
Broadview
www.broadview.com.cn
前端架构
从入门到微前端
黄峰达（Phodal）◎著
中国工信出版集团
電子工業出版社

前端架构
从入门到微前端

黄峰达（Phodal）◎著

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

内 容 简 介

本书是一本围绕前端架构的实施手册，从基础的架构规范，到如何设计前端架构，再到采用微前端架构拆分复杂的前端应用。本书通过系统地介绍前端架构世界的方方面面，来帮助前端工程师更好地进行系统设计。

前端架构包含以下五部分内容。

- 设计：讲述了架构设计的模式，以及设计和制定前端工作流。
- 基础：通过深入构建系统、单页面应用原理、前端知识体系等，来构建出完整的前端应用架构体系。
- 实施：通过与代码结构的方式，介绍如何在企业级应用中实施组件化架构、设计系统和前后端分离架构。
- 微前端：引入 6 种微前端的概念，以及如何划分、设计微前端应用，并展示了如何实现这 6 种微前端架构。
- 演进：提出更新、迁移、重构、重写、重新架构等架构演进方式，来帮助开发人员更好地设计演进式架构

本书适合想要成为优秀前端开发工程师（初中级），或致力于构建更易于维护的系统架构的开发人员、技术主管、软件架构师和软件项目经理等。

图书在版编目（CIP）数据

前端架构：从入门到微前端 / 黄峰达著. -- 北京：电子工业出版社，2019.5
ISBN 978-7-121-36534-8

Ⅰ. ①前… Ⅱ. ①黄… Ⅲ. ①程序设计 Ⅳ. ①TP311.1

中国版本图书馆 CIP 数据核字（2019）第 092407 号

责任编辑：董 英
印 刷：三河市君旺印务有限公司
装 订：三河市君旺印务有限公司
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：20.5 字数：410 千字
版 次：2019 年 5 月第 1 版
印 次：2019 年 5 月第 1 次印刷
定 价：79.00 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 zlts@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。

前　言

1. 关于前端架构

独立开发一个前端项目，或身为前端技术负责人，编写代码时，总得多一分考虑。不同的场景、不同的时间、不同的角色，考虑的东西也是不同的。在多种因素的共同作用下，我们往往找不到最好的方案，只能找到最合适的方案。于是，我们需要从几个不适合的方案里选出最合适的一个，换句话说，就是从不太差的中选出最好的那个。“最合适”，便是这样相比较出来的。

过去，当人们讨论架构时，往往指的是后端架构。对于只使用后端 API 的前端来看，后端看上去像只做 CRUD（增加（Create）、读取查询（Retrieve）、更新（Update）、删除（Delete））。然而，后端并不像看上去那么简单。从架构层面考虑，后端是要实现高并发和高可用的。在多数情况下，数据库是后端最大的瓶颈，存储的时候要考虑原子性、一致性、隔离性和持久性，使用的时候要考虑通过分表、存储、主从同步来提高性能和并发量，在这个过程中还要考虑备份、迁移、查询速度、效率等问题。此外，在代码实现上还有一系列的复杂问题：使用消息队列来解耦依赖，使用微服务来拆分单体应用……

架构，在前端会更复杂——涉及领域更广。前端在实现的过程中，除了考虑代码的可用性、性能、模型构建、组件复用等问题，还有前端特有的平台设定、浏览器兼容、交互设计、用户体验等相关的问题。而在“大前端”的背景之下，还需要深入移动端设计、桌面应用、物联网等相关的领域。

后端的发展比较稳定，更注重代码实现。而前端的发展比后端晚了好多年。大概从 Google Map 在 2005 年使用了大量的 Ajax 之后，人们才意识到原来前端还可以这么做。随

后，产生了单页面应用，并诞生了一个又一个的前端应用，直至出于解耦的目的，前后端也开始分离了。

Web 前端应用越来越复杂，使得架构在前端设计与开发也越来越受关注。规范、原则、模式、架构，是我们在前端架构中需要关注的内容。

2. 本书结构

本书从结构上分为 11 个章节，除第 9 章、第 10 章存在一定的依赖性外，每一章都可以独立阅读而不受影响。

第 1 章　前端架构。介绍什么是架构、如何进行架构设计，以及相应的架构设计原则。同时，还介绍了前端架构的发展历史，以及如何通过层次设计来设计前端架构。

第 2 章　项目中的技术架构实施。介绍软件开发过程与架构的关系是怎样的，如何正确地实施架构，以及如何在过程中提升团队的能力。

第 3 章　架构基础：工作流设计。从代码的角度出发，展示前端架构的基础内容：基础规范、文档化、通过流程化提高代码质量和测试策略。

第 4 章　架构基础：设计构建流。从构建工具（webpack、Gulp、Grunt 等）入手，讲述前端应用的构建流程、如何设计构建流、持续交付的构建等，它们用于构建完整的前端应用。

第 5 章　架构设计：多页面应用。介绍如何开发传统的多页面应用，并结合前端框架的知识体系，例如模板与模板引擎、双向绑定、前端路由等，讲解如何使用合适的技术栈开发多页面应用。

第 6 章　架构设计：单页面应用。关注如何开发单页面应用，相关的内容有前端 MV* 原理、前端应用如何选型、前端应用如何启动及服务端如何渲染。

第 7 章　架构设计：组件化架构。讲述组件化架构及其三种不同的展现形式：基础的风格指南、用于重用的模式库，以及进阶的设计系统。

第 8 章　架构设计：前后端分离架构。主要关注前后端分离及其 API 管理，介绍了不同的 API 管理模式、前后端独立的模拟 API 服务 MockServer，以及服务于前端的后端（BFF）。

第 9 章　架构设计：微前端架构。包括 6 种微前端的介绍，从经典的路由分发式到多框架运行的前端微服务化等。并介绍如何划分、设计微前端应用，以及“微”害架构的相关概念。

第 10 章　微前端实战。以实战的方式，对第 9 章中介绍的 6 种微前端方案进行实践。

第 11 章　架构演进：演进式架构。介绍 5 种架构演进的方式：更新、迁移、重构、重写、重新架构，以帮助开发人员更好地设计演进式架构。

3. 本书目标

本书的目标是带领读者探索前端架构世界。我们将关注于解决有关前端应用的基础架构问题：

- 如何快速启动一个项目？
- 如何在组织内有效地共享代码、组件库、函数库等？
- 如何提升项目的代码质量？
- 项目的文档化采用什么策略？
- 采用怎样的策略进行代码测试？

……

本书还将介绍，在进行更高级的架构设计时，需要考虑的诸多因素：

- 应用的哪些部分可以在其他应用中快速重用？
- 应用内的组件采用怎样的通信机制？
- 是否通过拆分应用来降低复杂度？
- 如何对架构进行演进，以降低开发、维护成本？
- 如何让多个团队高效地进行并行开发？能否将不同应用、项目的代码隔离开来，而不是所有的人工作在一个代码库上？

……

此外，这并不是一本纯理论性书籍。书中配套了大量的相关实践代码，它们可以帮助读者更好地理解架构的实践。

4 代码

本书相关的代码都可以从 GitHub 上下载：https://github.com/phodal/aofe.code。这些代码遵循 MIT 开源协议，读者可以将这些代码用在学习、商业等用途的项目中，而不需要笔者同意。同时，笔者也不对这些代码的衍生代码负责。

5 遇到问题

在遇到问题时，欢迎随时与笔者取得联系。遇到代码问题时，建议直接在 GitHub 上创建一个相关的 issue，以便帮助其他读者解决同样的问题。

遇到内容不清楚、代码相关等问题时，可以通过下面的方式联系笔者：

（1）GitHub 上的相关项目参与讨论：https://github.com/phodal/aofe.code。

（2）知乎、微博：[@phodal]（http://weibo.com/phodal）。

（3）邮件：h@phodal.com。

（4）微信公众号：phodal-weixin。

你也可以在知乎、SegmentFault 网站上进行提问，并@phodal 来帮助你解决问题。

6. 致谢

写一本书，初期靠的是激情，中后期靠的是耐心。编程亦如此，年少的时候，笔者进入这个行业靠的是比赛，长大后靠的是好奇心。17 年过去了，笔者仍然想着不断地提升自己，因为编程领域有太多的新事物值得我们去探索。

感谢电子工业出版社博文视点的支持。

感谢女朋友@花仲马，在写作期间给予笔者的支持——笔者才能在下班之后，腾出大量的时间用于写作、编程及设计。

编码的乐趣在于创建“轮子”，在撰写本书相关内容的时候，笔者也编写了一个专用的 Markdown 编辑器 Phodit（https://www.phodit.com/）。

目　录

1

第 1 章
前端架构

没有一种架构能满足未来的需求。

在软件的生命周期中，架构可以不断地优化，持续地变好，使得架构可以适用于当前的场景。所以架构是可以改变的，架构是需要变化的。

架构往往都是在一定的约束条件下设计出来的。在设计的时候，架构师总会担心设计出来的架构不能满足未来的需求，容易遭到后来人的吐槽。可受限于当前的时间、人力、财力、环境、能力，我们设计出来的架构，只是符合当前约束的架构。既然如此，我们就要以一种开放的心态来看待这个问题。未来的接任者，在经历了一定的练习之后，能带领团队演进出更好的架构。

架构也是分层级的，在不同的阶段里形式是不一样的，当面向不同的人群时，模式也是不一样的。所以每个人所理解的架构就会有所差异，那么到底什么是架构呢？

1.1 为什么需要软件架构

1.1.1 什么是软件架构

对于软件架构，不同的人有不同的理解，不同的组织有不同的定义。如维基百科上说：

软件架构是指软件系统的高级结构，以及创建这种结构和系统的约束。每个结构包括软件元素、元素之间的关系，以及元素和关系的属性。软件系统的架构是一种隐喻，类似于建筑物的体系结构。它作为系统和开发项目的蓝图，列出了设计团队必须执行的任务。

而在 IEEE（1471 2000）中，架构是指体现在它的组件中的一个系统的基本组织、组织之间的关系、组织与环境的关系及指导其设计和发展的原则。

又或者相似的其他定义，无一不在强调设计架构的整体及内部各个部分的关系。与此同时，还涉及实施过程中的原则和相关的设计实践。软件开发的过程就好比盖房子：

- ◎ 如果不能完成地基的建设，那么房子就无法继续往上盖。
- ◎ 开始盖房子时，便是在实施对应的架构。若地基有问题，房子就难以长期存在。
- ◎ 在浇筑钢筋水泥的过程中，若不注意，房屋就可能歪歪扭扭。
- ◎ 虽然房子盖好了，但是不注意后期的保养和维护，仍然会出现问题。

盖房子和软件开发是相似的，需要规划、设计、实施，而后才能使用。盖房子，最先设计的也是建筑的架构，有了建筑的蓝图，还需要考虑美观、实用、坚固等要素，从各种各样的材料中选择合适的，完成这一系列决策才能进入实施阶段。在实施的时候，还需要严格按照建筑的蓝图来进行施工，并保证施工的质量，才能造出所需的建筑。

对于软件开发来说也是相似的。最初，设计出软件的总体架构蓝图，思考各个模块之间的关系，实施一系列相关的架构决策。然后，选择软件开发所需要的一系列技术栈、框架等，讨论关于应用的上线、部署等流程问题。最后，才能进入软件的开发阶段。在开发的过程中，还需要保证软件的质量，才能设计出符合要求的系统。

而无论是建造房屋，还是开发软件架构，其中的一个重中之重的步骤便是结构设计。对于房屋来说是建筑结构，对于软件来说则是软件架构。

1.1.2　开发人员需要怎样的软件架构

不以实现为目的的架构，都是无意义的。因此，我们对系统架构的基本判断如下：

◎　一个无法上线的应用架构，算不上好的软件架构。

◎　一个没有人能完成开发的软件架构，算不上具有可行性的软件架构。

◎　一个在现有的技术上不可行的架构，算不上合理的软件架构。

所以，一旦我们谈及软件架构，需要讨论的第一个重点便是因地制宜。在 BAT 这一量级的公司里，他们实施的架构往往无法在小公司里照搬。这一量级的组织拥有大量的资源、基础设施，与之相匹配的优秀人才。若想实施相似的架构，小公司所需要的则是在时间上的长时间投入，或者有强力的技术人员，才能驱使系统往这个方面靠拢。其所依靠的是循序渐进的方式，只有这样才能完成架构目标。

除此，谈及软件架构的时候，还得有这么一些人——他们能按时、高质量（或者说有质量）地完成系统的设计。只凭一系列软件的架构蓝图，没有对应的实施者、维护者，那么系统就可能不会以预计的方式来构建；只凭一系列的架构蓝图，没有相应的实施规范和原则，便无法按我们预期的方式来实施项目。

为此，我们期望的软件架构，应该是贯穿在它被应用的生命周期里的，应该包含以下的内容：

◎　系统间关系。明确地指出该系统与其他系统之间的关系，是调用关系，还是依赖关系等。

◎　系统内关系。系统内各子系统之间的关系，如前端应用与后端应用，以怎样的方式通信，需要怎样的通信机制。

◎　应用内架构。包含应用相关的框架、组件，并清楚地表示出它们之间的关系。

◎　规范和原则。用于指导项目中的开发人员，编写出符合需求的代码，以构建出设计中的架构。

对于瀑布型项目而言，它还包含类级别的架构，即一个类应该提供怎样的接口，存在

怎样的继承关系。而对于敏捷型项目而言，它们则会在进入项目后才完成。这部分的内容，偏向于系统的详细设计。同理于前端应用，我们决定开发、使用哪些 UI 组件，便属于系统的抽象架构部分。而关于这个组件提供的接口，则偏向于组件的详细设计，但是它也属于要考虑的架构范围——但并非是必须考虑的范围。

1.2 架构的设计

架构设计并非只是一个技术工作，它包含了一系列复杂的工作，其范围包括软件工程、开发实践、业务交付等相关的领域。为此，在进行架构设计的时候，需要进行一系列技术及非技术相关的工作，如图 1-1 所示。

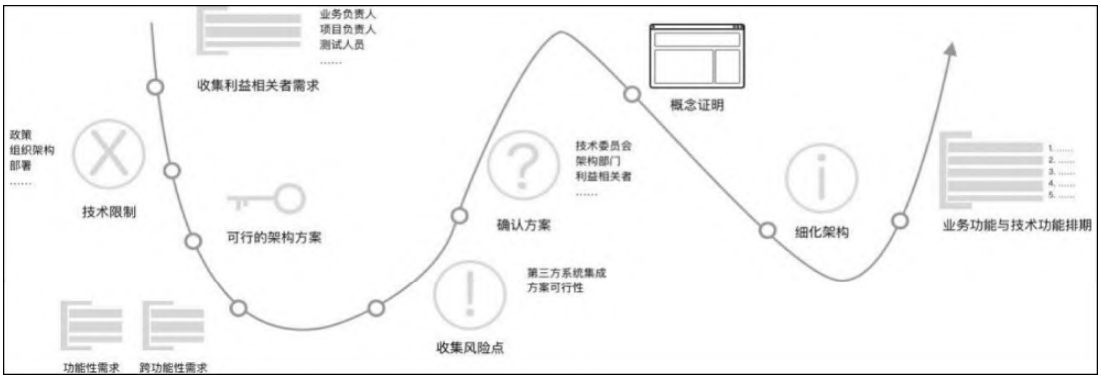

图 1-1

相应的步骤如下：

（1）收集利益相关者的需求。倾听业务人员、项目负责人等相关者的需求，进行用户访谈，收集相关的需求。

（2）与相应的技术人员（如开发人员、测试人员）讨论，了解架构上的潜在限制。

（3）寻找潜在的可行性技术方案。

（4）整理出功能列表中的功能性需求和跨功能性需求。

（5）找出会严重影响开发的风险点。

（6）和技术委员会、利益相关者反复确认方案（可选）。

（7）对架构设计进行概念证明。

（8）细化架构的部分实施细节。

（9）结合技术和业务，进行需求排期。

对于不同项目来说，上述步骤都会有所不同，有的可以直接忽略一些步骤，有的则会包含更多的步骤。在项目实施的过程中，如果有部分需求无法确认，就需要延迟决定。

在这个过程中，最重要的还是收集相关的架构需求。有了这些基本的信息之后，才能进行相关的技术决策。因为对于大部分项目来说，技术方案往往都是现成的。从过往经验、书籍、互联网、同学、同事等渠道，都可以获得一些不错的技术方案。只是每个项目因为需求的不同，所需要的技术方案也略有差异——往往需要整合多个方案，或者对某个方案进行改进，而需求能决定这些架构方案的细节。

1.2.1　收集架构需求

不得不强调的一点是，架构并非完全从技术角度来考虑问题。它需要从多方的利益出发，在满足利益相关者需求的同时，还要具备技术上的可实施性。为了寻找技术相关的因素，我们需要进行相关的讨论，收集一系列的相关资料：

◎　了解相关者的利益。

◎　寻找架构关注点。

◎　明确跨功能需求。

◎　罗列技术风险点。

有了这一系列的内容才能进一步明确，我们的架构需要解决什么问题。

1. 了解相关者的利益

架构设计的目的是解决一个应用的设计与实施，而应用则是为了满足利益相关者的需求而创建的。为此，在构建应用、设计应用架构的时候，需要考虑相关者的利益。这些利益相关者如图 1-2 所示。

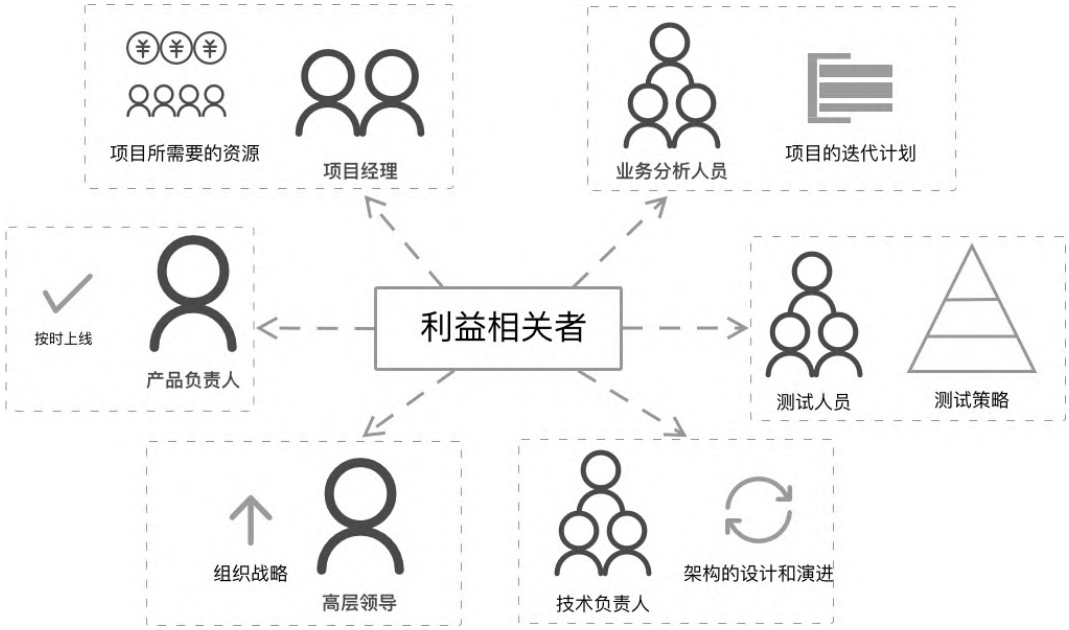

图 1-2

他们有各自的关注点：

◎ 产品或业务负责人（PO），关心是否能按时上线。

◎ 项目经理，根据架构来决定项目计划及项目人选。

◎ 架构师、开发人员，关心系统的构建、演进及维护。

◎ 业务分析人员，关心如何分配和安排项目的迭代计划。

◎ 测试人员，设计合理的测试计划，如对系统集成部分的测试等。

还可能存在级别更高的利益相关者，诸如创业公司的投资人，或者公司中的 CTO、CIO 等级别的人物——对于中大型公司而言，项目会因为战略目的由他们来发起，但不会有足够的精力投入相关的实践上。只是实施的过程中，我们需要听取他们对于项目的期望，以及与之相对应的预期（如战略上的预期）。

因此在设计架构的时候，需要考虑到各方的利益诉求，再坐到一起讨论出折中的方案。由于各方的利益诉求并不会一致，往往需要某一方做出一定的妥协。如业务方需要更快的页面响应速度，但受限于数据量大或者查询复杂，并不能给出对应的速度。那么，作为一

种折中方案，可能是用户体验设计师和前端开发人员，设计一些加载动画，来响应用户的行为。

尽管不能一一满足各方的所有诉求，但是满足了部分诉求之后，仍然是一个让人满意的架构——它让不同利益的相关者达成共识。至少在达成一致之后，我们可以继续往下设计系统。

2. 寻找架构关注点

在设计架构的过程中，各个组织、开发人员、架构师都拥有自己的关注点。在大部分的项目中，都会拥有相似的一些关注点，也因此它们会来自于其他项目的经验。常见的关注点如表 1-1 所示。

表 1-1

关注点	说明
性能	应用需要达到怎样的性能指标、可以实现多少用户的并发
安全	应用如何保障用户数据的安全、如何应对客户端的攻击、如何应对服务端的攻击
平台化	应用是否需要作为一个平台，来承载其他系统
代码维护	是不是稍有经验的开发人员都能快速上手
用户体验	用户体验是否比其他几个维度更重要

不同的项目因为自身的需求，对于各种特性的优先级考虑往往是不一样的。有的项目认为安全更重要，由安全带来的用户体验下降的问题是可以允许的；而有的项目认为用户体验更重要，也便能忍受其影响到其他特性。

我们在进行相应决策的时候，也需要和相关者进行讨论。如因安全问题导致用户的体验下降，而安全又是基线，则需要告知相应的利益相关者。在反复确认之后，我们便会得到这些关注点的优先级顺序。这个优先级，对于我们后续的工作非常有帮助，可以减少在设计上出现的返工。

3. 明确跨功能需求

按功能性来划分，需求可以分为功能性需求和跨功能性需求（又称为非功能性需求）。

功能性需求定义了一个软件系统或组件的功能，也是一个系统需提供的功能及服务[1]。而跨功能性需求也是需求的一个重要组成部分，它指的是依靠一些条件判断系统运作的情形或其特性，而不是针对系统特定行为的需求。这些非功能性需求一般是隐性的，往往难以直接观察得出。

因此在设计架构的过程中，我们需要详细列出实现跨功能需求的计划。它们从源头可以分为如下两类。

◎ 运行质量（Execution Qualities）：即可以在系统运作时观察到的质量，例如安全性、易用性等。

◎ 演进质量（Evolution Qualities）：它们体现在系统的静态结构中，例如软件可测试性、可维护性、可扩展性、可伸缩性（Scalability）等。

这些需求，有一部分是与性能（后端）指标相关的，例如同时支持的并发数，接受的宕机时间等。它们往往用各种“xx 性”来描述，举例如下。

◎ 可用性：是指在一段时间内，系统能够正常运行的概率。如何保证这样的可用性？

◎ 可维护性：其表现的指标是，在接到修改后，可以在相应的时间内（如小功能 1～3 天）修改完成。

◎ 可变性：在未来，应用的哪些地方会发生变化？是否需要提前做好准备？

◎ 容错性：系统中可能会出现哪些故障？如何确保这些故障不会让系统无法运行？

◎ 可伸缩性：当更大规模的用户使用系统时，哪些服务会出现问题？

多数时候，对于后端服务来说，它们只会间接影响架构的决策。但是对于前端应用来说，它们往往会影响应用的架构——部分相关的内容是一些兼容性、跨平台相关的需求，举例如下。

◎ 浏览器的支持范围：明确指出我们要针对哪些浏览器，以及浏览器的哪些版本做兼容。

◎ 移动端设备支持的版本：常见的设备如 Android 和 iOS，设备最低的支持版本，以及支持哪些设备制造商。

[1] https://zh.wikipedia.org/wiki/功能需求

例如，我们决定支持早期的 IE8 浏览器，那么就无法选择流行的前端框架——React、Angular、Vue，便不得不从早期的前端框架（如 jQuery、Knockout、Backbone）中选择。从这一点来看，它会严重影响前端系统的架构。如果我们决定支持 IE9 相关的浏览器，那么还需要花费一些时间来进行相关的适配。此外，在日常的开发中选择一些辅助的开发库时，也不得不考虑这样的因素，因为它也会为测试带来一定的工作量。

同样的问题也存在于客户端应用中，如在开发 Android 应用时，希望能支持 Android 4.4 以下的系统，同样也会大大地影响系统的架构。在低版本的 Android 系统中，缺乏一些系统特性，导致在实现的过程中需要有针对性地调整系统的架构。

注意：其中的部分跨功能性需求可能是由业务人员、产品经理提出来的，他可能对这些问题不是非常了解。对于这一类问题，要坚持以数据来说服人——要么提供不同浏览器、系统的占比，要么提供支持向下兼容的成本估算——真实的估算。

4. 罗列技术风险点

在设计架构的过程中，还需要寻找系统中的一系列技术风险点，并努力降低它们带来的风险。因为系统的风险部分，往往才是最影响应用开发的部分——不确定性意味着其带来的风险是未知的。常见的风险类型如下。

- 技术风险：如果在应用中某一个功能的实现比较复杂，或者团队缺乏某一领域的经验，如 AI，那么在实践的时候可能会影响系统的架构。
- 第三方系统集成：在大中型公司内部，这是一个令人头痛的问题。如果集成后端服务，那么只需要确认对方的 API 能否按时上线，并按照我们的规范编写即可。如果集成客户端的 SDK 或者组件，那么第三方要以 Web 或原生代码的形式集成到系统中。不论哪种方式，都需要提前进行技术方面的评审，并在上线前安排联调。
- 受限制的线上运行环境：在开发人员和运维人员部门分离的组织里，往往会在内部限制某些语言，如选定 Java 等语言，限制其他语言的使用。如果选定的是其他语言，如 JavaScript + Node.js，那么运维人员便难以维护，需要进行培训或者招聘新的运维人员。

◎ 与此同时，若是一些需求带来额外的技术膨胀，则会为按时完成应用的开发带来风险。比如，想要实现一个复杂的前端动画效果，经过评估可能需要两星期的开发时间，如果实现这个动画效果将无法按时上线。这时，要么努力去说服业务人员修改需求——适当地降低交互效果，要么将相应的功能排期靠后。

当系统中要大量地集成第三方系统时，系统的风险就会变大。对接第三方系统时，会涉及接口不一致，导致两边需要反复修改才能对接 API。对于这些系统的对接，我们往往只能估计一个大致的时间，并预留一些调试时间。在调试过程中，可能会出现一些意外，如人员休假、bug 不好修复等问题，导致出现一定的延迟，影响应用上线。而当出现大量的第三方系统时，总的延迟可能性就更高，风险也就越大。

1.2.2 架构模式

从计算机发明至今，软件架构从混沌时代的大泥球模式，发展到丰富多彩的多架构模式，已经积累了大量成熟的架构。也产生了一系列架构的开发方法，以及相应的架构模式。在日常的开发中，这些模式和方法都有相应的参考价值。不得不强调的是，这些方法和模式都有各自的场景，并不适合直接套用。

架构风格便是架构模式的展现形式。

1. 架构风格

架构风格是描述某一特定应用领域中系统组织方式的惯用模式。

应用的软件架构并非是凭空想象出来的，它们往往是在实践的基础上总结出来的。不同的行业有不同的特性，相关的架构模式也不同。对于某一类型、领域的应用来说，它们在架构上存在一些相似之处，这些相似之处在于组织系统的方式，也就是架构的风格。当然了，在类似的应用和领域会存在相似的架构风格。

在日常的开发中，常见的架构风格有：

◎ 分层风格。这是最常见的架构风格，它将系统按照水平切分的方式分成多个层。一个系统由多层组成，每层由多个模块组成。每一层为上层提供服务，并使用下层提供的功能。最为人所知的分层架构应用是 OSI 七层模型和 TCP/IP 五层模型，在开发后端服务的时候得到了广泛的应用。如在采用 Spring MVC 开发的后端应用中，Controller 层在接收后端请求时，将通过 Service 层向 DAO（Data Access

Object，数据访问对象）层请求数据，而不是直接向 DAO 层请求数据。

◎ MVC 架构风格。这种风格应用得相当广泛，它强调职责分离，将软件系统分为三个基本部分：模型（Model）、视图（View）和控制器（Controller）。由视图和控制器一起完成用户界面的功能，并设计一套变更机制，来保证用户界面与模型的一致性。它是一种常见的架构风格，在涉及图形用户界面时，往往都有它的身影，如前端应用、移动端应用等。

◎ 发布-订阅风格。这种风格又可以称为基于事件的架构风格，它是一种一对多的关系，一个对象状态发生改变时，所有订阅它的对象都会得到通知。这种架构风格带来的最大好处是，代码上的解耦。发布者不关心事件的订阅者，只需要在适当的时候发布相应的事件即可；而订阅者则依赖于事件，而非事件的发布者。在前端的日常开发中，为了解耦不同的 UI 组件的依赖，会经常采用这种模式。

◎ 管理和过滤器。这是一种适合于处理数据流的架构模式，它将每个步骤都封装在一个过滤器组件中，数据通过相邻过滤器之间的管道传输。最典型的管道-过滤器架构是 UNIX shell 的设计。在类 Unix 系统中，使用“|”作为管理符号，当我们需要编写复杂的 Shell 脚本来处理内容时，便会使用这个符号。诸如 ls-l|grep.jpg，便会先执行 ls-l 命令，再将结果交由 grep 程序，查找以.jpg 结尾的文件名。它也适用于相关的数据处理场景，如我们在采用 Hadoop、Spark 等编写数据处理相关的代码时，便会采用这种模式来编写。如前端框架的 Angular，也直接内置了管理（Pipe）系统。

除了上述几种风格，还有诸多的架构风格也比较常见，但是它们偏向于后端架构，与前端架构没有直接的关系，有兴趣的读者可以从本章后面的参考书籍中进行更详细的了解。

和设计模式一样，架构风格可以体现架构的一致性，提高人们对架构的可理解性，可以在新项目上实现重用。在设计架构之前，可以了解一些相关的架构风格，以便更好地设计架构。

1.2.3　架构设计方法

时至今日，我们已经拥有一系列架构设计的方法。这些架构的开发方法和开发流程，也和架构开发的模式有关。但是由于这些设计方法往往是为了解决复杂的问题而存在的，

因而对于一般的项目来说，充满大量冗余的步骤，并不很实用。而且，它会使开发人员只关注于工具的使用，而不是设计出合理的架构。

在这里，我们将简单地介绍一下 4 + 1 视图法及 TOGAF，以配合本章的其他部分来进行架构设计。与此同时，这两种方式，侧重于从系统的角度来考虑问题，若只是从前端的角度来考虑，未免有些大材小用。

1. 架构开发方法：4 + 1 视图法

每个项目都会有一张系统架构图，里面包含了系统的主要层级之间的关系，以及一些第三方系统之间的关系。这张图不仅会涉及系统的软件架构部分，还会涉及系统相关的部署内容。

我们可以通过开发、部署、流程相关的内容来了解系统的架构，也可以利用 Philippe Kruchten 提出的“RUP 4+1 视图方法”来进行相应部分的设计和架构探索，如图 1-3 所示。

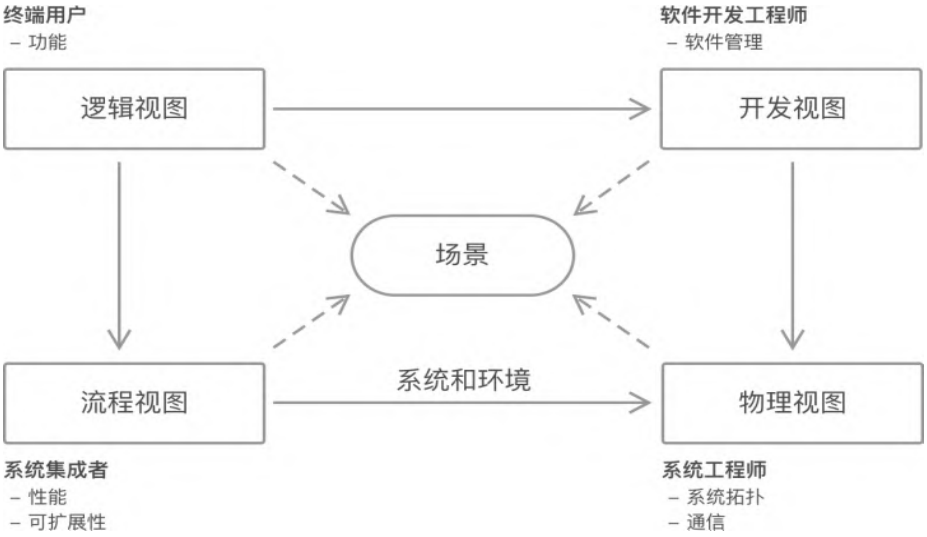

图 1-3

图 1-3 中的逻辑视图、流程视图、开发视图、物理视图和场景的含义如下。

◎ 逻辑视图（Logical View）：在设计期的模块、接口划分，职责及协作方式等。

◎ 流程视图（Process View）：在运行期运行的数据同步，如在微前端中的数据流、控制流等。

◎　开发视图（Development View）：在开发期的框架、库、技术选型及其对应的编译。

◎　物理视图（Physical View）：又称为部署视图。在部署期与持续交付相关的技术决策。

◎　场景（Scenarios），又称为用例视图，它使用一组用例或场景来说明架构。

从图 1-3 可以看出，4 + 1 视图法的设计流程如下：

（1）架构人员根据需求创建相应的逻辑架构，开发人员进行详细设计。

（2）架构人员和开发人员根据需求设计物理架构，再由开发人员根据物理架构进行对应的详细设计。

值得注意的是，4 + 1 视图法是倚靠软件建模和 OO 技术来进行架构设计的，尤其在逻辑架构设计方面更符合传统的瀑布模式，需要事先设计好各类接口，才能进入项目的开发阶段。这种烦琐的设计方式，既不适合于互联网应用，又不适合于前端应用。

尽管如此，这种 4 + 1 视图法的展现形式，也是我们需要的形式。即从四个不同的维度（逻辑架构、开发流程、部署架构、运行时）来考虑软件技术的设计。

2. 架构开发方法：TOGAF 及 ADM

针对大型的企业架构来说，可以采用 TOGAF（The Open Group Architecture Framework，开放组体系结构框架）的标准化方式来设计企业架构。从 1995 年发布第一个版本至今，已经发布了 9 个版本——在笔者写本书的时候，最新版本为 9.2 版本，该版本将企业架构分为如下 4 个架构域（Architecture Domains）。

◎　业务架构（Business Architecture）：定义业务战略、治理方法和关键业务流程。

◎　应用架构（Application Architecture）：为将要部署的各个应用程序提供蓝图，并展示它们的交互及与核心业务流程的关系。

◎　数据架构（Data Architecture）：描述了一个组织的逻辑、物理数据资产及数据管理资源的架构。

◎　技术架构（Technology Architecture）：定义了支持部署业务、数据和应用程序服务所需的逻辑软件和硬件功能，它包含了 IT 基础设施、中间件、网络、通信、处理和标准。

在没有足够的时间和精力的情况下，往往很难建立一个涵盖 4 个架构域的详尽的架构描述。此外，还有一套架构开发方法叫作 ADM，用于创建企业级架构。ADM 一共分为 8 个阶段，如图 1-4 所示。

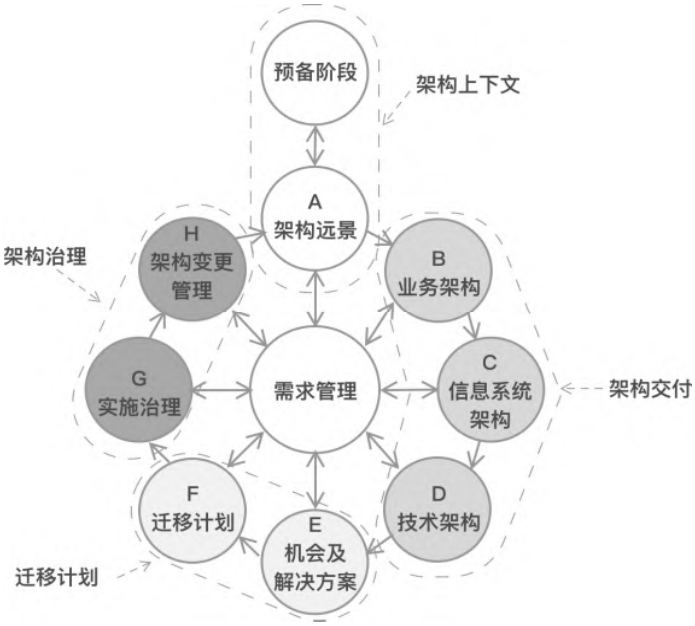

图 1-4

下面对 8 个阶段进行解释：

A. 架构愿景。设定项目的范围、约束和期望。

B. 业务架构。开发业务架构，用于支持设定的架构愿景。

C. 信息系统架构。开发信息系统架构，用于支持设定的架构愿景。

D. 技术架构。开发技术架构，用于支持设定的架构愿景。

E. 机会及解决方案。为前几个阶段定义的架构进行初步实施计划和交付工具的识别。

F. 迁移计划。通过最终确定的详细实施和迁移计划，阐述如何从基准体系结构迁移到目标体系结构。

G. 实施治理。准备和发布架构契约，并对实施的架构进行监督，以确保实施项目符合架构的要求。

H. 架构变更管理。对架构变更进行持续的监控。

每一阶段都有详细的步骤，及相应的产出规范。由于 ADM 是面向大型企业架构的，其过程和步骤比较复杂。有需要的读者可以阅读国际开放标准组织官方提供的文档（http://www.opengroup.org/togaf），以深入地了解实施方案。

从某种程度上说，TOGAF 是面向大型企业设计的架构方法，内容广泛而且过程烦琐。对于中小型企业及中小型应用来说过于复杂，需要按照自己的需求进行一定的裁剪。

3. 其他

事实上，现有的架构设计方法针对的都是大中型的后台应用——它们需要大量的领域建模、复杂的部署流程和服务调用关系。上述方式中的多个步骤和细节，对于前端应用而言并不存在。与此同时，在与后台一起设计架构的过程中，便会整合出前端所需要的内容。由于篇幅所限，这里就不展开相关的设计方法的讨论，仅做简单的介绍。

从笔者的经验来看，前端架构可以通过层次设计的方式来进行，即由顶至底进行一层一层的技术决策，再由底至顶逐层验证方案的可行性。关于这部分内容，我们将在本章的最后一节进行详细的介绍。

1.2.4　生成架构产出物

不论采用哪种架构设计方式，都需要留下相应的架构文档，它们将为团队开发打下基础。在进入开发阶段时，作为一个普通的开发人员希望得到的内容如下。

- ◎ 架构图：它包含了系统的整体架构，用于显示地告诉开发人员，它们是如何构成整个系统的，以及每个部分之间的关系。同时，显式地表明哪些部分是第三方系统，以及它们与该系统之间的关系。
- ◎ 迭代计划：按照业务和技术的要求，按时间顺序排列出项目的实施计划。由于其中也包含了上线时间，所以也可以从上线时间往前推算出迭代时间。开发流程：定义开发人员的工作方式，诸如采用敏捷还是瀑布的开发模式、何种源码方式（主分支、GitFlow 或者 Feature Branch（功能分支）工作流），必要时还要提前准备相应的工具和设备。然后，有针对性地对开发流程进行一定程度的裁剪，以真正满足团队的开发。

- ◎ 技术栈及选型：确定项目中使用的语言、框架、库等相关的技术栈，以及相应的依赖等。
- ◎ 示例代码：在这些代码中展示架构的风格及相应的设计规范。
- ◎ 测试策略：明确项目的测试类型、测试流程，以及相应的人员在哪些层级进行测试。
- ◎ 部署方式：定义应用的部署方式及相应的部署方案。

持续集成方案：描述系统的各个模块之间（如前后端）如何集成，以及采用怎样的时间和频率来集成相关的模块。

有了这些基本流程，开发人员就可以寻找合适的工具，开始搭建项目。反过来，也可以从我们所需要的部分反推出我们所需要的内容。

1.3 架构设计原则

不同的人在设计架构时会出现不同的风格，在细节的把握上也会出现特有的风格，这便是架构的设计原则。笔者根据自己的项目经验，总结了三个设计原则，如下。

- ◎ 不多也不少：不做多余的设计，也不缺少关键的部分。
- ◎ 演进式：不断地演进以使架构适应当前的环境。
- ◎ 持续性：长期的架构改进比什么都重要。

它们不是真实的架构需求，而是隐藏在背后的设计思想，也是不同人的设计价值观。

1.3.1 不多也不少

一个好的架构设计，内容不多也不少。增加新的元素，觉得有些多余；想删除某个元素，却又找不到更合适的。无论是艺术设计领域，还是计算机领域，原则都是相似且适用的。

设计过多则过度设计。即我们针对未来，提前准备好相应的设计。这些设计都是假想出来的，在实现的时候设计本身便不再适用。反而，我们还要解决之前的设计带来的问题，重新修改、删除原先的设计，它们会进一步增加项目的成本。不过，过度设计多出自规模

较大的项目，参与人数众多，某些地方如果不能进行预先设计，那么在实践过程中会遇到一系列问题。

设计过少则设计不足。设计不足会使架构扩展性不强，不能灵活地应对变化。值得一提的是，设计过多有可能是设计者的癖好或是炫技，而设计不足则可能是能力有限。

对于过度设计或者设计不足，往往很难做到真正的平衡及适宜。尤其在出现过其中一个问题之后，很容易往另外一个极端发展。如我们在某个项目里设计过度，在下一个相似的项目里，就可能会因为刻意减少设计而导致设计不足；相似的，如果在某个项目里缺少了设计，那么在下一个项目里，可能又会出现过度设计。

此外，我们还经常作两种假设，一是未来的人会比我们聪明，所以少做了一些设计；二是未来的接手者会比我们略逊一点点，那么就会为他们多做一些设计。

对于低级别的设计来说，应尽量避免进入细节设计，宝贵的资源应该投入更紧迫的业务开发中。对于高级别的设计来说，不预先设计模块和组件，仅从架构上考虑，即为未来预留扩展的空间。比如对于前端开发而言，为了将来在内部开发自己的组件库，当前可以通过适配者、代理模式对第三方组件进行二次封装。

如果在代码中为未来的代码预留一定的空间，那么大多是会产生问题的。因为多余的设计，会影响系统的后续扩展，并且在修改相关代码时，不敢放手去改，以满足现在的需求。比如在设计前端组件的过程中，想到未来会添加某些功能，便预留相关的接口，便有过度设计的嫌疑。

架构都只是适合当前的情况，一谈论到各种需求变化时，会发现架构设计上有各种不足，这并非是架构的问题。架构要不断根据需求演进变化，以满足新的需求。

1.3.2　演进式

适应环境能够生存下来的物种,并不是那些最强壮的，也不是那些最聪明的,而是那些对变化做出快速反应的。——达尔文

开发模式不同，软件架构的设计要求也不同：

◎　在瀑布模式下，往往会预先设计好系统所需要的一系列要素，进行软件建模、详细的架构设计、文档编写，直至设计的工作完成。在项目实施的过程中，会严格按照这些设计来执行。倘若架构有问题，那便是一个相当严重的事故。

◎ 在敏捷模式下，则是先有架构蓝图，实现对应架构的 PoC（概念验证）。然后在实现的过程中，基于 PoC 产生的代码叠加业务代码。接着，在项目实施的过程中，不断地完善应用架构的设计，如软件建模等。

这便是敏捷模式和瀑布模式的对比，采用敏捷模式是为了应对用户需求的不断变化。而需求的变化，则可能会和早期确认的方向不一致，与最初设计的架构不匹配。互联网应用都是敏捷型应用，需要应对不断变化的用户需求，及时改进和调整架构。

瀑布模式采用的是完全计划式设计，即在编码前，由顶层至底层进行的一系列架构设计，包含了各个模块、服务、函数和接口。而演进式架构，则是预先设计好重要的部分如模块功能划分、领域划分等粗粒度，随后在编码的过程中，再进行函数和接口的设计与实现。

事实上，瀑布模式也无法采用完全的计划式设计，因为无法事先对系统进行全面的了解。而即使是纯粹的敏捷项目，也无法采用完全的演进式架构，还是需要结合计划式设计。以软件开发和设计领域的专家 Martin Fowler 的观点来看，两者间最合理的分配是 20%的计划式设计，80%的演进式设计。这样的设计需要我们注意以下几个方面：

（1）在项目初期进行计划式设计，以确保架构能处理最大的风险。

（2）在迭代 0 进行初步的架构实践，编写示例的架构性代码——以将设计原则、风格融入项目中。

（3）在项目实施过程中，采用局部设计或演进式设计，以应对需求变化。

（4）配合重构、测试驱动设计与持续集成等敏捷实践，来驱动架构的实施，并防止架构腐烂。

在计划设计阶段，可以制定前后端分离架构的基本规范，如使用 JSON 作为 API 的数据格式，使用 RESTful 作为 API 的规范。同时，进行核心的技术选型，如选择后台服务的框架、数据库等。对于前端界面的框架，则可以在项目的迭代 0 里进行更进一步的技术选型。在项目实施的过程中，如果发现工具不合适，也可以在适当的时候尝试使用更好的工具，它需要我们以一种变化的心态来看待架构在项目中的意义。

相似的还有诸如 API 的设计，我们可以在计划设计阶段进行 API 的功能定义。而真正的接口定义——返回的字段、详细内容和值类型则属于详细设计，可以在后续实现的过程中加以完善。

1.3.3　持续性

从某种意义上来说，持续性和演进式有些类似。两者的区别是，演进式是指架构上的一些变化，而持续性针对的是开发人员的变化。架构的持续性原则的意图是，敢于修正架构中的错误部分，在修正的过程中尽管可能会带来一些不合适的中转式架构，但是也会很快被纠正过来。具体地，持续性包括以下几个方面：

技能水平的持续改进。随着代码的增多，架构也在不断地变化，也需要不断地被纠正，以符合现有的需求。这就要求团队里的成员既要懂得现有的架构，还要有能力来做一些改变。为此，团队成员需要在项目中进行相应的工程实践。在日常的开发中，使用敏捷方法实践相关的持续集成、持续部署。此外，还要关注与学习相关的技能和实践，以协助我们改善架构。如 Neal Ford 所说，开发中善于发现抽象与模式，并借助测试驱动开发，利用重构进行导向设计。并使用一些质量改善工具，用它们产生的指标来发现问题。

应用的持续改进。除了业务应用本身，互联网应用还需要一系列的配套服务日志、用户行为日志、性能监控等。如果以互联网公司的思维来看，这些事件都不会是一蹴而就的，“先上线，后解决问题”才是真理。在项目开始阶段，某些部分可以手动来完成。然后，在项目演进的过程中不断开发相应的功能，以满足自己的需求。

如常见的日志功能，开始时可以手动进行相关的查询。而后将其暴露到后台服务应用中，再有针对性地对数据进行展示。多数组织对于这些工具的完善是抱着一种持续改进的态度来进行的。

设计能力的持续提升。如果出于人力、物力所限，设计的架构不够合理，也没关系——人的能力是在不断提升的。若是有机会回到之前的场景，再慢慢思考这个问题，仍然有可能感觉当时的方案是最合适的。因此，对于方案而言，不存在最佳答案。建议大家在设计的时候为未来的改进留下一定的空间，以便于在未来隔离出这部分设计。

值得注意的是，关于架构相关的重要决定，还有一点很重要——延迟决策。如果架构上有多个可演进方向，无法做一个合适的决策，那么可以在条件更加充分的时候再做决策，而不是花费大量的时间盲目地修改架构，那样只会造成资源浪费。

1.4 前端架构发展史

最初，前端是没有架构的，因为功能简单的代码没有架构可言。通过操作 DOM 就能完成的工作，不需要复杂的设计模式和代码管理机制，也就不需要架构来支撑起应用。前端开发的发展历史分为以下几个阶段：

- 古典时期。由后端渲染出前端 HTML，用 Table 布局，用 CSS 进行简单的辅助。
- 动效时期。前端开始编写一些简单的 JavaScript 脚本来做动画效果，如轮播广告。
- Ajax 异步通信时期。2005 年，Google 在诸多 Web 应用中使用了异步通信技术如 Google 地图，开启了 Web 前端的一个新时代。

一旦前端应用需要从后端获取数据，就意味着前端应用在运行时是动态地渲染内容的，这便是 Model（模型）UI 层解耦。jQuery 能够提供 DOM 操作方法和模板引擎等。这时的开发人员需要做下面两个事情：

- 动态生成 HTML。由后端返回前端所需要的 HTML，再动态替换页面的 DOM 元素。早期的典型架构如 jQuery Mobile，事先在前端写好模板与渲染逻辑，用户的行为触发后台并返回对应的数据，来渲染文件。
- 模板分离。由后端用 API 返回前端所需要的 JSON 数据，再由前端来计算生成这些 HTML。前端的模板不再使用 HTML，而是使用诸如 Mustache 这样的模板引擎来渲染 HTML。

由于 HTML 的动态生成、模板的独立与分离，前端应用开始变得复杂。后端的 MVC 架构进一步影响了前端开发，便诞生了一系列早期的 MVC 框架，如 Backbone，Knockout，等等。

与此同时，在 Ryan Lienhart Dahl 等人开发了 Node.js 之后，前端的软件工程便不断地改善：

- 更好的构建工具。诞生了诸如 Grunt 和 Gulp 等构建工具。
- 包管理。产生了用于前端的包管理工具 Bower 和 NPM。
- 模块管理。也出现了 AMD、Common.js 等不同的模块管理方案。

随着单页面应用的流行，前后端分离架构也成为行业内的标准实践。由此，前端进入了一个新的时代，要考虑的内容也越来越多：

◎ API 管理，采用了诸如 Swagger 的 API 管理工具，各式的 Mock Server 也成为标准实践。

◎ 大前端，由前端来开发跨平台移动应用框架，采用诸如 Ionic、React Native、Flutter 等框架。

◎ 组件化，前端应用从此由一个个细小的组件结合而成，而不再是一个大的页面组件。

系统变得越来越复杂，架构在前端的作用也变得越来越重要。MVC 满足不了开发人员的需求，于是采用了组件化架构。而组件化+ MV*也无法应对大型的前端应用，微前端便又出现在我们的面前，它解决了以下问题：

◎ 跨框架。在一个页面上运行，可以同时使用多个前端框架。

◎ 应用拆分。将一个复杂的应用拆解为多个微小的应用，类似于微服务。

◎ 遗留系统迁移。让旧的前端框架，可以直接嵌入现有的应用运行。

复杂的前端应用发展了这么久，也出现了一系列需要演进的应用——考虑重写、迁移、重构，等等。

1.5　前端架构设计：层次设计

从笔者的角度来看，架构设计本身是分层级的，面向不同级别的人时所展示的内容也是不一样的。如我们作为一个架构师、技术人员，在面对同一级别、更高一级别的架构师、技术人员时，说的便是形而上学的东西，如微前端、前后端分离等各种概念。这些概念，对于接触过相关知识的程序员而言很容易理解。而当我们面对经验略微丰富的程序员的时候，说的可就不是：去实现微前端这样的东西，而是需要落实到怎样去做这样的一件事。

不同阶段构成架构的因素是不同的，基于这个思路，架构设计可以分为四个层级，如图 1-5 所示。

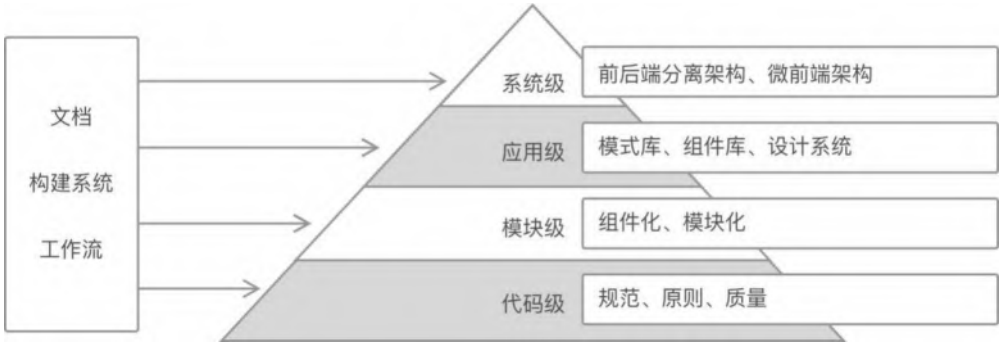

图 1-5

相应的层级解释如下：

◎ 系统级，即应用在整个系统内的关系，如与后台服务如何通信，与第三方系统如何集成。

◎ 应用级，即应用外部的整体架构，如多个应用之间如何共享组件、如何通信等。

◎ 模块级，即应用内部的模块架构，如代码的模块化、数据和状态的管理等。

◎ 代码级，即从基础设施来保障架构实施。

在设计的时候，既要用自上而下的方式来设计架构，又要用自下而上的方式来完善架构。从演进式设计的角度来看，我们需要在前期设计的时候，对所有系统级架构及部分应用级架构进行技术决策，而其余部分的架构则可以在实施的过程中考虑。

1.5.1 系统内架构

在设计前端架构的时候，首先考虑的是应用在整个系统中的位置——它与系统中的其他子系统是怎样的。这种关系包含了架构和业务上的关系，以及它们之间的协作机制。对于前端应用来说，这一部分的子系统包含了如下两方面内容：

◎ 其他前端应用。关注如何与这些应用进行交互、通信等。

◎ 对接的后台服务。关注如何与后台服务进行通信，诸如权限、授权、API 管理等。

如果是系统间的数据通信（如与后台服务之间的交互），那么只需要规定数据通信、数据传递的机制即可。这一类的通信机制，不仅包含了前后端分离架构的 API 设计，还包

含了各类的数据传递，如 OAuth 跳转的 Token 验证等。此外，对于传统的多页面应用来说，也需要关注其中的数据传递，如将 Cookie 作为用户凭据等。

因此，对于前端与后端的关系，我们所要考虑的主要因素是前后端分离架构的设计。

前后端分离架构其实是一个笼统的概念，它是指前后端分离如何实施的技术决策。它包含了一系列的决策、用户鉴权、API 接口管理与设计、API（契约）文档管理、Mock Server 使用、BFF（服务于前端的后端）、是否需要服务端渲染等。与此同时，当我们开发应用的时候，所涉及的并不只是我们所在团队的工作，往往还需要与其他团队，或者团队内的其他成员的沟通，才能与后端一起设计出整个前后端分离的方案。

当在一个系统内时，微前端是一个应用间的架构方案，当在多个应用间时，微前端则是一种系统间的架构方案。

微前端是将多个前端应用以某种形式结合到一起，当系统中存在多个前端应用（或者单个前端应用的模块较大）时，就需要考虑使用微前端的方式来拆分。此外，还需要做一系列的技术决策来支持应用的整合。

然后，我们还需要考虑前端的客户端展现形式。在大前端的背景下，它可能是 PC Web 应用、移动 Web 应用、混合移动应用（结合 Cordova 构架）、混合桌面应用（结合 Electron 框架）、原生移动应用（结合 React Native）等，具体选择哪一种技术，取决于我们先前调查的利益相关者的需求。当我们做完上述三个核心的架构决策之后，就需要考虑应用的部署架构了。有的客户端形式可能需要服务端渲染，会在某种程度上影响到前端应用的部署，但是总的影响并不大，往往只需要通过反向代理的配置就可以完成部署的配置。如果与后台服务不在一个域，则需要考虑支持跨域请求，或者让后台做一层代码。

有了这些基本的架构设定，便可以继续设计下一层级的应用架构。

1.5.2　应用级架构

应用级架构指的是，单个应用与外部应用的关系，如微服务架构下的多个应用的协作。它可以是一个团队下的多个前端应用，也可以是一个组织内的前端应用。其在组织中所处的位置，也进一步决定了我们所需要设计的架构方案。

若是从相关的定义来看，它与系统级应用存在一定的交集。但是，笔者将其视为系统级架构的进一步细化。比如，在系统架构的层级里，我们定义了微前端架构，而具体的实施细则则会放在各个应用中实现。至于应用间的数据如何交换，不同的应用有不同的实现

方式，通常是通过在相应的层级里定义相应的接口来实现。

由于各应用之间需要通过复用代码、共享组件库、统一设计等减少工作量，因此，我们要考虑以下几方面内容。

脚手架。作为一个基础的模块应用，脚手架用于快速生成、搭建前端应用。它除了包含一个前端项目所需要的要素，还包含着与组织内部相关的规范和模式，如部署模板、构建系统等。团队内的多个应用之间往往会使用同一套构建系统，该构建系统会包含在脚手架中。除了包含一系列的构建脚本，它还可以包含编写好的统一的 CLI 工具。比如在笔者曾经历的混合应用开发的项目里，就曾经通过编写 CLI 工具支持多个应用的开发。

模式库。它是一系列可复用代码的合集，如前端组件、通用的工具函数等。其作用是在多个应用之间共享代码，降低修改成本。在设计架构的时候，如果考虑内建相应的 UI 组件库，就需要考虑结合装饰器模式，将模式库作为一层代理来封装外部的 API，以降低后期的修改成本。模式库还包含了用于多个前端应用通信的数据通信库。

设计系统。它相当于更高级别的 UI 组件库，在这个层级里，我们关注抽象通用的 UI 模式，用于在多个系统之间共享设计。与模式库/组件库不同，设计系统偏向于设计人员的模式，而非开发人员的视角。

在应用级架构中，我们将决定选择哪种前端框架，并进行相关的技术选型。

1.5.3 模块级架构

模块级架构是深入单个应用内部、更细致地设计应用内部的架构，它所涉及的内容是我们在日常开发中经常接触的。我们所要做的是，制定一些规范或者更细致的架构设计。这部分内容会在我们开始业务编码之前进行设计。在敏捷软件开发中，它被称为迭代 0/Sprint 0/Iteration 0，其相关的内容有以下两个方面：

- ◎ 模块化。它包含了 CSS、JavaScript、HTML/模板的模块化。对于 JavaScript 或者模板而言，其模块化的设计受框架的影响比较深。对于 CSS 来说，我们也需要设计一个合理的方式来进行管理，既需要考虑全局样式以用于样式复用、局部样式以用于隔离变化、通用变量以方便修改，又需要考虑相应的工具来辅助设计。此外，还需要定义相应的 CSS、JavaScript、模块的代码组织方式。
- ◎ 组件化。它主要考虑的是，在应用内如何对组件进行封闭，以及相应的原则和粒度。

此外，对于不同的框架，还涉及一些框架特定的模式与架构设计，它们会在一定程度上影响单个应用的架构。对于不同的框架来说，所涉及的范围有所不同。如在 Angular 框架中，不需要关心相关的模式，只需要掌握框架定义的规范即可，如使用 Service 来保存应用的状态、使用 Pipe 来处理数据等。而在 React 框架中，则需要设计状态和数据流的管理方式，即需要诸如 Flux 或者 Redux 这样的状态管理方案。

1.5.4　代码级：规范与原则

当我们真正编码的时候，考虑的架构因素是更低层级的内容，这部分架构设计被称为代码级的架构设计，它关注于实践过程中的相关规范和原则。这部分内容相当多并且烦琐，包含但不限于下述内容：

开发流程。它包含了开发一个功能所需要的完整流程——从源码管理方式、代码合并方式、代码提交信息规范、代码规范自动化，到测试编写等一系列的过程。编写开发流程的目的是，保证编写、创建出来的代码能够符合项目的要求。

代码质量及改善。在实施过程中不仅要注重代码整洁，还要注重 TDD（测试驱动开发）等相关的实践，并且遵守 SOLID 原则，以保证代码的质量。此外，还需要制定代码的测试策略，测试的目的并非减少 bug，而是用测试来保证现有的功能是正确的。

规范而非默契。在整个架构中，我们会更关注规范化。小的团队可以依赖于默契，大的团队则需要规范。它需要我们关注几个方面：代码风格的统一，如统一化编辑器、IDE 等的配置、使用几个空格；代码的命名；如何保持一致性等。

此外，在日常的开发中，还需要注重代码的可维护性——简单的代码更容易被读懂和维护。笔者在维护一个 Scala 项目的过程中，就深有体会——越是写得抽象的代码，越难以维护。诸如函数式编程是一个好东西，但是好的东西也容易被滥用，导致人们不喜欢这个东西。

1.6　小结

本章首先介绍了什么是软件架构，以及为什么我们需要软件架构；然后介绍了如何设计软件架构——从架构的需求收集、简单地介绍了一些现有的架构风格、两种主流的架构

设计方法，到生成所需的架构产出物；接着介绍了三个基本的架构设计原则，它用于指导我们进行架构设计。好的架构设计是不多也不少的，它以演进式的方式不断演化。

本章的最后部分则关注于前端相关的架构设计。首先，介绍了前端架构在不同阶段的一些变化，即前端架构的发展史；然后，介绍了前端架构的一种设计方法——层级设计，我们从四个层级（系统内架构、应用级架构、模块化架构、代码级架构）的架构里了解了在每一个部分中我们所需要做的与架构相关的事情；最后，展开对架构的讨论，以此来熟悉本书的内容。[1]

1 参考书籍：《面向模式的软件架构》《架构实战：软件架构设计的过程》《恰如其分的软件架构》。

2

第 2 章

项目中的技术架构实施

成功的项目才是架构成功的证明，若是将一个失败的项目说成成功的架构案例，恐怕连门外汉都觉得不对了。成功的架构，只有设计是不行的，还需要有配套的流程和规范。既要在过程中保障实施质量，又要注重后期的维护。

在笔者经历的一些项目里，有的项目周期短，在 3～6 个月之间；有的项目周期长，在 1～2 年之间。短期项目和长期项目都有各自不同的特点。长期项目面临的主要挑战是团队的士气、能力的增长及架构的演进，而短期项目面临的主要挑战是技术实践与业务进度的冲突。即我们在追赶业务进度的同时，如何做好项目的技术实践。

无论是短期项目还是长期项目，它们都包含了相似的启动和实施周期，如图 2-1 所示。

图 2-1

这三个时期的工作内容介绍如下。

◎ 技术准备期：开始与架构实施相关的一系列工作，如搭建脚手架、测试及部署等。

◎ 业务回补期：填补第一个时期造成的业务进度落后的问题，以技术实践业务来证明技术的价值。

◎ 成长优化期：持续地对项目的技术和业务进行优化，以实现开发及业务人员的诉求。

在项目的不同时期，需要考虑的因素是不一样的，也会出现一定程度的偏差。关注这些因素，并对项目的架构负责的人，便是这个项目的技术负责人。每个时期都有不同的关注点，也都需要面临各种不同的挑战。

在本章中，我们将分析这三个时期的四个层级架构间的关系。

2.1 技术负责人与架构

优秀的软件开发人员，带有强烈的工匠精神，当遇到问题时，便会想着如何去解决它。时间久了，对软件系统就会有自己的思考，还会面临职业生涯的一个挑战，即要不要成为

一个 Tech Leader，也就是技术负责人。这时，优秀的软件开发人员就带着系统思维来考虑这个问题。这也是资深程序员和核心贡献者的一个区别，从更大的角度来考虑问题。

技术负责人是一个项目架构实施的保证。如果开发团队是房屋的建设团队，那么技术负责人 Tech Leader 便是带着施工队的队长，队长日常要做的事情有以下几方面：

◎　适当地平衡业务的进度与技术方案。

◎　解决重要、复杂的技术问题。

◎　帮助团队的其他成员成长。

◎　从全局考虑整个项目的技术和业务问题。

项目的技术负责人，可能是项目架构的最初设计者，也可能并没有参与项目架构的设计，他可能是在中途参加到项目中的。然而，一旦成为项目的技术负责人，就要开始对这个项目的架构负责。当然，我们并不需要为之前设计的架构负责，而是要为现在和将来的架构负责。如果过去的架构出现问题，那么要一点一点地去纠正。另外，还需要保证架构在项目中的成功实施。

作为一个技术负责人，当我们设计软件架构的时候，考虑的不仅是架构技术的方案，还需要包含如下内容：

（1）技术方案的设计。

（2）技术方案的落地。

（3）保证技术方案的实施。

（4）确保技术方案的上线。

（5）关注技术方案的后续维护。

因此，在整个系统中，架构方案只是其中的一个环节，或许是最不起眼的环节。因为有一定经验的程序员设计出来的应用架构和优秀的程序员设计出来的应用架构，相差也不是很远。甚至，有可能因为他们的见多识广，反而能设计出更好的架构。只是设计出来的架构，可能因为能力的问题而不能落地，或者无法维持到最后。

对于整个系统而言，技术负责人需要保证架构的成功实施，并使它能持续演进下去。在项目实施的过程中，有三个要素。第一，保证项目在开发过程中的质量；第二，提升人员的能力；第三，确保功能和应用上线。这三个要素的每一个要素都是令人头疼的，而我

们要保证这三个要素的同时推进，以确保项目和架构的成功。为此，我们还需要不断地在三者之间做一个平衡。

幸运的是，在项目的实施过程中，质量、能力和进度并不是同时展开的，而是按照一定的先后顺序展开的。

2.2 技术准备期：探索技术架构

这个阶段的业务与技术的优先级是，技术第一，业务第二。在这一个时期里，我们关注的三个点是：

- 架构设计，即设计系统的架构。
- 概念验证，验证先前设计的架构是否可行。
- 迭代 0，搭建系统的基础设施。

这三点都是与强技术相关的内容，需要花费大量的时间。

2.2.1 架构设计

当开始一个新的项目时，我们便要做大量的技术准备工作——搜索相关的技术资料，评估相关的技术方案等。这些在第 1 章中介绍了，这里不再赘述。

2.2.2 概念验证：架构的原型证明

在完成架构的设计之后，我们需要从技术上证明它的可行性。这个可行性的阶段，称为概念验证（PoC）：

概念验证（Proof of Concept，简称 PoC）是对某些想法的一个较短而不完整的实现，以证明其可行性，示范其原理，其目的是验证一些概念或理论。概念验证，通常被认为是一个有里程碑意义的实现原理。[wiki_poc]

在这个概念验证阶段里，我们所实践的是对架构理论的探索。比如我们选择了微服务、微前端等新技术，又或者 GraphQL 等新框架，并且尝试在这个项目中使用。那么我们就

需要去验证它，看看它是否能真正地被用上？

为此，我们需要使用这些新技术编写一个简单的“Hello,World!”，将我们所设计的各个部分串联到一起，构建一个完整的、基础的系统。如果我们计划使用微前端技术结合 React + Angular 框架开发的应用，那么第一步便是将空白的 React 和 Angular 应用结合到一起。从概念上证明它是可行的，下一步才是将现有的应用进行整合。

当我们预先设想的技术和架构不能应用时，我们应该采用原有的系统架构，还是重新设计一个合理的架构，这是一种考验，在这个时候到底第一优先级是什么，是技术、业绩还是业务？然而不可避免的是，我们又得花费大量的时间在一个新的概念验证上。

因此，在尝试新的架构和设计之前，请务必先在业余时间有所实践，再拿到项目中使用，这也是笔者所推荐的模式。直接在项目中使用的弊端，便是在时间上的浪费。项目上每多一个人，这个浪费的成本就会扩大一些。笔者就曾经在项目上经历过这样的事件，我们花费了两三周的时间来证明四种不同的技术方案的优缺点。由于架构未定，其他成员也不能编写相应的业务代码，只能尝试练习相关的东西，搭建持续集成等，直到完成相应的架构证明。

在这个概念阶段，我们并不会进行业务代码的编写，只给出简单的架构相关示例。

2.2.3　迭代 0：搭建完整环境

完成概念验证之后，就开始了迭代 0，以完成项目配套的技术准备工作。

迭代 0（Sprint 0），又名为 I0（Iteration 0），从名称就可以发现它与敏捷软件开发的关系。当我们开始一个项目的时候，进入的是迭代 1 的开发，那么迭代 0 呢？我们可以将其视为在所有迭代之前的准备工作。尽管我们从概念上将迭代 0 与概念验证阶段划分开来，但是在这种定义之下，如果项目不复杂，那么概念验证阶段会被列入迭代 0 的工作范围中。而对于复杂的项目而言，概念验证阶段则会独立于迭代 0。不过，在人力充足的情况下，会有一部分人进入与迭代 0 相关的工作。

在迭代 0，我们所要做的基本事项有：

◎　创建应用脚手架。

◎　创建项目的代码库。

◎　搭建持续集成、持续交付。

◎ 进行各种权限配置，如各种不同的环境账号准备、开发人员的账号配置等。

◎ 配置配套的工具，如代码审查、自动化原生应用上传等。

◎ 更细粒度的技术选型。

除了技术准备工作，迭代 0 还需要进行内部的技术培训——只是简单的技术培训，用于介绍系统的架构，开发注意事项等。当然，即使已经有了相应的培训，还需要准备基础的架构方面的文档，以及必要的一些规范。

此外，在概念验证阶段，我们编写的是粒度较粗的代码，并且为了追求效率和验证，可能有大量的 Code Smell（代码坏味道），诸如注释的代码、未使用的代码、不经测试的代码，等等。我们还要进行部署的准备工作，尝试进行第一次测试环境的部署。

这个阶段结束的标志是：项目成员可以进行正常的项目开发，并且此时的开发方式和未来没有太大的区别。

2.2.4 示例项目代码：体现规范与原则

在大部分项目中，经验丰富的开发人员往往只是少数，有些经验丰富的开发人员会分配到重要的项目上，成为新团队的技术负责人，或者某个团队的骨干。因此，除了这些经验丰富的开发人员，往往还存在一定数量的缺少编程经验的程序员。对于这些“年轻”的程序员来说，需要有人能指点其编写代码，以提升团队的平均技术水平，使之能按时完成任务。

为了有针对性地规范代码，并帮助其他成员了解代码，一个相应的举措便是编写相应的项目示例代码。通过这些示例代码，可以展现好的编程模式、范式，将它们融入项目中。有了这样一个基本的雏形，哪怕是刚毕业的学生，也能照猫画虎地编写业务代码。

例如，对于前端页面来说，登录的功能，便是一个相对比较完整的示例。它涉及一系列与前端相关的内容：状态管理、网络请求、数据模式、表单提交、UI 交互等。其中每一个小部分，几乎都是我们在第 1 章中提到的代码级架构。在日常编码中，对于这些基本的东西更要花费精力。稍有不慎，就会与我们最初设计的风格相去甚远。

2.3　业务回补期：应对第一次 Deadline

当团队成员能正常地开发业务逻辑时，我们的关注点就变成了：应对第一次 Deadline。在这个阶段里，业务开发虽然已经进入正轨，但是可能存在一定的进度落后。于是在优先级上变成了业务第一、技术第二。

这是一个价值证明的阶段，也是调配落后的进度与先进生产力的时期。毕竟欠下的债（业务）总是要还的。如果在上一个阶段中，我们花费了大量的时间在概念证明上，那么必须在这个阶段偿还这些时间。

在项目开发的过程中，人员能力不足也是大部分项目都会遇到的问题。就好像带兵打仗，老兵并不会一直待着，总有走的时候。而随着团队的规模不断扩大，会不断添加新的成员。若是这些新成员的能力不提升，或者提升比较慢，就会影响项目的进度。

此外，由于内部技术的问题已经解决了，内部的业务已经步入正轨，我们开始关注于第三方系统集成，并着手准备应用的测试和第一次上线。

2.3.1　追补业务

当技术不再是问题时，关注点便在业务上。虽然技术是业务价值的实现方式，但是业务才是赚钱的直接证明。如果业务无法存活下去，那么技术就无法证明其价值。

如果因为先前的概念验证导致一定的进度落后，那么在适当的时候，我们还会临时性地后置部分的技术需求，以用于按时完成业务。比如在必要的时候，单元测试、UI 测试都可以适当地减少，直到业务平衡之后，再回来填补测试。每当这个时候，总会有人为此感到困惑和不理解。

先进的架构，并不一定会为业务带来价值；先进的技术，也并不一定会为业务带来价值。这就是为什么每当我们采用新的架构和技术时，总需要通过一系列的会议来讨论新的架构是否能够带来更多的价值。新的架构和技术带来的往往是一些架构上的可能性，它节省的往往是开发时间。而实施这些新的架构，也会花费一定的开发时间，因此需要适当地计算收益，以实现业务的最大价值。如此一来，利益相关者才会考虑架构的价值。在大多数时候，业务人员考虑的是业务价值，进度是他们考虑的第一因素。

2.3.2 测试：实践测试策略

在编写业务代码的过程中，还要不断面临测试上的压力。

在敏捷项目里，测试人员真正开始测试，是在项目进行一段时间后才开始的。一方面，前后端之间的联调可能没有完全准备好，无法进行完整的测试。另一方面，可测试的业务内容相对比较少，部分功能可能无法完整地串联起来。随后，便一直在项目中进行日常的功能测试。在上线前，才会进行一次相关的、完整的回归测试。

对于瀑布型项目而言，一个项目的测试和其他项目的测试并没有太大区别。只是在新的项目中，往往也需要大量地准备、熟悉项目、编写测试案例等，才能进入测试的流程。因此，测试人员往往在上线前的几周里才开始进行大规模的测试。

日常的测试，除了带来更稳定的功能，还给开发人员带来 bug，这些 bug 会进一步地影响项目的进度。随着上线时间的逼近，bug 数量在不断增加，甚至可能会出现一些混乱的场面——测试人员在不断地提 bug，而开发人员需要不断地完善新业务，导致出现一些矛盾。

对于一个前后端分离的项目，在平时的开发里已经进行了大量的联调，基本可以应对上线。只是我们在设计架构时考虑的一些情况，可能会在这个阶段里出现；一些尚未考虑的情况，可能也会在这个阶段里出现。诸如：

◎ 没有进行集成第三方测试，无法验证业务的完整性。

◎ 没有准备好一个稳定的测试环境，以提供给测试人员进行测试。

◎ 没有接近真实的数据，以验证一些极端条件是否会出错。

……

每出现一个导致测试不完整的情况，都相当于一个风险问题。

此外，在这个时期和上个时期里，开发人员编写的自动化测试（单元测试、UI 测试等）的覆盖率比较低——大量的时间被花费在相关业务功能的实现上。尽管如此，我们还需要尝试在这个时期里，定下一个覆盖率的基值，比如 30%，先从 0 开始，然后在下一个阶段里进入更高的数值——就当前而言，它不能变得更差了。这样在后期，我们才有机会进一步提升代码的质量。

这一系列的混乱会随着项目的进行被不断地改进。

2.3.3　上线准备

在追补业务的过程中，我们在另外一方面着手准备了应用的上线事宜。当第三方依赖已经准备好时，便可以着手准备与上线相关的事宜。小公司要购买相应的服务器，大公司要申请相应的资源、审批流程，等等。

如果项目依赖于第三方服务和平台，并且他们的上线周期与我们的上线时间接近。那么，这个时候与他们联合进行调试就是一种挑战。特别是上线日期临近时，如果遇上 Bug，那么就需要反反复复地修改和测试。

与此同时，需要根据部署架构练习相关的技术实践。若是计划使用服务端渲染，那么我们需要准备好与 Node.js 相关的线上调试环境，并让团队拥有基本的调错（debug）能力。若是计划使用 Docker，那么也需要不断尝试练习与 DevOps 相关的技能——哪怕是公司内部拥有相关的 DevOps，开发人员也需要拥有基本的能力，才能应对线上的问题。

总之，我们需要在上线前准备好所有相关的内容。

2.3.4　第一次部署：验证部署架构

无论怎样，第一次部署都会比较痛苦。哪怕在我们有了其他项目经验之后，一旦一段时间内没有经历过新应用的部署，就有可能出现问题——通常是一些配置问题，比如某个软件升级，导致之前的配置出错。哪怕使用 Docker 这样的工具，也可能出现一些意想不到的情况。应对这些情况的最好方式是，如果能提前尝试使用上线流程，那么就提前走上线流程。

对于前端项目来说，部署并不是一个痛苦的事情。大部分应用只是单纯的静态文件，可以直接打包交给后端人员上线，也可以结合 Docker 进行自动化部署。若是带有服务端渲染的前端应用，部署会稍微麻烦一些，还需要配套对应的进程管理、服务监控等一系列的工具。

不管怎样，在第一次上线之后，我们实现了从 0 到 1 的阶段。有了这一次的经验，往后基本不会出现太多的问题——但是仍有可能出现问题。

2.3.5 提升团队能力

受团队能力影响，越是进度困难的时候，越需要提升团队的能力。它可以避免我们陷入一个误区，即我们因为能力不足而加班，却没有时间提升能力，又进一步导致加班。一旦团队里出现这样的问题，我们就不得不正视这个问题。尽管能力可以通过个人的练习得来，但是通过参与项目的方式来提升能力，则是一种更高速、有效的方式。

提升团队能力的方式有很多，但是受限于业务进度，我们需要选择一种合适的方式，在时间有限的情况下发挥出更好的效果。在项目实施的过程中，相关能力的提升方式如图 2-2 所示。

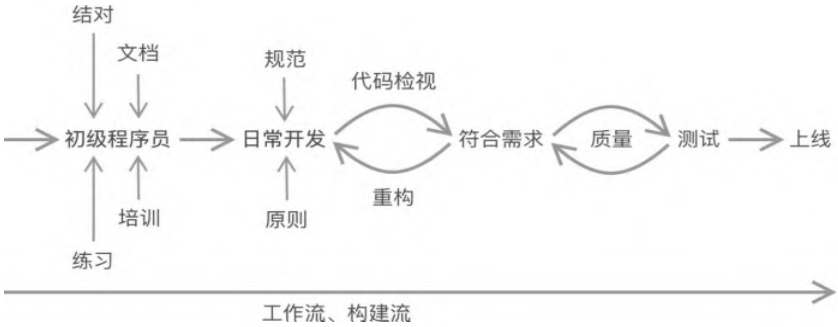

图 2-2

日常培训的过程如下：

（1）通过进入日常开发之前的培训、阅读文档等一系列的方式，来帮助他们培养基本的能力。

（2）进入开发后，则通过代码检视、代码规范及原则等一系列的方式，来帮助他们写出符合需求的代码。

（3）在项目开发过程中，我们可能会通过结对编程的方式，来帮助他们更好地成长。

如图 2-2 所示，在这个阶段里，我们提升能力的目标是，让每个人在完成功能的同时，还要保证质量。

1. 技术分享（Sessions)

能力提升有多种方式，其中使用得最多的招式便是技术分享。继续按照是和否的分法，

我们依旧可以将其分为两类：

- ◎ 项目相关的技术分享。它所围绕的是项目所使用到的技术。
- ◎ 非项目相关的技术分享。做一些项目相关的技术栈，或者完全以项目为主的技术分享。

对于非项目相关的技术分享来说，主要目的在于，知道有这样一件事，以及它能做些什么。对于绝大部分的人来说，仅仅是听半小时、一小时的分享，是不可能掌握相关的技能的。整个分享结束，掌握得最好的人，便是那个做技术分享的人。因此，就结果而言，做分享的人是最能学到东西的。我们也往往会将相关的分享，交给团队的新人来做。

当我们做一个技术分享的时候，必然需要准备一系列的资料，还要反复加工相关的东西，这样才能向其他人介绍相关的技术。如果团队中有技术经验丰富的人，就能指出讲述过程中的一些错误。做分享也能加深双方的印象。此外，技术分享还能帮助新人练习一系列非技术相关的技能，诸如表达、资料搜集，等等。

如果想在项目内制定分享的机制，需要人人都进行技术分享，那么就要有相应的鼓励政策。此外，需要注意时间和地点的选择。若是分享可以在上班时间完成，就应该在上班时间完成。否则，团队中的大部分人会觉得这是一种累赘，觉得分享是一个不好的事情。

通常开启技术分享的人是项目的技术负责人。在项目启动阶段，介绍一系列与项目相关的内容和知识。对于这种类型的分享，一般在时间长度上控制在 30～60 分钟之间。除了介绍理论，还需要配合实践示例，以加深大家的印象。

2. 工作坊（Workshop）

对于听的人而言，听别人讲与技术相关的内容，哪怕再有兴趣，一段时间后，我们也会忘得一干二净——这大抵是人的本能吧。平时工作中用不到的内容，我们早晚会忘记的。如果觉得一个技术有用，需要用更有效的方式来学习和练习技术，例如工作坊（Workshop）。

工作坊是一个以练习为主，以理论为辅的掌握新技术的方式。它在介绍新技术的过程中，会设计与之配套的大量练习。参加工作坊的人，都需要参与到这些练习中，以掌握相关的技能。由于这些练习是现场进行，它能让参与者相互进行沟通和交流。在练习过程中，若是遇到一些问题，也能互相帮助，快速解决遇到的问题。正因为如此，与私下的个人练习相比，它能更快速地掌握相关的技能。

与工作坊相似的，还有一个名为 Bootcamp 的练习事件。它适用于复杂的、抽象的技

术，如面向对象和设计模式。工作坊着重于让参与的人掌握一门技术；而 Bootcamp 则只能帮我们技术入门。

3. 面向新人的结对编程

结对编程，是现代软件工程一直在寻找的一种有效的知识传递方式。结对编程存在多种模式：

（1）Navigator-Driver（领航员-驾驶员式）。Navigator 关注如何实现功能，Driver 则负责实现。并且由 Navigator 告诉 Driver 如何实现相关的代码。

（2）Ping-Pong 模式。常见于 TDD 开发模式，由 A 编写某个功能，B 实现测试，随后调反过来，由 B 编写功能，由 A 实现测试。

（3）键鼠模式。即编程时，由一方掌握鼠标，一方掌握键盘。这种模式的主要目的是，帮助新人快速熟悉使用编辑器的快捷键。

对于新人而言，我们往往会采用 Navigator-Driver 的模式来进行结对编程。过程中，只提供相应的思路，由项目中的新人来实现相应的业务。而作为一个经验丰富的程序员，我们除了指导他/她更好地使用工具，诸如如何更好地使用 IDE 和 Git 命令等。我们还需要观察新人在过程中犯的错误，有些错误，我们可以直接指出来，并帮助其改正；有些错误，则是等他犯了之后，帮助其解决问题，以积累经验——有些错误若是不犯，可能并不会意识到有错误。

4. 对内输出和对外输出

输出是最好的输入方式。在输出的过程中，需要重新梳理知识体系，也因此输出变成了一个输入的过程。这种重新输入的输出，可以分为对内输出和对外输出。

对内输出。我们前面提到的技术分享、工作坊、结对编程等，都是一些对内输出的形式，它可以加快新技术在团队和组织内部的实施。

对外输出。这部分内容便是我们在日常的技术社区里经常看到的各式各样的技术团队的输出方式。在国内，越来越多的公司和企业，都在进行对外输出和分享，比如撰写某项技术在公司的实践、某些新框架的试用，等等。对外输出的目的有两个：

（1）练习技术上的技能，提升相应的软技能。

（2）扩大团队的影响力，以便未来招聘到更多人才。

输出方式多种多样，有翻译、写作、开源项目、技术分享等。我们既可以翻译技术文章，也可以翻译技术书籍，还可以撰写团队相关的技术文章、技术书籍。诸如 360 奇舞团的相关开发框架和技术周刊，饿了么团队开源的前端组件库，广发证券团队翻译的 Angular 相关书籍等，这些都是很好的对外输出示例。

5. 其他

此外，一旦内部没有资深的人可以对我们进行培训的时候，我们便需要考虑使用外部力量来提升我们，即外部培训。由于笔者并没有遇到内部无法消化的问题，对于外部培训的了解并没有那么深刻。从笔者的角度来看，如果只是为了学习新技术，那么可以尝试在团队中一起学习和练习。在学习和练习的过程中寻找专业的人士、搜索相关的资料，以更细致地掌握相关的内容。

不得不提及的一点是，以上几种方式，都是笔者在 ThoughtWorks 经历的一些提升方式。在不同的公司里，也会有各自不同的提升方式，总体上存在一定的差异。但是，大体上并不会有太大的偏差，只是有的公司不一定有这么全面的培训体系。

2.4　成长优化期：技术债务与演进

经历一两次上线后，项目进入一个稳定上线、交付的阶段。笔者将其称为成长优化期，这是一个技术提升开发体验，技术带来更多业务价值的阶段。不过，实际上这已经是一个稳定的时期——我们可以抽时间来解决各种各样的问题。

在设计架构和完善业务的过程中，会暴露出团队的一系列问题：架构不完善、开发流程不便利等。短期内，这些问题并不会影响我们的开发。然而就长期而言，这些问题还是有可能会影响开发进度。先前我们在追赶业务时也遗留了一些技术问题，尤其是代码的质量问题，这些问题会随着时间的推移和业务代码的堆砌变得越来越严峻。当然，如果一个项目的时间短，那么它就不会遇到这些挑战。不论怎样，程序员作为一个“匠人”，总得有点“追求”，要不断地提高自己的水平。

2.4.1 偿还技术债务

在技术准备期，我们在构建技术基础方面花费了大量时间；在业务回补期，我们在支持业务的开发方面花费了大量时间。在这两个时期，我们都或多或少地采取了一些妥协方案，为的是能加快速度开发流程。这些问题都将在未来成为我们的开发负担。这种方式就和债务一样，可以在短期内得到好处，但是在未来必须偿还它们。这部分开发负担，我们称其为技术债。

技术债包含的内容有如下几个方面：

（1）代码质量。常见的问题有接口、函数的重复实现，即 a 成员在自己的功能内，实现过这个功能，但是 b 成员又实现一份，没有提取到公共的方法中。实现方式或模式不统一，比如我们采用某个框架来解决问题，但是在真实场景下，可能又会采取过去的实践方式，诸如使用 Lambda、RxJava、Rx.js、Ramda 等框架来进行函数式编程。少部分代码未按规范进行实践，这个问题更加常见，不仅存在于没有代码检视（Code Review）的项目，还存在于拥有代码检视的项目。未检视过的代码，往往容易被遗漏。

（2）测试覆盖率。在面对技术不熟悉、业务又过急的情况时，UI 自动化测试、单位测试往往是最先被抛弃的一环。一方面，自动化测试的目的在于，保持功能不被破坏；另一方面，大部分的项目都拥有专业的测试人员进行测试。值得商榷的是，国内的互联网公司都不会有测试这种东西，所以它们就存在这种长期的债务了。

（3）依赖问题。依赖对于短期项目来说不是问题，对于长期项目来说，依赖没有及时更新是一个很严重的问题。例如，我们使用 Redux 3.0 的版本，当 4.0 版本发布的时候，按照语义化版本的规则，它可能修改了大量的代码，而我们不得不追随这个变化——除非，我们决定在未来重写该应用，否则大量依赖过旧的问题，会导致我们难以对代码进行重构，因而不得不重写应用。

有些问题不是一天两天造成的，比如测试覆盖率低的问题，要解决这些问题也不是两三天就能完成的。这往往需要制定一个长期的计划，才能将它们一个一个地修正过来。多数情况下，我们并没有足够的时间在短期内修复，相关的技术债都是项目在不断演进的过程中，一步步提升的。比如测试覆盖率，它依赖所有的人编写测试，并不断限定测试覆盖率的下限。这样一来，在经历了几个迭代之后，我们便会拥有不错的测试覆盖率。

想要改善代码质量也不是一件容易的事，改进代码质量要依赖开发人员的水平，以及

团队的能力。如果团队中的代码质量不忍直视，充满了各种 Code Smell（代码的坏味道），那么可以通过代码检视的方式来不断提高团队成员的水平。然后，有针对性地进行相应的培训。

值得注意的是，与日常的业务代码编写相比，改进过去的代码会带来更多的成长和技术挑战——我们更容易从错误的代码中学习，而不是从成功的经验中学习。举个例子，我们直接看别人写的与设计模式相关的代码，并不会直接学到相关的内容，如果从过去写的代码重构看到设计模式，就能更深刻地理解相应的技术实践。

2.4.2　优化开发体验

提升开发体验，也是在稳定时期值得考虑的另外一个因素。在我们的日常工作中，有很多是手动完成的，我们可以通过自动化来减少重复性工作。

有这样一些例子：我们想创建一些测试数据，需要在数据库中手动创建，但是缺少对应的批量创建脚本或命令；在调试时经常需要手动输入相关的账号，这也可以通过插件来自动化登录；在开发移动应用时，一旦提交了代码，就应该有相应的工具来自动化构建应用，并在构建成功后上传到某个包管理中心，同时安装到对应的测试机。相似的，还有其他各种方式的自动化流程，它们所做的便是减少重复的工作，以不断提升开发体验。

在这个过程中，我们可能写了大量的接近重复的代码。而这些代码表面看上去并不是重复的，但是从抽象层来看是重复的。为了应对这种问题，我们可以进行代码重构，也可以通过诸如创建领域特定语言的方式来进一步抽象出内部 DSL 的代码。这样，我们就可以减少花费在业务代码上的时间。

此外，还可以思考怎样将一些代码的编写实现自动化，例如在 UI 层通过采用 Sketch2Code 来生成模板页面，或者编写相应的拖曳生成 UI 界面的工具。一旦我们优化了这些开发流程——尤其在相应的自动化功能完成之后，开发人员就开始面对一些新的挑战。

2.4.3　带来技术挑战

堆砌业务代码对大部分技术人员来说是一件难熬的事情——每天都在重复工作，总想寻找一些挑战。而对于周期长的应用来说，这种事情更为可怕。不论怎样的项目，技术人员都需要获得一定的能力增长。如果不能满足这种诉求，那么就不能进步，也就相当于是一种能力方面的退步。因为技术在不断地进步，新的技术总会很快地淘汰旧的技术。一个

长期项目一旦结束，新的技术体系就可能与旧的技术体系完全不一样了。

面对这种情况，一种常见的做法是引入新的技术栈。它既可以是一个框架，又可以是一门语言。我们可以引入 Rxjs 进行响应式（Reactive）开发，或者引入 Rambda.js 进行函数式编程。也可以采用新的语言，如在开发 Android 应用时，除了使用 C++，还可以使用 Kotlin 语言来完成部分功能；在开发 iOS 应用时，可以结合 Objective-C 来使用 Swift 语言进行开发。

对于前端应用而言，我们还可以尝试使用新的前端框架。目前维护大型的前端应用用的是 Angula 框架，如果要开发一些新的简易的应用，那么可以尝试使用 React 或者 Vue 框架来实现。

此外，对于稍有余力的项目团队来说，可以尝试进行一些小型的模拟项目。在这些小型的模拟项目上，我们关注于使用新的技术来开发现有的应用。一方面，可以为以后的架构演进做准备；另一方面，让我们的技术与主流接轨。在笔者经历过的项目中，曾经有一个项目，每隔几个月开展一次 Hackday 活动，活动内容是使用新技术来重写旧的应用。此外，我们还会做一些 Workshop 来练习使用项目潜在的新技术栈。

当然，对于项目管理者来说，堆砌业务代码会轻松很多，毕竟出现新技术风险的可能性较低。但是在保证进度的情况下，也需要适当地带来一些成长的机会。

2.4.4 架构完善及演进

经过大量的业务沉淀，我们也会发现架构中存在的一些问题。这些问题大都是一些架构设计上的偏差，经过调整，有的纠正到原来的设计上，有的使用新的架构设计。如果使用新的设计，我们还需要将那些使用过去设计的部分进行相应的修改，这意味着会有一定的返工工作量。

我们还可能遇到由于业务变更导致架构需要修改的情况，这时不仅需要调整，还需要进行全新的设计，才能适应新的业务需求。比如，如果在前端应用中包含大量的第三方应用，就要考虑插件化的方案，设计一个新的插件化架构。再举一个例子，如果我们最开始开发应用的时候使用 Cordova 的 WebView 容器，那么出于安全考虑，会迁移到自己开发的混合应用框架上。这两个例子都只是在架构设计上发生一些改变，但是从代码的角度来看，所需要的修改并不会太多。

当然，这也并不是说我们预先设计的架构有问题，因为最初的架构满足的是创建架构

时设计的业务场景。当业务发生一些变化时，架构也需要做相应的调整。我们不可能让过去的架构一直适应新的业务变化，在变化发生的时候演进系统的架构才是一种正常的形态。

此外，在更低级别的代码层里，我们会发现代码中存在一些复杂、混乱的相互调用，如果不加以规范并重构代码，那么会使得新增的代码加剧这个问题的产生。比如在项目中后期参与开发的人员编写的代码，可能与我们最初的架构风格不一致。如果原先的架构风格更符合需求，那么还需要帮助这些后来的开发人员解决相关的问题。

2.5　小结

我们在进行设计架构时需要关注架构的设计和实施。在第 1 章中我们介绍了如何设计项目的架构，在本章中则关注于架构的实施。我们通过深入项目的三个时期——技术准备期、业务回补期和成长优化期，详细了解了在整个项目的生命周期里，如何一步一步地实施和改进项目的架构。

想要高质量地实施一个架构，需要有相应的实施者。作为一个技术负责人，需要关注对应的事项。因此，在本章的最后，我们介绍了如何通过多种方式来提升团队的能力。

3

第 3 章

架构基础：工作流设计

前端基础架构是一系列工具与流程的集合，它是我们在启动一个项目时所需要制定的一系列规范和规划。每当我们启动一个新的项目，所要做的事情会很多。

在 ThoughtWorks，我们称这个时间段为 Iteration 0，即 I0 或者迭代 0。不同的时期，我们对于项目启动会有不同的看法。以笔者为例：

1.0 时期，迭代 0 就是选定一个前端框架。刚工作时，这段时间所要做的事情，就是从众多前端框架中选一个。选定框架本身就是一件麻烦的事，要进行一堆细致的分析才能决定使用哪一个。如果组织内没有规定好框架，那么选择框架便相当复杂。当笔者真正去启动一个前端项目的时候，发现迭代 0 并不是选一个框架、写一个“Hello,world”那么简单，还需要搭建一个持续集成环境。在搭建持续集成环境的过程中，仍需要编写前端应用所需要的构建脚本。于是，迭代 0 所做的事情就变得更复杂。

2.0 时期，迭代 0 应该是选定前端框架 + 完整的构建脚本和构建系统。在笔者随后经

历的项目中，这个模式很好。原因是在这些项目里，笔者只是负责一个小团队的架构。当需要负责一个更大的项目、更大的团队时，需要从组织的层面去考虑这个问题。对于一个大的前端项目，其前端架构要让所有的团队都可以一起构建这个应用，但相应的代码不能在同一个代码库里。此外，还有一个麻烦的问题就是规范化。在小的团队里，规范可以口头约定，而在大的团队里，规范一定是可以自动化的。

3.0 时期，迭代 0 应该是选定前端框架+完整的构建脚本和构建系统+流程规范化。在笔者接触一些更大规模的团队时，发现在团队里要做的事情就更多了。从组织本身的安全和能力提升出发，我们应该开发自己的前端开发框架。从用户体验和开发人员的便利性上看，我们应该做出自己的前端 Design System。为了降低开发成本，我们还需要拥有一些前端应用脚手架、自己的开发工具或者协作工具。于是，工作的重心便从业务转向了构建工具。

这样仔细一看发现：哦，这是和团队规模密切相关的啊。为了保障团队能顺利合作开发，我们需要制定一系列的开发流程，并创建相应的开发规范。

技术在不断演进，帮助我们不断地提高生产力、使流程规范化。过去我们手动下载依赖，现在可以一键安装依赖；过去需要单独进行测试，现在可以在提交代码后自动测试。随着我们不断地优化开发流程，这些工作越来越自动化。

原先，我们只是出台一系列规范，期盼团队中的成员能遵循，然而规范在这个时候只是规范而已。现在规范被硬编码到流程中，一系列的操作都由程序来进行检测——使用 Lint 工具对代码风格进行检验，使用测试覆盖率检测测试是否测试过等。这些流程旨在通过规范代码来提升代码的可阅读性，减少代码中可能出现的 bug。

当然，硬规范有好的方面也有不好的方面。好的方面是，它可以产出更规范的代码；不好的方面是，它让我们千篇一律。至于使用哪种方式，具体还要看情况——从两个方案中选择时，我们往往是从两个不好的中选出一个较好的。

这些流程与规范，就是我们要在本章中讨论的内容。

3.1　代码之旅：基础规范

在设计架构的时候，要考虑由下而上的模式，底层的实践最终会影响整个系统的架构。

再好的架构，如果没有辅以有效的工程实践，那么最终我们得到的只是一个空有其表的架构方案。能自下而上影响软件架构的，就只有代码了。

代码本身是一种难以衡量的实践，同一个业务功能有不同的代码实现。想象一个场景，我们对外提供了一个 RESTful API，是不是只要我们能以规范的方式提供这个 RESTful API 接口，代码的实现方式和质量就变得不重要了？

从短期来看，如果一个 API 能快速地提供功能以驱动业务增长，那么它就是一个成功的 API。不论其设计多么丑陋，代码质量多差，只要不影响性能，未来就有改进的空间。可是从长期来看，API 是要能够面向变化而快速扩展的，如果我们不能方便地在 API 中扩展功能，那么它就真的会影响业务了。尽管重构旧的代码可以帮助我们走向更好的架构，但是在业务进度不合理的情况下，我们只能在旧的、丑陋的代码上不断堆砌功能。直至有一天，我们愉快地选择重写系统。

在本节里，我们将讨论代码中的一些基础规范，它们更多地关注代码的可读性，而不是代码的质量，我们会在后面的章节里关注代码质量。为了提升代码的可读性，我们需要做到以下几方面：

- 规范代码组织结构。
- 统一代码风格，即源代码的书写风格。
- 组件、函数等命名规范。
- 开发工具规范。

光看这几点要求，总觉得似乎多了很多条条框框。尽管这种统一性会扼杀团队的多样性，但是对于代码层次的风格统一是相当有必要的。

在这些实践中，有些并不仅仅是实践，它还反映了架构的模式，如代码组织结构——从代码的组织构建上，我们可以真真切切地感受到它与系统架构的相似之处。由于应用内的代码复用采用组件化的架构，所以我们应该隔离不同的组件。比如，在 Angular 生成的组件（component）中，我们就可以看到一种组件完全独立的存在形式。

3.2　代码组织决定应用架构

开发人员在构建架构之后面临的第一个挑战是，如何读懂代码结构。在接触代码之前，我们所要面对的就是代码的组织方式。我们可能会做如下一些事情：

- 打开 README 了解应该阅读哪些相关的资料。
- 阅读 package.json 了解系统的基础设施、使用了哪些组件库，以及配置了哪些构建脚本。
- 浏览主目录下的一个个文件，了解系统的一些插件的配置。
- 进入项目代码中阅读和了解。

不同框架有不同的文件组织方式，不同团队也有不同的规范。比如使用 Java 或 Spring 框架，默认会遵循这样的一些文件夹规范：model 文件夹用于存放与模型相关的代码，repository 文件夹用于存放与数据源相关的代码，service 文件夹用于存放与服务相关的代码等。这些代码的组织形式实际上反映了应用的架构。

Python 语言的 Django 框架使用的方式是基于业务的组织形式，即每个“应用”（即某一业务领域，如用户管理）是一个单独的文件夹，在它的目录中拥有自己完整的相关代码。下面是一个 Django 应用的代码结构：

```
.
├── __init__.py
├── admin.py
├── apps.py
├── migrations
│   └── __init__.py
├── models.py
├── tests.py
└── views.py
```

对于前端来说也是相似的。下面是 Angular 生成的应用（通过 ng new xxx 可生成）的主目录下的结构：

```
e2e                                    //E2E 测试文件夹
src                                    //源码文件夹
.editorconfig                          //编辑器统一格式配置文件
.gitignore                             //Git 忽略文件的配置文件
angular.json                           //Angular 配置文件
package.json                           //软件包相关信息的配置文件
README.md                              //文档
tsconfig.json                          //TypeScript 编译配置文件
tslint.json                            //TypeScript 代码风格配置文件
```

通过上面的文件名信息，我们可以快速地了解系统的一些基础组成。这对于其他前端框架也是相似的，这些配置文件实则是定义了一系列的规范。对于不同的开发者，它们可以起到规范的作用。

在 src 目录的 app 文件夹下，有如下一些文件：

```
├── app.component.css
├── app.component.html
├── app.component.spec.ts
├── app.component.ts
├── app.module.ts
└── header
    ├── header.component.css           //CSS 样式文件
    ├── header.component.html          //HTML 模板文件
    ├── header.component.spec.ts       //测试文件
    └── header.component.ts            //TypeScript 代码逻辑
```

面代码中的 header 是通过 Angular 的 CLI 生成的。举 Angular 的例子是为了说明代码组织结构的重要性。此外，还需要注意文件名。如果这里的 header 组件实际上是一个用于页面底部的组件，那么就会很尴尬。

从这里的代码组织形式来看，它更像是基于组件的架构，而不再是类似于 MVC 形式的架构。对于其他前端框架比如 React，如果我们通过状态来管理应用，那么从组织构建上看，它像 MVC 架构。

从文件名上就可以了解这是一个组件（Component），而由于 CSS、HTML、TypeScript 的分离，我们又可以快速地在修改代码时找到对应的文件。这种特性对于庞大工程的项目来说相当有帮助，但是对于简单的业务，比如加载（Loading）组件时，使用与 React、Vue 相似的结构会更简洁，即一个文件包含 CSS、HTML、JavaScript 代码。

注意：这里主要针对框架默认生成的代码组织架构，一个 Vue 组件也可以拆分成多个文件，一个 Angular 组件也可以写成一个文件。而默认生成的，意味着这个框架是约定俗成的，在生成的时候需要加以注意。

因此，当我们定下了前端应用的架构时，我们应该按照与架构相似的方式来编写，以免随着代码架构的变化而腐烂。当架构在未来发生变化的时候，我们就需要相应地更改代码的组织方式。

3.3　统一代码风格，避免架构腐烂

代码是写给人看的，给计算机执行的是二进制码。

现在，我们已经准备好了代码，也大概地了解了项目的架构。接下来，便是真正了解一个项目代码的工作——阅读代码。迎面而来的是代码的风格、写法，然后才会注意到函数的实现。如果使用比较智能的 IDE，那么可能得到如图 3-1 所示的一堆错误。

```
if(direction !== undefined) {
  if(currHeadingLevel <= 0) {
    if(direction == "bigger") {
      text = "###### " + text;
    } else {
      text = "# " + text;
    }
  } else if(currHeadingLevel == 6 && direction == "smaller") {
    text = text.substr(7);
  } else if(currHeadingLevel == 1 && direction == "bigger") {
    text = text.substr(2);
  } else {
    if(direction == "bigger") {
      text = text.substr(1);
    } else {
      text = "#" + text;
    }
  }
} else {
```

图 3-1

作为一个工程师，每天不得不面对其他来源的代码，或从 StackOverflow/Google 上搜索到的代码，或从 GitHub 上复制的代码，又或者阅读别人的面试作业。看到代码的第一眼，可能就觉得这份代码不适合自己的口味。原因有很多，比如，代码里使用四个空格，而不是两个。在日常的工作中，如果你都不想多看几眼项目中别人的代码，那可就糟糕了。

3.4 使用 Lint 规范代码

为了规范代码，我们需要诸如 TSLint 及 CSSLint 类型的代码扫描工具。我们可以利用这些 Lint 工具，通过配置语法检测规则来对代码风格进行检测。也可以关闭所有的规则，只运行基本语法验证，这一切都取决于我们的意图。

对于代码风格来说，有太多需要规范的地方，比如下面这些事项：

◎ 是否以分号（;）结束语句。

◎ 缩进四个空格，还是两个空格。

◎ 判断是否相等的时候，使用===，以避免类型转换。

◎ 函数大括号是否换行。

通常，我们可以生成自己的配置选项，也可以使用其他团队、组织创建好的风格配置。比如，Airbnb 创建的 eslint-config-airbnb 规范，对于编写 JavaScript 的人来说就相当不错。我们可以在它的基础上修改出符合要求的规范，从而节省大量的时间。

这些风格问题很难讨论出一个好坏，但是我们需要统一成一种风格——如果在一个文件里，有的以两个空格做缩进，有的以四个空格做缩进，那么它的阅读体验就会相当差。

幸运的是，在现有的前端应用模板里，都会生成相应的 Lint 配置。我们只需要在构建时、提交时、测试时运行相应的脚本，即可对代码进行 Lint 扫描。不同组织有不同的使用倾向，特别是在那些有一定年代的系统、公司里，要改变现有的命名方式很难。因为使用哪种方式并不重要，重要的是保持一致。

事实上，这些 Lint 工具不仅能帮助我们分析代码风格，还对代码质量起着一定的监督作用。如在 TSLint 里，当我们继承了一个类，但是没有实现其方法时，它就会提醒我们这里有问题（笔者在这里使用的工具是 WebStorm），代码如图 3-2 所示。

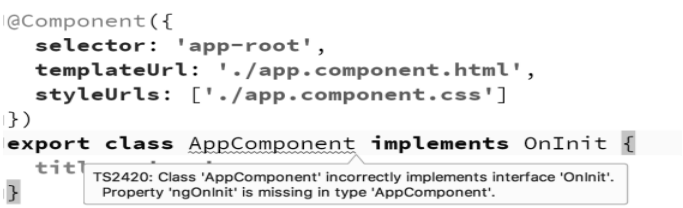

图 3-2

AppComponent 类想实现 OnInit 接口，但是却缺少 ngOnInit 属性。另外，在诸如 TypeScript 这样的静态语言里，它能进行更细致的语法检测。

当然，这些工具在提升质量的同时，也在一定程度上限制了代码的灵活性，毕竟，灵活性和规范是难以兼得的。

3.5　规范化命名，提升可读性

我们开始进一步了解业务逻辑、阅读项目的代码和函数。于是，我们又进一步将注意力放到命名上。

命名，对于程序员来说是一件相当头疼的事，对于英语不是母语的开发人员来说更是如此。为了取一个变量名，我们需要打开翻译软件、网站，输入对应的中文名，以获取可能的英语单词。再按自己的经验来决定使用哪个单词。即便如此，一个英文单词又可能拥有多个中文含义，我们又需要进一步翻译成中文。即便最后我们决定使用这个单词，在和其他程序员讨论代码的时候，可能还会对这些代码有一定的争议。

这个头疼的问题不是我们使用规范能解决的，但是我们可以通过制定命名规则来统一命名方法。

3.5.1　命名法

对于阅读代码来说，一个简单有效的函数名、变量名，远远比冗长的注释来得更加有用。而这些函数名、变量名本身应该也是容易阅读的，比如 gettypebyid 这种全小写的方式，对于开发人员来说相当不友好。如果读者也接触过不同的语言就会发现，语言间的差距不仅

体现在使用上，还会体现在函数、变量的命名上。下面是几种常用的命名规则：

- 驼峰命名法，译自 CamelCase，类似于骆驼的后背形状一样高低起伏。这个命名法来源于 Perl 语言，其展现形式是在单字之间通过首字母大写来展现，如“getTypeById”。在前端开发中，这种命名规则较为常见。
- 下画线命名法，即通过下画线来分割单词，如“get_type_by_id”就是采用这种命名方法。下画线（_）的分割方式非常显眼，它也更容易让开发人员区别单词。在 Python 语言中，这种命名方式特别流行。
- 匈牙利命名法，最初是由一个匈牙利程序员 Charles Simonyi 发明的，其命名规则是：属性 + 类型 + 对象描述。如“strFirstName”便是这样的形式，变量中的“str”表示这个变量的类型是 string，再比如“iAge”中的“i”表示这个变量的类型是 int。

上述几种方式都是为了提升代码的可读性而出现的，并不存在哪种命名方式更好，都是随习惯而来的。

不同语言之间存在一些不同的文化，在使用其他语言的时候，这些文化会体现出来。比如在 JavaScript 里，通常会使用 that = this 来解决 this 作用域的问题，而对于一些程序员（如 Ruby 语言的使用者）来说，他们会将它写成 self = this——这种方式对他们来说更好理解。

对于前端团队来说，我们需要统一项目的命名规则，以降低项目的成本。

3.5.2 CSS 及其预处理器命名规则

除了代码，CSS 的命名规则也是值得进行规范的。它存在于 HTML 文件和 JavaScript 代码中，以及自身的 CSS 文件中。

在前端技术快速发展的今天，我们已经不再直接编写 CSS 文件，而是通过 CSS 预处理器将 LESS、SCSS 等新增了很多编程特性的语言转换为 CSS。对 CSS 进行命名规范的主要原因是，如果不同页面、组件中定义的样式发生冲突，就会导致页面 UI 受到影响。而在诸如 Angular 这样的 Web Components 框架里，它会将一个组件内的 CSS 代码进行编译，使它只在自己的组件里生效。

同样，在规模比较大、技术栈统一的前端团队里，会开发一套统一的编码规范。比如

“Airbnb CSS/Sass 指南”[1]，它里面会定义一些常见风格的写法，并且会编写 CSS Lint 工具。

下面是在上述的指南中提到的 CSS 命名示例：

```
.ListingCard { }
.ListingCard--featured { }
.ListingCard__title { }
.ListingCard__content { }
```

下面是一些对应的解释：

◎ .ListingCard 是一个块（Block），表示高层次的组件。

◎ .ListingCard__title 是一个元素（Element），它属于.ListingCard 的一部分，因此块是由元素组成的。

◎ .ListingCard--featured 是一个修饰符（Modifier），表示这个块与.ListingCard 有着不同的状态。

需要注意的是，原版的 Airbnb 写法中，“ListingCard”采用全小写的形式，“listing-card”在使用 React 的 JSX 语法的时候，变成了驼峰式写法。

3.5.3　组件命名规则

同样，对于团队而言，组件的命名规则也需要规范，特别是对于使用组件化框架的项目而言，有如下几种不同的命名方式：

◎ 按照功能来命名，比如 SideBar 就是一个侧边栏功能的组件。

◎ 按页面来切分组件，比如 NewsItem 就是用于展示新闻的组件，它既用于列表页，又用于相关新闻页。

◎ 按上下文来命名组件，如 NewsChildItem 就是按需要将上一个组件的某个共用元素拆分出来。

不过这些命名方式并不是对立的，它们可能同时存在于一个项目中。

[1] https://github.com/airbnb/css

3.6 规范开发工具，提升开发效率

笔者曾与多个团队的开发人员用同一个代码库工作，经常发现他们提交的代码有问题。原因是他们使用的编辑器不够智能——缺少插件。有些问题是新手程序员注意不到的，有些则可以显式地通过工具来提醒开发者：这里有问题。

因此，统一开发工具是必要的。在前端开发的时候，可以选择编辑器如 Visual Studio Code，或者专业的 IDE 如 WebStorm。建议初学者从 IDE 上手，因为 IDE 更关注编码，而非在学习编辑器。

然而，对于编辑器的统一，同样会扼杀团队的多样性。因此退而求其次，我们可以追求使用相同的插件。下面是一些适合在项目中使用的插件：

- EditorConfig，它可以让我们读取项目中的.editorconfig 配置，以遵循统一的编辑器规范，诸如两个空格的缩进。
- Lint 插件，如 ESLint、HTMLHint，可以帮助我们在 IDE 及编辑器上显示 Lint 问题。
- 单词拼写检测，直接在代码中显示拼错的单词，可以帮助其他人阅读代码。
- 路径补全（Path Intellisense），能自动提醒我们可以引用的资源、库路径。
- 代码自动补全，包含不同语言的代码补全。
- Emmet 插件，可以帮助我们快速编写 HTML、CSS 等。
- 代码格式化，可以帮助我们格式化代码。

同样，编辑器在帮助我们的同时，也在一定程度上限制了我们。但是当它的收益远大于损失的时候，就是值得的。

3.7　项目的文档化：README 搭建指南

现在，我们已经大致了解了前端项目的一些基础规范，但是仍然可能错过一些关键的信息。这时，我们就需要通过文档来深入了解细节。在讨论文档时，有可能说的是需求文档、设计文档、测试文档、用户手册等，而这里特指的是与项目相关的技术文档。

文档对于一个项目来说，是一个重中之重的存在。当团队进来一个新人时，需要提供一份快速上手指南；当我们想要了解当前的系统架构时，需要一份能方便查阅的文档；当我们对过去的技术决策表示怀疑时，需要查看文档了解原因，并查看架构演进设计。

搭建指南，几乎是每个代码库的必做工作。它最好能以某种形式存在于项目的代码中，如 README，开发人员在复制代码到本地时，就可以直接查看。并且在项目演进的过程中，持续不断地对它进行更新，以便其他人能顺利地开展活动。

一份好的搭建指南，应该和我们在 GitHub 上看到的开源项目是相似的，具有如下特点：

◎　支持运行的环境。

◎　必要的依赖准备，以及如何搭建。

◎　项目的安装指南。

◎　线上的示例或最后的运行环境。

◎　相关的文档链接。

◎　相关人员的联系方式，讨论群。

在大部分项目里，这些必要的资源放在 README 中，能大大地提高开发人员的效率。

3.8 绘制架构图：减少沟通成本

对于架构复杂的项目而言，架构图是必不可少的。架构图能让一个团队中的新成员快速地了解现有系统的各个组成部分。对于复杂的系统，架构图一般展示的是各个子系统之间如何通信；对于简单的系统，架构图则可以是由项目的技术栈组成的。

在简单的前端项目里，架构图可能只表现不同框架之间的关系，以及不同层级的组件库之间的使用关系。即使不看架构图，我们也可以清晰地了解项目的架构。

对于复杂的前端项目来说，我们可能采用微前端的方式来设计应用架构。各个应用之间可能存在一定的关联性，以及底层的一些共用依赖等。在微应用化的前端方案里，架构图的作用更多是描述项目的构建过程。

绘制项目架构图有多种方式，既可以使用代码生成，也可以使用专业工具绘制。当然，也可以是某次会议上的讨论结果，然后将拍照记录到文档中。总之，我们需要为后来者建立一个有效的文档机制，以方便未来的开发者能够在需要的时候，找到原有的系统设计和现在架构之间的一些差异。

3.8.1 代码生成

对于可能不断变化的事物而言，能使用版本工具记录是最好的选择。对于架构图的记录也是如此，因此我们要考虑的第一种方式就是：代码生成架构图。

在笔者的工具箱里，Graphviz 就是一个不错的工具。它是一个由 AT&T 实验室启动的开源工具包，用于绘制 DOT 语言脚本描述的图形。它也提供了可供其他软件使用的库，比如 Darge.js 是一个可以直接在浏览器渲染 Graphviz 的库。

Graphviz 之所以方便是因为语法特别简单——和我们平时表达的方式是一样的，即：

```
"包管理" -> "包发布" -> "自动部署"
"CLI" -> "部署"
"脚本语言(Bash,Perl,Ruby,Python etc)" -> "部署"
"脚本语言(Bash,Perl,Ruby,Python etc)" -> "构建"
"*nix" -> "软件编译" -> "部署"
"构建" -> "软件编译"
```

我们只需要关注如何编写它们之间的关系，就能快速地绘制出流程图。

Graphviz 也有一些明显的缺点，比如它的自动化生成连线，容易导致线条间重叠；它生成的 UI 图，看上去不是那么美观。因此，它更适合在开发者之间使用，当我们有一些更高级的需求时，则可以考虑使用其他工具。

3.8.2　专业工具

与代码生成相比，使用专业的工具来绘制架构图是一种更方便的选择。但是因为它们专业，所以在价格上也体现了一定的专业性。比如 Windows 系统的 Visio、macOS 上的 OmniGraffle 都是一些非常不错的软件，只是它们价格昂贵。当然，除了架构图，这些专业的软件还能做出更丰富的与项目管理相关的软件图。

除了这些专有的闭源软件，还可以选择使用开源软件，比如 Dia。它是开源的流程图软件，是 GNU 开源计划中的一部分，作者是 Alexander Larsson。它可以将多种需求以模块化的方式进行设计，如流程图、网络图、电路图等。

3.8.3　软件附带工具

专业的绘图软件都是相当昂贵的，并且不是很好用。笔者有时会使用 macOS 上自带的 Keynote 来绘制层级架构图。对于不复杂的架构来说，它相当方便，并且快速有效。macOSKeynote 的示例如图 3-3 所示。

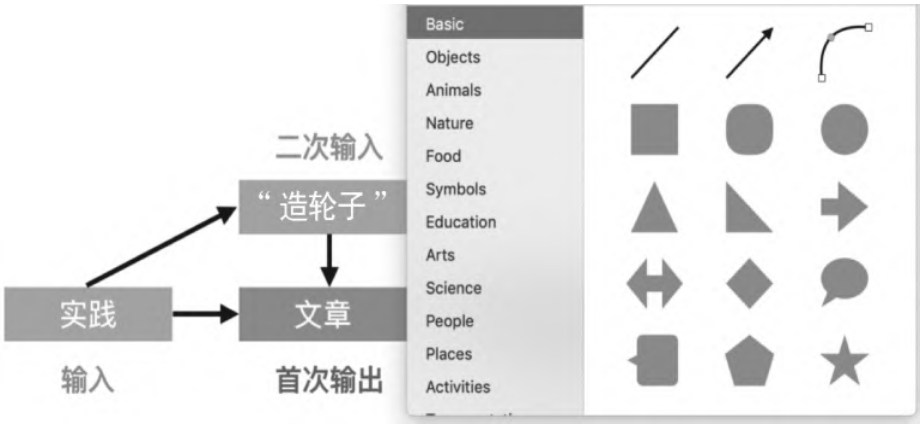

图 3-3

由于写作需要，笔者购买了 Office 365 套装。其中的 Word 和 Powerpoint 是自带的，SmartArt 也适用于架构图的绘制。SmartArt 自带一系列的模板，可以帮助我们快速地画出系统架构，如图 3-4 所示。

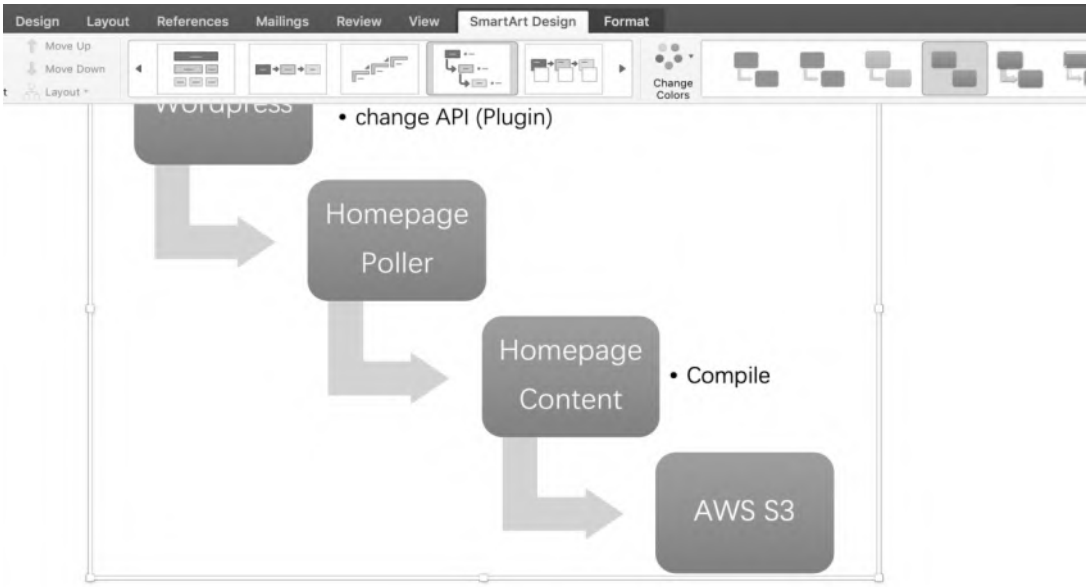

图 3-4

仔细探索，也许就会发现正在使用的编辑器可能也可以直接生成架构图，而不需要专门绘制。

3.8.4 在线工具

除了使用上述离线工具，我们还可以使用在线工具来绘制架构及流程图。由于篇幅限制，这里就不展开讨论了。

值得注意的是，如果不是会员，这些在线工具会在超出容量限制之后，将超出容量的图直接发布到互联网上。

3.9　可编辑文档库：提升协作性

绘制完项目架构图后，我们就需要找到一个可编辑的文档中心。同样的，我们优先关注于其是否可以版本化、图形可视化，以及追踪历史修改。它可以是：

◎　可以记录版本历史的维基（Wiki）软件。

◎　专业的协作工具，比如 Atlassian 的 Confluence。

◎　基于 Git + Markdown 的代码管理工具。

它应该还能支持以下功能：实时多人编辑、内容索引、图形可视化、实时搜索、打标签、检查清单、导入导出等。

我们之所以对于编辑的文档中心有这样的要求，原因在于遇到过太多的“坑”。过去接收的二进制文档，诸如 word 等，太容易过期了，它们不能进行版本管理，讨论问题的时候，需要进一步确认。如果中间文档有更新，我们需要重新获取一份文档。在反复沟通之后，我们会产生对文档价值的怀疑。

一个好的文档中心能够从某种程度上解决这个问题。但是，它并不能解决所有的问题。对于一个长期的项目而言，笔者更希望这个文档是放置在代码仓库中的。

3.10　记录架构决策：轻量级架构决策记录

架构是一种在持续变化，并且不断演进的事物。在一个长期的项目里，我们会不断地打开 Wiki、文件中心来更新文档。而遗憾的是，我们可能会忽略做出这个架构设计的原因。因此，我们需要在项目中实施轻量级架构决策记录。它同代码一起，记录软件架构在其生命周期里发生的变化、演进，等等。

轻量级架构决策记录是一种轻量级的架构记录方式，通常使用像 Markdown 这样的轻量级文本格式语言，它们既方便于开发人员编写，又适合于版本软件管理。同时，它还能结合到项目的代码中。架构决策记录将按顺序和数字编号，并且不会被重复使用。这些记录包含了影响架构、非功能需求、依赖关系、接口或构建系统技术等内容。

下面是一个由 adr 命令生成的 adr/0001-examples.md 架构决策记录的示例文件：

```
# 1. examples
日期：2018-07-30
## 状态
2018-07-30 提议
## 背景
在这里补充上下文...
## 决策
在这里补充上决策信息...
## 后果
```

在这个记录中，它由以下六部分组成：

◎ 标题。

◎ 日期。

◎ 描述决策相关的状态，包含提议、通过、完成、已弃用、已取代等。

◎ 价值中立的、用于描述事实上下文的背景。

◎ 应对这种场景的相应的决策。

◎ 记录应用决策后产生的结果。

通过它们，我们可以清楚地了解与技术决策相关的知识，给未来的开发人员做好知识铺垫，以便对系统进行进一步的演进。

3.11 可视化文档：注重代码的可读性

相比传统的文档方式，使用 Web 技术制作出来的文档更具表现力、更容易吸引用户。在现今的前端框架里，为了更好地吸引开发者，并方便开发者使用，项目的文档通常是一

些可交互的文档，如图 3-5 所示是混合应用框架 Ionic 的文档示例，左侧是一些相关的文档说明，右侧则是相应代码的示例。

图 3-5

我们不需要引入这个框架，就可以了解这个组件运行的效果。有时，按照官方的文档并不能达到预期的效果，我们可以查看右侧演示部分的源码。

在过去的某个项目里，因为项目的技术栈多而且复杂，加之项目组多了几个毕业生，急需一个框架来展示项目技术栈。笔者首先想到 ThoughtWorks 的技术雷达，然而技术雷达只会发现一些新出现的技术，以及其对应的一些趋势。对于现有的技术栈的一些趋势发现得不够全面，于是便着手构建一个新的技术趋势图。该图用于展示在项目架构演进过程中的技术栈变化，如图 3-6 所示。

虽然工具比较简单，但是能从一定程度上反映项目的技术趋势。而构建这样的工具，并不需要太多的时间，只需要几个小时就能搞定。与传统的文档相比，对于读者来说更容易使用和读懂。

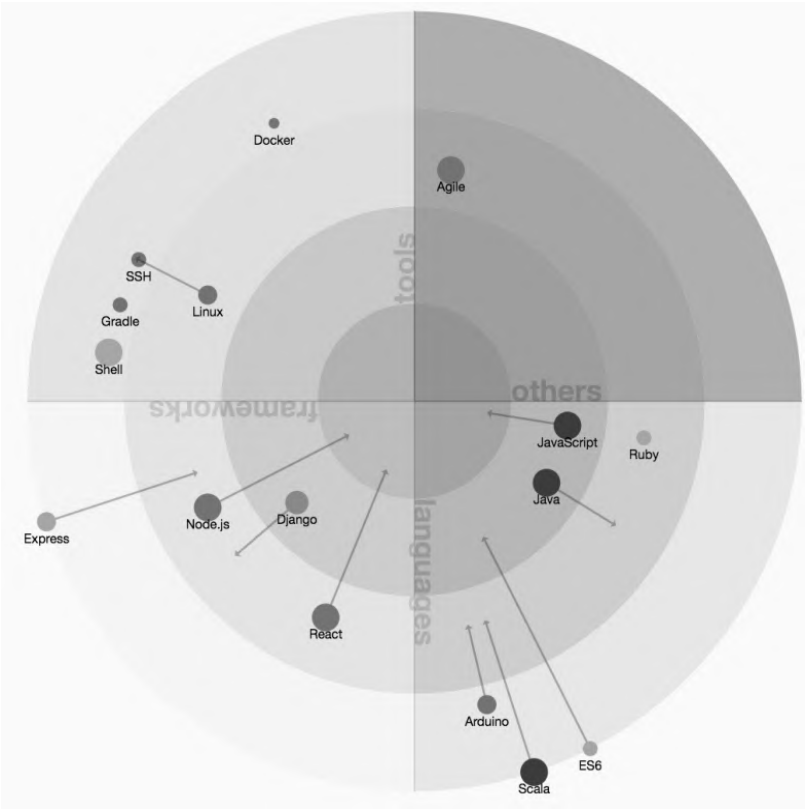

图 3-6

3.12 看板工具：统一管理业务知识

看板，对于敏捷项目来说是一个必不可少的工具。除了有了解项目进度、分配资源、发现问题的作用，它更重要的功能是记录用户故事，即我们拥有一个实时可翻阅相关业务内容的文档。在阅读代码的过程中，我们可以查找出代码修改的业务原因。

作为一个开发人员，过去笔者并不觉得这样的工具有什么帮助，直至遇到了一个有历史的遗留系统。由于系统差不多有 10 年的历史，在修改每一段代码前，都要了解项目的上下文。而由于人员的更换，写代码的开发人员、相应的业务分析师大都已经联系不上了。

即使能找到相应的开发人员，他也记不住相应的业务信息。每当这个时候，我们就只能寄希望于系统曾经记录过这些需求及其变更历史。

3.13　提交信息：每次代码提交文档化

在团队协作中，使用版本管理工具 Git、SVN 几乎都是这个行业的标准。当我们提交代码的时候，需要编写提交信息。

3.13.1　项目方式

提交信息的主要用途是，告诉这个项目的其他人，在这次代码提交中做了些什么。比如，在更新了 React Native Elements 的版本后，提交信息为：[T] upgrade react native elements。对应修改的是：package.json 和 yarn.lock 中的文件。一般来说，建议小步提交，即按自己的 Tasking 步骤来提交，每一小步都有对应的提交信息。这样做的目的是，防止在一次修改中修改过多的文件，导致后期修改、维护、撤销等困难。

在笔者经历的一个项目里，曾使用 Jira 作为看板工具，Bamboo 作为持续集成服务器，并采用结对编程的方式进行。在 Jira 里每一个功能卡都有对应的卡号，而 Bamboo 支持使用 Jira 的任务卡号关联的功能。这个卡号是指在持续构建服务器上与示例对应的任务卡号，即相应的提交人。因此，这个时候我们的规范稍微有一些特别：

```
[任务卡号] xx & xx: do something
```

比如：[PHODAL-0001] ladohp & phodal: update documents，解释如下：

- PHODAL-0001，业务的任务卡号，它可以帮我们找到某个业务修改的原因，即找出相应 bug 的来源。
- ladohp & phodal，结对编程的两个人的名字，后者（phodal）一般是写代码的人，出于礼貌就放在后面了。由于 Git 的提交人只显示一个，所以写上两个人的名字。当提交的人不在时，就可以向另外一个人询问修改的原因。
- update documents，在代码提交过程中做了什么事情。

只需要将这里的任务卡号与看板工具相对比，我们就更容易清楚地了解代码为什么这么写，原因是与其他方式相比，它的效率更高，更符合开发人员的习惯。

3.13.2 开源项目方式

与我们日常工作稍有不同的是：工作中的发布计划一般都是事先安排好的，不需要changelog。而开源应用、开源库需要有对应的 CHANELOG 文件，其包含或添加了什么功能、修改了什么，等等。这个文件如果由人力来维护则是一件麻烦的事——我们需要翻阅过去的提交历史，并整理出所有重要的修改。因此，在开源世界里，就采用标准化提交信息，来自动化生成 CHANGELOG 文件。

在这里，让我们以做得比较好的开源项目 Angular 为例来做展示。Angular 团队建议采用以下形式：

```
<type>(<scope>): <subject><BLANK LINE><body><BLANK LINE><footer>
```

下面举一个提交的例子：docs(changelog): update change log to beta.5，对应的解释如下：

- ◎ docs 对应于修改的类型，即文档更新。
- ◎ changelog 是影响的范围，即 changelog 文件。
- ◎ subject 是对应的事件，即相应的提交内容。

采用这样的模式，通过对提交类型的统一来生成 changelog 文件，相关参数如下。

- ◎ build: 影响构建系统或外部依赖关系的更改（示例范围：gulp、broccoli、NPM）。
- ◎ ci: 更改持续集成文件和脚本（示例范围：Travis、Circle、BrowserStack、SauceLabs）。
- ◎ docs: 只是更改文档。
- ◎ feat: 添加一个新功能。
- ◎ fix: 修复某个错误。
- ◎ perf: 改进性能的代码更改。
- ◎ refactor: 代码更改，既不修复错误也不添加功能。

◎　style：不影响代码含义的变化（空白、格式化、缺少分号等）。

◎　test：添加缺失测试或更正现有测试。

在 Angular 中同时还对应了 20+的 Scope，由于它是一种项目特定的 Scope，所以它并不需要适用于日常的项目开发，这里不再展开讨论。

这样做的优点是，它可以通过工具如 standard-version ，来生成一个 changelog 文件，如图 3-7 所示。

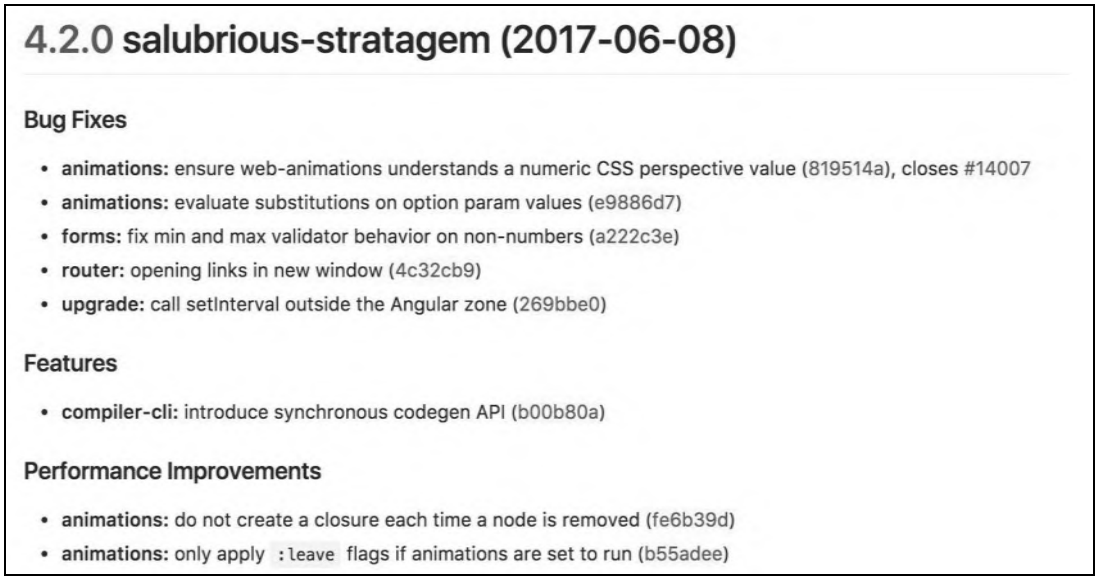

图 3-7

这样一来，除了规范化提交信息，还可以生成对应的修改日记。对于传统型的 IT 公司来说，它可以大大方便编写上线计划。

3.13.3　对比不同文档方式

下面让我们对上面的文档化方式进行一个简单的总结，如表 3-1 所示。

表 3-1

类型	适用场景	使用频率	性价比
搭建指南	项目搭建	低	高
架构图	新人进入项目	低	高
架构决策记录	架构决策变更	低	中
知识要点	遇到常见问题	高	低
可视化文档	高效展示数据	高	中
看板工具	了解业务	中	-
代码提交规范	了解代码修改原因	高	高

我们花费大量的时间来讨论文档化是因为它们真的非常重要。尽管对于开发人员来说，写文档是一件痛苦的事，但是在未来它会被证明是有价值的。当然，过去笔者也认为写文档是没有价值的，可是时间久了就发现，哪怕是一点点的记录，对于后来的人都是有帮助的。

3.14 通过流程化提高代码质量

Web 应用通常面临着上线和质量之间的博弈——只要不影响用户体验，小的 bug 往往对于项目来说是可以“容忍”的。这样就可以早些上线，实现用户价值。此外，Web 开发还受其开发的影响：一个是类似于敏捷方式的迭代式开发，另一个则是可以多次上线。

Web 开发与一些特殊领域及行业的软件开发不同，一个开发成本上亿的软件，可能只会运行、部署一次，不会有第二次机会，如原子弹的控制系统。还有一些类型的案例，就是智能汽车上的自动驾驶系统，稍有不慎就会车损人亡。Web 应用虽然更新困难，可它们还是能远程更新的。而在这些系统上，它们就更追求质量，而不是开发速度。Web 应用部署如果失败可以回滚，虽然会带来一定的财务上的损失，但是极少会带来生命危险。

因此在质量和速度方面，在 Web 开发上保持着一个微妙的平衡。

可软件开发不仅只有质量和速度的问题，还有一个产品问题——即能做出符合用户需求的产品。于是，就变成了质量-速度-需求，一个更复杂的平衡。为了交付更符合用户需求的产品，就不得不经常做一些需求变更。而这些变更的时间，往往会影响代码质量和开

发速度——实现一个需求的时间越短，测试的时间就越短，Bug 出现的可能性就越高。过去笔者遇到过，今晚上线，下午临时改需求。可想而知，测试人员是没有时间测试的。

在这个时候，持续集成只能显式地告诉我们，测试挂了，某些功能失效了，我们不应该部署这个新版本。然而并不是持续集成出问题了，我们就不能部署，仍然还是能部署的。

3.14.1　代码预处理

在代码提交之前，我们还可以进行一些常见的操作：

◎　静态代码分析（Lint），用于进行静态代码分析，常见的如 Lint4j、TSLint、ESLint。

◎　运行测试，为了不影响持续集成，我们需要在代码提交之前进行测试。

现在的编辑器（使用相应插件）、IDE 可以提高技术手段，在开发的过程中分析静态代码，并随时提高建议。如 Intellij IDEA 和 WebStorm 就会根据 TSLint 来提醒开发者注意关于 TypeScript 代码的一些规范问题。

这些分析工具主要进行代码分析，如《全栈应用开发：精益实践》一书中所说，一般会进行如下一系列的风格检测：

◎　规范函数名及变量。

◎　代码格式规范。

◎　限制语言特性。

◎　函数行数限制。

◎　多重嵌套限制。

◎　未使用代码。

……

这些规范如果没有强制就是一个游戏，于是我们通常会依赖于 Git Hooks 来做这样的事。对于一个使用 Git 来管理源码的项目来说，Git Hooks 可以做这样一些事情，即在.git/hooks 目录下查看一些内容，相应的命令如下。

```
applypatch-msg    post-merge         pre-auto-gc      prepare-commit-msg
commit-msg        post-receive       pre-commit       push-to-checkout
post-applypatch   post-rewrite       pre-push         update
post-checkout     post-update        pre-rebase
post-commit       pre-applypatch     pre-receive
```

一般而言，我们只会在两个阶段做相应的事情：

◎ pre-commit，预本地提交。通常会在该提交之前进行一些语法和 Lint 的检测。

◎ pre-push，预远程提交。通常会在该提交之前运行一些测试。

于是，在我们的这个前端项目里又写了两个 scripts，代码如下：

```
{
   "precommit": "lint-staged",
   "prepush": "ng test && ng build --prod"
}
```

在执行 pre-commit 时，我们配合 lint-staged 和 prettier 来进行代码格式化：

```
"lint-staged": {
   "src/app/*.{css,scss}": [
     "stylelint --syntax=scss",
     "prettier --parser --write",
     "git add"
   ],
   "{src,test}/**/*.ts": [
     "prettier --write --single-quote",
     "git add"
   ]
 }
```

事实上，使用 ng lint --fix 也是一种不错的方式。

我们在 push 代码之前进行测试及 Angular 的构建。单元测试运行得相当快，它可以在几分钟内完成，快速地对问题做出响应，而不是等到持续集成出问题时再去修复。

Git 提供了一个--no-verify 参数，可以让开发者不需要进行上面的验证就能提交代码。我们往往无法阻止别人做这样的事情，特别是在多个团队协作的时候。

3.14.2　手动检视代码

除了使用 Lint + Git Hooks 的方式来自动化检测代码问题，手动地去检视代码也是一个不错的检查代码的方式。这种方式可以带来一系列的好处：

（1）提高新成员的编程水平。

（2）保证代码规范得以实施。

（3）每个人对项目的代业务都会很熟悉。

（4）提供项目代码的质量。

（5）帮助成员熟悉工具和快捷键的使用。

有意思的是，就一般而言，代码检视是难以发现代码中的 Bug 的，但是它能督促其他成员认真写代码。而就代码检视来说，常见的有两种方式：

◎　常规代码检视。

◎　阻塞式代码检视。

每种方式都有一定的适应场景，下面让我们来展开讨论。

1. 常规代码检视

常规代码检视（Code Review），即团队成员聚在一起，听其他成员讲述自己最近（通常是今天）写的代码。一个完整的代码检视是从讲述相关的业务功能、技术功能开始的，而非打开 IDE 直接讲代码的实现。项目中的成员对代码有了上下文的了解之后，才能更明确地知道代码为什么需要这么写。

除了上面的基本规范，常规代码检视通常还需要规定一个固定的时间。比如在笔者的公司里，通常从每天下午 5 点开始。写了一天的代码，这个时候大概也累了，可以讨论一些技术问题。固定时间还有一个好处是，在进入每天的检视环节之前，可以提前回顾一下自己写的代码。

值得注意的是，下午下班前可能是一天最累的时候，代码检视不一定有显著的效果，因此适时地调整代码检视的时间，也可以取得不错的效果。我们曾将代码检视的时间定为

下午两点，一方面是因为午休后需要缓冲，另一方面是因为在代码检视完后要立即修改代码。

2. 阻塞式代码检视

阻塞式代码检视，是由代码提交者通知项目中的其他人，代码需要检视才能合并入代码库。上面谈到的常规代码检视是一种非阻塞式代码检视，即代码可以直接提交到主分支上。

在阻塞式代码检视里，代码提交通常是一个 Pull Request，即要求其他人把代码拉到主分支，这就要求项目中的其他开发者来帮助我们检视代码。当代码没有问题的时候，通过这个代码 Pull Request 请求，代码便会合并到主分支上。它通常与功能（Feature）分支工作流、GitFlow 工作流或 Fork 工作流一起使用。

作为一个喜欢自由的开发者，笔者不是很推荐这种方式，它会影响持续集成的构建。即我们的主（Master）分支上应该包含所有的代码，而不是将代码分散在功能（Feature）分支上。但是它仍然有相当多的适用场合，当项目中的代码质量真的不好时，就可以采用这种方式——前提是，其他检视者能够带责任感地去检视代码。

3. 其他

此外，对于技术负责人来说，平时还需要经常检视团队成员写的代码。比如每天在提交代码的时候，我们都需要从远程获取最新的代码。在这个时候，或者运行测试的时候，我们可以顺便阅读一下其他人写的代码，如果从中发现问题，可以直接展开讨论，以免影响其他开发者的开发。

3.15 使用工具提升代码质量

除了上面这些手段，事实上还有一些不错的工具，可以帮助我们改善代码质量。

3.15.1 代码扫描工具

过去，发现代码中 Bug 的方式往往是测试人员测试出来的，或者在编写测试的过程中

发现有问题。随着技术的发展，已经有一些工具能直接扫描代码，并帮我们发现代码中的 Bug，如 Sonar 就是这样的一个工具。这些代码扫描工具通常都能通过静态分析代码执行自动检视，并检测多种编程语言上的错误、代码坏味道和安全漏洞。它可以报告出各种常见的、显式的代码问题，如 switch 语句没有 default。当然，这些语法问题通常可以用 CheckStyle 来解决。

代码扫描工具还能与大部分构建工具、持续集成工具、源码管理工具配合使用。在这个时候，我们只需要配合源码管理工具或者持续集成工具，在提交代码后，进行相应的代码扫描命令，再生成测试报告。

3.15.2　IDE 快速重构

IDE 重构，即借助于 IDE 来对代码进行重构，它通常是由快捷键来触发的。它将重构的主要工作交给 IDE，而不是由开发人员通过修改代码来完成。多数时候，我们只需要按下快捷键，再输入一些必要的名称、参数，重构就完成了。因此，它是在我们每天的开发中不可或缺的一部分。可以随时随地进行，而不需要预留一个专用的重构时间。

在日常开发中，我们编写的一些代码坏味道，都可以直接由 IDE 的快捷键完成重构。如 Intellij IDEA 自带的强大的重构功能，只需要按下快捷键，就可以进行大量的代码重构。以下是 Intellij IDEA 中使用 Commond + N 进行内联方法的示例，如图 3-8 所示。

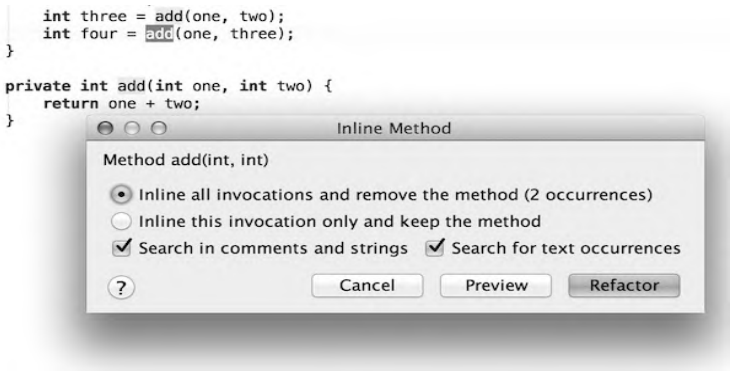

图 3-8

它可以自然而然地发生，而其犯错的成本又相当小。

此外，还有一种方式可以提高代码的质量，那就是测试。

3.16 测试策略

测试策略是架构中重要的一环，它用于保证项目质量及代码质量。测试并不能直接地减少 Bug，它可以间接地减少代码反复修改出现的 Bug——每当我们修改代码的时候，就极有可能破坏系统原有的功能。

过去，在 Web 应用中有一个很经典的“测试金字塔”的概念。随着 BFF 和前后端分离架构的实现，现在的测试金字塔概念与过去不一样，其呈现的是四层结构，如图 3-9 所示。

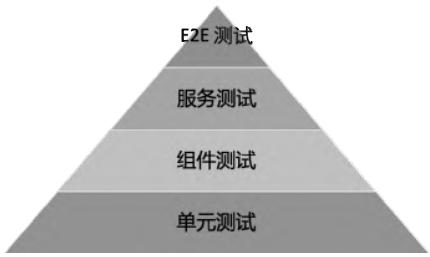

图 3-9

金字塔中的四个层级如下：

◎ 单元测试。

◎ 组件测试。含快照测试（Snapshot）。

◎ 契约测试/接口测试。相当于服务测试。

◎ E2E 测试。端对端测试、集成测试。

在项目初始时，我们不可能一次性地引入所有的测试，而是一步步慢慢往上堆加测试的类型。从这个角度来看，周期越长的项目越容易有完整的测试链。反之，周期越短的项目只会拥有基本的测试，极端情况下便是没有测试的。如国内的大部分前端项目，这些项目都是没有测试的。正因为如此，国内关于前端测试的内容不多。编写测试的时间花费多，是人们不写测试的一个重要原因。当编写测试的时间不被领导所承认时，测试就变得难以推广。

与此同时，越是底层的测试，比如单元测试，在实践上越容易，所花费的时间也越少，成本相对也越低。而诸如 E2E 这样的测试，在实践上相对困难，单个测试的编写需要的时间长，成本高，但是 E2E 测试与单元测试相比更加全面。

3.16.1　单元测试

对于单元测试来说，我们需要做的事情有：

（1）选定一个单元测试框架。

（2）创建测试规则。

（3）制定测试覆盖率的最小值。

在前端领域里，由于开源作者和贡献者的喜好作者，受不同的前端框架的适配影响，在其文档上都为开发者推荐不同的单元测试框架。有的框架，默认会使用某些测试框架。这些单元测试框架之间的区别并不是太大，其用法都是相似的。编写测试的时候，采用的往往也都是三段式风格 Given-When-Then，对应的中文便是：假设-当-那么。三段式风格解释如下。

◎　假设（Given）：一个上下文，指定完成测试所需要的条件。

◎　当（When）：进行一系列操作，即所要执行的操作，如单击某个按钮。

◎　那么（Then）：得到一系列可观察的后果，即需要检测的断言，如按钮被隐藏。

下面是使用 Jasmine 测试一个 Angular 登录功能的示例：

```
it('should request login if not logged in', () => {
    // Given 给定以下条件
  userService.isLoggedIn = false;
    // When 执行事件
  fixture.detectChanges();
    // Then 预期结果
  const content = el.textContent;
  expect(content).not.toContain('Welcome', 'not welcomed');
  expect(content).toMatch(/log in/i, '"log in"');
});
```

而 React 框架则推荐使用 Facebook 自家的 Jest，其测试的编写过程也和 Angular 相似。下面是一个使用 Jest 测试的示例：

```
describe('helper test ios', () => {
  it('should call linking function', () => {
    // Given 给定以下条件
    const spy = jest.spyOn(Linking, 'canOpenURL');
    // When 执行事件
    Helper.openLink('https://forum.growth.ren');
    // Then 预期结果
    expect(spy).toBeCalledWith('https://forum.growth.ren');
  });
});
```

上述代码中的 spyOn 是测试替身（Test Double）的一种，它的出现是为了达到测试的目的，并且减少被测试对象的依赖。上述方式是测试替身中的 Mock，它用于确保某个方法被调用，即 Linking 对象中的 canOpenURL 方法。对应的代码中体现便是，我们断言它被调用了，并且带有参数'https://forum.growth.ren'。另外一种常用的测试替身的方式是 Spy，其表现方式是：使用简单的行为来替换复杂的行为，或返回一个特定的结果。下面是示例代码：

```
spyOn(authService, 'login').and.returnValue(
   Observable.of({
      isSuccess: true,
      response: {
         token: ''
      }
   }));
```

我们需要定义一个单元测试规则。在正常情况下，我们应该测试大部分函数。但是我们不一定有足够的时间来对每个功能进行测试，对于单元测试来说，下面是工作中常见的一些规则。

（1）必须进行测试的是通用、公用的 Utils 函数。

（2）复杂交互操作需要进行一定的测试。

（3）网络请求可以交给契约测试，或者不进行测试。

单元测试通常只用于测试代码中的逻辑，而对于隐藏在模板中的逻辑来说，单元测试往往难以进行测试。因此，在编写业务逻辑的时候，尽可能把代码写在 JavaScript 或

TypeScript 代码中。可以尝试在项目中采用驱动测试开发，以测试来驱动业务逻辑。采用这种方式来开发，可以保证功能都得到测试。

3.16.2　组件测试

组件测试是指，对项目中编写的组件进行测试。对于项目中用到的第三方组件应该是被组件的开发者测试过的——如果一个组件的测试不够、采用的范围比较小，那么我们在选型的时候需要慎重。

现代的前端应用多数采用组件化的架构，组件的测试和普通的单元测试区别并不大。但是，它们应该是被分开来进行讨论的。

当我们对组件进行测试的时候，它包含了两部分内容，一部分是组件自身功能的测试，这部分是单元测试；另外一部分则是对组件进行测试。在包含了单元测试之后，组件测试所要测试的便是组件的行为。当我们单击组件的一个按钮的时候，除了对应的模型、数据发生变化，对应的 DOM 也会发生一定的变化，而我们所测试的便是这里的 DOM。对 DOM 的行为测试，采用的方式与单元测试是相似的。

在一些框架里，则有更简单的测试方式——快照（Snapshot）测试。如在 React 框架中，可以通过 Virtual DOM 生成的快照来验证组件的行为是否正确。下面是 React 框架的快照的测试示例：

```
it('renders correctly', () => {
  const todoList = renderer.create(
    <TodoList data={TODO_LISTS['zh-cn'].hello} />,
  );
  const todoListJson = todoList.toJSON();
  expect(todoListJson).toMatchSnapshot();
});
```

执行完测试后会生成视图对应的快照文件，用于后期对比快照。当组件中传入不同的参数，或者组件间进行相应的交互时会产生状态的变化。通过记录这些状态和属性的变化，便能验证组件是否被修改或是否受到影响。值得关注的是，快照测试容易陷入为了测试而测试的情景，即我们可能忽略了状态的变化，而直接提交对应的快照修改。

3.16.3 契约/接口测试

契约测试（Contract Test），即对前后端约定的 API 进行测试，这个 API 约定又可以称为契约。在 Web 应用里会分别交由前端和后端进行测试。对于后端来说，契约测试用于验证后台生成的 API 是否和契约 API 一致。而对于前端来说，主要测试后台的 Mock Server 返回的 API 是否满足前端需求。对于有些不需要授权服务的 API，则可以直接验证后台 API 是否和前端保持一致。因此在测试的时候，需要依赖于对应的 Mock Server 或者后台服务，才能确保测试能正常运行。

编写契约测试与单元测试和 E2E 测试的主要区别在于，测试的对象不同。它和服务测试类似，不再单纯地对一个系统进行测试，而是结合其依赖的后端服务进行测试。与 E2E 测试不同的是，编写契约测试只是从 API 级别对代码进行测试，而不是从用户的级别来进行测试。

由于契约测试依赖于契约，这部分内容我们会在第 8 章中进行详细的介绍。

E2E 测试是一个很熟悉的话题，其表现行为类似于真实的用户行为：启动浏览器，模拟用户行为进行操作。E2E 测试要花费较长的时间编写，由于需要真实的运行，其运行时间也比较长，并且还需要反复修改。因此在考虑 E2E 测试的时候，应该考虑经常出错的、核心的功能。

诸如 Angular 框架，也推荐了自己的 E2E 测试框架 Protractor。此外，在选定 E2E 测试的时候，我们可以考虑是否采用 BDD 的方式进行。

BDD，英文全称 Behavior Driven Development，中文含义为行为驱动开发，它是一种敏捷软件开发的技术，它鼓励软件项目中的开发者、QA 和非技术人员或商业参与者之间的协作。

与一般的自动化测试（如单元测试、服务测试、UI 测试）不一样，BDD 是多方参与的测试开发方式。如在使用 Protractor 写 Angular 的 E2E 测试的时候，所有的测试都是前端测试人员编写的。BDD 最重要的一个特性是：由非开发人员编写测试用例，而这些测试用例是使用自然语言编写的 DSL（领域特定语言）。下面是一个 BDD 测试的业务层实现：

```
* 当我在网站的首页
* 输入用户名 "demo"
* 输入密码 "mode"
```

```
* 提交登录信息
* 用户应该跳转到欢迎页
```

BDD 测试在代码底层的实现如下：

```
defineSupportCode(function({Given, When, Then}) {
    Given('当我在网站的首页', function() {
        return this.driver.get('http://0.0.0.0:7272/');
    });

    When('输入用户名 {string}', function (text) {
        return this.driver.findElement(By.id('username_field')).sendKeys(text)
    });

    When('输入密码 {string}', function (text) {
        return this.driver.findElement(By.id('password_field')).sendKeys(text)
    });
...
});
```

这样的测试可以尽可能保证业务的准确性，也能真正体现出 E2E 测试的价值，即模拟真实业务场景下的用户行为。在一些公司里，这些测试都是由专业的测试人员来编写的，不需要开发人员来编写。此时的测试不再局限于前端的角度来进行测试，而是站在整个系统的角度来考虑测试。

3.17　小结

在本章中，我们从浏览项目的代码开始，讨论关于代码不同部分的规范制定——从代码的组织架构、风格，到命名规范、开发工具等。然后，我们从搭建指南开始入手，介绍了在前端项目中的各种文档类型，以及通用的文档工具。最后，我们回到代码中，一步步地将代码从本地提交转到了测试环节。此外，还讨论了与代码质量相关的内容，以及如何采用合适的代码测试策略。

4

第 4 章

架构基础：设计构建流

在第 3 章中讲述了如何去搭建和熟悉开发环境这样的工作，作为一个资深的前端开发人员，我们要有能力去设计一个完整的应用开发流程。而这个流程实际上就是要设计一个构建系统，通过它将第 3 章中设计的流程以代码、工具的形式来加以规范。

构建系统是一个 Web 应用不可缺少的一部分，其核心用途是，帮助开发者从源代码开发开始，构建出最后可用的目标软件。开发人员还能通过构建系统中的构建工具添加更多实用的功能，如修改监测代码触发代码编译后自动地刷新浏览器，在构建系统时配置不同的环境参数，在本地运行的时候做反向代理，等等。通常，在一个前端应用中，构建系统需要做下面的一些事情：

- ◎ 依赖管理及安装。
- ◎ 优化开发环境。
- ◎ 代码质量检测。

◎　编译及打包。

◎　测试及部署。

……

这也是接下来我们所要学习的内容。

与过去相比，前端应用的构建已经变得相当复杂。前端应用需要管理依赖版本，以防止在不同的开发机器上因依赖版本不一致而出错；需要编译代码，而不再是直接合并代码、导致混淆；需要编译 SCSS 为 CSS，而不是直接使用 CSS。对于一个现代的前端应用而言，通常需要做一些步骤，如图 4-1 所示。

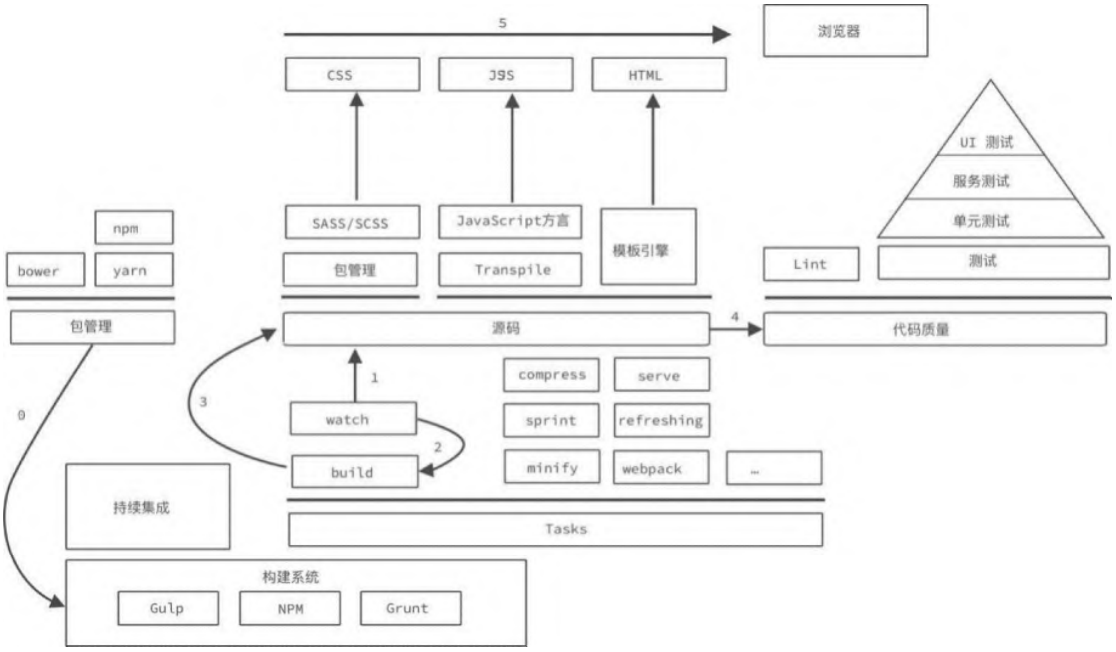

图 4-1

尽管现有的前端框架提供了一些工具，帮我们屏蔽了底层构建的复杂度，但是在实际开发的过程中会遇到一些问题，使得我们不得不去优化底层的构建系统。下面是混合应用框架 Ionic，其命令行工具执行 ionic serve 时的启动日志如下：

```
[11:43:58]  ionic-app-scripts 1.3.4
[11:43:58]  watch started ...
[11:43:58]  build dev started ...
```

```
[11:43:58]  clean started ...
[11:43:58]  clean finished in 1 ms
[11:43:58]  copy started ...
[11:43:58]  transpile started ...
[11:44:03]  transpile finished in 5.17 s
[11:44:03]  preprocess started ...
[11:44:03]  deeplinks started ...
[11:44:03]  deeplinks finished in 132 ms
[11:44:03]  preprocess finished in 132 ms
[11:44:03]  webpack started ...
[11:44:03]  copy finished in 5.49 s
[11:44:26]  webpack finished in 22.94 s
[11:44:26]  sass started ...
[11:44:29]  sass finished in 3.21 s
[11:44:29]  postprocess started ...
[11:44:29]  postprocess finished in 85 ms
[11:44:29]  lint started ...
[11:44:29]  build dev finished in 31.57 s
[11:44:29]  watch ready in 31.69 s
```

在这个过程中，它会完成如下工作：

（1）启动“代码修改监测”服务。

（2）开始构建 Dev（开发）版本的代码。

（3）清除上次构建的代码。

（4）将 TypeScript 转译（或称编译）为 JavaScript，以便于直接在浏览器上调试。

（5）对 DeepLinks（在 iOS 上称为通用链接（Universal Links））进行预处理。

（6）复制资源文件到对应的目录中。

（7）将 Sass 编译为 CSS，以便于直接在浏览器上调试。

（8）执行一些构建后的处理。

（9）开始执行 Lint。

与早期的前端应用相比，这样的一个 Web 应用构建流程相当复杂。如果是自己从头编写这样一个构建流程，那么它可能比完成项目的时间还长。从上面的日志来看，这个框架的构建工具是 webpack。它帮助我们自动完成了大量的底层细节，使得我们只需要编写

相应的配置就可以实现构建流。

那么，接下来让我们探索一下其中比较重要的几部分内容。

4.1　依赖管理工具

在项目的源码管理中，依赖的管理是相当重要的一部分。它可以使代码在项目的各个开发机器、持续集成服务器上快速地搭建、运行、发布，而不需要确认依赖版本等额外的工作。早期没有依赖管理软件时，项目的依赖放在代码库中——每次更新依赖的时候，均需向项目的代码库提交一个新的版本，这导致项目的代码库越来越大，对新成员越来越不友好。在有了依赖管理之后，我们只要执行安装命令，如 npm install，再执行运行服务就可以进行开发了。

（1）不同的模块化方案

在继续讨论依赖管理之前，我们不得不再次提及 AMD、CommonJS 和 UMD，它们是三种不同的 JavaScript 模块化方案。这些模块化方案，在一定程度上影响着项目的依赖管理。

AMD（Asynchronous Module Definition）即异步模块定义，其典型的表现形式是通过构建好的 JavaScript 库文件，直接在浏览器中引入依赖，典型的框架有 jQuery。为了引用这些采用 AMD 方式的库，我们还需要使用诸如 require.jsjs 这样的 AMD 管理库来管理依赖。如果这些 AMD 库是在浏览器上通过 script 直接引入并运行的，那么它们就可以使用 bower 来进行管理。

CommonJS 最初被用于在非浏览器端使用 JavaScript 来管理模块，比如使用 Node.js 来开发服务器应用。设计的目的是避免模块定义全局对象，即 JavaScript 的作用域问题。在 CommonJS 库里，每个文件就是一个模块。开始时，CommonJS 在 Node.js 服务端运行，随着 Browserify 和 webpack 能将这些 CommonJS 编译为浏览器能运行的 JavaScript，越来越多的前端框架直接使用 CommonJS 来管理模块。

随着 AMD 和 CommonJS 的流行，又产生了 UMD（Universal Module Definition，通用模块定义），它是 AMD 和 CommonJS 的糅合，它会判断是否是 Node.js 的模块。下面是一个 UMD 库的示例代码：

```
    (function (global, factory) {
      typeof exports === 'object' && typeof module !== 'undefined' ? factory
(exports) :
      typeof define === 'function' && define.amd ? define(['exports'], factory) :
      (factory((global.oan = {})));
    }(this, (function (exports) { 'use strict';
    ...
```

这段代码中的 typeof exports === 'object'就是用于判断 Node.js 中的模块（Exports）是否存在。如果不存在，则使用 AMD 的方式来加载模块。

那么，让我们从 AMD 方式讲起。

（2）AMD 方式：通过 Bower 进行管理

谈到 AMD 库的管理，一般会有两种方式，一种是直接下载对应的库，另一种是可以由后来的 Bower 进行管理。

过去前端的基础设施不完善，没有统一的软件包中心，开发者需要在搜索引擎或者 GitHub 上找到代码库，然后下载解压，再从中复制出代码文件、资源文件等。

在有了 Bower 之后，开发者只需要在 bower.json 文件中添加所需要的依赖，就可以由 Bower 自动下载这个依赖的相关内容——该库打包到 bower 的服务器上（Bower 是一个包管理工具），它可以管理包含 HTML、CSS、JavaScript、字体甚至图像文件的组件。Bower 的工作原理是，从各个地方获取安装包，并搜索、查找、下载和保存正在寻找的内容。因此，Bower 在过去的某段时间里非常流行。

值得注意的是，由于现今的大部分新的前端应用都是通过 CommonJS 及 AMD 的方式来引入依赖的，所以 Bower 的使用场景越来越少。

（3）CommonJS 方式：通过 NPM/Yarn 管理

由于 CommonJS 早先是用在 Node.js 项目上的，所以其包管理工具 NPM 也与 CommonJS 有了更多的交集。

NPM（Node Package Manager，Node 包管理器）是 JavaScript 世界的包管理工具，并且是 Node.js 平台的默认包管理工具。开发人员通过 NPM 可以安装、共享、分发代码，并管理项目依赖关系。此外，这些包会分发到 NPM 的官方网站 npmjs.org 上，以便对 NPM 包进行更好的管理。

除了 NPM，还可以使用 Yarn 来管理依赖。Yarn 是由 Facebook 推出的一款新的

JavaScript 包管理工具，其特色是 Yarn 会缓存它下载的每个包，因此不用重复下载就可以加快安装速度。随着 Yarn 提出缓存包的功能，NPM 也提供了相似的功能。

目前主流的前端应用框架（React、Angular、Vue）支持并采用的方式主要以编译构建为主，大都采用 NPM + CommonJS 的形式来管理依赖。

4.2　软件包源管理

在确认了安装的工具之后，我们需要考虑软件包源的问题。NPM 使用 package.json 来管理依赖，在这个文件中定义依赖的内容和版本。软件包源有以下几种方式：

- 公有/私有软件包源服务器。
- 基于源码版本管理服务器，比如直接使用 Git 服务器来管理和分布库的版本。
- 本地的软件包，使用相对路径导入本地仓库。

这几种方式可以单独使用，也可以结合起来使用。公有/私有软件包源服务器对于直接使用源服务器来说，有以下几种方式：

- npmjs.org 提供的官方源。它可以提供最新的软件包，但是国内开发者使用的时候会受到影响。
- 淘宝的 cnpm 源。它可以为国内的开发者提供更快速的下载服务。
- 搭建私有的 NPM 服务器，提供更为安全快速的 Node.js 环境。

对于小型组织来说，软件包通常来源于 NPM 包管理中心。而在大中型企业中，对于软件包的管理则会更谨慎一些。除了放置那些不适合公开的库，它还可以保证应用的构建不受外部的干扰——比如外部软件包中心宕机。对于那些因为安全原因隔离内部/外部网络的公司来说，则更需要这样的软件包仓库。

他们会搭建自己的内部软件包仓库，比如流行的 Java 语言里的 Maven 私有仓库软件 Nexus。在 Node.js 领域里，则可以使用 cnpmjs 来搭建自己的私有化包服务器。

1. 基于源码版本管理服务器

对于一些内部的前端库，除了自己搭建 NPM 服务器，还可以通过 Git 服务器及 Git Tag 的方式进行管理。下面是一个新版本的发布代码：

```
git tag v1.0.7
git push origin v1.0.7
```

在依赖 package.json 中安装对应的版本：

```
"dependencies": {
  "adr": "git+http://github.com/phodal/adr/#v1.0.7"
}
```

需要注意的是，使用这种方式管理的包，需要将构建完的源码提交到代码库。比如，使用 TypeScript/ES6 编写的库，需要提供最后编译生成的目标文件（通常位于 dist 或者 build 目录下），并将其提交到代码库中，这会使得源码库由于版本越来越多而变得越来越臃肿。

因此，在这种情况下，可以创建一个新的代码库来管理构建后的包，将源码与目标代码进行隔离。

2. 本地的软件依赖包

本地的软件依赖包是指，把应用的依赖放置在本地。它通过包管理工具来支持 file 协议并引入依赖，即在 package.json 中通过"aofe":"file:aofe"的形式来引入依赖。

在拥有 Git 服务器的环境下，这种方式已经不多见，其使用场景如下：

◎ 构建的包过大，不适合使用 Git 下载。

◎ 在开发应用的同时开发依赖。在开发一个前端库的时候，我们会在这个库里包含相应的 Demo，方便对该库的调试。但是当用户的应用、第三方库与我们的库冲突时，就不得不通过下载代码来进行调试。这时需要通过相对文件夹路径的方式来引入库，以方便我们进行调试。

在前端不断扩张的未来，这种本地的管理方式不会消失。

3. 前端应用的编译

获取软件依赖，并在本地安装依赖之后，我们就可以尝试进行编译和打包。按照惯例，我们执行 npm build 就能构建出这个项目的目标应用。在安装依赖后，构建应用的目的是

确保依赖是正确安装的。在笔者的搭建经验里，这一步并不总是成功的——其他人提交的未经测试的代码可能是有问题的，它会影响我们的构建。

在进行构建的过程中，除了与钩子相关的操作，很大一部分是在编译相关的源码。

浏览器在渲染页面的时候使用 HTML + JavaScript + CSS 的形式。不论我们在编写代码的时候采用什么形式，最后都需要以这种形式来运行。于是，我们就需要将应用编译成 JavaScript、CSS，以及对应的 HTML DOM。

4. JavaScript 的编译

JavaScript 是一门重要的语言，尽管它最初是在 10 天内设计出来的。它具有各种神奇的语法，它像魔法一样不时地让我们发现一些“惊喜”。由于 JavaScript 中糟粕的比重超出了预料，以至于大量的使用者在吐槽其设计。正是因为 JavaScript 的流行，使得人们越来越正视这个问题。不满足现状的开发人员，开始创建一门又一门基于 JavaScript 的语言，以获得更好的开发体验。这些行动使得原先可以直接在浏览器上运行的 JavaScript，改为编译后再运行。

CoffeeScript 是早期较为流行的 JavaScript 语言，现在使用的人已经越来越少，人们可以运行越来越好的 ES x（ES6、ES2015、ES2016、ES2017）或者是 TypeScript。过去受限于 JavaScript 语言的各种问题，有大量的应用是用 CoffeeScript 来编写的。因此，CoffeeScript 让喜欢“造轮子”的开发者意识到这么做也是相当合适的——先使用其他语言编写，再将其编译到 JavaScript——通过 coffee 命令来编译，就能到浏览器上解释运行。即我们可以使用一个更完备的语言来编写应用，而不是受限于语言本身。

ES6，即 ECMAScript 6，尽管是下一代的正统 JavaScript，受限于浏览器的支持，它仍然需要编译为 ECMAScript 5 才能在大部分浏览器上运行。主流的前端框架都支持使用 ES6 来开发，只需要配合编译器 Babel 的 CLI，就可以使用 babel 来编译代码，比如 babel src -d lib 就可以将 ES6 代码编译成 ES5 代码。因此，ES6 与 TypeScript 都是目前主流的框架。

Typescript 是 JavaScript 的一个严格超集，它除具有 ECMAScript 的特性外，还带来前端大规模应用中最需要的静态类型检查和基于类的面向对象编程。TypeScript 使用自己的 TypeScript 编译器，通过 tsc 将 TypeScript 代码转换为 ES5 或者 ES6 的代码，配合 tsconfig.json 就可以大规模地应用于前端应用开发。同样地，现有的主流框架也可以支持 TypeScript。

此外，还有 ClojureScript 等不同的语言。我们唯一需要做的就是：找一个喜欢的

JavaScript 语言版本。不同的语言有自己的适用场景，作为一个习惯规范的开发者，笔者更喜欢 TypeScript 这门语言。它烦琐，但是规范性很强。

选择语言，也会在一定程度上影响我们引入包的方式。不过，现今的大部分应用都使用 NPM 来管理依赖。代码在编译的过程中，会自动导入所需要的依赖，然后构建出对应的目标文件。

5. CSS 的编译

在现在的前端应用里，CSS 的展现方式不再像过去那么简单，出现了一些新的方式。架构上的差异，也决定了我们使用哪种方式来对 CSS 进行编译。

传统方式是指，通过<link>标签加载，在全局中使用上 CSS 的方式。我们只需要按自己的偏好，选定一个 CSS 预处理器（如 Sass、LESS、Stylus），在构建的时候将其编译为 CSS，就可以引入使用了。

对于小型项目来说，全局化的 CSS 并没有问题。但是当项目的规模越来越大时，全局的 CSS 就变得越来越难以使用。在这种时候，使用哪个 CSS 预处理器就不再重要了。CSS 与生俱来的继承与覆盖，会导致我们在 CSS 代码中写入大量的“!imporntant”。

```
p {
    word-break:break-all !important;
}
```

6. CSS In JavaScript

为了进一步避免这种问题的发生，就有了 CSS in JavaScript——即 CSS 在 JavaScript 代码中。实际上，CSS in JavaScript 是组件化架构不断发展的一个产物，它可以解决 CSS 全局作用域的问题。传统的 CSS 加载方式是，将所有的样式写在 CSS 文件里；而 CSS in JavaScript 则是通过 JavaScript 将 CSS 样式附加到元素上。这意味着，组件的样式可以独立运行，并且能在 JavaScript 和 CSS 之间共享函数和变量。

下面是 Angular 框架生成的 CSS in JavaScript 代码示例：

```
xb=tr({encapsulation:0,styles:[["p[_ngcontent-%COMP%]{word-break:break-all}"]],data:{}})
```

从代码中，我们可以清晰地看到 styles 对象中绑定的 p[_ngcontent-%COMP%]{word-break: break-all}。它在浏览器运行的时候会被解析为：

```
p[_ngcontent-c8] {
    word-break: break-all;
}
```

对比之后，会发现这里的%COMP%是一个变量模板。在运行时，对应的组件会绑定新的属性，对应的属性也会动态添加 CSS。具体实现代码如图 4-2 所示：

图 4-2

因此，当我们选择组件化架构的时候，更多地会考虑使用这种方式来实现。尽管如此，两种方式之间并不是对立的，我们也需要通过传统的 CSS 方式来实现一些全局的样式，比如颜色、字体等的设置。我们使用 CSS in JavaScript 是为了更好地控制组件中的样式。

7. 模板编译

HTML 是网页结构的一个重要支撑，在经典的网页里，HTML 都是由静态的 HTML 代码或工具生成的。与 JavaScript 和 CSS 不同的是，多数的 HTML 并不是直接编译生成的，而是间接地通过模板来生成 HTML DOM 节点的。

早先我们并不会直接编译模板为 JavaScript 代码，而是在运行时由模板引擎来实时编译模板。由于模板在网页运行的时候，还需要将其解释为 JavaScript 可以处理的数据，所以会进一步导致一定的性能问题。

进一步地，在构建的过程中，直接将相应的模板编译到源码中。当应用程序在线上运行的时候，直接调用相应的方法来更新数据或者触发对应的方法。我们编写的模板文件与之前相比并没有太大的区别。

下面是 Angular 编译后的模板：

```
    (t()(),Ir(15,0,null,2,2,"a",[["href","tel:987654321"],["mat-raised-butto
n",""]],[[1,"tabindex",0],[1,"disabled",0],[1,"aria-disabled",0],[2,"_mat-an
imation-noopable",null]],[[null,"click"]],function(t,e,n){var  i=!0;return"click"
===e&&(i=!1!==io(t,16)._haltDisabledEvents(n)&&i),i},Ly,Fy))
```

其原先的 HTML 模板如下：

```
    <mat-list-item role="listitem">
      <a mat-raised-button href="tel:987654321">Tel 987654321</a>
    </mat-list-item>
```

从编译并 minify 后的代码里，我们可以清晰地看到与 DOM 节点相关的内容——如数组中的“href”,“tel:987654321”，即对应的 HTML 中的 href 属性。

相似地，对于 React、Vue 这一类框架也是如此，将模板编译为 JavaScript 可以提高运行效率。因此在今天看来，尽管我们依然在编写 HTML，但是实际上它们只是像模板一样，相应的细节都交给前端框架的脚手架来完成了。

4.3 前端代码的打包

在主流的前端框架里，编译及打包的工作都已经交给对应框架的 CLI（Command-Line Interface，命令行界面）来完成了，例如 Angular 的 CLI、Vue 的 CLI，以及 Facebook 推出的 Create-React-App。在这些工具里，只有 Angular 可以真正地实现开箱即用，这和 Anguar 本身大而全的设计有关。因此，当我们选择使用 Vue、React 等其他框架时，或多或少地需要进行一些定制，这就要求我们对应用本身的打包有一定了解。

代码打包的最后一步就是，一一将代码“拼接”到一起，压缩并混淆代码，最后构建出目标代码文件。期间，还会进行可选的摇树优化（Tree Shaking，顾名思义就是去掉不需要用到的代码），其主要依赖于 ES6 模板及其带来的静态分析。这个目标文件是按我们的需要而产生的，如下是 Angular 项目构建出来的代码相关文件（通过 ng build--prod 生成）：

```
    chunk {0} runtime.a66f828dca56eeb90e02.js (runtime) 1.05 kB [entry]
[rendered]
    chunk {1} styles.3ee11a5cb61aea461490.css (styles) 58.4 kB [initial]
```

```
[rendered]
    chunk {2} polyfills.299c94e64165a17ec3cb.js (polyfills) 98.3 kB [initial]
[rendered]
    chunk {3} main.344eb905a64866d91056.js (main) 601 kB [initial] [rendered]
```

上面的代码中涉及我们需要的几个目标文件：

◎ 用于管理运行时（如路由懒加载）的 runtime.js 文件（在 Angular 6.0 之前的版本里是 inline.js）。

◎ 样式相关的 styles.css。

◎ 解决 JavaScript 在不同浏览器兼容问题的 polyfills.js。

◎ 程序相关逻辑的 main.js。

上面这段代码，除了进行代码编译、打包，还对 CSS 和 JS 等静态资源文件进行了重命名。

4.4　设计构建流

构建流的设计是逐步演进的，很难在项目初始时便设计出完整的构建流。对于复杂的应用来说，在开发应用的过程中会不断地完善。对于简单的应用来说，事先设计好的流程可能用不上。

构建流通常来源于：

◎ 过去的开发经验。

◎ 前端框架提供的命令。

◎ 应用脚手架提供的功能。

不管哪种形式提供的构建流，多数时候都需要经过一定的改造才能满足我们的需求。

1. 如何设计构建流

摆在我们面前的第一个考验是，如何设计应用的构建流？我们需要进一步了解如下内容：

（1）我们需要做的任务有哪些？

（2）每个任务具体的步骤拆解有哪些？

（3）部署时，其对应的形式是怎样的？

（4）有没有现成的插件可使用？

◎ 任务。应用的多数流程都是通用的，对应的任务也是通用的。这些命令都是我们耳熟能详的，比如用于安装依赖的命令 npm install，用于测试的命令 npm test，等等。稍有区别的是，其背后对应的细节处理不一致，也就是对于这些任务的拆解。

◎ 步骤拆解。任务本身的区别并不大，其背后的步骤往往差异甚大，尤其是常见的 build 命令。build 命令一方面受前端框架的影响，另一方面则受项目复杂度的影响。越是复杂的项目，其背后的构建流越复杂，比如其需要依赖于不同的子模块、定义度过高的自定义脚本。尽管 build 命令只有一个，我们还是需要将其拆解为多个子命令，如 build:icon、build:lib，等等。

◎ 展现形式。虽然多数的应用都是以 Web 应用的形式部署的，但是并非所有的应用都是需要部署的。当我们提供底层库时，提供的是一个库和相应的文档；当我们提供一套 UI 框架时，构建的是一个库+UI 组件的演示库。不同类型创建的应用，其展现形式都有所不同。

◎ 插件。在设计构建流程的过程中，不得不考虑插件对我们的影响。比如，我们要打包图标字体（Icon Font），如果没有现成的工具，则可能不会考虑这种形式——多数时候，我们并不会自己来编写这样的插件。当我们只能找到一个对应的插件时，如果它与所需的构建流程不符合，那么我们要修改与插件相应的代码，或重新设计流程。当且仅当有多个插件时，我们才能打造更流畅的构建流。

在实施的过程中，我们还需要一个构建工具，它可以是 Grunt、Gulp、webpack 等中的一个，或者直接使用 NPM 来完成。这些工具并没有太大的区别，要做的事情的复杂性决定了使用的工具。

2. 构建工具：自动化任务

构建工具，相当于是一个自动化任务工具，也可以称之为任务管理器。我们通过这个构建工具和其提供的接口来提供应用程序的构建服务。有的构建工具如 NPM 或者 make 只是简单地帮助我们运行命令，它不能提供一些可编程的接口。有的工具如 gulp 和 grunt，

会提供流（Pipe）以帮助我们更好地设计构建流。

过去，受到后端项目的影响，在 Node.js 还没有成熟之前，我们可能会使用 Gradle（构建脚本以 build.gradle 文件的形式存在于项目中）和 Make（构建脚本以 makefile 文件的形式存在于项目中）等工具进行构建。而现在，基本上以下面的三种形式存在：

◎　NPM 脚本，适用于脚本简单的应用构建。

◎　Gulp/Grunt，适用于复杂的项目工作流构建。

◎　webpack，进行项目的模块化打包。

每个工具都有一定的适用场景。

3. 使用包管理工具构建 NPM

我们可以通过 NPM 脚本来执行构建工作。这些脚本存在于 package.json 中的 scripts 部分中。它们可以通过执行 npm run[任务]来运行，如 npm run build。

如果只是简单的构建，如 Angular 框架中的 ng build 外加一些参数，那么直接使用 NPM 自带的 package.json 就可以完成相应的配置，代码如下：

```
    "scripts": {
      "build": "tsc && cp -a views dist",
      "build:app":  "electron-packager  --ignore=components  --ignore=editor/
node_modules --ignore=nlp . Phodit --platform=darwin --icon=assets/imgs/icons/
mac/icon.icns --overwrite",
      "watch": "tsc -w",
      "lint": "tslint -c tslint.json -p tsconfig.json",
      "start": "npm run build && electron ./dist/native/main.js"
    }
```

比如在上面的示例中，我们定义了 5 个任务：build、build:app、watch、lint、start，其对应的值便是命令执行的详细内容。

然而，对于构建稍微复杂一点的项目来说，它相当难维护。如上面的 build:app 任务，实际上是一段字符串。当字符串越来越长时，它本身就难以阅读，更不用说添加更多的任务。我们使用 JSON 文件来管理任务，而 JSON 文件本身不能编写注释，所以任务很难在代码中文档化。

当脚本越来越复杂时，我们就考虑使用脚本文件的形式来管理，如 node build.js。既

然这么复杂，我们就可以考虑更实用的方式，找一些现成的工具来管理。

尽管如此，在实现的时候，我们一般都会结合 NPM 脚本与 webpack 或 Gulp/Grunt 使用，代码如下：

```
"build": "webpack --config webpack.dist.config.js"
```

这样做的好处是：第一，使用一致的命令行接口能提供友好的开发体验；第二，未来在替换 webpack 的时候，对于其他开发者来说是无感知的。

4. 使用构建工具构建 Grunt/Gulp

对于复杂的构建流，我们就可以考虑使用 Grunt 或者是 Gulp 来完成构建。

- ◎ Grunt 是最早的前端构建和原生构建工具之一，只需要找到适当的插件，加上适当的配置，就可以完成构建。作为早期的构建工具，它拥有大量的插件，因为配置简单，所以相当受欢迎。
- ◎ Gulp 是一个基于流的构建工具，适当的插件外加适量的流操作代码，可以管理复杂的任务。由于使用流编程的方式，它更能胜任复杂的构建项目。

两者都是相当流行的构建工具，只是稍有区别。Grunt 通过 Gruntfile 进行配置，而 Gulp 则通过 Gulpfile 进行编程。随着项目变得越来越复杂，开发人员发现通过配置的构建脚本，越来越难以使用——有相当多的任务是不能简单地通过配置来完成的。尽管使用流编程稍微复杂一些，但是它能带来更灵活的自定义任务。

下面是一个 Grunt 用于编译 Sass 为 CSS 的任务示例：

```
//代码位于 "chapter04/gulp-grunt-webpack-compare"目录下
module.exports = function (grunt) {
  grunt.initConfig({
    sass: { // 编译 Sass 任务
      dist: {
        options: {
          style: 'expanded'
        },
        files: {
          'dist/css/styles.css': 'src/styles.scss',
        }
      }
    },
```

```
    cssmin: { // 压缩 CSS 任务
      dist: {
        files: {
          'dist/css/styles.min.css': 'dist/css/styles.css'
        }
      }
    },
  });

  grunt.loadNpmTasks('grunt-contrib-sass');
  grunt.loadNpmTasks('grunt-contrib-cssmin');

  grunt.registerTask('default', ['sass', 'cssmin']);
};
```

在代码中的 initConfig 函数中传入的参数，就是我们的配置。在这个配置中，主要由编译 Sass 为 CSS 的 Sass 对象和将 CSS 压缩为 cssmin 对象来完成。在这两个对象里，我们还配置了源文件，目标文件等内容。loadNpmTasks 函数则是用于加载对应的 Grunt 插件。registerTask 函数，则是用于配置默认的任务，它依赖于两个子任务 Sass 和 cssmin。

在这个流程中，我们先将 src/styles.scss 文件进行编译，并输出形成 dist/css/styles.css 文件。然后，再将这个文件压缩成 dist/css/styles.min.css。

而 Gulp 写出的构建任务则是以代码为主。下面是一个 gulpfile 的相应任务的示例：

```
//代码位于"chapter04/gulp-grunt-webpack-compare"目录下
var gulp = require('gulp');
var sass = require('gulp-sass');
var minifyCss = require('gulp-minify-css');
var rename = require('gulp-rename');

gulp.task('sass', function(done) {
  gulp.src('./src/styles.sass')
    .pipe(sass())
    .pipe(gulp.dest('./dist/css/'))
    .pipe(minifyCss({
      keepSpecialComments: 0
    }))
    .pipe(rename({ extname: '.min.css' }))
    .pipe(gulp.dest('./dist/css/'))
```

```
        .on('end', done);
    });

    gulp.task('default', ['sass']);
```

Gulp 借鉴了 UNIX 操作系统的管道（Pipe）思想，所以在上面代码中的 Sass 任务里，也是以多个 JSON 为主的链式调用。在任务开始的时候，通过 src 方法配置了输入文件，然后像流水线一样往下执行：

（1）进行 sass()编译。

（2）输出一个目标文件到 dist/css 目录下。

（3）使用 minifyCss 进行代码压缩。

（4）通过 rename 将文件的扩展名取为.min.css。

（5）再输出这个文件到 dist/css 目录下。

（6）结束任务和工作流。

通过这个过程我们可以看到，编程的方式比配置的方式更能提高可管理性。这也是 Gulp 与 Grunt 越来越流行的原因。在早期，选择哪个构建工具，还会有一个决定因素，那就是构建工具中是否有对应的插件。但是，现在这个因素已经越来越不重要了，因为两个构建工具都有我们所需要的大部分插件。从 npmjs.org 的下载情况来看（2019 年 04 月），Gulp 的每周下载量（近 120 万次）接近 Grunt（60 余万次）的两倍。

然而，构建工具的故事到这里并没有结束。随着前端工程的模块化，webpack 也在某种程度上成为一个构建工具。

5. 使用打包工具构建 webpack

在了解 webpack 之前，不得不提到一点：webpack 并不是一个构建任务运行器，但是它提供了灵活的配置，可以进行一些与任务相关的操作，如其官网介绍的那样：

webpack 是一个现代 JavaScript 应用程序的静态模块打包器（Static Module Bundler）。在 webpack 处理应用程序时，它会在内部创建一个依赖图（Dependency Graph），并映射到项目需要的每个模块，然后将所有这些依赖生成到一个或多个 bundle 中。

当前主要的前端框架（Angular、React、Vue 等）都使用 webpack 来打包应用，除了

承担打包的工作，它还负责一部分常见的构建任务。然而对于 webpack 而言，它仍然是一个以配置方式为主的构建过程。下面是同上面的 Gulp 和 Grunt 类似的用于编译 Sass 成 CSS 的步骤：

```
//代码位于"chapter04/gulp-grunt-webpack-compare"目录下
module.exports = {
  entry: ['./src/styles.scss'],
  module: {
    rules: [
      {
        test: /\.scss$/,
        use: [
          {
            loader: 'file-loader',
            options: {name: '[name].css',}
          },
          {loader: "extract-loader"},
          {loader: 'css-loader'},
          {loader: 'postcss-loader'},
          {loader: 'sass-loader'}
        ],
      }
    ]
  }
};
```

上述配置脚本，看上去和 Grunt 更加相似，而未采用 Gulp 式的流编程。因为其主要作用仍然是代码编译，而非构建。对于复杂的构建来说，还是结合 Gulp 更加合理。

对于前端框架来说，他们倾向于在底层屏蔽这些复杂的配置。例如使用 create-react-app 脚本生成的应用，需要执行 npm run eject 才能生成对应的 webpack 配置。

6. 实现构建流

尽管应用的构建相当重要，但是在编写前端应用的时候，我们往往不会自己从头去编写构建代码。第一，它需要花费大量的时间；第二，现成的构建代码可以使用。这些代码及配置的来源，可能是前端框架自带的，也可能是前端应用的脚手架提供的。在需要编写某些功能的时候，还可以直接找到现成的插件来实现。

7. 插件定义构建流

在框架自带构建的情况下，只需要执行 build 命令就可以生成项目所需要的 JavaScript、CSS、HTML 等文件。但是总会有一些意外，有时需要一些额外的静态资源。比如，当我们使用的图标源自图标字体（Icon Fonts）时，每次都需要通过 SVG 重新生成。此时，我们就需要在构建脚本里编写一些相应的代码来实现这个功能。

我们遇到的这些构建问题，多是其他人曾经遇到过的，可以尝试通过以下方式来解决问题：

（1）查找是否有对应的插件。

（2）对比不同的插件区别。

（3）创建插件的适配层。

（4）验证插件是否有效。

（5）重复（2）～（4）步。

（6）如果插件都不适用，那么就编写新插件或改写原有的插件。

针对第 3 章中提到的与模板相应的编译，我们要做的内容有，编译 HTML、JavaScript 和 CSS，对应的 task 就是：gulp.task('default', ['html', 'css', 'js'])。

我们可以在具体的编译任务中处理相应的编译工作，此外，我们还要做的事情有：

（1）确认源文件类型、输出文件内容及格式。

（2）根据源文件输出查找相应的插件。

（3）编写对应的配置或者代码。

一般而言，如果输入文件夹是 src 目录，那么输出文件则是 build 或者 dist。按照这个基础模式，我们只需要找到相应的插件，编写处理的工作流即可。

下面是一个前端项目编译的相关示例：

```
//代码位于"chapter04/gulp-examples"目录下
var gulp = require('gulp');
var pug = require('gulp-pug');
var less = require('gulp-less');
var minifyCSS = require('gulp-csso');
```

```
var concat = require('gulp-concat');
var sourcemaps = require('gulp-sourcemaps');

gulp.task('html', function(){
  return gulp.src('src/templates/*.pug')
    .pipe(pug())
    .pipe(gulp.dest('build/html'))
});

gulp.task('css', function(){
  return gulp.src('src/css/*.less')
    .pipe(less())
    .pipe(minifyCSS())
    .pipe(gulp.dest('build/css'))
});

gulp.task('js', function(){
  return gulp.src('src/javascript/*.js')
    .pipe(sourcemaps.init())
    .pipe(concat('app.min.js'))
    .pipe(sourcemaps.write())
    .pipe(gulp.dest('build/js'))
});

gulp.task('default', [ 'html', 'css', 'js' ]);
```

在这个代码示例中，我们设计了编译 JavaScript、CSS 和 HTML 三个任务，每个任务都依赖于相应的插件来编译代码：

- ◎ 在 HTML 任务里，使用 gulp-pug 插件来编译 pug，模板为 HTML。
- ◎ 在 CSS 任务里，使用 gulp-less 插件来编译 less，代码为 CSS 代码，并使用 gulp-csso 来压缩 CSS 代码。
- ◎ 在 JavaScript 任务里，使用 gulp-concat 插件来合并 JavaScript 代码文件，并使用 'gulp-sourcemaps 插件来输出代码映射的 SourceMap 文件。

构建完成后在 build 目录下会生成相应的配置：

```
├── css
│   └── styles.css
├── html
```

```
|   └── index.html
└── js
    └── app.min.js
```

在这几个不同的任务中，我们可以清晰地看到插件起到了关键性的作用。插件所能做的事情决定了工作流的设计效果。

8. 框架定义构建流

前端应用的构建离不开基本的三要素 HTML、CSS 和 JavaScript。不论我们选用哪一个前端框架，最后输出的内容是一样的。随着前端应用变得越来越复杂，越来越多的前端框架提供了完整的应用脚手架，脚手架中也提供了完整的构建流。对于多数应用来说，使用官方提供的脚手架提供的构建流就可以了，完全不需要自己编写。

让我们来看一个 React 的示例。首先，使用 React 官方的 create-react-app 来创建应用。

对于 npm 5.2+（执行 npm --version 查看）的用户来说，可以直接使用 npx create-react-app react-webpack-demo 来创建应用。

对于 NPM 5.2 以下的用户，则使用以下的命令来创建：

```
npm install -g create-react-app
create-react-app react-webpack-demo
```

安装完成后，执行 npm run eject 或 yarn eject 来生成对应的脚本文件和配置文件：

```
scripts
├── build.js
├── start.js
└── test.js
config
├── env.js
├── jest
|   ├── cssTransform.js
|   └── fileTransform.js
├── paths.js
├── polyfills.js
├── webpack.config.dev.js
├── webpack.config.prod.js
└── webpackDevServer.config.js
```

从代码中我们可以看到，React 是以 webpack 打包为主的构建方式。这里有几个不同的 webpack 配置文件，分别对应不同的使用环境。以 webpack.config.dev.js 为例，下面的代码是其 JavaScript 编译的核心逻辑：

```
...
  entry: [
    require.resolve('./polyfills'),
    require.resolve('react-dev-utils/webpackHotDevClient'),
    paths.appIndexJs,
  ],
  output: {
    pathinfo: true,
    filename: 'static/js/bundle.js',
    chunkFilename: 'static/js/[name].chunk.js',
    publicPath: publicPath,
    devtoolModuleFilenameTemplate: info =>
      path.resolve(info.absoluteResourcePath).replace(/\\/g, '/'),
  },
...
```

在这段代码中，entry 指明了进入编译的文件入口，output 对象则是输出的路径，即相应的配置。

然而，在这个过程中发生了一个巨大的变化——这个脚手架本身提供了构建流，在执行 npm run eject 之前，我们是看不到构建脚本的。当然，我们也不需要看到这些代码，框架本身已经提供了这个功能。只有在应用和构建变得复杂时，我们才需要去了解它是如何构建的，才需要去修改构建脚本。也就是说，框架已经定义了构建流。

4.5　持续交付问题

如果应用构建成功，那么就可以着手部署应用了。大部分前端应用的部署并不复杂，只需要将最后构建生成的目标文件，安装或者复制到指定的线上目录即可——如 HTTP 服务器指向的固定路径。由于其部署简单，有一定 Linux 使用经验的开发者都可以独立完成部署。在笔者经历的多数项目里，尽管身为前端的技术主管，但都不需要半夜去上线，而是交由后端开发人员来完成。

1. 部署方式

对于前端应用来说，其部署方式和后端的区别不大，从方式上可以分为如下三种。

（1）持续部署。构建完成即部署，常见于测试环境。

（2）自动化部署。在持续部署的基础上稍微弱化，即需要人为的介入才能自动化部署。

（3）手动部署。即全程需要人为操作的部署流程。

对于第一种持续部署方式，它不需要有一个上线计划，任何时间都在上线。持续部署更类似于 GitHub Pages 的形式，提交代码后就自动上线了。

在笔者经历的项目里，基本上都具备 Dev 环境的自动化部署，部署的成功也就意味着持续集成是成功的。对于其他测试环境如 ST、UAT 来说，也往往具备这种持续部署的可能性，只是出于流程考虑，通常采用手动触发的自动化部署方式。对于生产环境来说，这种持续部署更加少见。在笔者经历的某个项目里，前端部署采用持续部署的模式，任何修改都可以直接部署到生产环境中。它的实践依赖于成体系的代码质量控制，项目的单元测试覆盖率要在 90%以上，并且拥有大量的自动化 UI 测试。

不论是自动化部署还是手动部署，都需要一个确定的发布策略——不论是敏捷模式还是瀑布模式。敏捷模式的上线发布计划是一个分布迭代（通常两周一次），随后不断地发布；而瀑布模式则是一次上线就完成大部分主要功能。

持续不断地发布应用的优点在于，让团队成员适应快速变化和发布的节奏，并有能力快速解决应用发布过程中出现的问题，而不是在一次上线之后匆忙地应对各种可能发生的情况。对于大型组织下的项目而言，受限于组织内部的上线策略，并不是所有的敏捷项目都可以按迭代上线，但是我们都可以在内部发布，并在过程中持续地改进发布流程。

2. 设计持续集成

在 ThoughtWorks 里完成本地的“hello, world”编写之后，下一步要设计持续集成。这种模式有如下优点：

- 拥有一个随时可发布的应用软件。
- 方便测试人员进行测试。
- 及时地展示项目的代码集成情况。

与之相对应的是：我们要采用第 3 章中提到的何种代码管理方式，越是能持续集成代码，那么后期我们要面对的风险也就越小。在选择代码管理方式时，要选择那种能够持续集成代码的方式，以降低风险。前端项目持续集成的主要内容如下：

◎　对应用进行构建。

◎　进行应用的测试，利用代码来完成测试有助于减少 bug。

◎　部署应用到对应的环境，以提供一个联调和测试环境。

我们需要做的事情有：

（1）找到相应的工具。

（2）进行相应的配置。

（3）运行本章中完成的构建脚本。

（4）编写部署脚本。

相应地，我们需要选择一个合适的持续集成工具——专业的收费工具有 Bamboo；开源的工具有 Jenkins、GoCD；用于开源软件的工具有 Travis CI。对于这一类工具来说，Jenkins 往往是最好的选择，开源、免费又拥有大量的中文文档。在进行配置的时候，我们可以采用流水线即代码（Pipeline as Code）方式进行，将构建流水线的代码编写在代码库中。再次声明一下，我们推荐把一切与代码相关的东西，都放在代码库中管理，如一些与项目相关的文档、架构决策记录，等等。比如 Jenkins 的配置文件，以及 Jenkinsfile 和 Travis CI 的.travis.yml 配置文件，都是用来运行相应的构建的。

如下是一个 Jenkins 的流水线即代码（Pipeline as Code）方式的 Jenkinsfile 配置：

```
pipeline {
  agent any

  triggers {
    pollSCM('* * * * *')
  }

  stages {
    stage('Setup') {
      steps {
        sh 'npm install'
      }
```

```
      }


    stage('Build') {
      steps {
        sh 'npm build'
      }
    }

    stage('Test') {
      steps {
        sh 'npm test'
      }
    }
  }
}
```

注意：在运行上面的测试脚本时，可以使用 Docker 运行一个 Jenkins 应用，在 Docker 官方提供的 Kitematic 中可以找到对应的镜像，再配合代码：chapter04/react-webpack-demo 即可使用。

该配置与构建脚本类似，定义了持续集成所要做的基本事项：安装依赖、执行构建、运行测试，它们分别对应不同的步骤，也对应于前端的构建任务中的几个核心任务。只是现在执行者变了，由开发者变为持续集成服务器，由手动变为全自动执行。

与构建脚本保持基本一致，可以上前端构建与持续集成平台进行解耦，以便向不同的平台迁移。关于持续集成的好处，这里不再赘述。如果读者还没有使用过持续集成，那么可以从现在开始构建一个更好的开发体验。事实上，在完成上面的步骤后，我们还需要编写对应的部署脚本。

3. 自动化部署

相比于后端，自动化部署对于前端来说是一件特别容易的事。我们通过 npm run build 就可以构建出一个前端项目：

```
drwxr-xr-x  5 phodal staff  160 Aug 22 23:06 assets/
-rw-r--r--  1 phodal staff 5.4K Aug 22 23:06 favicon.ico
-rw-r--r--  1 phodal staff  736 Aug 22 23:06 index.html
-rw-r--r--  1 phodal staff  16K Aug 22 23:06 main.js
-rw-r--r--  1 phodal staff 222K Aug 22 23:06 polyfills.js
```

```
-rw-r--r--  1 phodal staff 5.1K Aug 22 23:06 runtime.js
-rw-r--r--  1 phodal staff  16K Aug 22 23:06 styles.js
-rw-r--r--  1 phodal staff 6.7M Aug 22 23:06 vendor.js
```

首先，将这些静态文件部署到服务器上，使用 Nginx 配置域名及相应后台服务的地址即可，代码如下：

```
server {
    listen 8080;
    server_name https://aofe.phodal.com;
    root /www/home/aofe;
    index index.html index.htm;

    location / {
        try_files $uri $uri/ /index.html;
    }
}
```

然后，搭配 Docker，编写 Dockerfile 就能完成快速的应用部署。

再编写一个对应的 NPM 脚本，配置到 Jenkinsfile 中，就完成了整个持续部署的流程。

使用 Puppet、Chef 这样的工具也可以直接在本地运行脚本，将应用部署到线上环境。然而，事情往往并不是这么简单的，只有当我们拥有服务器的直接操作权时，才可以轻松地运行我们所需要的服务。

尽管前端部署很简单，还要适当地考虑“回滚机制、蓝绿部署、灰度发布”等因素。并在应用部署后，使用 UI 自动化测试来测试部分功能是否正常，以及测试对应的后端服务连接。这里不推荐人工测试，考虑到前端的部署比较简单，就不详细地进行说明了。

4. 环境配置

不同的环境拥有不同的功能、代码及配置事项。当我们编写构建脚本的时候，还需要针对不同的环境（开发、线上）编写构建脚本。

5. 开发环境配置

对于开发配置而言，有如下不同环境的考虑因素：

（1）在本地环境开发时，我们需要将 HTTP 请求指向本地的 Mock Server。

（2）在本地集成后端服务时，需要将 HTTP 请求直接指向开发环境的后台服务。

（3）针对不同环境的构建配置来配置不同的构建脚本，以用于部署或者调试。

（4）在运行时，针对不同环境的产品进行相关服务的配置。

我们需要考虑如下配置：

构建配置。在前面的 webpack 部分，我们介绍了针对不同环境的构建脚本。这个功能是构建工具本身所支持的，不需要我们操心——当然，通过一些额外的配置和参数，也可以实现同样的功能。因此，只需要在编写构建脚本时，将这些因素考虑进去。

代码配置。多数前端框架都提供了环境变量的功能，如 Angular 可以通过不同的 environment.*.ts 文件来区分不同的环境。代码中的配置主要用于运行时的环境配置，比如用于不同环境（开发、测试、线上）的第三方服务配置，以及广告、第三方登录授权服务配置等。这些配置可能是硬编码在项目中的，也有可能是动态地从后台获取的。两种方式都有各自的优缺点，使用哪种方式要看需求、配置的变更程度，以及部署是否便利。

代理配置。对于遵循流程的 ST、UAT 环境来说，由于其可能拥有半年前、一年前的线上数据，因此这些环境和线上的环境更加相似。在这个环境里更容易发现一些 Bug，这时需要将 HTTP 请求指向这些不同环境的服务器。因此，我们几乎需要对所有的环境编写相应的配置。

下面是一个 Angular 应用针对不同环境的配置示例：

```
    {
      "name": "aofe.code",
      "version": "0.0.0",
      "license": "MIT",
      "scripts": {
        "ng": "ng",
        "start": "ng serve --open -e dev --proxy-config config/proxy.conf.json",
        "start:dev": "ng serve --open -e dev --proxy-config config/proxy.conf.
dev.json",
        "start:st": "ng serve --open -e st --proxy-config config/proxy.conf.
st.json",
        "start:uat": "ng serve --open -e uat --proxy-config config/proxy.conf.
uat.json",
        "build": "ng build --prod"
      }
    }
```

此外，我们还有一些线上运行时与调试相关的配置。

6. 线上调试

对于项目开发而言，我们经常要写一些调试代码。为了捕获难以察觉的 bug，有时需要在对应的环境上运行代码，有时则需要在线上的环境中运行代码。

为了在线上运行和调试代码，通常需要有一些额外的“开关”（toggle），如：

◎　在 URL 中添加一些参数，前端代码在运行的过程中去读取这个参数。

◎　在 LocalStorage 中根据某些值是否存在来运行和调试代码。

◎　对特定的账号进行权限处理，以获取调试功能。

具体采用哪种方式，取决于不同项目的需要。对于一些特殊的应用来说，我们还可能需要通过输入参数来获取后台返回的相应的 Debug 用的 JavaScript 代码。

有意思的是，另外一个重要的开关功能：功能开关（Feature Toggles），也使用了类似的注入变量的调试方式。

功能开关是一种强大的技术，允许团队在不更改代码的情况下修改系统行为。比如，我们想在月底的上线中添加一个重要的 A 功能。但是 A 功能不太稳定，我们想在发布的过程中控制 A 功能的使用。这个时候，我们就需要使用功能开关来控制，当我们打开 A 功能开关的时候，就运行带 A 功能的部分的代码；当我们关闭 A 功能开关的时候，就运行原始的代码。在代码中，我们通过某一个变量来控制，而变量的管理则可以通过上述的方式来实现。

4.6　小结

在本章中，我们介绍了通用的前端应用构建流程，以及在其背后的依赖管理、包管理和发布机制。在了解基本的构建之后，我们开始了解隐藏在浏览器端的三元素 HTML、JavaScript、CSS 的编译过程。通过这些基本的介绍，我们对前端应用的构建流有了一个基本的认识。

接着，我们展开了对前端应用的构建流设计。先介绍了一些常用的构建工具：Gulp、Grunt 及 webpack，同时还有其背后的一些基本思想——基于配置的构建、基于代码和流的构建方式。并分别使用这两种思想，来完成一个前端应用的构建流设计。通过这些实例，我们可以更容易地探索现有应用的构建流程。

最后，我们结合了配置构建和持续集成，对前端应用的持续部署进行了简要的介绍。此外，还关注了隐藏在构建背后的环境配置，以及一些与配置相关的线上调试设置。

现在，我们对应用之外的基础流程和框架有了更详细的了解。尽管这些内容比较基础，但是它们背后的思想相当重要。在我们的工作中，只是不断地换着使用工具而已。

5

第 5 章

架构设计：多页面应用

单页面应用，是指只有单个 Web 页面的应用，它通过动态重写当前页面来与用户进行交互，而不是通过远程服务器来加载新的页面。对于那些从远程服务器加载新页面的应用来说，由于它存在多个页面，所以为了方便区分，我们将其称为多页面应用。

在单页面应用中，通常会在第一次加载时，便加载全部的静态资源（HTML、CSS、JavaScript、图片等），而多页面应用则是在每次加载新页面时会重新加载（受 HTTP 缓存的影响，大部分资源不会重新加载）。单页面应用的这个特性，使它可以适应于 Web 应用，还可以用于移动应用（混合应用）和桌面应用（基于 Electron、NW.js 等）。

尽管单页面应用很流行，但是我们并不总是需要它。在学习编程的初级阶段里，经常会掉入一个陷阱：一旦学会一个新的框架，就会在我们的大部分项目中使用这个框架。即所谓的“锤子定律”：

我认为，假设你所拥有的工具只有一个锤子时，你把所有的事物当作钉子来对待是很有吸引力的。

但是事实上，并不是所有的前端应用都是单页面应用，都需要单页面应用的框架。这些应用可以是传统的多页面应用。而作为一个专业的前端开发人员来说，非单页面的应用也是我们要考虑的范围。

5.1 为什么不需要单页面应用

人与技术之间的关系是很微妙的，我们喜欢某项技术，就会擅长某项技术；我们擅长某项技术，可能就会喜欢某项技术；我们习惯了某项技术，也就可能不愿考虑其他技术。当我们习惯了单页面应用，就有种种理由讨厌多页面应用；当我们习惯了多页面应用，就有可能想拒绝单页面应用。

于是乎，当我们作为一个多页面应用的开发者，面对提出使用单页面应用建议的人，就会提出一系列客观或不客观的理由。而如果单纯从技术的角度来考虑这个问题，那么我们会得到这样一些结论：

- 构建单页面应用的成本很高。
- 前端生态快速发展，学习成本高。
- 单页面应用对 SEO（搜索引擎优化）不友好。
- 应用架构更加复杂。

这些都是从技术的复杂度和实施成本来考虑的。有意思的是，从非技术的角度来讲，比如 KPI、技术远景，使用单页面应用可能才是合适的。最合适的框架并不意味着仅仅是对这个项目合适，还可能是对这个项目的开发人员合适。

5.1.1 构建成本

在第 4 章中，我们花费了一个章节讨论构建系统。一个不使用单页面应用架构的原因也在于此，一个现代的前端单页面应用构建相当复杂。

当我们的应用是一个报名表单时，我们需要如下步骤：

（1）编写页面样式，以提供更好的页面效果。

（2）动态获取数据，以渲染下拉菜单供用户选择。

（3）在用户提交前，实施对应的表单验证，以提供更好的用户体验。

因此，在使用传统的前端技术时，我们只需要找到一个表单插件，添加到代码中，编写一些相应的代码就可以了。相应的一些模板的生成，都是由后端提供的。

当使用单页面应用的时候，我们要考虑使用哪个框架，使用什么语言和技术栈，还需要考虑是否分离前后端。而不论是否分离前后端，我们的应用构建都会变得相当复杂。除了需要在本地处理构建流程，还需要在一台公用的服务器上运行构建（如持续集成服务器），以提供最后用于发布的软件包。一旦框架本身提供的构建流程不符合我们的需求时，我们就不得不花费大量的精力在重写构建脚本上。

对于不了解前端的程序员来说，还需要实践第 4 章的内容，才能厘清这一步所需的成本。

5.1.2　学习成本

前端在过去的几年里经历了快速的发展、变化和更新。由于技术变革过快，所以今年使用某版本框架编写的应用，在明年可能就不再被支持。这个时候，回过头去看旧的应用，就变得非常陌生。我们不仅要学习新的框架，又要学习旧的框架。

以前端的数据获取为例，在过去的几年里发生了一些显著的变化，并且在不同的框架中使用方式还略有区别。在早期，我们使用 jQuery 的 Ajax 函数来获取远程数据：

```
$.ajax({
  url: "/api/article/comments/html", success: function (content) {
    $("#comments").html(content);
  }
});
```

其背后是对 XMLHttpRequest 的封装。由于 XMLHttpRequest（XHR）的设计混乱——输入、输出和使用事件来跟踪的状态混杂在一个对象里。于是，在一些浏览器中开始提供新的 Fetch API 来替换旧的 XMLHttpRequest API，示例代码如下：

```
fetch("/api/article/comments").then(function (res) {
  if (res.ok) {
    res.json().then(function (data) {
      that.contents = data.contents;
    });
  }
});
```

在不同的框架中，推荐使用不同的 API 数据获取库，其对于异常的处理还是不一样的。下面是 Vue.js 框架推荐的 Axios 的示例：

```
axios
  .get('/api/article/comments')
  .then(response => (this.contents = response.contents))
  .catch(error => console.log(error))
```

对比之后，会发现在上面的 Fetch 示例中，这里的异常需要额外的代码去处理。而在使用 RX.js 来处理数据的 Angular 框架中，则又有所不同：

```
this.http.get<CommentsModel>('/api/article/comments')
  .subscribe(
    data => this.contents = data,
    err => console.log(err)
  );
```

与它前面的示例相比，Angular 框架中多做了一层数据模型的验证。这只是框架进化中的一个例子。此外，还有不同的模板引擎设计、不同的路由规则的变化，以及不同的组件化方案。

当未来的某一天，又出现一个新的框架，它能解决旧的系统中的一些问题时——这些遗留问题持续地影响了我们的开发进度，如果能替换它们有助于效率的提升。我们写了一个又一个 demo，又对一系列的指标进行测试，发现它能满足需求。于是计划使用这个框架，在切换框架的过程中遇到的第一个问题是框架的学习成本。特别是当我们使用“大而全”或“小而美”的框架时，学习成本很高。

因此，我们不得不去衡量框架切换的成本。

5.1.3 后台渲染成本

一个网站的流量来源有多种方式：直接访问、外部链接、用户搜索、社区分享等。不

同网站的主要流量来源各有不同，也各有区别，他们对流量的策略也各有不同。其中有一类是比较“轻松”地获取流量的方式，即用户搜索。网站的访客通过在搜索引擎上搜索，看到我们的网站提供了他们想要的内容，便访问了我们的网站，而我们所做的只是向搜索引擎提供一份可以供搜索引擎爬虫抓取的内容。

早期，搜索引擎爬虫只支持从 HTML 中解析出内容，再处理解析的结果，并以某种形式展示到搜索的结果页。而随着技术的发展，有些搜索引擎爬虫可以支持抓取 JavaScript 动态渲染的内容。

对于面向 Google 的网站而言，它可以提供更好的 JavaScript 渲染支持，可以直接抓取 JavaScript 渲染的应用。然而，对于国内的开发者而言，多数时候我们要面对百度、搜狗等搜索引擎，它们对 JavaScript 渲染页面的支持并不是很好。如果网站的主要流量来源是搜索引擎，那么我们就不可能放弃这些“免费”的流量，我们的应用就需要支持爬虫，此时我们就要尽可能地提供一份由 HTML 构成的内容。

如果是单页面应用，就要考虑使用额外的方式来支持：

◎　预渲染，即面向搜索引擎提供一份可以被索引的 HTML 代码。

◎　同构应用，由后端运行 JavaScript 代码生成对应的 HTML 代码。

◎　混合式后台渲染，由后端解析前端模板，生成对应的 HTML 代码。

如果交互不复杂，那么反而不如多页面应用来得简单——直接进行后台渲染提供一份 HTML。因此，如果交互少，那么使用多页面应用就显得更加合适。

5.1.4　应用架构的复杂性

单页面应用不仅带来了前端开发的变化，还带来了后端开发的变化。

架构方式变成了前后端分离架构。原先，业务逻辑的处理基本都交由后端来完成，而现在数据处理的逻辑则部分交给前端，尤其是数据的展示部分。为此，后端需要提供一个又一个 RESTful API，将控制权交由前端进行交互，后端则将数据的展示部分解耦出来。这也意味着，后台的数据通过 API 直接暴露在所有攻击者面前。在设计的时候，我们不得不更加仔细地考量某些字段要怎么展示，是否采用二次请求的方式来完成等。

复杂的应用，则会进一步孵化出 BFF（服务于前端的后端）层，在这一层针对不同的客户端提供不同的 API。这样的中间层，可以明确职责，但也让架构变得复杂。

为了提供更好的用户体验，一些代码逻辑会重复地在前后端出现，比如表单的校验。对于一个长的或者复杂的表单，在用户填写的过程中，应该实时提示用户哪些字段出错。而不是在用户提交表单后才告诉用户哪里有错。如果二次提交的表单仍有错，又需要再次修改，如此反复会给用户带来极差的体验。逻辑的重复实现也就意味着，当一种业务发生变更时，需要同时修改多处。

除了表单一类，还需要处理权限的管理。通常情况下，单页面应用一次性提供给用户所有的代码，通过对后台 API 的请求来获取数据。由于代码已经运行在客户端上了，我们需要控制用户能访问哪些页面。当用户没有权限时，不仅要从后台对用户权限进行校验，还要从前端控制用户的页面访问。用户只应该看到他们能访问的功能，对其他功能要加以限制。因此，在一些复杂的应用中，我们会根据不同的角色，编写出不同的前端应用代码。

最后，为了提供一个可供前端测试的 API，我们还需要编写一个 Mock Server 来提供一个临时的 API。这部分内容，我们将在后面的章节中介绍。

因此，让我们抛开单页面应用的一些想法，来思考多页面应用需要怎么做。

5.2 简单多页面应用的开发

多页面应用往往是一些轻量级的前端应用，多数时候会被划定为“前端页面”——往往只需要编写页面样式，即编写一些 JavaScript 就能完成工作了。

常见的这一类应用有：

- ◎ 门户类、资讯类、数据展示类应用，比如博客、CMS 系统，等等。
- ◎ 弱交互型应用，比如 GitHub、Gitlab 这样的代码管理站点，就不需要太多交互。
- ◎ 面向资源受限的设备应用，嵌入式系统由于资源限制，往往运行的不是完整的 JavaScript 引擎，会导致其难以支持完整的单页面应用。

对于这一类型的应用来说，寻找一个合适的 UI 组件库，外加一些用于交互的框架或者库，就可以完成大部分功能。偶尔存在复杂的交互，也可以通过编写一些 JavaScript 代码来完成。

5.2.1　选择 UI 库及框架

在多页面的时代，由于前端应用的简单化，几乎大部分前端团队都拥有自己的前端框架和 UI 库。因为在这个时期，自己编写、维护一个前端框架和 UI 库是一件容易的事。

随着时间的推移及单页面应用的流行，现在组织内部维护的框架也逐渐被社区的框架所取代。内部的框架受限于对不同部门的支持，整体的技术发展速度往往不如外部框架。

因此，在未来的某个时期往往容易被抛弃。复杂的单页面应用几乎都采用了主流的前端框架——对于一些大型机构而言，仍然会存在自己的单页面框架，它们有足够的能力去维护这样一个框架。

于是，我们需要从外部选择一个合适的框架。

5.2.2　jQuery 和 Bootstrap 仍然好用

jQuery 是一个起源于 2006 年的开源前端库，它大大地简化了 HTML 与 JavaScript 的操作。由于天性自由，笔者曾在自己的某个开源项目里尝试过不使用 jQuery，但是后来发现使用原生 JavaScript 来处理不同浏览器的 DOM，是一件痛苦的事情。因此 jQuery 仍然是目前最流行的前端框架，这与大量旧网站的存在不无关系，也从某种程度上反映了大量的网站仍是多页面应用。

Bootstrap 作为流行的响应式 UI 框架，其布局可以根据显示网页的设备（桌面、平板电脑、手机）来进行动态调整。由于其响应式的处理能力，使得开发人员可以更关注于实现，而不是 CSS 样式，也因此成了一些开源 CMS、框架所提供的默认 UI 框架。

如果你想快速地使用多页面技术来开发应用，那么使用 jQuery 的生态能感受到显著的开发优势，再将 Bootstrap 作为 UI 框架能提升开发体验。尽管这样做会成为一个“普通”的方案，即没有亮点，看上去和其他应用类似，但是在后期我们可以快速、方便地进行定制。

在这里不得不提及一个我曾经遇到的很有意思的问题。

◎　开发人员：“我们想开发一个多页面应用，但是不想使用 jQuery，因为它看上去很 low，有没有别的选择？”

◎ 我：“那想必只有 Zepto。”（Zepto 是一个类 jQuery 的框架，大部分的 API 和 jQuery 兼容）。

我们选择技术栈是因为它能解决问题。如果使用一个新的技术要花费大量的时间，既不能让自己高高兴兴地解决问题，又不能带来一些利益，如 KPI。那么，为什么不节省时间，去做一些更有意义的事情？比如自己去造一个类似 React 的“轮子”？

5.2.3 不使用框架：You Don’t Need xxx

标题来源于 GitHub 上受欢迎的各种 You Don’t Need 系列的开源项目。对于那些只是简单地显示、隐藏 DOM 等操作的应用来说，我们完全不需要任何框架。

在新的 HTML 标准形成后，如果我们只是做简单的元素选择，可以使用原生的 querySelectorAll，再配置一个 pollyfill。在一些面向程序员的网站，如在 GitHub 上已经使用浏览器原生的 DOM 操作元素，来替换诸如 jQuery 这样的框架。如基本的 class 查询，在 jQuery 中使用的是$('.class');，而使用原生的代码则是：

```
document.querySelectorAll('.class');
// 或者
document.getElementsByClassName('class');
```

两种方式并没有太大的区别，只是使用原生的代码会相对较长。然而，我们可以将其封装成一个代理方法，以便未来替换。在这种对比之下，使用原生的代码会更加灵活，并且更容易在组织内部积累经验。尽管在一些开源库中能够找到替代方案，但是自己去造一个“轮子”也是一个不错的方式。

同样，这种替换方式也适用于 lodash、moment.js 等的实践，当我们不再需要支持低版本的 IE 浏览器时，就可以使用一些高版本浏览器自带的 JavaScript 语言特性，比如 users.find(function (o) { return o.age < 40; })，它可以用来替换 lodash 的_.find(users, function (o) { return o.age < 40; })。

对于简单的应用来说，上述工具和方式能解决我们的问题。但是对于更复杂的页面应用来说，我们则需要一些更好的工具。

5.3　复杂多页面应用的开发

应用往往不会那么理想化，让我们想用单页面应用技术，就使用单页面应用技术。单页面应用与多页面应用存在一些交集。这些交集，往往会影响我们的技术决策。使用多页面应用过于简单，而使用单页面应用过于复杂。我们要想办法在中间状态做一个最好的选择，即自己选择和创建合适的框架来完成功能。

我们选择 MVC 框架，多数是因为我们需要其中的一些特性，比如：

◎　模板引擎，动态生成、创建页面。

◎　双向绑定，实时修改数据。

◎　前端路由，路由变化映射到对应的逻辑上。

……

正是这些特性组成了一个前端框架，它也是处在单页面应用和多页面应用交集的一些技术。这时除了选用框架，我们还可以自制一些简单的框架来完成相应的开发工作。考虑到市面上已经有几本关于创建 MVC 框架的书籍（比如《JavaScript 框架设计》），并且本书并不是主要关注于此，所以接下来，我们将简单介绍其中的一些基本原理。

5.3.1　模板与模板引擎原理

对于稍微复杂的多页面应用，我们常常会采用模板来动态生成一些 HTML，比如弹窗（Popup）、对话框（Dialog），或者一些固定的业务模板。由于重复编写这些内容的 HTML 必然不是一个好的方式，所以简单的 HTML 可以通过字符串拼接或者模板字符串的方式来生成，下面是开源前端框架 artDialog 中的前端 HTML 拼接示例：

```
html +=
  '<button type="button"' +
  ' i-id="' + id + '"' + style +
  (val.disabled ? ' disabled' : '') +
  (val.autofocus ? ' autofocus class="ui-dialog-autofocus"' : '') + '>' +
  val.value + '</button>';
```

但是要生成复杂的 HTML，则需要使用模板引擎来实现。模板引擎，又可以称为模板处理器，它是一个结合模板和数据模型来输出最终文件的处理程序。在前端领域里，这种文档是 HTML。一般情况下，我们会将前端模板引擎分为两种，基于字符串的模板引擎和基于 JavaScript 的模板引擎。两种不同类型的模板引擎，各有各的优势。

5.3.2 基于字符串的模板引擎设计

顾名思义，基于字符串的模板引擎，就是通过字符串替换的方式，来渲染出 HTML，再将 HTML 插入 DOM 节点中，其代表性框架有 Mustache 和 Handlebars.js。

从软件架构上来看，Mustache 提供了多语言的版本，即官方提供了诸如 Python、Java、Ruby 等语言的模板引擎。因为前后台共用这些模板，它可以使得后台渲染变得相当容易。只需要读取对应的模板文件，就可以将 Model 填充到数据中。使用这种方式时，每次更新值都需要全节点替换 DOM 节点，这样会造成一定的性能问题。

其基本原理是，全局正则匹配模板关键字，如{% name %}，再从传入的 data 中找到是否有对应的值，如 data 中存在 name 的对象，则使用该值进行替换，否则该值为空。

```
//代码位于 chapter05/template-engine 目录下
function simple(template, data) {
  return template.replace(/\{\{([\w\.]*)\}\}/g, function (str, key) {
    var keys = key.split(".");
    var value = data[keys.shift()];
    for (var i = 0; i < keys.length; i++) {
      value = value[keys[i]];
    }
    return (typeof value !== "undefined" && value !== null) ? value : "";
  });
}
```

这是一个简易的模板引擎示例，在这个示例中，我们输入 data 数据，并通过正则表达式/\{\{([\w\.]*)\}\}/g 将模板中对应的键（代码中的 key）替换为相应 data 中的值。为了做示范，这里只处理一种模板规则，下面是基于这个模板引擎编写的模板代码：

```
<p>用户：{{user.name}}，地址：{{info.address.city}} </p>
再结合 JavaScript 代码及数据模型，就可以将其渲染成 HTML：
var data = {user: {name: 'Phodal'}, info: {address: {city: '漳州'}}};
var temp = simple("<p>用户：{{user.name}}，地址：{{info.address.city}} </p>", data);
document.getElementById("result").innerHTML = temp;
```

首先，我们创建一个 data 变量，将示例数据放入变量中；然后，调用 simple 方法来获得填充完数据的 HTML；最后，找到对应的 DOM 元素，替换其中的 HTML。

这里只是一个简单的字符串模板引擎示例，读者如果对编写模板引擎有兴趣，可以从本书第 5 章代码的 template-engine 目录的 README.md 下查看与之相关的索引，或者从 GitHub 上找到相应的示例。

字符串模板引擎除了用在多页面应用的动态更新上，还可以用于静态站点生成器，如用于 GitHub Pages 的 Jekyll、Hexo 框架，以及更通用的 Assemble 框架。它们的原理都是类似的，从某个地方获取数据（如数据库），再通过数据 + 模板来渲染 HTML。

基于字符串的模板引擎在更新 DOM 的时候会更新所有 DOM 节点，这时浏览器需要重新渲染所有的节点。当我们拥有大量的 HTML 元素或 DOM 操作时，如果每次都刷新整个 DOM 节点，显然是不合适的。这时，我们就需要更高级一点的方案。

5.3.3　基于 JavaScript 的模板引擎设计

基于 JavaScript 的模板引擎的表现形式是，将模板转义为 JavaScript，在执行的过程中，再动态更新所需要的 DOM 节点。相应的逻辑如下：

（1）将模板编译为某种 DSL（领域特别语言），比如 HyperScript 或者 JavaScript 对象（代码 + 数据）。

（2）在使用时，调用 JavaScript 来渲染出 DOM 结点。

（3）（可选）当发生变更时，通过 DOM Diff 算法来替换对应的修改结点。

它和基于字符串的模板引擎的区别主要在于第三点，是否全局替换 DOM 节点。比如 Virtual DOM，就是带一个 Dom Diff 的 JavaScript 模板引擎，在进行大量 DOM 操作的时候，由于只对变化的 DOM 进行替换，可以提高前端应用的性能。

普通的 JavaScript 模板引擎，形如我们在第 4 章中提供的 Angular 模板编译后的示例：

```
    (t()(),Ir(15,0,null,2,2,"a",[["href","tel:987654321"],["mat-raised-butto
n",""]],[[1,"tabindex",0],[1,"disabled",0],[1,"aria-disabled",0],[2,"_mat-an
imation-noopable",null]],[[null,"click"]],function(t,e,n){var  i=!0;return
"click"===e&&(i=!1!==io(t,16)._haltDisabledEvents(n)&&i),i},Ly,Fy))
```

但是 Angular 的例子过于复杂，且它并不拥有 DOM Diff 功能。这里我们将基于开源的 Virtual DOM 实现（https://github.com/Matt-Esch/virtual-dom）来介绍一下。基于 Virtual DOM 技术的 JavaScript 模板引擎的基本逻辑如下：

- ◎ 使用一种名为 HyperScript 的 DSL（领域特定语言）来创建虚拟的树。
- ◎ 通过这个虚拟树来创建一个 DOM 节点。
- ◎ 在数据发生变化的时候，diff DOM 节点也会发生变化，并通过更新对应修改的 DOM 来更新模板。

HyperScript 是一个用于创建带 HTML 结构的 script 脚本的工具，可以用于渲染出一个虚拟的 DOM 树。该 Virtual-DOM 项目的 DSL 的写法示例为：h('a', {href: 'https://aofe.phodal.com/'}, 'aofe')，这里的 h 是 hyperscript 的缩写的方法名，后面的三个参数分别对应了 a 标签的几个属性。

如上所述，这是一个虚拟的 DOM 树，我们还需要解析这个 DOM 树，把它渲染成真正的 DOM，并将其添加到页面上：

```
//代码在"chapter05/js-template"目录下。
var h = require('virtual-dom/h');
var createElement = require('virtual-dom/create-element');

var result = h('a', {href: 'https://aofe.phodal.com/'}, 'aofe');

//渲染出 DOM 节点
var node = createElement(result);
document.body.appendChild(node);
```

由于 Virtual-DOM 是采用 Common.js 的模块化方案编写的，我们需要借助于一些工具，比如 browerify，来将其编译为可以在浏览器上运行的代码。先通过 npm install 安装 browserify，再执行相应的编译命令，就可以将生成的脚本直接在浏览器上运行：

```
browserify vdom-sample.js -o bundle.js
```

如果使用原生的 DOM 操作，写出来的代码便是：

```
var a = document.createElement('a');
a.href = "https://aofe.phodal.com/"
a.text = "aofe"
document.body.appendChild(a);
```

在没有修改 DOM 的时候，两者的区别并不是太大。而在进行 DOM 修改之后，我们的代码变成了：

```
var h = require('virtual-dom/h');
var diff = require('virtual-dom/diff');
var patch = require('virtual-dom/patch');
var createElement = require('virtual-dom/create-element');

//1:创建一个函数来声明 DOM 对应的属性
function render(count)  {
  return h('a', {href: 'https://aofe.phodal.com/'}, count);
}

//2:初始化 document
var count = 0;                              //创建应用 data，比如用于存储的 count

var tree = render(count);                   //初始化树
var rootNode = createElement(tree);         //创建一个初始化的 DOM 节点
document.body.appendChild(rootNode);        //将 DOM 节点添加到 document

// 3: 整合起来来更新逻辑
setInterval(function () {
  count++;

  var newTree = render(count);
  var patches = diff(tree, newTree);
  rootNode = patch(rootNode, patches);
  tree = newTree;
}, 1000);
```

我们将上述代码分为三步：第一步，将 a 标签的文本变成了 count 变量；第二步，将 count 变量赋值为 0，并初始化 DOM 节点，添加到页面中；第三步，生成 count 值变化后的虚拟 DOM 树，并通过 diff 方法来对比新旧虚拟 DOM 树，再通过 patch 方法将变化后的 DOM 一一修改到原始的 rootNode 节点上。

我们可以发现整个过程依赖于虚拟的 DOM 树，它实际上相当于一个 DSL（领域特定语言），提供了一个模板与真实的 DOM 节点的代理，所有相关的修改在这层 DSL 数据上进行优化，以此来应对复杂的 DOM 操作的性能问题。这里的 hyperscript 并不是一个容易编写的模板。

在使用 Virtual DOM 的前端框架时，通常使用类 XML（HTML 也是类 XML）的形式来编写模板，比如 React 的模板代码如下：

```
class ShoppingList extends React.Component {
  render() {
    return (
      <div className="shopping-list">
        <h1>Shopping List for {this.props.name}</h1>
        <ul>
          <li>Instagram</li>
          <li>WhatsApp</li>
        </ul>
      </div>
    );
  }
}
```

再将这个模板转换为中间 DSL 状态：

```
return React.createElement('div', {className: 'shopping-list'},
  React.createElement('h1', /* ... h1 children ... */),
  React.createElement('ul', /* ... ul children ... */)
);
```

这样我们编写的时候就不会发生太多变化，使用的仍然是类 XML 的形式。在编译的时候将其转换为 DSL，从而在复杂的 DOM 操作里实现一定程度上提高性能的目的。

不过，在多页面应用中使用基于 JavaScript 模板引擎的情况并不常见。对于一些复杂的、拥有大量的 DOM 操作的组件，如表格组件，则更需要这种类 XML 的组件，以提高性能。

5.3.4 双向绑定原理及实践

双向绑定，即双向数据绑定，它是指视图（View）的变化能实时地让数据模型（Model）发生变化，而数据的变化也能实时更新到视图层。在进一步深入双向绑定之前，应当对单向数据绑定的定义有一定的了解。

从双向绑定的定义里，我们可以清晰地将单身绑定分为两类。

第一类，视图的变化能实时地让数据模型变化。常见的有用于用户输入数据的输入框

（input 标签），在输入模型中输入数据时，我们需要监听这些数据的变化，再将这些值传递到模型中进行处理：

```
//代码位于 chapter05/vanilla-binding 目录下
var inputData = '';
var inputText = document.querySelector('#inputText');
inputText.addEventListener('input', function(event){
  inputData = event.target.value;
});
```

在上面的代码中，我们监听了一个 ID 为 inputText 的输入框，当有用户输入时，更新数据。

第二类，当数据发生变化时，对应的视图上的值也发生了变化，此时需要更新视图的值，一个简单的示例代码如下：

```
//代码位于 chapter05/vanilla-binding 目录下
var button = document.querySelector('#change-button');
button.addEventListener('click', function (event) {
  inputText.value = 'change';
});
```

从上面代码中可以看到，数值更新是通过更新 inputData 的值触发输入框的值的更新，这种实现方式需要做一个双向绑定。

如果多页面应用只有一两个双向绑定的场合，倒也还好。重复编写这样的代码，问题并不会很突出。但是，当我们的页面是一个表单时，就会麻烦一些了。倘若不使用相关的表单插件，长表单算是最难对付的场景。为了通过实时验证来提高用户体验，用户输入值的同时，需要再进行一定的正则校验，以验证输入项是否符合我们预期的要求。

解决问题的思路和模板引擎相差无几，要么找一个合适的框架来实现，要么自己实现一个简单的双向绑定。关于相关的框架，读者可以在本书的源码（相应章节的 README.md）中找到相应的资源。双向绑定有几种不同的实现方式：

- ◎ 手动绑定。即两个单身绑定的结合，通过手动 set 和 get 数据来触发 UI 或数据变化。
- ◎ 脏检查机制。即在发生指定的事件（如 HTTP 请求、DOM 事件）时，遍历数据相应的元素，然后进行数据比较，对变化的数据进行操作。

◎ 数据劫持。即通过 hack 的方式（Object.defineProperty()）对数据的 setter 和 getter 进行劫持。在数据变化时，通知相应的数据订阅者，以触发相应的监听回调。

考虑到数据劫持可以独立于框架和事件运行，并且代码量比较小，这里便以其为例，来介绍相关的绑定流程。

依照 MDN 的解释，Object.defineProperty 方法会直接在一个对象上定义一个新属性，或者修改一个对象的现有属性，并返回这个对象。我们所要做的是，为一个作用域中的对象进行 setter 和 getter 的劫持。这里的 setter 指的是，当我们改变一个属性的值时，它会执行一些对应的 setter，如我们执行 o.b = 38 时，会触发其对应的 setter 方法。而 getter，则是我们获取的这个属性的值。

下面是按照该原理进行的数据劫持示例：

```
var o ={}; var bValue;
Object.defineProperty(o, "b", {
  get : function(){
    return bValue;
  },
  set : function(newValue){
    bValue = newValue;
  },
  enumerable : true,
  configurable : true
});

o.b = 38;
```

这段代码，创建了一个 bValue 变量，并为 o 对象的 b 属性重新定义了 getter 和 setter。当我们为 o.b 对象设置值时（对应于代码中的 o.b=38），我们将调用 set 方法，将新的值赋予 bValue 变量；当获取 o.b 对象的值时，它将返回 bValue 的值。

于是，这时 bValue 的值就是 38，如果我们修改 bValue 的值，对应的 o.b 的值就会随着变化。如果将代码中的 set 函数的赋值语句去掉，那么即使执行了 o.b = 38，o.b 的值将始终是 undefined，因为 set 函数的操作被清空了。代码中的 configurable 特性表示对象的属性是否可以被删除，以及除 writable 特性外的其他特性是否可以被修改。

有了这个操作，我们就可以在更新数据模型的时候关联到相应的 UI 修改逻辑。这个示例的 UI 是一个输入框：

```
<input id="inputText" data-tw-bind="name" type="text">
```

下面是和该输入框相关的双向绑定的部分代码（完整的代码位于 chapter05/binding-simple）：

```
var scope = {};
var elements = document.querySelectorAll('[data-tw-bind]');
...
var value;
Object.defineProperty(scope, prop, {
  set: function (newValue) {
    value = newValue;
    elements.forEach(function (element) {
      if (element.getAttribute('data-tw-bind') === prop) {
        if (element.type && (element.type === 'text')) {
          element.value = newValue;
        } else if (!element.type) {
          element.innerHTML = newValue;
        }
      }
    });
  },
  get: function () {
    return value;
  },
  enumerable: true
});
```

这段代码与前一步中代码的主要差异在 set 函数部分。首先，我们更新了对象的值；然后，从已有的元素里寻找是否有绑定的值，即通过 element.getAttribute('data-tw-bind') === prop 来判断是否有绑定的值；最后，根据元素的类型来进行相应的处理。为了示例方便，我们只处理类型是 text 的值，如果类型是 text，就改变它的 value 为新的值。

这样，我们就可以在更新 model 值的时候，同时更新到 UI 上，并在 UI 的值发生变化的时候，将值更新到 model 上。由于 Object.defineProperty 只支持 IE9 及以上的浏览器，所以 Vue 采用这一类技术的框架，就不支持 IE8 浏览器。

有了这样一个双向绑定的功能，我们就可以很方便地处理大量数据与 UI 的交互，而不是手动去编写更新值的方法。

5.3.5 前端路由原理及实践

传统的多页面应用，路由及相关的状态都是由后端控制的，发生变化时也由后端来做相应的处理。当应用的重心发生转换——一部分由后端转移至前端时，路由的控制权也相应地发生了一些变化。发展至单页面应用时，几乎都是由前端来控制路由的。路由是前端应用的一个核心功能，它用于连接分散在各处的控制逻辑。

谈及路由映射时，我们讨论的是监听路由的变化，以调用函数来处理对应的逻辑。即使在多页面应用中也是如此，当用户执行一些操作，并且可以和别人分享时，记录下这些操作。当其他用户打开带操作状态的链接时，多页面应用立即响应，并更新页面状态。

5.3.6 两种路由类型

从前端路由的基本原理来看，要实现一个前端路由并不复杂。在实现之前，我们要先了解一下两种类型的前端路由。

基于 History API 的路由和传统的路由基本一样，区别是它可以通过 JavaScript 来控制。HTML5 中的 History API 无刷新更改地址栏链接，配合 Ajax 可以做到无刷新跳转。依据其提供的几个基本方法，我们可以自动操作单页面应用的所有逻辑：

- back，返回前一页。
- forward，在浏览器记录中前往下一页。
- go，在当前页面的相对位置从浏览器历史记录加载页面。
- pushState，按指定的名称和 URL 将数据 push 进会话历史栈。
- replaceState，指定的数据、名称和 URL，更新历史栈上最新的入口。

由于 History API 是基于真实的 URL 操作的，如果没有配置好相应的 HTTP 服务器，那么在刷新页面的时候就找不到相应的页面。

基于 Hash 的路由在浏览器地址栏是通过#号及后面的部分来代表 URL 的，其背后的原理要从 DOM API 中的 location 讲起，location.href 会获取当前页面的 URL，而 location.hash 将会获取#后面的内容，如我们的 URL 是 https://www.phodal.com/#test=true，那么我们就可以通过 location.hash 来获取#test=true 的值。当一个窗口的 Hash（URL 中#后面的部分）

改变时就会触发 hashchange 事件。监听相应的事件：window.addEventListener ("hashchange", funcRef,false)。绑定路由与函数的映射关系，就可以完成一个基础的路由系统。

此外，路由的 Hash 还具有以下一些特点：

（1）Hash 值的改变不会导致页面重新加载。

（2）Hash 值由浏览器控制，不会发送到服务器端。

（3）Hash 值的改变会记录在浏览器的访问历史中，因此可以在浏览器中前进和后退。

还有一点需要注意：使用浏览器访问网页时，如果网页 URL 中带有 Hash，页面就会被定位到 id（或 name）与 Hash 值一样的元素的位置。

它的这些性质表明，它适合用于前端路由。同时，由后端控制路由的多页面应用，往往只会采用 Hash 的方式来进行路由。那么，让我们来探索一下如何自制一个 Hash 路由器，以便于让我们进行页面间的参数传递，简化复杂应用的开发流程。

5.3.7　自造 Hash 路由管理器

在继续深入之前，让我们先来看一下 Angular 框架的路由示例：

```
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'hero/:id',      component: HeroDetailComponent },
  { path: '**', component: PageNotFoundComponent }
];
```

在这个示例中，我们清晰地看到路由和组件之间的映射。在 appRoutes 数组中的对象里：path 参数使用正则表达式定义了要跳转的路由；component 则是对应的要调用的组件。

由于我们要实现的路由管理器比较简单，所以我们直接从代码展开：

```
//代码位于 chapter05/router/easy-router 目录下
function Router() {
  this.routes = {};
  this.currentUrl = '';
}

Router.prototype.add = function (path, callback) {
  this.routes[path] = callback || function () {};
```

```
};
Router.prototype.refresh = function () {
  this.currentUrl = location.hash.replace(/^#*/, '');
  this.routes[this.currentUrl]();
};
Router.prototype.load = function () {
  window.addEventListener('load', this.refresh.bind(this), false);
  window.addEventListener('hashchange', this.refresh.bind(this), false);
};
Router.prototype.navigate = function (path) {
  path = path ? path : '';
  location.href = location.href.replace(/#(.*)$/, '') + '#' + path;
};

window.Router = new Router();
window.Router.load();
```

整体的逻辑很简单，在应用初始化的时候添加所有路由。随后，在 hashchange 发生变化的时候调用相应的处理逻辑。对应的几个不同函数的相关说明如下：

- ◎ add，用于创建路由集、添加路由的 key 及其对应的函数。
- ◎ refresh，解析出当前路由的 key，再根据 key 从路由集中找到并调用对应的路由处理函数。
- ◎ load，初始化路由相应的监听事件。
- ◎ navigate，跳转到对应的路由。

如果想使用如 Angular 一样的正则表达式路由方式，只需要在 refresh 函数中加以处理即可。接着，只需要添加路由，以及对应的处理逻辑，就可以完成一个完整可用的路由管理器，具体实现代码如下：

```
//代码位于 chapter05/router/easy-router 目录下
Router.add('/', function () {
  document.getElementById('demo').innerHTML = "Router Home";
});
Router.add('/blue', function () {
  document.getElementById('demo').innerHTML = "Router Blue";
});
```

在这个路由示例中，我们添加了两个路由：

◎ “/”，用作 Hash 为“#/”的路由，即首页。

◎ “/blue”，用作 Hash 为“#/blue”的路由。

可以看到，两个路由添加了在跳转至该路由时修改页面的 HTML 的功能。对于如 Angular 这样的框架级路由来说，它们做的事情会稍微复杂一些，它们以完整的类的形式存在，有自己的生命周期。组件在运行的过程中按照这个生命周期来执行，但是对于多页面应用的前端路由来说，只需要适应前端页面的 Hash 变化，就可以做出一个不亚于多页面应用的复杂交互。

当前，这种路由变化引起直接的 DOM 更新的方式不合理。结合之前的 Virtual DOM 来进行操作，会是一种更合理的方式。

5.4　避免散弹式架构

在编写多页面应用的时候，不同的函数有可能都在操作同一个节点，而这些函数可能位于不同的代码文件中，故而可以称之为散弹式架构。其来源是《重构》一书中的散弹式修改（Shotgun Surgery）。

如果每遇到一种变化，你都必须在许多不同的 classes 内做出许多小修改以响应之，你所面临的坏味道就是散弹式修改（Shotgun Surgery）。如果需要修改的代码散布四处，你不但很难找到它们，也很容易忘记某个重要的修改。——《重构》

现今的单页面框架已经不太可能存在这样的问题，而多页面应用仍然还存在这样的问题。

5.4.1　散弹式架构应用

在笔者刚开始编写大型前端应用的时候学习的是 Backbone，因为并没有一个好的 MVC 框架，在当时的情况下仍然是最适合的选择。在当时来看，即使是比较早的移动 SPA 应用，也具有它的一系列问题。散弹式架构应用包含如下几个部分：

◎ Backbone 轻量级 MVC。

- jQuery 使用户能更方便地处理 HTML、events、实现动画效果。
- Mustache 模板引擎。
- Require.js 依赖模块管理。

这样的架构看上去似乎并不存在问题。我们使用 Backbone 来创建一个又一个页面，每个页面对应一个特别的类 Controller 文件，对应的操作页面元素的逻辑也就分散在不同的 Controller 文件里。对于一个复杂的应用，它则会出现一个 PageView、ListPageView 等的页面，而 ListPageView“继承”自 PageView，PageView 则“继承”自 Backbone.View。在一些复杂的情况下，还会有 SubListPageView 这样的页面出现。

当我们需要操作 DOM 时，就会用到 jQuery/Zepto，就会出现问题：jQuery 带来的散弹性架构问题。

不同的页面可能会对页面的某个元素（如 Header）进行修改。如在列表页面对搜索元素进行修改，在详情页面对返回元素进行修改，在修改搜索结果页面也是对返回元素进行修改，等等。随着业务的进一步复杂，页面修改就失去控制了。每次当我们要改 header 的时候，都需要重新梳理相关的逻辑，重新理解其运行的生命周期。到最后，控制这个 header 返回按钮的已经分配到七八个 Controller 文件里。

据我们所知，JavaScript 并不是一门完整的面向对象的语言。我们在写代码的过程中，由于 Code Diff 和结对编程的存在，减少了一些潜在的问题。同时，我们自己在原始的 View 里采用了 LifeCycl 的设计。而在下一层 View、PageView 中则会继承这样的设计，以此类推。

全局搜索相应的 ID，再寻找其继承关系，逐一进行调试。除了每一层 View 的关系，还会在全局中对一些 DOM 进行处理。当你在某一层级修改 DOM 时，就只能“祝你好运”了。

在新的 MV*框架里可以使用组件化解决问题，而在旧的多页面应用中，我们得想办法去解决这个问题。

5.4.2 如何降低散弹性架构的出现频率

在上述例子中可以发现，这种情况的原因主要来源于对 DOM 的操作。在组件化的单页面应用中，通常通过参数、属性、事件传递来通知对应的组件哪里需要修改，这在多页

面应用中则会复杂一些。我们有太多需要添加交互的地方，越复杂的交互越容易出错。

对于一个多页面应用来说，完全避免散弹性修改几乎是不可能的，但是可以尽量降低它们的出现。

统一交互处理。既然 DOM 处理逻辑分散在不同的地方，解决方法可以是寻找一种方式来统一管理。类似于组件化的方式，可以提供一个类、函数来统一管理，需要的时候调用相应的方法来控制。也可以通过全局事件来管理，处理部分监听相应的修改。这种方式相当理想——复杂的交互导致复杂的代码，多数是因为设计初期没料到这个地方会变得如此复杂。设计初期，如果知道交互演进得如此复杂，那么我们就会有相应的策略和方式来进行管理。

按页面拆分脚本。这种方式可能是历史的一个倒退。但是，对于复杂交互的多页面应用来说也是一个可行的方案。常规的打包是将所有 JavaScript 脚本打包到一个 JavaScript 文件中。这时，大部分对于 DOM 的事件监听，可能就直接运行了。当我们操作某个 DOM 的时候，会触发相关订阅者的逻辑。而现在，每个页面加载独立的 JavaScript，因此降低了相互之间的影响。然而，这种方式依赖于更加复杂的构建系统——我们需要一个共用库，以及独立的页面脚本，然后才能构建出独立的脚本。

ID 并非 class。如果在一个元素上即有 ID 又有 class，那么我们应该使用 ID。这一点毫无疑问，可一旦某个元素上没有 ID，那么我们往往直接使用 class 来选择元素。这时容易导致一个常见的 bug，即当我们没有意识到这个 class 在多个元素上使用时，相应的 DOM 操作就会影响到其他元素。因此，在编写相应的 HTML 时，尽量将相关的 DOM 用 ID 进行标识。其他开发人员复用 CSS 时，就不会因为使用了相同的 class 而出现 bug。

唯一的选择器。在一个页面里 ID 通常是唯一的，需要确保它在全局（所有页面）是唯一的，以合理地标识某一特定的 DOM 节点。为了唯一性保障，可以制定一些基本的规范：

- 集中处理那些通用的 CSS。通用的样式，以统一的规则进行命名。
- 按照不同的业务（领域）命名 CSS。如果难以确定业务（领域）的界限，那么就通过 URL 来命名，一个页面的 URL 是唯一的。
- 依据不同的页面类似细分。如 list、detail，都可以用于标识相关的 ID 和 class。

当然，还有其他方式可以避免出现这种问题，然而要对付复杂交互，更有效的方式是使用组件化方案。它能解决当前的问题，由于逻辑是在组件内部，再复杂也是能进行测试的。

5.5 小结

在本章中，我们对比了单页面应用和多页面应用，了解了这两种应用的适用场景。随后详细地深入了：为什么在某些场合下，使用多页面应用是一种更有效的选择？单页面应用的学习成本、构建成本、后台渲染成本，以及其带来的应用架构的复杂性，使我们在有些时候应该考虑多页面应用。

然后，我们简单地了解了如何使用现有的技术来开发简单的多页面应用。对于复杂的多页面应用来说，我们学习了 MVC 框架的一些核心要素：双向绑定、模板引擎、路由映射。它们都是一个 MVC 框架核心的、可以独立出来的元素。通过直接使用这些核心组件，我们既可以不绑定框架，又可以灵活地扩展应用的架构。同时，在未来还能更方便地进行架构演进。

有了这一章的基础，我们将能更好地把握单页面应用背后的思想。

6

第 6 章

架构设计：单页面应用

在业务不断发展的过程中，由于前端项目变得越来越复杂，所以我们要考虑拆分出前端应用部分，使之成为一个能独立开发、运行的应用，而非依赖于后端渲染出 HTML 的多页面应用。我们不得不采用更符合复杂客户端、UI 层开发的 MVC 架构，以便更快速地开发前端应用。前端应用采用了 MVC 框架，也变成了单页面应用，前后端便开始分离了。

在开发一个单页面应用时需要了解什么呢？这便是本章要关注的核心内容：

- 前端应用的 MV*架构是什么？它是如何实现的？
- 选择怎样的前端框架？Angular、React、Vue 要怎么选择？
- 开发一个单页面应用时，还需要什么储备？脚手架、UI 库、模板？
- 如何做好一个单页面应用的后台渲染？

这样看来，开发单页面应用没有开发多页面应用那么简单，让我们继续探索前端应用，本章从 MVC 架构谈起。

6.1 前端 MV*原理

架构，并非是凭空想象出来的，而是从持续不断的实践中总结出来的。开始，软件工程师按照需求写下一行一行代码，一个一个又长又复杂的函数。时间久了，软件工程师发现这样的代码不好维护。因此，产生了面向对象编程，它可以帮助我们更好地进行软件的抽象化思维。在不断抽象的过程中，它能将一个勉强可以用在其他项目的架构抽取出来，慢慢地在不断应用这个架构的过程中，也就演进为一个框架。

同时，人们还在不断地反思这其中的复杂过程，整理出一些好的设计模式和架构模式，MVC 架构就是其中的一个。Martin Folwer 在《企业应用架构模式》（2004 年出版）一书中，介绍了最初的 MVC 架构的三个层次的作用，如表 6-1 所示。

表 6-1

层次	职责
表现层	提供服务、显示信息、用户请求、HTTP 请求和命令行调用
领域层	逻辑处理，系统中真正的核心
数据层	与数据库、消息系统、事物管理器和其他软件包进行通信

在前端看来则是：

◎ Model（模型层），用来获取、存放所有的对象数据。

◎ View（表现层），呈现信息给用户。

◎ Controller（控制层），模型和视图之间的纽带。

两种解释差距并不大。就前端而言，当我们更新数据模型（Model）时，就需要通过控制器（Controller）来更新 UI（View）。想要更好地了解它们的关系，可以先来看一个前端 MVC 架构的示例。

6.2　前端 MVC 架构原理

了解前端 MVC 最好的形式，便是从代码中来看 MVC 架构。为此，我们先准备了一个与 MVC 架构相关的示例。在这个示例中，我们要实现的效果是：单击页面上的一个名为 hello 的按钮，然后将按钮的显示形式变成 world。功能很简单，但是要以 MVC 的形式来呈现。如果以非 MVC 的形式来呈现，那么代码如下：

```
<button id="demo-button"></button>
<script>
var text = "hello"
var demoButton = document.getElementById('demo-button');
demoButton.innerText = text;
demoButton.addEventListener('click', {
    handleEvent: function(event) {
        event.target.innerText = "world"
    }
})
</script>
```

在上面的代码中，初始化时会更新按钮的显示文字，单击按钮时也会更新按钮上的文字。为了方便后面的代码解释，我们传入的 addEventListener 变量是一个带 handleEvent 参数的变量，而非一个函数，两者的效果是等价的。

由于我们要修改的是按钮背后的文字，所以需要将文字变成一个变量，存于 Model 函数中。对应的 Model 函数代码如下：

```
//代码位于 chapter06/basic-mvc 目录下
function Model() {
  this.text = "hello";
}
```

在这个 Model 函数里，我们创建了一个 text 对象，在 text 对象中存储了一个“hello”字符串，所以 Model 函数的作用是：提供和存放数据。

随后，在我们的 MVC 架构中，还需要有一个 Controller 实现：

```
//代码位于 chapter06/basic-mvc 目录下
function Controller(model) {
  var that = this;
  this.model = model;
  this.handleEvent = function (e) {
    e.stopPropagation();
    switch (e.type) {
      case "click":
        that.clickHandler(e.target);
        break;
      default:
        console.log(e.target);
    }
  };
  this.getModelByKey = function (modelKey) {
    return that.model[modelKey];
  };
  this.clickHandler = function (target) {
    that.model.text = 'world';
    target.innerText = that.getModelByKey("text");
  }
}
```

Controller 主要包含三个函数：

◎ handleEvent，用于响应相关的事件。

◎ getModelByKey，返回对应的模型数据的值。

◎ clickHandler，用户单击后对应的操作。

与上面的按钮示例类似，主要的交互逻辑在 handleEvent 函数里。在这个函数里，我们判断事件的类型，如果是一个 click 事件，那么我们就调用 clickHandler 函数。

不同的 HTML 元素有不同的事件，如输入框，有 input、keydown、keyup、keypress 等，对应不同的事件，需要有不同的处理逻辑。它与我们之前编写双向绑定相关的实现极为相似，需要针对不同类型的 DOM 做双向绑定的处理。

在有了 Controller 之后，还需要实现一层 View，代码如下：

```
function View(controller) {
  this.controller = controller;
```

```
    this.demoButton = document.getElementById('demo-button');
    this.demoButton.innerText = controller.getModelByKey("text");
    this.demoButton.addEventListener('click', controller);
  }
```

View 层实现了相应的初始化和事件绑定。初看应该在 Controller 进行绑定，但是有了双向绑定之后，都是由 View 层来自动绑定的。

最后，我们将它们（Model、View、Controller）结合到一起，便是一个粗糙的 MVC 示例：

```
  function main() {
    var model = new Model();
    var controller = new Controller(model);
    var view = new View(controller);
  }
```

以这种 MVC 的方式实现事件的响应和逻辑处理，有助于帮助我们处理复杂的前端应用。但是由于不可能在每个小的部分中都以这种方式去实现每个功能，所以相关的操作变得非常烦琐。而且，当我们拥有大量的交互时，使用原生的 JavaScript 来开发应用，代码将变得难以维护。这时，我们需要用前端框架来解决：

（1）保持 UI 与模型的同步（即双向绑定）的问题。

（2）原生的 JavaScript 难以编写复杂、高效、易维护的 UI 的问题。

因此，我们可以结合上一章中提出的几个核心要素（路由、双向绑定等），来完成一个前端框架所需要的核心部分。

6.3　进阶：设计双向绑定的 MVC

让我们回顾一下第 5 章中的双向绑定示例。在该示例中，我们实现的是一个观察者模式，即定义对象间的一种一对多的依赖关系，使得每当一个对象状态发生改变时，其相关的依赖对象皆得到通知并被自动更新。

为此，我们需要在 MVC 应用里添加两种类型的观察者模式：

◎　基于数据劫持的观察者模式。当数据模型的值发生变化时，调用对应的函数。

◎ 常规的观察者模式。提供可供调用的调用方式。

当数据模型发生变化时，对 Model 函数进行修改，代码如下：

```
//代码位于 chapter06/basic-mvc-twb 目录下
function Model() {
  var that = this;
  var text = "hello";
  this.listeners = [];

  Object.defineProperty(that, "text", {
    get: function () {
      return text;
    },
    set: function (value) {
      text = value;
      that.notify();
    }
  });
}
Model.prototype.subscribe = function (listener) {
  this.listeners.push(listener)
};
Model.prototype.notify = function (value) {
  var that = this;
  this.listeners.forEach(function (listener) {
    listener.call(that, value);
  });
};
```

我们使用了和第 5 章中类似的数据劫持，与之不同的是，数据更新（Setter）的相关函数被抽取出来并放到 notify 函数中。一旦某个数据的值发生变化，就会触发相应的监听者的调用逻辑。而在 View 层里，我们则会进行之前的双向绑定逻辑，并注册成为一个订阅者，以便在数据发生更新的时候更改对应的 View。

于是，View 层得到进一步的完善：

```
function View(controller) {
  var that = this;
  this.controller = controller;
  var elements = document.querySelectorAll('[data-tw-bind]');
  elements.forEach(function (element) {
```

```
      if (element.type === 'button') {
        element.innerText = controller.getModelByKey("text");
        that.call = function (data) {
          element.innerText = data.text;
        };
        element.addEventListener('click', controller);
      }
    });
    this.controller.model.subscribe(this);
  }
```

与之前的双向绑定类似，我们会遍历所有的 data-tw-bind 属性，为 button 类型的元素的初始化值自动绑定 click 事件。同时，添加了订阅的响应函数 call，当 Model 层的数据发生变化时更新 button 上的显示文本。此外，将 View 注册为数据模型的监听者。

最后，Controller 通过 clickHandler 方法对数据进行更新。

```
function Controller(model) {
  ...
  this.clickHandler = function (target) {
    that.model.text = 'world';
  }
}
```

当这个数据发生变化时，会调用数据劫持 Setter 中的相关 notify 方法。也就会调用所有订阅者的响应函数 call 方法，并且还会更新按钮上的文字。为了验证双向绑定是可以工作的，我们可以在 Controller 中对 setTimeout 进行设置，让其在 3 秒后改变 model 的值，以观察对应的文本是否更新：

```
  setTimeout(function(){
    that.controller.model.text = "3s"
  }, 3000)
```

前端 MV*框架的原理就是通过观察和订阅来进行联动操作，以自动触发各种逻辑函数。

自己编写 MV*框架，是一个很好的提升自身能力的方式。但是在开发应用的时候，我们还是选择前端框架。一方面是因为，前端框架能满足我们快速开发的需求；另一方面是因为，前端框架拥有自己的生态，能大大地提高开发效率。但是框架那么多，我们应该选择哪个呢？

6.4 前端框架选型

刚开始学习前端的时候，SPA（单页面应用）还没有现在这么流行，可以选择的框架也很少。现在，随便打开一个与该技术相关的网站或应用，只需要简单地看几页，就可以看到丰富的与前端框架相关的文章——Angular、React、Vue。

当笔者还是一个新手程序员时，从不考虑技术选型的问题，因为不需要做技术选型和更换架构，觉得框架丰富和笔者没关系，反正还是用现在的技术栈。等到真正需要技术选型和更换架构的时候，依靠之前的基础知识，仍能很轻松地上手。

可是到了需要考虑选型的时候，真觉得天仿佛要塌下来一般。如果选择 A 框架，则使用过 B 框架的人可能会有些不满。如果选用 B 框架，则使用 A 框架的人会有些不满。选择一个过时的框架，则大部分人都会不满。这点“小事”，也足以让你几天几夜睡不上一个好觉。与笔者在《全栈应用开发：精益实践》一书中所讲的类似，技术选型考虑的一系列因素如图 6-1 所示。

图 6-1

选定一个前端框架时，我们还要考虑的因素有：

◎ 框架是否能满足大部分应用的需求？如果不能，那么需要使用哪个框架？

◎ 框架是否有丰富的组件库？如果没有，我们的团队和组织是否有独立开发的能力？

◎ 框架的社区支持怎样？在遇到问题时能否快速方便地找到人解答？

◎ 框架的替换成本如何？假如我们的新项目将使用 B 框架，那么我们还需要额外学习什么内容？

每个问题都不是一个简单、独立的问题，它们都值得我们深入地探索一番。

6.4.1　选型考虑因素

如果旧的框架逐渐被淘汰，那么就需要选择一个新的框架，以便帮助大家平衡地过渡。这时如果你也成为一个前端的技术负责人，那么需要评估哪个框架更适合我们。

恐怕不是一句话或者几句话就能解决这个问题的，我们需要组织讨论会，进行各种框架的对比分享等。待讨论的事项如下：

◎ 团队成员能否快速掌握该框架。框架拥有自己的设计思想，它可能有更适合的用户群。如 Angular 其总体的架构思想依赖注入、强类型等，更适合那些有后端经验的开发者。如 Vue 框架更容易上手，适合初学者进行 Web 开发。而对于一个没有经验的新手，直接使用 Angular 框架，则需要较长的学习时间。

◎ 框架的生态是否丰富？是否拥有我们所需要的功能组件？技术选型要考虑的一个因素是生态系统，即是否有对应的可选择的组件、库是否足够丰富。多数时候不同的框架之间都可以找到一些相似的组件，但有些时候一些复杂的交互组件，可能没有相对应的框架实现。一旦业务上对这类组件的需求比较大，那么我们可能就只能放弃某个框架，同时考虑消耗的成本。

◎ 不同框架对于不同浏览器的支持程度如何？由于实现机制的不同，框架会对浏览器有一定的要求。这部分内部，我们会在随后的章节里，进一步展开讨论。

◎ 框架的后期维护成本和难度怎样？项目的维护难度与其花费的时间和代码量是成正比的，即时间越长、代码量越大，维护成本也就越高。越是大型的项目，到后期也就越难维护；而小型的项目，则不存在这样的问题。因此在选择框架的时候，可以考虑项目的规模，越有可能变大的项目，选择一个大而全的框架，往往会更容易维护。

◎ 是否能以最小的代价迁移现有的应用？当我们选定 A 框架时，往往后面编写的代码都难以迁移到 B 框架。比如我们使用 React 编写的组件，无法直接在 Angular 或者 Vue 中使用。尽管随着 Web Components 技术的发展，跨框架使用变得越来越有希望，但是这也意味着我们需要做大量的适配。

随着时间的流逝，A 框架在未来不一定适合我们，可能 B 框架更适合现在的我们，那么我们可能会踏入原来的坑中，继续探索 B 的可能性。

此外，还有一个场景是，当一个框架的 API 不断变化并且向下不兼容的时候，就会带来额外的维护成本。一个典型的例子就是 React Native，在笔者使用 React Native 开发一个应用的过程中，中途更新了版本，由于 API 的变更影响了第三方组件，并且额外带来几天时间的升级成本。在这个应用的生命周期里，我们还需要不断地追随这部分变化。

当我们踏入新手村时，考虑的往往是容易上手的框架；小的团队、组件选择框架的时候，也往往考虑容易上手的框架。但是项目变大后，大而全的框架往往是更容易使用的。

6.4.2 框架类型：大而全还是小而美

工作的时候往往对协作和技术选型会有一些要求，而平时练习的时候就不一样了，往往会不断地尝试新的技术、框架和库，以提升自己的技术视野。并且，在平时练习的时候，我们可以承担使用新技术带来的风险。如果我们平时进行了大量的练习，那么在工作的时候就可以轻松地引入使用。

对前端框架的选型，笔者一直有自己的偏好行为，工作时往往选择大而全的框架，平时练习的时候往往选择是小而美的框架。一方面担心变更会带来额外的麻烦，另一方面可以学习尝试新的框架、新的可能性。

1. 大而全

大而全，顾名思义框架大、功能又全，开箱即用。相对于小而美的框架而言，它可以提供给开发者一个完整应用的所有要素。也因此，在提供大量功能的同时，体积上也比较大，所需要了解的知识也比较多。另外，大而全的框架还有着集中统一的详细文档，能提供与编程、架构相关的规范。这种框架带来的明显问题是：上手成本高、框架限制高。

比如 Angular 是一个大而全的框架，对于一个通用应用而言，我们可以用它来实现大部分功能。这种类型的框架，与大型组织的组织结构一样，有些庞大和臃肿，但是却可以有效率、有组织地进行开发。与此同时，它也更容易被这些组织内部所采用。

但是，刚上手 Angular 框架时，需要开发者具有比其他前端框架更深厚的计算机基础。对于使用弱类型语言 JavaScript 为主的开发者来说，刚上手强类型语言 TypeScript，会有诸多的不习惯，还需要学习一系列的语法知识。与此同时，还要遵循 Angular 框架制定的

“强制”规范和框架准则。

2. 小而美

小而美，即框架本身只提供核心的功能，要完成一个前端应用，还需要寻找其他组件，如路由、数据获取等。因此小而美的框架，更像是基于 Linux 内核的操作系统。Linux 本身只提供内核，要变成一个完整的操作系统，至少需要有一个桌面软件。为此还需要一系列的工具，如 gcc、glibc、binutils、make 等。相比之下，大而全的框架更类似于 macOS 或者 Windows 系统，可以开箱即用。

小而美的框架灵活性比较高，可定制度也比较高。我们可以寻找合适的组件，使用到我们的系统上，使它更加灵活。哪怕某天，一个组件不好用，也可以进行相应的修改，或者重写。即使是核心的前端框架出现问题，如出现版权问题，也能切换到一个合适的替代框架上。相应的组件部分，也可以复用，而不需要重写。但是对于一个大而全的框架来说，则相对比较困难，一旦出现问题只能尝试使用新的框架。

小而美的框架的主要问题是：维护成本太高——这里讨论的是维护成本，也就是在这个软件上可能还要继续开发一年、两年，而不是几个月。在小而美的框架里，组件间的依赖，版本间的限制，上下游的同步等，在时间线上就是各种麻烦的问题。在工作上，花费大量时间解决这种技术问题，真的会影响开发体验——只会导致加班。

以 React 为例，为了搭建一个基于 React 的框架，我们一般需要这样一些东西：

◎　Redux，react-redux，用于管理组件状态。

◎　react-router-dom，react-router-redux，用于管理前端路由。

◎　whatwg-fetch，用于与后台交互的数据请求。

◎　redux-immutable, redux-saga 用于更好的管理状态。

……

时刻关注这些关键组件的版本，几乎也成了日常工作的一部分。一旦有一个组件不再被维护，就需要快速尝试其他可替换的组件，并将其配合到系统中使用。因而多数时候使用小而美的架构是一种快速的解决方案，而受限于不同水平的开发者，新手程序员往往不能编写高质量的代码。如果使用大而全的框架，由于要遵循相关的规范，所以从一定程度上间接地提升了开发者编写代码的水平。

在了解了大而全或是小而美之后，让我们继续了解一下 Angular、React、Vue 三个前端框架，以及笔者认为的一些合适的使用场景。

6.4.3 框架：React

接下来，让我们谈谈 React，如其官方介绍：React – A JavaScript library for building user interfaces，React 是一个为数据提供渲染为 HTML 视图的开源 JavaScript 库。从这个定义来看，可以发现 React 是一个 View 层，它的作用是提供了一套数据机制。

React 是第一个采用 Virtual DOM 的、流行的前端框架。传统的 DOM 操作是直接在 DOM 上操作的，当需要修改一系列元素中的值时，就会直接对 DOM 进行操作。而采用 Virtual DOM 则会对需要修改的 DOM 进行比较（DIFF），从而只选择需要修改的部分。因此，对于不需要大量修改 DOM 的应用来说，采用 Virtual DOM 并不会有优势。

除了在编写应用时不需要对 DOM 进行直接操作、提高了应用的性能，React 还有一个重要的思想即组件化，即 UI 中的每个组件都是独立封装的。并且，由于这些组件独立于 HTML，所以它们不仅可以运行在浏览器里，还能作为原生应用的组件来运行。组件之间可通过属性和事件来进行通信。同时，在 React 中还引入了 JSX 模板，即在 JS 中编写模板，而且还需要使用 ES6。

令人遗憾的是，如我们在 6.4.2 节所述，React 只是一个 View 层，它是为了优化 DOM 的操作而诞生的。为了完成一个完整的应用，我们还需要路由库、执行单向流库、web API 调用库、测试库、依赖管理库等，这简直是一场噩梦。因此为了搭建出一个完整的 React 工程，我们还需要做大量的额外工作。

1. 适用场景

选择 React 而不是别的框架还有一个重要的原因：React 的思想不局限于前端领域，它还有 React Native、React VR 等，它们可以在不同的平台之上运行类 React 的 View 层。使用与 React 语法类似的 React Native，我们可以编写出原生的移动应用。此外，我们还可以在 Web 应用程序与 iOS、Android 应用程序之间，共享大部分业务逻辑。

如阅文集团在其网站起点海外版中，采用了 React 对原有的应用进行重构。其原因是，由于之前的 App 部分采用 Hybrid 架构模式，因此便采用了 React Native 来解决性能问题。React Native 不仅能解决开发上的性能瓶颈问题，还适合前端开发人员上手开发应用。

阿里巴巴的一些部门（如智能服务事业部）里采用 React 框架的主要原因是生态，其在公司内部已经有大量的与 React 相关的基础设施如 Ant Design、Ant Design Pro 及 Fusion 等 UI 组件库。如 Ant Design Pro 则非常适合中台前端的解决方案设计。

与此同时，出现一些相关框架比如 ReactXP，可以在共享逻辑的基础上实现跨多个目标平台共享视图定义、样式和动画，而不需要做出太多修改，进一步实现代码在原生应用与 Web 应用的复用。随着这项技术在不同平台如 VR 的发展，在遥远的未来也可以实现复用业务逻辑及 View 层。

6.4.4　框架：Angular

Angular 是一个大而全的框架，它提供了开发一个完整应用所需的所有要素。同时，作为背后的开发公司，Google 有一个适用于 Angular 框架的 Material Design UI 库。我们结合 Angular 框架及 UI 库就能完成大部分的前端开发工作。

Angular 官方还提供了开发应用所需的脚手架，包含测试、运行服务、打包等部分。前端开发人员使用官方的命令行工具就可以快速生成 Angular 应用：ng new my-dream-app。在这个官方生成的项目里，可以直接运行和构建 Angular 框架，而不需要像 React 或者 Vue 一样，寻找一个合适的脚手架。

下面的代码是通过 Angular 命令行工具 Angular CLI 生成的项目的依赖（package.json 文件中）：

```
"dependencies": {
    "@angular/animations": "^6.0.3",
    "@angular/common": "^6.0.3",
    "@angular/compiler": "^6.0.3",
    "@angular/core": "^6.0.3",
    "@angular/forms": "^6.0.3",
    "@angular/http": "^6.0.3",
    "@angular/platform-browser": "^6.0.3",
    "@angular/platform-browser-dynamic": "^6.0.3",
    "@angular/router": "^6.0.3",
    "core-js": "^2.5.4",
    "rxjs": "^6.0.0",
    "zone.js": "^0.8.26"
}
```

从这个依赖清单可以看出，项目的主要依赖都是与 Angular 框架相关的内容。此外，Angular 项目默认能提供的一些额外的功能有：

◎ animations，用于制作浏览器动画。

◎ HTTP，用于处理 HTTP 相交的数据交互。

◎ forms，用于处理表单相关的内容。

◎ router，用于进行路由操作。

依赖中剩下的 core-js 是用来作浏览器兼容性（参见 Angular 项目的 pollyfil.ts 文件）的，rxjs 则是响应式编程框架；Zone.js 则是 Angular 团队开发的用于封装和拦截浏览器中的异步活动的库。

同时，在这个 Demo 里，还集成有如下内容：

◎ 单元测试，集成运行环境 Karma 及单元测试框架 Jasmine。

◎ 端到端（E2E）测试，集成端到端测试框架 Protractor。

◎ 静态代码分析，通过工具 codelyzer 及 tslint 进行静态代码的分析。

而在其他框架里，与这些测试相关的内容，则需要开发人员手动来集成。在其官网 https://angular.io/上，它不仅提供了我们需要的文档，还有一些配套的与 Angular 相关的资源。

当然，大而全也意味着在编码的过程中需要严格地按照官方的规范来执行，大而全的框架出现问题的时候，要修改是不容易的。

在最近两三年的时间里，笔者一直使用 Angular 来作为项目的前端框架，原因主要是 Angular 的规范性。Angular 不仅提供了一个前端框架所需要的开发要素，还提供了一系列开发规范和指南。这些规范有些被写在官方的文档上，有些以配置代码的形式存在于项目中，有些则存在于 CLI（命令行工具）中。

这些严格的规范更适合于大公司的规模化运作，尤其在非互联网行业的传统公司里，比如金融、保险等。Angular 大而全的体系方便进行项目管理，既能降低风险，又不会在制定开发规范上花费太多时间。尤其是那些后端出身的部门经理，更容易上手 Angular 的框架。

比如，华为公司在其 DevCloud 平台上选择了 Angular 框架，主要原因是可以确保开发人员严格地遵守规范。另外，华为公司还看重 Typescript、组件化等带来的前端研发效

率、质量及可维护性的提升。

与 React 相比，虽然 Angular 没有官方的 Web 领域之外的平台方案，但是在社区拥有一些相应的框架。如用于跨平台原生应用开发的 NativeScript，及用于混合应用的 Ionic 框架，它们都可以在某种程度上实现与 Web 平台共用逻辑。

6.4.5　框架：Vue

对于没有 Angular 和 React 经验的团队来说，Vue 是一个非常好的选择。Vue 借鉴了 Angular 和 React 的一些思想，在其基础上开发了一套更易上手的框架。它既不像 Angular 需要理解大量的基础知识，也不像 React 在使用 Virtual DOM 的同时需要学习 JSX 及其相关的语法。

当然，使用 Vue 也需要学习基于 Template 的语法 。两者有颇大的区别，但是很显然，使用 React 需要重写之前的业务逻辑，而不能嵌入使用。正是这一点区别，决定了 Vue 在针对传统多页面应用的时候更有优势——我们可以将 Vue 嵌入应用中，而使用 React 或者 Angular 基本意味着重写整个应用。

Vue 对比于 Angular 和 React 框架的一个优势是，对于传统的多页面应用，直接引入 vue.min.js 就可以使用了。直接拿代码库就可以发布了，不需要打包。对于那些需要迁移前端框架的项目来说，它可以以一种渐近式的方式来进行，在成熟后便可作为单页面应用框架来开发前端应用。

Vue 的开发者尤雨溪是中国人，框架本身提供了大量丰富的中文文档，这也为 Vue 的发展和使用带来巨大的优势。

Vue 框架适合于需要快速上手、上线的应用，还适用于迁移传统的多单面应用。如笔者曾因为业务需要创建一个新的移动 Web 应用，要求几天内上线。因为时间短，所以直接排除了 React——没有一天的时间，怕是搭建不好 React 全家桶的。而 Angular 也被笔者排除了，因为它要构建包发布，从流程规范上比较麻烦。最后的选择是 Vue 框架，它可以满足快速上线的需求，同时在后期也可以演进成单页面应用。

Vue 框架还拥有使用类似语法的 Weex 框架，两者的关系类似于 React 和 React Native 框架的关系。前端开发人员在熟悉了 Vue 之后就能快速上手 Weex。值得注意的是，受限于 Weex 的发展，其在移动应用的发展并不是很理想。

比如，滴滴出行选择 Vue 框架是看中了 Vue.js 在移动应用开发中的优势，并对它的未来充满信心。而在早期采用 Vue 框架的组织，如饿了么，已经在内部拥有大量的相关生态。饿了么在后期进行技术选型的时候，也倾向于选择 Vue 作为前端框架。

6.4.6 选型总结

在这么多框架中，到底如何选择呢？

对笔者而言，如果是一个大中型的企业级应用，那么基本上会选择使用 Angular。统一的规范和设计模式对于大型团队来说可代替无数的沟通成本，并且不需要花费大量的时间进行讨论——使用哪个基础的路由、状态管理器等。就像康威定律一样，它也更符合大型组织内部的架构设计。

对于移动应用项目来说，如果采用 React Native，那么 React 在当前是一个更好的选择。

对于一个中小型项目来说，如果团队的新人比较多，并且基础比较薄弱，那么 Vue 就很适合开发人员使用。并且，它也可以使用 Vue 语法的 Weex 来开发移动应用，只是这个 Weex 框架并不像 React Native 那样被大量地采用。

6.5 启动前端应用

选择好前端框架后就可以进行下一步开发了：

（1）寻找合适的脚手架，编写出第一个“Hello, world”。

（2）选择合适的 UI 框架，以快速开发前端页面。

（3）确认浏览器的支持范围，以明确测试边界。

（4）明确响应式设计的需求，以明确在编码的过程中支持哪些设备。

这部分只是那些独立于组织、公司的通用内容。对于流程规范化的组织内部，还会更复杂。

6.5.1　创建应用脚手架

尽管使用框架自带的 CLI 简化了大量的步骤，但当我们直接使用这个框架时，仍然需要自定义大量的工作，而这些工作有很多是重复的，可以提取出来通用的，这就是应用脚手架所做的事情：

- 通用的业务相关模块，比如登录、授权、Token 管理。
- 页面模板页，比如首页。
- 业务模板，比如中后台应用模板。
- 持续部署脚本，比如持续集成、部署脚本。
- 常用的依赖，比如 UI 组件库。

值得注意的是，应用脚手架并非一次编写就完事，而是需要持续不断地优化，特别是及时地跟进依赖的版本。现有的前端项目无一例外地都使用第三方框架、库、组件等，而一个应用的开发周期少则 1～2 个月、多则几年，在这个过程中这些依赖都在不断地更新。为了能在其他项目中使用这个脚手架，我们需要及时地响应依赖更新。

其中一种方式就是使用依赖扫描工具，运行时会读取 package.json 文件，并用 NPM 服务器做比较，以发现依赖的最新版本。这只是其中一小步，更复杂的是，我们要去更新依赖。依赖更新的麻烦之处在于，某个依赖发布了一个大版本，那么其 API 可能发生了变化，需要更新相关的一系列代码。

上述讨论的是内部脚手架的构建方式，在这之外还有一种方式是外部脚手架。这些外部脚手架往往是指开源的脚手架，而越是在 GitHub/Gitlab 上流行的（star 数多）脚手架，在这方面做得越好。开源项目往往符合马太效应——star 数越多的项目使用的人越多，使用的人越来越多的项目，其 star 数也会越来越多。

于是，凡是 star 数靠前的脚手架，在社区使用它的人数也就越多，能及时反映脚手架、修复脚手架问题的人也就越多。因此，对于大部分团队来说，与自己维护脚手架相比，使用社区的脚手架往往是一种更合适的选择。在社区的脚手架做进一步优化即可。

要寻找这样一个脚手架，只需要在使用 Google 或者 GitHub 搜索的时候，加上一些与脚手架相关的词，如 react boilerplate、react starter、react starter kit、react seed 等。克隆（clone）一下代码，试一试这个脚手架是否符合团队的习惯，就可以进一步修改开发了。

值得注意的是，即使像 Angular 这样的框架，也可以寻找一些合适的脚手架。特别在开发一些非桌面应用如移动应用、PWA、Electron 桌面应用的时候，直接使用脚手架会比较方便，因为它不需要额外的配置。

6.5.2 构建组件库

前端应用是一个以用户界面（UI）为主的应用程序，与后端应用相比，它既关注于业务逻辑，又关注于 UI 实现。不同的 UI 和交互设计，也会影响 UI 库的选择。而 UI 交互并不是一种简单的事情，复杂的 UI 交互可以花费几天、几星期、几个月的时间，简单的 UI 也需要几分钟的时间。在没有基础和沉淀的情况下，一般不会考虑做内部的 UI 库。因此，让我们考虑一下，如何选择一个合适的组件库？

1. 选择组件库

选择组件库要考虑的因素有：

◎ 是否需要跨框架的组件库，即可以支持 React、Angular、Vue 等不同的框架？

◎ 组件库是否容易替换？如果替换难度比较大，需要考虑是否进行二次封装。

当我们只有一个前端 MV*框架的时候，选择 UI 组件库并不是一件困难的事。但是，当我们有多个 MV*框架的项目时，选择 UI 组件库则有一些困难。比如，在大型组织、公司里，不同的部门团队会选择适合自己的框架。框架不同，对应的 UI 库也会有所差距。幸运的是，市面上有一些 Design System 或 UI 库，可以在不同的前端 MV*框架上找到与之对应的实现。

Bootstrap UI，作为最普通（流行）的 UI 前端框架，在技术社区里有着不同于前端 MV*框架的相关实现，比如 React-Bootstrap、NG Bootstrap。只需要使用框架名 + Bootstrap 进行搜索，就能找到不止一个非官方实现的版本。不同的流行版本之间可能有一些细微的差异，有一些框架可能新增了一些样式，但是大抵上差不多——基于官方及社区的组件，提供了不同框架的封装。使用 Bootstrap 的优点就是可以在不同的平台上有大量的实现。在大型团队的内部，只要使用了多个前端框架，做一层封装，就可以实现一套适合于内部的 UI 框架。

Material Design，是由 Google 推出的全新的设计语言，旨在为移动设备、平板电脑、台式机和“其他平台”提供更一致、更广泛的“外观和感觉”。作为一个成熟的设计语言，

其被大量地应用在 Google 的产品线中，如 Chrome 浏览器界面、PC Web 端的网页、移动应用（Android、iOS）等，以提供一致的跨平台和应用程序体验。Google 除了提供 Material Design 的相关设计规范、指南，还提供了对应的 UI 框架实现，如用于 Angular 框架的 Material 2 库、用于多页面应用的 Material Design Lite、用于 Web Components 应用的 Material Web Components 库等。虽然官方并没有提供对应的 React 和 Vue.js 版本，但是可以从社区上找到对应的实现。

Ant Design，是由阿里巴巴旗下的蚂蚁金服推出的服务于企业级产品的设计体系。其提炼自企业级中后台产品的交互语言和视觉风格，提供了全链路开发和设计工具体系。在当前（2018 年），其在官网宣称是世界第二流行的 React UI 框架（自称最流行的 React UI 框架是基于 Material Design 的 Material UI 库）。作为一个由中国公司开发的前端框架，其在社区和文档的支持上显然比原生英语的框架要好得多。其官方提供了 React 版本的实现，并且在社区上也能找到不同前端框架的实现。

此外，我们还可以寻找那些专用于不同框架的 UI 库，如专用于 Vue.js 框架下的 Element 等。由于大部分 UI 库替换成本相对比较低，因此更有机会尝试不同的设计。由于这些 UI 库的组件多数是独立的，所以在开发的过程中遇到不合适的组件时，也方便添加一些新的组件。因此，与框架选型相比，UI 库的选择不是一个复杂的工作。

2. 创建组件库

创建自己的组件库，不仅能方便我们设计出符合自己需求的组件，还能进一步扩大团队的影响力。创建组件库的步骤如下：

（1）寻找一个现成的组件库。

（2）构建出组件库的基础设施。从步骤（1）的组件库中删除所有的组件，修改项目名等。

（3）编写一两个测试组件，引入项目中进行测试。

（4）持续不断地更新组件库。

再结合第 4 章“架构基础：设计构建流”中提到的软件包源管理的方式，便可以拥有一个能够对外发布的组件库。

6.5.3 考虑浏览器的支持范围

前端开发，最麻烦的便是考虑浏览器的支持范围。这项工作并不容易，有时候我们修复了针对 A 浏览器的 bug，却因此产生了 B 浏览器上的 bug。谈及旧版本浏览器的支持，我们通常指的是 IE 相应的浏览器支持。对于国内的开发者来说，还不得不考虑使用老旧版本的 IE。旧版本的 IE 浏览器除了兼容性差，还非常不好调试。

如果我们开发的是基于 Electron 框架的桌面应用，那么能极有效率地提供的生产力，因为我们不需要考虑浏览器的兼容性。对于基于某特定平台的应用，如果仅面向 iOS 设备的应用，那么所需要考虑的浏览器要素也就少了。对于 iOS 设备而言，并不存在太大的问题，只要设备少、浏览器内核基本一致，就不会出现太大的问题。而 Android 则不同，由于厂商众多，并且涉及操作系统的版本，同一个厂商的同一款机型的不同 Android 版本，就有可能出现不一致的兼容性问题。

幸运的是，当我们选择了一定的前端框架之后，也就决定了对旧的浏览器的支持范围：

- Angular 只支持 IE9+的浏览器。
- Vue 只支持 IE9+的浏览器。
- React 只支持 IE9+的浏览器。

Angular、Vue、React 都是只支持 IE9 及以上的浏览器，从官方的 issues 或者与社区相关的网站上，我们可以找到一些常见的 bug 是如何修复的。

1. JavaScript 兼容（polyfill）

谈及浏览器支持，有一个很重要的问题是，不同浏览器对于 JavaScript 新特性、API 等的支持，对于新的 JavaScript 语言特性的支持各有不同。为了在上面“平衡”地运行，我们需要一些额外的与兼容相关的 JavaScript 代码，这种代码称为 polyfill。

如在 JavaScript 的数组中有一个新的 find API，即 Array.prototype.find()，下述代码是相应的示例：

```
var array1 = [5, 12, 8, 130, 44];

var found = array1.find(function(element) {
  return element > 10;
});
```

IE 浏览器的 Array 不支持 find 方法，有如下两种解决方式。

方式一：引入 polyfill 文件来模拟对这些 API 的支持，这种方式是通过为旧的浏览器添加方法来实现的。下面是 find 方法的 polyfill 的一种实现：

```
Array.prototype.find = Array.prototype.find || function(callback) {
  ...
}
```

当 Array 中没有 find 方法时，浏览器便会调用我们编写的方法。在这个时候，我们需要了解用到哪些特性。为了方便起见，我们可能引入所有相关的兼容代码，但是这样会导致构建出来的 polyfill 变得过于庞大。

通过 polyfill 的方式只能解决缺少 API 的问题，而当我们要使用一些新的语法特性，比如箭头函数（(参数 1, 参数 2, …, 参数 N) => { 函数声明 }）时，则需要使用转译的方式来实现。

方式二：转译（Transpiling），即通过使用 Babel 等编译器，来将代码编译成 ES5 版本的 JavaScript 代码。通过这种方式可以将我们使用的新 API、新特性都进行转译，以实现对旧式浏览器的支持。而这种转译实际上是使用其他代码来替换相应部分的代码，比如使用一个新的 find 来实现替换代码中的 Array.prototype.find 方法，而不是提供对其的支持。

在一些框架如 Angular 中，既会通过转译来使用新的语法特性，又会通过 polyfills.ts 来引入 core-js 对应的模板，以支持所需要的语法特性。例如：

```
import 'core-js/es6/object';
import 'core-js/es6/function';
import 'core-js/es6/parse-int';
import 'core-js/es6/array';
...
```

在代码中，API 的 polyfill 可以更好地帮助我们向下兼容语法特性，及适当地减少包的大小。

2. CSS 兼容

同样的，浏览器对于 CSS 的支持程度不同，也会影响我们使用 CSS 的特性如 Flexbox 等。尤其在移动端上，移动端不同的 WebView 限制了我们使用特性——不同的制造商改造的浏览器有所差异，会产生一定的适配成本。并且，即使是同一个设备制造商，它们的不

同版本的手机也存在旧版本的浏览器特性落后的问题，这也进一步提高了适配的成本。

如果我们计划使用新的 CSS、JavaScript 特性、API，那么可能没有一种好的方式来支持。使用旧的特性、API 是一个保守的方式，它可以保证代码是可以工作的。但是，总有一天我们要使用新的特性、API，因此我们就需要不断地向前看。

与 JavaScript 拥有 polyfill 库相似的是，多数时候我们并不会自己编写全部的 CSS。我们会采用一些已有的 CSS 框架，这些框架拥有一整套成熟方案，如响应式布局方案、CSS 动画方案、UI 组件样式方案等。在这些方案中，它已经帮助我们解决了大部分的兼容性问题，我们只需要使用即可。

此外，有一些工具，如 Can I Use（https://caniuse.com/）就是这样一个不错的网站，它可以方便地让我们查询不同浏览器对于某一特性的支持程度，如图 6-2 所示。

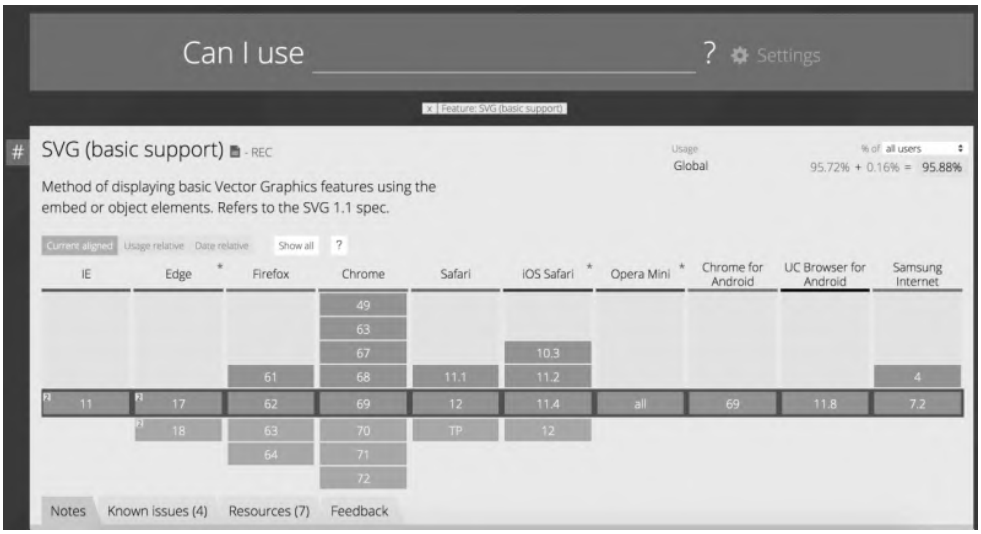

图 6-2

如图 6-2 所示是该网站用来展示 SVG 在不同浏览器上支持的程度。在开发的过程中，当我们计划使用一些新的 API 及新特性时，通过这样的查询可以让我们提前知道在哪些浏览器上会有问题，以便我们在使用时减少与兼容性相关的 bug 出现。

3. 响应式支持

在设备屏幕越来越多的今天，开发一个前端应用，就不得不考虑设备的响应式支持。

在多数情况下，响应式设计考虑的是在不同分辨率下能正常显示，另外一个重点则是在不同分辨率下以不同的方式来展示。后者考验的是产品经理、业务分析师的设计能力，而前者则是考验开发人员。

尽管已经有 UI 框架可以帮我们解决不同分辨率的适配问题，这些适配也能满足我们的需求，但是并非所有情况都是那么理想的。有的时候我们需要支持的分辨率可能不在框架的列表里。而且，对于真正的响应式问题——在什么分辨率下显示什么内容，还需要由我们来决定。

4. 分辨率适配

针对不同的屏幕大小展示不同的内容，只需要结合媒体查询（Media Queries）及 JavaScript 就可以实现。下面是一个媒体查询的示例：

```
    @media (min-width: 768px) and (max-width: 1024px) and (orientation:
landscape) {
      //CSS
    }
```

当设备的屏幕宽度大小位于 768～1024 像素之间，且处于横屏状态时，便会采用上述的 CSS。这时我们只需要获取一些常用的分辨率情况。一般采用业内的数据为主，比如百度统计的分辨率使用情况，如下是 PC 端常见的分辨率：

◎　1280×768 像素。

◎　1366×768 像素。

◎　1440×768 像素。

◎　1920×1080 像素。

◎　2560×1440 像素。

对于移动端来说，可以采用如腾讯移动分析提供的与设备分辨率相关的数据。移动端主要以 Android 和 iOS 为主，两种设备的分辨率又有所不同。Android 主要的分辨率如下：

◎　1920×1080 像素。

◎　1280×720 像素。

◎　1820×1080 像素。

◎ 2280×1080 像素。

……

根据上述的这些不同的分辨率，我们就可以实施相应的响应式支持：

（1）保证显示器在上述的分辨率下能正确显示。

（2）在特定的显示器上显示特定的内容。

与此同时，拥有一个确定好的分辨率范围，也能方便测试人员进行兼容性测试。

5. 移动设备适配

与特定分辨率内容显示相比，还有一个比较麻烦的是设备的响应式适配，即对特定设备进行设备特定的 UI 展示。常见的这类应用如混合应用、移动 Web 应用、PWA 应用等。在 Android、iOS 拥有不同的 UI 设计规范，就需要采用不同的 UI 方案。

如图 6-3 所示是 Ionic 框架为两个平台提供的适配方案：

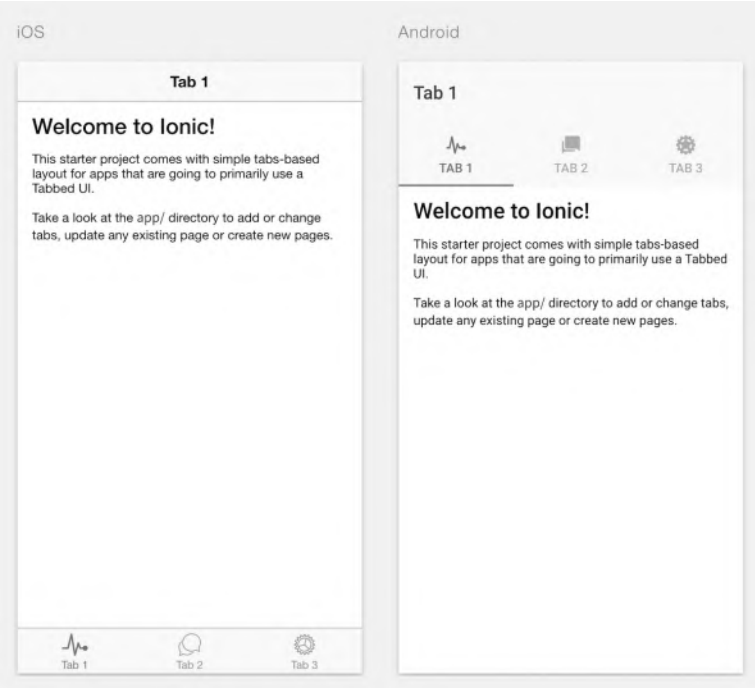

图 6-3

在图 6-3 中，Android 和 iOS 各自拥有不同的标签栏（Tabbar）：iOS 的 Tabbar 默认在应用的最下方，而 Android 的 Tabbar 则默认在应用的上方。这就是遵循一定的设计规范的结果。而在真正实施的过程中，Web 应用则会偏向于统一两端的 Tabbar 设计，将其放在应用的下方。这样的好处是可以减少两个操作系统之间的差距，方便开发人员进行开发。

此外，选项卡（Tabs）、开关（Toggle）等在不同的移动设备上，在 UI 上的显示也不相同。

6.6　服务端渲染

在开发单页面应用的过程中，还有一个非常值得考虑的事情：服务端渲染。在没有前后端分离的时期，服务端渲染是默认提供的——由后端将数据渲染到模板上，当用户、爬虫访问时，直接返回这些 HTML。这样做可以达到两个目的：

◎　SEO（搜索引擎优化）。提供一个更好的搜索引擎。

◎　更快的内容到达速度。直接由服务端返回 HTML，会比由客户端加载到 JavaScript 后渲染出页面更快。

单页面应用的服务端渲染与纯服务端渲染有所不同，多数情况下，单页面应用的服务端渲染依赖于服务端拥有一个 JavaScript 引擎，由其运行 JavaScript 后返回相应的模板。但是，当服务端所使用的语言拥有相同的模板引擎时，也可以不需要这种方式。依笔者的开发经验来看，单页面应用的服务端渲染可以分成三种类型：

◎　非 JavaScript 语言的同构渲染。

◎　基于 JavaScript 语言的同构渲染。

◎　预渲染。

每种方式都各有特色，也各自适合不同的场景。

6.6.1　非 JavaScript 语言的同构渲染

同构渲染，即前后端共享模板文件，由各自的模板引擎结合数据来渲染出页面。非

JavaScript 语言的同构渲染是同构渲染的早期演变形式，考虑到它与现今流行的 Node.js 同构渲染方式的差距，我们将其独立出来。

在这种场景下开发单页面应用时，需要寻找一个后端支持的模板类型，如果我们使用的后端语言是 Java，模板引擎是 Mustache，那么就要寻找一个 Java 语言的 Mustache 引擎。在用户或爬虫通过链接访问 URL 时，会由后端获取数据及对应的模板文件，然后渲染出 HTML 提供给用户或爬虫。最后，后端通过<Script>的方式将一些数据及相应的应用状态，嵌入 HTML 中，以提供给前端应用处理。

在这个过程中有两个步骤比较麻烦，一个是寻找合适的渲染引擎，另一个是手动地将状态同步给前端。

在笔者经历过的一个项目里，后台使用 Spring MVC 作为基础架构、Mustache 作为模板引擎，和使用 JSP 作为模板引擎相比没有多大区别——由 Controller 去获取对应的 Model，再渲染出页面。多数时候搜索引擎都是依据站点地图（Sitemap）来进行索引的，所以后台就可以处理这些请求。同样的当用户访问相应的页面的时候，也返回同样的页面内容。当完成页面渲染的时候，就交由前端框架来处理相应的逻辑了。

换句话说，这时候它从一个多页面应用变成了一个单页面应用，如：

◎ 当用户用搜索引擎或 URL 直接访问 article/1c2fa746 时，将直接由后端返回 HTML。

◎ 当用户从该文件跳转到别的页面时，则由前端来渲染页面，不需要经过后端处理。

◎ 当用户刷新页面时，重新由后端返回渲染好的 HTML，再经由前端来处理。

这种渲染方式的优势是，不需要修改现有的后端技术栈就可以直接使用，缺点是需要手动地将后端的状态同步给前端。因此，这种渲染方式在维护上会比较困难。使用非 JavaScript 语言来生成 JSON，往往并不是那么方便。特别对于那些强类型的语言如 Java，我们需要不断地声明类型，判断是否为空等。

对于那些使用 JavaScript 及 Node.js 环境开发后端应用的项目来说，这种方式更加简单——支持浏览器使用的模板引擎，往往也都支持在 Node.js 环境（基于 JavaScript 语言）的同构渲染。

6.6.2　基于 JavaScript 语言的同构渲染

对于前端开发人员来说，使用 JavaScript 语言进行同构渲染，更容易上手，也更具有优势。不需要学习新的语言，利用现有的知识，外加 Node.js 的一些基础，就可以进行同构渲染的开发。

与非 JavaScript 语言的同构渲染相比，基于 JavaScript 语言的同构渲染，更容易将后端的状态提供给前端。这些状态都可以直接在同一个代码中处理，如果是非 JavaScript 语言的同构渲染，那么我们需要在后端语言中保持状态，又在 JavaScript 中处理状态。这是一个反复切换代码库的修改的过程，往往容易出错——因为我们可能忘记去添加对新的状态的处理，或者在后端中提供了新的状态。

更重要的是，基于 JavaScript 语言的同构渲染有一个更大的优势，几乎是所有的主流框架都支持的。如 React 和 Vue 都提供了 renderToString 的支持，Angular 框架则是使用 renderModuleFactory，但是它们之间的差距并不大。下面是 Vue.js 官网的示例：

```
const Vue = require('vue')
const server = require('express')()
const renderer = require('vue-server-renderer').createRenderer()

server.get('*', (req, res) => {
  const app = new Vue({
    data: {
      url: req.url
    },
    template: `<div>访问的 URL 是： {{ url }}</div>`
  })

  renderer.renderToString(app, (err, html) => {
    if (err) {
      res.status(500).end('Internal Server Error')
      return
    }
    res.end('
      <!DOCTYPE html>
      <html lang="en">
        <head><title>Hello</title></head>
        <body>${html}</body>
```

```
        </html>
      ')
    })
  })

  server.listen(8080)
```

其中的主要逻辑都是相似的，在 renderer.renderToString 方法中引用要处理的组件，就可以直接渲染出 HTML。再借助于 Node.js 环境下的后端框架如 express 来返回对应的 HTML。在这个过程中，还可以将路由及应用的状态传递给应用。传递的路由状态与传统的同构渲染的路由状态有很大差异，非 JavaScript 方式的同构渲染的路由状态，需要由后端手动进行维护。

当然，这种渲染方式也存在一定的问题：

◎ 服务端内存溢出的风险。

◎ 组件及模块需要兼容浏览器和后端 Node.js 环境。

◎ 提供状态的获取机制。

当前需要单页面应用的服务端渲染是因为现有的搜索引擎不够强大，如百度等国内的搜索引擎不支持，或者支持不好。未来，一旦搜索引擎支持客户端渲染，就不需要如此复杂的处理过程了。

6.6.3 预渲染

预渲染（PreRender）指的是预先渲染 HTML，并针对爬虫返回特定的 HTML。这种类型的渲染方式并不常见，一方面是因为不适合数据量大的应用，另一方面是因为更新比较麻烦。

因为公司内有项目采用这种方式，于是笔者曾经尝试过用这种方式来渲染，有如下方法可以尝试采用：

◎ 爬虫生成静态页面。在本地运行应用，用爬虫抓取所有页面，再上传到文件存储服务器即可。

◎ 程序生成静态页面。在本地运行应用，内部带有真实的线上数据，由 PhantomJS/Chrome Headless 来渲染页面，再保存为对应的页面。

◎　静态站点生成器。编写一个独立的应用程序，该应用程序将从服务器获取数据，再通过模板来渲染出静态页面。

前两种方式相对来说比较简单，第三种方式略微麻烦。但是不论哪种方式，都存在一些限制：

（1）运行时，需要获取真实的线上数据。

（2）数据发生更新时，需要重新渲染一遍 HTML。

即使我们可以在线上运行生成的代码，还要考虑一些额外的问题：

（1）生成速度。大量的数据生成的时间往往较长，从几小时到几天不等。

（2）数据的实时性。如果产品数据每天都大量地更新，那么就需要每天定时地生成 HTML。

作为一个很有经验的 SEO 开发人员，笔者一点都不喜欢这种做法。要知道 Google 有时候会模拟成真实的用户，不带有爬虫的那些参数和标志，去访问页面。如果我们返回给 Google 的两个页面差异太大——可能是忘记更新了，那么 Google 可能就会认为该网站在作弊。

不过，如果是一个页面数据有限的应用程序，那么这种方式也是一个快速的、便捷的解决方案——它不需要占用大量的服务器资源。

6.7　小结

在本章中我们首先结合 MVC 框架应用介绍了 MVC 架构的基本原理，并结合双向绑定的内容开发了一个支持双向绑定的 MVC 应用/框架。然后，我们介绍了如何选择一个合适的前端框架，并介绍了几个不同的前端框架，以及如何在它们之间进行选择。

接着，我们对如何开发前端应用展开了讨论。前端应用介绍从寻找合适的脚手架开始，到选择合适的 UI 框架，再到确认浏览器的支持范围、明确响应式设计的需求等。最后，还介绍了在开发单页面应用的过程中要考虑的服务端渲染的问题，即用三种不同的方式来进行服务端渲染。尽管有的方式已经不再实用了，但是它能帮助我们更合理地迁移传统的多页面应用。

随着应用的进一步复杂化，我们需要一些更合适的架构来代替当前臃肿的应用：

- 基于组件的架构。
- 微前端架构。
- 微应用架构。

这便是我们在下一章要讲述的内容。

7

第 7 章

架构设计：组件化架构

有了第 6 章的基础，对于单页面应用和前端框架的架构，我们已经有了大致的了解，也能理解其背后的原理。现在我们开发一个应用，当应用复杂化时，需要关注如何降低系统的复杂度。

而这种应用的复杂度像力一样不会消失，也不会凭空产生，它总是从一个物体转移到另一个物体，或从一种形式转化为另一种形式。如果一个页面的逻辑过于复杂，那么我们会将其拆解为多个子页面、模板、组件等，其表现形式是基于组件（Component-Based）的架构，又称为前端组件化。

7.1 前端的组件化架构

在给定的软件系统中，基于组件的架构侧重于对广泛使用的功能进行关注点分离。即

将不同的复杂性、关注点分离出来，分别进行处理，让每一小部分都拥有自己的关注焦点。通过定义、实现松散耦合的独立组件，将其组合到系统中，以降低整个系统的复杂度。

用上述方式来看待应用架构，是一种由顶至底的方式。而在日常开发中，我们则是由底至上来开发应用的，即先构建 UI 组件，再构建页面。从前端来看，组件可以被视为构成用户界面的一个小功能，其表现形式是组件库，而组件库的作用是，通过复用已有的组件来快速构建 UI 应用。从 UI 设计来看，单单只有组件是不够的，还需要关注它们之间的配合。

组件化具有一系列的优点：可重用、代码简洁、易测试等。在这个过程中，它将一个页面的单一 MVC 架构拆解成了多个组件 MVC +一个页面的 MVC 架构，如图 7-1 所示。

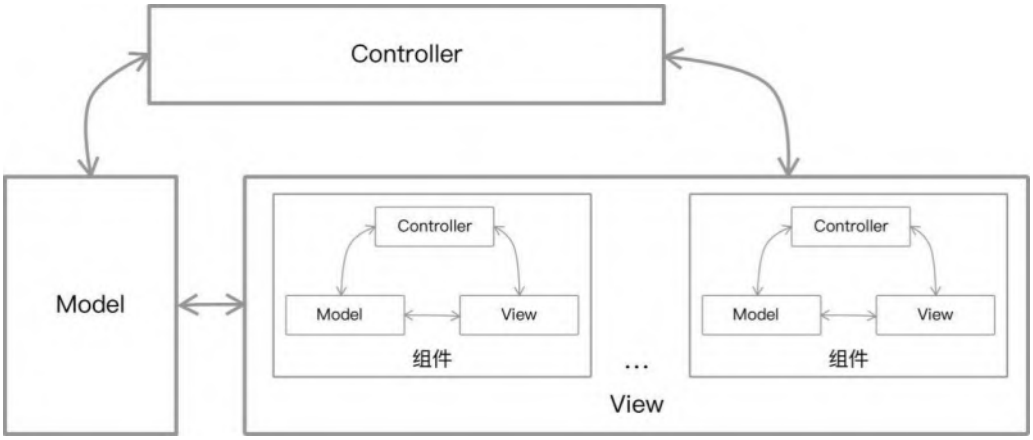

图 7-1

因此，在本章中我们将梳理一遍组件的相关内容，以此来从底至顶地了解前端页面的开发。在这个过程中，我们主要通过组件的发展过程来学习相关的内容：

- ◎ 风格指南（Style Guide）。早期侧重视觉，对设计的文字、颜色、LOGO、ICON 等设计做出规范，产出物一般称为 Guideline，Guideline 通常是 UI 规范。
- ◎ 模式库（Pattern Library），即 UI 组件库。模式库更侧重于前端开发，对界面元素的样式进行实现，其代码可供预览使用，产出物一般称为组件库 UI 框架等，如 Bootstrap 库。
- ◎ 设计系统（Design System）。设计系统在某种程度上结合了风格指南和模式库，并附加了一些业务特别的元素，并且进一步地完善了组件化到页面模板相关的内容。

上述三种类型，仿佛是前端三个不同时期的设计的演进。首先，设计人员设计出系统的 UI 图，由前端开发人员来实现。为了统一设计风格，我们制定了一套规范，即 Style Guide。然后，为了统一相互之间的代码，我们又创建了 UI 组件库。最后，为了统一开发人员之间的语言，我们又创建了设计系统。

在三种类型的模式里，有一些是互相包含的关系。为了讲述方便，我们将各部分内容进行了一定的调整，将合适的内容放在合适的模式上。

7.2　基础：风格指南

风格指南（Style Guide）通常是指，用于文档编写和设计的标准，或者用于特定出版物、组织或领域的通用用法。前端风格指南是 UI 界面中所有元素的模块化集合，以及实现这些元素的代码片段。设计人员可以根据风格指南设计出符合系统统一风格的页面和 UI。风格指南只是一份索引——设计、组件的列表，该列表内容如下：

- 其展现形式，通常是以网站的形式来展现的。
- 设计人员，通过风格指南来查找对应的设计准备及常见的 UI 样式。
- 开发人员，从风格指南上直接复制风格的相关代码。

而构建 Style Guide 并不是一件容易的事，它往往需要长期的积累。哪怕是购买、借助开源的 Style Guide，也并不能直接和项目相匹配，还需要一定时间的磨合，才能做出适合自己团队的指南。

因此，小型的团队可能没有能力维护 Style Guide，但是在有余力的情况下，仍然可以进行尝试。毕竟，这有助于团队的成长。

7.2.1　原则与模式

风格指南是一个包含了常用的 UI 组件的设计，如按钮、表单输入元素、导航菜单、模态框和图标，以及相应的一些设计规范。从它的名字“指南”就可以知道它只是一个指南，而非一个强制的规范。

基础的 UI 组件如何摆放成一个页面，它们摆放的位置和相互的关系就是一个 UI 系统特有的模式。如图 7-2 所示是 Material Design 中关于卡片（Card）元素的一些基本模式。

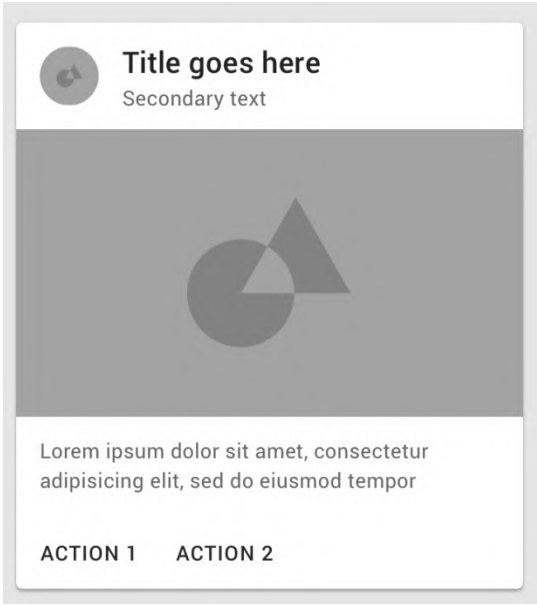

图 7-2

图 7-2 是一个卡片的相关示例，它以明确的大小标明了每个元素该有的大小，以及每个元素应该摆放的位置。当一个新的设计师开始浏览指南的时候，相当于接收其中的模式：某个地方的字体应该多大、需要用什么颜色。在浏览过程中，设计师还会发现相关位置的摆放会遵循一定的设计原则，有一些原则十分常见，比如《写给大家看的设计书》一书中强调的四个设计原理：

◎ 亲密性，即将相关的项（组件）组织到一起。

◎ 对齐，每一项都应当与页面上的内容存在某种视觉联系。

◎ 重复，重复元素以体现一致性。

◎ 对比，对比产生强调，以强调产生强烈的反差。

作为一个指南，除了在文档或网站上明确指出如何使用这些原则，还需要有足够的示例。在这一点上我们可以从开源社区的实践上找到一些对应的案例，比如 Ant Design 直接采用了上述的四个原则，并添加了自己的设计系统特有的原则如：直截了当、足不出户、

简化交互等。然后，在相关的模式中采用了上述的原则来形成自己的设计。案例数量越多，越能对后来者有启发。

既然原则与模式已经定义出来了，就需要去使用、遵守它们。如 Ant Design 官方所说，在它们的最底层是设计的价值观。设计原则体现的是设计的价值观，而模式则体现了设计原则，它们都是设计语言的一部分。受限于笔者的能力，建议有意向往这方面发展的前端开发人员，寻找更专业的书籍来学习。

7.2.2　色彩

颜色是设计指南中最重要的部分之一。打开一个网站，映入眼帘的便是一个又一个的色块，如顶部导航栏的颜色。这种设计模式相当常见，如 GitHub 顶部的黑色导航，Overflow 顶部的白色导航。回忆一下常用的 App，每个应用都有自己的主色调，支付宝是蓝色的、微信是绿色的，淘宝是橙色的。

它们时刻在提醒着我们，选择一个好的颜色的重要性。一家互联网公司的主要（有影响力）的产品就只有那么几个，应用的颜色便也和公司、组织的形象相关联。若是组织与应用的色调不一致，那么会损坏品牌的形象。

在 Web 应用中，由于黑底白字不能突出重要的内容，所以需要一系列的颜色来创建用户友好的界面。在这些颜色中，通常都会做如下分类：

◎　主题色，又可以称为品牌色，用于体现产品的特性及宣传时使用。

◎　功能色，用来展示数据和状态，以及提醒用户。在 Material Design 中则被称为次主题色。

◎　中性色，用于常规的页面显示和过渡，通常是浅色和深色的变种，如白色和灰色。

如在 Bootstrap 中定义了几种类型的颜色，其对应了上述三部分中的功能色+主题色：

◎　primary（主题色）。

◎　secondary（次主题色）。

◎　success（成功色）。

◎　danger（危险色）。

◎　warning （警示色）。

◎ info（提示色）。

不过，要从那么多颜色中选择自己的主题色，不是一件容易的事。Web 应用的色彩通常使用 hex（16 进制）来表示，共有 6 位，累计有 16,777,216 种颜色。早期，为了保证网页的颜色，能够在不同的平台下显示效果一致，出现了 216 种 Web 安全颜色。现在基本不存在这种问题，可以放心地选择合适的颜色。幸运的是，在一些设计系统中，会提供相应的基础色板，如 Material Design，在那之上可以创建出适合自己的色彩。如图 7-3 所示是基础色板。

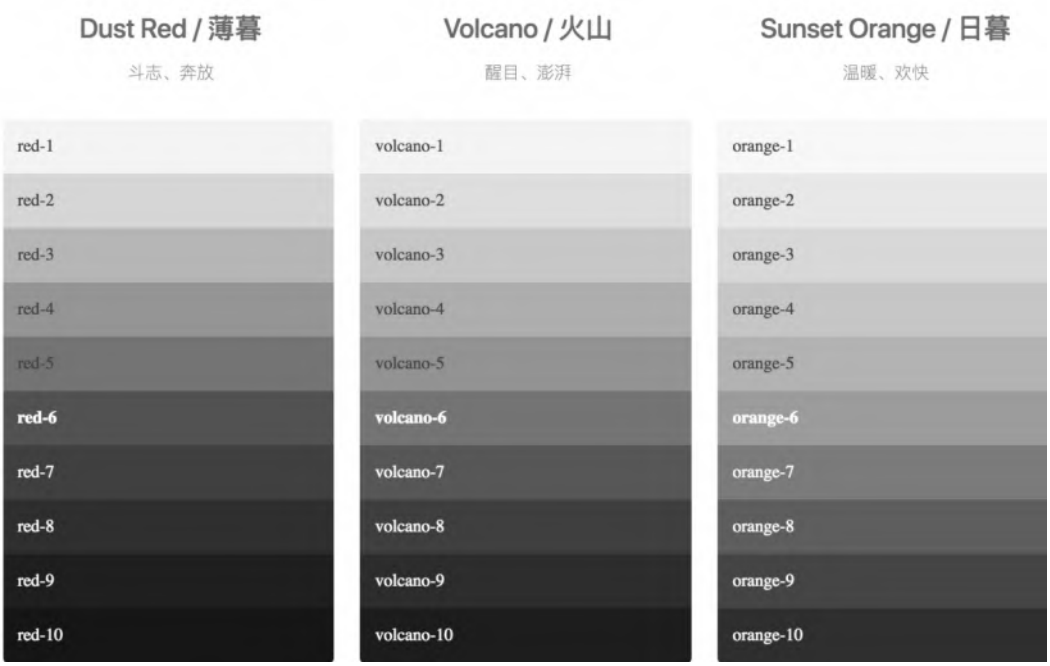

图 7-3

此时，还需要对色彩进行一定的命名，以方便不同角色进行沟通。对于前端开发人员来说，色彩采用变量来存储，结合 CSS 预处理器如 SCSS 定义全局的变量，即可随意使用及替换。最后，将选定完的色彩及其名称、值，以某种形式展现出来，如图 7-4 所示的网页上的色彩：

在如图 7-4 所示的网页里，我们所选定的颜色以大的色块在网页上展示，还有其对应在前端代码中的色值及变量名。无论设计人员还是开发人员，只需要单击相应的颜色，便可以自动拷贝对应的色值，应用到设计或应用中。

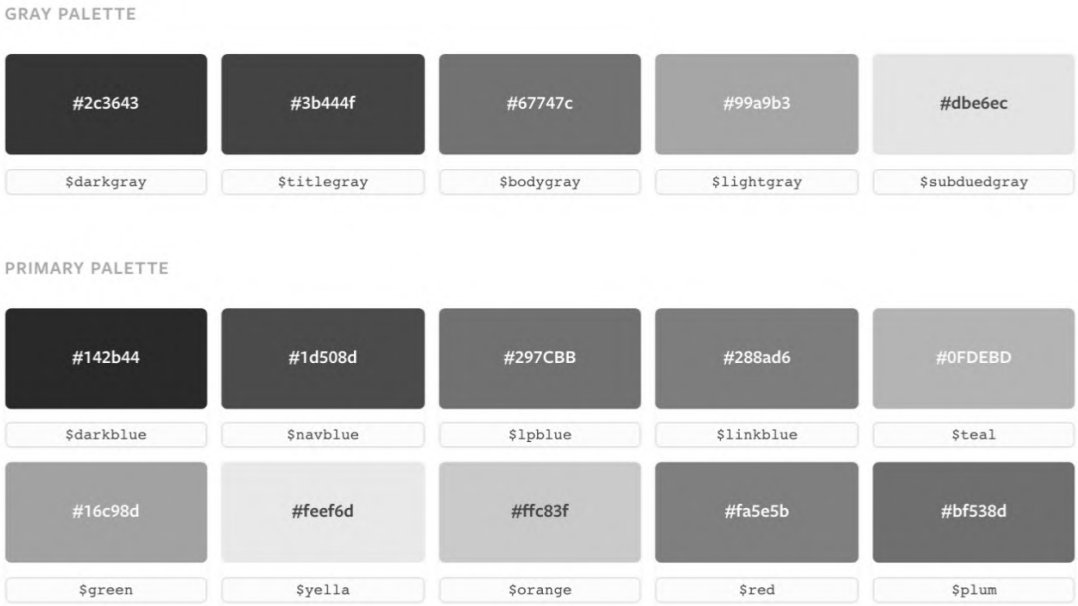

图 7-4

7.2.3　文字排印

我们细看网页时，便会发现页面上的那些大字体的区块比一般的文字更加显眼。比如在新闻、资讯、博客类的站点中，页面的标题总会比内容更先吸引眼球。诸如 Google 之类的搜索引擎，在收录页面的时候，也会注意页面 h1、h2、h3 的使用，级别越高的标题越有可能被收录到搜索的结果页。

对于字体而言，在开发应用时要考虑的因素有字体大小、字体颜色、行高（Line-Height）、字重（Line-Height）及字体家族（Font-Family）。对于风格指南而言，我们更关注于选择大小合适的字体。字体大小所针对的是常用的标题 h1～h6，以及普通的段落（p）。从 h6 到 h1，它们以一定的字节在不断地变大。在早期的 Windows 系统里，某些字没有 13px 大小的字体，因而多数采用的是偶数。这种奇怪的习惯也被保留到今天，因而多数字体的大小是偶数。通常，设计人员在大于 14px 的 Font-Size 中选择合适的大小。

宋体、楷体和黑体，是日常开发中最常接触的中文字体类型，其他中文字体也会有一些，但并不常用。在不同的系统上，字体使用情况如下所示。

◎ macOS，苹方简体：PingFang SC。

◎ Windows，微软雅黑：Microsoft YaHei。

◎ Linux，开源字体文泉驿微米黑：WenQuanYi Micro Hei。

对应的，我们的全局 font-family 设置可能就是这样的：

```
    body {
      font-family:-apple-system,BlinkMacSystemFont,Helvetica Neue,PingFang SC,
Microsoft YaHei,Source Han Sans SC,Noto Sans CJK SC, WenQuanYi Micro Hei,
sans-serif;
    }
```

当不存在字体时，一直往下找合适的，直至系统默认的无衬线体。如果想要突出设计感，那么就要考虑使用一些个性化的字体。中文字体因为体积大的缘故，通常不会从远程加载。而对于英文字体，则因为体积小所以在这方面比较容易。随着近几年来网络速度的不断加快和网络资费的下降，中文字体也变得可行。在使用网络字体时，务必要对相关的版权有所了解。或者购买相应的字体，或者使用系统自带的字体。

英语字体的数量比较少，大写字母和小写字母一共 52 个字，设计出一套完整的字体比较容易。而对于中文字体来说，设计字体就不是一件容易的事。在 1988 年出版的《现代汉语常用字表》中的常用字有 2500 个，次常用字有 1000 个，设计一套字体最少要几个月的时间。

7.2.4 布局

在项目开发中，布局是一个值得着重考虑的因素，合理的页面布局应该是符合规范化的。所谓的规范化，即这些布局是能用数值化体验的。数值能友好地规范设计人员设计的页面，也有助于避免产生复杂的、难以维护的前端代码。

开发一些简单的前端页面时，我们会选择 Bootstrap 作为 UI 库来使用。除了它本身提供的 UI 功能，它还有自己的布局系统。选择 Bootstrap 也许算得上是一个“平庸”的选择，但是选择它并不会导致失败。有时像 Bootstrap 这样的框架又显得有些臃肿——它太庞大了，修改东西不方便。轻量级的 CSS 框架如 Skeleton、Milligram 往往更适合我们使用，它们只有基本的布局和简单的样式。

样式是我们最需要修改的内容，而布局则往往是通用的内容。就当前而言，主要的布

局仍然是固定布局，如栅格布局、流体布局等，而基于 Flex 布局则是未来的一种趋势。由于篇幅所限，这里的架构以最常使用的栅格布局、未来布局和 Flex 布局为主。

1. 固定布局：栅格布局

栅格（Grid）设计有很多别称：网格设计系统、标准尺寸系统、程序版面设计、瑞士平面设计风格、国际主义平面设计风格，等等。

在平面设计中，栅格是一种由一系列用于组织内容的相交直线（垂直的、水平的）组成的结构（通常是二维的），它广泛应用于打印设计中的设计布局和内容结构。在网页设计中，它是一种用于快速创建一致的布局和有效地使用 HTML 和 CSS 的方法。——维基百科

栅格系统的参数根据项目的实际情况，尽量建立 10 的倍数或 8 的倍数（Google Material Desig 推荐）。从某种意义上说，它是设计人员的设计与开发人员的数值化相结合的一种产物。既能满足设计人员对于数值的要求，又能让开发人员快速使用。

根据不同的设计有一些栅格系统的实现，如 960 栅格系统，旨在通过提供基于 960px 宽度的常用尺寸来简化 Web 开发的工作流程，它能提供 12 列和 16 列的布局设计。随着显示设备的不断更新，又出现了更大的栅格系统，如 1200 栅格系统。与此同时，为了应对小屏设备，它大量地使用了我们在第 6 章中提到的媒体查询（Media Queries）。

这样的布局方式实现起来很方便。下面是 Skeleton CSS 框架中栅格系统的部分代码示例：

```
/*  示例位于：chapter07/grid-layout/index.html */
.one.columns {
    width: 4.66666666667%;
}

...

.twelve.columns {
  width: 100%;
  margin-left: 0;
}
```

从上面的代码中可以看出 Skeleton 框架的栅格系统是一个 12 列的布局。如图 7-5 所示是该 CSS 框架的栅格示例。

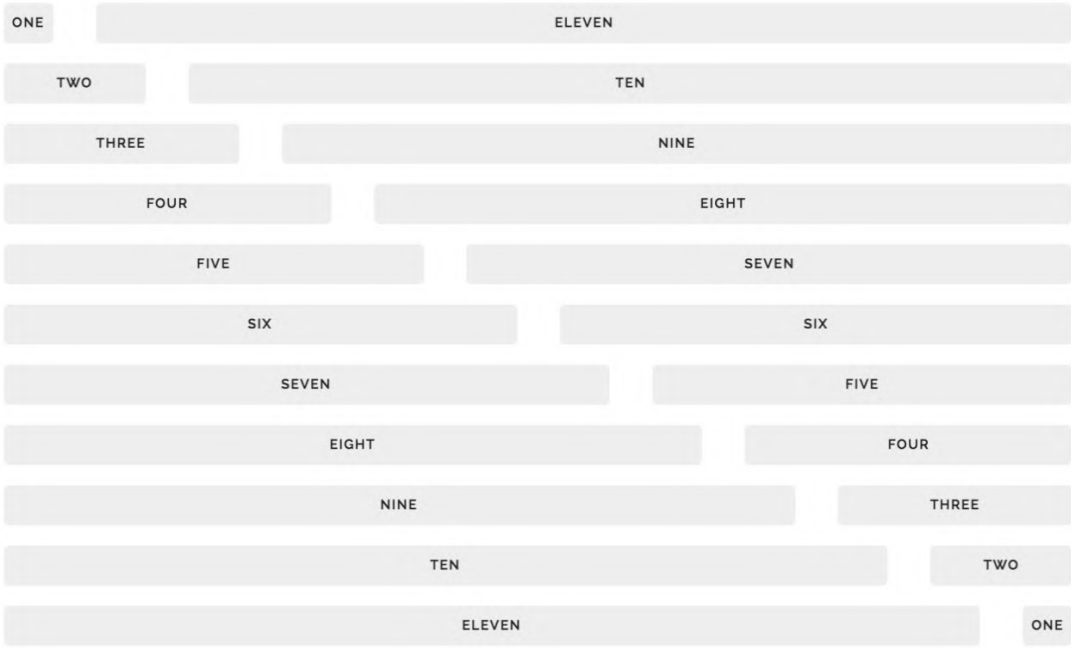

图 7-5

从图 7-5 中可以看到实际某个区块的可能宽度，即它应该满足上述 12 列中某一列的宽度。而在 Element UI 上则采用了 24 栅格系统，即将区域进行 24 等分，它能应对更高的分辨率带来的显示问题。列数越多，带来的设计可能性——我们可选择元素区块的大小就越多。缺点是维护成本高。

从上述一系列的例子中，我们可以看到栅格系统可以轻松应对左右布局的页面。然而，对于上下布局的页面来说，栅格系统基本上很难实现，或者说现有的 CSS 布局很难完成这样的功能。在真实的业务开发中，上下对齐的元素很常见。既然现有的布局不能满足我们，那么我们就设计一个新的布局吧。

于是，出现了新的 Flexbox 布局，它的定义为：如何在 CSS 中进行布局。

2. Flex 布局

CSS3 弹性盒子（又称为 Flexible Box 或 Flexbox）布局是 CSS 的模块之一，定义了一种针对用户界面设计而优化的 CSS 盒子模型。在弹性布局模型中，弹性容器的子元素可以在任何方向上排布，也可以“弹性伸缩”其尺寸。这样，既可以增加尺寸以填满未使用的

空间，也可以收缩尺寸以避免父元素溢出。子元素的水平对齐和垂直对齐操控起来都很方便。

与常规的布局相比，Flex 布局对于设计师来说更好用，对开发人员来说亦是如此。Flex 布局可以在各个方向上进行布局，并且能以弹性的尺寸来适应显示空间。Flex 布局具有以下特点：

◎　在任何流动的方向上（上、下、左、右）都能进行良好的布局。

◎　能以逆序或者任意顺序排列布局（即视觉顺序可以独立于源和语音顺序）。

◎　可以线性地沿着主轴或侧轴换行排列。

◎　能以弹性的方式在任意容器中伸缩大小。

◎　可以与容器对齐，或彼此对齐。

◎　可以沿主轴动态折叠或不折叠，同时保留容器的交叉大小。

如图 7-6 所示是 MDN 文档中关于弹性盒布局的介绍。

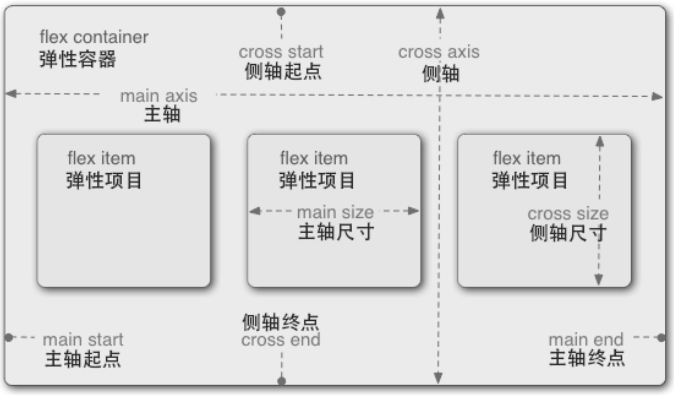

图 7-6

从图 7-6 中可以看到，我们可以在上下左右轻松地进行布局。其中的一些概念解释如下：

◎　弹性容器（Flex Container），包含弹性项目的父元素。通过设置 display 属性的值为 flex 或 inline-flex 来定义弹性容器。

◎　弹性项目（Flex Item），弹性容器的每个子元素都称为弹性项目。弹性容器包含的文本将被包覆成匿名弹性单元。

- ◎ 轴（Axis），每个弹性框的布局包含两个轴。弹性项目沿其依次排列的那根轴称为主轴（Main Axis），垂直于主轴的那根轴称为侧轴(Cross Axis）。
- ◎ 方向（Direction），弹性容器的主轴起点（Main Start)/主轴终点（Main End）和侧轴起点（Cross Start）/侧轴终点（Cross End）描述了弹性项目排布的起点与终点。
- ◎ 行（Line），根据 flex-wrap 属性，弹性项目可以排布在单个行或者多个行中。此属性控制侧轴的方向和新行排列的方向。
- ◎ 尺寸（Dimension），根据弹性容器的主轴与侧轴，弹性项目中对应主轴的宽和高称为主轴尺寸（Main Size），对应侧轴的宽和高称为侧轴尺寸（Cross Size）。

更详细的内容可以参见 MDN 相关的文档[1]。由于篇幅所限，我们就不详细展示 Flex 布局的相关细节了。

有了这样的布局系统后，我们不再担心上下布局的问题。尽管 Flex 布局已经很方便了，但是在单页面应用中独立维护 CSS 并不是一件容易的事。因此，我们不得不考虑兼容传统的栅格布局。让我们看一下使用 Flex 布局来实现一个类似栅格系统的示例：

```
/* 示例位于：chapter07/flex-layout/index.html */
.row {
  display: flex;
  flex-direction: column;
  padding: 0;
  width: 100%;
}
.row .column.column-10 {
  flex: 0 0 10%;
  max-width: 10%;
}

.row .column.column-20 {
  flex: 0 0 20%;
  max-width: 20%;
}
...
```

[1] [^flexbox]

在.row 选择器中，我们通过 display:flex 定义了一个弹性容器，flex-direction: column 则是定义了显示的方式，即以“列”（Column）显示。随后在与 column 相关的选择器里，我们同样通过定义宽度的方式来进行。以.column-10 为例，这里的 flex: 0 0 10%分别对应了扩展比例 flex-grow、收缩比例 flex-shrink，伸缩基准值 flex-basis。代码中将扩展比例、收缩比例设置为 0，将伸缩的基准值设置为 20%，同时设置最大宽度为 20%。这样，我们既定义了它在页面上的显示方式，又设置了它的宽度。

在一些前端框架里，有一些更好的实现方法，可以用于创建 Flex 布局。如在 Angular 框架里，由官方团队提供的 Flex-Layout 可以在 HTML 中提供与 Flexbox 相关的支持。它以编写 HTML 的方式来替代 CSS 的相关内容：

```
<div fxLayout="row" fxLayout.xs="column"></div>
```

以属性的方式来传递布局，可以更方便地对样式进行控制，如是否显示元素、对某个 flex 属性的值进行修改，等等。下面代码是对应的显示、隐藏示例：

```
<div fxShow [fxShow.xs]="isVisibleOnMobile()"></div>
```

虽然 Flex 布局能提供更好的布局方式，以更便捷的方式对页面进行排版，但是其浏览器的支持程度不是很乐观。如在 IE9 及之前的浏览器上都是不支持的，在 IE11 浏览器上也只是部分支持。更详细的支持细节可以参见：https://caniuse.com/#search=flex。

7.2.5　组件

对于设计人员来说，风格指南的组件只是在设计时使用的——从某个统一的组件库中复制出一个组件的样式，再粘贴到具体的页面设计中。对于开发人员来说，早期的风格指南中的组件，更多地使用于 CSS 上。单击相应的组件便可以复制相应的 CSS，或相应的 class，然后可以直接粘贴到代码中。

前端开发人员可以从这个统一的风格指南中获益，只需要复制、粘贴便可轻松地完成任务，其中麻烦和困难的地方就是创建组件和模式库。

随着模板库的发展，它变得越来越强大。考虑到这并不是风格指南的主要内容，我们将在 7.2 节进行详细的介绍。

7.2.6 文档及其他

除上述重要的内容外，对于设计指南来说，我们还要考虑的因素是文档。在文档中我们要指明：

◎ 设计人员如何使用设计指南。

◎ 开发人员如何维护设计指南。

◎ 如何在项目中使用设计指南。

◎ 常见问题的解决方式。

……

此外，风格指南中还应该包含一系列的设计细节和规范：

◎ 使用图片规范，不同位置图片的大小，一般采用什么格式的图片，等等。

◎ 留白间距，诸如 padding 默认的是 n*4px，即对应的 4 的倍数。

◎ 统一的圆角大小。

……

由于笔者更偏向于开发者的视角，对于设计方面的研究并没有那么深入，有需要的读者可以向更专业的社区和人士寻求帮助，比如可以读一下《UX 最佳实践：提高用户体验影响力的艺术》。

7.2.7 维护风格指南

注：对于搭建风格指南有兴趣的读者，可以直接阅读本书提供的相关代码，代码位于 chapter07/designsystem。相关的搭建步骤编写在项目的 README.md 文件中。

风格指南不是创建出来就能坐享其成的，而是需要持续不断地改进的。对于项目周期长的应用来说，其风格指南在周期里会不断地演进，直至能满足现有的设计。因此，在开发应用和风格指南的整个周期里，都需要开发人员和设计人员在改造设计指南上不断地投入精力。哪怕一个细微的变化，也要及时地更新。

有了设计指南，开发人员不用再重复地创造颜色不一致的按钮、方向不一样的动画效

果、大小差距比较大的标题等，只要使用统一的 class 就行了。设计与代码不同，设计多以主观为主，很难制定相应的规范。但是，设计人员也会从中受益，他们花费更少的时间在实现高保真设计上，并且关注于创建出更好的用户体验。设计人员不再纠结于使用什么颜色，只需要从色板上选择合适的那个即可；也不再纠结于某一元素的 px 大小，只需要按照规范实施即可。

前端应用随着业务的发展变得越来越复杂，单纯的风格指南越来越难以满足前端框架的需求。前端的组件不再是使用 CSS 和 HTML 就能完成的，它也需要编写一定的 JavaScript 才能使用，框架也从 CSS 库变成了真正的组件库（模式库）。

7.3　重用：模式库

模式库与面向设计人员为主的风格指南略有不同的是，它是一个面向开发者视角的组件库、代码集。其作用是，帮助开发人员创建易于维护的代码库。模式库具有一系列的优点：

- ◎ 开发效率。只需要通过参数便可以复用先前编写的代码，而不需要重复写代码。
- ◎ 一致化。减少了重复编写样式和组件带来的不一致的问题。
- ◎ 可维护性。通过重用与抽象减少了重复代码的出现率，代码量的下降也进一步降低了系统维护难度。

模式库和组件库，是一个容易混淆的概念，它们是包含的关系。从名称上来说，模式库包含了项目、应用程序中的所有可重用元素，如组件、通用代码等；而组件库只包含应用程序中与组件相关的代码。这一点十分有趣，在社区上我们看到的组件库多数是以组件为核心的 UI 库，而在项目内部落地的时候，为了方便维护，组件库中不仅包含了组件库，还带有一些常用的与组件相关的处理逻辑。

不同项目的可重用代码往往与业务场景相关，因此本节就不展开相关内容的讨论了，我们将以组件为主来介绍模式库。

7.3.1 组件库

风格指南关注的是元素在视觉上呈现的方式，其更像是一个与品牌设计相关的指南。而组件库则更多地关注与组件相关的内容，它们如何交互、何时使用，并且除了设计还包含代码。无论是网站、移动应用还是桌面应用，它都将包含由诸多小的元素构建出来的页面。组件库是构成应用程序的基础模块，它收集所有元素中可重复使用的部分，以确保所有元素一起工作，并保持设计的一致性。

在第 6 章中，我们已经讨论了如何选择组件库（UI 库），这里我们将进一步讨论如何创建一个组件库。

1. 创建组件库

从头创建一个组件库是一项复杂的工作，要考虑的因素有构建、发布、测试等一系列的环节。因此，寻找一个已有的、不复杂的组件库，往往是一个更好的选择。笔者经历的这些项目都是基于开源项目来构建组件库的：从中删除不需要的组件，添加自己的组件；修改包名、组件名，在项目内部发布——由于项目的版权限制，某些组件库只能在内部发布。

在这个过程中，我们还需要考虑如下一系列的因素：

- 组件提供怎样的方式？选择一次性引入所有组件的包，还是使用某个组件再引入某个组件？
- 组件如何发布？组织内部包是否有相应的发布方式？是否直接使用 Git 服务器作为包中心？
- 复杂的组件是否直接使用外部组件，而非自己开发？
- 对于复杂的组件，是否拆分成一个或多个组件，以降低维护成本？
- 组件的反馈渠道。如何提供对其他项目的支持？遇到问题时如何处理，比如 GitHub 的管理方式是什么？
- 组件的发布策略。是定期发布版本，还是以语义化版本的方式发布？

组件库的创建过程和创建应用程序的过程是类似的，但是与创建应用程序相比要更加复杂，具体步骤如下：

（1）创建脚本架。它可以从框架自带的工具生成，或是从其组件库修改。

（2）第一个组件。创建一个基本的组件，引入项目上，进行脚手架测试。

（3）进行发布测试。选择合适的发布策略，发布组件库，提供快捷的使用方式。

（4）创建组件的文档中心。可以通过与 JSDoc 类似的方式从代码中生成文档，也可以用更专业的编写方式。

（5）提供组件示例。创建一个专门的网站，来提供组件交互访问。

（6）持续改进。添加新的组件，修复 bug，并不断优化组件库。

与创建组件库相比，创建组件是一个相对简单的工作。创建组件的通用模式是，将接口分解为可重用的构建块，对它们进行排列、分组、命名，并在它们之间建立规则。此外，编写所有组件的概述、使用指南、示例代码，以便整个团队在构建和设计时使用。

在创建组件的过程中，最复杂、最让人头疼的一步恐怕就是命名了，好在多数组件都有业内通用的命名方式，尽管不同的组件命名方式略有差异，但都是相差不多的。

2. 如何维护组件库

组件的质量和数量是通过两种方式来提升的：需求、新的 Bug。能让一个组件变得更健壮的方式也在于此，从使用者上接收反馈。接收反馈的方式多种多样，从早期的邮件群组、非专业的论坛，到专业的敏捷工具如 Jira，或 GitHub、GitLab 自带的 issues 管理工具，还有 QQ 群、微信群都有这样的例子。

每种方式都有各自的优势与劣势，但是它们都提供了一个有效的反馈渠道。不论是来自哪种渠道的反馈，看到的第一时间应该及时地响应。尤其对于开源软件来说，哪怕说一句“收到问题，得花时间解决”，对于使用者来说也是一颗定心丸。使用者知道这个项目还是有希望的，一旦我们解决了这个问题，使用者就会觉得这个组件库是靠谱的。

解决了反馈的问题之后，我们就拥有了一定数量的组件，开始着重于提供这些组件的质量。编写测试，是一种有效提高质量的方式，足够的测试覆盖率，也更能提高使用者的信心——但是它也为开发者带来了一定的负担。此外，我们可以借助一些工具来提供代码的质量。

在组织内部发布组件库时，则可以使用开源的 Jenkins CI 来搭建持续集成、开源的 SonarQube 来对代码质量进行管理，还有私有化的 NPM 包服务器等。

对于那些在 GitHub 上发布项目的组件库来说，有各种各样的工具能使一部分维护工作实现自动化，例如 GitHub 官方推荐的两个工具：

◎ 自动化版本发布的 semantic-release。

◎ 自动复查代码工具的 Danger。

此外，还有关注于质量改善方式的工具：免费的持续集成工具 Travis CI、代码质量工具 Code Climate。自动化版本的发布除了依赖于各种自动化构建脚本，还需要有一个合适的发布策略。特别是在为下游的开发人员提供脚手工具时，这一点更为重要，需要有渠道来告诉这些用户：我们的代码库更新了，它解决了哪些旧的 Bug，实现了哪些新的功能。

组件库和风格指南采用一样的方式，提供一个网站，供开发人员查询及使用。这个网站相当于一份文档，并且是一份可交互的文档，除了说明组件的展示样式，它还包含了组件的用法。另外，我们可以修改这些代码，并直接在线运行代码，查看相应的效果。

和之前不同的是，网站并不是组件库的重点，组件库的重点是优先提供一个可用的 UI 组件。

7.3.2 组件类型

在创建这些组件的过程中，我们可以进一步地对组件进行分类。分类的目的是，帮助我们更好地对组件的目的进行把握，也方便开发人员寻找合适的组件。比如，在 Material Design 中将组件分为以下几类：

◎ 表单控制类，包含自动完成、输入框、单选框、选择器、滑动输入条等。

◎ 导航类，包含菜单、侧边导航、工具栏等。

◎ 布局类，包含卡片、标签页、树型选择器、栅格列表、步骤条等。

◎ 按钮和指示器类，包含按钮、图标、进度条等。

◎ 弹框及对话框类，包含底部弹出框、对话框、提示框等。

◎ 数据表格类，包含分页器、排序表单等。

上述组件中，有一些比较复杂的组件，这些复杂的组件不算是简单的组件。依笔者的观点来看，以组件的架构来进行分类，也是一个不错的方式：

◎　基础 UI 组件。这是最小化的组件，它们不依赖于其他组件。

◎　复合组件。由多个组件组成的组件，它们依赖于现有的组件。

◎　业务组件。带有业务功能的大量重复使用的组件。

三者是按照复杂度和业务的相关性进行分类的。

1. 基础 UI 组件

基础 UI 组件是最小的 UI 组件，它不能再往下拆分成更小的组件。它作为页面中最少的元素而存在，比如按钮、下拉菜单、对话框等。其中大部分是对原生 Web 元素的封装，例如：<input>、<select>、<button>，它们以简单的形式存在。

在创建基础组件的过程中，要遵循一个基本原则：基础组件是独立存在的。它们可以共享配置，但是不能相互依赖，依赖意味着它不是基础组件。其实维护这些基础组件也不是一件容易的事。下面是 Ant Design 中 Button 的参数列表：

```
export interface BaseButtonProps {
  type?: ButtonType;
  icon?: string;
  shape?: ButtonShape;
  size?: ButtonSize;
  loading?: boolean | { delay?: number };
  prefixCls?: string;
  className?: string;
  ghost?: boolean;
  block?: boolean;
  children?: React.ReactNode;
}
```

每当我们想支持某个属性可配置时，为了复用代码，需要进一步添加参数。由于参数在某种程度上也造成后期维护的困难，因此让每个组件都独立，可以进一步降低修改代码的难度。

在基础组件里，字体图标（Icon Font）相对比较复杂。字体图标能代替图片是因为它具有一系列的优势：矢量的无限缩放特性、像字体一样灵活地进行样式修改、体积更少，等等。它复杂的原因在于，为了定制图标，我们需要创建一个新的独立的项目来完成相关的自动化工作。

如果项目的团队规模较少、需要添加的字体也少，那么我们可能直接使用现有的 Icon 库，例如 font-awesome。也可以在现有 Icon 库的基础上，再去定制自己的 Icon 库，并去除一些不需要的组件。

如果团队具备一定的规模，那么可以自己开发字体图标库，图标库可以直接由矢量图片来生成，例如 svg2ttf。这也意味着我们需要不断地更新字体图标库。

2. 复合组件

复合组件是在多个基础的 UI 组件上的进一步结合。这个层级的组件相对来说比较复杂，原因是它将原本划分在业务部分的逻辑，抽象到了组件层面。原本需要开发人员编写的重复代码，现在放置在组件中实现，其复杂程度便从应用程序转移到组件层面。而原先复杂的组件，也进一步变得更加复杂。

如图 7-7 所示是 Material Design 的 Angular 版本的官方示例中的文档组件。

图 7-7

图 7-7 中的文档组件是一个复合组件，它由多个基础组件结合而成：

◎ 卡片组件，用于展示标题与源码。

◎　标签页组件，用于切换不同的代码类型。

◎　源码展示组件，一个不在 Material Design 中的带语法高亮的组件。

因为这个文档组件的重复使用，使得我们将其抽象成一个独立的组件，于是该组件变得相当有价值。

大部分复合组件，包含了一些复杂的组件，它们可能要花费几个月、甚至几年的时间，才能变成一个可供商业用户稳定使用的版本。复合组件包含以下几个部分：

◎　表格。表格往往带有复杂的交互，比如固定行、固定列、可编辑、虚拟滚动等。由于其数据量大，往往又对性能有很高的要求。

◎　图表。图表的门槛相对比较高，并且种类繁多，对于显示、交互的要求也高，所以图表相当复杂。

◎　富文本编辑器。几乎是最复杂的组件，其功能需求往往与 Word 进行对比，其代码量可能接近 Word 的数量级。

这一类组件功能都比较强大，实现这样一个组件，可能比实现一个项目要复杂得多。对于这些复杂的组件，要么直接使用第三方组件，要么以提供二次封装的方式进行，几乎不可能从头开始去实现这样的组件，即使是 BAT（百度、阿里巴巴、腾讯）这种量级的公司，也都是由专门的团队维护了多年。

复合组件在一定程度上会与业务组件有一定的交集，两者间的区别在于，是否混入了业务特定（领域特定）的代码。

3. 业务组件

业务组件是我们在实现业务功能的过程中抽象出来的组件，其作用是在应用中复用业务逻辑。当它们涉及一些更复杂的业务情形时，就要考虑是否将这些组件放入组件库中。还要考虑这个组件是否只会在我们的项目或系统中使用，别的项目或系统会不会用到？

按是否在特定领域（行业）使用，是否只能在当前项目中使用，业务组件可以分为如下两类。

第一类，应用相关组件，即组件与应用逻辑绑定的组件。对于登录页面而言，可能有多种形式，比如弹窗形式的登录、嵌入页面中的登录、独立的登录页面等。尽管使用的地方不同，但是它们在这个系统中是带有业务价值的。应用相关组件往往难以在其他系统中

使用，它需要内置以下内容：

- 用户名、密码的前端校验规则。
- 对后端错误规则的响应。
- 对后端响应的处理。
- 完成登录后的逻辑处理。

这一类组件便是与我们的业务密切相关的组件，它们往往是难以在其他系统中复用的，即不能通过修改参数就直接使用。对其他系统的应用来说，直接复制代码会是一个更好、更快的选择。

第二类，领域特定组件，可以用于特定行业的组件，但并非是通用的。比如，我们系统中有多处地方结合了搜索表单 + 表格，以便在选择时自动从后台获取数据，并显示在表格上。实现一个这样的功能，然后到处复制代码，显然不是一个合理的方式。抽取、提炼成一个组件，就会更加容易维护。这时，我们需要在多选框组件上绑定一个触发的事件，以便由业务代码实现相应的数据处理逻辑，然后在这个事件中返回对应的数据即可。

尽管我们已经进一步抽象了组件，但它还是被限制在该场景上。

抽取现有的页面元素，将其变成组件是一个复杂的工作。采用支持组件化的前端 MV* 框架是一种措施，它可以在需要的时候直接抽象成组件。但是，既然某部分的功能极有可能被重用，是否意味着我们在开发业务的时候就假设这个组件会被重用，这时是否需要进行预先的接口设计？

从某种意义上来说，多数情况下的提前设计是一种过度设计。如果一个组件在可见的未来（通常几星期内）会被再次使用，那么就有预留的必要。然而，由于两者的需求可能不一致，会导致预留的接口和所需要的接口不一致。如果时间太长，那么相关的需求可能就会被砍掉，也意味着我们可能会白白浪费时间。

此外，关于组件复用还有一个重要的议题：简单的功能，是否值得抽象成组件？比如，在创建表单的时候，有一个对应的信息填写表单。而在表单修改页里，修改的表单和创建的表单是差不多的，除了有一些不可修改的值、不显示的值，以及对应的提交 URL 不一样。于是，这个表单在表单数据量多的时候，就应该复用。而如果我们的表单只有两三个字段，那么不复用这个组件，往往是一个更好的选择。

对于数据的请求和处理的方式，决定了它是否能成为一个业务组件。比如，在组件中

统一进行相应的数据处理会减少开发人员的成本，但是该组件就变得不再通用，是一个专有的业务组件。而如果不在组件中处理这些数据，则需要将代码分散到项目中。此时，进行二次的封装，会是一种更好的方式。

7.3.3　隔离：二次封装

在一个成熟的项目里，自己开发组件是一种方式。另外一种方式，则是使用现成的组件。可是现成的组件往往也不一定能满足我们的需求。如果没有满足自己的“轮子”，那么就造一个。而许多时候，我们也不需要自己去创造一个“轮子”，直接在别人提供的组件上，添加自己需要的特性，便可以完成自己的功能。

这种二次封装的方式，并没有任何不妥。事实上，所有组件都是在基础的 HTML 组件上进行的二次封装。唯一需要注意的是，原有组件的开源协议（License）是否允许我们用在商业项目上，是否一定要开源。

即使我们直接使用第三方的 UI 库，也需要对 UI 组件进行封装，以隔离系统与组件间的依赖。未来，如果依赖的第三方组件库出现问题，我们只需要替换组件层，不需要进行大规模的代码修改。

组件的二次封装不外乎两种模式，可以直接使用设计模式来说明：

中介者模式，用一个中介对象（中介者）来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。对于一个新组件，可以不加修改地直接引用原来的组件。例如，我们定义了一个<aofe-button>组件，它提供了与原有组件一致的属性、输入和输出。在开发的时候，只需要引用原有的组件即可。未来即使再次替换成新的组件，也只需要引用<aofe-button>即可 。

装饰者模式，在不改变原类文件及不使用继承的情况下，动态地将责任附加到对象上，从而实现动态拓展一个对象的功能。这种设计模式为原有的组件提供了我们需要的功能的接口，如果某个组件的属性、输入、输出不符合 API 规范，那么我们就创建一个新的接口，来保证它符合规范。在未来，我们也不需要花费时间在修改代码上。我们仍然可以调用<aofe-button>这个组件，只是它支持了更多的输入和输出。

两种模式结合之下，我们就可以将原来的所有组件做一层代理。原本我们应该调用的组件接口是<react-xxx>，现在都改成了<aofe-xxx>。我们一旦决定对所有组件进行处理，就会产生新的模式。

釜底抽薪模式。如果有些组织想拥有自己的组件库，而当前又没有时间和精力，那么这种釜底抽薪的模式，倒也是颇为合适的。只需要在有余力的时候，一个接一个地替换原有的组件即可。随着时间的推移，我们便可以完成对所有组件的替换。而在这个过程中，它对于组件库的使用方是无感知的，使用方只需要升级相应的版本即可——前提是，API 兼容做得好。

在我们决定封装的那一刻，也决定了我们将编写一个新的组件库。

即使有了组件库，结合之前的风格指南，我们也难以设计出风格一致的系统。组件库存在的一个主要问题是，虽然它们提供了组件的概述，但是通常会给解释留下很多空间——组件可以以各种方式组合，看上去是一致的，但是从设计上来看是不一致的。因此，我们需要更多的东西来帮助我们构建出更一致的应用。

7.4 进阶：设计系统

设计系统是一组相互关联的设计模式与共同实践的结合，以连贯组织来达成数字产品的目的。模式是重复组合以创建界面的元素，如用户流、交互、按钮、文本、图标、颜色、排版、微观等。实践则是选择、创建、捕获、共享和使用这些模式的方法，特别是在团队中工作时。——出自 Alla Kholmatova 的 *Design Systems* 一书

有了风格指南，设计人员只需要从指南中寻找合适的元素，便可以拼凑出一个合适的页面。有了模式库，开发人员从组件库中寻找合适的组件，也能完成一个页面业务的开发。虽然页面风格看上去是一致的，但是它可能并不像是我们的设计。问题出在哪里呢？

在创建风格指南和模式库的过程中，我们很容易忽略一点——无论是风格指南还是模式库，它们都在创建通用的设计语言，以规范化工程实践和设计模式。模式库通过创建统一的组件、组合的组件命名、统一的使用方式来规范化组件的使用；风格指南通过创建统一的风格、规范的样式、合适的元素大小来规范化设计。但是这些模式、规范等，并没有明确地标明出来，在不同的设计人员的心里，可能有不同的规则、原则，而没有统一。团队越大则问题越突出，毕竟 1000 个读者心中有 1000 个哈姆雷特。于是，这些原则等便在设计系统上体现出来了。

设计系统从本质上来看是规则、约束和原则的集合，其分别由设计和代码实现。从名

称上来看，更偏向于设计为主的体系。从开发人员的角度来看，它是一个尝试解决设计人员的交互一致性的系统。从设计人员的角度来看，它是一个规则的合集。

于是设计系统做的第一件事，便是加入风格指南和模式库，然后明确提出了设计原则及模式。如图 7-8 所示，该图展示了设计系统与模式库、风格指南的关系。

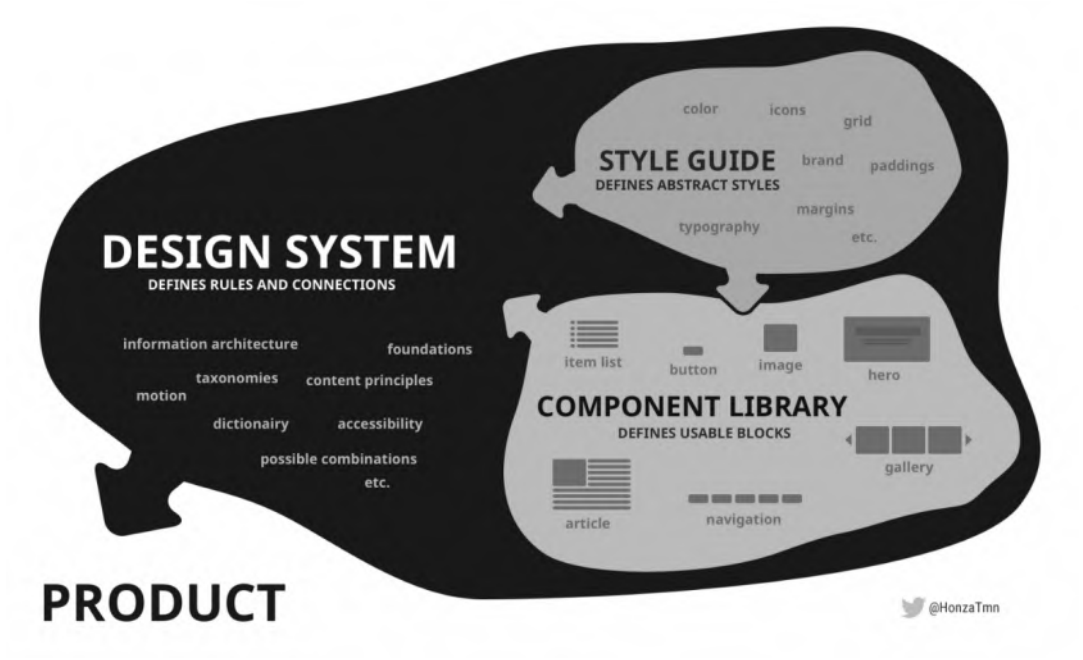

图 7-8

这时，设计系统包含了完整的设计标准、文档、原则及工具包（UI 模式和代码组件）。这一系列的原则和规范，包含了如下的一些指南：

◎　动作（Motion，即包含过滤和动画设计等）。

◎　无障碍。

◎　内容原则。

◎　信息架构。

……

设计系统是一个开发人员与设计人员协作的产物。建立一个设计系统，意味着建立一

个规则、一套规范，需要大家共同遵循这些规矩。设计人员拥有自己的与设计相关的术语，开发人员也拥有与自己开发相关的领域知识，在这个系统中需要融合这两者，建立出一个共享的、通用的语言。

有了先前的设计指南和模式库的开发经验，我们的关注点变成了如何去设定这些原则与模式。

7.4.1 设立原则，创建模式

“造轮子”，最简单的方式便是从模仿现有的“轮子”开始。如果我们计划去创建模式，那么不妨先看看别人的原则有哪些，它们又是如何创建的。

原则要保持适当的数量，才能保证容易被记住。过于复杂或者数量过多，会容易遗漏而导致要素的丢失。而过于简单，则起不了一点作用，即俗话说的，说了等于没说。而如果这些原则能缩写，它就变得相当友好，例如面向对象设计中的 SOLID（单一功能、开闭原则、里氏替换、接口隔离及依赖反转）原则。开发人员、设计人员能在第一印象里记起这五个字母及其背后的含义。

在 Ant Design 中一共出现了十个原则：亲密性、对齐、对比、重复、直截了当、简化交互、足不出户、提供邀请、巧用过渡、即时反映。其中的前四个原则参考了《写给大家看的设计书》。然而，Ant Design 多达十条的原则，会影响设计人员去运用它们。如果原则不能简化，那么它更多的就是负担。

Salesforce 的 Lightning 设计系统相对简单一些，它具有如下特点：

◎ 明晰。消除歧义。让人们看到、理解并充满信心地采取行动。

◎ 效率。简化和优化工作流程。智能地预料需求，以帮助人们更好、更智能、更快地工作。

◎ 一致性。对同一问题应用相同的解决方案，以创造熟悉度，并加强直觉。

◎ 美观。通过周到和优雅的工艺，来表现出对人们的关注的尊重。

如果没有后面对于文字的解释，那么仅仅只是前面的词组，我们往往也很难达成一致性意见。一个不错的例子，便是 Material Design 设计系统中的原则：

◎ 配目、图形、精心设计（Bold，Graphic，Intentional）。

◎　动作来表达隐喻（Motion Provides Meaning）。

◎　灵活的根基（Flexible Foundation）。

◎　跨平台（Cross-Platform）。

在这里，不得不提供 Material Design 的 Angular 组件提供的灵活性。Google 官方提供的 Angular 框架的 Material Design 实现，以一种灵活的方式提供了其组件库。在使用这个框架的时候，我们只需要引入所需要的组件（不改变原有的组件），就能引入 Material Design 的相关实现。

有了模仿设计原则的对象之后，我们便可以尝试去创建自己的设计原则。下面是专业的 UX 设计平台 UXPin 提出的创建设计原则的步骤：

（1）寻找产品类比。

（2）在产品类比中寻找设计原则。

（3）通过用户调研，让列表变得真实。

（4）建立价值主张。

（5）抽象原则。

对于新创建的团队来说，寻找自己的设计原则并不是一件容易的事。没有现有的产品，没有过去的设计，很难进入下一步的提取。好在那些要创建设计系统的组织，都是有了一定规模的设计团队，也有了一定的产品。在这种情况下，要梳理出属于自己的设计原则，并总结出来。

在有了设计原则之后，我们还需要总结出常见的一些设计模式。这些模式涉及一些常用的展示：色彩、数据录入、数据展示、空状态、布局、加载、本地化、字体、搜索、导航、消息、标记和风格等。在过去的设计指南中，这些往往只是一个组件、一个示例，现在则变成了一个个的原则。以 loading 组件为例，它可以有多种不同的形态：

◎　SVG 或者 Gif 的加载动画。

◎　展示加载比例的进度条。

◎　显示加载的占位图。

◎　系统相关的提示操作。

在我们设计出来的应用中，也会多多少少地使用其中的几种形式。为了统一 loading 组件的处理，应当明确地在设计系统的文档中指出什么是 loading，指出在哪些时候、使用哪种 loading 的展示方式，并且每种 loading 的展示方式都应该有相应的示例。一一整理一致、不一致的交互方式，便是在实践设计模式的过程中需要做的事情。

道理说了这么多，要做的事情这么多，但是如何才能规范化地实践呢？

7.4.2 原子设计

构建设计系统时，需要模块化的分层方法，并且需要能与模式库相结合。这时，我们便需要原子设计原则，它是由 Brad Frost 提出的能与设计师的设计系统概念相吻合、又能满足模式库的原则。原子设计是一个设计方法论，由五种不同的阶段组合而成，它们协同工作，以创建一个有层次、有计划的界面系统：

- ◎ 原子，即事物的基本组成部分。它是最小的单元，不能再往下细分，也就是基本的 HTML 标签，比如表单标签、输入、按钮。
- ◎ 分子，即由原子聚合而成的粒子。在 UI 设计中，分子是由几个基本的 HTML 标签组成的简单组织，如移动 Web 应用的底部导航栏。
- ◎ 有机体，是由分子、原子或其他有机体组成的相对复杂的 UI 组分 。它用于创建一些独立、可复用的内容，比如 header、footer 等。
- ◎ 模板，顾名思义就是整合前面的元素，构建整体的布局，它将组件在这个布局的上下文中结合到一起。比如，一个博客包含 header、footer 及博客内容，三者可以构建一个基本的外观。
- ◎ 页面。用真实的内容（数据）展示出来的最终产品，它还能测试设计系统的弹性。

如图 7-9 所示是基于以上五个阶段，构建出的一个设计系统的步骤。

上述模式也与我们构建一个组件化的前端应用架构颇为相似。

开发人员和设计人员，从基础组件的原子部分开始工作，积累底层的基础。然后，通过原子构造出一个更大的分子，也就是复合组件。再由复合组件组成独立内容，即组织。接着，我们通过不断地组合这些内容，便可以创建出页面模板。最后，我们编写的应用，只需要获取真实数据，渲染到模板上即可。

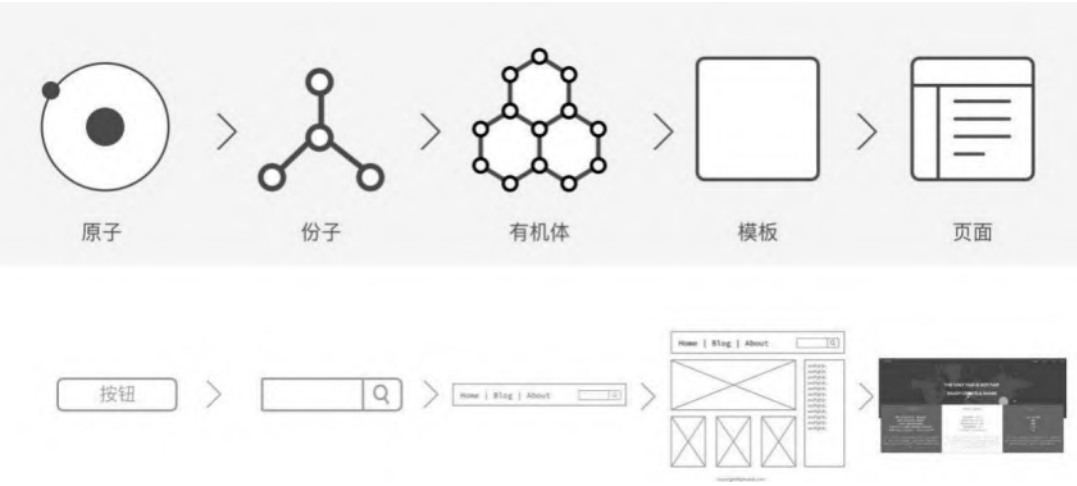

图 7-9

1. 原子

原子是网页构成的基本元素。在 Web 应用中，大规模地使用组件化的思想，应该来自于前端框架 React。它可以将一个大的用户界面的各个部分，各自封装到自己管理的独立系统中。

对于前端开发人员来说，设计系统的原子部分，便是模式库中的基础组件。由各种基础标签组成，如封装<input>标签的输入框架，封装<a>、<button>构成的按钮。对于设计人员来说，原子还包含基础的色调、字体排印等内容。正是这些基础元素，为后来构建出的页面打下了基础。每个原子组件都拥有自己的基础样式，它作为一个参考指南而存在。

2. 分子

按自然科学来看，分子是由原子构成的。它不是由基础的 UI 组件构建的组件，而是我们在先前提到的复合组件。一个输入框和一个搜索框，可以构建一个搜索框，它既是设计人员的组件，也是开发人员的组件。

与原子级别的组件相比，分子级别的组件才是日常开发中使用最多的。一个简单的原子往往很难代表具体的功能，但分子则可以被一眼看出可以用在什么场所。比如，从多个页面中拿出一个输入框，我们往往很难判断其出处。它可以来自表单的输入框，也可以来自搜索中的输入框，或者来自一个评论用的输入框。但是，如果它是一个搜索框，那么我

们就会知道，什么场景下需要使用它。

分子级别的组件，往往也需要在多个地方被重复使用。因此过于复杂的组件，也不太适合在这一级别出现。

3. 有机体

有机体是由原子及分子组成的相对复杂的 UI 构成物。如果只是从复合组件的定义来看，它与分子组件类似。从某种意义上来说，它与分子最大的区别在于，有机体级别的组件在设计的时候，体现了设计系统中的设计模式。

在 *Atomic Design* 一书中所举的例子是：网页的头部（header），它是包含几个具有自己独特的属性和功能的小界面。不同的网站 header 的设计往往是不同的。出于简洁考虑，有的应用的 header 会显得非常简单，隐藏一些不常用的功能。而有的设计，则会将 header 设计成带动画和过渡的，当我们打开页面的时候，看到的是完整的 header；而当我们往下滚动一段距离的时候，看到的是简洁的、修改的 header。所有的这些，都体现了设计系统中的设计模式。

从 header 这一组别的组件来看，这个结果更加显而易见。与 header 同样级别的组件，基本上是业务相关的组件，比如带过滤的搜索区域、列表页、footer 等。因此，在某种情况下，我们可以将有机体和业务组件画上等号。

4. 模板

原子设计的模板，和我们理解的模板是类似的。只是对于开发人员和设计人员来说，模板是两种不同的存在。对于设计人员来说，模板更多的是粗粒度的线框图，它用于展示有机体、分子、原子之间如何互相结合。而对于开发人员来说，模板则是一些接近完成的页面的编码，只是代码中的数据是仿造的，而非真实的数据。

无论如何，模板的目的都是为了提供一个可重用的模板，它可以方便地用于构建多个不同实例，可以方便后来的人快速地上手项目。新进入项目的人，可以愉快地 Ctrl + C、Ctrl + V，也可以参考着进行修改。顺带一提：在一些框架里，如 Angular，可以直接通过模板来生成用于页面中的代码。下面是适用于 Angular 的 Material Design 的导航生成的相关命令：

```
yarn ng generate @angular/material:material-nav --name myNav
```

然后，设计人员确认设计，便可以绘制出高保真图；开发人员编写真实的业务逻辑，并创建出真实的页面。

5. 页面

页面，是一个一个具有真实业务功能和真实数据的组件集合，是一个真实的模板，也是在日常的业务开发中，前端人员和设计人员的重点关注所在。当我们不曾拥有模板，而又需要展示模板的时候，页面就是最好的模板。它拥有最真实的模板数据，足以让人信服。

在实现页面的时候，分别关注于设计和代码的可维护性。之前的设计原则、模式库，理应毫无偏差地应用到页面上。编写代码的时候，考虑的是重用之前的组件库（模式库）。一旦现有的组件不能满足需求，就修改原有的组件，将其提取到模式库中。

前端代码是否好维护，以及模式库是否方便维护，会在实现页面的过程中体现出来。就好比是，前面我们说了一堆大道理，可等到落地的时候，发现无法落地。这种苦闷的现实，怕是多数人不想面对的。

7.4.3　维护与文档

目前，我们看到的设计系统（可以由互联网访问），几乎都是以网站的形式存在的。网站是一个快速、有效的文档展示方式，由于网站背后都是由源码管理系统和版本发布相结合的，所以不存在文档落后的问题。

设计人员除了在设计系统上体现组件和设计的原则，还应当提供一些相关的资源和文档：

- ◎ 搭建指引。包含诸如如何使用设计系统，如何修改设计系统等的内容。
- ◎ 内容指南。随着服务化设计等概念的展开，我们应当考虑添加内容指南等一系列应用相关的规则。
- ◎ 工具指南。向使用这个设计系统的成员，提供一些开发者工具、设计师工具等的指南。

……

由于篇幅所限，这里就不展开设计系统的创建。有兴趣的读者可以从 GitHub 上下载对应的代码，并按照 README.md 中的过程进行搭建。相关的资源，后续也会更新在

GitHub 上：

◎ 使用 Storybook 框架创建的示例，位于 chapter07/designsystem。

◎ 使用 Vue Design System 创建的示例，位于 chapter07/vue-designsystem。

设计系统与维护风格指南、模式库类似，一旦我们添加了新的组件，就需要即时地更新文档，即更新网站。如果模式库和设计系统（即包含模式库的 DEMO 部分）位于两个不同的代码仓库下，那么修改起来就不是一件容易的事。一种常见的方式是，将设计系统与模式库以某种方式集成到一个项目中——它可以以目录的形式存在，也可以以 git submodule 的形式存在。既方便前端开发人员测试新组件，又方便添加组件的 Demo 到设计系统中。这时，设计系统就是组件库的 Demo 工程。

在我们开发这个设计系统时，遇到的第一个难题是，选择哪个前端 MV*框架？如果模式库支持 Vue，那么我们的设计系统也应该使用 Vue 来开发。但是，当我们的组织内部存在多个前端框架的时候，我们是否也要提供多个模式库？

7.5 跨框架组件化

过去我们谈论前端的组件化架构时，通常指的是框架限制的组件化架构。而当拥有基础的 UI 组件库时，我们的架构则是基于 UI 组件库的组件化架构，两者间的不同在于共性的第一次提取。当我们在业务组件的基础上，对一些通用业务组件进行封装时，我们的架构则基于 UI 组件库和业务组件的组件化架构。不论是哪种方式，最后都限定于框架限制——将系统绑定在框架上。

对于团队的技术决策者来说，绑定框架的实现是一种冒险的做法。未来，这些都是风险，那么有没有可能将底层的 UI 组件库、复合组件和业务组件库通用呢？

7.5.1 框架间互相调用：Web Components

在笔者创作这本书的时候，先开发了一个简单的 Markdown 编辑器 Phodit，随后边写作，边完善编辑器。过程中，笔者有意无意地将编辑器中的功能，拆分成一个一个的小组件，每个小组件使用不同的技术构建，React、Angular、Stencil.js、原生 JavaScript，等等。如：

◎ Stencil.js + Web Components 来放置 Terminal 的关闭窗口。

◎ React.js 制作了左侧的树形文件树。

◎ Angular 6 完成了重命名文件的交互。

◎ sweetalert 用来做 Dialog 提醒。

……

编辑器仍然在开发中，这并不是最后的所有技术。引入这么多框架的“hello,world”，然后构建一个个简单的组件，也许是为了练习。虽然这么说，实际上 SimpleMDE 已经封装了 CodeMirror 的一系列 API，为了能快速用上自己的编辑器，笔者决定直接基于 SimpleMDE 来修改。而 SimpleMDE 并不能直接用在 Angular 等前端框架上，这也意味着，由于 Editor 的存在，我们不得不将页面撕裂成左侧菜单、Terminal 窗口栏、辅助栏、状态栏等几部分。

不同的框架开发方式不同，导致不同前端框架之间的组件不能一起使用。即使用 React 开发的组件，也很难直接用在 Angular 的应用上。同理，Angular 的框架也难以应用于 Vue 应用中。好在这些应用都可以开发成一个 Web Components（Web 组件），我们可以在其他框架里引用这些组件。

Angular 框架提供了一个 createCustomElement 的接口，可以直接将组件定义成 Web Components 组件。而在 React、Vue 框架中，则是可以支持引入这些 Web Components 组件，也可以引入其他框架的组件。不过直接使用 Angular 构建出来的组件，体积上稍微大一些。一个更合适的方式则是，使用第三方框架来构建 Web Components 组件，如 Stencil，然后在我们的框架、应用中引入这些组件。

关于 Angular、React 相互调用的 Demo，可以参考这里的代码：chapter07/wc-angular-demo，搭建指南也在相应的 README.md 文件上，这里就不详细展开叙述了。

7.5.2　跨平台模式库

有了上面的技术基础，我们就可以构建跨 UI 框架的组件库了，它可以解决在构建内部 UI 库时面对不同技术框架的问题。那么，为什么还没有这样的 UI 库？原因主要有两个：

技术不够成熟。现有的前端框架对于 Web Components 的支持并不是很好，比如，笔者尝试使用 React 时遇到一些问题。并且，既然前端框架都能支持构建出这样的组件，那

么也需要浏览器对于 Web Components 的支持。我们需要比如 custom-elements-es5-adapter.js 等的支持，而像 Polymer 这样的 Web Components 框架也需要 IE11+等浏览器的支持。

Web Components 技术重写组件不够好。是的，我们需要将之前使用 TypeScript、JSX 或者.Vue 编写的组件，换成更轻量级的框架。UI 框架中的很多要素，是我们在编写组件的时候不需要的——我们只在需要的时候引入我们所需要的组件即可。

一旦包含浏览器支持的基础设施得到了进一步的完善，我们就能开发出这样的跨平台组件库。

7.6 小结

组件化架构与 MV*架构是现今前端应用的基本架构，它们往往相互结合，以用于降低 Web 应用的复杂度——将应用中的部分复杂度，由应用程序转移向组件库（模式库）。

为此，本章从组件化架构入手，展开组件化背后的思想，及其设计相关的研究。从本章中我们会发现，组件与设计之间的关系是相辅相成的。风格指南、模式库和设计系统，都是开发人员和设计人员在合作过程中探索的产物。每种类型都各有特点：

- ◎ 风格指南，侧重于设计。其在设计的实现是定义 UI 规范，其在前端的实现是通过 CSS 框架来体现的。
- ◎ 模式库，侧重于前端开发。其重点是基础组件、复合组件及业务组件的相互结合。
- ◎ 设计系统，结合风格指南与模式库，加强了模式和原则的运用。

此外，我们还在各部分中介绍了如何去创建对应的风格指南、模式库及设计系统。通过这些实践和练习，我们能从中找到适合于大型组织的开发人员和设计人员的协作方式。

最后，我们介绍了如何通过 Web Components 在不同前端框架间相互调用组件，以证明跨框架组件的可能性，它可以进一步降低大型组织中多个框架下的组件开发成本。[1]

[1] 参考书籍：*Atomic Design.*

8

第 8 章

架构设计：前后端分离架构

在现今的前端开发方式里，单页面应用与前后端分离是主流的趋势。采用单页面应用框架，也就意味着前后端分离，以及一系列开发模式的改变——代码库分离、独立部署等。其中多多少少对于传统应用来说，是一些新的挑战。可一说到前后端分离，人们最大的痛楚莫过于对 API 的管理和维护：

- ◎ 前端找不到某个字段。原因可能是后端删除了该字段，或者进行了重命名。
- ◎ 某个返回值是错误的。原因可能是计算错误，或者是业务理解不一致。
- ◎ 前端获取不到任何数据。原因可能是后端误删了数据库，或者在重新部署应用。
- ◎ 状态码预期不一致。某个结果预期应该是服务端错误，却是用 200 的 HTTP 状态码返回的。

对于上述问题的解决，便是本章我们要讨论的内容之一。结合前后端分离及 API 管理相关的内容，本章将介绍如下内容：

◎ 前后端分离是如何协作进行的。

◎ 如何进行 API 的文档管理。

◎ 如何创建 API 的 MockServer。

◎ 使用测试来保证 API 是符合预期的。

◎ 采用 BFF 架构来提升前后端 API 的开发效率。

这些仍然是我们在每个项目开始时所要面临的挑战。在进入细致的讨论之前，我们先了解一下前后端分离应用是如何进行开发的。

8.1 前后端分离

前后端分离，即前后端各自作为一个半自治的技术独立团队，协作开发同一业务功能。统一中的不统一性，便是前后端分离的有趣之处。

8.1.1 为什么选择前后端分离

前端与后端之间，原本就是技术栈近乎独立的两端。只是在传统的 Web 应用中，使用后端的模板罢了。因此，即使前后端分离了，带来的技术挑战也不大。对于传统的技术团队来说，之所以存在大量的技术挑战是因为，由多页面应用转向了单页面应用。

前后端分离更多的是，带来工程、协作、沟通、测试上的挑战。同一个团队的人，如果共同实现前端逻辑和后端逻辑，那么就不存在这一类型的挑战。一旦分离开来，同一个团队就仿佛变成两个部门，相互间在斗争。后端开发人员认为：“前端不就是数据展示嘛！”，前端开发人员认为：“后端不就是 CRUD 嘛！”当然这只是个玩笑，我们不得不承认前端交互的复杂度，不得不承认后端并发并行的复杂度。

前后端分离的原因是多种多样的，移动优先（Mobile First）战略将单页面应用带到了移动领域，使得后端需要提供 RESTful 风格的 API，也进一步加剧了前后端分离；并且，知识的专业化使得传统的后端工程师，已经无法满足前端开发的要求。

前后端分离所面临的挑战是：除了提供一个 API（如 RESTful API 的方式），还要面临

API 管理带来的挑战。特别是在后端还没有准备好的情况下，需要去推进前端的开发。这个时候，前端还需要一个“勉强”可以用于开发的 API，并且在后端完成后，尽可能少地进行修改。

除了上述的一些缺点，它也带来了一系列的优点：

◎ 独立部署。前端应用可以独立运行在自己的服务器上，而不受后端上线计划的影响。

◎ 分清职责。后端将视图层（View）从系统架构中拆分出去，让系统变得更加简洁。

◎ 技术栈独立。分离之前，技术选型受一定限制，如模板引擎等。分离之后，只要保证 API 是一致的，前后端之间就会互不影响。

◎ 方便系统演进。一旦前后端都使用自己的技术栈，转换技术栈就变得相当容易。后端可以迁移到微服务，前端可以迁移到微前端架构。

◎ 提高效率（相对的）。对于复杂项目而言，拆分可以降低维护成本；而对于简单的项目而言，拆分则会提高维护成本。

无论如何，一旦选择了前后端分离，就意味着我们要熟悉相关的开发模式。

8.1.2　前后端分离的开发模式

在不同的软件开发模式下，前后端分离模式也略微有所差异。瀑布模式的前后端分离，仍然是预先制定 API 的文档，再进行联调。敏捷模式的前后端分离，则是一个业务一个 API，每个 API 单独集成。

两者之间，敏捷模式更能响应变化，瀑布模式更依赖于前期的设计能力。如果是团队内的项目，那么大抵采用的都是敏捷模式，是团队的成员所能接受的。可一旦对接第三方的 API 和服务，大家想要的就是瀑布模式，让第三方先把 API 测试好，再提供给团队使用。但是，有时第三方的 API 也是在开发中的，所以集成的过程可能会相当痛苦——停下手中的工作，及时地响应 API 变化。

无论哪种方式都需要从业务逻辑出发，从业务的角度出发，Web 应用的前后端是一个整体，它们无法独立运行。于是，我们的开发模式变成了下面这样，如图 8-1 所示。

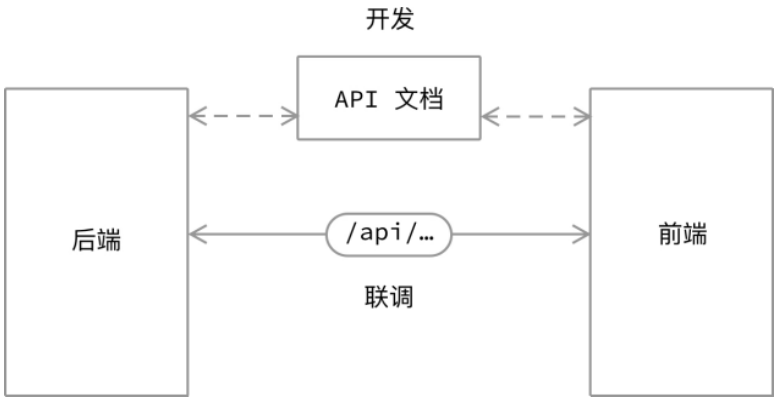

图 8-1

图中的 API 文档，可以是一个模拟后端接口的 JSON API，也可以是文档编写的 API。在这个过程中，我们要做的事情有：

（1）按业务的展示逻辑，确认出待展示的内容。

（2）前后端根据内容，一起细致化每个字段名，直至接口确认完毕。

（3）遇到对接第三接口时，需要往复进行第（2）步。

（4）当各种开发完成时，在测试环境进行集成。

（5）将完整的业务功能交给 QA 进行测试。

如果这几点，都能按我们的预期来进行，那就很完美。可实际上，并不会完全按上面的模式来进行。在开发的过程中，我们不得不面对 API 的变化，它可能来自于业务变化、性能影响或者接口限制。如果 API 发生变更时没有及时通知使用方，那么当应用上线时，事故就不可避免。对于大中型的组织，有测试人员能进行回归测试；小型团队，则往往只能听天由命了。

在进一步深入 API 管理之前，让我们先对前后端分离中的 API 有一个基础的认识。

8.1.3 前后端分离的 API 设计

由于 API 设计的话题，不是一小节、一大章能介绍得完的，有相当多的书籍在介绍相关的内容。这里我们只介绍与前后端分离相关的 RESTful API 设计和安全。当然，关于 RESTful API 和安全相关的书籍，又已经有一大堆了。API 并不是和前端开发人员没有任

何关系的。作为一个专业的前端开发人员，考虑后端接口对于前端的合理性，也是日常工作的一部分。后端 API 不合理、不符合规范，必然带来前端的额外工作量。

1. RESTful API

RESTful API，几乎是前后端分离的标准实践。值得注意的是，它并没有明确数据的格式是 JSON、XML，只是 JSON 格式更适合于前端开发。顺带一提，实际上很多自己宣称是 RESTful API 的后端服务，都不能符合 RESTful API 的所有要求——毕竟要实现按规范完成太难了。

虽然后端才是 API 的实现者，但是作为一个前端开发人员，我们还要确认接口的规范性，才能保证代码不出问题。下面是一个 HTTP Get 请求的示例：

```
this.http.get('https://aofe.phodal.com').subscribe(
  data => {
    //获取数据成功后，执行相关的逻辑
  },
  (err: HttpErrorResponse) => {
    //获取数据失败后，执行相关的错误处理逻辑
  })
```

如果在这个示例中，后端将一个返回的状态码 200 变成 404，那么逻辑就永远在错误的逻辑里。当然，这种不合理的状态码一般是不会存在的，通常我们要借助于 HttpInterceptor 进行全局统一的非法授权处理，即 401 处理。通过统一的非法授权处理，可以清除用户权限相关的显示。

总之，后端都应该提供靠谱的 RESTful API。而前端开发人员作为使用方，是确保后端 API 规范的又一个环节。为此，作为一个 Web 开发人员，我们还是应该了解一下相关的基本规范：

- 标准的 HTTP 动词（又称为 HTTP 请求方法）。GET、PUT、POST、DELETE、PATCH 等，每个动词的用法应该和它的行为一致。
- 状态码。20x、40x、50x 等常见的状态码，都应该正确地使用。
- 资源路径。RESTful API 中的 URL 用于代表资源，应该确保资源能遵循相关的规范，例如，/comments/1 返回第一条评论，/comments/返回所有的评论。

◎ 参数处理。如果存在大量的参数，那么我们就需要通过 GET 带查询字符串（Query String）的方式，或者 POST 带 body 的方式来进行传递。如果能统一，那么开发起来也就方便了。

此外，还有一系列和 RESTful 相关的细节。这里罗列的只是在日常工作中，前端开发人员经常接触的一些相关实践。除了与 API 相关的话题，另外一个有趣的关乎前后端的话题，便是安全。

2. API 与安全

安全，在哪都是一个重要的话题。对于前端开发来说，也存在一些与安全相关的内容。前端的安全措施能大大地拖延黑客的破解时间。从这种意义上来说，前端安全只处于一个“聊胜于无”的状态——真正的安全措施，都需要毫无保留地在后端实施。

依笔者过去的经验来看，前端有如下一些与 API 和安全相关的要素，是我们在每个项目中要考虑的。

◎ Token 管理。对于前端开发人员来说，无论是保存会话状态的 Cookie，还是无状态的 Token，都没有太大区别，无非是在管理方式上的区别。后端出于安全考虑，会有各种复杂的管理机制，常用的有超过时间跨度则过期、二次登录失效、多个客户端可以同时登录、Token 永不过期等。而对于前端来说，只是在遇到 401 未授权的时候进行相应的逻辑处理。

◎ 表单校验。表单是一个特别有意思的话题，因为所有的表单都是两端校验的，既需要前端校验，又需要后端校验。前端进行表单校验往往是出于用户体验的目的，提醒用户哪里做错了，需要重新填写。后端进行检验时默认前端是不可信的。后端开发人员应该确保后端的健壮性。稍显麻烦的一点在于，保证前后端校验逻辑的一致性。在测试人员进行测试的时候，会发现诸多前后端不一致的逻辑校验问题。

◎ 权限管理。从某种意义上来说，用户只能访问自己能访问的数据，这看上去又是一个与后端相关的话题。前端也有与之相关联的一部分，前端往往只是根据相关的角色和权限来展示现在的页面，至于权限则要在后端进行判断。与这些权限相关的处理，要么在路由跳转时进行判断，要么在页面初始化时进行判断。在复杂的权限管理系统里，它几乎充斥在每个页面中。

这些 API 看上去很平凡，但却是我们在实现业务的过程中不得不多次考虑、经常接触的问题。在开发相关业务时，时刻与后端对齐，同时还可能不断地修改 API。

3. 应对 API 变更

在实施前后端分离架构的过程中，最让人苦恼的莫过于 API 发生了变化。API 发生变化的原因很多，如业务变化、字段名出错、第三方接口不匹配等，但是这些都不重要——一个字段从 string 变成 number，对于前端的影响并不大。重要的是，当后台 API 发生变化时，前端却不知道或者没有人对此做出响应。例如，某一天你下班后，后台的同事告诉你，这个 API 变了。结果，第二天这事被忘记了，于是就造成了一个 bug。

API 变更是不可避免的，如果不修改 API，那么会带来更多问题。长痛不如短痛，如果将来需要修改，那么不如从现在开始修改。相比于后端进行的修改，早期的修改因为改动范围较小，修改的成本往往较低。合理的 API 修改，大部分前端开发人员都能接受。可是如果 API 反复修改，从 A 变成 B，又从 B 变成 A，那么就会遭到一番异议。

修改 API 时，对于前端来说，往往意味着字段的改变。字段需要从代码和模板里发生变化，修改对应的值。如果只是简单的字段，修改起来难度也不大，无非就是 data.name 和 data.username 的区别。如果修改的地方很多，或者是修改的 API 字段多，那么就会引起开发者的怨言。开了一场会议后的结果是，作为 API 使用方，可能还要修改相应的字段。

这时，我们可以尝试用以下方式来降低 API 修改带来的 bug。

- 统一 API 接口服务。将领域相关的、同一类型的 API 请求集成到一个服务之下，如与账户相关的 AccountServices。它所做的事情是统一接口请求，以及简化相关参数。当修改 API 地址时，我们只需要修改相应的文件，就可以全局生效。而一旦遇到参数、变量修改时，通过寻找被依赖的代码，就可以快速实现相应的修改，这也是编辑器自带的功能。如 WebStorm 编辑器的 Find Usage 快捷功能、Visual Studio Code 的 Find All References 快捷功能。
- API 数据模型。这里依赖于使用 TypeScript 编写应用的前端项目。编写 API 相关逻辑的时候，对应于后台返回的数据模型，编写一个接口（Interface）文件。如果我们修改了 API 模型的 Interface 文件，那么使用 API 的地方就会在编辑器上进行提示。并且，在编译的过程中，也会直接显式地告诉开发人员，该类型不存在某一属性。
- 一致化处理方式。我们从 API 返回的结果中获取了数据，我们可能将这些数据命名为 result、data、response，同时给它赋予一个变量。对应于后端返回的模型或者接口，可以给这个变量命名为 userData 等类似的名称。即使发生变更，也能方便地进行查找、修改和替换。

◎ 可选的模型适配层。即从后端返回的数据中取回我们需要的数据，并赋予新的变量名称。对于面向匈牙利命名法的 Web 应用来说，这种方式相当合理。匈牙利命名法对代码阅读不友好，如果有一层适配，可以降低代码的阅读难度。它的缺点是，需要同时维护两个数据模型。它的优点是，修改 API 时，不需要修改对应的前端代码，只需要修改适配模型即可。与此同时，可以在这层映射里，对字段进行统一的处理。

而在这个过程中，我们还会遇到一系列与 API 相关的难题。

8.2 API 管理模式：API 文档管理方式

每当我们开启一个新的项目时，总会去寻找合适的方式来管理 API 文档。总是觉得过去的那种 API 管理方式存在诸多的问题，想试试新的管理方式。

当我们拿到一个 API 文档时，会预期在这个文档里拥有使用 API 时需要的所有信息。对于 SDK 类型的 API 文档，我们预期包含有关函数、类、返回类型、参数等的详细信息，并且还需要附有教程和示例。对于前后端分离项目的 API 文档，则是预期在 API 文档中包含有后端 URL 地址、HTTP 请求方法、输入参数、返回参数、返回格式等相关的内容。

API 在此时便是一种契约，将不同的团队关联起来，能够让他们高效协作。而文档则是记录这个契约的工具之一。

1. 传统方式

传统的 API 管理方式多种多样，它们涉及的应用有桌面应用、移动客户端开发等。这一类型的应用往往与互联网模式不同，传统模式的 API 更新比较缓慢。追求稳定的同时，还需要支持不同版本应用的开发。这个版本的 SDK 功能或许比较多，也意味着收费的价格会比较高。因此，便出现了大量的 API 相关的管理方式。

在没有专业的 API 工具之前，API 文档都是手写的。有的写在 Word 里，有的写在专业的文档工具里，还有的写在项目目录下 README.md 中。每个公司都有各自的写法，无所谓好坏。

口头约定 API 是最不靠谱的方式，在今天大抵是已经不存在的。即使在同一个项目里，它也远没有聊天工具和邮件可靠。聊天工具和邮件，都可以作为核对 API 的证据。

离线 API 文档。采用 Word 文档来传递 API 文档，仍然在一定范围内是可见的。对于提供 SDK 的软件厂商来说，仍然存在这样的文档，如提供移动应用的 SDK，或者提供闭源的收费组件。通过单一文件来记录文档，难免存在滞后的问题。在文档上，既要标明相应的软件版本，又要拥有一个相应的文档中心，以知晓客户当前使用的 API 是哪个版本。如果不向第三方提供 API，那么 Web 应用的 API 就不存在版本的问题。URL 上也没有了/v3/这种版本号，也不需要维护多个版本号的后端。而如果没有了版本化，就要进行 API 变更。

在线协作 API 文档。由于我们的 API 文档需要多人、多地、多项目协作，所以采用本地的 API 文档，必然是不可行的。既然本地不行，就采用在线的工具。这类工具可以是 Google Docs，也可以是 Wiki，还可以是其他在线协作文档工具。它们都能满足协作的需求，也能保证 API 文档尽可能是最新的。

版本化 API 文档。在有了 Git 和 Git 服务器之后，我们对 Web 开发的前后端分离应用就有了更好的方式：文档代码化。与在线协作工具相比，它更加灵活，让我们拥有了可追溯的历史，以及可回退的版本工具。并且，每个开发人员都拥有一份离线版本，可以随时使用和修改。

代码即文档。在代码编写函数和功能的同时，也编写应用相应的文档。然后，通过诸如 Javadoc、JSdoc 等来生成应用的相应文档。

无论怎样，对于敏捷模式的互联网应用来说，上面的方式都不太适用。尤其是前后端分离的应用，API 文档应该不只是文档，还可以作为前端的工具来使用。

2. 互联网模式

对于互联网企业来说，文档化的方式，一方面落后，另一方面需要花费大量的时间和成本来维护。为了追求开发效率，维护 API 文档变成了一个负担。如果不采用文档的方式，那么我们就需要一种额外的方式，来作为前后端 API 的桥梁。

这种方式便是代码。它可不是普通的代码，而是一份可运行的代码，可以在后端不可用时，提供可供前端使用的 API。此外，如果我们愿意，那么只需要补充细节，便可以成为完整的 API 文档。最重要的是，它可以作为服务使用，并随时修改。

代码化的方式如下：

◎ HTTP 服务即 API 文档。对于多数互联网应用来说，前端需要的只是一个可运行的 HTTP 服务，可以在没有后端接口的情况下进行开发。因此，API 文档并不是必需的，可运行的 HTTP 服务才是最重要的。如果只有 API 文档，前端开发人员还需要自己去创建一个 Mock Server，即 HTTP 服务，来逐一模拟 API 接口。这时我们可以将 HTTP 服务当成 API 文档来使用，这种 HTTP 服务的形式比较简单，通常由一个个 JSON 文件构成。

◎ 代码生成可交互的 API 文档。它可以提供一个可编辑的在线工具如 Swagger，它以代码的方式保存 API，还能提供生成 HTTP 服务的功能。通过编写相应的 API 文档代码和 HTTP 服务代码，就可以在网页上直接测试 API，如图 8-2 所示。

图 8-2

每种方式都有自己的特点，适合于不同的场景。Mock Server 也存在不同的方式，有的使用 JSON 格式，有的使用 YAML 格式（.yml 文件）。面对这种情况，我们需要在组织内部拥有一个可用的 API 规范。

如果只有文档，那么对于前后端分离的项目没有多大用处。过去我们会在前端代码里硬编码（Hard Code）一些数据，这些数据又被称为假数据，它往往不能验证接口和请求的准确性。因此，我们需要 Mock Server 来做更多的事情。

8.3　前后端并行开发：Mock Server

Mock Server（仿造服务器），即用于仿造后端接口的模拟 HTTP 服务器。它是一个简单的 HTTP 服务，在后端未准备好的情况下，它可以为前端提供一个可用的 API 服务。在工程实践做得好的项目里，它几乎是前后端分离应用的标准配置。

8.3.1　什么是 Mock Server

在迭代 0 的时候，我们还需要讨论 Mock Server 的形式，它是前后端关于 API 开发的最重要的会议。创建 Mock Server 就意味着，我们开始创建基本的 API 规范。如果我们在 Mock Server 中使用 JSON 作为数据格式，那么往后的 API 都以 JSON 来提供。

不同项目有不同的业务场景，因此对于 Mock Server 的要求各有不同。根据模仿后台 API 的程度不同，划分出了不同的模仿精度：

◎　低。只用于前端显示相关的逻辑，即只返回某个 API 的相关字段。

◎　中。带权限相关的功能，需要在使用时，进行权限验证。

◎　高。除了上述的功能，还需要做出更高级的响应，如二次请求结果不一样。

可以看到，精度越高，所需要的开发成本就越高。好在情况并没有那么差，这些成本主要是前期的设计成本。模仿精度高的 Mock Server 虽然开发成本高，但在后期开发时，大都只需要 Ctrl + C / Ctrl + V（复制、粘贴）。基于精度考虑，可以用几种不同类型的 Mock Server 来实现：

普通 Mock Server。我们在 API 配置文件中定义了什么，便返回什么内容。其特点是简单、易维护，缺点是不容易模拟所有情况。如果只是简单的页面显示，不涉及复杂的权限逻辑，那么就可以考虑这种类型的 Mock Server。

DSL 形式的 Mock Server。DSL，即领域特定语言，它是专门针对某一特定问题的计算机语言。这种方式与普通的 Mock Server 有所不同，其配置文件（通常是 JSON）是通过特定格式编写的，返回的数据只是 API 配置的一部分。在这个文件中，我们还需要登录生成一个 Token，根据不同的请求和 Token 返回不同的内容。相比之下，比较接近于真实

的服务，登录完成后获取 Token，每次请求的时候都会验证 Token。一旦 Token 不匹配，就返回错误的结果。

编程型 Mock Server。它需要我们编写简单的代码，才能返回对应的 API 数据。它的优点是灵活性好，但是缺点是维护成本高。它需要花费一两个小时来编写代码，才能返回对应的数据，这会带来额外的开发成本。这种接口适合于复杂的项目（大部分 API 需要四五天的时间来实现），如果项目不复杂，那么使用这种类型的 Mock Server，则会有画蛇添足的味道。

因此，要按照自己的需求来选择合适的方式。从笔者的角度看，第二种类型的 DSL 更为合适。一方面，它不是一个简单的 JSON 服务器；另一方面，它的简易的 DSL 设计能满足我们大部分的业务场景。

1. 是否保持一致的业务逻辑

在 Mock Server 中，需要尽可能地包含一切 API 响应。要打造一个与业务高度一致的 Mock Server 是一件可怕的工作。编写这样的 API 接口，相当于预先编写 API 文档，同时思考可能的错误因素。

考虑 Happy Pass（成功情况）的代码，是我们需要做的事情，但是是否返回大量的异常情况，则是一件值得考虑的事情。很多时候，我们面向异常编程，即处理错误的代码，这种情况可能比处理正常代码的情况还多。所以，返回失败的 API 也就有了一定的数量，编写和维护都不是一件容易的事。

对于初次接触 Mock Server 的后端人员来说，他们会觉得考虑这么多情况的 Mock Server 是不能接受的。开发一个模拟 API，会加重开发人员的负担，其开发成本接近于真实 API 的成本。因为在编写这个模拟 API 的过程中，需要梳理一遍业务逻辑。

倘若拥有统一的规范来处理返回失败情况的 API，那么这部分的模拟 API 可以适当地减少。或者如果觉得编写相应的 API 没有必要，也可以适当地裁剪。至于返回成功的 API，则一个也不能少。

在实施 Mock Server 的过程中，还有一个问题值得考虑：这个 Mock Server 由前端还是后端开发？

2. 由前端还是后端开发?

维护 Mock Server，意味着一些额外多出来的工作量。这些工作量根据业务，分配到相关的开发人员，并由相应的前后端开发人员来完成。

相对于由前端开发人员来维护 Mock Server，由后端开发人员维护 Mock Server 则会更加方便。后端是接口的提供方，他们是经验丰富的 API 编写者，能提供准确的模拟 API。此外，作为 API 和 Mock Server 的提供方，一旦修改了 API 接口，就会去更新模拟 API。因此，在多数时候，往往由后端的开发人员来维护 Mock Server。

如果后端开发人员的进度落后于前端，那么重任便会落到前端开发人员的头上。完成这个工作，需要依赖于一定的接口经验，同时进行二次确认。如果项目已经实施了一段时间，自然问题不大。如果项目实施的时间比较短，则需要多次确认，才能确保后期 API 的修改较少。

一旦后端在实现 API 的时候不能提供某个功能，便需要修改对应的模拟 API。这种 API 变动会进一步带来契约上的变动，便需要进一步通知到使用方。若只是日常的一两个 API 修改，倒也是能理解的。一旦后端的 API 因为某种原因，如安全，需要进行大规模的修改。那么，这种大规模的 API 变动会导致开发人员花费大量的时间。

8.3.2　三种类型 Mock Server 的比较

如 8.3.1 节所述，不同类型的 Mock Server 都有各自的特色。因此我们在建立 Mock Server 的时候，需要对这方面的内容进行细致的比较。在这里，我们将进一步地对不同类型的 Mock Server 进行比较。

1. 普通 Mock Server：HTTP 服务器

普通的 Mock Server 看上去就像一个简单的 HTTP 文件服务器。在文件中定义好相关的 API，启动服务时，它便是这些模拟 API 的服务器。当然，有些框架提供的功能不只这些：

- 支持相关字段的查询、过滤。
- 支持文件中内容的全文搜索。

◎ 支持正则表达式路由。

……

主流的后端 API 都使用 JSON 格式提供的数据，它相当于一个 JSON Server，顾名思义，使用 JSON 文件快速创建 Mock Server。下面是 Node.js 中的 Mock Server 框架 json-server 的示例：

```
{
  "projects": [
    {"id": 1, "title": "aofe", "author": "phodal"},
    {"id": 2, "title": "adr", "author": "phodal"}
  ]
}
```

我们只需要访问 http://localhost:3000/projects 就可以返回对应的 JSON API。

当我们在前端应用里请求/projects API 时，会返回所有的数据；当我们访问/projects/1 或/projects?id=1 API 时，则会返回 id 为 1 的那条数据。而在这个过程中，我们只需要编写上述配置文件即可。

通过路由配置，我们可以简化大量的 JSON 文件编写。下面是 json-server 配置路由的示例：

```
//代码位于 "chapter08/json-mock-server"目录下
{
  "/api/*": "/$1",
  "/:resource/:id/show": "/:resource/:id",
  "/projects/:category": "/projects?category=:category",
  "/projects\\?id=:id": "/projects/:id"
}
```

在这个框架里，如果想提供自定义 header、授权等功能，则需要通过插件和编码来完成。

如果我们想要的 Mock Server 用途比较简单（只用于提供 API），那么我们就可以使用这个框架。在笔者经历的前端项目上，都需要进行授权等一系列相关的操作。使用普通 Mock Server 的方式，不是很友好。这时，我们需要寻找其他类型的 Mock Server。

2. DSL 形式的 Mock Server

如我们之前所说，代码的复杂度是不会消失的，它只是以一种形式转换为另外一种形式。既然我们希望拥有功能更全的 Mock Server，那么就寻找一个支持这种形式的 Mock Server，这种方式便是 DSL（领域特定语言）形式的 Mock Server。

与普通的 Mock Server 相比，DSL 形式的 Mock Server 的最大特点是用配置代替代码。即将原本需要实现的代码，变成一行行的配置。虽然我们降低了编写代码的复杂度，但是也从某种程度上提升了编写配置的复杂度。因此，总的复杂度大大地降低了。

接下来，让我们来看一个 DSL 形式的 Mock Server 示例。笔者将使用 Moco 来作为 mock server。Moco 是一个使用 Java 语言编写的简易 Mock Server 服务器。该框架使用 Java 语言，其配置都是通过 JSON 格式来提供的，而非 XML 格式。

下面是一个带 Token 认证的 API 相关示例：

```
[{
  "request": {
    "method": "post",
    "uri": "/verify-token",
    "headers": {
      "authorization": "Bear 32423424324"
    }
  },
  "response": {
    "status": 200,
    "text": "{ \"success\": true}"
  }
}]
```

运行这个服务也同样很简单，只需要一行代码：java -jar moco-runner-0.12.0-standalone.jar http -p 12306 -g config.json。当我们请求这个 API 时，如果不带 Token 或者带不正确的 Token，那么就会和正常的 API 一样返回错误。

```
$ curl -X POST -H http://localhost:12306/verify-token

> HTTP/1.1 400 Bad Request
> content-length: 0
```

而当我们带上正确的 Token，发起对 API 的请求时，就会返回对应的配置好的 Token。

```
    $ curl -X POST -H "authorization: Bear 32423424324" -I -s
http://localhost:12306/verify-token

    > HTTP/1.1 200 OK
    > content-length: 0
```

注：相关的代码位于 chapter08/moco-server 目录下，并包含相关的搭建指南。

从编写的形式上看，其在普通的 Mock Server 的基础上，添加了更多自定义的配置，即 DSL，通过 DSL 来提供更为强大的功能。当然，这种 DSL 也存在一定的局限性，一旦我们绑定了 DSL 类型的 Mock Server，便是绑定了特定的配置，这时要切换 Moco Server 就不是一件容易的事了。

此外，如果我们想要的某些特性在 DSL 中无法提供，那么可能需要通过插件的方式来支持，或者可以通过自己编写相应的代码来完成。

3. 编程型 Mock Server

与上述两种方式相比，编程型 Mock Server 能提供最大化的定制功能。在适当地配置和编程之后，可以直接使用 JSON 文件来提供服务。在需要的时候，可以方便地进行扩展，定制出适合自己的 DSL。

从维护的角度来看，笔者并不推荐使用编程型 Mock Server。一旦涉及编程，便需要选择一门语言，而前端使用的是 JavaScript，后端语言则是多种多样的。如果我们选择了一门编程型的 DSL，那么就需要选定维护的主力——到底是前端还是后端？如果选择一个非 JavaScript 的语言，并且这门语言较难上手，那么这对于前端是不友好的。而如果选择 JavaScript + Node.js，则会为后端带来一定的学习成本。

但是，当我们使用 Swagger 来提供 API 文档时，使用这种类型的 Mock Server，便能提供更好的支持。下面是使用 Swagger 生成的模拟 API 的示例：

```
    function hello(req, res) {
      var name = req.swagger.params.name.value || 'stranger';
      var hello = util.format('Hello, %s!', name);
      res.json(hello);
    }
```

Swagger 的 Mock API 是基于 Node.js 的后端 Web 框架 Express 封装而来的。因此，从上面的代码来看，我们仿佛是在使用 Express 框架编写后端的 API。

我们只需要在 swagger.yaml 文件中指定对应 URL 的响应，就可以在 API 文档里测试 API：

```
...
paths:
  /hello:
    # 绑定控制器的逻辑到路由上
    x-swagger-router-controller: hello_world
...
```

注：相关的代码位于 chapter08/swagger-demo 目录下，并包含相关的搭建指南。

两者相结合，我们就拥有了一个“活”的 API 文档。前端开发人员可以了解每个参数的输入值及示例，然后进行相应的 API 的测试。结合 Swagger 的编程 API，特别适合于对外提供 API，如图 8-3 所示。

图 8-3

此外，这些编程型 Mock Server 框架又可以提供灵活的数据创造功能。比如 Faker.js，它可以在浏览器和 Node.js 中生成大量的模拟数据，再以 API 的形式返回前端应用中。而如 Spring Cloud，则可以按类型（String、Integer 等）来接收相应的输入数据。

总的来说，编程型 Mock Server 更具灵活性，但上手成本高。

4. Mock Server 选型指南

有了上面的对比，想从上述三种 Mock Server 中，选择适合自己项目的 Mock Server，倒也是不难了。

- ◎ 如果只是前端开发人员用来简化开发，而后端不能提供支持，那么使用普通的 JSON Mock Server 就可以了。
- ◎ 如果前后端同时维护 Mock Server，那么可以尝试使用 DSL 形式的 Mock Server，可以提供更多的特性。
- ◎ 需要更多的定制化功能，则可以使用编程型 Mock Server。

笔者经历的项目，都是以敏捷类型为主的。前后端开发人员是坐在一起协作开发的，因而往往需要同时修改 mock server。采用编程方式的 Mock Server，需要选择一门语言。而无论哪种语言，对于其他技术栈的开发人员来说，都是一个新的挑战。因此，往往偏向于采用 JSON 形式，这样对两端更为友好。

笔者曾经也在一些项目上使用其他 Mock Server 工具，如 Spring Contract，它使用 groovy 的语法来编写模拟 API。对于采用 Java 语言和 Gradle 的后端开发人员来说，使用 groovy 还是相当容易上手的；但是对于前端开发人员来说，groovy 的语法和 JSON 相差甚大，写出模拟 API 略微麻烦。

8.3.3 Mock Server 的测试：契约测试

业务修改会带来一定量的 API 修改，一旦 API 修改便需要通知使用方。通知的方式再好，都比不上持续集成的失败，它以直接的状态（红色）显示当前的构建有问题。那么，要怎样做才能在 API 修改时，影响到持续集成呢？答案：契约测试，即对模拟 API 或真实的 API 进行测试。

契约测试，又称为消费者驱动的契约测试（Consumer-Driven Contracts，简称 CDC），是指从消费者业务实现的角度出发，驱动出契约，再基于契约对提供者进行验证的一种测试方式。

契约是一个高级版的 Mock Server。两者的区别在于，契约是双方或多方共同协议的，而 Mock Server 则更多是从使用方的角度来考虑的。当我们要往 Mock Server 里添加一个

新的契约（模拟 API）时，这个 API 是需要双方认可和协定的。一旦定下了这个契约，我们就可以验证，API 产出是否与契约保持一致。对应的，我们需要在前后端里，各自对契约进行测试，如图 8-4 所示。

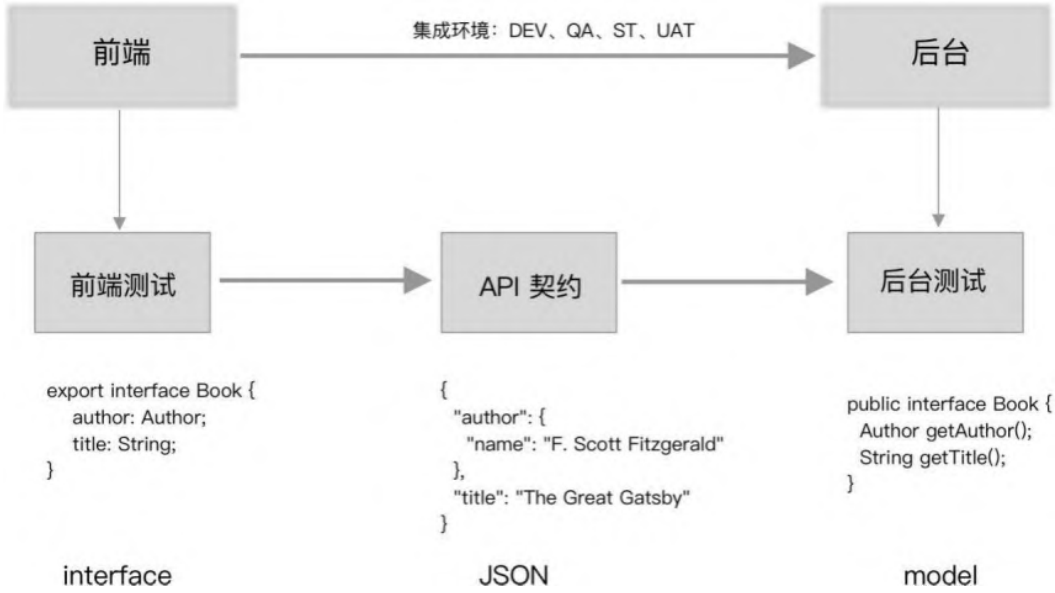

图 8-4

各端以自己的方式，验证契约是否符合自身要求。在编写测试的过程中，值得关注的点是，如何以合理的方式来测试契约。其中最常见的关键点是 API 的一致度。如我们的 API 返回了 20 个字段，是否对每个字段进行校验？可能的情况如下：

◎　校验所有的字段名和返回结果是否一致。

◎　只校验所有的字段名是否一致。

◎　校验部分字段名和返回结果是否一致。

◎　只校验部分字段名是否一致。

◎　是否校验所有可能的返回结果？如在 A、B、C 等条件下，是否都测试？

校验所有的字段，就意味着编写测试和维护代码时的复杂度会提高。校验部分的字段，则可能缺少一些重要的信息。一个折中的方案是：校验所有的字段名、部分的返回结果是否一致。具体在每个项目上实践时，都各有差异，需要慎重考虑。既不要让契约测试成为

累赘，又不要让它可有可无。

在笔者经历过的项目里，曾经使用 JSON 文件来比对 API 的返回结果。在测试中，仿造出一个与返回 API 完全一致的数据，代码中有 10 个字段，将这些字段与 JSON 文件进行比对。此外，我们还测试了 API 在不同条件下返回的结果。如果某个字段需要修改，那么所有 JSON 文件都需要修改。它可以保证代码在前端的健壮性，但是对于后期维护项目的人而言，体验就不是很友好了。

1. 后端契约测试

后端作为 API 的生产者，验证编写出来的 API 能否和 Mock Server 上的字段、类型相匹配。其大体的测试步骤如下：

（1）先运行 Mock Server 服务。

（2）发起对 Mock Server 服务的 API 请求，获取相应的返回数据。

（3）判断相应的数据、字段与 API 中的一致。

因此在进行技术选型的时候，会出现消费者驱动的技术选型的情况。后端要进行契约的测试，因而其在选择契约服务器（即 Mock Server）时，会偏向于与后端更吻合的契约测试工具，例如 Spring Cloud Contract。这样的工具和框架适合于后端进行开发和测试，但是不一定适合前端人员编写。如接下来我们要展示的 Spring Cloud Contract，其采用 Groovy 语言来编写契约，过于复杂的场景不适合前端人员编写模拟 API。

下面给大家介绍使用 Spring 框架官方的契约测试示例，相应的 Mock Server API 的部分代码如下所示：

```
Contract.make {
    description "should return person by id=1"

    request {
        url "/person/1"
        method GET()
    }

    response {
        status 200
        headers {
```

```
            contentType applicationJson()
        }
        body (
            id: 1,
            name: "foo",
            surname: "bee"
        )
    }
}
```

这是一个模拟 API，在 request 字段里定义了请求的 API 的 URL 和 HTTP 请求的方法；在 response 里则定义了对应的返回状态码、header 和返回数据。在总体写法上，与我们之前使用 JSON 的格式是类似的。

接着，我们运行 Mock Server，就可以发出 API 请求，获得相应的响应结果。然后，就可以验证相应的结果是否正确：

```
    @Test
    public void get_person_from_service_contract() {
        // given:
        RestTemplate restTemplate = new RestTemplate();

        // when:
        ResponseEntity<Person> personResponseEntity = restTemplate.
getForEntity("http://localhost:8100/person/1", Person.class);

        // then:
        BDDAssertions.then(personResponseEntity.getStatusCodeValue()).
isEqualTo(200);
        BDDAssertions.then(personResponseEntity.getBody().getId()).
isEqualTo(1l);
        BDDAssertions.then(personResponseEntity.getBody().getName()).
isEqualTo("foo");
        BDDAssertions.then(personResponseEntity.getBody().getSurname()).
isEqualTo("bee");
    }
```

后端拥有对应的数据模型，要验证每个字段也比较容易，只需要从 body 里获取对应的值即可。如果 Mock Server 中的值被修改了，如上述 Mock Server 中的 name 值被修改为 bar，那么这里的测试也就失败了。继而影响整个系统的持续集成，以及自动化构建。

这时，我们就遇到一个问题，契约的修改是否影响构建？答案是不确定的。它取决于我们对于契约的重视程度。如果契约和 API 的修改经常带来各种问题，而这些问题推动起来很困难（如异地开发的沟通问题），那么我们就可以让契约测试来影响构建。而如果契约和 API 的修改很少带来问题，那么我们还是要利用契约测试来保障应用程序的质量。

不过，如果在契约进行修改时，前后端双方都能做出快速的响应，契约测试的重要性就没有那么大了。正因为团队协调不容易、响应比较慢，所以对于规则的要求会更高。

依靠后端的测试，只能保证后端 API 与契约是一致的。前端也需要自己的契约测试，以保证前端 API 和契约是吻合的。

2. 前端契约测试

前端作为契约的消费者和使用者，才是更关心契约修改的一方。可是如果后端开发人员没有对 API 与契约的一致性负责，那么前端的契约测试就会变成一个枷锁。因此，前端契约测试存在必要不充分条件，即存在后端的契约测试。

前端的契约测试与后端的契约测试相差无几：请求 API、获取数据、校验数据。下面是一个使用 Jest 编写的契约测试的示例：

```
describe('#API Contract Test', () => {
  it('should return correctly data', async () => {
    const data = await mockHttp.getBook('1')
    expect(data).toBeDefined()
    expect(data.entity.name).toEqual('前端架构')
  });
});
```

对于不同的项目，我们所需要校验的内容，也有所差异。毕竟完完整整地去测试一个契约，成本太高了。一旦业务修改带来新的返回值的变化，我们就需要修改对应的测试。有了后端的契约做保证，我们只需要做一些简单的校验：

- 字段名一致校验。逐一校验每个 API 的返回值过于烦琐，而且 API 字段多数时候也不需要修改。我们只需要检验后端返回的 API 字段是否与我们需求的一致即可。就是说，我们只需要遍历返回数据的 key，与我们所需的 key 列表进行对比，就可以完成测试。对于那些经常修改字段名、采用匈牙利命名法命名字段的项目来说，这种测试是非常适合的。

◎ 对于使用 TypeScript 编写应用的前端项目来说，可以尝试使用项目中的 interface 接口（即面向对象中的类）来进行 API 测试。笔者编写的 mest 框架，便是借用 interface 文件来生成数据，将其与后端的字段名进行对比。

◎ 校验逻辑字段。即我们并不验证 API 的所有字段是否正确，往往只验证带逻辑部分的 API 是否正确。如上所述，逐一校验每个字段，开发和维护成本太高。对于后端来说，仍然有必要验证接口。对于获取 API 的前端页面来说，它们往往以展示为主。如果展示的字段不带额外的处理逻辑，就没有理由去验证这个字段。只需要关注需要逻辑处理部分的 API，再进行绑定和测试即可。

在契约测试里，我们关注的是 API 本身，这与单元测试、集成测试有所不同。在实现的过程中，我们很容易将契约测试混入单元测试等代码中——因为这样可以方便我们测试从 API 返回到显示的逻辑。但是，在笔者看来，对于契约测试应该与其他测试代码相分离，形成独立的契约测试代码。

8.3.4　前后端并行开发总结

在有了契约和 Mock Server 之后，前后端的开发模式变成：

◎ 前后端约定契约 API，并完成对应的 Mock Server 实现。

◎ 前后端根据各自的逻辑实现对应的业务代码。

◎ 前后端编写各种契约测试，并确定 API 的修改能够反映到持续集成。

◎ 前后端进行 API 集成。

◎ 在 API 修改时，修正对应的 API 修改。

它能确保前后端以各自独立的方式运行，但是它并不能保证后端能提供前端需要的 API。因为“CRUD 式的 API”往往返回的是所有的字段和未经处理的值。当我们拥有 Android、iOS、移动 Web、桌面 Web 时，如果某个 API 上需要运行一些处理代码，那么就需要在四个客户端上进行各种处理。这个时候需要怎么做呢？

8.4 服务于前端的后端：BFF

前端和后端经常会有各种关于业务处理的讨论。某个业务处理的代码，放在前端还是后端？这个问题需要一方去说服另一方，也不容易讨论出一个结果来。前后端各自都包含一些业务逻辑，逻辑放在前端或后端，并没有太大的区别。

一个简单的共识是，如果多端都需要一个计算如时间转换，那么应该由后端提供。相同的逻辑，不应该在不同的客户端上实现多遍。如一个关于文章、资讯等的时间展示例子，按内容发布时间的不同，有不同的处理逻辑：

（1）今天，显示今天 + 对应的时间，如今天 7:30。

（2）昨天，显示昨天 + 对应的时间，如昨天 7:30。

（3）昨天之前的本周的其他时间，则显示星期几 + 对应的时间，如星期几 7:30。

（4）如果时间超出本周，且不是昨天，则显示日期 + 对应的时间，如 2018 年 11 月 6 号 7:30。

如果存在 4 种客户端（Android、iOS、桌面 Web、移动 Web），一个功能以 0.5 天的实现时间来计算，那么总共需要 2 天。加上对应的测试和 bug 修复，至少要 3 天。而如果是服务端实现，那么加上测试，总体上只需要 1.5 天左右的时间。如果这种情形多，如 20 个需求 * 1.5 = 30 天，从时间上说，由后端来实现是相当划算的。如果在这个过程中，业务需求发生了变化，那么就会产生一系列的修改成本。

大量诸如此类的业务逻辑放在后端实现，能在某种程度上降低应用的开发和维护成本。这些与业务相关的处理逻辑，直接放在原有的后端代码中也不是很合适。它们类似于胶水代码，难以整理和维护，并且它们需要经常修改。为了将这些业务代码抽离出来，形成更好的系统架构，便出现了前端、后端之间的中间层，这样的中间层称为 BFF。

8.4.1 为什么使用 BFF

BFF，即 Backends For Frontends（服务于前端的后端），是指在服务器设计 API 时会考虑客户端的使用情况，在服务端根据不同的设备类型返回不同客户端所需要的结果。BFF

模式不会为所有的客户端创建通用的 API，而是创建多个 BFF 服务：一个用于 Web 前端，另一个用于移动客户端（甚至一个用于 iOS，另一个用于 Android）。BFF 下的 API Gateway 如图 8-5 所示。

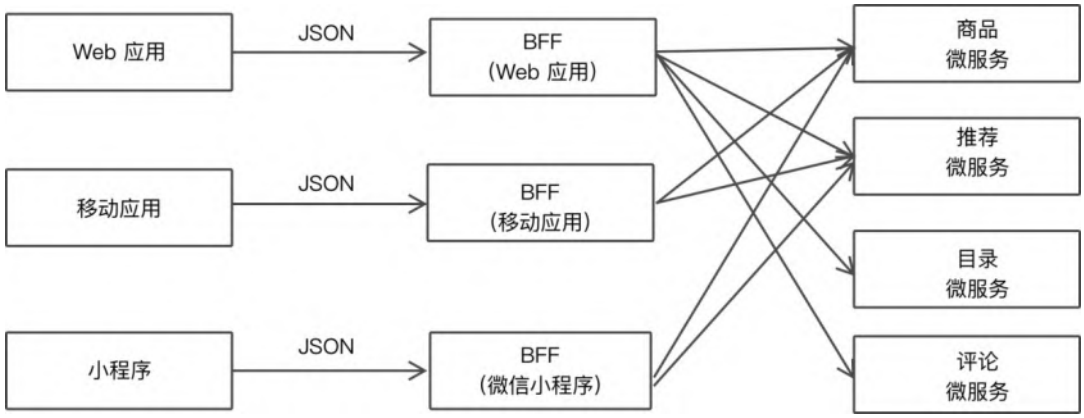

图 8-5

在图 8-5 的架构中，每种类型的客户端都有自己的 BFF 服务。每个客户端都可以在自己的 BFF 里添加所需要的业务逻辑。前端可以针对自己需要的部分，在面向前端的 BFF 服务里添加自己所需要的逻辑。移动端也是如此，可以自定义自己的 API 响应。

如果针对每一种类型的设备都有自己的 BFF，难免会在维护系统时带来一些额外的成本。倘若这些都是业务独立的应用，就不会带来太多的困扰。只要应用业务逻辑保持一致，维护多个 API 就会显得多余。这时可以考虑一种折中的方案：一个主要的、完整的 API，加上一个或两个针对移动设备的变种，或者针对移动设备做一个 BFF 层，而针对桌面应用做另外一个 BFF 层。

如果前端要对接一个已经完备的后端 API，而非从头开发，那么我们要考虑开始一个 BFF。如果后端 API 的提供方不提供 API 封装支持，那么就要由使用方来创建一层 BFF；如果对方愿意提供 BFF，那么我们什么事都不用做。大都是前者居多，需要由我们来封装 API。

1. BFF 真的需要吗?

在不同项目下，使用 BFF 的意图各有不同。我们使用 BFF 的目的可能是：

◎　应对多端应用。一方面，对不同的客户端进行特定的业务处理；另一方面，集中

处理统一逻辑，降低开发成本。

- 聚合后端微服务。当一个业务的处理和展示，需要多个后端服务时，可以通过APIGateway 来聚合后端服务，以加快应用响应的速度。
- 代理第三方 API。客户端不直接访问第三方 API，而是通过 BFF 作为中介来访问第三方 API。同时，按自方系统的逻辑来实现符号自身需要的 API。当更换第三方服务时，可以不更换 API。
- 遗留系统的微服务化改造。从单体架构迁移至微服务时，先通过 BFF 来调用单体应用的功能。再逐一创建微服务，当完成一个服务时，将请求指向新的微服务。慢慢地，就可以从遗留代码转移到新的演进方案上。

问题是，由于每个 BFF 是独立的服务，它们的独立程度相当高。这意味着，要同时维护这么多个 BFF 并不是一件容易的事——好在作为一个前端开发人员，我们只需要维护前端的 BFF 层。但是，我们真的需要 BFF 吗？

我们要考虑的几个因素是：

- 是否需要提供多种接口，来适应不同的客户端？
- 是否需要针对某一特定客户端，进行后端接口优化？
- 是否需要为第三方提供 API？
- 是否存在大量的后端服务需要聚合？
- 是否需要为客户端进行业务逻辑处理？

不论怎样，对于快速变化的 UI 和业务来说，BFF 是一个比较好的解决方案。它处于两端之间，能更好地响应变化。

2. 对比 API Gateway

如图 8-6 所示是一个常见的 API Gateway 在系统中的架构：

其在架构上与 BFF 的作用类似。API Gateway 是一个位于前端与后台服务之间的代理，也是后台服务的唯一入口。它将请求由客户端路由到对应的服务，并执行身份验证、监控、负载均衡等任务。比如 Netflix 开源的微服务网关组件 Zuul，通过过滤器可以实现身份认证与安全、审查与监控、动态路由、静态响应处理、压力测试等功能。

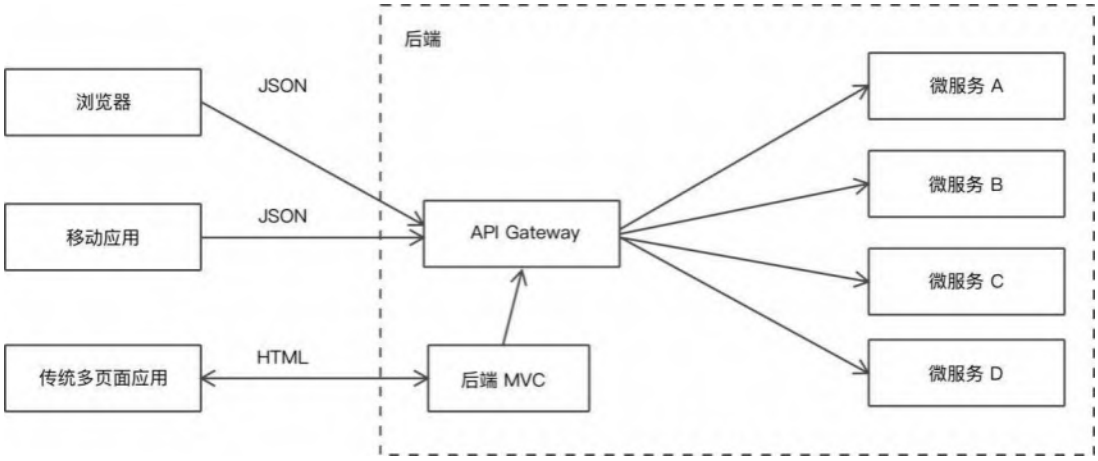

图 8-6

简易的 API Gateway 可以是一个如 Nginx、HAProxy 这样的反向代理，统一前端请求的 API 入口，并分发到不同的服务上。复杂的 API Gateway，可以聚合后端的服务，还能根据客户端的需要返回不同的内容。或者通过查询参数来向不同的客户端提供接口，/api/news?type=web 用于向 Web 提供接口，而/api/news?type=mobile 用于向移动端提供接口。

API Gateway 与 BFF 最大的区别在于，API Gateway 只拥有一个 API 入口，而 BFF 则是针对不同客户端，拥有各种 API Gateway。此外，BFF 会根据业务逻辑进行编码。而 API Gateway 只做数据的转发，不做额外的数据。因此从某种程度上来说，BFF 是一种高级的 API Gateway。

8.4.2　前后端如何实现 BFF

在实现 BFF 之前，我们要商榷一下，这层 BFF 到底是由前端来实现的，还是由后端来实现的，或者是两端一起实现的。多数时候，不会同时由两端来实现，往往由某一方来主导。前端实现 BFF 和后端实现 BFF 的主要差别在于其所选的语言。也会有后端开发人员想去尝试前端的技术栈，前端的开发人员想去尝试后端的技术栈。可受限于内部因素，这也只是少数。

前端开发人员擅长编写 JavaScript/TypeScript，所以由 Node.js 来实现 BFF 层，几乎是前端的第一选择。前端开发人员可以选择合适的 Web 框架（如 Koa、Express、Egg.js）来打造 BFF，也可以使用更合适的 GraphQL 来完成。

后端开发人员在实现 BFF 时，出于维护或者组织内部规则，会偏向于使用后端现有的技术栈。前端开发人员按照自己习惯的 Ajax/Fetch 请求写代码，再按一定的逻辑展示到页面上。

1. 传统后端技术栈下的 BFF

尽管 Node.js 对于前端开发人员更友好，但是在前端团队经验不够丰富的情况下，采用传统的后端技术栈是一种更稳妥的方案。修改内容主要是字段处理，如果有了一部分 BFF 代码，那么不需要对后端的数据库等有深入的了解，只需要按业务逻辑 对数据进行一些特殊的处理即可。

对于使用后端技术栈的 BFF，前端开发人员几乎很少有话语权，但是这也带来了极大的优势。前端开发人员不需要维护 BFF，只需要在合适的时候提出自己对 API 的需求即可。说到后端的技术栈，偶尔尝试不同类型的语言，会对自己的成长有帮助。长期处在某一特定的计算机领域里，多少会让自己的思维形成一些固化。进入一个新的领域，有时会以第三视角来重新审视原来的技术栈。

由于后端部分不是重点，并且后端部分与其他 API 编写的差异不大，这里就不再详细地展开讨论了。

2. 前端技术栈下的 BFF

采用 Node.js 作为前端的 BFF 层是前端人员最有可能的技术栈选择。JavaScript 的动态语言特性加上 Node.js 的性能，使得 Node.js 在快速实现与业务相关的 BFF 层时有极大的优势。由后端 APIGateway 获取的数据是 JSON 格式的，JavaScript 可以快速地对 JSON 进行解析。前端技术栈没有静态类型语言的类型问题，可以相当便捷地修改结果，并返回给前端使用。即：

（1）通过 Node.js 来接收前端发送过来的请求。

（2）根据请求的类型向对应的后台 API 服务发起请求。

（3）获得返回结果后进行处理，并向前端返回对应的结果。

通过以上步骤，前端开发人员就可以在这层 BFF 里添加自己所需要的业务逻辑。一旦业务上发生一些变化，也可以直接在 BFF 上进行修改，而不需要修改后端服务或者前端代码。实现相应的业务逻辑，对于开发人员来说并不算复杂，复杂的部分有如下几个地方：

◎ 难以推进 Node.js 后端服务的使用。它可能受限于组织内部的审查和安全策略。

◎ 上线时，需要前端人员配合上线。在传统的上线方式里，前端只是作为静态文件来部署，可以交由后端来完成上线。

此外，尽管 Node.js 已经越来越稳定，但是与 Node.js 相关的调试、内存泄漏问题的排查，也需要有相关的能力才能完成。如果组织内部缺乏相关的专家，想要大规模地采用 BFF 并不是一件容易的事。

8.4.3　使用 GraphQL 作为 BFF

与普通的 Node.js + Web 框架实现 BFF 相比，更流行的方式是采用 GraphQL。GraphQL 既是一种用于 API 的查询语言，又是一种标准，也相当于一个满足开发者数据查询的运行时。

用过数据库、搜索引擎的人，大都能理解查询语言是什么。开发人员只需要编写查询语句就可以获取数据源中的相关数据。如果在前端编写查询语句，那么前端就可以轻松地获取自己需要的数据，而不需要后台做大量的处理。下面是 GraphQL 查询一个 id 为 2 的记录的代码，功能是获取其中的 id、category 和 featured_image 三个字段的值：

```
query make {
  make(id: "2") {
    id,
    category,
    featured_image
  }
}
```

然后，将这个请求发送给 GraphQL，而 GraphQL 则会按照定义的 RESTDataSource 向对应的后台服务请求数据，并返回结果：

```
{
  "data": {
    "make": {
      "id": "2",
      "category": "2",
      "featured_image": "uploads/make/.thumbnails/connected-smart- cities.jpg/
connected-smart-cities-600x360.jpg"
```

```
        }
      }
    }
```

下面是获取数据相应的代码：

```
    //代码位于chapter08/graphql-demo/目录下
    class MakeAPI extends RESTDataSource {
      constructor() {
        super();
        this.baseURL = 'https://www.wandianshenme.com/api/';
      }

      async getMake(id) {
        return this.get(`make/${id}`);
      }

      async getMakes(args) {
        const data = await this.get('make', args);
        return data.results;
      }
    }
```

它与常规的 BFF 有极大的不同：前端只需要传入自身需要的字段，GraphQL 便返回我们所需要的字段。如果使用传统的 BFF 方式，那么需要对代码进行适度修改，才能返回前端需要的字段。一旦业务发生变化，就要反复修改常规 BFF 层的代码。

如果采用 REST 架构，那么上述 API 请求应该是：

```
    GET https://aofe.phodal.com/api/make/:id
```

其返回结果如下：

```
    {
        "id": 1,
        "category": 1,
        "featured_image": "uploads/make/.thumbnails/amazon-echo.jpg/amazon-
echo-600x360.jpg"
    }
```

定义数据源类（如上述代码中的 RESTDataSource）相当于创建对应的数据源。在数据源中向对应的后端服务发送 API 请求，相当于一个高度定义的 APIGateway。这也是

GraphQL 中编码的主要部分，它会处理前端的请求，并从对应的后端服务中获取数据。如图 8-7 所示是在 Web 端调试的示例。

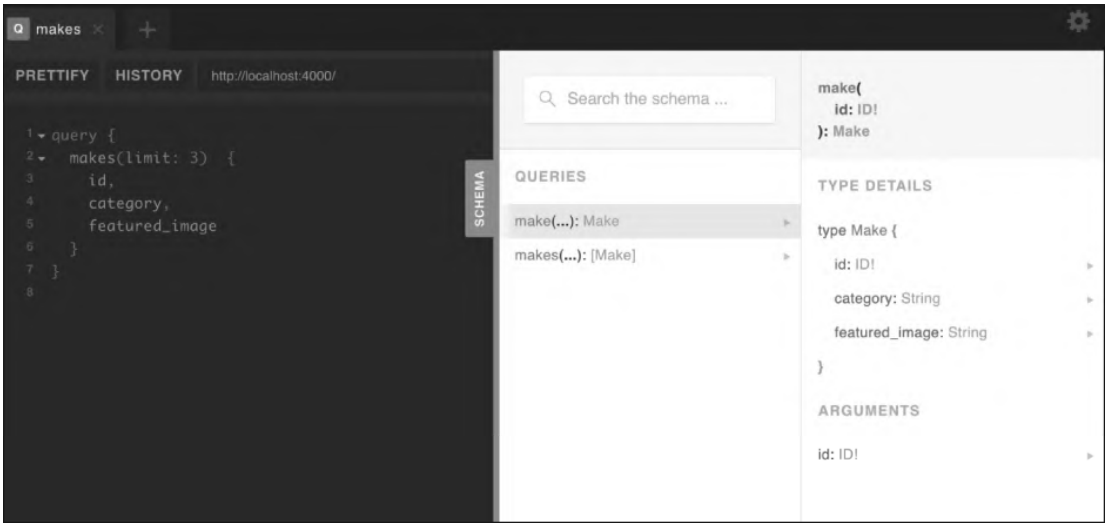

图 8-7

值得注意的是，GraphQL 只定义了这种查询语言语法、语句如何执行等。GraphQL 真正的实现则依赖于不同的服务端库。这些服务端在实现时有一些差异。框架的开发者会按照设想和用户需求进行一些额外的扩展，以获取更多的开发者。因此，在选择 GraphQL 服务端库的时候，会存在一定的功能差异。在客户端上使用时有丰富的选择，既可以使用 Facebook 推出的 Relay，又可以使用 GraphQL 服务端库 Apollo Server 的开发商提供的对应客户端库 Apollo Client。

1. GraphQL vs 常规 BFF

从 GraphQL 的种种优点来看，它也被视为 REST 架构的替代方案。从形态上看，它是一个高度标准的架构，我们需要按照它的规范来编写应用。不同的团队对于 REST 架构的理解略有不同，一些特定的资源处理方式有多种实现方案。在 REST 架构中，资源和大小是由服务器确定的，在 GraphQL 中服务器只是声明可用的资源，客户端可以取得自己需要的东西。

尽管从某种意义上将它和 REST 架构对比并不合理，但是我们可以将这里的 REST 架构视为 REST 架构的 BFF。GraphQL 的比较对象应该是常规的 BFF，相比之下 GraphQL

具有下面的优势：

- 按需获取。客户端可以按自己的需要，从服务端获取已经定义好的资源和数据，而非进行与服务端 BFF 相关的编程。
- 代码即文档。与参数相比，GraphQL 编写的查询语句更像是一份文档。换句话说，它适合于人类阅读。
- 易于使用的 API 调试工具。如图 8-7 所示，多数的 GraphQL 实现都能提供一个开发用的前端调试 API 界面，可以进行 API 请求、验证等。
- 强类型的 API 检查。面向前端的接口都有强类型的 Schema 做保证，能快速地定位问题。
- 易于版本化的 API。其可以通过 Schema 扩展 API，而 REST 则需要通过 URI 或 HTTPheader 等来接收版本。

当然，它也有一些缺点：

- HTTP 请求无法被缓存。由于所有 HTTP 的请求只能在 App 级别上实现缓存，即通过 GraphQL 客户端库来实现。
- 错误码处理不友好。GraphQL 统一返回 200 的结果，在其中对错误信息进行包装。对于传统的 HTTP 客户端来说，需要额外的处理才能走到异常分支。
- 学习成本。使用 GraphQL 就要学习一门查询语言，同时还需要写一大堆 Schema 才能使用这种特定格式。从某种意义上来说，相当于学习一门数据库语言。
- 实现复杂。普通的场景下，需要开发人员编写 Schema 声明，手动编写 Resolver 来关联字段。一旦遇到复杂的场景，就会难以控制。

相比之下，很难在常规的 BFF 和 GraphQL 中做出选择。如果业务一直在变动，或者需要对外提供 API，选择 GraphQL 更合适。而如果业务变动不频繁，或者客户端数据量少（如只有 Web），那么使用 BFF 也可以。

2. 基于 GraphQL 的 BFF 架构

一旦我们要尝试在系统中引入 GraphQL，就要考虑它在系统中的位置。考虑到 GraphQL 是 BFF 的一种类型，而 BFF 又是一个进阶版的 BFF，我们可以采用如下几种架构方式：

- GraphQL -> API Gateway ->后端服务。即我们只是将 GraphQL 作为系统的一个随时可替换的“装饰器”存在，它和常规的 BFF 一样，随时可以被新的技术替换掉，而不影响系统的整体设计。
- GraphQL -> REST、RPC 等接口的后端服务。API Gateway 和 BFF 做的事情有一部分是重复的，如果我们打算进一步使用 GraphQL，那么就可以直接由 GraphQL 来对接后台服务。而这些后台服务与 GraphQL 通信的方式，可以是 REST 式、也可以是 RPC 式。在这一点上，和常规的 BFF 并没有太大区别。
- GraphQL -> GraphQL 接口的后端服务。当我们下决心使用 GraphQL，并且从头构建基于 GraphQL 架构的应用时，可以尝试直接在后端服务中直接使用 GraphQL 来提供接口，而不是 REST 或者 RPC 等的形式。只是未来一旦 GraphQL 不适合了，就需要重新修改接口。

如图 8-8 所示是采用 GraphQL 作为 APIGateway，同时后端服务都提供 GraphQL 接口的前后端分离架构示例图。

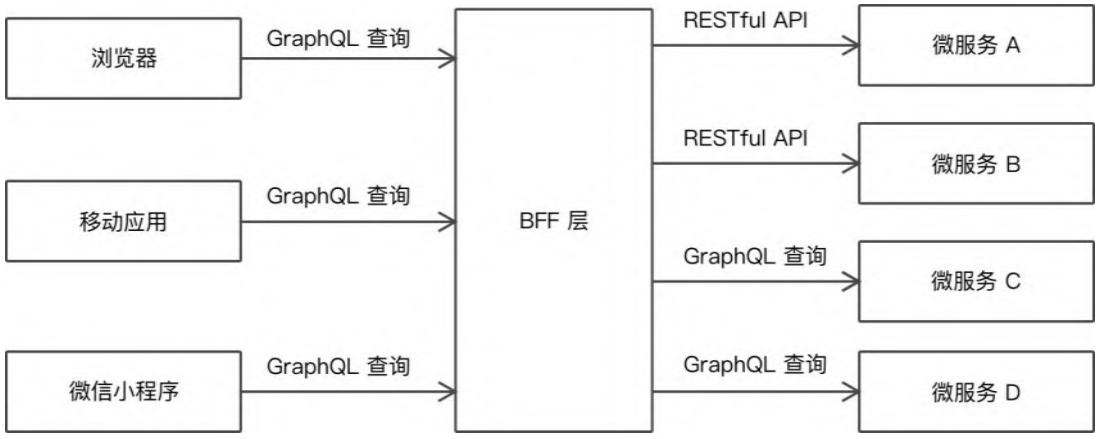

图 8-8

与 GraphQL 相似的还有搜索引擎，使用搜索引擎构建 BFF 层是一个烦琐的过程，我们需要构建索引、设计应用程序的架构，并设计一套与搜索引擎同步的逻辑。

GraphQL API 的强类型特性，也非常适合模拟 Mock Server 的存在。同时，结合测试框架 Jest + Relay，便可以进行快照（Snapshot）测试。

8.5 小结

在本章中，我们详细介绍了前后端分离架构下的前端应用开发。首先，介绍了不同的API 的管理方式（离线文档、在线协作、版本管理等），以及使用 API 服务作为 API 文档的方式。然后，介绍了什么是 Mock Server、它在前后端分离架构中的作用，以及如何确保在开发过程中互不影响。此外，演示了三种类型的 Mock Server：

- 普通的 Mock Server。
- DSL 形式的 Mock Server。
- 编程型 Mock Server。

本章还介绍了如何通过将 Mock Server 中的模拟 API 作为契约进行契约测试来降低系统的 bug，提高稳定性。

另外，我们在本章引入了一个降低业务耦合性的 BFF 层，即服务于前端的后端。它可以降低多端应用的开发成本，并提高应用开发的灵活性。同时，还介绍了当前流行的用 BFF 实现 GraphQL 的方法，并演示了其如何在 Web 架构中发挥作用。

9 第 9 章 架构设计：微前端架构

大型组织的组织结构、软件架构在不断地发生变化。移动优先（Mobile First）、App 平台（One App）、中台战略等，各种口号不断地被提出、修改和演进。同时，其业务也在不断地发展，由线下到线上、从无到有，这些进一步地导致组织的应用不断地膨胀，进一步地映射到软件架构上。

这些让我们联想到“康威定律”：设计系统的组织，其产生的设计和架构等价于组织间的沟通结构，可它并不是在反映康威定律，而是相似的推论：系统的组织在不断变化的同时，其设计和架构也在不断地调整。

而在组织结构变化的同时，架构也随着产生了一系列的变化，毕竟天下大势，分久必合，合久必分。与数据库的分库分表一样，既然一个组织的部门已经过于庞大，就进一步将它细化。同理，软件的不同部分又被拆分到不同的部门之下。随着不同部门的业务发展，技术栈也因此而越来越难以统一，出现了多样化。在走向多样化后，用户越来越厌倦一家公司的应用软件（App）分散在多个不同的应用上。应用的获客成本越来越高，应用又一

次走向聚合。

在分离了前后端之后，拆分降低了系统的复杂度，并进一步提高了软件的开发效率。随着业务的不断扩张，需求也不断扩张，应用又开始变得臃肿。既然应用变大了，我们就继续往下拆分，拆分成更小的单位。在本章中，关注于如何采用微前端架构，来解决复杂的前端应用。接下来将要学习的内容有如下几个部分：

◎ 什么是微前端架构？它是如何形成的，以及有什么优缺点。

◎ 如何设计一个微前端架构的系统？

◎ 如何合理地拆分前端应用？

最后，我们还将引入“微”害架构的概念，即不合理地实施微架构将对系统产生什么影响。

9.1 微前端

微前端是一种类似于微服务的架构，它将微服务的理念应用于浏览器端，即将单页面前端应用由单一的单体应用转变为把多个小型前端应用聚合为一的应用。各个前端应用还可以独立开发、独立部署。同时，它们也可以进行并行开发——这些组件可以通过 NPM、Git TagGit 或者 Submodule 来管理。

9.1.1 微前端架构

微前端的实现意味着对前端应用的拆分。拆分应用的目的并不只是为了在架构上好看，它还可以提升开发效率。比如 10 万行的代码拆解成 10 个项目，每个项目 1 万行代码，要独立维护每个项目就会容易得多。而我们只需要实现应用的自治，即实现应用的独立开发和独立部署，就可以在某种程度上实现微前端架构的目的。

1. 应用自治

微前端架构，是多个应用组件的统一应用，这些应用可以交由多个团队来开发。要遵循统一的接口规范或者框架，以便于系统集成到一起，因此相互之间是不存在依赖关系的。

我们可以在适当的时候，替换其中任意一个前端应用，而整体不受影响。这也意味着，我们可以使用各式各样的前端框架，而不会互相影响。

2. 单一职责

与微服务类似的是，微前端架构理应满足单一职责的原则。然而，微前端架构要实现单一职责，并非那么容易。前端面向最终用户，前端需要保证用户体验的连续性。一旦在业务上关联密切，如 B 页面依赖 A 页面，A 页面又在一定的程度上依赖 B 页面，拆分开来就没有那么容易。但是如果业务关联少，如一些关于“我们的联系方式”等的页面，使用不多，没有多少难度。因此，一旦面临用户体验的挑战，就要考虑选择其他方式。

3. 技术栈无关

在后端微服务的架构中，技术栈无关是一个相当重要的特性。后端可以选用合适的语言和框架来开发最合适的服务，服务之间使用 API 进行通信即可。但是对于微前端架构来说，虽然拥有一系列的 JavaScript 语言，但是前端框架是有限的，即使在某个微前端架构里实现了框架无关，也并不是那么重要。框架之间的差距并不大，一个框架能做的事情，另一个框架也能做，这一点便不如后端。使用 Java 解决不了的人工智能部分，可以交给 Python；如果觉得 Java 烦琐，可以使用 Scala，如图 9-1 所示。

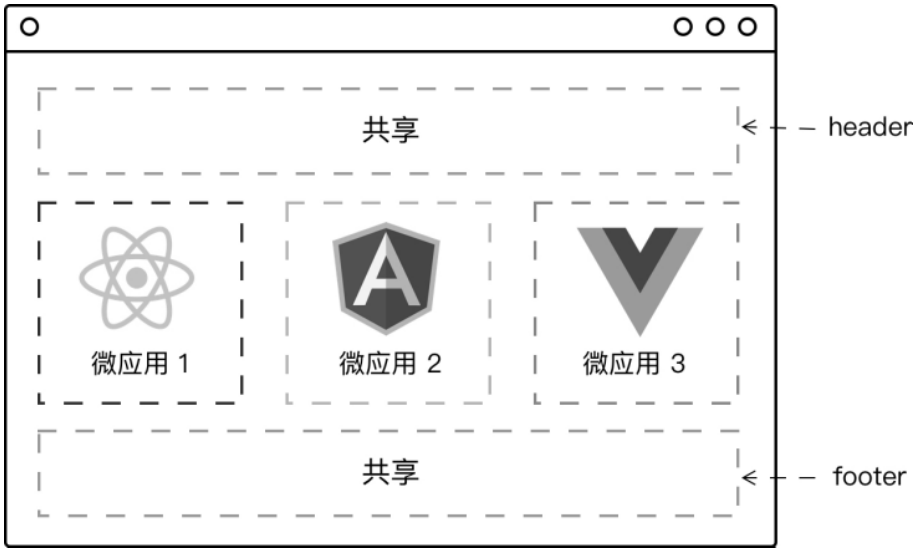

图 9-1

对大部分公司和团队来说，技术无关只是一个无关痛痒的话术。如果一家公司的几个创始人使用了 Java，那么极有可能在未来的选型上继续使用 Java。对于前端框架来说也是相似的，如果我们选定 Angular，除非出现新的框架来解决 Angular 框架的问题，否则大概率继续使用原有的框架——毕竟已经拥有大量成熟的基础设施。

此外，技术栈无关也有一系列的缺点：

◎ 应用的拆分基础依赖于基础设施的构建，如果大量应用依赖于同一基础设施，那么维护就变成了一个挑战。

◎ 拆分的粒度越小，意味着架构变得越复杂、维护成本越高。

◎ 技术栈一旦多样化，便意味着技术栈是混乱的。

那么，我们到底应该在什么时候采用微前端架构呢？

9.1.2 为什么需要微前端

虽然微前端不是能在 10 年内提高 10 倍生产力的银弹。然而除了上述微前端的优点，仍然有其他理由让我们去采用微前端架构：

◎ 遗留系统迁移。

◎ 聚合前端应用。

◎ 热闹驱动开发。

不同的出发点，在实践方式上都略有差异，值得我们花费时间去尝试。

1. 遗留系统迁移

笔者曾在 GitHub 上开源有微前端框架 Mooa 及对应的文档《微前端的那些事儿》。自内容发布以来，笔者陆续收到一些微前端架构的咨询。过程中发现了：解决遗留系统，才是人们采用微前端方案最重要的原因。因为在这些咨询里，开发人员所遇到的情况与之前遇到的情形并不相似。笔者的场景是设计一个新的前端架构，而这些开发人员要考虑前端微服务化是因为遗留系统的存在。

过去那些使用 Backbone.js、Angular.js、Vue.js 等框架所编写的单页面应用，已经在线上稳定地运行了，也没有新的功能。对于这样的应用来说，我们也没有理由浪费时间和精

力重写旧的应用。而这些应用是使用旧的、不再使用的技术栈编写的，由于框架本身已经不更新（不增加新功能或者不再维护），因此应用可以称为遗留系统。既然应用可以使用，就不花太多的力气重写，而是直接整合到新的应用中去。

不重写原有系统，同时抽出人力来开发新的业务，这对业务人员来说，是一个相当有吸引力的特性，而且对技术人员来说，也是一件相当不错的事情。人生苦短，请尽量不重写。

2. 后端解耦，前端聚合

后台微服务的初期有一个很大的卖点：使用不同的语言、技术栈来开发后台应用。事实上，采用微服务架构的组织和机构，一般都是大中型规模的。相对于中小型组织来说，对于框架和语言的选型要求比较严格，如在内部限定了语言和框架。因此充分使用不同的技术栈，以发挥微服务的优势，几乎是很少出现的。在这些大型组织机构里，采用微服务的原因主要还是，使用微服务架构可以解耦服务间的依赖，如图 9-2 所示。

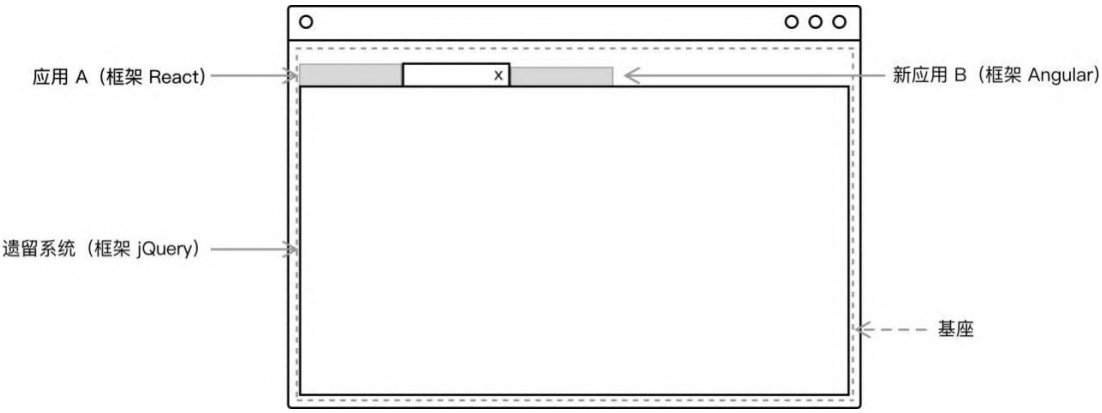

图 9-2

而在前端微服务化上，则恰恰与之相反，人们更想要的结果是聚合前端应用，尤其是那些 To B（to Bussiness，面向企业）的应用。

最近几年，移动应用出现了一种趋势，即用户不想装那么多应用了。而一家大的商业公司，往往会提供一系列的应用。这些应用从某种程度上反映了这家公司的组织架构。然而，在用户的眼里，他们就是一家公司，他们就只应该有一个产品。相似的，这种趋势也在桌面 Web 出现。聚合成为客户端的一个技术趋势，实现前端聚合的就是微前端架构。

3. 热闹驱动开发

所谓热闹驱动开发指的是，软件开发团队所做的软件架构或技术栈的决策，其中很多决策并没有经过踏实的研究和对目标成果的认真思考，而是不准确的意见、社交媒体的信息，或者就是些“热闹”的玩意。

换句话说，“流行”就应该采用某个新的技术和架构，在各种会议、文章中不断被提及，不经过细致研究就直接采用这样的技术。典型的场景有，项目的领导参加了一个会议，或者某一核心成员了解到一个新的技术、架构，便想尝试使用这种技术。多数情况下，往往是经过了一定的考虑，才会使用这样的技术。但是，我们并不排除一些情况，例如只是为了 KPI 或者是别人觉得好，那便是好。于是在技术社区，常常会看到一些“有意思”的提问：“领导让我来看看 xxx 技术……”。

因为“热闹”去学习一项新的技术，并没有什么问题。新的技术意味着学习，它可以在一定程度上提升技术水平和技术影响力。可是，对于这样一个“热闹”的技术，没有多加研究，便直接在项目上使用，难免会遇到一些挫折。失败了，不免会得出该微前端不好用的结论。

微服务、微前端便是这样一些“热闹”的玩意，可以预见的是，未来将有越来越多的前端应用采用这样的架构。热闹了，对于整个技术社区来说，起着一定的促进作用——变得更加热闹。但是，采用之前记得先看看别人的失败经验，再想方设法进行一些细致的调查：构建原型、测试应用、寻找合适的人等。

微前端可以实现的方式比较多，当前也没有标准的实现方式。在短期的未来，也不会有可以在不同项目都适用的实践。但是，它们的基本原理都是相似的，也都需要详尽的设计。

9.2 微前端的技术拆分方式

从技术实践上，微前端架构可以采用以下几种方式进行：

（1）路由分发式。通过 HTTP 服务器的反向代理功能，将请求路由到对应的应用上。

（2）前端微服务化。在不同的框架之上设计通信和加载机制，以在一个页面内加载对应的应用。

（3）微应用。通过软件工程的方式，在部署构建环境中，把多个独立的应用组合成一个单体应用。

（4）微件化。开发一个新的构建系统，将部分业务功能构建成一个独立的 chunk 代码，使用时只需要远程加载即可。

（5）前端容器化。将 iframe 作为容器来容纳其他前端应用。

（6）应用组件化。借助于 Web Components 技术，来构建跨框架的前端应用。

实施的方式虽然多，但都是依据场景而采用的。在有些场景下，可能没有合适的方式；在有些场景下，则可以同时使用多种方案。

9.2.1　路由分发式

路由分发式微前端，即通过路由将不同的业务分发到不同的独立前端应用上。其通常可以通过 HTTP 服务器的反向代理来实现，或者通过应用框架自带的路由来解决，如图 9-3 所示。

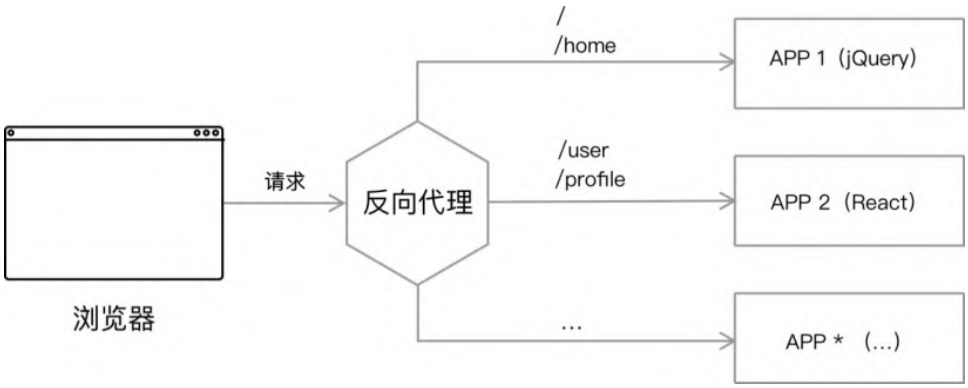

图 9-3

就当前而言，路由分发式的架构应该是采用得最多、最容易的“微前端”方案。但是这种方式看上去更像是多个前端应用的聚合，即我们只是将这些不同的前端应用拼凑到一起，使他们看起来像一个完整的整体。但它们并非是一个整体，每当用户从 A 应用转换到 B 应用的时候，往往需要刷新一下页面、重新加载资源文件。

在这个架构中，我们只需要关注应用间的数据传递方式。通常，我们只需要将当前的

用户状态，从 A 应用传递到 B 应用即可。如果两个应用在同一个域里运行，就更加方便了，它们可以通过 LocalStorage、Cookies、IndexedDB 等方式共享数据。值得注意的是，在采用这种应用时，缺少了对应用状态的处理，需要用户重新登录，这种体验对用户来说相当不友好。

9.2.2 前端微服务化

前端微服务化，是微服务架构在前端的实施，每个前端应用都是完全独立（技术栈、开发、部署、构建独立）、自主运行的，最后通过模块化的方式组合出完整的前端应用。其架构如图 9-4 所示。

图 9-4

采用这种方式意味着，一个页面上同时存在两个及以上的前端应用在运行。而路由分发式方案则是，一个页面只有唯一一个应用。

当我们单击指向某个应用的路由时，会加载、运行对应的应用。而原有的一个或多个应用，仍然可以在页面上保持运行的状态。同时，这些应用可以使用不同的技术栈来开发，如页面上可以同时运行 React、Angular 和 Vue 框架开发的应用。

我们这样实施的原因是，不论基于 Web Components 的 Angular，还是 VirtualDOM 的 React，都是因为现有的前端框架离不开基本的 HTML 元素 DOM。因此，我们只需要做到如下两点：

第一点，在页面合适的地方引入或者创建 DOM。

第二点，用户操作时，加载对应的应用（触发应用的启动），并能卸载应用。

对于第一点，创建 DOM 是容易解决的。而第二点，则一点儿也不容易，特别是移除 DOM 和相应应用的监听。当我们拥有一个不同的技术栈时，我们需要有针对性地设计出一套这样的逻辑。

同时，我们还需要保证应用间的第三方依赖不冲突。如应用 A 中使用了 z 插件，而应用 B 中也使用了 z 插件，如果一个页面多次引入 z 插件会发生冲突，那么我们应该尝试去解决这样的问题，可以通过向上游开发者提 Pull Request 来修复这个问题。

9.2.3　组合式集成：微应用化

微应用化是指，在开发时应用都是以单一、微小应用的形式存在的，而在运行时，则通过构建系统合并这些应用，并组合成一个新的应用，其架构如图 9-5 所示。

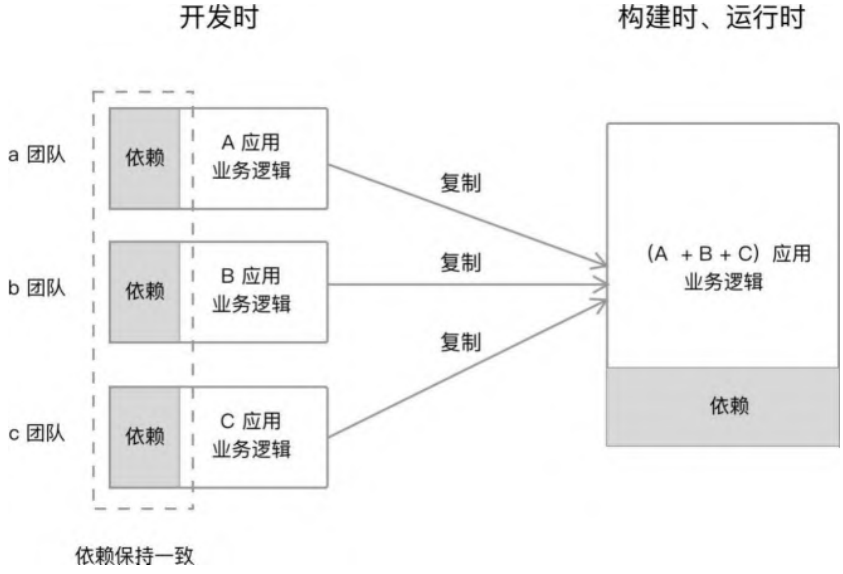

图 9-5

微应用化大都是以软件工程的方式来完成前端应用的开发的，因此又可以称之为组合式集成。对于一个大型的前端应用来说，采用的架构方式往往是通过业务作为主目录的，然后在业务目录中放置相关的组件，同时拥有一些通用的共享模板，例如：

```
├── account
├── dashboard
├── reports
├── ...
└── shared
```

当我们开发一个这样的应用时，从目录结构上看，业务本身已经被拆分了。我们所要做的是，让每个模块都成为一个单独的项目，如将仪表盘功能提取出来，加上共享部分的代码、应用的基本脚手架，便可以成为一个单独的应用。拆分出每个模块之后，便只需要在构建的时候复制所有的模块到一个项目中，再进行集成构建。

微应用化与前端微服务化类似，在开发时都是独立应用的，在构建时又可以按照需求单独加载。如果以微前端的单独开发、单独部署、运行时聚合的基本思想来看，微应用化就是微前端的一种实践，只是使用微应用化意味着我们只能使用唯一的一种前端框架。大团队通常是不会同时支持多个前端框架的。

9.2.4 微件化

微件（Widget），是一段可以直接嵌入应用上运行的代码，它由开发人员预先编译好，在加载时不需要再做任何修改或编译。而微前端下的微件化则指的是，每个业务团队编写自己的业务代码，并将编译好的代码部署（上传或者放置）到指定的服务器上。在运行时，我们只需要加载相应的业务模块即可。在更新代码的时候，我们只需要更新相应的模块即可。如图 9-6 所示是微件化的架构示意图。

在非单页面应用时代，要实现微件化方案是一件特别容易的事。从远程加载 JavaScript 代码并在浏览器上执行，生成对应的组件嵌入页面。对于业务组件也是类似的，提前编写业务组件，当需要对应的组件时再响应和执行。在未来，我们也可以采用 WebComponents 技术来做这样的事情。

而在单页面应用时代，要实现微件化就没有那么容易了。为了支持微件化，我们需要做下面一些事情。

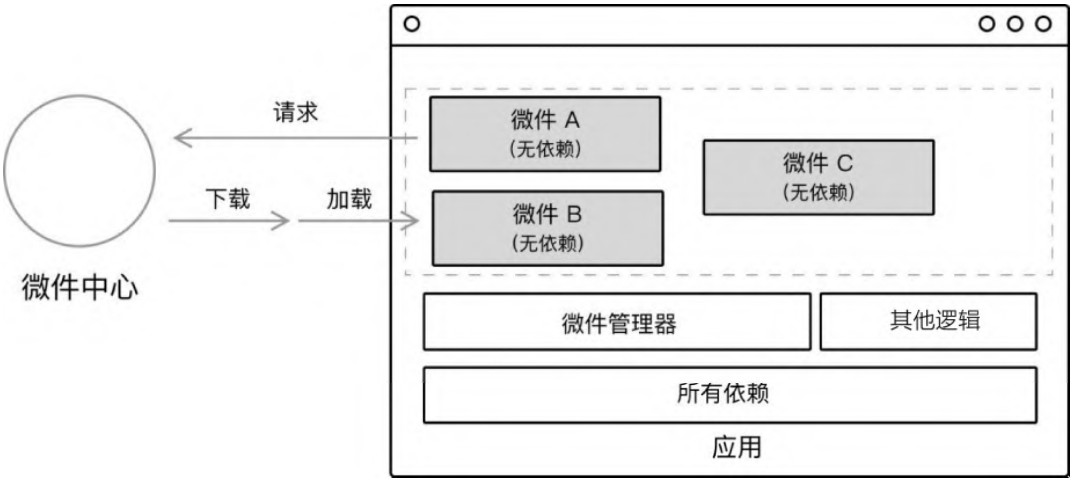

图 9-6

（1）持有一个完整的框架运行时及编译环境。这用于保证微件能正常使用，即可调用框架 API 等。

（2）性能受影响。应用由提前编译变成运行时才编译，会造成一些性能方面的影响——具体视组件的大小而定。

（3）提前规划依赖。如果一个新的微件想使用新的依赖，需要从上游编译引入。

此外，我们还需要一个支持上述功能的构建系统，它用于构建一个独立的微件模块。这个微件的形式如下：

◎　分包构建出来的独立代码，如 webpack 构建出来的 chunk 文件。

◎　使用 DSL 的方式编写出来的组件。

为了实现这种方式，我们需要对前端应用的构建系统进行修改，如 webpack，使它可以支持构建出单个的代码段。这种方式的实施成本比微应用化成本高。

9.2.5　前端容器：iframe

iframe 作为一个非常“古老”的、人人都觉得普通的技术，却一直很管用。它能有效地将另一个网页/单页面应用嵌入当前页面中，两个页面间的 CSS 和 JavaScript 是相互隔离的——除去 iframe 父子通信部分的代码，它们之间的代码完全不会相互干扰。iframe 便相

当于创建了一个全新的独立的宿主环境，类似于沙箱隔离，它意味着前端应用之间可以相互独立运行。

当然采用 iframe 有几个重要的前提：

◎ 网站不需要 SEO 支持。

◎ 设计相应的应用管理机制。

如果我们做一个应用平台，会在系统中集成第三方系统，或多个不同部门团队下的系统，显然这仍然是一个非常靠谱的方案。此外，在上述几个微前端方案中，难免会存在一些难以解决的依赖问题，那么可以引入 iframe 来解决。

无论如何当其他方案不是很靠谱时，或者需要一些兼容性支持的时候，只能再度试试 iframe。

9.2.6 结合 Web Components 构建

Web Components 是一套不同的技术，允许开发者创建可重用的定制元素（它们的功能封装在代码之外），并且在 Web 应用中使用它们。

真正在项目上使用 Web Components 技术，离现在的我们还有些距离，可是结合 Web Components 来构建前端应用，是一种面向未来演进的架构。或者说在未来，可以采用这种方式来构建应用。比如 Angular 框架，已经可以将当前应用构建成一个 Web Components 组件，并在其他支持引入 Web Components 组件的框架中使用，如 React。我们还可以使用 Web Components 构建出组件，再在其他框架中引入。

为此，我们只需要在页面中通过 Web Components 引入业务模块即可，其使用方式类似于微件化的方案，如图 9-7 所示。

目前困扰 Web Components 技术推广的主要因素在于浏览器的支持程度。在 Chrome 和 Opera 浏览器上，对 Web Components 支持良好，而对 Safari、IE、Firefox 浏览器的支持程度，并不是很理想。有些不兼容的技术，可以引入 polyfill 来解决，有些则需要浏览器支持。

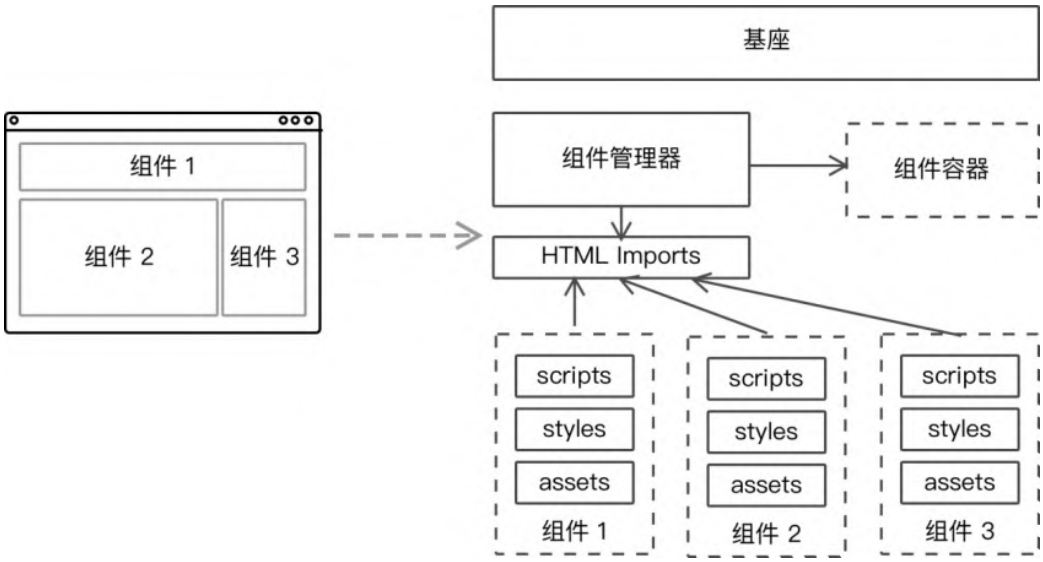

图 9-7

9.3　微前端的业务划分方式

与微服务类似，要划分不同的前端边界不是一件容易的事。就当前而言，以下几种方式是常见的划分微前端的方式：

◎　按照业务拆分。

◎　按照权限拆分。

◎　按照变更的频率拆分。

◎　按照组织结构拆分。

◎　跟随后端微服务划分。

因为每个项目都有自己特殊的背景，所以切分微前端的方式就不一样。即使项目的类型相似，也存在一些细微的差异。

9.3.1 按照业务拆分

在大型的前端应用里，往往包含了多个业务。这些业务往往在某种程度上存在一定的关联，但并非是强关联。如图 9-8 所示是一个常见的电商系统的相关业务[1]。

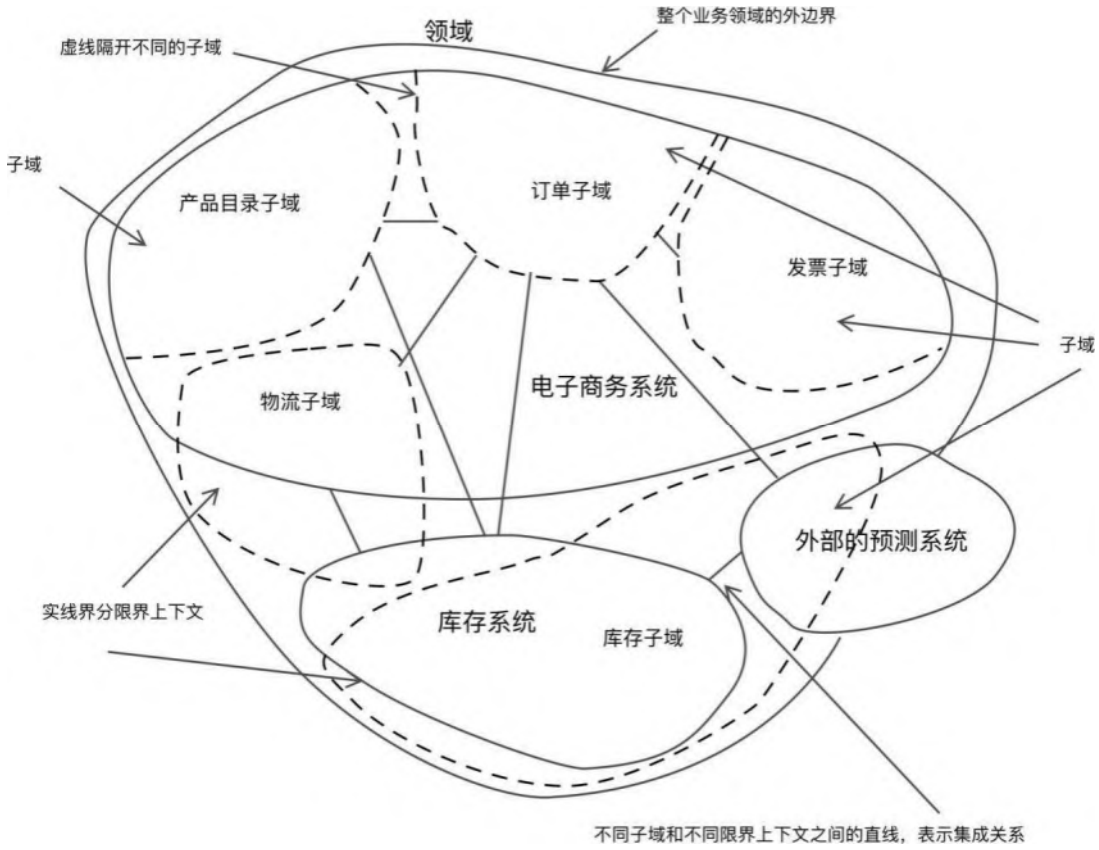

图 9-8

在这样的一个系统里，它可能同时存在多个系统：电子商务系统、物流系统、库存系统等。每个系统都代表自己的业务，它们之间的关联可能并不是很紧密——对于前端应用来说，只需要一个系统内对象的 ID，加上用户的 Token，便能轻松地从一个系统跳转到另

[1] 出自《实现领域驱动设计》

外一个系统中。这种业务本身的高度聚合，使得前端应用的划分也变得更加轻松。

如果业务间本身的耦合就比较严重（如一个电子商务的运营人员，可能需要同时操作订单、物流等多个系统），那么要从前端业务上分离它们，就不是很容易。

因此，对于由业务性质决定的应用，往往只能依据业务是否隔离来进行拆分。

9.3.2　按照权限拆分

对于一个同时存在多种角色及多种不同权限的网站来说，最容易采用的方案就是通过权限来划分服务和应用。尤其这些权限在功能上是分开的，也就没有必要集中在一个前端应用中。

在一个带后台管理功能和前台展示页面的网站里，它们的功能有时候是绑定在一起的，如各种 CMS、博客应用 WordPress；而在一些大型系统中，它们往往又是独立的，有独立的入口来访问后台，有独立的入口来访问前台。多数时候，这取决于大部分用户是否同时拥有两种权限？如果只有管理员拥有后台权限，那么分开是一种更好的选择。因此，多数应用会在项目创建的初期将管理系统划分出去。

但是对于初始时期的管理来说，往往不会有这种划分方式。为了方便管理人员使用，它们需要结合在一起。可是随着后台管理的功能越来越多，应用会变得越来越臃肿。特别在单页面应用中，出于组件复用等目的，往往会将其设计在同一个前端工程中。

是否按照权限来划分应当取决于应用是否臃肿，或者是否正在变得臃肿，导致难以维护。还需要考虑是否为每种角色和权限划分出不同的前端应用。如果只有一种权限的功能比较高，而其他权限业务少，那么是否就只拆分成前台与后台两部分。

9.3.3　按照变更的频率拆分

在一个前端应用中，并非所有模块和业务代码都在不断地修改、添加新的功能。不同的业务模块拥有不同的变更频率。有些功能可能在上线之后，因为用户少而几乎不修改；有些功能则可能为了做而做，即证明有这个技术能力，或者有这个功能。而有一些功能，因为是用户最常用的，所以在不断迭代和优化中。因此，可以依照变更频率来拆分前端应用。

不常用的功能，虽然业务少、变更少导致代码也相对较小，但是因为非核心业务数量

多，从应用中拆分出去也更容易维护。比如 Word 这样的文字处理软件，我们日常使用的功能可能不到 10%。而其他一些专业性的需求，则仍然有 90%的空间，它们也需要花费大量的开发时间。若是将应用中频繁变更的部分拆分出来，不仅更容易维护其他部分的代码，还可以减少频繁的业务修改给其他部分带来的问题。

经常变更的业务也可以进一步进行拆分——拆分成更多的前端应用或者服务。使用变更的频率进行拆分的前提是，我们使用数据统计来计算各部分的使用情况。对于一个大型的前端应用来说，这部分几乎是不存在问题的。

9.3.4 按照组织结构拆分

如“康威定律”所说，“团队的组织方式必然会对它产生的代码有影响。”既然如此，就会存在一种合理的微前端划分方式，即根据不同团队来划分不同的微前端应用及服务。

对于后端来说，按照组织结构拆分服务，几乎是一个默认的做法。团队之间使用 API 文档和契约，就可以轻松地进行协作。对于前端应用来说，同样可以采用这种方式来进行。

这时，作为架构的提出方和主要的核心技术团队，我们需要提供微前端的架构方案。如使用路由分发式微前端，需要提供一个 URL 入口；使用前端微服务化，需要提供一个 API 或者接入方式，以集成到系统中。

值得注意的是，它与业务划分方式稍有区别，一个团队可能维护多个业务。如果某些业务是由一个团队来维护的，那么在最开始的阶段，他们可能倾向于将这些业务放在同一应用中。然后，由于业务的增多或者业务变得复杂，则会进一步拆分成多个应用。

对于跨团队协作来说，集成永远都是一个复杂的问题。尤其在团队本身是异地开发的情况下，沟通就变成一个麻烦的问题。技术问题更适合于当面讨论，如指着代码或页面进行讨论。一旦有一方影响了系统构建，就需要优先去解决这个问题。

9.3.5 跟随后端微服务拆分

幸运的是，与微架构相关的实施，并不只有前端才有，往往是后端拥有相应的实施，前端项目才会进行进一步的拆分。而一旦后端拥有相关的服务，前端也可以追随后端的拆分方式。

然而，后端采用的拆分方式，并不都适合于前端应用——可能多数时候都不适合。如

后端可能采取聚合关系来划分微服务，这时对于前端应用来说并没有多大的启发，但是有些时候还是可以直接采用一致的拆分模型。毕竟如果在后端服务上是解耦的，那么在前端业务上也存在一定解耦的可能性。

9.3.6　DDD 与事件风暴

在后端微服务（Microservices）架构实践中，常常借助于领域驱动设计（Domain Driven Design，DDD）进行服务划分。DDD 是一套综合软件系统分析和设计的面向对象建模方法。DDD 中的一个限界上下文（Bounded Context），相当于一个微服务。而识别限界上下文的核心是，识别出领域的聚合根，这时便依赖于事件风暴来进行。

事件风暴（Event Storming）是一项团队活动，旨在通过领域事件识别出聚合根，进而划分微服务的限界上下文。事件风暴就是把所有的关键参与者都召集到一个很宽敞的屋子里来开会，并且使用便利贴来描述系统中发生的事情。它们会通过以下步骤来确定各个业务的边界，同时划分出每个服务：

（1）寻找领域事件。

（2）寻找领域命令。

（3）寻找聚合。

（4）划分子域和限界上下文。

由于篇幅所限，这里就不展开详细的介绍了，有兴趣的读者可以寻找相关资料，或者从本章代码中找到 README 来阅读相关的资料。

9.4　微前端的架构设计

有了微服务之后，遇到一个新的项目，总会考虑是否需要微服务，是否有能力应对微服务带来的技术挑战。在有了微前端之后，也有同样的疑问，我们是否真的需要微前端？我们是否能应对微前端带来的技术挑战？毕竟，没有银弹。在考虑是否采用一种新的架构的时候，除了考虑它带来的好处，还要考量存在的大量风险和技术挑战。微前端，也是这样一个技术架构，和微服务一样，要进行实践，并做好一系列的技术储备。

9.4.1 构建基础设施

在基础设施上，微前端架构与单体应用架构有相当大的差异。在单体应用里，共享层往往只有一个。而在微前端架构里，共享层则往往存在多个，有的是应用间共用的共享层，有的是应用内共用的共享层。在微前端设计初期，构建基础设施要做如下几件事情：

- 组件与模式库。在应用之间提供通用的 UI 组件、共享的业务组件，以及相应的通用函数功能模块，如日期转换等。
- 应用通信机制。设计应用间的通信机制，并提供相应的底层库支持。
- 数据共享机制。对于通用的数据，采取一定的策略来缓存数据，而不是每个应用单独获取自己的数据。
- 专用的构建系统（可选）。在某些微前端实现里，如微件化，构建系统用于构建出每个单独的应用，又可以构建出最后的整个应用。

这些技术实践，只是一些相对比较通用的内容。对于不同的微前端方案来说，又存在一些细微的差异，具体需求我们将在第 10 章中讨论。

9.4.2 提取组件与模式库

系统内有多个应用采用同一框架的微前端架构，模式库作为微前端架构的核心基础，可以用于共享代码。通过之前的组件库、设计与系统相关的实践，我们已经有了一个基础的共享模式库。在这个库里，它会包含我们所需要的基础组件，可以在多个前端应用中使用。同时，结合第 4 章的相关内容为组件库创建版本机制，设计相应的发布周期，以支持整个系统内应用的开发。下面介绍样式、业务组件及共享库。

1. 样式

在实施微前端的过程中经常会遇到一个头疼的问题：样式冲突。如果在一个页面里同时有多个前端应用，那么就会存在以下几种形式的样式：

- 组件级样式，只能用于某一特定组件的样式。
- 应用级样式，在某一个前端应用中使用的样式。

◎　系统级样式，可在该页面中使用的样式，往往会影响多个应用。

对于组件级样式来说，有些框架可以从底层上直接支持组件模式隔离，如 Angular，这是不太需要花费精力考虑的。此外，对于组件库来说，我们也会创建对应的 CSS 前缀来保证唯一性，只要在开发的过程中多加注意即可。

对于应用级样式而言，则需要制定一个统一的规范，可以根据应用名加前缀，如 dashboard-，也可以根据路由来增加相应的前缀，以确保应用本身的样式不会影响到其他应用。此外，我们往往会为这些应用，创建一个统一的样式库，以提供一致的用户体验。

系统级样式，大抵只存在于基座模式设计的微前端架构里。在这种方案里，由基座应用来控制其他应用，也存在部分的样式。在编写这些样式的时候，需要注意对其他应用的影响。此外，它也可以作为统一的样式库承载的应用来使用。

2. 业务组件及共享库

对于在多个应用中使用的业务组件和共享函数，我们既可以提供 NPM 包的方式，又可以提供 git submodule 的方式，引入其他应用中。

对于通用的组件，它在开发的前期需要频繁地改动，这时可以将其抽取成为子模块（Submodule）的形式在项目中使用。当我们需要的时候，可以轻松地修改，并在其他应用中更新。当这些组件趋于稳定的时候，可以尝试将其作为 NPM 包发布。如果有这种打算，就需要在这个子模块中使用 package.json 及 NPM 的管理方式，方便后期直接扩展。

此外，不得不提及的是，这种类型的修改应当是兼容式修改。在难以兼容的情况下，需要对系统中使用到的部分，逐一进行排查，直到确认已更新下游 API。然后，还要进行相应部分的测试，以确保组件修改带来的影响都已经被修复。

9.4.3　应用通信机制

解决了应用间共享代码的问题，我们还需要设计出一个应用间通信的机制。在微前端架构里，从应用间的关系来看，存在如下两种类型的通信：

◎　同级通信，即挂载在同一个 HTML Document 下的应用间的通信。

◎　父子级通信，即采用 iframe 形式来加载其他应用。

前者往往通过全局的自定义事件（CustomEvent）便可以实现。值得一提的是，在 IE

浏览器上需要使用 Polyfill 兼容库。此外，由于应用之间共享一个 Window，所以我们还可以开发自己的发布-订阅模式组件。与自定义事件相比，它拥有更高的可定制度。

如果采用父子级通信机制，则稍显麻烦一些。普通的父子级通信可以做到以下几方面：

◎ 通过 PostMessage 在父子窗口之间进行通信。

◎ 透过 parent.window 寻找到父窗口，再发出全局的自定义事件。

◎ 当其他应用加载时，将消息发送给父窗口，由父窗口发出自定义事件。

◎ 当其他应用未加载时，先将消息传递给父窗口，再由父窗口进行存储，提供一个获取通信的机制。

在实施过程中，具体采用哪种或哪几种方式，取决于我们在设计的时候有哪些需要。也可以在前期设计出所有的机制，方便后期使用。

值得注意的是，在实现的过程中往往会出现两种结果：

◎ 嵌入业务的特定通信机制。

◎ 剥离业务的通用通信机制。

后者是一个通用的通信机制，开发成本相对较高。前者则是一个业务绑定的模式，一旦添加新功能，便需要进行修改。

9.4.4 数据管理

单页面应用是指对于业务状态的管理及处理。过去这部分内容往往是由后端来实现的，即由 Session 来维持一个用户的状态，现在这个状态转交给了前端。前端在与后端进行交互时，需要传递大量的状态，这时的状态主要由两个部分来组成：URI（GET 的传递参数会转换到 URI 中）+请求 body。由于一个应用持有这些状态，所以为了实现方便，需要在应用间共享这些数据。

应用的数据管理可以分为两部分，一部分是状态，另一部分则是应用数据。只是从某种意义上来说，状态是一种特殊的应用数据，它更加显式地展示数据。通过上部分的应用通信机制，可以解决部分数据共享问题，通用部分的数据，则可以选择一个合适的数据管理策略。为此，我们需要一个去中心化的管理数据，或者基于基座应用的数据管理机制。

常见的数据交互方式，有以下几种：

◎ URI 参数传递。

◎ 使用 LocalStorage 共享数据。

◎ 其他客户端存储，如 IndexedDB、Web SQL 等。

◎ 服务端存储客户端状态，可以采用 JSON 格式存储。

在这方面，我们很难通过实施一套有效的方案来管理，往往是通过规范来保持一致，如在某些情况下使用 URI 传递参数，在某些情况下应用自身从 LocalStorage 获取。当然，如果设计了自己的安全存储策略，就另当别论了。

9.4.5　专用的构建系统

首先需要声明一下，并非所有的微前端架构都需要一个专用的构建系统。只有那些依赖于构建及构建工具创建出来的微前端应用，才需要设计出一个专用的构建系统，以支撑系统的开发。

如果我们采用了微应用化、微件化的架构方案，就需要设计自己的构建流程、构建系统，每种方式在具体实现上各有差异。微件化需要修改构建工具，如添加对应的构建插件，使它能支持构建出组件包；微应用化则依赖于设计构建流程，而不需要对构建工具进行修改，详细的过程我们会在第 10 章中讨论。

9.5　微前端的架构模式

从微前端应用间的关系来看分为两种：基座模式（管理式）、自组织式，分别对应两种不同的架构模式：

◎ 基座模式。通过一个主应用来管理其他应用。设计难度小、方便实践，但是通用度低。

◎ 自组织模式。应用之间是平等的，不存在相互管理的模式。设计难度大，不方便实施，但是通用度高。

就当前而言，基座模式实施起来比较方便，方案上也是蛮多的。

不论哪种方式，都需要提供一个查找应用的机制，在微前端中称为服务的注册表模式。和微服务架构相似，不论哪种微前端方式，都需要有一个应用注册表的服务，它可以是一个固定值的配置文件，如 JSON 文件，或者是一个可动态更新的配置，又或者是一种动态的服务。它主要做以下一些事情：

- 应用发现。让主应用可以寻找到其他应用。
- 应用注册。即提供新的微前端应用，向应用注册表注册的功能。
- 第三方应用注册。即让第三方应用接入系统中。
- 访问权限等相关配置。

应用在部署的时候，可以在注册表服务中注册。如果基于注册表来管理应用，那么使用基座模式来开发就比较方便。

9.5.1 基座模式

在这种模式的微前端架构中，基座承担了微前端应用的基础与技术核心。基座模式，是由一个主应用和一系列业务子应用构成的系统，并由这个主应用来管理其他子应用，包括从子应用的生命周期管理到应用间的通信机制。

基座模式中的主应用，类似于 API Gateway 的概念，它作为系统的统一入口，负责将对应的请求指向对应的服务。子应用，则是负责各个子模块的业务实现，其架构如图 9-9 所示。

这个主应用，既可以只带有单纯的基座功能，也可以带有业务功能。它所处理的业务功能指的是核心部分的业务功能，如：

- 用户的登录、注册管理。
- 系统的统一鉴权管理。
- 导航菜单管理。
- 路由管理。

◎　数据管理。

◎　通信代理。

……

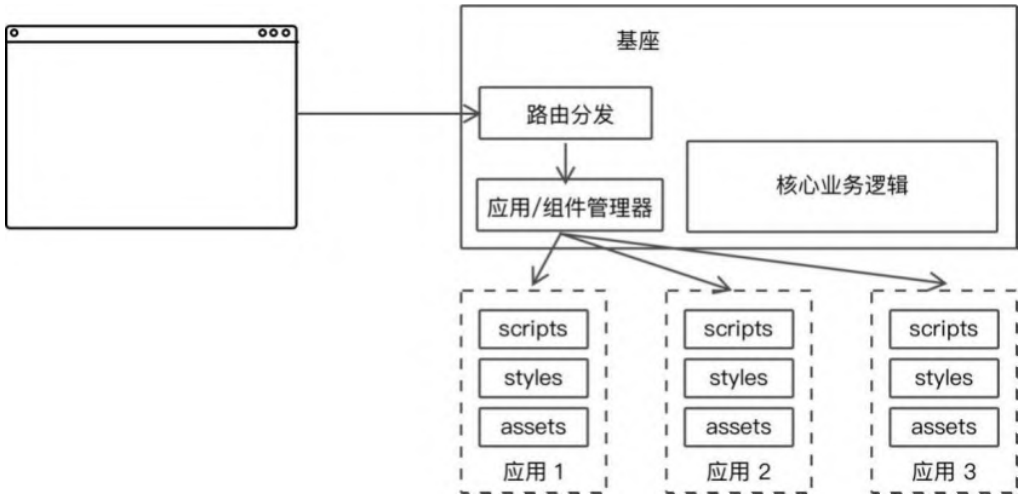

图 9-9

作为应用的基础核心，它还需要：

◎　维护应用注册表。在应用注册表上表明系统有多少个服务、能否找到对应的应用等。

◎　管理其他子应用。如在何时加载应用、何时卸载应用等。

要实现这种模式的微前端架构，只需要设计好对应的应用加载机制即可，因此在实施的时候也比较方便。

9.5.2　自组织模式

自组织指的是，系统内部各子系统之间能自行按照某种规则形成一定的结构或功能。采用这种模式可以使系统内的各种前端应用，都各自拥有一个小型的基座管理功能，也相当于每个应用都可以是基座。

在采用基座模式时，用户要想访问 A 应用需要先加载主应用，然后才能加载 A 应用。

采用自组织模式时，用户想要访问 A 应用则只访问 A 应用，不需要加载主应用，这也因此使它拥有了更高的自主性。

不过多数时候，我们并不需要自组织模式的微前端架构，因为它设计起来复杂、拥有大量的重复代码。

9.6 微前端的设计理念

在笔者实践微前端的过程中，发现以下几点是我们在设计时需要关注的内容。

- 中心化：应用注册表。
- 标识化应用。
- 应用生命周期管理。
- 高内聚，低耦合。

它们也是我们在实践微前端的过程中所要考虑的要点。

9.6.1 中心化：应用注册表

微服务从本质上说应该是去中心化的。但是，它又不能完全去中心化。因为对于一个微服务来说，它需要一个服务注册中心：服务提供方要注册通告服务地址，服务的调用方要能发现目标服务。

对于一个前端应用来说，我们也需要拥有一个应用注册表，它将拥有每个应用及对应的入口。在前端领域里，入口的直接表现形式可以是路由，或是对应的应用映射。应用在构建完成，或者部署到服务器后，应该在这个应用注册表中注册，才能及时地向其他应用提供访问的权限。

以路由形式的注册表为例，当我们添加了一个新的应用时，就相当于在网页上添加了一个菜单链接，用户就能知道哪个页面是可以使用的，也就能访问到这个新的应用。从代码上来说，就是我们需要有一个地方来管理应用：目前存在哪些应用，哪个应用使用哪个路由。

如之前在“架构模式”（9.5 节）中所述，应用注册表可以是一个配置文件，也可以是一个后端服务。由于它是面向前端提供的，其表现形式往往是 JSON，在这个文件中，可以配置微前端加载规则。

9.6.2　标识化应用

标识化应用是指，建立某种规则来区分不同的应用，类似于唯一标识符，即 ID。我们需要这个 ID 来标识不同的应用，以便在安装和卸载的时候，能寻找到指定的应用。

在笔者设计一个微前端框架的时候，曾困扰于如何以合适的方式为每个子应用取一个名称——怎么去规范化子应用。由此，笔者再一次想到了康威定律：

系统设计、产品结构等同组织形式，每个设计系统的组织，其产生的设计等同于组织之间的沟通结构。

换句话说就是，在同一个组织下，不可能有两个项目的名称是一样的。既然如此，那么就由开发系统的人为开发应用的人分配一个 ID 就可以了。

对于第三方应用来说，我们往往会给它们添加对应的前缀，如 *3rd-xxx*，以区分不同的应用。并且第三方应用需要向系统申请，才能接入这个系统中。

如果存在大量的不需要审核的应用，那么可以由系统后台来生成唯一的标识符。

9.6.3　生命周期

当用户单击某个链接时，相应的系统需要加载相应的应用，在这个过程中，还需要用加载动画来响应用户的行为，并创建应用所需要的 DOM 节点，将应用挂载到相应的 DOM 节点上，然后运行应用。当用户不需要这个应用时，我们可以选择卸载应用，或者继续保留应用。这几个步骤所做的事情，就体现了应用的生命周期。

前端微架构与后端微架构的最大不同之处，也在于此——生命周期。微前端应用作为一个客户端应用拥有自己的生命周期，生命周期包括如下 3 个部分：

（1）加载应用。

（2）运行应用。

（3）卸载应用。

在微前端框架 Single-SPA 中设计了一个基本的生命周期（虽然它没有统一管理），其包含如下 5 种状态。

- load：决定加载哪个应用，并绑定生命周期。
- bootstrap：获取静态资源。
- mount：安装应用，如创建 DOM 节点。
- unload：删除应用的生命周期。
- unmount：卸载应用，如删除 DOM 节点、取消事件绑定。

这部分的内容事实上就是微前端的一个难点所在，如何以合适的方式来加载应用呢？每个前端框架都各不相同，其所需要的加载方式也是不同的。当我们决定支持多个框架的时候，便需要在这一部分进行更细致的研究。

9.6.4 高内聚，低耦合

最后，在设计各个子应用及主应用的过程中，需要遵循高内聚、低耦合的原则。

高内聚，即模块内的关系，一个软件模块只由相关性很强的代码组成。设计的过程就是识别、度量模块内的联系，再将相关的行为聚集在一起，把不相关的行为放在别处。如果想要修改模块中某一部分的行为，只需要修改一处代码即可，而不需要修改多处。在实践的过程中，主要基于单一职责和关注点分离两个原则来实现。

低耦合，即模块间的关系。对于微架构系统来说，在服务之间、应用之间如果实现了松耦合，那么修改一个服务和应用就不需要修改另一个服务和应用。除了基座应用，每个应用都不应该关心协作应用的相关信息。

这两个词说起来很简单，但是真正在实现的过程中却相当麻烦。应用之间往往存在一定的依赖关系，要解耦这些依赖，就需要将数据独立出来，并使用通信的方式来传递状态。

9.7 “微”害架构

在对微前端架构有了一定的了解后，在继续实施之前，我们不得不强调一下，在实施

微架构的过程中会出现的一些问题，它会导致架构的弊大于利。如《人月神话》所说，没有银弹。即一个架构有好的一方面，实施不好则有害。

“微”害架构，即微架构以不合理的方式运行着，它采用“微架构”（微服务、App 插件化、微前端等）技术拆分臃肿的单体应用，导致软件架构进一步复杂化，难以维护，使得原本具有优势的微架构出现一些问题。

为什么一个设计良好的架构，会变成一个人人嫌弃的架构呢？我们可以轻松地列举出如下理由：

◎　架构的设计不符合演进的需求。

◎　开始设计时，架构就不合理。

◎　架构本身是合理的，后继的开发者能力不足。

◎　架构本身是合理的，然而开发的过程中走歪了。

关于能力不足导致的架构问题就不进一步展开讨论了，我们主要讨论的是“走歪了的流程”，它才是导致“微”害架构的元凶。在开发能力完备的情况下，架构走向不合理的原因是 KPI。由 KPI 导向的系统架构设计，必然会出现一定的不合理性。

◎　“我们的竞争对手采用了微服务架构，你看看我们有没有办法也用这个架构？”

◎　“隔壁开发小组的团队使用了微前端架构，我们也上这个架构吧？”

◎　“小李啊，听说最近微服务很火啊，你看能不能用到我们的项目上？”

……

如果只是吐槽 KPI，对于读者来说一定是没有价值的，因为 KPI 是一定要有的。从这种角度来看，架构是不是最好的不重要，重要的是大家都满意——只有领导满意，才可以多涨工资；只有被团队采用，才可以向外宣传，吸引新成员。

那么到底是怎么产生“微”害架构的呢？从定义上我们可以知道，架构本身应该是合理的，只是出现了一些问题。问题的原因多种多样，如没有因地制宜地实施架构——A 项目有 50 个后端开发，后端由 20 个微服务组成；B 项目有 10 个后端开发，也由 20 个微服务组成。每次上线的时候，不得不出动所有的人，上线到半夜才算完成。

下面让我们了解、回顾一下系统的架构形成。

9.7.1 微架构

在过去的几年里，笔者曾参与了一系列“微架构”相关系统的开发。对于不同类型的应用程序，它们的改变方式也有所不同。后端应用从单体应用拆分成一个一个微服务，它们对于用户来说是不可见的，但对于自身的客户端而言，仍然是一个整体；前端和移动应用则正在从一个一个的应用，整合成一个大的整体。然而客户端又稍有不同，它们需要实现在开发时拆分，构建时聚合。与微架构开发类似的有：

- ◎ 后端拆分。其典型的形式是微服务，以微服务的实现来说，不同部门的服务也往往是独立存在的，在架构上后台本身也是互不干扰的。只要保证接口 API 是正常的，就不需要关心其项目背后的技术栈。将其与 App 和前端进行对比，就会发现后端微服务是特别容易实现的。组织内部的不同开发团队，只需要保障 API 是稳定可用的、API 是丰富的，不需要过多地讨论会议。
- ◎ App 拆分。App 存在多种容器，有插件化、组件化、小程序等不同的方案。由于一个大型组织的 App 所承载的功能相当丰富，其有可能拥有不逊色于操作系统的代码量。要实现跨团队共同在一个代码库开发，并不是一件容易的事。因此采用一个 App 平台的方案，让不同部分的业务可以在上面运行，是一个更符合组织架构的开发模式。
- ◎ 前端拆分。大规模前端应用的开发历史要比 App 短得多，但是也出现了一系列的拆分方案：前端微服务化、微应用化、微件化，等等。前端走向微前端架构的原因，除了庞大的单体应用，还有一部分是要聚合旧的遗留应用。我们希望旧的遗留应用，能不经修改或者少量的修改，就可以嵌入新的系统中。

如果读者也同笔者一样走过弯路，就会发现微架构对于人员素质的要求。对于某些组织来说，不合理地采用微架构，可能会带来更多的问题，如前端基础设施不完善，会导致各个应用间带有大量的重复代码。

9.7.2 架构的演进

回过头来看，大型项目的架构演进，无非就是：

（1）我有了一个 Idea，快速地设计出原型 1.0 版本，并推向市场，效果还不错，赚到一些钱。

（2）市面上的竞争对手越来越多，为了赢得这场战争，我们在 2.0 版本的系统里添加了越来越多的功能，一个臃肿的单体应用。

（3）随着时间的推移，添加新功能的难度越来越大，于是我们设计出了 3.0 版本的系统——在今天这个节点里，就是微服务架构。

这就是《Linux/Unix 设计思想》一书中所提到的三个系统，毫无维和感：

（1）在背水一战的情况下，人类设计出了第一个系统。

（2）专家们在第一个系统的基础上，做出了伟大而臃肿的第二个系统。

（3）受累于第二个系统的人，设计出了第三个系统。

可故事并不能因此而结束。随着时间的推移，受益于第三个系统的人们，又有一部分组织会回到第二个系统。

大部分组织会意识到三个系统的存在。于是，在根据 Idea 设计一个新系统的同时，采用了第三个系统的架构，然后出错了——因为他们缺少专家。这里的专家并非单纯指技术方面的专家，而是技术专家 + 业务专家，他/她可以是一个懂业务的技术专家，或者是一个懂技术的业务专家。架构在底层是由技术实现的，而在顶层则是由业务代码实现的。

业务层的耦合直接影响了技术上的耦合。这个困境导致了技术架构在不断变化。

9.7.3　微架构带来的问题

在不同的微架构领域里，他们有着不同的问题。在考察是否使用某项技术的时候，我们就会考虑这些问题。而在这个时候，我们的第一答案是：“直接使用这种架构，具体遇到什么问题再解决”。是的，现在让我们来戳破这些问题，看看有没有对应的解决方式。

1. 后端：微服务到应用

实施过后端微服务的开发人员，想必对微服务也有一系列的看法。在不正确的实施之后，不少人会对微服务有所不满。尽管如此，他们都会承认微服务在隔离服务之间的价值。但是当带来的问题远大于收益时，微服务就变得不值得采用了——要么问题不好解决，要么问题解决不了。早在 2014 年，笔者翻译了一篇微服务代价的相关文章《微服务架构——不是免费的午餐》，其中提到的多数都是技术问题。

在项目上实施的时候，笔者发现有一些额外的问题：

◎ 代码架构。在不同的组织之间采用微服务架构，对于相互之间都是有好处的。但是在一个小型的内部组织里，掌握代码独立的微服务数量就值得考虑了——既然不需要和别人沟通，为什么需要在十多个 repo 间切换？

◎ 部署流程。微服务讲究独立开发、独立部署，如果一个微服务不能独立部署（有些固定的上线周期），那么部署就变成一件痛苦的事。如果在一次 release 计划里，我们需要同时上线十几个服务，就变得……

那么，在这种情况下，这些开发者就在考虑，采用微服务架构是否值得。实际上这个问题的本质可能是，是否有独立拆分微服务的必要？或者采取集成构建的方式，在开发和构建时来独立两个应用，在部署时只部署一个应用。

2. 客户端：插件化、组件化

App 可以分为插件化和组件化两种方式。插件化是集成构建完成后的 APK 包，而组件化（Module）则是集成构建后的模块（Lib）。Web 前端的拆分则可以是微前端、微应用和跨框架组件化几种形式。

不论采用哪种方式，每一个微小的部分，最后都可以在同一框架内运行。不同的实现方式，稍有一些区别。有些架构实施起来很轻松，但是在维护后却变得相当麻烦。实际上往往有很多架构在实施的时候就很麻烦，更不用说在后期维护了。

业务的价值往往以大前端的形式呈现，因此移动应用在代码的拆分上，往往比后端容易。不论采用动态加载、UI 跳转、路由或者其他形式，都要考虑如何来拆分代码。于是我们就遇到问题了：

一个简单的业务，是否有必要成为一个简单的服务？当我们有一系列简单的功能时，它们真的要独立运行。它们是否有必要成为多个简单的组件？它们不能是一个组件来接受多个参数吗？

面对这些问题，我们可以采取合适的代码拆分策略，以便在未来合适的时候进行拆分。限制我们这么做的主要原因是，后期拆分的时候会面临巨大的挑战。

9.7.4　解决方式：可拆分式微架构

要合并多个后端服务，就要统一它们的技术栈，但是他们的技术栈会是一样的吗？在多数组织里，技术异构是存在的——只存在于少量数据服务中。在大多数情况下，他们都是使用同一种技术进行构建的。同一个语言或同一个框架，可以降低学习成本。一些不易实现的特性如机器学习、深度学习等，则使用 Python 等拥有丰富 AI 生态的技术栈来构建。

此时，我们只需要保证可以成功构建即可，如微应用化就是一种构建时集成的方式。当然，这有可能又一次回到“锤子定律”的问题中了。

这种方式并不违反微服务的一系列原则，服务自治、专注的服务等。如果我们主要修改的代码在某一个特定的服务里，那么我们只需要将它们拆分出来独立部署。同样的道理适用于 App 插件化和前端微应用化。如果没有规范化的开发流程，架构之间可能会出现进一步的耦合。

9.8　小结

在本章中，我们引入了微前端架构，介绍了我们为什么需要微前端？其中的缘由有：遗留系统迁移、聚合前端应用及热闹驱动开发。接着，我们了解了单体前端应用拆分成微前端架构的几种方式：

- 路由分发式。
- 前端微服务化。
- 微应用。
- 微件化。
- 前端容器化。
- 应用 Web Components 化。

不同的方式都有各自的适用场景。而技术和业务的关系决定了我们在拆分的时候往往需要从业务的角度来考虑技术问题。因此，我们又进一步介绍了如何从业务的角度来拆分

前端应用：

- 按照业务拆分。
- 按照权限拆分。
- 按照变更的频率拆分。
- 按照组织结构拆分。
- 跟随后端微服务进行划分。

同时，我们还介绍了微前端设计的知识——基础设施构建、架构模式及设计理念，它将有助于我们设计出更好的微前端架构应用。最后，我们还了解了不合理使用微前端架构将会带来的一些问题，这些内容将有助于我们在第 10 章中实践微前端架构。

10

第 10 章
微前端实战

在第 9 章中，我们介绍了微前端一系列的相关知识，都是在为本章做铺垫。微前端作为一个新的前端架构，在实施的过程中会遇到一个又一个挑战。或许第 9 章中的内容对我们来说有些抽象，那么在这一章中，我们将在实践微前端的过程中，探索微前端架构带来的挑战。

无论是对于从头开发的微前端应用，还是正在迁移的微前端应用，这种架构的演进都需要一个实施时间。出于以下目的，我们需要快速“发布”MRV 版本（最小可发布版本）的微前端架构应用：

- ◎ 架构在项目中的可行性验证。
- ◎ 向领导和团队证明架构的可能性。
- ◎ 增强团队对于新技术的信心。

微前端实战更像是将 Demo 应用结合到我们的项目中，成为一个真正的应用。在既有

项目里，如果我们计划将前端应用，拆分成十几个子应用的微前端架构，那么在这个阶段，我们应该将那个最小的、成本最低的应用迁移到微前端架构中。再依我们的经验，来判断迁移其他应用，可能会遇到的问题。同时，提前做好相应的技术储备。

在新的项目里，我们反而可以将带通用性的、成本最低的应用，使用微前端架构来编写。但是无论如何，我们所做的是一个最小的试错成本：

（1）有些问题会在第一个阶段发生，我们可以留出空间和计划一一解决。

（2）如果问题太多那么可以放弃这个方案。

在我们实施的过程中会遇到一个又一个问题，正是这些问题让我们考虑，如何更好地完善一个微前端架构的应用。我们在团队中会听到各种声音，如“微前端并不是那么好用”，如“我们应该回到原来的方案上”。如果因为技术困难、实施困难而走向原来的架构，多少会有些遗憾；如果只是团队成员不能适应这种变化，那么还是要继续往这个方面前进。有时候我们不得不正视这样的挑战，偶尔在不舒适的区域，可以提升团队的适应能力。

总之，无论如何我们会遇到一系列的挑战，还有各种内部的技术讨论会议。如果大家都认可微前端真的适合这个项目，那么就一点点地往前走。

注意：在本章的小结里包含了各种微前端方式的优缺点对比，有需要的读者可以根据对比的结论选择自己需要的架构方式。

10.1 遗留系统：路由分发

在一个单体前端、单体后端应用中，有一个典型的特征，即路由是由框架来分发的，框架将路由指定到对应的组件或者内部服务中。微服务在这个过程中做的事情是，将调用由函数调用变成了远程调用，如远程 HTTP 调用。微前端与之类似，它是将应用内的组件调用变成更细粒度的应用间组件调用，即原先我们只是将路由分发到应用的组件执行，现在则需要根据路由来找到对应的应用，再由应用分发到对应的组件上。

采用路由的方式将请求分发到不同的应用上，并不是一个新的事物。稍微回想一下，我们日常使用的应用，便会发现这种模式相当常见。

10.1.1　路由分发式微前端

路由分发式微前端，即通过路由将不同的业务分发到不同的独立前端应用上。其通常可以通过 HTTP 服务器的反向代理来实现，或者通过应用框架自带的路由来解决。

就当前而言，通过路由分发式的微前端架构应该是采用最多、最易采用的“微前端”方案。但是这种方式看上去更像是多个前端应用的聚合，即我们只是将这些不同的前端应用拼凑到一起，使他们看起来像一个完整的整体，但是它们并不是一个整体，每次用户从 A 应用到 B 应用的时候，需要刷新一下页面。

在几年前的一个项目里，我们当时正在进行遗留系统重写。我们制定了一个迁移计划：首先，使用静态网站生成动态网页；其次，使用 React 计划栈重构详情页；最后，替换搜索结果页。

整个系统并不是一次性迁移过去的，而是一步步往下进行的。因此在完成不同的步骤时，我们就需要上线这个功能，于是就需要使用 Nginx 来进行路由分发。这种重构或者重写系统的方式，又可以称为绞杀者模式，即一小部分一小部分地迁移系统，而非一次性迁移整个系统。

下面是一个基于路由分发的 Nginx 配置示例：

```
# 代码位于 chapter10/router-dispatch 目录下
http {
  server {
    listen      80;
    server_name  aofe.phodal.com;
    location /api/ {
      proxy_pass http://http://172.31.25.15:8000/api;
    }
    location /web/admin {
      proxy_pass http://172.31.25.29/web/admin;
    }
    location /web/notifications {
      proxy_pass http://172.31.25.27/web/notifications;
    }
    location / {
      proxy_pass /;
    }
  }
}
```

在这个示例中，不同页面的请求被分发到不同的服务器上。

接着，我们在别的项目上也使用了类似的方式，其主要原因是**跨**团队的协作。当团队达到一定规模的时候，我们不得不面对这个问题。此外，还有 Angular 跳崖式升级的问题。在这种情况下，用户前台使用 Angular 重写，后台继续使用 Angular.js 等保持现有的技术栈。在不同的场景下，都有一些相似的技术决策。

因此在这种情况下，它适用于以下场景：

◎ 不同技术栈之间差异比较大，难以兼容、迁移、改造。

◎ 项目不想花费大量的时间在这个系统的改造上。

◎ 现有的系统在未来将会被取代。

◎ 系统功能已经很完善，基本不会有新需求。

在满足上面场景的情况下，如果为了得到更好的用户体验，还可以采用 iframe 的方式来解决。由于这种方式相当常见，在这里我们就不详细介绍了，重点让我们关注一下微前端的测试。

10.1.2 路由分发的测试

测试是系统质量的保证。无论是普通的单体架构应用，还是微前端架构，测试都是必不可少的。即使因为组织内的条件限制导致没有时间编写所有的测试，对于关键部分的测试，也会使得系统更加强壮——一旦后期的修改影响了之前的功能，就会在测试中体现出来。因为测试只能保证现有的功能是正常的，并不能保证没有 bug。

依笔者的经验来看，微前端路由分发的测试主要分为两种：

◎ 单元测试。验证生成的跳转 URI 是否符合我们创建的规则。

◎ 集成测试。验证生成的 URI，以及其最后（重定向）的地址是否符合要求。

单元测试用于保障应用内的集成是正确的，而集成测试则用于保障系统内、应用间的集成是正确的。

1. 单元测试：URL 验证

路由分发实际上是一个 URI 生成及跳转的逻辑，其结果是生成一个跳转的 URI。生成

URI 有几种不同的方式：

- ◎ 前端应用内生成。即只有一个前端应用，会根据一定的规则来生成 URI；当出现多处使用时，需要由业务库来实现。
- ◎ 后端代码生成。即生成跳转的 URI 是由后端生成的，这个时候就不是由前端测试来覆盖了。
- ◎ 编写通用库生成。如果我们的代码需要在多处使用，那么就要考虑编写一个通用的业务库来实现。

在笔者经历过的一个项目里，有一个搜索类型的网站，用户进行一次搜索所要经历的跳转方式如下：

- ◎ 首页->列表。在这个过程中，我们需要在 URI 中带上所有的搜索条件。
- ◎ 列表->详情。在跳转到详情页时，我们需要带上详情面的 ID 及一些基本信息。

在用户进行搜索的时候，先由首页生成搜索的 URI，这个生成的 URI 可能是 /location-guangdong-shenzhen-price-5000-to-10000，再带着用户跳转到新的 URI 上。然后，由列表页的相关应用来处理这个逻辑，其所做的事情是解析这个 URI，从 URI 中获取相应的参数。在上述 URI 中带有的信息是地理位置：广东-深圳，价格范围是：5000～10000 元。

对于首页的应用来说，只需要生成这样的 URI 即可，因此测试起来也比较方便。我们只需要采用如 Jasmine 这样的框架便可以验证这部分的功能。

作为接收方的列表应用，需要根据对应的 URI 解析出参数，并根据 URI 中解析的参数对后台发起相应的 HTTP 请求。作为列表应用，我们需要测试 URI 的解析规则是否正确。同样的，这部分解析逻辑，也可以由单元测试进行覆盖。

2. 集成测试：URL 重定向测试

一旦页面跳转规则比较复杂时，就不得不考虑使用 E2E 测试来覆盖这些跳转。E2E 测试与 URL 测试相比，更能测试系统间的跳转是否正常。因此，除了可以验证跳转 URL，还可以验证跳转后页面的 Title 是否正确。

当然，对于一般的微前端应用来说，这种集成测试也是相当重要的。只是对于路由分发类型的微前端架构应用来说，测试路由是一种可靠、便捷的方式。

10.2 遗留系统微前端：使用 iframe 作为容器

在第 9 章中，我们已经介绍了一些适用于 iframe 的情形。当我们在实现一个平台级应用时，会在系统中集成第三方系统，或者集成多个不同部门团队下的系统，显然这是一个不错的方案。典型的场景如将传统的 Desktop 应用迁移到 Web 应用。

在采用 iframe 的时候，我们需要做两件事：

◎ 设计管理应用机制。

◎ 设计应用通信机制。

两者都需要经过一定的设计，而当前并没有完整的解决方案：

加载机制。是指在什么情况下，我们会加载、卸载这些应用。在这个过程中，采用怎样的动画过渡，让用户看起来更加自然。

通信机制。直接在每个应用中创建 postMessage 事件并监听，并不是一个友好的事情。其本身对于应用的侵入性太强，因此通过 iframeEl.contentWindow 去获取 iframe 元素的 Window 对象是一个更简化的做法。此外，我们需要定义一套通信规范：事件名采用什么格式、什么时候开始监听事件、什么时候解绑监听事件等。

有兴趣的读者，可以看看笔者之前写的微前端框架：Mooa（GitHub：https://github.com/phodal/mooa），其在内部系统中设计了一套简单的机制。

虽然这种方案很普通，但它是有效的。当我们尝试了其他几种方案并不是很有效时，最后这一种可能就是我们想要的有效的解决方案。

10.3 微应用化

微应用化是指，在开发时应用都是以单一、微小应用的形式存在的，而在运行时则通过构建系统合并这些应用，组合成一个新的应用。

微应用化与前端微服务化架构类似，它们在开发时都是独立应用，在构建时又可以按

照需求单独加载。如果以微前端的单独开发、单独部署、运行时聚合的基本思想来看，微应用化就是微前端的一种实践。只是使用微应用化意味着我们只能使用唯一的一种前端框架。如果从框架不限的角度来定义，微应用化怕是离微前端有些远，好在大部分团队不会想着同时支持多个前端框架。当然，在一些非核心业务上，可以尝试使用新框架。

除了限制开发人员使用同一个框架，它还有一些缺点：

（1）所有应用的依赖需要统一。一旦依赖版本不一致，可能会带来其他问题。

（2）高度依赖于持续集成。每个子应用在提交的时候，都会重新构建出整个应用。一旦一个子应用出错，系统就会出错。它的优点是：实现简单、成本较低。

10.3.1　微应用化

微应用化的形式是，在构建时以单体应用的形式构建；在运行时，以应用模块的形式存在。从原理上说，我们做的只是从多个项目中复制出代码，然后合并到一个项目中。下面是一个 Angular 应用的目录结构：

```
//代码位于 chapter10/microapps 目录下
├── app
│   ├── app-routing.module.ts
│   ├── ...
│   └── reports.module.ts
├── dashboard
│   ├── ...
│   └── dashboard.module.ts
├── reports
│   ├── ...
│   └── reports.module.ts
└── settings
    ├── ...
    └── settings.module.ts
```

除了主的 app 模块，还包含了 dashboard、settings、reports 三个模块。在微应用化里，上述应用便是最后构建过程中的应用。在构建之前这些模块是以独立 App 的形式存在的。而主的 App 模块下的功能，则可以变成主工程。这时，一共会有四个代码仓库：

◎　主代码库。只包含一个空白的框架式代码，它是一个单独的应用，可以独立构建，构建完成则成为包含另外三个应用的完整工程。

◎ dashboard 应用。在构建时复制对应模块、目录的代码到主工程。

◎ settings 应用。同上。

◎ reports 应用。同上。

当系统开始构建时，我们会从独立的 dashboard 应用中拷贝相应的 dashboard 路径下的代码到上述的这个工程里，然后替换相应的空白应用。在这个 dashboard 应用内，自己又是一个完整的 Angular 应用，它可以独立地开发运行。下面是 dashboard 应用的主要结构：

```
//代码位于 chapter10/microapps/dashboard 目录下
├── app
│   ├── app-routing.module.ts
│   ├── ...
│   └── dashboard
│       ├── dashboard.component.css
│       ├── dashboard.component.html
│       ├── dashboard.component.spec.ts
│       ├── dashboard.component.ts
│       └── dashboard.module.ts
```

同理，对于其他两个应用 settings 和 reports 也是类似的——只需要在构建时，从对应的目录拷贝相应的文件夹并替换即可。

注意：这里的目录只用于演示，在真实的业务场景下，在 app/dashboard 目录下还会包含一系列的子模块。并且，我们还会为这个模块创建相应的路由，以提供对复杂应用的支持。

为了实现以上功能，我们只需要做如下事情：

（1）设计一个合理的目录结构，以容纳不同的应用。

（2）解耦模块间的依赖，使它们独立运行。

第一点我们已经基本实现了，在真实的场景中，可能会根据路由或者应用名来划分应用。对于依赖解耦，有一个很不错的方式是，通过路由懒加载来帮助我们解耦。

路由 LazyLoading，即路由懒加载、惰性加载，它是根据不同的路由来将对应的模块、组件切分成不同的代码块，然后当路由被访问的时候才加载对应模块、组件。在每个路由对应的模块里，模块本身接近于独立，构建前以独立文件夹的形式，构建后以独立代码的形式而存在。即使有一些依赖是不独立的，也可以放到共用模块里。它的这种特性，使得

整体架构无限逼近于我们所需要的微前端架构。

当前的主流前端框架如 Angular、React、Vue 是可以支持的。事实上，路由 LazyLoading 依赖的是 webpack 对于 chunk 的支持。下面是一个 AngularLazyLoading 的路由示例，其对于 Vue 或者 React 也是相似的：

```
//代码位于 chapter10/lazyload-demo 目录下
export const ROUTES: Routes = [
  { path: '', pathMatch: 'full', redirectTo: '' },
  { path: 'dashboard', loadChildren: 'dashboard/dashboard.module#DashboardModule' },
  { path: 'settings', loadChildren: 'settings/settings.module#SettingsModule' },
  { path: 'reports', loadChildren: 'reports/reports.module#ReportsModule' }
];
```

这些模块以相对依赖的方式导入项目，当用户单击/dashboard 时，就会去寻找 dashboard 对应的模块（dashboard/dashboard.module#DashboardModule）。应用在构建的时候会将 module 独立构建成*.chunk.js。这些 chunk 文件只在对应的路由被触发时，才会加载进来并渲染相应的模块。

这种懒加载模式意味着：

（1）不需要依赖于其他模块就可以构建出应用。如我们的 dashboard 应用，在本地开发时只需要有 DashboardModule 相关的代码，其他模块可以不存在，或者以空白应用的形式存在。

（2）可以更新主工程的依赖模块，而不影响系统的整体构建。

再加上组件化的抽离及业务代码的共享，便能完成这种模式的应用开发。

10.3.2　架构实施

从上面的例子来看，这种架构模式并不复杂，它的实施也相对比较简单：

◎　持续集成设计。

◎　设计占位方式。

当然，在第 9 章中提到的组件化和业务共享，这里就不重复描述了。

1. 持续集成设计

与常规的持续构建不同的是，对于这样的一个系统来说，其持续集成的触发机制可以由这几部分构成：

（1）子应用代码更新，触发对应的模块的持续构建。

（2）主应用代码更新，触发整个系统的持续构建。

（3）一旦步骤（1）集成成功，就会触发整个系统的持续构建。

一旦多个项目同时提交代码，就会出现一些问题，整个系统的构建可能会不断触发。我们不得不考虑将这个主应用变成手动构建，或者按时间间隔来构建，如每 30 分钟构建一次。

此外，为了实现在持续集成服务器上合并子应用及主应用，我们还需要一个构建脚本来执行相关的内容。它会执行如下的相关内容：

（1）从远程拉取最新的代码。

（2）使用新的子应用代码替换旧的占位模块。

（3）执行一些文本替换（可选）。

读者可以参考本书相关的示例代码，这部分内容位于 chapter10/microapps/src/scripts 目录下。

2. 设计占位方式

由于应用在编译的过程中依赖于这些模块，所以需要在主工程中引入一些占位模块来代替真实的模块，以避免主工程在编译的过程中出错。

这个占位模式会通过以下方式来体现：

（1）子应用中指向其他子应用的路由。即是否展示完整的页面，是否展示对应的空白路由页面。

（2）主应用中的子应用模块。即以何种方式来展示相关的应用，是否有机制来屏蔽未实现的应用。

上述两种因素并不影响整个系统的构建，但是会影响开发人员的开发体验。对于不同

的团队，其实现方式也略有差异。建议读者在实施的过程中根据自己的需要进行定制。在笔者经历过的项目里，也是经过反复修改才最终选定空白页面展示的占位模式。

10.3.3　测试策略

考虑到微前端架构在实施上的一些特殊性，传统的测试金字塔不能满足需求。只能添加一些额外的测试：

◎　应用依赖一致性检测。

◎　功能模块生成测试。

这些测试可以在一定程度上判断构建是否成功。

1. 依赖一致性测试

鉴于这种模式的独特性，我们需要在不同的子应用间保持一致的依赖版本。因此有必要对依赖版本进行测试和对比，以避免在线上依赖不一致的时候，出现一些意料之外的 bug。

对于前端项目来说，这个依赖管理配置文件就是 package.json。我们只需要从不同的项目中读取这个文件，然后对比其中的版本即可。我们要使每个工程的依赖尽可能保持一致。

2. 功能模块生成测试

由于项目加载模块的方式是通过前端框架自带的 LazyLoad 功能来实现的。理论上，我们就不需要测试 lazyLoad 的功能是否正确。功能模块生成测试包括下面几方面内容：

◎　测试复制的模块能否复制到对应的目录上。

◎　测试生成的模块代码大小是否正常。

◎　E2E 测试。

对于模块是否能正确复制进行测试，最简单的方式是编写脚本，在持续集成的过程中运行测试脚本，如果没有检测到 exit(-1)，则持续集成构建失败。在拥有了自动化测试之后，我们并不需要这种模式。

测试模块代码大小是否合理的原因在于，检测是否正确地替换功能模块。正常情况下每个 chunk.js 文件要大于空白的模块。但是这部分测试依赖于构建阶段的测试日志，同样需要在持续集成中编写脚本，并执行 exit(-1)。

使用 E2E 测试对于微前端或者微服务化架构来说是一种特别有效的方式，它唯一的问题是，运行起来比较慢。

10.4 前端微服务化

尽管我们已经介绍了三种微前端方式，但是从它们相关的实践来看，并不像微服务化的前端应用，为此我们可以进一步了解、采用前端微服务化。如我们在第 9 章中所说，前端微服务化是指，在不同的框架之上设计通信、加载机制，以在一个页面内加载对应的应用。当我们谈及前端服务化时所希望的是：

- 应用可以自动加载、运行，并能够与应用注册表进行联系。
- 每个应用的开发是完全隔离的，开发时互不影响。它可以接入某个框架，以更好地支持构建。

应用配置，针对不同的项目各有不同。有的项目已知应用数量，出于拆分需要而采用配置文件。有的项目应用数量多，并且需要支持第三方应用，需要有一个服务来进行相应的配置和管理。虽然如此，它们也有一定的适用性，如：

- 应用的挂载 DOM 节点。
- 应用的服务地址。
- 应用的唯一标识符。
- 应用的名称。
- 应用所需要加载的脚本文件。

对于应用的隔离，则需要深入不同的框架代码中了解不同框架的生命周期，才能有针对性地写出每个框架的加载机制。好在社区已经有如 Single-SPA 这样的微前端架构方案，它已经有针对不同框架的应用加载示例。

此外，我们需要注意的是：

◎　在加载应用时的事件绑定及应用入口。

◎　在卸载应用时的事件解绑。

首先，让我们关注于如何设计出适合自己的前端微服务化方案。

10.4.1　微服务化设计方案

从相关的前端微服务化方案上来看，我们将其分为两种：

（1）通用型微前端方案。即可以适配不同的前端框架，更适合于迁移型的微前端项目。

（2）定制型微前端方案。即只适配一种前端框架，适合于从头开发的微前端项目。

采用通用型微前端方案或者定制型微前端方案，取决于我们是否进行深度定制？如果我们只是保证旧系统的代码可以直接运行在新的服务上，那么通过通用型方案就能解决问题。而如果我们还需要进行一系列的定制，那么就要编写适合自己的架构加载方案。而定制也意味着，需要投入额外的成本进行研发。

不论哪种方法，目前都采用基座化的方式来加载其他应用。基座化方式可以支持加载不同的前端框架，以及在基座工程上绑定业务逻辑。依这种方案来看，我们可以将基座分为两种模式：

（1）瘦基座。其只包含微前端方案相关的框架代码，而核心的业务代码则是由基座加载过来的。

（2）胖基座。前端方案相关的代码与业务逻辑代码包含在同一个代码库中。

两种模式从本质上看，区别主要在于业务逻辑是否绑定。

1. 基座控制系统

在第 9 章里，我们描述的基座的功能，也是前端微服务架构所需要承担的功能。对于不同项目而言，基座所要承担的功能也有所不同。有的项目需要绑定更多功能，有的项目只需要基本功能即可。如图 10-1 所示是笔者之前经历的一个设计基座相关功能的流程图。

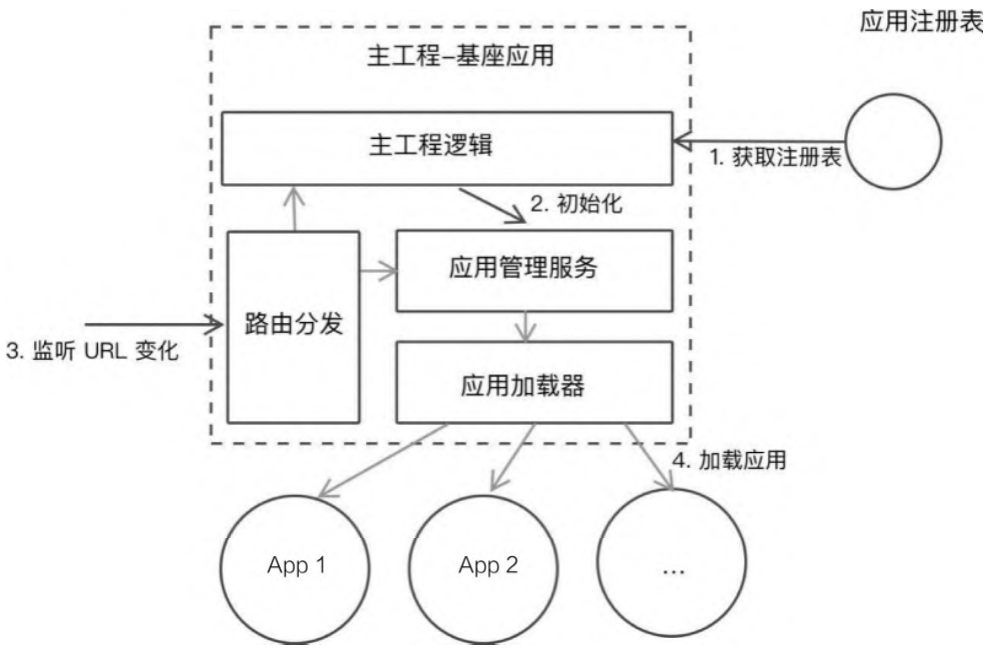

图 10-1

其整体的工作流程如下所示：

（1）主工程在运行的时候，会去服务器获取最新的应用配置。

（2）主工程在获取配置后，将逐一创建应用，并为应用绑定生命周期。

（3）当主工程监测到路由变化的时候，将寻找是否有对应的路由匹配到应用。

（4）当匹配到对应的应用时，则加载相应的应用。

对于笔者的项目而言，基座包含以下功能：

◎ 管理其他子应用。

◎ 负责应用间的通信。

◎ 设计路由的响应机制。

◎ 支持嵌入常规及 iframe 模式。

但是，我们并没有在基座及主工程上维护应用表——前端应用表是一个静态的 JSON 文件。

2. 应用间事件

在应用间的事件机制上，前端微服务化和其他微服务化方案的差距几乎是一致的，并不需要太多的修改和设计。一个 HTML Document 下可能同时存在多个不同的前端框架，需要注意事件名称的规范问题。即我们需要针对 CustomEvent 中的名称进行一定的规范，避免应用间错误地使用同一事件名。因此也可以采用之前的方法，即将应用名作为前缀，这个前缀可以用来处理应用内的事件，这样会减少与全局的通知事件之间的冲突。

3. 设计路由响应机制

前端微服务化还有一个棘手的问题，即路由。如果在一个页面里有多个单页面应用，且应用的路由发生了变化，那么它们会对路由的变化进行响应。如果其中有一个路由，直接将这个存在的路由引向了 404 页面，或者使用了默认的路由，那么应用的行为相当于发生了变化，这种体验对用户来说相当不友好。

为此，需要设计一个路由的响应机制，即什么时候由什么应用来处理路由？下面是笔者在之前的一个项目中采用的方案：

（1）基座应用只负责自己相应的路由，剩下的路由用一个统一的模块来处理，如 Angular 中的入口模块 AppModule

（2）当基座应用的路由发生变化后，会发出对应的路由变化的全局事件。

（3）在子应用启动后，它们将监听是否有相应的路由变化。

下面是一个路由的关联响应示例：

```
    window.addEventListener(MOOA_EVENT.ROUTING_CHANGE, (event: CustomEvent)
=> {
      if (event.detail.app.name === appName) {
        let urlPrefix = 'app'
        if (urlPrefix) {
          urlPrefix = '/${window.mooa.option.urlPrefix}/'
        }
        router.navigate([event.detail.url.replace(urlPrefix + appName, '')])
      }
    })
```

如果发出事件的应用名称和当前的应用名称一致，那么子应用可以路由到相应的页面。

10.4.2 通用型前端微服务化：Single-SPA

Single-SPA 自称是一个 JavaScript 的元框架（Meta-Framework）。它可以用于构建可共存的微前端应用，每个前端应用都可以用自己的框架编写。它能实现如下内容：

- ◎ 在同一页面上使用多个框架（React、AngularJS、Angular、Ember 等），而不用刷新页面。
- ◎ 使用新的框架编写前端代码，而无须重写现有的应用程序。
- ◎ 延迟加载前端应用和代码，用于改善初始加载时间。

Single-SPA 是一种瘦基座的架构模式，这种基座只负责按照生命周期规范来处理应用，并不负责应用的生命周期实现。子应用具体的生命周期是由自身来完成的，基座只负责按自己定义的规范来执行。

1. 基座部分

Single-SPA 的基座部分（在其框架定义里是指根应用程序）只负责定义应用相关的生命周期，以及应用基本的配置。下面是一个 Single-SPA 的根应用程序的代码示例：

```
    //代码位于 chapter10/single-spa-sample/src/root-application 目录下
    import * as singleSpa from 'single-spa';

    singleSpa.registerApplication('app-1', () => import ('app1/app1.js'),
pathPrefix('/app1'));
    singleSpa.registerApplication('app-2', () => import ('app2/app2.js'),
pathPrefix('/app2'));

    singleSpa.start();

    ...
```

如上代码所示，基座并不需要关心子应用的技术栈，由子应用来完成相应的生命周期实现。但是，在某些复杂的前端框架，如 ember，需要加载额外的库才能实现加载功能。在这个示例里的 app-1 是一个 React 框架编写的应用，而 app-2 则是一个 Angular 框架编写的应用。

对于基座应用来说，只需要导入相关的生命周期的代码，如这里的 app1.js。当然，不

同框架的生命周期是不同的，在不同应用的生命周期中，创建 DOM 相关的逻辑也不同。

2. 子应用部分

对于一个 Single-SPA 的子应用来说，我们需要做如下几件事：

◎　根据不同的框架设计生命周期。

◎　根据不同的应用创建加载的 DOM 节点。

下面是一个基于 Single-SPA 及 Single-SPA-React 的 React 子应用的应用配置代码：

```
//代码位于 chapter10/single-spa-sample/src/app1 目录下
import React from 'react';
import ReactDOM from 'react-dom';
import singleSpaReact from 'single-spa-react';
import Root from './root.component.js';

const reactLifecycles = singleSpaReact({
  React,
  ReactDOM,
  rootComponent: Root,
  domElementGetter,
});

export function bootstrap(props) {
  return reactLifecycles.bootstrap(props);
}

export function mount(props) {
  return reactLifecycles.mount(props);
}

export function unmount(props) {
  return reactLifecycles.unmount(props);
}
```

配置的核心部分是创建 reactLifecycles 定义的部分。在初始化这个变量时，需要传入应用程序相关的信息及根组件（rootComponent），通过这些信息来创建一个 React 应用。

对于 Angular 等框架也是类似的，只是不同框架的传入参数不同。下面是一个 Single-SPA 的 Angular 应用的配置示例：

```
//代码位于 chapter10/single-spa-sample/src/app2 目录下
const ngLifecycles = singleSpaAngular({
  domElementGetter,
  mainModule,
  angularPlatform: platformBrowserDynamic(),
  template: `<app2 />`,
  Router,
})
```

由于不同的前端应用会定义自己的 DOM 节点，以挂载应用到节点上。因此在完成相关的配置后，我们还需要创建对应的 DOM 节点，或者直接传入相应的元素。如上述的 React 或者 Angular 配置，都依赖于 domElementGetter 元素。当页面上还不存在对应的元素时，需要手动地在这个配置文件中创建。

```
//代码位于 chapter10/single-spa-sample/src/app1 目录下
function domElementGetter() {
  // Make sure there is a div for us to render into
  let el = document.getElementById('app1');
  if (!el) {
    el = document.createElement('div');
    el.id = 'app1';
    document.body.appendChild(el);
  }

  return el;
}
```

当我们拥有 DOM 节点、定义完子应用的生命周期，以及基座应用根据路由匹配到对应的应用时，就可以加载对应的应用。

3. 优缺点

从上面的实践和介绍中，我们可以发现 Single-SPA 的一些优势：

◎ 能支持大部分主流的前端框架，也能支持传统的前端框架。

◎ 提供更好的用户体验。即不需要页面跳转，直接在当前页面载入。

◎　方便于迁移旧的遗留系统。

Single-SPA 拥有以下的缺陷：

◎　系统构建复杂。应用需要集成在一起进行构建。

◎　不支持不同应用的部署分离。

◎　代码结构复杂。

◎　有额外的大量学习成本。

对于系统构建和分离部署来说，结合之前微应用化的组合式集成方案，也是一个不错的选择——先创建占位模块，然后在构建时进行替换。因此，在 Single-SPA 的各种兼容方案的基础上，创建自己的微前端方案是一种更好的选择。

10.4.3　定制型前端微服务化：Mooa

既然 Single-SPA 不能满足我们的需求，我们就结合 Single-SPA 的一些好的实践制定出符合自己需要的前端微服务化方案。如果在 Single-SPA 中将生命周期放置在子程序中会影响系统的构建，那么将其移到基座应用中。对于复杂的应用来说，其架构往往是基于多个架构演进而成的。因此，基于某个“成熟”的方案来修改出适合自己的方案，才是最好的方法。

笔者曾编写过一个面向 Angular 的微前端架构 Mooa，它是基于 Single-SPA 的源码及生命周期思想而编写的，但是其设计思想与 Single-SPA 相距甚远。从上面的 Single-SPA 的示例中可以看出，基座应用是必须的，可以将基座应用和基础核心功能相集成，作为一个胖基座型的架构模式。下面是基于 Mooa 框架的基本架构模式：

◎　主工程，负责加载其他应用，以及用户权限管理等核心控制功能。

◎　子应用，负责不同模块的具体业务代码。

在这种模式下，由主工程来控制整个系统的行为，子应用则做出一些对应的响应。

1. 主工程部分

当我们将应用的控制权收集到主工程之后，与应用相关的操作配置，以及每个应用的配置都需要由主工程来获取。这时主工程要做的事情有：

（1）初始化微前端框架相关的配置。

（2）从远程获取所有子应用的配置。

（3）配置所有的子应用。

（4）进行主工程和子应用的路由配置。

（5）监测路由变化，一旦匹配到对应的路由，则加载对应的应用，并卸载旧的应用。

对于以配置为主的微前端架构来说，它们的流程是相似的。下面是 Mooa 框架的初始化（app.component.ts 文件）：

```
//代码位于 chapter10/mooa-examples/mooa-hosts/src/app/目录下
this.mooa = new Mooa({
  debug: false,
  parentElement: 'app-home',
  urlPrefix: 'app',
  switchMode: 'coexist',
  preload: true,
});
```

在初始化参数里我们配置了如下内容：

◎ 不开启微前端框架的调试功能。

◎ 子应用的父级节点（parentElement）是 app-home 元素。

◎ 子应用的 URL 前缀是 app。

◎ App 间的切换模式（switchMode）是同时存在（coexist）的。

对于各种定制型的方案其具体配置都是不同的，但是基本思想都是类似的。这些配置在我们实践的过程中会不断地丰富。

对于子应用相关的配置也是类似的。下面是 Mooa 框架的子应用配置示例：

```
//代码位于 chapter10/mooa-examples/mooa-hosts/src/assets/目录下
[
  {
    "name": "app1",
    "selector": "app-app1",
    "baseScriptUrl": "/assets/app1",
    "styles": [
```

```
        "styles.3bb2a9d4949b7dc120a9.css"
      ],
      "prefix": "app/app1",
      "scripts": [
        "main.fbb4c7398448b0ebe06f.js",
        "polyfills.908282e8e7f434e6eebf.js",
        "runtime.ec2944dd8b20ec099bf3.js"
      ]
    }
  ]
```

上面的大部分配置都是指向与 App 相关的静态文件的，用于在加载应用时使用。在有了上述配置和应用数据之后，我们就可以在 app.component.ts 中创建应用，并绑定相应的路由事件。当路由发生变化时，可以通过框架来通知其他应用：

```
  //代码位于 chapter10/mooa-examples/mooa-hosts/src/app/目录下
  private createApps(data: IAppOption[]) {
    data.map((config) => {
      this.mooa.registerApplication(config.name, config, mooaRouter.matchRoute
(config.prefix));
    });

    this.router.events.subscribe((event) => {
      if (event instanceof NavigationEnd) {
        this.mooa.reRouter(event);
      }
    });

    return this.mooa.start();
  }
```

由于篇幅所限，并且源码并不复杂，这里就不展开源码的讨论了，读者可以从 chapter10/mooa-examples/中进行更深入的了解。

2. 子应用部分

对于子应用的设计来说，我们既要考虑其在系统中的形式，也要能让它可以独立地运行。Single-SPA 框架使用集成构建的模式，子应用并不一定能独立运行——它们需要一起集成才能进行构建。对于日常的开发而言，我们并不希望发生这样的情况。我们在设计的时候需要考虑如下内容：

（1）降低微前端架构相关代码的侵入性。

（2）设计一套机制来应对微前端和独立运行的两种模式。

对于第（2）个事项来说，我们可以注入一个全局变量，如在基座工作中为 window 绑定 isSingleSPA 对象，并在全局范围内标识它是一个微前端架构的应用。只要这个变量的值不是 true，就表明系统处于独立运行的模式，相应的路由等机制将继续保持原来的模式。

下面是笔者编写的 Mooa 框架中的子应用的代码修改，一共需要三种对应的修改逻辑。

路由更新。在 App 开始运行的时候，需要监测是否在子应用中跳转到对应的路由。如我们在基座中为应用 A 配置多个路由入口，当用户单击基座的链接时，应用 A 无法监测到变化，因此需要应用 A 做出响应。对于应用 A 而言，需要在初始化时监听基座的路由事件。

监听路由代码的逻辑如下：

```
export class AppComponent {
  constructor(private router: Router) {
    mooaPlatform.handleRouterUpdate(this.router, 'app1');
  }
}
```

适配 Angular 框架的基准 URL。对于 Angular 框架而言，它还需要配置基准的 URL（形如<base href="/">），对于子应用来说，它需要位于对应的 app 路径下——假设应用名称是 app1，那么它的基座 URL 应该是/app/app1，而不是/。在独立运行时，它的基准 URL 则是/。因此，需要在对应的 app.module.ts 中进行 APP_BASE_HREF 的配置：

```
providers: [
  {provide: APP_BASE_HREF, useValue: mooaPlatform.appBase()},
],
```

解决依赖冲突。在 Angular 中使用 Zone 作为变化检测机制，因此每个应用都会在 polyfill.ts 中引入 Zone。如果在一个页面中存在多个 Zone 就会发生冲突，对于这个冲突，我们不得不手动解决，比如只在基座应用中引入 Zone.js 文件（子应用不引入 Zone.js 文件）。

3. iframe 隔离

由于应用间的冲突并不好解决，因此在设计的过程中，我们可以考虑引入 iframe 来隔离应用。并且，我们可以提供特制的 iframe 模式，对于使用 Mooa 框架的开发者来说，只

需要进行一个配置改动就可以。

要想控制不同的 iframe 需要做到以下几点：

（1）为不同的子应用分配 ID。

（2）在子应用中进行 hook，以通知主应用：子应用已加载。

（3）在子应用中创建对应的事件监听，来响应主应用的 URL 变化事件。

（4）在主应用中监听子程序的路由跳转等需求。

这些相关的实现机制，都由框架本身来实现。

10.4.4　前端微服务化总结

从上面的例子中可以看出，实施前端微服务化是一件复杂的事。我们需要深入框架的原理，了解各种相关的实现，才能封装出适合自己的方案。尽管当前的这些实践，并不一定是最好的实践。但是随着技术的发展，会出现一些更好的方式，来实现上述的目标。

就当前而言，它们可以解决大部分与微前端架构相关的问题。仅凭这两个例子，并不能覆盖大部分的情况，但是相信它们能启发读者做出更好的前端微服务化架构。

10.5　组件化微前端：微件化

如我们在第 9 章中所说，在微前端下的微件化指的是，每个业务团队编写自己的业务代码，并将编译好的代码部署（上传或者放置）到指定的服务器上。在运行时，我们只需要加载相应的业务模块即可。

尽管微件化的定义中部署的代码是已编译好的代码，但是实际上，它可以是源码，又或者是某种特定的语言。按照广义的微件化方式可以将微件化方式划分为三种：

- DSL 微件化，即通过创建领域特定语言（DSL）来实施微件化。
- 预编译微件化。
- 运行时编译微件化。

对于 DSL 微件化而言，它不是一种真正的微件化方案。并且，我们也不可能使用 DSL 来编写复杂的业务组件，因此，它并不适用于此。对于使用微件化的应用来说，它们的模式是差不多的：

◎ 完成本地微件的开发后上传到指定的微件注册中心。

◎ 应用在获得对应的微件请求时，从微件注册中心查找微件。

◎ 一旦匹配到对应的微件，就下载相应的微件代码。

◎ 将微件嵌入页面中执行。

对于微件化的方案来说，如果存在第三方开发组件，就不得不考虑寻找如 iframe 这样的隔离方案。从上面的定义我们发现，微件化并不适合于复杂的前端应用，但是它可以作为微前端方案的一种。

对于传统的多页面应用来说，微件化是相当容易实施的方案。在传统的多页面应用结构中，只需要编写相应的 HTML、CSS，以及 JavaScript，就可以在加载微件时创建相应的组件和元素，再替换到相应的 DOM 结点上。在今天看来，这种方式更类似于 WebComponents。

对于多页面应用而言，只要支持运行时编译模板的框架都可以支持微件化。目前主流前端框架（React、Vue 和 Angular）都可以支持这种方式，而三者的实现略有差异。Vue 使用传统的 ES5 + HTML 模板语法，因为可以直接嵌入页面使用，故其实现方式比较简单，不需要额外的编译工作。因为 Angular 默认使用 TypeScript 语言，React 使用 JSX 语法，所以它们需要编译后才能嵌入页面中运行。

10.5.1 运行时编译微件化：动态组件渲染

运行时编译微件化，即组件在运行时才编译和运行微件化。不过，运行时的微件化需要在运行时编译，并引入整个框架的编译器，这样才能创建相应的组件，并加载到页面中。运行时编译微件化也对应用的体积和性能造成了一定的影响。

1. Vue 微件化

Vue 是一个渐进式框架，它既能以单页面应用的形式进行构建，又能结合传统的多页面应用来开发前端应用。因此，其微件化的实现也相对比较简单，通过官方的示例，就可

以看到它的微件化样貌。

下面是 Vue 的模板示例，它可以直接嵌入现有的页面中：

```
<div id="app-2">
  <span v-bind:title="message">
    鼠标悬停几秒钟查看此处动态绑定的提示信息!
  </span>
</div>
```

相关的 JavaScript 代码如下：

```
var app2 = new Vue({
  el: '#app-2',
  data: {
    message: '页面加载于' + new Date().toLocaleString()
  }
})
```

只需将上述两部分代码放置在远程。当我们需要的时候，创建 ID 为 app-2 的元素，再运行相应的 JavaScript 脚本即可。

2. Angular 微件化

由于 Angular 框架很复杂，所以其微件化的过程也相对复杂一些，具体内容如下：

◎　在核心工程中提供微件化所需要的环境，即应用相关的依赖，如 Angular 相关的依赖。

◎　在微件工程里，构建出能运行在核心工程的代码。

在 Angular 微件化的过程中，我们依赖 System.js 来完成依赖环境的构建和微件构建。System.js 是一个通用的动态模块加载器，它为浏览器和 Node.js 提供加载 ES6 模块、AMD、CommonJS 和全局脚本的能力。

下面是一个使用 Angular + System.js 构建出来的组件代码示例：

```
//代码位于 chapter10/micro-components/widgets 目录下
System.register(['@angular/core', '@angular/common/http', '@angular/common'],
function (exports, module) {
……
});
```

它与 AMD 形式类似，只要在模块中引入这些依赖，组件就能拥有对应的依赖来运行。

首先，我们在核心工程中使用 System.js 提供这些依赖，具体代码如下：

```
    //代码位于 chapter10/microcomponents/src/app/dashboard/dashboard 目录下
    SystemJS.set('@angular/core', SystemJS.newModule(angularCore));
    SystemJS.set('@angular/common', SystemJS.newModule(angularCommon));
    SystemJS.set('@angular/common/http',
SystemJS.newModule(angularCommonHttp));
    ...
```

然后，配合 Angular 的动态组件加载机制，就可以加载对应的微件并运行，具体代码如下：

```
    //导入微件
    const module = await SystemJS.import(widget.moduleBundlePath);

    // 编译模块
    const moduleFactory = await this.compiler.compileModuleAsync(module
[widget.moduleName]);

    // 解析 ComponentFactory
    const moduleRef = moduleFactory.create(this.injector);
    const componentProvider = moduleRef.injector.get(widget.name);
    const componentFactory = moduleRef.componentFactoryResolver.
resolveComponentFactory(componentProvider);

    // 编译组件
    this.content.createComponent(componentFactory);
```

对于微件而言，需要单独配置编译、构建脚本，详细的配置可以参见本章的相关代码，目录位于 chapter10/micro-components/widgets/widget-quotes。

读者可以尝试在上述代码里构建组件，并查看最后的生成组件，会发现其中的模板、样式等都没有编译。它只有在运行时才能编译，也因此可能出现一定的性能问题。除了编译时的性能问题，还有应用的体积——所有依赖都直接加入项目中，而不是需要什么加入什么。

10.5.2　预编译微件化

既然在运行时编译有可能产生性能问题，那么我们就尝试提前编译这些微件。提前编译组件并嵌入应用中使用，并不是一件容易的事。

对于某些框架（如 React）来说，可以像 Vue 一样直接嵌入页面中使用，但是需要提供编译过后的版本。它的微件化实施起来也比较简单。

对于另外一些框架来说，要做到这种程度并不容易。它需要依赖大量的编译、打包相关的经验，并深入理解相关的打包工具、框架原理，才能完成相应的开发。

1. React 微件化

想要使用 React 框架的微件化开发，需要先将其编译为可在浏览器上运行的 ES5 代码，再放到浏览器上运行。相关的实施步骤如下：

（1）编写统一的 API，封装 React 的根组件。

（2）修改构建系统的配置，让其输出可在浏览器上直接运行的版本。

（3）加载对应的 JavaScript 脚本，并调用组件的初始化方法。

从以上步骤来看，React 微件化比 Vue 的方式复杂。下面是封装接口的示例代码：

```
//代码位于 chapter10/react-micro-components 目录下
export const init = (config) => {
  ReactDOM.render(<App/>, document.getElementById('root'));
  serviceWorker.unregister();
}
```

通过修改 webpack.*.js 配置文件，将构建的目标平台配置为 umd，同时修改库为 MyApp，相关代码如下：

```
var config = {
  // ...
  output: {
    // ...
    library: 'MyApp',
    libraryTarget: 'umd',
```

```
      umdNamedDefine: true,
    }
  }
```

接下来，只要在页面上引入对应的组件，并执行相应的初始化代码即可，相关代码如下：

```
<script src="./bundle.js" type="text/javascript"></script>
<script type="text/javascript">
  MyApp.init({
    some: "config"
  });
</script>
```

基于这个基本的原则，我们就可以构建出基于 React 框架的微件化方案。

2. 通过构建工具进行微件化

在笔者了解的项目中，有一些项目通过 webpack 的 hack 实现了微件化，实现过程很麻烦。

在微件化实现的过程中，需要深入了解 webpack 的编译过程，以及框架的相应原理。由于场景有限，从实用的角度来看，不如前端微服务化方便，而从实现的角度来看不如微应用化实施方便，笔者并没有深入地进行研究。建议有需要的读者深入研究 webpack 的编译流程，来构建这种类型的微前端架构。

其基本思想和运行时的微件化是类似的，即为微件提供一个可运行的环境。作为预编译的微件，其流程中比较麻烦的地方是，提供一个 Mock 的包依赖环境，它能让两个不同项目中编译出来的组件（chunk）交叉运行。通常情况下，我们使用 webpack 打包出来的 chunk 文件，因为编译及 minify 的关系，其 ID 等都会发生变化。

10.6 面向未来：Web Components

我们在第 7 章介绍过如何使用 Web Components 来构建跨框架的组件。既然它能通过跨框架来实现组件复用，那么它也可以实现微前端里的微件化、微服务化等。

10.6.1　Web Components

在这里，让我们再次探索一下 Web Components。

Web Components 是一套技术，允许开发人员创建可重用的定制元素（功能封装在代码之外），可以在 Web 应用中使用它们[1]。

从定义中我们可以发现，Web Components 无论从名称还是定义上看，都特别适合作为一个组件、微件而存在。Web Components 主要由如下四项技术组件组成[1]：

- Custom Elements（自定义元素），允许开发者创建自定义元素。即常见的 HTML 标签以外的元素，如<app-root></app-root>等。
- Shadow DOM（影子 DOM），它是一组 JavaScript API，用于将封装的“影子”DOM 树附加到元素（与主文档 DOM 分开呈现）中，并控制其关联的功能。通过它可以保持元素的私有化，而不用担心与文档的其他部分发生冲突。
- HTML Templates（HTML 模板），如<template>和<slot>元素等，可以用于编写不在页面中显示的标记模板。
- HTML Imports，用于引入自定义组件。目前的做法都是将组件的定义细节存储在一个单独的文件中，再通过导入文件来使用。

这几项技术光从理论上来说有些抽象，让我们看一个简单的例子。首先，Web Components 的组件由 link 标签引入，具体代码如下：

```
...
<link rel="import" href="components/di-li.html">
<link rel="import" href="components/d-header.html">
</head>
```

然后，在需要使用组件的地方引入这些组件，具体代码如下：

```
<d-header></d-header>
<div class="container">
  <di-li (click)="world()" [draggable]="true" [text]="'import'"></di-li>
</div>
```

[1] https://developer.mozilla.org/zh-CN/docs/Web/Web_Components

这些 Web Components 附加在各种元素结点上，如图 10-2 所示。

```
▼<di-li (click)="world()" draggable="true">
  ▼#shadow-root (open)
    <style>
        :host {
          color: #dd4633;
          font: 18px Arial, sans-serif;
        }
      </style>
    "Hello..."
    <span>{{text3}}</span>
    <br>
    "World"
    <input type="text" (change)="onTextChange($event)">
  <style>
```

图 10-2

接着，在各自 Web Components 的 HTML 文件里创建相应的组件元素，并编写组件逻辑，其架构如图 10-3 所示。

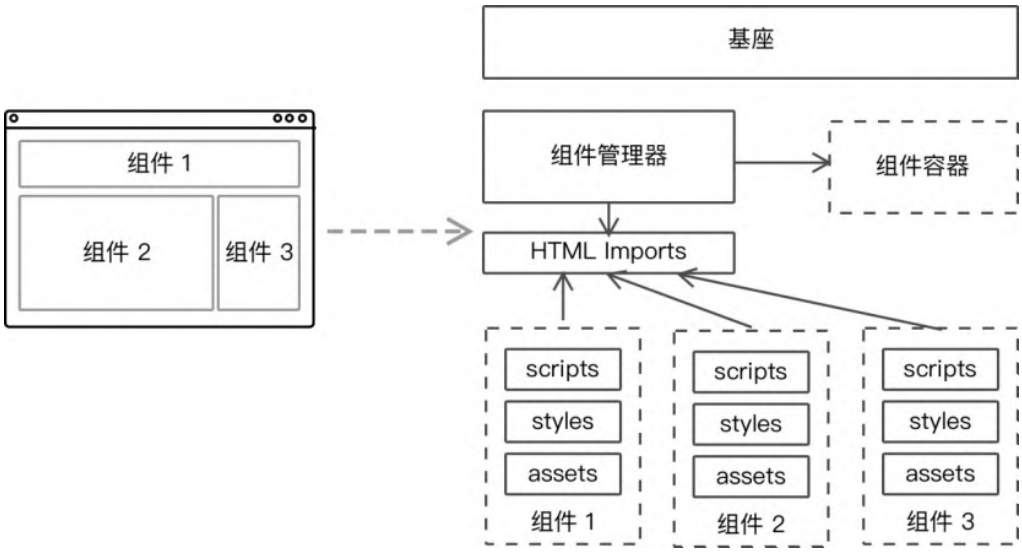

图 10-3

这种方式与之前使用 iframe 的方式相似，组件拥有自己独立的脚本和模式，以及对应的用于单独部署组件的域名。在使用的时候，我们只需要引入这些组件即可。从它的结构形式上来看，它非常适合于微件化架构。在需要使用组件的时候，可以做如下事情：

◎　创建组件的 DOM 结点。

◎　通过 HTML Imports 导入组件。

这和我们之前定制的微件化、微服务化几乎是一致的，唯一的区别是，通过 Web Components 方式实现的事情都是原生的，不需要开发人员自己编写。根据是否与现有的框架结合，我们可以将 Web Components 的微前端方案分为两种：

◎　纯 Web Components 方式。

◎　结合 Web Components 的方式。

光从名称上就可以看出两者的不同之处。

10.6.2　纯 Web Components 方式

直接使用 Web Components 及原生的 JavaScritp 来构建应用是相当复杂的一件事，我们需要编写自己所需要的基本组件。下面是一个基础的 Web Components 示例：

```
//代码位于 chapter10/wc-demo/hello-world/目录下
class HelloElement extends HTMLElement {
  constructor() {
    super();
    var shadow = this.attachShadow({mode: 'open'});
    shadow.innerHTML += `
        <style>
            #tabs { background-color: red; }
            span { color: purple; }
        </style>
        <div id="tabs">
            hello,
        </div>
        <span>world</span>
        `;
  }
}

customElements.define('hello-element', HelloElement);
```

首先，定义一个 HelloElement 的元素，它继承自 HTMLElement。然后，通过 attachShadow() 给指定的元素挂载一个 Shadow DOM，并且返回它的 ShadowRoot。通过为 innerHTML 赋值在 ShadowRoot 定义要创建的样式和 HTML。最后，将自定义标签<hello-element>与相关的代码进行绑定。这样我们就完成了一个 Web Components 组件，可以随时在其他项目中使用。

然而，使用纯 Web Components 构建应用会更复杂，它没有一个现代的框架所需要的双向绑定、路由、模板引擎等要素。因此，在构建的时候可以考虑使用基于 Web Components 的前端框架，如 Polymer 或者 Stencil。下面是使用 Stencil 构建 Web Components 组件的示例：

```
//代码位于 chapter10/stencil-demo 目录下
@Component({
  tag: 'my-component',
  styleUrl: 'my-component.css',
  shadow: true
})
export class MyComponent {
  @Prop() first: string;
  @Prop() middle: string;
  @Prop() last: string;

  private getText(): string {
    return format(this.first, this.middle, this.last);
  }

  render() {
    return <div>Hello, World! I'm {this.getText()}</div>;
  }
}
```

这个框架的风格与 Angular 类似，只是它构建出来的是独立于 Web Components 的组件。就当前而言，我们不太可能使用纯 Web Components 来开发前端应用，因此我们将面临如下挑战：

- ◎ 重写现有的前端应用。我们需要通过使用 Web Components 来完成整个系统的功能。
- ◎ 上下游生态系统不完善。纯 Web Components 方式缺乏相应的第三方控件支持，这也是 jQuery 相当流行的原因。

只有当我们考虑从头开发一个应用或者团队相关的底层技术时，才会考虑使用纯 Web Components。

10.6.3　结合 Web Components 方式

既然不能从头使用 Web Components，那么就尝试结合 Web Components 来构建应用。就当前而言，有两种方式可以结合 Web Components 来构建微前端应用：

◎　使用 Web Components 构建独立于框架的组件，并在对应的框架中引入这些组件。

◎　在 Web Components 中引入现有的框架，类似于 iframe 的形式。

前者是一种组件式的方式，后者像是在迁移未来的“遗留系统”到未来的架构上。

1. 在 Web Components 中集成现有框架

在现有的 Web 框架里，已经有一些框架可以支持 Web Components 形式的组件，比如 Angular 支持的 createCustomElement 就可以实现一个 Web Components 形式的组件：

```
@NgModule({
  declarations: [InteractBar],
  imports: [BrowserModule],
  entryComponents: [InteractBar]
})
export class AppModule {
  constructor(private injector: Injector) {
    const interactBar = createCustomElement(InteractBar, {injector});
    customElements.define('interact-bar', interactBar);
  }
}
```

只需要这样做就可以在 HTML 中传递参数：<interact-bar filename="phodal.md"></interact-bar>，或者监听对应的@Output 事件：

```
const bar = document.querySelector('interact-bar');
bar.addEventListener('action', (event: any) => {
  ...
})
```

可以通过采用微件化或者微应用化的形式来构建应用：

（1）将使用不同前端框架开发的应用逐一构建成 Web Components 组件。

（2）编写一个面向 Web Components 组件的应用加载器。

（3）结合业务逻辑在这个 Web Components 容器里管理不同的前端应用。

从这个形式上看，它是一个更现代化的管理组件方式。在这个过程中，我们需要面对依赖的重复加载，以及依赖的冲突问题。

2. 集成在现有框架中的 Web Components

结合 Web Components 的另外一种方式，是类似于 Stencil 的形式，将组件直接构建成 Web Components 形式的组件，然后在对应的如 React 或者 Angular 中直接引用。下面是一个在 React 中引用 Stencil 生成的 Web Components 的例子：

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import registerServiceWorker from './registerServiceWorker';

import 'test-components/testcomponents';

ReactDOM.render(<App />, document.getElementById('root'));
registerServiceWorker();
```

在这种情况下，我们就可以构建出独立于框架的组件和应用。与在 Web Components 中集成现有的应用不同的是，这种方式需要使用 Web Components 来重新编写应用。从实践上看，要做到这样的事情相当困难——使用现有的框架编写 Web Components 组件，会比使用新的框架更容易。

不过，与在 Web Components 中集成现有应用相比，使用 Web Components 编写的应用和组件，由于其裁减了一些前端框架的特性，从体积上看，要比前端框架小得多。

总之，Web Components 技术依赖于浏览器的支持程度。在采用 Web Components 技术来构建应用时，需要考虑这个因素。因此，在采用 Web Components 技术前，需要有效地确认浏览器的支持范围，再决定是否采用。

10.7　小结

在本章中，我们主要关注于如何通过六种不同的方式来实现微前端架构。每一种微前端架构方案都有各自的优缺点及适应场景，在选型的时候，需要按照自己的真实需要来选择。当出现一些不可抗力（如领导、KPI）影响的时候，也应该从中选择合适的方案。

最后，让我们来对比一下不同的微前端架构方案。如表 10-1 所示是本章中不同方案的简要对比。

表 10-1

方式	开发成本	维护成本	可行性	同一框架要求	实现难度	潜在风险
路由分发	低	低	高	否	★	这个方案太普通了
iframe	低	低	高	否	★	这个方案太普通了
应用微服务化	高	低	中	否	★★★★	针对每个框架做定制及 Hook
微件化	高	中	低	是	★★★★	针对构建系统，如 webpack 进行 hack
微应用化	中	中	高	是	★★★	统一不同应用的构建规范
WebComponents	高	低	高	否	★★	新技术，浏览器的兼容问题

更详细的比较可以参考下面的方式，如表 10-2 所示。

表 10-2

架构目标	描述
a.独立开发	独立开发，而不受影响
b.独立部署	能作为一个服务来单独部署
c.支持不同框架	可以同时使用不同的框架，如 Angular、Vue、React
d.摇树优化	能消除未使用的代码
e.环境隔离	应用间的上下文不受干扰
f.多个应用同时运行	不同应用可以同时运行
g.共用依赖	不同应用是否共用底层依赖库
h.依赖冲突	依赖的不同版本是否导致冲突
i.集成编译	应用最后被编译成一个整体，而不是分开构建

如表 10-3 所示，a～j 分别表示上面的几种不同的架构考虑因素。

表 10-3

方式	a	b	c	d	e	f	g	h	i
路由分发	O	O	O	O	O	O			
iframe	O	O	O	O	O	O			
前端微服务化	O	O	O			O			
微件化	O	O			-	-	O	-	
微应用化	O	O		O	-	-	O	-	O
Web Components	O	O		O	O	O	-	-	O

图中的 O 表示支持，空白表示不支持，-表示不受影响。

这些都是笔者相应的实施经验，可能有所欠缺或不足。如果各位读者有更好的方式，欢迎在这本书代码的 GitHub 上，与大家分享和讨论。

11

第 11 章

架构演进：演进式架构

有些前端应用在开发完成之后，由于种种原因不再进行维护。有些应用，则还没到开发完成的时候就不再进行维护了。随着业务不断增加及代码不断膨胀，架构也因此发生变化，已经不再开发的应用可能还需要迁移到新的系统中。只要业务和用户还在，架构就拥有变化的可能性。而那些老旧框架系统中的代码，我们不想再维护。可是只要它存在，就还需要去阅读它们，或者改写相关的代码、重构旧的应用、重写部分代码，以使它们能满足新的需求。

在我们的职业生涯里，会不断地面临这种架构上的挑战。比如，采用新的技术栈重写应用，或挑出大梁重构现有应用，又或者将应用迁移到合适的架构上。每种挑战都有各自的特色：

- 更新，让旧的应用的依赖和环境不断更新，以免成为一个不可维护的遗留系统。
- 迁移，在改变小量代码的情况下，让旧的应用可以运行在新的架构上。

◎ 重构，对于架构的重构，往往从模块、服务级别上对应用进行代码改善。

◎ 重写，对系统中的部分功能、应用或全部功能，使用新的技术栈、语言进行重写。

◎ 重新架构，从架构的层面上，对应用进行重新设计、拆分。

在编写应用的过程中，我们学会如何使用某个框架、语言。在演进架构的过程中，我们学会如何去做应用。前者是一个开发人员的基线，后者则是一个开发人员成长的机会。对于前端开发人员来说，我们会关注架构的成长，因为它会为自己带来更多的成长。

11.1 更新

大部分的前端应用并非一直在开发，或者编写后不再维护。它们可能开发一段时间，然后停滞一段时间，再去开发一段时间，再休息一段时间，如此往复地循环。

在这种往复的过程中，如果我们只是修改业务代码，而不适当地更新依赖和架构，那么总有一天这些代码会腐烂。因为随着时间的积累，没及时更新的依赖会越来越多，我们需要维护的版本也越来越多。直到有一天，我们内联（改写）了所有的依赖，或者重写了这个应用。

笔者曾经历过一个项目，项目中的一部分是使用 Ruby On Rails 编写的框架，但是由于已经有一两年没有维护了，其中的版本已经落后，需要我们升级依赖、语言。我们在升级 Rails 的版本时，发现它和 Ruby 版本不匹配，而在升级 Ruby 的版本后，发现很多与语言不匹配的问题。最后，我们放弃了对依赖和语言的升级，以便于将来进行重写。

从那以后，笔者一直不断思考可维护性的问题。如果我们已经有了一个前端应用，它不太需要更新，但应用又有一定数量的用户，那么就要考虑这个应用在未来大规模修改的可能性。在修改过程中需要不断地维护，以防止这个应用变成一个遗留的、难以维护的系统。而在不同的升级方式里，我们所要面临的挑战都是不一样的：

◎ 依赖升级。

◎ 框架升级。

◎ 语言升级。

它们是根据各自的更新频率所产生的顺序。如第三方依赖更新频率会比较高，且数量

比较多，其更新对系统影响比较小。而核心的框架部分则变动略大，在进行大的版本升级时，需要评估升级难度。至于语言，往往几年才更新一个版本，在多数情况下，要升级起来并不容易。

11.1.1　依赖和框架版本升级

我们在选择依赖库的时候，通常会选择那些比较成熟的依赖。成熟的框架、组件、库，往往采用语义化版本的形式来发布。其遵循的版本号格式是：主版本号.次版本号.修订号，如 1.2.3 中的 1 是主版本号，2 是次版本号，3 是修订号。其版本号递增规则如下。

◎　主版本号：当你做了不兼容的 API 修改。

◎　次版本号：当你做了向下兼容的功能性新增。

◎　修订号：当你做了向下兼容的问题修正。

注意：有些框架并不会严格按照这种形式来发布，如 React 框架的版本号直接从 0.15.6 改为了 16.0。

依照这种形式，在针对依赖的升级时，我们会采取相应的应对措施：

◎　当修订号发生变更时，通常只是进行一些 Bug 的修复，不需要我们做出响应。

◎　当次版本号发生变更时，往往可能会发生一些 API 变更，我们要视向下兼容情况来决定是否跟进。

◎　当主版本号变化时，可能有些 API 已经不存在了，我们需要大量地改动应用。

当发生修订号变更和次版本号变更时，我们可以视情况不予理睬，或者有选择地更新，如果该库提升了性能，那么就可以更新；如果该库修改了我们未使用的 API 的 bug，也就没有理由再更新了。

当主版本号发生了变更，我们不得不跟进这些变化。这也意味着，我们可能需要花费几天、几星期的时间，才能解决相关的问题。根据官方的升级指南更新日志（Release Notes）和相应的文档，并改进我们的代码，以完善相应的调用接口。

在这个过程中，我们还会遇到依赖树（上下游依赖）冲突的问题。比如，在我们的项目中使用了依赖 A 的 1.1.0 版本，又需要引入依赖 B，而 B 依赖于 A 的 2.5.0 版本。当我们将 A 升级到 2.5.0 版本的时候，还得顺带更新其他代码。对于如 Angular 这样大而全的

框架而言，基本不存在这样的问题。但是如 React 这种由多个组件组合而成的框架，则经常会遇到这样的问题。

一旦我们决定对一个项目采取维护模式，就需要制定一些基本的策略，比如：

◎ 确认合理的时间间隔，如三个月一次。

◎ 定期检查依赖或者使用工具自动检测。

◎ 为更新预留时间及精力

◎ 准备文档策略，以记录过程中遇到的问题。

但是无论如何，难点不在于规范和策略的制定，而在于升级依赖的决心。

11.1.2 语言版本升级

对于前端而言，语言的版本升级，并不会带来太大问题。不论我们使用 ES6 还是 TypeScript，到目前都没有发生严重的升级问题。其最后在浏览器上呈现的方式是一致的，都需要编译为 ES5，都需要能兼容当前的主流特性。由于需要向下兼容 ES5，在升级上不会出现太多的问题。这一点与后端语言形成了鲜明的对比，在后端语言里，如 Python 或者 Ruby，一旦升级了某个版本，就需要重写应用。

尽管如此，我们还是要考虑语言在未来会发生的升级，这些升级可能会删除一些特性。因此在编写的时候，应该尽量考虑某些语法是否能够被大部分浏览器所采纳。例如在 Chrome 浏览器上，往往会包含一些新的特性，但是这些新的特性可能在其他浏览器中不被支持。越是如此，就越需要慎重地采用新特性。

当然，如果我们想从一种语言切换到另外一种语言，如从 CoffeeScript 切换到 TypeScript，或者从 ES6 切换到 TypeScript 都不是一件容易的事，这种工作等价于重写应用。

11.1.3 遗留系统重搭

如果上面的两种机制都失败了，那么到我们手上就变成一个遗留系统。如果我们在一个具有一定规模的组织里，就有可能遇到这种情况。越是有历史包袱的公司，就越有可能存在这样的项目，总有一天某个项目会被挖出来。

曾经，笔者所在团队在 GitHub 上找到一个基本符合我们需求的项目，但是其中有几

个依赖的版本比较旧，需要更新。对于这一类系统来说，与其说是升级，其实更像是应用的迁移，实施起来并不是一件容易的事。按笔者之前的升级和迁移的经验来看，该应用迁移可以分为如下几个步骤：

（1）创建全新的运行环境。

（2）准备接近线上环境的测试数据，如 Staging。

（3）执行更细粒度的版本管理控制，以方便回滚。如每一次小的变更，每一个新特性的升级，都需要版本管理工具来记录。

（4）优先升级次要组件的版本，以方便向上兼容。

（5）逐一升级核心框架，以查找对应的更新日志。

（6）必要时自己编写依赖。在升级依赖的过程中，我们极有可能遇到的一个问题是，某个依赖不再更新，此时需要自己内联这个依赖。

（7）清理掉不需要的代码和文件。

（8）进行完整的用户验收测试。

（9）上线前使用线上的环境进行预部署。

当然，这样升级太复杂，对于笔者来说，还有一种粗暴的迁移方式：

（1）使用最新版本的依赖，创建一个新的“hello,world”。

（2）将旧应用的部分业务代码直接复制到新的项目中。

（3）修改其中不合适的代码。

这种粗暴的方式并非总是可以采用的，我们往往需要回到上一种方式。

11.2　迁移

迁移与重构、重写最大的不同之处在于，迁移只需要改动少量的代码，就可以使旧的系统恢复生机。不需要深入理解代码中的业务逻辑，只是从架构、框架上略微对系统进行改造，就可以完成目标。此外，我们也不可能花费大量的时间和精力在重写应用上，也没

有必要重构原有的代码，只需要让旧的系统集成到新的系统中继续使用即可。

11.2.1 架构迁移的模式

如我们在第 9 章和第 10 章所说，使用微前端技术来容纳旧的遗留系统是一个不错的方向。但是如果没有银弹，在迁移到微前端之前，我们就要考虑一件事，即我们真的需要微前端吗？微前端并不总是适合的，它的适用场景是有限的。

因此，在使用新技术迁移旧的应用之前，我们需要思考如下几个问题：

◎ 旧的系统出现了什么问题？

◎ 拆解部分应用是否能够解决问题？

◎ 有没有平滑迁移的策略？

◎ 是否可能带来新的业务价值？

◎ 如何降低未来的演进风险？

我们也可以采用 5W1H 分析法来分析。如果自己并非是最早的迁移应用的开发者，那么可以寻找、咨询其他项目在迁移的过程中会遇到哪些问题。

事实上，对于这个问题我们已经有答案了，只是在找对应的论据来支撑自己。一旦论据充足了，信心也就坚定了。

决定了迁移之后，我们可以尝试迁移应用，步骤如下：

（1）构建和提取基础设施，如组件库、代码库。

（2）确认用于练手的边缘应用。如果失败了、不合适了，那么可以尝试使用其他模式。

（3）寻找合适的解耦方式，包含数据、事件等。

（4）尝试对系统的其他部分进行拆分。

（5）编写对应的文档及相应的培训。

值得一提的是，无论从零开发创建一个应用，还是迁移到微前端架构，我们都不是从一开始就做最复杂的事情。而是不断地在边缘试探，直到觉得合适的时候才开始真正动手去做。其过程无非是创建项目，并编写一些 Demo，再评估该项目是否适合我们。

11.2.2　迁移方式：微前端

在第 9 章、第 10 章里，我们已经在微前端上花费了大量的时间和精力，这里就不重复介绍了。我们主要关注迁移到微前端架构的几个步骤：

◎　寻找合适的微前端技术。

◎　确认可行的微前端方案。

◎　尝试使用其中的某些方案。

◎　对比、讨论几种不同方案的利弊。

◎　决定适合当前项目实施的方案。

◎　尝试在一个项目中使用新架构开发。

◎　编写架构决策记录及文档，记录实施过程中常见的问题。

◎　对项目成员进行相关培训。

在整个过程中最复杂的部分是确认微前端方案。在笔者之前的一个项目里，由于当时并没有太多的方案可以选择，我们花费了大概一个月的时间在讨论合适的方案。最后，我们从几个前端方案里选择出一个方案，并实施了相关的方案。

11.2.3　迁移方式：寻找容器

除了微前端的方式，对于前端项目而言还有一种迁移系统的方式：寻找更大的容器。这种容器视不同的应用场景有不同的实现，如针对浏览器是 iframe，针对移动端是 App 的 WebView 等。

对于浏览器应用来说，iframe 是当前用得比较广泛的前端容器，它可以极大地方便我们迁移旧的前端应用。Web Components 作为一个新的组件级容器，可以帮助我们兼容不同的应用和组件。只需要经过略微的修改，便可以将前端应用嵌入其他应用中去。

我们在开发内部应用时还可以采用如 Electron 这种桌面 GUI 框架，它可以提供用于运行 WebView 的容器，使得应用能直接运行，还能提供“直接”访问系统 API 的能力，拥有较高的自定义能力。

除了桌面应用，移动应用还存在大量的容器，如 Cordova + Ionic 的混合应用容器、React Native 中的 WebView 容器，或者 Flutter 中的 WebView 容器。拥有了这些容器，便拥有了新的可能性。可以将前端应用迁移到新的容器上，继续发挥原来的价值，而不需要花费大量的功夫，下面是举例子：

（1）早先我们做一个移动 Web 应用，它可以运行在微信、移动应用里。随着应用越来越复杂，我们想将其作为一个独立的应用而存在。为了在短期内看到效果，证明这种方式的可行性，我们思考的第一个方案是将现有应用开发成一个混合应用。于是，我们进行一些少量的修改，把它运行在诸如 Cordova 这种混合应用框架上。

（2）后来，有了更多的开发人员想提升更好的用户体验，便尝试编写原生应用。由于开发人员仍然都是前端领域的开发人员，便使用 React Native 的 WebView 作为容器。在这种情况下，我们可以一边运行旧的应用，一边使用新的技术栈重写旧的应用。在这个过程中，既保证了业务的稳定运行，又提供了一种迁移方案。

一旦迁移完成，就可以慢慢地抛弃旧的代码，重写前端应用。

11.3 重构

重构不是一件简单的事，决定做应用的重构是一个了不起的决定。作为一个资深的咨询师，笔者经常在文章、书籍中强调重构的重要性。但是，笔者强调的往往是代码层级的重构，而非应用的重构。如果需要对应用进行重构，那么就要深入代码中去，从代码中逐一解决问题。

与更新和迁移稍有不同的是，应用的重构是在软件开发过程中的行为，而不是维护期进行的工作。只有应用还在开发进行中，我们才能深刻体会代码的坏道，以及其带来的软件架构问题。

11.3.1 架构重构

在没有测试的情况下实施重构，无异于在悬崖边上行走。对于多数项目来说，重构是一个不存在的方案，原因也在于此。只有测试才能保证重要的重构能往下进行，否则只能大量依赖于手工测试。而手工测试，也是一种容易出错的方式——作为一个开发人员，我

们的测试经验往往并不丰富。因此，不得不依赖于测试人员，让他们帮助我们进行回归测试。

因此，在时间充沛的情况下，我们要做的第一件事情是引入测试，具体步骤如下：

◎　选择合适的测试框架，并引入项目中。

◎　找到需要重构的代码，编写相应的单元测试用例。

◎　提交相关的测试代码。

◎　准备进行代码重构。

在没有测试的情况下，还可以适当地采用 IDE 重构的方式，它能在有限的空间里改善代码的质量。IDE 重构，即借助于 IDE 来对代码进行重构，其通常是由快捷键来触发的。它将重构的主要工作交给 IDE，而不是由开发人员通过痛苦地修改代码来完成的。在多数时候，我们只需要按下快速键，再输入一些必要的名称和参数，重构就完成了。但是 IDE 重构的范围是有限的，只能帮助我们解决简单的代码问题。

不论曾经是否引入测试，重构的流程都是相差无已的：

◎　创建一个新的重构分支。

◎　从简单的重构开始练习。如目录调整、重命名变量。

◎　小步提交。使用细致化的版本管理，方便出错后的回退管理。

◎　对复杂的代码进行样式拆分、逻辑拆分、组件拆分。对于函数来说，可以采取提取变量的方式进行。

◎　修改或者编写测试，保障业务功能正常。

◎　对于复杂的功能，寻找合适的人一起完成重构。

关于如何对代码进行重构，已经有一本专业的书籍《重构：改善既有代码的设计》在讲述相关的内容，这里就不重复引入相关的内容了。我们只介绍那些前端需要注意的问题。

11.3.2　组件提取、函数提取、样式提取

代码的复制、粘贴是我们在前端应用中经常看到的操作。开发人员在编写逻辑的时候往往倾向于直接复制代码，而非重用代码，这两种方式各有利弊。可是如果在代码逻辑上

是重复的，就需要对其进行手术，删去重复的代码，以免代码进一步复杂化。回到前端的模式来看这个问题，就是三种（HTML、JavaScript、CSS）类型代码的提取：

◎ 组件提取。一个合理的前端应用，包含一个 UI 组件库。从组件的层面上看，可能出现代码重复的地方在复合组件和业务组件中。对于那些相似功能的组件，我们要决定是否提取出通用的模式，避免在业务变化时进行多处修改。对于复合组件来说，组件提取是在业务发展的过程中呈现的，需要在多次出现后进行模式提取。

对于组件而言，不得不再强调的一点是，多个小的组件比一个复杂的组件容易维护。但是，不同的人对于组件有自己不同的偏好，所以对于这一点往往难以达成共识。

◎ 函数提取。在我们的代码中往往包含着各种通用的处理逻辑，如统一的 HTTP 的错误处理，它们可以集中起来进行处理。随着我们的编程经验越来越丰富，在创建一些函数的时候就会自觉地创建 xxxUtil.js、xxHelper.js，或者 xx.services.ts。

在编写代码的过程中，可能会出现多种同样逻辑的代码，如同事 A 在实现 x 功能的时候，与同事 B 在实现 y 功能的时候，使用相同的处理逻辑。这时会发现功能重复实现的问题，这就需要我们在浏览代码的时候进行适当地提取和重构。在日常工作中，适当的代码检视（Code Review）能解决部分问题——但无法解决全部问题。

◎ 样式提取。样式是笔者一直觉得头疼的内容。哪怕引入一个公共的样式库，我们也只能在样式上尽量统一，统一的颜色、字体大小、border 等。但是，在实现业务功能的时候，非统一的样式往往会出现问题，如某一类元素的宽高、元素的定位等。由于它对我们来说并不是那么重要，所以也就不会那么在意。只是在出现需要多处修改的时候，才会重视对样式的重构。

11.3.3 引入新技术

由于进度限制或者开发人员能力的限制，在编写某些业务功能的时候，往往使用的不是最好的方式，充满了各种 hack 式的写法。随着技术不断地更新换代，便拥有了更好的解决旧问题的方式。新的技术栈、依赖库，可以在不侵入现有应用的情况下，改善这些现有的应用。

笔者对于这一类型的重构颇有印象，最典型的例子就是引入 promise 来解决回调地狱的问题。回调地狱，意味着逐层地回调嵌套，嵌套层数越多则问题越复杂，后期在理解回调的时候也相当困难。在没有 promise 之前，为了重构这一类多层嵌套的代码，往往会将

回调抽取成很多的匿名函数。在引入 promise 之后，则可以采取 promise.all 这类方法，以实现进一步解耦和灵活化回调函数。

就前端领域来看，以下几种代码的重构方式比较常见：

◎ 新的编程理念。随着时间的推移，编程理念也在不断地发生变化。早期流行过程式编程，现今主流的编程方式是面向对象编程，未来可能流行函数式编程。开发人员会尝试新的方式，以解决一些在日常开发中的痛点。比如，使用 promise 替换 callback，或者在复杂的前端处理逻辑中，采用 reactive.js 或者 rambda 来解决问题。当然，它也是有好也坏的，开发人员因为喜欢，可能会过度使用。在不合适的地方采用，或者写得过于复杂，会造成难以维护的问题。

◎ 引入新的框架特性。不仅在应用中会引入新的编程理念，在前端框架中更是经常通过引入新的特性来解决开发人员的痛点。比如，在使用 React 编写应用的时候，需要维护应用的状态，简单的应用可以自己编写，而复杂的应用则可以通过引入 Redux 来解决。然后 React 在自己的新版本里又引入 React Hook，它可以在某种程度上替换 Redux。如果框架的新特性比库更容易使用，那么重构这部分代码也不是不能接受的。

◎ 解决语言糟粕。由于设计时间很短，JavaScript 这门语言有诸多的细节设计得不是很好。它充满了糟粕，这些糟粕进一步影响了我们编写出来的代码的质量，比如在 ES6 之前的 JavaScript 虽然有类，但是没有类继承的概念。针对这一类问题，我们需要引入一些新的技术栈。比如，当我们要解决大量的浮点计算的问题时，需要引入如 bignumber.js 这样的库。尤其是当我们的代码中充满了各种相关的语句判断时，就需要通过这些库来重构代码。

尽管我们需要使用这些新的技术来解决旧的问题，但是还应该以适当地引入为主，以免未来需要再次对它们进行重构。当然，对于短期项目来说，这些因素可能并不重要；但是对于长期项目来说，它值得我们去深入考虑。

11.4　重写

一旦上述的几种方式都无能为力——更新、重构、迁移，那么应用就会成为遗留系统，

不得已就只能考虑重写这些应用。

过去笔者认为，重写一个应用是一件简单的事，而演进一个应用则是一件复杂的工作。所以，笔者有幸经历了一个大型前端应用的重写，发现重写应用一点都不简单——当一个应用有一定的历史时，没有人能讲清楚完成的业务逻辑，也没有人知道代码中的 2 是什么意思，大概就只能是 2 吧。要是我们在重写的过程中，没有把 2 变成诸如 varHALF = 2 这样的变量，也没有写上对应的注释，恐怕又会多一次重写。直到有一天，这个 2 被删除了，或者知道了 2 的含义。

此外，应用在其生命周期里，经过了不同的开发人员、不同的业务变更，必然有大量的遗留代码——没有人知道具体业务的代码。有大量的业务已经不存在了，然而代码仍然保留着，还包含了相关的注释，也就没有人想去或敢去修改这些代码。

因此，在拥有足够人力和物力的情况下，对于旧系统或者遗留系统而言，最好的使用方式就是重写应用。但是，基本上不存在这样的情况，除非我们远远领先于其他竞争对手。如此一来，我们还是能够接受重写的成本的。

11.4.1 重写能解决问题吗

在年轻的时候，我们对于有一定年限的应用总有一个相似的看法，为什么我们不重写呢？而资深的程序员，又会同情地告诉我们“重写不会带来业务价值”。那么，怎样才能带来业务价值？下面是问题的答案。

- 更少的功能完成时间。旧的系统需要 3 天才能完成的功能，新的系统现在只需要 1 天就能完成。
- 更好的用户体验。
- 提升应用的性能。旧的应用需要 3 秒才能响应结果，新的系统只需要 0.5 秒。

这些有年代感的应用，往往会增加新的功能上的麻烦。哪怕是修复一个简单的 bug，也需要花费大量的时间——因为可能会带来新的 bug。从这个角度来看，别人可能认为我们对代码库不熟悉。那么我们只能找到一些合适的证据，以证明这个系统确实需要重写了：

- 过时的依赖或技术。如使用过时的前端框架，找不到合适的开发人员。
- 出现表达力更好的技术。原来需要 20 行代码的功能，现在只需要 5 行代码。

◎　旧代码无法维护。值得注意的是，难以维护和无法维护是两个概念。

从过往的经验来看，还有一种重写是因为应用过于庞大，需要拆分为几个更小的应用，如微前端技术。但是庞大算不上是一个合适的重写理由，除非涉及多团队写作。

在我们决定重写之前，需要考虑几点要素：

◎　重构、迁移、升级真的不能解决问题吗？

◎　重写的时间预期是多少？重写的时间花费往往比预期更长，比技术上花费的时间更短，但理清业务的时间更长。

◎　能接受重写的成本吗？重写不会对业务带来额外的好处，反而是在浪费时间。

◎　是否整理出完整的功能列表？只有清晰的功能列表，才能保证重写不被阻碍。

◎　产品是否领先于市场？在重写期间，我们的开发速度可能会落后于其他产品。

◎　是否有能力同步维护两个产品和团队？在重写期间，需要在新旧应用里实现同样的功能。

◎　在重写完成之前，是否可能变为遗留系统？要进行合理的技术选型，以避免重写失败。

因此只有当重写应用的花费 + 维护应用的花费<维护现有系统的花费时，我们才应该考虑重写。同时在重写的过程中，还需要继续维护现有的应用。因此，一旦带宽不够，就无法支持开发人员重写应用了。

值得注意的是，在重写的过程中需要考虑开发人员的感受——在有些人看来，重写是在浪费时间。有足够的重写热情的人，能推动重写的完成；而有遗留系统经验的人，能保证在重写的过程中少走弯路。

11.4.2　梳理业务

从技术层面上看，重写应用并不复杂，复杂的地方在于理清原有的业务逻辑——看着旧的业务代码，编写新的业务代码，找寻公司内部相关的人寻问业务。

在整理的过程中会发现相关的开发人员、业务人员，往往只记得某个时刻的业务，并不清楚最终的业务是怎样的。为了理清这些业务，需要找到所有的利益相关人，按时间线，从头到尾理清业务变化，以确认最终的业务逻辑。然而，即便如此，我们也很难整理出所

有的业务逻辑。

在笔者过往的经验里，还有大量的业务逻辑隐藏在代码中。这些隐藏的业务逻辑，往往是被业务人员遗忘的。毕竟再详细的需求列表，也需要一些细节的优化。这些小的优化，可能并不会记录到任何文档里。这种情况是我们在业务讨论中无法逐一整理出来的，需要进一步对原有代码进行梳理，才有可能获得相关的业务知识。但这种做法也许会拖慢重写进度。在重写应用的时候，还可能存在的一种情况是，原来的应用还存在 bug，而这个 bug 在过去可能未被发现。对于重写的应用而言，测试人员和业务人员验收功能的时候，往往是依照原有的应用来对照的。而新的逻辑由于组件等原因与旧的逻辑不一致，此时才发现旧的应用中存在 bug。这一类 bug 往往不影响系统的主要功能。

当出现大量的业务遗漏的情况时，反复地修改会使开发人员沮丧——“为什么我们要重写这个应用？”

11.4.3 沉淀新架构

虽然，我们是在重写新的应用，但并非是从头设计的，而是已经拥有了一个旧的架构。在旧的应用计划重写之前，我们尝试过修复它，评估了很多方案，尝试了很多种可能性，最后无可奈何地选择了重写。

有了过往的尝试，从全貌上对旧系统有了全面的认识。再经过一系列的会议讨论，大部分开发人员心中就有了一个新系统的大概雏形。如果在重写新系统的时候没有一个可行的方案，那么重写可能会变成一场新的灾难。

在有些情况下，我们可能真的没有一个可行的方案，需要寻找合适的外援来帮助我们解决问题，如外部的技术咨询公司。如果真的无计可施，那么就需要对旧系统的架构进行总结：吸收旧系统的优点，总结旧系统的设计缺陷。如果不去总结旧系统的问题，直接开发设计新系统，那么无异于浪费时间。

新旧系统的相似之处如下：

◎ 选择更合适的技术栈。

◎ 进行组件和系统的防腐设计。

◎ 系统与应用间的解耦设计。

这些是我们在重写应用的过程中总结出来的。如果在旧系统中遇到组件、依赖的问题，

那么我们会在设计新架构的时候对它们进行隔离。如果原有的技术栈不合适，那么在选择的时候就会更加谨慎。经验，在有些时候是一无是处的，而在有些时候则是如虎添翼。

一旦我们有了重写系统的经历就会更加注意：在过程中改进系统，而非在未来重写应用。

11.5　重新架构

虽然，重写能带来更好的开发体验，还有更快的功能交付速度。但是，作为一个开发人员，笔者还是讨厌重写项目的。第一，难以说服业务人员，第二，重写并不是一种很好的体验——它好像只是在练习技术栈而已，不能带来技术上的成长。而与重写相比，在改写部分代码的情况下，重新对应用的架构进行设计，是一种更有意思的挑战。

11.5.1　重搭架构

重搭架构，与重构相比，是一种更高阶的应用重构。它从应用的层级上对架构进行重新设计，而非在代码层级上进行改进。比如，结合微前端架构来拆分前端应用，并使用绞杀者模式一点一点地重写应用。

代码难以维护导致我们想重写系统，其中具体原因有：代码结构不合理、依赖设计不合理等。这些问题几乎只存在于大的代码库里，小的代码库往往不会出现这样的问题。在这种情况下，我们所要做的是细分代码库，将其拆分成多个依赖、组件、应用。我们的核心目标是将系统变成高内聚、低耦合的系统。

因此，在实际的过程中，是在重新应用先前学习的相应内容：

◎　重新设计构建系统，以支撑新架构。

◎　设计模块化的应用架构，以帮助我们解耦模块。

◎　抽象化组件、提取函数库，以减少重复代码。

◎　采用微前端技术来解耦前端应用。

……

无论如何，在这个过程中，我们仍然会遇到一系列的挑战。对于一个单体的前端应用代码库来说，需要将其拆分为多个模块。模块之间的共用组件、模式库等需要逐个提取，以降低组件的耦合度，提高系统的复用率。

11.5.2 增量改写

增量改写，即将系统拆分为多个独立模块或应用。当我们决定对应用进行重写的时候，只需要重写其中的一部分，其他功能仍然是正常运行的。而那些需要新功能的部分，又可以运行在新的应用之上。直到有一天，我们重写完所有的模块，系统相当于重写完成了。这种模式，又称为绞杀者模式，它可以在满足系统演进的情况下实现如下目标：

- 大幅度减少对于业务的影响，降低了风险。
- 可以随时停止重写，而不影响用户使用。
- 用户、客户可以随时看到网站的变化，带来更多的价值。
- 为开发人员带来更多的自由和乐趣。
- 每个模块重写时，只关心相关部分的业务，而非所有的业务。

当然，这种方式拥有这么多优点而没有被采用，主要是因为实现起来有难度。该拆分方式仍依赖于微架构。对于后端来说，往往是使用 BFF 作为防腐层。对于前端来说，如果想采用这种模式，需要使用微前端架构来拆分现有的前端应用。它和我们前面所讲的内容是相似的：

- 设计全新的路由分发机制。
- 使用新技术编写某一部分应用。
- 将路由指向新的前端应用，剩下的部分仍然指向旧的应用。
- 不断添加新功能、重写模块，直至完成重写。

从增量改写相关的内容来看，它结合了重搭架构和重写两部分内容，也因此在设计上需要更多的精力。在笔者经历过的一个将多页面应用迁移到单页面应用中的项目，就采用了这种模式。它是一种依赖于路由分发式的微服务架构，系统切分出一个部分，并编写完成一个新的模块，便将路由指向新的应用。相应的旧的模块，也就可以不维护了，直到有一天，我们完成了系统的重写。

11.6　小结

过去，笔者总以为从头写一个应用是相当有挑战的一件事。而事实是，这种挑战不一定存在，或者只是存在搭建脚手架和架构的挑战。在经历了这个过程之后，我们就会觉得项目索然无味。事实上，此时此刻，才是我们真正挑战自己的时候。见证一个系统的成长，见证它的变化——好与坏，同时不断寻找提升自己的可能性。

在我们的职业生涯里，维护应用的时间远远长于启动一个新的应用。当我们觉得启动应用更有挑战的时候，是因为维护应用的时间太长。在经历多个项目之后，笔者才明白维护一个系统有多难，如何让一个我们觉得“烂”的系统慢慢变好？旧的系统给我们带来一系列深入的思考，我们会思考平时哪里做得不好，应该怎么做才能让系统变得更好？如果能不断地在新旧系统之间切换，那么我们就能理解好的架构的意义，好的设计的初衷。并且，只有经历过遗留系统，才能让自己真正地成长。

因此在本书的结尾，通过介绍更新、迁移、重构、重写、重新架构等几种不同的方式，展示了如何演进现有的应用的架构，它可以在未来帮助我们从多个角度考虑技术架构，以做出更好的技术决策。

反侵权盗版声明

前端架构

本书是一本围绕前端架构的实施手册，从基础的架构规范，到如何设计前端架构，再到采用微前端架构拆分复杂的前端应用。通过系统地介绍前端架构世界的方方面面，来帮助前端工程师更好地进行系统设计。

ISBN：978-7-121-36534-8

书价：79.00