# Web 应用新架构研究

16222010272 田杰

摘要：本文调查研究并总结了Web应用新架构在前端、后端以及整体的表现。并使用这些新架构做了一个可以演示的示例（<https://fire.pa-pa.me/> ）。

## 前端架构

前端技术是集中在浏览器端的技术，最终使用的是 HTML、CSS和JavaScript。

### HTML

* + - HTML： HTML是超文本标记语言，使用标签描述了网页的结构。HTML元素是HTML网页的基石，使用标签表示。浏览器并不展示这些标签，而是使用这些标签来渲染页面的内容。
    - HTML已经发展到HTML 5。
    - HTML5 浏览器支持情况：HTML5 能被所有现代浏览器支持。而且，不管是新的还是旧的浏览器，都会自动地将不认识的元素视为行内元素。
    - 除了浏览器的原生支持的HTML，在后面会详细讲到有JavaScript框架中出现的新的标记语言：jsx，它看上去很类似HTML，但是需要由JavaScript来解析。它出现的原因是为了简化动态更新页面的工作，以及提高页面改变渲染呈现时的性能。

### CSS

* + CSS是一门用来描述HTML文档样式的语言，它定义了HTML元素该如何展示。
  + CSS 已经发展到了CSS3

### JavaScript:

JavaScript经常被缩写为JS，是一门高级解释型编程语言。也被归为动态语言、弱类型语言、基于原型的语言以及多范式的语言。JavaScript让网页具有交互性，所有是Web应用的核心部分。

随着JavaScript的发展，现在JavaScript不仅可以运行在浏览器端，也能运行在服务器端（NodeJs）。浏览器端JavaScript，曾经风靡一时的jQuery已经日渐式微，浏览器端JavaScript进化出三足鼎立的Angular、React以及Vue前端框架。除此之外，更多的开源框架层出不穷。

1. 目前流行的前端框架：

MVX = MVC + MVP + MVVM

* MVC：Model（模型）+View（视图）+Controller（控制器），主要是基于分层的目的，让彼此的责任分开。  
  View通过Controller来和Model联系，Controller是View和Model的协调者，View和Model不直接联系，基本联系都是单向的。用户通过Controller来操作Model从而达到View的变化。
* MVP：是从MVC模式演变而来，都是通过Controller/Presenter负责逻辑的处理+Model提供数据+View负责显示。  
  在MVP中，Presenter完全把View和Model进行了分离，主要的程序逻辑在Presenter里实现。并且，Presenter和View是没有直接关联的，是通过定义好的接口进行交互，从而使得在变更View的时候可以保持Presenter不变。
* MVVM：MVVM是把MVC里的Controller和MVP里的Presenter改成了ViewModel。Model+View+ViewModel。  
  View的变化会自动更新到ViewModel，ViewModel的变化也会自动同步到View上显示。这种自动同步是因为ViewModel里的属性实现了观察者模式，当属性变更时都能触发对应的操作。

常见框架对比：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 框架 | 模式 | 说明 | 特点 | 缺点 |
| Riot js | MVP |  |  |  |
| Angular | MVVM | 优秀的前端JS框架，已经被用于Google的多款产品当中 | 支持指令（内置、自定义）  支持过滤器（内置、自定义）  双向数据绑定  良好的应用程序结构  HTML模板  可嵌入、注入和测试  坚实的基础和社区支持 | 学习成本高  Watcher 越多，性能越差  不支持低端浏览器 |
| Vue | MVVM | 严格地说它不是一个框架，因为它只聚焦视图层，是一个构建数据驱动的Web界面库 | 轻量级  双向数据绑定  支持指令（内置、自定义（内置、自定义）  插件化  不支持低端浏览器  基于依赖追踪的观察并且使用异步队列更新，性能好  .vue 特殊文件格式需要编辑后使用  一切都是组件  支持混入  DOM 模板 | 新生儿，不如Angular成熟  影响度不大  不支持IE8 |
| React | V | 主要用于构建UI。 | 采用特殊的JSX语法，需要编译后使用  一切都是组件  插件化  支持混入  依赖Virtual DOM  声明式设计  高效  灵活  速度快  跨浏览器兼容  模块化  单向数据流  同构、纯粹的JavaScript | 只是一个V而已，不是一个完整的框架。  如果是大型项目，基本要加上ReactRouter和Flux等。 |
| Knockout js | MVVM |  |  |  |
| Ember js | MVVM |  |  |  |

## 后端架构

### SOA（Service-Oriented Architecture）Hub（服务总线）

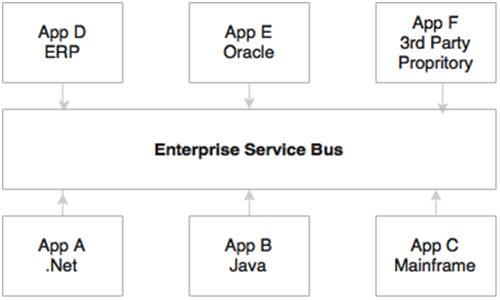
开放小组这样定义SOA：

SOA是支持面向服务的一种架构风格。面向服务是一种基于服务开发并且产出其他服务的思考方式。

* 它是自洽的
* 可以是其他服务组合的一种组合
* 对服务的消费者来说它是一个黑盒

#### 面向服务的集成：

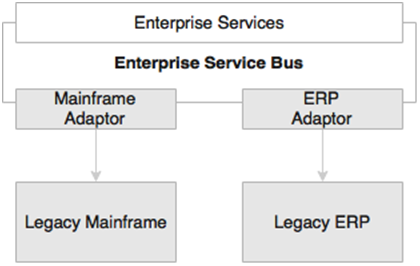
面向服务的集成是指很多组织采用的基于服务的集成方式：



很多组织使用SOA主要是解决集成复杂度问题，也就是所谓的集成意大利面条。一般地，这被称为面向服务集成（SOI）。在这种情况下，应用间通过公共集成层交流，这种公共集成层使用标准协议和消息格式，如基于SOAP/XML的Web Service或者HTTP或者Java Message Service（JMS）。这种组织类型将焦点放在企业集成模式（EIP）来建模他们的集成需求。这种途径强烈依赖重量级企业服务总线（ESB），如TIBCO Business Works，WebSphere ESB，Oracle ESB或者类似产品。多数ESB提供商将一系列相关产品打包在一起，如规则引擎、业务进程管理引擎等等，作为一个SOA套件。这种组织集成深度根源于这些产品，他们要么在ESB层编写重量级的编排逻辑或者在服务总线中编写业务逻辑本身。无论哪种情况，所有的企业级服务的部署和访问都发生在ESB上。这些服务通过企业管理模型来管理。

#### 遗留现代化

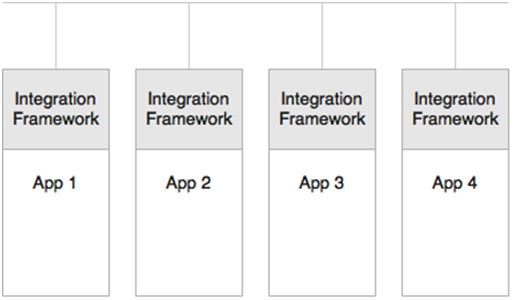
SOA也会在遗留应用之上构建服务时用到，如下图所示：



另一类型的组织将SOA用在过渡项目或者遗留现代化的项目中。在这种情况下，服务被构建并部署在ESB上，ESB通过ESB适配器来连接后端系统。

#### 面向服务的应用

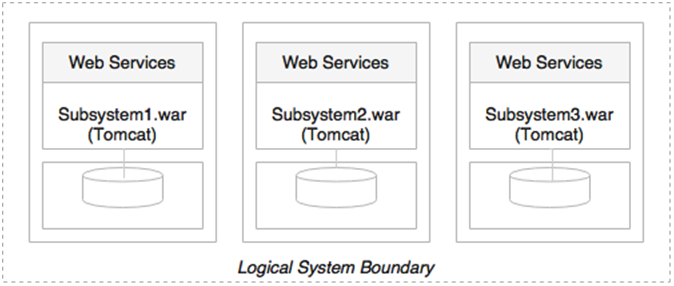
还有一些企业将SOA用在应用层级：



这种途径将轻量级的集成框架如Apache Camel或者Spring集成嵌入在应用中来处理服务相关的跨领域能力，如协议中介、并行执行、编排以及服务集成。因为一些轻量集成框架内置支持Java对象，这样的应用甚至会使用内置Plain Old Java Objects（POJO）服务来做集成以及服务间的数据交换。这样的结果是，所有的服务都必须被打包进一个单一Web压缩归档。这样的企业下一步很自然地就会使用微服务。

#### 使用SOA做整体迁移

下图展示了逻辑系统边界：



最后的一种可能性是当到达单一系统的临界点后将单一系统向更小的单元转化。他们可能会将应用分解成更小的物理可部署的子系统，与纵向扩展途径类似，并且把它们在Web服务器上作为Web压缩归档部署或者在系统自带的容器上作为Jar包部署。这些作为服务的子系统会使用Web Service或者其他轻量级的协议来在不同服务间交换数据。他们也会使用SOA以及服务设计原则来达到这一目的。对这些组织来说，他们会倾向于认为所谓微服务，不过是新瓶装旧酒。

### MicroService（微服务）

微服务是一种架构风格，是为了满足现代业务需求而演化出的软件开发途径。它能提升组织敏捷度，交付速度和扩展能力。微服务给你一种开发更多的物理区隔的模块化应用的开发方式。

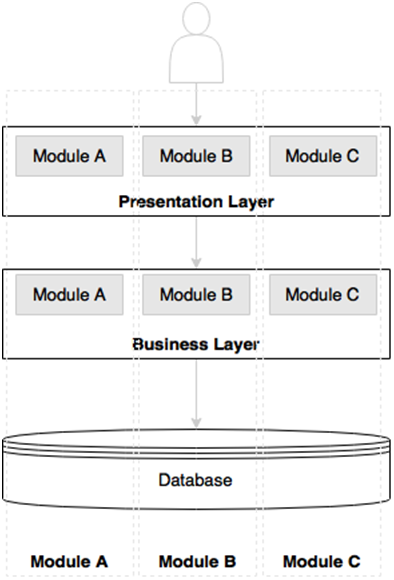
微服务与传统的面向服务架构有很多类似的地方。

在很多应用微服务的组织里，成功应用了著名的“分而治之”的技术，将它们的单一应用分区成更小的原子性单元。每一个单元执行单一职能——一个服务。这些组织解决了在单一应用时代碰到的大量的问题，于是越来越多的组织开始在重构他们的单一应用时应用此模式。后来这个模式被正式称为微服务架构。

微服务起源于六边形架构的主意。六边形架构或者说六边形模式也被称为端口和适配模式。Cockburn这样定义微服务：

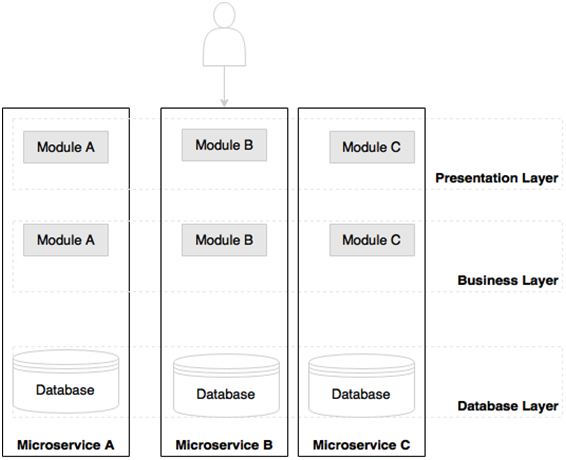
一种将IT系统作为一组自动化的、自洽并且低耦合的业务能力集合的构架风格或者方式。

下图是传统的多层系统架构，包括展示层、业务层和数据库层：

v

模块A、B和C代表了三种不同的业务能力。图中的层次代表了架构关注的分离。每一层都保有适合于该层的业务能力。展示层有所有三个模块的网页组件，业务层有所有三个模块的业务组件，而数据库层包含了所有三个模块的数据表。大多数情况下，层次之间可以物理分开，但是同一层次的模块是紧密结合着的。

让我们看看一个基于微服务的架构：



我们可以看到在微服务架构中边界反过来了。每个垂直的切片代表了一个微服务。每个微服务会有其特有的表示层、业务层和数据库层。微服务与业务能力保持一致，从而对一个微服务的改动不会影响其他的微服务。

微服务之间的交流或者传输机制并没有统一标准。一般来说，微服务之间通过接受度广的轻量级协议来沟通，如HTT和REST，或者消息协议，如JMS或者AMQP。特殊情况下，有些微服务会通过经过更优化的协议如Thrift、ZeroMQ、Protocol Buffers或者Avro 来沟通。

因为微服务与业务能力保持得更一致并且有着独立的可管理的生命周期，它们是强调DevOps以及云平台的企业的理想选择。DevOps和云平台是微服务的两个方面。

### BFF（Backend for Frontend）

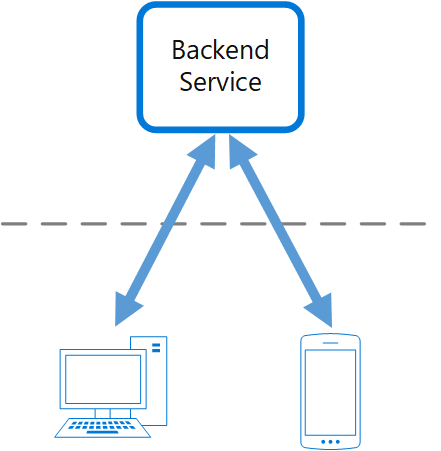
创建单独的后端服务来让特有的前端应用或者界面来消费使用。如果你想避免为多个界面定制同一个后端，那么这个模式非常有用。

背景：

一个应用一开始可能只针对桌面Web UI。典型地，一个后端服务是与相应的UI并行开发的。随着该应用的用户基数的增长，移动应用被开发出来了，而且必须和同样的后端打交道。这个后端服务变成了一个通用的、既服务于桌面UI又服务于移动UI的后端了。

但是移动设备的能力与桌面浏览器截然不同，比如说屏幕尺寸、性能、以及显示限制等。这样就形成了移动应用与桌面Web应用的需求显著分化的结果。

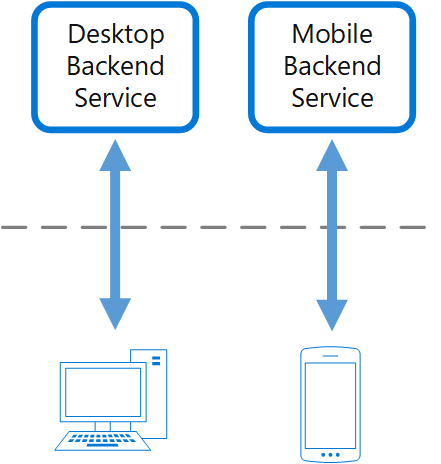
这个显著差异造成了后端的需求争夺。后端需要经常性地做大改动来同时服务于桌面Web UI和移动应用。经常性地，不同的界面团队工作在各自的前端，导致后端变成了开发过程中的瓶颈。互相冲突的需求更新，以及要保持服务同时为两端工作，会导致在一个单一可部署的资源上花费太多努力。



因为开发活动集中在后端服务，可以创建一个单独的团队来管理和维护后端。这个终极后果是造成界面与后端的联系断开，从而把平衡来自不同界面团队的互相冲突的需求的负担加给了后端团队。当一个界面团队要求后端做改动时，在将这些改动集成进后端之前必须和其他的界面团队做确认。

解决方案：

为每个用户界面创建一个后端。微调每个后端的行为与表现来匹配前端环境的需要的同时不担心影响其他的前端体验。



因为每个前端都配有一个单独的后端，这个后端就能专门为其前端做优化。从而这个后端会变得更小，复杂度更低，并且很可能比深度满足所有界面需求的通用后端跑起来更快。每个界面团队能掌控各自的后端，不必再依赖一个中心的后端开发团队。这给界面团队在语言选择、发布节奏、工作负荷的优先级调整、以及后端特性集成上很大的灵活性。

问题和值得考虑的地方：

* 考虑要部署多少个后端
* 如果不同的界面（像不同的移动客户端）有着同样的请求，考虑是否值得去为每个不同界面部署不同的后端，在这种情况下，可能一个后端就足够了。
* 应用此模式，不同的服务间很可能充斥着重复的代码。
* 专门服务于前端的后端服务只应该包含与客户端有关的逻辑和行为。通用的业务逻辑和其他全局特性应该在应用的其他地方来管理。
* 想一想这个模式如何在开发团队中通过他们的职责体现出来。
* 考虑应用这个模式需要多长时间实现。在支持已有的通用后端的同时，构建新后端的努力会带来技术债吗？

应用这个模式的时机：

* 在维护共用的或者通用的后端服务时需要大量的开发负担。
* 你想为不同的客户端界面优化相应的后端
* 为适应多个界面，通用的后端被做了大量的定制化。
* 某些用户界面的后端使用另一种语言开发会更好

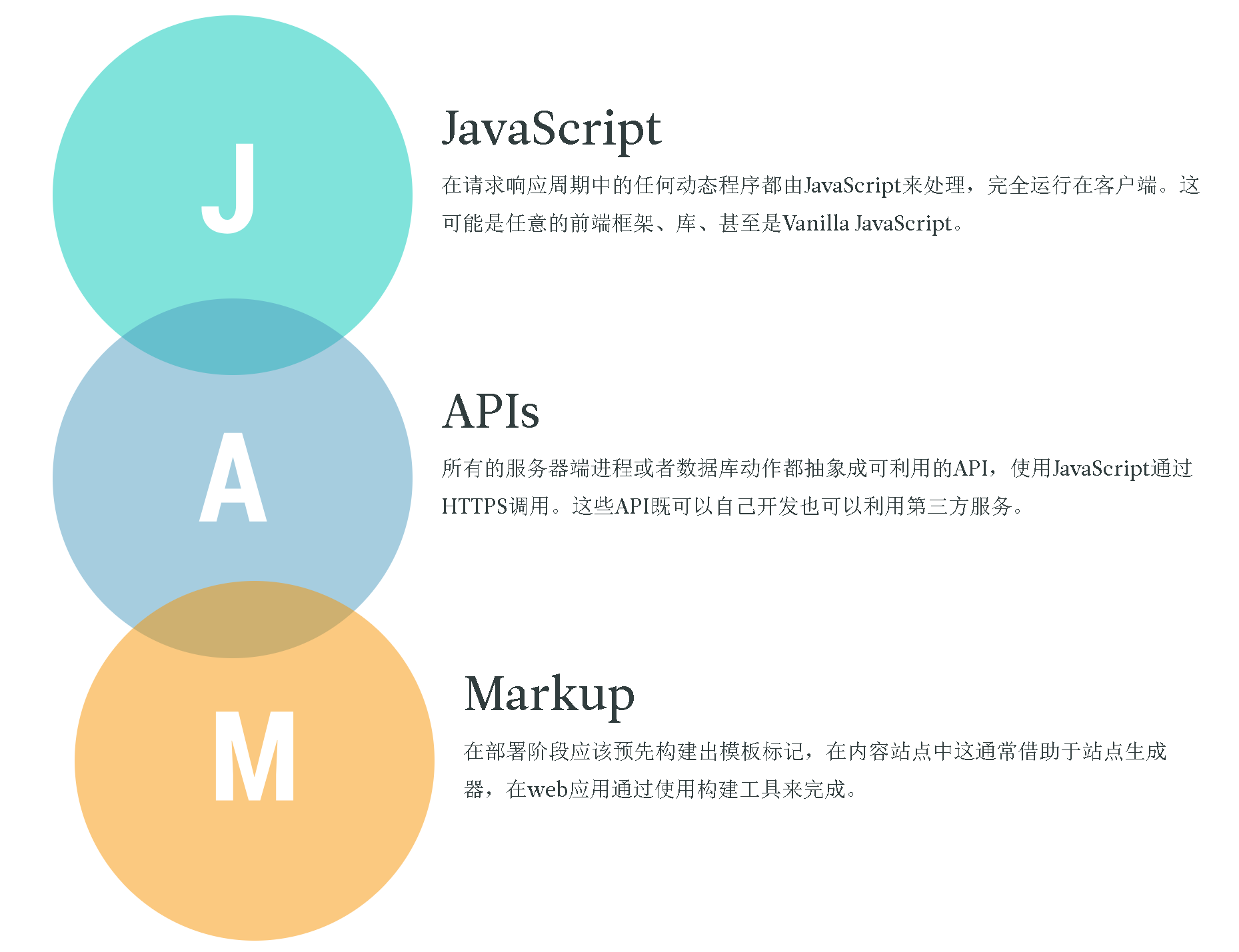
不适合使用这个模式的时机：

* 不同的界面发出同样的或者类似的请求到后端
* 与后端打交道的只有一个界面

## 整体应用架构

### JAMstack：

基于客户端JavaScript、可复用的API以及预编译好的标签。这是一种新型的构建网站与应用的方式，它交付更好的性能、更高的安全性、更低成本的扩展以及更好的开发体验。



* JavaScript

在请求响应周期中的任何动态程序都由JavaScript来处理，完全运行在客户端。这可能是任意的前端框架、库、甚至是Vanilla JavaScript。

* APIs

所有的服务器端进程或者数据库动作都抽象成可利用的API，使用JavaScript通过HTTPS调用。这些API既可以自己开发也可以利用第三方服务。

* Markups

在部署阶段应该预先构建出模板标记，在内容站点中这通常借助于站点生成器，在web应用通过使用构建工具来完成。

不是以JAMstack构建的站点都有哪些标志？

* 使用Wordpress、Drupal、Joomla或者SquareSpace等服务器端CMS系统构建的站点
* 使用同构渲染在服务器的运行时构建视图的单页应用。
* 依赖Ruby、Node或者其他后端语言的运行在服务器上的单一Web应用

JAMstack的好处：

* 更好的性能：明明在部署时就能生成的页面，为什么要在线等呢？在不断降低到达用户的第一比特时间的今天，再也没有比通过CDN来服务预先生成的文件更好的解决方案了。
* 更便宜、更易扩展：当你的部署可以放置在任何地方的文件堆服务时，扩展问题就只是在更多的地方去放置这些文件罢了。
* 更高安全性：当服务器端进程被抽象成微服务API，攻击表面就被缩小了。你也能够利用第三方服务的领域专家级安全保护。
* 更好的开发体验：检耦合和控制分离允许更有针对性的开发和调试，而且因为生成站点的CMS选项被大大拓宽，维护不同的内容与市场的应用堆的需要就不再有了。

最佳实践：

* 整个项目都扔在CDN上：因为JAMstack项目不依赖服务器端代码，他们不需要存在单个服务器上而是可以分布式部署。直接从CDN服务，速度与性能无与伦比。你能越多地应用尖端技术在你的应用上，用户体验就越好。
* 自动化构建：因为JAMstack标记是预生成的，除非做个新的部署，否则内容不会更新。自动化这个过程让你不会因此感到疲惫。你可以通过WebHook，或者一个发布系统来自动完成这件事情。
* 所有的东西都通过Git跟踪变化：每个人都能通过做一个git clone、通过标准流程安装必要依赖（如npm install），就能在本地完全运行这个项目。不需要复制数据库，没有复杂的安装，这减少了贡献者的阻力，也简化了预演和测试工作量。
* 原子化部署：随着JAMstack项目增长得非常庞大，新的改动可能需要重新部署成百上千个文件。一个一个上传这些文件在部署完成前会造成不一致的状态。你能通过支持“原子性部署”的系统来避免这种情况，除非所有的文件都上传完成，否则不会有任何改动上线。
* 现代化的构建工具：利用好现代构建工具。由于构建工具多如牛毛，可能在刚开始使用它们时会不知西东，但是利用好它们，你不用等到明天的浏览器出来，就可以在今天开始使用明天的Web标准。目前这意味着Babel，PostCSS，Webpack等等。
* 即刻缓存失效：当构建-部署循环变成日常，你需要确保当部署上线时，就真的上线了。要消除所有的疑虑你需要确保你的CDN能够处理即刻缓存清除。

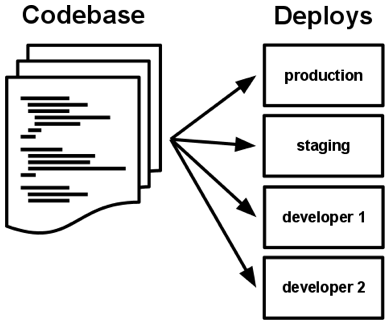
### 12-因子Web应用：

* + - 简介：如今，软件通常会作为一种服务来交付，它们被称为网络应用程序，或软件即服务（SaaS）。12-Factor 为构建如下的 SaaS 应用提供了方法论：
      1. 使用标准化流程自动配置，从而使新的开发者花费最少的学习成本加入这个项目。
      2. 和操作系统之间尽可能的划清界限，在各个系统中提供最大的可移植性。
      3. 适合部署在现代的云计算平台，从而在服务器和系统管理方面节省资源。
      4. 将开发环境和生产环境的差异降至最低，并使用持续交付实施敏捷开发。
      5. 可以在工具、架构和开发流程不发生明显变化的前提下实现扩展。

这套理论适用于任意语言和后端服务（数据库、消息队列、缓存等）开发的应用程序。

12-因子：

1. 基准代码：一份基准代码，多份部署



基准代码和应用之间总是保持一一对应的关系：

* 一旦有多个基准代码，就不能称为一个应用，而是一个分布式系统。分布式系统中的每一个组件都是一个应用，每一个应用可以分别使用 12-因子 进行开发。
* 多个应用共享一份基准代码是有悖于 12-Factor 原则的。解决方案是将共享的代码拆分为独立的类库，然后使用 依赖管理 策略去加载它们。

尽管每个应用只对应一份基准代码，但可以同时存在多份部署。每份 部署 相当于运行了一个应用的实例。通常会有一个生产环境，一个或多个预发布环境。此外，每个开发人员都会在自己本地环境运行一个应用实例，这些都相当于一份部署。

所有部署的基准代码相同，但每份部署可以使用其不同的版本。比如，开发人员可能有一些提交还没有同步至预发布环境；预发布环境也有一些提交没有同步至生产环境。但它们都共享一份基准代码，我们就认为它们只是相同应用的不同部署而已。

1. 依赖：显式声明依赖关系

大多数编程语言都会提供一个打包系统，用来为各个类库提供打包服务，就像 Perl 的 CPAN 或是 Ruby 的 Rubygems 。通过打包系统安装的类库可以是系统级的（称之为 “site packages”），或仅供某个应用程序使用，部署在相应的目录中（称之为 “vendoring” 或 “bunding”）。

12-Factor规则下的应用程序不会隐式依赖系统级的类库。 它一定通过 依赖清单 ，确切地声明所有依赖项。此外，在运行过程中通过 依赖隔离 工具来确保程序不会调用系统中存在但清单中未声明的依赖项。这一做法会统一应用到生产和开发环境。

例如， Ruby 的 Bundler 使用 Gemfile 作为依赖项声明清单，使用 bundle exec 来进行依赖隔离。Python 中则可分别使用两种工具 – Pip 用作依赖声明， Virtualenv 用作依赖隔离。甚至 C 语言也有类似工具， Autoconf 用作依赖声明，静态链接库用作依赖隔离。无论用什么工具，依赖声明和依赖隔离必须一起使用，否则无法满足 12-Factor 规范。

显式声明依赖的优点之一是为新进开发者简化了环境配置流程。新进开发者可以检出应用程序的基准代码，安装编程语言环境和它对应的依赖管理工具，只需通过一个 构建命令 来安装所有的依赖项，即可开始工作。例如，Ruby/Bundler 下使用 bundle install，而 Clojure/Leiningen 则是 lein deps。

| **语言** | **包管理工具** | **安装包** | **依赖声明** | **隔离工具** |
| --- | --- | --- | --- | --- |
| Perl | CPAN |  |  |  |
| Ruby | Rubygems | bundle install | Gemfile | bundle exec |
| Python | pip | pip install -r requirements.txt | requirements.txt | virutalenv |
| C |  |  | Autoconf | static linking |
| Node | npm | npm install | package.json |  |

12-因子应用同样不会隐式依赖某些系统工具，如 ImageMagick 或是curl。即使这些工具存在于几乎所有系统，但终究无法保证所有未来的系统都能支持应用顺利运行，或是能够和应用兼容。如果应用必须使用到某些系统工具，那么这些工具应该被包含在应用之中。

1. 配置：在环境中存储配置

通常，应用的 配置 在不同 部署 (预发布、生产环境、开发环境等等)间会有很大差异。这其中包括：

* 数据库，Memcached，以及其他 后端服务 的配置
* 第三方服务的证书，如 Amazon S3、Twitter 等
* 每份部署特有的配置，如域名等

有些应用在代码中使用常量保存配置，这与 12-Factor 所要求的代码和配置严格分离显然大相径庭。配置文件在各部署间存在大幅差异，代码却完全一致。

判断一个应用是否正确地将配置排除在代码之外，一个简单的方法是看该应用的基准代码是否可以立刻开源，而不用担心会暴露任何敏感的信息。

需要指出的是，这里定义的“配置”并不包括应用的内部配置，比如 Rails 的 config/routes.rb，或是使用 Spring 时 代码模块间的依赖注入关系 。这类配置在不同部署间不存在差异，所以应该写入代码。

另外一个解决方法是使用配置文件，但不把它们纳入版本控制系统，就像 Rails 的 config/database.yml 。这相对于在代码中使用常量已经是长足进步，但仍然有缺点：总是会不小心将配置文件签入了代码库；配置文件的可能会分散在不同的目录，并有着不同的格式，这让找出一个地方来统一管理所有配置变的不太现实。更糟的是，这些格式通常是语言或框架特定的。

12-Factor推荐将应用的配置存储于 环境变量 中（ env vars, env ）。环境变量可以非常方便地在不同的部署间做修改，却不动一行代码；与配置文件不同，不小心把它们签入代码库的概率微乎其微；与一些传统的解决配置问题的机制（比如 Java 的属性配置文件）相比，环境变量与语言和系统无关。

配置管理的另一个方面是分组。有时应用会将配置按照特定部署进行分组（或叫做“环境”），例如Rails中的 development,test, 和 production 环境。这种方法无法轻易扩展：更多部署意味着更多新的环境，例如 staging 或 qa 。 随着项目的不断深入，开发人员可能还会添加他们自己的环境，比如 joes-staging ，这将导致各种配置组合的激增，从而给管理部署增加了很多不确定因素。

12-Factor 应用中，环境变量的粒度要足够小，且相对独立。它们永远也不会组合成一个所谓的“环境”，而是独立存在于每个部署之中。当应用程序不断扩展，需要更多种类的部署时，这种配置管理方式能够做到平滑过渡。

争议:

* 环境变量隐式地对进程可见。这就很难跟踪环境变量的开放情况，比如很容易就可能把环境变量全部打印出来作为错误报告的一部分发送出去了。
* 如果没有显式的筛选，整个环境变量会被传递给所有的子进程。所以你的秘钥隐式地就对第三方工具可见了。很难说第三方工具会对环境变量做什么，特别是在极端情况下会崩溃。
* 外部开发者并不清楚环境变量中有隐秘信息，多数期待环境变量里并没有什么特别信息，只有一般的系统配置，这很危险。

解决方案：使用配置文件，交给服务器的配置管理软件（如chef）管理。通过这种方式，我们把秘钥存储在代码库之外，并且使用管理服务器相同的方式管理配置文件。

结论：

不管采用什么方式，重点是将配置与代码分离，并了解各种方案的优缺点，保护好秘钥的安全。

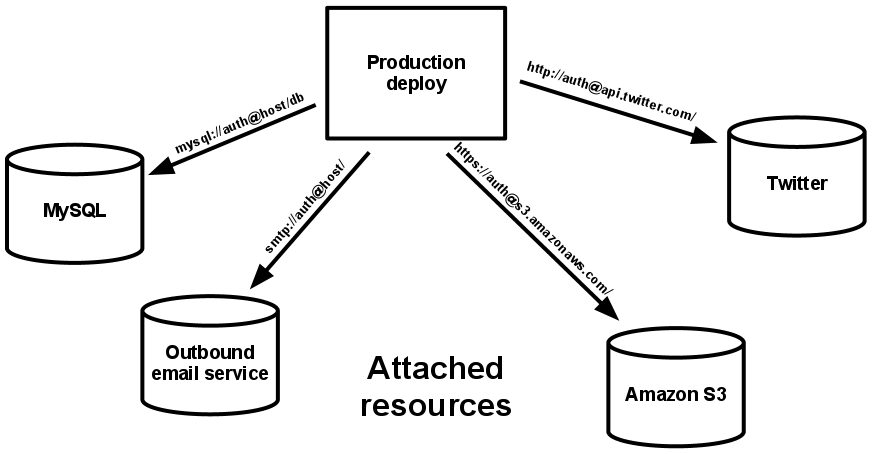
1. 后端服务：把后端服务当作附加资源

后端服务是指程序运行所需要的通过网络调用的各种服务，如数据库（MySQL，CouchDB），消息/队列系统（RabbitMQ，Beanstalkd），SMTP 邮件发送服务（Postfix），以及缓存系统（Memcached）。

类似数据库的后端服务，通常由部署应用程序的系统管理员一起管理。除了本地服务之外，应用程序有可能使用了第三方发布和管理的服务。示例包括 SMTP（例如 Postmark），数据收集服务（例如 New Relic 或 Loggly），数据存储服务（如 Amazon S3），以及使用 API 访问的服务（例如 Twitter, Google Maps, Last.fm）。

12-Factor 应用不会区别对待本地或第三方服务。 对应用程序而言，两种都是附加资源，通过一个 url 或是其他存储在 配置 中的服务定位/服务证书来获取数据。12-Factor 应用的任意 部署 ，都应该可以在不进行任何代码改动的情况下，将本地 MySQL 数据库换成第三方服务（例如 Amazon RDS）。类似的，本地 SMTP 服务应该也可以和第三方 SMTP 服务（例如 Postmark ）互换。上述 2 个例子中，仅需修改配置中的资源地址。

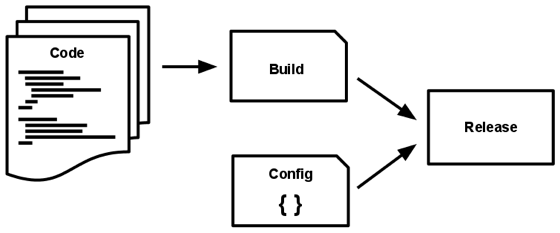
每个不同的后端服务是一份 资源 。例如，一个 MySQL 数据库是一个资源，两个 MySQL 数据库（用来数据分区）就被当作是 2 个不同的资源。12-Factor 应用将这些数据库都视作 附加资源 ，这些资源和它们附属的部署保持松耦合。



1. 构建，发布，运行：严格分离构建和运行

基准代码 转化为一份部署(非开发环境)需要以下三个阶段：

* 构建阶段是指将代码仓库转化为可执行包的过程。构建时会使用指定版本的代码，获取和打包 依赖项，编译成二进制文件和资源文件。
* 发布阶段会将构建的结果和当前部署所需 配置 相结合，并能够立刻在运行环境中投入使用。
* 运行阶段（或者说“运行时”）是指针对选定的发布版本，在执行环境中启动一系列应用程序 进程。



12-factor 应用严格区分构建，发布，运行这三个步骤。 举例来说，直接修改处于运行状态的代码是非常不可取的做法，因为这些修改很难再同步回构建步骤。

部署工具通常都提供了发布管理工具，最引人注目的功能是退回至较旧的发布版本。比如， Capistrano 将所有发布版本都存储在一个叫 releases 的子目录中，当前的在线版本只需映射至对应的目录即可。该工具的 rollback 命令可以很容易地实现回退版本的功能。

每一个发布版本必须对应一个唯一的发布 ID，例如可以使用发布时的时间戳（2011-04-06-20:32:17），亦或是一个增长的数字（v100）。发布的版本就像一本只能追加的账本，一旦发布就不可修改，任何的变动都应该产生一个新的发布版本。

新的代码在部署之前，需要开发人员触发构建操作。但是，运行阶段不一定需要人为触发，而是可以自动进行。如服务器重启，或是进程管理器重启了一个崩溃的进程。因此，运行阶段应该保持尽可能少的模块，这样假设半夜发生系统故障而开发人员又捉襟见肘也不会引起太大问题。构建阶段是可以相对复杂一些的，因为错误信息能够立刻展示在开发人员面前，从而得到妥善处理。

1. 进程：以一个或多个无状态进程运行应用

运行环境中，应用程序通常是以一个和多个 进程 运行的。

最简单的场景中，代码是一个独立的脚本，运行环境是开发人员自己的笔记本电脑，进程由一条命令行（例如python my\_script.py）。另外一个极端情况是，复杂的应用可能会使用很多 进程类型 ，也就是零个或多个进程实例。

12-Factor 应用的进程必须无状态且 无共享 。 任何需要持久化的数据都要存储在 后端服务 内，比如数据库。

内存区域或磁盘空间可以作为进程在做某种事务型操作时的缓存，例如下载一个很大的文件，对其操作并将结果写入数据库的过程。12-Factor应用根本不用考虑这些缓存的内容是不是可以保留给之后的请求来使用，这是因为应用启动了多种类型的进程，将来的请求多半会由其他进程来服务。即使在只有一个进程的情形下，先前保存的数据（内存或文件系统中）也会因为重启（如代码部署、配置更改、或运行环境将进程调度至另一个物理区域执行）而丢失。

源文件打包工具（Jammit, django-compressor） 使用文件系统来缓存编译过的源文件。12-Factor 应用更倾向于在 构建步骤 做此动作——正如 Rails资源管道 ，而不是在运行阶段。

一些互联网系统依赖于 “粘性 session”， 这是指将用户 session 中的数据缓存至某进程的内存中，并将同一用户的后续请求路由到同一个进程。粘性 session 是 12-Factor 极力反对的。Session 中的数据应该保存在诸如 Memcached 或 Redis 这样的带有过期时间的缓存中。

1. 端口绑定：通过端口绑定提供服务

互联网应用有时会运行于服务器的容器之中。例如 PHP 经常作为 Apache HTTPD 的一个模块来运行，正如 Java 运行于 Tomcat 。

12-Factor 应用完全自我加载 而不依赖于任何网络服务器就可以创建一个面向网络的服务。互联网应用 通过端口绑定来提供服务 ，并监听发送至该端口的请求。

本地环境中，开发人员通过类似http://localhost:5000/的地址来访问服务。在线上环境中，请求统一发送至公共域名而后路由至绑定了端口的网络进程。

通常的实现思路是，将网络服务器类库通过 依赖声明 载入应用。例如，Python 的 Tornado, Ruby 的Thin , Java 以及其他基于 JVM 语言的 Jetty。完全由 用户端 ，确切的说应该是应用的代码，发起请求。和运行环境约定好绑定的端口即可处理这些请求。

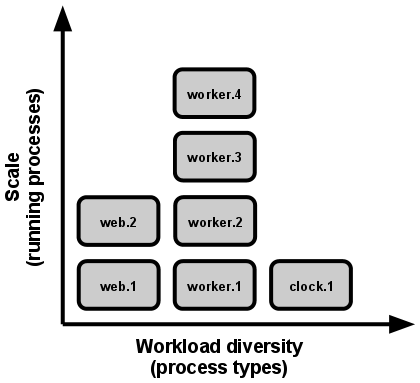
HTTP 并不是唯一一个可以由端口绑定提供的服务。其实几乎所有服务器软件都可以通过进程绑定端口来等待请求。例如，使用 XMPP 的 ejabberd ， 以及使用 Redis 协议 的 Redis 。

还要指出的是，端口绑定这种方式也意味着一个应用可以成为另外一个应用的 后端服务 ，调用方将服务方提供的相应 URL 当作资源存入 配置 以备将来调用。

1. 并发：通过进程模型进行扩展

任何计算机程序，一旦启动，就会生成一个或多个进程。互联网应用采用多种进程运行方式。例如，PHP 进程作为 Apache 的子进程存在，随请求按需启动。Java 进程则采取了相反的方式，在程序启动之初 JVM 就提供了一个超级进程储备了大量的系统资源(CPU 和内存)，并通过多线程实现内部的并发管理。上述 2 个例子中，进程是开发人员可以操作的最小单位。

在 12-factor 应用中，进程是一等公民。12-Factor 应用的进程主要借鉴于 unix 守护进程模型 。开发人员可以运用这个模型去设计应用架构，将不同的工作分配给不同的 进程类型 。例如，HTTP 请求可以交给 web 进程来处理，而常驻的后台工作则交由 worker 进程负责。



这并不包括个别较为特殊的进程，例如通过虚拟机的线程处理并发的内部运算，或是使用诸如 EventMachine, Twisted, Node.js 的异步/事件触发模型。但一台独立的虚拟机的扩展有瓶颈（垂直扩展），所以应用程序必须可以在多台物理机器间跨进程工作。

上述进程模型会在系统急需扩展时大放异彩。 12-Factor 应用的进程所具备的无共享，水平分区的特性 意味着添加并发会变得简单而稳妥。这些进程的类型以及每个类型中进程的数量就被称作 进程构成 。

12-Factor 应用的进程 不需要守护进程 或是写入 PID 文件。相反的，应该借助操作系统的进程管理器(例如 systemd ，分布式的进程管理云平台，或是类似 Foreman 的工具)，来管理 输出流 ，响应崩溃的进程，以及处理用户触发的重启和关闭超级进程的请求。

1. 易处理：快速启动和优雅终止可最大化健壮性

12-Factor 应用的 进程 是 易处理（disposable）的，意思是说它们可以瞬间开启或停止。 这有利于快速、弹性的伸缩应用，迅速部署变化的 代码 或 配置 ，稳健的部署应用。

进程应当追求 最小启动时间 。 理想状态下，进程从敲下命令到真正启动并等待请求的时间应该只需很短的时间。更少的启动时间提供了更敏捷的 发布 以及扩展过程，此外还增加了健壮性，因为进程管理器可以在授权情形下容易的将进程搬到新的物理机器上。

进程 一旦接收 终止信号（SIGTERM） 就会优雅的终止 。就网络进程而言，优雅终止是指停止监听服务的端口，即拒绝所有新的请求，并继续执行当前已接收的请求，然后退出。此类型的进程所隐含的要求是HTTP请求大多都很短(不会超过几秒钟)，而在长时间轮询中，客户端在丢失连接后应该马上尝试重连。

对于 worker 进程来说，优雅终止是指将当前任务退回队列。例如，RabbitMQ 中，worker 可以发送一个NACK信号。 Beanstalkd 中，任务终止并退回队列会在worker断开时自动触发。有锁机制的系统诸如 Delayed Job 则需要确定释放了系统资源。此类型的进程所隐含的要求是，任务都应该 可重复执行 ， 这主要由将结果包装进事务或是使重复操作 幂等 来实现。

进程还应当在面对突然死亡时保持健壮，例如底层硬件故障。虽然这种情况比起优雅终止来说少之又少，但终究有可能发生。一种推荐的方式是使用一个健壮的后端队列，例如 Beanstalkd ，它可以在客户端断开或超时后自动退回任务。无论如何，12-Factor 应用都应该可以设计能够应对意外的、不优雅的终结。Crash-only design 将这种概念转化为 合乎逻辑的理论。

1. 开发环境与线上环境等价：尽可能的保持开发，预发布，线上环境相同

从以往经验来看，开发环境（即开发人员的本地 部署）和线上环境（外部用户访问的真实部署）之间存在着很多差异。这些差异表现在以下三个方面：

* 时间差异: 开发人员正在编写的代码可能需要几天，几周，甚至几个月才会上线。
* 人员差异: 开发人员编写代码，运维人员部署代码。
* 工具差异: 开发人员或许使用 Nginx，SQLite，OS X，而线上环境使用 Apache，MySQL 以及 Linux。

12-Factor 应用想要做到 持续部署 就必须缩小本地与线上差异。 再回头看上面所描述的三个差异:

* 缩小时间差异: 开发人员可以几小时，甚至几分钟就部署代码。
* 缩小人员差异: 开发人员不只要编写代码，更应该密切参与部署过程以及代码在线上的表现。
* 缩小工具差异: 尽量保证开发环境以及线上环境的一致性。

将上述总结变为一个表格如下：

| **传统应用** | **12-因子应用** |
| --- | --- |
| 每次部署间隔 | 数周 | 几小时 |
| 开发人员 vs 运维人员 | 不同的人 | 相同的人 |
| 开发环境 vs 线上环境 | 不同 | 尽量接近 |

后端服务 是保持开发与线上等价的重要部分，例如数据库，队列系统，以及缓存。许多语言都提供了简化获取后端服务的类库，例如不同类型服务的 适配器 。下列表格提供了一些例子。

| **类型** | **语言** | **类库** | **适配器** |
| --- | --- | --- | --- |
| 数据库 | Ruby/Rails | ActiveRecord | MySQL, PostgreSQL, SQLite |
| 队列 | Python/Django | Celery | RabbitMQ, Beanstalkd, Redis |
| 缓存 | Ruby/Rails | ActiveSupport::Cache | Memory, filesystem, Memcached |

开发人员有时会觉得在本地环境中使用轻量的后端服务具有很强的吸引力，而那些更重量级的健壮的后端服务应该使用在生产环境。例如，本地使用 SQLite 线上使用 PostgreSQL；又如本地缓存在进程内存中而线上存入 Memcached。

12-Factor 应用的开发人员应该反对在不同环境间使用不同的后端服务 ，即使适配器已经可以几乎消除使用上的差异。这是因为，不同的后端服务意味着会突然出现的不兼容，从而导致测试、预发布都正常的代码在线上出现问题。这些错误会给持续部署带来阻力。从应用程序的生命周期来看，消除这种阻力需要花费很大的代价。

与此同时，轻量的本地服务也不像以前那样引人注目。借助于Homebrew，apt-get等现代的打包系统，诸如Memcached、PostgreSQL、RabbitMQ 等后端服务的安装与运行也并不复杂。此外，使用类似 Chef 和 Puppet 的声明式配置工具，结合像 Vagrant 这样轻量的虚拟环境就可以使得开发人员的本地环境与线上环境无限接近。与同步环境和持续部署所带来的益处相比，安装这些系统显然是值得的。

不同后端服务的适配器仍然是有用的，因为它们可以使移植后端服务变得简单。但应用的所有部署，这其中包括开发、预发布以及线上环境，都应该使用同一个后端服务的相同版本。

1. 日志：把日志当作事件流

日志 使得应用程序运行的动作变得透明。在基于服务器的环境中，日志通常被写在硬盘的一个文件里，但这只是一种输出格式。

日志应该是 事件流 的汇总，将所有运行中进程和后端服务的输出流按照时间顺序收集起来。尽管在回溯问题时可能需要看很多行，日志最原始的格式确实是一个事件一行。日志没有确定开始和结束，但随着应用在运行会持续的增加。

12-factor应用本身从不考虑存储自己的输出流。 不应该试图去写或者管理日志文件。相反，每一个运行的进程都会直接的标准输出（stdout）事件流。开发环境中，开发人员可以通过这些数据流，实时在终端看到应用的活动。

在预发布或线上部署中，每个进程的输出流由运行环境截获，并将其他输出流整理在一起，然后一并发送给一个或多个最终的处理程序，用于查看或是长期存档。这些存档路径对于应用来说不可见也不可配置，而是完全交给程序的运行环境管理。类似 Logplex 和 Fluentd 的开源工具可以达到这个目的。

这些事件流可以输出至文件，或者在终端实时观察。最重要的，输出流可以发送到 Splunk 这样的日志索引及分析系统，或 Hadoop/Hive 这样的通用数据存储系统。这些系统为查看应用的历史活动提供了强大而灵活的功能，包括：

* 找出过去一段时间特殊的事件。
* 图形化一个大规模的趋势，比如每分钟的请求量。
* 根据用户定义的条件实时触发警报，比如每分钟的报错超过某个警戒线。

1. 管理进程：后台管理任务当作一次性进程运行

进程构成（process formation）是指用来处理应用的常规业务（比如处理 web 请求）的一组进程。与此不同，开发人员经常希望执行一些管理或维护应用的一次性任务，例如：

运行数据移植（Django 中的 manage.py migrate, Rails 中的 rake db:migrate）。

运行一个控制台（也被称为 REPL shell），来执行一些代码或是针对线上数据库做一些检查。大多数语言都通过解释器提供了一个 REPL 工具（python 或 perl） ，或是其他命令（Ruby 使用 irb, Rails 使用 rails console）。

运行一些提交到代码仓库的一次性脚本。

一次性管理进程应该和正常的 常驻进程 使用同样的环境。这些管理进程和任何其他的进程一样使用相同的 代码 和 配置 ，基于某个 发布版本 运行。后台管理代码应该随其他应用程序代码一起发布，从而避免同步问题。

所有进程类型应该使用同样的 依赖隔离 技术。例如，如果Ruby的web进程使用了命令 bundle exec thin start ，那么数据库移植应使用 bundle exec rake db:migrate 。同样的，如果一个 Python 程序使用了 Virtualenv，则需要在运行 Tornado Web 服务器和任何 manage.py 管理进程时引入 bin/python 。

12-factor 尤其青睐那些提供了 REPL shell 的语言，因为那会让运行一次性脚本变得简单。在本地部署中，开发人员直接在命令行使用 shell 命令调用一次性管理进程。在线上部署中，开发人员依旧可以使用ssh或是运行环境提供的其他机制来运行这样的进程。

# 示例：

<https://fire.pa-pa.me/>

通过git提交代码，自动构建，部署上CDN。后端使用lambda，整体面向无服务器架构，前端采用GatsbyJS框架。