目录

Growth: 全栈增长工程师指南 **9**

全栈工程师是未来 9

技术的革新史 9

软件开发的核心难题：沟通 11

大公司的专家与小公司的全栈 14

全栈工程师的未来：无栈 17

基础知识篇 **18**

工具只是辅助 20

WebStorm 还是 Sublime? 20

语言也是一种工具 21

提高效率的工具 22

快速启动软件 22

IDE 23

DEBUG 工具 23

终端或命令提示符 24

包管理 26

环境搭建 27

OS X 27

Windows 28

GNU/Linux 29

学好一门语言的艺术 30

一次语言学习体验 30

输出是最好的输入 33

如何应用一门新的技术 34

Web 编程基础 35

从浏览器到服务器 35
从 HTML 到页面显示 38
HTML 39
hello,world 39
中文? 41
其他 HTML 标记 42
小结 43
CSS 44
简介 45
样式与目标 49
选择器 49
更有趣的 CSS 51
JavaScript 52
hello,world 52
JavaScriptFul 53
面向对象 59
其他 60

前端与后台 **62**
后台语言选择 63
JavaScript 63
Python 63
Java 63
PHP 64
其他 64
MVC 64
Model 66
View 67

Controller 68
更多 68
后台即服务 69
API 演进史 69
后台即服务 73
数据持久化 73
文件存储 73
数据库 74
搜索引擎 75
前端框架选择 77
Angular 77
React 77
Vue 77
jQuery 系 77
前台与后台交互 78
Ajax 78
JSON 79
WebSocket 81

编码 **81**

编码过程 82
Web 应用的构建系统 83
Web 应用的构建过程 84
Web 应用的构建实战 85
Git 与版本控制 88
版本控制 88
Git 88
Tasking 92

- 如何 Tasking 一本书 92
- Tasking 开发任务 93

写代码只是在码字 94

内置索引与外置引擎 96

- 门户网站 96
- 内置索引与外置引擎 97

如何编写测试 97

- 测试金字塔 97
- 如何测试 100

测试替身 103

- Stub 103
- Mock 104

测试驱动开发 104

- 红-绿-重构 104
- 测试先行 106

可读的代码 108

- 命名 109
- 函数长度 110
- 其他 110

代码重构 111

- 重命名 111
- 提取变量 111
- 提炼函数 112

Intellij Idea 重构 112

- 提炼函数 112
- 内联函数 116
- 查询取代临时变量 117

重构到设计模式 120
过度设计与设计模式 120

上线 **121**

隔离与运行环境 121
隔离硬件：虚拟机 122
隔离操作系统：容器虚拟化 124
隔离底层：Servlet 容器 127
隔离依赖版本：虚拟环境 128
隔离运行环境：语言虚拟机 128
隔离语言：DSL 132
LNMP 架构 132
GNU/Linux 133
HTTP 服务器 134
Web 缓存 134
数据库端缓存 135
应用层缓存 135
前端缓存 136
客户端缓存 136
HTML5 离线缓存 136
可配置 136
环境配置 136
运行机制 138
功能开关 138
自动化部署 139
依赖与包仓库 141
构建软件包 142
上传和安装软件包 142

数据分析 **144**

构建-衡量-学习 144

想法-构建 145

产品-衡量 146

数据-学习 146

数据分析 147

识别需求 147

收集数据 147

分析数据 148

展示数据 148

用户数据分析：Google Analytics 149

受众群体 149

流量获取 150

移动应用 151

网站性能 151

网站性能监测 151

网站性能 153

SEO 154

爬虫与索引 154

什么样的网站需要 SEO? 156

SEO 基础知识 156

内容 158

UX 入门 160

什么是 UX 160

什么是简单? 163

进阶 164

用户体验要素 164

认知设计 167
流 167

持续交付 168

持续集成 168
持续集成系统 169
持续集成流程 169
持续交付 171
基础设施 171
持续部署 173
持续学习 173
持续阅读 174
持续编程 174
持续写作 174

遗留系统与修改代码 175

遗留代码 176
遗留代码 176
如何修改遗留代码 176
修改遗留代码 177
网站重构 177
速度优化 178
功能加强 178
模块重构 179

回顾与架构设计 179

自我总结 179
吾日三省吾身 179
Retro 180

- Retro 的过程 180
- 三个维度 181
架构模式 183
- 预设计式架构 184
- 演进式架构：拥抱变化 185
浮现式设计 185
- 意图导向 186
- 重构 187
- 模式与演进 188
每个人都是架构师 188
- 如何构建一个博客系统 188
- 相关阅读资料 191
架构解耦 192
- 从 MVC 与微服务 192
- CQRS 198
- CQRS 结合微服务 203

Growth: 全栈增长工程师指南

这是一本不止于全栈工程师的学习手册，它也包含了如何成为一个 Growth Hacker 的知识。

全栈工程师是未来

> 谨以此文献给每一个为成为优秀全栈工程师奋斗的人。

技术在过去的几十年里进步很快，也将在未来的几十年里发展得更快。今天技术的门槛下降得越来越快，原本需要一个团队做出来的 Web 应用，现在只需要一两个人就可以了。

同时，由于公司组织结构的变迁，以及到变化的适应度，也决定了赋予每个人的职责将会越来越多。尽管我们看到工厂化生产带来的优势，但是我们也看到了精益思想带来的变革。正是这种变革让越来越多的专家走向全栈，让组织内部有更好的交流。

你还将看到专家和全栈的两种不同的学习模式，以及全栈工程师的未来。

技术的革新史

从开始的 CGI 到 MVC 模式，再到前后端分离的架构模式，都在不断地降低技术的门槛。而这些门槛的降低，已经足以让一两个人来完成大部分的工作了。

CGI 二十年前的网站以静态的形式出现，这样的网站并不需要太多的人去维护、管理。接着，人们发明了 CGI (通用网关接口，英语：Common Gateway Interface) 来实现动态的网站。下图是一个早期网站的架构图：

当时这种网站的 URL 类似于：

```
https://www.phodal.com/cgi-bin/getblog
```

(PS：这个链接是为了讲解而存在的，并没有真实存在。)

用户访问上面的网页的时候就会访问，cgi-bin 的路径下对应的 getblog 脚本。你可以用 Shell 返回这个网页：

```sh
#!/bin/sh
echo Content-type: text/plain
echo hello,world
```

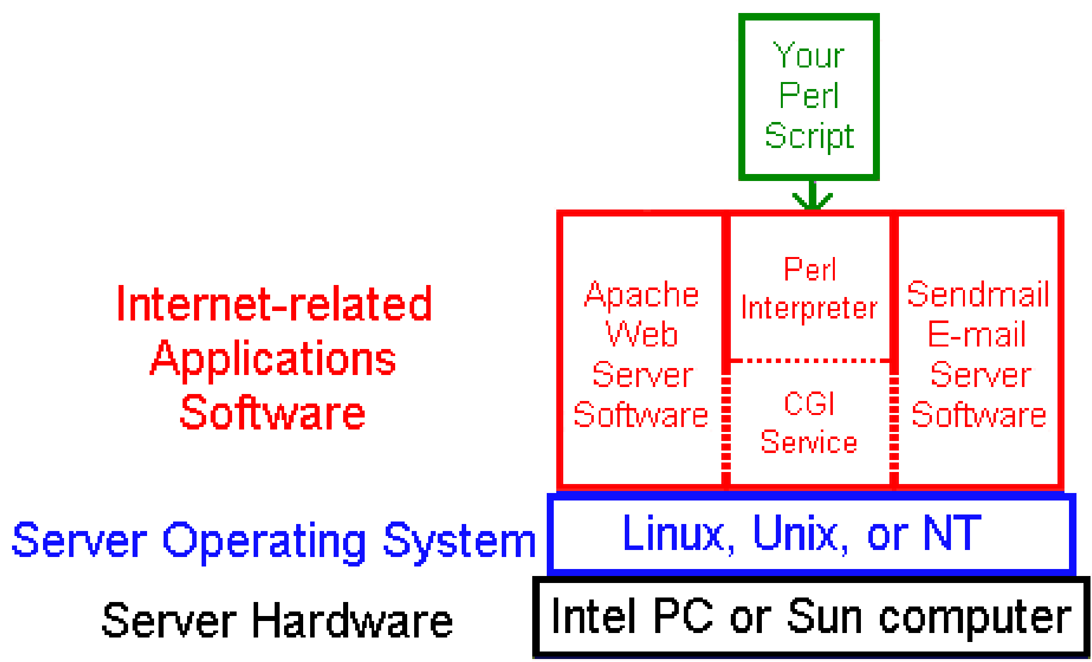


图 1: CGI 网站架构

Blabla，各种代码混乱地夹杂在一起。不得不说一句：这样的代码在 2012 年，我也看了有一些。简单地来说，这个时代的代码结构就是这样的：

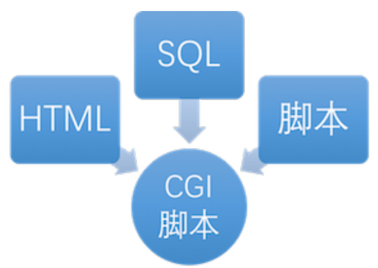


图 2: CGI 脚本文件

这简直就是一场恶梦。不过，在今天好似那些 PHP 新手也是这样写代码的。

好了，这时候我们就可以讨论讨论 MVC 模式了。

MVC 架构 我有理由相信 Martin Fowler 的《企业应用架构模式》在当时一定非常受欢迎。代码从上面的耦合状态变成了：

相似大家也已经对这样的架构很熟悉了，我们就不多解释了。如果你还不是非常了解的话，可以看看这本书后面的部分。

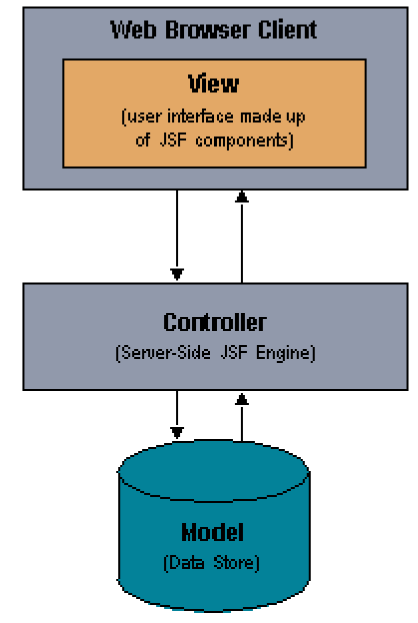


图 3: MVC 架构

后台服务化与前端一致化架构 在今天看来，我们可以看到如下图所示的架构：

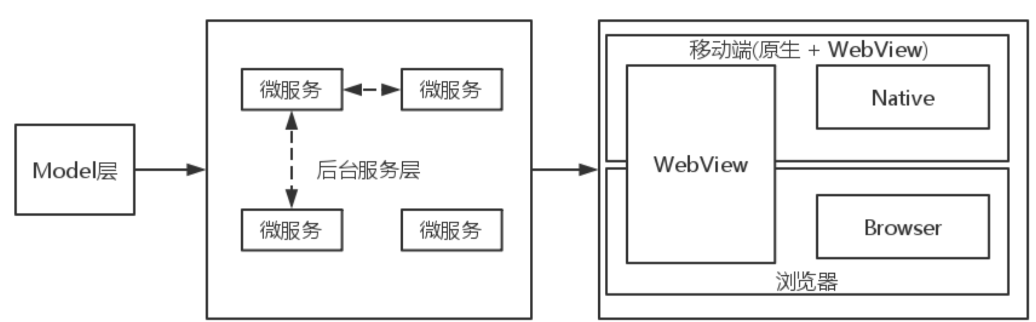


图 4: 后台服务化与前台一致化架构

后台在不知不觉中已经被服务化了，即只提供 API 接口和服务。前端在这时已经尽量地和 APP 端在结合，使得他们可以保持一致。

软件开发的核心难题：沟通

软件开发在过去的几十年里都是大公司的专利，小公司根本没有足够的能力去做这样的事。在计算机发明后的几十年里，开发软件是大公司才能做得起的。一般的非技术公司无法定制自己的软件系统，只能去购买现有的软件。而随着技术成本的下降，到了今天一般的小公司也可以雇佣一两个人来做同样的事。这样的演进过程还真是有意思：

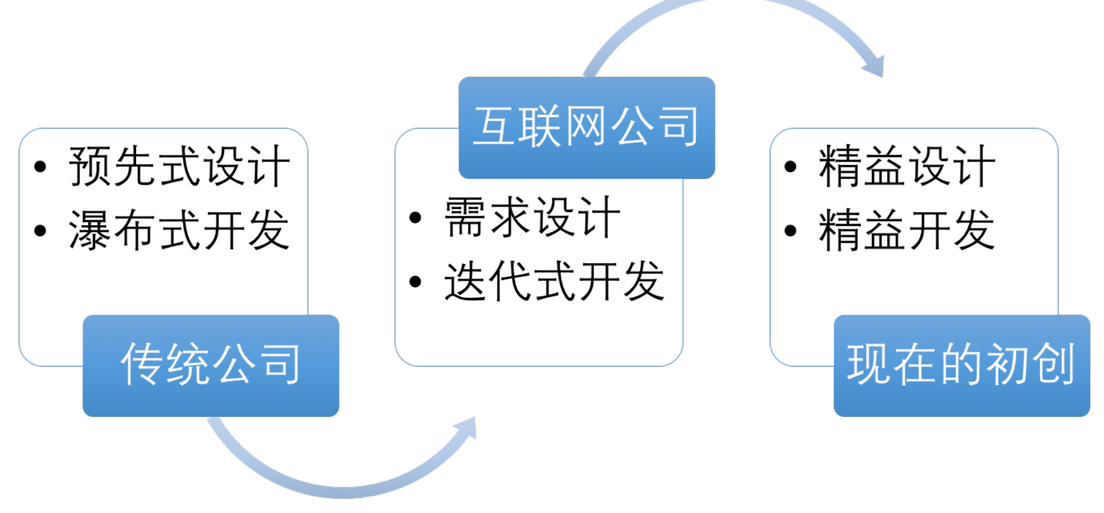


图 5: 开发演进

在这其中的每一个过程实质上都是为了解决沟通的问题。从瀑布到敏捷是为了解决组织内沟通的问题，从敏捷到精益不仅仅优化了组织内的沟通问题，还强化了与外部的关系。换句话说，精益结合了一部分的互联网思维。

瀑布式　在最开始的时候，我们预先设计好我们的功能，然后编码，在适当的时候发布我们的软件：

传统的开发流程 —— 瀑布模型

需求分析
分析设计
编码/单元测试
集成测试
系统测试

图 6: 预先式设计的瀑布流

然而这种开发方式很难应对市场的变化——当我们花费了几年的时间开发出了一个软件，而这个软件是几年前人们才需要的。同时，由于软件开发本身的复杂度的限制，复制的系统在后期需要大量的系统集成工作。这样的集成工作可能要花费上大量的时间——几星期、几个月。

图 7: 瀑布流的沟通模型

当人们意识到这个问题的时候，开始改进工作流程。出现了敏捷软件开发，这可以解释为什么产品经理会经常改需求。如果一个功能本身是没必要出现的话，那么为什么要花功夫去开发。但是如果一个功能在设计的初期就没有好好设计，那么改需求也是必然的。

敏捷式 现有的互联网公司的工作流程和敏捷软件开发在很多部分上是相似的，都有迭代、分析等等的过程：

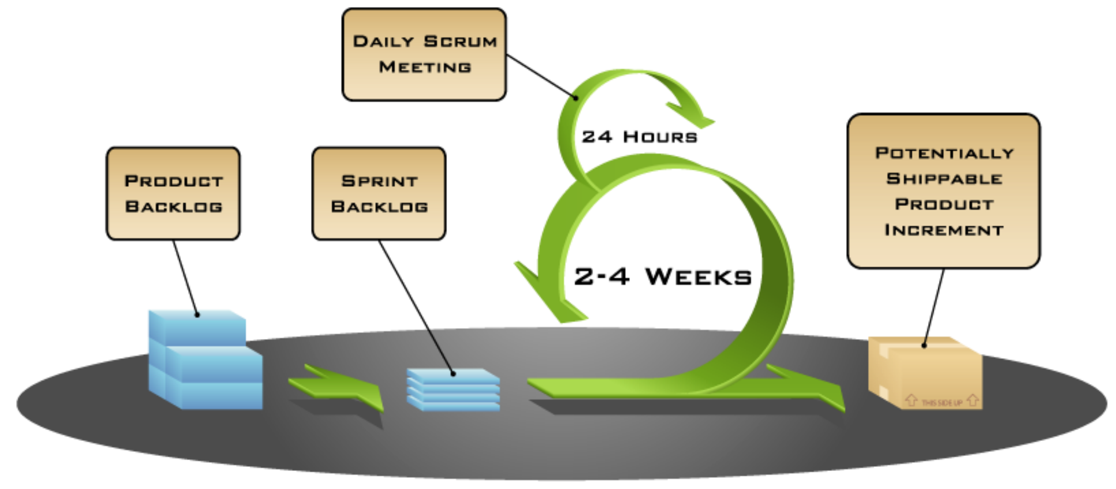


图 8: 敏捷软件开发

但是据我的所知：国内的多数互联网公司是不写测试的、没有 Code Review 等等。当然，这也不是一篇关于如何实践敏捷的文章。敏捷与瀑布式开发在很大的区别就是：沟通问题。传统的软件开发在调研完毕后就是分析、开发等等。而敏捷开发则会强调这个过程中的沟通问题：

在整个过程中都不断地强调沟通问题，然而这时还存在一个问题：组织结构本身的问题。这样的组织结构，如下图所示：

图 9: 敏捷软件开发的沟通模型

图 10: 组织结构

如果市场部门/产品经理没有与研发团队坐一起来分析问题，那么问题就多了。当一个需求在实现的过程中遇到问题，到底是哪个部门的问题?

同样的如果我们的研发部门是这样子的结构:

那么在研发、上线的过程中仍然会遇到各种的沟通问题。

现在，让我们回过头来看看大公司的专家与小公司的全栈。

大公司的专家与小公司的全栈

如果你经常看一些关于全栈和专家的技术文章的时候，你就会发现不同的人在强调不同的方向。大公司的文章喜欢强调成为某个领域的专家，小公司喜欢小而美的团队——全栈工程师。

如我们所见的：大公司和小公司都在解决不同类型的问题。大公司要解决性能问题，小公司都活下去需要依赖于近乎全能的人。并且，大公司和小公司都在加班。如果从这种意义上来说，我们可以发现其实大公司是在剥削劳动力。

专家

我们所见到的那些关于技术人员应该成为专家的文章，多数是已经成为某个技术领域里的专家写的文章。并且我们可以发现很有意思的一点是：他们都是管理者。管理者出于招聘的动机，因此更需要细分领域的专家来帮助他们解决问题。

全栈

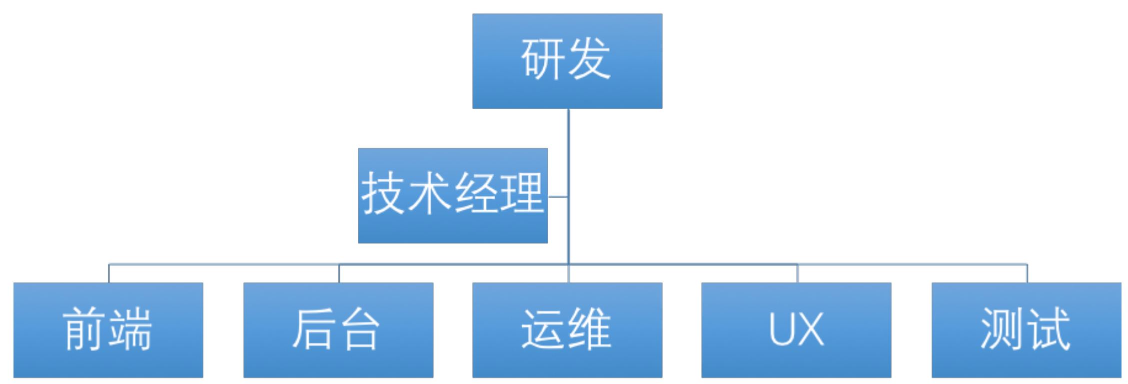


图 11: 研发部门

相似的，我们所看到的那些关于成为全栈工程师的文章，多数是初创公司的 CTO 写的。而这些初创公司的 CTO 也多数是全栈工程师，他们需要招聘全栈工程师来帮助他们解决问题。

两种不同的学习模型 而不知你是否也注意到一点：专家们也在强调 "一专多长"。因为单纯依靠于一个领域的技术而存在的专家已经很少了，技术专家们不得不依据于公司的需求去开拓不同的领域。毕竟 "公司是指全部资本由股东出资构成，以营利为目的而依法设立的一种企业组织形式；"，管理人们假设技术本身是相通的，既然你在技术领域里有相当高的长板，那么进入一个新的技术也不是一件难的事。

作为一个技术人员，我们是这个领域中的某个子领域专家。而作为这样一个专家，我们要扩展向另外一个领域的学习也不是一件很难的事。借鉴于我们先前的学习经验，我们可以很快的掌握这个新子域的知识。如我们所见，我们可以很快地补齐图中的短板：

在近来的探索中发现有一点非常有意思：如果依赖于 20/80 法则的话，那么成为专家和全栈的学习时间是相当的。在最开始的时候，我们要在我们的全栈工程和专家都在某个技术领域达到 80 分的水平。

那么专家，还需要 80% 的时间去深入这个技术领域。而全栈工程师，则可以依赖于这 80% 的时候去开拓四个新的领域：

尽管理论上是如此，但是专家存在跨领域的学习障碍——套用现有模式。而全栈也存在学习障碍——如何成为专家，但是懂得如何学习新的领域。

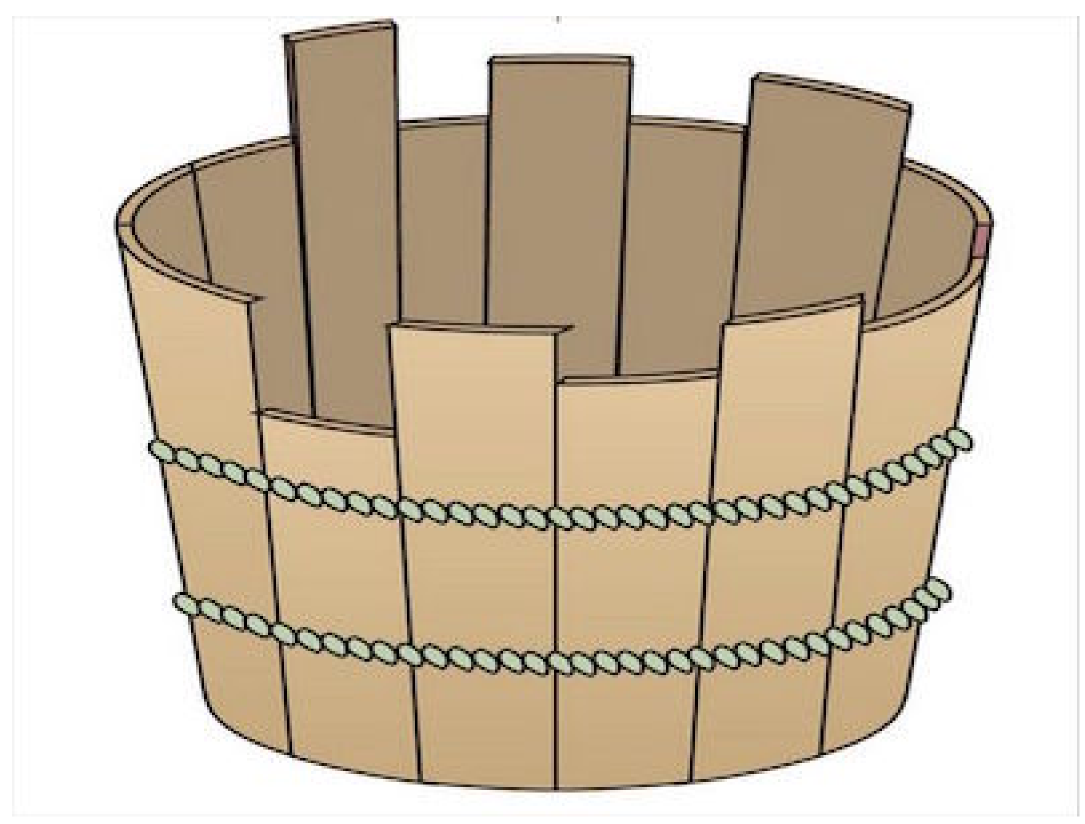


图 12: 木桶

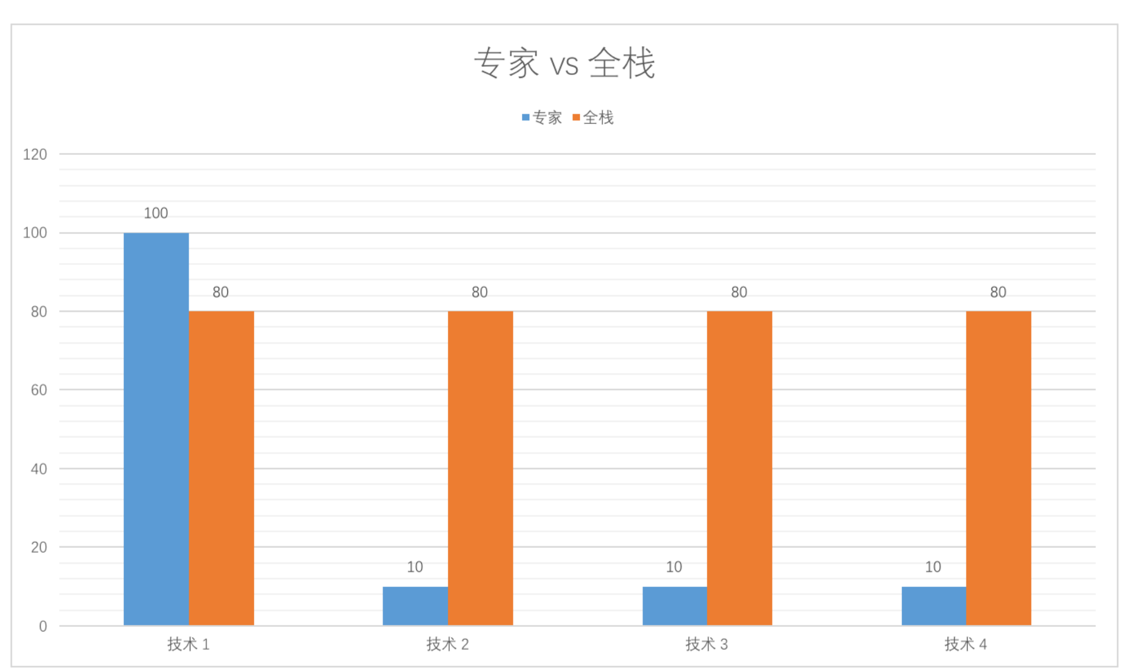


图 13: 全栈与专家学习时间

解决问题的思路：不同的方式　有意思的是——成为专家还是成为全栈，取决于人的天性，这也是两种不同的性格决定的。成为管理者还是技术人员看上去就像一种简单的划分，而在技术人员里成为专家还是全栈就是另外一种划分。这取决于人们对于一个问题的思考方式：这件事情是借由外部来解决，还是由内部解决。下面这张图刚好可以表达我的想法：

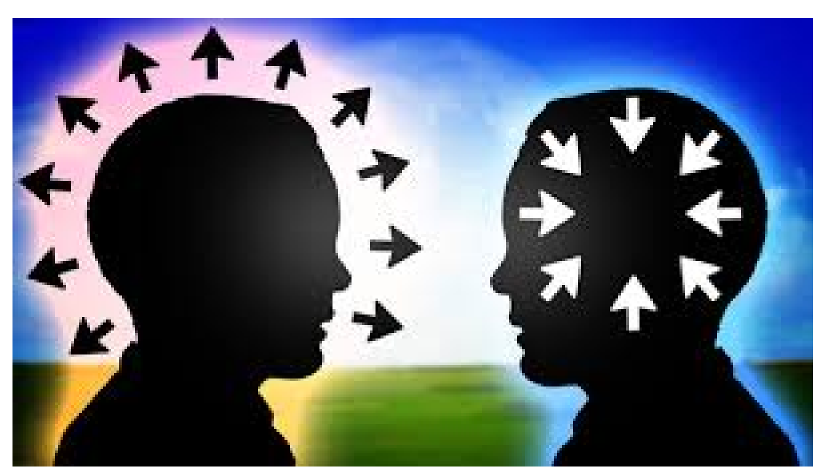


图 14: 内向与外向思维

而这种思维依据于不同的事情可能会发生一些差异，但是总体上来说是相似的。当遇到一个需要创轮子的问题时，我们就会看到两种不同的方式。

对于全栈工程师来说，他们喜欢依赖于外部的思维，用于产生颠覆式思维。如 AngularJS 这样的框架便是例子，前端结合后端开发语言 Java 的思维而产生。而专家则依赖于内部的条件，创造出不一样的适应式创新。如之前流行的 Backbone 框架，适应当时的情况而产生。

全栈工程师的未来：无栈

全栈工程师本身不应该仅仅局限于前端和后台的开发，而可以尝试去开拓更广泛的领域——因为全栈本身是依赖于工程师本身的学习能力，正是这种优秀的学习能力可以让他们可以接触更广泛的知识。

全栈的短板　如果你也尝试过面试过全栈工程师，你会怎么去面试他们呢？把你知道的所有的不同领域的问题都拿出来问一遍。是的，这就是那些招聘全栈工程师的公司会问你的问题。

人们以为全栈工程师什么都会，这是一个明显的误区——然而要改变这个误区很难。最后，导致的结果是大家觉得全栈工程师的水平也就那样。换句来说，人们根本不知道

什么是全栈工程师。在平时的工作里，你的队伍都知道你在不同领域有丰富的知识。而在那些不了解你的人的印象里，就是猜测你什么都会。

因此，这就会变成一个骂名，也是一个在目前看来很难改变的问题。在这方面只能尽可能地去了解一些通用的问题，并不能去了解所有的问题。在一次被面试全栈工程师的过程中，有一个面试官准备了几个不同语言（JavaScript、Java、Python、Ruby)的问题来问我，我只想说 Ciao —— 意大利语：你好！

除了这个问题——人们不了解什么是全栈工程师。还有一个问题，就是刚才我们说的成为专家的老大难问题。

无栈　让我毫不犹豫地选择当全栈工程师有两个原因：

1. 这个世界充满了未解的迷，但是我只想解开我感兴趣的部分。
2. 没有探索，哪来的真爱？你都没有探索过世界，你就说这是你最喜欢的领域。

当我第一次看到全栈工程师这个名字的时候，我发现我已然是一个全栈工程师。因为我的学习路线比较独特：

中小学：编程语言 -> 高中：操作系统、内核、游戏编程 -> 大学：硬件、Web 开发 -> 工作：后端 + 前端

而在当时我对 SEO 非常感兴趣，我发现这分析和 Marketing 似乎做得还可以。然后便往 Growth Hacking 发展了：

而这就是全栈学习带来的优势，学过的东西多，学习能力就变强。学习能力往上提的同时，你就更容易进入一个新的领域。

参考书籍

- 《精益企业：高效能组织如何规模化创新》
- 《企业应用架构模式》
- 《敏捷软件开发》
- 《技术的本质》

基础知识篇

在我们第一次开始写程序的时候，都是以 Hello World 开始的。或者：

```
printf("hello,world");
```

图 15: Growth Hacking

又或许：

```
alert('hello,world');
```

过去的十几年里，试过用二十几种不同的语言，每个都是以 hello,world 作为开头。在一些特定的软件，如 Nginx，则是 **It Works**。

这是一个很长的故事，这个程序最早出现于 1972 年，由贝尔实验室成员布莱恩・柯林汉撰写的内部技术文件《A Tutorial Introduction to the Language B》之中。不久，同作者于 1974 年所撰写的《Programming in C: A Tutorial》，也延用这个范例；而以本文件扩编改写的《C 语言程序设计》也保留了这个范例程式。工作时，我们也会使用类似于 hello,world 的 boilerplate 来完成基本的项目创建。

同时需要注意的一点是，在每个大的项目开始之前我们应该去找寻好开发环境。搭建环境是一件非常重要的事，它决定了你能不能更好地工作。毕竟环境是生产率的一部分。高效的程序员和低效程序员间的十倍差距，至少有三倍是因为环境差异。

因此在这一章里，我们将讲述几件事情：

1. 使用怎样的操作系统
2. 如何去选择工具

3. 如何搭建相应操作系统上的环境
4. 如何去学习一门语言

工具只是辅助

一个好的工具确实有助于编程，但是他只会给我们带来的是帮助。我们写出来的代码还是和我们的水平保持着一致的。

什么是好的工具，这个说法就有很多了，但是有时候我们往往沉迷于事物的表面。有些时候 Vim 会比 Visual Studio 强大，当你只需要修改的是一个配置文件的时候，简单且足够快捷——在我们还未用 VS 打开的时候，我们已经用 Vim 做完这个活了。

> “好的装备确实能带来一些帮助，但事实是，你的演奏水平是由你自己的手指决定的。”– 《REWORK》

WebStorm 还是 **Sublime?**

作为一个 IDE 有时候忽略的因素会过多，一开始的代码由类似于 Sublime text 之类的编辑器开始会比较合适。于是我们又开始陷入 IDE 及 Editor 之战了，无聊的时候讨论一下这些东西是有点益处的。相互了解一下各自的优点，也是不错的，偶尔可以换个环境试试。

刚开始学习的时候，我们只需要普通的工具，或者我们习惯了的工具去开始我们的工作。我们要的是把主要精力放在学习的东西上，而不是工具。刚开始学习一种新的语言的时候，我们不需要去讨论哪个是最好的开发工具，如 Java，有时候可能是 Eclipse，有时候可能是 Vim，如果我们为的只是去写一个 hello,world。在 Eclipse 上浪费太多的时间是不可取的，因为他用起来的效率可不比你在键盘上敲打来得快，当你移动你的手指去动你的鼠标的时候，我想你可以用那短短的时间完成编译，运行了。

工具是为了效率　寻找工具的目的和寻找捷径是一样的，我们需要更快更有效率地完成我们的工作，换句话说，我们为了获取更多的时间用于其他的事情。而这个工具的用途是要看具体的事物的，如果我们去写一个小说、博客的时候，word 或者 web editor 会比 tex studio 来得快，不是么。我们用 TEX 来排版的时候会比我们用 WORD 排版的时候来得更快，所以这个工具是相对而论的。有时候用一个顺手的工具会好很多，但是不一定会是事半功倍的。我们应该将我们的目标专注于我们的内容，而不是我们的工具上。

我们用 Windows 自带的画图就可以完成裁剪的时候，我们就没必要运行起 GIMP 或者 Photoshop 去完成这个简单的任务。效率在某些时候的重要性，会比你选择的工具

有用得多，学习的开始就是要去了解那些大众推崇的东西。

了解、熟悉你的工具 Windows 的功能很强大，只是大部分人用的是只是小小一部分。而不是一小部分，即使我们天天用着，我们也没有学习到什么新的东西。和这个就如同我们的工具一样，我们天天用着他们，如果我们只用 Word 来写写东西，那么我们可以用 Abiword 来替换他。但是明显不太可能，因为强大的工具对于我们来说有些更大的吸引力。

如果你负担得起你手上的工具的话，那么就尽可能去了解他能干什么。即使他是一些无关仅要的功能，比如 Emacs 的煮咖啡。有一本手册是最好不过的，手册在手边可以即时查阅，不过出于环保的情况下，就不是这样子的。手册没有办法即时同你的软件一样更新，电子版的更新会比你手上用的那个手册更新得更快。

Linux 下面的命令有一大堆，只是我们常用的只有一小部分——20% 的命令能够完成 80% 的工作。如同 CISC 和 RISC 一样，我们所常用的指令会让我们忘却那些不常用的指令。而那些是最实用的，如同我们日常工作中使用的 Linux 一样，记忆过多的不实用的东西，不比把他们记在笔记上实在。我们只需要了解有那些功能，如何去用他。

语言也是一种工具

越来越多的框架和语言出现、更新得越来越快。特别是这样一个高速发展的产业，每天都在涌现新的名词。如同我们选择语言一样，选择合适的有时候会比选得顺手的来得重要。然而，这个可以不断地被推翻。

当我们熟悉用 Python、Ruby、PHP 等去构建一个网站的时候，JavaScript 用来做网站后台，这怎么可能——于是 Node.js 火了。选择工具本身是一件很有趣的事，因为有着越来越多的可能性。

过去 PHP 是主流的开发，不过现在也是，PHP 为 WEB 而生。有一天 Ruby on Rails 出现了，一切就变了，变得高效，变得更 Powerful。MVC 一直很不错，不是么? 于是越来越多的框架出现了，如 Django，Laravel 等等。不同的语言有着不同的框架，JavaScript 上也有着合适的框架，如 AngularJS。不同语言的使用者们用着他们合适的工具，因为学习新的东西，对于多数的人来说就是一种新的挑战。在学面向对象语言的时候，人们很容易把程序写成过程式的。

没有合适的工具，要么创造一个，要么选择一个合适的。

小结 学习 Django 的时候习惯了有一个后台，于是开始使用 Laravel 的时候，寻找 Administartor。需要编译的时候习惯用 IDE，不需要的时候用 Editor，只是因为有效

率，嵌入式的时候 IDE 会有效率一点。

以前不知道 WebStorm 的时候，习惯用 DW 来格式化 HTML，Aptana 来格式化 JavaScript。

以前，习惯用 WordPress 来写博客，因为可以有移动客户端，使用电脑时就不喜欢打开浏览器去写。

等等

等

提高效率的工具

在提交效率的 N 种方法里：有一个很重要的方法是使用快捷键。熟练掌握快捷键可以让我们随着自己的感觉编写程序——有时候如果我们手感不好，是不是就说明今天不适合写代码！笑 ~~

由于我们可能使用不同的操作系统来完成不同的工具。下面就先说说一些通用的、不限操作的工具：

快速启动软件

在我还不和道有这样的工具的时候，我都是把图标放在下面的任务栏里：

图 16: Windows 任务栏

直到有一天，我知道有这样的工具。这里不得不提到一本书《卓有成效的程序员》，在书中提到了很多提高效率的工具。使用快捷键是其中的一个，而还有一个是使用快速启动软件。于是，我在 Windows 上使用了 Launcy:

通过这个软件，我们可以在电脑上通过输入软件名，然后运行相关的软件。我们不

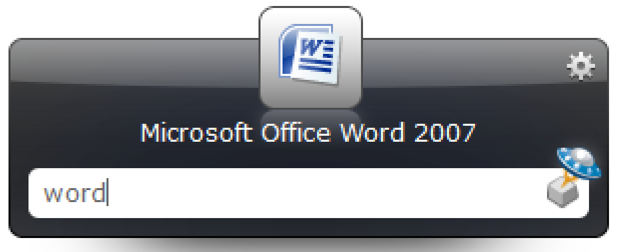


图 17: Launchy

再需要点击某个菜单，再从菜单里选中某个软件打开。

IDE

尽管在上一篇中，我们说过 IDE 和编辑器没有什么好争论的。但是如果是从头开始搭建环境的话，IDE 是最好的——编辑器还需要安装相应的插件。所以，这也就是为什么面试的时候会用编辑器的原因。

IDE 的全称是集成开发环境，顾名思义即它集成了你需要用到的一些工具。而如果是编辑器的话，你需要自己去找寻合适的工具来做这件事。不过，这也意味着使用编辑器会有更多的自由度。如果你没有足够的时间去打造自己的开发环境就使用 IDE 吧。

一般来说，他们都应该有下面的一些要素：

- 快捷键
- **Code HighLight**
- **Auto Complete**
- **Syntax Check**

而如果是编辑器的话，就需要自己去找寻这些相应的插件。

IDE 一般是针对特定语言才产生的，并且优化更好。而，编辑器则需要自己去搭配。这也意味着如果你需要在多个语言上工作时，并且喜欢折腾，你可以考虑使用编辑器。

DEBUG 工具

不得不提及的是在有些 IDE 自带了 Debug 工具，这点可能使得使用 IDE 更有优势。在简单的项目是，我们可能不需要这样的 Debug 工具。因为我们对我们的代码库比较熟悉，一个简单的问题一眼就知道是哪里的问题。而对于那些复杂的项目来说，可能就

没有那么简单了。特别是当你来到一个新的大中型项目，一个简单的逻辑在实现上可能要经过一系列的函数才能处理完。

这时候我们就需要 Debug 工具——对于前端开发来说，我们可能使用 Chrome 的 Dev Tools。但是对于后端来说，我们就需要使用别的工具。如下图所示的是 Intellij Idea 的 Debug 界面：

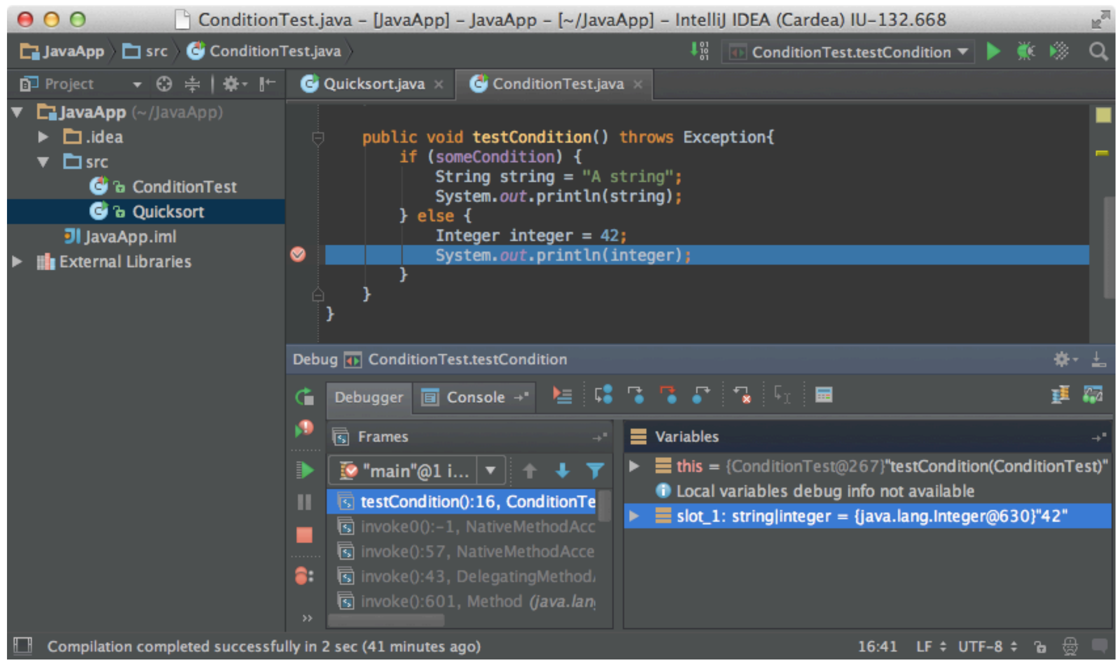


图 18: Intellij Idea Debug

在 Debug 的过程中，我们可以根据代码的执行流程一步步向下执行。这也意味着，当出现 Bug 的时候我们可以更容易找到 Bug。这就是为什么他叫 Debug 工具的原因了。

终端或命令提示符

在开始写代码的时候，使用 GUI 可能是难以戒掉的一个习惯。但是当你习惯了使用终端之后，或者说使用终端的工具，你会发现这是另外一片天空。对于 GUI 应用上同样的菜单来说，在终端上也会有同样的工具——只是你觉得记住更多的命令。而且不同的工具对于同一实现可能会不同的规范，而 GUI 应用则会有一致的风格。不过，总的来说使用终端是一个很有益的习惯——从速度、便捷性。忘了提到一点，当你使用 Linux 服务器的时候，你不得不使用终端。

使用终端的优点在于我们可以摆脱鼠标的操作——这可以让我们更容易集中精力于完成任务。而这也是两种不同的选择，便捷还是更快。虽是如此，但是这也意味着学习 Linux 会越来越轻松。

```
Last login: Mon Mar  7 04:32:54 2016 from 210.74.157.146

       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-ami/2015.09-release-notes/
No packages needed for security; 3 packages available
Run "sudo yum update" to apply all updates.
[ec2-user@ip-172-31-18-86 ~]$ sudo yum update
Loaded plugins: fastestmirror, priorities, update-motd, upgrade-helper
Loading mirror speeds from cached hostfile
 * amzn-main: packages.us-west-2.amazonaws.com
 * amzn-updates: packages.us-west-2.amazonaws.com
amzn-main/latest
amzn-updates/latest
28 packages excluded due to repository priority protections
Resolving Dependencies
--> Running transaction check
---> Package newrelic-daemon.x86_64 0:5.5.0.154-1 will be updated
---> Package newrelic-daemon.x86_64 0:6.0.0.155-1 will be an update
---> Package newrelic-php5.x86_64 0:5.5.0.154-1 will be updated
---> Package newrelic-php5.x86_64 0:6.0.0.155-1 will be an update
---> Package newrelic-php5-common.noarch 0:5.5.0.154-1 will be updated
---> Package newrelic-php5-common.noarch 0:6.0.0.155-1 will be an update
--> Finished Dependency Resolution

Dependencies Resolved

==================================================================================================
 Package                                        Arch                              Version
==================================================================================================
Updating:
 newrelic-daemon                                x86_64                            6.0.0.155-1
 newrelic-php5                                  x86_64                            6.0.0.155-1
 newrelic-php5-common                           noarch                            6.0.0.155-1
```

图 19: Linux 终端截图

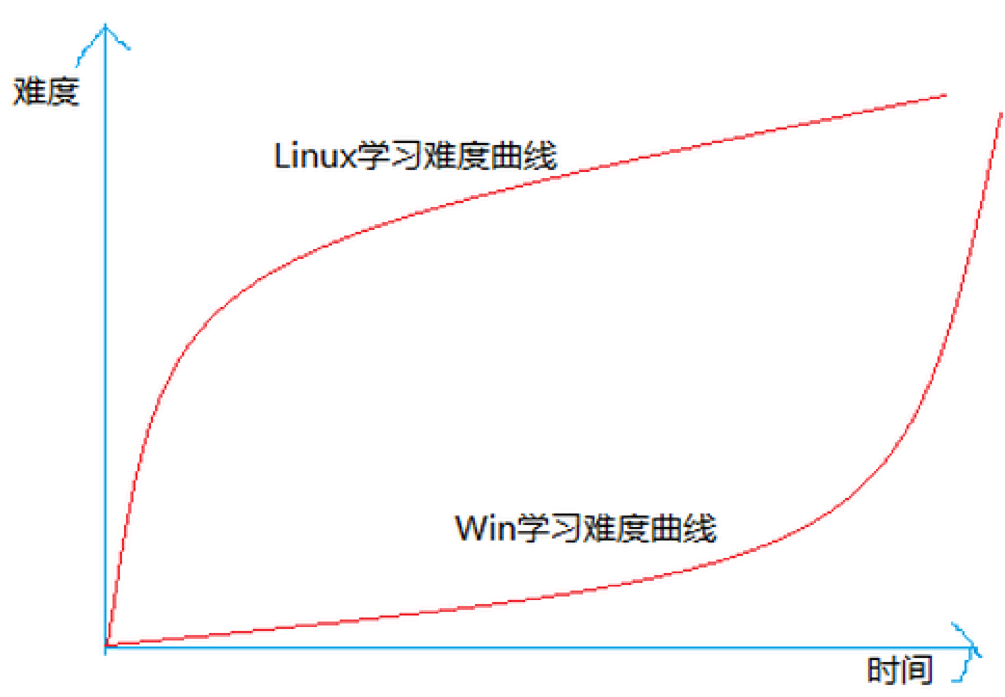


图 20: Linux 与 Windows 的学习曲线

虽然这是以 Linux 和 Windows 作了两个不同的对比，但是两个系统在终端工具上的差距是很大的。Linux 自身的哲学鼓励使用命令行来完成任务，这也意味着在 Linux 上会有更多的工具可以在命令行下使用。虽然 Windows 上也可以——如使用 CygWin 来完成，但是这看上去并不是那么让人满意！

包管理

虽然包管理不仅仅存在于操作系统中，还存在着语言的包管理工具。在操作系统中安装软件，最方便的东西莫过于包管理了。引自 OpenSUSE 官网的说明及图片：

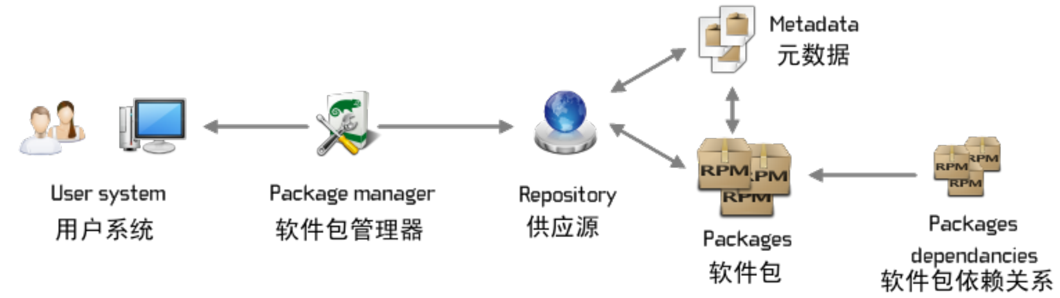


图 21: 包管理

Linux 发行版无非就是一堆软件包 (package) 形式的应用程序加上整体地管理这些应用程序的工具。通常这些 Linux 发行版，包括 OpenSUSE，都是由成千上万不同的软件包构成的。

- 软件包：软件包不止是一个文件，内含构成软件的所有文件，包括程序本身、共享库、开发包以及使用说明等。
- 元数据 (metadata) 包含于软件包之中，包含软件正常运行所需要的一些信息。软件包安装之后，其元数据就存储于本地的软件包数据库之中，以用于软件包检索。
- 依赖关系 (dependencies) 是软件包管理的一个重要方面。实际上每个软件包都会涉及到其他的软件包，软件包里程序的运行需要有一个可执行的环境（要求有其他的程序、库等），软件包依赖关系正是用来描述这种关系的。

我们经常会使用各式各样的包管理工具，来加速我们地日常使用。而不是 Google 某个软件，然后下载，接着安装。

环境搭建

由于我近期工具在 Mac OS X 上，所以先以 Mac OS X 为例。

OS X

Homebrew

包管理工具，官方称之为 The missing package manager for OS X。

Homebrew Cask

brew-cask 允许你使用命令行安装 OS X 应用。

iTerm2

iTerm2 是最常用的终端应用，是 Terminal 应用的替代品。

Zsh

Zsh 是一款功能强大终端（shell）软件，既可以作为一个交互式终端，也可以作为一个脚本解释器。它在兼容 Bash 的同时 (默认不兼容，除非设置成 emulate sh) 还有提供了很多改进，例如：

- 更高效
- 更好的自动补全
- 更好的文件名展开（通配符展开）
- 更好的数组处理
- 可定制性高

Oh My Zsh

Oh My Zsh 同时提供一套插件和工具，可以简化命令行操作。

Sublime Text 2

强大的文件编辑器。

MacDown

MacDown 是 Markdown 编辑器。

CheatSheet

CheatSheet 能够显示当前程序的快捷键列表，默认的快捷键是长按 ⌘。

SourceTree

SourceTree 是 Atlassian 公司出品的一款优秀的 Git 图形化客户端。

Alfred

Mac 用户不用鼠标键盘的必备神器，配合大量 Workflows，习惯之后可以大大减少操作时间。

上手简单，调教成本在后期自定义 Workflows，不过有大量雷锋使用者提供的现成扩展，访问这里挑选喜欢的，并可以极其简单地根据自己的需要修改。

Vimium

Vimium 是一个 Google Chrome 扩展，让你可以纯键盘操作 Chrome。

相关参考：

- Mac web developer apps
- 强迫症的 Mac 设置指南

Windows

Chocolatey

Chocolatey 是一个软件包管理工具，类似于 Ubuntu 下面的 apt-get, 不过是运行在 Windows 环境下面。

Wox

Wox 是一个高效的快速启动器工具，通过快捷键呼出，然后输入关键字来搜索程序进行快速启动，或者搜索本地硬盘的文件，打开百度、Google 进行搜索，甚至是通过一些插件的功能实现单词翻译、关闭屏幕、查询剪贴板历史、查询编程文档、查询天气等更多功能。它最大的特点是可以支持中文拼音的模糊匹配。

PowerShell

Windows PowerShell 是微软公司为 Windows 环境所开发的壳程序（shell）及脚本语言技术，采用的是命令行界面。这项全新的技术提供了丰富的控制与自动化的系统管理能力。

cmder

cmder 把 conemu，msysgit 和 clink 打包在一起，让你无需配置就能使用一个真正干净的 Linux 终端！她甚至还附带了漂亮的 monokai 配色主题。

Total Commander

Total Commander 是一款应用于 Windows 平台的文件管理器，它包含两个并排的窗口，这种设计可以让用户方便地对不同位置的“文件或文件夹”进行操作，例如复制、移动、删除、比较等，相对 Windows 资源管理器而言方便很多，极大地提高了文件操作的效率，被广大软件爱好者亲切地简称为：TC 。

GNU/Linux

Zsh

Zsh 是一款功能强大终端（shell）软件，既可以作为一个交互式终端，也可以作为一个脚本解释器。它在兼容 Bash 的同时 (默认不兼容，除非设置成 emulate sh) 还有提供了很多改进，例如：

- 更高效
- 更好的自动补全
- 更好的文件名展开（通配符展开）
- 更好的数组处理
- 可定制性高

Oh My Zsh

Oh My Zsh 同时提供一套插件和工具，可以简化命令行操作。

ReText

> ReText 是一个使用 Markdown 语法和 reStructuredText (reST) 结构的文本编辑器，编辑的内容支持导出到 PDF、ODT 和 HTML 以及纯文本，支持即时预览、网页生成以及 HTML 语法高亮、全屏模式，可导出文件到 Google Docs 等。

Launchy

> Launchy 是一款免费开源的协助您摒弃 Windows “运行” 的 Dock 式替代工具，既方便又实用，自带多款皮肤，作为美化工具也未尝不可。

环境搭建完毕！现在，就让我们来看看如何学好一门语言！

学好一门语言的艺术

一次语言学习体验

在我们开始学习一门语言或者技术的时候，我们可能会从一门 hello,world 开始。

好了，现在我是 Scala 语言的初学者，接着我用搜索引擎去搜索『Scala』来看看『Scala』是什么鬼：

> Scala 是一门类 Java 的编程语言，它结合了面向对象编程和函数式编程。

接着又开始看『Scala ‘hello,world’』，然后找到了这样的一个示例：

```
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello, world!")
  }
}
```

GET 到了 5% 的知识。

看上去这门语言相比于 Java 语言来说还行。然后我找到了一本名为『Scala 指南』的电子书，有这样的一本目录：

- 表达式和值
- 函数是一等公民
- 借贷模式

- 按名称传递参数
- 定义类
- 鸭子类型
- 柯里化
- 范型
- Traits
- ...

看上去还行，又 GET 到了 5% 的知识点。接着，依照上面的代码和搭建指南在自己的电脑上安装了 Scala 的环境：

```
brew install scala
```

Windows 用户可以用:

```
choco install scala
```

然后开始写一个又一个的 Demo，感觉自己 GET 到了很多特别的知识点。

到了第二天忘了！

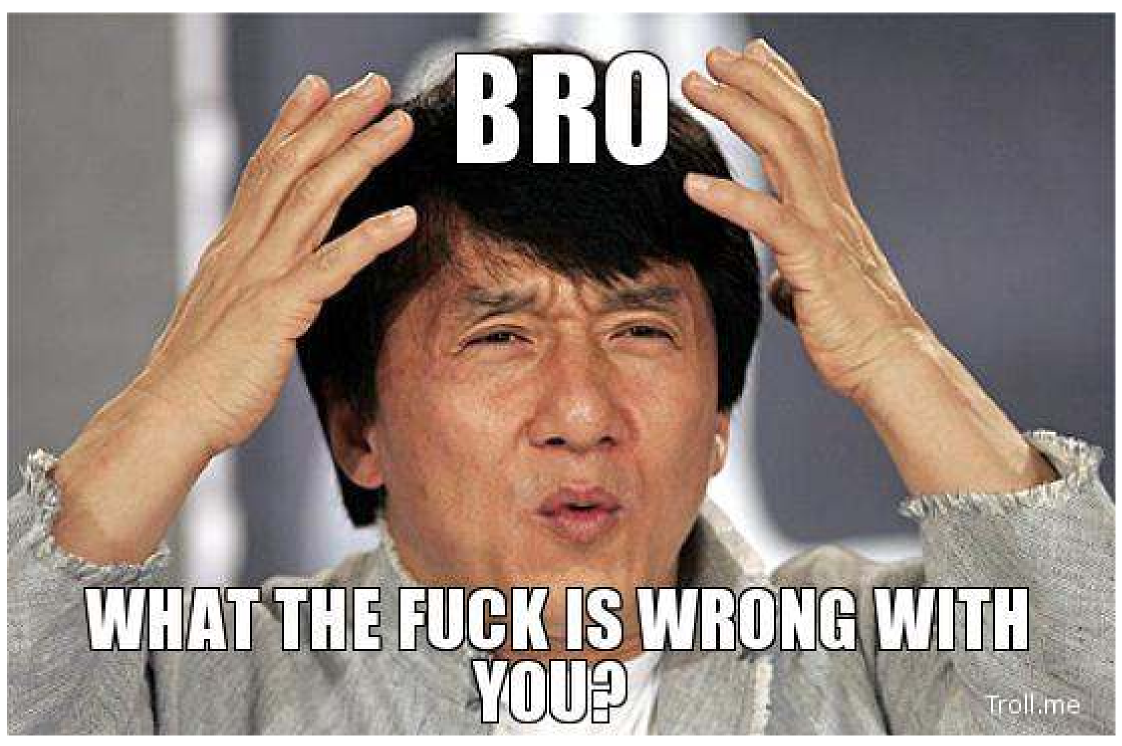


图 22: Bro Wrong

接着，你又重新把昨天的知识过了一遍，还是没有多大的作用。突然间，你听到别人在讨论什么是这个世界上最好的语言——你开始加入讨论了。

于是，你说出了 Scala 这门语言可以：

- 支持高阶函数。lambda，闭包...
- 支持偏函数。match..
- mixin，依赖注入..
- 等等

虽然隔壁的 Python 小哥赢得了这次辩论，然而你发现你又回想起了 Scala 的很多特性。

图 23: 最流行的语言

你发现隔壁的 Python 小哥之所以赢得了这场辩论是因为他把 Python 语言用到了各个地方——机器学习、人工智能、硬件、Web 开发、移动应用等。而你还没有用 Scala 写过一个真正的应用。

让我想想我来能做什么？我有一个博客。对，我有一个博客，我可以用 Scala 把我的博客重写一遍：

1. 先找一 Scala 的 Web 框架，Play 看上去很不错，就这个了。这是一个 MVC 框架，原来用的 Express 也是一个 MVC 框架。Router 写这里，Controller 类似这个，就是这样的。
2. 既然已经有 PyJS，也会有 Scala-js，前端就用这个了。

好了，博客重写了一遍了。

感觉还挺不错的，我决定向隔壁的 Java 小弟推销这门语言，以解救他于火海之中。

『让我想想我有什么杀手锏?』

『这里的知识好像还缺了一点，这个是什么?』

好了，你已经 GET 到了 90% 了。如下图所示：

图 24: Learn

希望你能从这张图上 GET 到很多点。

输出是最好的输入

上面那张图『学习金字塔』就是在说明——输出是最好的输入。

如果你不试着去写点博客、整理资料、准备分享，那么你可能并没有意识到你缺少了多少东西。虽然你已经有了很多的实践，然并卵。

因为你一直在完成功能、完成工作，你总会有意、无意地漏掉一些知识，而你也没有意识到这些知识的重要性。

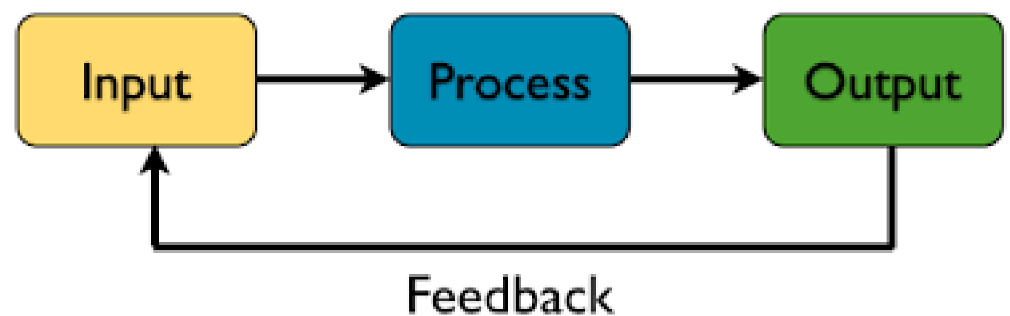


图 25: Output is Input

从我有限的（500+）博客写作经验里，我发现多数时候我需要更多地的参考资料才能更好也向人们展示这个过程。为了输出我们需要更多的输入，进而加速这个过程。

而如果是写书的时候则是一个更高水平的学习，你需要发现别人在他们的书中欠缺的一些知识点。并且你还要展示一些在别的书中没有，而这本书会展现这个点的知识，这意味着你需要挖掘得更深。

所以，如果下次有人问你如何学一门新语言、技术，那么答案就是写一本书。

如何应用一门新的技术

对于多数人来说，写书不是一件容易的事，而应用新的技术则是一件迫在眉睫的事。

通常来说，技术出自于对现有的技术的改进。这就意味着，在掌握现有技术的情况下，我们只需要做一些小小的改动就更可以实现技术升级。

而学习一门新的技术的最好实践就是用这门技术对现有的系统行重写。

第一个系统 (v1)：`Spring MVC` + `Bootstrap` + `jQuery`

那么在那个合适的年代里，我们需要单页面应用，就使用了 Backbone。然后，我们就可以用 Mustache + HTML 来替换掉 JSP。

第二个系统 (v2)：`Spring MVC` + `Backbone` + `Mustache`

在这时我们已经实现了前后端分离了，这时候系统实现上变成了这样。

第二个系统 (v2.2)：`RESTful Services` + `Backbone` + `Mustache`

或者

第二个系统 (v2.2)：`RESTful Services` + `AngularJS 1.x`

Spring 只是一个 RESTful 服务，我们还需要一些问题，比如 DOM 的渲染速度太慢了。

第三个系统 (v3)：`RESTful Services` + `React`

系统就是这样一步步演进过来的。

尽管在最后系统的架构已经不是当初的架构，而系统本身的业务逻辑变化并没有发生太大的变化。

特别是对于如博客这一类的系统来说，他的一些技术实现已经趋于稳定，而且是你经常使用的东西。所以，下次试试用新的技术的时候，可以先从对你的博客的重写开始。

Web 编程基础

从浏览器到服务器

如果你的操作系统带有 cURL 这个软件 (在 GNU/Linux、Mac OS 都自带这个工具，Windows 用户可以从http://curl.haxx.se/download.html下载到)，那么我们可以直接用下面的命令来看这看这个过程 (-v 参数可以显示一次 http 通信的整个过程)：

```
curl -v https://www.phodal.com
```

我们就会看到下面的响应过程:

```
* Rebuilt URL to: https://www.phodal.com/
*   Trying 54.69.23.11...
* Connected to www.phodal.com (54.69.23.11) port 443 (#0)
* TLS 1.2 connection using TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
* Server certificate: www.phodal.com
* Server certificate: COMODO RSA Domain Validation Secure Server CA
* Server certificate: COMODO RSA Certification Authority
* Server certificate: AddTrust External CA Root
> GET / HTTP/1.1
> Host: www.phodal.com
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 403 Forbidden
< Server: phodal/0.19.4
< Date: Tue, 13 Oct 2015 05:32:13 GMT
< Content-Type: text/html; charset=utf-8
```

```
< Content-Length: 170
< Connection: keep-alive
<
<html>
<head><title>403 Forbidden</title></head>
<body bgcolor="white">
<center><h1>403 Forbidden</h1></center>
<hr><center>phodal/0.19.4</center>
</body>
</html>
* Connection #0 to host www.phodal.com left intact
```

我们尝试用 cURL 去访问我的网站，会根据访问的域名找出其 IP，通常这个映射关系是来源于 ISP 缓存 DNS（英语：Domain Name System）服务器 [^DNSServer]。

以 “*” 开始的前 8 行是一些连接相关的信息，称为响应首部。我们向域名 https://www.phodal.com/发出了请求，接着 DNS 服务器告诉了我们网站服务器的 IP，即 54.69.23.11。出于安全考虑，在这里我们的示例，我们是以 HTTPS 协议为例，所以在这里连接的端口是 443。因为使用的是 HTTPS 协议，所以在这里会试图去获取服务器证书，接着获取到了域名相关的证书信息。

随后以 “>” 开始的内容，便是向 Web 服务器发送请求。Host 即是我们要访问的主机的域名，GET / 则代表着我们要访问的是根目录，如果我们要访问 https://www.phodal.com/about/页面在这里，便是 GET 资源文件 /about。紧随其后的是 HTTP 的版本号（HTTP/1.1）。User-Agent 通过指向的是使用者行为的软件，通常会加上硬件平台、系统软件、应用软件和用户个人偏好等等的一些信息。Accept 则指的是告知服务器发送何种媒体类型。

这个过程，大致如下图所示：

在图中，我们会发现解析 DNS 的时候，我们需要先本地 DNS 服务器查询。如果没有的话，再向根域名服务器查询——这个域名由哪个服务器来解析。直至最后拿到真正的服务器 IP 才能获取页面。

当我们拿到相应的 HTML、JS、CSS 后，我们就开始渲染这个页面了。

HTTP 协议 说到这里，我们不得不说说 HTTP 协议——超文本传输协议。它也是一个基于文本的传输协议，这就是为什么你在上面看到的都是文本的传输过程。

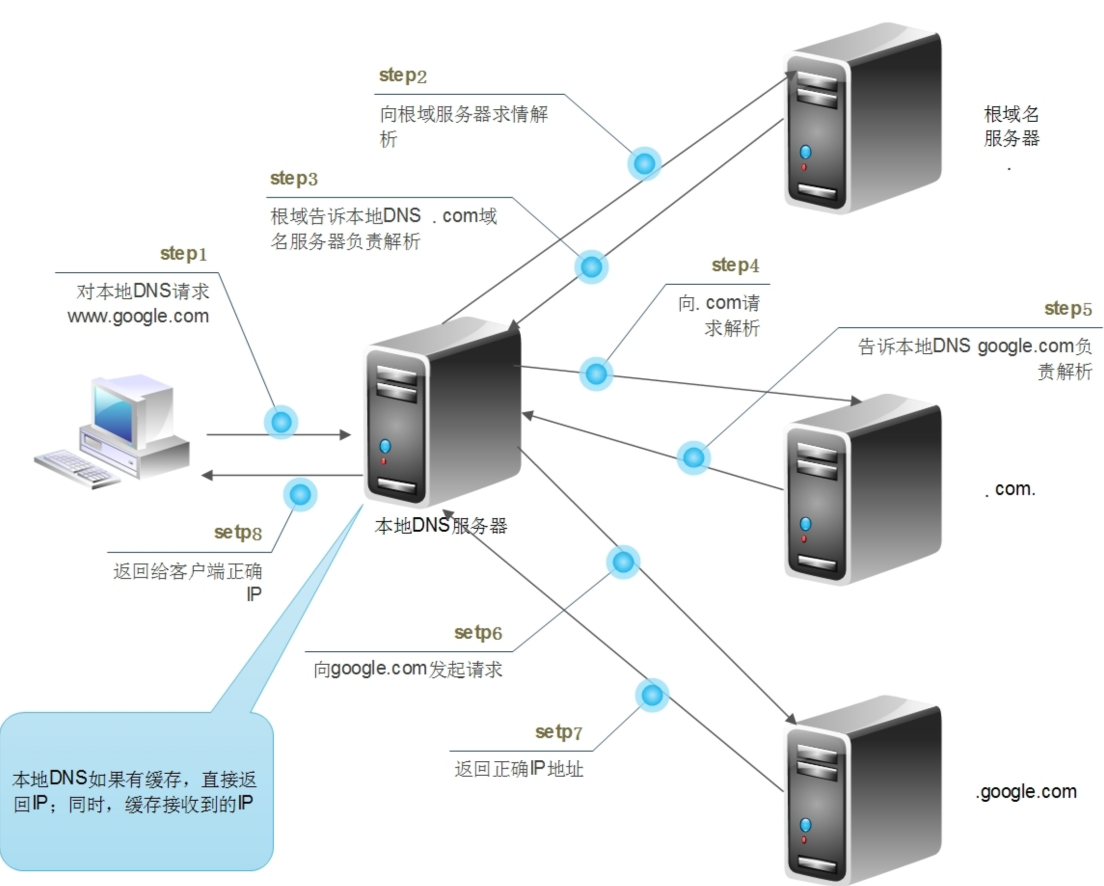


图 26: DNS 到服务器的过程

从 **HTML** 到页面显示

而浏览器接收到文本的时候，就要开始着手将 HTML 变成屏幕。下图是 Chrome 渲染页面的一个时间线：

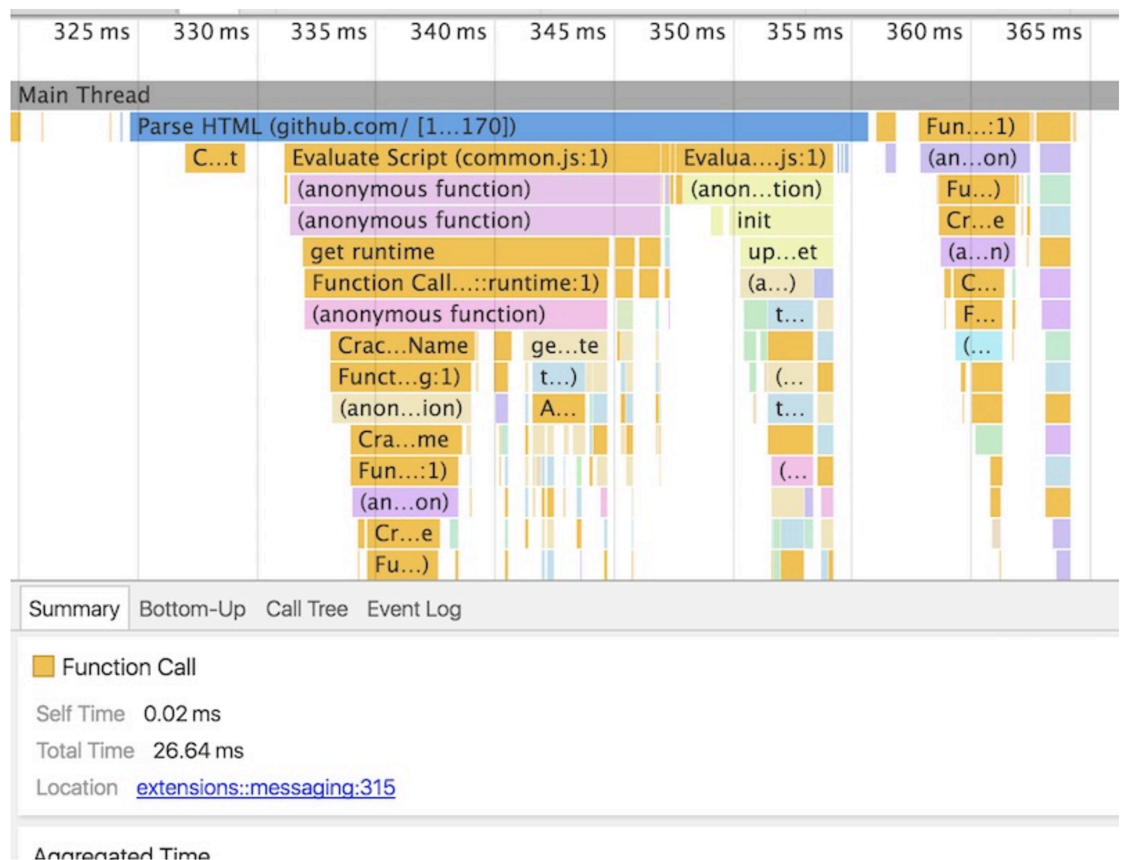


图 27: Chrome 渲染的 Timeline

及其整个渲染过程如下图所示：

（PS: 需要注意的是这里用的是 WebKit 内核的渲染过程，即 Chrome 和 Safari 等浏览器所使用的内核。）

从上面的两图可以看出来第一步都 Parser HTML，而 Paser HTML 实质上就是将其将解析为 DOM Tree。与此同时，CSS 解析器会解析 CSS 会产生 CSS 规则树。

随后会根据生成的 DOM 树和 CSS 规则树来构建 Render Tree，接着生成 Render Tree 的布局，最后就是绘制出 Render Tree。

详细的内容还得参见相关的书籍 ~~。

相关内容：

- 《How browsers work》

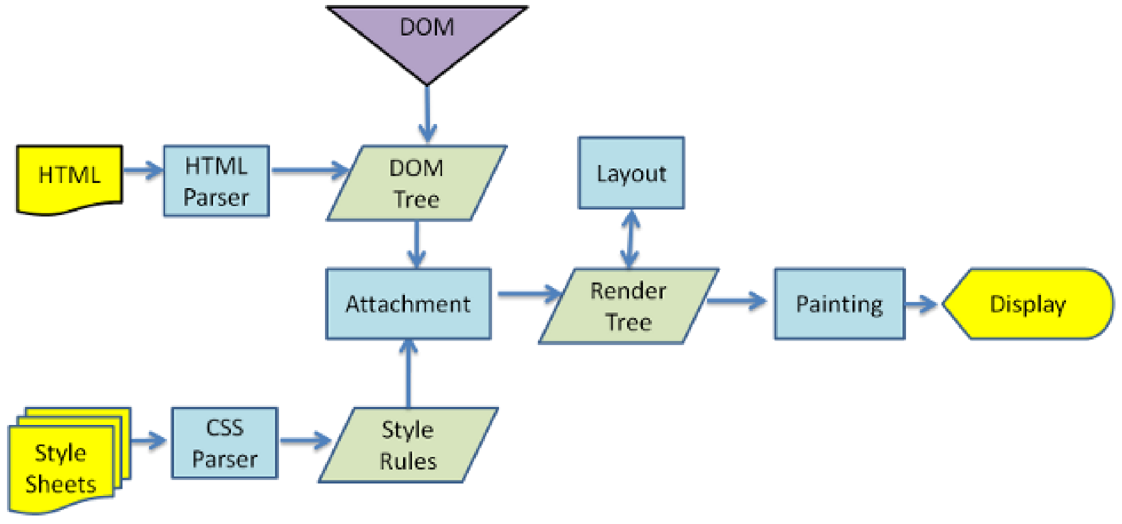


图 28: Render HTML

HTML

让我们先从身边的语言下手，也就是现在无处不在的 HTML+Javascript+CSS。

之所以从 HTML 开始，是因为我们不需要配置一个复杂的开发环境，也许你还不知道开发环境是什么东西，不过这也没关系，毕竟这些知识需要慢慢的接触才能有所了解，尤其是对于普通的业余爱好者来说，当然，对于专业选手言自然不是问题。HTML 是 Web 的核心语言，也算是比较基础的语言。

hello,world

hello,world 是一个传统，所以在这里也遵循这个有趣的传统，我们所要做的事情其实很简单，虽然也有一点点 hack 的感觉。——让我们先来新建一个文并命名为 “helloworld.html”。

(PS: 大部分人应该都是在 Windows 环境下工作的，所以你需要新建一个文本，然后重命名，或者你需要一个编辑器，在这里我们推荐用 **Sublime Text** 。破解不破解，注册不注册都不会对你的使用有太多的影响。)

1. 新建文件
2. 输入
3. 保存为->“helloworld.html”,
4. 双击打开这个文件。正常情况下都应该是用你的默认浏览器打开。只要是一个正常工作的现代浏览器，都应该可以看到上面显示的是 “Hello,world”。

这才是最短的 hello,world 程序，但是呢？在 Ruby 中会是这样子的

```
2.0.0-p353 :001 > p "hello,world"
"hello,world"
    => "hello,world"
2.0.0-p353 :002 >
```

等等，如果你了解过 HTML 的话，会觉得这一点都不符合语法规则，但是他工作了，没有什么比安装完 Nginx 后看到 It works! 更让人激动了。

遗憾的是，它可能无法在所有的浏览器上工作，所以我们需要去调试其中的 bug。

调试 **hello,world** 我们会发现我们的代码在浏览器中变成了下面的代码，如果你和我一样用的是 Chrome，那么你可以右键浏览器中的空白区域，点击审查元素，就会看到下面的代码。

```html
<html>
    <head></head>
    <body>hello,world</body>
</html>
```

这个才是真正能在大部分浏览器上工作的代码，所以复制它到编辑器里吧。

说说 **hello,world** 我很不喜欢其中的 <*></*>，但是我也没有找到别的方法来代替它们，所以这是一个设计得当的语言。甚至大部分人都说这算不上是一门真正的语言，不过 HTML 的原义是

> 超文本标记语言

所以我们可以发现其中的关键词是标记——markup，也就是说 HTML 是一个 markup，head 是一个 markup，body 也是一个 markup。

然而，我们真正工作的代码是在 body 里面，至于为什么是在这里面，这个问题就太复杂了。打个比方来说：

1. 我们所使用的汉语是人类用智慧创造的，我们所正在学的这门语言同样也是人类创造的。
2. 我们在自己的语言里遵循着 桌子是桌子，凳子是凳子的原则，很少有人会问为什么。

中文?

所以我们也可以把计算机语言与现实世界里用于交流沟通的语言划上一个等号。而我们所要学习的语言，并不是我们最熟悉的汉语语言，所以我们便觉得这些很复杂，但是如果我们试着用汉语替换掉上面的代码的话

<语言>
    <头><结束头>
    <身体> 你好，世界<结束身体>
<结束语言>

这看上去很奇怪，只是因为是直译过去的原因，也许你会觉得这样会好理解一点，但是输入上可就一点儿也不方便，因为这键盘本身就不适合我们去输入汉字，同时也意味着可能你输入的会有问题。

让我们把上面的代码代替掉原来的代码然后保存，打开浏览器会看到下面的结果

<语言> <头><结束头> <身体> 你好，世界<结束身体> <结束语言>

更不幸的结果可能是

<璇 □█> <澶 □><缁撴潫澶 □> <韬 □ 綋> 浣犲ソ锛屼笘鐣 □<缁撴潫韬 □ 綋> <缁撴潫璇 □█>

这是一个编码问题，对中文支持不友好。

我们把上面的代码改为和标记语言一样的结构

<语言>
    <头></头>
    <身体> 你好，世界</身体>
<结束语言>

于是我们看到的结果便是

<语言> <头> <身体> 你好，世界

被 Chrome 浏览器解析成什么样了?

```
<html><head></head><body><语言>
        <头><!--头-->
        <身体> 你好，世界<!--身体-->
    <!--语言-->
</body></html>
```

以

结尾的是注释，写给人看的代码，不是给机器看的，所以机器不会去理解这些代码。

但是当我们把代码改成

```
<whatwewanttosay>你好世界</whatwewanttosay>
```

浏览器上面显示的内容就变成了

你好世界

或许你会觉得很神奇，但是这一点儿也不神奇，虽然我们的中文语法也遵循着标记语言的标准，但是我们的浏览器不支持中文标记。

结论:

1. 浏览器对中文支持不友好。
2. 浏览器对英文支持友好。

刚开始的时候不要对中文编程有太多的想法，这是很不现实的:

1. 现有的系统都是基于英语语言环境构建的，对中文支持不是很友好。
2. 中文输入的速度在某种程度上来说没有英语快。

我们离开话题已经很远了，但是这里说的都是针对于那些不满于英语的人来说的，只有当我们可以从头构建一个中文系统的时候才是可行的，而这些就要将 CPU、软件、硬件都包含在内，甚至我们还需要考虑重新设计 CPU 的结构，在某种程度上来说会有些不现实。或许，需要一代又一代人的努力。忘记那些吧，师夷之长技以制夷。

其他 **HTML** 标记

添加一个标题，

```
<html>
    <head>
        <title>标题</title>
    </head>
    <body>hello,world</body>
</html>
```

我们便可以在浏览器的最上方看到“标题”二字，就像我们常用的淘宝网，也包含了上面的东西，只是还包括了更多的东西，所以你也可以看懂那些我们可以看到的淘宝的标题。

```
<html>
<head>
    <title>标题</title>
</head>
<body>
hello,world
<h1>大标题</h1>
<h2>次标题</h2>
<h3>...</h3>
<ul>
    <li>列表 1</li>
    <li>列表 2</li>
</ul>
</body>
</html>
```

更多的东西可以在一些书籍上看到，这边所要说的只是一次简单的语言入门，其他的东西都和这些类似。

小结

美妙之处　我们简单地上手了一门不算是语言的语言，浏览器简化了这其中的大部分过程，虽然没有 C 和其他语言来得有专业感，但是我们试着去开始写代码了。我们可能在未来的某一篇中可能会看到类似的语言，诸如 Python，我们所要做的就是

```
$ python file.py
=>hello,world
```

然后在终端上返回结果。只是因为在我看来学会 HTML 是有意义的，简单的上手，然后再慢慢地深入，如果一开始我们就去理解指针，开始去理解类。我们甚至还知道程序是怎么编译运行的时候，在这个过程中又发生了什么。虽然现在我们也没能理解这其中发生了什么，但是至少展示了

1. 中文编程语言在当前意义不大，不现实，效率不高兼容性差
2. 语言的语法是固定的。（ps: 虽然我们也可以进行扩充，我们将会在后来支持上述的中文标记。）
3. 已经开始写代码，而不是还在配置开发环境。
4. 随身的工具才是最好的，最常用的 code 也才是实在的。

更多　我们还没有试着去解决 “某商店里的糖一颗 5 块钱，小明买了 3 颗糖，小明一共花了多少钱” 的问题。也就是说我们学会的是一个还不能解决实际问题的语言，于是我们还需要学点东西，比如 JavaScript, CSS。我们可以将 JavaScript 理解为解决问题的语言，HTML 则是前端显示，CSS 是配置文件，这样的话，我们会在那之后学会成为一个近乎专业的程序员。我们刚刚学习了一下怎么在前端显示那些代码的行为，于是我们还需要 JavaScript。

CSS

如果说 HTML 是建筑的框架，CSS 就是房子的装修。那么 JavaScript 呢，我听到的最有趣的说法是小三——还是先让我们回到代码上来吧。

下面就是我们之前说到的代码，CSS 将 Red 三个字母变成了红色。

```html
<!DOCTYPE html>
<html>
<head>
</head>
<body>
    <p id="para" style="color:red">Red</p>
</body>
    <script type="text/javascript" src="app.js"></script>
</html>
```

只是，

```
var para=document.getElementById("para");
para.style.color="blue";
```

将字体变成了蓝色，CSS+HTML 让页面有序的工作着，但是 JavaScript 却打乱了这些秩序，有着唯恐世界不乱的精彩，也难怪被冠以小三之名了——或许终于可以理解，为什么以前人们对于 JavaScript 没有好感了——不过这里要讲的是正室，也就是 CSS，这时还没有 JavaScript。

Red

图 29: Red Fonts

简介

这不是一篇专业讲述 CSS 的书籍，所以我不会去说 CSS 是怎么来的，有些东西我们既然可以很容易从其他地方知道，也就不需要花太多时间去重复。诸如重构等这些的目的之一也在于去除重复的代码，不过有些重复是不可少的，也是有必要的，而通常这些东西可能是由其他地方复制过来的。

到目前为止我们没有依赖于任何特殊的硬件或者是软件，对于我们来说我们最基本的需求就是一台电脑，或者可以是你的平板电脑，当然也可以是你的智能手机，因为他们都有个浏览器，而这些都是能用的，对于我们的 CSS 来说也不会有例外的。

CSS(Cascading Style Sheets)，到今天我也没有记得他的全称，CSS 还有一个中文名字是层叠式样式表，事实上翻译成什么可能并不是我们关心的内容，我们需要关心的是他能做些什么。作为三剑客之一，它的主要目的在于可以让我们方便灵活地去控制 Web 页面的外观表现。我们可以用它做出像淘宝一样复杂的界面，也可以像我们的书本一样简单，不过如果要和我们书本一样简单的话，可能不需要用到 CSS。HTML 一开始就是依照报纸的格式而设计的，我们还可以继续用上面说到的编辑器，又或者是其他的。如果你喜欢 DreamWeaver 那也不错，不过一开始使用 IDE 可无助于我们写出良好的代码。

忘说了，CSS 也是有版本的，和 Windows，Linux 内核等等一样，但是更新可能没有那么频繁，HTML 也是有版本的，JS 也是有版本的，复杂的东西不是当前考虑的内容。

代码结构 对于我们的上面的 Red 示例来说，如果没有一个好的结构，那么以后可能就是这样子。

```
<!DOCTYPE html>
<html>
<head>
</head>
<body>
    <p style="font-size: 22px;color:#f00;text-align: center;padding-left: 20p
    <p style="  font-size:44px;color:#3ed;text-indent: 2em;padding-left: 2em;
</body>
</html>
```

虽然我们看到的还是一样的:

如果没有一个好的结构

那么以后可能就是这样子。。。。

图 30: No Style

于是我们就按各种书上的建议重新写了上面的代码

```
<!DOCTYPE html>
<html>
<head>
    <title>CSS example</title>
    <style type="text/css">
        .para{
            font-size: 22px;
            color:#f00;
            text-align: center;
            padding-left: 20px;
        }
        .para2{
```

```
            font-size:44px;
            color:#3ed;
            text-indent: 2em;
            padding-left: 2em;
        }
    </style>
</head>
<body>
    <p class="para">如果没有一个好的结构</p>
    <p class="para2">那么以后可能就是这样子。。。。</p>
</body>
</html>
```

总算比上面好看也好理解多了，这只是临时的用法，当文件太大的时候，正式一点的写法应该如下所示:

```
<!DOCTYPE html>
<html>
<head>
    <title>CSS example</title>
    <style type="text/css" href="style.css"></style>
</head>
<body>
    <p class="para">如果没有一个好的结构</p>
    <p class="para2">那么以后可能就是这样子。。。。</p>
</body>
</html>
```

我们需要

```
<!DOCTYPE html>
<html>
<head>
    <title>CSS example</title>
    <link href="./style.css" rel="stylesheet" type="text/css" />
</head>
<body>
```

```
    <p class="para">如果没有一个好的结构</p>
    <p class="para2">那么以后可能就是这样子。。。。</p>
</body>
</html>
```

然后我们有一个像 app.js 一样的 style.css 放在同目录下，而他的内容便是

```
.para{
    font-size: 22px;
    color:#f00;
    text-align: center;
    padding-left: 20px;
}
.para2{
    font-size:44px;
    color:#3ed;
    text-indent: 2em;
    padding-left: 2em;
}
```

这代码和 JS 的代码有如此多的相似

```
var para={
    font_size:'22px',
    color:'#f00',
    text_align:'center',
    padding_left:'20px',
}
```

而 22px、20px 以及 #f00 都是数值，因此：

```
var para={
    font_size:22px,
    color:#f00,
    text_align:center,
    padding_left:20px,
}
```

目测差距已经尽可能的小了，至于这些话题会在以后讨论到，如果要让我们的编译器更正确的工作，那么我们就需要非常多这样的符号，除非你乐意去理解:

```
(dotimes (i 4) (print i))
```

总的来说我们减少了符号的使用，但是用 lisp 便带入了更多的括号，不过这是一种简洁的表达方式，也许我们可以在其他语言中看到。

```
\d{2}/[A-Z][a-z][a-z]/\d{4}
```

上面的代码，是为了从一堆数据中找出“某日/某月/某年”。如果一开始不理解那是正则表达式，就会觉得那个很复杂。

这门语言可能是为设计师而设计的，但是设计师大部分还是不懂编程的，不过相对来说这门语言还是比其他语言简单易懂一些。

样式与目标

如下所示，就是我们的样式

```css
.para{
    font-size: 22px;
    color:#f00;
    text-align: center;
    padding-left: 20px;
}
```

我们的目标就是

如果没有一个好的结构

所以样式和目标在这里牵手了，问题是他们是如何在一起的呢？下面就是 CSS 与 HTML 沟通的重点所在了:

选择器

我们用到的选择器叫做类选择器，也就是 class，或者说应该称之为 class 选择器更合适。与类选择器最常一起出现的是 ID 选择器，不过这个适用于比较高级的场合，诸如用 JS 控制 DOM 的时候就需要用到 ID 选择器。而基本的选择器就是如下面的例子:

```
p.para{
    color:#f0f;
}
```

将代码添加到 style.css 的最下面会发现“如果没有一个好的结构”变成了粉红色，当然我们还会有这样的写法

```
p>.para{
    color:#f0f;
}
```

为了产生上面的特殊的样式，虽然不好看，但是我们终于理解什么叫层叠样式了，下面的代码的重要度比上面高，也因此有更高的优先规则。

而通常我们可以通过一个

```
p{
    text-align:left;
}
```

这样的元素选择器来给予所有的 p 元素一个左对齐。

还有复杂一点的复合型选择器，下面的是 HTML 文件

```
<!DOCTYPE html>
<html>
<head>
    <title>CSS example</title>
    <link href="./style.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <p class="para">如果没有一个好的结构</p>
    <div id="content">
        <p class="para2">那么以后可能就是这样子。。。。</p>
    </div>
</body>
</html>
```

还有 CSS 文件

```
.para{
    font-size: 22px;
    color:#f00;
    text-align: center;
    padding-left: 20px;
}
.para2{
    font-size:44px;
    color:#3ed;
    text-indent: 2em;
    padding-left: 2em;
}

p.para{
    color:#f0f;
}
div#content p {
    font-size:22px;
}
```

更有趣的 **CSS**

一个包含了 para2 以及 para_bg 的例子

```
    <div id="content">
        <p class="para2 para_bg">那么以后可能就是这样子。。。。</
p>
    </div>
```

我们只是添加了一个黑色的背景

```
.para_bg{
    background-color:#000;
}
```

重新改变后的网页变得比原来有趣了很多，所谓的继承与合并就是上面的例子。

我们还可以用 CSS3 做出更多有趣的效果，而这些并不在我们的讨论范围里面，因为我们讨论的是 be a geek。

或许我们写的代码都是那么的简单，从 HTML 到 JavaScript，还有现在的 CSS，只是总有一些核心的东西，而不是去考虑那些基础语法，基础的东西我们可以在实践的过程中一一发现。但是我们可能发现不了，或者在平时的使用中考虑不到一些有趣的用法或者说特殊的用法，这时候可以通过观察一些精致设计的代码中学习到。复杂的东西可以变得很简单，简单的东西也可以变得很复杂。

JavaScript

JavaScript 现在已经无处不在了，也许你正打开的某个网站，他便可能是 node.js +json+javascript+mustache.js 完成的，虽然你还没理解上面那些是什么，也正是因为你不理解才需要去学习更多的东西。但是你只要知道 JavaScript 已经无处不在了，它可能就在你手机上的某个 app 里，就在你浏览的网页里，就运行在你 IDE 中的某个进程里。

hello,world

这里我们还需要有一个 helloworld.html，JavaScript 是专为网页交互而设计的脚本语言，所以我们一点点来开始这部分的旅途，先写一个符合标准的 helloworld.html

```
<!DOCTYPE html>
<html>
    <head></head>
    <body></body>
</html>
```

然后开始融入我们的 JavaScript，向 HTML 中插入 JavaScript 的方法，就需要用到 HTML 中的 <script> 标签，我们先用页面嵌入的方法来写 helloworld。

```
<!DOCTYPE html>
<html>
    <head>
        <script>
            document.write('hello,world');
        </script>
    </head>
```

```
    <body></body>
</html>
```

按照标准的写法，我们还需要声明这个脚本的类型

```
<!DOCTYPE html>
<html>
    <head>
        <script type="text/javascript">
            document.write('hello,world');
        </script>
    </head>
    <body></body>
</html>
```

没有显示 hello,world？试试下面的代码

```
<!DOCTYPE html>
<html>
    <head>
        <script type="text/javascript">
            document.write('hello,world');
        </script>
    </head>
    <body>
        <noscript>
            disable Javascript
        </noscript>
    </body>
</html>
```

JavaScriptFul

我们需要让我们的代码看上去更像是 js，同时是以 js 结尾。就像 C 语言的源码是以 C 结尾的，我们也同样需要让我们的代码看上去更正式一点。于是我们需要在 helloworld.html 的同一文件夹下创建一个 app.js 文件，在里面写着

```
document.write('hello,world');
```

同时我们的 helloworld.html 还需要告诉我们的浏览器 js 代码在哪里

```
<!DOCTYPE html>
<html>
    <head>
        <script type="text/javascript" src="app.js"></script>
    </head>
    <body>
        <noscript>
            disable Javascript
        </noscript>
    </body>
</html>
```

从数学出发　让我们回到第一章讲述的小明的问题，从实际问题下手编程，更容易学会编程。小学时代的数学题最喜欢这样子了——某商店里的糖一个 5 块钱，小明买了 3 个糖，小明一共花了多少钱。在编程方面，也许我们还算是小学生。最直接的方法就是直接计算 3x5=?

```
document.write(3*5);
```

document.write 实际也我们可以理解为输出，也就是往页面里写入 3*5 的结果，在有双引号的情况下会输出字符串。我们便会在浏览器上看到 15，这便是一个好的开始，也是一个糟糕的开始。

设计和编程　对于实际问题，如果我们只是止于所要得到的结果，很多年之后，我们就成为了 code monkey。对这个问题进行再一次设计，所谓的设计有些时候会把简单的问题复杂化，有些时候会使以后的扩展更加简单。这一天因为这家商店的糖价格太高了，于是店长将价格降为了 4 块钱。

```
document.write(3*4);
```

于是我们又得到了我们的结果，但是下次我们看到这些代码的时候没有分清楚哪个是糖的数量，哪个是价格，于是我们重新设计了程序

```
tang=4;
num=3;
document.write(tang*num);
```

这才能叫得上是程序设计，或许你注意到了 “;” 这个符号的存在，我想说的是这是另外一个标准，我们不得不去遵守，也不得不去 fuck。

函数　记得刚开始学三角函数的时候，我们会写

```
sin 30=0.5
```

而我们的函数也是类似于此，换句话说，因为很多搞计算机的先驱都学好了数学，都把数学世界的规律带到了计算机世界，所以我们的函数也是类似于此，让我们从一个简单的开始。

```
function hello(){
    return document.write("hello,world");
}
hello();
```

当我第一次看到函数的时候，有些小激动终于出现了。我们写了一个叫 hello 的函数，它返回了往页面中写入 hello,world 的方法，然后我们调用了 hello 这个函数，于是页面上有了 hello,world。

```
function sin(degree){
    return document.write(Math.sin(degree));
}
sin(30);
```

在这里 degree 就称之为变量。于是输出了 -0.9880316240928602，而不是 0.5，因为这里用的是弧度制，而不是角度制。

```
sin(30)
```

的输出结果有点类似于 sin 30。写括号的目的在于，括号是为了方便解析，这个在不同的语言中可能是不一样的，比如在 Ruby 中我们可以直接用类似于数学中的表达:

```
2.0.0-p353 :004 > Math.sin 30
=> -0.9880316240928618
2.0.0-p353 :005 >
```

我们可以在函数中传入多个变量，于是我们再回到小明的问题，就会这样去编写代码。

```
function calc(tang,num){
    result=tang*num;
    document.write(result);
}
calc(3,4);
```

但是从某种程度上来说，我们的 calc 做了计算的事又做了输出的事，总的来说设计上有些不好。

重新设计　我们将输出的工作移到函数的外面，

```
function calc(tang,num){
    return tang*num;
}
document.write(calc(3,4));
```

接着我们用一种更有意思的方法来写这个问题的解决方案

```
function calc(tang,num){
    return tang*num;
}
function printResult(tang,num){
    document.write(calc(tang,num));
}
printResult(3, 4)
```

看上去更专业了一点点，如果我们只需要计算的时候我们只需要调用 calc，如果我们需要输出的时候我们就调用 printResult 的方法。

object 和函数　我们还没有说清楚之前我们遇到过的 document.write 以及 Math.sin 的语法为什么看上去很奇怪，所以让我们看看他们到底是什么，修改 app.js 为以下内容

```
document.write(typeof document);
document.write(typeof Math);
```

typeof document 会返回 document 的数据类型，就会发现输出的结果是

```
object object
```

所以我们需要去弄清楚什么是 object。对象的定义是

> 无序属性的集合，其属性可以包含基本值、对象或者函数。

创建一个 object，然后观察这便是我们接下来要做的

```
store={};
store.tang=4;
store.num=3;
document.write(store.tang*store.num);
```

我们就有了和 document.write 一样的用法，这也是对象的美妙之处，只是这里的对象只是包含着基本值，因为

```
typeof story.tang="number"
```

一个包含对象的对象应该是这样子的。

```
store={};
store.tang=4;
store.num=3;
document.writeln(store.tang*store.num);

var wall=new Object();
wall.store=store;
document.write(typeof wall.store);
```

而我们用到的 document.write 和上面用到的 document.writeln 都是属于这个无序属性集合中的函数。

下面代码说的就是这个无序属性集合中的函数。

```
var IO=new Object();
function print(result){
    document.write(result);
};
IO.print=print;
IO.print("a obejct with function");
IO.print(typeof IO.print);
```

我们定义了一个叫 IO 的对象，声明对象可以用

```
var store={};
```

又或者是

```
var store=new Object{};
```

两者是等价的，但是用后者的可读性会更好一点，我们定义了一个叫 print 的函数，他的作用也就是 document.write，IO 中的 print 函数是等价于 print() 函数，这也就是对象和函数之间的一些区别，对象可以包含函数，对象是无序属性的集合，其属性可以包含基本值、对象或者函数。

复杂一点的对象应该是下面这样的一种情况。

```
var Person={name:"phodal",weight:50,height:166};
function dream(){
    future;
};
Person.future=dream;
document.write(typeof Person);
document.write(Person.future);
```

而这些会在我们未来的实际编程过程中用得更多。

面向对象

开始之前先让我们简化上面的代码，

```
Person.future=function dream(){
    future;
}
```

看上去比上面的简单多了，不过我们还可以简化为下面的代码。。。

```
var Person=function(){
    this.name="phodal";
    this.weight=50;
    this.height=166;
    this.future=function dream(){
        return "future";
    };
};
var person=new Person();
document.write(person.name+"<br>");
document.write(typeof person+"<br>");
document.write(typeof person.future+"<br>");
document.write(person.future()+"<br>");
```

只是在这个时候 Person 是一个函数，但是我们声明的 person 却变成了一个对象 一个 **Javascript** 函数也是一个对象，并且，所有的对象从技术上讲也只不过是函数。 这里的 “<br>” 是 HTML 中的元素，称之为 DOM，在这里起的是换行的作用，我们会在稍后介绍它，这里我们先关心下 this。this 关键字表示函数的所有者或作用域，也就是这里的 Person。

上面的方法显得有点不可取，换句话说和一开始的

```
document.write(3*4);
```

一样，不具有灵活性，因此在我们完成功能之后，我们需要对其进行优化，这就是程序设计的真谛——解决完实际问题后，我们需要开始真正的设计，而不是解决问题时的编程。

```
var Person=function(name,weight,height){
    this.name=name;
    this.weight=weight;
    this.height=height;
    this.future=function(){
        return "future";
    };
};
var phodal=new Person("phodal",50,166);
document.write(phodal.name+"<br>");
document.write(phodal.weight+"<br>");
document.write(phodal.height+"<br>");
document.write(phodal.future()+"<br>");
```

于是，产生了这样一个可重用的 JavaScript 对象, this 关键字确立了属性的所有者。

其他

JavaScript 还有一个很强大的特性，也就是原型继承，不过这里我们先不考虑这些部分，用尽量少的代码及关键字来实际我们所要表达的核心功能，这才是这里的核心，其他的东西我们可以从其他书本上学到。

所谓的继承，

```
var Chinese=function(){
    this.country="China";
}

var Person=function(name,weight,height){
    this.name=name;
    this.weight=weight;
    this.height=height;
    this.futrue=function(){
        return "future";
    }
}
Chinese.prototype=new Person();
```

```
var phodal=new Chinese("phodal",50,166);
document.write(phodal.country);
```

完整的 JavaScript 应该由下列三个部分组成:

- 核心 (ECMAScript)——核心语言功能
- 文档对象模型 (DOM)——访问和操作网页内容的方法和接口
- 浏览器对象模型 (BOM)——与浏览器交互的方法和接口

我们在上面讲的都是 ECMAScript，也就是语法相关的，但是 JS 真正强大的，或者说我们最需要的可能就是对 DOM 的操作，这也就是为什么 jQuery 等库可以流行的原因之一，而核心语言功能才是真正在哪里都适用的，至于 BOM，真正用到的机会很少，因为没有完善的统一的标准。

一个简单的 DOM 示例,

```
<!DOCTYPE html>
<html>
<head>
</head>
<body>
    <noscript>
        disable Javascript
    </noscript>
    <p id="para" style="color:red">Red</p>
</body>
    <script type="text/javascript" src="app.js"></script>
</html>
```

我们需要修改一下 helloworld.html 添加

```
<p id="para" style="color:red">Red</p>
```

同时还需要将 script 标签移到 body 下面，如果没有意外的话我们会看到页面上用红色的字体显示 Red，修改 app.js。

```
var para=document.getElementById("para");
para.style.color="blue";
```

接着，字体就变成了蓝色，有了 DOM 我们就可以对页面进行操作，可以说我们看到的绝大部分的页面效果都是通过 DOM 操作实现的。

美妙之处　这里说到的 JavaScript 仅仅只是其中的一小小部分，忽略掉的东西很多，只关心的是如何去设计一个实用的 app，作为一门编程语言，他还有其他强大的内制函数，要学好需要一本有价值的参考书。这里提到的只是其中的不到 20% 的东西，其他的 80% 或者更多会在你解决问题的时候出现。

- 我们可以创建一个对象或者函数，它可以包含基本值、对象或者函数。
- 我们可以用 JavaScript 修改页面的属性，虽然只是简单的示例。
- 我们还可以去解决实际的编程问题。

前端与后台

> 前端 Front-end 和后端 Back-end 是描述进程开始和结束的通用词汇。前端作用于采集输入信息，后端进行处理。

这种说法给人一种很模糊的感觉，但是他说得又很对，它负责视觉展示。在 MVC 结构或者 MVP 中，负责视觉显示的部分只有 View 层，而今天大多数所谓的 View 层已经超越了 View 层。前端是一个很神奇的概念，但是而今的前端已经发生了很大的变化。你引入了 Backbone、Angluar，你的架构变成了 MVP、MVVM。尽管发生了一些架构上的变化，但是项目的开发并没有因此而发生变化。这其中涉及到了一些职责的问题，如果某一个层级中有太多的职责，那么它是不是加重了一些人的负担?

后台在过去的岁月里起着很重要的作用，当然在未来也是。就最几年的解耦趋势来看，它在变得更小，变成一系列的服务。并向前台提供很多 RESTful API，看上去有点像提供一些辅助性的工作。

因此在这一章里，我们将讲述详细介绍:

1. 后台语言与选型
2. 前端框架与选型
3. 前端一致化，后台服务化的趋势
4. 前后端通讯

后台语言选择

如何选择一门好的后台语言似乎是大家都很感兴趣的问题？大概只是因为他们想要在一开始的时候去学一门很实用的语言——至少会经常用到，而不是学好就被遗弃了。或者它不会因为一门新的语言的出现而消亡。

JavaScript

在现在看来，JavaScript 似乎是一个性价比非常高的语言。只要是 Web 就会有前端，只要有前端就需要有 JavaScript。与此同时 Node.js 在后台中的地位已经愈发重要了。

对于 JavaScript 来说，它可以做很多类型的应用。这些应用都是基于浏览器来运行的，有：

- Electron + Node.js + JavaScript 做桌面应用
- Ionic + JavaScript 做移动应用
- Node.js + JavaScript 网站前后台
- JavaScript + Tessl 做硬件

So，这是一门很有应用前景的语言。

Python

Python 诞生得比较早，其语言特性——做事情只有一件方法，也决定了这门语言很简单。在 ThoughtWorks University 的学习过程中，接触了一些外国小伙伴，这是大多数人学习的第一门语言。

Python 在我看来和 JavaScript 是相当划算的语言，除了它不能在前端运行，带来了一点劣势。Python 是一门简洁的语言，而且有大量的数学、科学工具，这意味着在不远的将来它会发挥更大的作用。我喜欢在我的各种小项目上用 Python，如果不是因为我对前端及数据可视化更感兴趣，那么 Python 就是我的第一语言了。

Java

除此呢，我相信 Java 在目前来说也是一个不错的选择。

在学校的时候，一点儿也不喜欢 Java。后来才发现，我从 Java 上学到的东西比其他语言上学得还多。如果 Oracle 不毁坏 Java，那么他会继续存活很久。我可以用

JavaScript 造出各种我想要的东西，但是通常我无法保证他们是优雅的实现。过去人们在 Java 上花费了很多的时间，或在架构上，或在语言上，或在模式上。由于这些投入，都给了人们很多的启发。这些都可以用于新的语言，新的设计，毕竟没有什么技术是独立于旧的技术产生出来的。

PHP

PHP 呢，据说是这个『世界上最好的语言』，我服务器上运行着几个不同的 WordPress 实例。对于这门语言，我还是相当放心的。并且这门语言由于上手简单，同时国内有大量的程序员已经掌握好了这门语言。不得不提及的是 WordPress 已经占领了 CMS 市场超过一半的份额，并且它也占领了全球网站的四分之一。还有 Facebook，这个世界上最大的 PHP 站点也在使用这门语言。

其他

个人感觉 Go 也不错，虽然没怎么用，但是性能应该是相当可以的。

Ruby、Scala，对于写代码的人来说，这是非常不错的语言。但是如果是团队合作时，就有待商榷。

MVC

人们在不断地反思这其中复杂的过程，整理了一些好的架构模式，其中不得不提到的是我司 Martin Fowler 的《企业应用架构模式》。该书中文译版出版的时候是 2004 年，那时对于系统的分层是

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通讯。

化身于当时最流行的 Spring，就是 MVC。人们有了 iBatis 这样的数据持久层框架，即 ORM，对象关系映射。于是，你的 package 就会有这样的几个文件夹：

```
|____mappers
|____model
|____service
```

```
|____utils
|____controller
```

在 mappers 这一层，我们所做的莫过于如下所示的数据库相关查询：

```
@Insert(
        "INSERT INTO users(username, password, enabled) " +
                "VALUES (#{userName}, #{passwordHash}, #{enabled})"
)
@Options(keyProperty = "id", keyColumn = "id", useGeneratedKeys = true)
void insert(User user);
```

model 文件夹和 mappers 文件夹都是数据层的一部分，只是两者间的职责不同，如：

```
public String getUserName() {
    return userName;
}

public void setUserName(String userName) {
    this.userName = userName;
}
```

而他们最后都需要在 Controller，又或者称为 ModelAndView 中处理：

```
@RequestMapping(value = {"/disableUser"}, method = RequestMethod.POST)
public ModelAndView processUserDisable(HttpServletRequest request, ModelMap m
    String userName = request.getParameter("userName");
    User user = userService.getByUsername(userName);
    userService.disable(user);
    Map<String,User> map = new HashMap<String,User>();
    Map <User,String> usersWithRoles= userService.getAllUsersWithRole();
    model.put("usersWithRoles",usersWithRoles);
    return new ModelAndView("redirect:users",map);
}
```

在多数时候，Controller 不应该直接与数据层的一部分，而将业务逻辑放在 Controller 层又是一种错误，这时就有了 Service 层，如下图：

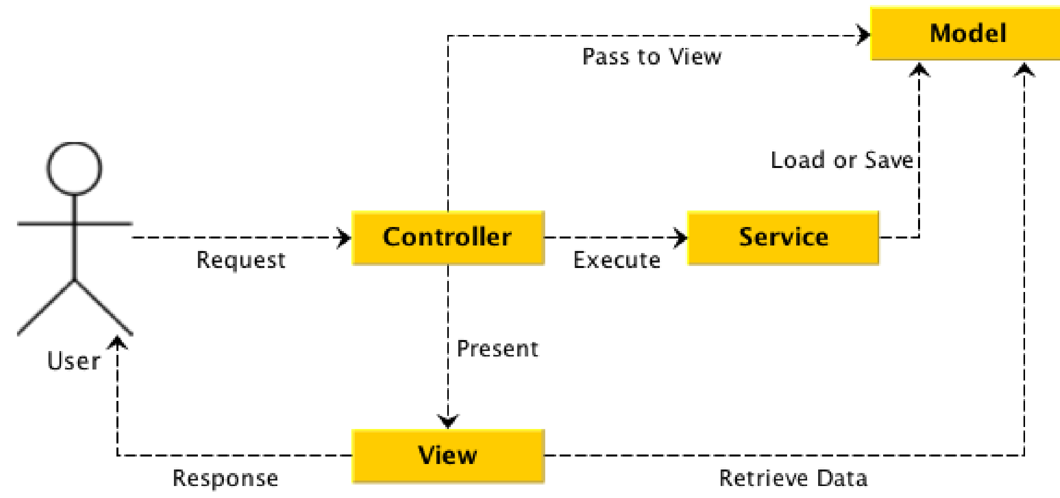


图 31: Service MVC

Domain（业务）是一个相当复杂的层级，这里是业务的核心。一个合理的 Controller 只应该做自己应该做的事，它不应该处理业务相关的代码：

我们在 Controller 层应该做的事是：

1. 处理请求的参数
2. 渲染和重定向
3. 选择 Model 和 Service
4. 处理 Session 和 Cookies

业务是善变的，昨天我们可能还在和对手竞争谁先推出新功能，但是今天可能已经合并了。我们很难预见业务变化，但是我们应该能预见 Controller 是不容易变化的。在一些设计里面，这种模式就是 Command 模式。

Model

模型用于封装与应用程序的业务逻辑相关的数据以及对数据的处理方法。

它是介于数据与控制器之间的层级，拥有对数据直接访问的权力——增删改查(CRUD)。Web 应用中，数据通常是由数据库来存储，有时也会用搜索引擎来存储

因此在实现这个层级与数据库交付时，可以使用 SQL 语句，也可以使用 ORM 框架。

SQL(Structured Query Language，即结构化查询语言)，语句是数据库的查询语言

ORM(Object Relational Mapping)，即对象关系映射，主要是将数据库中的关系数据映射称为程序中的对象。

View

View 层在 Web 应用中，一般是使用模板引擎装载对应 HTML。如下所示的是一段 JSP 代码:

```
<html>
<head><title>First JSP</title></head>
<body>
  <%
    double num = Math.random();
    if (num > 0.95) {
  %>
      <h2>You'll have a luck day!</h2><p>(<%= num %>)</p>
  <%
    } else {
  %>
      <h2>Well, life goes on ... </h2><p>(<%= num %>)</p>
  <%
    }
  %>
  <a href="<%= request.getRequestURI() %>"><h3>Try Again</h3></a>
</body>
</html>
```

上面的 JSP 代码在经过程序解析、处理后，会变成相对应的 HTML。而我们可以发现在这里的 View 层不仅仅只有模板的作用，我们会发现这里的 View 层还计划了部分的逻辑。我们可以在后面细细看这些问题，对于前端的 View 层来说，他可能是这样的:

```
<div class="information pure-g">
    {{#.}}
    <div class="pure-u-1 ">
        <div class="l-box">
            <h3 class="information-head"><a href="#/blog/{{slug}}" alt="{{tit
            <p>
```

```
                发布时间:<span>{{created}}</span>
            <p>
                {{{content}}}
            </p>

            </p>
        </div>
    </div>
    {{/.}}
</div>
```

在这里的 View 层只是单纯的一个显示作用，这也是我们推荐的做法。业务逻辑应该尽可能的放置于业务层。

Controller

控制器层起到不同层面间的组织作用，用于控制应用程序的流程。

更多

在前后端解耦合的系统中，通常系统的架构模式就变成了 MVP，又或者是 MVVM。

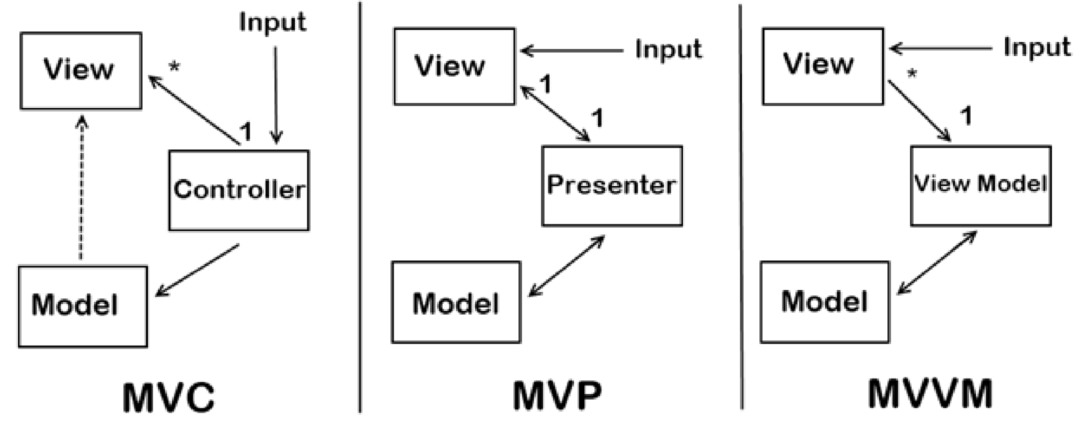


图 32: MVC、MVVM、MVP 对比

三者间很大的不同在于层级间的通讯模型、使用场景。

MVP

> MVP 是从经典的模式 MVC 演变而来，它们的基本思想有相通的地方：Controller/Presenter 负责逻辑的处理，Model 提供数据，View 负责显示。

MVVM MVVM 是 Model-View-ViewModel 的简写。相比于 MVC 悠久的历史来说，MVVM 是一个相当新的架构，它最早于 2005 年被由的 WPF 和 Silverlight 的架构师 John Gossman 提出，并且应用在微软的软件开发中。而 MVC 已经被提出了二十多年了，可见两者出现的年代差别有多大。

MVVM 在使用当中，通常还会利用双向绑定技术，使得 Model 变化时，ViewModel 会自动更新，而 ViewModel 变化时，View 也会自动变化。所以，MVVM 模式有些时候又被称作：model-view-binder 模式。

后台即服务

> BaaS（Backend as a Service）是一种新型的云服务，旨在为移动和 Web 应用提供后端云服务，包括云端数据/文件存储、账户管理、消息推送、社交媒体整合等。

产生这种服务的主要原因之一是因为移动应用的流行。在移动应用中，我们实际上只需要一个 API 接口来连接数据库，并作一些相应的业务逻辑处理。对于不同的应用产商来说，他们打造 API 的方式可能稍有不同，然而他们都只是将后台作为一个服务。

在一些更特殊的例子里，即有网页版和移动应用端，他们也开始使用同一个 API。前端作为一个单页面的应用，或者有后台渲染的应用。其架构如下图所示：

API 演进史

在早期的工作中，我们会发现我们会将大量的业务逻辑放置到 View 层——如迭代出某个结果。

而在今天，当我们有大量的逻辑一致时，我们怎么办，重复实现三次?

如下所示是笔者之前重构的系统的一个架构缩略图：

上面系统产生的主要原因是：技术本身的演进所造成的，并非是系统在一开始没有考虑到这个问题。

从早期到现在的互联网公司都有这样的问题，也会有同样的过程：

第一阶段：因为创始人对于某个领域的看好，他们就创建了这样的一个桌面网站。这个时间点，大概可以在 2000 年左右。

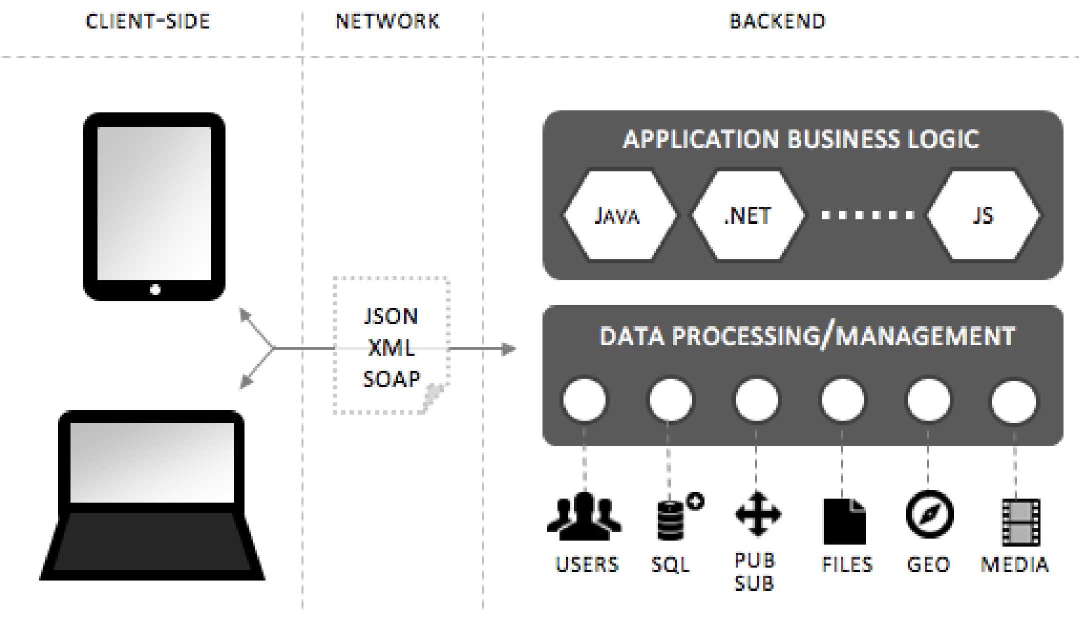


图 33: Backend As A Service

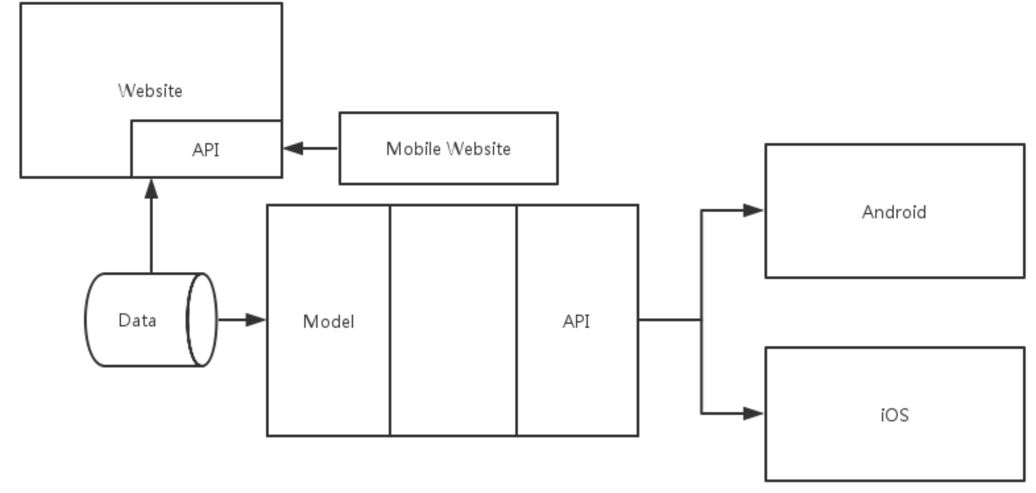


图 34: 重复逻辑的系统架构

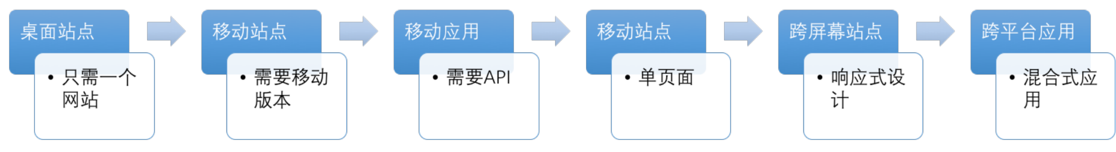


图 35: API 演进史

第二阶段：前“智能手机”出现了，人们需要开发移动版本的网站来适用用户的需要。这时由于当时的开发环境，以及技术条件所限，当时的网站只会是桌面模板的简化。这时还没有普及 Ajax 请求、SPA 这些事物。

第三阶段：手机应用的制作开始流行起来了。由于需要制作手机应用，人们就需要在网站上创建 API。由于当时的业务或者项目需求，这个 API 是直接耦合在系统中的。

第四阶段：由于手机性能的不断提高，并且移动网络速度不断提升，人们便开始在手机上制作单页面应用。

由于他们使用的是相同业务逻辑、代码逻辑相同而技术栈不同的代码，当有一个新的需求出现时，他们需要重复多次实现，如下图所示：

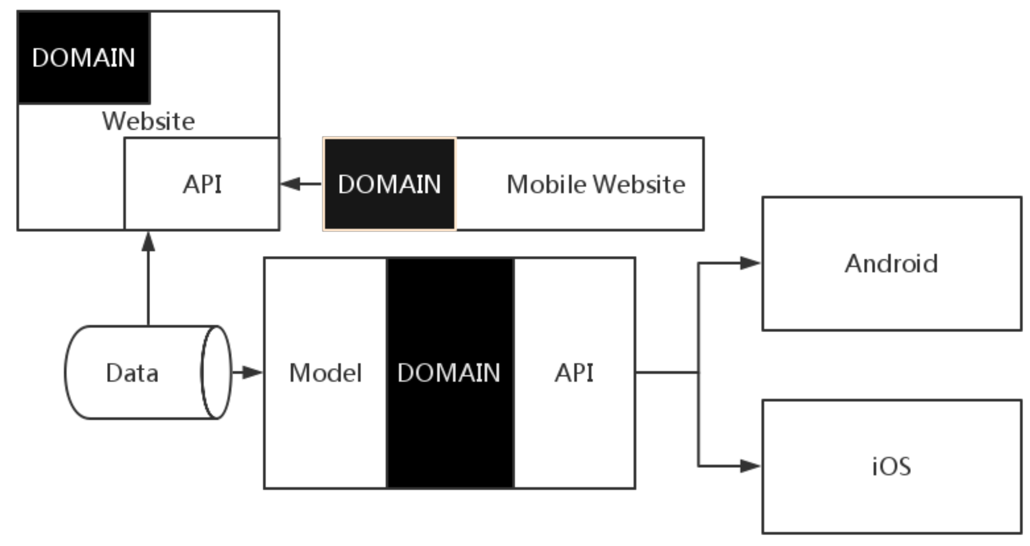


图 36: 重复业务逻辑的系统架构

随后——也就是今天，各种新的解决方案出现了，如 React、混合应用、原生 + Web 的混合式应用、他们的目的就是解决上述的问题。不过，这些解决方案只是为了解决在前端中可能出现的问题，详细的内容可以见《前端演进史》。

而人们也借此机会在统一后台——因为我们可以借助于混合应用或混合式应用（即原生 + 内嵌 WebView，可以同时解决性能和跨平台问题）统一移动端，借助于响应式设计的理念可以统一桌面、平板和手机端。

因此，我们需要的就只是这样的一个 API：

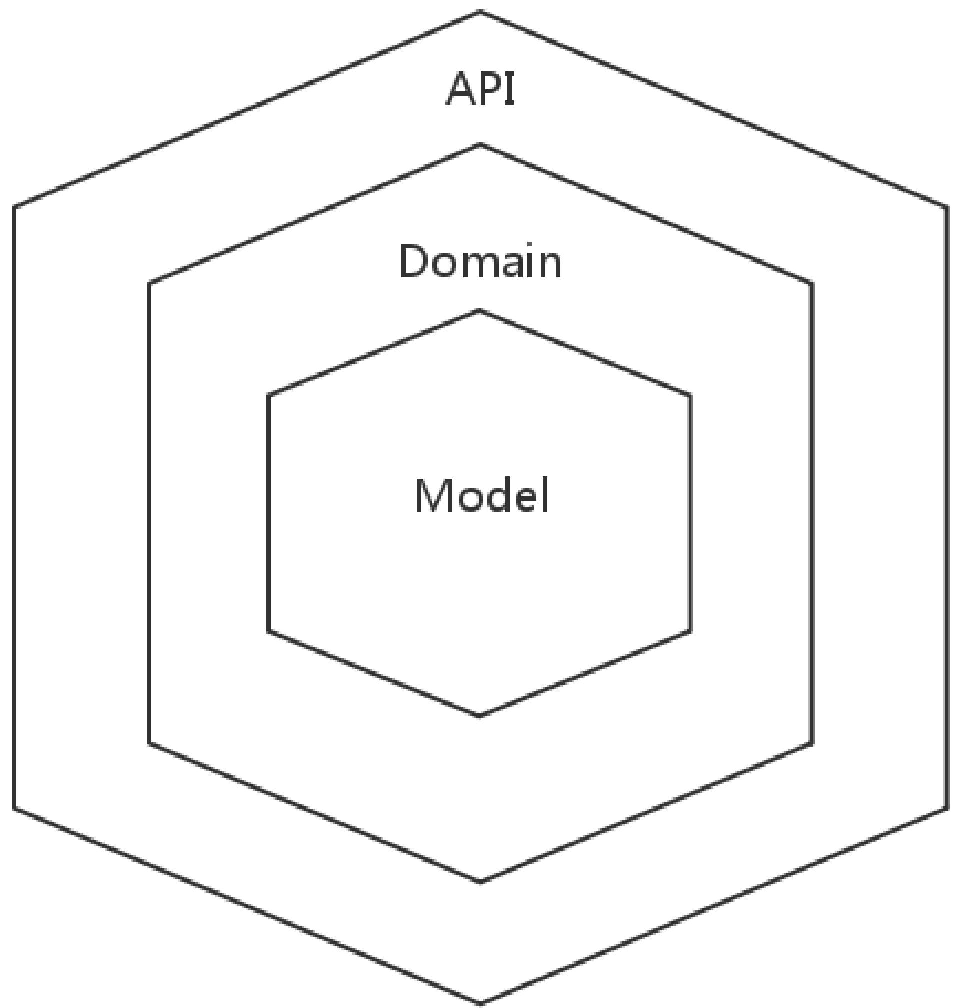


图 37: One API

后台即服务

现在，让我们来看看一个采用后台即服务的网站架构会是怎样的?

数据持久化

信息源于数据，我们在网站上看到的内容都应该是属于信息的范畴。这些信息是应用从数据库中根据业务需求查找、过滤出来的数据。

数据通常以文件的形式存储，毕竟文件是存储信息的基本单位。只是由于业务本身对于 Create、Update、Query、Index 等有不同的组合需求就引发了不同的数据存储软件。

如上章所说，View 层直接从 Model 层取数据，无遗也会暴露数据的模型。作为一个前端开发人员，我们对数据的操作有三种类型:

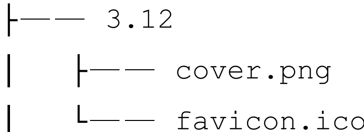
1. 数据库。由于 Node.js 在最近几年里发展迅猛，越来越多的开发者选择使用 Node.js 作为后台语言。这与传统的 Model 层并无多大不同，要么直接操作数据库，要么间接操作数据库。即使在 NoSQL 数据库中也是如此。
2. 搜索引擎。对于以查询为主的领域来说，搜索引擎是一个更好的选择，而搜索引擎又不好直接向 View 层暴露接口。这和招聘信息一样，都在暴露公司的技术栈。
3. RESTful。RESTful 相当于是 CRUD 的衍生，只是传输介质变了。
4. LocalStorage。LocalStorage 算是另外一种方式的 CRUD。

说了这么多都是废话，他们都是可以用类 CRUD 的方式操作。

文件存储

通常来说，以这种方式存储最常见的方式是 log(日志)，如 Nginx 的 access.log。像这样的文件就需要一些专业的软件，如 GoAccess、又或者是 Hadoop、Spark 来做对应的事。

在数据库出现之前，人们都是使用文件来存储数据的。数据以文件为单位存储在硬盘上，并且这些文件不容易一起管理、修改等等。如下图所示的是我早期存储文件的一种方式:



```
└── 3.13
    └── template.tex
```

每天我们都会修改、查看大量的不同类型的文件。而由于工作繁忙，我们可能没有办法一一地去分类这些文件。有时选择的便是，优先先按日期把文件一划分，接着再在随后的日子里归档。而这种存储方式大量的依赖于人来索引的工作，在很多时候往往显得不是很靠谱。并且当我们将数据存储进去后，往往很难进行修改。大量的 Log 文件就需要专门的工作来分析和使用，依赖于人来解析这些日志往往显得不是很靠谱。这时我们就需要一些重量级的工具，如用 Logstash、ElasticSearch、Kibana 来处理 Nginx 访问日志。

而对于那些非专业人员来说，使用 Excel 这样的工具往往显得比较方便。他们不需要去操作数据库，也不需要专业的知识来处理这些知识。只是从某种意义上来说，Excel 应该归属于数据库的范畴。

数据库

当我们开始一个 Web 应用的时候，如创建一个用户管理系统的时候，我们就需要不断由于经常对文件进行查询、修改、插入和删除等操作。不仅仅如此，我们还需要定义数据之前的关系，如这个用户对应这个密码。在一些更复杂的情况下，我们还需要寻找中这些用户对应的一些操作数据等等。如果我们还是这些工作交给文件来处理，那么我们便是在向自己挖坑。

> 数据库，简单来说可视为电子化的文件柜——存储电子文件的处所，用户可以对文件中的数据运行新增、截取、更新、删除等操作。

在操作库的时候，我们会使用到一名为 SQL（英语：Structural Query Language，中文：结构化查询语言）的领域特定语言来对数据进行操作。

> SQL 是高级的非过程化编程语言，它允许用户在高层数据结构上工作。它不要求用户指定对数据的存放方法，也不需要用户了解其具体的数据存放方式。

数据库里存储着大量的数据，在我们对系统建模的时候，也在决定系统的基础模型。

ORM 在传统 SQL 数据库中，我们可能会依赖于 ORM，也可能会自己写 SQL。在使用 ORM 框架时，我们需要先定义 Model，如下是 Node.js 的 ORM 框架 Sequelize 的一个示例：

```
var User = sequelize.define('user', {
  firstName: {
    type: Sequelize.STRING,
    field: 'first_name'
  },
  lastName: {
    type: Sequelize.STRING
  }
}, {
  freezeTableName: true
});

User.sync({force: true}).then(function () {
  // Table created
  return User.create({
    firstName: 'John',
    lastName: 'Hancock'
  });
});
```

上面定义的 Model，在程序初始化的时候将会创建相应的数据库字段。并且会创建一个 firstName 为 'John'，lastName 为 'Hancock' 的用户。而这个过程中，我们并不需要操作数据库。

像如 MongoDB 这类的数据库，也是存在数据模型，但说的却是嵌入子文档。在业务量大的情况下，数据库在考验公司的技术能力，想想便觉得 Amazon RDS 挺好的。

搜索引擎

尽管百科上对于搜索引擎的定义是这样的：

> 搜索引擎指自动从因特网搜集信息，经过一定整理以后，提供给用户进行查询的系统。

但是这样说往得不是非常准备。因为有相当多的网站采用了搜索引擎作为基础的存储服务架构，而且他们并非自动从互联网上搜索信息。搜索引擎应该分成三个部分来组成：

1. 索引服务
2. 搜索服务
3. 索引数据

索引服务便是用于将数据存储到索引数据中，而搜索服务正是搜索引擎存在的意义。对于查询条件复杂的网站来说，采用搜索引擎就意味着减少了非常多的繁琐数据处理事务。在一些架构中，人们用数据库存储数据，并使用工具来将数据注入到搜索引擎中。

从架构上来说，使用搜索引擎的优点是：分离存储、查询部分。从开发上来说，它可以让我们更关注于业务本身的价值，而不是去实现这样一个搜索逻辑。

如下图所示的 Lucene 应用的架构：

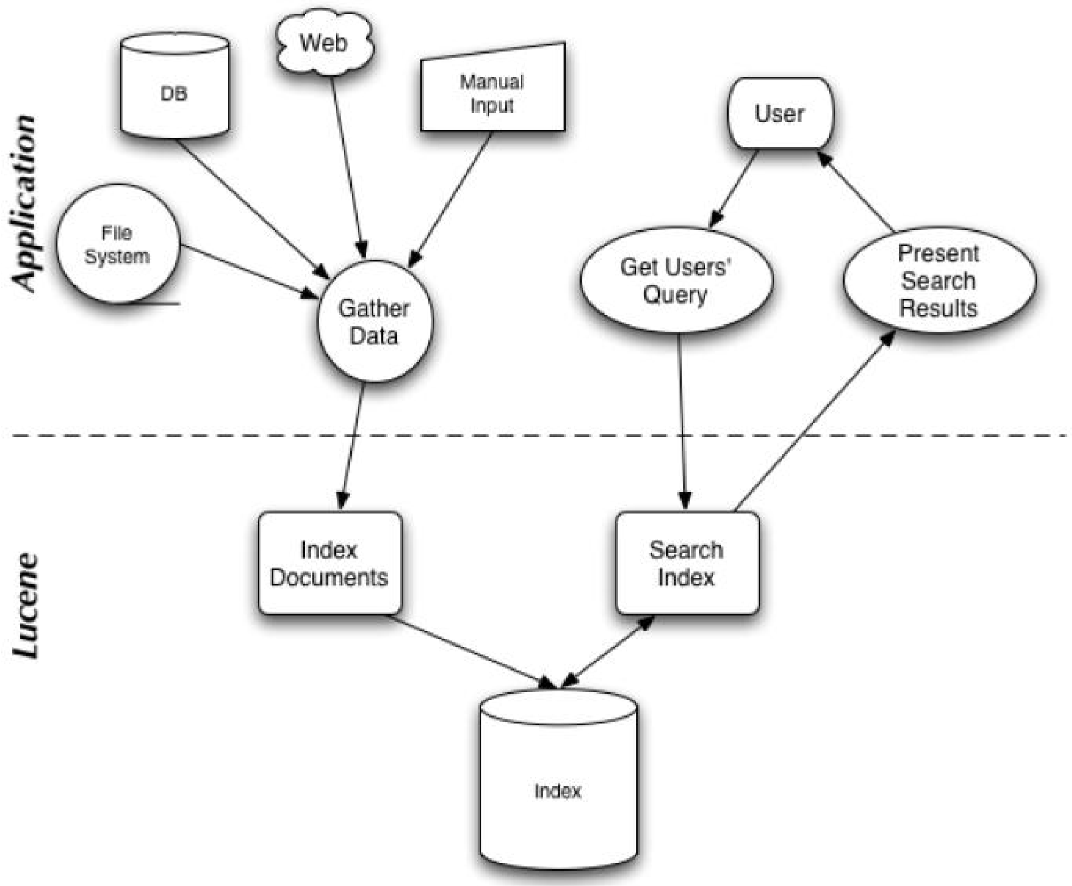


图 38: Lucene 应用架构

可以从图中看到系统明显被划分成两部分：

1. Index Documents。索引文档部分，将用于存储数据到文件系统中。
2. Search Index。搜索部分，用于查询相应的数据。

前端框架选择

选择前端框架似乎是一件很难的事，然而这件事情并不是看上去那么难。只是有时候你只想追随潮流，或者因为你在技术选型受到一些影响。但是总的来说，选择一个框架并不是一件很难的事。同时也不是一件非常重要的事，因为框架本身是相通的。如果我们不尽量去解耦系统，那么选择什么框架也都是一样的。

Angular

AngularJS 对于后端人员写前端代码来说，是一个非常不错的选择。Angular 框架采用并扩展了传统 HTML，通过双向的数据绑定来适应动态内容，双向的数据绑定允许模型和视图之间的自动同步。

并且类似于 Ionic 这样的混合框架，也将 Ionic 带到了移动应用的领域。

React

React 似乎很受市场欢迎，各种各样的新知识——虚拟 DOM、JSX、Component 等等。React 只是我们在上面章节里说到的 View 层，而这个 View 层需要辅以其他框架才能完成更多的工作。

并且 React 还有一个不错的杀手锏——React Native，虽然这个框架还在有条不紊地挖坑中，但是这真的是太爽了。以后我们只需要一次开发就可以多处运行了，再也没有比这更爽的事情发生了。

Vue

Vue.js 是一个轻量级的前端框架。它是一个更加灵活开放的解决方案。它允许你以希望的方式组织应用程序，你可以将它嵌入一个现有页面而不一定要做成一个庞大的单页应用。

jQuery 系

jQuery 还是一个不错的选择，不仅仅对于学习来说，而且对于工作来说也是如此。如果你们不是新起一个项目或者重构旧的项目，那么必然你是没有多少机会去超越 DOM。而如果这时候尝试去这样做会付出一定的代价，如我在前端演进史所说的那样——晚点做出选择，可能会好一点。

因为谁说 jQuery 不会去解放 DOM，React 带来的一些新的思想可能就比不上它的缺点。除此，jQuery 耕织几年的生态系统也是不可忽略。

Backbone + Zepto + Mustache 这是前几年（今年 2016）的一个技术方向，今天似乎已经不太常见了。在这种模式下，人们使用 Backbone 来做一些路由、模型、视图、集合方面的工作，而由 jQuery 的兼容者 Zepto 来负责对 DOM 的处理，而 Mustache 在这里则充当模板系统的工作。

前台与后台交互

在我们把后台服务化后，前端跨平台化之前，我们还需要了解前台和后台之间怎么通讯。从现有的一些技术上来看，Ajax 和 WebSocket 是比较受欢迎的。

Ajax

AJAX 即 “Asynchronous JavaScript And XML”（异步 JavaScript 和 XML），是指一种创建交互式网页应用的网页开发技术。这个功能在之前的很多年来一直被 Web 开发者所忽视，直到最近 Gmail、Google Suggest 和 Google Maps 的出现，才使人们开始意识到其重要性。通过在后台与服务器进行少量数据交换，AJAX 可以使网页实现异步更新。这意味着可以在不重新加载整个网页的情况下，对网页的某部分进行更新。传统的网页如果需要更新内容，必须重载整个网页页面。

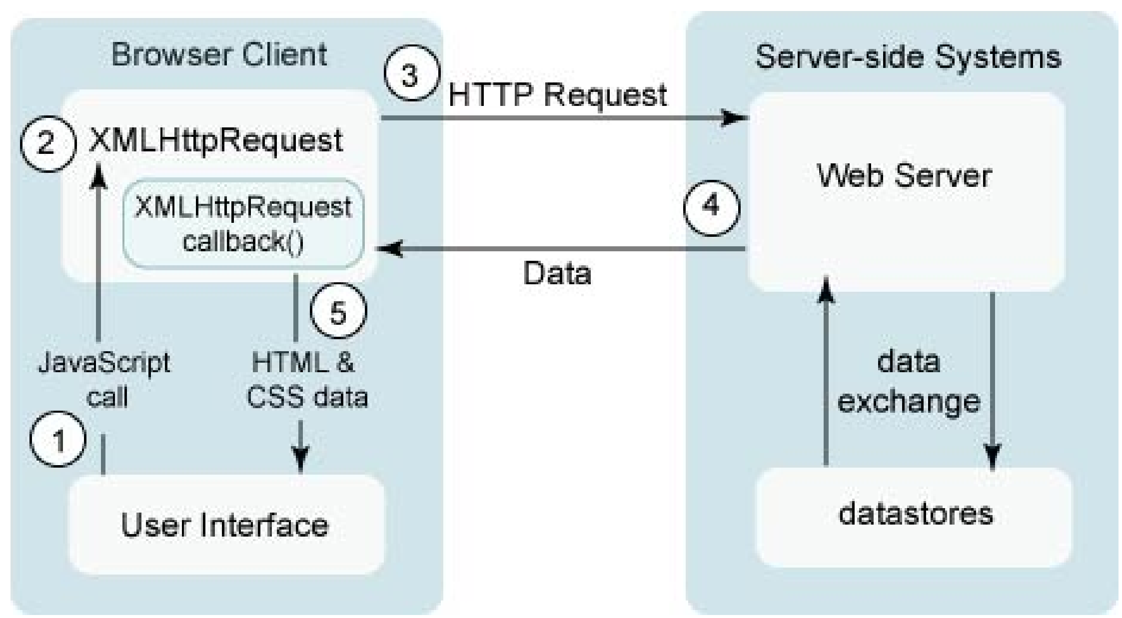


图 39: Ajax 请求

说起 Ajax，我们就需要用 JavaScript 向服务器发送一个 HTTP 请求。这个过程要从 XMLHttpRequest 开始说起，它是一个 JavaScript 对象。它最初由微软设计，随后被 Mozilla、Apple 和 Google 采纳。如今，该对象已经被 W3C 组织标准化。

如下的所示的是一个 Ajax 请求的示例代码：

```
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
    if (xhr.readyState == XMLHttpRequest.DONE) {
        alert(xhr.responseText);
    }
}
xhr.open('GET', 'http://example.com', true);
xhr.send(null);
```

我们只需要简单的创建一个请求对象实例，打开一个 URL，然后发送这个请求。当传输完毕后，结果的 HTTP 状态以及返回的响应内容也可以从请求对象中获取。

而这个返回的内容可以是多种格式，如 XML 和 JSON，但是从近年的趋势来看，XML 基本上已经很少看到了。这里我们以 JSON 为主，来简单地介绍一下返回数据的解析。

JSON

> JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。它基于 ECMAScript 的一个子集。JSON 采用完全独立于语言的文本格式，但是也使用了类似于 C 语言家族的习惯（包括 C、C++、C#、Java、JavaScript、Perl、Python 等）。这些特性使 JSON 成为理想的数据交换语言。易于人阅读和编写，同时也易于机器解析和生成 (一般用于提升网络传输速率)。

XML VS JSON JSON 格式的数据具有以下的一些特点：

- 容易阅读
- 解析速度更快
- 占用空间更少

如下所示的是一个简单的对比过程：

```
myJSON = {"age" : 12, "name" : "Danielle"}
```

如果我们要取出上面数值中的 age，那么我们只需要这样做：

```
anObject = JSON.parse(myJSON);
anObject.age === 12 // True
```

同样的，对于 XML 来说，我们有下面的格式:

```
<person>
    <age>12</age>
    <name>Danielle</name>
</person>
```

而如果我们要取出上面数据中的 age 的值，他将是这样的：

```
myObject = parseThatXMLPlease();
thePeople = myObject.getChildren("person");
thePerson = thePeople[0];
thePerson.getChildren("age")[0].value() == "12" // True
```

对比一下，我们可以发现 XML 的数据不仅仅解析上比较麻烦，而且还繁琐。

JSON WEB Tokens

JSON Web Token (JWT) 是一种基于 token 的认证方案。

在人们大规模地开始 Web 应用的时候，我们在授权的时候遇到了一些问题，而这些问题不是 Cookie 所能解决的。Cookie 存在一些明显的问题：不能支持跨域、并且不是无状态的、不能使用 CDN、与系统耦合等等。除了解决上面的问题，它还可以提高性能等等。基于 Session 的授权机制需要服务端来保存这个状态，而使用 JWT 则可以跳过这个问题，并且使我们设计出来的 API 满足 RESTful 规范。即，我们 API 的状态应该是没有状态的。因此人们提出了 JWT 来解决这一系列的问题。

通过 JWT 我们可以更方便地写出适用于前端应用的认证方案，如登陆、注册这些功能。当我们使用 JWT 来实现我们的注册、登陆功能时，我们在登陆的时候将向我们的服务器发送我们的用户名和密码，服务器验证后将生成对应的 Token。在下次我们进行页面操作的时候，如访问 /Dashboard 时，发出的 HTTP 请求的 Header 中会包含这

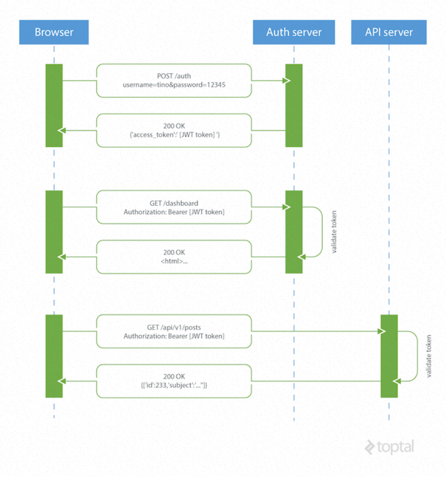


图 40: JWT 流程

个 Token。服务器在接收到请求后，将对这个 Token 进行验证并判断这个 Token 是否已经过期了。

需要注意的一点是：在使用 JWT 的时候也需要注意安全问题，在允许的情况下应该使用 HTTPS 协议。

WebSocket

在一些网站上为了实现推送技术，都采用了轮询的技术。即在特定的的时间间隔里，由浏览器对服务器发出 HTTP 请求，然后浏览器便可以从服务器获取最新的技术。如下图所示的是 Google Chrome 申请开发者账号时发出的对应的请求：

从上图中我们可以看到，Chrome 的前台正在不断地向后台查询 API 的结果。由于浏览器需要不断的向服务器发出请求，而 HTTP 的 Header 是非常长的，即使是一个很小的数据也会占用大量的带宽和服务器资源。为了解决这个问题，HTML5 推出了一种在单个 TCP 连接上进行全双工通讯的协议 WebSocket。

WebSocket 可以让客户端和服务器之间存在持久的连接，而且双方都可以随时开始发送数据。

编码

在我们真正开始去写代码之前，我们可能会去考虑一些事情。怎么去规划我们的任务，如何去细分这个任务。

1. 如果一件事可以自动化，那么就尽量去自动化，毕竟你是一个程序员。

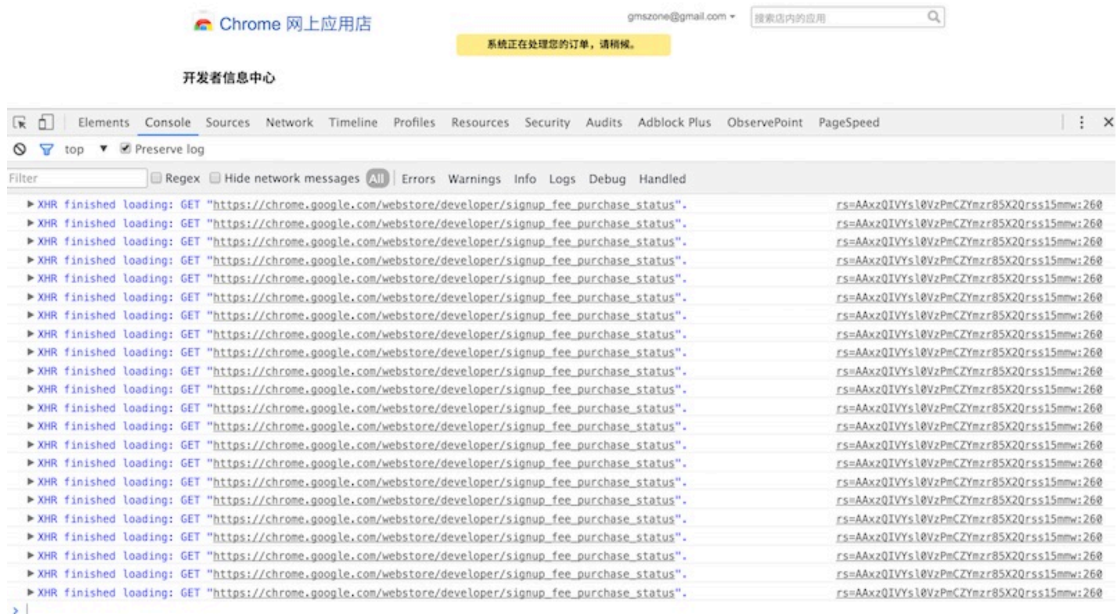


图 41: Chrome Ajax 轮询

2. 快捷键！快捷键！快捷键！
3. 使用可以帮助你快速工作的工具——如启动器。

不过不得不提到的一点是：你需要去考虑这个需求是不是一个坑的问题。如果这是个一个坑，那么你应该尽早的去反馈这个问题。沟通越早，成本越低。

编码过程

整个编程的过程如下图所示：

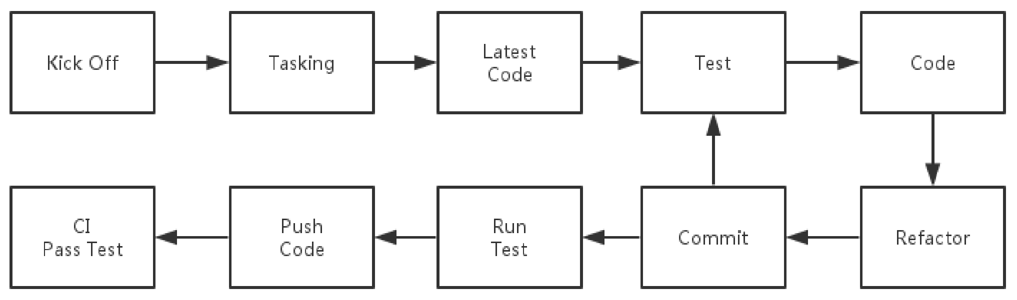


图 42: 编码过程

步骤如下所示：

1. Kick Off。在这个步骤中，我们要详细地了解我们所需要做的东西、我们的验收条件是什么、我们需要做哪些事情。
2. Tasking。简单的规则一下，我们需要怎么做。一般来说，如果是结对编程的话，还会记录下来。
3. 最新的代码。对于使用 Git 来管理项目的团队来说，在一个任务刚开始的时候应该保证本地的代码是最新的。
4. Test First。测试优先是一个很不错的实践，可以保证我们写的代码的健壮，并且函数尽可能小，当然也会有测试。
5. Code。就是实现功能，一般人都知道。
6. 重构。在我们实现了上面两步之后，我们还需要重构代码，使我们的代码更容易阅读、更易懂等等。
7. 提交代码。这里的提交代码只是本地的提交代码，因此都提倡在本地多次提交代码。
8. 运行测试。当我们完成我们的任务后，我们就可以准备 PUSH 代码了。在这时，我们需要在本地运行测试——以保证我们不破坏别人的功能。
9. PUSH 代码。
10. 等 CI 测试通过。如果这时候 CI 是挂的话，那么我们就需要再修 CI。这时其他的人就没有理由 PUSH 代码，如果他们的代码也是有问题的，这只会使情况变得愈加复杂。

不过，在最开始的时候我们要了解一下如何去搭建一个项目。

Web 应用的构建系统

> 构建系统 (build system) 是用来从源代码生成用户可以使用的目标的自动化工具。目标可以包括库、可执行文件、或者生成的脚本等等。

常用的构建工具包括 GNU Make、GNU autotools、CMake、Apache Ant（主要用于 JAVA）。此外，所有的集成开发环境（IDE）比如 Qt Creator、Microsoft Visual Studio 和 Eclipse 都对他们支持的语言添加了自己的构建系统配置工具。通常 IDE 中的构建系统只是基于控制台的构建系统（比如 Autotool 和 CMake ）的前端。

对比于 Web 应用开发来说，构建系统应该还包括应用打包 (如 Java 中的 Jar 包，或者用于部署的 RPM 包、源代码分析、测试覆盖率分析等等。

Web 应用的构建过程

在刚创建项目的时候，我们都会有一个完整的构建思路。如下图便是这样的一个例子：

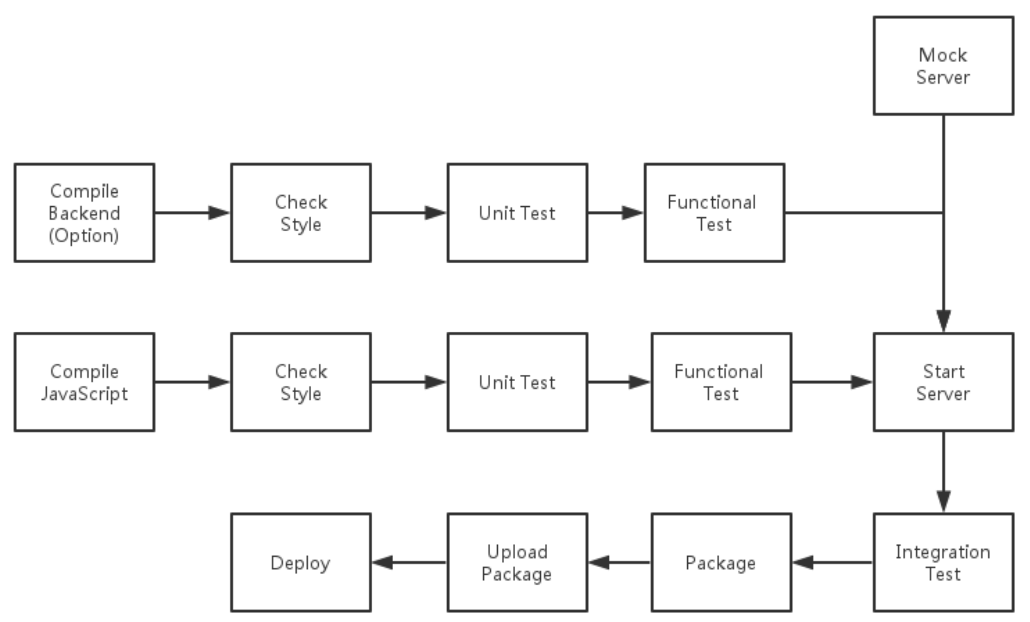


图 43: 构建过程

这是一个后台语言用的是 Java，前台语言用的是 JavaScript 项目的构建流程。

Compile。对于那些不是用浏览器的前端项目来说，如 ES6、CoffeeScript，他们还需要将代码编译成浏览器使用的 JavaScript 版本。对于 Java 语言来说，他需要一个编译的过程，在这个编译的过程中，会检查一些语法问题。

Check Style。通常我们会在我们的项目里定义一些代码规范，如 JavaScript 中的使用两个空格的缩进，Java 的 Checkstyle 中一个函数不能超过 30 行的限制。

单元测试。作为测试中最基础也是最快的测试，这个测试将集中于测试单个函数的是不是正确的。

功能测试。功能测试的意义在于，保证一个功能依赖的几个函数组合在一起也是可以工作的。

Mock Server。当我们的代码依赖于第三方服务的时候，我们就需要一个 Mock Server 来保证我们的功能代码可以独立地测试。

集成测试。这一步将集成前台、后台，并且运行起最后将上线的应用。接着依据于用户所需要的功能来编写相应的测试，来保证一个个的功能是可以工作的。

打包。对于部署来说，直接安装一个 RPM 包，或者 DEB 包是最方便的事。在这个包里会包含应用程序所需的所有二进制文件、数据和配置文件等等。

上传包。在完成打包后，我们就可以上传这个软件包了。

部署。最后，我们就可以在我们的线上环境中安装这个软件包。

Web 应用的构建实战

下面就让我们来构建一个简单的 Web 应用，来实践一下这个过程。在这里，我们要使用到的一个工具是 Gulp，当然对于 Grunt 也是类似的。

Gulp 入门指南

> Gulp.js 是一个自动化构建工具，开发者可以使用它在项目开发过程中自动执行常见任务。Gulp.js 是基于 Node.js 构建的，利用 Node.js 流的威力，你可以快速构建项目并减少频繁的 IO 操作。Gulp.js 源文件和你用来定义任务的 Gulp 文件都是通过 JavaScript（或者 CoffeeScript ）源码来实现的。

1. 全局安装 gulp：

```
$ npm install --global gulp
```

2. 作为项目的开发依赖（devDependencies）安装：

```
$ npm install --save-dev gulp
```

3. 在项目根目录下创建一个名为 gulpfile.js 的文件：

```
var gulp = require('gulp');

gulp.task('default', function() {
  // 将你的默认的任务代码放在这
});
```

4. 运行 gulp：

```
$ gulp
```

默认的名为 default 的任务（task）将会被运行，在这里，这个任务并未做任何事情。接下来，我们就可以打造我们的应用的构建系统了。

代码质量检测工具　当 C 还是一门新型的编程语言时，还存在一些未被原始编译器捕获的常见错误，所以程序员们开发了一个被称作 lint 的配套项目用来扫描源文件，查找问题。

对应于不同的语言都会有不同的 lint 工具，在 JavaScript 中就有 JSLint。JavaScript 是一门年轻、语法灵活多变且对格式要求相对松散的语言，因此这样的工具对于这门语言来说比较重要。

2011 年，一个叫 Anton Kovalyov 的前端程序员借助开源社区的力量弄出来了 JSHint，其思想基本上和 JSLint 是一致的，但是其有一下几项优势：

- 可配置规则，每个团队可以自己定义自己想要的代码规范。
- 对社区非常友好，社区支持度高。
- 可定制的结果报表。

下面就让我们来安装这个软件吧：

安装及使用

```
npm install jshint gulp-jshint --save-dev
```

示例代码:

```
var jshint = require('gulp-jshint');
var gulp   = require('gulp');

gulp.task('lint', function() {
  return gulp.src('./lib/*.js')
    .pipe(jshint())
    .pipe(jshint.reporter('YOUR_REPORTER_HERE'));
});
```

自动化测试工具　一般来说，自动测试应该从两部分考虑：

- 单元测试
- 功能测试

Mocha 是一个可以运行在 Node.js 和浏览器环境里的测试框架，

```
var gulp = require('gulp');
var mocha = require('gulp-mocha');

gulp.task('default', function () {
    return gulp.src('test.js', {read: false})
        // gulp-mocha needs filepaths so you can't have any plugins before it
        .pipe(mocha({reporter: 'nyan'}));
});
```

编译　对于静态型语言来说，编译是一个很重要的步骤。不过，对于动态语言来说也存在这样的工具。

动态语言的编译

可以说这类型的语言，是以我们常见的 JavaScript 为代表。

1. CoffeeScript 是一套 JavaScript 的转译语言，并且它增强了 JavaScript 的简洁性与可读性。
2. Webpack 是一款模块加载器兼打包工具，它能把各种资源，例如 JS（含 JSX）、coffee、样式（含 less/sass）、图片等都作为模块来使用和处理。
3. Babel 是一个转换编译器，它能将 ES6 转换成可以在浏览器中运行的代码。

打包　在 GNU/Linux 系统的软件包里通过包含了已压缩的软件文件集以及该软件的内容信息。常见的软件包有

1. DEB。Debian 软件包格式，文件扩展名为.deb
2. RPM（原 Red Hat Package Manager，现在是一个递归缩写）。该软件包分为二进制包（Binary）、源代码包（Source）和 Delta 包三种。二进制包可以直接安装在计算机中，而源代码包将会由 RPM 自动编译、安装。源代码包经常以 src.rpm 作为后缀名。
3. 压缩文档 tar.gz。通常是该软件的源码，故而在安装的过程中需要编译、安装，并且在编译时需要自己手动安装所需要依赖的软件。在软件仓库没有最新版本的软件时，tar.gz 往往是最好的选择。

由于这里的打包过程比较繁琐，就不介绍了。有兴趣的读者可以自己了解一下。

上传及发布包　上传包之前我们需要创建一个相应的文件服务器，又或者是相应的软件源。并且对于我们的产品环境的服务器来说，我们还需要指定好这个软件源才能安装这个包。

以 Ubuntu 为例，Ubuntu 里的许多应用程序软件包，是放在网络里的服务器上，这些服务器网站，就称作 "源"，从源里可以很方便地获取软件包。

因而在这一步中，我们所需要做的事便是将我们打包完的软件上传到相应的服务器上。

Git 与版本控制

版本控制

> 版本控制是一种记录一个或若干文件内容变化，以便将来查阅特定版本修订情况的系统。

虽然基于 Git 的工作流可能并不是一个非常好的实践，但是在这里我们以这个工作流做为实践来开展我们的项目。如下图所示是一个基于 Git 的项目流：

我们日常会工作在 "develop" 分支（那条线）上，通常来说每个迭代我们会发布一个新的版本，而这个新的版本将会直接上线到产品环境。那么上线到产品环境的这个版本就需要打一个版本号——这样不仅可以方便跟踪我们的系统，而且当出错的时候我们也可以直接回滚到上一个版本。如果在上线的时候有些 Bug 不得不去修复，并且由于上线的新功能很重要，我们就需要一些 Hotfix。而从整个过程来看，版本控制起了一个非常大的作用。

不仅仅如此，版本控制的最大重要是在开发的过程中扮演的角色。通过版本管理系统，我们可以：

1. 将某个文件回溯到之前的状态。
2. 将项目回退到过去某个时间点。
3. 在修改 Bug 时，可以查看修改历史，查出修改原因
4. 只要版本控制系统还在，你可以任意修改项目中的文件，并且还可以轻松恢复。

常用的版本管理系统有 Git、SVN，但是从近年来看 Git 似乎更受市场欢迎。

Git

从一般开发者的角度来看，Git 有以下功能：

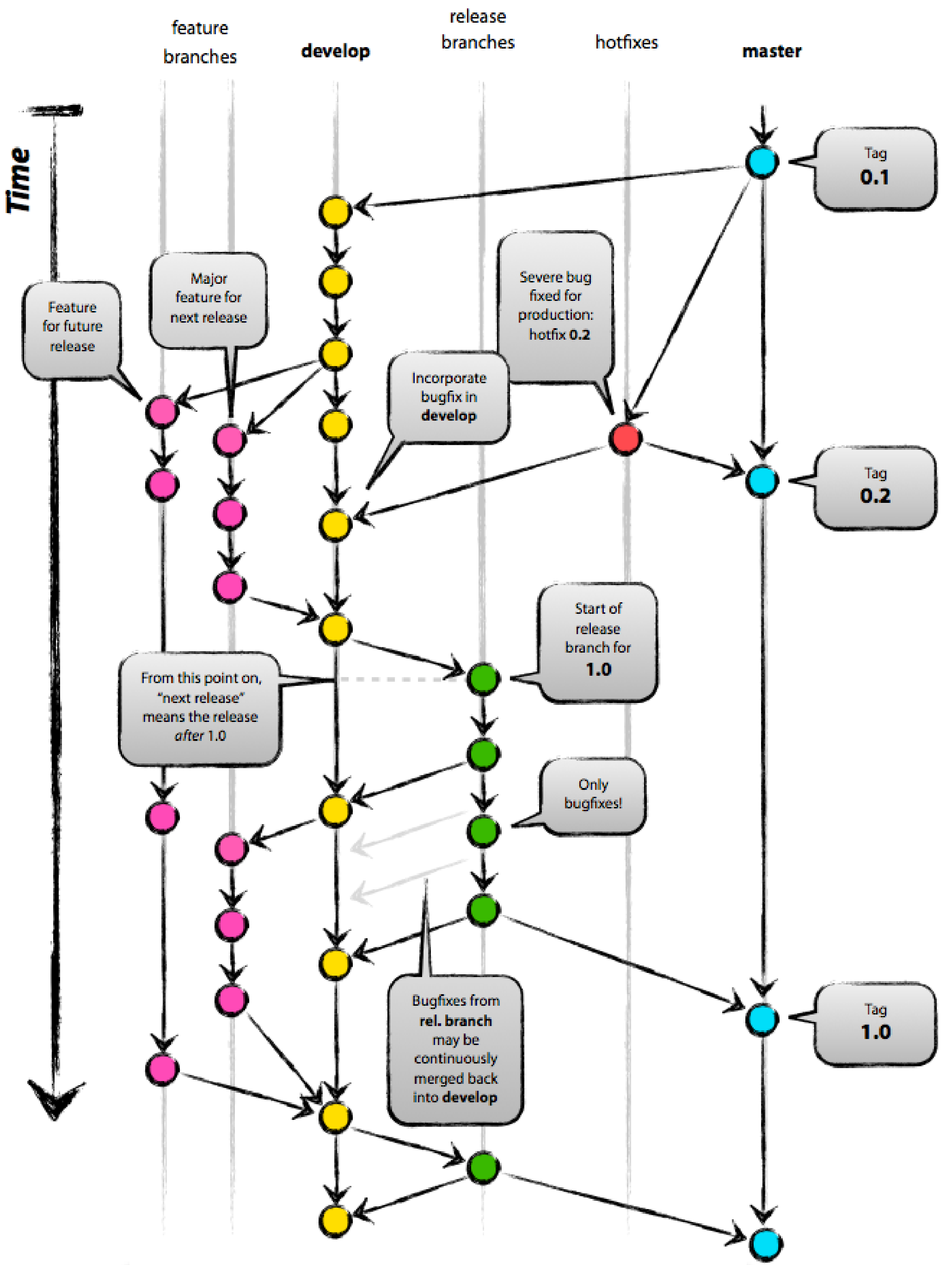


图 44: 基于 Git 的工作流

1. 从服务器上克隆数据库（包括代码和版本信息）到单机上。
2. 在自己的机器上创建分支，修改代码。
3. 在单机上自己创建的分支上提交代码。
4. 在单机上合并分支。
5. 新建一个分支，把服务器上最新版的代码 fetch 下来，然后跟自己的主分支合并。
6. 生成补丁（patch），把补丁发送给主开发者。
7. 看主开发者的反馈，如果主开发者发现两个一般开发者之间有冲突（他们之间可以合作解决的冲突），就会要求他们先解决冲突，然后再由其中一个人提交。如果主开发者可以自己解决，或者没有冲突，就通过。
8. 一般开发者之间解决冲突的方法，开发者之间可以使用 pull 命令解决冲突，解决完冲突之后再向主开发者提交补丁。

从主开发者的角度（假设主开发者不用开发代码）看，Git 有以下功能：

1. 查看邮件或者通过其它方式查看一般开发者的提交状态。
2. 打上补丁，解决冲突（可以自己解决，也可以要求开发者之间解决以后再重新提交，如果是开源项目，还要决定哪些补丁有用，哪些不用）。
3. 向公共服务器提交结果，然后通知所有开发人员。

Git 初入 如果是第一次使用 Git，你需要设置署名和邮箱：

```
$ git config --global user.name "用户名"
$ git config --global user.email "电子邮箱"
```

将代码仓库 clone 到本地，其实就是将代码复制到你的机器里，并交由 Git 来管理：

```
$ git clone git@github.com:someone/symfony-docs-chs.git
```

你可以修改复制到本地的代码了（symfony-docs-chs 项目里都是 rst 格式的文档）。当你觉得完成了一定的工作量，想做个阶段性的提交：

向这个本地的代码仓库添加当前目录的所有改动：

```
$ git add .
```

或者只是添加某个文件：

```
$ git add -p
```

我们可以输入

```
$ git status
```

来看现在的状态，如下图是添加之前的：

```
fdhuang  ⓔtest  gh-pages ↑2  ~/learing/github-roam  git st
On branch gh-pages
Your branch is ahead of 'origin/gh-pages' by 2 commits.
  (use "git push" to publish your local commits)
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   chapters/01-introduction.md
        modified:   chapters/02-github-fundamentals.md
        modified:   github-roam.md
        modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")
fdhuang  ⓔtest  gh-pages ↑2  ~/learing/github-roam
```

图 45: Before add

下面是添加之后的

```
fdhuang  ⓔtest  gh-pages ↑2  ~/learing/github-roam  git st
On branch gh-pages
Your branch is ahead of 'origin/gh-pages' by 2 commits.
  (use "git push" to publish your local commits)
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   chapters/01-introduction.md
        copied:     chapters/01-introduction.md -> chapters/02-github-fundamentals.md
        modified:   github-roam.md
        modified:   index.html

fdhuang  ⓔtest  gh-pages ↑2  ~/learing/github-roam
```

图 46: After add

可以看到状态的变化是从黄色到绿色，即 unstage 到 add。

在完成添加之后，我们就可以写入相应的提交信息——如这次修改添加了什么内容、这次修改修复了什么问题等等。在我们的工作流程里，我们使用 Jira 这样的工具来管理我们的项目，也会在我们的 Commit Message 里写上作者的名字，如下：

```
$ git commit -m "[GROWTH-001] Phodal: add first commit & example"
```

在这里的 `GROWTH-001` 就相当于是我们的任务号，Phodal 则对应于用户名，后面的提交信息也会写明这个任务是干嘛的。

由于有测试的存在，在完成提交之后，我们就需要运行相应的测试来保证我们没有破坏原来的功能。因此，我们就可以 PUSH 我们的代码到服务器端：

```
$ git push
```

这样其他人就可以看到我们修改的代码。

Tasking

初到 ThoughtWorks 时，Pair 时候总会有人教我如何开始编码，这应该也是一项基础的能力。结合日常，重新演绎一下这个过程：

1. 有一个明确的实现目标。
2. 评估目标并将其拆解成任务 (TODO)。
3. 规划任务的步骤 (TODO)
4. 学习相关技能
5. 执行 Task，遇到难题就跳到第二步。

如何 **Tasking** 一本书

以本文的写作为例，细分上面的过程就是：

1. 我有了一个中心思想——在某种意义上来说就是标题。
2. 依据中心思想我将这篇文章分成了四小节。
3. 然后我开始写四小节的内容。
4. 直到完成。

而如果将其划分到一个编程任务，那么也是一样的：

1. 我们想到做一个 xxx 的 idea。
2. 为了这个 idea 我们需要分成几步，或者几层设计。
3. 对于每一步，我们应该做点什么
4. 我们需要学习怎样的技能
5. 集成每一步的代码，就有了我们的系统。

现在让我们以这本书的写作过程为例，来看看这个过程是怎么发生的。

在计划写一本书的时候，我们有关于这本书主题的一些想法。正是一些想法慢慢地凝聚成一个稳定的想法，不过这不是我们所要讨论的重点。

当我们已经有了一本书的相关话题的时候，我们会打算去怎么做？先来个头脑风暴，在上面写满我们的一些想法，如这本书最开始划分了这七步：

- 从零开始
- 编码
- 上线
- 数据分析
- 持续交付
- 遗留系统
- 回顾与新架构

接着，依据我们的想法整理出几个章节。如本书最初的时候只有七个章节，但是我们还需要第一个章节来指引新手，因此变成了八个章节。对应于每一个章节，我们都需要想好每一章里的内容。如在第一章里，又可以分成不同的几部分。随后，我们再对每一部分的内容进行任务划分，那么我们就会得到一个又一个的小的章节。在每个小的章节里，我们都可以大概策划一下我们要写的内容。

然后我们就可以开始写这样的一本书——由一节节汇聚成一章，由一章一章汇聚成一本。

Tasking 开发任务

现在，让我们简单地来 Tasking 如何开发一个博客。作为一个程序员，如果我们要去开始一个博客系统的话，那么我们会怎么做？

1. 让规划一下我们所需要的功能——如后台、评论、Social 等等，并且我们还应该设计我们博客的 Mockup。
2. 随后我们就可以简单地设计一下系统的架构，如传统的前后端结合。
3. 我们就可以进行技术选型了——使用哪个后端框架、使用哪个前端框架。
4. 创建我们的 hello,world，然后开始进行一个功能的编码工作。
5. 编码时，我们就需要不断地查看、添加测试等等。
6. 完成一个个功能的时候，我们就会得到一个子模块。
7. 依据一个个子模块，我们就可以得到我们的博客系统。

与我们日常开发一致的是：我们需要去划分任务的优先级。换句话来说，我们需要先实现我们的核心功能。

对于我们的博客系统来说，最主要的功能就是发博客、展示博客。往简单地说，一篇博客应该有这么基础的四部分：

1. 标题
2. 内容
3. 作者
4. 时间
5. Slug

然后，我们就需要创建相应的 Model，根据这个 Model，我们就可以创建相应的控制器代码。再配置下路由，添加下页面。对于有些系统来说，我们就可以完成博客系统的展示了。

写代码只是在码字

编程这件事情实际上一点儿也不难，当我们只是在使用一个工具创造一些东西的时候，比如我们拿着电烙铁、芯片、电线等去焊一个电路板的时候，我们学的是如何运用这些工具。虽然最后我们的电路板可以实现相同的功能，但是我们可以一眼看到差距所在。

换个贴切一点的比喻，比如烧菜做饭，对于一个优秀的厨师和一个像我这样的门外汉而言，就算给我们相同的食材、厨具，一段时间后也许一份是诱人的美食，一份只能喂猪了——即使我模仿着厨师的步骤一步步地来，也许看上去会差不多，但是一吃便吃出差距了。

我们还做不好饭，还焊不好电路，还写不好代码，很大程度上并不是因为我们比别人笨，而只是别人比我们做了更多。有时候一种机缘巧遇的学习或者 bug 的出现，对于不同的人的编程人生都会有不一样的影响 (ps: 说的好像是蝴蝶效应)。我们只是在使用工具，使用的好与坏，在某种程序上决定了我们写出来的质量。

写字便是如此，给我们同样的纸和笔 (ps: 减少无关因素)，不同的人写出来的字的差距很大，写得好的相比于写得不好的，只是因为练习得更多。而编程难道不也是如此么，最后写代码这件事就和写字一样简单了。

刚开始写字的时候，我们需要去了解一个字的笔划顺序、字体结构，而这些因素相当于语法及其结构。熟悉了之后，写代码也和写字一样是简简单单的事。

学习编程只是在学造句

? 多么无聊的一个标题

计算机语言同人类语言一样，有时候我们也许会感慨一些计算机语言是多么地背离我们的世界，但是他们才是真正的计算机语言。

计算机语言是模仿人类的语言，从 if 到其他，而这些计算机语言又比人类语言简单。故而一开始学习语言的时候我们只是在学习造句，用一句话来概括一句代码的意思，或者可以称之为函数、方法 (method)。

于是我们开始组词造句，以便最后能拼凑出一整篇文章。

编程是在写作

> ? 编程是在写作，这是一个怎样的玩笑? 这是在讽刺那些写不好代码，又写不好文章的么

代码如诗，又或者代码如散文。总的来说，这是相对于英语而言，对于中文而言可不是如此。如果用一种所谓的中文语言写出来的代码，不能像中文诗一样，那么它就算不上是一种真正的中文语言。

那些所谓的写作逻辑对编程的影响

- 早期的代码是以行数算的，文章是以字数算的
- 代码是写给人看的，文章也是写给人看的
- 编程同写作一样，都由想法开始
- 代码同文章一样都可以堆砌出来 (ps: 如本文)
- 写出好的文章不容易，需要反复琢磨，写出好的代码不也是如此么
- 构造一个类，好比是构造一个人物的性格特点，多一点不行，少一点又不全
- 代码生成，和生成诗一样，没有情感，过于机械化
- 。。。

然而好的作家和一般的写作者，区别总是很大，对同一个问题的思考程度也是不同的。从一个作者到一个作家的过程，是一个不断写作不断积累的过程。而从一个普通的程序员到一个优秀的程序员也是如此，需要一个不断编程的过程。

当我们开始真正去编程的时候，我们还会纠结于“僧推月下门”还是“僧敲月下门”的时候，当我们越来越熟练就容易决定究竟用哪一个。而这样的“推敲”，无论在写作中还是在编程中都是相似的过程。

> 写作的过程真的就是一次探索之旅，而且它会贯穿人的一生。

因此:

> 编程只是在码字，难道不是么?

真正的想法都在脑子里，而不在纸上，或者 IDE 里。

内置索引与外置引擎

门户网站

让我们先来看看门户网站。

百科上说：

> 门户网站（英语：Web portal，又稱入口網站，入门网站）指的是将不同来源的信息以一种整齐划一的形式整理、储存并呈现的网站

从某种意义上来说门户网站更适合那些什么都不知道，从头开始探索互联网的人。换句话说，这类似于有点于类似我们在学第一门计算机语言——我们不需要去寻找什么，我们也不知道一些复杂的概念。

这时候我们只能随便的看一本别人推荐的书籍，读一读别人写的笔记，开始一点点构建我们的知识体系。

而在我们学习第二门计算机语言的时候，我们有了更多的诀窍——我们知道怎么去搜索。在我们的知识体系里，我们知道如何去搜索，这时我们就可以通过搜索引擎来学习。

百科上大致将搜索引擎分成了四部分：搜索器、索引器、检索器、用户接口。

1. 搜索器：其功能是在互联网中漫游，发现和搜集信息。
2. 索引器：其功能是理解搜索器所搜索到的信息，从中抽取出索引项，用于表示文档以及生成文档库的索引表。
3. 检索器：其功能是根据用户的查询在索引库中快速检索文档，进行相关度评价，对将要输出的结果排序，并能按用户的查询需求合理反馈信息。
4. 用户接口：其作用是接纳用户查询、显示查询结果、提供个性化查询项。

我想这部分大家都是有点印象的就不多介绍了 (即：Ctrl + C, Ctrl + V)。

对于一个新手来说，使用搜索引擎的最大障碍就是——你知道问题，但是你不知道怎么搜索。这也是为什么，你会在那么多的博客、问答里，看到如何使用搜索引擎。

但是这并不能解决根本性问题——你需要知道你的问题是什么。顺便，推荐一本书叫做《你的灯亮着吗?》

内置索引与外置引擎

(ps: 为了和搜索引擎对应起来，这里就将内置门户改成内置索引。)

所以，再仔细回到上面的问题里。要成为一名可以完成任务的程序员，你就需要不断地构建你的门户网站。我们要学习 Web 开发，我们就需要对整个知识体系有一个好的理解。不断理解的过程中，我们就不断也添加了新的文档，构建新的索引。每遇到一个新的知识点，我们就开始重新生成新的索引。

然后又会引入一个问题：

> 人的大脑如同一间空空的阁楼，要有选择地把一些家具装进去。

我们需要不断地整理一些新的技术，并且想方设法地忘记旧的知识。

有时，不得不说笔记和博客是这样一个很好的载体。在未来的某一天，我们可以重新挖掘这些技术，识别技术的旧有缺陷，发展出新的技术——水能载舟，亦能覆舟。

如何编写测试

写测试相比于写代码来说算是一种简单的事。多数时候，我们并不需要考虑复杂的逻辑。我们只需要按照我们的代码逻辑，对代码的行为进行覆盖。

需要注意的是——在不同的团队、工作流里，测试可能是会有不同的工作流程：

- 开发人员写单元测试、集成测试等等
- 测试团队通过界面来做黑盒测试
- 测试人员手动测试来测试功能

在允许的情况下，测试应该由开发人员来编写，并且是由底层开始写测试。为了更好地去测试代码，我们需要了解测试金字塔。

测试金字塔

测试金字塔是由 Mike Cohn 提出的，主要观点是：底层单元测试应多于依赖 GUI 的高层端到端测试。其结构图如下所示：

从结构上来说，上面的金字塔可以分成三部分：

1. 单元测试。

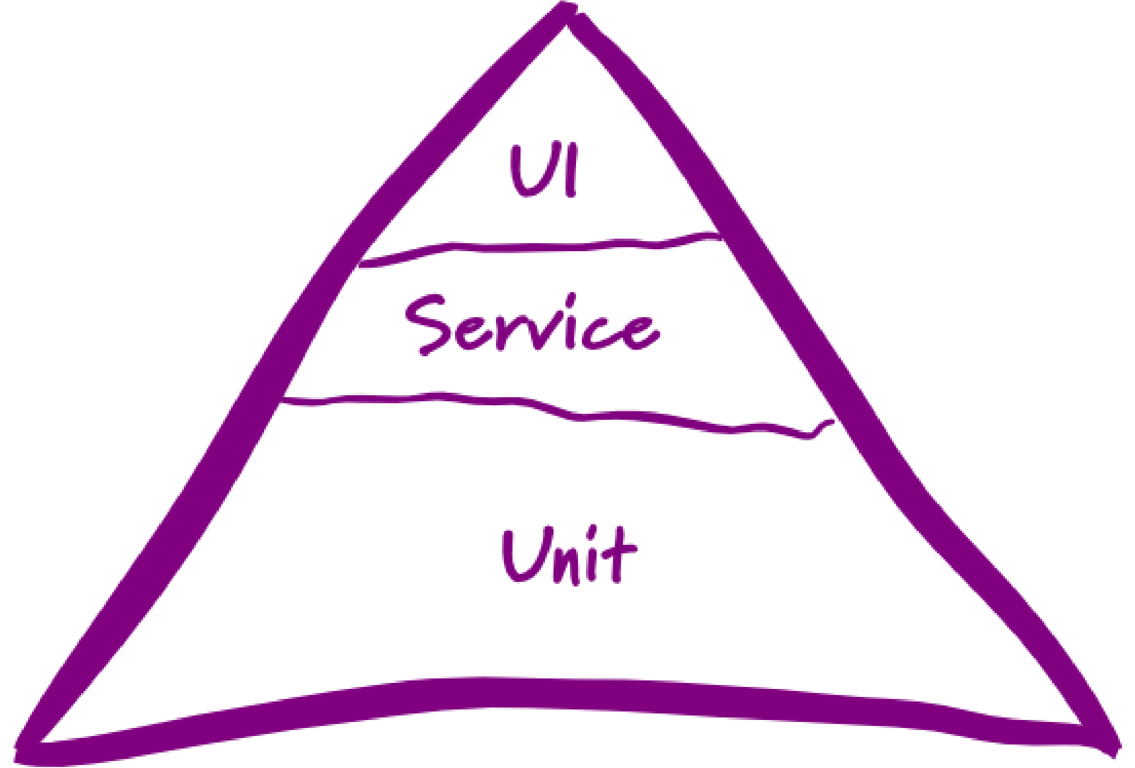


图 47: 测试金字塔

2. 服务测试
3. UI 测试

从图中我们可以发现：单元测试应该要是最多的，也是最底层的。其次才是服务测试，最后才是 UI 测试。大量的单元测试可以保证我们的基础函数是正常、正确工作的。而服务测试则是一门很有学问的测试，不仅仅只在测试我们自己提供的服务，也会测试我们依赖第三方提供的服务。在测试第三方提供的服务时，这就会变成一件有意思的事了。除此还有对功能和 UI 的测试，写这些测试可以减轻测试人员的工作量——毕竟这些工作量转向了开发人员来完成。

单元测试 单元测试是针对程序模块（软件设计的最小单位）来进行正确性检验的测试工作。它是应用的最小可测试部件。举个例子来说，下面是一个 JavaScript 的函数，用于判断一个变量是否是一个对象：

```
var isObject = function (obj) {
    var type = typeof obj;
    return type === 'function' || type === 'object' && !!obj;
};
```

这是一个很简单的功能，对应的我们会有一个简单的 Jasmine 测试来保证这个函数是正常工作的：

```
it("should be a object", function () {
    expect(l.isObject([])).toEqual(true);
    expect(l.isObject([{}])).toEqual(true);
});
```

虽然这个测试看上去很简单，但是大量的基本的单元测试可以保证我们调用的函数都是可以正常工作的。这也相当于是我们在建设金字塔时用的石块——如果我们的石块都是经常测试的，那么我们就不怕金字塔因为石块的损坏而坍塌。

当单元测试达到一定的覆盖率，我们的代码就会变得更健壮。因为我们都需要保证我们的代码都是可测的，也意味着我们代码间的耦合度会降低。我们需要去考虑代码的长度，越长的代码在测试的时间会变得越困难。这也就是为什么 TDD 会促使我们写出短的代码。如果我们的代码都是有测试的，单元测试可以帮助我们在未来重构我们的代码。

并且在很多没有文档或者文档不完整的开源项目中，了解这个项目某个函数的用法就是查看他的测试用例。测试用例（Test Case）是为某个特殊目标而编制的一组测试输入、执行条件以及预期结果，以便测试某个程序路径或核实是否满足某个特定需求。这些测试用例可以让我们直观地理解程序程序的 API。

服务测试　服务测试顾名思义便是对服务进行测试，而服务可以是有不同的类型，不同层次的测试。如第三方的 API 服务、我们程序提供的服务，虽然他们他应该在这一个层级上进行测试，但是对他们的测试会稍有不同。

对于第三方的提供的 API 服务或者其他类似的服务，在这一个层级的测试，我们都不会真实地去测试他们能不能工作——这些依赖性的服务只会在功能测试上进行测试。在这里的测试，我们只会保证我们的功能代码是可以正常工作的，所以我们会使用一些虚假的 API 测试数据来进行测试。这一类提供 API 的 Mock Server 可以模拟被测系统外部依赖模块行为的通用服务。我们只要保证我们的功能代码是正常工作的，那么依赖他的服务也会是正常工作的。

而对于我们提供的服务来说，这一类的服务不一定是 API 的服务，还有可能是多个函数组成的功能性服务。当我们在测试这些服务的时候，实际上是在测试这个函数结合在一起是不是正常的。

一个服务可能依赖于多个函数，因而我们会发现服务测试的数量是少于单元测试的。

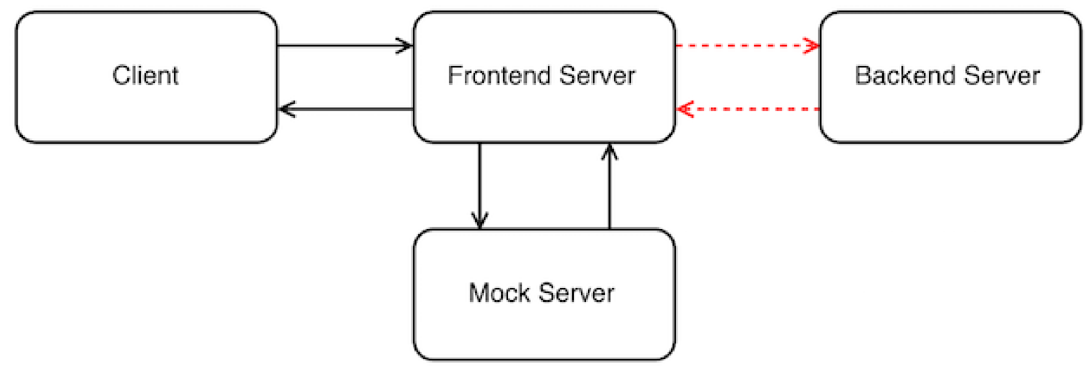


图 48: Mock Server

UI 测试　在传统的软件开发中，UI 测试多数是由人手动来完成的。而在稍后的章节里，你将会看到这些工作是可以由机器自己来完成的——当然，前提是我们要编写这些自动化测试的代码。需要注意的是 UI 测试并不能完全替代手工的工作，一些测试还是应该由人来进行测试——如对 UI 的布局，在现阶段机器还没有审美意识呢。

自动化 UI 测试是一个缓慢的过程，在这个过程里我们需要做这么几件事：

1. 运行起我们的网站——这可能需要几分钟。
2. 添加一些 Mock 的数据，以使网站看上去正常——这也需要几分钟到几十分钟的时间。
3. 开始运行测试——在一些依赖于网络的测试中，运行完一个测试可能会需要几分钟。尽管可以并行运行测试，但是一个测试几分钟算到最后就会累积成很长的时间。

所以，你会发现这是一个很长的测试过程。尽可能地将这个层级的测试往下层级移，就会尽可能的节省时间。一个 UI 测试需要几分钟，但是一个单元测试可能不到 1 秒。这就意味着，这样的测试下移可以节省上百个数量级的时间。

如何测试

现在问题来了，我们应该怎么去写测试？换句话来说，我要测什么？这是一个很难的问题，这足够可以以一本书的幅度来说明这个问题。这个问题也需要依赖于不同的实践，不同的时候我们可能对问题的看法都有不同。

编写测试的过程大致可以分成下面的几个步骤：

1. 了解测试目的 (Why)？即我们需要测什么，我们是为了什么而编写的测试。

2. 我们要测哪些内容 (What)? 即测试点，我们即要从功能点上出发来寻找需要我们测试的点，在不同的条件下这个测试点是不一样的。
3. 我们要如何进行测试 (How)? 我们要使用怎么样的方法进行测试?

测试目的　我们在上面提到过的测试金字塔，也表明了我们在每个层级要测试的目的是不一样的。

在单元测试这一层级，因为我们所测试的是每一个函数，这些函数没有办法构成完成的功能。这时候我们就只是用于简简单单的测试函数本身的功能，没有太多的业务需求。

而对于服务这一层级，我们所要测试的就是一个完整的功能。对于以 API 为主的项目来说，实际上就是在测返回结果是否是正确的。

最后 UI 这一层级，我们所需要测试的就是一个完整的功能。用户操作的时候应该是怎样的，那么我们就应该模仿用户的行为来测试。这是一个完整的业务需求，也可以称之为验证测试。

测试点　在了解完我们要测试的目的之后，我们要测试的点也变得很清晰。即在单元测试测试我们的函数的功能，在我们的服务测试我们的服务，在我们的 UI 测试测试业务。

而这些都理想的情况，当系统由于业务的原因不得不耦合的时候。究竟是单元测试还是功能测试，这是一个特别值得思考的问题。如果一个功能即可以在单元测试里测，又可以在服务测试里测，那么我们要测试哪一个? 或者说我们应该把两个都测一遍? 而如果是花费时间更长的 UI 测试呢? 这样做是不是会变得不划算。

如何写测试代码　先让来们来简单地看一下测试用例，然后再让我们看看一般情况下我们是如何写测试代码的。下面的代码是一个用 Python 写的测试用例:

```
class HomepageTestCase(LiveServerTestCase):
    def setUp(self):
        self.selenium = webdriver.Firefox()
        self.selenium.maximize_window()
        super(HomepageTestCase, self).setUp()

    def tearDown(self):
        self.selenium.quit()
        super(HomepageTestCase, self).tearDown()
```

```python
    def test_can_visit_homepage(self):
        self.selenium.get(
            '%s%s' % (self.live_server_url,  "/")
        )

        self.assertIn("Welcome to my blog", self.selenium.title)
```

在上面的代码里主要有三个方法，setUp()、tearDown() 和 test_can_visit_homepage()。在这三个方法中起主要作用的是 test_can_visit_homepage() 方法。而 setUp() 和 tearDown() 是特殊的方法，分别在测试方法开始之前运行和之后运行。同时，在这里我们也用这两个方法来打开和关闭浏览器。

而在我们的测试方法 test_can_visit_homepage() 里，主要有两个步骤：

1. 访问首页
2. 验证首页的标题是 “Welcome to my blog”

大部分的测试代码也是以如何的流程来运行着。有一点需要注意的是：一般来说函数名就表示了这个测试所要做测试的事情，如这里就是测试可以访问首页。

如上所示的测试过程称为“四阶段测试”，即这个过程分为如下的四个阶段：

1. **Setup**。在这个阶段主要是做一些准备工作，如数据准备和初始化等等，在上面的 setup 阶段就是用 selenium 启动了一个 Firefox 浏览器，然后把窗口最大化了。
2. **Execute**。在执行阶段就是做好验证结果前的工作，如我们在测试注册的时候，那么这里就是填写数据，并点击提交按钮。在上面的代码里，我们只是打开了首页。
3. **Verify**。在验证阶段，我们所要做的就是验证返回的结果是否和我们预期的一致。在这里我们还是使用和单元测试一样的 assert 来做断言，通过判断这个页面的标题是 “Welcome to my blog”，来说明我们现在就是在首页里。
4. **Tear Down**。就是一些收尾工作啦，比如关闭浏览器、清除测试数据等等。

Tips 需要注意的几点是：

1. 从运行测试速度上来看，三种测试的运行速度是呈倒金字塔结构。即，单元测试跑得最快，开发速度也越快。随后是服务测试，最后是 UI 测试。
2. 即使现在的 UI 测试跑得非常快，但是随着时间的推移，UI 测试会越来越多。这也意味着测试来跑得越来越久，那么人们就开始不想测试了。在我们之前的项目

里，运行完所有的测试大概接近一个小时，我们开始在会议会争论这些测试的必要性，也在想方设法减少这些测试。
3. 如果一个测试可以在最底层写，那么就不要在他的上一层写了，因为他的运行速度更快。

参考书籍:

- 《优质代码——软件测试的原则、实践与模式》
- 《Python Web 开发：测试驱动开发方法》

测试替身

测试替身（Test Double）是一个非常有意思的概念。

> 有时候对被测系统 (SUT) 进行测试是很困难的，因为它依赖于其他无法在测试环境中使用的组件。这有可能是因为这些组件不可用，它们不会返回测试所需要的结果，或者执行它们会有不良副作用。在其他情况下，我们的测试策略要求对被测系统的内部行为有更多控制或更多可见性。如果在编写测试时无法使用（或选择不使用）实际的依赖组件 (DOC)，可以用测试替身来代替。测试替身不需要和真正的依赖组件有完全一样的的行为方式；他只需要提供和真正的组件同样的 API 即可，这样被测系统就会以为它是真正的组件！——Gerard Meszaros

当我们遇到一些难以测试的方法、行为的时候，我们就一些特别的方式来帮助我们测试。Mock 和 Stub 就是常见的两种方式:

1. Stub 是一种状态确认，它用简单的行为来替换复杂的行为
2. Mock 是一种行为确认，它用于模拟其行为

通俗地来说：Stub 从某种程度上来说，会返回我们一个特定的结果，用代码替换来方法；而 Mock 只是确保这个方法被调用。

Stub

Stub 从字面意义上来说是存根，存根可以理解为我们保留了一些预留的结果。这个时候我们相当于构建了这样一个特殊的测试场景，用于替换诸如网络或者 IO 口调度等高度不可预期的测试。如当我们需要去验证某个 API 被调用并返回了一个结果，举例在最小物联网系统设计中返回的 json，我们可以在本地构建一个

```
[{"id":1,"temperature":14,"sensors1":15,"sensors2":12,"led1":1}]
```

的结果来当我们预期的数据，也就是所谓的存根。那么我们所要做的也就是解析 json，并返回预期的结果。当我们依赖于网络时，此时测试容易出现问题。

Mock

Mock 从字面意义上来说是模仿，也就是说我们要在本地构造一个模仿的环境，而我们只需要验证我们的方法被调用了。

```
var Foo = function(){};
Foo.prototype.callMe = function() {};
var foo = mock( Foo );

foo.callMe();

expect( foo.callMe ).toHaveBeenCalled();
```

测试驱动开发

测试驱动开发是一个很“古老”的程序开发方法，然而由于国内的开发流程的问题——即开发人员负责功能的测试，导致这么好的一项技术没有在国内推广。

红-绿-重构

测试驱动开发的主要过程是: 红 —> 绿 -> 重构

1. 先写一个失败的单元测试。即我们并没有实现这个方法，但是已经有了这个方法的测试。
2. 让测试通过。实现简单的代码来保证测试通过，就算我们用一些作弊的方法也是可以的。我们写的是功能代码，那么我们应该提交代码，因为我们已经实现了这个功能。
3. 重构，并改进功能代码，让它变得更加合理。

TDD 有助于我们将问题分解成更小的部分，再一点点的添加我们所需要的业务代码。随着这个过程的不断进行，我们会发现我们已经接近完成我们的功能代码了。并且到了最后，我们会发现我们的代码都会被测试到。

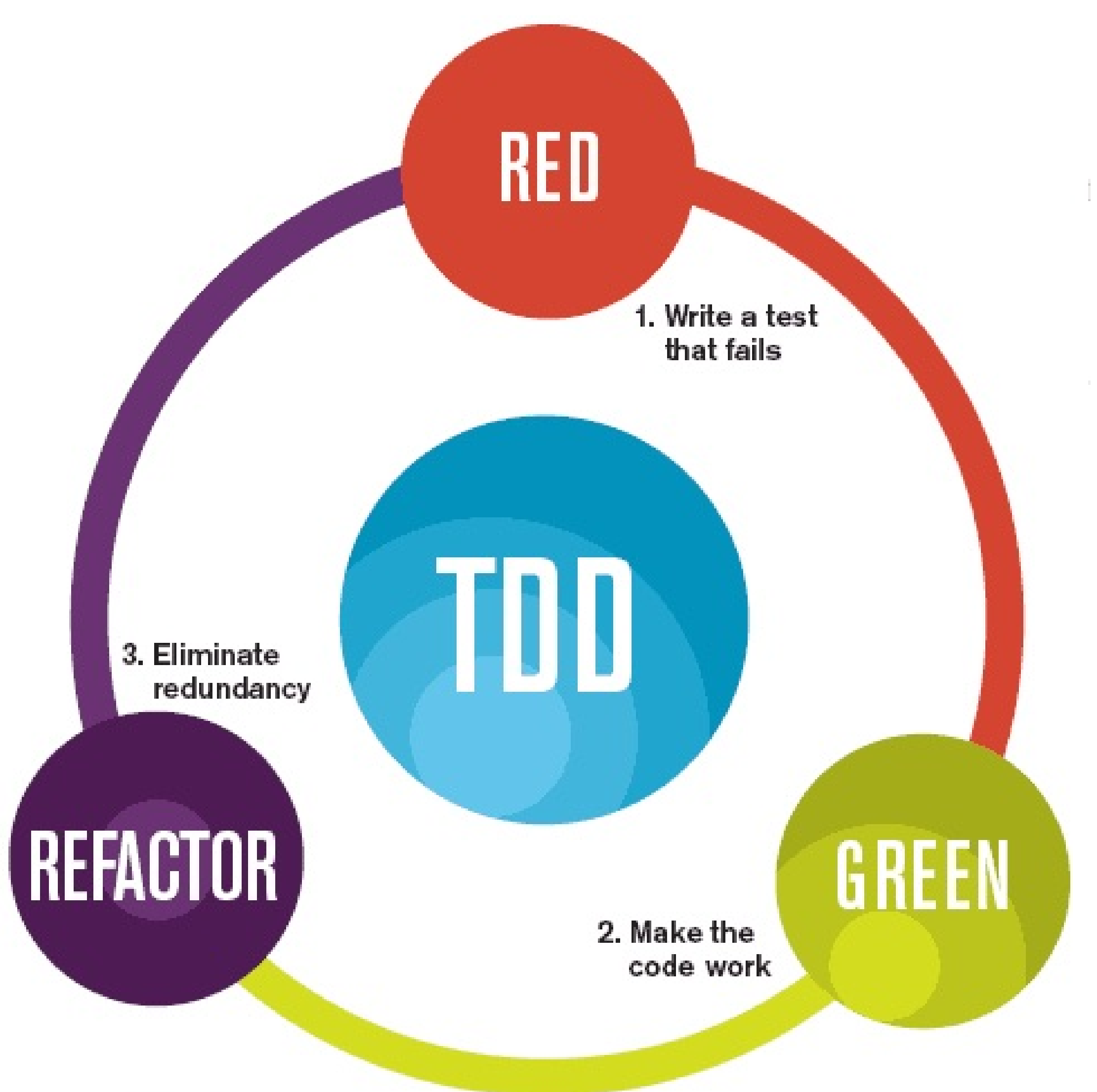


图 49: TDD

虽然说起来看上去很简单，但是真正实现起来并不是那么容易。于我而言我只会在我自己造的一些轮子中使用 TDD。因为这个花费大量的时间，通常来说测试代码和功能代码的比例可能是 1:1，或者是 2：1 等等。在自己创建的一些个人应用，如博客中，我不需要与其他人 Share 我的 Content。由于我使用的是第三方框架，框架本身的测试已经足够多，并且没有复杂的逻辑，我就没有对我的博客写测试。而在我写的一些框架里，我就会尽量保证足够高的测试覆盖率，并且在适当的时候会去 TDD。

通常来说对于单元测试我会采用 TDD 的方式来进行，但是功能测试仍会选择在最后添加进去。主要的缘由是：在写 UI 的过程中，元素会发生变化。这一点和我们在写 Unit 的时候，有很大的区别。div + class 会使得我们思考问题的方式发生变化，我们需要去点击某个元素，并观察某个元素发生的变化。而多数时候，我们很难把握好一个页面最好的样子。

不得不说明的一点是，TDD 需要你对测试比较了解后，你才容易使用它。从个人的感受来说，TDD 是在一开始是一件很难的事。

测试先行

对于写测试的人来说，测试先行有点难以理解，而对于不写测试的人来说，就更难以理解。这里假定你已经开始写测试了，因为对于不写测试的人来说，写测试就是一件难以理解的事。既然我们都要写测试，那么为什么我们就不能先写测试呢？或者说为什么后写测试存在一些问题?

依据 J.Timothy King 所总结的《测试先行的 12 个好处》:

1. 测试可证明你的代码是可以解决问题的
2. 一面写单元测试，一面写实现代码，这样感觉更有兴趣
3. 单元测试也可以用于演示代码
4. 会让你在写代码之前做好计划
5. 它降低了 Bug 修复的成本
6. 可以得到一个底层模块的回归测试工具
7. 可以在不改变现有功能的基础上继续改进你的设计
8. 可以用于展示开发的进度
9. 它真实的为程序员消除了工作上的很多障碍
10. 单元测试也可以让你更好的设计
11. 单元测试比代码审查的效果还要好
12. 它比直接写代码的效率更高

但是在我个人的感觉里，多比较喜欢的是：写出可以测试的函数。这是一个一直困

扰着我的难题，特别是当我的代码里存在很多条件的时候，在后期我编写的时候，难度越来越大。当我只有一个简单的 IF-ELSE 的时候，我的代码测试起来也很简单:

```
if (hour < 18) {
    greeting = "Good day";
} else {
    greeting = "Good evening";
}
```

而当我有复杂的业务逻辑时，后写测试就会变成一场恶梦:

```
if (EchoesWorks.isObject(words)) {
    var nextTime = that.parser.parseTime(that.data.times)[currentSlide + 1];
    if (that.time < nextTime && words.length > 1) {
        var length = words.length;
        var currentTime = that.parser.parseTime(that.data.times)[currentSlide
        var time = nextTime - currentTime;
        var average = time / length * 1000;
        var i = 0;
        document.querySelector('words').innerHTML = words[i].word;

        timerWord = setInterval(function () {
            i++;
            if (i - 1 === length) {
                clearInterval(timerWord);
            } else {
                document.querySelector('words').innerHTML = words[i].word;
            }
        }, average);
    }
    return timerWord;
} else {
    document.querySelector('words').innerHTML = words;
}
```

我们需要重新理清业务的逻辑，再依据这些逻辑来编写测试代码。而当我们已经忘记具体的业务逻辑时，我们已然无法写出测试。

思考

通常在我的理解下，TDD 是可有可无的。既然我知道了我要实现的大部分功能，而且我也知道如何实现。与此同时，对 Code Smell 也保持着警惕、要保证功能被测试覆盖。那么，总的来说 TDD 带来的价值并不大。

然而，在当前这种情况下，我知道我想要的功能，但是我并不理解其深层次的功能。我需要花费大量的时候来理解，它为什么是这样的，需要先有一些脚本来知道它是怎么工作的。TDD 变显得很有价值，换句话来说，在现有的情况下，TDD 对于我们不了解的一些事情，可以驱动出更多的开发。毕竟在我们完成测试脚本之后，我们也会发现这些测试脚本成为了代码的一部分。

在这种理想的情况下，我们为什么不 TDD 呢?

参考资料

J.Timothy King 《Twelve Benefits of Writing Unit Tests First》

可读的代码

过去，我有过在不同的场合吐槽别人的代码写得烂。而我写的仅仅是比别人好一点而已——而不是好很多。

然而这是一件很难的事，人们对于同一件事物未来的考虑都是不一样的。同样的代码在相同的情景下，不同的人会有不同的设计模式。同样的代码在不同的情景下，同样的人会有不同的设计模式。在这里，我们没有办法讨论设计模式，也不需要讨论。

我们所需要做的是，确保我们的代码易读、易测试，看上去这样就够了，然而这也是挺复杂的一件事:

- 确保我们的变量名、函数名是易读的
- 没有复杂的逻辑判断
- 没有多层嵌套 (事不过三)
- 减少复杂函数的出现（如, 不超过三十行）
- 然后，你要去测试它。这样你就知道需要什么，实际上要做到这些也不是一些难事。

只是首先，我们要知道我们要自己需要这些。对于没有太多编程经验的人，建议先从两个基本点做起：

- 命名

- 函数长度

首先要说的就是程序员认为最难的一个话题了——命名。

命名

命名是一个特别长的，也是特别忧伤的故事。我想作为一个程序员的你，也相当恐惧这件事。一个好的函数名、变量名应该包含着这个函数的信息，如这个函数是干什么的，或者这个函数是怎么来的，这个变量名存储的是什么。

正因为取名字是一件很重要的事，所以它也是一件很难的事。一个好的函数名、变量名应该能正确地表达出它的涵义。如你可以猜到下面的代码中的 i 是什么意思吗?

```
fruits = ['banana', 'apple',  'mango']
for i in fruits:         # Second Example
   print 'Current fruit :', i
```

那如果换成下面的代码会不会更容易阅读呢?

```
fruits = ['banana', 'apple',  'mango']
for fruit in fruits:         # Second Example
   print 'Current fruit :', fruit
```

而命令还存在于对函数的命名上，如我们可能会用 getNumber 来表示去获取一个数值，但是要知道这样的命名并不是在所有的语言中都可以这样用。如在 Java 中存在 getter 和 setter 这种模式，如下的代码所示:

```
public String getNumber() {
  return number;
}
public void setNumber(String number) {
  this.number = number;
}
```

如果我们是去取某个东西的数值，那么我们应该使用 retrieveNumber 这样的更具代表性的名字。

在《编写可读代码的艺术》也提到了这几点:

1. 选择专业的词。最好是可以和业务相关的，它应该极具表现力。
2. 避免像 tmp 和 retval 这样泛泛的名字。不得不提到的一点是，tmp 实在是一个有够烂的名字，将其变为 timeTemp 或者类似的会更直观。它只应该是名字中的一部分。
3. 用具体的名字代替抽象的名字。
4. 为名字赋予更多的信息。
5. 名字应该有多长。
6. 利用名字的格式来传递含义。

函数长度

> 函数是指一段在一起的、可以做某一件事儿的程序。

这就意味着从定义上来说，这段函数应该只做一件事——但是什么才是真正的一件事呢？实际上还是 TASKING，将一个复杂的过程一步步地分解成一个个的函数，每个函数只做他的名称对应的事。对于一个任务来说，他有一个稳定的过程，在这个过程中的每一步都可以变成一个函数。

因此，长的代码意味着一件事——这个函数可能违反了单一职责原则，即这个类做了太多的事。通常来说，一个类，只有一个引起它变化的原因。当一个类有多个职责的时候，这些代码就容易耦合到一起了。

对于函数长度的控制是为了有效控制分支深度。如果我们用一个函数来实现一个复杂的功能，那么不仅仅在我们下次阅读的时间会花费大量的时间。而且如果我们的代码没有测试话，那么这些代码就会变得越来越难以理解。而在我们写这些函数的时候就没有测试，那么这个函数就会变得越来越难以测试，它们就会变成遗留代码。

其他

虽然只想介绍上面的简单的两点，但是顺便在这里也提一下重复代码 ~~。

重复代码　在《重构》一书中首先提到的 Code Smell 就是重复代码（Duplicate Code）。重复代码看上去并不会影响我们的阅读体验，但是实际上会发生这样的事——重复的代码阅读体验越不好。

DRY(Don't Repeat Yourself) 原则是特别值得玩味的。当我们不断地偏执的去减少重复代码的时候，会导致复杂度越来越高。在适当的时候，由于业务发生变更，我们还需要去拆解这些不重复的代码。

代码重构

> 重构，一言以蔽之，就是在不改变外部行为的前提下，有条不紊地改善代码。

代码重构（英语：Code refactoring）指对软件代码做任何更动以增加可读性或者简化结构而不影响输出结果。在经历了一年多的工作之后，我平时的主要工作就是修 Bug。刚开始的时候觉得无聊，后来才发现修 Bug 需要更好的技术。有时候你可能要面对着一坨一坨的代码，有时候你可能要花几天的时间去阅读代码。而，你重写那几十代码可能只会花上你不到一天的时间。但是如果你没办法理解当时为什么这么做，你的修改只会带来更多的 Bug。修 Bug，更多的是维护代码。还是前人总结的那句话对:

> 写代码容易，读代码难。

假设我们写这些代码只要半天，而别人读起来要一天。为什么不试着用一天的时候去写这些代码，让别人花半天或者更少的时间来理解。

重命名

在上一节中，我们提到了命名的重要性，这里首先要说到的也就是重命名~。让再看看《编写可读代码的艺术》也提到了这几点:

1. 选择专业的词。最好是可以和业务相关的，它应该极具表现力。
2. 避免像 tmp 和 retval 这样泛泛的名字。不得不提到的一点是，tmp 实在是一个有够烂的名字，将其变为 timeTemp 或者类似的会更直观。它只应该是名字中的一部分。
3. 用具体的名字代替抽象的名字。
4. 为名字赋予更多的信息。
5. 名字应该有多长。
6. 利用名字的格式来传递含义。

提取变量

先让我们来看看一个简单的情况:

```
if ($scope.goodSkills.indexOf('analytics') !== -1) {
  skills.analytics = 5;
}
```

在上面的代码里比较难以看懂的就是数字 5，这时候你会怎么做？写一行注释？这里的 5 就是一个 Magic Number。

而实际上，最简单有效的办法就是把 5 提取成一个变量：

```
var LEVEL_FIVE = 5;
if ($scope.goodSkills.indexOf('analytics') !== -1) {
  skills.analytics = LEVEL_FIVE;
}
```

提炼函数

这个简单有效的方法就是为了对付之前太长的函数，抽取提炼函数出应该抽取出来的部分成为一个新的函数。引自《重构》一书的说法，短的精巧的函数有以下的特点：

1. 如果每个函数的粒度都很小，那么函数被复用的机会就更大；
2. 是这会让高层函数读起来就像一系列注释一样，容易理解；
3. 是如果函数都是细粒度，那么函数的复写也会更加容易。

在提炼函数中我们所要做的就是——判断出原有的函数的意图，再依据我们的新意图来命名新的函数。然后判断依赖——变量值，处理这些变量。提取出函数，最近对其测试。

这里只简单地对重构进行一些介绍，更多详细信息请参阅《重构：改善既有代码的设计》。

Intellij Idea 重构

下面简单地介绍一下，一些可以直接使用 IDE 就能完成的重构。这种重构可以用在日常的工作中，只需要使用 IDE 上的快捷键就可以完成了。

提炼函数

Intellij IDEA 带了一些有意思的快捷键，或者说自己之前不在意这些快捷键的存在。重构作为单独的一个菜单，显然也突显了其功能的重要性，说说提炼函数，或者说提出方法。

快捷键

Mac: `alt+command+M`

Windows/Linux: `Ctrl+Alt+M`

鼠标: Refactor | Extract | Method

重构之前

以重构一书代码为例，重构之前的代码

```
public class extract {
    private String _name;

    void printOwing(double amount){
        printBanner();

        System.out.println("name:" + _name);
        System.out.println("amount" + amount);
    }

    private void printBanner() {
    }
}
```

重构

选中

```
System.out.println("name:" + _name);
System.out.println("amount" + amount);
```

按下上述的快捷键，会弹出下面的对话框

输入

```
printDetails
```

那么重构就完成了。

重构之后

IDE 就可以将方法提出来

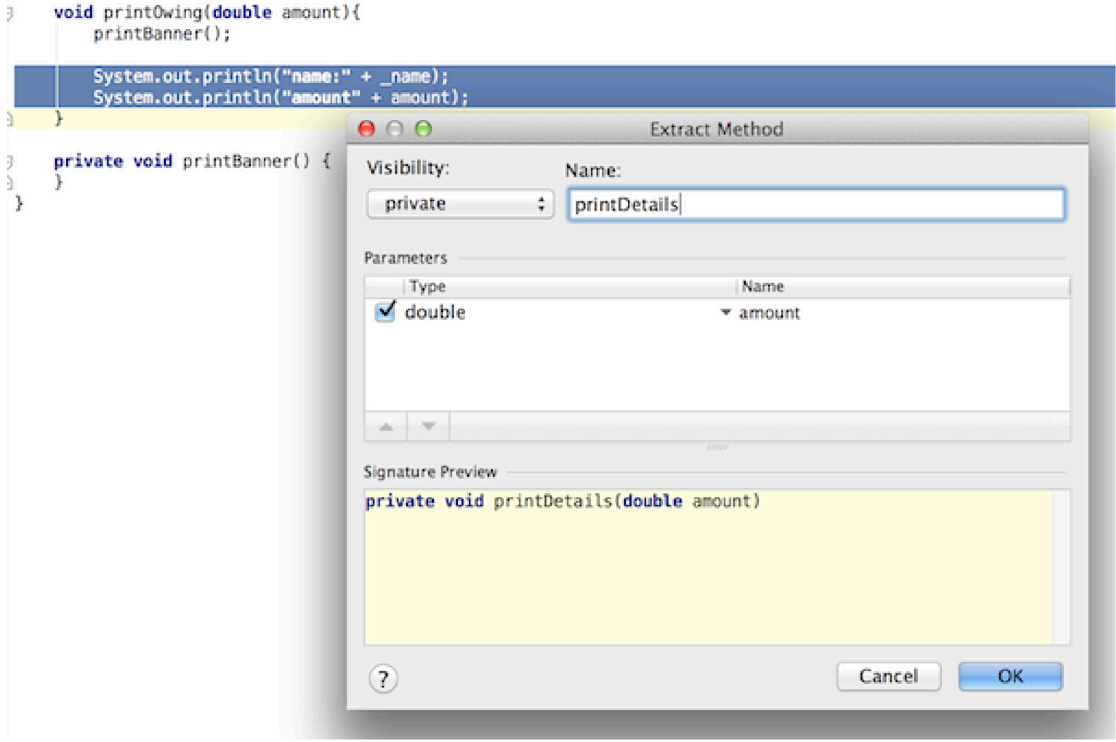


图 50: Extrct Method

```
public class extract {
    private String _name;

    void printOwing(double amount){
        printBanner();
        printDetails(amount);
    }

    private void printDetails(double amount) {
        System.out.println("name:" + _name);
        System.out.println("amount" + amount);
    }

    private void printBanner() {
    }
}
```

重构

还有一种就以 Intellij IDEA 的示例为例，这像是在说其的智能。

```
public class extract {
    public void method() {
        int one = 1;
        int two = 2;
        int three = one + two;
        int four = one + three;
    }
}
```

只是这次要选中的只有一行，

```
int three = one + two;
```

以便于其的智能，它便很愉快地告诉你它又找到了一个重复

```
 IDE has detected 1 code fragments in this file that can be replaced with a c
```

便返回了这样一个结果

```
public class extract {

    public void method() {
        int one = 1;
        int two = 2;
        int three = add(one, two);
        int four = add(one, three);
    }

    private int add(int one, int two) {
        return one + two;
    }

}
```

然而我们就可以很愉快地继续和它玩耍了。当然这其中还会有一些更复杂的情形，当学会了这一个剩下的也不难了。

内联函数

继续走这重构一书的复习之路，接着便是内联，除了内联变量，当然还有内联函数。

快捷键

Mac: `alt+command+M`

Windows/Linux: `Ctrl+Alt+M`

鼠标: Refactor | Inline

重构之前

```
public class extract {

    public void method() {
        int one = 1;
        int two = 2;
        int three = add(one, two);
        int four = add(one, three);
    }

    private int add(int one, int two) {
        return one + two;
    }

}
```

在 `add(one,two)` 很愉快地按上个快捷键吧，就会弹出

再轻轻地回车，Refactor 就这么结束了。。

Intellij Idea 内联临时变量

以书中的代码为例

```
double basePrice = anOrder.basePrice();
return (basePrice > 1000);
```

同样的，按下 `Command+alt+N`

```
return (anOrder.basePrice() > 1000);
```

```
    int three = add(one, two);
    int four = add(one, three);
}

private int add(int one, int two) {
    return one + two;
}
```

Inline Method

Method add(int, int)

- (•) Inline all invocations and remove the method (2 occurrences)
- ( ) Inline this invocation only and keep the method
- [x] Search in comments and strings [x] Search for text occurrences

? Cancel Preview Refactor

图 51: Inline Method

对于 python 之类的语言也是如此

```
def inline_method():
    baseprice = anOrder.basePrice()
    return baseprice > 1000
```

查询取代临时变量

快捷键

Mac: 木有

Windows/Linux: 木有

或者: `Shift+alt+command+T` 再选择 `Replace Temp with Query`

鼠标: **Refactor** | `Replace Temp with Query`

重构之前

过多的临时变量会让我们写出更长的函数，函数不应该太多，以便使功能单一。这也是重构的另外的目的所在，只有函数专注于其功能，才会更容易读懂。

以书中的代码为例

```
import java.lang.System;
```

```java
public class replaceTemp {
    public void count() {
        double basePrice = _quantity * _itemPrice;
        if (basePrice > 1000) {
            return basePrice * 0.95;
        } else {
            return basePrice * 0.98;
        }
    }
}
```

重构

选中 basePrice 很愉快地拿鼠标点上面的重构

图 52: Replace Temp With Query

便会返回

```java
import java.lang.System;

public class replaceTemp {
    public void count() {
        if (basePrice() > 1000) {
            return basePrice() * 0.95;
        } else {
            return basePrice() * 0.98;
        }
    }

    private double basePrice() {
        return _quantity * _itemPrice;
    }
}
```

而实际上我们也可以

1. 选中

   `_quantity * _itemPrice`

2. 对其进行 `Extrace Method`
3. 选择 `basePrice` 再 `Inline Method`

在 Intellij IDEA 的文档中对此是这样的例子

```java
public class replaceTemp {

    public void method() {
        String str = "str";
        String aString = returnString().concat(str);
        System.out.println(aString);
    }

}
```

接着我们选中 aString，再打开重构菜单，或者

Command+Alt+Shift+T 再选中 Replace Temp with Query

便会有下面的结果:

```
import java.lang.String;

public class replaceTemp {

    public void method() {
        String str = "str";
        System.out.println(aString(str));
    }

    private String aString(String str) {
        return returnString().concat(str);
    }

}
```

重构到设计模式

> 模式和重构之间存在着天然联系，模式是你想到达的目的地，而重构则是从其他地方到达这个目的地的条条道理——Martin Fowler《重构》

过度设计与设计模式

过度设计和设计模式是两个很有意思的词语，这取决于我们是不是预先式设计。通过以往的经验我们很容易看到一个环境来识别一个模式。遗憾的是使用设计模式来依赖于我们整个团队的水平。对于了解设计模式的人来说，设计模式就是一种沟通语言。而对于了解一些设计模式的人来说，设计模式就是复杂的代码。

并且在软件迭代的过程中需求总是不断变化的，这就意味着如果我们对我们的代码设计越早，那么在后期失败的概率也就越大。设计会伴随着需求而发生变化，在当时看起来合理的设计，在后期就会因此而花费过多的代价。

而如果我们不进行一些设计，就有可能出现设计不足。这种情况可能出现于没有时间写出更好的代码的项目，在这些项目里由于一些原因出现加班等等的原因，使得我们

没有办法写出更好的代码。同时，也有可能是因为参考项目的程序员的设计方面出现不足。

我们没有对设计模式介绍的一个原因是——它需要有大量的编程经验，才可以让我们实现：重构到设计模式。

上线

作为一个开发人员，我们也需要去了解如何配置服务器。不仅仅因为它可以帮助我们更好地理解 Web 开发，而且有时候很多 Bug 都是因为服务器环境引起的——如臭名昭著地编码问题。

- 一些简单的 Ops 技能。
- 了解服务器的相关软件
- 搭建运行 Web 应用的服务器
- 自动化部署应用

为了即时的完成工作，你是不是放弃了很多东西，比如质量? 测试是很重要的一个环节，不仅可以为我们保证代码的质量，而且还可以为我们以后的重构提供基础条件。

作为一个在敏捷团队里工作的开发人员，初次意识到在国内大部分的开发人员是不写测试的时候，我还是有点诧异。

尽管没有写测试可以在初期走得很快，但是在后期就会遇到一堆麻烦事。传统的思维下，我们会认为一个人会在一家公司工作很久。而这件事在最近几年里变化得特别快，特别是在信息技术高速发展的今天。人们可以从不同的地方得到哪里缺人，从一个地方到另外一个地方也变得异常的快，这就意味着人员流动是常态。

而代码尽管还在，但是却会随着人员流动而出现更多的问题。这时如果代码是有有效的测试，那么则可以帮助系统更好地被理解。

隔离与运行环境

为了将我们的应用部署到服务器上，我们需要为其配置一个运行环境。从底层到顶层有这样的运行环境及容器：

1. 隔离硬件：虚拟机
2. 隔离操作系统：容器虚拟化
3. 隔离底层：Servlet 容器

4. 隔离依赖版本：虚拟环境
5. 隔离运行环境：语言虚拟机
6. 隔离语言：DSL

实现上这是一个请求的处理过程，一个 HTTP 请求会先到达你的主机。如果你的主机上运行着多个虚拟机实例，那么请求就会来到这个虚拟机上。又或者是如果你是在 Docker 这一类容器里运行你的程序的话，那么也会先到达 Docker。随后这个请求就会交由 HTTP 服务器来处理，如 Apache、Nginx，这些 HTTP 服务器再将这些请求交由对应的应用或脚本来处理。随后将交由语言底层的指令来处理。

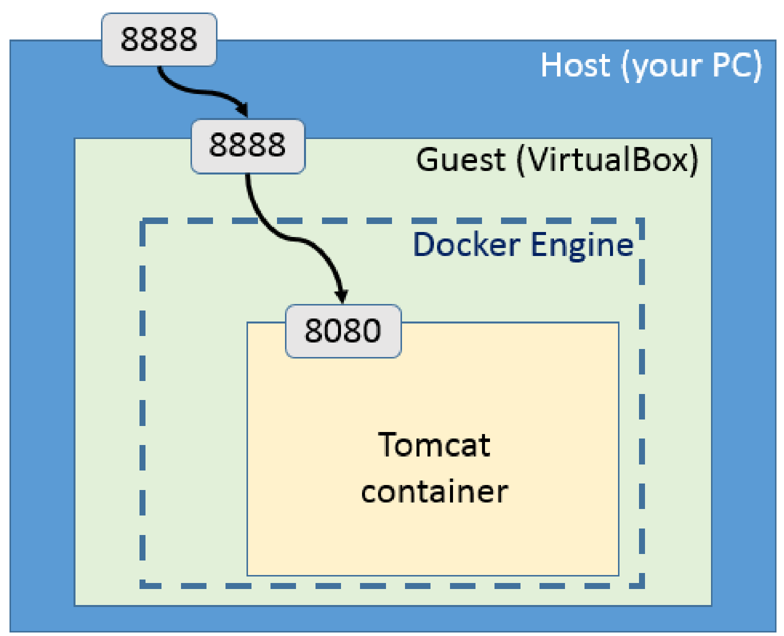


图 53: Docker Tomcat

不同的环境有不同的选择，当然也可以结合在一起。不过，从理论上来说在最外层还是应该有一个真机的，但是我想大家都有这个明确的概念，就不多解释了。

隔离硬件：虚拟机

在虚拟机技术出现之前，为了运行不同用户的应用程序，人们需要不同的物理机才能实现这样的需求。对于 Web 应用程序来说，有的用户的网站访问量少消耗的系统资源也少，有的用户的网站访问量大消耗的系统资源也多。虽然有不同的服务器类型可以选择，然而对于多数的访问少的用户来说他们需要支付同样的费用。这听上去相当的不合理，并且也浪费了大量的资源。并且对于系统管理员来说，管理这些系统也不是一件容易的事。在过去硬件技术革新特别快，让操作系统运行在不同的机器上也不是一件容易的事。

> 虚拟机（Virtual Machine）指通过软件模拟的具有完整硬件系统功能的、运行在一个完全隔离环境中的完整计算机系统。

这是一个很有意思的技术，它可以让我们在一个主机上同时运行几个不同的操作系统。我们可以为这几个操作系统使用不同的硬件，在这之上的应用可以使用不同的技术栈来运行，并且从理论上互相不影响。其架构如下图所示：

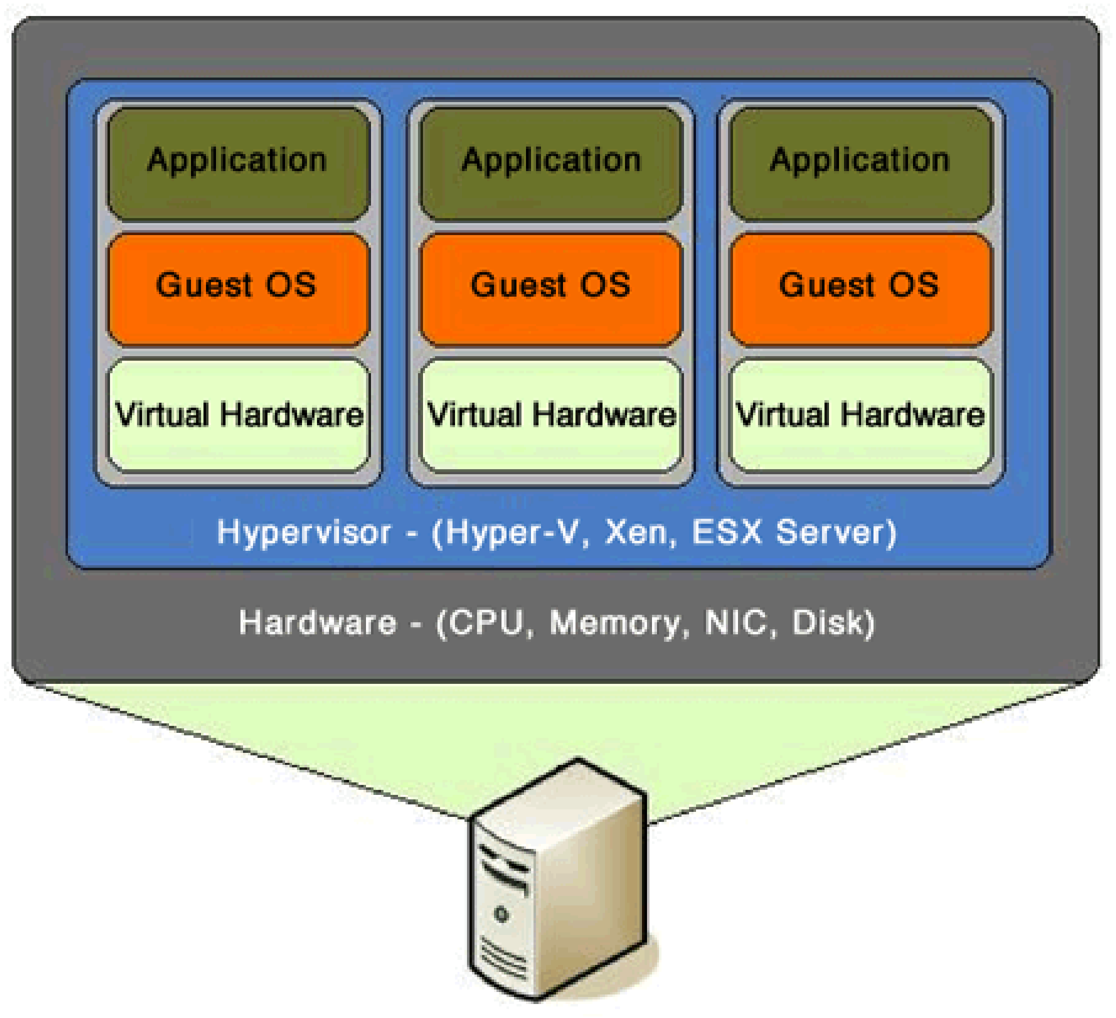


图 54: 虚拟机

借助于虚拟机技术，当我们需要更多的资源的时候，创建一个新的虚拟机就行了。同时，由于这些虚拟机上运行的是同样的操作系统，并且可以使用相同的配置，我们只需要编写一些脚本就可以实现其自动化。当我们的物联机发生问题时，我们也可以很快将虚拟机迁移或恢复到另外的宿主机。

隔离操作系统：容器虚拟化

对于大部分的开发团队来说，直接开发基于虚拟机的自动化工具不是一件容易的事，并且他从使用成本上来说比较高。这时候我们就需要一些更轻量级的工具容器——它可以提供轻量级的虚拟化，以便隔离进程和资源，而且不需要提供指令解释机制以及全虚拟化的其他复杂性。并且，它从启动速度上来说更快。

LXC 在介绍 Docker 之前，我们还是稍微提一下 LXC。因为在过去我有一些使用 LXC 的经历，让我觉得 LXC 很赞。

> LXC，其名称来自 Linux 软件容器（Linux Containers）的缩写，一种操作系统层虚拟化（Operating system–level virtualization）技术，为 Linux 内核容器功能的一个用户空间接口。它将应用软件系统打包成一个软件容器（Container），内含应用软件本身的代码，以及所需要的操作系统核心和库。通过统一的名字空间和共用 API 来分配不同软件容器的可用硬件资源，创造出应用程序的独立沙箱运行环境，使得 Linux 用户可以容易的创建和管理系统或应用容器。

我们可以将之以上面说到的虚拟机作一个简单的对比，其架构图如下所示：

我们会发现虚拟机中多了一层 Hypervisor——运行在物理服务器和操作系统之间，它可以让多个操作系统和应用共享一套基础物理硬件。这一层级可以协调访问服务器上的所有物理设备和虚拟机，然而由于这一层级的存在，它也将消耗更多的能量。据爱立信研究院和阿尔托大学发表的论文表示：Docker、LXC 与 Xen、KVM 在完成相同的工作时要少消耗 10% 的能耗。

LXC 主要是利用 cgroups 与 namespace 的功能，来向提供应用软件一个独立的操作系统运行环境。cgroups（即 Control Groups）j Linux 内核提供的一种可以限制、记录、隔离进程组所使用的物理资源的机制。而由 namespace 来责任隔离控制。

与虚拟机相比，LXC 隔离性方面有所不足，这就意味着在实现可移植部署会遇到一些困难。这时候，我们就需要 Docker 来提供一个抽象层，并提供一个管理机制。

Docker

> Docker 是一个开源的应用容器引擎，让开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。Docker 可以自动化打包和部署任何应用、创建一个轻量级私有 PaaS 云、搭建开发测试环境、部署可扩展的 Web 应用等。

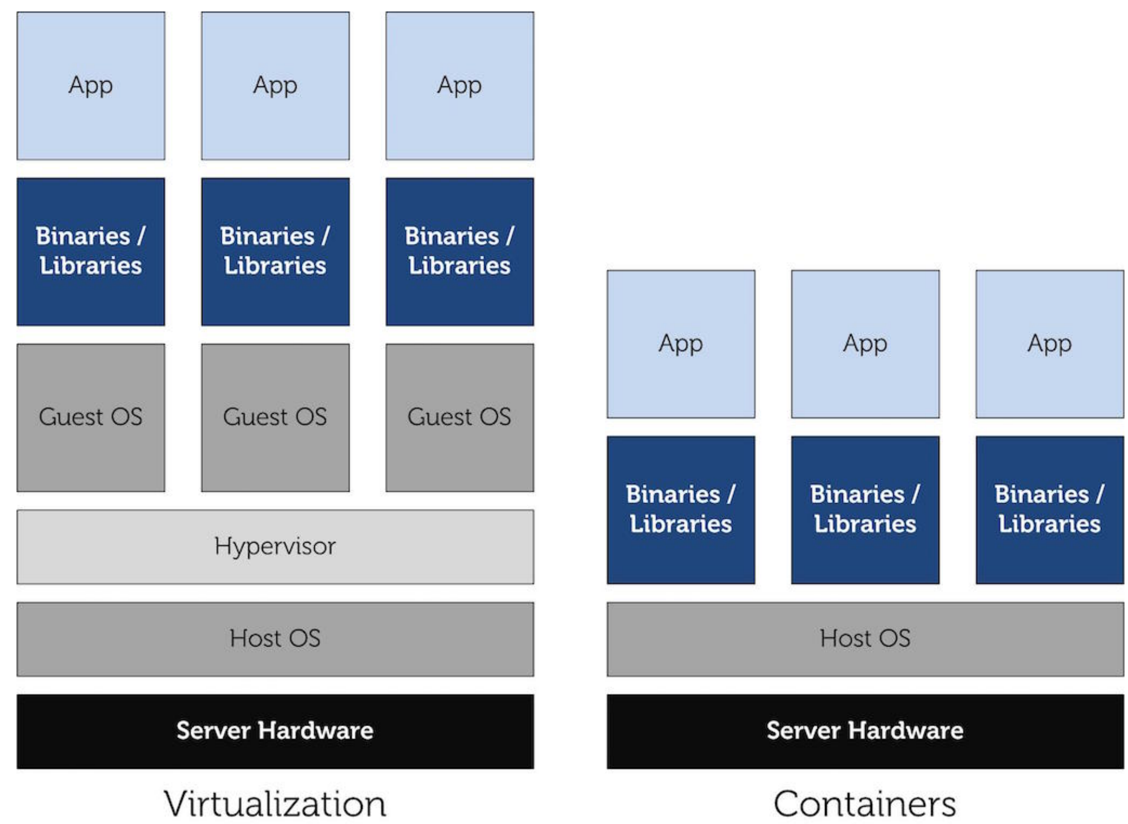


图 55: LXC vs VM

构建出 Docker 的 Container 是一个很有意思的过程。在这一个过程中，首先我们需要一个 base images，这个基础镜像不仅包含了一个基础系统，如 Ubuntu、Debian。他还包含了一系列的模块，如初始化进程、SSH 服务、syslog-ng 等等的一些工具。由上面原内容构建了一个基础镜像，随后的修改都将于这个镜像，我们可以用它生成新的镜像，一层层的往上叠加。而用户的进程运行在 writeable 的 layer 中。

从上图中我们还可以发现一点：Docker 容器是建立在 Aufs 基础上的。AUFS 是一种 Union File System，它可以不同的目录挂载到同一个虚拟文件系统下。它的目的就是为了实现上图的增量递增的过程，同时又不会影响原有的目录。即如下的流程如下：

其增量的过程和我们使用 Git 的过程中有点像，除了在最开始的时候会有一个镜像层。随后我们的修改都可以保存下来，并且当下次我们提交修改的时候，我们也可以在旧有的提交上运行。

因此，Docker 与 LXC 的差距就如下如图所示：

LXC 时每个虚拟机只能是一个虚拟机，而 Docker 则是一系列的虚拟机。

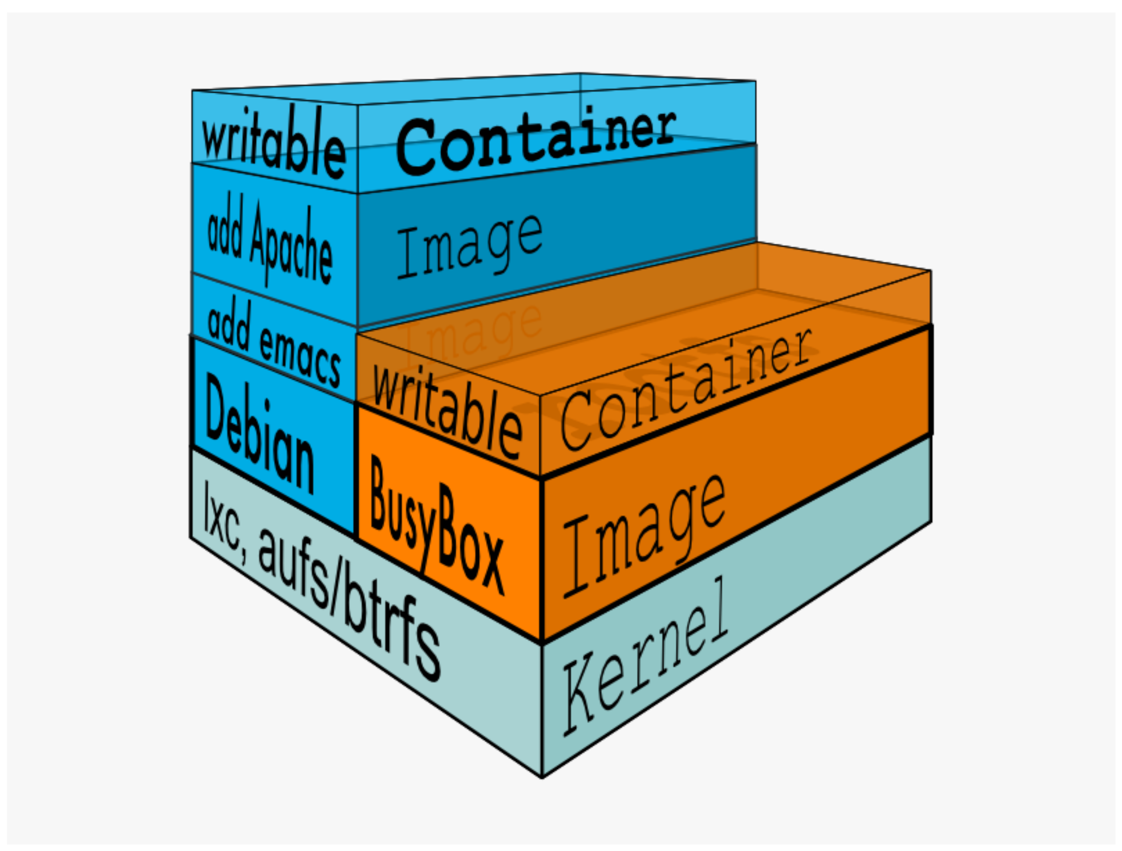


图 56: Docker Container

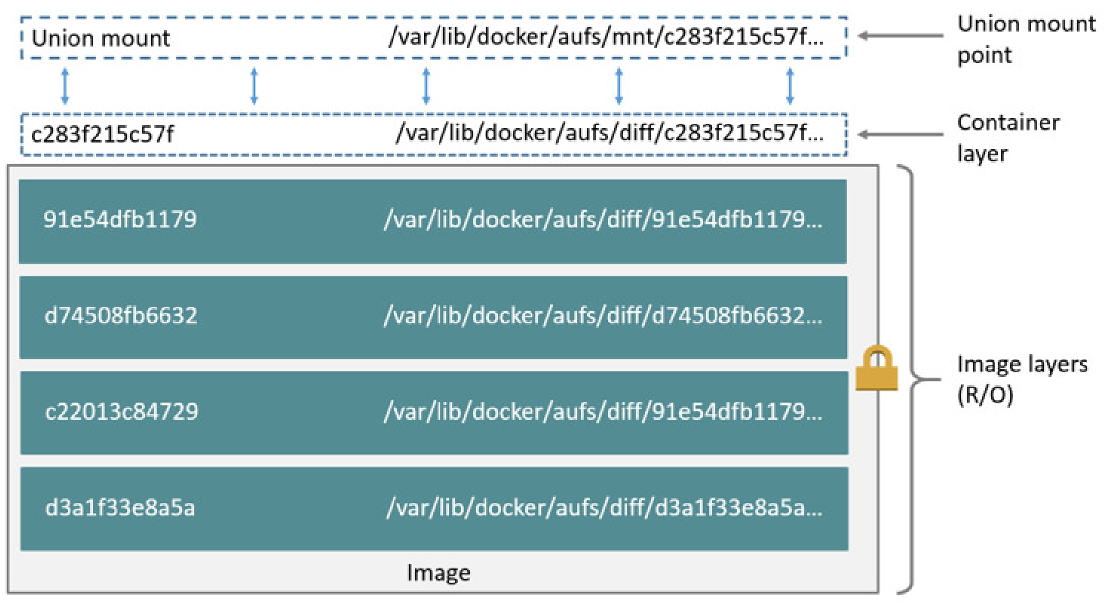


图 57: AUFS 层

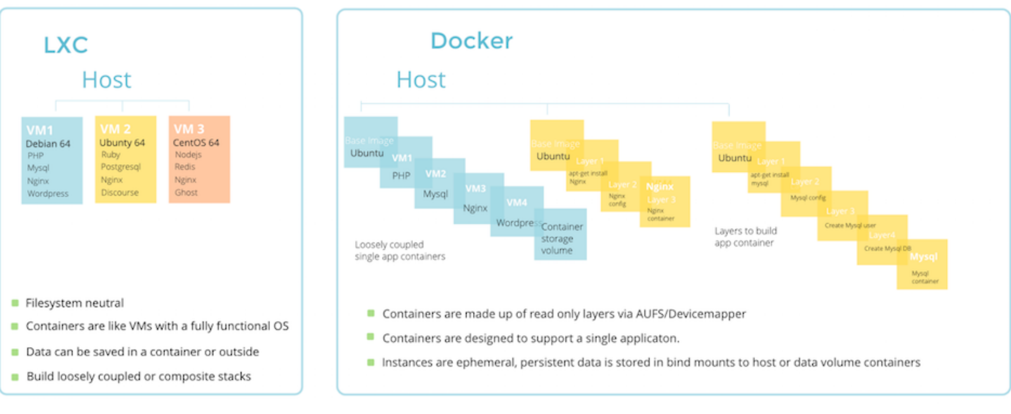


图 58: LXC 与 Docker

隔离底层：**Servlet** 容器

在上面的例子里我们已经隔离开了操作系统的因素，接着我们还需要解决操作系统、开发环境引起的差异。早期开发 Web 应用时，人们使用 CGI 技术，它可以让一个客户端，从网页浏览器向执行在网络服务器上的程序请求数据。并且 CGI 程序可以用任何脚本语言或者是完全独立编程语言实现，只要这个语言可以在这个系统上运行。而这样的脚本语言在多数情况下是依赖于系统环境的，特别是针对于 C++ 这一类的编译语言来说，在不同的操作系统中都需要重新编译。

而 Java 的 Servlet 则是另外一种有趣的存在，它是一种独立于平台和协议的服务器端的 Java 应用程序，可以生成动态的 Web 页面。

Tomcat 在开发 Java Web 应用的过程中，我们在开始环境使用 Jetty 来运行我们的服务，而在生产环境使用 Tomcat 来运行。他们都是 Servlet 容器，可以在其上面运行着同一个 Servlet 应用。Servlet 是指由 Java 编写的服务器端程序，它们是为响应 Web 应用程序上下文中的 HTTP 请求而设计的。它是应用服务器中位于组件和平台之间的接口集合。

Tomcat 服务器是一个免费的开放源代码的 Web 应用服务器。它运行时占用的系统资源小，扩展性好，支持负载平衡与邮件服务等开发应用系统常用的功能。除此，它还是一个 Servlet 和 JSP 容器，独立的 Servlet 容器是 Tomcat 的默认模式。其架构如下图所示：

Servlet 被部署在应用服务器中，并由容器来控制其生命周期。在运行时由 Web 服

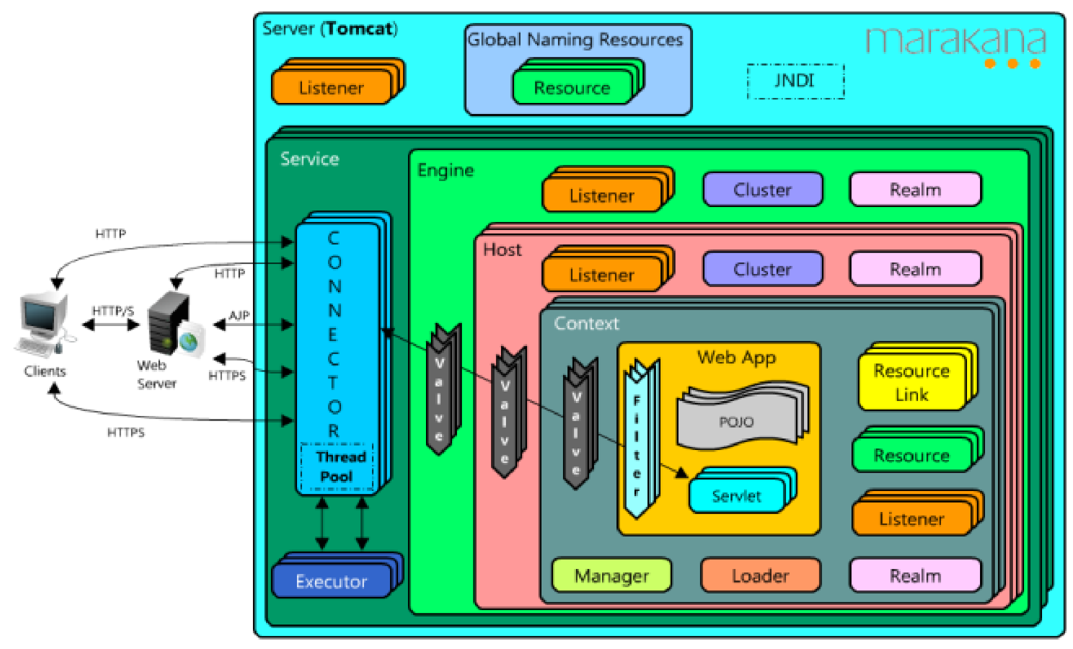


图 59: Tomcat 架构

务器软件处理一般请求，并把 Servlet 调用传递给 "容器" 来处理。并且 Tomcat 也会负责对一些静态资源的处理。

隔离依赖版本：虚拟环境

对于 Java 这一类的编译语言来说，不存在太多语言运行带来的问题。而对于动态语言来说就存在这样的问题，如 Ruby、Python、Node.js 等等，这一个问题主要集中于开发环境。当然如果你在一个服务器上运行着几个不同的应用来说，也会存在这样的问题。这一类的工具在 Python 里有 VirtualEnv，在 Ruby 里有 RVM、Rbenv，在 Node.js 里有 NVM。

下图是使用 VirtualEnv 时的不同几个应用的架构图：

如下所示，在不同的虚拟环境里，我们可以使用不同的依赖库。在这上面构建不同的应用，也可以使用不同的 Python 版本来构建系统。通常来说，这一类的工具主要用于本地的开发环境。

隔离运行环境：语言虚拟机

最后一个要介绍的可能就是更加抽象的，但是也是更加实用的一个，JVM 就是这方面的一个代表。在我们的编程生涯里，我们很容易就会遇到跨平台问题——即我们在我

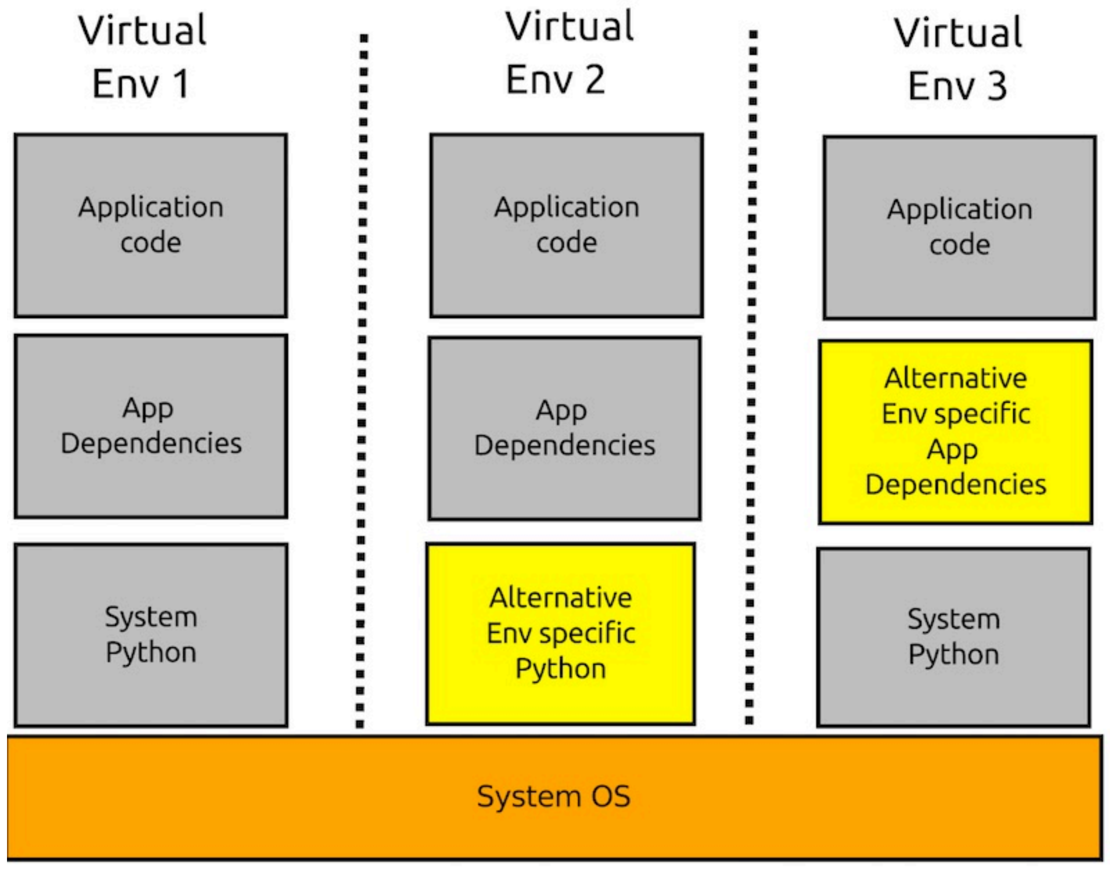


图 60: VirtualEnv

们的开发机器上开发的软件，在我们的产品环境的机器上就没有办法运行。特别是当我们使用 Mac OS 或者 Windows 机器上开发了我们的应用，然后我们需要在 Linux 系统上运行，就会遇到各种问题。并且当我们使用了一个需要重新编译的库时，这种问题就更加麻烦。

如下图所示的是 JVM 的架构示意图

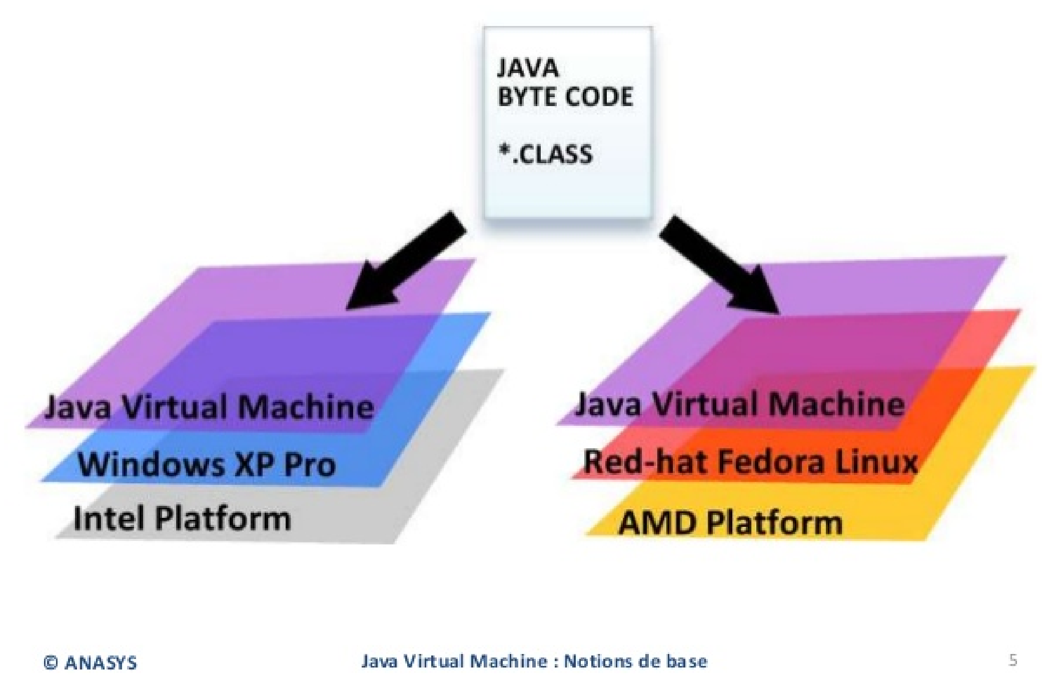


图 61: JVM

JVM 是一种用于计算设备的规范，它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。它可以实现 “编写一次，到处运行”。

换句话来说，它在底层实现了环境隔离，它屏蔽了与具体操作系统平台相关的信息，使得 Java 程序只需生成在 Java 虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。

基于此，只要其他编程语言的编译器能生成正确 Java bytecode 文件，这个语言也能实现在 JVM 上运行。如下图所示的是基于 JVM 的 Jython 语言的架构图：

其底层是基于 JVM，而编写时则是用 Python 语言，并且他可以使用 Java 的模块来编程。

常见拥有同样架构的工具，还有 MySQL，如下图是所示的是 MySQL 的架构图：

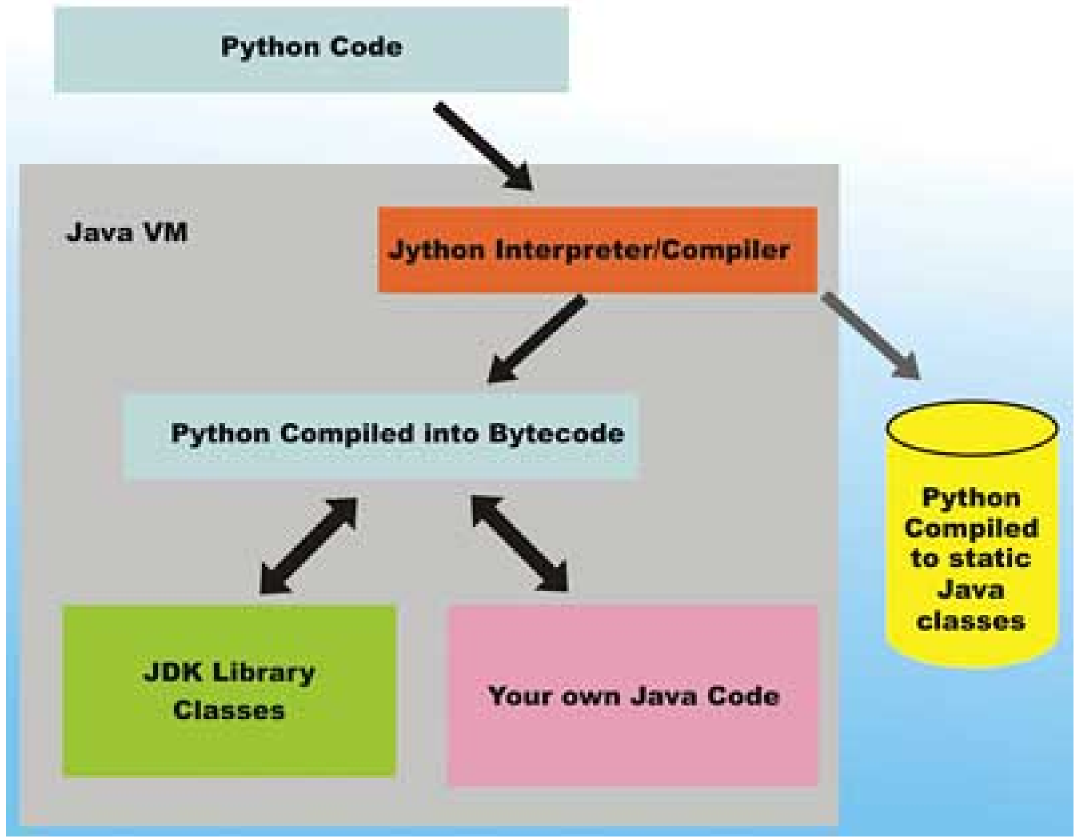


图 62: Jython

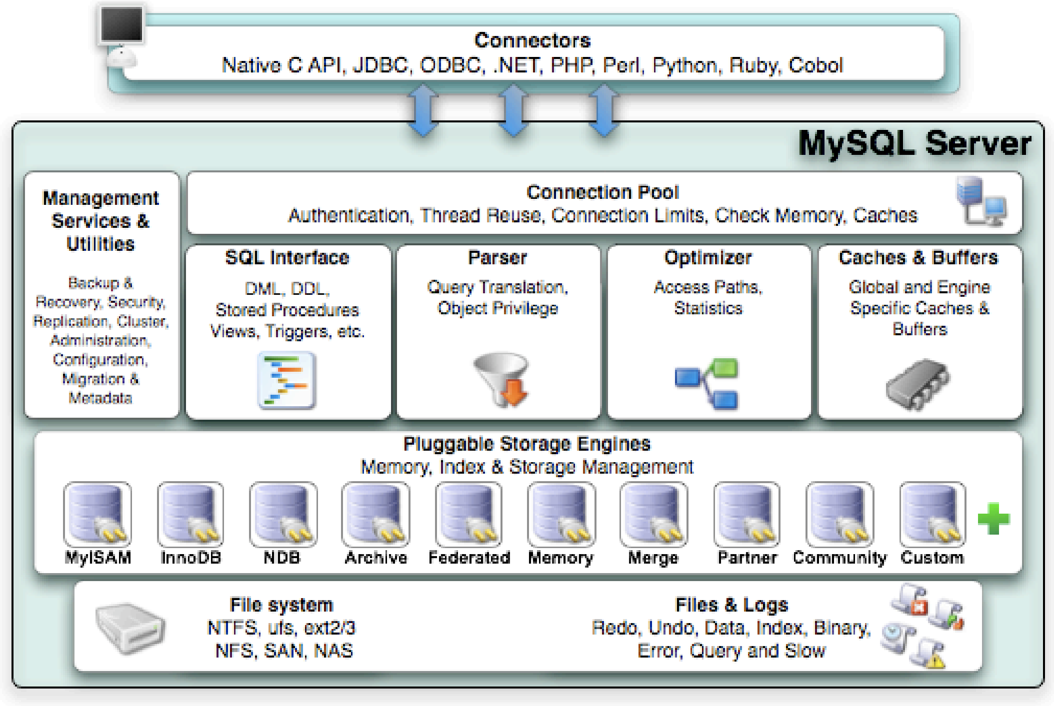


图 63: MySQL

MySQL 在最顶层提供了一个名为 SQL 的查询语言，这个查询语言只能用于查询数据库，然而它却是一种更高级的用法。它不像通用目的的语言那样目标范围涵盖一切软件问题，而是专门针对某一特定问题的计算机语言，即领域特定语言。

隔离语言：**DSL**

这是一门特别有意思也特别值得期待的技术，但是实现它并不是一件容易的事。

作为讨论隔离环境的一部分，我们只看外部 DSL。内部 DSL 与外部 DSL 最大的区别在于：外部 DSL 近似于创建了一种新的语法和语义的全新语言。如下图所示是两中 DSL 的一种对比：

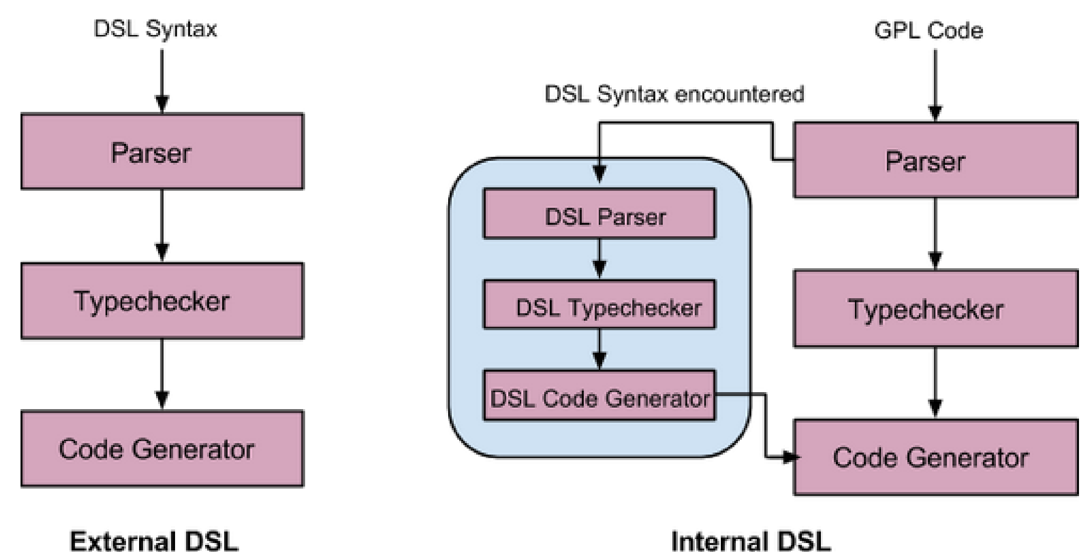


图 64: 内部 DSL 和外部 DSL

在这样的外部 DSL 里，我们有自己的语法、自己的解析器、类型检测器等等。最简单且最常用的 DSL 就是 Markdown，如下图所示：

如果我们可以将我们的业务逻辑写成 DSL，那么我们就不需要担心底层语言的变动过多地影响原有的业务逻辑。换句话说，这相当于创建了我们自己的语言隔离环境，我们不需要思考用何种语言来实用我们的业务。

LNMP 架构

> LNMP 是一个基于 CentOS/Debian 编写的 Nginx、PHP、MySQL、phpMyAdmin、eAccelerator 一键安装包。可以在 VPS、独立主机上轻松的安装 LNMP 生产环境。

图 65: Markdown

由于在前面我们已经介绍过了数据库和编程语言，这里我们就只介绍 LN 两项

GNU/Linux

GNU 工程创始于一九八四年，旨在开发一个完整 GNU 系统。GNU 这个名字是“GNU's Not Unix!”的递归首字母缩写词。“GNU”的发音为 g'noo，只有一个音节，发音很像“grew”，但需要把其中的 r 音替换为 n 音。类 Unix 操作系统是由一系列应用程序、系统库和开发工具构成的软件集合, 并加上用于资源分配和硬件管理的内核。

Linux 是一种自由和开放源码的类 UNIX 操作系统内核。目前存在着许多不同的 Linux 发行版，可安装在各种各样的电脑硬件设备，从手机、平板电脑、路由器和影音游戏控制台，到桌上型电脑，大型电脑和超级电脑。Linux 是一个领先的操作系统内核，世界上运算最快的 **10** 台超级电脑运行的都是基于 **Linux** 内核的操作系统。

Linux 操作系统也是自由软件和开放源代码发展中最著名的例子。只要遵循 GNU 通用公共许可证, 任何人和机构都可以自由地使用 Linux 的所有底层源代码，也可以自由地修改和再发布。严格来讲，**Linux** 这个词本身只表示 **Linux** 内核，但在实际上人们已经习惯了用 **Linux** 来形容整个基于 **Linux** 内核，并且使用 **GNU** 工程各种工具和数据库的操作系统（也被称为 **GNU/Linux**）。通常情况下，Linux 被打包成供桌上型电脑和服务器使用的 Linux 发行版本。一些流行的主流 Linux 发行版本，包括 Debian（及其衍生版本 Ubuntu），Fedora 和 openSUSE 等。Linux 得名于电脑业余爱好者 Linus

Torvalds。

HTTP 服务器

> Web 服务器一般指网站服务器，是指驻留于因特网上某种类型计算机的程序，可以向浏览器等 Web 客户端提供文档，也可以放置网站文件，让全世界浏览；可以放置数据文件，让全世界下载。

目前最主流的三个 Web 服务器是 Apache、Nginx、IIS。

Apache Apache 是世界使用排名第一的 Web 服务器软件。它可以运行在几乎所有广泛使用的计算机平台上，由于其跨平台和安全性被广泛使用，是最流行的 Web 服务器端软件之一。它快速、可靠并且可通过简单的 API 扩充，将 Perl/Python 等解释器编译到服务器中。

Nginx Nginx 是一款轻量级的 Web 服务器/反向代理服务器及电子邮件（IMAP/POP3）代理服务器，并在一个 BSD-like 协议下发行。由俄罗斯的程序设计师 Igor Sysoev 所开发，供俄国大型的入口网站及搜索引擎 Rambler（俄文：Рамблер）使用。其特点是占有内存少，并发能力强，事实上 Nginx 的并发能力确实在同类型的网页服务器中表现较好，中国大陆使用 Nginx 网站用户有：百度、新浪、网易、腾讯等。

IIS Internet Information Services（IIS，互联网信息服务），是由微软公司提供的基于运行 Microsoft Windows 的互联网基本服务。最初是 Windows NT 版本的可选包，随后内置在 Windows 2000、Windows XP Professional 和 Windows Server 2003 一起发行，但在 Windows XP Home 版本上并没有 IIS。

代理服务器

> 代理服务器（Proxy Server）是一种重要的服务器安全功能，它的工作主要在开放系统互联 (OSI) 模型的会话层，从而起到防火墙的作用。代理服务器大多被用来连接 INTERNET（国际互联网）和 Local Area Network（局域网）。

Web 缓存

Web 缓存是显著提高 Web 站点的性能最有效的方法之一。主要有:

- 数据库端缓存
- 应用层缓存
- 前端缓存
- 客户端缓存

不同的缓存类型适用于不同的环境下使用。

数据库端缓存

这个可以用以“空间换时间”来说。比如建一个表来存储另外一个表某个类型的数据的总条数，在每次更新数据的时候同时更新数据表和统计条数的表。在需要获取某个类型的数据的条数的时候，就不需要 select count 去查询，直接查询统计表就可以了，这样可以提高查询的速度和数据库的性能。

应用层缓存

应用层缓存这块跟开发人员关系最大，也是平时经常接触的。

- 缓存数据库的查询结果, 减少数据的压力。这个在大型网站是必须做的。
- 缓存磁盘文件的数据。比如常用的数据可以放到内存，不用每次都去读取磁盘，特别是密集计算的程序，比如中文分词的词库。
- 缓存某个耗时的计算操作，比如数据统计。

应用层缓存的架构也可以分几种:

- 嵌入式，也就是缓存和应用在同一个机器。比如单机的文件缓存，java 中用 hashMap 来缓存数据等等。这种缓存速度快，没有网络消耗。
- 分布式缓存，把缓存的数据独立到不同的机器，通过网络来请求数据，比如常用的 memcache 就是这一类。

分布式缓存一般可以分为几种:

- 按应用切分数据到不同的缓存服务器，这是一种比较简单和实用的方式。
- 按照某种规则（hash, 路由等等）把数据存储到不同的缓存服务器
- 代理模式，应用在获取数据的时候都由代理透明的处理，缓存机制有代理服务器来处理

前端缓存

我们这里说的前端缓存可以理解为一般使用的 cdn 技术，利用 squid 等做前端缓冲技术，主要还是针对静态文件类型，比如图片，css,js,html 等静态文件。

客户端缓存

浏览器端的缓存，可以让用户请求一次之后，下一次不在从服务器端请求数据，直接从本地缓存读取，可以减轻服务器负担也可以加快用户的访问速度。

HTML5 离线缓存

application cahce 是将大部分图片资源、js、css 等静态资源放在 manifest 文件配置中。当页面打开时通过 manifest 文件来读取本地文件或是请求服务器文件。

离线访问对基于网络的应用而言越来越重要。虽然所有浏览器都有缓存机制，但它们并不可靠，也不一定总能起到预期的作用。HTML5 使用 ApplicationCache 接口可以解决由离线带来的部分难题。前提是你需要访问的 Web 页面至少被在线访问过一次。

可配置

让我们写的 Web 应用可配置是一项很有挑战性，也很实用的技能。

起先，我们在本地开发的时候为本地创建了一套环境，也创建了本地的配置。接着我们需要将我们的包部署到测试环境，也生成了测试环境的相应配置。这其中如果有其他的环境，我们也需要创建相应的环境。最后，我们还需要为产品环境创建全新的配置。

下图是 Druapl 框架的部署流:

在不同的环境下，他们使用不同的 Content。这些 Content 的内容不仅仅可以是一些系统相当的配置，也可以是一些不同环境下的 UI 等等。而在这其中也会涉及到一些比较复杂的知识，下面只是做一些简单的介绍。

环境配置

最常见的例子就是我们需要在不同的环境有不同的配置。大原则就是我们不能直接使用产品的环境测试，因此我们就需要为不同的环境配置不同的数据库：

- 开发环境。即开发者用于开发的环境，大部分的数据都是由我们自己注入的，在开发的过程中我们也会添加一些数据。

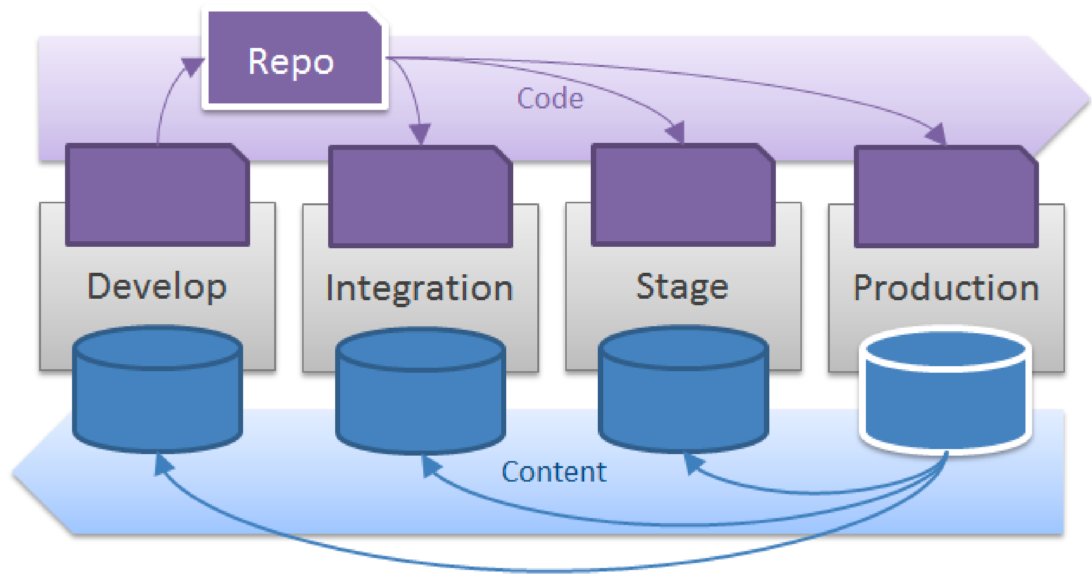


图 66: Drupal Deployment Flow

- 集成测试环境/测试环境。和开发环境一样，这些数据也是由我们注入的，而这些数据主要是为于测试目的。当我们的应用出现 Bug 的时候，我们可能就需要添加新的测试及其测试数据。
- 模拟环境（Stageing）。在软件最终发布前，开发或者设计人员对软件进行调整后可以及时预览改变的测试环境，这个环境更接近于产品最终发布后的运行环境。因此，这个环境的数据一般来说就是产品环境的一些旧数据——可能是几个月前，几年前的数据。
- 产品环境。即线上环境，都是真实的用户数据。

因此从理论上来说，我们就需要 4~5 个不同的数据库配置。而这些不同的数据库配置并不代表着他们使用的是相同的数据库。我们可以在本地环境使用 SQLite，而在我们的产品环境使用 MySQL。不过，最好的情况是我们应该使用同一个配置。这样当出现问题的时候，我们也很容易排查、

而除了数据库配置之外，我们还有一些其他配置。因此针对于不同的环境的配置最好独立地写在不同的文件里。并且这些配置最好可以以文件名来区分，如针对于开发环境，就是 `dev.config.js`，针对于测试环境就是 `test.config.js`。

因此，为了实现不同的环境使用不同的配置，我们就需要有一个变更控制。如果我们只有相应的配置，而没有对应的运行机制那就有问题了。

运行机制

当我们的应用程序在服务器上运行得好好的时候，我们可能就不想因为修改配置而去重启机器，这时候我们就需要配置热加载。即我们修改配置后，不需要重启服务即可以使用新的配置。对应的还有一种，便是我们需要重启机器才能实现配置。

无论是哪种方式都需要修改配置来实现。而在我们使用的过程中热加载可能需要消耗一些系统资源，因为我们的系统需要不断地读取配置的状态并对其进行判断。并且如果我们的应用运行在多个机器上的时候，我们可能需要一个个的上支个性。而如果我们是冷启动的话，我们就可以考虑使用自动部署的方式来完成。

对应的，我们也需要在我们的代码中实现判断这些配置的逻辑。

功能开关

当我们上线了我们的新功能的时候，这时候如果有个 Bug，那么我们是下线么？要知道这个版本里面包含了很多的 Bug 修复。如果在这个设计这个新功能的时候，我们有一个可配置和 Toogle，那么我们就不需要下线了。只需要切的这个 toggle，就可以解决问题了。

对于有多套环境的开发来说，如果我们针对不同的环境都有不同的配置，那么这个灵活的开发会帮助我们更好的开发。

Feature Toggle 它是一种允许控制线上功能开启或者关闭的方式，通常会采取配置文件的方式来控制。其过程如下图所示：

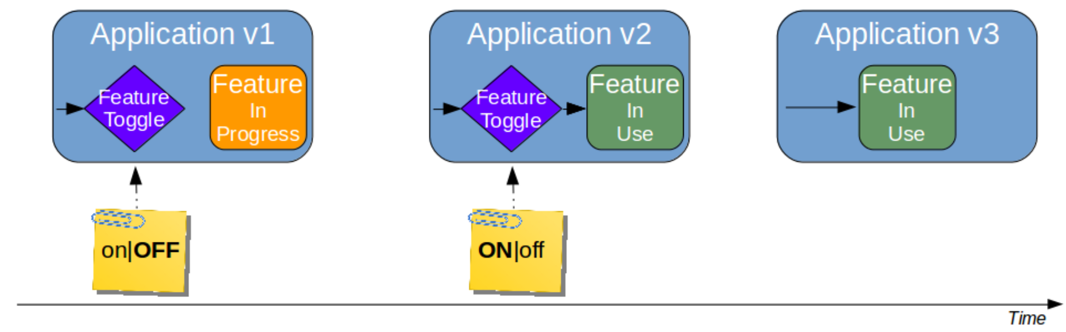


图 67: Feature Toggle

当我们需要 A 功能的时候，我们就只需要把 A 功能的开关打开。当我们需要 B 功能，而不需要 A 功能的时候，我们就可以把相应的功能关掉。像在 Java 里的 Spring 框架，就可以用 PropertyPlaceHolder 来做相似的事。使用 bean 文件创建一个 properties

```
<util:properties id="myProps" location="WEB-INF/config/prop.properties"/>
```

然后向注入这个值:

```
@Value("#{myProps['message']}")
```

我们就可以直接判断这个值是否是真，从而显示这个内容。

```
<spring:eval expression="@myProps.message" var="messageToggle"/>

<c:if test="${messageToggle eq true}">
    message
</c:if>
```

这是一种很实用，而且很有趣的技术。

参考书籍：《配置管理最佳实践》

自动化部署

优化我们开发流程有一个很重要的步骤就是：让部署自动化。通过部署自动化，我们可以大大缩减我们的开发周期，加快软件交付流程。下图是一个自动化部署的流程图:

从下图中我们可以得到下面的五个步骤:

- 获取源码
- 获取依赖
- 构建软件包
- 生成/上传安装包
- 目标平台安装/配置

这个过程可能和之前的 Web 项目构建过程差不多，然而却多了好几步。

在前面的章节里，我们已经使用了版本管理系统来管理我们的源码。因此，在这里对于获取源码的介绍就比较简单了——我们只需要在我们的 CI（持续集成）服务器上使用 `git clone` 这一类的方法来获取我们的源码即可。

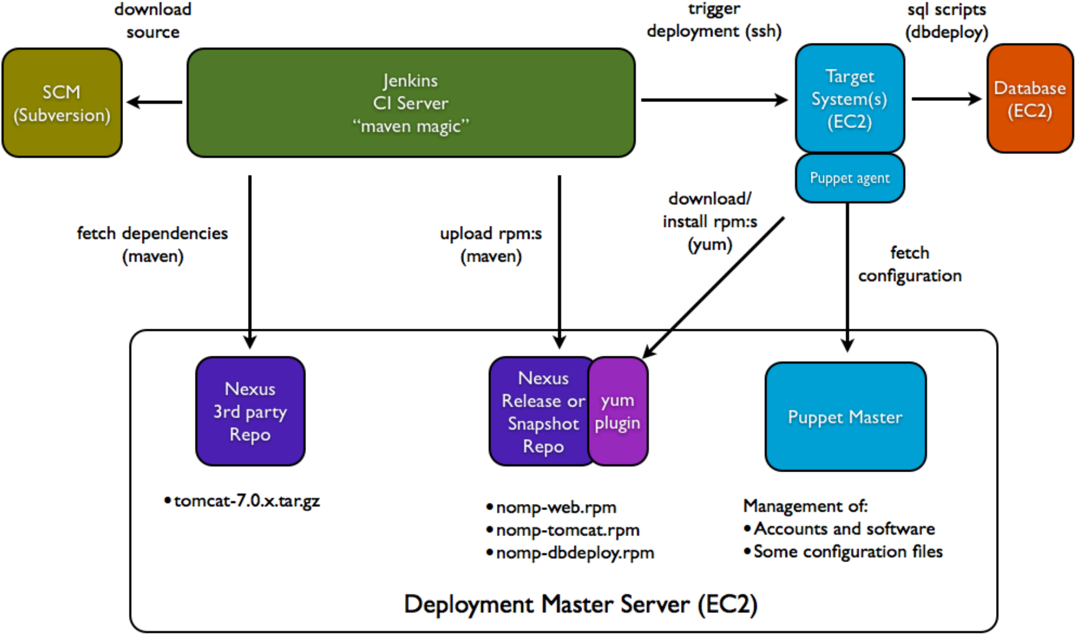


图 68: 自动化部署

依赖与包仓库

获取完源码后，我们就需要开始下载软件包依赖。无论是 Python、Ruby、Java，还是 JavaScript 都需要这样的一个过程。软件开发已经从大教堂式的开发走向了集市——开源软件改变了这一切。

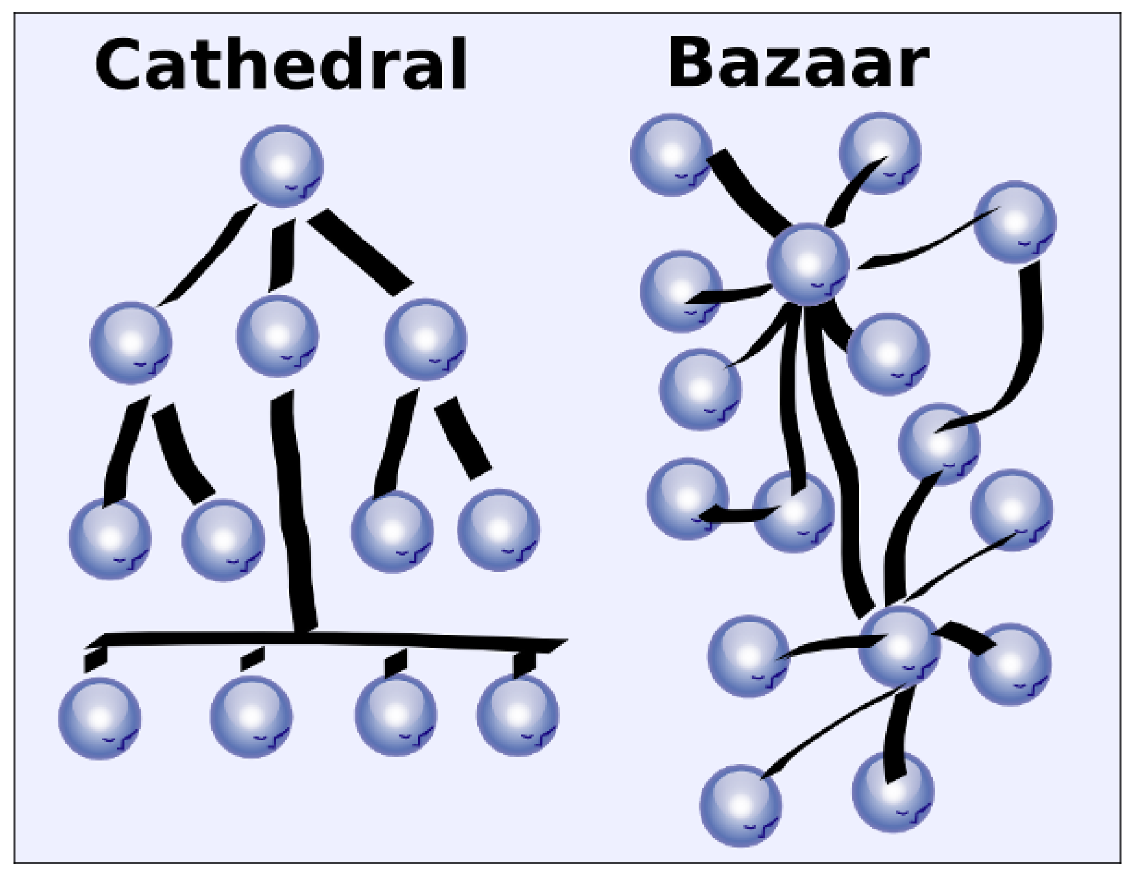


图 69: 大教堂与集市

过去我们需要大系统的内部构建我们使用的依赖，现在我们更多地借助于外部的库来实现这些功能。这也意味着，如果在这一个节点里出现了意外——软件被删除，那么这个系统将陷入瘫痪的状态。如之前在 NPM 圈发生了“一个 17 行的模块引发的血案”——即 left-pad 工具模块被作者从 NPM 上撤下，所有直接或者间接依赖这个模块的 NPM 的软件包都挂掉了。因为我们依赖于公有的包服务，所以系统便严重依赖于外部条件。

这时候一种简单、有效的方案就是搭建自己的包服务。如使用 Java 技术栈的项目，就会使用 Nexus 搭建自己的 Maven 私有服务。我们的软件依赖包将会依赖于我们自己的服务，此时会产生的主要问题可能就是：我们的软件包不是最新的。但是对于追求稳定的项目来说，这个并不是必须的需求，反而这也是一个优势。

构建软件包

在一些编译型语言里，在我们运行包测试后，我们将会得到一个软件包。如 Jar 包，它是 Java 中所特有一种压缩文档。Jar 包无法直接安装使用，虽然我们可以直接运行这个 Jar 包，但是我们需要通过一些手段将这个 Jar 包拷贝到我们的服务器上，然后运行。在特定的时候，我们还需要修改配置才能完成我们的工作。

因此，使用 RPM 或者 DEB 包会是一种更好的选择。RPM 全称是 Red Hat Package Manager（Red Hat 包管理器），它工作于 Red Hat Linux 以及其它 Linux 和 UNIX 系统，可被任何人使用。如下图是 RPM 包的构建过程：

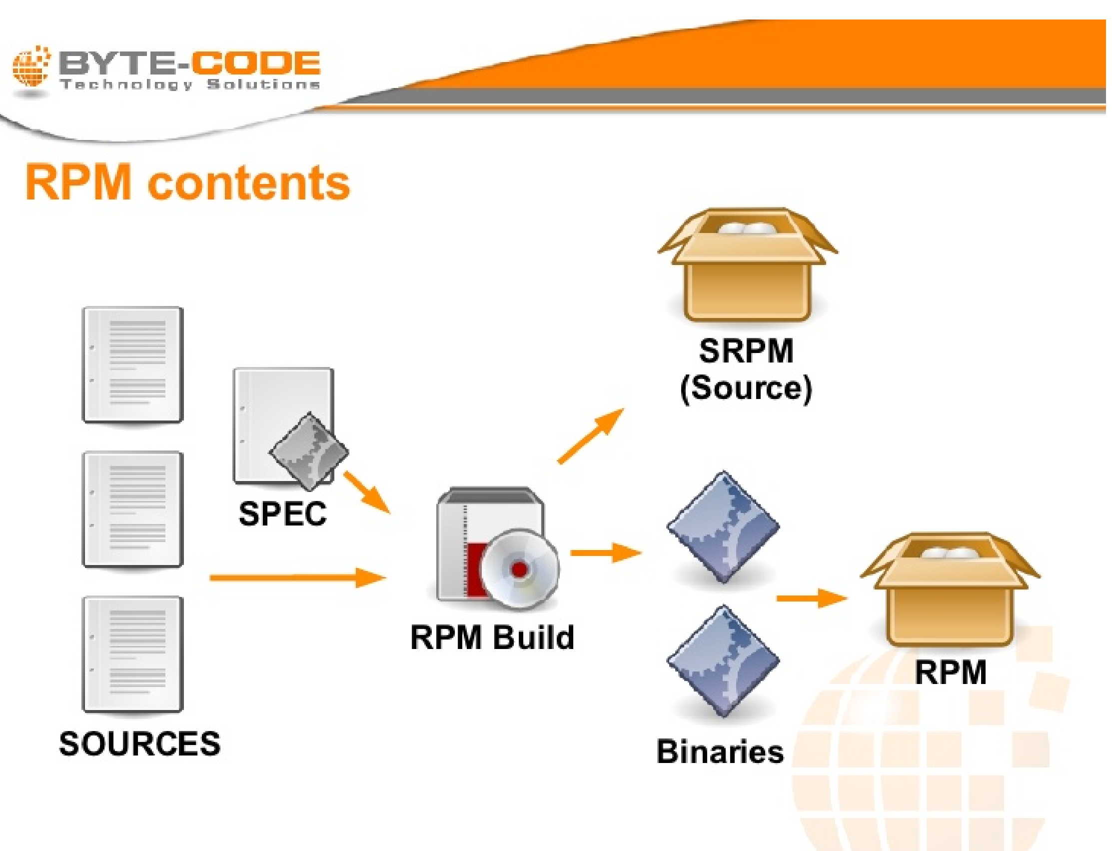


图 70: RPM Build Process

要构建一个标准的 RPM 包，我们需要创建.spec 文件，这个文件包含软件打包的全部信息——如包的 Summary、Name、Version、Copyright、Vendor 等等。在产生完这一个配置文件事，执行 rpmbuild 命令，系统会按照步骤生成目标 RPM 包。

上传和安装软件包

生成对应的软件包后，我们就可以将其上传到 Koji 上，它是 Fedora 社区的编译系统。如下图所示：

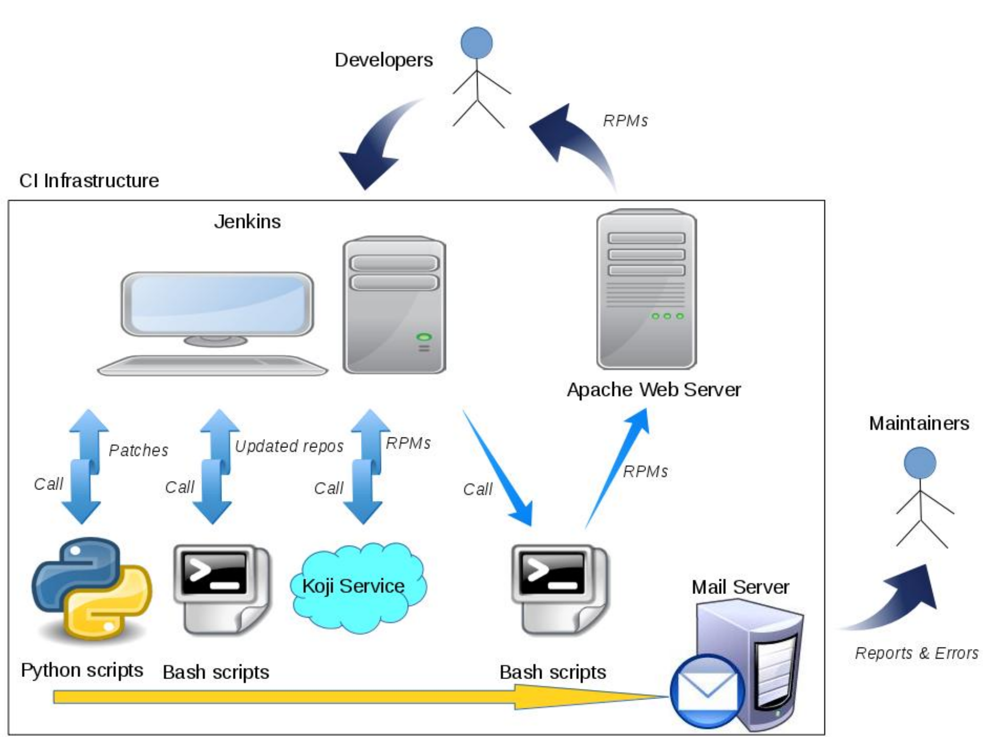


图 71: RPM Build Process

如果我们已经对我们的所有目标操作系统配置过，即配置好了软件源，那么我们就可以直接在我们的服务器上使用包管理工具安装，如 `yum install`。

数据分析

有时候，对于我们的决定只要有一点点的数据支持就够了。也就是一点点的变化，可能就决定了我们产品的好坏。我们可能会因此而作出一些些改变，这些改变可能会让我们打败巨头。

这一点和 Growth 的构建过程也很相像，在最开始的时候我只是想制定一个成长路线。而后，我发现这好像是一个不错的 idea，我就开始去构建这个 idea。于是它变成了 Growth，这时候我需要依靠什么去分析用户喜欢的功能呢？我没有那么多的精力去和那么多的人沟通，也不能去和那么多的人沟通。

我只能借助 Google Analytics 来收集用户的数据。从这些数据里去学习一些东西，而这些就会变成一个新的想法。新的想法在适当的时候就会变成一个产品，接着我们就开始收集用户数据，然后循环。

构建-衡量-学习

构建-衡量-学习是在《精益创业》中的一个核心概念，这结合了客户开发、敏捷软件开发方法和精益生产实践。他们是非常重要的一个循环：

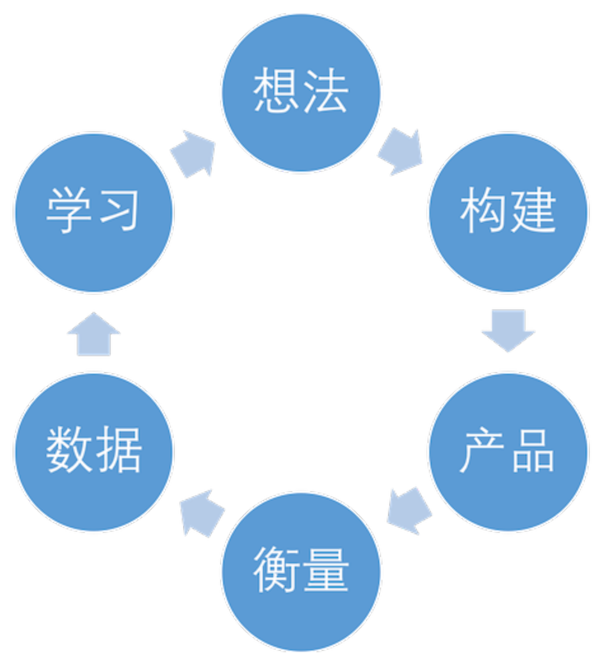


图 72: 数据分析过程

这一过程不仅仅可以改进我们的产品，也可以用于初创企业。它并不是独立的一个环节，实现上它应该是一整个环节：我们根据我们的想法去创建我们的产品，在使用产品的过程中我们收集一些数据，再依据这些数据来改进我们的产品。

想法-构建

想法实际上便是解决一个痛点的解决方案。如果你和我一样也经常记录自己的想法，你会发现每个月里，你总会跳出一个又一个的想法。正如，我在那篇《如何去管理你的Idea》中说的一样：

> 我们经常说的是我们缺少一个 Idea。过去我也一直觉得我缺少一些 Idea，今天发现并非如此，我们只是缺少记录的手段。

我们并不缺少 Idea，我们只是一直没有去记录。随着时间的增长，我发现我的GitHub 上的 Idea 墙 (ideas) 一直在不断地增加。以至于，我有一个新的 Idea 就是整理这个 Idea 墙。

而作为一个程序员，我们本身就可以具备构建一个系统的能力，只是对于大多数人来说需要多加的练习。有意思的一点是，这里的构建系统与一般的构建系统有一点不太一样的是，我们需要快速地构建出一个 MVP 产品。MVP 简单地来说，就是最小可用的产品。如下图的右边所示：

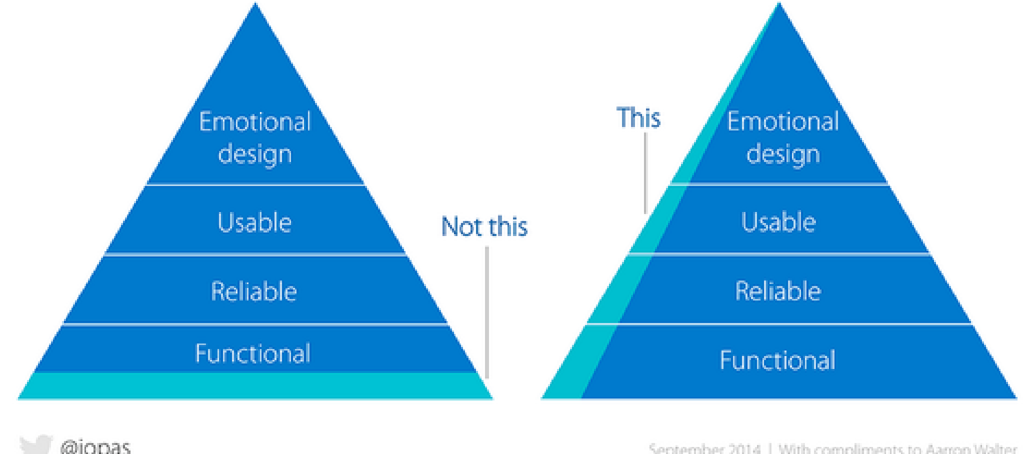


图 73: MVP

在每一层级上都实现一定的功能，使得这个系统可用，而非构建一个非常完整的系

统。随后，我们就可以寻找一些种子用户来改进我们的产品。

产品-衡量

按照上面的步骤，到了这里应该就是客户开发。而如《精益客开发》一书所说，客户开发可以分成五个步骤：

- 形成假设。即我们觉得用
- 找到可以交谈的潜在客户
- 提出恰当的问题
- 从答案中找到有用的信息
- 弄明白现阶段需要构建什么样的产品来保持下一个学习循环

在整个过程中，我们其实就是在了解我们的客户是谁，以及他们的需求。并且在这个过程中，我们可以为我们的开发确认出清晰的假设，我们可以一点点地打造出用户喜爱的产品。

数据-学习

当我们收集到一定的用户数据，如网站、应用的数据，我们就开始去分析数据。如《精益创业》所说，在分析数据之前，我们需要确定我们的增长模型，即：

- 黏着式增长引擎——其重点是让用户成为回头客，即让客户持续使用我们的产品。这就意味着，我们在分析数据和学习的过程中，我们要侧重于关注流失率和使用频率。
- 病毒式增长引擎——其只做一件事：让名声传播出去。即通过用户间的不断传播来扩散产品，我们需要考虑所谓的病毒式传播系数，还有用户之间的特定行为。
- 付费式增长引擎——赚钱是识别商业模式是否可持续的指标。

针对不同的增长引擎有不同的学习过程，如媒体网站，我们通过不同的方式来导入流量，这些流量最终会有一些会转化成价值。这些价值会以不同的形式出现，如订阅率、在线参与度、广告营收等等。

而从这些数据中学习就需要一些特殊的技巧，详情请见下面的参考书籍。

参考书籍：

- 《精益数据分析》

- 《精益客户开发》
- 《精益创业》

数据分析

数据分析是一个很有意思的过程，我们可以简单地将这个过程分成四个步骤：

- 识别需求
- 收集数据
- 分析数据
- 展示数据

值得注意的是：在分析数据的过程中，需要不同的人员来参与，需要跨域多个领域的知识点——分析、设计、开发、商业和研究等领域。因此，在这样的领域里，回归敏捷也是一种不错的选择（源于：《敏捷数据科学》）：

- 通才高于专长
- 小团队高于大团队
- 使用高阶工具和平台：云计算、分布式系统、PaaS
- 持续、迭代地分享工作成果，即使这些工作未完成

识别需求

在我们开始分析数据之前，我们需要明确一下，我们的问题是什么？即，我们到底要干嘛，我们想要的内容是什么。

> 识别信息需求是确保数据分析过程有效性的首要条件，可以为收集数据、分析数据提供清晰的目标。

当我们想要提到我们的网站在不同的地区的速度时，我们就需要去探索我们的用户主要是在哪些地区。即，现在这是我们的需求。我们已经有了这样的一个明确的目标，下面要做起来就很轻松了。

收集数据

那么现在新的问题来了，我们的数据要从哪里来？

对于大部分的网站来说，都会有访问日志。但是这些访问日志只能显示某个 IP 进入了某个页面，并不人详细地介绍这个用户在这个页面待了多久，做了什么事。这时候，这些数据就需要依赖于类似于 Google Analytics 这样的工具来统计网站的流量。还有类似于 New Relic 这样的工具来统计用户的一些行为。

在一些以科学研究为目的的数据收集中，我们可以从一些公开的数据中获取这些资料。

而在一些特殊的情况里，我们就需要通过爬虫来完成这样的工作。

分析数据

现在，我们终于可以真正的去分析数据了——我的意思是，我们要开始写代码了。从海量的数据中过滤出我们想要的数据，并通过算法来对其进行分析。

一般来说，我们都利用现有的工具来完成大部分的工作。要使用哪一类工具，取决于我们如要分析的数据的数量级了。如果只是一般的数量级，我们可以考虑用 R 语言、Python、Octave 等单机工具来完成。如果是大量的数据，那么我们就需要考虑用 Hadoop、Spark 来完成这个级别的工作。

而一般来说，这个过程可能是要经过一系列的工具才能完成。如在之前我在分析我的博客的日志时 (1G 左右)，我用 Hadoop + Apache Pig + Jython 来将日志中的 IP 转换为 GEO 信息，再将 GEO 信息存储到 ElasticSearch 中。随后，我们就可以用 AMap、leaflet 这一类 GEO 库将这些点放置到地图上。

展示数据

现在，终于来到我最喜欢的环节了，也是最有意思，但是却又最难的环节。

我们过滤后我们的数据，得到我们想要的内容后，我们就要去考虑如何可视化我们的数据。在我熟悉的 Web GIS 领域里，我可以可视化出我过滤后的那些数据。但是对于我不熟悉的领域，要可视化这些数据不是一件容易的事。在少数情况下，我们才能使用现有的工具完成需求，多数情况下，我们也需要写相当的代码才能将数据最后可视化出来。

而在以什么形式来展示我们的数据时，又是一个问题。如一般的数据结果，我们到底是使用柱形图、条形图、折线图和面积图中的哪一种？这依赖于我们有一些 UX 方面的经验。

参考来源: 精益数据分析。

用户数据分析：**Google Analytics**

Google Analytics 是一个非常赞的分析工具，而且它不仅仅可以用于 Web 应用，也可以用于移动应用。

受众群体

如下图是 Growth 应用最近两星期的数据：

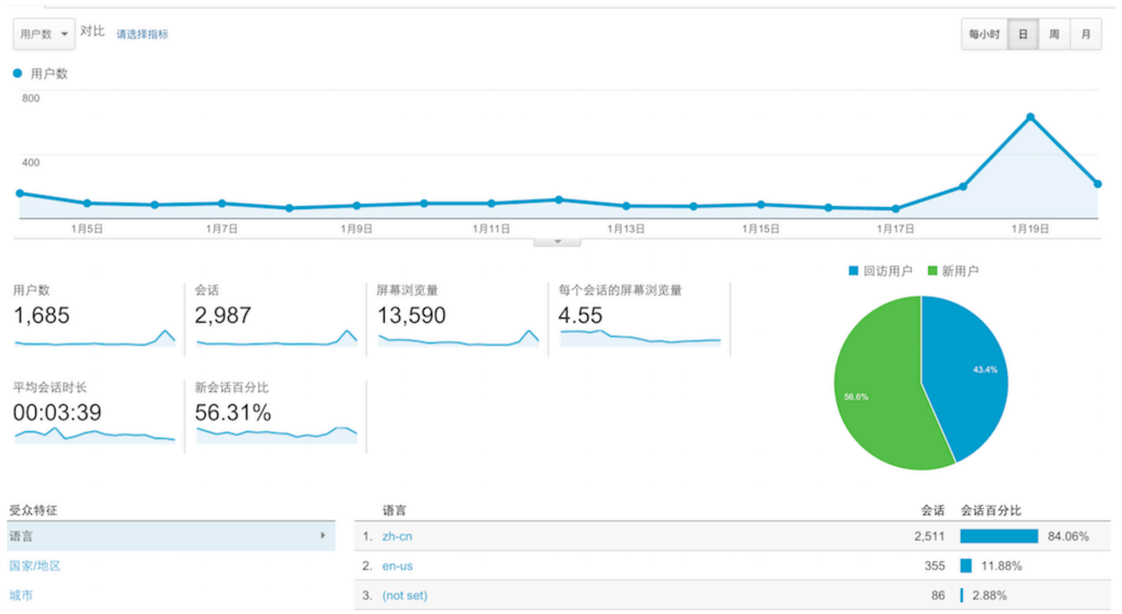


图 74: Growth GA

这是 Google Analytics 中的“受众群体”的概览，在这个视图中：

1. 折线图就是每天的用户数。
2. 下面会有用户数、会话、屏幕浏览量等等的一些信息。
3. 右角的饼图则是回访问用户和新用户的对比。
4. 最下方便是受众的信息——国家、版本等等。

从图中，我们可以读取一些重要的信息，如用户的停留时间、主要面向的用户等等。在浏览器版本会有：

1. 浏览器与操作系统
2. 移动设备

这样的重要数据，如下表是我网站 20160104-20160120 的访问数据：

浏览器	会话	新会话百分比
Chrome	5048	75.99%
Firefox	694	78.39%
Safari	666	78.68%
Internet Explorer	284	87.68%
Safari (in-app)	92	86.96%
Android Browser	72	87.50%
Edge	63	79.37%
Maxthon	51	68.63%
UC Browser	41	80.49%
Opera	34	64.71%

可以从上表中看到访问我网站的用户中，IE 只占很小的一部分——大概 4%，而 Chrome + Safari + Firefox 加起来则近 90%。这也意味着，我可以完全不考虑 IE 用户的感受。

类似于这样的数据在我们决定我们对某个浏览器的支持情况时会非常有帮助的。也会加快我们的开发，我们可以工作于主要的浏览器上。

流量获取

除此，不得不说的一点就是流量获取，如下图所示是我博客的热门渠道：

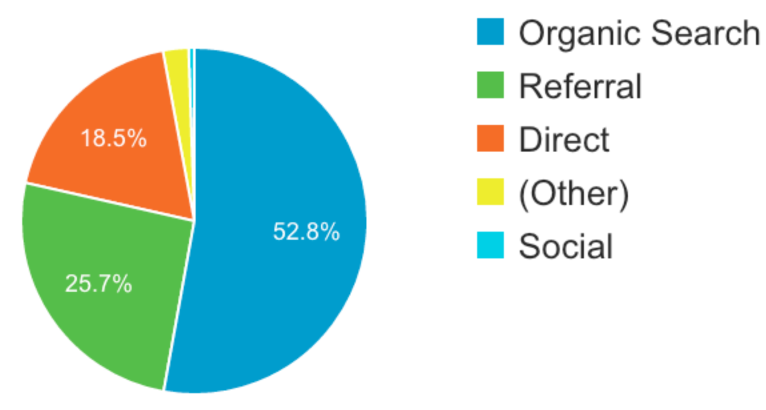


图 75: Phodal.com Traffic

可以直接得到一个不错的结论是我的博客的主要流量来源是搜索引擎，再细细一看数据：

来源/媒介	会话
baidu / organic	2031
google / organic	1314
(direct) / (none)	1311
bing / organic	349
github.com / referral	281

主要流量来源就是 Baidu 和 Google，看来国人还是用百度比较多。那我们就可以针对 SEO 进行更多的优化：

1. 加快访问速度
2. 更表意的 URL
3. 更好的标题
4. 更好的内容

等等等。

除此，我们可以分析用户的行为，如他们访问的主要网站、URL 等等。

移动应用

除此，我们还可以使用它来分析移动应用，不过这受限于 Google 在国内的访问程度。如下图是 GA 收到的应用的使用数据：

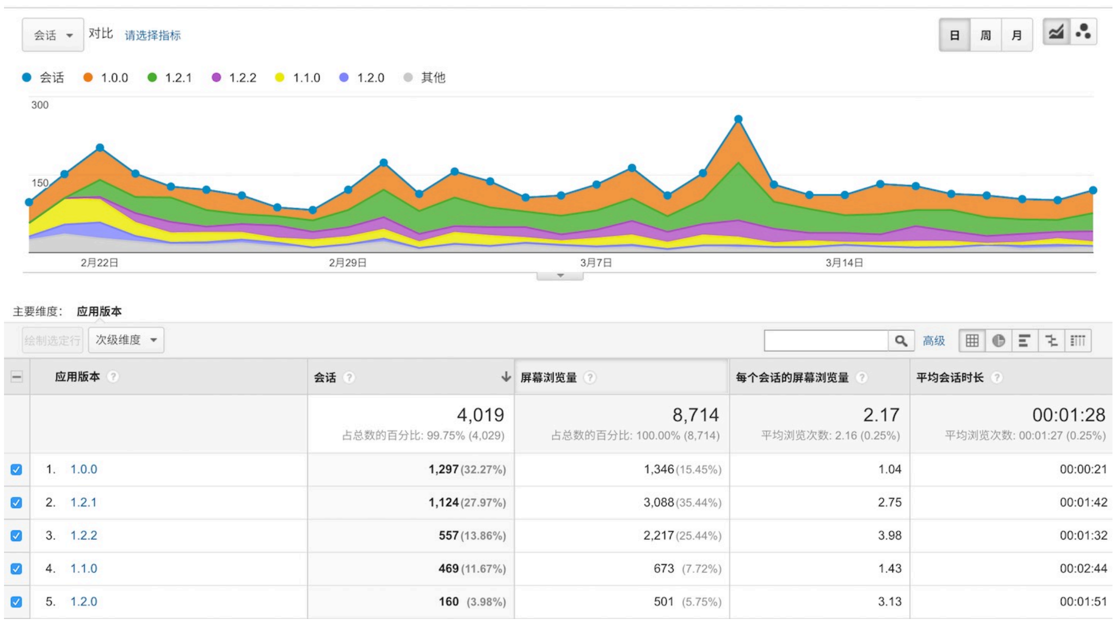
我们也可以从上面看到 APP 的安装来源等等。

网站性能

网站性能直接影响到了网站的响应时间、吞吐量等等，也是运维、开发一系列技术的体现。

网站性能监测

网站性能监测



	应用版本	会话	屏幕浏览量	每个会话的屏幕浏览量	平均会话时长
		4,019 占总数的百分比: 99.75% (4,029)	8,714 占总数的百分比: 100.00% (8,714)	2.17 平均浏览次数: 2.16 (0.25%)	00:01:28 平均浏览次数: 00:01:27 (0.25%)
1.	1.0.0	1,297 (32.27%)	1,346 (15.45%)	1.04	00:00:21
2.	1.2.1	1,124 (27.97%)	3,088 (35.44%)	2.75	00:01:42
3.	1.2.2	557 (13.86%)	2,217 (25.44%)	3.98	00:01:32
4.	1.1.0	469 (11.67%)	673 (7.72%)	1.43	00:02:44
5.	1.2.0	160 (3.98%)	501 (5.75%)	3.13	00:01:51

图 76: Growth 应用数据

网站可用性是网站性能监测的重要指标之一，表示在一段时间内，网站处于“正常状态”的机率。

- DNS 解析
- 内存、硬盘等等
- 网页打开速度

应用性能指数

Apdex 联盟，一个由众多网络分析技术公司和测量工业组成的联盟组织，它们联合起来开发了“应用性能指数”即“Apdex”(Application Performance Index)，用一句话来概括，Apdex 是用户对应用性能满意度的量化值。它提供了一个统一的测量和报告用户体验的方法，第一次把最终用户的体验和应用性能联系在了一起。

任务响应时间定义为：当用户操作（鼠标点击、输入、回车）开始到系统（客户机、网络、服务器）响应从而用户能继续这个过程所经过的时间。这些等待时间定义了应用程序的“响应度”。该指数是基于应用程序响应度的三个方面：

- 满意: 用户充分工作。这就是目标时间（T 秒），即在此时间里用户的工作没有因应用程序的响应时间而受阻，如 3 秒。
- 容忍: 用户感觉到响应滞后，响应时间大于 T，但能继续这个过程，如 3 ～ 12 秒。

- 挫折: 响应时间大于 F 秒的性能是不能接受的，用户可能放弃这个过程。F 等于 T×4，在本例子中为 12 秒。

网站性能

针对网站性能优化领域，网上已经有相当多的总结，这里只罗列一些常见（我用过）的策略。

减少 **HTTP** 请求　从网上查找的情况分类来看，有下面的一些情况：

- 合并 JavaScript 和 CSS。只是这种方式需要好好评估，因为合并过多的 JavaScript，可能会导致 JavaScript 文件过大。一个大的文件将增加 Load 时间，导致不好的用户体验。
- CSS Sprites。即将一个页面涉及到的所有零星图片都包含到一张大图中去。值得注意的是，像 Logo 这一类文件将不要加到里面去了。
- 拆分初始化负载。将页面加载时需要的一堆 JavaScript 文件，分成两部分：渲染页面所必需的和其他的。页面初始化时，只加载必须的，其余的等会加载。
- 划分主域。将资源划分的请求划分到几个不同的域上，来加速资源请求。

对于我这样的懒人来说，我使用 Google 出品的 PageSpeed。它主要的功能是针对前端页面而进行服务器端的优化，对前端设计人员来说，可以省去优化 css、js 以及图片的过程。它可以对 CSS 和 JavaScript 压缩、合并、级联、内联，生成一个新的 Script 和 CSS 文件。还有图像优化：剥离元数据、动态调整，重新压缩，如针对 Chrome 浏览器生成 WebP 文件。还可以推迟图像和 JavaScript 加载。

页面内部优化　HTML 页面内的优化的目的便是：尽快渲染出页面。常见的优化策略便是：

- 将 CSS 放在顶部，即早点渲染出页面及其样式。
- 将 JavaScript 放在底部。如果有后台渲染机制，那么就应该将 JS 放到页面底部来加速页面加载。如果是单页面应用，那么这个 JS 就应该在页面顶部。
- 压缩 HTML。在我们写模板的过程中，一些判断可能会导致页面有过多的空格。压缩这些 HTML，可以稍微提高一下页面速度。

这里的大部分内容都应该通过修改代码来完成。

启用缓存　前面的缓存一节里，我们说过了一些缓存的策略，我们再稍微提一下。

- 后台优化，如数据库端缓存
- 启用页面缓存，即应用层缓存

减少下载量　简单地来说，就是减少对服务器的请求：

- 使用 CDN
- 使用外部 JavaScript 和 CSS
- 缓存：使用 gzip 压缩、添加 Expires 头、配置 ETag、使 AjaX 可缓存

网络连接上的优化　主要就是对域名到服务器进行优化，因此从方法上有：

- DNS 域名解析加速
- 减少 DNS 查找

SEO

> 这是一个老的，有些过时纸，但非常平易近人，甚至在我们中间的非白皮书的读者图标微笑什么每个程序员都应该知道的关于搜索引擎优化和他们绝对概念的解释更详细，我只提一笔带过。

搜索时发生什么了**?**

- 用户输入查询内容
- 查询处理以及分词技术
- 确定搜索意图及返回相关、新鲜的内容

为什么需要 **SEO?**

这是一个有趣的问题，答案总会来源于为网站带来更多的流量。

爬虫与索引

我们先看看来自谷歌的爬虫工作的一点内容

> 抓取是 Googlebot 发现新网页并更新这些网页以将网页添加到 Google 索引中的过程。

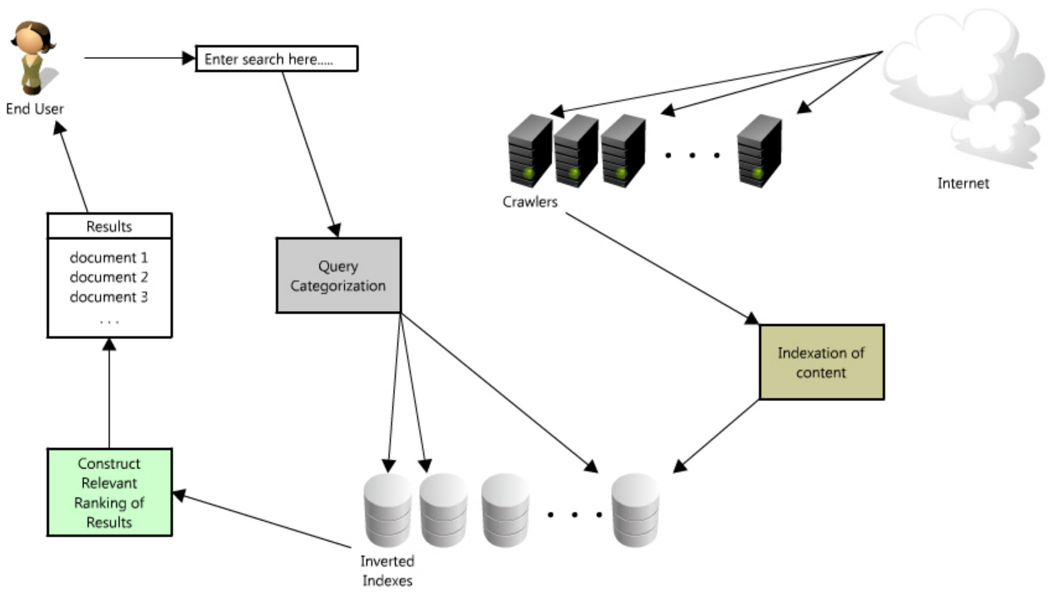


图 77: search-engine-arch

> 我们使用许多计算机来获取（或“抓取”）网站上的大量网页。执行获取任务的程序叫做 Googlebot（也被称为漫游器或信息采集软件）。Googlebot 使用算法来进行抓取：计算机程序会确定要抓取的网站、抓取频率以及从每个网站中获取的网页数量。
>
> Google 的抓取过程是根据网页网址的列表进行的，该列表是在之前进行的抓取过程中形成的，且随着网站管理员所提供的站点地图数据不断进行扩充。Googlebot 在访问每个网站时，会检测每个网页上的链接，并将这些链接添加到它要抓取的网页列表中。新建立的网站、对现有网站所进行的更改以及无效链接都会被记录下来，并用于更新 Google 索引。

也就是如原文所说:

> 谷歌的爬虫 (又或者说蛛蛛) 能够抓取你整个网站索引的所有页。

为什么谷歌上可以搜索整个互联网的内容？因为，他解析并存储了。而更有意思的是，他会为同样的内容建立一个索引或者说分类，按照一定的相关性，针对于某个关键词的内容。

PageRank 对于一个网站来说是相当重要的，只是这个相比也比较复杂。包括其他网站链接向你的网站，以及流量，当然还有域名等等。

什么样的网站需要 **SEO**?

下图是我的博客的流量来源

默认渠道分组	流量获取 会话
	9,057 占总数的百分比: 100.00% (9,057)
1. Referral	**4,110** (45.38%)
2. Organic Search	**3,333** (36.80%)
3. Direct	**1,594** (17.60%)
4. Social	**20** (0.22%)

图 78: What Site Need SEO

正常情况下除了像腾讯这类的 QQ 空间自我封闭的网站外都需要 SEO，或者不希望泄露一些用户隐私如 Facebook、人人等等

- 如果你和我的网站一样需要靠搜索带来流量
- 如果你只有很少的用户访问，却有很多的内容。
- 如果你是为一个公司、企业工作为以带来业务。
- 。。。

SEO 与编程的不同之处**?**

SEO 与编程的最大不同之处在于: 编程的核心是技术，**SEO** 的核心是内容。

内容才是 SEO 最重要的组成部分，这也就是腾讯复制不了的东西。

SEO 基础知识

确保网站是可以被索引的

一些常见的页面不能被访问的原因

- 隐藏在需要提交的表格中的链接

- 不能解析的 JavaScript 脚本中的链接
- Flash、Java 和其他插件中的链接
- PowerPoint 和 PDF 文件中的链接
- 指向被 meta Robtots 标签、rel="NoFollow" 和 robots.txt 屏蔽的页面的链接
- 页面上有上几百个链接
- frame(框架结构) 和 iframe 里的链接

对于现在的网站来还有下面的原因，通过来说是因为内容是动态生成的，而不是静态的

- 网站通过 WebSocket 的方法渲染内容
- 使用诸如 Mustache 之类的 JS 模板引擎

什么样的网页可以被索引

- 确保页面可以在没有 JavaScript 下能被渲染。对于现在 JavaScript 语言的使用越来越多的情况下，在使用 JS 模板引擎的时候也应该注意这样的问题。
- 在用户禁用了 JavaScript 的情况下，保证所有的链接和页面是可以访问的。
- 确保爬虫可以看到所有的内容。那些用 JS 动态加载出来的对于爬虫来说是不友好的
- 使用描述性的锚文本的网页
- 限制的页面上的链接数量。除去一些分类网站、导航网站之类有固定流量，要不容易被认为垃圾网站。
- 确保页面能被索引。有一指向它的 URL
- URL 应该遵循最佳实践。如 blog/how-to-driver 有更好的可读性

在正确的地方使用正确的关键词

- 把关键词放 URL 中
- 关键词应该是页面的标签
- 带有 H1 标签
- 图片文件名、ALT 属性带有关键词。
- 页面文字
- 加粗文字
- Descripiton 标签

内容

对于技术博客而言，内容才是最需要考虑的因素。

可以考虑一下这篇文章，虽然其主题是以 SEO 为主 用户体验与网站内容

不可忽略的一些因素是内容才是最优质的部分，没有内容一切 SEO 都是无意义的。

复制内容问题　一个以用户角度考虑的问题: 用户需要看到多元化的搜索结果

所以对于搜索引擎来说，复制带来的结果：

- 搜索引擎爬虫对每个网站都有设定的爬行预算，每一次爬行都只能爬行 trpgr 页面数
- 连向复制内容页面的链接也浪费了它们的链接权重。
- 没有一个搜索引擎详细解释他们的算法怎样选择显示页面的哪个版本。

于是上文说到的作者给了下面的这些建议:

> 避免从网上复制的内容（除非你有很多其他的内容汇总，以使它看起来不同 - 我们做头条，对我们的产品页面的新闻片段的方式）。这当然强烈适用于在自己的网站页面以及。内容重复可以混淆搜索引擎哪些页面是权威（它也可能会导致罚款，如果你只是复制粘贴别人的内容也行），然后你可以有你自己的网页互相竞争排名！
>
> 如果你必须有重复的内容，利用相对 = 规范，让搜索引擎知道哪个 URL 是一个他们应该被视为权威。但是，如果你的页面是另一个在网络上找到一个副本？那么开始想出一些策略来增加更多的文字和信息来区分你的网页，因为这样重复的内容是决不可能得到好的排名。

——待续。

保持更新　谷歌对于一个一直在更新的博客来说会有一个好的排名，当然只是相对的。

对于一个技术博客作者来说，一直更新的好处不仅可以让我们不断地学习更多的内容。也可以保持一个良好的习惯，而对于企业来说更是如此。如果我们每天去更新我们的博客，那么搜索引擎对于我们网站的收录也会变得越来越加频繁。那么，对于我们的排名及点击量来说也算是一个好事，当我们可以获得足够的排名靠前时，我们的 PR 值也在不断地提高。

更多内容可以参考:Google Fresh Factor

网站速度

> 谷歌曾表示在他们的算法页面加载速度问题，所以一定要确保你已经调整您的网站，都服从最佳做法，以使事情迅速

过去的一个月里，我试着提高自己的网站的速度，有一个相对好的速度，但是受限于域名解析速度以及 VPS。

网站速度分析与 traceroute

UX 与网站速度优化——博客速度优化小记

Nginx ngx_pagespeed nginx 前端优化模块编译

保持耐心

> 这是有道理的，如果你在需要的谷歌机器人抓取更新的页面，然后处理每一个页面，并更新与新内容对应的索引的时间因素。
>
> 而这可能是相当长一段时间，当你正在处理的内容 PB 级。

SEO 是一个长期的过程，很少有网站可以在短期内有一个很好的位置，除非是一个热门的网站，然而在它被发现之前也会一个过程。

链接　在某种意义上，这个是提高 PR 值，及网站流量的另外一个核心，除了内容以外的核心。

- 链接建设是 SEO 的基础部分。除非你有一个异常强大的品牌，不需要干什么就能吸引到链接。
- 链接建设永不停止。这是不间断营销网站的过程

关于链接的内容有太多，而且当前没有一个好的方法获取链接虽然在我的网站已经有了

Links to Your Site

Total links

`5,880`

> 同时寻求更多的链接是更有利更相关的链接可以帮助一样多。如果你有你的内容的分销合作伙伴，或者你建立一个小工具，或其他任何人都会把链接

> 回你的网站在网络上 - 你可以通过确保各个环节都有最佳的关键字锚文本大大提高链路的相关性。您还应该确保所有链接到您的网站指向你的主域（http://www.yourdomain.com ，像 http://widget.yourdomain.com 不是一个子域）。另外，你要尽可能多的联系，以包含适当的替代文字。你的想法。
>
> 另外，也许不太明显的方式，建立链接（或者至少流量）是使用社交媒体 - 所以设置你的 Facebook ，Twitter 和谷歌，每当你有新的链接一定要分享。这些通道也可以作为一个有效的渠道，推动更多的流量到您的网站。

由社交渠道带来的流量在现在已经越来越重要了，对于一些以内容为主导的网站，而且处于发展初期，可以迅速带来流量。一些更简单的办法就是交换链接，总之这个话题有些沉重，可能会带来一些负面的影响，如黑帽 SEO。。。。

参考来源:

《SEO 艺术》(The Art of SEO)

UX 入门

用户体验设计（英语：User Experience Design），是以用户为中心的一种设计手段，以用户需求为目标而进行的设计。设计过程注重以用户为中心，用户体验的概念从开发的最早期就开始进入整个流程，并贯穿始终。其目的就是保证：

- 对用户体验有正确的预估
- 认识用户的真实期望和目的
- 在功能核心还能够以低廉成本加以修改的时候对设计进行修正
- 保证功能核心同人机界面之间的协调工作，减少 BUG。

关于 UX 的定义我觉得在知乎上的回答似乎太简单了，于是在网上寻寻觅觅终于找到了一个比较接近于答案的回答。原文是在: Defining UX，这又是一篇不是翻译的翻译。

什么是 UX

用户体验设计（英语：User Experience Design），是以用户为中心的一种设计手段，以用户需求为目标而进行的设计。设计过程注重以用户为中心，用户体验的概念从开发的最早期就开始进入整个流程，并贯穿始终。其目的就是保证：

- 对用户体验有正确的预估

- 认识用户的真实期望和目的
- 在功能核心还能够以低廉成本加以修改的时候对设计进行修正
- 保证功能核心同人机界面之间的协调工作，减少 BUG。

UX 需要什么

从下图中我们可以看到一些 UX 所需要的知识体系：

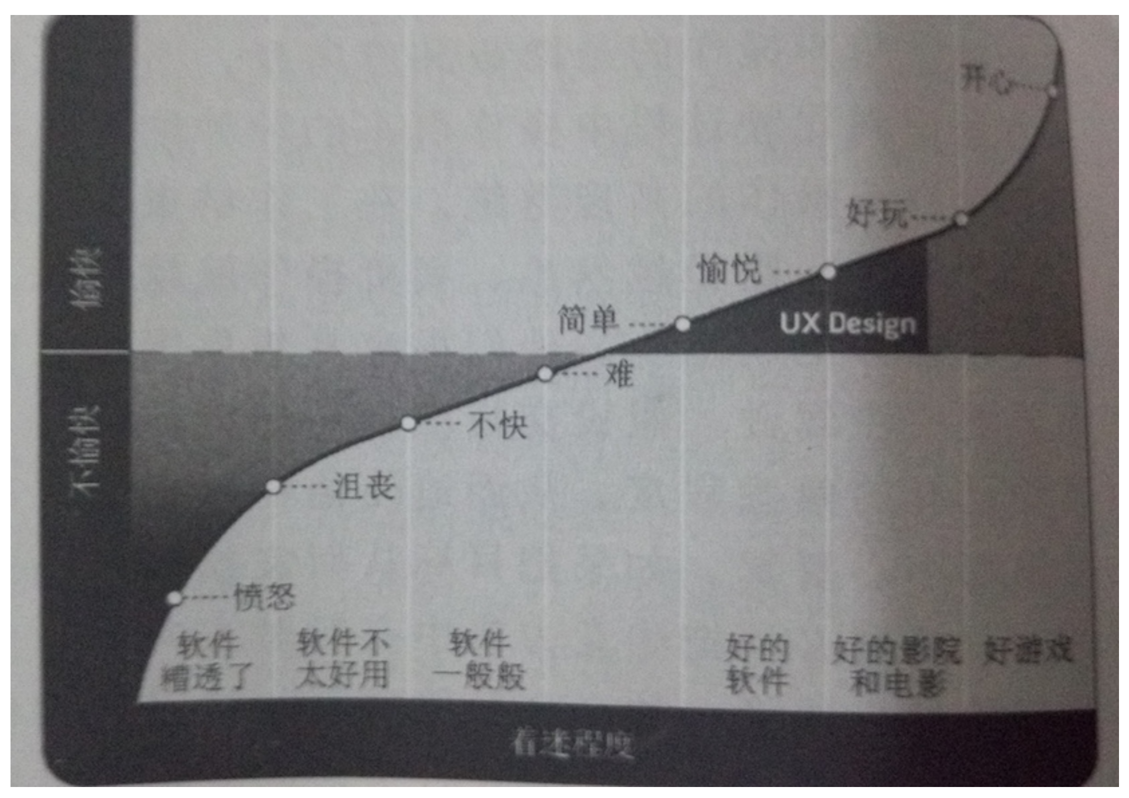


图 79: UX

即

- 信息构架
- 构架
- 工业设计
- 人为因素 (人因学)
- 声音设计 (网页设计中比较少)
- 人机交互
- 可视化设计
- 内容 (文字, 视频, 声音)

交互设计便是用户体验设计的重点。我们再来看看另外的这张图片

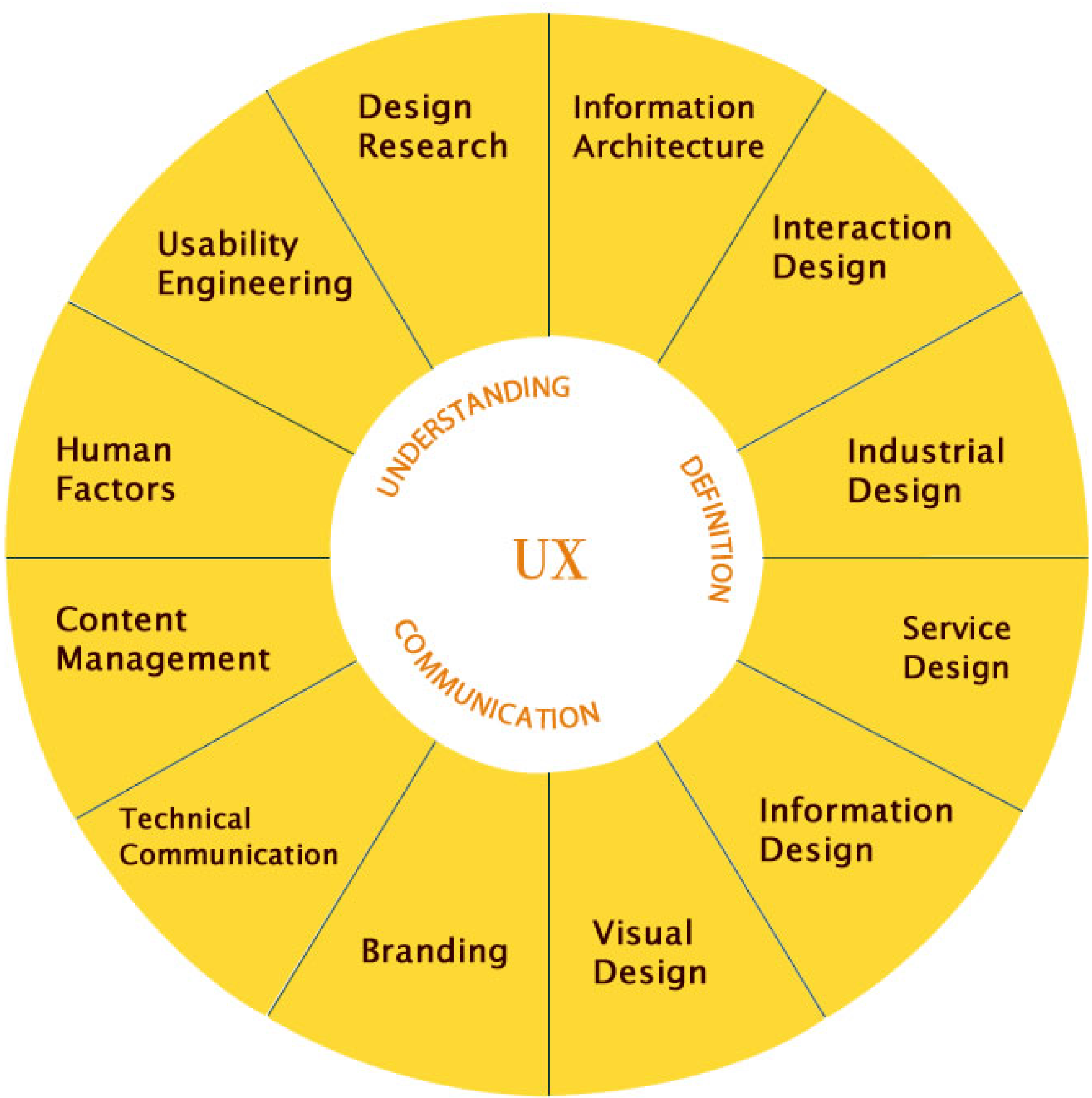


图 80: Fields Of User Experience Design

什么是简单?

一个好的软件应该是简单的，并且是令人愉快的。

在不同的 UX 书籍里，似乎就会说到【简约至上】。简单就是“单纯清楚、不复杂”。而这里的简单并不仅仅只是不复杂那么简单。对于一个软件来说，简单实际上是你一下子就能找到你想要的东西，如：

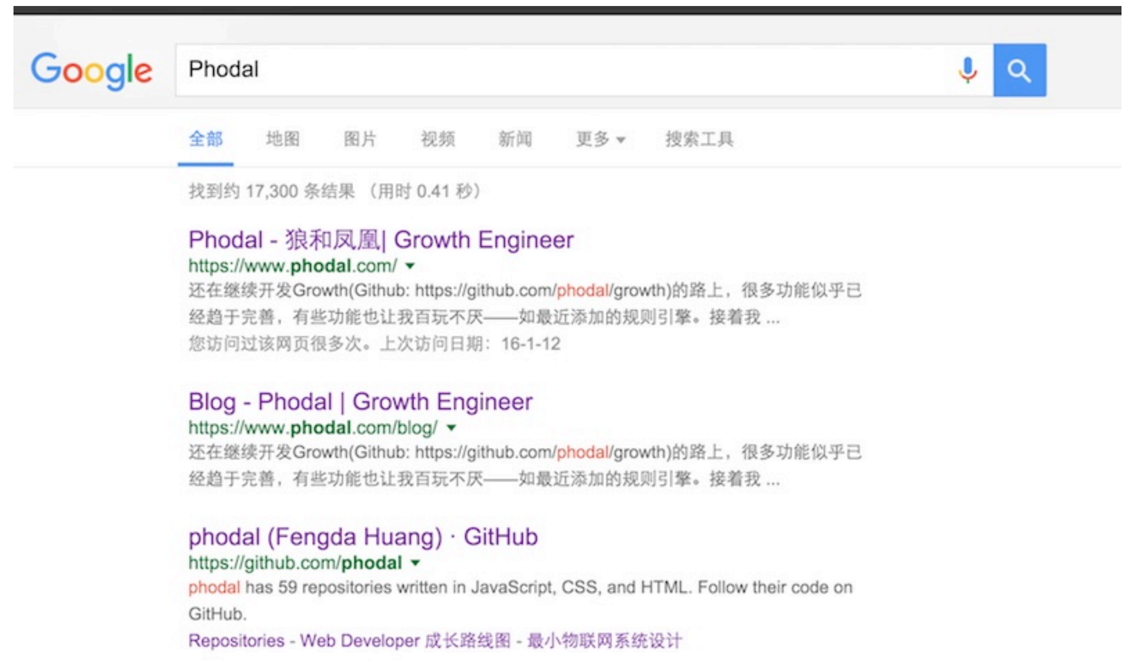


图 81: Search Phodal

而我们并不可能在一开始就得到这样的结果，这需要一个复杂的过程。而在这个过程开始之前，我们一定要知道的一点是：我们的用户到底需要什么?

如果我们无法知道这一点，而只是一直在假想客户需要什么，那么这会变成一条死路。

接着在找寻的过程中，发现了一个有意思的图，即精益画布：

首先，我们需要知道几个用户而存在的问题——即客户最需要解决的三个问题。并且针对三个问题提出对应的解决方案，这也就是产品的主要功能。

那么，这两点结合起来对于用户来说就是简单——这个软件能解决客户存在的主要问题。

如果我们可以完成这部分功能的话，那么这就算得上是一个有用的软件。

【问题】 客户最需要解决的三个问题	【解决方案】 产品最重要的三个功能	【独特卖点】 用一句简明扼要但引人注目的话阐述为什么你的产品与众不同，值得购买	【门槛优势】 无法被对手轻易复制或者买去的竞争优势	【客户群体分类】 目标客户
	【关键指标】 应该考核哪些东西		【渠道】 如何找到客户	
【成本分析】 争取客户所需花费 销售产品所需花费 网站架设费用 人力资源费用等		【收入分析】 盈利模式 客户终身价值 收入 毛利		

图 82: Lean

进阶

而实际上有用则是位于用户体验的最底层，如下图所示：

这时候就需要尽量往可用靠拢。怎样对两者进行一个简单的划分?

下图就是实用的软件：

而下图就便好一点了：

要达到可用的级别，并不是一件困难的事：遵循现有软件的模式。

换句话说，这时候你需要的是一本 **Cookbook**。这本 **Cookbook** 上面就会列出诸多现有的设计模式，只需要参考便使用就差不多了。

同样的，我便尝试用《移动应用 UI 设计模式》一本书对我原有的软件进行了一些设计，发现它真的可以达到这样的地步。

而这也类似于在编程中的设计模式，遵循模式可以创造出不错的软件。

用户体验要素

尽管对于这方面没有非常好的认识，但是还是让我们来看看我们现在可以到哪种程度。如在一开始所说的，我们需要满足用户的需求，这就是我们的目标：

而在最上面的视觉设计则需要更专业的人来设计。

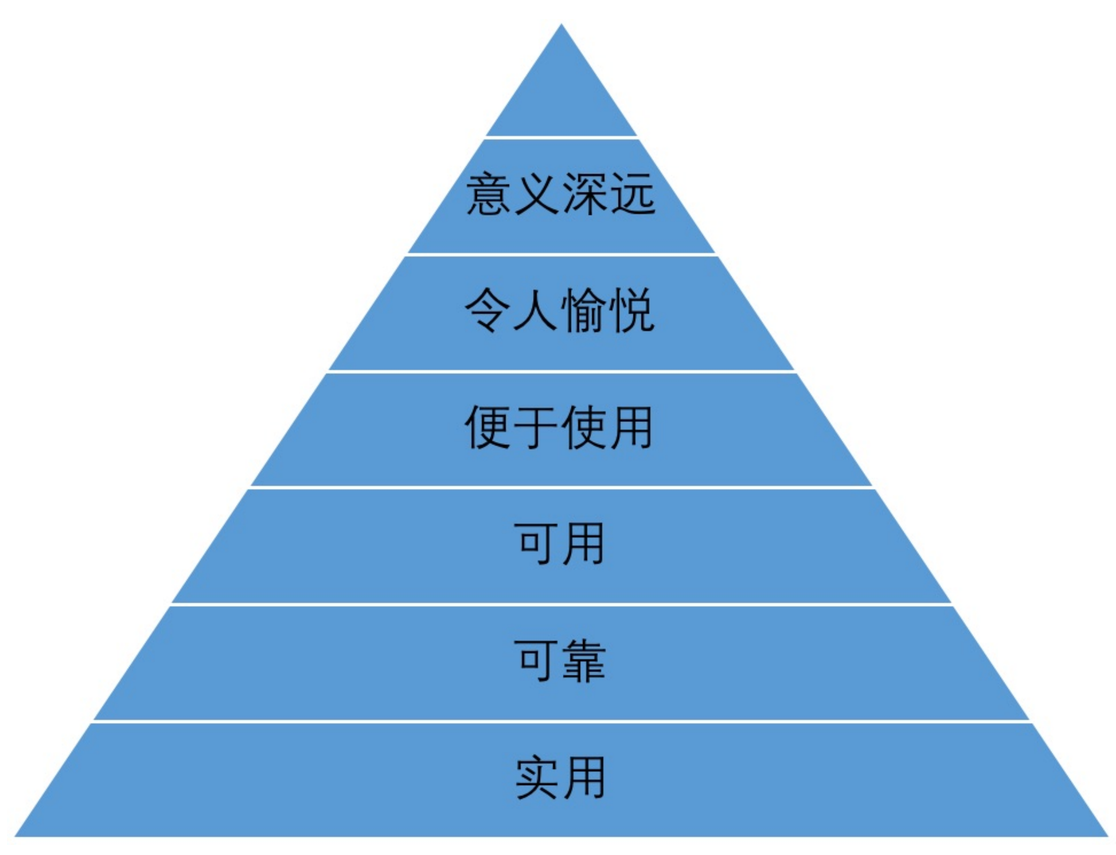


图 83: UX

图 84: IE Alert

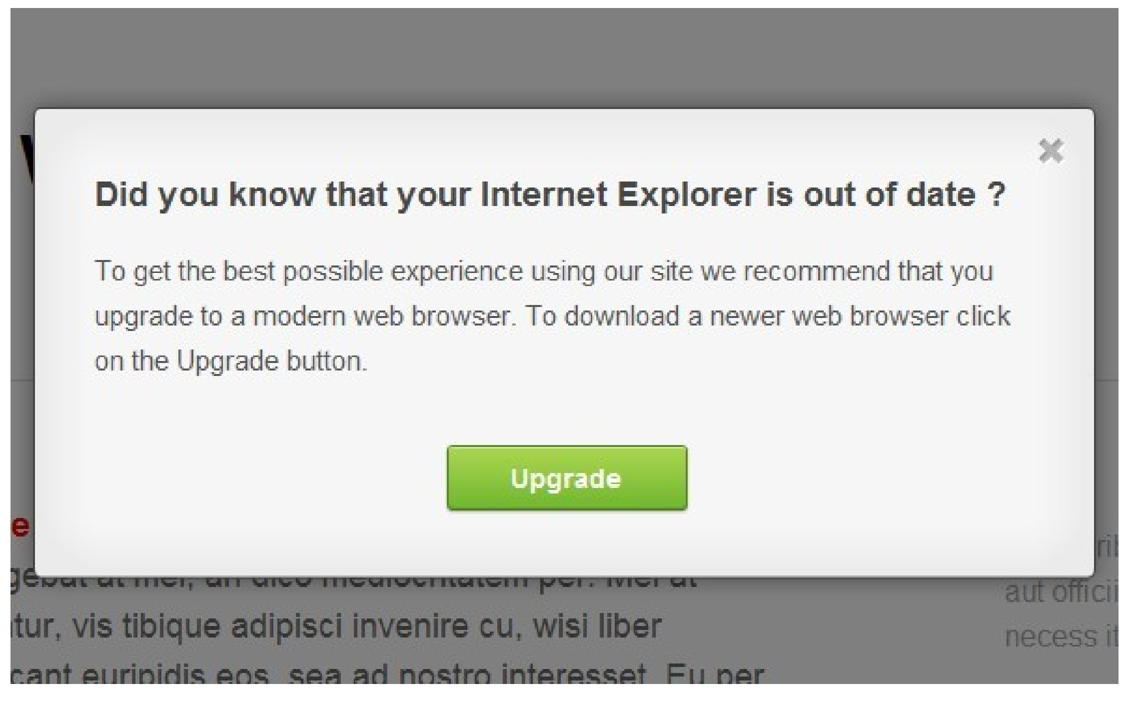


图 85: jQuery Popup

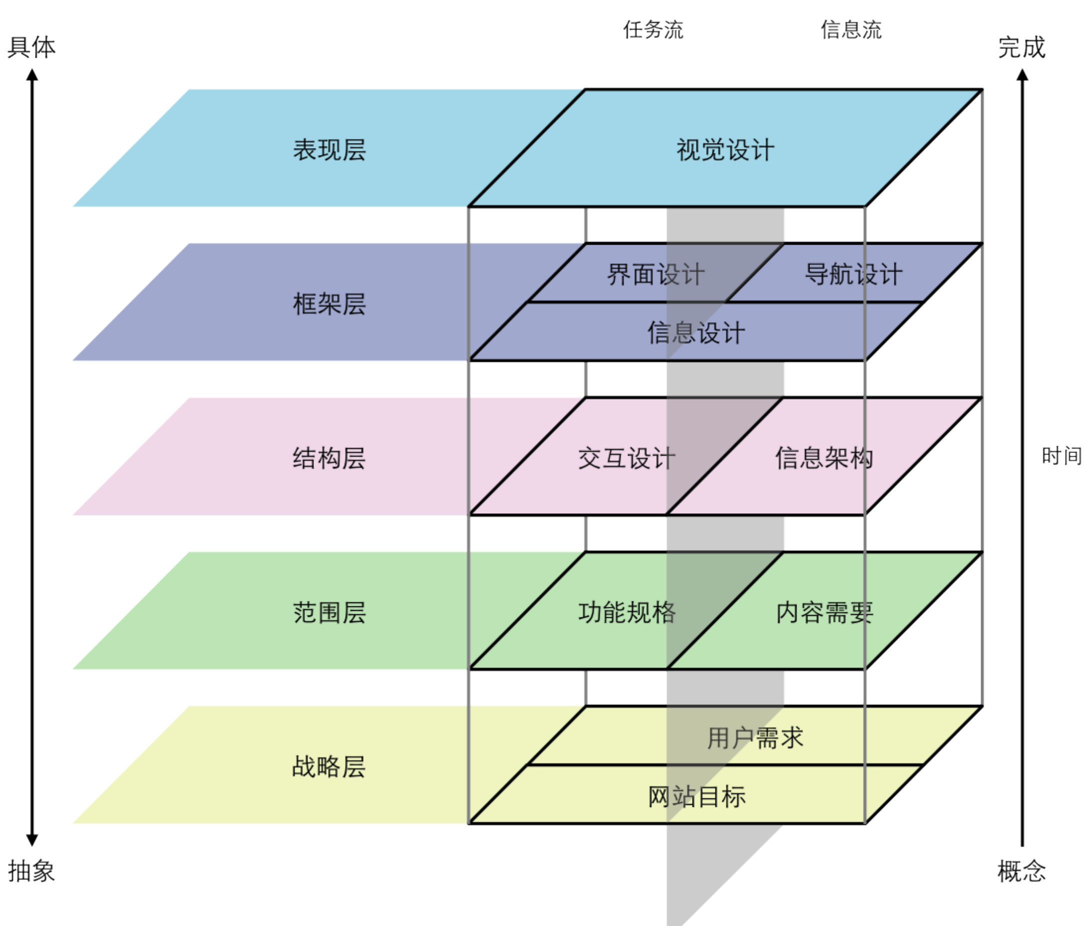


图 86: 用户体验要素

参考目录

- 《怦然心动——情感化设计交互指南》
- 《用户体验要素》
- 《移动应用 UI 设计模式》

认知设计

第一次意识到这本书很有用的时候，是我在策划一个视频。第二次，则是我在计划写一本书的时候。

流

在《认知设计》一书中，提到了下面的学习体验，即 “流” (Flow)。而在我们学习的过程中，我们也会有类似的学习过程。

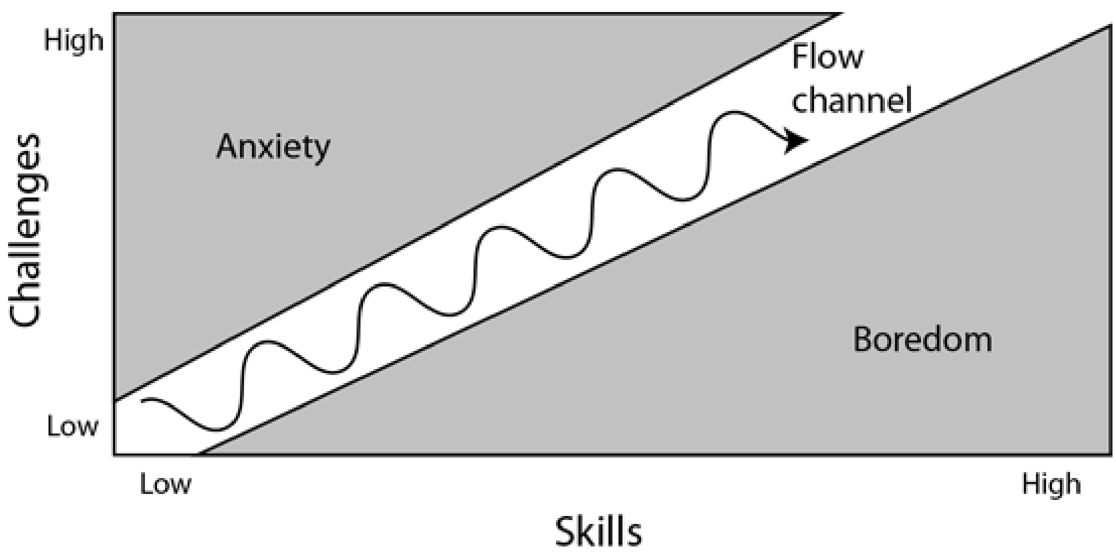


图 87: Learn Design

如在早期我学习 Emcas 和 GNU/Linux 的时候，也曾经放弃过，虽然在当时我已经读过 Linux 内核。然而，在应用之前进行理论学习并没有卵用。

通常我们会有类似于下面的学习体验，对于一本书来说有下面的体验似乎也是一件很不错的事:

1. 在最开始学习的时候，我们需要一点理论基础，以及我们需要学点什么。
2. 然后，我们需要构建一个简单可用的系统，以获取信心。如果我们在这一步没有想象中，那么简单，那么我们可能会放弃学习。或者等到某个时期成熟的时刻，如

在我开始学习《设计模式》的时候，那么本书的高度太高了。直到有一天，我了解到了一本叫《Head First 设计模式》的书，才重新把 GoF 的书看了一遍，发现其实也没有想象中的难。

3. 接着在我完成了某个功能之后，那么我可能继续学习某个理论，用于支撑我的下一步计划。
4. 在那之后，我觉得这一步可能也不是那么难，因为已经有了前面的基础。如果在一步失败的时候，那么我们可能会继续寻找某些可靠的方案，又或者是理论支撑。
5. 。。。
6. 直到有一天，我们来到了一个瓶颈的前面，现有的方案已经不满足我们的需求。对于这个问题，我们可能已经没有一个更好的解决方案。于是，我们可能就需要创建一个轮子，只是在这时，我们不知道怎样去造轮子。
7. 于是我们开始学习造轮子。
8. ....

只有当我们保持一个学习的过程，才会让我们在这一步步的计划中不会退缩，也不能退缩。

持续交付

> 交付管道的建立和自动化是持续交付的基础

持续集成

更关注代码质量。持续集成是为了确保随着需求变化而变化的代码，在实现功能的同时，质量不受影响。因此，在每一次构建后会运行单元测试，保证代码级的质量。单元测试会针对每一个特定的输入去判断和观察输出的结果，而单元测试的粒度则用来平衡持续集成的质量和速度。

持续集成的核心价值在于[1]：

1. 持续集成中的任何一个环节都是自动完成的，无需太多的人工干预，有利于减少重复过程以节省时间、费用和工作量；
2. 持续集成保障了每个时间点上团队成员提交的代码是能成功集成的。换言之，任何时间点都能第一时间发现软件的集成问题，使任意时间发布可部署的软件成为了可能；

[1]基于 Jenkins 快速搭建持续集成环境

3. 持续集成还能利于软件本身的发展趋势，这点在需求不明确或是频繁性变更的情景中尤其重要，持续集成的质量能帮助团队进行有效决策，同时建立团队对开发产品的信心。

持续集成系统

在前面的内容里，我们已经介绍了持续集成的各项基础设施——如使用版本管理、编写测试、自动化部署。要构建这样的一个持续集成系统需要下面的内容：

- 支持自动构建
- 源码服务器
- 持续集成服务器

我们已经实现了前两点，针对于第三点——持续集成服务器，我们可以以 Jenkins 为例做一些简单的说明。它是一种基于 Java 开发的持续集成工具，并提供了用于监控持续重复工作的软件平台。

它可以让整个开发流程到部署都实现自动化。由于每个功能可以一点点的加在 build 中，那么这样就能保证每次的新 build 可以交付新的功能。同时，我们可以根据用户的反馈情况及时调整开发方向，降低项目风险。

持续集成流程

我们就可以对这个工作流展开进入介绍。持续集成重要就是要保证整个过程是可持续的。如下图是一个持续集成的工作流：

不同的开发者在自己的机器上开发功能代码。在完成一定的本地提交后，这些代码将会提交给源代码控制服务器。不过，在那之前我们应该在本地跑测试来减少持续集成失败的情况。接着，我们的 CI 会定时去获取源码服务器是否有代码修改。如果有代码修改，那么我们的集成服务器将会获取这些代码。然后，构建这个项目，运行测试，再输出返回结果。最后，我们可以开发一些小工具来提醒用户 CI 是否运行成功。

如果这个过程中，我们我们的 CI 运行失败的话，那么我们就不能再提交新的代码——除了修复 CI 的代码外。持续集成变红不能过夜，要么快速解决，要么回退。

在这个过程中，有两点值得注意，一个是小步前进，一个是迟早反馈。

小步前进　小步前进是一系列步骤的集合，其目的是：集成越早问题越小。即当我们的代码越早的提交到源码服务器，那么其他人就可以尽可能早的和我们的代码集成到一

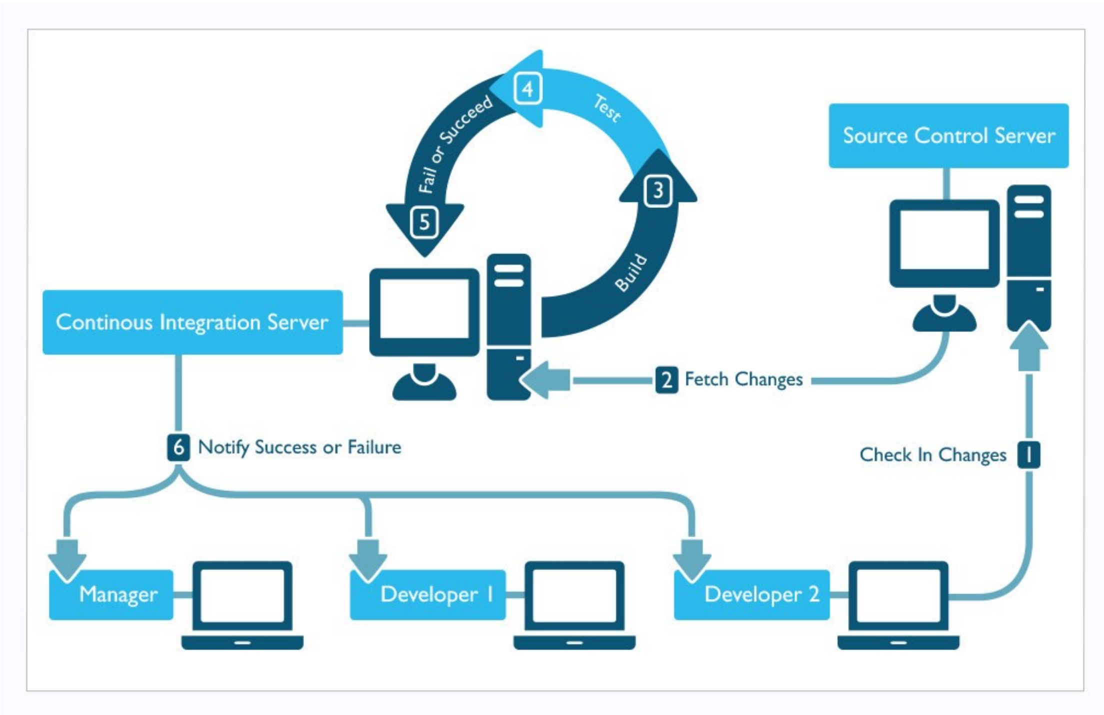


图 88: CI Workflow

起。这也意味着，每天结束时，我们在本地的修改要尽可能小，并且这些修改还要保证不会破坏持续集成。

我们需要频繁地在本地提交我们的代码，编写独立的测试——如果我们在最后才编写测试，这会拖慢整个流程，它使得我们不能尽可能早的提交代码。

尽早反馈

> 反馈越早，那么问题越小。

无论是精益还是敏捷都在强调一点——尽早反馈，反馈对于提高敏捷开发流程效力非常重要。从日常的：

- Code Review
- 静态代码分析
- 自动集成测试
- 自动验收测试
- 高频率发布

我们都在做尽可能小的反馈，这些实践对于我们完成一个好的持续集成来说是非常重要的基础。

参考资料：

- 《持续交付：发布可靠软件的系统方法》

持续交付

持续交付依赖于一系列的工具和实践，下图是一个持续交付的工作流：

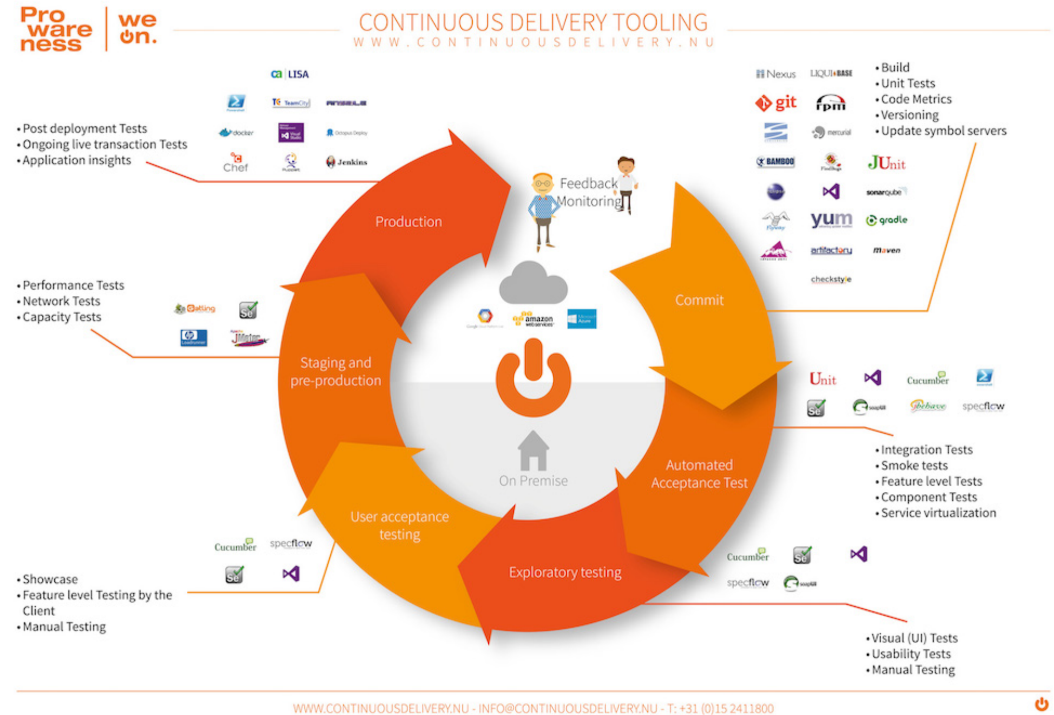


图 89: CD Workflow

还有一系列与开发无关的技能：

1. 自动化
2. DevOps
3. 云基础设施
4. 以软件为中心的哲学

基础设施

在我们使我们的项目可以持续交付软件包的时候，我们需要

本地开发环境　在本地编写代码时，我们需要设置本地的开发环境。假设我们要开始一个 Java Web 项目，在我们的开发机器上，我们需要安装：

- 版本管理工具，如 git，用于管理源代码。
- IDE，如 Intellij IDEA，用于搭建开发环境。
- 构建工具，如 gradle，用于安装依赖、运行测试、构建工程等等。
- 语法检测工具，如 checkstyle，用于检查代码语法。
- 单元测试框架，如 JUnit，用于进行单元测试。
- 集成测试框架，如 Cucumber、Selenium，用于做行为测试。

除此，在我们的项目代码里，我们还需要：

- `CI` 运行脚本，用于在 CI 上运行指定的测试。
- 上传包脚本，用于上传 build 完的软件包。
- 部署脚本，用于在本地部署包到测试环境。
- 监控代码，用于监测网站性能和用户行为。

当然我们还需要辅助一些测试工具来测试网站，如性能测试、网络测试等等。

持续集成环境　为什么在这里会出现一个持续集成环境？我也不知道，只是想到了这里。由于我们需要持续集成，所以我们也需要一个运行持续集成服务器的机器。

持续集成服务器是由两部分组成的：Master 和 Agent。即一个用于控制其他运行持续集成服务的机器，以及执行指令的机器。因此，我们需要在一台机器上安装 Master 软件，在另外一台机器上作为 Agent。在我们的 Agent 上，我们需要安装相对应的运行服务的软件，如

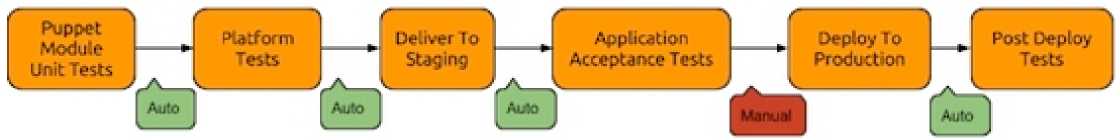
- 指定版本的语言环境，如 Java、Python。
- 构建工具。
- 版本管理工具，及对应的密钥。
- 打包工具，如 RPM。
- 虚拟桌面，即可以模拟桌面浏览器的软件。

同时，我们还需要有一个地方放置我们的 RPM 包。

测试环境　相比于上面两个环境来说，测试环境就比较简单了。我们只需要创建几个不同的环境，即开发者的测试环境、QA 环境、模拟线上环境，在这几个不同的环境上有不同的配置。

持续部署

在持续交付之外的，还有持续部署——这个就更依赖于团队的组织结构了。其与持续交付的对比如下图所示：



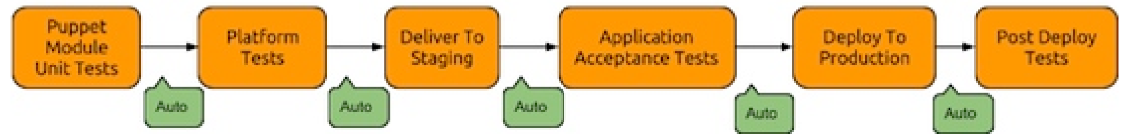


图 90: 持续部署

我们可以从图中看到，两者的最大不同之处在于：持续部署会直接将构建生成的部署到产品环境。这就意味着，我们不仅要有强大的技术实力，也要有足够的组织支持才能做到。而这部分已经超出了软件开发的内容了 ~~。

持续学习

如果说持续交付则是一种对卓越的追求，那么持续学习应该就是追求软件卓越。如果说持续集成是一种软件开发实践，那么对于技术人员来说——持续写作应该就是持续学习的实践

生活总会遇到压力，来自工作上的也好，来自对于技术上的兴趣也罢，我们需要持续来断地学习。没有一直能立于不败的方法，在传说中的武林上也是如此。

对于持续学习来说，通常会有以下的

- 阅读
- 编程
- 写作

有意思的是持续学习有额外的好处便是

- 持续学习可以降低危机感

持续阅读

看过如此多的金庸、古龙小说我们都会发现有那么多的人都在追求武功上的卓越，有的走火入魔了，有的铤而走险杀人放火，暂且不讨论这些。我们简单的以大部分的主角为例，大部分的主角自小就练得一手好武艺，少部分除外，而他们通过会比前辈厉害，只是因为我们看了前人的说，现在也是如此。

20 年前要建一个淘宝怕是没有两三个月十几个是不行的，但是今天要建出原来淘宝的模样，也许一个人单枪匹马一两天就能搞定了，还能上线。

有意思的是武林小说的武林秘籍少之又少，正常情况下能学到的或许就是教科书上的种种。而现在，如果我们要学习 UX 的话，我们很容易可以从亚马逊上拿到一个书单，又或者是某个博客里面列举出来的:《用户体验要素》、《交互设计沉思录》、《怦然心动——情感化交互设计指南》等等。

我们可以更加方便快捷地获取我们所需要的知识从书上、网上等等。

阅读更多的书籍是持续学习的基础。

总会听到有些人在工作之余看了更多的书，在某种情况下来说是有意义的。我们需要不断地去阅读。

持续编程

编程算是一个开发人员工作时一直在做的，而对于工作之后来说，到底还会有多少人继续编程就是一个有意思的问题。

对于一个有兴趣的程序员来说，工作和兴趣都是分开的，可以将工作视之为无味的东西，但是休息时间呢？可以用来创造自己觉得有意义的东西，可以用来认识更多志同道合的人，对于不满现状的人更是如此，或许为自己创造了更多的机会。

如果工作之后编程，不应该是为了工作而编程，应该为了兴趣而编程，或者其他。如果没有时间，是不是因为加班了，对于刚开始养家糊口来说加班是没有办法的，但是如果不是的话，又没时间，是不是……

持续写作

对于一个技能人员来说，写作可能不是一件有意思的事，但是也不是一件很难的事，没有必要将大量的文字用文本表示。写给其他技术人员看的，有时候更多的是代码、思路、图。写作对于学习的意思怕是有一大把，写作是最好的输入，也是最好的输出。你需要为你的这篇文章

- 去参考更多的资料
- 更深入的学习
- 更多的时间付出

然而这些都是有价值的，你也许可以从中得到

- 一份工作
- 一些志同道合的朋友
- 一个博客
- 一种习惯
- 还有人生
- 或许还能写书。

对于我来说，更多的是对于读者和 SEO 的兴趣，SEO 是一门艺术。

遗留系统与修改代码

尽管维基百科上对遗留系统的定义是:

> 一种旧的方法、旧的技术、旧的计算机系统或应用程序。

但是实际上，当你看到某个网站宣称用新的框架来替换旧的框架的时候，你应该知晓他们原有的系统是遗留系统。人们已经不想在上面工作了，很多代码也不知道是干什么的，也没有人想去深究——毕竟不是自己的代码。判断是否是遗留代码的条件很简单，维护成本是否比开发成本高很多。

- 几乎无法维护
- 代码遗失
- 逻辑不清
- 没有文档或者不够详细、看不懂
- 关键点遗失

在维护这一类系统的过程中，我们可能会遇到一些原因来修改代码。如《修改代码的艺术》的一书中所说，修改软件有四大原因:

- 增加特性

- 修复 Bug
- 改善设计
- 优化

当我们修改代码之后，我们将继续引进新的 Bug。

参考阅读

- 《修改代码的艺术》

遗留代码

我们生活息息相关的很多软件里满是错误、脆弱，并且难以扩展，这就是我们说的"遗留代码"。

相信你也经常看到某某网站的高架构之路，会发现其中一个很有趣的过程就是他们会把之前的架构抛弃掉。接着，他们又做了一个这样的系统，然后过些年这个系统又被重做了。究其原因，会发现这个架构是在几年前设计的。在几年前，他是非常好的架构。但是随着时间的演变，他还是几年前的架构。这是为什么呢?

遗留代码

什么是遗留代码?

> 没有自动化测试的代码就是遗留代码，不管它是十年前写的，还是昨天写的。

从一个新手程序员到一个老鸟，我们的编程水平都在不断增加。但是我们过去写的代码一直都在那里，但是我们一直都没有足够的勇气去动他们。因为我们知道如果我们一不小心改错了什么，就会导致一些意外的 Bug。这些 Bug 可能会对我们的编程生涯造成一些影响。

而我们不知道这样做的后果，是因为我们没有对原来的代码进行测试。如果我们的代码都是经过测试的，那么我们在修改中出的错，都会在测试中加以体现。长此以往，没有人敢去修改这些代码。

既然他在旧的系统中工作得很好，那么我们就没有理由去修改他们。当有新的需求出现时，我们就可以重新设计一个新的系统。

如何修改遗留代码

> 即使是最训练有素的开发团队，也不能保证始终编写出清晰高效的代码。

然而，如果我们不去尝试做一些改变，这些代码就会遗留下去——成为遗留代码，再次重构掉。即使说，重构系统是不可避免的一个过程，但是在这个过程中要是能抽象中领域特定的代码、语言也是件不错的事。

修改遗留代码

So，如何开始修改代码？如《修改代码的艺术》一书所说，应该是下面的五个步骤：

1. 代码修改点
2. 找到测试点
3. 打破依赖
4. 编写测试
5. 修改并重构

在有测试的情况下重构现有的代码才是安全的。而这些测试用例也是功能的体现，功能首先要得到保证了，然后才能保证一切都可以正常。不过，我更喜欢以下面三点概括他们：

- 守: 找到测试点。守，即保证原有的功能是正确的。在这基础上，我们需要添加测试
- 破: 打破依赖。会导致遗留代码的一个原因还有，原有代码的耦合度比较高。因此，我们需要去打破这些耦合，重新构建依赖。
- 离: 修改并重构。

不过，我想你只要有前面的那些步骤。你为什么还需要看这一章的内容呢？

参考书籍：

《修改代码的艺术》 《持续交付指南：修改代码的 9 条最佳实践》

网站重构

> 网站重构应包含结构、行为、表现三层次的分离以及优化，行内分工优化，以及以技术与数据、人文为主导的交互优化等。

从我所了解到的网站重构，它大概可以分为下面的几类：

1. 速度优化

2. 功能加强
3. 模块重构

下面就我们来看这三类的网站重构

速度优化

通常来说对于速度的优化也包含在重构中

- 压缩 JS、CSS、image 等前端资源
- 程序的性能优化 (如数据读写)
- 采用 CDN 来加速资源加载
- 对于 JS DOM 的优化
- HTTP 服务器的文件缓存

如对于压缩前端资源这一类的重构，不仅仅需要从代码层级来解决问题，也可以借由服务器缓存来解决问题。在这时候就需要去判断应该由哪个层级来做这样的事情——如果一件事可以简单地由机器来解决，但是由人来解决需要花费大量的时间，这时就应该交由机器来解决。而如果由人来解决是一个长期受期，并且成本比较低的事，那么就应该由人来解决。如我们只需要在我们的构建脚本中引入 minify 库就可以解决的事，那么应该交由人来做。

如，采用 CDN、HTTP 服务器的文件缓存这一类应该交由机器来做。

同时像程序性能优化、JS DOM 优化都应交由人来解决的事。特别是像程序性能优化，从长期来看可能是一件长期受益的事。当且仅当，我们遇到性能问题时，我们重构这部分代码才可能带来优势。如果我们的网站的访问量不是特别大，那么优化可能就是徒劳的。但是这种优化对于个人的成长还是挺有帮助的。

功能加强

一般来说功能加强，应该是由于需求的变动才引起对系统的重构需求：

- 解耦复杂的模块 -> 微服务
- 对缓存进行优化
- 针对于内容创建或预留 API
- 需要添加新的 API
- 用新的语言、框架代替旧的框架 (如 Scala, Node.js, React)

模块重构

深层次的网站重构应该考虑的方面

- 减少代码间的耦合
- 让代码保持弹性
- 严格按规范编写代码
- 设计可扩展的 API
- 代替旧有的框架、语言
- 增强用户体验

回顾与架构设计

在我开始接触架构设计的时候，我对于这个知识点觉得很奇怪。因为架构设计看上去是一个很复杂的话题，然而他是属于设计的一部分。如果你懂得什么是美、什么是丑，那么我想你也是懂得设计的。而设计是一件很有意思的事——刚开始写字时，我们被要求去临摹别人的字体，到了一定的时候，我们就可以真正的去设计。

自我总结

总结在某种意义上相当于自己对自己的反馈：

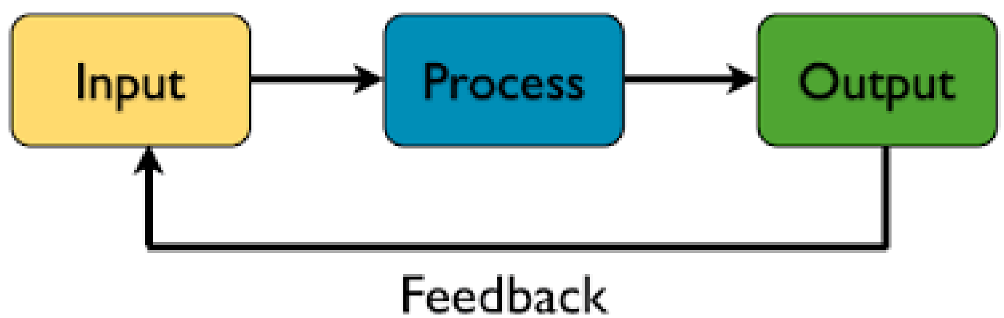


图 91: Output is Input

当我们向自己输入更多反馈的时候，我们就可以更好地调整我们的方向。它属于输出的一部分，而我们也在不断调整我们的输入的时候，我们也在导向更好地输出。

吾日三省吾身

为什么你不看不到自己的方向？

Retro

Retro，又可以称为回顾，它的目的是对团队的激励、改进。它的模式的特点就是让我们更关注于 Less Well，即不好的地方。当我们无法变得更好的时候，它可以帮助我们反观团队自身，即不要让现状变得更差，避免让破窗效应[2]难以发生。

在敏捷团队里，Retro 通常会发生一个迭代的结束与下一个迭代的开始之间，这看上去就是我们的除旧迎新。相信很多人都会对自我进行总结，随后改进。而 Retro 便是对团队进行改进，即发生了一些什么不好的事，而这些事可以变好，那么我们就应该对此进行改进。

Retro 是以整个团队为核心去考虑问题的，通常来说没有理由以个人为对象。因为敏捷回顾有一个最高指导原则，即：

> 无论我们发现了什么，考虑到当时的已知情况、个人的技术水平和能力、可用的资源，以及手上的状况，我们理解并坚信：每个人对自己的工作都已全力以赴。

下面就让我们来看看在一个团队里是如何 Retro 的。

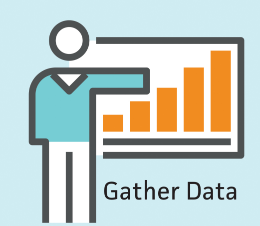
Retro 的过程

它不仅仅可以帮助我们发现团队里的问题，也可以集思广益的寻找出一些合适的解决方案。Retro 的过程和我们之前说的数据分析是差不多的，如下图所示：

即：

1. 设定会议目标。在会议最开始的时候我们就应该对会议的内容达成一种共识，我们要回顾的主题是啥，我们要回顾哪些内容。如果是一般性的迭代 Retro，那么我们的会议主题就很明显了。如果是针对某一个特定项目的 Retro，那么主题也很明显。
2. Retro 的回顾。即回顾上一个 Retro 会议的 Action 情况，并进行一个简单的小结。
3. 收集数据。收集数据需要依赖于我们收集数据的模式，要下面将会说到的四种基本维度，或者是雷达图等等。不同的收集数据的形式有不同的特别，团队里的每个人都应该好好去参与。

[2]以一幢有少许破窗的建筑为例，如果那些窗不被修理好，可能将会有破坏者破坏更多的窗户。最终他们甚至会闯入建筑内，如果发现无人居住，也许就在那里定居或者纵火。又或想像一条人行道有些许纸屑，如果无人清理，不久后就会有更多垃圾，最终人们会视为理所当然地将垃圾顺手丢弃在地上。因此破窗理论强调着力打击轻微罪行有助减少更严重罪案，应该以零容忍的态度面对罪案。



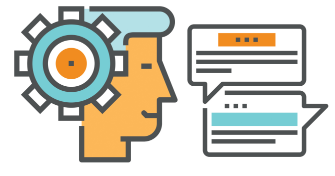


图 92: Retro 流程

4. 激发灵感。当我们寻找到团队中一个值得去庆祝的事，或者一个出了问题的事，我们就应该对这个问题进行讨论。并且对其展开了解、调查，让大家进一步看到问题，看到问题的根源。
5. 决定做什么。现在我们已经做了一系列的事，最重要的来了，就是决定我们去做什么。我们应该对之前的问题做出怎样的改进。
6. 总结和收尾。记录会议成果，更新文档等等。

三个维度

以我们为例，我们以下面的三个维度去进行 Retro:

1. Well.
2. Less Well.
3. Suggestion

当然最后还会有一个 Action：

1. Action

该模式的特点是会让我们更多的关注 Less Well，关注我们做的不好的那些。

Well。我们在 Well 里记录一些让我们开心的事，如最近天气好、迭代及时完成、没有加班等等，这些事从理论上来说应该继续保持（KEEP）下去。

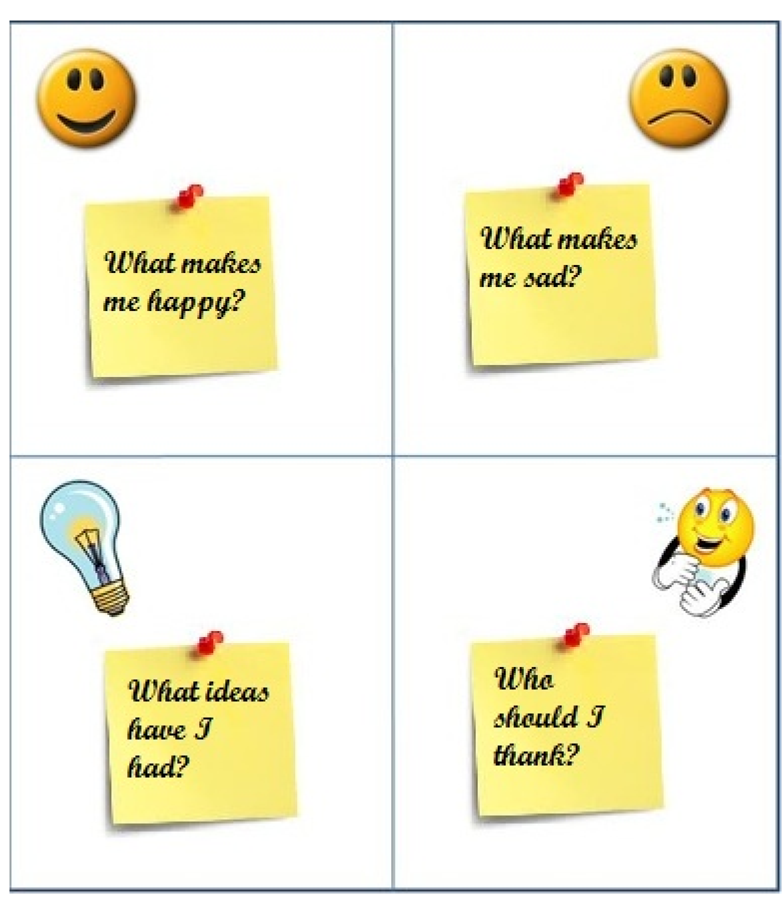


图 93: Retro

Less Well。关注于在这个迭代的过程中，发生了一些什么不愉快的事。一般来说，我们就会对 Less Well 加以细致的讨论，找出问题的根源，并试图找到一个解决方案。换句话来说，就是改变（CHANGE)。

Suggestion/Puzzle。如果我们可以直接找到一些建议，那么我们就可以直接提出来。并且如果我们对当前团队里的一些事情，有一些困惑那么也应该及早的提出来。

Action。当我们对一些事情有定论的时候，我们就会提出相应的 Action。这些 Action 应该有相应的人去执行，并且由团队来追踪。

架构模式

模式就是最好的架构。

架构的产生 在我开始接触架构设计的时候，我买了几本书然后我就开始学习了。我发现在这些书中都出现了一些相似的东西，如基本的分层设计、Pipe and Filters 模式、MVC 模式。然后，我开始意料到这些模式本身就是最好的架构。

MVC 模式本身也是接于分层而设计的，如下图是 Spring MVC 的请求处理过程：

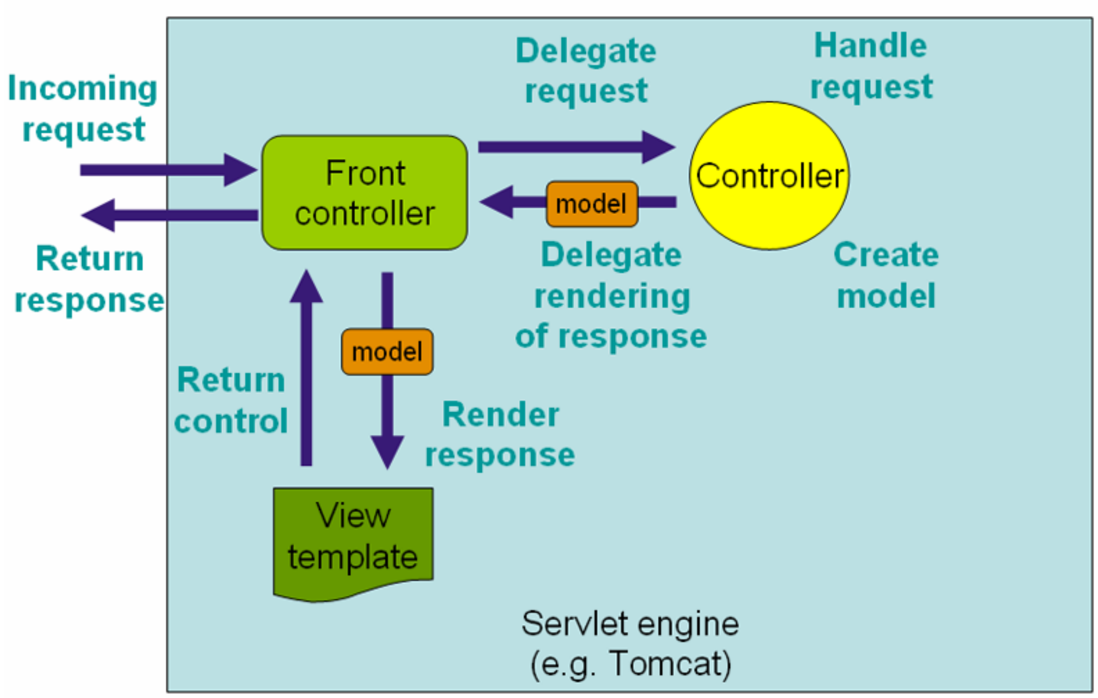


图 94: Spring MVC

而这个框架只是框架本身的架构，这一类也是我们预先设计好的框架。

在框架之上，我们会有自己本身的业务所带来的模式。如下图是我的网上搜罗到的

一个简单的发送邮件的架构：

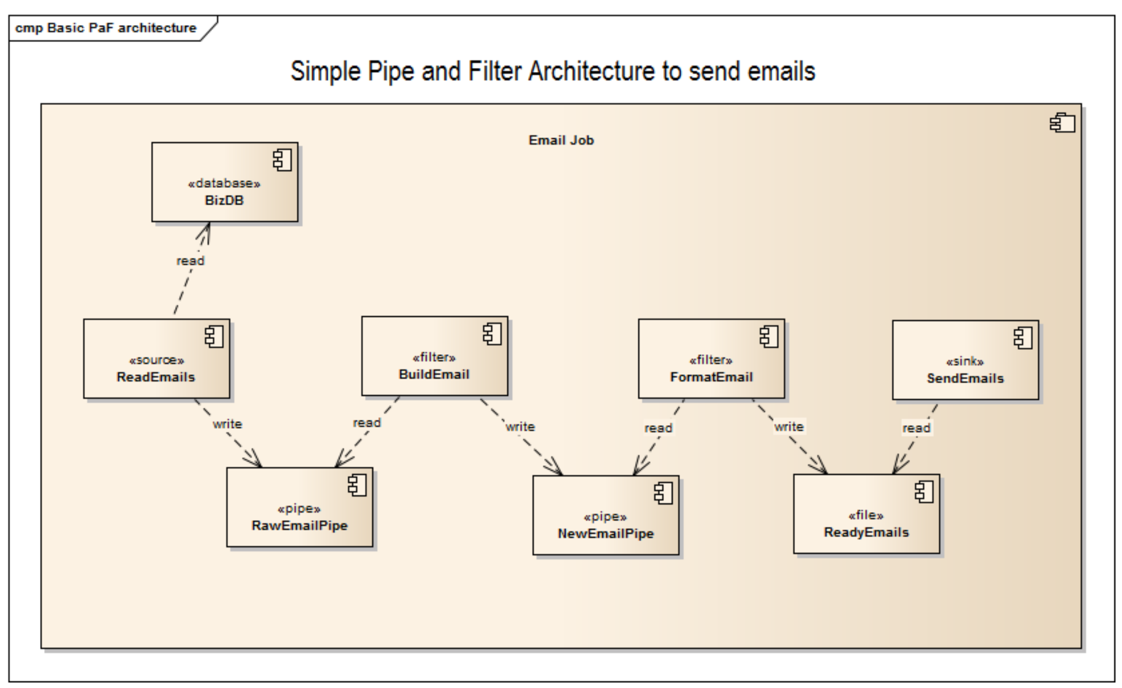


图 95: 发送邮件中的 Pipe and Filters 模式

这样的模式则是由业务发展的过程中演进出来的。

预设计式架构

在我们日常使用的框架多数是预先设计的构架，因为这个架构本身的目标是明确的。系统会围绕一定的架构去构建，并且在这个过程中架构会帮助我们更好地理解系统。如下图所示的是 Emacs 的架构：

它采用的是交互式应用程序员应用广泛的模型-视图-控制器模式。

无论是瀑布式开发——设计好系统的框架，然后对系统的每个部分进行独立的完善和设计，最后系统再集成到一起。还是敏捷式开发——先做出 MVP，再一步步完善。他们都需要一定的预先式设计，只是传统的开发模式让两者看上去是等同的。

在过去由于 IT 技术变革小，新技术产生的速率也比较低，预先设计系统的架构是一种很不错的选择。然而，技术的发展趋势是越来越快，现有的设计往往在很短的一些时间里就需要推倒重来。

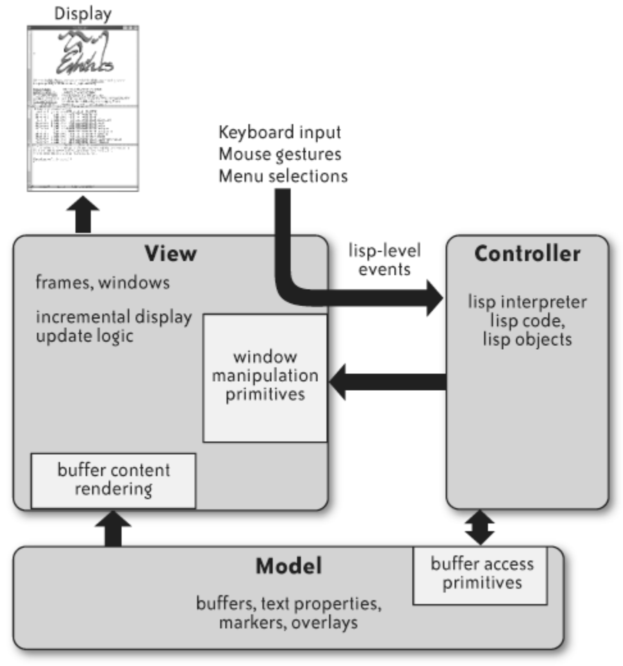


图 96: Emcas 架构

演进式架构：拥抱变化

演进式架构则是我们日常工作的业务代码库演进出来的。由于业务本身在不断发展，我们不断地演进系统的架构。也就是这样模式下产生的架构系统会更加稳定，也更加优美。仅仅依赖于事先的设计，而不考虑架构在后期业务中的变化是一种不可取的设计模式。

这不并意味着不采用预先式设计，而是不一味着去靠原先系统的架构。

浮现式设计

设计模式不是一开始就有的，好的软件也不是一开始就设计成现在这样的，好的设计亦是如此。

导致我们重构现有系统的原因有很多，但是多数是因为原来的代码变得越来越不可读，并且重构的风险太大了。在实现业务逻辑的时候，我们快速地用代码实现，没有测试，没有好的设计。

而下图算是最近两年来想要的一个答案：

浮现式设计是一种敏捷技术，强调在开发过程中不断演进。软件本身就不应该是一开始就设计好的，他需要经历一个演化的过程。

图 97: 浮现式设计

意图导向

就和 Growth 一样在最开始的时候，我不知道我想要的是怎样的——我只有一个想法以及一些相对应的实践。接着我便动手开始做了，这是我的风格。不得不说这是结果导向编程，也是大部分软件开发采用的方法。

所以在一开始的时候，我们就有了下面的代码：

```
if (rating) {
    $scope.showSkillMap = true;
    skillFlareChild[skill.text] = [rating];

    $scope.ratings = $scope.ratings + rating;
    if (rating >= 0) {
      $scope.learnedSkills.push({
        skill: skill.text,
        rating: rating
      });
    }

    if ($scope.ratings > 250) {
      $scope.isInfinite = true;
```

```
    }
  }
```

代码在不经意间充斥着各种 Code Smell:

1. Magic Number
2. 超长的类
3. 等等

重构

还好我们在一开始的时候写了一些测试，这让我们可以有足够的可能性来重构代码，而使得其不至于变成遗留代码。而这也是我们推崇的一些基本实践：

> 红 -> 绿 -> 重构

测试是系统不至于腐烂的一个后勤保障，除此我们还需要保持对于 Code Smell 的嗅觉。如上代码：

```
if ($scope.ratings > 250) {
  $scope.isInfinite = true;
}
```

上面代码中的“250”指的到底是？这样的数字怎么能保证别人一看代码就知道 250 到底是什么？

如下的代码就好一些：

```
var MAX_SKILL_POINTS = 250;
if ($scope.ratings > MAX_SKILL_POINTS) {
  $scope.isInfinite = true;
}
```

而在最开始的时候我们想不到这样的结果。最初我们的第一直觉都是一样的，然而只要我们保持着对 Code Smell 的警惕，情况就会发生更多的变化。

重构是区分普通程序员和专业程序员的一个门槛，而这也是练习得来的一个结果。

模式与演进

如果你还懂得一些设计模式，那么想来，软件开发这件事就变得非常简单——我们只需要理解好需求即可。

从一开始就使用模式，要么你是专家，要么你是在自寻苦恼。模式更多的是一些实现的总结，对于多数的实现来说，他们有着诸多的相似之处，他们可以使用相同的模式。

而在需求变化的过程中，一个设计的模式本身也是在不断的改变。如果我们还固执于原有的模式，那么我们就会犯下一个又一个的错误。

在适当的时候改变原有的模式，进行一些演进变显得更有意义一些。如果我们不能在适当的时候引进一些新的技术来，那么旧有的技术就会不断累积。这些技术债就会不断往下叠加，那么这个系统将会接近于崩塌。而我们在一开始所设定的一些业务逻辑，也会随着系统而逝去，这个公司似乎也要到尽头了。

而如果我们可以不断地演进系统——抽象服务、拆分模块等等。业务在技术不断演进地过程中，得以保留下来。

每个人都是架构师

每一个程序员都是架构师。平时在我们工作的时候，架构师这个 Title 都被那些非常有经历的开发人员占据着。然而，如果你喜欢刷刷 Github，喜欢做一些有意思的东西，那么你也将是一个架构师。

如何构建一个博客系统

如果你需要帮人搭建一个博客你先会想到什么？ 先问一个问题，如果要让你搭建一个博客你会想到什么技术解决方案？

1. 静态博客（类似于 GitHub Page）
2. 动态博客（可以在线更新，如 WordPress）
3. 半动态的静态博客（可以动态更新，但是依赖于后台构建系统）
4. 使用第三方博客

这只是基本的骨架。因此如果只有这点需求，我们无法规划出整体的方案。现在我们又多了一点需求，我们要求是独立的博客，这样我们就把第 4 个方案去掉了。但是就现在的过程来说，我们还是有三个方案。

接着，我们就需要看看 Ta 需要怎样的博客，以及他有怎样的更新频率？以及他所能接受的价格？

先说说价格——从价格上来说，静态博客是最便宜的，可以使用 AWS S3 或者国内的云存储等等。从费用上来说，一个月只需要几块钱，并且快速稳定，可以接受大量的流量访问。而动态博客就贵了很多倍——我们需要一直开着这个服务器，并且如果用户的数量比较大，我们就需要考虑使用缓存。用户数量再增加，我们就需要更多地服务器了。而对于半动态的静态博客来说，需要有一个 Hook 检测文章的修改，这样的 Hook 可以是一个客户端。当修改发生的时候，运行服务器，随后生成静态网页。最后，这个网页接部署到静态服务器上。

从操作难度上来说，动态博客是最简单的，静态博客紧随其后，半动态的静态博客是最难的。

整的性价比考虑如下：

x	动态博客	静态博客	半动态的静态博客
价格	几十到几百元	几元	依赖于更新频率几元 ~ 几十元
难度	容易	稍有难度	难度稍大
运维	不容易	容易	容易
数据存储	数据库	无	基于 git 的数据库

现在，我们已经达到了一定的共识。现在，我们已经有了几个方案可以提用户选择。而这时，我们并不了解进一步的需求，只能等下面的结果。

客户需要可以看到文章的修改变化，这时就去除了静态博客。现在还有第 1 和第 3 种方案可以选，考虑到第 3 种方案实现难度比较大，不易短期内实现。并且第 3 种方案可以依赖于第 1 种方案，就采取了动态博客的方案。

但是，问题实现上才刚刚开始。

我们使用怎样的技术？ 作为一个团队，我们需要优先考虑这个问题。使用怎样的技术解决方案？而这是一个更复杂的问题，这取决于我们团队的技术组成，以及未来的团队组成。

如果在现有的系统中，我们使用的是 Java 语言。并不意味着，每个人都喜欢使用 Java 语言。因为随着团队的变动，做这个技术决定的那些人有可能已经不在这个团队里。并且即使那些人还在，也不意味着我们喜欢在未来使用这个语言。当时的技术决策都是在当时的环境下产生的，在现在看来很扯的技术决策，有可能在当时是最好的技术决策。

对于一个优秀的团队来说，不存在一个人对所有的技术栈都擅长的情况——除非这个团队所从事的范围比较小。在一个复杂的系统里，每个人都负责系统的相应的一部分。尽管到目前为止并没有好的机会去构建自己的团队，但是也希望总有一天有这样的机会。在这样的团队里，只需要有一个人负责整个系统的架构。其中的人可以在自己擅长的层级里构建自己的架构。因此，让我们再回到我们的博客中去，现在我们已经决定使用动态的博客。然后呢？

作为一个博客我们至少有前后台，这样我们可能就需要两个开发人员。

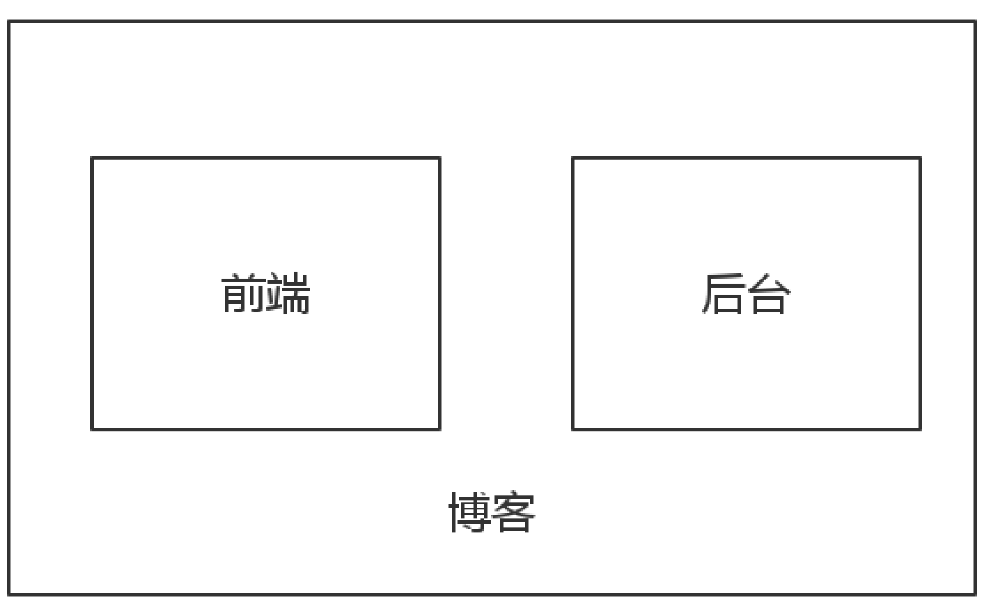


图 98: 前后台

（PS：当然，我们也可以使用 React，但是在这里先让我们忽略掉这个框架，紧耦合会削弱系统的健壮性。）

接着，作为一个前端开发人员，我们还需要考虑的两个问题是：

1. 我们的博客系统是否是单页面应用？。
2. 要不要做成响应式设计。

第二个问题不需要和后台开发人员做沟通就可以做决定了。而第一个问题，我们则需要和后台开发人员做决定。单页面应用的天然优势就是：由于系统本身是解耦的，他与后台模板系统脱离。这样在我们更换前端或者后台的时候，我们都不需要去考虑使用何种技术——因为我们使用 API 作为接口。现在，我们决定做成单页面应用，那么我们就需要定义一个 API。而在这时，我们就可以决定在前台使用何种框架：AngularJS、

Backbone、Vue.js、jQuery，接着我们的架构可以进一步完善：

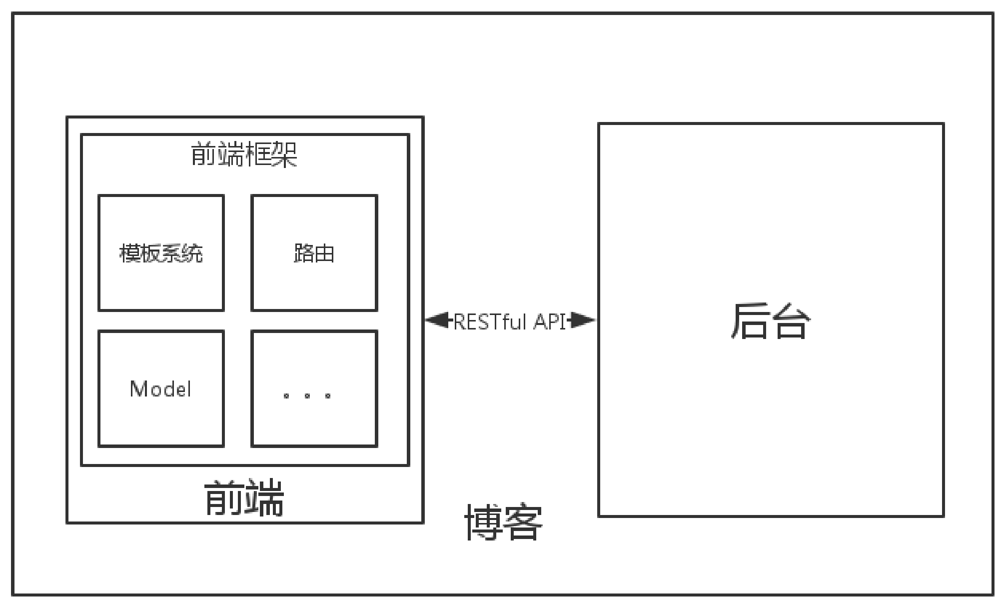


图 99: 含前端的架构

在这时，后台人员也可以自由地选择自己的框架、语言。后台开发人员只需要关注于生成一个 RESTful API 即可，而他也需要一个好的 Model 层来与数据库交付。

现在，我们似乎已经完成了大部分的工作？我们还需要：

1. 部署到何处操作系统
2. 使用何处数据库
3. 如何部署
4. 如何去分析数据
5. 如何做测试
6. 。。。

相信看完之前的章节，你也有了一定的经验了，你也可以成为一个架构师了。

相关阅读资料

- 《程序员必读之软件架构》

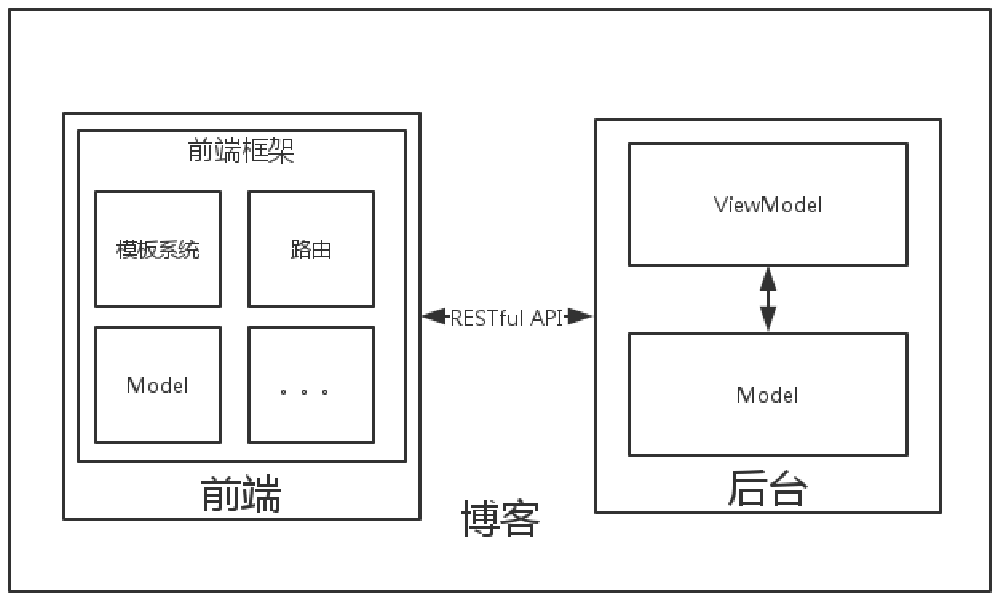


图 100: 含前端后台的架构

架构解耦

解耦是一件很有意思的过程，它也能反应架构的变迁。

从 MVC 与微服务

在我初识架构是什么的时候，我看到了 MVC 模式架构。这种模式是基于分层的结构，要理解起逻辑也很简单。这个模式如下图所示：

由我们的 Front controller 来处理由客户端（浏览器）发过来的请求，实际上这里的 Front controller 是 DispatcherServlet。DispatcherServlet 负责将请求派发到特定的 handler，接着交由对应的 Controller 来处理这个请求。依据请求的内容，Controller 将创建相应 model。随后这个 model 将传到前端框架中渲染，最后再返回给浏览器。

但是这样的架构充满了太多的问题，如 view 与 controller 的紧密耦合、controller 粒度难以把控的问题等等。

Django MTV 我使用 Django 差不多有四年了，主要是用在我的博客上。与 MVC 模式一对比，我发现 Django 在分层上还是很有鲜明特性的：

在 Django 中没有 Controller 的概念，Controller 做的事都交由 URL Dispatcher，

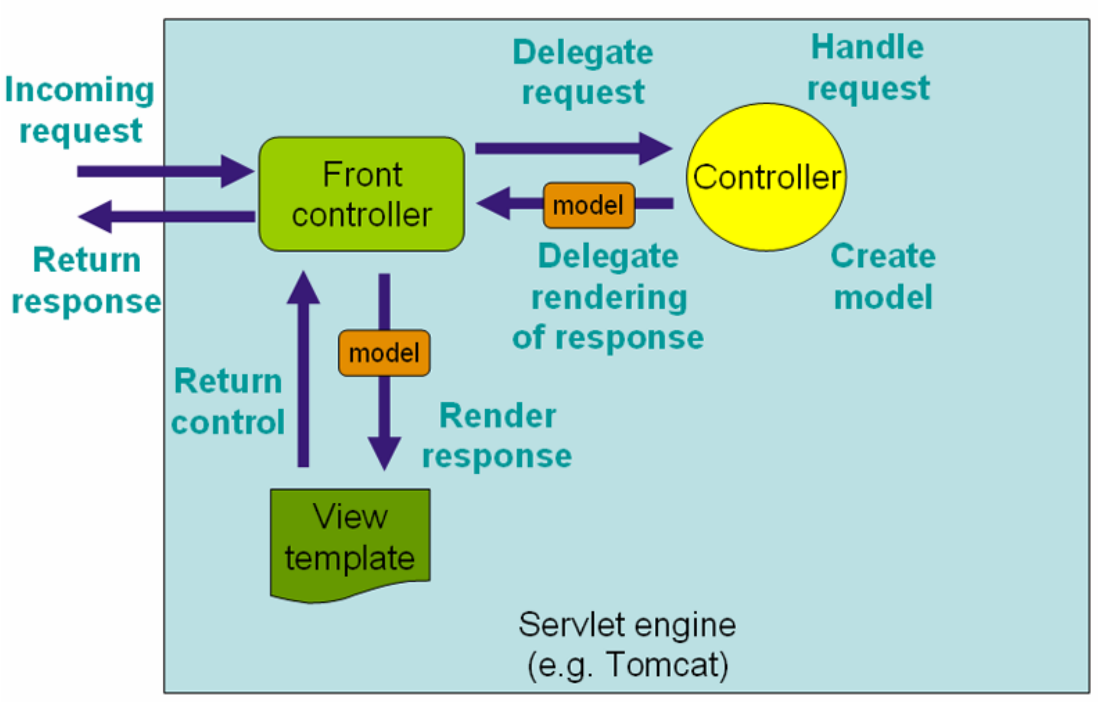


图 101: Spring MVC

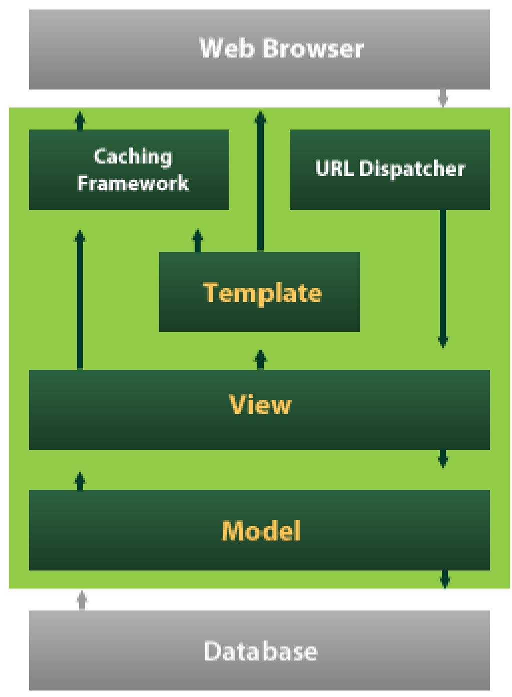


图 102: Django MTV 架构

而这是一个高级的 URL Dispatcher。它使用正则表达式匹配 URL，然后调用合适的 Python 函数。然后这个函数就交由相应的 View 层来处理，而这个 View 层则是处理业务逻辑的地方。处理完后，Model 将传到 Template 层来处理。

对比如下图如示：

传统的 MVC 架构	Django 架构
Model	Model(Data Access Logic)
View	Template(Presentation Logic)
View	View(Business Logic)
Controller	Django itself

从上面的对比中，我们可以发现 Django 把 View 分层了。以 Django 对于 MVC 的解释来说，视图用来描述要展现给用户的数据。而在 ROR 等其他的 MVC 框架中，控制器负责决定向用户展现哪些数据，而视图决定如何展现数据。

联想起我最近在学的 Scala 中的 Play 框架，我发现了其中诸多的相似之处：

虽然在 Play 中，也有 Controller 的概念。但是对于 URL 的处理先交给了 Routes 来处理，随后再交给 Controller 中的函数来处理。

不过与一般 MVC 架构的最大不同之处，怕是在于 Django 的 APP 架构。Django 中有一个名为 APP 的概念，它是实现某种功能的 Web 应用程序。如果我们要设计一个博客系统的话，那么在这个项目中，Blogpost 是一个 APP、评论是一个 APP、用户管理是一个 APP 等等。每个 APP 之中，都会有自己的 Model、View 和 Controller。其架构如下图所示：

当我们需要创建一个新的功能的时候，我们只需要创建一个新的 APP 即可——为这个 APP 配置新的 URL、创建新的 Model 以及新的 View。如果功能上没有与原来的代码重复的话，那么这就是一个独立的 APP，并且我们可以将这个 APP 的代码 Copy/Paste 到一个新的项目中，并且不需要做修改。

与一般的 MVC 架构相比，我们会发现我们细化了这些业务逻辑原来的三层结构，会随着 APP 的数量发生变化。如果我们有三个 APP 的话，那么我们相当于有 3* 三层，但是他不是等于九层。这样做可以从代码上直接减少逻辑的思考，让我们可以更加集中注意力于业务实现，同时也利于我们后期维护。

虽是如此，后来我意识到了这样的架构并没有太多的先进之处。而这实际上是一个美好但是不现实的东西，因为我们还是使用同一个数据库。

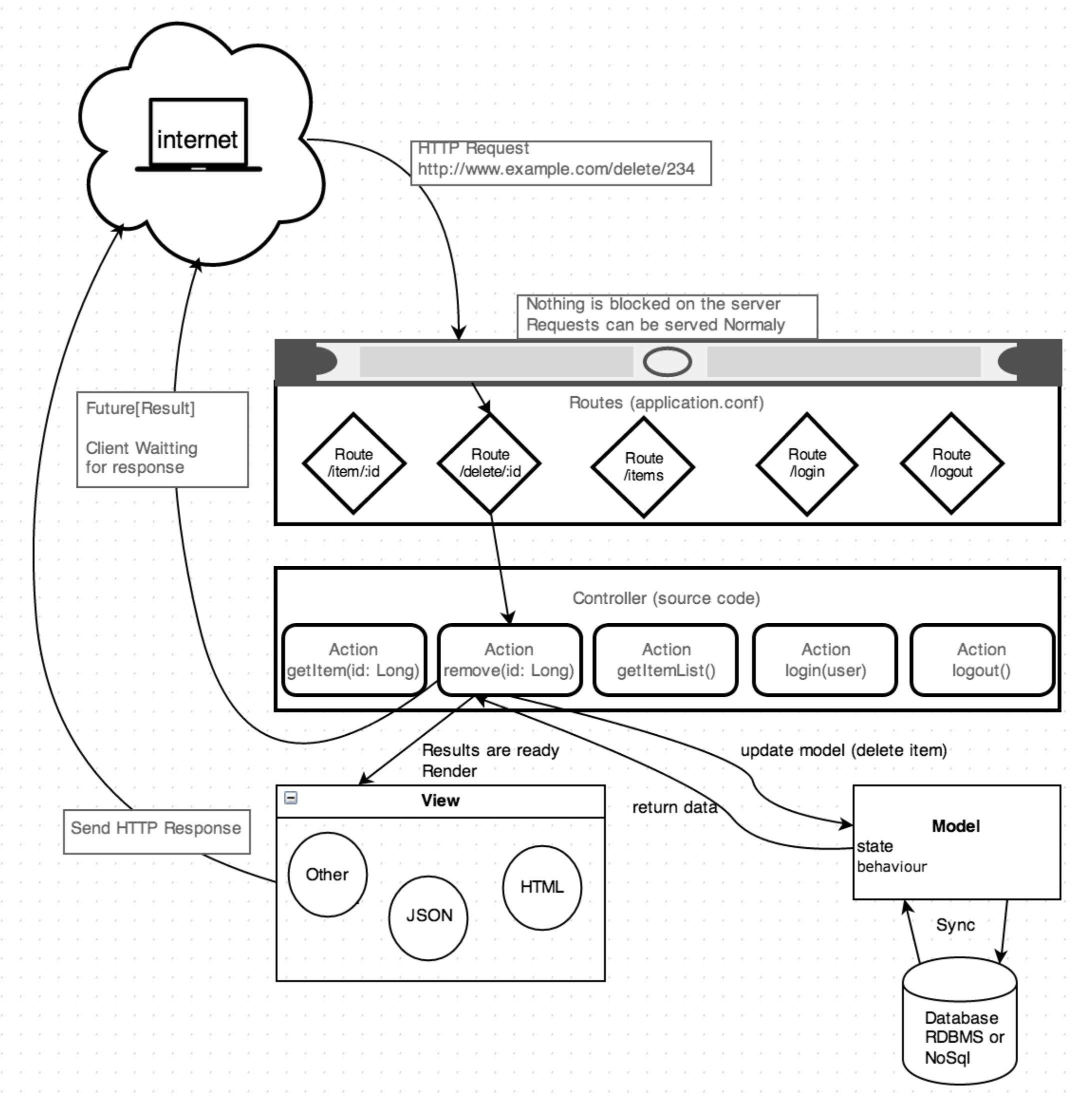


图 103: Play 框架异步请求

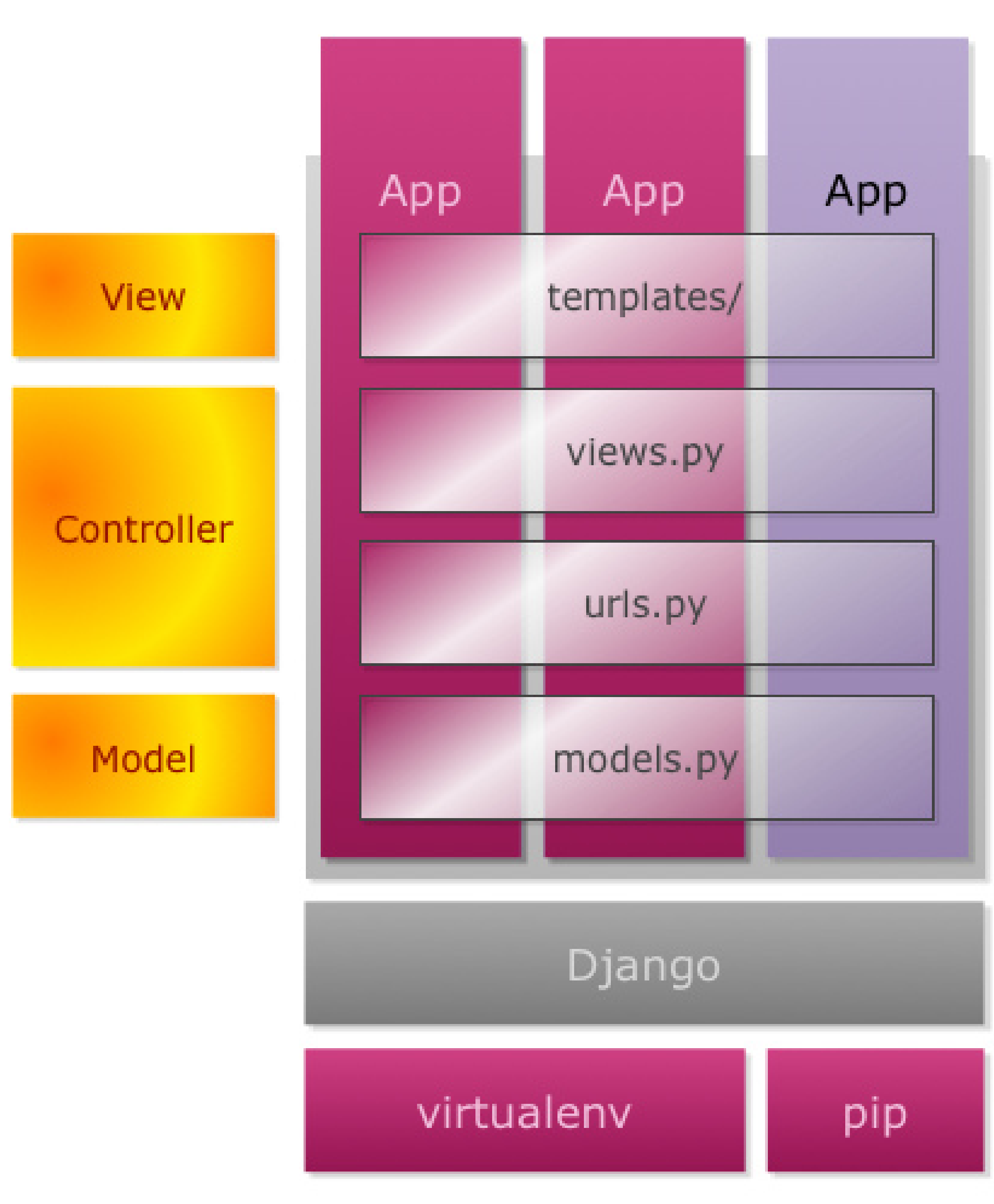


图 104: Django APP 架构

微服务与 Reactive 在微服务架构中，它提倡将单一应用程序划分成一组小的服务，这些服务之间互相协调、互相配合。每个服务运行在其独立的进程中，服务与服务间采用轻量级的通信机制互相沟通。每个服务都应该有自己独立的数据库来存储数据。

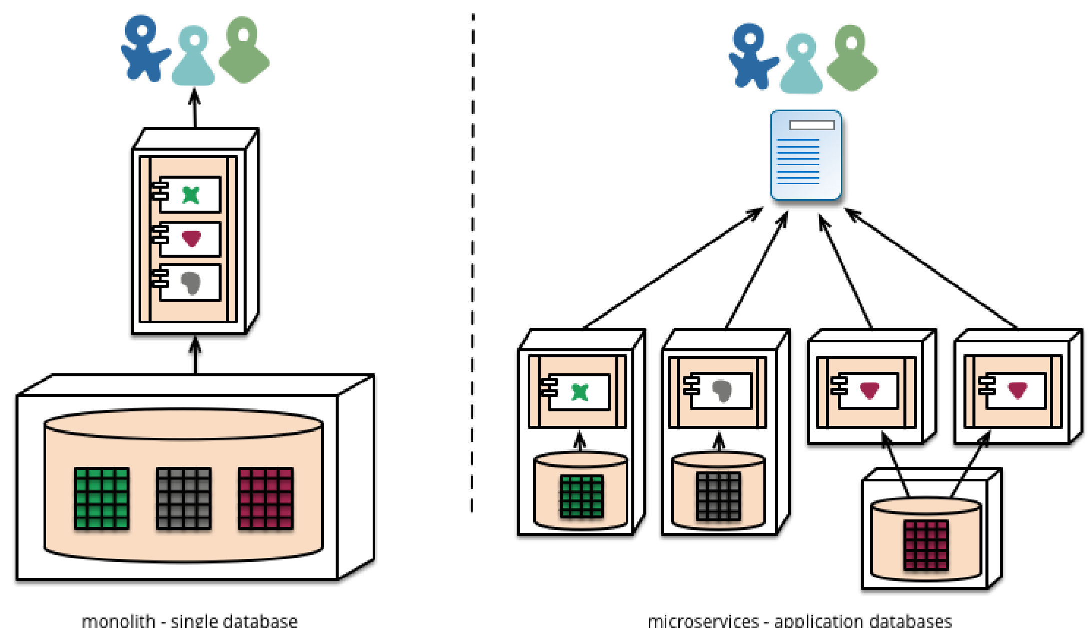


图 105: 分散数据

Django 从某种意义上有点接近微服务的概念，只是实际上并没有。因为它没有实现 Play 框架的异步请求机制。抱句话来说，应用很容易就会在调用 JDBC、Streaming API、HTTP 请求等一系列的请求中发生阻塞。

这些服务都是独立的，对于服务的请求也是独立的。使用微服务来构建的应用，不会因为一个服务的瘫痪让整个系统瘫痪。最后，这一个个的微服务将合并成这个系统。

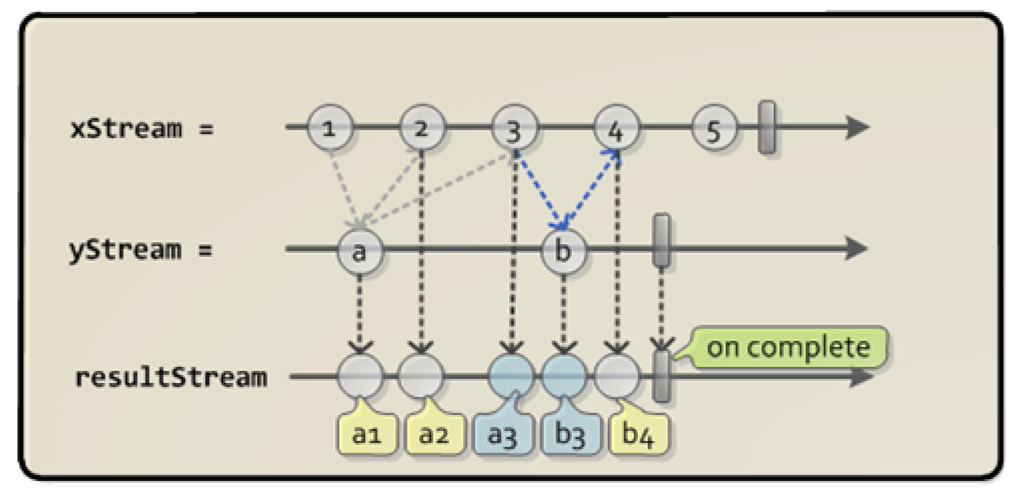


图 106: Combined List

我们将我们后台的服务变成微服务的架构，在我们的前台使用 Reactive 编程，这样我们就可以结合两者的优势，解耦出更好的架构模式。然而，这其中还有一个让人不爽的问题，即数据库。如果我们使用多个数据库，那么维护成本也随着上升。而如果我们可以在后台使用类似于微服务的 Django MTV 架构，并且它可以支持异步请求的话，并在前台使用 Reactive 来编程，是不是就会更爽一点?

CQRS

对于复杂的系统来说，上面的做法做确实很不错。但是对于一个简单地系统来说，这样做是不是玩过火了？如果我们要设计一个博客系统的话，那么我们是不是可以考虑将 Write/Read 分离就可以了?

> 命令和查询责任分离 Command Query Responsibility Segregation（CQRS）是一种将系统的读写操作分离为两种独立模型的架构模式。

CQS 对于这个架构的深入思考是起源于之前在理解 DDD。据说在 DDD 领域中被广泛使用。理解 CQRS 可以用分离 Model 和 API 集合来处理读取和写入请求开始，即 CQS（Command Query Separation，命令查询分离）模式。CQS 模式最早由软件大师 Bertrand Meyer（Eiffel 语言之父，面向对象开-闭原则 OCP 提出者）提出。他认为，对象的行为仅有两种：命令和查询。

这个类型的架构如下图所示：

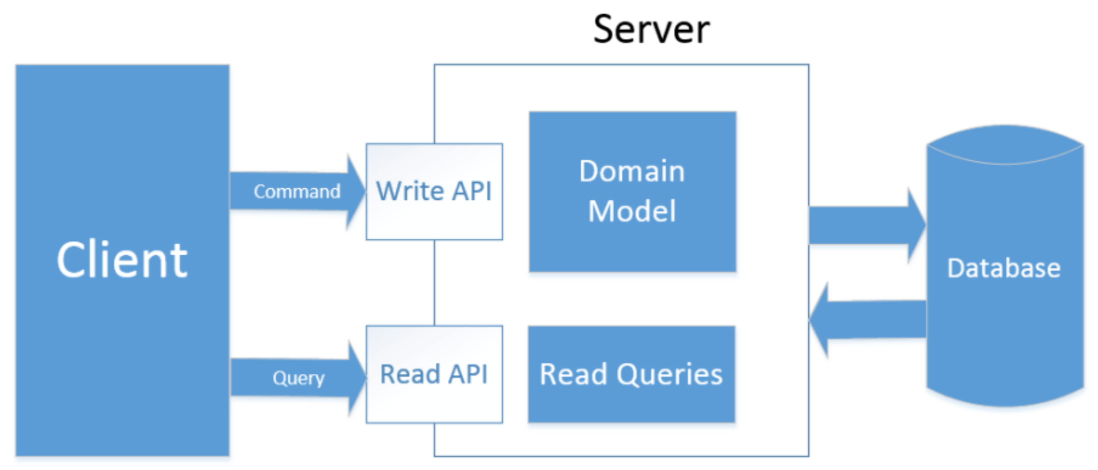


图 107: CQS Basic

> 除了编写优化的查询类型，它可以让我们轻松换 API 的一部分读一些缓存机制，甚至移动读取 API 的请求到另一台服务器。

对于读取和写入相差不多的应用来说，这种架构看起来还是不错的。而这种架构还存在一个瓶颈问题，使用同一个 RDBMS。对于写入多、读取少的应用来说，这种架构还是存在着不合理性。

为了解决这个问题，人们自然是使用缓存来解决这个问题了。我们在我们的应用服务外有一个 HTTP 服务器，而在 HTTP 服务器之外有一个缓存服务器，用于缓存用户常驻的一些资源。如下图所示：

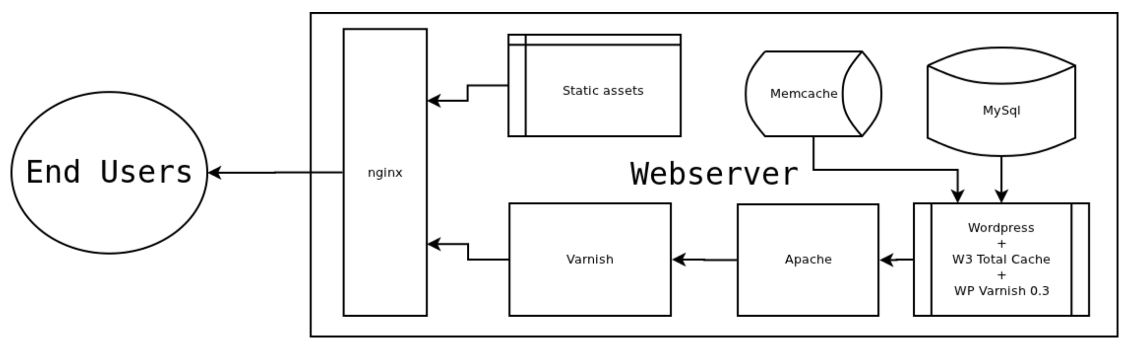


图 108: 带缓存的 Web 架构

而实际上这样的服务器可能是多余的——我们为什么不直接生成 HTML 就好了？

编辑-发布分离 或许你听过 Martin Folwer 提出的编辑-发布分享式架构：即文章在编辑时是一个形式，而发表时是另一个形式，比如用 Markdown 编辑，而用 HTML 发表。

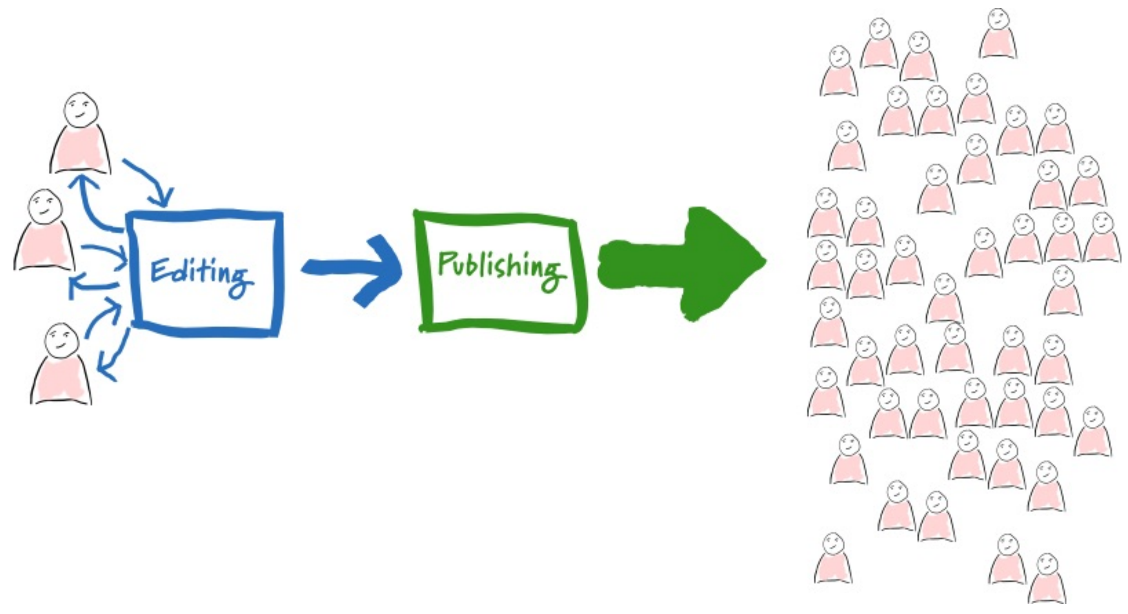


图 109: 编辑-发布分离

而最典型的应用就是流行于 GitHub 的 Hexo、Jekyll 框架之类的静态网站。如下图所示的是 Hexo 的工作流：

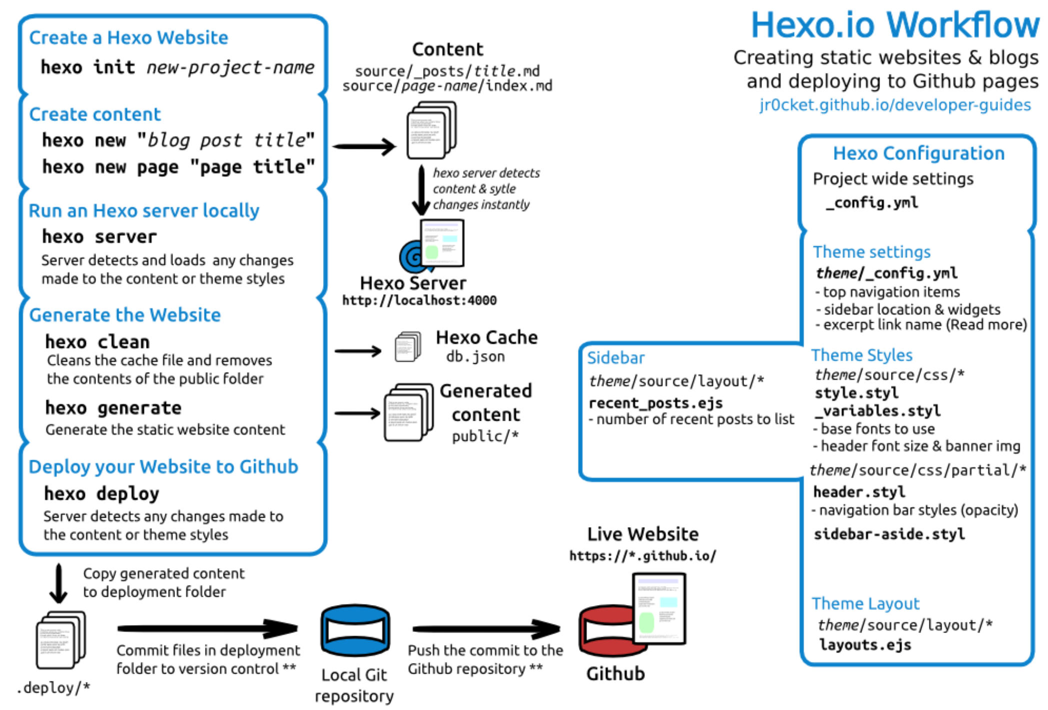


图 110: Hexo 站点工作流

我们在本地生成我们的项目，然后可以创建一个新的博客、开始编写内容等等。接着，我们可以在本地运行起这个服务，除了查看博客的内容，还可以修改样式等等。完成上面的工作后，我们就可以生成静态内容，然后部署我们的应用到 GitHub Page 上。这一切看上去都完美，我们有两个不同的数据源——一个是 md 格式的文本，一个是最后生成的 HTML。它们已经实现了读写/分离：

但是作为一个前端开发人员，没有 JSON，用不了 Ajax 请求，我怎么把我的博客做成一个单页面应用？

编辑-发布-开发分离 因为我们需要交我们的博客转为 JSON，而不是一个 hexo 之类的格式。有了这些 JSON 文件的存在，我们就可以把 Git 当成一个 NoSQL 数据库。同时这些 JSON 文件也可以直接当成 API 来

其次，这些博客还需要 hexo 一样生成 HTML。

并且，开发人员在开发的时候不会影响到编辑的使用，于是就有了下面的架构：

在这其中我们有两种不同的数据形式，即存储着 Markdown 数据的 JSON 文件和最后生成的 HTML。

对博客数量不是很大的网站，或者说一般的网站来说，用上面的技术都不是问题。

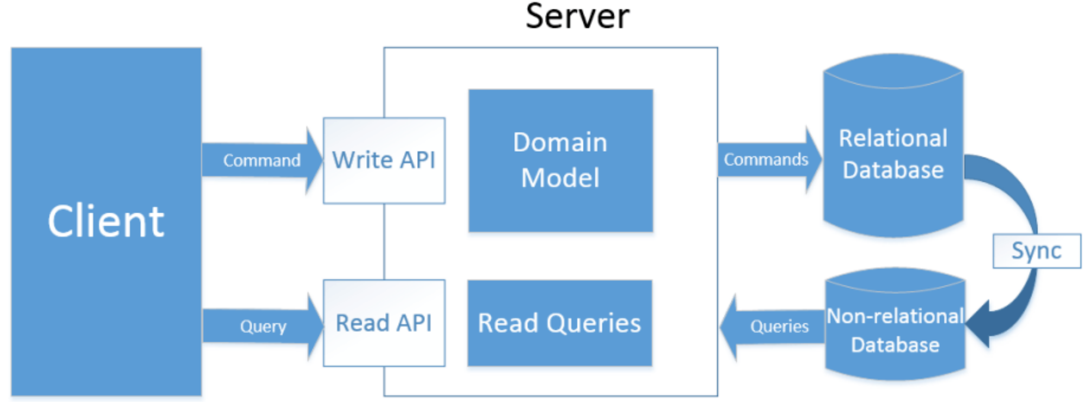


图 111: CQRS 进阶

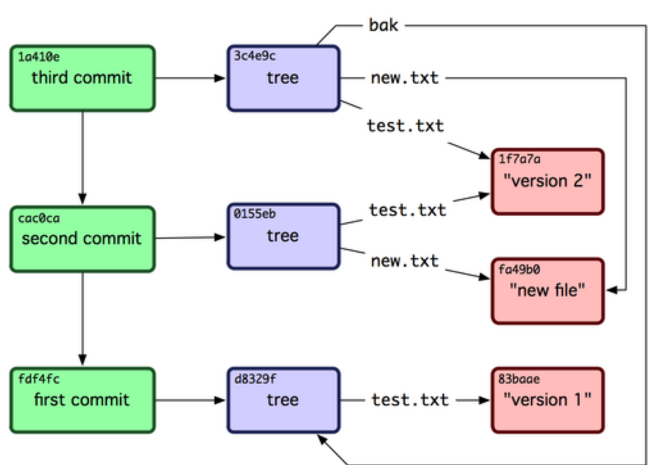


图 112: Git As NoSQL DB

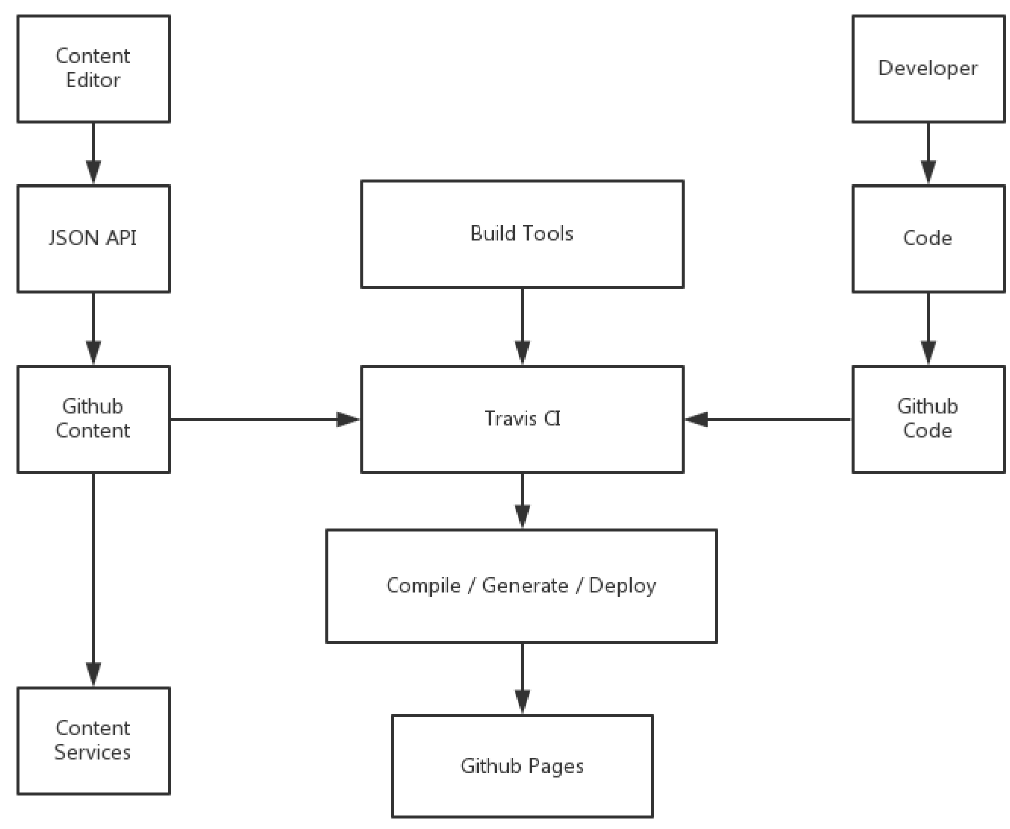


图 113: 基于 Git 的编辑-发布分离

然而有大量数据的网站怎么办？使用 EventBus：

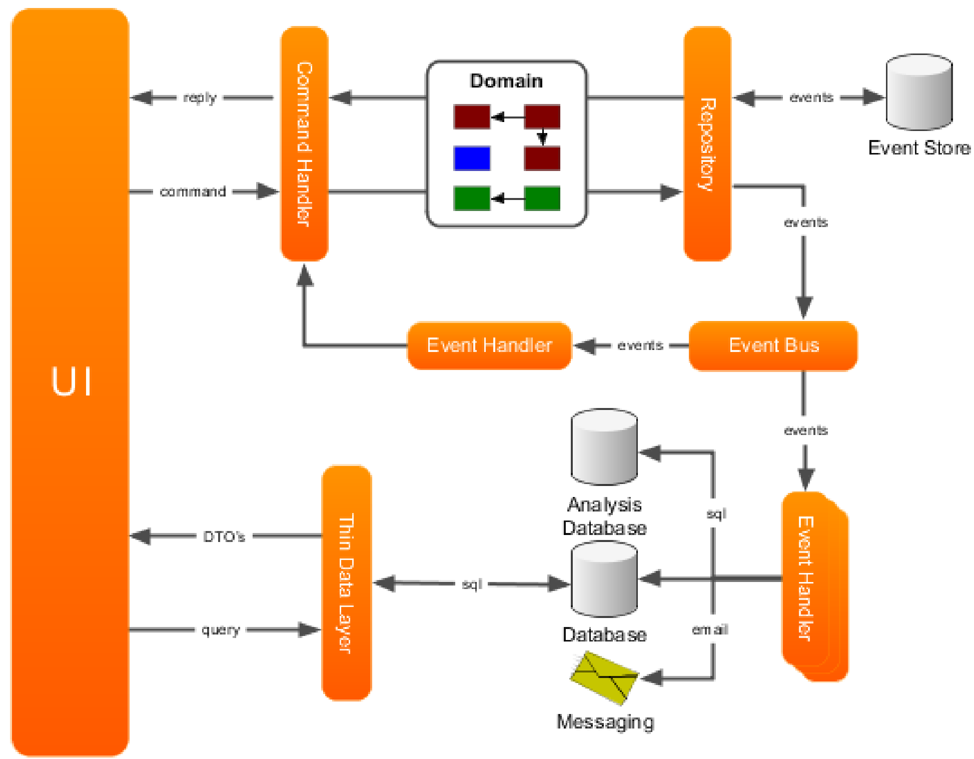


图 114: CQRS 和 EventBus

在我之前玩的一个 Demo 中，使用 Python 中的 Scrapy 爬虫来抓取现有的动态网站，并将其变成静态网站部署到 AWS S3 上。

但是上面仅仅只是实现了文章的显示，我们还存在一些问题：

1. 搜索功能
2. AutoComplete

等等的这些服务是没有用静态 API 来实现的。

CQRS 结合微服务

既然可以有这么多分法，并且我们都已经准备好分他们了。那么分了之后，我们就可以把他们都合到一起了。

Nginx as Dispatcher 最常见的解耦应用的方式中，就有一种是基于 Nginx 来分发 URL 请求。在这种情况下，对于 API 的使用者，或者最终用户来说，他们都是同一个 API。只是在后台里，这个 API 已经是不同的几个 API 组成，如下图所示：

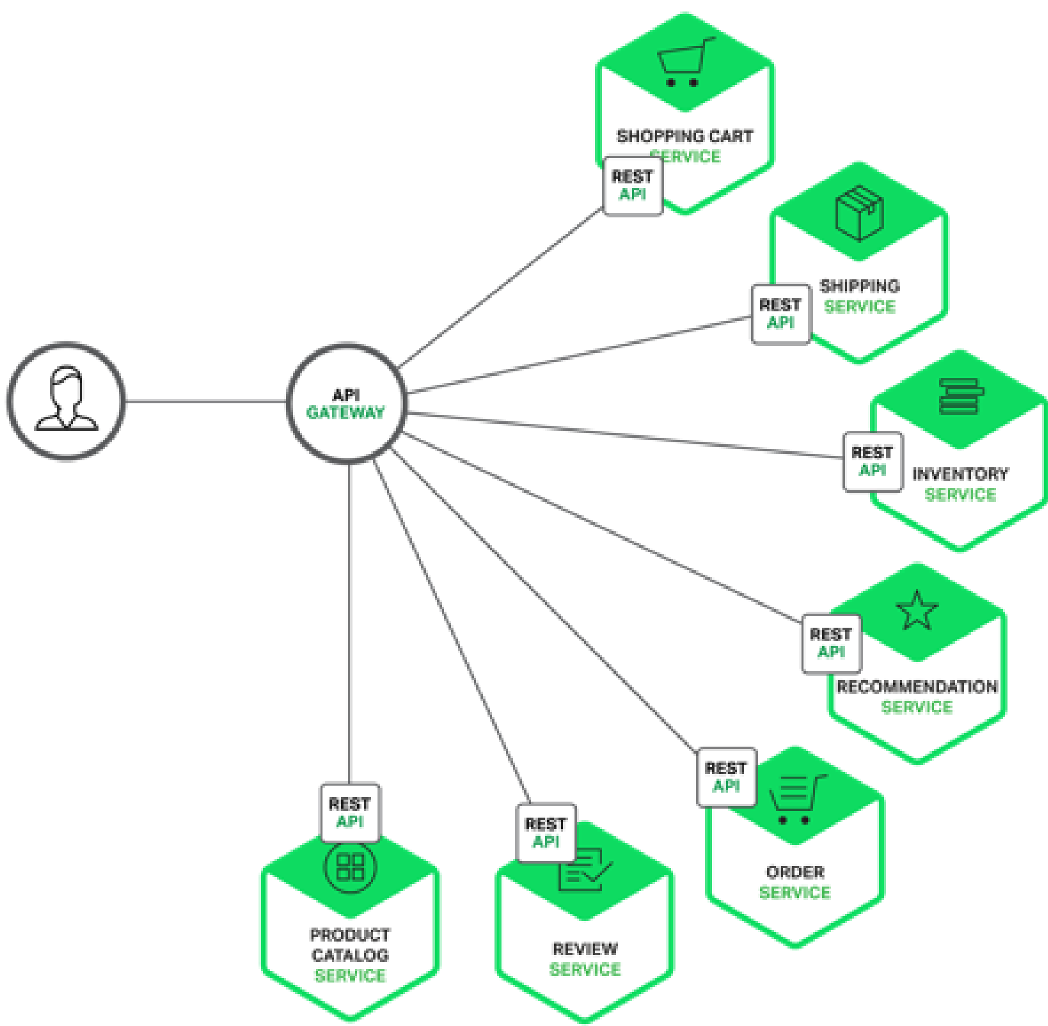


图 115: Nginx 解耦微服务

客户端的请求来到 API Gateway，根据不同的请求类型，这些 URL 被分发到不同的 Service，如 Review Service、Order Service 等等。

对于我们想要设计的系统来说也是如此，我们可以通过这个 Dispatcher 来解耦我们的服务。

CQRS 结合微服务 现在，我们想要的系统的雏形已经出现了。

从源头上来说，我们把能缓存的内容变成了静态的 HTML，通过 CDN 来分发。并且，我们还可以将把不同的服务独立出来。

从实现上来说，我们将博客的数据变成了两部分：一个以 Git + JSON 格式存在的 API，它除了可以用于生成 HTML，另外一部分作为 API 来使用。

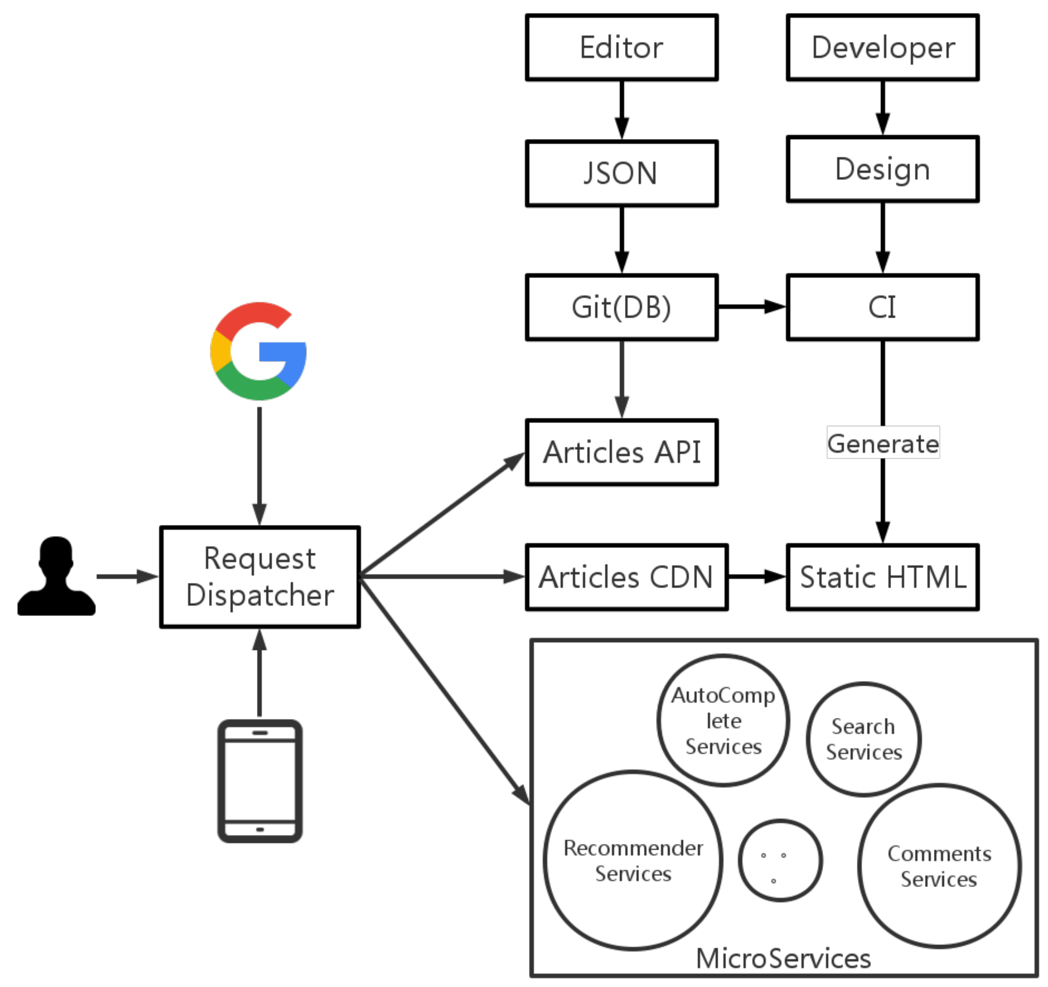


图 116: CQRS 结合微服务

最后，我们可以通过上面说到的 Nginx 或者 Apache 来当这里的 Request Dispatcher。