工具通过暴露贯穿 JavaScript 应用的模式来实现一致性，而且这些模式在哪里都是一致的。当应用随着功能增长而不断扩展时，我们能够反复应用同样的框架组件。框架也可以促进我们的实现模式中的一致性。如果观察任何一个框架的内部，都会看到它有自己的通用组件，他们被扩展开来，为我们提供可用的组件。

内建的性能

开源架构有最多的开发人员来查看代码，在生产中采用这种架构的项目也是最多的。所以他们可以获得大量的用户反馈，其中包括了性能提升。关注性能应该从第三方工具着手，因为它们可能是在一个既定应用中使用最多的代码。让性能依赖浏览器供应商或者 JavaScript 类库是不明智的，利用我们一直使用的组件来调整性能才是正道。

利用社区智慧

成功的 JavaScript 框架拥有强大的社区。它比详实的文档有用得多，因为我们可以碰到问题时随时提问。别人很可能在他们的项目中也使用了和我们一样的框架来实现类似的事情。开源项目就像是一个知识引擎，即使那里没有想要的确切答案，但总能借助社区的智慧获得足够多的灵感，然后自己找到答案。

框架并非天生支持扩展

一个框架并不会比另一个框架的扩展能力更好，写一个 TODO 应用作为标准来衡量框架的扩展性也几乎是无用的。写 TODO 应用是为了大致了解一个框架，知道它和其他框架相比的优劣势。如果不确定哪个框架更适合我们的风格，写一个 TODO 应用是一个不错的开始。

我们的目标是实现能够积极响应影响因素变动的良好扩展性，但这些变动都是独特且无法预知的，我们能做的就是预测未来会遇到哪些影响扩展的因素。基于这些影响因素以及应用的属性，有些框架会比其他框架更为合适。框架有助于我们实现扩展，但是它们不能替代我们自动进行扩展。

小结

扩展一个 JavaScript 应用和扩展其他类型的应用并不一样。尽管能使用 JavaScript 来创建大型后端服务，但我们关注的是随着应用的扩展，用户在浏览器里的交互行为。此外，在设计可扩展的架构时，有大量影响因素可以指导我们做出决策。

我们已经讨论过一些影响因素，了解它们如何以一种自上而下的方式产生影响，并为 JavaScript 前端开发带来独特的挑战。我们考察了更多用户、更多功能和更多开发人员所带来的影响，可以看到，需要考虑的事情有很多。尽管浏览器正在成为一个强大的平台，我们也在浏览器上实现应用，但与其他平台相比，它还是有一些局限。

设计并实现可扩展的 JavaScript 应用要有一个架构。软件最终实现的就是为该设计提供输入，扩展影响因素也是关键。有鉴于此，在本章中从不同角度讨论了如何选择架构。其中组件组合关系以及它们在扩展中的响应性，都是在架构中值得注意的方面。

鉴于这些扩展影响因素会随着时间的推移而改变，我们以架构的角度来调整设计或者修改产品，以应对扩展所面临的挑战。下一章将进一步讨论这些影响扩展的各项因素，深入了解它们并准备一个核对清单，帮助我们实现可扩展的 JavaScript 应用来响应这些事件。

2

可扩展性的影响因素

提到可扩展性的影响因素，首先要提软件的使用者。他们是首要的影响因素，因为他们是我们开发软件的终极原因。在第 1 章中提到，用户影响软件的功能，最终影响我们编写的代码以及开发者。当我们停下脚步，开始思考这些可扩展性影响因素时，我们认识到一个健壮的 JavaScript 架构可以处理这些影响因素。带着思考后的结果，可以从不同架构的角度来审视我们的代码。从下一章开始，我们将会深入研究每一个角度。

但在深入研究之前，让我们先更仔细地看一看这些可扩展性的影响因素。我们希望认真了解一下这些影响因素。因为设计上的每个决策很大程度都取决于这些影响因素。也许更重要的是，我们需要设计一个可以适应未预料到的场景的架构。

先仔细看一下使用我们软件的用户：他们为什么使用我们的软件？我们的软件如何满足他们的需要？不管你信不信，这些问题与如何编写 JavaScript 代码是有关系的。接着，将焦点从用户转到软件功能，看一下应用的外向人格（outward-facing personality）。或许有些功能与我们的应用并不契合，但是有时候这并不重要，因为我们没有发言权。如果打算扩大规模、取悦用户，就必须将这些功能做到最好。

最终负责开发这些功能的人力资源也是一个影响可扩展性的因素。它可以成就一个产品，也可以毁掉一个产品。我们将了解开发团队所面临的挑战，以及他们如何被功能的影响因素所限制。在本章末尾，将给出一个影响因素的清单，来确保影响扩展能力最紧迫的问题都已经被考虑到。

扩展用户

最重要的用户就是我们自己——软件开发者。虽然我们的使命是通过交付可扩展的软件让用户满意，但同时还需要让自己满意，这就需要一个可行的商业模式。我们之所以关心这点，因为不同的商业模式意味着需要使用不同方法来获取新用户、管理已有用户。从这一点开始，扩大用户基数的复杂度开始加大。我们需要考虑如何组织用户，用户是如何使用我们的软件进行彼此交流的，如何为用户提供帮助、收集反馈、收集用户数据等。

对 JavaScript 应用来说，可行的商业模式可以是部署广告支撑的免费服务，也可以是为私人现场部署我们的软件，从中收取许可费。决定究竟哪种方法最合适很可能会超出我们的权利范围。然而，我们的责任是，了解所选择的商业模式，将之与软件当前及未来的用户相关联。

商业模式可能会非常复杂。比如，企业往往会从一个明确的、可以满足用户的，同时满足企业需求的方式开始。然而，随着企业的发展和成熟，曾经清晰良好的业务模式变得模糊、不太平易近人，对我们的架构会产生不可预知的影响。让我们来看看其中的一些商业模式会如何影响我们用户群的可扩展性。

许可证费用

软件授权是一个复杂的话题，我们不打算在这里深入探讨。最重要的问题是，我们的商业模式是否依赖出售软件许可。如果是，很可能会请其他组织到现场部署我们的 JavaScript 应用程序，而不太可能让个人购买软件许可证。虽然不是不可能——但要取决于软件的具体性质。销售许可证的一种可能的情况是，我们的软件将被多个组织私有地部署。

对于这种商业模式，有两个有趣的扩展性属性需要考虑。首先，特定组织中的用户数量是有限的。虽然组织可以很大，可以卖给多个大型企业，但通常的情况是，使用许可证模式只有较少的用户。其次，每个组织都有定制的需求。这涉及到可配置性、用户的组织，等等。在许可证模式下，我们更容易在这些方面收到用户对软件进行改进的要求。

因此，尽管需要支持的用户并不多。由于使用软件的组织结构不同，所以技术支持性质会变得更加复杂，从而更难扩展。这些环境下的依赖管理也变得更具有挑战性，因为约束条件决定了我们的软件如何扩展。而在其他环境中，这些限制是相对比较宽松的。

订阅费用

订阅费用是用户使用我们的软件时定期收取的费用。大部分情况下，订阅可以使用户花费得更少。同时这也是一种更灵活的商业模式，因为它可以很容易地应用到现场部署或公开部署的软件上。

相较于许可证模式，订阅模式对组织来说更便宜，所以我们的软件就更有可能被更多组织使用。你要知道，一个组织可能分为多个部门，每个部门都有自己的预算。

然而，在规模上，订阅模式面临着和许可证模式类似的挑战，即复杂的定制需求。如果订阅模式能带给我们更多现场部署的机会，那么也会带来更多功能需求。订阅模式的另一个可扩展性问题就是如何留住客户。如果软件不能为用户提供持续价值，用户将不会继续支付订阅费用。

因此，如果打算使用订阅模式，就需要努力提供新功能，以证明我们的软件物有所值。

消耗费用

另一种模式是按需购买。由于用户不需要购买用不到的资源，这种模式对用户是有吸引力的。当然这种模式并不适合每个应用，比如，没有用户需要的资源或应用消耗的资源对我们来说无关紧要。

在其他情况下，资源的使用是明显的。比如用户执行了花费较大的计算任务，或在一段时间内存储了大量的数据。这种情况下，按需购买的模式对我们和用户都是合理的。用户消耗的资源越少，付出的金钱就越少。用户的行为可能是飘忽不定的，消耗也会有峰值。尽管如此，相对于其他使用软件的事件来说，这些事件都是短暂的。

这种商业模式对可扩展性的挑战是，需要额外的工具来衡量应用的核心数据。首先，需要一个工具来测量、记录消费。其次，需要一个工具来准确、可视化地描述这些消费指标。根据用户消费的内容，以及我们期望达到的集成度级别，会考虑一个第三方组件来满足这些需求。

广告支持

另一个选择是将应用程序部署到互联网上，通过显示广告获取收入。因为是免费的应

用程序，所以更有可能被用户使用。但另一方面，很多人会抵制广告，这一点也会抵消免费带来的吸引力。

也许，使用这种方法的目标并不是广告收入，而是大规模的使用量。事实上这二者是并存的，因为更多的用户意味着更多的广告收入。同时，流量巨大的 JavaScript 应用程序可以吸引投资者的关注。因此，大量用户本身就是很有价值的。

这些类型的应用与其他商业模式的应用在扩展性方面不大相同。互联网上吸引大量用户的应用为不同的人解决不同的问题。采用这一模式意味着我们需要触及大量用户，这也意味着我们需要降低进入门槛。采用这种商业模式时关注的重点是易用性和社会意义。

开源

我们要考虑的最后一个商业模式是开源模式。不要笑，开源软件对 Web 的运作非常重要。JavaScript 应用程序不大可能不使用任何开源组件，更有可能我们使用的组件都是开源的。但为什么人们会花费宝贵的时间来开发每一个人甚至包括竞争对手都可以使用的工具？

其中第一个误解是：为我们开发开源软件的人是一群坐在家里、失业的人。而事实是，我们使用的大多数工具的开发人员大部分就职于做同样技术并处于强势地位的公司。或许他们还启动了开源项目，以解决公司的问题，提供其发展过程中缺失的工具。

第二个误解是，启动或促成开源项目是在帮助竞争对手。其实通过开源软件，仅凭一己之力，是不太可能把自己置于比竞争对手更差的境地的。当然，通过其他方式，是绝对可以通过伤害自己来帮助竞争对手的。

另外，开源项目对组织是有益的。开源项目必须是有效的项目，是有用的、通用的。如果项目持续足够长的时间，就会有更多的使用相关技术的参与者。而这是一件好事，围绕开源项目的社区是非常宝贵的。虽然开源项目本身不足以支撑一个组织，但不可否认的是，它是任何 JavaScript 应用商业模式的一个基本组成部分。

可以利用组和角色组将用户分类。将角色作为用户类型是一个强大的抽象概念，因为它允许我们通过角色类型来概括各种功能。例如，通过角色的属性而不是单个用户的属性来进行检查，这样比修改逻辑更容易将用户从一个角色迁移到另一个角色。

理清用户角色以及如何将他们转化为用户组是一个棘手的问题，唯一可以指望的是调整用户的组织结构。因此，首要目标是使分组机制尽可能地通用。不过这也需要做出取舍，毕竟任何完全通用的组件都会对性能带来负面的影响。

有些分组决策在一开始是显而易见的，比如用户是否能意识到系统中其他的用户。如果是这样，就可以开始研究用户如何使用我们的应用程序进行沟通的具体问题了。同样的，这可能很明显是依赖于应用所拥有的不同特性。我们遵循的商业模式也会影响用户管理方面的设计：如果是基于许可证模式，很可能会现场部署，并且会有很多用户角色方面的需要，以及随之而来的用户分组；如果在互联网上公开部署，用户分组就没有那么重要了一—那么，或许可以选择有利于软件性能的更简单的做法。

随着我们的软件变得越来越复杂，以及更多功能的添加带来了更多的客户，我们开始需要隔离应用程序的某些部分。也就是说，基于访问控制权限需要限制某些功能。相对而言，使用一套用户、分组和访问控制系统更简单些，而不是让不同的用户角色安装独立的软件系统。

作为 JavaScript 架构师，这些对我们有更深一层的含义。因为一旦决定引入访问控制，就没有回头路了。从此以后，我们必须保持一致——每个功能都需要检查相应的权限。更复杂的问题是，如果这样对用户进行分组，那么很可能也需要对系统的其他实体进行类似的分组。只有这样才有意义，尤其是对最终用户来说，因为该组相关的功能、资源可以由该组用户访问并使用。

与用户沟通

另一个需要考虑的关于用户以及用户之间的关系的，是提供给用户用来沟通的渠道。用户是明确选择某个用户进行沟通还是更隐晦地进行沟通？比如，一个与我们同组的用户正在看一张图表，生成图表的数据由组内其他成员放入系统。除去那些明显的沟通渠道，思考这些隐晦的沟通渠道是否值得？

应用程序的性质决定了开放哪些沟通渠道给用户，这可能还取决于用户自身。有些应用程序的用户可能需要自己熟练地执行任务，而不需要与其他用户通信。另外，我们可能会发现自己开发的東西有一些社会意识。事实上，我们甚至可以依赖于外部的社交网络服务。

如果要依靠第三方用户管理、社交网络或其他服务，必须小心与这些服务耦合的紧密程度。从可扩展性的角度讲，采用第三方认证机制可能会给我们带来想要的额外好处，特别是考虑到大多数用户都希望不需要另外注册账户就可以使用我们的应用程序。从其他角度来讲，一旦开始实现新的功能，使第三方集成变得非常复杂，这种方法对用户管理将是一个问题。例如，一个照片编辑应用程序如果使用 Facebook 登录可能会有更好的扩展性，因为 Facebook 是大多数用户的照片源。

如果我们的应用程序非常实用或有趣，用户会找到相互进行交流的方式。我们可以抵制它，也可以利用用户通信来帮助我们实现扩展。也就是说，扩大信息的透明度，可以使我们的用户与同伴分享有用的东西，而不必四处搜罗。

支持机制

如果 JavaScript 应用程序可以正常工作那再好不过了。但是即使一切都按计划进行，已经部署完成并且没有 Bug，仍然要在用户需要、误操作或与可用性相关时，提供迅速的技术支持。

如果技术支持机制不能很好地扩展就可能阻碍业务的发展甚至导致业务停顿。所以，除了软件需要可扩展性，用户支持系统同样需要。支持系统可以集成在我们的软件里，也可以外包给第三方软件或人员。

如果用户不需要技术支持更好，这就是我们设计时考虑易用性的原因。通过专家或实际用户了解各种用户体验，并将他们的设计集成到我们的软件中，这是在用户支持方面可以做到的最显而易见的事情。因为如果通过易用性设计能做到这一点，那么就可以在扩展的同时很大程度上消除潜在的用户支持方面的问题。

不管怎样，还是要假设，在部署完之后会有意想不到的支持方面的案例出现。因为用户是有好奇心的，就算一切正常，他们仍然会提出问题。所以，我们不能对用户说：“我们的用户体验是一流的，一切运转正常，走开。”对于用户的问题和疑虑都应该回应。因为如果对用户的问题不屑一顾，就无法扩展我们的应用程序了。

JavaScript 组件能否用来支持我们的用户？如果这就是我们想要的，绝对可以！事实上，根据用户所处环境提供帮助是最有效的方法。如果用户对特定组件有疑问，而他们在这个组件上看到一个帮助按钮，可以通过单击这个按钮提交问题，那么在我们在接收到问题后，

就会对用户所处环境更加明确。我们确切地知道用户想做什么，而不必再花时间来模拟用户所处的环境。

说起来容易做起来难，而且这对可扩展性也会产生其他影响，因为上下文帮助系统是需要工作量的。而如果决定采用它，就必须考虑为所有功能增加上下文帮助。这样做是否可扩展呢？

我们可以考虑的另一种方法是，建立一个拥有来自软件制造者、使用者的信息知识库。处于某种目的而使用软件的使用者比我们更容易给出更好的答案。而这些答案对用户、对我们都是非常有价值的。

反馈机制

是否有必要将反馈与支持相区分？支持本身就是一种反馈。如果你留意在过去一段时间里遇的各种支持问题，就会发现我们可以将其转化为反馈，并利用这些信息作为反馈。但是，反馈与支持两种形式仍然值得区分，因为在两种形式中用户的心态是不同的。虽然遇到需要支持的问题时，用户可能会感到轻微受挫，也可能是严重受挫。此时用户根本不关心如何帮你提高产品质量，他们只关心怎么才能解决问题，完成他们的工作。

另一方面，用户在使用一段时间我们的软件后，会敏锐地觉察到他们在某个工作流程中效率低下。收集这类反馈是至关重要的。如何获得这类反馈呢？一种选择是在应用程序中提供反馈按钮，就像上下文相关的支持按钮一样。另一种选择是让第三方来处理收集的反馈意见。就像技术支持一样，将用户所在的环境自动地添加到反馈信息中，这对我们理解用户的反馈是有好处的，而且不会花费很多时间。

反馈的关键是让客户积极参与。并不是每个使用我们软件的用户都愿意与我们分享他们的想法。但总会有人是愿意的，就算他们只是在发泄挫败感。我们必须对他们的抱怨做出响应来建立起对话，同时提供这类反馈的用户也希望我们做出回应。正是在与这些用户的不断交谈中我们的产品才能不断完善，而不能指望用户一开始就提出绝妙的主意。

随着用户数量的增长，我们是否能跟上并保持对用户反馈的响应？显然，以我们目前的状况和应用成长的速度来看，这将是一个挑战。围绕一段给定的用户数据创建对话框是一件事，对这些反馈做出响应是另一件事。假设我们启用了很棒的反馈机制，并植入了软件中，我们还需要让这个机制运行起来。因此，我们需要考虑如何将根据用户反馈产

生需求的过程变得可扩展。如果，用户反馈从来没有转化为行动，就无法在这方面做到可扩展。

提示用户

JavaScript 应用程序需要为用户显示通知提示。特别是当我们关心的主要是响应用户的操作时，可以简单直接地实现通知提示。例如，当用户做了某些操作，导致向后端发起 API 请求时，需要向用户显示一个通知，提示用户该动作是成功还是失败。这样的提示可以在整个应用程序中使用，即可以用同一个工具来显示大多数通知。

在设计一个可扩展的 JavaScript 架构时，很容易忘记通知。通知是一个很大的话题，有上下文敏感的通知、普通通知，还有发生在用户离线时的通知。离线通常是指发送电子邮件给用户，提示他们登录，或在需要时采取行动。

上下文敏感的通知可能是这几种通知中最重要的，因为它们提供的反馈与用户正在做的事情相关。这在可扩展性方面是具有挑战性的，我们必须确保这些类型的通知在整个应用中的用户界面对所有类型的实体都保持一致。普通的通知一般用来提示后台操作的结果。

用户的某些资源会在意料之中或是意料之外改变状态。不管怎样，用户可能想了解这些事件。理想情况下，如果用户登录使用系统，那么通用通知会自动发出。但是，我们可能还希望这些类型的通知以邮件形式发送给用户。

通知系统面临的挑战是容量。如果用户量很大且很活跃，就会产生大量通知需要发送。这无疑会影响其他组件的性能，同时还需要面对通知系统的可配置性。我们的通知系统不可能迎合所有的用户，所以需要在某种程度上调整通知系统。具体如何调整才能使应用可扩展，取决于 JavaScript 的架构师和开发者。

用户维度

数据是了解用户如何与软件进行交互的最佳途径。某些数据是不能靠猜测或手工收集的，而需要借助工具，自动化地收集用户与软件交互的相关数据。有了原始数据，就可以进行分析并做出决定。

虽然将收集任务自动化是合乎情理的，但是一开始可能并不需要收集。只有对某个功能的未来方向不确定，或者想要进一步了解下一步工作的优先级时，才值得去收集用户数据。大多数情况下，我们可以毫不费力地得到答案，也就不需要分析工具了。有些情况，比如现场部署情况下，我们可能不允许收集这些数据。

市面上有很多第三方的数据采集工具。这些工具包含了很多我们需要的很有用的报表，不过也包含了很多我们用不上的报表。同时还有一个第三方组件集成度的问题，因为总有些时候会有需要关闭某些功能，或者至少需要允许我们设置数据存储位置。

除了决定产品方向，这些数据还有很多其他用途。我们可以通过这些数据提高用户体验。提高用户体验可以通过根据以往经验简单地提出建议，还可以基于这些数据来进行有效的优化，这都取决于我们的用户想要什么。弄清楚用户想要什么也是一个可扩展性问题，因为随着软件的壮大，会获得更多的用户，而这些用户想要不同的东西。此时，收集到的用户数据就会成为应对可扩展性问题的有力武器。

扩展用户示例

假设我们的公司在开发在线贷款应用，前端简单明了没有太复杂的东西，申请者创建账号，就可以申请新的贷款或者管理已有贷款。这个应用的商业模型是基于消费的，我们通过贷款的利息盈利。所以，贷款越多挣得越多。

很明显，影响可扩展性的因素是用户量和易用性。低息小额贷款是我们服务的部分价值所在。在申请新的贷款时用户希望付出尽量小的成本、最少的输入、最短的等待时间。这些是应该高度关注的、要提供给用户的价值，以及要面临的可扩展性影响因素。

再来考虑一下我们应用的可扩展性。在已定的应用类型下，我们不太可能预见到社交功能方面的需求。在大多数情况下，用户可以被视作黑盒，他们在使用应用时只存在于自己的小宇宙中。由于应用的易用性对我们来说非常重要，所以应用只有很少的部分会经常变化。因此，对于可扩展性来说，支持和反馈不是问题。我们无法彻底取消支持和反馈，但是花费在它们上面的精力可以尽量减小。

另一方面，需要向市场推出我们的服务，但我们不清楚客户贷款做什么，最流行的还款计划是什么样的，等等。对于这些，我们或许可以提供更有效的市场信息，并提高整体的用户体验。这就是说，收集有关应用程序的元数据非常重要。由于我们的用户量庞大，

所以需要存储的元数据量也很大。在设计功能时还必须考虑到如何收集元数据并进行存储以供日后使用，这也使设计变得复杂。

扩展功能

现在把注意力转向扩展软件的功能。用户是最终的影响因素，我们现在对扩展用户有了大概的了解，就可以把这些知识运用到功能开发上。当我们考虑扩展用户时，就需要考虑为什么这么做。为什么选择了这种商业模式而不是那种商业模式？为什么需要为某个用户角色开启一个功能，而对其他用户角色禁用这个功能？一旦开始实际设计和实现时，就要开始思考如何去做。我们不仅要关注正确性，而且还要关注可扩展性。对于用户来说，可扩展功能的影响是决定性的。

应用价值

我们希望很好地实现应用的各个功能，并且引入新的功能为用户提供价值。思考这一点是有价值的。因为从本质上讲，增加我们软件的价值，让更多的用户受益，正是我们为之努力的目标。于此相反的例子是，由于过于关注扩展新的领域，导致用户依赖的软件已有功能被忽视，导致用户感到失望。

当忘记开发应用的初心时就容易出现这种问题。听起来有些荒谬，但是在各种因素影响下，很容易走向一个完全不同的方向。通常情况下都会导致软件的失败，只有在极少数情况下，才会产生极为成功的软件。而这确实是一个可扩展性问题。我们的软件应该有一组始终可以稳定交付的核心价值，这是我们软件的本质所在，不应动摇。我们常常会面临其他扩展性影响因素，比如新用户希望从软件得到更多核心价值以外的价值。如果我们无法处理，就意味着我们无法扩展软件的主要价值。

当我们对软件的当前价值和理想价值发生混淆时就暗示着我们走向了歧途。换句话解释这两种价值就是，软件当前能做的事情与希望它将来能做的事情。毫无疑问，我们必须前瞻性地思考这个问题。但是，需要对未来计划的各种可能性进行持续的完整性检查，而这往往意味着回溯到我们为什么创建软件。

如果应用程序真如我们所愿非常引人注目。那么就必须与其他扩展性影响因素做斗争，以使我们的软件保持当前的势头。也许这就意味着评估新功能的流程中需要增加一个环节，

用来保证新功能在某种程度上能够有助于体现我们软件的核心价值。并不是所有的功能都会有帮助，所以更值得仔细推敲，是否真的值得牺牲扩展能力来改变方向？

“杀手级”功能与“杀死”应用的功能

我们希望我们的软件可以从众多竞争对手中脱颖而出。最好的情况是，我们所在的市场足够细分以至于没有竞争对手，这样就可以轻易实现一个稳定的、可工作的、没有任何花哨功能的软件。但事实并非如此，所以我们的软件必须与其他软件区分开来——一种方法就是实现杀手级的功能。所谓杀手级功能就是只有我们的软件才有，别家没有，而且用户深度关注的功能。

我们面临的挑战是，杀手级功能很少是计划出来的，它们一般是应用程序顺利交付的副产品。随着应用的不断成熟、细化和调整，会突然发现一些功能演变成了一个杀手级功能。杀手级功能的这种产生方式并不奇怪。倾听用户的声音，满足扩展的需求，就可以演化出特有的功能。如果在足够久的时间内不断地增加一些新功能，去掉一些旧功能，修改已有的功能，那么杀手级功能就会自动浮现出来。

有时我们会提出明确的计划，某个功能将成为杀手级功能。这样做不是最优的，对用户也没有价值。他们选择我们的软件，不是因为我们的产品路线图中有很多杀手级功能，而是因为我们做了他们需要完成的事情，很可能因为我们做的比竞争对手更高效。如果我们开始时就从杀手级功能本身来思考这个问题，就会远离应用的核心价值。

解决这个问题的最佳方案就是创建一个开放的环境，在软件功能初期阶段欢迎来自团队所有成员的意见。尽早扼杀一个坏点子，就能避免在上面浪费时间。不幸的是，现实中好点子和坏点子并不好区分，只有对某个功能开发一段时间后才会发现它在某些方面无法扩展。很多时候会是这样，但不至于损失惨重。即使在某个功能开发之后希望停止，还是可以从中学到一些宝贵经验的。

当不能扩展时，会决定终止该功能。我们不会在架构上强加无法工作的功能。在开发每个功能的过程中都会到达某个点，在这个点需要问自己，看中这个功能是否超过现有的架构。如果是，是否愿意改变体系结构以适应这个功能。一般情况下，体系结构比功能更有价值。因此，停止无法适应框架的正在开发的功能，可以作为一个宝贵的经验。基于被取消的功能，今后可以更好地理解哪些功能可以扩展，哪些不行。

数据驱动的功能

一个应用拥有大量且多样化的用户群是一回事，能否通过收集数据了解用户与软件之间的互动行为、并有效利用这一信息又是另一回事。用户行为指标（user metrics）是一个强大的工具，能通过收集相关信息，为软件的未来发展方向提供决策支持。我们称之为数据驱动的功能。

起初，当没有用户或者只有少量用户时，当然无法收集用户行为指标，所以只能依赖团队的集体智慧或其他信息。我们都曾做过 JavaScript 项目，所以大概知道如何让产品起步。一旦项目启动，需要让工具就位，来更好地支持功能方面的决策，特别是用来决定需要加入哪些功能、不需要哪些功能。随着软件日益成熟，可以收集更多的用户行为指标，从而进一步对功能进行修正，满足用户的实际需要。

拥有足够的数据来实现一项由数据驱动的功能对于扩展来说颇具挑战，因为在一开始就需要一套机制来收集并提炼数据。这要求一些我们可能并不具备的开发投入。此外，必须基于这些数据来决定功能，但数据不会自己变成明确的指示和要求，这要求我们对数据进行分析和解读。

对于那些一直要求要实现的功能，我们也想预测它们的生命力。如果没有数据来支持我们的假设，这项任务就比较困难。比如，我们是否有该应用未来运行环境的数据？简单的数据点足以决定一项功能是否值得去实现。

以数据驱动功能从两个角度来说都是切实可行的——我们自动采集的数据和我们提供的数据。尽管两者都难以扩展，但两者都有扩展的必要性。真正的解决方案只有确保功能的数量足够精简，这样我们才能够应对每个功能所产生的数据量。

与竞品比较

除非在一个完全细分的市场，否则很可能会遇到竞争对手。即使是在一个小众的市场，仍然可能与其他应用程序在一些功能上发生重叠。由于软件开发公司越来越多，很可能会有直接的竞争。通过创造卓越的功能与同类产品竞争意味着，不仅要继续提供一流的软件，而且需要了解竞争对手以及对方用户的想法。这是扩展能力的限制因素之一，因为必须花时间去了解竞争对手使用的技术是如何工作的。

如果我们有一个产品销售队伍，他们往往可以知道其他人在做什么。因为潜在客户经常会问他们：你们的软件有没有这个或那个其他某个软件已有的功能。也许最引人注目的卖点就是我们不但可以实现该功能，并且可以做得更好。

这是我们必须要小心的地方。因为这个因素也会限制我们赢得客户的能力。我们对现有和潜在客户的承诺必须做到可扩展。承诺太多，将无法实现功能，使用户失望。承诺过少，或干脆没有承诺，就无法赢得客户。最好的应对办法是，确保销售人员能够一直接触软件最真实的一面。包括软件能做什么、不能做什么、未来有哪些可能性、有哪些不切实际的选择，等等。

销售人员在推销产品时对如何兑现承诺及其影响无法全部了解，所以必须有一定的回旋余地。否则，将无法获得客户，因为没有为产品制造出兴奋点。如果要扩展这种方法来赢得新客户，就需要一个行之有效的方法将承诺转化为现实。一方面，不能在架构上做妥协。另一方面，必须找到某种方案以满足用户的需求。

修改已有的功能

在成功部署 JavaScript 应用程序之后的很长一段时间内，我们还要不断地改进代码和总体架构。所以说，唯一不变的就是变化。为了改善用户体验，返回去修改软件的现有功能需要相当的纪律性才能做到。因为我们会受到来自股东增加新功能的压力。这是一个长期的问题，如果无法提升已有的功能，就永远无法添加新功能。

最不可能的情况是不需要修改任何东西，而且现有的所有用户都很满意，他们不希望有任何改变。有些用户害怕改变，这意味着他们喜欢这个软件的各个方面，我们做得很好。很明显，我们希望实现更多像这样，让用户普遍感到满意，不需要改进的功能。

如何做到这一点呢？必须倾听用户的反馈，以路线图为基础，根据反馈来修改功能。要保持对用户和用户要求的可扩展性，达到开发新功能和修改现有功能之间的平衡。如果要检查我们是否正朝着正确的增强功能的方向前进，一种方法是，将变化的内容发送给用户，然后统计得到的反馈。事实上，这可能会吸引一些相对安静的用户给我们提一些具体的建议。这是一种将决定权抛给用户的方法——这是我们的想法，你是怎么想的？

除了搞清楚需要改进哪些功能，以及相对于新功能何时改进已有功能之外，还有架构方面的风险。代码是如何紧密耦合的？是否可以在不破坏其他功能的前提下隔离某个功能？

这种风险永远无法完全消除——只能减少耦合。这里的可扩展性问题是，为了重构、回归，我们花在修改给定功能上的时间有多少？组件耦合越松，我们花在这些活动上的时间就越少，就有更多的时间来增强已有功能。从管理的角度来看，通过改变带来的冲突总是会给组织的其他人带来风险。

支持用户分组和角色

随着不同类型的商业模型的出现以及用户基数的增加，用户管理成为一个可扩展性问题。因为它触及我们实现的每个功能，而且用户管理很可能会随着功能需求的变化而变得更加复杂。随着应用的成长，我们需要处理角色、分组和权限控制。

复杂的用户管理会带来很多副作用。刚开发的功能一开始也许可以很好地工作，但是在用户可能面临的其他场景中会失败。这就意味着需要更多的事件来测试这些功能，QA 团队已经不堪重负了，更不用提复杂用户管理在安全与隐私方面给每个功能带来的额外工作量。

我们对复杂的用户管理方案真的是无能为力，因为这经常与使用软件的组织结构紧密相关。我们更可能以现场部署的方式来面对这种复杂性。

增加新服务

到了某一个阶段，现有的后端服务会不足以支持新功能的开发。此时对后端的依赖非常小，可以更好地扩展前端的开发投入。如果听起来有悖直觉，不要担心，因为我们确实需要后端服务来实现客户需求。所以，对后端的需要一直都在。需要避免的是对 API 进行不必要的更改。

如果可以利用现有的 API 就可以实现功能，那就可以这样做。这样后端团队能够专注于修改 bug，以确保稳定性和性能表现。如果为了支持功能而不断改变 API，后端就无法专注。

有时，无法避免地需要新增后端服务。为了扩展开发进程，需要知道新服务在何时是必要的，并了解如何实现它们。

第一个问题是新服务的必要性。当所需 API 不可能实现时，这就变得十分简单，因为

我们不得不利用现有的资源。第二个问题是新服务的可行性。由于需要一个新的 API，所以很可能是由我们来初步构建一个 API，然后必须要听一下后端团队的反馈。如果我们的团队拥有全栈工程师，那么经费开支会更少，因为我们都在同一个团队，彼此之间沟通紧密。

现在已经决定构建一个新的 API，就必须相应地在前端和后端同步实现功能。没有一劳永逸的方案可采用，因为实现服务的难易程度有别。我们的功能可能要求若干个新的服务。关键是在 API 上达成共识，并拥有一套已经就位的模拟机制。一旦实际服务上线，取消模拟机制只是时间的问题。

然而，从整体应用整体扩展的角度来讲，这仅仅是前端功能和后端服务集成点中的一个。为系统引入新功能带来的影响是未知的，只能通过测试和先验经验知识进行猜测。只有到在最终的生产环境中，才能看到新功能的可扩展性。不同的功能虽然使用相同的服务，但也会对请求负载、错误率等各方面产生不同的影响。

在 JavaScript 应用中为了保持用户会话与实际同步，经常使用 Socket 连接后端数据。这样做虽然可以简化一部分代码，但同时也使其他代码变得复杂，由此带来的影响是巨大的。通过 Web Socket 发送的实时数据称为“推送数据”。在 Web Socket 出现以前，流行使用长轮询 HTTP 请求。这意味着客户端需要负责检查数据是否发生了变化，而不是由后端负责检查并在发生变化时将数据发送给客户端。

至今，同样关于实时数据的可扩展问题仍然存在。有了 Web Socket，部分工作从前端转移到了后端：当相关数据发生变化时，后端服务负责推送 Web Socket 消息。尽管如此，还是有一些方面需要我们注意。比如，从整体来看，我们的架构是否依赖实时数据的分发？或者我们是否仅考虑了某个功能的实时数据？如果为了更好地支持新功能而首次引入 Web Socket，就必须问自己：在架构演变过程中，这是我们想要的吗？如果实时数据只是影响到一两个功能，是远远不够的。对于开发人员来说，如果一个功能有实时数据，而另一个没有，在开发软件过程中会很难解决一致性的问题。

将实时数据恰当地集成到前端架构中，会使应用更加合理并且在多个方面更好扩展。这基本上就意味着任何一个组件都应该使用与其他组件相同的方式来访问实时数据。同以往一样，数据从用户和其组织向下流动的过程中所面临的可扩展性问题，最终决定了实现功能的类型，而这又会影响实时数据发布的速率。实时数据发送到浏览器的频率会根据应

用的结构和用户数据连接方式的不同而产生很大的变化。我们在实现每个功能时都应该充分考虑到这些方面。

扩展功能示例

我们的视频会议软件在大型组织中十分流行。主要原因在于它具有稳定、高性能和基于浏览器无须插件的特点。有一个客户希望实现聊天功能，因为他们非常喜欢我们的软件以至于希望用我们的软件来完成所有的实时交流，而不仅仅是视频会议。

在 JavaScript 层面上实现聊天功能并不很困难。我们最终会复用 Web 视频会议功能中的若干组件，通过简单的重构就可以开发出所需组件。但在可扩展性方面，文字聊天和视频聊天之间还是有细微差别的。

主要差别是文字聊天与视频聊天持续的时长不同，后者通常比较短暂。这就意味着需要为长时间的聊天制定合理的策略。我们的视频聊天不需要用户账号就可以参加，因为有时可能会邀请组织外的人来参加视频聊天。这与文字聊天不同，因为我们不能准确地邀请某个匿名聊天者，并且在他们离开后简单地结束会话。同时我们还需要对用户管理组件做出修改。比如，聊天组与视频组是否需要互相对应？

因为只有一个用户提出了这个需求，我们很可能需要使用某种方式来关闭它。不仅是因为这个新功能有可能会减损应用的核心价值“视频会议”，更重要的是它会在为其他用户部署应用时带来麻烦。新的后端服务会增加接口的复杂度，还会需要更多的培训和支持，这就是为什么不是所有的组织都需要开启这个功能。如果我们的架构没有开关组件的能力，那么就是有其他因素影响了我们的可扩展能力。

开发的可扩展性

最后一个需要克服的，影响可扩展性的因素就是软件开发者。任何足够复杂的 JavaScript 应用都不太可能是由一个开发者单独开发的。即便是在特殊的自发组织的开源项目中，通常也会由一个团队来开发。在其他机构中，团队及团队中的角色都有更具体的定义。无论团队是如何组织的，本章将会直接讨论如何扩展团队。

寻找开发资源可能是我们在项目初期遇到的第一个问题。一个团队组成不是静态的，

随着软件代码和解决方案的增长，必须加入新的开发资源。无论喜欢与否，最好的资源最有可能离开，因为他们是最受追捧的。理想情况下，我们可以抓住优秀的开发人员不放，但同时必须继续招聘新的开发资源。聘用 JavaScript 程序员的方式和时间由需要实现的功能以及为实现这些功能而提供的架构来决定。

从日常的角度来看，团队的每个成员都应该负责实现一个特定的应用程序块。这是一个复杂的问题，同时也是一个可扩展的影响因素。我们必须小心定义团队中的角色，不要附加过多的限制。当受到影响因素影响时，需要调整并进行交付。硬性的角色定义帮不上什么忙，恰恰相反，如果组件的开发方式有一定的自主性，我们需要尝试尽量减少边界的设置。


最后，尝试找出一个可靠的方法来确定我们是否拥有过多的开发资源。这听起来像是一件坏事：我们拥有了所有的这些人才，他们还有要做的工作。这二者似乎看起来总是如影随形，不是吗？不，不总是。

寻找开发资源

对于产品管理者来说为将来的计划雇佣更多的开发人员，比为当前的工作雇佣恰到好处开发人员更具有诱惑性。但是出于某些原因，这并不能很好地进行扩展。这个场景下新员工可能会面对的第一个问题就是无法通过真实的功能学习已有的代码。记得吗？我们雇佣他们是希望他们来开发路线图中尚未开发的新功能。所以，即使他们想要帮忙，但是我们还没有赋予他们真正的职责。经过一两个星期，他们就会开始极力远离那些试图结束工作的人们。

最好思考一下我们现在正在做什么。软件下一个版本要交付的功能与我们的能力之间是否有明显的差距？如果没有明显的差距，就不需要招收新的程序员来弥补。招收新人只会带来不必要的沟通成本。这样做的缺点是一旦我们明确了在开发某些功能的能力上有差距，就无法立刻找到我们需要的资源。而这种招聘压力可能会导致雇佣不适合的开发者，进而导致新雇佣的开发者因为某种原因而无法融入团队。

比较好的可扩展的开发资源的成长方式是，等到有了开发能力的差距之后再进行招聘。并非要等到世界末日或公司即将破产，而是等到我们可以把工作做得更好。如果可以避免，就不应该一次雇佣多个开发人员。只要花时间，就能找到可以填补差距的合适的开发者。

【 Fred Brooks 的《人月神话》是介绍软件开发生命周期过程中沟通开销的经典书籍。】

开发职责

Web 浏览器是囊括了很多技术和部件的复杂平台。Web 平台的某些组件与其他组件相比，很多都处于未开发状态。但是对我们来说，理解它仍然很重要。这些新兴技术是网络的未来。那么，我们的团队选择谁去学习这些新技术，并在整个组织传播呢？Web 平台带来的挑战是，需要掌握的知识已经超出了一个人在工作同时可以掌握的合理范围。这就是为什么我们在某种程度上需要开发角色的原因。

这些开发角色之间的界线取决于所在的组织和文化。同时，需要开发的应用本身也会影响开发角色的设置。角色的设定没有现成的方法，同时也应该避免严格的设定。这是因为我们需要适应可扩展性影响因素带来的各种改变，而严格的角色设定会阻碍有能力的开发者的发挥。通常，随着最后交付日期的邻近，根本没有时间去争论角色的界线。

前端架构师是最有可能看清，哪些角色对给定应用是有意义的。而这些角色并不是长期不变的角色，它们由架构师指导，由成员有机地形成。在开源项目中，这种特点尤其明显。开源项目中的成员们做自己喜欢并且擅长的事情。虽然我们不能照搬这种模式，但是可以从中找到线索——根据大家在应用功能需求的范围内都善于做什么事情来决定角色。这样可以帮助开发人员得到他们需要的指导。对 JavaScript 开发的某些方面感兴趣并不意味着他们达到了需要的水平。这时就需要资深开发者作为导师给以指引，让他们做自己喜欢做的事情。长远来看，这样做有巨大的收益。

资源过多

我们已经提到了一部分容易雇佣过多开发资源的问题。当已经有由产品管理定义好的清晰的路线图时，我们希望充分确定自己确实有充足的开发资源来履行路线图。雇人速度过快不可避免地会导致开发资源过多。如果已经处于开发资源过多的状态，我们应该怎么办？

如果我们对团队成员不满意，答案很明显，我们的资源比实际需要的要多。不过还有

另一种方式来看待这件事情。如果我们有太多不想失去的资源，就必须调整产品路线图，以适应已经招到的开发人员。这意味着我们需要寻找一种渠道，让有关产品的点子从开发达到产品管理。这是一门艺术，而非科学。

这是一个具有挑战性的工作。作为一个前端架构师，搞清楚谁去做什么很重要。调整开发资源的最佳方法就是为实现架构的开发者提供精确的架构图。如果有矛盾的地方就找出正确的前进方向。比如，我们需要更多的 JavaScript 开发者来填坑，或者由于资源过多而需要在产品上有所变化。

扩展开发示例

我们的应用程序已经发布了一段时间，有了一些成果，并且部署在多种环境下。瑞安是我们的一个核心开发者，他写的代码涉及许多领域。他会帮助其他开发人员提高代码质量，向他们提出建议，等等。我们已经注意到自己的应用程序已经大到让所有功能都开始有性能上的下降。

我们需要瑞安做一些性能优化。优化需要重构部分代码，基本上占据了他所有的时间。但是与此同时，如果我们打算满足客户的需求，就仍然需要持续交付新的功能。但另一方面，从性能角度看，我们的扩展性能遇到了瓶颈。

这时我们意识到，需要雇佣一个新的开发人员来开发新功能。他/她不需要像瑞安那样厉害。只需要基本的开发能力，熟悉我们正在使用的技术。幸运的话，我们会雇到合适的人来承担更多的责任。但是眼下，瑞安留下的缺口并不大。所以如果是为了可扩展，我们并不急需另一个瑞恩那么厉害的开发者。

影响因素检查表

本章结束时会给出一些检查清单。清单中有一些简单的问题，这些问题没有唯一的答案。而有些答案在我们整个软件的生命周期会保持不变。例如，我们希望自己的商业模式不会经常更改。其他一些答案依赖于事物的状态，这也是清单的目的所在。我们一遍一遍地检查清单，只要一有变化就随时检查。可能是检查需求、用户、新的部署或已经发生变化的开发环境。这些问题无非是提醒我们那些影响 JavaScript 应用程序可扩展的微妙因素。如果看到它们会带来更多的问题而非答案，那么清单的目的就已经达到了。

用户检查清单

用户是我们开发软件的第一原因。这个清单包括了为什么应用需要可扩展性的最基本的方面。以下问题贯穿了软件的整个生命周期，而不仅仅发生在用户管理角度出现问题的时候。当功能需求发生变化时，就需要检查一下这个清单。

软件的商业模型是什么？

- 基于许可证的？
- 基于订阅的？
- 基于按需缴费的？
- 基于广告的？
- 开源的？

软件有不同的用户角色吗？

- 某些功能是否对某些用户角色不可见？
- 每个功能是否都是需要考虑用户角色的？
- 角色是如何定义并管理的？
- 我们的商业模型如何影响应用的角色？

用户是否通过我们的软件进行交流？

- 用户是否需要协作来有效使用软件？
- 用户交流是否影响我们的数据模型？
- 用户角色如何影响用户交流？

如何为应用提供支持？

- 技术支持是内建在应用中还是在外部处理？
- 用户是否可以通过知识库互相帮助？
- 商业模型和用户角色如何影响我们提供的支持？

如何收集用户反馈？

- 反馈系统是内建在应用中还是在外部处理？
- 如何激励用户提供反馈？
- 我们提供的支持如何影响我们期望获得的反馈？

如何为用户提供相关的信息？

- 应用是否具有通用的、与内容无关的通知系统？
- 如何保证在某个时间点只发送给用户相关的通知？
- 用户是否可以审核通知？

应该收集哪些用户信息？

- 是否使用收集的用户信息改进产品？
- 产品是否可以在运行时利用用户信息改进用户体验？
- 商业模式如何影响我们对收集信息的需求？

功能清单

以上清单是源自用户的对可扩展性的影响因素，以下清单则是源自我们的软件功能，包括了新功能以及功能变化等问题。这些问题将帮助我们应对与功能相关的，影响可扩展性的一些因素。

软件的核心功能是什么？

- 正在实现或改进的功能是否有助于实现产品的核心价值？
- 目前的价值主张是否过于广泛关注？
- 用户数量和用户角色如何影响我们对应用价值相关功能的关注度？

如何确定功能的可行性？

- 是否在实现杀手级功能，而不是顺其自然？
- 是否花时间来决定某个提议的功能是否可行，而不是草草实现它？
- 软件的价值和用户期望的功能如何影响最终要实现的功能的可行性？

是否可以做出明智的关于软件功能的决定？

- 是否有可以作为决策依据的用户数据？
- 是否有以前实现过的类似功能的历史数据？
- 商业模型如何影响那些我们可以采集并用之做决策的应用功能方面的数据？

竞争对手是谁？

- 我们是否提供了与竞品类似的功能？是否做得更好？
- 我们是否属于某个细分市场？
- 我们是否可以从竞品中学到什么？
- 我们的商业模型如何影响我们所面临的竞争以及需要实现的功能类型？

如何将已有的功能做得更好？

- 以我们增加功能的速度来看，是否有足够的事件来维护已有的功能？
- 从架构上讲，修改功能是否安全且不破坏其他功能？
- 用户如何影响我们改进已有功能？
- 商业模型如何影响我们部署改进的产品？

如何将用户管理集成到软件功能中？

- 访问控制机制是否普遍到平日的功能开发不需要关心的程度？
- 我们是否可以将功能分组？
- 用户是否可以开、关某些功能？
- 我们构建的应用类型，以及其用户和用户角色如何影响软件功能的复杂性？

软件功能是否与后端服务紧密耦合？

- 已有服务是否足够通用，可以处理新的需要实现的功能？
- 是否可以模拟后端服务，完全在浏览器中运行？
- 软件的功能如何影响后端服务的设计与能力？

前端如何保持与后端数据同步？

- 是否可以利用 Web Socket 连接来推送通知？
- 频繁的用户操作是否会导致更多的消息发送给其他用户？
- 消耗实时数据如何影响功能的复杂性？

开发者清单

最后，我们要在整个软件开发过程中检查关于开发资源的清单。这个清单不会像用户清单和功能清单那样被频繁检查。尽管如此，我们还是要确保已经强调了在开发资源方面的考虑。

如何找到合适的开发资源？

- 现有的开发资源是否够用？
- 是否需要重新审视已有的开发资源能否适应开发中的功能？
- 是否有合适的开发资源来开发当前的产品？

如何分配开发责任？

- 责任区间应当有多少重叠的部分？
- 当前的责任区是否反映了我们正在构建的功能？
- 团队的各种技能集合如何影响责任区的划分？

是否可以避免雇佣过多的开发者？

- 是否雇佣了过多的开发者？
- 是否因为开发者过多而带来了沟通负担？
- 并行开发的功能数量如何影响开发者越多越好的认知？

小结

当谈到影响 JavaScript 应用可扩展性的因素时，有三个主要领域需要关注：用户、功能和开发。自上而下，每个领域直接影响其下方领域。

首先是软件用户。有很多与用户相关的影响因素决定了软件的可扩展性需求。例如，商业模式潜移默化地影响着软件架构。基于许可证的部署很可能被现场部署到某处，因此更有可能需要定制。复杂性的组合是多种多样的，且都源自软件用户。

下一个主要的领域是应用的功能。我们必须思考用户及用户对可扩展性的影响，并尽可能深入地理解这些影响，然后将这些理解应用到功能设计中。比如，一旦人们开始使用我们的软件，在很短的时间里就可能发生很多事情，这些事情将如何分散我们对应用的核心价值的注意力？不管你信不信，专注力也是需要可扩展的。

最后，还有软件的开发。开发团队需要打造。找到合适的人才并不容易，即使我们已经有了这群靠谱的开发者，每个人的责任以及软件功能和用户对他们的影响，也需要考虑进来。随着应用程序的开发，我们必须确保正确的资源就位。

既然我们已经奠定了前端可扩展性的基础，接下来就可以深入到其中的具体细节了。本书其余部分会将前两章的概念转化为 JavaScript 代码。我们已经知道了什么是可扩展性的影响因素，现在需要做出架构上的取舍。这是最有趣的部分，因为我们可以编写代码了。

3

组件组合

大规模 JavaScript 应用实际上相当于一系列组件之间的互相通信。本章的重点是组件之间的组合 (Composition)，下一章我们再讨论组件之间的通信。组件组合是一个大话题，同时也是与可扩展 JavaScript 代码相关的一个话题。当开始思考组件之间的组合时，就会注意到组件设计上的缺陷和局限性。正是这些缺陷和局限性，导致我们无法扩展自己的代码和设计。

组件并不是随意组合的，现在已经有了一些流行的 JavaScript 组件模式。在本章的开始，将先了解一些在每个 Web 应用中都会用到的、封装了常用模式的通用组件。为了能规模化地扩展通用组件，理解组件实现的某些模式是至关重要的。

从纯技术角度正确恰当地组合组件是一回事儿，轻松地将组件映射到应用的功能特性就是另一回事儿了。对于已经编写实现的组件来说，也是这样的。我们的编写代码需要提供某种程度上的透明度。这样，不管在运行时还是设计时，都可以分解我们的组件、理解它们的功能。

最后，将了解如何将业务逻辑与组件解耦。这并不是什么新鲜的思想，这种分离的思想由来已久。JavaScript 应用带来的挑战在于它涉及太多方面，以至于很难把业务逻辑和其他涉及到的方面泾渭分明地分离开来。我们组织代码的方式（相对于使用它们的组件）对可扩展性有着很大的影响。

通用组件

如今，编写大规模 JavaScript 应用或框架，不太可能不使用类库。我们统称这些为工具，因为我们更关心哪些工具可以帮助我们做到可扩展，而不是哪个工具比另一个更好。最终还是由开发团队决定哪个工具最适合应用的开发，而不是个人的偏好。

在选择要使用的组件工具时，指导因素是工具能提供的类型和功能。比如，一个大型 Web 框架有我们需要的所有通用组件。另一方面，函数变成工具库能提供许多我们需要的低阶功能。如何将这些有机地结合起来，就是我们要思考的内容。

解决方法是找到拥有我们所需要的通用功能的工具。通常我们会扩展这些组件，编写应用特有的功能。本章将介绍一些大型 JavaScript 应用需要的典型组件。

模块

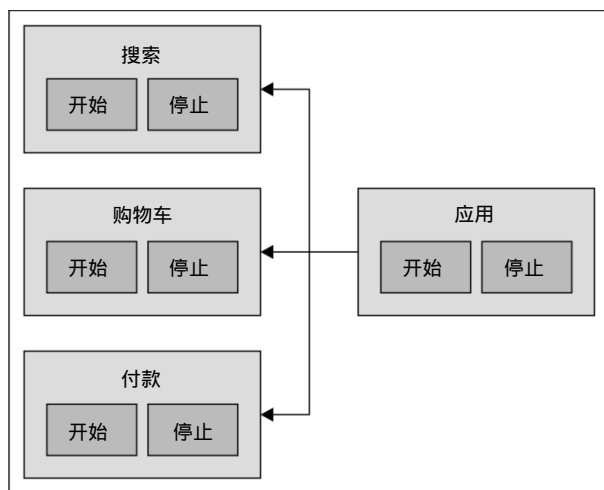
有人说除了 JavaScript 几乎所有编程语言都以某种形式支持模块。其实这并不正确：在写作本书时，ECMAScript6 的最终草案中已经引入了模块的概念。尽管如此，现在已经有一些工具不依赖 `script` 标签就可以模块化代码。大规模的 JavaScript 工程仍然是相对比较新的事物，`script` 标签也不是用来解决代码模块化和依赖关系管理的。

RequireJS 可能是当下最流行的模块加载、依赖解析工具。仅仅是为我们的前端应用加载模块就需要一个库，这说明问题比较复杂。比如把网络延时和竞争态条件考虑进来时，模块依赖就没有那么简单了。

另一个选择是使用诸如 Browserify 的转译器。这种方法引起人们注意，是因为它可以用 CommonJS 的格式来声明模块。而这种格式正是 NodeJS 使用的格式。比起 AMD，这种格式更接近 ECMAScript 的模块标准。所以它的优势是，使用这种格式写的代码可以更好地与后端代码和未来的标准相兼容。

一些框架有自己的模块概念，比如 Angular 和 Marionette，不过更抽象些。

这些模块更多的是关于如何组织代码，而不是如何将代码从服务器发送到浏览器。这种类型的模块甚至可以更好地映射到框架的其他功能。比如，若有一个用来管理模块的中央应用实例（centralized application instance），那么框架可能会提供一种管理这些模块的方法，如下图所示。



图注：一个全局应用组件使用其他模块作为基础构建模块。这些模块可以小到只有一个功能，也可以大到包含很多功能。

上面这种组织结构让我们可以执行更高层次的任务（比如关闭某个模块或设置模块的参数）。本质上，模块代表功能，是一种打包机制，这种机制允许我们将一组功能封装起来，不用关心应用的其他部分如何实现。模块通过为功能增加高层次操作、将各个功能视为建筑模块，帮助我们扩展应用。如果没有模块，就没有好的方法做到这些。

模块的构成会根据声明模块的机制不同而不同。模块可以简单地将一组功能、对象通过一个名字空间暴露出来。如果选择了某个特定框架的模块声明方式，可能就不止这么简单了。比如自动事件生命周期，或者用来执行例行设置任务的方法，等等。

无论如何划分，在可扩展的 JavaScript 应用中，模块都是用来创建基础构建模块、处理复杂依赖关系的工具。

```
// main.js
// 从util.js 中引入log函数。
import log from 'util.js';
log('Initializing...');

// util.js
// Exports a basic console.log() wrapper function.
'use strict';

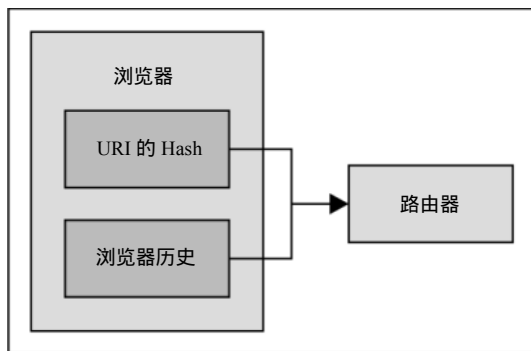
export default function log(message) {
  if (console) {
    console.log(message);
  }
}
```

很容易通过模块级别的基础构建来构建大型应用程序，将模块剥离应用并独立运行也同样容易。如果应用程序是无模块的一个大程序或者模块太多粒度太细，都很难从代码中发现问题，也很难进行测试。我们的组件本身可能工作得很好，只是影响了系统的某个地方而产生了问题。但是如果我们像拼图那样，轻松地一块一块移除代码，就可以使排除故障的过程变得可扩展。

路由器

任何一个大型的 JavaScript 应用都会有很多的 URI，URI 是用户看到的页面的地址。用户可以通过单击链接跳转到相应的资源，也有可能是代码为响应用户的某些操作而自动跳转的。网页在大型 JavaScript 应用程序问世之前就一直依赖于 URI。URI 指向资源，而资源可以是任何东西。应用程序越大，资源也就越多，URI 也就越多。

路由器组件是前端用来监听并响应路由变化的工具。这样就可以更少地依赖于后端服务器来解析 URI 并返回对应内容。现在大多数网站仍然使用这种方法，但是这种方法在构建应用时有一些缺点。



图注：当URI发生变化时浏览器触发对应事件，路由组件响应这些变化。URI的变化可以是浏览历史API触发的，有可能是`location.hash`变化触发的。

最主要的问题是我们希望用户界面是可移植的。也就是说不管后端是用什么技术开发的，前端应用都可以正常地部署和工作。既然我们不是为后端的 URI 组装页面，那么也就不需要让后端来解析路由。

我们在路由器组件中声明所有用到的 URI 模式，这些简称为**路由**。如果把路由当作蓝图，URI 就是蓝图的一个事实例。这就意味着当路由器收到一个 URI，就会将之与一个路由相关联。这其实就是路由器的职责。小型应用还比较简单，但如果考虑到可扩展性，那么还有更多路由器的设计问题需要考虑。

首先，需要考虑一下我们想要设计一个怎样的 URI 机制。有两个基本的选择，一是监听 `location.hash` 改变事件，二是使用浏览历史 API。第一种方法相对简单，第二种方法可以在所有的现代浏览器中使用。同时第二种方法不会在 URI 中出现“#”，所以看起来更实像一个真的 URI。框架中的路由组件可以只支持第一种或第二种方案，使用者无须选择。有的路由器组件同时支持两种方案，这时就需要选择其中最适合我们应用的方案。

其次，需要考虑的是如何响应路由变化。通常有两种方法。第一种方法是声明一个路由并绑定回调函数。当时当路由比较多时这种方法就不太理想了。第二种方法是路由被激活时触发事件。这就意味着没有什么事件直接与路由绑定，而是由一些组件监听这些事件。当路由很多时这种方法就非常有用。因为路由器对这些组件一无所知，而只知道路由。

以下示例展示了路由组件监听路由事件。

```
// router.js

import Events from 'events.js'

// 路由器是一种事件代理 (Event Broker), 可以触发路由并监听路由变化。
export default class Router extends Events {

  // 当传入一个路由配置对象时, 我们遍历它上面的路由,
  // 并对每一个路由名称调用listen()函数。
  // 这样就将路由对应到了事件监听器。
  constructor(routes) {
    super();

    if (routes !== null) {
      for (let key of Object.keys(routes)) {
        this.listen(key, routes[key]);
      }
    }
  }

  // 当调用者准备好响应路由事件时调用start()方法。
  // 在这个方法里监听onhashchange事件。
  // 并手动调用onHashChange方法来处理当前路由。
  start() {
    window.addEventListener('hashchange',
      this.onHashChange.bind(this));

    this.onHashChange();
  }

  // 当发生路由变化时触发相应事件。
  // 记住, 这个路由器同时还是一个事件代理。
  // 事件名是当前URI。
  onHashChange() {
    this.trigger(location.hash, location.hash);
  }
}
```

```
};

// 创建一个路由器实例，然后用两种不同的方式监听路由变化。
//
// 第一种方式，将配置传给路由器。键是实际路由，值是对应的回调函数。
//
// 第二种方式，使用路由器的listen()方法，事件名是实际路由。
// 当路由被激活时，对应回调函数被调用。
//
// 在start()函数被调用前不会触发任何代码，这就给我们提供了设置的时机。
// 比如，响应路由变化的回调函数在可以调用前可能需要某些配置。

import Router from 'router.js'

function logRoute(route) {
  console.log(`${route} activated`);
}

var router = new Router({
  '#route1': logRoute
});

router.listen('#route2', logRoute);

router.start();
```

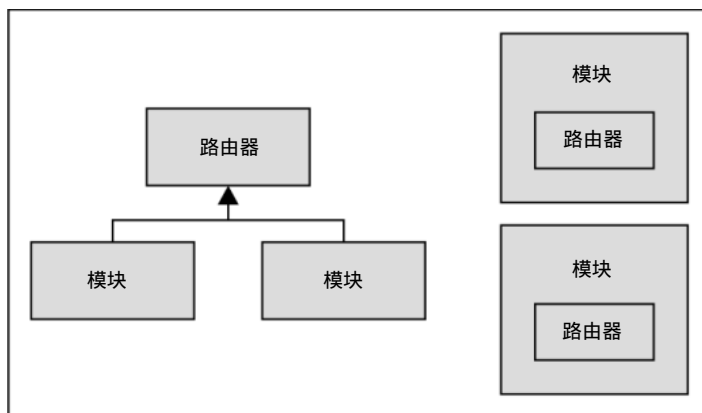


运行以上示例所需的部分代码并没有完全列出。比如 `event.js` 模块。这部分代码包含在随书代码包中。没有列出是因为它与示例关系不那么密切。同样，出于版面考虑，示例代码避免使用特定的类库。实际中，我们不会自己编写路由器或事件 API。因为我们使用的框架已经有了相关组件。我们只是用原始的 ES6 JavaScript 代码来展示与可扩展性相关的内容。

在架构方面关于路由需要的考虑是，我们希望设计一个全局单一路由，还是每个模块一个路由，或是其他组件。全局单一路由的问题是当不断地增加功能和路由时，应用会变

得越来越大，它会变得越来越难以扩展。它的优点是所有路由都声明在同一个地方，单一路由还可以触发所有组件都可以监听到的事件。

在第二种方案中每个模块都有一个路由器实例。比如，如果应用中有五个组件，那么每个组件都有它们自己的路由器。这样的好处是每个模块都是自包含的，使用者不需要查找对应路由。另外这种方案中路由定义和响应函数之间耦合更紧，这意味着代码会更简洁。这种方案的缺点是路由声明分散在各个模块，而不是集中存放，如下图。



图注：左边的是一个全局路由器。所有的模块都使用它来响应URI事件。右边的模块拥有自己的路由器。这些路由器拥有自己单独的配置，而非全局配置。

路由器组件是否支持多路由器取决于使用的框架。每个路由可能只有一个回调函数。每个路由的事件可能还有一些不知道的细微差别。

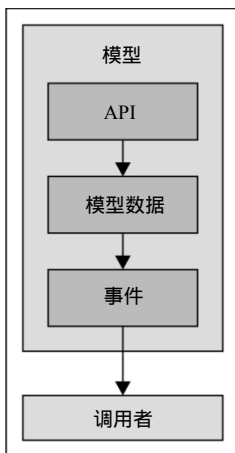
模型/集合

与我们应用交互的 API 会暴露一些实体。一旦这些实体通过网络传送到浏览器，我们将会把它们保存为数据模型。集合是一组相关的实体，通常有着相同的数据类型。

我们使用的工具可能会也可能不会提供通用模型和集合组件，或者他们可能有一些相似但命名不同的组件。API 数据建模的目标是近似接近 API 实体。可以简单地将模型存储为 JavaScript 对象，将集合存储为数组。

简单地将 API 实体保存为对象和数组有如下问题：其他组件需要实现调用 API，在数据发生变化时触发事件，对数据进行变换。我们希望其他组件可以对模型和数组进行变换

以完成他们的任务，但不希望有很多重复代码。最好可以将诸如数据变换、API 调用、事件生命周期的公共用代码封装起来，如下图。



图注：模型封装了与API的交互、解析数据、当数据发生变化时触发的事件。这可以让模型外其他部分的代码变得更加简单。

将 API 数据如何加载进浏览器、如何发出命令的实现细节隐藏起来，可以使应用程序在成长过程中变得可扩展。随着我们不断为 API 增加实体，代码的复杂度也在不断增加。我们可以通过约束与模型和集合组件交互的 API 来控制代码的复杂度。

我们面临的另一个关于模型和集合的可扩展性的问题是，这两个组件应该被放在什么地方？应用程序是一个大组件，由很多小的组件组成。模型和数据集合可以很好地映射到 API，但并不一定能很好地映射到应用的功能。API 实体比特定的功能更加通用，而且会有更多功能使用。这就给我们带来了一个开放性问题：在众多组件中我们的模型和集合应该处于什么位置？

下面例子中的特定视图继承自通用视图，同一个模型可以传递给任意一个视图。

```
// 一个非常简单的模型类。
class Model {
    constructor(first, last, age) {
        this.first = first;
        this.last = last;
        this.age = age;
    }
}
```



```
}

// 视图基类，有一个name()方法输出一些信息。
class BaseView {
  name() {
    return `${this.model.first} ${this.model.last}`;
  }
}

// 扩展BaseView。构造函数接受唯一参数model，并保存到this.model属性
class GenericModelView extends BaseView {
  constructor(model) {
    super();
    this.model = model;
  }
}

// 使用特定参数扩展GenericModelView。
class SpecificModelView extends BaseView {
  constructor(first, last, age) {
    super();
    this.model = new Model(...arguments);
  }
}

var properties = [ 'Terri', 'Hodges', 41 ];

// 确保视图之间的数据是相同的。
// name() 方法应该返回相同内容.....
console.log('generic view',
  new GenericModelView(new Model(...properties)).name());
console.log('specific view',
  new SpecificModelView(...properties).name());
```

一方面，相对于组件使用的数据和集合，组件可以是完全范化（Generic）的；另一方面，一些组件是针对特定需求的，他们可以直接实例化对应的数据集合。在运行时用模型和集合配置通用组件带来的好处是有条件的：通用组件必须足够通用，能在多处使用。否

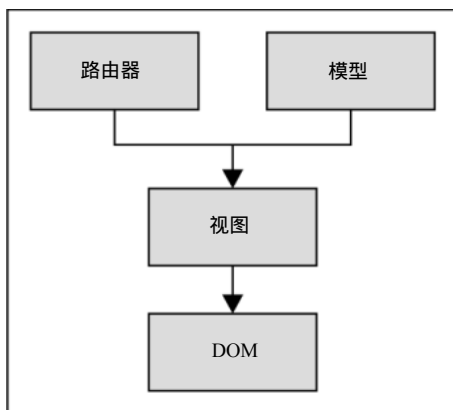
则我们不如将模型和组件封装在一起。选择正确的方法可以帮助我们实现可扩展，因为组件不是全部通用或全部特定的。

控制器/视图

团队使用的框架和遵循的设计模式不同，控制器和视图所代表的就可能不同。由于有太多的 MV* 模式，所以很难从可扩展性的角度，给出它们之间的有意义区别。这些 MV* 模式相似却又不同，他们之间的细微差异各有利弊。为了方便讨论大规模 JavaScript 工程，我们将视它们为同类型的组件。如果我们决定在实现中分离这两个概念，本章中的一些观点对这两种类型同样适用。

我们会在后面使用视图这个术语，本书中视图指视图和对应的控制器。视图组件会与其他众多组件进行交互，包括路由器、模型、集合以及模板。下一章将会讨论模板。当有事情发生的时候，我们需要通知用户。视图的任务就是更新 DOM。

可以是简单地更新 DOM 节点的属性，也可以是重新渲染新的模版。



图注：视图组件更新DOM以响应路由器和模型事件。

很多类型的事件都可以引起视图更新 DOM。比如：路由发生变化，或者模型发生了变化，也可以是一个方法直接调用了视图。更新 DOM 并不像想象中那样直接了当，因为我们需要考虑性能问题。比如，当视图接收大量事件时会发生什么？我们还需要考虑代码的延时。比如，在允许 DOM 渲染之前，JavaScript 代码调用栈会有多深？

视图的另一个职责是响应 DOM 事件，这种事件通常是用户与界面交互时触发的。交

互可能起于视图也止于视图。比如，我们可能根据用户输入或模型的状态，用一条消息更新 DOM，亦或是在事件处理函数被**消除抖动**（debounce）时什么都不做。

消除抖动函数会把多次调用合并成一次。比如 10 毫秒内调用二十次 `foo()` 实际上只调用了一次。更详细的解释可以参见以下文章：<http://drupalmotion.com/article/debounce-and-throttle-visual-explanation>。多数情况下 DOM 事件会转换为函数调用或其他事件。比如，我们会调用模型上的方法或者对集合进行变换。多数情况下，最终会通过更新 DOM 来提供反馈。

我们可以直接或间接地更新 DOM。直接更新 DOM 相对比较容易可扩展化，间接更新或通过副作用更新就比较有挑战性。这是因为随着应用需要的部件越来越多，越来越难将间接调用的原因和影响想清楚。

下面的例子中展示了视图正在监听 DOM 事件和模型事件。

```
import Events from 'events.js';

// 一个简单的模型。它继承自Events。
// 所以它可以监听其他组件触发的事件。
class Model extends Events {
  constructor(enabled) {
    super();
    this.enabled = !!enabled;
  }

  // enabled属性的setter和getter。
  // 设置这个属性会触发事件。
  // 所以其他组件可以监听enabled事件。
  set enabled(enabled) {
    this._enabled = enabled;
    this.trigger('enabled', enabled);
  }

  get enabled() {
    return this._enabled;
  }
}
```

```
}

// 视图组件需要一个模型实例和一个DOM元素作为构造参数。
class View {
  constructor(element, model) {

    // 当模型触发了enabled事件，就调整DOM属性。
    model.listen('enabled', (enabled) => {
      element.setAttribute('disabled', !enabled);
    });

    // 当元素被单击后设置模型的状态，这会触发上面的监听器。
    element.addEventListener('click', () => {
      model.enabled = false;
    });
  }
}

new View(document.getElementById('set'), new Model());
```

代码变得复杂带来的一个好处是我们可以获得可重用的代码。视图对它所监听的模型和路由是如何更新的一无所知，它所关心的只是特定组件上的特定事件。事实上这会帮助我们减少对特殊情况的处理。

在运行时渲染视图得到的 DOM 结构也需要考虑进来。如果查看一下顶级的 DOM 节点，你会发现它还有内嵌的 DOM 结构，正是这些顶级节点构成了页面的布局。它可能被应用的主视图渲染，而其他视图都是它的子视图。但是请记住，复杂的视图关系很难做到可扩展。

模板

模板引擎大多位于后端框架，但如今并不尽然。这主要得益于前端成熟的模板引擎库。在大型应用中我们很少与后端服务交换 UI 相关的内容。我们不会让后端帮助渲染一个 URL 对应的页面。目前的趋势是给组件自主性，让它们自己决定如何渲染。

让组件和其对应的 HTML 相耦合是件好事。因为这就意味着，我们可以很容易地分辨

出 DOM 中的 HTML 是在哪里生产的。从而可以更好地诊断问题，优化大型 JavaScript 应用的设计。

模板有助于组件之间的分离。浏览器渲染的 HTML 大多来自于模板，这有助于将模板相关的代码与其他逻辑相分离。前端模板引擎并不是简单的字符串替换工具。前端模板通常会提供工具来减少 JavaScript 样板代码。比如，在合适的地方我们可以插入条件判断或 for-each 循环。

应用特定的组件

目前我们讨论的组件类型有助于编写可扩展的 JavaScript 代码。但它们又比较通用，所以在开发中不可避免地会遇到问题。我们遵循的组件组合模式可能并不能满足特定功能的需求。不过这也可能是我们思考架构增加新组件的时机。

比如考虑增加小部件。小部件是一些通用组件，主要关注展现以及用户交互。比如很多视图都使用相同的 DOM 元素和相同的事件处理函数，那么就没有必要在应用中的视图中重复这些内容。如果把它抽离为一个公用组件是不是更好呢？但可能没必要抽离成一个视图，所以也许我们需要一个新的小部件组件。

有时候，我们仅仅出于组合目的而创建组件。比如我们会创建组件，将路由器、视图、模型/集合和模板组件连接为一个单元。模块可以达到这个目的，但并不是所有情况都够用。有时候，我们缺少一种组件之间通信的方法。下一章将会讨论这部分的内容。

扩展通用组件

我们通常会在开发过程的后期发现组件缺少一些需要的功能。如果基础组件设计良好，我们可以扩展它，加入需要的属性和功能。在本节中将会介绍一些需要在扩展应用中广泛使用组件的场景。

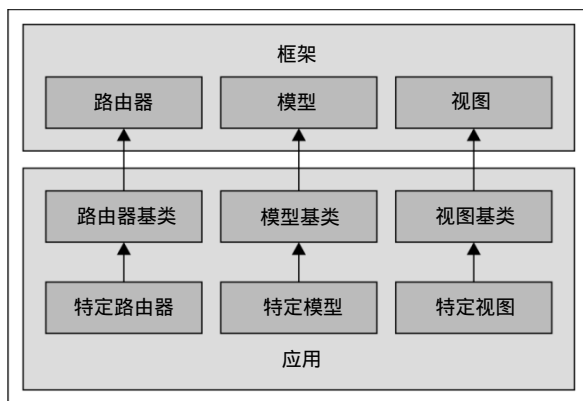
如果希望我们的代码可扩展，就需要充分利用这些基础组件。

识别公用数据、功能

在扩展特定组件类型之前，值得花时间思考一下，哪些属性和方法是所有组件公用的。

有些是显而易见的，另一些则不然。组件的可扩展能力一部分取决于我们识别公共部分的能力。

如果我们有一个全局应用对象实例，那么全局变量和方法都可以放在这里。这在大型 JavaScript 应用中很常见。但随着公用部分的增多，这种方法会变得难以驾驭。另一种方法是创建多个全局模块来代替一个全局应用实例，或者是二者同时使用。但从可理解性来讲，这种方法是不可扩展的。



图注：理想的组件架构不会超过三层，应用依赖的组件通常在框架的最顶层。

以往经验来看，应该避免对任何组件扩展在三层以上。这就意味着任何需要扩展通用视图的组件都不会变得特别复杂。比如，我们可以通过扩展框架中的通用视图组件得到我们自己的通用版本的视图组件。这个通用版本视图组件会包含我们应用中每个视图实例需要的属性和功能。这样只有两层结构，便于管理。这就意味着任何需要扩展我们的通用视图的组件都可以放心扩展，而无须担心将事情弄复杂了。任何类型的最大扩展层级应该是三层，正好可以避免不必要的全局数据。多于三层就会带来可扩展性问题，因为整体层次会难以掌握。

扩展路由器组件

我们的应用可能只需要一个路由器实例。尽管如此，可能仍然需要重写通用路由器的某些扩展点。如果是多个路由器实例，就会有我们不希望重复的公共属性和方法。如果应用中的每个路由都遵循相同的模式，仅仅是有细微差别，那么就可以在基路由器（base router）中实现相同的部分，以避免重复的代码。

除了声明路由之外，路由被激活时都会触发路由事件。并且，根据应用架构图不同，会有不同的事件发生。甚至不管哪个路由被激活，有些事件总会发生。而这正是扩展路由器以提供便利方法的地方。比如，我们需要对某个路由进行权限检查。那么完全没有必要检查每一个组件，因为复杂的权限控制和众多的路由将会不利于扩展。

扩展模型/集合

无论具体怎样实现，模型或集合都会共享一些公共属性。尤其当它们以同一 API 为目标时，这更是最常见的情况。特定的模型或集合是以 API、返回的数据和可能采取的动作（action）为中心的。对于所有实体，我们很可能以相同的 API 为目标，而这些实体拥有少部分的共享属性。我们更倾向于提取出共享数据，而不是在每个模型和集合里都重复一遍。

除了在数据和集合中共享属性，还可以共享相同的行为。比如，对于某个功能，对应的模型很可能没有足够的数据。也许这些数据可以从模型中变换而来，这时公用的变换就可以抽取到基模型和集合中。这很大程度上取决于我们实现的功能类型和它们之间的一致程度。如果应用成长迅速，并有很多意料之外的功能，就会更倾向于在视图中进行数据变换，因为这些变换会对使用的数据和集合进行一次性的修改。

大多数框架都会帮助我们处理通过 XHR 获取数据的细枝末节。不过很遗憾，这并不是全部事实。因为很少有功能可以和 API 实体一一对应。大多数情况下个功能需要若干个相关集合和一个变换后的集合。这时操作很快会变得非常复杂，因为需要多个 XHR 请求。

我们通常会用 Promise 去同步这些请求，然后在拿到所有需要的数据之后再进行数据变换。

下面的代码示例展示了特定的模型通过扩展通用模型来提供新的获取数据的行为。

```
// BaseModel中的fetch() 设置一些属性值并返回一个Promise实例。
class BaseModel {
  fetch() {
    return new Promise((resolve, reject) => {
      this.id = 1;
      this.name = 'foo';
      resolve(this);
    });
  }
}
```

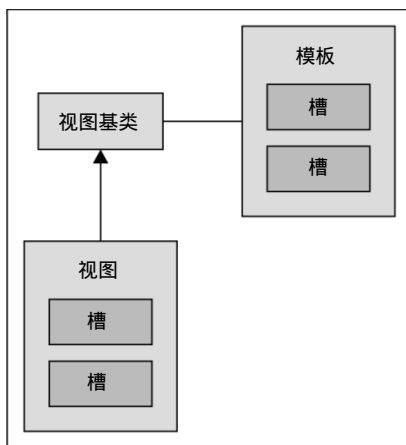
```
    }  
  }  
  
  // 特定模型类扩展BaseModel，并重写了fetch()。  
  class SpecificModel extends BaseModel {  
  
    // 重写基类的fetch()方法。  
    // 调用基类的fetch方法并返回fetchSettings()函数的结果。  
    fetch() {  
      return Promise.all([  
        super.fetch(),  
        this.fetchSettings()  
      ]);  
    }  
  
    // 返回一个新的Promise实例，同时设置一个新的模型属性。  
    fetchSettings() {  
      return new Promise((resolve, reject) => {  
        this.enabled = true;  
        resolve(this);  
      });  
    }  
  }  
  
  // 在fetch()调用完成之后，确保所有属性和希望的一样。  
  new SpecificModel().fetch().then((result) => {  
    var [ model ] = result;  
    console.assert(model.id === 1, 'id');  
    console.assert(model.name === 'foo');  
    console.assert(model.enabled, 'enabled');  
    console.log('fetched');  
  });
```

扩展控制器/视图

当我们使用基模型（Base Model）和基集合（Base Collection）时，控制器或视图之间通常会共享一些属性。因为控制器或视图的任务就是渲染模型和数据集合。比如，如果一

个视图一遍又一遍地渲染同一个模型的属性，那么这部分代码就可以移到基视图中，然后再扩展它。也许模板中也有一些重复的部分，这就意味着我们需要考虑在基视图里创建一个基模板。如下页图所示，所有子视图继承相同的模板。

是否可以使用这种方式继承模板取决于使用的框架或类库，或者是要实现的功能很难使用这种方式。比如，虽然有公用的基模板，但会有很多更小的视图和模板能够插入较大的组件中。



图注：视图可以使用其基视图的模版，也可以继承其他基视图的功能。

视图还需要响应用户交互。可能是直接做出反应，也可能是将事件发送到继承层中的上层组件。不管是哪种情况，如果应用的功能一致，就会有一些公用的 DOM 处理代码，希望将这些代码提取到公用基视图中。这对实现应用的可扩展性有极大的帮助，因为当我们不断地增加新功能时，DOM 事件处理代码的增加量是最少的。

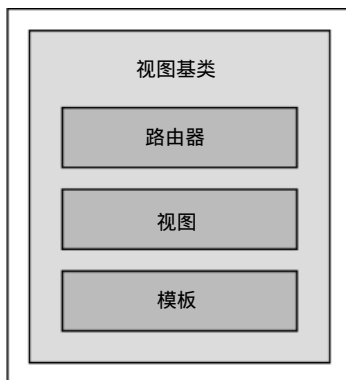
将功能映射到组件

现在我们对通用组件以及如何在应用中扩展它们有了一定的了解，是时候思考如何将这些组件结合起来了。单独使用路由器、模型、模板或控制器是没有用的，需要将它们组合成一个内聚单元来实现应用的功能。

为此需要将应用的功能映射为组件，但是也不能随意地进行映射。需要思考应用功能中有哪些是公用的，哪些是独特的，这样才能写出可扩展的应用。

通用功能

组件组合中最重要的部分就是一致性和重用性。根据应用面临的可扩展性的影响因素，可以列出一些组件必须实现的特性：比如用户管理、访问控制，以及应用独有的一些特点。这些再加上其他架构方面的角度（其他角度会在本书其他部分深入讲解），构成了通用功能的核心。



图注：一个通用组件由框架中的其他通用组件构成。

应用程序各种功能的通用部分为我们提供了编写更大的基础部件的设计蓝图。这些通用功能会帮助我们实现可扩展性。如果在编写聚合组件时考虑到这些因素，就可以轻松地扩展应用。

设计组件时的挑战是，我们不仅要从架构可扩展性的角度考虑，还要从功能完备性的角度考虑。如果每个功能的行为都相同，那么一切都已经准备就绪。只有当每个功能都遵循相同的模式，扩展时才不会有局限。

但是百分之百一致的功能仅仅是幻觉，是不可能存在的。而程序员比用户更容易产生这种幻觉。打破模式是有必要的，而以可扩展的方式来响应这种打破也是很重要的。这也是为什么成功的 JavaScript 应用会持续关注功能的通用性，来确保这些功能反映了现实需求。

特定功能

当需要实现不符合模式的功能时我们将面临着可扩展性的挑战。我们需要权衡引入这

些功能对架构带来的影响，当模式被打破时架构就需要做出改变。这并不是坏事，而是必要的。面对新的需求，可扩展能力取决于已有功能的通用性。这就意味着通用功能组件不能过于死板，否则注定会失败。

在对不寻常的需求做出决定前，请认真思考它对可扩展性带来的影响。比如新的功能是否真的需要使用一套不同于其他功能的布局和模板。可扩展 JavaScript 的最高水平就在于设计出组合组件需要遵循的少数基本蓝图。剩下的问题就是如何遵循并执行。

解构组件

组件的组合是一种创建秩序的活动，但在开发过程中经常会反其道而行。甚至在开发结束之后，还可以通过将组件剥离，来了解组件在不同场景下是如何工作的。解构组件意味着我们可以将系统肢解，以结构化的方式查看独立的部分。

维护和调试组件

随着应用程序的开发，组件会积累各种抽象层。这样做是为了更好地支持功能的需求，同时支持一些有利于扩展的架构方面的属性。但其中的问题是，随着抽象层的不断积累，我们实现了组件功能，但是失去了组件的透明性。这一点不但对问题的诊断和修复很重要，对代码的易学性也很重要。

举个例子，如果间接层过多，程序员就需要花更长的时间进行跟踪，查找原因。把时间浪费在追踪代码上，会降低开发上的可扩展能力。现在我们面对两个相对立的问题：首先，需要抽象层来应对真实世界的功能需求和架构上的约束。其次，由于缺乏透明度，我们无法完全掌控代码。

以下示例中展示了一个渲染组件和一个功能组件。功能组件使用的渲染组件可以被轻松替换。

```
// Renderer的构造函数唯一参数是一个渲染函数。
// render方法返回调用这个函数的结果。
class Renderer {
  constructor(renderer) {
    this.renderer = renderer;
```

```

    }

    render() {
        return this.renderer ? this.renderer(this) : '';
    }
}

```

// feature实例定义一种输出样式，它的构造函数接受三个参数：

// header, content, 和footer。这三个参数都是Renderer的实例。

```

class Feature {
    constructor(header, content, footer) {
        this.header = header;
        this.content = content;
        this.footer = footer;
    }

    // render方法渲染对应视图并返回渲染结果。
    // 视图中的每个区域对一个或零个renderer。
    render() {
        var header = this.header ?
            `${this.header.render()}\n` : '',
            content = this.content ?
            `${this.content.render()}\n` : '',
            footer = this.footer ?
            this.footer.render() : '';
        return `${header}${content}${footer}`;
    }
}

```

// 用负责渲染三个区域的render创建一个feature实例。

```

var feature = new Feature(
    new Renderer(() => {
        return 'Header';
    }),
    new Renderer(() => {
        return 'Content';
    }),

```

```
new Renderer(() => {
    return 'Footer';
})
);

console.log(feature.render());

// 删除header区域。用新的renderer替换footer，并检查输出。
delete feature.header;
feature.footer = new Renderer(() => {
    return 'Test Footer';
});

console.log(feature.render());
```

可替换性可以帮助我们解决这两个对立的可扩展性影响因素。所谓可替换性就是我们是否可以轻易地替换某个组件或子组件。在介绍抽象层前,需要考虑替换组件的容易程度。这可以帮助我们学习、调试代码。

比如,若我们可以将一个复杂的组件从系统中剥离,然后用一个简单的组件替换掉它,就可以简化调试过程。如果替换之后问题消失了,那么问题就出在这个被拿掉的组件。否则就可以排除这个组件,继续查看其他组件。

重构复杂组件

要做到可替换性说起来简单做起来难,尤其当工期短,时间紧时。如果组件替换不太现实,就需要重构代码了,至少是重构那部分替换不了的代码。这样做需要权衡,使封装和透明度都恰到好处。

可替换性可以是更细粒度的。如果某个视图的方法既长又复杂,而方法执行过程可以分为几个阶段,那么最好将方法分解为几个可重写的方法。

可插拔的业务逻辑

并不是所有的业务逻辑都应该被封装在组件中。实际上,最理想的是将业务逻辑放在

一组函数中。理论上这样可以更清晰地分离相关逻辑，组件可以处理架构相关的事情来帮助我们实现可扩展，业务逻辑也可以插到任何组件中。在实践中，将业务逻辑从组件剥离十分重要。

扩展与配置

有两种构建组件的方法。其一，可以扩展已有的库和框架，通过不断扩展逐渐实现特定功能。其二，通过给组件传入配置参数，告诉组件如何工作。

扩展相对于配置的优势是调用者不必关心如何配置。如果可以利用扩展的各种优势，代码就会变得更加简洁，尤其是使用组件的代码。另一方面，可以得到可以用于各种特定情况的通用组件，不过前提是组件必须支持相应配置项。这种方式的好处是组件继承层次简单，数量较少。

有时最好使组件尽可能通用。这样，当需要通用组件实现特定功能时就不需要重新定义组件层次了。当然，因为必须提供相应的配置，组件调用者在使用组件时会变得更复杂。

JavaScript 应用架构设计方面的取舍由我们决定。是将所有东西都封装起来，还是允许各种配置，亦或是在二者之间取得平衡，都取决于我们。

无状态的业务逻辑

函数式编程语言的函数没有副作用。无副作用对某些函数式编程语言是强制的，但 JavaScript 不是。不过我们仍然可以在 JavaScript 中实现无副作用的函数。如果传入相同的函数参数时总会返回相同的结果，那么这个函数就是无状态的。它不依赖于组件的状态，也不改变组件的状态，只是单纯地运算。

如果可以实现一个无状态的业务逻辑库，就可以设计出非常灵活的组件。我们可以把业务行为传入组件，而不是在组件中直接编写，这样不同的组件就可以利用相同的无状态业务逻辑函数。

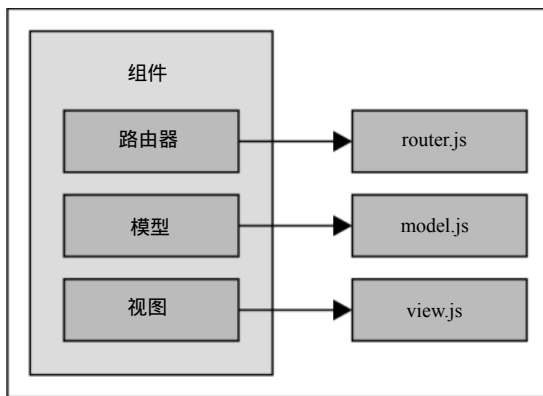
棘手的是如何找到适合这样实现的函数，因为在前期实现并不是一个好主意。但是随着应用不断地开发迭代，可以利用这个策略将代码重构为无状态的函数。这些函数可以在

需要的组件中共享 ,这样就可以使业务逻辑代码更加关注于相应的业务 ,组件更小更通用 ,并且可以在多样的上下文环境中重用。

组织组件代码

组件除了要有助于应用的可扩展 ,还需要考虑源码模块的结构。项目建立时 ,代码文件能很好地与运行在浏览器的代码相对应。随着功能和组件的增多 ,早期的代码组织会削弱这种对应关系。

在跟踪运行时行为时 ,越省力越好。这样 ,就可以把更多的精力集中在当下的设计问题上 ,将应用的功能做得更加稳定。而这正是可以直接提供用户价值的地方。



图注：图表展示了组件到具体实现的映射关系。

从架构方面讲 ,还有另一种维度来组织代码。这个维度赋予了我们隔离特定代码的能力。我们可以将代码视为运行时组件 ,这些组件是自维持 (self-sustained) 的 ,可以打开或关闭。这意味着我们可以不费力地找到同一个组件的所有代码。如果一个组件需要 10 个源文件——JavaScript、HTML 和 CSS 代码——理想情况下这些代码都应该在同一文件夹下。

当然 ,所有组件共用的通用代码是个例外。这部分的代码应该尽可能地放在距工程根目录较近的地方 ,这样有助于了解组件之间的依赖关系。如果组件依赖的代码分散在各处 ,依赖图 (Dependency Graph) 会变得很难扩展。

小结

这一章介绍了组件组合的概念。组件是可扩展 JavaScript 应用程序的基础。常见组件包括模块、模型/数据集合、控制器/视图和模板等。虽然这些模式能帮助我们达到某种程度上的一致性，但仅仅这些是无法让代码在众多因素的影响下良好工作的。这也是为什么我们需要扩展这些组件，来提供自己的通用实现。这样应用才可以进一步扩展以实现其特定功能。

根据应用程序遇到的可扩展性影响因素的不同，我们需要采取不同的方法，在组件中实现通用功能。一种方法是继续扩展组件的层级结构，封装内部实现，与外界完全隔离。另一种方法是在创建组件时，将逻辑和属性注入组件。这样做的代价是使用组件的代码会变得更加复杂。

最后，讨论了如何组织源码。合理的文件组织可以很好地反映组件的设计逻辑，这可以帮助我们扩展开发效能、隔离组件之间的代码。下一章，将着重介绍组件之间的通信。编写可以良好运行的独立组件是一回事，实现可扩展的组件间通信则是另一回事。

4

组件的通信与职责

上一章着重介绍了组件是什么、由什么组成、为什么这样组成。本章的重点在于 JavaScript 组件之间起到粘合剂功能的部分，也就是如何组成。如果组件是为了某种特定的目的而设计的，就需要和其他组件进行通信以实现更多功能。比如，路由器组件不太可能更新 DOM 或直接调用 API，但是有专门的组件做这些事情，所以其他组件可以让这些专门的组件执行这些功能。这就需要通过通信来完成。

在本章的开头部分先介绍了前端开发中比较流行的一些组件通信模型。我们不太可能开发自己的组件间通信框架，因为已经有很多强健的库可供选择。我们更关心的是，从可扩展的角度来讲，通信模型是如何影响应用的扩展性的，以及我们所能做什么。

给定组件的职责会影响它与其他组件间的通信，同时还会影响它无法控制的服务之间的通信。比如后端 API 和 DOM API。一旦我们开始实现应用程序的组件，应用中的组件就会逐渐分层。如果明确这些分层，将有助于组件间通信的可视化。这样我们就可以预见组件通信在可扩展性方面的各种问题。

通信模型

有多种通信模型可以用来实现组件间通信，最简单的就是方法调用或者说函数调用。这种模型是最直接、最易实现的。但是，它又使调用组件和被调用组件之间耦合得很深，所以这种方法很难扩展到多个组件。

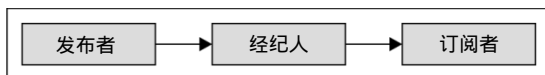
我们需要在组件之间增加一个间接层，用来协调组件之间的通信。它可以将组件间通信变得可扩展。因为组件间不再直接与彼此通信，而是依赖通信机制完成消息的投递。这

类通信机制对应的两种流行的模型是消息传递（message passing）和事件触发（event triggering）。下面来比较一下这两种模型。

消息传递模型

消息传递通信模型在 JavaScript 应用中非常普遍。比如，在本地机器上，消息可以从一个进程传递到另一个进程，可以从一台主机传递到另一台主机，也可以在一个进程内传递。虽然消息传递有些抽象，但它仍然是比较低级的概念，还有进一步解读的空间。而在组件之间的通信机制提供了更高级的抽象。

比如，**发布—订阅（Publish-Subscribe）**模型就是一种具体的消息传递模型。一个组件订阅某个主题的消息，其他组件发布这个主题的消息，完成消息传递的则称为“经纪人”。这个设计的关键点在于，组件之间是互不知晓的。这种设计增进了组件间的松耦合，有助于在组件较多时增强可扩展性。



图注：发布—订阅模型。“经纪人”将发布的消息传递给订阅者。

另一个消息传递的类型是命令—响应模型。一个组件对另一个组件发起命令并得到一个响应。这个场景中的组件耦合得相对紧一些，因为调用者指定了完成任务的组件。

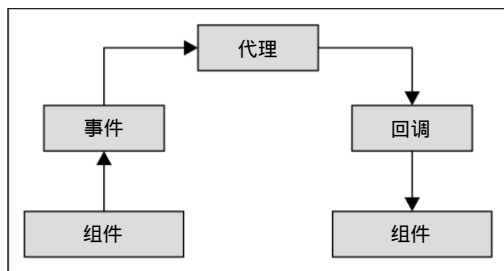
但是，这仍然比直接的方法调用更好一些，因为我们仍然可以轻松地替换调用者和接受者。

事件模型

我们经常听到用户界面代码是事件驱动的。一些事件发生，会引起 UI 渲染某个部分，或者用户发起某些动作，触发一些事件，使得我们的代码必须做出解读并响应。从通信的角度讲，UI 只是一组声明的可视化元素，被触发事件和响应事件的回调函数。

这就是为什么发布—订阅模型适合 UI 开发。我们开发的大多数组件会触发一种或多种类型的事件，与此同时其他组件会订阅这些事件，并且在事件触发时运行代码做出响应。从高层次来看，这就是大多数我们的组件将如何与其他组件通信——通过事件机制通信，即发布—订阅模型。

之所以称为事件、触发，而不是消息、发布—订阅，是因为前端程序员对前者更加熟悉。因为 DOM 的事件系统采用这些术语。它们是与 Ajax 调用和 Promise 对象有相关的异步事件，并且应用程序使用的框架也有一套自己的事件系统。



图注：某个组件触发事件，其他组件监听这类事件并执行回调函数。这个过程由“事件经纪人”（event-broker）进行协调。

分离的事件系统一定会通过应用程序中的组件触发所有的事件，这使得我们很难弄清楚某个动作触发后究竟发生了什么。这的确也是一个与可扩展性相关的问题。本书之后的章节中将会就这个问题给出解决方案，以使组件通信变得可扩展。

通信数据结构

事件的数据不是晦涩、难以理解的——它有具体的含义。回调函数会根据具体含义进行决策。有时这个数据可以被回调函数忽略，但是我们不希望过早地决定某些后续增加的回调函数是否不需要这些数据。这一点正好可以有助于通信机制的可扩展，换句话说就是在正确的地方提供正确的数据。

数据不仅应该准备就绪等待传递给回调函数，还应该要有可预见的数据结构。接下来会讲解如何建立事件命名约定，以及传递给事件处理函数的数据的约定。通过确保事件数据在需要的地方出现，并且不会被错误地解释，可以将组件间通信变得更加透明，这样更有助于实现可扩展。

命名约定

起一个有意义的名字很难，尤其是需要起很多名字的时候。为事件命名也是一样的。一方面，我们希望事件的名称是有意义的。看一眼事件名就知道其含义，能够有助于实现

可扩展性。另一方面，如果视图用多种含义重载（overload）事件名，就失去了快速解读事件名的优势。

起一个短小且有意义的好名字主要是为使用事件的开发人员考虑。基本思想是，当开发人员的代码与事件打交道时，开发人员可以很快想清楚事件是如何流动的。要知道，这只是众多实践中，有助于实现整体可扩展的事件架构中的一个，不过也是很重要的一个。

比如，我们可能有一个基础事件类型以及这一事件更具体的版本。也可以有多个这种基础事件类型，和多个更具体实例，以覆盖更直接的场景。如果有太多的特定事件名和事件类型，就意味着它们无法被重用，同时也意味着还会有更多的事件需要开发者分析理解。

数据格式

除了事件名称之外，事件还应该携带其他数据，我们称之为负载（payload）。数据负载应该总是包含被触发事件的相关数据，很有可能还包含触发该事件的组件数据。关于事件数据最需要留意的是，应当总是携带与订阅这类事件的处理函数的相关信息。通常，回调函数可能会基于事件数据的某个属性状态来决定什么都不做，完全忽略这个事件。

比如，仅仅为了得到那些我们做决定所需的数据或执行某些动作需要的数据，就让事件的每个回调函数都查找对应的组件，这显然是不可扩展的。不过，预见具体需要哪些数据也不是那么容易。如果知道需要哪些数据，就会直接调用相应的函数，而不会争论是否需要事件触发机制。关键是在松耦合的情况下，提供可预测的数据。下面是事件数据的一个简化示例。

```
var eventData = {  
  type: 'app.click',  
  timestamp: new Date(),  
  target: 'button.next'  
};
```

如果想明白哪些数据与给定的事件有关，可以进行以下思考：从事件处理函数中可以得到什么？哪些数据几乎永远不会需要？例如，我们不建议计算事件相关的数据，然后四处传递。如果事件处理函数可以计算，就让它们计算。如果我们在事件处理函数中开始看到重复的代码，那就另当别论了。这时我们应该考虑将重复的部分提取出来作为公共事件数据。

公共数据

事件数据将会始终包含触发该事件的组件信息——可能是组件本身。这始终是一个不错的选择，因为我们只知道事件被触发了，但完全不知道之后事件的回调函数会如何响应该事件。所以只要不引起困惑和误导，最好还是多给回调函数一些数据。

如果同类型的组件总是触发相同的事件，我们就可以据此设计回调函数，并期望回调函数中总是能得到相同的数据。我们可以使事件数据更加通用，并且可以给回调函数提供关于事件本身的数据，例如时间戳、事件的状态，等等。这些数据与组件无关却与事件本身有关。

下面的例子展示了一个基类事件定义了所有公共数据。子类事件通过附加额外属性来扩展基类事件。。

```
// click-event.js
// 所有实例都有type和timestamp属性，另外加上传入的额外属性。
// 最重要的是，任何使用ClientEvent的地方都能确保type和timestamp总是存在的。
export default class ClickEvent {

  constructor(properties) {
    this.type = 'app.click';
    this.timestamp = new Date();
    Object.assign(this, properties);
  }
};

// main.js
import ClickEvent from 'click-event.js';

// 创建一个ClickEvent并传入一些属性。
// 可以重写一些已有的属性。
var clickEvent = new ClickEvent({
  type: 'app.button.click',
  target: 'button.next',
  moduleState: 'enabled'
});
```

```
console.log(clickEvent);
```

再说一次，在数据重用上不要自作聪明，允许必要的重复并接受它。更好的处理办法是创建一个基础事件数据结构，然后将重复的属性移到公用结构中。

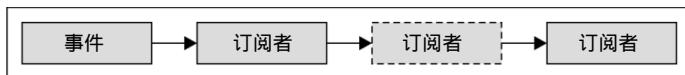
可追踪的组件通信

也许构建大型 JavaScript 应用最大的挑战是始终清楚事件从哪里开始，在哪里结束，换句话说就是追踪事件在组件间的流动。不可追踪的代码会带来可扩展性方面的危机，因为我们无法预测给定事件将会引发哪些反应。

在开发过程中，为了减轻理解事件流的痛苦，有多种策略可以采用，甚至可以修改设计使之更加简化。简单的设计是可扩展的，但我们无法简化那些自己不理解的东西。

订阅事件

发布—订阅消息模型的一个优点是我们可以随时增加新的订阅。这意味着，如果不确定某些代码是如何工作的，可以从多种角度针对问题注册事件回调函数，直到对问题有了更好的了解。这算是一种黑客工具，这种工具可以 hack 我们的软件，有助于实现软件的可扩展。这是因为我们授权开发者自己动手，如果他们对某些事情不清楚，而代码相对容易破解，他们就会自己动手解决。



图注：在某个特定点订阅事件，或以特定的顺序订阅，有可能会改变事件的生命周期。拥有这种能力很重要，但是过度使用也会带来不必要的复杂度。

在某些极端案例中，甚至需要使用这种订阅者的方式来修复生产环境中的问题。比如，某个回调函数能够阻止某个事件的执行，取消接下来处理函数的执行。对于在代码中经常被触发的事件来说，如果能有类似上述这种“入口点”，那应该算是一件好事。

全局事件日志

事件处理回调函数可以输出日志，但有时需要从事件机制本身的角度记录日志。比如，如果我们正在处理一些棘手的代码，并且需要知道相对于其他回调函数我们的回调函数是

什么时候被调用的。这时，事件触发机制应该有一个选项来控制与事件生命周期相关的日志记录。

这意味着对于任何被触发的给定事件，我们都可以看到相关的日志。这部分记录日志的代码独立于事件响应函数。我们称它为“元事件”——关于事件的事件。比如，事件的回调函数在运行之前、运行完毕后、没有更多回调函数时的触发时间点。这些为我们在回调函数中记录日志进行追踪调试的代码提供了急需的上下文信息。

以下代码展示了一个开启日志的事件代理。

```
// events.js
// 一个简单的事件代理。
export default class Events {

  // 构造函数接受一个log()函数。
  // 这个函数在触发事件时会被用到。
  constructor(log) {
    this.log = log;
    this.listeners = {};
  }

  // 调用所有监听name对应事件的函数。
  // 如果在代理被创建的时候有传入了log()函数，将data传给每个函数。
  // 那么在每个回调函数调用之前和之后都调用这个log函数。
  trigger(name, data) {
    if (name in this.listeners) {
      var log = this.log;
      return this.listeners[name].map(function(callback) {
        log && console.log('BEFORE', name);

        var result = callback(Object.assign({
          name: name
        }, data));

        log && console.log('AFTER', name);

        return result;
      });
    }
  }
}
```



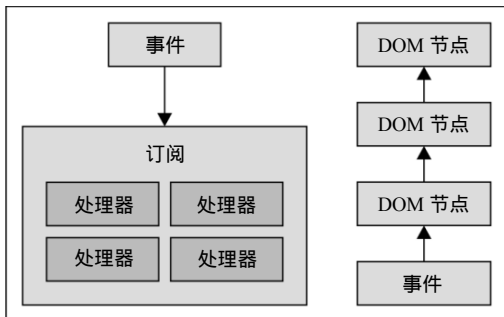
```
        });  
    }  
}  
};  
  
// main.js  
import Events from 'events.js';  
  
// 两个事件回调函数用于输出日志。  
// 第二个是异步的，因为它使用了setTimeout()。  
function callbackFirst(data) {  
    console.log('CALLBACK', data.name);  
}  
  
function callbackLast(data) {  
    setTimeout(function() {  
        console.log('CALLBACK', data.name);  
    }, 500);  
}  
  
var broker = new Events(true);  
  
broker.listen('first', callbackFirst);  
broker.listen('last', callbackLast);  
  
broker.trigger('first');  
broker.trigger('last');  
  
//  
// BEFORE first  
// CALLBACK first  
// AFTER first  
// BEFORE last  
// AFTER last  
// CALLBACK last  
//  
// 请注意如何跟踪事件代理的调用。  
// 同时，请注意CALLBACK last明显是异步。  
// 这是因为它不在BEFORE last和AFTER last之间。
```

事件的生命周期

不同事件触发机制中的事件有不同的生命周期。花时间了解每种机制如何工作、如何控制是值得的。我们先来看 DOM 事件。用户界面的 DOM 节点构成树状结构，任何一个节点都可以触发 DOM 事件。如果触发事件的节点上有事件处理函数，那么该函数会在事件触发时被执行。之后事件会向上传播，在传播过程中的每个节点上查找该事件的处理函数并执行，直至达到文档节点。

事件处理函数实际上可以改变默认的 DOM 事件传播行为。

比如，如果我们不希望后续 DOM 节点上的事件处理函数被执行，可以在 DOM 树的底层节点停止事件传播。



图注：对比各种框架的组件事件系统的事件处理方法和浏览器处理的DOM事件

其他需要关注的主要就是我们正在使用的框架中的事件触发机制。作为一门语言，JavaScript 没有通用的事件触发机制，只有专门的事件触发机制：DOM 事件、Ajax 调用和 Promise 对象。从内部实现来看，它们都使用任务队列。只是从外界来看，它们像是独立的系统。这正是用到框架并提供必要的抽象概念的地方。这些类型的事件派发器非常简单，某个事件的订阅者以先进先出的顺序被执行。某些事件系统支持本节讨论过的高级生命周期选项，比如全局事件日志和事件的提前终止。

通信的开销

直接调用组件上方法的一个优点是开销极小。如果所有组件间通信都是通过事件触发机制来完成，那么不可避免地会产生一些很小的开销。事实上，这种开销几乎是无法察觉

的，反而是其他方面的开销才会引起可扩展性问题。

在这一节我们会了解事件触发频率、回调函数的执行以及回调函数的复杂性。这些方面都有可能降低软件的性能，甚至导致软件不可用。

事件的频率

当软件只有少量的组件时，事件的频率是有限的。但是如果组件增多，甚至某些组件在响应事件时还会触发其他事件，事件的频率会迅速成为问题。这意味着，当用户在做快速操作或者多个 Ajax 响应一起返回时，需要某种手段来防止这些事件阻碍 DOM 渲染。

JavaScript 面临的一个挑战是单线程，但是 Web Worker 超出了本书讨论的范围，因为它引入了全新的架构性问题。假设用户在一秒钟内点击了四次，通常情况下这对我们的事件系统不算什么，但是如果此时正在处理一个耗时的 Ajax 响应，那么用户界面将会无响应。

为了避免界面无响应，可以为事件加设节流阀（throttle）。这意味着限制回调函数执行的频率。没有节流的回调函数执行完会立即执行下一个回调，而节流后的回调函数执行完会延迟一小段事件然后再执行下一个回调函数。这样做的优点是可以给 DOM 更新和 DOM 事件回调函数运行的机会，缺点是事件的生命周期可能受到长时间运行的更新，或其他代码的负面影响。

以下示例展示了事件代理如何将触发的事件节流到某个频率：

```
// events.js
// 事件代理，将事件触发阈值设置为100毫秒。
export default class Events {

  constructor() {
    this.last = null;
    this.threshold = 100;
    this.size = 0;
    this.listeners = {};
  }

  // 只有在阈值范围之内才触发事件。
  trigger(name, data) {
```

```
var now = +new Date();

// 如果超过阈值或者首次触发, 可以调用_trigger(),
// 在该函数中事件回调函数被调用。
if (this.last === null || now - this.last > this.threshold) {
    this._trigger(name, data);
    this.last = now;
// 如果我们刚触发过事件, 就需要设置一个超时器,
// size是一个乘数, 用来在阈值间隔之后调用后续的触发事件。
} else {
    this.size ++;
    setTimeout(() => {
        this._trigger(name, data);
        this.size --;
    }, this.threshold * this.size || 1);
}

// 这是实际触发事件的地方, 称为trigger(),
// 在检查完阈值之后会调用这个方法。
_trigger(name, data) {
    if (name in this.listeners) {
        return this.listeners[name].map(function(callback) {
            return callback(Object.assign({
                name: name
            }, data));
        });
    }
}

};

//main.js
import Events from 'events.js';

function callback(data) {
    console.log('CALLBACK', new Date().getTime());
}
```

```
}  
var broker = new Events(true);  
  
broker.listen('throttled', callback);  
  
var counter = 5;  
  
// 在一个循环中连续触发counter次事件，  
// 会导致事件代理对连续触发做出限制。  
while (counter--) {  
    broker.trigger('throttled');  
}  
//  
// CALLBACK 1427840290681  
// CALLBACK 1427840290786  
// CALLBACK 1427840290886  
// CALLBACK 1427840290987  
// CALLBACK 1427840291086  
//  
// 注意回调中输出的时间戳之间的间隔（约等于阈值）。
```

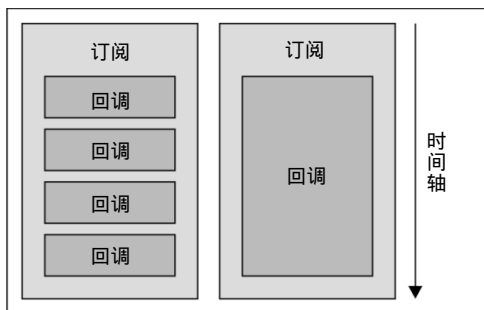
回调函数执行时间

尽管事件触发机制一定程度上可以控制回调函数被执行的时机，但我们并不一定要控制回调函数花费的时间。从事件系统的角度来看，由于 JavaScript 的单线程特性，每一个回调函数可以看成一个小黑盒。那么，如果一个回调函数抛出了异常，我们如何确定是哪个回调函数出现了错误，从而诊断并修复它？

有两种方法可以应对这个问题。如本章前面所提到的，事件触发机制可以开启全局事件日志。通过查看事件日志中的时间戳我们可以推断出回调函数的执行时间。但这并不是计算回调函数执行时间最有效的方法。

另一个方法是在回调函数开始执行时设置一个超时函数。比如，在超过一秒后检查回调函数是否仍在运行，如果仍在运行就抛出异常。这样，系统可以在回调函数执行时间过长时提示系统故障。

但这种方法仍然有一个问题：如果回调函数中有执行时间很长的循环，那么检测它的函数都没机会执行。



图注：执行时间短的和执行时间长的回调函数之间的对比，后者导致DOM更新和DOM事件队列没有时机进行处理。

事件复杂度

当一切都失败时，就轮到我们，大规模 JavaScript 应用程序的设计者，来确保事件处理程序的复杂性保持在一个适当的水平。太高的复杂度意味着潜在的性能瓶颈和用户界面无响应——这是很差的用户体验。如果回调函数粒度过细，或由于事件本身的某些原因，我们仍然会面临性能问题。因为事件触发机制本身也会带来开销——处理更多的回调函数意味着更多的开销。

支持组件间通信的大多数 JavaScript 框架有一个很好的地方就是他们的事件系统十分灵活。这些框架在默认情况下会触发它们认为重要的事件，这些事件带来的性能开销对我们来说基本观察不到，可以忽略。同时框架也允许我们在需要时触发自定义的事件，所以如果开发一段时间后，我们发现事件的粒度过小，就可以适当地做出调整。

一旦对应用程序事件的粒度有了把握，就可以调整回调函数做出反映。甚至可以开始编写较小的回调函数，以使它们可以组成更高阶的函数并提供更多粗粒度的功能。

这里有一个例子，展示了回调函数触发了其他事件，而其他回调函数监听了这些事件。

```
import Events from 'events.js';  
// 这些回调函数触发logic事件。  
// 加入这个间接层可以使逻辑代码  
// 和可能执行其他必要例行事务的事件处理函数解耦。
```

```
function callbackFirst(data) {
  data.broker.trigger('logic', {
    value: 'from first callback'
  });
}

function callbackSecond(data) {
  data.broker.trigger('logic', {
    value: 'from second callback'
  });
}

var broker = new Events();

broker.listen('click', callbackFirst);
broker.listen('click', callbackSecond);

// logic回调小巧紧凑。它不必关心DOM访问或获取网络资源这类事情。
broker.listen('logic', (data) => {
  console.log(data.name, data.value);
});

broker.trigger('click');
//
// logic from first callback
// logic from second callback
```

通信责任区

当思考 JavaScript 组件通信时，从应用程序与外界的边缘思考会很有帮助。我们目前为止都集中关注组件间通信，即同一个 JavaScript 应用中组件之间的通信。这些组件间的通信不会自主发起或结束。可扩展的 JavaScript 代码还需要考虑流入和流出应用程序的事件。

后端 API

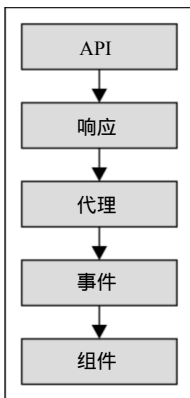
最明显的出发点是后端 API，因为它定义了应用的域，前端真的只是这些 API 的门面

而已。当然，还不止这些，API 数据最终限制了应用可以做什么、不可以做什么。

思考一下哪些组件是负责直接与后端通信，从组件和职责角度来讲还是有好处的。当应用程序需要数据时，正是这些组件发起对后端 API 的请求，获取数据，并在拿到数据后通知我们以便将数据传给其他组件。所以，实际上有一些组件间的通信间接地与后端通信组件有关联。

比如，我们有一个组件集合，为了填充这个集合，需要调用一个方法。集合是否知道它需要填充自己或创建自己？更可能是其他组件发起创建集合，然后让它从 API 获取一些数据。我们知道发起创建的组件并不直接调用 API，同时还知道它在通信中扮演了重要的角色。

当要扩展到很多组件时，就需要好好思考哪些组件负责直接与后端通信了。因为他们需要遵循可预见的模式。



图注：前端的事件代理直接或间接地将API的响应和数据翻译为组件可以订阅的数据。


Web Socket 用于更新状态

Web Socket 连接减轻了长轮询在 Web 应用程序中的需求。Web Socket 越来越多地被使用，得益于浏览器对其强有力的支持，同时后端服务器也有很多支持 Web Socket 连接的库。而最具有挑战性的部分是记录事务，它使我们能够检测到改变并通过发送消息通知相关的会话（Session）。

抛开后端的复杂性不提，Web Socket 确实解决了很多前端软实时（soft real time）更新

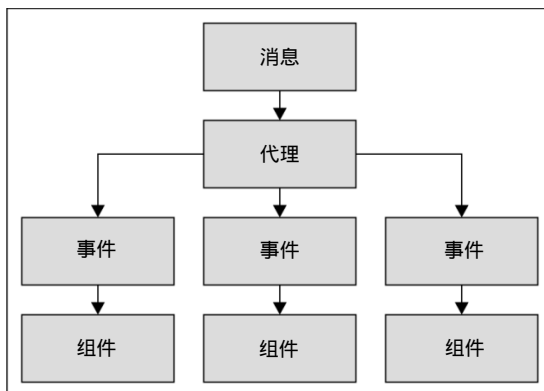
方面的问题。Web Socket 是与后端通信的双向通道,但真正大放异彩的是在接收更新方面,例如接收模型改变的状态。

这样就允许从模型中显示数据的组件在数据更新时重新渲染自己。具有挑战性的部分在于,在任何给定的前端会话中,我们只允许使用一个 Web Socket 连接,这意味着响应这些消息的处理函数需要清楚如何处理。你可能还记得,在前面的章节中,提到过事件数据、有意义的事件名称,以及事件的数据结构。Web Socket 消息事件给出了一个很好的例子,它的消息类型有多种,我们需要弄清楚如何处理这些消息。

 注:由于 Web Socket 连接是有状态的,且可能断开。这意味着,我们将不得不面对额外的挑战:实现代码重新连接断开的 Web Socket。

让一个回调函数处理所有 Web Socket 消息并直接操作 DOM 是一个坏主意。一种方法是每个 Web Socket 消息类型对应一个处理函数。但这会很快变得无法控制,因为有很多回调函数需要运行,而从责任方面讲,众多组件将必须与 Web Socket 连接紧耦合。

如果组件不关心更新的数据是否来自于 Web Socket 连接而只关心数据是否改变的话又会如何?也许我们需要为组件引入一种关于数据变化的新事件类型,然后 Web Socket 处理函数时只需要将消息转换为这些类型的事件。这是一种可扩展的 Web Socket 通信方式。因为我们可以彻底地把 Web Socket 隔离,这样它就不会与我们的系统耦合太紧。



图注:一个事件将某种类型的Web Socket消息翻译为实体特定的事件,只有对这个特定事件感兴趣的组件才需要订阅。

DOM 更新

不必说，对于一个运行在浏览器中的 Web 应用来说，我们的组件势必要和 DOM 打交道。那么哪些组件涉及 DOM、哪些不涉及就很值得我们去思考。视图组件是很常见的，因为它将应用程序的数据以某种可见的形式展现在了浏览器窗口中。

视图这类组件的可扩展性具有挑战性，很大程度上是因为事件在视图中是双向流动的。另一个原因是，当我们不知道新代码应该放在哪儿的时候，通常会把它放在视图中。当视图变得越来越臃肿时，我们会将一部分代码放在控制器或专门放置工具类代码的文件中，亦或是其他什么地方。不过，肯定有某种更好的方法。

先让我们思考一下视图事件通信。首先是流入的事件，流入的事件会通知视图数据发生了变化，应该更新 DOM 了。在收到这些事件后，视图就会更新 DOM。以上这种工作方式很可靠，当视图只监听一个组件的事件时可以很好地工作。但是当我们的应用的功能逐渐增多增强，视图就开始需要“思考”了。不过，视图还是“笨”一点的好。

比如，某个视图刚开始只负责在某个数据发生变化时渲染一个元素，而现在需要做更多的事情。在完成渲染后，它需要计算一些数据中没有直接给出的数值，然后更新另一个元素。这个过程中，视图变得“聪明”起来，但是会渐渐地失控，直到不可扩展。

从通信的角度讲，我们希望视图是一个简单数据到 DOM 的一对一的绑定。如果能保持这样，就可以轻松地预知数据发生变化时会发生什么事情。因为我们清楚地知道哪个视图监听这个数据以及这些数据与哪个 DOM 结构绑定。

现在思考一下另一个绑定方向——监听 DOM 的变化。所存在的挑战同样是我们倾向于让视图变得“聪明”起来。当输入数据出现问题时，我们会重载视图的事件处理函数。而这个函数会响应 DOM 事件并履行非它应有的职责。视图“笨”一点，会工作得更好。它们应该将 DOM 事件翻译为应用特定的事件，这样其他组件就可以监听应用特定的事件，如同上面提到的将 Web Socket 事件翻译为应用特定事件一样。我们“聪明”的组件发起了某些业务过程，但它们其实并不关心发起是不是源自于 DOM。这样，通过创建少数几个通用组件我们就可以做到可扩展，并不需要做很多的事情。

松耦合的通信

如果组件间通信是松耦合的，当可扩展性影响因素出现时我们便可以更容易地适应。首先，事件驱动是关于组件间通信的一个很好的设计，它使我们可以移动替换组件。我们可以将有问题、低性能的组件移除，用其他相同功能的组件替换。如果不能替换组件，意味着我们必须修复原有组件的问题。而这对于交付软件是一个很大的风险，从开发角度来说也是一个可扩展性的瓶颈。

松散耦合的组件间通信的另一个作用是，当出现问题的时候，我们可以将有问题的组件隔离。当用户在组件错误的状态下试图做其他事情时会导致进一步的问题，而我们可以防止异常组件导致的其他组件处于错误的状态。对这样的问题进行隔离，可以帮助我们修复故障组件。

替换组件

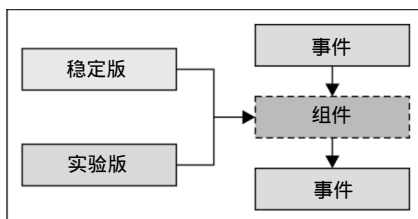
基于给定组件触发和响应的事件，我们可以轻松地用不同版本的组件替换它。不过我们还是需要了解一下组件内部是如何工作的，毕竟我们不希望完全改变它。但这只是最容易的部分，实现组件的过程中比较困难的部分是如何将它们组织起来。实现可扩展的组件意味着尽可能地将组件的组织简单易行、逻辑一致（approachable and coherent）。

但是为什么组件的可替换性这么重要呢？我们会认为由少部分连接在一起的组件组成的稳定代码不会经常改变。从这个角度来看，组件的可替代性的确没有那么重要。那还有什么必要考虑可替代性呢？这种心态唯一的问题是，如果我们认真对待代码的可扩展性，我们就不应该只对某一部分组件应用设计原则而忽略另一些。

事实上，不愿意重构稳定的代码不一定是件好事。比如，它会阻碍我们那些需要进行重构的新的想法，而组件的可替换性给我们带来的好处就是可以实施这些新的想法。因为如果将稳定的组件替换为新组件实现起来很容易的话，我们更倾向于将新的设计理念放到产品中去。

替换组件并不只是在设计时。我们可以引入可变性，准备一些组件以备不时之需，究竟选择哪个组件可以在运行时决定。这种灵活性意味着我们可以轻松地扩展功能以应对可

扩展性的影响因素。比如适时增加新的用户角色。一部分角色对应一个组件，另一部分对应另外一个相互兼容的组件，亦或是不对应任何组件。能这样做的关键就是要支持这种灵活性。



图注：只要组件遵循相同的通信协议、事件触发和处理，开发实验性的功能会变得比较容易。

应对意外事件

松耦合的组件可以帮助我们增强应对有缺陷组件的能力。这主要是因为可以隔离引起问题的单个组件，然后快速定位问题并修复。另外，如果有问题的组件运行在生产环境中，可以在降低其负面影响的同时交付响应的修复方案。

缺陷总会发生的，我们需要接受这一点并从设计上避免它。发现缺陷时要从中吸取教训，这样以后才能不重蹈覆辙。由于我们的工期很紧，经常需要提前发布，所以 bug 总会悄悄沿着产品的缺陷爬进产品中。还会有没有测试到的边界情况或者被单元测试忽略的独特的编程错误。不管怎样，我们还是需要设计组件故障模型以应对这些状况的发生。

隔离有缺陷的组件的一种方法是将所有的事件回调放到 try/catch 中。如果发生任何异常，我们的回调函数就会通知事件系统组件处于错误状态，这就给其他组件提供了恢复状态的机会。如果事件回调链中有一个出错的组件，我们可以安全地提示用户某些操作无法完成。而得益于出错组件的通知，其他组件仍处于可用状态，所以用户还可以继续使用其他功能。

下面代码示例展示了可以捕获回调错误的事件代理：

```
// events.js
export default class Events {

  constructor() {
    this.listeners = {};
  }
}
```

```
    }

    // 触发事件。
    trigger(name, data) {
        if (!(name in this.listeners)) {
            return;
        }

        // 我们需要这个变量来跟踪错误状态。
        var error = false,
            mapped;

        mapped = this.listeners[name].map((callback) => {
            // 如果前一个回调函数出错，我们就不再调用后续的回调函数。
            // 后续函数对应的输出值都是undefined。
            if (error) {
                return;
            }

            var result;

            // 捕获回调函数抛出的异常。
            // result对象的error属性设置为true。
            try {
                result = callback(Object.assign({
                    name: name,
                    broker: this
                }, data));
            } catch (err) {
                result = { error: true };
            }

            // 回调函数可以抛出异常，
            // 也可以返回一个error属性为true的对象。
            // 结果都是一样的——我们停止处理后继回调函数。
            if (result && result.error) {
                error = true;
            }
        });
    }
}
```

```
    }

    return result;
  });

  // 当出现错误时触发相应的事件，以便让其他组件知道。
  if (error) {
    this.trigger(`${name}:error`);
  }
}

}

// main.js
import Events from 'events.js';

// 回调函数在失败时返回error属性为true的对象。
function callbackError(data) {
  console.log('callback:', 'going to return an error');
  return { error: true };
}

// 回调函数在失败时抛出异常。
function callbackException(data) {
  console.log('callback:', 'going to raise an exception');
  throw Error;
}

var broker = new Events();

// 监听成功事件和对应的出错事件。
broker.listen('shouldFail', callbackError);
broker.listen('shouldFail:error', () => {
  console.error('error returned from callback');
});

broker.listen('shouldThrow', callbackException);
broker.listen('shouldThrow:error', () => {
```

```
        console.error('exception thrown from callback');
    });

    broker.trigger('shouldFail');
    broker.trigger('shouldThrow');
    // callback: going to return an error
    // error returned from callback
    // callback: going to raise an exception
    // exception thrown from callback
```

组件分层

在任何足够大的 JavaScript 应用中，当组件数量超过某个阈值时，组件间的通信就会产生可扩展性问题。主要的瓶颈就是我们创建的复杂性以及我们无法理解它。要应对这种复杂性，可以引入分层。所谓分层就是抽象的分类术语，有助于可视化地理解运行时究竟发生了什么。

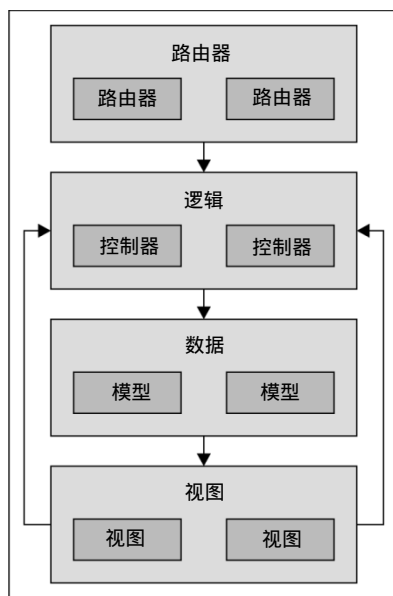
事件流向

首先，通过分层设计，可以以事件流向的形式展示代码中组件间通信的复杂性。比如说，我们的应用有三层：顶层涉及路由和其他用户界面的进入点，中间层遍布数据和业务逻辑，底层是视图。如下页图所示。虽然这些层中包含多少组件也是其中一个因素，但并不重要，重要的是从分层的角度来看穿越到其他层的箭头类型。

比如，以上面三层架构为例，我们可能注意到最直接的层间连接就是路由器和数据与业务逻辑层之间的连接。这是因为这些事件大多自上而下流动，从路由器直接流向下层。从那里开始，模型和控制器组件之间很可能有一些通信。但最终，事件还会向下流动。

在数据/业务逻辑层和视图层之间，代表通信的箭头变为双向且有些令人迷惑。这是因为这两层代码中的事件流动方向也是双向且令人迷惑的。这不是可扩展的，因为我们不能很容易地掌握我们触发的事件所带来的影响。分层设计很有用的一点就是可以帮助我们找到一种可以移除双向事件流动的方法。这意味着引入一个间接层（indirection），它将负责源和目标之间的事件流转。

如果我们做得巧妙，事件将会流转得更加清晰而不会使分层图更加凌乱，而其对性能的影响基本上可以忽略。



图注：组件间明确可辨的事件流动方向对可扩展性有巨大影响。

开发者的职责

分层只是一种辅助手段而不是一个正式的架构规范，这意味着我们可以将它们用于任何可以发挥作用的地方。不同群体的人可以有他们自己的分层设计来满足他们在理解复杂度上的需求，但如果开发团队作为一个整体遵循相同的分层设计会更有用。分层最好保持简单，不要超过四到五层，否则就违背了使用它们的目的。

开发者可以使用分层作为自组织的手段。他们了解架构，接下来有工作需要完成。比如说，我们有两个开发者开发同一个功能，就可以使用组件分层架构来计划他们的开发并避免互相干扰。当有一个类似分层设计的概览图作为参考时，两个人的工作最终会无缝地合成一个整体。

构建代码思维导图

即使没有分层图，了解当前组件代码属于哪一层也可以帮助我们理解组件的功能，这

一点对系统其他部分也同样适用。知道我们正在编写的组件属于哪一层，可以让我们在潜意识中知道哪些组件与当前组件相邻，哪些事件什么时候会跨越层的边界。

在分层的约束下，相对于已有组件以及它们在分层间的通信模式，新组件的设计问题就会格外明显。分层的存在，以及分层经常被开发者用来作为非正式的辅助工具的事实，可能足以在早期就将设计问题暴露出来。或许并没有什么问题，但分层足够用于促进对设计的讨论。团队中的有些人可能从中学到些什么，有些人可能会确信自己的设计是可靠的。

小结

组件是构建 JavaScript 应用的基础，将组件紧密连接起来的粘合剂就是应用使用的通信模型。从底层讲，组件间通信就是一个组件通过某种代理机制将消息传递给另一个组件，而这种机制通常简单地抽象为事件系统。

我们知道其实从一个组件传递到下一个组件的是事件数据，这些数据必须是一致可预见并且有意义的。我们还了解了可追踪的事件：是否可以通过事件触发机制在事件被触发时开启全局事件日志？

JavaScript 代码的边界是通信的端点（Endpoints）。我们了解了与 DOM、Ajax 调用、本地存储等外部系统通信的各种组件，并且需要将“聪明”的组件与系统隔离。

可替换性和分层对可扩展性是至关重要的。可替换组件可以降低风险快速开发新的代码。分层帮助我们多方面保持大局观和可控性，也可以将设计上错误的假设提前暴露出来。

下一章该考虑应用可寻址能力的可扩展问题了，而且还会看一看前两章讨论的内容在下一章还有没有意义。

5

寻址和导航

Web 应用依赖于**可寻址 (addressable)** 的资源，URI 是互联网中很关键的技术。因为我们可以把一些与资源相关的信息编码到 URI 字符串中，所以 URI 消除了一系列复杂性问题，这就是可寻址的**道 (policy)**。而寻址的**术 (mechanism)** 则依赖于浏览器，或者说我们的 JavaScript 代码——来查找并且展示所请求的资源。

在过去，往往是后端来处理 URI。当用户输入一个 URI 时，浏览器的职责仅仅是将请求发送到后端，然后展示响应的资源。然而，在大规模的 JavaScript 应用中，处理 URI 的工作最大限度地转向了前端。我们利用工具就能在浏览器端实现复杂的路由，从而也能减少对后端技术的依赖。

不过，一旦我们的软件功能复杂起来，前端路由带来收益的同时也需要耗费一定的代价。本章会深入研究随着应用结构的扩张和成熟，可能会面临的路由场景。对于框架里的路由组件，大多数基础的实现细节并不重要。我们更关心的是，在扩展性的影响下，路由组件能否更好地适应这些影响因素。

实现路由的方法

在 JavaScript 里实现路由有两种方法。第一种是利用基于 hash 的 URI，这些 URI 以 # 字符开头，这种方法更为常见一些。另一种不太常用的方法是利用浏览器的 history API 生成更多传统的 URI，也就是 Web 应用上常见的 URI。这项技术更复杂，而且最近才得到足够的浏览器支持来实现该方法。

Hash URI

URI 里的 hash 部分原先是为了指向文档里的特定位置而设计的。因此浏览器会查找# 字符左边的所有内容，将这部分信息发送到后端，请求一些页面内容。只有当页面到达浏览器，并且被渲染之后，# 字符右边的内容才有意义。这时候浏览器会在页面内根据 URI 的 hash 部分定位本地页面的相关位置。

现在，URI 里的 hash 部分有了不同的用处。虽然在 URI 改变时，我们依旧用它来避免给后端发送无关数据。但不同的是，如今我们处理的是 Web 应用和功能，而不是网站和静态内容。既然地址改变时应用的大部分已经加载到浏览器，就没必要给后端发送不必要的请求，而只需要加载新的 URI 所需的额外数据，通常用程序内部的一个 API 请求就能解决。

当提及在 JavaScript 中使用 hash 实现路由以及改变 URI 时，通常只有 hash 部分会被改变。这意味着相关的浏览器事件将会被触发，从而让代码知道 URI 已经改变。但是它并不会自动向后端请求新的页面内容，这也是关键所在。实际上我们使用 hash 方式的原因之一就是，这样的前端路由会给性能和效率会带来很大的提升。

这个方法不仅很有效，而且容易实现。想要实现一个能够根据 hash 变化抓取相关数据，然后更新页面的事件的监听器并不难。另外，浏览器会自动帮我们处理 history 的改变。

传统 URI

对于一些用户和开发者来说，hash 方法像是一种 hack 的方式，SEO 也会碰到问题。因此，他们更倾向于传统的以斜线分割的资源名格式。得益于现代浏览器对 history API 的增强，现在这种方法在所有现代浏览器里都能够实现。本质上，路由机制可以监听那些被推进 history 栈中的状态，当状态改变时，它会阻止请求发向后端，直接在本地响应状态变化。

显然这种方法需要写更多代码，还要考虑更多的极端情况。比如，用户可以向应用里输入任何有效的 URI，所以后端需要支持所有前端路由所支持的 URI。针对这种情况，一种处理方式是在服务器端重写规则，将 404 错误重定向到应用首页，从而交给真正的路由程序来处理。

大部分 JavaScript 应用框架中的路由组件都封装了这两种方法，并提供无缝调用它们

的方法。至于为了增强功能或者改善可扩展性问题而采用了哪一种方法并不重要。不过说到可扩展性，我们有必要知道有以上两种方法，而不应只局限在其中一种。

路由是如何工作的

现在我们进一步看看路由的实现。我们希望了解路由的职责，以及 URI 改变时路由的生命周期。简单来说，就是路由拿到新 URI，然后分析是否感兴趣。如果是，就触发对应的路由事件，并且把解析后的 URI 数据作为参数传递进去。

从底层理解路由对于扩展应用很重要，因为应用的 URI 越多，就越有可能面临可扩展性问题。只有深入了解路由的生命周期，才能适当平衡影响可扩展性的因素。

路由的职责

可以简单地把路由理解成一张映射表，它把一些路径、字符串、以及正则表达式的定义映射到回调函数。但是这个映射过程一定要足够快、容易预测，还要足够稳定。达到这个标准很难，特别是当应用中的 URI 越来越多时。以下是一些通用的路由职责：

- 存储从路由模式到对应事件名的映射。
- 监听 URI change 事件——*hash change* 或 *pop state*。
- 查找路由模式，将新的 URI 与每个映射模式相比较。
- 匹配成功时，根据匹配的模式解析新的 URI。
- 触发路由事件，传递解析数据。



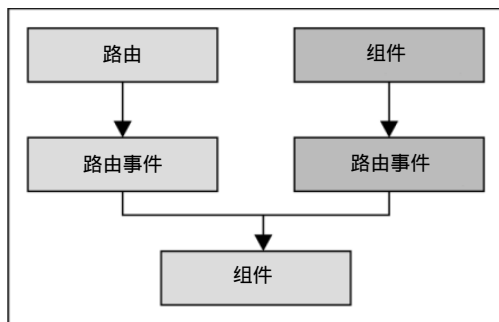
路由查找需要线性查找路由映射表，最终得到匹配的结果。假如定义了很多路由，就会严重影响性能。如果用对象数组来表示路由映射，路由性能很可能会不稳定。例如，某一个路由在数组的末，匹配到最后才成功，自然很耗时。而如果这个路由在数组的开头，耗时就少得多。

当匹配频繁访问的 URI 时，为了避免出现性能问题，可以扩展路由组件，让它根据优先级属性对路由映射数组进行排序，或者用 *trie* 结构来避免线性查找。当然，只有在路由规则过多、严重影响路由性能时，才会考虑这些优化方式。

当 URI 改变时路由组件还要处理很多情况，这也是为什么有必要理解一个路由的生命周期——从地址栏里的 URI 变成完成所有的事件处理函数。从性能角度考虑，路由数过多会影响我们的应用。从构造角度考虑，要跟踪创建了哪些组件以及影响了哪些路由极具挑战性。只有当我们了解了一个路由的生命周期时，处理起前面这些问题才会容易些。

路由事件

路由组件一旦找到了 URI 匹配的路由模式，根据匹配模式解析了 URI，就会触发路由事件，路由事件会存放在映射表中。URI 里面可能包含了变量，这些变量被解析后会作为参数传给路由事件处理函数。



图注：路由事件提供了一个抽象层，即使不是路由组件也能触发路由事件。

大部分框架都配备了路由组件，路由改变时不会触发路由事件，而是直接调用函数。这样更简单，在小型应用中更直接有效。但是，事件触发机制所具备的间接特性，更利于别的组件和路由组件解耦。

间接方式的好处在于，即使不同组件之间无法感知到彼此，也能监听相同的路由事件。应用规模扩大后，线上的路由需要新增功能，此时不用在原有代码上添加新功能，只需增加新的事件处理函数即可。这也是抽象的好处，监听路由事件的组件不用管是谁触发了事件。其他组件也可以去触发路由事件，而不必依赖路由组件。

URI 的结构和模式

在大型 JavaScript 应用中，设计路由组件需要考虑很多因素，而设计 URI 也是如此。

它们由什么构成？它们在教育中能保持一致性吗？不好的 URI 有哪些特点？一旦考虑的方向不对，扩展寻址能力就会变得相当困难。

编码信息

URI 的好处在于，客户端只需要将它传递到应用中，它包含的信息就足以完成有效的操作。最简单的 URI 表示的是应用里的资源类型或者静态地址，比如 `/users` 或者 `/home`。利用这些信息，路由组件就能触发路由事件，从而触发回调函数。这些回调函数并不需要参数，因为都是静态资源。

但在某些情况下，路由回调函数需要上下文，此时则要将信息编码到 URI 中。最常见的场景是，客户端用唯一标识请求某一资源的特定实例。例如，`user/31729`，这种情况下，路由组件需要找到匹配这一字符串的模式，并且该模式要指定提取 `31729` 变量。路由组件要将该变量传递到回调函数中，回调函数才有足够的上下文信息来完成任务。

如果将太多的信息编码到 URI 中，URI 就会变得冗长复杂。比如，为了显示一个表格页面，把查询参数都编码到 URI 中。在路由模式中包含全部的可能性不仅很难，而且容易出错。因为变量的组合太多，难免会出现变化和无法预期的边界情况，甚至有些变量也可能是可选的。

为了避免这种复杂的情况，最好不要在 URI 模式中解析这些变量。可以让回调函数进一步解析 URI，得出上下文。这样做有助于维持路由规则干净整洁，还能让那些复杂奇怪的处理函数与其他逻辑解耦。

对于通用查询，尤其是以链接形式出现时，给用户提供的 URI 要足够简单。比如，最近文章可能会链接到 `/posts/recent`。像文章顺序和资源编号这类信息，如果不被编码到 URI 中，URI 的处理函数就需要用别的办法拿到。但是如果在 URI 中不包含这些信息，对用户体验和代码的可扩展性都是好事。

设计 URI

创建 URI 时，资源名就是一个不错的选择。比如，一个展示事件的页面，它对应的 URI 可以用 `events` 开头。我们要避免不直观的资源名。在没有上下文的前提下，还要避

免特定术语的缩写。

同时，还要避免相反的情况：URI 太啰唆。添加过多的含义容易引起混淆，不论单个单词的长度还是 URI 组件的个数，都不要太冗长。结构清晰、可读性强的 URI 通常由更小的部分构成，比如，先是事物类型，紧接着是事物标识符。另外，不要把类别等不必要的信息编码到 URI 中，因为这些信息能在页面展示。

对所有 URI 都要保证一致的标准。如果限制了一个资源名的字符个数，那所有资源名的字符数都要有同样的限制。如果用斜线分隔了 URI 的不同部分，那所有 URI 都应如此。这样设计的目的是，即使有很多 URI，用户也很容易理解，因为他们不用打开页面就能够猜出一个 URI 对应的是什么。

在保持一致性的前提下，可能还需要一些特定类型的 URI。比如，当某个页面让一个资源处于不同的状态，或者该页面需要用户输入时，则需要在这个动作后面加上不同的符号。例如，有个任务页面的 URI 可能是 `/tasks/131:edit`，为了保持一致性而将 URI 各部分用斜线分割，就可能是 `/tasks/131/edit`。虽然这会让该 URI 看起来同 `tasks/131` 是两个不同的资源，实际上它们只是同一个资源的不同状态而已。

以下示例中运用了一些正则表达式来验证路由：

```
// 用通配符匹配URI里的参数.....
console.log('second', (/^user\/(.*)/i).exec('user/123'));
// [ 'user/123', '123' ]

// 匹配相同的URI，只是加了限制.....
console.log('third', (/^user\/(\d+)/i).exec('user/123'));
// [ 'user/123', '123' ]

// 匹配失败，查找的是字母结果我们得到的是数字.....
console.log('fourth', (/^user\/([a-z])/i).test('user/123'));
// false

// 匹配成功，我们得到一串字母.....
console.log('fifth', (/^user\/([a-z]+)/i).exec('user/abc'));
// [ 'user/abc', 'abc' ]
```

将资源映射到 URI

在实际应用中，URI 最常见的形式就是应用里的链接。前面已经描述了路由组件处理 URI 的过程，下面介绍一下生成链接，以及插入 DOM 的过程。

生成链接有两种方式。第一种是手动创建，这种方式需要借助模板引擎和公用函数。第二种是自动生成，可以增强对多个 URI 的管理能力。

手动创建 URI

如果组件在 DOM 里渲染内容，就会潜在地创建 URI 字符串，并将其添加到链接元素上。当只有几个页面和 URI 时，这种方式会很简单。然而当规模扩大时，页面数量和 URI 数量就会一起上涨。

在实现视图的时候，可以参考路由模式的映射配置，映射配置指定了 URI 的形式以及对应的行为。借助大部分框架自带的模板引擎，利用模板特征，就能根据需要动态渲染链接。如果缺少成熟的模板机制，就需要借助单独的公用组件，来生成 URI 字符串。

URI 和模板数量较多时，这种方式很难实施。尽管借助一些模板语法的帮助，能减轻创建链接的麻烦。但是这仍然很耗时，并且容易出错。除此之外，在模板里构建链接的静态特性，会导致重复的模板内容，比如，硬编码的资源类型。

自动生成资源 URI

大多数链接资源都是 API 提供的，在代码中通过模型（model）或者集合（collection）来表示。在这种情况下，与其借助模板工具来构建 URI，倒不如在每个模型或集合中用同一个函数来构建 URI。这种方式下，模板的内容重复的问题将不复存在，只需关注抽象的 `uri()` 函数即可。

这种方式在简化模板的同时，也带来了一个问题：如何同步模型和路由组件。比如，模型生成的 URI 字符串需要匹配路由模式。所以要么遵循严格的规范，保证模型里 URI 与路由组件同步生成，要么模型需要基于路由模式来定义它生成 URI 字符串的方式。

如果在构建 URI 模式时，路由使用简化的正则表达式语法，那么，实现模型里的 `uri()`

函数与路由定义自动同步是可行的。但是，模型必须了解路由组件，这就会导致依赖问题，因为有时候只需要模型而不需要路由。假设能够将 URI 的模式存储到模型里，随后将这个模式注册到路由组件中，那么，模型就可以利用该模式生成 URI 字符串，而且，仍然只需要在模型里修改 URI 模式。当别的组件在路由组件中注册这个模式时，就不会跟模型耦合了。

以下示例展示了如何将 URI 字符串封装入模型，而不是其他组件中：

```
// router.js
import events from 'events.js';

// 路由组件还是事件触发器。
export default class Router {

  constructor() {
    this.routes = [];
  }

  // 增加一个特定“模式”，
  // 并且在被激活的时候，触发事件"name"。
  add(pattern, name) {
    this.routes.push({
      pattern: new RegExp('^' +
        pattern.replace(/:\w+/g, '(.*)'),
      name: name
    });
  }

  // 添加任意的已配置路由，开始监听导航事件。
  start() {
    var onHashChange = () => {
      for (let route of this.routes) {
        let result = route.pattern.exec(
          location.hash.substr(1));
        if (result) {
          events.trigger('route:' + route.name, {
            values: result.splice(1)
          });
        }
      }
    };
  }
}
```

```
                break;
            }
        }
    };

    window.addEventListener('hashchange', onHashChange);
    onHashChange();
}
}

// model.js
export default class Model {

    constructor(pattern, id) {
        this.pattern = pattern;
        this.id = id;
    }

    // 为模型生成URI字符串。
    // 将模式作为构造函数的参数传入。
    // 那么，生成URI字符串的代码（例如，操作DOM的代码）只需从模型中获取URI。
    get uri() {
        return '#' + this.pattern.replace(/:\w+/, this.id);
    }

}

// user.js
import Model from 'model.js';

export default class User extends Model {

    // 实例的URI模式被封装在这个静态方法里。
    static pattern() {
        return 'user/:id';
    }

}
```

```
    constructor(id) {
        super(User.pattern(), id);
    }

}

// group.js
import Model from 'model.js';

export default class Group extends Model {

    // pattern() 方法是静态方法，
    // 因为所有Group模型的实例都会用同样的路由模式。
    static pattern() {
        return 'group/:id';
    }

    constructor(id) {
        super(Group.pattern(), id);
    }

}

// main.js
import Router from 'router.js';
import events from 'events.js';
import User from 'user.js';
import Group from 'group.js';

var router = new Router()

// 用pattern()静态方法添加路由。
// 没有必要在这里硬编码任何路由。
router.add(User.pattern(), 'user');
router.add(Group.pattern(), 'group');

// 设置响应路由的函数
```

```
events.listen('route:user', (data) => {
  console.log(`User ${data.values[0]} activated`);
});

events.listen('route:group', (data) => {
  console.log(`Group ${data.values[0]} activated`);
});

// 构造新的模型，并且在DOM中使用它们的uri属性。
// 仍然不需要硬编码路由模型相关的东西。
var user = new User(1);
document.querySelector('.user').href = user.uri;

var group = new Group(1);
document.querySelector('.group').href = group.uri;

router.start();
```

触发路由

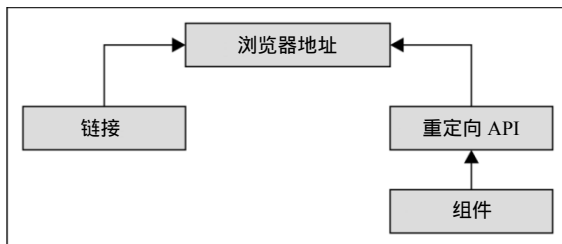
最常见的路由触发方式就是用户单击应用里的链接。在前面讨论过，我们需要让链接生成机制控制很多页面和 URI，而触发动作本身也会影响可扩展性。举个例子，在小型应用中链接显然更少，也意味着用户的点击事件越少。相反，越多的导航意味着事件触发的频率越高。

还要考虑一些不太常见的导航行为。包括在后端任务完成时将用户重定向，或者是直接将用户从 A 导航到 B。

用户行为

当用户单击了应用中的某个链接，浏览器会拿到这个链接并改变 URI。这一过程还可能包含应用的入口，因为可能是从别的网站或者书签栏打开的。因此，链接和 URI 非常灵活，它们可以来自任何地方，并且指向任何资源。我们要尽可能地使用链接，因为这说明了应用的连接性很好，而且处理 URI 变化是路由组件非常擅长的事情。

但是其他方式也能触发 URI 变化，导致路由 workflow 改变。比如，假设用户正在一个 `create` 事件表单上，提交表单后，后端成功返回了响应，这时候不应该让用户停留在 `create` 事件页面，而是要将其导航到事件列表页，这样他才能看见刚才添加的事件。在这种情况下，手动改变 URI 就很有必要并且很好实现。



图注：应用改变地址栏的不同方式。

重定向用户

API 成功响应后重定向到一个新的路由，这就是一个手动触发路由的典型示例。此外，为了响应用户行为，或者为了确保用户看到正确的信息，都需要将用户重定向到新的页面。

并非所有的重大操作都是在后端完成的，比如执行操作的本地 JavaScript 组件，执行完后，需要将用户带到应用里的其他页面。

重点是，效果比原因更重要，不必关心是什么导致了 URI 发生改变，真正需要关心的是，如何才能以不可预见的方式使用路由组件。当应用扩展时，有一些场景需要简单快捷的路由方案，而对导航的整体把握有助于控制应用实现可扩展的方式。

路由配置

路由组件的映射表往往比路由实现本身大得多。随着应用规模扩大，产生了更多的路由模式，映射表就会变大。为了满足扩展的需求，这种结果往往是不可避免的。应对的方法就是不要让大量的路由声明把自己压垮，因为这种事情可能会以各种形式发生。

有多种方式来配置路由，这些路由会被特定的路由实例响应。基于项目使用的框架，在配置方式上，路由组件比其他组件更灵活，一般来讲，有静态路由和事件注册两种方式。此外，在某些场景下，路由组件需要能够禁用特定路由。

静态路由声明

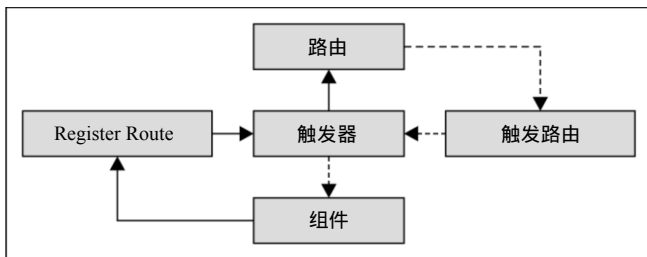
简单的应用通常用静态声明来配置路由。在路由创建时，通常要有一个从路由模式到回调函数的映射表。这种方式的好处是，所有路由模式相对集中，一眼就能看到路由配置中有什么，而且不必去查找特定的路由。然而，在路由过多时就行不通了，还是得搜索特定路由。同时，如果没有分离关注点，当开发人员想彼此独立开发时就比较困难。

注册事件

要定义很多路由时，应该关注封装的路由：哪些组件需要这些路由，它们如何告诉路由组件。大多数路由组件只要调用一个方法，就能添加一个新的路由配置。因此只需要引用这个路由组件，并且将目标组件的路由添加进去即可。

这种方式无疑向正确的方向迈进了一步。它允许将路由的声明保存在需要路由的组件内部，而不是将整个应用的路由配置拼凑到一个单独的对象中。不过，还需要再提升一下这个方式的可扩展性。

与其让组件直接依赖一个路由实例，倒不如触发一个添加路由的事件，这个事件能被任何监听事件的路由组件接收到。也许应用里有多个路由实例，每个实例都有自己特定的功能，比如记录日志，并且这些实例都能够监听遵循特定标准的路由。重点是，组件不需要关心路由实例，当某个模式匹配了 URI 时，就会有方法触发路由事件。



图注：使用事件将组件与路由隔离。

禁用路由

配置了一个路由后，这个路由在会话期间需要一直处于可用状态吗？路由组件是否要提供方法禁用某个路由呢？这取决于从响应性角度如何看待特定情况。

比如,假设发生了某一事件后,某个路由不该再被访问了(访问它只会抛出一个友好的错误提示),路由回调函数可以检查这个路由是不是可访问的。不过,这样会增加回调函数自身的复杂性,并且这种复杂性会遍布在整个应用的回调函数中,而不是仅仅只包含在一处。

另一种可选方案是使用校验组件,当某些组件进入了可禁用状态时,它能够禁用路由。同时,这个校验组件还能在组件变成可用状态时,重新启用路由。

第三种方式是,在第一次注册路由时添加一个可选的监视函数。一旦路由匹配上,就运行监视函数,假如监视函数通过,该路由就会被正常激活;否则,就会失败。这种方式是最好扩展的,因为被检查的状态与相关的路由紧耦合,所以不必为路由切换可用/禁用状态。考虑一下将监视函数加入到路由的匹配标准中。

以下示例展示了接受监视函数的路由组件。当有监视函数并且返回 `false` 时,路由事件才不会被触发。

```
// router.js
import events from 'events.js';

//路由组件根据路由变化触发事件。
export default class Router {

  constructor() {
    this.routes = [];
  }

  // 新增一个路由,同时有一个可选的监视函数。
  add(pattern, name, guard) {
    this.routes.push({
      pattern: new RegExp('^' +
        pattern.replace(/:\w+/g, '(.*)'),
      name: name,
      guard: guard
    });
  }
}
```

```
start() {
  var onHashChange = () => {
    for (let route of this.routes) {
      let guard = route.guard;
      let result = route.pattern.exec(
        location.hash.substr(1));

      // 如果匹配了路由，并且有监视条件，就用监视条件判断一下，
      // 事件只有在监视函数通过的情况下才会被触发。
      if (result) {
        if (typeof guard === 'function' && guard()) {
          events.trigger('route:' + route.name, {
            values: result.splice(1)
          });
        }
        break;
      }
    }
  };

  window.addEventListener('hashchange', onHashChange);
  onHashChange();
}

// main.js
import Router from 'router.js';
import events from 'events.js';

var router = new Router()

// 这个函数可以作为任何路由的监视条件。
// 它会返回一个随机值，用以演示不同的结果，
// 在实际应用中，它可以是希望传给路由的任何值。
function isAuthorized() {
  return !!Math.round(Math.random());
}
```



```
}

// 第一个路由没有监视条件，
// 所以总会触发一个路由事件。
// 第二个路由只有当给定回调函数返回true的时候，
// 才会触发路由事件。
router.add('open', 'open');
router.add('guarded', 'guarded', isAuthenticated);

events.listen('route:open', () => {
  console.log('open route is always accessible');
});

events.listen('route:guarded', (data) => {
  console.log('made it past the guard function!');
});

router.start();
```

故障排查

一旦路由数量增长到足够大，就不得不对复杂情况进行故障排查。如果提前知道问题所在，就能更好地解决。还可以在路由实例中构建故障排查工具，加快解决速度。扩展系统的寻址能力意味着要快速且可预测地响应问题。

路由器冲突

路由器冲突很令人头疼，因为跟踪这类问题相当棘手。当一个模式拥有更通用的版本时，再添加更具体的模式到路由组件中就会发生冲突。越通用的模式越容易冲突，因为它能匹配最具体的 URI，而这些 URI 本来是要用更具体的模式来匹配的。然而，这些 URI 永远都测不到，因为通用模式总会先执行。

这种情况发生时，无法立刻察觉是路由发生了问题，因为不正确的路由回调函数运行得相当好，UI 看起来一切正常，除了有一个地方略微偏移。假如路由是以先进先出的顺序

执行，URI 的具体性会影响匹配结果。也就是说，如果先添加的模式越通用，当它们被激活的时候，它们就总能匹配更具体的 URI 字符串。

对很多这种 URI 进行排序会非常耗时，因为需要对比新添加的路由的顺序和已有路由的模式。假如开发人员都往同一个地方添加路由，还可能造成提交时的冲突。这也再次证明，让组件分开管理各自的路由是有好处的。它使得定位和处理潜在的冲突路由更加简单，因为组件内部拥有相似 URI 模式的数量小得多。

接下来的例子展示了一个路由组件里两个冲突的路由：

```
// 在routes中查找第一个匹配的路由，对uri进行测试。
function match() {
  for (let route of routes) {
    if (route.route.test(uri)) {
      console.log('match', route.name);
      break;
    }
  }
}

var uri = 'users/abc';

var routes = [
  { route: /^users/, name: 'users' },
  { route: /^users\/(\w+)/, name: 'user' }
];

match();
// 匹配 users。
// 注意，假如我们仔细查看uri的话，得到的结果可能与预期不一样。
// 这说明在测试一系列URI规则的时候，顺序的重要性。

routes.reverse();

match();
// 匹配 user。
```

记录初始配置

不要配置完所有相关的路由后才让路由组件监听 URI 的改变。举个例子，假如每个组件是单独配置路由组件的，没必要等到组件配置自己的路由时，路由组件才开始监听 URI 的变化。

主应用组件可以先初始化从属组件，完成时再让路由组件开始监听。如果单独的组件封装了自己的路由，在开发过程中，就很难获取完整的路由配置。在这种情况下，需要在路由组件中添加一个选项，记录完整的配置，包括模式及触发的事件。这有助于扩展规模，因为不需要牺牲模块化路由就能得到整体的路由情况。

记录路由事件

除了记录原始的路由配置，当 URI 改变事件被触发时，记录这个生命周期也是很好的。它跟前面章节里讲的事件机制记录不一样，这些事件会在路由组件触发路由事件时被记录。

在构建一个路由很多的大规模 JavaScript 系统时，需要知道路由组件的所有细节，还有它在运行时的行为。路由组件对应用的可扩展性极为重要，因此即使最微小的细节也值得我们关注。

举个例子，当路由组件遍历路由查找匹配模式的时候，如果能知道路由组件的执行过程会有很大的帮助。同样重要的还有路由组件解析 URI 字符串的结果，这样就能与下游的事件处理函数所接收的字符串进行比较。但并不是所有的路由组件都支持这个级别的日志。假设需要这些日志，一些框架需要提供大量的组件接口，同时还要提供良好的扩展机制。

处理非法资源的状态

路由组件是无状态的，它只是将 URI 字符串当作输入，基于模式匹配标准来触发事件。一个寻址相关的可扩展性问题与路由组件的状态无关，而是跟监听路由的组件状态相关。

举个例子，假设从一个资源导航到另一个资源，当访问新资源时，前一个资源可能会发生变化。对于某个特定用户来说，这个资源很可能就变成非法资源了，但是它仍在用户的历史记录里，用户只需要单击浏览器的返回键就可以访问。

以上是路由组件和寻址可能导致的边缘情况，但处理边缘情况不是路由组件的职责。边缘情况可能是因为将很多 URI、很多组件还有复杂的业务逻辑组合到一起了。路由组件只是协助处理大规模方案的机制，并不负责实现方案。

小结

本章详细探讨了寻址能力以及在应用扩大时实现这种架构属性的方式。

本章开头展示了两种路由方法：利用 hash change 事件和利用现代浏览器支持的 history API。大多数框架都将这两种方式的差异抽象化了。接着介绍了路由组件的职责，以及在触发事件时它们应该如何与其他组件解耦。

URI 的设计本身也会影响软件的可扩展性，因为它们必须保持一致并且可预测。即便是用户，也能利用可预测性来帮助其扩展软件的使用。此外，URI 编码了信息，这些信息需要传到路由处理函数中，这一点在实现时也需考虑到。

再接下来，介绍了触发路由的不同方式，标准方式是单击链接。假如有很好的连接性，应用就会有很多链接。为了帮助扩展这么多的链接，需要一个方法来自动生成 URI 字符串。接下来，我们将带大家探索组件运行所需的元数据，即组件中的用户偏好和缺省值。

6

用户偏好和默认设置

任何大型 JavaScript 应用都需要配置组件。组件的配置根据应用不同，其范畴和本质都不一样。在配置组件时，需要考虑一些可扩展性的影响因素，我们在本章中都将一一讲解。


先来认识几种需要处理的偏好类型，本章之后的部分会针对这些偏好以及如何处理这些偏好，详细地论述具体的可扩展性问题。

偏好类型

在设计大规模 JavaScript 系统时，主要关心的偏好类型有三种，包括：地区、行为和外观。在这一节将分别定义这三种偏好类型。

地区

如今，一个应用想要在全球范围内获得成功，就不能只支持某个地区。出于全球化和互联网的考虑，是否能在世界上其他地方创建应用已经成为新的标准。因此，设计 JavaScript 架构时要能够无缝适配多个地区，使不同地区的用户能够同等轻松自信地使用应用。

 使组件在任何地区都可用的过程叫做国际化(internationalization)，
而针对特定地区为应用创建特定数据的过程叫做本地化
(localization)。


由于涉及到用户界面的每一个视觉因素，国际化/本地化非常困难。尽管很多组件不需要关心地区，比如控制器（controllers）和数据集（collections），但是国际化/本地化还是相当重要的。比如，任何原本可以硬编码到模板里的字符串标签，现在都需要先经过一个感知地区（locale-aware）的翻译机制进行处理。

语言翻译本身就很困难，更何况地区数据还涉及到一些特定文化内容。比如，日期/时间或者货币价格的格式，这些还只是最普遍和直接的元素。上至整个页面的布局，下至如何定量，这些都与地区相关。

行为

组件大部分的行为都写在代码里，并且是固定的。不同的偏好导致的行为变化既微妙又重要。当组件相互作用时，一定会有一些不兼容的组合，从而导致一些问题。

比如，实现组件时，一个函数可能会从配置信息里获取一个值来进行计算。这个值可能是用户偏好，也可能是为了方便维护而存储的信息。

 在本章剩下的部分，我们会讨论偏好的各种配置，以及在配置组件里所有偏好设置的总效应。

行为偏好对用户看到的内容会有不同的影响，比如将组件关掉，或者禁用。这种偏好会导致在 UI 里不渲染这个组件。另一种偏好则决定显示多少元素。比如，用户告诉应用程序希望看到每一页显示多少条搜索结果。

这几种偏好不一定会直接影响终端用户。也就是说，一个组件可能存在某种偏好，这种偏好不会直接暴露给用户。可能仅仅是为了让开发更灵活，用来减少代码量。可配置的组件有多种形式，因此，需要有针对性地解决这些问题，才能更好地扩展软件规模。

不仅要考虑前端组件，一个特定的偏好也可能改变后端行为。简单的偏好也许只是一个查询参数，复杂的偏好却可能导致使用一个不同的 API 端点（endpoint）。所有这些看起来无害的偏好加起来会造成深远的影响，甚至可能影响系统的其他用户。

外观

假如一个现代的 JavaScript 应用打算扩大受众范围，它的外观必须是可配置的。小至一个可配置的 logo，大到一个可能彻底改变 UI 风格的主题，都属于外观配置的范畴。

一般情况下，外观的改变主要集中在 CSS 属性，比如字体、颜色、宽度、圆角，等等。尽管 CSS 的实现大部分不会触及到 JavaScript 开发者的主要职责，但还是得注意主题的界限。

比如，假设应用的外观及其配置方式需要很灵活，就应该在运行时让用户选择自己的主题。因此我们需要实现一个切换主题的机制来让用户与之交互。甚至，需要在某个地方存储和加载主题 UI 的偏好。

以上只是粗粒度的主题配置，还要考虑细粒度的外观配置。尽管前者更流行，我们还是会讨论如何配置特定组件的具体样式。外观的粒度会与其他可扩展性影响因素同时存在，比如，软件在哪里部署以及配置 API 的能力。

支持地区

让组件支持国际化是一个好主意。事实上，有很多 JavaScript 工具可以帮助我们完成这个事情。有些工具是独立的，有些工具则是针对特定框架进行定制的。这些工具用起来很简单，但是还有许多与地区相关的因素需要考虑，尤其是在规模扩大的情况下。

决定支持哪些地区

一旦软件的生产环境支持国际化，下一步就要决定支持哪些地区。当完成了第一步，即确保了所有组件的国际化时，我们其实仅仅支持了一个地区，即默认地区。在刚开始的时候，做到这步就够了，因为可能还要好几年软件才会需要支持第二个地区。

较新的软件项目通常都会这样。明知道要把国际化排到日程上，但是经常被其他需求挤到一边去。有些人认为无须投入精力去支持其他地区，因为目前这个需求并不急迫。而反方认为，随着组件增加，事后再做国际化的实现成本会大大增加。所以这也带来了另一个需要权衡的可扩展性问题：我们是希望应用在不同文化之间传播，还是希望立刻上市投放？

一般情况下，我们认为必须支持国际化，于是就需要先确定优先支持哪个地区，其他地区可以暂时不支持。除非真正有需求，否则不要盲目确定很多的支持地区。每个地区都会占据物理空间，还需要人来进行维护。所以当没有用户为这种额外的复杂性买单时，是不值得为这些地区做本地化的。

我们应该只根据用户需求来选定地区。如果在某个地区有好几百人需要本地化支持，而另外的地区只有十几个人有需求，很明显应该支持人多的地区。我们应该将地区像功能一样进行优先级排序。

维护地区

要支持一个特定地区，首先需要翻译 UI 里的字符串消息。这些字符串有的被静态编码到模板文件里，有的则在 JavaScript 模块中。如果能查找到字符串并一次性将其翻译完最好，但几乎没有字符串会一直保持不变，通常都会稍微进行调整。另外，随着软件的扩展，会添加更多的组件，于是就需要翻译更多的字符串。

单就翻译字符串来说，影响可扩展性的因素就是地区的数量，所以尽管我们能搞得定翻译，但还是要谨慎地选择有限的地区。然而事情还远不只这些，因为有些消息字符串可以直接从源语言映射到目标语言，但是有些东西，比如语法的改变（不同的变体会有不同的含义）就没有这么直接了。事实上，这些用法有时需要使用专门的国际化库。

其他的本地化数据，比如日期/时间格式，则不需要太多维护。对于特定地区，在整个应用中也就会用到一到两种格式。在这些格式中，用户会倾向于使用他们文化中的标准格式。好在我们可以使用**通用区域数据仓库（CLDR）数据**，这是一个可下载的通用区域数据仓库。这是一个不错的开始，因为大部分情况下，这种数据都够用，并且很容易在请求时重写。


设置地区

一旦国际化库准备就位，并且敲定了地区，我们就可以开始从不同文化角度来验证应用的行为了。关于应用的行为，需要考虑很多因素。比如，需要让用户方便地选择地区，并且还要跟踪用户的选择。

选择地区

在 JavaScript 应用中实现地区选择有两种常见的方式：第一种方式是利用 `accept-language` 请求头，第二种方式在用户设置页面放一个选择工具。

用 `accept-language` 方式的好处是无须用户输入。应用把语言偏好设置发送给用户浏览器，然后就能设置地区了。但是这种方式从可用性和实现角度来说非常局限。比如，用户无法控制他们浏览器的语言偏好，或者浏览器并没有应用所支持的地区偏好。

 `accept-language` 请求头方法的另一个技术问题是：无法直接将请求头从浏览器传给 JavaScript 代码，尽管这两者都在浏览器里。比如，假设 JavaScript 代码需要得到地区偏好，以便加载合适的地区数据，那么它需要访问 `accept-language` 头。为了实现这个需求，还要后端技术支持。

更灵活的方式是给用户提供一个地区选择工具，用户一眼就能看到想选的地区。不过，我们还得想办法存储用户的选项，这样用户就不用反复选择了。

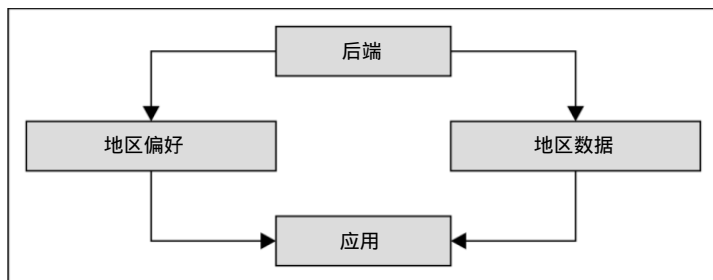
存储地区偏好

用户选择了地区偏好后，可以把偏好值存到 cookie 中。下一次浏览器加载应用时，就能直接使用地区偏好信息，然后我们就能将选择器标记为最合适的选项，同时加载相关地区的数据。

用 cookie 存储地区偏好的缺点是，一旦用户换了浏览器就得重选一次。现在越来越多的用户使用移动设备，用户在一台设备上做的改变需要在任何设备上打开应用时都有效。这还真是个大问题，用 cookie 没法解决。

假如使用后端 API 来存储地区偏好，用户则可以在任何地方使用应用。接下来要做的是，加载相关地区数据供其他组件使用。一般来说，我们希望在开始渲染数据之前，地区数据就能准备就绪，因此发给后端的首批请求就应该包含地区数据。有时后端会将所有地区作为一个资源传回，如果支持很多地区，加载这个资源的预先成本就会很明显。

但是，如果先加载地区偏好，就只需要加载目标地区数据。这能提升初始加载速度，但是需要权衡的是，用户切换新的地区时会较慢。这不太可能经常发生，所以最好不要加载用不着的数据。



图注：应用先加载地区偏好，然后用地区偏好来加载本地数据。

URI 中的地区

除了将地区偏好存储在后端或者存在 cookie 中，还可以把地区编码成 URI 的一部分，通常在 URI 的开头部分用两个字符的代号表示，例如 `en` 或者 `fr`。这种方式的好处是不需要存储偏好设置。如果仍然要给用户提供一个地区选择器，最后会生成一个新的 URI 而不是生成一个可存储的偏好设置值。

将地区偏好编码到 URI 中跟 cookie 方式有一样的缺点。尽管我们可以收藏一个 URI 或者将 URI 发给别人，别人会看到同样的地区，但这并不是一个永久的偏好。虽然可以将偏好永久存储并且在加载应用时更新 URI，但这种方式不便于扩展，因为它增加了路由和生成 URI 的复杂度。

通用组件配置

正如在前一节看到的地区偏好，我们需要加载偏好值，之后这个值才能被每个组件使用，或者像地区偏好一样，只被一个组件使用，但是这个偏好值会间接影响所有组件。从更高的角度来看，除了地区偏好，有很多别的东西需要配置到组件里。本节会从通用角度来看待这个问题。首先需要决定一个组件可配置的部分有哪些，然后采取一些方法在运行时给组件加载偏好。

选择配置的值

配置组件的第一步是决定偏好，即组件的哪些部分需要配置，哪些部分保持静态。这一步是无法做到精确的，通常会在事后才能发现有些静态部分本应该是可配置的。特别是当软件刚起步的时候，最好的方法是反复试验，找到可配置的偏好。过多地思考初始化配置反而会拖慢扩展的进度。

静态的部分更简单，更加结构化且可移植性更差。这就消除了潜在的边缘条件和性能问题。没人能完全做到预先判断可配置的值。当软件成熟时，我们看问题的角度也会更成熟，当某些偏好配置准备就绪，我们将会对预期有更好的想法。

比如说，我们会开始注意到组件里的重复内容。它们大体相同，只有微小的变化。如果继续添加新的组件类型，而这些类型只有极小的变化，项目就会陷入扩展的困境中。由于代码的增长不受控制，再加上特定组件的职责变模糊，会令开发人员感到困惑。

此时需要利用配置来进行扩展，引进偏好设置来支持新的组件类型。比如，假设需要一个新的视图，它跟另一个已有视图相同，只是在一个 DOM 事件的处理方式上不一样。与其实现一个新的视图类型，倒不如增强已有的视图，让其接收新的函数值来覆盖该事件的默认处理函数。

另一方面，不能只是贸然地引入组件偏好，因为这样做是在用新的可扩展瓶颈去替换旧的瓶颈。我们要考虑性能问题，因为每新增一个偏好配置，性能都会受到影响。还要考虑代码的复杂度，因为使用偏好远不如使用静态值直接。还要考虑不同开发人员同时进行开发时，可能引入偏好值不一致的问题。最后，还要考虑跟踪和记录某个组件的各种偏好。

存储和硬编码默认值

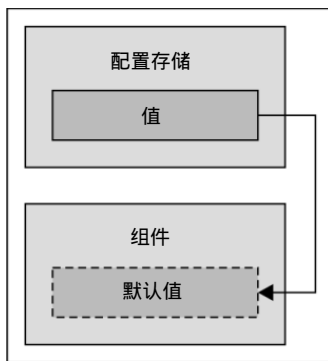
就组件而言，应当尽可能地将偏好同常规的 JavaScript 变量一样处理。这会让代码更灵活，比如，将偏好设置替换成静态值就不会造成太大的影响。通常我们都会给常规变量声明一个初始值，那么也应该给偏好设置声明一个默认值。即使由于某些原因无法得到后端存储的偏好，软件还是会用正常的默认值继续运行。

任何偏好都应该有一个备用的默认值，而且应该把这些值存起来。理想情况下，常见

的场景都能使用默认值，所以不需要提供偏好也能使用软件。假如因为某些原因无法得到后端存储的配置值，即使不是最理想的配置，软件用硬编码的默认值也能继续运行。

有时候，没有配置值的软件注定无法运行，这时应该让软件快速失败，而不是用默认值。尽管使用默认值时软件是全功能的，由于依赖客户和部署，这种模式会比软件不可用更糟糕。当部署大规模 JavaScript 应用时，需要考虑清楚。

假设后端要删除一个修改过的偏好值，默认偏好值会让应用更安全，这就好比重置出厂设置。换句话说，如果在调整偏好值时给软件引入了问题，可以只删除存储的值。如果不必在后端存储默认值，那么重写默认值就没有风险。



图注：默认值一直保留，但是可以轻易地用后端的偏好值重写默认值。

对后端的影响

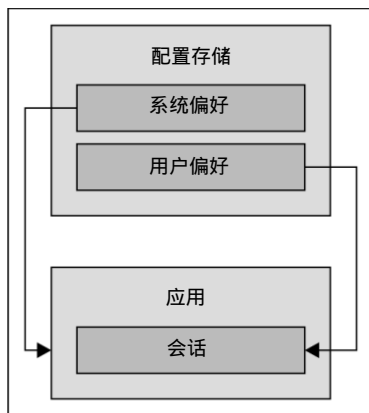
如果为了方便移植把偏好值存储在后端，那么需要提供一些机制往配置中写入新的偏好值，或者取出偏好值。理想情况下，后端应该提供一个 API，能定义任意的键值对形式的偏好，并且允许通过一个请求重新取回所有配置。

这样就可以在开发组件的时候给组件定义新的偏好，并且不会影响到后端团队，这对前端开发相当重要。对后端 API 来说，前端配置是随意的，无论有没有 UI，API 都一样有效。

有时候，这种方式带来的麻烦会超过好处。如果在整个应用中只需要用到几个配置值，

只需要考虑维护一份静态的 JSON 文件来作为前端配置。这种方式很随意，可以随时定义偏好，而且就抓取偏好值而言，跟 API 效果一样。

对于用户定义的偏好，这种方式就没那么好了。比如，如果用户位于偏好地区，那么应用可能会预先指定一个默认地区，直到用户主动选择新地区。如果用户只是为自己设置偏好，而不是为系统里的每个用户设置，就需要用前面提到的配置 API。这种方式通常将偏好地区存储在数据库，而且存储的数据一定是与用户相关联的。但并不是所有偏好值都与用户关联，有些是由部署操作者来设置的，用户碰不到这些值。



图注：当前用户的会话可以用来加载当前用户的偏好。这与系统设置不同，系统设置不会随着用户改变而变化。

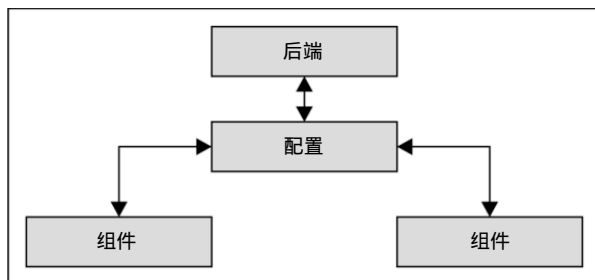
加载配置值

加载前端配置有两种方式。第一种方式是全部加载，因为任何配置都会在 UI 中呈现。那么在路由开始处理任何事务之前，必须等待配置文件加载完毕才能执行后续操作。这意味着要一直等待 promise 返回配置数据。全部加载的缺陷很明显：初始加载时间会变长。但是好处是可以一次获得后续的所有数据，不必再次请求。

可以使用浏览器的本地存储来缓存偏好值。用户很少修改偏好值，并且缓存还可能提升初始加载的性能。但是这样增加了复杂度，所以只有在配置值太多，加载很费时的时候，才会考虑这种方法。

除了预先加载所有配置，还可以按需加载。也就是说，当一个组件要被实例化时，才

发一个请求获取配置。这种方式的好处是高效。但话说回来，得加载多少配置才用得着这么复杂的方式？所以还是尽可能预先加载所有的配置吧。



图注：一个配置组件，用来与后端进行通信，它给操作偏好值的组件提供了抽象方法。

配置行为

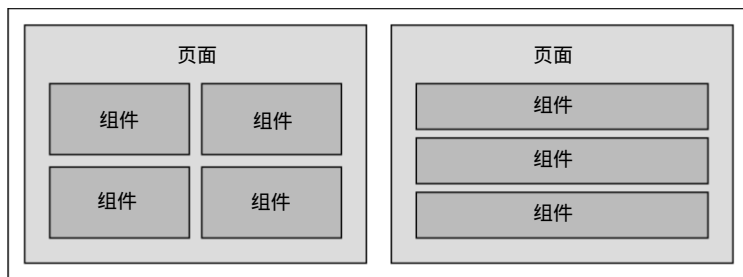
假如应用实现得好，组件的行为大多是包含在组件内部的。它们暴露在外面的只是一些偏好，用来对组件的行为进行细微的调整。行为可以是组件内部关注的东西，比如使用什么类型的模型或者偏好算法；也可能是面向用户的，比如启用某个组件或者设置显示模式。正是这些偏好帮助我们扩展组件，让它们能在不同的上下文运行。

启用和禁用组件

一旦软件达到一定的量级，不是所有功能都会被每个用户用到。不管是对软件供应商还是客户而言，像切换组件的启用/关闭状态这种简单的功能都是一个强大的工具。比如，尽管在软件里某些用户角色需要一些功能，但这并不是普遍情况。为了更好地针对普通用户进行优化，我们可以选择禁用掉某些不太常用的高级功能，以便让布局更干净、性能更高。

反之，如果默认开启所有功能，但是能够提供关闭组件的方式，用户就能自己决定需要哪些功能。如果用户能够按照自己喜欢的样子安排 UI，移除对他们没用的元素，就能创造更好的用户体验。

可以肯定的是，这种方式会对整体布局造成影响。假如不花点时间将布局设计成可扩展的，开关组件的功能就没有意义了。在设计布局时，需要彻底考察用户或者应用本身可能用到的各种配置场景。



图注：禁用页面上的组件可能会更新布局，应用的样式要能处理这种情况。

改变数量

最好在设计阶段就能估算出展现在 UI 里的元素数量。列表里的元素个数最好是最优的数量，并且不会因为改变数量偏好造成太大差异。问题是这些数量实在是太主观了，往往取决于在应用里执行操作的用户，他们习惯什么数量，在使用软件的同时他们在做什么，以及很多其他因素。数量偏好的默认值可能并不是最优的。

一个关于数量的常见问题是：我想在我的屏幕上展现多少实体？这里的实体可以是在应用中常见的表格工具，可以是一个搜索结果页，还可以是任何能够呈现一批事物的元素。为了提高效率可以使用默认较少的元素，同时支持用户选择更多。



检查用户偏好的安全性是很好的做法。其中一个安全措施是在页面放置选择器，只能选择合法值，不能接受用户的任意输入，就能避免在一个表格里展现 1000 个实体这种情况的发生。尽管如此，返回这些数据的 API 也应该校验和限制数量参数。

另一个数量问题是：要展示哪些实体属性？在表格中，也许用户想看到特定列，同时隐藏其他列。类似于这种偏好应该被持久化，因为如果用户费劲地设置了想要看到的数据，他肯定不想再次设置。

改变数量偏好会对后端造成影响。比如，在获取数据的时候，需要将展示的实体数量传给后端，这样能避免抓取不展示的数据。改变数量偏好还会对模型或集合造成影响。比如，对于在一个特定的 UI 区域展现的数据或属性，需要将其作为参数传递，这样只需要请求模型或集合里的子集。

改变顺序

数据集在 UI 里的顺序也是一种常见的行为偏好，也是我们最想支持的偏好。其中配置默认顺序带来的影响最大。比如，按照修改日期对每个集合进行排序就是很好的默认顺序，那样最近的实体就会先展示。

许多表格组件允许用户切换给定列的升、降顺序。这些属于动作，不一定是偏好。然而，假如默认顺序永远都不是用户想要的，这种动作就会让人很恼火。因此在提供默认顺序偏好的同时，可能还需要提供方法让用户能随时单击列头进行排序。

还可能需要更复杂的排序偏好，而可点击的列头不一定能满足这种需求。比如，需要根据不在 UI 里的内容（比如相关度或者畅销度）来排序，该怎么办呢？可以增加一个控制器，其实这也是另一种潜在的偏好，因为这能让用户体验更棒。

```
// users.js
export default class Users {

  // 接受一个collection数组，还有一个order字符串。
  constructor(collection, order) {
    this.collection = collection;
    this.order = order;

    // 创建一个迭代程序，能够对collection数组进行迭代，
    // 而不需要直接访问它。
    this[Symbol.iterator] = function*() {
      for (let user of this.collection) {
        yield user;
      }
    };
  }

  set order(order) {

    // order参数被拆分成两部分：key和direction。
    var [ key, direction ] = order.split(' ');
```

```
// 对 collection 进行排序。
// 假如属性值能被转换成小写形式，那就转换一下，避免格式不一致。
this.collection.sort((a, b) => {
    var aValue = typeof a[key].toLowerCase === 'function' ?
        a[key].toLowerCase() : a[key];

    var bValue = typeof b[key].toLowerCase === 'function' ?
        b[key].toLowerCase() : b[key];

    if (aValue < bValue) {
        return -1;
    } else if (aValue > bValue) {
        return 1;
    } else {
        return 0;
    }
});

//假如direction是desc，我们需要将顺序反转。
if (direction === 'desc') {
    this.collection.reverse();
}

}

// main.js
import Users from 'users.js';

var users = new Users([
    { name: 'Albert' },
    { name: 'Craig' },
    { name: 'Beth' }
], 'name');

console.log('Ascending order...');
```

```
for (let user of users) {
    console.log(user.name);
}
//
// Albert
// Beth
// Craig

users.order = 'name desc';

console.log('Descending order...');
for (let user of users) {
    console.log(user.name);
}
//
// Craig
// Beth
// Albert
```

配置通知

当用户执行某些动作，比如开、关某些组件的时候，我们需要提供一个状态反馈。这个动作是成功还是失败，还是正在运行中？这些一般都用通知来实现，比如在屏幕角落或者面板里短暂弹出的消息。

用户可能希望控制通知的方式，因为没有什么比弹出根本不想看的垃圾信息更让人恼火的了。因此通知的主题可能就是偏好之一。比如，剔除不相关的实体类型的通知。


另一个潜在偏好可能是通知在屏幕上的停留时间。比如，通知是应该一直出现在屏幕上直到被看到，还是应该在 3 秒后自动消失？极端情况下，假如通知让用户不能忍了，用户可能希望关掉所有通知。如有必要，要一直记录用户的动作日志以便之后提供更方便的浏览。

行内选项

如何收集用户的偏好输入呢？对于不活跃的全局偏好，比较好的方式是提供一个分好

类的设置页。然而，在一个设置页面配置单个工具的特定属性很烦人，这时最好提供行内选项。

行内意味着用户可以使用目标 UI 里的元素来设置偏好。比如说，选择一个表格里需要展示的特定列时，将这种偏好藏在一个设置页里就不太合适。把偏好控制放在他们想控制的事物附近，会更加一目了然。有了上下文，用户能更轻松地理解选项的用意。

 在上下文中进行偏好控制的缺点是可能会弄乱 UI。假如页面上有很多组件，每个都有偏好控制，那么很可能会制造困惑而不是创造方便。

改变外观

如今，应用的外观很少是静态不变的。相反，应用会提供几个主题给用户选择，或者用软件里内置的功能就可以轻松创建主题。这能让我们的客户来决定软件呈现给用户的样子。除了将主题打包到应用以便更新外观，可能还需要设置单独的样式偏好。

主题工具

如果想要应用能在请求时改变主题，就必须在 CSS 和标签中加入大量的设计和架构。这一话题超过了本书范畴，但还是值得看看那些能帮助生成主题的工具。

首先要介绍的是 CSS 框架。就像 JavaScript 框架一样，CSS 框架定义了统一的模式和规则。剩下的就取决于组件开发者的发挥了，开发者负责将这些 CSS 模式应用到组件以及它们生成的标签上。可以将一个主题看作一组样式偏好，当配置发生变化时，外观就会根据新的偏好值发生变化。当一个 CSS 模块跟其他主题使用了一致的属性时，它就是一个主题，只是属性的值不一样而已。

另一个工具是 CSS 编译器，这属于后端构建过程的一部分。这些工具会将用 CSS 方言（CSS 扩展语言）编写的文件作为输入，并对其进行处理。这种预处理器语言类型的好处是：可以更大程度地控制样式偏好实现的方式。比如，CSS 中没有变量，但是预处理器语言有，这个可配置功能非常方便。

选择一个主题

拥有一个可定制主题的用户界面后，就需要加载特定的主题实例。如果不能让用户选择自己的主题，通过改变偏好值来改变设计也是相当不错的。当要实现一个新设计时，这种方式肯定会让代码部署到生产环境变得更加容易。

可能不久以后，我们会让用户来选择自己的主题。比如，当我们已经积累了大量用户，产生了自定义主题的需求。接下来要创建主题选择器，就跟系统里的其他偏好值一样，需要在页面放一个主题选择工具，用户的选择会映射到一个路径，从一个主题切换到另一个主题就这么简单。

另一种可能情况是，给不同的用户角色设置不同的默认主题。比如，假设用户以管理员身份登录成功，应用就会展示不同的用户界面来告诉用户当前登录的角色很特殊。在需要截图或者类似场景下这种方式很有用。

单独的样式偏好

应用的外观还可以在单个元素的级别上进行修改。也就是说，假设需要改变某个元素的宽度，可以在屏幕上直接修改。或者用户不喜欢当前的字体，想改变它，但不想改变其他属性。

我们应该尽量避免这种细粒度的样式偏好，因为它们很难扩展。组件必须避免考虑特定的样式，因为在绝大部分情况下这都会让软件的真实目的降级。不过在某些情况下，在屏幕上挑出一个不同的布局不会有什么问题，因为这通常只需要换掉 CSS 类名。

另一个可能方法是利用拖拽的交互来设置元素大小。但最好是短暂的交互，不要当作永久的偏好。我们希望优化的是普遍的配置值，为迎合个人口味而提供的缩放元素功能完全没有普遍价值。

性能影响

最后，概括一下目前为止讨论到的各种配置导致的性能影响。假如在某方面真的需要配置值，但这些配置可能会影响整体性能——那么需要设法抵消这种代价。

可配置地区的性能

到目前为止，跟地区相关的最显著的性能瓶颈就是初始加载，因为需要在为用户呈现任何内容之前加载全部的地区数据，包括字符串信息译本，还有其他本地化数据。如果不止预加载一个地区数据，初始化的性能问题会更严重。

提高加载性能最好的方式是只加载用户真正想要的地区。一旦他们设置了偏好，就不太可能会频繁改变这些偏好。因此一开始就加载其他地区数据没有任何好处。

在渲染界面时有一个无法避免的速度问题，因为有太多数据需要传递到本地化机制中。不过单这一点不太可能造成性能问题，因为大部分操作都很小并且很高效，只是简单地查找以及字符串匹配。但是也算是一个额外的间接开销，需要考虑进来。

可配置行为的性能

改变组件行为的配置也会带来极小的性能影响。事实上，可配置行为的性能特征很像可配置地区的性能特征，最大的挑战也是初始配置的加载。之后只是一些查找操作，这些操作很快。

需要注意的是，当需要配置很多组件时，虽然单个查找很快，很多查找加起来就会对性能造成影响。尽管需要很久才会到达这种程度，但还是存在风险。

以下案例展示了当一个数据集被排序时，配置的方式影响了其他操作的性能，其他操作是顺序依赖的并且会被频繁调用。

```
// users.js
export default class Users {

  // users 集合不包含数据以及order属性。
  constructor(collection, order) {
    this.collection = collection;
    this.order = order;
    this.ordered = !!order;
  }
}
```

// 设置order属性时，需要对内部的collection数组进行排序。

```
set order(key) {  
  this.collection.sort((a, b) => {  
    if (a[key] < b[key]) {  
      return -1;  
    } else if (a[key] > b[key]) {  
      return 1;  
    } else {  
      return 0;  
    }  
  });  
}
```

// 查找集中最小的数据。

// 假如集合是有序的，那么只用返回第一条集合数据。

// 否则，需要迭代集合来查找最小的数据。

```
min(key) {  
  if (this.ordered) {  
    return this.collection[0];  
  } else {  
    var result = {};  
    result[key] = Number.POSITIVE_INFINITY;  
  
    for (let item of this.collection) {  
      if (item[key] < result[key]) {  
        result = item;  
      }  
    }  
  
    return result;  
  }  
}
```

// 与min()相对的方法，

// 假如集合是有序的，返回最后一个集合数据。

```
// 否则，查找最大的数据。
max(key) {
  if (this.ordered) {
    return this.collection[this.collection.length - 1];
  } else {
    var result = {};
    result[key] = Number.NEGATIVE_INFINITY;

    for (let item of this.collection) {
      if (item[key] > result[key]) {
        result = item;
      }
    }

    return result;
  }
}

}

// main.js
import Users from 'users.js';

var users;

// 创建一个“有序的”users集合。
users = new Users([
  { age: 23 },
  { age: 19 },
  { age: 51 },
  { age: 39 }
], 'age');

// 调用min()和max()并不会对整个集合进行迭代，
// 因为集合已经是有序的了。
console.log('ordered min', users.min());
console.log('ordered max', users.max());
```



```
//  
// 有序 最小 {age: 19}  
// 有序 最大 {age: 51}  
  
// 创建一个“无序的”用户集合。  
users = new Users([  
    { age: 23 },  
    { age: 19 },  
    { age: 51 },  
    { age: 39 }  
]);  
  
// 每当调用min()或者max()时,  
// 要对集合进行迭代,找到最小或最大的数据。  
console.log('unordered min', users.min('age'));  
console.log('unordered max', users.max('age'));  
//  
// 无序 最小 {age: 19}  
// 无序 最大 {age: 51}
```

行为偏好可以完整地切换两个函数。它们可能有相同的接口,但是有不同的实现。在运行时决定使用哪个函数并不会有太大的性能开销,但是同时也要考虑内存消耗。比如,假设应用里有很多偏好支持不同的函数,除了存储作为偏好值的函数之外,还需要额外存储默认函数。

可配置主题的性能

可配置主题唯一的延迟问题就是获取主题偏好的初始开销。接下来是下载和将样式表应用到标签,这和只有一个静态样式表的应用没有区别。如果允许用户切换主题,还会因为等待下载,渲染新的 CSS 和静态资源造成额外延迟。

小结

本章介绍了在大规模 JavaScript 应用中的配置化概念。主要的配置范畴包括地区、行为和外观。地区是当今 Web 应用很重要的一部分,因为我们不能阻止其他地区的客户使用

我们的应用。还有国际化带来的可扩展性问题，国际化给开发生命周期增加了复杂性，并且增加了维持地区信息的开销。

偏好需要存储在某些地方。存储在浏览器没问题，但是这种方式无法移植。更合适的方式是将偏好存储到后端，在应用初始化的时候加载。增加太多的偏好会带来很多问题，包括区分用户定义的偏好和系统偏好。如果引用了健全的硬编码默认值的话就没有问题。

应用的样式是另一个可配置的维度，一些框架和构建工具能帮助我们构建外观主题。可配置组件需要考虑的性能问题较小，下一章会介绍在扩展软件时突然出现的性能问题。

7

加载时间和响应速度

JavaScript 的可扩展性包括应用的加载时间和用户跟应用交互时的响应速度。我们所指的性能就是这两种架构质量的合称。对用户而言，性能是衡量质量的首要指标，必须重视。

随着应用新功能的增加以及用户基数的增长，必须设法避免与之相关的性能下降。初始加载会受 JavaScript 构件的负载大小等条件的影响，而 UI 的响应速度则更多地取决于运行时的代码特性。

本章主要介绍这两个性能维度，以及为此对系统其他部分影响的各种权衡。

组件构件

前面强调了大型 JavaScript 应用只是很多组件的集合。这些组件通过错综复杂的方式和彼此通信，而这些通信实现了系统的行为。在组件通信之前，需要先把组件发送给浏览器。了解这些组件由什么构成，以及它们如何被发送到浏览器，能够帮我们分析出应用的初始加载时间。

组件依赖

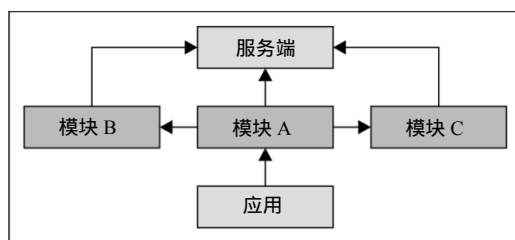
组件是应用的基础，这意味着我们要将组件发送到浏览器，然后条理清晰地执行它们。组件本身可以是完整的 JavaScript 文件，也可以是分散到若干个模块里的内容，所有“拼图”根据依赖图组合到一起。应用从一个组件开始作为应用的入口，它通过依赖加载找到所需要组件。比如，也许只有几个顶级组件映射到软件的关键功能。这是依赖树的第一层，

除非功能组件都是“一体成型”的，否则还可能进一步地处理模块依赖。

模块加载机制会遍历整个依赖树，直到得到所有需要的模块。关于模块和依赖，当拆分到比较合理的粒度时，就能屏蔽很多复杂性。不必掌握完整的依赖图，因为即使是中等规模的应用，这也是不切实际的。

这种模块化的结构，以及加载和处理依赖的机制，往往伴随着对性能的影响。也就是说，会影响初始加载时间，因为模块加载器需要遍历整个依赖图，还要向后端请求每个资源。尽管是异步请求，还是会有网络开销——这是初始加载过程中最耗时的部分。

然而，模块化并不意味着要忍受网络开销所带来的后果。当我们开始扩展功能和用户数量时，会有更多资源需要发送到每个客户端会话，越多的用户请求同一个资源，就会产生越多的资源竞争。好在模块依赖是可追踪的，这给构建工具提供了各种选择。



图注：JavaScript应用模块的加载方式，依赖是自动加载的。

构建组件

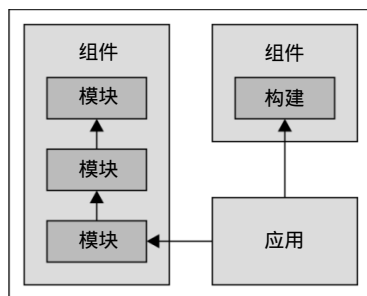
当组件到达一定的复杂度时，要实现全部功能，就得请求更多的模块。随着组件数量增多，所需的模块数量会成倍增长，同时还会有网络开销的问题。就算模块的负载很小，也必须要考虑网络开销。

实际上，我们应该尽量实现较小的模块，小模块更便于其他开发者使用。因为模块越小，变化就越少。前面提到过，模块和模块之间的依赖能够帮助我们分而治之，因为模块加载器能追踪依赖图并且按需拉取模块。

为了避免向后端发送太多请求，我们可以构建更大的组件构件作为构建工具链的一部分。很多现成的工具能直接用模块加载器来跟踪依赖，并构建相应的组件，比如 RequireJS 和 Browserify。这些工具很重要，使用这些工具能够选择合适的模块粒度，同时还能构建

更大的组件构件，或者是在运行过程中加载小模块。

网络开销对可扩展性的影响会很大。组件越多、越大，构建过程就越重要，尤其是构建过程中的混淆和压缩。另一方面，开发团队是否能够关闭这些构建步骤也会影响可扩展性。如果我们能变换发给浏览器的组件构件类型，开发过程就能进展得更快。



图注：构建组件会能够减少对构件的请求，也能减少网络请求。

加载组件

在这一节会介绍加载源模块和组件构件到浏览器的实现机制。现在有很多第三方工具可以构造模块并声明其依赖关系，但是目前更趋于使用新的浏览器标准来完成这些任务。除此之外，本节还会介绍懒惰模块加载，以及加载延迟对可用性的影响。


加载模块

很多生产环境的大型应用使用了 RequireJS 和 Browserify 等技术。RequireJS 是一个纯 JavaScript 的模块加载器，它的工具能构建更大的组件。Browserify 的目标是构建浏览器端的组件，它的代码是在 Node.js 环境下运行的。尽管这两者都能解决很多本章目前为止讨论的问题，但是新的 ECMAScript 6 模块方式才是更先进的方法。

基于浏览器的方式来加载模块和管理依赖的好处是，我们不再需要第三方工具了。如果语言本身就能够解决可扩展性问题，那么最好选择这种方式，因为它能减轻工作量。当然，这种方式并不是银弹，但它确实拥有很多理想的功能。

比如，我们不用再依靠发送 Ajax 请求并执行返回的 JavaScript 代码来实现模块加载了，因为现在这些工作都归浏览器负责。由于加入了标准的 `import export` 关键字，语法本身

也会更加整齐。但是，原生的 JavaScript 模块才刚刚出现，对于使用了其他模块加载器的代码，还没有足够的理由废弃掉。新项目可以尝试 ES6 的 transpiler 技术，它允许我们从头开始使用这些新的模块构造方法。

 应用所经历的和用户最终要承受的网络开销，有一部分是由 HTTP 标准造成的。最新的 2.0 版本草案解决了很多开销和性能问题。这对于加载模块来说意味着什么呢？假设能以最小的开销得到合理的网络性能，我们也许能简化构件。编译更大组件的需求就能降低优先级，然后就可以集中精力实现一个坚实的模块化体系结构。

懒惰的模块加载

整体编译组件会造成无法按需延迟加载某些模块。在编译的组件里，要么全有，要么全无，特别是当全部的前端文件被编译到一个 JavaScript 构件中时。这样做的好处是所需的模块都提前就绪了。初始加载完成 5 分钟后，如果用户想使用一个功能，由于代码已经在浏览器里，所以功能随时都可以运行。

相比之下，懒惰加载其实就是默认方式。简单来讲就是只有其他组件显式地请求某个模块时，才会加载该模块到浏览器。这个过程可以是一个 `require()` 调用或者是一个 `import` 声明。调用之后才会从后端抓取这些模块。这样做的好处是，初始加载会更快，因为只会拉取初始功能所需的模块。

但是当初始加载完成 5 分钟后，用户要使用某个功能，应用才会第一次 `require` 或者 `import` 一些模块。这意味着初始加载之后还会出现延迟问题。值得注意的是，在会话开始后，按需加载的模块从数量上来讲是很少的，因为肯定有一些共享的模块，它们在初始化的时候就已经加载了。

我们还要考虑系统的依赖问题。尽管延迟加载了特定模块，但是，可能会有间接依赖在不经意间加载了主界面上的模块，这些加载可能是多余的。开发者工具里的网络面板很适合处理这种情况，通过它就能看到是否加载了多余的内容。假如应用有很多功能，懒惰加载会特别管用。它能节省很多初始加载时间，可能有些功能用户根本不会用，也就省得加载了。

接下来是一个例子，展示了按需加载模块的概念：

```
// stuff.js
// export 一些功能，以便别的模块调用。
export default function doStuff() {
    console.log('doing stuff');
}

// main.js
// 链接被单击时再 import "doStuff()" 。

document.getElementById('do-link')
    .addEventListener('click', function(e) {
        e.preventDefault();

        // 在ES6中，只需要System.import()，但目前还不能兼容所有环境。
        var loader = new traceur.runtime.BrowserTraceurLoader();
        loader.import('stuff.js').then(function(stuff) {
            stuff.default();
        });
    });
```

模块加载的延迟

模块为响应事件而加载，而这些用户事件无处不在，比如开始运行应用或者选中了 tab，这些类型的事件都有可能加载新的模块。在这些代码模块传输或计算的过程中，我们能为用户做什么呢？由于等待中的代码不知何时才能完成，因此无法准确地执行那些优化加载体验的代码。

比如说，直到一个模块以及它依赖的模块都加载完成，我们才能执行一些操作，这些操作对用户可感知的 UI 响应速度至关重要。包括调用 API、操作 DOM 以响应用户。没有 API 数据，我们只能告诉用户，稍安勿躁，数据正在加载呢！由于模块还在加载，进度条还没走到头，如果用户忍受不了这种失落感，就会开始随便点击那些貌似可点击的元素。如果没有任何事件处理函数来应对这种操作，UI 看起来就会毫无响应。

以下案例展示了一个已经载入的模块运行耗时的代码时，会阻碍载入中的模块运行：


```
// delay.js

var i = 10000000;

// 吃掉一些CPU周期，
// 只要任意模块 import 了这个延时模块，引用者就会被延迟。

console.log('delay', 'active');
while (i--) {
    for (let c = 0; c < 1000; c++) {

    }
}
console.log('delay', 'complete');

// main.js

// 引入这个模块将会造成阻塞，因为它运行了一些耗时的代码。
import 'delay.js';

// 显示了链接，并且链接看起来可点击，但实际上什么也不会发生。
// 因为还没有设置事件处理函数。
document.getElementById('do-link')
    .addEventListener('click', function(e) {
        e.preventDefault();
        console.log('clicked');
    });
```

网络是不可预测的，应用在后端面临的可扩展性的影响因素也是不可预测的。大量的用户就意味着可能会在加载模块时出现高延迟。如果我们要扩展应用，就必须考虑这些情况，这时需要用到一些策略。在主应用加载之后，需要首先加载通知模块。

比如，UI 里有默认的加载元素，但是当第一个模块加载完成后，它要渲染更详细的信息，告诉用户正在加载的内容以及加载可能持续的时间，或者它只通知用户网络或者后端出现异常。当扩展应用时，必须尽早考虑这些情况，并且让 UI 看起来永远是可响应的，即使实际上并不是。

通信瓶颈

当应用拥有更多的动态模块时，它就需要更多的通信开销。因为组件需要彼此通信来实现功能的主要行为。如果我们迫切地想要解决这个问题，将组件之间的通信开销减少到 0，就会产生一套巨大且重复的代码。所以要想实现模块化的组件，通信所带来的成本问题就无法避免。

这一节会讨论扩展软件时会遇到哪些通信瓶颈。我们需要寻找一个平衡点，在不牺牲模块化的前提下提高通信性能。最有效的方式就是利用 Web 浏览器里的性能分析工具。这些工具还能暴露用户跟 UI 交互时响应速度的问题。

减少间接引用

组件彼此通信的机制，可以抽象化描述成事件经纪人。经纪人的工作就是维护所有特定事件类型的订阅者列表。JavaScript 应用主要在两方面扩展：特定事件类型订阅者的数量以及事件的数量。考虑到性能瓶颈，这些可能很快就会失控。

首先我们要特别注意功能的组成。为了实现一个功能，我们会遵循与已有功能相同的模式。这意味着会使用相同的组件类型，相同的事件，等等。虽然这些功能会有一些微妙的变化，但是最主要的模式是一样的。所有功能都遵循相同的模式，这是个不错的实践，将有助于减少开销。

比如，假设应用中的模式需要 8 ~ 10 个组件来实现特定功能，就会产生过多的开销。因为任何一个组件都要与其他组件进行通信，即使有些通信并不重要。在设计架构的时候，一切都构想得特别好，但是当模式实现了，初始的价值已经淡化，剩下就是性能问题。

接下来的例子中仅仅增加了一个组件就导致了通信开销呈指数级增长：

```
// component.js
import events from 'events.js';

// 一个通用组件。
export default class Component {
```

```
// 创建时，该组件触发一个事件。
// 同时还会添加一个该事件的监听器，并执行一些耗时操作。
constructor() {
  events.trigger('CreateComponent');
  events.listen('CreateComponent', () => {
    var i = 100000;
    while (--i) {
      for (let c = 0; c < 100; c++) {}
    }
  });
}

};

// main.js

import Component from 'component.js';

// 用来放置新增组件的数组。
var components = [];

// 单击add按钮时，就会创建一个新组件。
// 随着添加的组件越来越多，系统的整个等待时间会受到明显的影响。
// 一直单击按钮，足够长的时间后，浏览器标签会崩溃。

document.getElementById('add')
  .addEventListener('click', function() {
    console.clear();
    console.time('event overhead');
    components.push(new Component());
    console.timeEnd('event overhead');
    console.log('components', components.length);
  });
```

松耦合的组件比较好，因为关注点（功能）被分离了，我们能更自由地实现组件，同时还能避免破坏其他组件。组件耦合的方式形成了一种复用模式。但在初始实现后，随着软件日渐成熟，我们会发现以前很好的模式如今已经变得过重。因为组件的关注点现在已


经很好理解，不再需要过去那种“自由地实现”了。解决这个问题的办法就是改变接下来的模式，它相当于终极指示，会直接影响将来所有组件代码的实现方式。改变模式是通过移除不需要的组件来解决通信瓶颈的最好的方法。

分析代码

查看代码时凭直觉感受就能发现很多不必要的代码。我们在前面的小节中介绍过，这时选择合适的组件间的通信模式相当重要。我们能够直接看到逻辑设计层面有过多的组件，那么在运行时的执行层面该如何分析呢？

在开始重构代码、改变模式、移除组件之前，需要先分析代码，以便掌握代码运行时的性能特征，而不仅是了解它的表面特征。分析提供的信息能帮我们做出有效的优化决策。更重要的是，通过分析代码，能避免那些对用户体验影响甚微的微观优化，至少还能对需要处理的性能问题进行优先级排序。组件之间的通信开销很可能占较高优先级，因为它对用户有最切实的影响，并且是扩展应用的一个巨大障碍。

首先，我们可以用浏览器内置的分析工具。在与应用交互时，我们可以手动使用开发者工具界面来分析整个应用。这对于诊断 UI 中特定的响应问题十分有效。其次，可以写代码（其机制与浏览器内置分析机制一样）来分析更小的代码片段（比如独立的函数），这种方式能得出相同的结果。分析结果本质上是一个调用栈，同时还包含了在哪些地方消耗了多少 CPU 时间的分解结果。这一方式指明了需要改进的方向，我们能集中精力优化那些成本“昂贵”的代码。

 我们只介绍了分析 JavaScript 应用性能的一点皮毛。性能分析是一个巨大的话题，你可以在谷歌上搜索“Profiling JavaScript code”，这里有大量的优质资源。推荐一个入门资源：<https://developer.chrome.com/devtools/docs/cpu-profiling>。

接下来的例子展示了如何使用浏览器开发者工具来得到一个分析结论，这个分析比较了几个函数的性能：

```
// 吃掉一些CPU周期，调用其他函数来建立一个分析调用栈。  
function whileLoop() {
```

```
var i = 100000;

while (--i) {
    forLoop1(i);
    forLoop2(i);
}

// 吃掉一些CPU周期.....
function forLoop1(max) {
    for (var i = 0; i < max; i++) {
        i * i;
    }
}

// 吃掉更少的CPU周期.....
function forLoop2(max) {
    max /= 2;
    for (var i = 0; i < max; i++) {
        i * i;
    }
}

// 在开发者工具的profile标签创建分析数据。
console.profile('main');
whileLoop();
console.profileEnd('main');
// 1177.9ms 1.73% forLoop1
// 1343.2ms 1.98% forLoop2
```

除了浏览器之外还有其他分析 JavaScript 代码的工具,可以根据分析的目的进行选择。比如 benchmark.js 以及类似的工具可以用来衡量代码的原始性能 (raw performance), 分析结果会显示代码每秒执行的操作次数。这个方法真正的用处在于比较两个或两个以上的函数实现。它能够提供一个分解结果, 展示哪个函数是最快的, 快到什么程度。最终我们会发现这才是我们所需要的最重要的分析信息。

组件优化

前面已经解决了组件通信的性能瓶颈，现在进入组件内部，看看它们的实现细节和可能呈现的性能问题。比如，维护状态是 JavaScript 组件的一个公共需求，但是记录事务式的代码并不能很好地在性能方面进行扩展。我们还需要意识到，一些会改变其他组件所用数据的函数会带来一些副作用。最后，DOM 本身以及代码中跟 DOM 交互的部分，很可能出现无响应的情况。

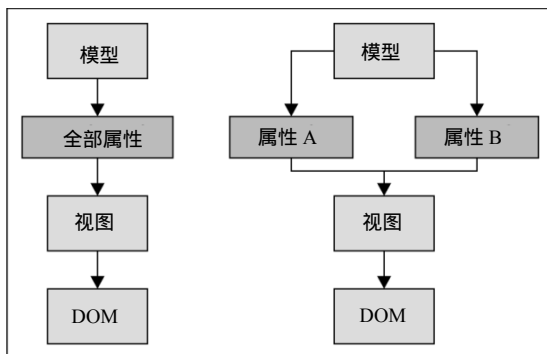
维护状态的组件

大部分组件都无可避免地需要维护状态。比如，如果组件由一个模型和一个视图组成，那么这个视图需要根据模型的状态决定什么时候进行渲染，还要引用 DOM 元素（直接引用或者是用选择器字符串引用），保证任何一个元素在任何时候都有状态。

所以状态实际上就是组件里的生命，这有什么重要意义吗？其实没有，真的。事实上我们需要写一些事件驱动的代码，用来响应状态的改变，响应的结果就是改变用户所看到的内容。当扩展规模的时候，问题就来了：在个别情况下，组件需要维护更多的状态，导致后端提供数据模型变得更加复杂，DOM 元素的数量也会增加。所有带状态的组件都会相互依赖，随着这样的系统不断扩大，会导致复杂性增加，还会影响性能。

所幸，框架帮我们处理了很多这种复杂性问题，还在很大程度上优化了这些改变状态的操作，因为框架给应用提供了很多基础方法。不同的框架使用不同的方法来处理组件的状态改变。有些框架使用了更自动化的方法，在监控状态改变时需要更多的成本。另外一些框架只在状态明显改变的情况下才进行处理，直接触发事件。这种方法对程序员要求更高，但是能减少成本。

在增加组件的数量和复杂度时，我们可以通过以下步骤来避免性能问题。首先，要确保只为有意义的事情维持状态。如果为一些永远都不会出现的状态设置处理函数，这无疑是一种浪费。同样，如果组件能改变状态并且触发事件，却从不更新到 UI 中，为之维持状态也是一种浪费。尽管这些问题很难排查，但如果避免掉这些隐藏的问题，就能避免将来与响应速度相关的扩展问题。



图注：视图可以对任何模型的属性变化做出一样的反应，或者可以针对特定的属性变化做出特定的响应。虚拟DOM视图帮我们自动完成这一处理过程。

处理副作用

在前面的小节中，我们看到了组件所维护的状态，以及疏忽之下会如何影响性能。那么这些状态的改变是怎么发生的呢？它们并不是自发出现的，只有代码显式地修改了一个变量值才会导致状态变化。这个叫做副作用，它会影响性能且无法避免。副作用会导致前面小节中所讲的状态的改变，而且如果不加注意的话，还会影响性能。

跟有副作用的函数相反的是纯函数。这些函数接受输入，返回结果。中间不会有改变状态的操作。这种函数也就是我们所说的引用透明（referential transparency），也就是说对于特定输入，不管调用多少次函数，都可以确保相同的输出。这种属性对于优化和并发都十分重要。比如说，如果我们总是能根据特定输入得到相同的结果，函数不管在哪里调用都不会影响其结果。

接下来考虑一下通用组件，它在应用内部被特定功能的组件公用。这些通用组件不太可能会维持状态，因为状态更多地存在于接近 DOM 的组件中。顶级组件里的函数适合实现成没有副作用的纯函数。作为一个经验法则，我们应该让状态和副作用尽可能地接近 DOM。

在第 4 章中可以看到，对于在一个复杂的发布/订阅事件系统里发生的事件，我们很难进行人工跟踪。在事件中，我们没有必要跟踪这些路径，但是在函数中，就截然相反了。如果函数改变了某些东西的状态，从而导致在系统中的其他地方出现了问题，我们很难跟踪这种类型的问题的根源。此外，使用无副作用的函数越多，所需的安全检查代码就越少。

我们经常看到这些代码被用来检查某个状态，而这种检查看上去毫无理由，唯一的理由就是让它生效。这种方式只会增加开发的成本。

下面的例子展示了一个有副作用的函数和一个没有副作用的函数之间的对比。

```
// 该方法改变了传进来的对象参数。
function withSideEffects(model) {
  if (model.state === 'running') {
    model.state = 'off';
  }

  return model;
}

// 相反，该方法不会引起副作用，
// 因为它会返回一个新的实例，而不是改变model。

function withoutSideEffects(model) {
  return Object.assign({}, model, model.state === 'off' ?
    { state: 'running' } : {});
}

var first = { state: 'running' },
    second = { state: 'off' },
    result;

// 我们会发现withSideEffects()会引起一些意想不到的副作用。
// 因为它改变了其他地方使用的对象的状态。
result = withSideEffects(first);
console.log('with side effects...');
console.log('original', first.state);
console.log('result', result.state);

// 通过创建一个新的对象withoutSideEffects(),
// 不会改变任何对象的状态。
// 不会在代码的任何地方引入副作用。
result = withoutSideEffects(second);
```



```
console.log('without side effects...');  
console.log('original', second.state);  
console.log('result', result.state);
```

DOM 渲染技术

更新 DOM 的代价很大，优化 DOM 更新最好的方式就是不要更新。换言之，要尽可能少地更新。扩展应用带来的问题是 DOM 操作更加频繁，甚至超出了需要。而且还要监控更多的状态，还要更频繁地通知用户。即便如此，除了框架所使用的技术，我们还能做一些事情来减轻 DOM 更新所带来的负担。

那么，相对页面中简单运行的 JavaScript，为什么 DOM 更新的代价会这么高呢？因为计算展现样式的过程会吃掉很多 CPU 周期。我们可以先减轻浏览器渲染引擎的负担，然后提升 UI 的响应速度，在组件视图里，使用对渲染引擎依赖更小的技术。

举个例子，重排是一种渲染事件，它会导致整体的计算。本质上，在元素发生改变时才会重排，但这会导致附近的元素也发生改变。整个过程会贯穿整个 DOM，一个看起来不起眼的 DOM 操作可能会产生很大的开销。现代浏览器的渲染引擎很快，我们可以容忍在 DOM 代码里出现一点点失误，即使这样 UI 也能表现得很完美。但是当增加了新的动态的部分时，DOM 渲染技术的扩展性就至关重要了。

因此首先要考虑的策略是，搞清楚哪些视图更新会导致重排。比如，改变元素内容的操作非常不起眼，看起来永远不会导致性能问题。而往页面插入新的元素，或者改变已有元素的样式来响应用户进行交互时，则有可能导致响应问题。



当前流行一种 DOM 渲染技术，虚拟 DOM (virtual DOM)。ReactJS 以及一些类似的库就利用了这一概念。主要思路是代码只需要将内容渲染到 DOM 中即可，即使初次渲染整个组件也是如此。虚拟 DOM 拦截了这些渲染请求，计算出已经渲染的和将要发生改变部分的区别。之所以称为虚拟 DOM 是因为代表 DOM 的数据存储在 JavaScript 内存中，可以与真实 DOM 进行比较，只在需要时才会操作真实 DOM。这种抽象的方式可以提供一些有趣的优化，同时保持视图代码的简洁。

将更新一个接一个地发给 DOM 也不是很理想。因为 DOM 会收到一系列变化清单，然后一个一个顺序响应。对于复杂的 DOM 更新，有可能会触发一个又一个的重排。最好是把将要更新的 DOM 分离出来，更新后，再重新添加到 DOM 树中。当重新添加元素后，昂贵的重排计算就能一次性完成，而不是一连串地重排好几次。

然而，有时候问题并不出在 DOM 本身，而是出在 JavaScript 单线程的特性上。当组件的 JavaScript 在运行时，DOM 并没有机会去渲染任何待更新的内容。如果 UI 在某些情况下不响应了，最好是设置一个定时器让 DOM 更新。这样也能让等待中的 DOM 事件有机会执行。在 JavaScript 代码运行过程中用户试图进行一些交互时，这是至关重要的。

接下来的例子展示了如何在 CPU 密集的计算过程中，延迟运行中的 JavaScript 代码，从而让 DOM 有机会更新：

```
// 设置了定时器后，调用传进来的func。
// 这能够“延迟”调用，到下个合适的时机再执行
function defer(func, ...args) {
    setTimeout(function() {
        func(...args[0]);
    }, 1);
}

// 执行一些耗时操作.....
function work() {

    var i = 100000;
    while (--i) {
        for (let c = 0; c < 100; c++) {
            i * c;
        }
    }
}

function iterate(coll=[], pos=0) {
    // 吃掉一些CPU周期.....
```

```
work();

// 在DOM中更新进度.....
document.getElementById('progress').textContent =
    Math.round(pos / coll.length * 100) + '%';

// 延迟下一个对iterate()的调用，给DOM一个机会来展现更新的百分比。
if (++pos < coll.length) {
    defer(iterate, [ coll, pos ]);
}

}

iterate(new Array(1000).fill(true));
```

对于耗时的 JavaScript 代码来说，Web Worker 是另一种解决方案。因为它们不操作 DOM，所以不会干涉 DOM 的响应速度。不过本书不讨论这个技术。

API 数据

随着应用规模扩大，还有一个重要的性能问题，它就是应用数据本身。这一部分需要我们格外小心，因为运行时有很多可扩展性的影响因素。虽然更多的功能并不意味着更多的数据，但事实往往如此：更多的功能产生了更多的数据类型和更大的数据量。其中数据量主要随着用户数的增加而膨胀。作为 JavaScript 架构师，我们需要扩展应用，使应用在浏览器端能够应对日益增长的加载时间和数据规模。

加载延迟

也许扩展应用性能最大的困难在于数据本身。应用数据随着时间的推移而发生改变和进化在某种程度上是一种普遍现象。前端增加新功能必然会影响数据的体积，但是 JavaScript 代码无法控制用户数量或者用户与软件的交互方式。因此会导致数据激增，假设前端程序无法应对这种情况，程序最终就会停下来。

在数据加载过程中，前端工程师面临的问题是没有数据能展示给用户。我们只能采取必要的措施，提供一个可接受的加载体验。这就带来一个问题：当数据加载时，是用加载

消息挡住整个屏幕还是只在等待数据的元素区域显示加载消息？第一种方式基本避免了用户执行不允许的操作，因为我们阻止了用户与 UI 的交互。但如果使用第二种方式，那么当存在大量网络请求时，就必须得考虑用户与 UI 之间的交互。

其实以上两种方式都不太理想，因为在数据加载的时候，应用的响应速度基本都会被限制。由于我们不希望完全阻止用户与 UI 交互，所以可能要对数据加载有严格的超时要求。这样做的好处是即便告诉用户后端超时，也能保证响应速度。但是有时候用户必须等待一些操作完成，那么设置超时就会产生问题。这种情况下，糟糕的用户体验（等待）反而更好，而不是在无意中造成更坏的体验。

前端需要做两件事来帮助扩展后端数据。首先，要尽可能缓存响应。这能减少后端负担，缓存数据还能提升客户端的响应速度，因为可以省掉一个请求。为了不缓存陈旧的数据，我们需要引进合适的失效机制。Web socket 就很不错，虽然它只是通知前端会话某个实体类型发生了改变，但这样能清空缓存。第二件事就是减少每个请求加载的数据量。大部分 API 方法都能接收参数，限制结果数量。这个数量要在合理范围内，这样才能决定用户优先看到的内容，然后围绕这部分内容进行设计。

处理大数据集

在前面的小节中介绍了应用数据相关的可扩展性问题。当应用扩大时，数据也会增长，加载数据也会面临挑战。把数据加载到浏览器后，还得处理比以往更大的数据量，这可能会导致用户交互无响应。比如，假设一个集合有 1000 条记录，某个事件将这个集合传给多个组件进行处理，用户体验就会受到影响。因此我们需要借助工具，将庞大而难以扩展的数据过滤成最有用的数据。

此时需要借助底层公用函数库，对大数据集进行复杂的变换。大型框架可能提供了类似的工具，它们可能使用了底层的公共库。对数据执行的变换具备映射—化简（map-reduce）多样性，这是一个抽象模式。一些功能型程序库，如 Underscore/lodash，提供了很多这种模式的变体。它如何帮助我们在大数据集的情况下进行扩展呢？借助这些库，我们可以编写简洁可复用的映射和化简函数，同时将大部分的优化工作留给这些库来完成。



理想情况下,应用只会加载当前页所需的数据。但是在大多数情况下,这几乎是不可能的,API 无法计算出功能所需的每个可能的查询场景。因此用 API 大致过滤数据后,等数据到达前端时,组件会用更加精确的标准来过滤数据。

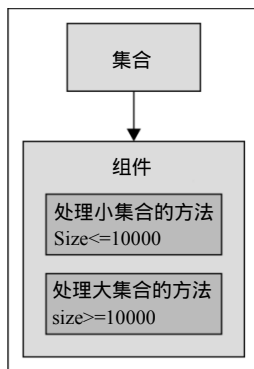
这样可能会造成一定的困惑。到底哪些数据是后端过滤的,哪些是前端过滤的?假设一个组件较多地依赖于后端 API 过滤,而其他组件更多地在前端进行过滤,开发者就会感到困惑,代码也会不直观。即使只是稍微改变了 API,组件使用的数据也会发生变化,甚至可能导致产生不可预测的 bug。

花在映射和化简上的时间越少,用户就会觉得 UI 响应越及时。因此要尽早地加载用户看得见的数据。比如说,当 API 数据到达时不要马上在事件中将数据分发。我们需要构造组件通信,使它尽快对大数据集进行计算上耗时的过滤。这样能够减轻所有组件的负载,因为它们现在处理的集合更小了。组件所处理的数据变少后,扩展更多的组件也不会有什么影响。

优化运行时组件

我们应该针对通用场景优化代码。因为随着功能和用户数增长,通用场景会增长,这不是边缘情况。不过我们经常需要处理两个同等通用的场景。设想一下,部署软件到几个客户环境后,经过一段时间,功能进化到满足客户需求时,任意一个功能就可能有两三个通用场景。

假设有两个函数能处理某个通用场景,那么必须明确运行时要使用哪个函数。这些通用场景都是非常粗粒度的。举个例子,某个通用场景可能是“集合很大”或者“集合很小”,检查这些条件成本并不大。那么假如我们对通用场景变化做了适配,软件将会比未适配时更具有响应性。比如,假设集合很大,函数就能采取其他方法来过滤它。



图注：一个组件能够在运行时基于广泛的分类范围（如较小或较大的数据集）改变其行为。

小结

对于用户，响应速度是衡量应用质量的一个重要指标。无响应的用户界面会让用户抓狂，并且这个应用对我们来说就没有进一步扩展的可能了。应用的初次加载是应用给用户的第一印象，同时也是最难提升速度的部分。我们探讨了加载所有资源带来的挑战，这需要模块、依赖，还有构建工具的共同作用。

组件之间的通信瓶颈则是影响 JavaScript 应用响应速度的另一个主要障碍。这些瓶颈通常源于过多的间接通信，还有为了实现特定功能而采取的事件设计方式。组件本身也会造成响应速度的瓶颈，因为 JavaScript 是单线程的。我们在这个范围内探讨了一些潜在的问题，包括维护状态以及处理副作用的成本。

API 数据是用户关心的内容，因此只有数据加载完毕，用户体验才有保障。我们讨论了扩展的 API 及其数据会造成哪些可扩展性问题。当数据加载完成后，组件需要能迅速地映射和化简这些数据，因为数据集会随着规模扩展而继续增长。现在我们已经知道很多提高架构性能的方法了，接下来看一下在各种环境下如何保证架构的测试以及运行能力。

8

可移植性和测试

Web 应用在短短的几年时间内，已经有了巨大的发展。人们已经不再像打补丁一样随意地在网页中嵌入 JavaScript。如今，我们通过 JavaScript 构建应用，请注意，我们不单单是构建应用，还要保证其可扩展。因此，在设计架构时，需要充分考虑到可移植性，以及可替换性。

可移植性与可测试性往往相伴随。编写大规模的 JavaScript 代码时，不能对后端作任何假设。这就意味着，应用需要具备脱离后端正常运作的能力。本章介绍可移植性与可测试性这两个联系紧密的主题，以及它们对可扩展性的影响。

与后端解耦

NodeJS 已经印证了 JavaScript 并不是只能用于网页编程。运行 JavaScript 不需要完整的浏览器环境，只需要 V8 JavaScript 引擎即可。虽然 Node 是一个后端服务器环境，但它仍然展示了 JavaScript 有多强大。同样，前端代码也应具备可移植性，能与任何后端架构完美协作。

本节将会解释为什么应该将前端的 JavaScript 代码与后端的 API 解耦；接着，会介绍如何模拟 API，完全解除对后端的依赖。

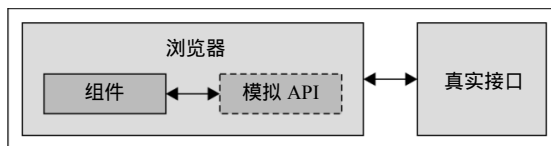
模拟后端 API

开发大型的 JavaScript 应用时，也需要搭建后端架构。那么问题来了，为何要将前端

代码与后端分离，使之不依赖于后端呢？为了提高程序的可扩展性，我们通常会保持组件之间松散耦合。同样地，将 Web 应用的前端与后端环境解耦，也能提高程序的可扩展性。即使后端的 API 不变，也不代表构建 API 的方法和框架不会改变。去除前端对后端的依赖还有其他的好处，例如，我们可以只修改 UI，而不修改系统的其他部分。但是，模拟后端 API 最大的优点主要体现在开发和测试方面。我们可以简单地模拟几个请求结果，来测试程序。

不管你喜不喜欢，开发的过程有时感觉像是在编写示例。在开发的途中，需要向利益相关者展示我们的成果。使用模拟数据会让我们更有信心，而不至于绝望。有了模拟数据，演示只是小菜一碟。当然，我们要在经理的心目中保持英勇的程序员形象。

模拟数据这么厉害，它就没有缺点吗？模拟数据跟程序中其他部分一样，也是需要维护的，而且总是伴随着风险。例如，当模拟 API 与真实 API 不再同步，或者人们分不清 UI 本来的功能和伪造的结果时，模拟 API 便失去了它的价值。为了避开这些风险，需要制定一套合适的流程来设计和实现我们的功能。以下是一个小例子：



图注：模拟数据独立于与真实API通信的组件之外，因此移除模拟API对组件毫无影响。

前端入口

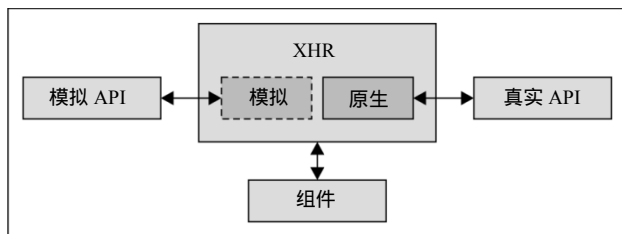
我们需要在前端与后端的对接之处设计开关，以便在真实数据和模拟数据之间切换。开关也可能处在服务端，在这种情况下，我们仍然会发送 HTTP 请求，只不过响应请求的并不是真实的服务器。另一种方式是将开关设立在浏览器端，HTTP 请求均会被模拟库的处理器截获，并不会发送到服务器。

在这两种模拟方式中，我们都努力尝试在前端与后端之间建立一个对接口。这就是使前端能够独立于后端的关键。生产环境中，前端与后端可以紧密联系，毕竟前后端本来就是相互依赖的。但是在开发阶段，能够控制组件如何请求 API，是提高可扩展性的关键。

有时候可以直接使用模型和集合创建模拟数据模块。例如，假设我们正处在模拟模式

中，可以引入这个模块来使用模拟数据。这种方式的问题是应用知道自己并不是真正在与后端通信。我们不希望如此，因为我們希望在切换到生产环境时无须修改代码。否则，将面临手动初始化模拟数据这样的麻烦事，我们应该尽可能地避免这种情况。

不管使用哪一种模拟机制，我们都应该将其模块化。换句话说，模拟代码应该能被快速关闭和完全移除。在生产环境中，不应该有模拟数据。这样在服务端实现模块化的模拟数据更为简单。如果是在浏览器端模拟数据，那么在建构应用时，我们需要通过一些选项将它们移除。本章将会介绍更多相关的构建工具。



浏览器端的 API 模拟器，会在 XHR 层拦截请求。如果应用中开启了数据模拟，那么它会寻找与请求匹配的模拟 API。在应用关闭了数据模拟后，原生的 HTTP 请求就会像往常一样工作。

模拟工具

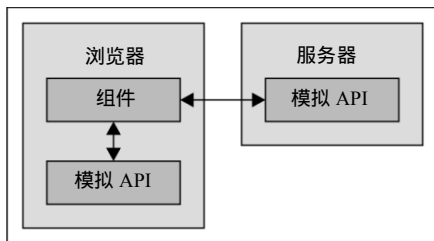
前面的部分提到了两种模拟后端 API 的方法，主要有两种模拟后端 API 的实现方式。第一种方法是在应用中引入类似 Mockjax 的库，来拦截 XHR 请求。第二种方法是搭建一个真实的 HTTP 服务器，但它并不会操作真实的应用数据——它会像 Mockjax 一样提供模拟数据。

Mockjax 的实现方式既简单又巧妙。它假设应用使用 jQuery 的 `ajax()` 方法来发送 HTTP 请求。这是一个合理的假设，因为大部分的框架都会使用 jQuery。当 Mockjax 被调用时，它会将 jQuery 的一些核心 XHR 方法重写。应用不管什么时候发送 XHR 请求，都会调用这些重写的方法。它会检查是否有路由规则匹配当前请求的 URI，如果有，则会执行相应的处理器。否则，它会跳过并尝试给后端发送一个真正的请求——当我们希望结合真实的 API 请求和模拟请求时，这个策略非常有用。之后的内容将会深入讨论这一点。

任何的处理器都能像真实 API 一样，返回 JSON 或其他格式的数据。模拟的关键在于

核心代码——发出请求的模型和集合——无须为 Mockjax 做出任何修改，这是因为 Mockjax 运作在较低层。在切换到使用真实数据的生产环境时，这些模型和集合都无须修改，只须移除调用 Mockjax 的模块，便可获得真实数据。

使用模拟 Web 服务器的技术，也可以达到相同的目标——部署时无须修改代码。它的原理实际上也是劫持 XHR 请求，只不过是在不同的地方完成。采用这种方式的主要好处是，我们在部署时无须做任何修改。处理请求的服务器有模拟服务器和真实服务器，但是在生产环境中，不太可能运行模拟服务器。采用这种方式的缺点是，我们需要运行一个额外的服务器，虽说这个要求不高，但毕竟多了一步，同时也略微降低了可移植性。例如，当我们把模拟环境的代码打包并发给别人时，我们更希望代码不依赖于 Web 服务器，因为如果代码不依赖于 Web 服务器，那么它可以在别人的浏览器上被完整地演示。



图注：在浏览器端或服务端模拟API。这两种实现方式都能达到相同的效果：我们的代码在使用模拟数据时和使用真实数据时都是相同的。

生成模拟数据集

在了解了几种模拟 API 的方法后，我们需要获得模拟数据。假设 API 返回的数据格式是 JSON，那么可以将模拟数据存储在 JSON 文件中。接下来，在模拟模块中引入这些依赖的 JSON 模块，并将它们作为模拟处理器的数据源。但是这些数据的内容从何而来呢？

在刚开始构建模拟数据时，通常已经有了可使用的 API。可以观察并参考真实 API 返回的数据，来组织模拟数据。如果有 API 文档，那么这个过程会更简单，因为可以了解不同实体的不同领域有哪些可取的值。但有时从哪里开始生成模拟数据并不明确。之后的功能设计过程部分中将会探讨这个问题。

手动创建模拟数据的好处是，可以保证数据的准确性。也就是说，我们不希望创造出无法反映真实数据的模拟数据，因为这违背了我们的最终目的。况且，保持与 API 同步也

是一个很大的难题。使用工具来自动生成模拟数据集是一个不错的办法，它只需要知道一个特定实体的结构，便可完成剩余的工作，例如接收参数和生成随机值。

另一种好用的模拟数据生成工具可以提取特定环境中真实的 API 数据，并保存到 JSON 文件中。假设代码在准生产环境中暴露出了一些问题，我们可以在该环境中使用这个工具提取出我们需要的模拟数据。采用这个实现方式可以或多或少地避免修改准生产环境中的代码，因为我们在诊断过程中对模拟数据的修改，都存在于内存中，可以被轻松地抹除。

执行操作

模拟 API 的一个难点是执行操作。这是 GET 以外的请求，通常需要改变一些资源的状态。例如，修改一个资源的属性，或移除一个资源。我们最好为处理器编写公用的操作处理逻辑，因为真实 API 在处理操作时会遵循通用的模式。

API 操作工作流的复杂度决定了实现模拟 API 的难度。修改某个资源的属性并返回成功状态码 200 是一个简单的操作例子。但是，应用的工作流通常都更为复杂，例如，某些操作需要较长时间。举个例子，某个操作会创建一个新资源并返回资源 ID，因此我们需要监控操作状态。我们已经在前端代码中检测了操作状态，因为与真实 API 通信时，也需要这么做。因此，此时的关键问题是如何根据应用的需求，模拟合适的数据。

这些操作可能很混乱，执行速度很快，尤其是当应用的规模变大，并拥有很多实体类型和操作的时候。我们应该争取以最小的代价模拟这些操作，而不要试图详细地模拟应用执行操作的每一步，因为这对可扩展性是没有帮助的。

功能设计过程

我们模拟 API 不是为了好玩，而是为了协助开发功能。考虑到将来可能要模拟庞大的 API 集合，因此，我们需要一个控制任务顺序的流程。例如，在开始实现一个功能之前，我们是否需要等待某个 API 就绪？虽然设计 API 需要时间，并且会关联到很多人，但如果我们可以模拟 API，那就不需要等待了。

在这一部分，我们将探讨模拟数据的必要步骤，以确保开发功能过程中代码的可扩展性。

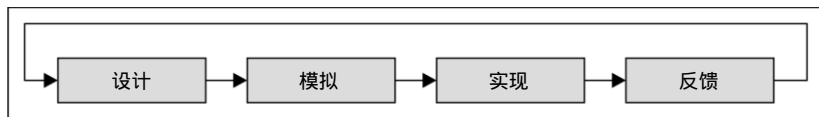
设计 API

有一些通用 API 可以支持多个功能，它们都是应用的关键。通常情况下，会有一部分 API 至关重要，并被大部分功能使用。另一方面，大部分新功能的开发都需要我们扩展原有的一个或多个 API。这与后端组织资源的方式有关，并且涉及到了设计层面的工作。

当我们试图开发新功能时，会面临一个问题，即开发新的 API 可能需要很长的时间。但如果我们必须等待 API 就绪才能开始开发前端功能，那么新功能就会被延误。这并不理想，因为我们希望趁任务还新鲜时就开始着手。如果一项工作被列入了积压的任务列表，那么它可能永远都会待在那儿。使用模拟 API 可以让我们避免延误新功能，这对开发的可扩展性至关重要。

当开始模拟新的 API 时，我们就进入了一个新的领域。这意味着可能要考虑前端以外的开发人员。我们可能会，也可能不会修改真实的 API——这由我们团队的构成决定。也就是说，不管谁是专家，都必须知道我们是如何设计 API 的，并且可以提出建议、更改，等等。我们没必要往死胡同里钻。另一种方式是让后端程序员提供 API 文档。设计 API 时，我们需要站在大局考虑，设计时只须考虑重要的 API 和它们的必要属性和操作，其他细节均可在模拟数据和真实代码中轻松调整。

在使用真实 API 之前使用模拟 API 来实现功能，有助于避免严重的错误。例如，假设我们借助模拟数据在前端实现并演示了某些功能，其他具有专业后端领域知识的程序员就有了一个检查功能可行性的机会，避免我们在未来面临更严重的问题。



图注：设计模拟API，并借助模拟API实现功能的周期。

实现模拟数据

现在，让我们来实现一个新功能，第一步是实现模拟 API，来支持前端代码的开发。正如之前所提到的，应该与最终要实现真实 API 的工程师保持联系，首先要在较高的层面上设计 API 的核心部分，接着在实现真实 API 的过程中调整其他部分。

但是，在开发模拟数据的过程中，不用总是依赖 API 小组。前端组件很可能已经使用了某些 API。也就是说，在这些已有的 API 中，很可能已经有可供遵循的固定模式。举一个典型的例子，模拟数据可能碰巧是一个遗漏的普通实体类型，我们完全可以参照已有的模拟数据进行补充。遵循一个良好的模式意味着一个好的开端，因为这降低了未来发生大的改动的几率。

在确定了模拟 API 的结构和功能后，需要添加模拟数据。假设已经在使用工具自动为某些 API 生成模拟数据，那么此时应该想办法扩展它，或者手动创建一些测试实体。我们应该用最少量的数据来证明实现的可行性，而非预先花费很多时间去输入数据。



不必总是先准备好模拟 API 再创建数据，我们还可以从数据开始着手——设计正确的实体结构，而非沉浸在 API 的实现细节。因为数据总归要被存储在某个地方，这是非常重要的一点。从数据着手可以让我们换一种思维方式，从而选择一种最适合手头任务的实现方式。

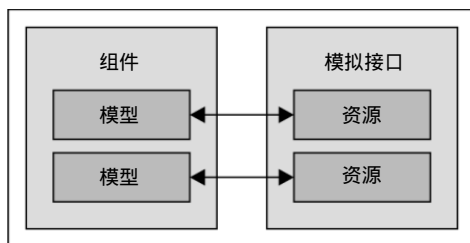
我们并不总是要实现一个全新的模拟 API。也就是说，模拟的 API 可能已经存在了，或者正在被实现。在这种情况下，模拟数据会更简单。因为通常情况下，可以向 API 的开发者请求示例数据或帮助来实现模拟数据。记住，如果想要提高可移植性，那必须具备将前端与后端分离的能力，这意味着需要模拟完整的 API。

实现功能

现在模拟 API 已经准备好了，是时候好好利用它了。模拟 API 不是固定不变的——它总是在不断地调整，但我们完全可以借助它来开发前端代码。很快地，就会发现一些问题：可能是 API，也可能是与 API 交互的组件有问题。我们不能因为这些问题而灰心丧气，因为这就是我们的目的——提前发现问题。如果没有模拟接口，就无法做到这点。

如果 API 是普遍适用的，并且组件能正常工作，那么可以开始寻找设计的性能瓶颈了。如果是通过工具来生成模拟数据的，那么这项任务将尤为简单，因为创建 100,000 个实体来测试代码并不是难事。为了解决性能问题，有时只需要一些微小的调整，但有时几乎需要修改整个实现方式。重要的是我们应尽早发现这些问题。

如果没有模拟数据，那么还有一个工作是难以完成的，那就是演示代码。当代码深度依赖于大量的后端环境时，演示是很难进行的。如果只需要几分钟就能开启某个功能并进行演示，那我们便可以充满信心地展示成果。也许我们目前所做的是错误的，也许利益相关者在看到演示时想起了他们之前遗漏的地方。这样，通过早期的、接连不断的反馈，模拟数据帮助我们扩展了功能开发生命周期。



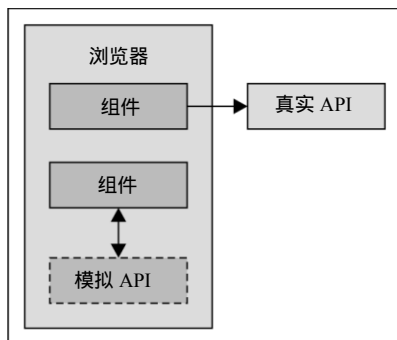
图注：开发中的组件内部，正在与模拟API通信。

协调模拟数据与真实数据

在完成功能开发后，我们需要切换到真实 API，切换时要做的工作取决于真实 API 的情况。例如，如果模拟的 API 已经存在了一段时间，那么只要模拟 API 与真实 API 高度统一，就可以假定无须采取任何操作。但是，如果我们模拟的是一个新的 API，那么它很有可能发生改变。所以必须捕获这些改变，并确保模拟接口与后续的更新保持一致。

这是模拟数据时难以扩展，让人烦恼的一部分。有很多因素可能会导致模拟 API 与真实 API 不一致，持续更新模拟 API 是一个令人望而生畏的任务。如果拥有生成模拟 API 的工具，就会简单很多。甚至可以基于 API 小组撰写的文档来生成所有的模拟 API。不过还有一个问题，虽然我们可以自动生成模拟 API，但需要先编写文档。因此，最好的方法是开发一个工具来自动生成模拟 API，而且由应用自身处理请求。只要没有太多重复的工作，并且 API 拥有一个合理的设计模式，应该就可以保持模拟 API 的同步。

另一个可行的优化，是在开启部分模拟 API 的同时，关闭其他的模拟 API，让模拟 API 的粒度可以被调整，而不是所有模拟 API 只能被统一开启或关闭。例如，当我们试图定位应用中的一个问题，并且需要依靠某个 API 返回特定的结果时，这项能力便可以发挥作用。我们可以在类似 Mockjax 这样的库中完成这个功能，当请求没有匹配的模拟接口时，会被交由原生的 XHR 对象直接处理。



图注：一个组件使用了模拟API，而另一个组件使用了真实API。

单元测试工具

在介绍了模拟 API 的基本知识后，是时候将注意力转移到测试上了。模拟 API 的能力与测试的能力是紧切相关的。因为我们可以针对相同的模拟数据进行测试。这意味着如果我们的测试失败，就可以使用导致测试失败的相同数据，来操作 UI 并定位问题。

在之后的内容中，我们将认识与 JavaScript 框架共同协作的单元测试工具以及它们的作用。我们也将寻找更加通用、独立、能检查任何代码的测试框架。在这一部分的末尾，我们将会介绍如何实现测试自动化，如何使测试自动化与开发工作流协作。

框架自带的工具

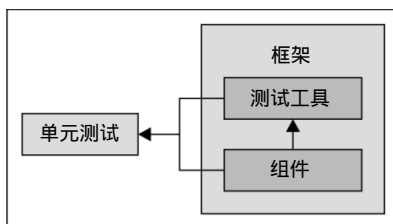
如果使用的是一个大型、无所不包的 JavaScript 应用框架，那么这个框架很可能自带单元测试工具。这里并不是提倡大家使用与框架无关的工具来代替已有的单元测试工具。这些工具当然是用来增强它们的——为编写框架中的测试用例提供专业的支持。

这减少了编写单元测试代码的工作量。如果按照框架的规范进行开发，那么框架自带的很多单元测试工具都可以开始工作了。例如，如果它知道我们将使用什么类型的组件来实现功能，就可以代替我们完成部分测试。这帮了我们一个大忙，我们不需要自己去重复测试。这些工具最终还可以帮助我们达到更高的测试覆盖率。

框架自带的测试工具除了提供测试框架外，还提供了实用函数。供我们在自己的测试用例中使用。这意味着我们需要维护的单元测试代码更少了。但只有当框架了解我们在测

试用例中会需要什么功能时，才能将它们提取，并以实用函数的形式提供给我们使用。

使用框架自带的测试工具的问题是，产品将会与特定框架有紧密的关系。但这不太可能成为一个问题，因为一旦选择了一个框架，我们将坚持使用它。好吧，在现在这个疯狂的 JavaScript 生态圈中，也许不是这样。达到优秀的可移植性要求架构高度敏捷，这意味着我们需要能适应改变。也许这就是为什么现在大部分的项目都减少了对大型框架的依赖，而增加了库的组合使用。



图注：单元测试与框架中的组件和单元测试工具有紧密的联系。

大型 JavaScript 应用中有很多异步代码，在单元测试中不应该忽略。例如，我们必须确保模型能够正确地获取数据和执行操作。这类函数返回的值是 promise 对象，要确保它们像预期一样成功或失败。



如果我们可以模拟 API，测试会简单很多。在浏览器端或服务端模拟都可以，因为代码仍然会将它们看做异步操作。有时我们也许会考虑模拟 Web Socket。在浏览器端模拟 Web Socket 有点难度，因为需要重写浏览器内置的 Web Socket 类。如果在服务器进行模拟，可以使用一个真实的 Web Socket 进行测试。

不管使用哪种方式，模拟 Web Socket 都是个难题，因为需要模拟 Web Socket 响应并回复消息的逻辑，例如，模拟一个 API 操作。但是，在完成更多的基础测试之后，仍然可以模拟 Web Socket，因为如果我们的应用依赖于它们，那我们必须使它们的测试自动化。

独立的单元测试工具

另外一种单元测试工具使用的是独立的框架，不依赖于特定的 JavaScript 应用框架或

库。Jasmine 是这类工具的标准，它为我们提供了一个简单明了的方法来声明测试规范。它创造性地在浏览器中运行了一个测试执行器，在测试通过或失败时就会展示一个工整的结果。

其他大部分独立单元测试工具都是在 Jasmine 的基础上进行扩展，提供额外的能力。例如，Jest 项目从本质上来说，就是一个拥有模块加载和内置模拟等额外功能的 Jasmine。再次强调，这些工具是不依赖于框架的，它们只专注于测试。使用这类独立的工具有助于提高可移植性。因为即使决定使用不同的技术方案实现应用，测试程序也仍然有效，有助于我们平滑过渡。

Jasmine 并不是唯一的标准，但它是最为通用和灵活。Qunit 在很久之前就出现了，可以被应用于任何框架，但它在最初是为 jQuery 项目设计的。当现成的测试工具太过庞大、不够灵活，或输出不够好时，我们甚至有想过自己开发测试工具。但这绝对不是我们希望发生的。我们通常会将单元测试加入到自动化的任务队列中，而不是心血来潮跑去执行单元测试。



有些代码相对容易测试。将它们分解为测试用例的难度取决于构建组件的方式。例如，当代码中有很多移动部件和副作用时，就需要编写大量的测试用例以达到足够高的测试覆盖率。如果代码耦合松散，副作用很少，那么编写测试用例会简单很多。

我们并不总能写出易测试的代码，因此有时牺牲覆盖率反而会更好。我们不希望为了创建更多测试用例，而编写重复的代码，甚至修改当前合适的架构。只有当组件已经足够庞大，并且需要更高的测试覆盖率时，才应该这么做。如果真的到了这个时候，也应该重新思考架构的设计了。优质的代码天生易于测试。

工具链和自动化

在持续的开发过程中，随着应用变得越来越庞大和复杂，很多任务需要改为“线下”进行。执行单元测试只是其中一个需要被自动化的任务。例如，在执行单元测试之前，通常会使用工具检查代码的合法性，以避免粗心导致的错误。通过测试后，需要编译组件，使它们能够在运行环境中工作。如果我们还编写模拟数据，这也会成为自动化流程中的一部分。

我们应该拥有一个可以自动执行以上所有任务的工具链。这些任务通常是大型任务的一小步，比如 `build production` 或 `build develop`。大型任务其实是由小型任务组合而成的。这个设计很灵活，因为**工具链**可以管理任务的执行顺序，或单独执行某个任务。例如，我们可以单独执行测试任务。

最受欢迎的工具链是 Grunt 任务执行器。其他类似的工具，例如 Gulp，也吸引了越来越多的关注。这些工具的一大好处是它们拥有丰富的插件——只需提供插件的配置便可定义好任务，并组织成为更大的任务。我们可以轻松地配置好自动化工具链——几乎不用自己实现代码。如果没有工具链，就难以享受来自这么多贡献者的资源。

使用工具链将任务自动化的另一个好处是可以实时修改编译文件的类型。例如，在开发某个功能时，我们并不想针对每一个改变都重新编译因为这么做会大大地降低效率。所以工具最好能直接处理原始模块，这样还有助于调试。在到达开发的尾声时，就会开始编译和测试。可以对原始代码和编译后的代码都进行单元测试——因为我们永远不知道编译后会出现什么问题。

测试模拟场景

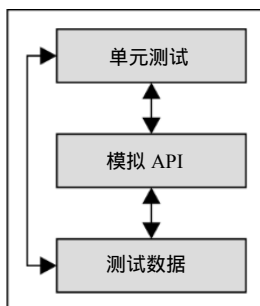
随着应用的扩展，我们需要处理的场景也越来越多。因为我们的用户数量、功能数量和代码复杂度都在提升。使用模拟数据和进行单元测试都有助于测试这些场景。在这一部分，将介绍几种创建模拟场景和测试场景的方法，会使用单元测试并假扮成用户与应用进行交互。

模拟 API 和测试固件

模拟数据的价值体现在很多方面，其中一面就是单元测试。如果使用模拟 API，可以像在使用真实 API 时一样执行测试用例。我们可以精确地控制模拟数据的内容，并按自己的想法改变它。这是一个沙箱，对外界没有副作用。即使是通过工具来生成模拟数据，我们仍然可以控制它们。

一些单元测试工具支持夹具（`fixture`），用来初始化测试时需要的资源和数据。这些数据与使用 Mockjax 模拟 API 时使用的数据没有很大的区别。主要的不同是我们很少在单元测试框架以外的地方使用它们。

我们可否在测试和模拟 API 时都使用一样的数据呢？例如，假设我们想使用单元测试框架自带的夹具数据功能，但如果不提供模拟数据，便无法使用它进行自动化测试。另一方面，模拟 API 有助于开发、测试功能和与后端解耦，等等。因此，完全可以为单元测试和模拟 API 提供测试数据。这样，便可以使用之前创建的模拟数据生成器，为测试和交互生成相应的场景。



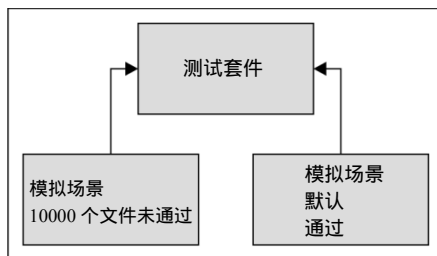
图注：单元测试可以请求模拟API，或者直接使用测试数据。如果它们使用的是同一份数据，那么当测试不通过时，就可以更轻松定位问题。

场景生成工具

随着时间的推移，我们将积累越来越多的功能和使用场景。因此，在工具链中加入自动生成模拟数据的工具，会帮我们一个大忙。如果想更进一步，这些工具最好是可配置的。即使这些配置是粗粒度的，通常情况下也足够生成我们需要的场景数据。

模拟的场景数据之间不会相差太大。值得提到的是，我们需要一组基础数据，因此当发现问题时，我们可以思考：这组数据有什么不同？如果可以通过工具生成大量场景数据，我们要确保拥有“黄金”数据集，即保证程序正常执行的数据。

我们通常会修改黄金模拟数据集合的实体数量。例如，如果我们想测试某个页面的某项功能，就可以在模拟数据中创建一百万个实体，来看看会发生什么。这个页面完全崩溃了——在后续的调查中发现，出现崩溃是因为一个 `reduce` 函数在执行加法操作的时候超出了整型的最大值。通过场景测试，可以找出类似的一些有趣的错误。即使场景看似牵强或者不太可能出现，仍然应该修复它，因为它肯定会被其他较常见的场景触发。



图注：修改场景可能会导致测试不通过，通常我们会通过扩大场景范围来定位问题。

我们可以模拟的场景非常之多。例如，我们可以通过删除实体属性来破坏一些数据，以确保前端组件定义了属性的默认值，或者能够优雅地报错。第二点实际上非常重要，因为随着 JavaScript 代码的扩展，会有越来越多无法修复的场景，我们需要确保相应的错误处理方式是可行的。

端到端测试和持续集成

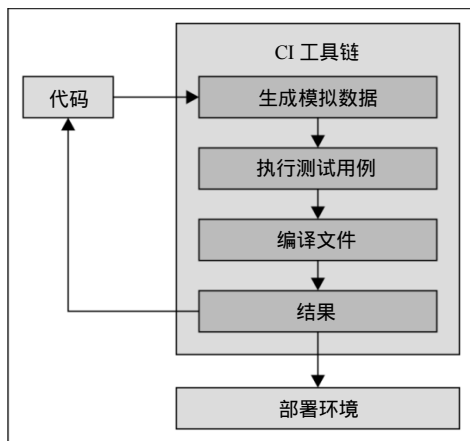
最后一步是对功能进行端到端测试，并将其添加到持续集成的过程中。单元测试是一回事，当代码通过单元测试后，我们可以确信组件是坚固可靠的。而端到端测试则是另一回事，它就像用户一样与 UI 交互。例如，我们实现的某些功能中可能就包含类似的用例，端到端测试就是为这种场景设计的。

Selemium 这类工具可以将端到端测试自动化。它们会将用户的操作记录下来，我们可以控制它重复地进行相同的操作。例如，一个端到端测试可能包含了对某个资源的创建、修改和删除操作。这类工具知道操作成功时程序该如何响应。如果结果与预期不符，则说明测试没有通过，需要修复一些错误。端到端测试的自动化对提高可扩展性是非常重要的，因为随着新功能不断地增加，用户与应用的交互情景会越来越多。

我们可以再次求助于工具链，它既然已经将其他任务自动化，那么也应该可以将端到端测试自动化。工具链对于持续集成过程也非常重要。我们通常共用一个 CI 服务器来构建系统的各个部分，只不过每个部分的构建过程不一样。借助工具链，可以方便地与 CI 进程集成，因为我们只需要编写合适的工具链命令即可。

使用模拟数据有助于端到端测试，因为当工具模拟用户行为时，它需要发送请求。这样不仅提高了一致性，而且有助于在寻找导致出错的数据时排除部分测试用的模拟数据。

使用模拟数据可以针对同一份数据开发单元测试和端到端测试。



图注：工具链、模拟数据和测试运行在一个CI环境，我们编写的代码正是输入。

小结

这一章介绍了何为可移植的 JavaScript 应用。在 JavaScript 应用中，维持可移植性意味着不能与后端紧密耦合。这么做的主要好处是，我们的 UI 可作为一个独立的应用，不依赖于任何后端技术。

为了提高前端代码的独立性，可以模拟它依赖的后端 API。模拟 API 还能帮助我们专注地开发 UI——避免了后端问题阻碍前端的开发。

模拟数据还有助于我们测试代码。现在有很多实现方式不同的单元测试库可供使用。如果使用相同的模拟数据来执行测试任务，可以通过观察浏览器的变化来排除不一致性。我们应该将测试自动化，让它与开发过程中的其他任务一起自动执行。

我们实现的工具链能与持续集成服务器完美地配合，持续集成服务器是一个有助于提高可扩展性的重要工具。我们通常也会使用它实现端到端测试的自动化，以便更清楚地了解真实的使用场景。现在，是时候看一下扩展应用的种种限制了。我们不可能无限地扩展应用，下一章将介绍在应用扩展到一定程度时，我们该如何避免碰壁。

9

缩小规模

我们往往认为扩展是单向的——在现有的基础往上扩展。不幸的是，事实并非如此。只有当现有的基础仍未崩溃时，我们才能基于它往上扩展。因此，扩展的关键在于定位并解决扩展的瓶颈。

这一章将介绍 JavaScript 架构师几乎在每一个浏览器环境中都会遇到的基础性扩展限制。我们将会把用户看作一个影响可扩展性的因素，观察新功能是如何与已有功能发生冲突的。将臃肿的设计精简化也是一件重要的事情。

应用的组合方式，决定了我们通过关闭功能来缩小它的难度。这与代码的耦合程度密切相关，如果仔细留意，会发现我们经常需要重构组件，以便日后能够轻松地移除它们。

扩展限制

应用被它的运行环境即硬件环境和浏览器所限制。有趣的是，在 Web 应用中，我们还需要考虑传输代码的成本。例如，如果我们正在编写后端代码，那么当遇到问题时，可以通过添加代码来解决它，因为后端代码不需要被传输到其他地方——它们只在一个地方运行。

但对于 JavaScript，文件体积会带来影响。我们没办法改变这个事实，因此应该重视网络带宽——不管是传输 JavaScript 文件，还是传输 API 返回的应用数据。

这一部分将介绍浏览器环境的硬性扩展限制。随着应用不断地扩展，这些限制将越来越严重。在计划添加新功能时，必须考虑每一个限制因素。

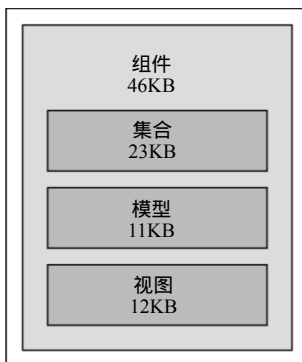
JavaScript 文件体积

JavaScript 文件的体积只会越变越大。最终，由于应用的加载时间太长，没有人愿意继续使用它。导致 JavaScript 文件庞大的原因，通常是由于某些功能过于膨胀。例如，如果我们往浏览器传输的文件太庞大，那么可能是由于文件中的某些逻辑过多，或者有些功能没被使用，又或者组件中有重复的代码。

不管是哪个原因，都会带来不好的影响。通常情况下，体积越小越好，但是在什么情况下，才会认为 JavaScript 文件的体积足够小呢？其实并没有统一的理想体积，需要分情况处理。比如，我们把应用部署在公网上？还是在合作伙伴的 VPN 上？不同系统的用户的接受标准也不一样。一般来说，公网用户对糟糕的加载性能和功能过剩的容忍度较低。而企业用户则希望有更多功能，对加载时间的要求也没那么高。

不断加入的新功能，是导致 JavaScript 文件变大的主要原因。需要编写新的组件来实现新功能，因此增加了文件体积。实现任何功能都需要编写一定大小的组件，并遵循已有功能的模式。如果模式是合理的，那我们应该能保持合理的组件大小。但是，当最后期限来临时，我们经常会编写重复的代码。而且即使我们已经最大程度地保持了代码的精简，仍然需要实现他们提出的新功能。

编译文件有助于缓解文件的体积问题。我们可以合并压缩文件、减少网络请求和总共的带宽。但是，任何新的功能都会导致这些编译后的文件变大，在遇到新的问题之前，都可以进行一定的扩展。如上所述，这些问题是相对的，它们依赖于应用的运行环境和用户。不管在什么样的情况下，都不能无限地增加 JavaScript 文件的体积。



图注：JavaScript文件的体积是所有构成组件的模块的总和。

网络带宽

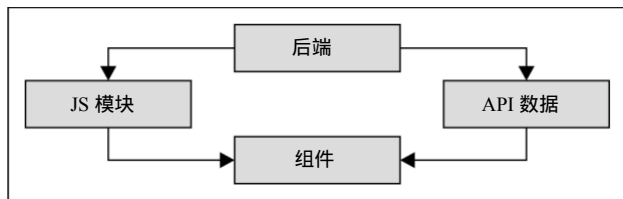
JavaScript 文件也消耗了应用的一部分网络带宽，尤其是在用户数量很多的情况下。用户数量始终是挑战架构设计的关键因素。JavaScript 代码关联着应用数据。这些 API 请求也会消耗网络带宽，带来可被用户感知的延时。

随着应用的使用区域越来越广泛，我们会发现不同程度的连接问题。世界上的很多地区都缺少高速网络，如果这些地区是我们的重要市场，那架构必须考虑网速慢的情况。使用 CDN 传输应用中使用的库会有所帮助，因为 CDN 考虑了请求的地理位置。

我们所面临的挑战在于任何的新功能都会增大网络的带宽消耗。这些消耗包括文件代码以及组件使用的新 API。需要注意的是，这些影响不会马上被感知。例如，这些组件可能不会在加载页面时请求 API，而是当用户导航到一个特定的 URI 时才会请求。

尽管如此，随着时间的推移，新的 API 总是会增加网络带宽的消耗。并且，当用户使用新功能时，发送的 API 通常不止一个。为了组织和展现数据，应用通常会发送三个或者更多的 API 请求。新增一个 API 看似不是什么大问题，但我们必须时刻留意，因为通常到最后增加的请求都不止一个，而这将意味着更多的带宽消耗。

那么是否存在一个统一的网络带宽限制呢？理论上是没有的，但是它与 JavaScript 文件体积一样——只要我们愿意，可以将它们增加到 10MB，只不过这不会提高，反而会降低用户体验。网络带宽消耗与此是同一个道理。



图注：组件请求的JavaScript模块和API数据消耗了网络带宽。

以下的例子展示了当发送更多请求时，会对应用总延时造成的影响。

```
// model.js
```

```
// 这个模块拥有一个形式上的fetch()方法，该方法不会设置任何数据。
export default class Model {

  fetch() {

    // 返回Promise供调用者使用。
    // 该异步方法在1秒后resolve，旨在模拟一个真实的网络请求。
    var promise = new Promise((resolve, reject) => {
      setTimeout(() => resolve(), 1000);
    });

    return promise;
  }
};

// main.js
import Model from 'model.js';

function onRequestsInput(e) {
  var size = +e.target.value,
      cnt = 0,
      models = [];

  // 基于requests创建模型。
  // number.
  while (cnt++ < size) {
    models.push(new Model());
  }

  // 开始计时，看下获取所有模型需要多久。
  console.clear();
  console.time(`fetched ${models.length} models`);

  // 使用Promise.all()以同时获取所有模型。在获取完成时，可以结束计时。
  Promise.all(models.map(item => item.fetch())).then(() => {
    console.timeEnd(`fetched ${models.length} models`);
  });
}
```

```
    });  
}  
  
// 注册DOM监听器，根据输入的requests创建和获取相应数量的模型。  
var requests = document.getElementById('requests');  
  
requests.addEventListener('input', onRequestsInput);  
requests.dispatchEvent(new Event('input'));
```

内存消耗

每当新增一个功能，都会多占用一块浏览器的内存。这是不言而喻的，但是依然值得一提。内存问题不仅仅会影响性能，有时甚至会拖垮整个网页。因此，我们需要密切关注代码的内存分配情况。浏览器内置的 Profiler 可以记录一段时间内的内存分配信息，这是诊断问题和观察代码运行情况的一个很好的工具。

频繁地创建和销毁对象可能会拖慢性能。这是因为不再被引用的数据会被当作垃圾回收。在垃圾回收器工作时，浏览器会暂停执行 JavaScript。因此，我们的要求是矛盾的——既希望代码被快速执行，又希望能够节省内存。



解决的办法是避免垃圾回收器进行不必要的回收。例如，有时候我们可以将一个变量提升到更外层的作用域，以此来减少应用在整个生命周期中创建和销毁该变量的次数。

另一个场景是短时间内（例如，在循环体内）频繁的内存分配行为。虽然 JavaScript 引擎可以智能地处理这些场景，但我们仍须留意。最好的底层库会考虑垃圾回收器，避免不必要的内存消耗。

API 返回的数据也会消耗内存。当返回的数据量大时，会消耗大量的内存。我们应该给 API 返回的数据设置上限。许多后端 API 都会自动设置，一次最多可返回 1000 个实体。如果想要获取集合中特定范围的数据，就需要提供偏移量。但是，我们应该更进一步地限制 API 返回。因为集合中的实体对应着浏览器中的模型，模型如果过多，就会占用大量内存。

随着用户不断地切换页面，这些集合会被当作垃圾回收，每一个新功能都有微小的

可能触发内存泄漏。这些小问题很难处理，因为内存泄漏是缓慢的，而且在不同的环境中表现不同。当内存泄漏严重并且明显的时候，问题较容易重现，因此更容易定位和修复。

以下是一个内存消耗快速失控的例子：

```
// model.js
var counter = 0;

// 该模型的每一个新的实例都会消耗更多的内存。
export default class Model {

  constructor() {
    this.data = new Array(++counter).fill(true);
  }
};

// app.js
// 这个简单的应用会将item添加到一个数组。
export default class App {

  constructor() {
    this.listening = [];
  }

  listen(object) {
    this.listening.push(object);
  }
};

// main.js
import Model from 'model.js';

function onRequestsInput(e) {
  var size = +e.target.value,
      cnt = 0,
```

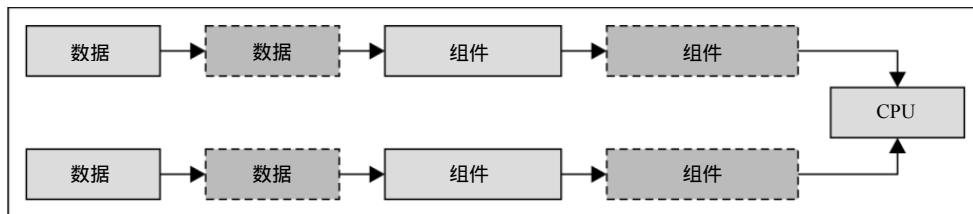
```
models = [];  
  
// 基于requests创建相应数量的模型。  
while (cnt++ < size) {  
    models.push(new Model());  
}  
  
// 开始计时，计算获取所有模型所用的时间。  
console.clear();  
console.time(`fetched ${models.length} models`);  
  
// 使用Promise.all()以同时获取所有模型。在获取完成时，可以结束计时。  
Promise.all(models.map(item => item.fetch())).then(() => {  
    console.timeEnd(`fetched ${models.length} models`);  
});  
}  
  
// 注册DOM监听器，根据输入的requests创建和获取相应数量的模型。  
var requests = document.getElementById('requests');  
  
requests.addEventListener('input', onRequestsInput);  
requests.dispatchEvent(new Event('input'));
```

CPU 消耗

CPU 是影响用户界面响应速度的一个重要因素。如果它随时都可以执行代码（例如，对一个点击事件做出响应），那 UI 的反应给人感觉是及时的。但如果 CPU 正忙于别的任务，那代码就需要静静等待，而用户也需要等待。通常操作环境中有很多软件排队等着使用 CPU 资源——大部分都是无法控制的。我们无法减少浏览器以外的应用对 CPU 的使用，但可以减少 JavaScript 应用对 CPU 的使用。但首先需要知道这些 CPU 消耗从何而来。

站在架构层面，我们不会考虑那些只能提高单个组件部分功能效率的小优化。我们更关心缩小规模，因为缩小规模能在保证应用正常运行的同时，大大地降低 CPU 消耗。第 7 章中介绍了如何分析代码。通过剖析代码，可以知道哪些代码消耗了 CPU 资源。有了 Profile 作为测量工具，就可以开始优化了。

站在架构层面,要关注两个影响 CPU 使用率的因素,分别是活跃功能的数量和这些功能用到的数据量。例如,随着新组件的加入,应用会消耗更多的 CPU 资源,因为当 UI 上发生事件时,新功能对应的组件需要做出某些响应。但这不太可能造成较大的影响,伴随新功能的 API 数据才是消耗昂贵的 CPU 资源的主要来源。



图注:更多的组件处理更多数据会吃掉更多的CPU时钟周期。

例如,如果在保证数据集不变的情况下不断地添加新功能,将会明显感到 CPU 的延迟。因为有了更多的间接调用,某一事件发生就可能会执行更多的代码。但是,这个问题形成得非常缓慢——我们可以毫不费力地添加成百上千的新功能去消耗 CPU。导致应用难以扩展的主要原因,是持续增长的数据。因为如果用功能数量乘以数据量,就会发现应用对 CPU 的消耗是呈现指数级增长的。

也许并不是所有的功能都会使用所有的数据,也许我们的应用中并没有很多间接调用。但是,在缩小应用规模时,仍然应该重视这一点。因此,如果希望减少 CPU 消耗,应该移除新功能和它们处理的数据——这是唯一能够快速获得成效的方法。

下面的例子展示了组件的数目和数据量的增长,是如何逐渐地消耗越来越多的 CPU 资源的。

```
// component.js
// 应用中的一个通用组件。
export default class Component {

  // 构造函数接收collection参数,并调用了它的reduces方法来消耗CPU资源。
  constructor(collection) {
    collection.reduce((x, y) => x + y, 0);
  }
}

// main.js
```

```
import Component from 'component.js';

function onInput() {
  // 根据输入的data数创建一个新数组。
  var collection = new Array(+data.value).fill(1000),
      size = +components.value,
      cnt = 0;

  console.clear();

  // 开始计时，计算x个组件处理y个集合使用的时间。
  console.time(`${size} components, ${collection.length} items`);

  // 根据输入的components数创建对应数量的组件。
  while (cnt++ < size) {
    new Component(collection);
  }

  // 处理完组件，停止计时。
  console.timeEnd(`${size} components, ${collection.length} items`);
}

// 注册DOM事件监听器。
var components = document.getElementById('components'),
    data = document.getElementById('data');

components.addEventListener('input', onInput);
data.addEventListener('input', onInput);

components.dispatchEvent(new Event('input'));
```

后端能力

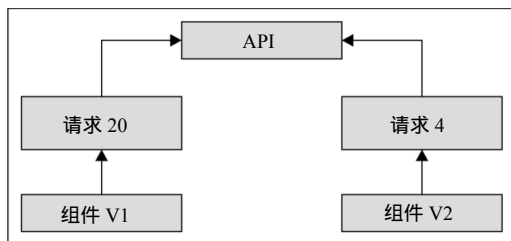
我们将要介绍最后一个限制应用扩展的因素，那就是提供静态资源和 API 数据的后端。它之所以会限制扩展，是因为代码在抵达浏览器前是无法被执行的，并且，在原始数据到达浏览器之前，应用也是无法展示信息的。两者都依赖于后端传输，但在开发前端功能时，

我们应该时刻留意后端的某些状态。

首先需要留意的是应用的使用情况。与运行 JavaScript 代码的浏览器一样，后端 API 也同样不能无限地扩展。即使后端可通过某些浏览器没有的方法进行扩展，但当请求量变大时，仍会受到冲击。其次要留意的是代码与 API 的交互方式。我们应该观察用户如何使用应用，这些交互发送了哪些 API 请求。如果可以优化一个用户发送的请求，那么用户数量的增长对后端的影响也会变小。

例如，我们希望避免发送不必要的请求。意思是，当不需要某些数据时，就不会加载它们，并且不用重复地加载相同的数据。如果某个用户在进入会话的前五分钟都没有使用任何功能，那么后端在这段时间内会拥有更多的空闲去处理其他请求。有时不同的组件会使用相同的 API。当这些组件同时被创建并请求了相同的 API 时，会发生什么呢？后端必须同时处理这两个请求，但这是不必要的，因为它们将返回相同的内容。

我们需要整理组件的通信，来应对后端加载这些影响扩展的因素。在这个例子中，第二个组件可以在挂起的请求中找到对应的 Promise 并将其返回，而不是创建一个全新的请求。



图注：新组件应该尽量减少带宽的消耗，一种方式是使用更少的API请求来完成相同的功能。

互相矛盾的功能

随着软件的不增大，功能之间的界线会变得越来越模糊。功能之间肯定存在一些重叠，这可能是件好事。如果功能之间一点重叠都不存在，那当用户试图从 UI 的一个区域过渡到另一个区域时，过程就会很困难。但当功能重叠的层级达到一定数量时，问题就会出现。每当有新的功能加入时，这个问题都会变得更严重，直到它被解决。

导致这个问题的其中一个原因是，随着时间的推移，应用的一些部分变得无关紧要，但它们并没有被移除，而是被继续留在了原处，成为障碍。这个影响在很大程度上是由顾

客的需求造成的，因为顾客的需求决定了产品未来的方向，也提示了我们应该修改哪些地方来迎合需求，或是在未来应该移除哪些功能。

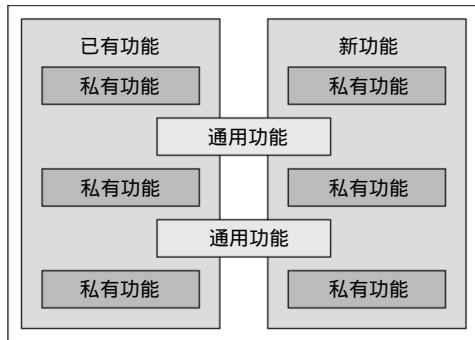
重叠的功能

在应用的生命周期中，会出现与已有功能重叠的新功能。这是软件开发的本质——在已有的基础上构建，而不是构建与已有功能毫不相关的另类功能。如果重叠的功能不但不会对原有功能造成破坏，而且有助于从已有功能过渡到加强的新功能时，那这种重叠就是可取的。

当重叠与已有功能发生冲突时，就会导致问题。这样就像在森林里建房子，却没有将树木事先移除。如果想要保证重叠是无缝并易于扩展的，我们必须调整已有代码来兼容新功能，或者重新思考新功能，以便更好地融入现有代码。这很有意思，因为在实现新功能之前，往往要考虑缩小它的规模，这会通常比在实现新功能之后做更容易。

荒谬的功能重叠会导致用户发现应用笨拙难用，抱怨连连。实际上我们经常告诉自己——虽然这样添加不好，但如果想在最后期限之前完成，就只能这么做了。这样做的代价是什么呢？除了用户的不满，还需要操心代码。我们几乎不会这么说——好吧，虽然用户不喜欢，但是代码很棒。糟糕的用户体验，通常是由糟糕的功能规划和实现导致的。

正如我们所见，解决的办法很简单，只要提供修改空间，或者调整新功能即可。我们经常忘记记录潜在问题。例如，当发现已有代码与计划中的新功能不兼容时，我们应该告知大家，并编写概要，描述不兼容的地方和原因。记录这些信息留作日后参考，总比忽略它们强。通过与团队成员的及时沟通，可以改进架构的设计。



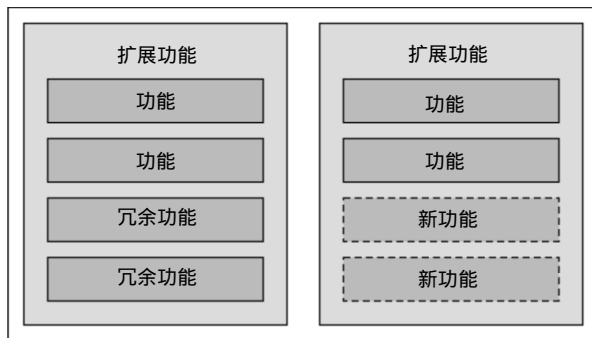
图注：去除多余代码可以从已有功能和新功能的重叠下手。

冗余的功能

随着时间的推移，一些功能被证明是有价值的。因为用户喜欢，并经常使用它们。更重要的是，几乎不需要修改，这些功能就可以一直正常运作。另一方面，一些功能并不如意，很快就“生锈”了。这种情况的表现有很多种：可能有少量用户喜欢这个功能，但它有很多问题，并且难以维护；也可能大部分用户都喜欢这个功能，但它阻碍了项目中多项计划的展开；但最常见的情况是没什么人使用它。

不管原因是什么，功能确实会变得冗余。我们这个行业有一个问题，就是喜欢囤积代码。有时我们会保留冗余的功能——可能是因为移除它会导致很多功能会失效，也可能是为了向后兼容。其他时候，这更多的是一个前端的问题，我们保留着这些功能，仅仅是因为没有收到明确的指令去移除它们。但是，如果希望未来能够扩展应用，恐怕必须要移除它们。

这是一个关于主动和被动的的问题。正如我们所知，每一个组件都会对扩展有一定的限制——不管是网络、内存、CPU、还是其他方面。也许即使不移除这些冗余代码，也不会出现问题。但最好还是移除它们，以降低它们限制应用扩展的可能性。也许这段代码看似无害，但将它完全移除难道不是更好吗？更进一步，我们最好向周围的人灌输这一观点——先移除冗余的东西，再思考下一步。如果有简化代码的先例，那我们更有可能说服利益相关者出售一个更精简的产品。



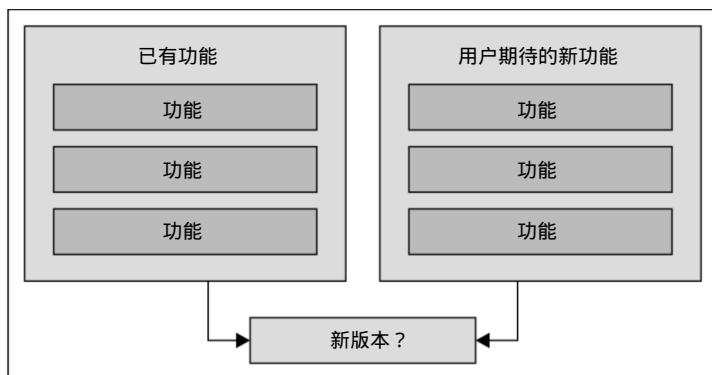
图注：应用的扩展空间是有限的，移除冗余的功能有助于释放扩展空间。

用户需求

我们会根据产品类型和用户类型，将用户需求转化为规范的计划和实现并快速响应。

我们都希望满足用户——这是构建软件的目的。但正是这些快速实现、让用户为之疯狂的功能破坏了我们的架构。这么做就好像修复 bug，当发现 bug 时，会尽可能快地修复代码，消灭 bug。

新功能不是 bug。不管用户和经理怎么说——没有这些功能他们依然能活下来。我们要想办法给自己争取足够的时间，将新功能融入架构。但这不代表我们可以不断地拖延，我们仍然应当及时完成。也许删除冷门的功能是改善架构最快的方式。



图注：整理出下一个版本的功能。它们可能是已有功能，也可能是用户期待的新功能。

设计失效

依靠在既有代码基础上修修补补来缩小规模只是一种方式。例如，通过删除功能或修改已有组件来兼容新功能。但相对于未来的发展，这种做法的成效不大。两年前适用的设计思想针对的是两年前的功能，不一定适用于现在。

为了使架构持久，我们需要修复失效的模式。这些模式之所以仍然能够正常运作，是因为我们对它们进行了调整，即便它们不是完成某些工作的最佳工具。架构的设计不是一次性的，随着软件的变化、对可扩展性的要求越来越高，我们需要不断地寻找合适的设计。

这一部分将介绍几种处理设计瑕疵的方法。也许代码中有冗余的移动部件，也许复杂的组件通信模型导致处理 API 数据的方式过于低效，也许复杂的 DOM 结构导致了笨重的选择器，因此拖慢了开发。这些只是一部分可能性——每个项目的问题都不一样。

多余的组件

在开始设计架构和开发软件时，我们会选择适合当前情况的模式。我们将组件之间的耦合设计得松散，为了实现松耦合，需要有所牺牲——添加更多的移动部件。例如，为了使每个组件更专注于某项职责，需要将大组件拆分成更小的组件。这些模式决定了功能组件的组合方式。如果我们遵循的模式中有多余的内容，那么任何新功能都会包含这些多余的内容。

难以确定哪个是正确的模式，是因为在决定使用什么模式时，我们没有足够的信息。例如，框架使用的设计模式非常通用，因为框架比我们的应用服务了更广泛的用户。因此，当我们希望使用与框架一样的模式时，需要对它们进行调整，以适应我们特有的功能。随着用户需求改变着产品的本质，这些模式会逐渐发生变化。我们可以投入时间修复模式，也可以保持最初的模式不变，只修复出现的问题。而扩展架构的最佳方法，就是敢于改变曾经的根基。

模式中最常见的问题就是多余的间接调用，即那些抽象并且没有任何价值的组件。它们的唯一的职责就是将组件从某些地方抽出。随着时间的推移，我们会发现使用这些模块的代码相对较少，并且看起来差异不大。这些模块之所以小，是因为它们的功能不多；之所以看起来一样，是因为我们坚持在代码中使用一致的模式。在构思模式时，这个组件非常有用。但在越来越多的组件实现之后，它的意义便会越来越小。删除这些组件不但不会破坏设计，反而会让整个项目更加轻巧。纸上画的设计模式和真实应用中的设计模式之间的区别是很有趣的。

在下面的例子中，展示了两个组件：一个使用了控制器；另一个未使用控制器并且少了一个移动部件。

```
// view.js
// 一个极简的视图，它会更新DOM中的某个元素的文本。
export default class View {

  constructor(element, text) {
    element.textContent = text;
  }
}
```

```
};

// controller.js
import events from 'events.js';
import View from 'view.js';

// 一个控制器组件，它接收并配置一个路由器实例。
export default class Controller {

  constructor(router) {
    // 添加路由，当路由被激活时，创建一个新的View实例，并更新内容。
    router.add('controller', 'controller');
    events.listen('route:controller', () => {
      new View(document.getElementById('content'),
'Controller');
    });
  }

};

// component-controller.js
import Controller from 'controller.js';

// 这个应用实际上只会创建控制器。它是否有存在的必要呢？

export default class ComponentController {

  constructor(router) {
    this.controller = new Controller(router);
  }

};

// component-nocontroller.js
import events from 'events.js';
import View from 'view.js';
```

// 这是一个不依赖于任何组件的应用组件。它完成了一个控制器应该完成的工作。

```
export default class ComponentNoController {

  constructor(router) {
    // 配置路由器，并创建新的视图实例，刷新DOM的内容。
    router.add('nocontroller', 'nocontroller');
    events.listen('route:nocontroller', () => {
      new View(document.getElementById('content'), 'No
Controller ');
    });
  }

};

// main.js
import Router from 'router.js';
import ComponentController from 'component-controller.js';
import ComponentNoController from 'component-nocontroller.js';

// 组件之间共享的全局路由器实例。
var router = new Router();

// 创建两种组件实例，并开启路由器。
new ComponentController(router);
new ComponentNoController(router);

router.start();
```

低效的数据处理

微小的优化难以为效率带来巨大的提升。另一方面，重复的处理过程可能对扩展有着巨大的限制。我们所面临的挑战在于，这些重复的代码难以被发觉。它们通常出现在组件之间传输数据的时候。第一个组件对 API 数据进行转换，接着，原始数据被传输到了第二个组件，第二个组件又对数据进行了一模一样的转换。随着组件越变越多，这些负面影响也会逐渐累积。

我们之所以很少发现此类问题，是因为被自己优雅的设计模式蒙蔽了双眼。由于我们始终坚持代码的一致性，因此有时这些有损用户体验的问题会被掩盖。意思是，我们不断地保持组件之间耦合松散，从而使得架构可以在多个方面扩展。

通常，少量的数据重复处理是完全可接受的。这取决于在处理其他可扩展性问题时，它是否能帮助我们提高灵活性。例如，如果有分离的部署，就可以轻松地修改配置、开启和关闭功能，那么这么做是有意义的。但是，可扩展性在某一方面的提高，通常意味着它在另一个方面的降低。例如，当数据量增长时，从一个组件传输到另一个组件的数据量也会增长。因此，虽然重复的数据转换过程在过去不是问题，但现在却是一个大问题。当这种情况发生时，我们需要减少数据处理。

再一次强调，我们无须进行微小的优化，而应该寻找能大大提高效率的突破点。我们应该从网络请求着手，因为减少刚加载页面时请求的数据量能大大提高前端效率。第二个要检查的地方是组件之间传输的数据。在这一步，我们要确保接收数据的组件没有与发送数据的组件执行相同的操作。

在下面的例子中，有一个组件会在 `fetch()` 方法被调用时获取模型数据。这个例子还展示了另一个可选的实现，这个实现在已经有待处理请求的情况下，不会再去获取模型。

```
// model.js
// 拥有一个虚拟方法fetch的虚拟模型。
export default class Model {

  fetch() {
    return new Promise((resolve) => {
      setTimeout(() => {

        // 通过在模型中记录日志，我们可以验证fetch是否被执行。
        console.log('processing model');

        // 设置虚拟数据，并使用模型实例resolve promise。
        this.first = 'First';
        this.last = 'Last';

        resolve(this);
      }, 1000);
    });
  }
}
```



```
    });  
  
  }  
  
};  
  
  
// component-duplicates.js  
import Model from 'model.js';  
  
// 拥有一个模型的标准应用组件。  
export default class ComponentDuplicates {  
  
  constructor() {  
    this.model = new Model();  
  }  
  
  // 一个不成熟的model.fetch代理。  
  // 之所以说它不成熟，是因为它不应在fetch请求未完成时，再次请求模型。  
  fetch() {  
    return this.model.fetch();  
  }  
  
};  
  
  
// component-noduplicates.js  
import Model from 'model.js';  
  
// 拥有一个模型的标准应用组件。  
export default class ComponentNoDuplicates {  
  
  constructor() {  
    this.promise = null;  
    this.model = new Model();  
  }  
  
}
```

```
// 一个机智的model.fetch代理。
// 它通过在请求完成前保存promise，避免了重复的API请求。
fetch() {

    // 已经存在promise，因此无须做任何事情——我们可以通过返回promise提前返回。
    if (this.promise) {
        return this.promise;
    }

    // 通过调用model.fetch()获取并保存promise。
    this.promise = this.model.fetch();

    // 一旦promise被resolve，立刻移除对promise的记录。
    this.promise.then(() => {
        this.promise = null;
    });

    return this.promise;
}

};

// main.js
import ComponentDuplicates from 'component-duplicates.js';
import ComponentNoDuplicates from 'component-noduplicates.js';

// 创建两种组件的实例。
var duplicates = new ComponentDuplicates(),
    noDuplicates = new ComponentNoDuplicates();

// 调用两次fetch()方法。
// 可以看到，两次调用中，都执行了模型的fetch方法，即使这是没必要的。
duplicates.fetch();
duplicates.fetch().then((model) => {
    console.log('duplicates', model);
});
```

```
// 我们再次调用两次fetch()方法。
// 这个组件在第二次调用时不会执行模型的fetch方法。
noDuplicates.fetch();
noDuplicates.fetch().then((model) => {
  console.log('no duplicates', model);
});
```



当组件之间耦合松散时，难以避免会发送重复的 API 请求。举个例子，某个功能创建并获取了一个模型。同一页中的另一个功能也需要相同的模型，但是它对第一个组件一无所知——因此它又重复地创建和获取了数据。

这样的结果是发送了一个完全相同的 API 请求，明显是多此一举的。这样不仅会由于重复执行回调函数降低前端效率，还会影响整体系统。不必要的请求会阻塞后端的请求队列，影响其他用户。我们必须留意这类重复的请求，并相应地调整架构。

过度创建标记

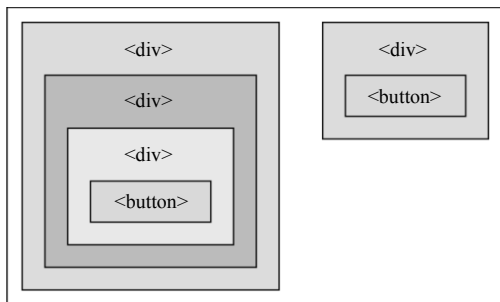
用于渲染 UI 组件的标记可能会爆发并难以控制。为了实现特殊的视觉和体验，我们需要对标记进行一些细微的调整。接着，因为它在某些浏览器上看起来不太美观，我们会再次对它进行调整。最终的结果就是，元素之间嵌套过深，反而失去了原有的语义。所以我们应该尽量保证标签的语义化——测试应该被放在 `p` 标签中，可点的按钮应该用 `button` 元素表示，页面区域应该由 `section` 元素划分，等等。

真正的难点在于，设计通常是由线框图表示的，我们需要将它实现得易于划分，便于框架和组件使用。因此，我们在试图保持语义化的时候，丢失了原本的简洁，同时，划分成独立的视图并不总是可以做到的。

我们应该尽可能简化 DOM 结构，因为它对 JavaScript 的简洁程度和性能有直接的影响。例如，组件经常需要定位页面上的元素，可能是为了改变它们的状态，也可能是为了读取它们的值。我们可以通过编写选择器字符串来查询 DOM，选中需要的元素。这些字符串贯穿视图代码，并反映了标记的复杂程度。

当在代码中发现复杂的选择器字符串时（甚至可能是自己写的），我们完全不知道它查

询的是哪个元素——因为 DOM 结构和标签名通常帮不了什么忙。所以使用语义化的标记可以在很大程度上帮助我们编写 JavaScript 代码。复杂的 DOM 结构对性能也会有影响——频繁地深度遍历 DOM 结构会损害性能。



图注：过深的元素嵌套通常能被简化成更少的元素。

应用组合

本章最后再讨论一下应用组合。这一部分对应用进行了高度的概括，以便分清它的各个组成部分。第 3 章介绍了组件的组合，其原则仍然适用于此处，只不过适用问题的层次更高了一些。

第 6 章介绍了可配置性，这与应用组合的思想也是相关的。例如，默认开启或关闭功能。应用的组合方式对缩小规模也有很大的影响。

功能的启动

关闭功能是一个可以快速缩减规模的手段，难点在于如何说服利益相关者这么做。接着我们便可以移除功能，这样就完成了吗？这可不一定，因为移除功能可能需要一些时间。例如，它可能涉及到系统的多个入口，并且这些入口无法通过任何配置来关闭。但这并不是什么大问题，只不过需要花费更多的时间来移除它们。

唯一的问题是移除功能后的测试工作。如果无法通过配置关闭功能，那么在测试前就需要花费时间写代码将它们移除。例如，我们可以花费五分钟修改配置来关闭功能，然后立即看到效果。那么，说不定我们能在从系统中真正移除这些功能之前，提前发现许多其他要做的工作。

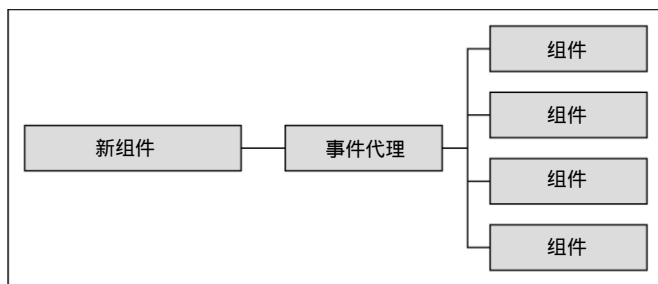
在移除功能后，除了要测试应用的运行时行为，还要修改编译选项。如果生产代码最终会被编译成少量 JavaScript 文件，就需要想办法在编译时排除这些功能。相当于通过配置关闭组件。这意味着在代码运行时，特定的文件不会被加载。如果我们直接将功能从源码仓库中移除，那么要考虑的问题就更少了——工具无法构建不存在的东西。但是，如果有成百上千个可能被引入的潜在组件，那么我们就需要想办法排除它们。

新功能的影响

影响应用的下一个主要因素是新功能的添加。是的，虽然我们讨论的是如何进行缩减，但不能忽略应用中待添加的新功能。毕竟这是我们进行缩减的原因——缩减的目的不是为了构建一个功能更少的、更小的应用，而是为了给用户想要的新功能提供空间，并随着时间的推移，提高产品的总体质量。

添加功能和移除功能经常是同时进行的。例如，在火速开发的过程中，当一个小组在实现新功能时，另一个小组则负责移除一些会导致问题的功能。由于这两项工作都会对应用产生很大的影响，我们必须考虑周全，尽量使影响最小化。

我们应该确保移除旧功能不会影响新功能。例如，新功能可能依赖于旧功能的某些部分。如果设计合理，那么组件之间不会存在任何直接的依赖。但是，我们对复杂度的理解是不够清楚的，尤其是间接的因果关系。因此，通常我们不会并行处理这两类任务。



图注：基于内部的组件通信模型进行开发，在系统中添加新组件带来的影响是相当小的。

重要的库

影响应用组合的最后一个因素是使用的框架和库。当然，我们只想使用需要的东西——要么用，要么不用。问题的关键在于怎么引入那些更小的库。相比之下，框架则包含了大

部分功能，它几乎包含了所有我们需要的功能。虽然未必总是正确的，通常使用框架有助于减少我们对第三方库的依赖。

如今，甚至框架都是模块化的，因此可以挑选自己需要的部分，忽略剩下的。即使如此，还是很可能引入没有用的功能（不管是来自框架还是其他地方）。这种情况经常发生，例如，某个库已经实现了我们需要的某个功能，因此我们不想自己再实现一遍。但随着页面的增多，依赖会变得混乱，应用中会充斥着没用的代码。我们应准确地辨别对应用至关重要的核心依赖。

小结

这一章告诉了大家，应用并不是在每一个方面都可以无限地扩展。实际上，应用在任何一方面都无法无限扩展。因为每一个方面都被不同的因素所限制。这些因素以独特的方式组合在一起，我们应进行必要的权衡。如果希望持续扩展，就必须在某些方面进行缩减。

新功能来自用户的需求，并且经常与已有功能重合。导致重合的原因可能是我们没有很好地定义新功能，也可能是系统已有的入口设计不当。不管怎么样，这会增加工作的难度，包括移除已有功能、添加新功能。我们需要时常移除重叠的部分，因为它们不管是在代码层面还是在使用层面都会带来混乱。

缩减并不只是一系列的工作——我们还需要考虑设计模式。在移除了某个功能后，我们需要审视使用的模式，并问问自己，以后是否仍然希望这么实现？修复设计模式，是更好、更有助于提高可扩展性的方式。即使我们已经缩减了代码，仍然可能存在错误。在下一章中，将更仔细地研究失效组件，并学习如何处理它们。

10

处理错误

此时，我们对自己的架构甚是满意，因为已经考虑了可扩展性、做出了所有合理的权衡、适当地牺牲了性能以提高可配置性，等等。为了使 JavaScript 架构可扩展，还需要考虑一个因素，那就是人为因素。我们虽然聪明，但仍是最脆弱的一个环节，因为是我们设计和实现了应用——并且，我们真的很容易犯小错误。

在结束软件开发之旅之前，我们还应该知道在设计组件时必须考虑错误处理。这包括错误模式——应该快速失效还是从错误中恢复？这涉及错误的特性，有的错误相对容易处理，有的则不然。但它还关乎我们对极限的理解，因为检测并从每一个可能出现的错误中恢复是不现实的。

在扩展应用的同时，还需扩展处理错误的方式。这是影响可扩展性的众多因素之一，我们还需做出权衡。接下来，将通过一个快速失效的错误模式来开始这一章节。

快速失效

采用快速失效错误模式的系统或组件会在出错时停止运行。这听起来可能不是一个令人满意的设计，但是试想一下其他方案：一个系统或组件出现了错误，但仍然继续运行。这些组件可能会保持在错误的状态运行，而且永远不会终止。

在某些情况下，我们偏向于恢复一个出错的组件，这会在稍后介绍。在这一部分，会介绍决定 JavaScript 组件是否应采用快速失效模式的一些标准，以及快速失效对用户的影响。有时候，甚至快速失效机制也会出错，我们也要考虑到这一点。

使用质量约束

组件快速失效通常都是由一个已知的问题导致的。但也可能出现无法预测的问题。这两种情况都会使组件处于错误状态，我们不希望应用若无其事地继续运行。在介绍质量约束之前，让我们先理解快速失效。快速失效是关于应用该如何运行的断言。例如，尝试发送 API 请求不应超过三次，等待响应的时间不应超过 30 秒，某个某型的某个属性不能为空，等等。

当断言失效时，应停止执行——不管这个断言属于某个组件，还是整个系统。这么做的目的并不是惹怒用户。对待任何错误，我们都希望尽量避免它。可以将快速失效看作汽车事故中的安全气囊——事故发生后，汽车便不可以再行驶。

我们不能轻易地使整个组件或系统失效。例如，如果某个小组使某个功能快速失效，而未通知其他组，则很可能会导致整个系统崩溃。当其他小组终于定位到问题时，却发现这是特意的设计。这种错误模式需要严谨的设计。快速失效场景的主要好处，是能避免由于应用未及时终止而导致的灾难性后果。



图注：超出限制时，组件会快速失效，很可能会导致整个系统也快速失效。

提供有意义的反馈

我们不希望在应用失效时为用户或开发小组提供错误的信息。因此必须区分快速失效和完全不受控制的错误。后者会导致应用崩溃，甚至可能导致浏览器标签页崩溃。更糟糕的情况，是应用仍然苟延残喘地运行，表单正常，然而实际上，它的内部已经出错了。

因此，当采用快速失效模式时，一定要明显地提醒用户某些功能出现了故障，不应该继续使用。不管出错的是一个单独的组件，还是整个应用，都要保证提示简明扼要。用户并不总是需要知道错误原因，他们只需要知道组件或应用现在崩溃了，他们的任何操作都不会生效。

在架构中使用快速失效模式有一个重要的作用——能在出错时接收到提示。用户永远不需要猜测。使用中的软件发生崩溃确实很恼人，但这肯定不会比等待和尝试了半天后才发现它崩溃了更糟糕。在清楚地提示了应用或部分功能已经出错后，我们也许希望阻止用户继续使用。比如在页面顶层添置一个 div，或者注销 DOM 的事件监听器。

下面的例子展示了两个错误处理函数。第一个例子通过禁用按钮，隐式地处理了错误。另外一个函数也做了相同的处理，但同时展示了一条错误消息。

```
// DOM元素。
var error = document.getElementById('error'),
    fail1 = document.getElementById('fail1'),
    fail2 = document.getElementById('fail2');

// The first event merely disables the button.
// 第一个事件处理函数仅仅禁用了按钮。
function onFail1(e) {
    e.target.disabled = true;
}

// 第二个事件处理函数不但禁用了按钮，还显式地提醒用户出现了什么错误。
function onFail2(e) {
    e.target.disabled = true;
    error.style.display = 'block';
}

// 注册事件处理函数。
fail1.addEventListener('click', onFail1);
fail2.addEventListener('click', onFail2);
```

当无法快速失效时.....

我们可以为组件设计快速失效机制，但是无法保证这些机制不会出错。因为这些用于检查自己程序的代码，是我们自己写的。我们可以编写一层又一层快速失效的错误处理代码。但是什么时候能结束呢？

在追求可扩展性的同时，应该意识到我们无法预言所有的错误。因为在某些时刻，我

们应该将注意力放在新功能上，而不是总放在架构上。不必要的错误处理代码不会给产品带来任何好处，只会增大代码体积。只要我们专注地实现功能，那么那些需要快速失效的场景就会自然而然地出现。

编写错误探测代码时，我们面临着一个问题——错误处理也需要随着系统、外部因素的影响一同扩展。例如，更多的用户会为后端带来更多需求。这意味着错误探测代码永远都不够完整——我们要如何解决这个问题呢？不用解决。因为解决这类问题无法提高可扩展性，在最开始就努力避免错误才是更有效的方法。

容错

拥有容错能力的系统可以恢复一个出错的组件。系统可以纠正组件的错误，或者使用一个新的实例来代替出错的实例。我们可以把容错想象成可以只使用一个引擎降落的飞机——而乘客则是我们的用户。

通常情况下，我们会在大型服务器环境中使用容错机制，但它也同样适用于复杂的前端场景。本节首先会讨论如何区分关键和非关键组件。接着会介绍如何探测错误，以及如何处理错误才能使应用继续工作。

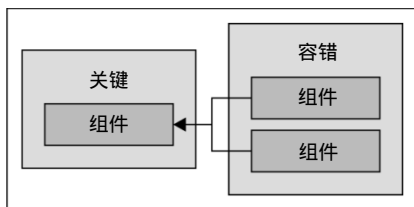
区分关键行为

正如代码中的关键部分不能中断，应用中的某些组件也不能优雅地失效。有些组件必须正常工作，一旦出错，必须快速失效，以避免出现更严重的错误。因此，我们要对组件进行分类。虽然似乎大家都知道如果能让组件正常工作，就应该不断对它们进行分类。但我们仍然应该将这种观念发扬光大。

对组件进行了分类后，可以清楚了解哪些是关键组件。这些关键组件必须正常工作，在任何情况下，它们都无须从错误中恢复。一旦这些组件出错，就应该被立刻修复。我们可通过单元测试更准确地定位关键组件。

我们不必为组件进行过度分类。例如，第一级包含绝对关键的组件，下一级包含不关键但必须正常工作的组件，等等——过度分类违背了我们的初衷。当某个组件出错时，应用要么继续工作，要么停止。我们可以按照这个简单的规则将组件划分为两种类型，这种

方法更简单和直接。任何非关键组件都应该拥有容错能力。因此，在完成分类后，我们可以开始为非关键组件设计错误探测和恢复策略了。



图注：对比关键组件和其他能够容错的组件。

探测和控制错误行为

如果我们设计的架构拥有优秀的可扩展性，那组件之间应该是耦合松散的。这意味着组件的错误处理之间也是耦合松散的。一个组件出错，不应导致其他组件也出错。如果我们可以实现这个愿望，那么所有事情都会变得简单。因为当一个组件出错时，我们可以确保这个错误不是由其他组件导致的。因此，定位和解决问题会更简单直接。

使用事件代理有利于分离组件之间的错误解耦。如果通过事件代理实现了组件内部的所有通信，那么可以在代理层方便地探测错误，避免错误影响到其他组件。例如，如果一个组件接收到事件后，在执行回调函数的过程中出错，那它可能会给整个应用带来副作用，甚至可能导致应用完全崩溃。

反之，事件代理可以探测错误，可以通过一个抛出的异常，也可以通过回调函数返回的错误状态码。当出现异常时，它不会继续执行，因为它已经被捕获了。事件队列中的下一个处理函数可以获得前者出错的消息——因此它可以决定下一步如何进行，或是什么也不做。重要的是错误已经被控制，并且已被其他组件知晓。

下面的例子展示了一个事件代理，它可以探测错误，并将此错误告知该事件的下一个回调函数。

```
// events.js
// 事件代理。
class Events {

    // 触发一个事件。
```

```
trigger(name, data) {
  if (name in this.listeners) {
    // 我们需要知道上一个处理器返回的结果，因此每个结果都在此处被保存。
    var previous = null;

    return this.listeners[name].map(function(callback) {
      var result;

      // 获得当前回调函数返回的结果。
      // 注意，这项操作被放置在一个异常处理器中。
      // 并且，每一个回调函数还可以接收上一个回调函数返回的结果。
      try {
        result = previous = callback(Object.assign({
          name: name
        }, data), previous);
      } catch (e) {
        // 如果回调函数出现了异常，那么这个异常会被返回，
        // 并被传给下一个回调函数。
        // 回调函数就是通过这种方式知晓前者是否执行失败的。
        result = previous = e;
      }

      return result;
    });
  }
}

var events = new Events();

export default events;

// main.js
import events from 'events.js';

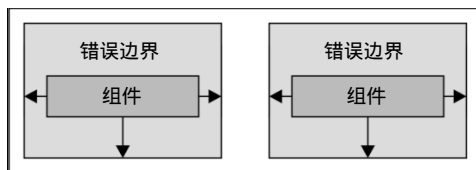
// 通过该方法，可从obj中读取并返回错误消息。
```

```
// 如果obj是一个异常，那我们可以返回它的message属性。
// 如果obj拥有一个error属性，我们可以返回它。
// 否则，可确定它不是一个错误，并返回"undefined"。
function getError(obj) {
    if (obj instanceof Error) {
        return obj.message;
    } else if (obj && obj.hasOwnProperty('error')) {
        return obj.error;
    }
}

// 这是第一个被执行的回调函数，因为它是第一个注册的。
// 它会随机地抛出错误。
events.listen('action', (data, previous) => {
    if (Math.round(Math.random())) {
        throw new Error('First callback failed randomly');
    } else {
        console.log('First callback succeeded');
    }
});

// 这是第二个回调函数。
// 它会检查上一个回调函数是否出错。
// 如果出错了，它会提前终止并返回错误。
// 否则，它会随机地抛出错误或成功返回。
events.listen('action', (data, previous) => {
    var error = getError(previous);
    if (error) {
        console.error(`Second callback failed: ${error}`);
        return previous;
    } else if (Math.round(Math.random())) {
        throw new Error('Second callback failed randomly');
    } else {
        console.log('Second callback succeeded');
    }
});
```

```
// 最后一个回调函数。  
// 它会检查上一个回调函数是否出错。  
// 要注意的是，之前的回调函数中，只可能有一个函数出错。  
// 因为如果第一个回调函数出错，那么第二个回调函数什么都不会做。  
events.listen('action', (data, previous) => {  
  var error = getError(previous);  
  if (error) {  
    console.error(`Third callback failed: ${error}`);  
    return previous;  
  } else {  
    console.log('Third callback succeeded');  
  }  
});  
  
events.trigger('action');
```



图注：通过有效地控制错误，可以避免一个组件引发的错误影响其他组件正常工作。

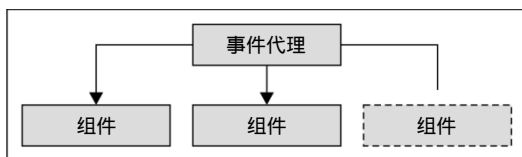
禁用出错组件

使整个组件快速失效，是为了避免出现更严重的错误。但是，如果某个组件完全独立于其他组件呢？我们可以尝试将其从错误中恢复，但这并不总是可行的——当出现 bug 时，恢复的唯一方法就是修复代码；但当无法恢复组件时，我们可以禁用它。

这么做有两个主要原因。第一，可降低出错组件拖累整个应用的几率。第二，禁用或隐藏组件，可避免用户使用它。这意味着由于用户不断地重复尝试而导致出现别的 bug 的几率会更小。这种情况理论上不会发生，因为这个组件是完全独立的。但是，我们无法保证设计万无一失。

禁用出错的组件并不会太糟糕，毕竟对用户来说也只是系统中的某个功能无法使用。我们可以在这个时间段内定位并修复出错的组件。

设计的关键问题是：应该由谁来禁用组件，是组件自身，还是某些负责探测错误的核心组件？一方面，由组件禁用自身是一个好想法，因为为了确保其他组件正常工作，禁用的过程可能有很多步骤。另一方面，如果使用类似事件代理的方法来禁用出错的组件，可以将错误处理集中在一处。我们应该选择最简单可行的方法。如果事件代理可以保险地完成这项任务，那么它应该就是我们的最佳选择。



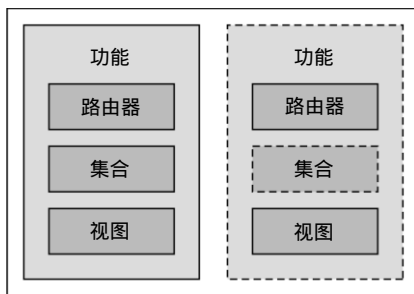
图注：被禁用的组件不会与系统的其他部分交互，因此降低了出错组件造成其他问题的几率。

优雅地降级功能

探测到错误时禁用组件是一回事。处理出错的组件并将其从 UI 中优雅地移除是另一回事。即使我们努力保持组件之间松耦合，但涉及到 DOM 时，这项工作会变得很有难度。例如，我们是否真的可以在不影响周边元素的情况下移除出错组件的 DOM 元素呢，或者这样做是否会更好——将元素保留在原处，只是将它们隐藏并禁用所有的 JavaScript 处理函数？

我们采取的实现方式取决于我们在构建什么，即应用的本质。有的应用采用合理的方式组合组件和布局页面，使得添加和移除功能甚为方便。不要以为视觉设计只是一块可以从系统中随意拆除的皮肤，不会带来任何后果。理论上说，它应该独立于系统的其他部分，但实际上，这种思想是不可行的。页面元素的布局对扩展是有影响的，例如，当组件出错时，我们是否能够在不影响系统其他部分的情况下，将他们禁用或修复。

我们可将处理出错组件的过程比喻成关机，因为这通常涉及一系列的操作——以便优雅降级。功能很少整体失效，通常是某个组件失效，导致功能不正常。例如，路由器出错并导致某个功能失效。此时，除了关闭失效的路由处理器，我们还需要关闭该功能涉及的一系列其他组件，将它们从 UI 中移除，并提示错误信息。我们应为任何功能考虑和测试这种关机流程。我们要保护的并不是功能本身，而是移除功能后系统的剩余部分。



图注：一个集合组件出错，导致整个功能都不可用，但是完整的应用还在正常运作。

故障恢复

在上一个部分，我们开始在前端代码中考虑容错。容错是指应用在某个组件出错后仍能够正常运作，至少在短期内。但如果应用无法从某些错误中恢复呢？为了持续满足用户，在探测到错误后，应该采取一些别的措施来代替禁用组件。

这一部分将介绍多种可让组件从失败操作中恢复的方法。例如，重新尝试操作，或者重启组件以清除错误状态。有时候，了解用户希望如何从错误中恢复也是个好办法。

重试失败操作

如果组件在执行某项操作时失败了，可以重新尝试。这项操作甚至不一定要属于组件。但由于组件依赖于此项操作，因此只要操作失败，那么组件也会出错。例如，一个后端 API 请求可能会失败，因而导致发送请求的组件处于一个未知状态。API 请求这种操作，适合在失败时进行重试。

不管我们重试的是一个 API 请求，还是涉及到其他组件的操作，都要保证这项操作是**幂等的**。意思是，在初次调用操作后，再次调操作不会有副作用。换句话说，就是多次成功调用操作不会对系统的其他部分产生负面影响。请求数据的操作（向 API 请求数据，同时不会改变后端资源的状态）适合在失败时被重新尝试。例如，如果获取数据的请求由于后端处理的时间太长而失败了（很可能是因为与其他用户争夺资源），我们可以再次尝试，并会快速获得结果。我们也许不愿继续等待，但可以进行安全的重试。下面的例子展示了一个模型，它会在请求失败时进行重试。

```
// api.js
// 通过返回一个promise对象模拟API请求。
function fetch() {
    return new Promise((resolve, reject) => {

        // 一秒后，随机地完成和拒绝promise对象。
        setTimeout(() => {
            if (Math.round(Math.random())) {
                resolve();
            } else {
                reject();
            }
        }, 1000);
    });
}

export default fetch;

// model.js
import fetch from 'api.js';

// 一个会请求API数据的实体模型。
export default class Model {

    // 初始化retries数和attempts计数器，会在请求失败时使用它们。
    constructor(retries = 3) {
        this.attempts = 0;
        this.retries = retries;
    }

    // 返回一个执行fetchExecutor()的promise对象，并且很可能会重试请求API。
    fetch() {
        return new Promise(this.fetchExecutor.bind(this));
    }

    fetchExecutor(resolve, reject) {
```

```
// 调用API并完成promise对象。同时重置attempts计数器
fetch().then(() => {
    this.attempts = 0;
    resolve();
}).catch(() => {
    // 再次发送API请求，除非在可以拒绝promise对象的情况下我们已经进行过太
    多次的重试。

    if (this.attempts++ < this.retries) {
        console.log('retrying', this.attempts);
        this.fetchExecutor(resolve, reject);
    } else {
        this.attempts = 0;
        reject(`Max fetch attempts
            ${this.retries} exceeded`);
    }
});
}

};

// main.js
import Model from 'model.js';

var model = new Model();

// 请求模型，并通过日志观察总共进行了多少次重试。
model.fetch()
    .then(() => {
        console.log('succeeded');
    })
    .catch((e) => {
        console.error(e);
    });
```



我们必须了解正在执行的操作类型和接收到的错误类型。例如，提交表单并创建资源的操作可能会出现很多种错误。如果在尝试执行此操作时，它返回了 503 错误码，就可以确定重试是安全的——因为后端资源没有被改动。另一方面，也可能接收到 500 错误码——在这种情况下，我们没办法知道在后端出现了什么错误。

对于获取数据的请求，无须操心它返回的是什么错误，因为我们不会改变任何东西的状态。因此，在重试操作之前，需要考虑操作的类型，如果这个操作会修改资源，那我们还需要考虑错误响应的类型。

重启组件

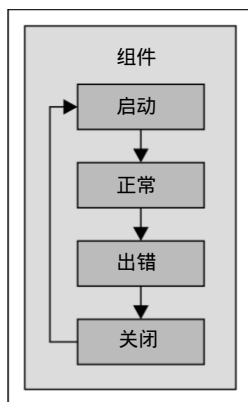
组件通常拥有生命周期——开启、关闭、和中间的很多阶段，它由组件的类型决定。通常情况下，组件的生命周期由创建开始，随着组件的运行，它的内部状态会发生改变。这些状态很可能就是导致组件之后出错的原因。

例如，如果组件处于忙碌状态，并且没有处理任何外部请求，那我们通常会在系统的其他部分寻找问题。可能组件的忙碌是合理的，也可能是某些原因导致组件被错误地卡在忙碌状态。如果是第二个原因，那么重启组件应该能解决所有问题，并且可以使组件再次回到运行状态，处理外部请求。

实际上，重启组件应该是从错误中恢复的最后一搏。走到这一步意味着我们不知道组件出错的原因，只知道有些功能没有正常工作，并对整个应用造成极大的破坏。在出现错误时重启组件的真正难点在于，清理了错误的内部状态后，组件需要重启并恢复到它出错的那个状态。如果某个组件保存着从后端获取的集合，那么将其重启之后，需要再次获取该集合，才能让组件回到当时的状态。

因此，在为组件设计重启功能之前，需要进行如下考虑。首先，应该在什么情况下重启组件？这通常要根据应用的具体情况而定，但大多数重启行为都集中在组件出错的边缘情况。当组件中存在一个 bug 时，重启它不见得有所帮助，但也不见得有所损害。另一个要考虑的方面，是对数据源的存储，即应用使用的数据，而不是内部状态。这是两个不同的东西——内部状态是由组件计算出来的，而数据则是由外部输入的源。

我们实现组件重启机制，不是为了掩饰代码中的其他问题。它是一个优秀的组件设计思想，迫使我们思考组件在运行环境中可能出现的各种情况。即使只是提出问题，也是有意义的——如果在运行时重启组件或使用一个新的实例代替它，会发生什么事情？我们可能永远不会去做这些事情，即使想做，也不一定可行。但是，通过这些练习，可以将组件设计得在这些场景中更具弹性。



图注：从较高层次看待组件的状态周期。

用户手动干涉

如果组件有能力重启并从错误中恢复，那我们也许希望让用户控制何时重启组件。例如，当组件出错时，我们可以禁用组件，并提示用户某些功能已出错，询问用户是否希望重启功能。

重试失败的操作也可以采取类似的方法——询问用户是否希望重试。当然，对于那些更普通的重试或重启操作，应该自动为用户完成。当用户明显希望某个操作执行成功，并且没有等太久时，我们不应该打扰他们，但这样会与我们及时响应的想法背道而驰。

我们可以设定自动重启/重试的次数上限，在达到上限之后，再去询问用户。例如，试图对 API 数据的请求超时了两次，这时用户很可能已经失去了耐心。因此，此时应该停止自动重试，并告诉用户出现的情况——应用无法从后端获得响应。应该继续重试还是停止？当组件遇到这种不确定的情况时，最好将决定权交给可能更具想法的用户。

不断地重启或重试对组件没什么影响，但前提是用户不介意。如果用户不希望继续，

会发生什么事情呢？此时用户已经被折磨够了，并且想采取更加确切有效的措施，而不是让应用继续重试下去。这种情况下，我们应该为用户提供引导。除了继续重试还可以采取什么操作？组件是否知道错误的原因，并可以将错误翻译成用户能看懂的信息？例如，也许用户可通过修改某个偏好设置来修复某个错误。此时，向用户展示一个友好的建议，可以引导他们更好地修复错误。

💡 最好将修复建议描述为也许可行，而不是绝对可行。这样可以帮你免除烦人的支持请求。

当我们无法从故障中恢复……

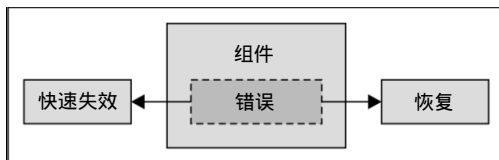
如果到了让用户手动干预的环节，应用仍未达到用户的预期，那我们就无能为力了。正如这部分标题所暗示的，并不是所有错误都可以被恢复。后端 API 并不总是能正确做出响应。组件在生产环境中是会存在 bug，有些 bug 甚至在很多年后才会被发现。

这种重大错误就像在人群中摔了个脸朝地。不断重试返回的是相同的结果，重启组件也不会有影响。此时向用户寻求输入是没意义的，或许用户不可能再重现失败的情形，或许我们根本就没有提供输入方式。

不管是哪种情况，解决的办法都是切换到快速失效模式——在异常情况下禁用组件或整个应用。如果只是禁用组件，就必须确保应用可以脱离该组件正常工作。这又回到了飞机使用一个引擎降落的例子——它是否能工作？如果不能，我们必须关闭整个应用。

这种解决方法可能乍一看有点激进。但是，这么做可以帮助支持小组排除一大部分麻烦。出错组件导致系统在运行中出现新错误的几率也会更小。

可扩展的错误处理方式能提高胜算，只要不在恢复错误时自作聪明，成功的机会就更大。



图注：组件的两种失效模式。采用哪种模式可在运行时决定，并不一定要预先决定。

性能和复杂度

既然我们已经拥有了强健的错误检测和恢复机制，那么是时候将注意力放到性能和复杂度的影响上了。在大型 JavaScript 应用中，任何行为都是有代价的——任何收获，都会给可扩展性带来新的挑战。错误处理也同样如此。

错误处理直接影响了性能和复杂度。软件出错的原因很奇特，当找不到优雅的处理方法时，我们需要复杂的实现修复它们。复杂的代码通常不利于性能。因此，让我们来看看，是什么拖慢了异常处理代码的性能。

异常处理

在处理 JavaScript 异常时，通常会捕获所有抛出的错误。不管错误是由我们抛出的，还是意料之外的，异常处理器都得找到处理错误的方法。比如，它应该禁用组件，还是重试操作？使用 `try/catch` 声明的好处是可以避免漏掉错误。否则，错误就会对其他组件带来副作用。

我们可以在事件代理中实现异常处理机制以避免错误穿越。这时，我们会将所有对事件回调函数的调用都放置在 `try/catch` 块中。通过这个方法，不管事件回调函数的结果是什么，异常处理函数都可以检测并处理异常。

但还有一个问题——异常处理器中的代码会损耗一定的性能。JavaScript 引擎会优化代码性能，但是某些代码是 JavaScript 引擎无法优化的，异常处理器就是其中之一。当调用栈中有很多层的异常处理器时，问题就会被放大。

这个影响有多明显，会带来多少延迟呢？这由应用的规模决定——组件越多，说明无法被优化的代码就越多。但通常，这不能决定应用的快慢。将错误处理放在事件代理层是一个合理的折中。虽然所有的代码都执行在 `try` 块中，但带来了很多好处——只有合理处理错误，我们才能快速前进。

在每个组件中添加嵌套的异常处理器很可能损耗更多性能并增加复杂度。例如，如果事件回调函数捕获了错误，但没有合理处理，这么做是弊大于利的。通常比较好的做法是在同一个地方统一捕获异常。如前所述，这样做仍然会带来性能损耗。当代码较高层级捕

获错误时，我们尚可处理，但我们绝对不希望在每个组件中都捕获错误，尤其当组件的数量还在不断增长时。

状态检查

除了异常处理，我们还应在执行操作前检查组件状态。如果当前状态下不适合执行某操作，就应该立刻停止，因为继续执行会导致出错。这像是一种提前的异常处理，在尝试执行任何操作之前，提前处理了所有潜在的问题，这种异常处理方式更为可行。

虽然组件自身的状态很简单，但当我们检查边缘情况时，除了要检查组件本身，通常还要检查其相关组件的状态。由于组件之间是松耦合的，因此通常不会直接检查其他组件，而是间接检查。例如，向主应用发起查询。这可能很复杂。并且，随着更多组件的加入，需要添加更多的状态检查，很可能导致已有的状态检查代码变得更加复杂。

使用 `if` 声明和一些附加的代码实现简单的状态检查是没问题的。但是随着测试失败，会出现越来越多的边缘情况，我们随之也在混乱中加入了越来越多的边缘情况处理器。如果将应用的状态看作一个整体，会发现这其实就是所有组件状态的总和。由于组件的数量庞大，每个组件都拥有独特的状态和操作限制，我们难以预测应用将会如何出错。一旦我们开始这么做，就很容易向系统引进更多的问题。这个代价就是复杂度的增加——错误处理带来了新的错误。



为了方便地处理错误，有一种方法可以降低组件状态检查的复杂度，就是以声明的方式将操作与必要条件绑定。例如，我们可以将操作名称与需检查的条件集合一一映射。这样便可以使用一套通用的机制，查看映射并决定操作是否可执行。在组件中使用这种一致的方法可以减少会带来问题的 `if` 声明。

通知其他组件

在设计 JavaScript 架构时，面对的另一个挑战，就是在组件相互解耦的系统中处理错误。我们希望组件之间低耦合，因为这意味着它们是可互换的，这有利于系统的构建和扩展。对于错误处理，这种分离可被看做出错组件与系统其余部分之间的一个安全网。这很棒，但除了正常的事件通信，还需要传递组件的错误信息。我们应如何完成这个工作，并

同时保持组件之间低耦合呢？

让我们通过事件代理——所有内部组件通信的指挥中心，来展开思考。如果它能传递所有组件事件，当然也能传递错误消息。比如说事件代理执行了一个回调函数，并且该函数抛出了异常。事件代理捕获了异常，并将该错误的详细信息作为参数之一传递给了事件的下一个回调函数。

通常情况下，回调函数会接收到一个错误参数，并对它进行检查，只需要一点小小的开销。如果该函数不在意之前发生的事情，就可以忽略此参数。或者，它也可以在接收到错误时，查看错误详情，并决定下一步行动。如果是 a 错误——检查 a 状态，否则那么做，等等。重点是组件之间能传递错误，因为如果我们不希望某个组件带来副作用，这时就需要在另一个组件中采取一些矫正操作，但我们首先得知道有这样一个错误。

记录日志和调试

在大型 JavaScript 应用中，提供正确信息是错误处理的一部分。很明显，我们应该从错误控制台展开工作，这里会记录未捕获的异常或使用 `console.error()` 生成的简单错误消息。有些错误消息可以帮助我们快速修复问题，但有些错误消息只会让我们不知所措。

除了记录出现的错误，还应记录可能导致出错的场景。即警告消息，但我们在前端应用中并没有充分地使用它。警告消息有助于诊断代码隐患，因为它提供了导致错误的一系列线索。

只要用户不打开开发者工具，就看不到这些日志。并且，用户通常也不会打开。通常，我们只为用户展示与他们相关的错误。并不会把所有错误日志都展示在他们眼前。在控制台打印了错误和警告后，我们应着手下一步了。

有意义的错误日志

有意义的错误日志对开发有很大帮助。错误消息的有效性直接决定了开发者能否在短时间内解决问题，这对可扩展性有一定的影响。如果错误消息未包含任何有用的信息，那么开发者将需要更多的时间来定位错误。我们能通过开发者工具追踪到错误的起点，但这仅仅是代码位置。我们需要更好的引导来定位错误。

有时错误消息模棱两可并不是什么大问题，只要追踪到错误在代码中的源头，很快就能找到原因。通常这些错误只是被忽略的边缘案例，通过几行代码就能修复。但有时候问题比较严重。例如，如果错误是由其他组件的副作用导致的，这是否意味着我们需要修复架构的设计，因为开发是建立在组件没有副作用的前提之下的。

举个例子：`Uncaught TypeError:component.action is not a function`。诊断这个问题需要很多工作——除非我们每天都与这些代码打交道，对它们已经非常熟悉了。可是，随着组件不断增加，应用也不断扩大，久而久之，我们无法将这些庞大的代码熟稔于心。平均下来，我们花费在每一段代码上的时间更少了，一旦程序出错，将很难被快速修复。但如果错误消息提供了有意义的信息，那就不一样了。比如把前面的错误消息改成这样：`ActionError: The "query" compnent does not support the "save" action`。

不可否认，为错误添加细节会增加代码的复杂度。然而，只要能合理地权衡是否添加错误细节，最终就能获益。例如，为永远都不会出现的场景添加错误检测和详细介绍是毫无意义的。我们只须关注最具价值的场景。如果这些场景很可能出错，那么与之相匹配的错误消息将有助于将其快速修复。

当代码快速失效时，应该抛出自定义异常。错误将被精确地展示在控制台上，为开发者提供有意义的信息，帮助他们诊断错误。抛出异常是快速失效的一种简单方法，因为一旦抛出异常，当前执行的代码堆栈便会马上停止运行。

为潜在故障发出警告

与错误消息不同的是，出现警告消息时系统仍然在正常运行，只是未处于最佳状态。举个例子，假设系统对某个集合的条目数有限制，那么当条目数接近限制数时，就可以发出警告——但代码的数量和复杂度也会随之增加。

既然我们已经拥有强健的错误处理机制，为什么还要发出警告呢？开发者工具终端通过高亮突出警告，这一点很棒。错误消息的出现往往代表系统中有部分功能失效，这不同于警告的目的。警告是为了提示当前的状态可能导致错误。例如，我们在加速发动机时，可能会发现转速指针进入了红色区域。这就是一种警告：如果继续当前的行为，可能会产生不良影响。

模棱两可的警告消息其实很有帮助，但对于错误消息，我们力求准确。警告消息要足

够通用，才能充分地说明应用状态。这意味着日志不会被不断重复的小消息塞满。否则警告将失去意义。通用的警告可以帮助我们诊断错误，它们通常是寻找错误的有利线索。如果为一个会打开开发者工具的用户排查错误，可以让他告诉我们警告消息。如果对方不会，则需要一个更友好的排错方法。

通知和指导用户

到目前为止，我们谈论的都是开发者工具的终端中展示的错误消息和警告消息。这些消息对用户来说并不重要。对于有必要让用户知晓的消息，我们应该将他们展示到界面上——总不能指望着用户总是打开开发者工具。前面提到的一些错误消息原则，也可以被应用到这儿。例如，出现错误时，除了在控制台提示错误，也应该通知用户。错误提示的准确性取决于我们。这时候必须考虑受众——告诉用户在某个方法被调用前某个组件的状态必须是这样或那样是没有用的。

但是，如果展示的是用户能够理解和使用的功能名称，而非程序中的错误名称，就有助于用户理解问题。他们将会知道什么出错了。通常用户并不在意出错的原因——就算知道又能如何？因此，我们最好提供有效的指引。这个功能出错了，您需要这样这样做。这样做很有必要，因为软件解决了很多问题，减少了人工干预的成本。同时还能让用户继续使用应用，这大大提高了可扩展性。

有时候，我们无法提供很好的指引。因为用户需要的功能崩溃了，而且我们无能为力。但是，我们仍然可以通知用户该功能已停止工作。在这种情况下，开发者工具控制台中展示的错误信息通常更有助于定位问题。但即便如此，也不应该在出现错误时忽略用户。因此，我们应对开发者和用户都照顾周全。

改进架构

为了保证架构能够扩展，应该实现强健的错误处理机制。但它的作用有限——一次次地处理相同的错误帮助不大。尽可能减少错误的出现，倒是更有用处。新组件通常会带来新的错误，因此我们需要通过解决旧错误，来抵消它们。

我们应该在设计过程中构思强健的错误处理机制，特别是在改进设计的过程中。需要做出的改动可能很微小，也可能巨大，这依赖于错误出现的频率、严重性和增长率。在考

虑到所有这些因素后，我们才能做出合理的权衡，继续下一步。

有很多实用的技巧。例如，我们可以坚持记录新的错误场景，将他们合理地分为关键与非关键组件。一如既往，保持简单。

记录错误场景

端到端测试是一个记录场景的好办法，特别是记录导致软件出错的场景。我们也许会在忙碌地设计和实现功能时突然想到一个场景，但端到端测试在复现生成环境中的真实错误时最有用。这些测试不仅能重现场景，便于我们确认错误已被修复，还能作为历史存档。

慢慢地，我们会积累越来越多的真实用例。例如，一些导致出错的真实用户操作。这会让软件实现得更加健壮。某种程度上，我们的软件设计从一开始就存在缺陷，需要通过端到端测试来验证。我们希望通过改进架构，保证一些错误不会发生。

假设应用在请求某个集合的数据时出错了，经过分析，我们发现原因是请求的参数不对。进一步检查，发现解析响应数据的方式也不正确——某些部分是静态的。这些属于架构层面的改进，因为它们对所有数据模型都是可用的，并且消灭了特定类型的错误。

改进组件分类

关键组件是核心应用中不可或缺的一部分——一旦出错，会导致整个应用崩溃。因此关键组件的数量并不多。这些关键组件可能影响所有组件，因此必须正常运行。非关键组件则与之不同，它们出错不一定会导致整个应用崩溃。非关键组件能从错误中恢复，保证应用正常运行。

虽然区分关键组件与非关键组件的标准相对稳定，但这并不是一成不变的。例如，假设系统中存在一个非关键组件，系统在这个组件出错时仍能正常运行。也许在过去，这样分类是合理的，但随着应用的变化，这个组件与其他所有组件都有着隐秘的联系，因此它变得很关键，不能出错。

关键组件是否会变得不关键呢？在这种时候，我们通常会将它从系统中完全移除，而不是将其降级为非关键组件。但是，我们一定要对关键组件有深刻的理解。这是架构设计中一个重点——有些组件不能出错。一旦出错，整个应用便会崩溃。在扩展应用的同时，我们要确保这一点。因此，在新增组件时，要确保能辨认出关键组件。

复杂导致出错

复杂的组件由很多部分组成，并与周围的环境紧密相连。当组件变得复杂时，内部会存在很多隐式状态，通常只有当组件出错时，我们才能注意到它们。我们确实无法掌控复杂的设计。当设计者自身无法掌控程序的设计时，更别提所有可能出现的错误了。

复杂会从两方面给我们带来不便。第一，容易导致出错。当组件的组成越来越复杂时，很可能会忽略平时显而易见的边缘情况。为了处理复杂的代码，我们需要加入更多的错误处理函数，这又会导致组件更加复杂，触发更多错误。如此循环，周而复始。

代码复杂带来的第二个难题是处理错误。例如，简单组件出错的原因很明显，即使之前忽略了，后来再补救，也能快速修复，因为我们要检查的代码很少。所以，代码简单能提高安全性，代码复杂时则不然。

小结

这一章介绍了大型 JavaScript 应用中的多种错误模式。快速失效模式是指探测到错误时，立即停止所有功能，以避免更多的损坏。我们通常会对应用的关键组件采用快速失效模式。

容错是架构设计中的一个重要机制，拥有容错机制的系统能够探测错误，并防止它们破坏常规工作。它在 JavaScript 中通常表现为捕获异常，避免影响其他组件。从错误中恢复组件的方式有很多种，包括重试操作、重启组件、清空错误状态。

错误处理会增加代码的复杂度，并且如果处理不当，会影响性能。为了避免以上问题，应尽量将组件实现得简单，不用维护状态，并且避免添加过多的异常处理代码。准确的错误消息能帮助开发者和用户更好地处理错误。我们最终的目标是通过错误改进设计，消灭所有问题。

大型可扩展的 JavaScript 并没有想象中的那么遥不可及。想要获得正确的答案，首先要找到正确的问题。希望这本书为大家提供了足够的知识，帮助大家解决扩展应用时可能遇到的问题。在合适的环境、合适的时间，找对限制扩展的原因，答案就会浮出水面。