# 引言

回到一年前的今天(2014.09.29)，一边在准备着去沙漠之旅，一边在准备国庆后的印度培训。

当时我还在用我的Lumia 920，上面没有各式各样的软件，除了我最需要的地图、相机。所以，我需要为我的手机写一个应用，用于在地图上显示图片信息及照片。

今天Github已经可以支持geojson了，于是你可以看到我在之前生成的geojson在地图上的效果[gps.geojson](https://github.com/phodal-archive/onmap/blob/master/gps.geojson)。

## Re-Practise

在过去的近一年时期里，花费了很多时间在提高代码质量与构建架构知识。试着学习某一方面的架构知识，应用到某个熟悉领域。

1. 所谓的一万小时天才理论一直在说明练习的重要性，你需要不断地去练习。但是并不是说你练习了一万小时之后就可以让你成为一个专家，而练习是必须的。
2. 让我想起了在大学时代学的PID算法，虽然我没有掌握好控制领域的相关理论及算法，但是我对各种调节还算有点印象。简单地来说，我们需要不断调整自己的方向。

现在还存在的那些互联网公司或者说开源项目，我们会发现两个不算有趣的规律：

1. 一个一直在运行的软件。
2. 尝试了几个产品，最后找到了一个合适的方向。

我发现我属于不断尝试地类型。一直想构建一个开源软件，但是似乎一直没有找对合理的用户？但是，我们会发现上述地两者都在不断地retry，不断地retry归根于那些人在不断的repractise。与之成为反例的便是：

1. 一个成功发布几次的软件，但是最后失败了
2. 尝试了不同的几个产品，但是失败了

所谓的失败，就是你离开人世了。所以，在我们还活着的时候，我们总会有机会去尝试。在那之前，我们都是在不断地re-practise。

这让我想到了Linux，这算是一个不错地软件，从一开始就存活到了现在。但是有多少开源软件就没有这么幸运，时间在淘汰越来越多的过去想法。人们创造事物的能力也越来越强，但是那只是因为创造变得越来越简单。

在我们看到的那些走上人生巅峰的CEO，还都在不断地re-practise。

## 技术与业务

于是，我又再次回到了这样一个现实的问题。技术可以不断地练习，不断地调整方向。但是技术地成本在不断地降低，代码的长度在不断地降低。整个技术的门槛越来越低，新出现的技术总会让新生代的程序员获利。但是不可避免地，业务地复杂度并没有因此而降低。这就是一个复杂的话题，难道业务真的很复杂吗？

人们总会提及写好CSS很难，但是写好Java就是一件容易的事。因为每天我们都在用Java、JavaScript去写代码，但是我们并没有花费时间去学。

因为我们一直将我们的时候花费的所谓的业务上，我们可以不断地将一些重复的代码抽象成一个库。但是我们并没有花费过多的时间去整理我们的业务，作为程序员，我们切换工作很容易只是因为相同的技术栈。作为一些营销人员，他们从一个领域到一个新的领域，不需要过多的学习，因为本身是相通的。

技术本身是如此，业务本身也是如此。

从技术到技术-领域是一条难走通的路？

## 资讯爆炸

回顾到近几年出现的各种资讯程序——开发者头条、极客头条、掘金、博乐头条等等，他们帮助我们的是丰富我们的信息，而不是简化我们的信息。

作为一个开发人员，过去我们并不需要关注那么多的内容。如果我们没有关注那么多的点，那么我们就可以集中于我们的想法里。实现上，我们需要的是一个更智能的时代。

业务本身是一种重复，技术本身也是重复的。只是在某个特定的时刻，一个好的技术可以帮助我们更好地Re-Practise。如推荐算法本身依赖于人为对信息进行分类，但是我们需要去区分大量地信息。而人本身的经历是足够有险的，这时候就需要机器来帮我们做很多事。

今天我在用MX5，但是发现不及Lumia 1020来得安静。功能越强大的同时，意味着我在上面花费的时间会更多。事情有好的一面总会有不好的一面，不好的一面也就意味着有机会寻找好的一面。

我们需要摒弃一些东西，以重新纠正我们的方向。于是，我需要再次回到Lumia 1020上。

## Lost

一开始就输在起跑线上

这是一个很有意思的话题，尽管试图将本章中从书中删除，但是我还是忍了下来。如果你学得比别人晚，在很长的一段时间里(可能直到进棺材)输给别人是必然的——落后就要挨打。就好像我等毕业于一所二本垫底的学校里，如果在过去我一直保持着和别人(各种重点)一样的学习速度，那么我只能一直是Loser。

需要注意的是，对你来说考上二本很难，并不是因为你比别人笨。教育资源分配不均的问题，在某种程度上导致了新的阶级制度的出现。如我的首页说的那样: THE ONLY FAIR IS NOT FAIR——唯一公平的是它是不公平的。我们可以做的还有很多——CREATE & SHARE。真正的不幸是，因为营养不良导致的教育问题。如果你还有机会正常地思想，那说明这个世界对你还是公平的。

# 前端篇: 前端演进史

细细整理了过去接触过的那些前端技术，发现前端演进是段特别有意思的历史。人们总是在过去就做出未来需要的框架，而现在流行的是过去的过去发明过的。如，响应式设计不得不提到的一个缺点是：**他只是将原本在模板层做的事，放到了样式（CSS）层来完成**。

复杂度同力一样不会消失，也不会凭空产生，它总是从一个物体转移到另一个物体或一种形式转为另一种形式。

如果六、七年前的移动网络速度和今天一样快，那么直接上的技术就是响应式设计，APP、SPA就不会流行得这么快。尽管我们可以预见未来这些领域会变得更好，但是更需要的是改变现状。改变现状的同时也需要预见未来的需求。

### 什么是前端？

维基百科是这样说的：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。计算机程序的界面样式，视觉呈现属于前端。

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？

## 前端演进史

过去一直想整理一篇文章来说说前端发展的历史，但是想着这些历史已经被人们所熟知。后来发现并非如此，大抵是幸存者偏见——关注到的都知道这些历史。

### 数据-模板-样式混合

在有限的前端经验里，我还是经历了那段用Table来作样式的年代。大学期间曾经有偿帮一些公司或者个人开发、维护一些CMS，而Table是当时帮某个网站更新样式接触到的——ASP.Net（maybe)。当时，我们启动这个CMS用的是一个名为aspweb.exe的程序。于是，在我的移动硬盘里找到了下面的代码。

<TABLE cellSpacing=0 cellPadding=0 width=910 align=center border=0>  
 <TBODY>  
 <TR>  
 <TD vAlign=top width=188><TABLE cellSpacing=0 cellPadding=0 width=184 align=center border=0>  
 <TBODY>  
 <TR>  
 <TD><IMG src="Images/xxx.gif" width=184></TD></TR>  
 <TR>  
 <TD>  
 <TABLE cellSpacing=0 cellPadding=0 width=184 align=center   
 background=Images/xxx.gif border=0>

虽然，我也已经在HEAD里找到了现代的雏形——DIV + CSS，然而这仍然是一个Table的年代。

<LINK href="img/xxx.css" type=text/css rel=stylesheet>

**人们一直在说前端很难，问题是你学过么？？？**

**人们一直在说前端很难，问题是你学过么？？？**

**人们一直在说前端很难，问题是你学过么？？？**

也许，你也一直在说CSS不好写，但是CSS真的不好写么？人们总在说JS很难用，但是你学过么？只在需要的时候才去学，那肯定很难。**你不曾花时间去学习一门语言，但是却能直接写出可以work的代码，说明他们容易上手**。如果你看过一些有经验的Ruby、Scala、Emacs Lisp开发者写出来的代码，我想会得到相同的结论。有一些语言可以让写程序的人Happy，但是看的人可能就不Happy了。做事的方法不止一种，但是不是所有的人都要用那种方法去做。

过去的那些程序员都是**真正的全栈程序员**，这些程序员不仅仅做了前端的活，还做了数据库的工作。

Set rs = Server.CreateObject("ADODB.Recordset")  
sql = "select id,title,username,email,qq,adddate,content,Re\_content,home,face,sex from Fl\_Book where ispassed=1 order by id desc"  
rs.open sql, Conn, 1, 1  
fl.SqlQueryNum = fl.SqlQueryNum + 1

在这个ASP文件里，它从数据库里查找出了数据，然后Render出HTML。如果可以看到历史版本，那么我想我会看到有一个作者将style=""的代码一个个放到css文件中。

在这里的代码里也免不了有动态生成JavaScript代码的方法：

show\_other = "<SCRIPT language=javascript>"  
show\_other = show\_other & "function checkform()"  
show\_other = show\_other & "{"  
show\_other = show\_other & "if (document.add.title.value=='')"  
show\_other = show\_other & "{"

请尽情嘲笑，然后再看一段代码：

import React from "react";  
import { getData } from "../../common/request";  
import styles from "./style.css";  
  
  
export default class HomePage extends React.Component {  
 componentWillMount() {  
 console.log("[HomePage] will mount with server response: ", this.props.data.home);  
 }  
  
 render() {  
 let { title } = this.props.data.home;  
  
 return (  
 <div className={styles.content}>  
 <h1>{title}</h1>  
 <p className={styles.welcomeText}>Thanks for joining!</p>  
 </div>  
 );  
 }  
  
 static fetchData = function(params) {  
 return getData("/home");  
 }  
}

10年前和10年后的代码，似乎没有太多的变化。有所不同的是数据层已经被独立出去了，如果你的component也混合了数据层，即直接查询数据库而不是调用数据层接口，那么你就需要好好思考下这个问题。你只是在追随潮流，还是在改变。用一个View层更换一个View层，用一个Router换一个Router的意义在哪？

### Model-View-Controller

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 Folwer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);

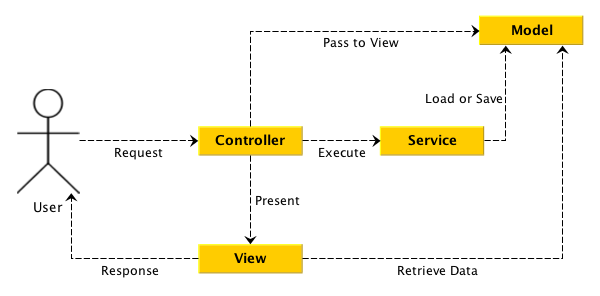
model文件夹和mappers文件夹都是数据层的一部分，只是两者间的职责不同，如：

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：

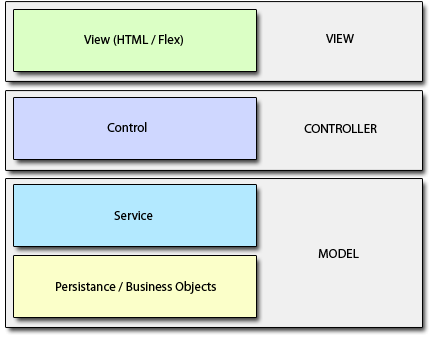
@RequestMapping(value = {"/disableUser"}, method = RequestMethod.POST)  
public ModelAndView processUserDisable(HttpServletRequest request, ModelMap model) {  
 String userName = request.getParameter("userName");  
 User user = userService.getByUsername(userName);  
 userService.disable(user);  
 Map<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：

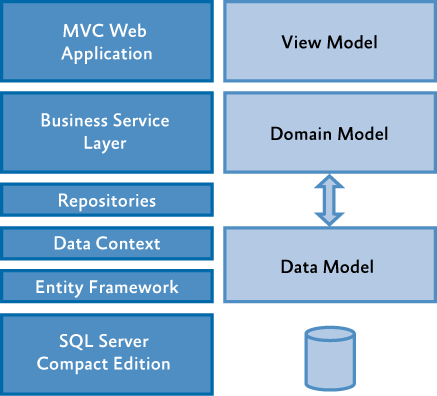


Service MVC

然而对于Domain相关的Service应该放在哪一层，总会有不同的意见：



MVC Player



MS 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：

if (isNewnameEmpty == false && newuser == null){  
 user.setUserName(newUsername);  
 List<Post> myPosts = postService.findMainPostByAuthorNameSortedByCreateTime(principal.getName());  
  
 for (int k = 0;k < myPosts.size();k++){  
 Post post = myPosts.get(k);  
 post.setAuthorName(newUsername);  
 postService.save(post);  
 }  
 userService.update(user);  
 Authentication oldAuthentication = SecurityContextHolder.getContext().getAuthentication();  
 Authentication authentication = null;  
 if(oldAuthentication == null){  
 authentication = new UsernamePasswordAuthenticationToken(newUsername,user.getPasswordHash());  
 }else{  
 authentication = new UsernamePasswordAuthenticationToken(newUsername,user.getPasswordHash(),oldAuthentication.getAuthorities());  
 }  
 SecurityContextHolder.getContext().setAuthentication(authentication);  
 map.clear();  
 map.put("user",user);  
 model.addAttribute("myPosts", myPosts);  
 model.addAttribute("namesuccess", "User Profile updated successfully");  
 return new ModelAndView("user/profile", map);  
}

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。

View层是一直在变化的层级，人们的品味一直在更新，有时甚至可能因为竞争对手而产生变化。在已经取得一定市场的情况下，Model-Service-Controller通常都不太会变动，甚至不敢变动。企业意识到创新的两面性，要么带来死亡，要么占领更大的市场。但是对手通常都比你想象中的更聪明一些，所以这时**开创新的业务是一个更好的选择**。

高速发展期的企业和发展初期的企业相比，更需要前端开发人员。在用户基数不够、业务待定的情形中，View只要可用并美观就行了，这时可能就会有大量的业务代码放在View层：

<c:choose>  
 <c:when test="${ hasError }">  
 <p class="prompt-error">  
 ${errors.username} ${errors.password}  
 </p>  
 </c:when>  
 <c:otherwise>  
 <p class="prompt">  
 Woohoo, User <span class="username">${user.userName}</span> has been created successfully!  
 </p>  
 </c:otherwise>  
</c:choose>

不同的情形下，人们都会对此有所争议，但只要符合当前的业务便是最好的选择。作为一个前端开发人员，在过去我需要修改JSP、PHP文件，这期间我需要去了解这些Template：

{foreach $lists as $v}  
<li itemprop="breadcrumb"><span{if(newest($v['addtime'],24))} style="color:red"{/if}>[{fun date('Y-m-d',$v['addtime'])}]</span><a href="{$v['url']}" style="{$v['style']}" target="\_blank">{$v['title']}</a></li>  
{/foreach}

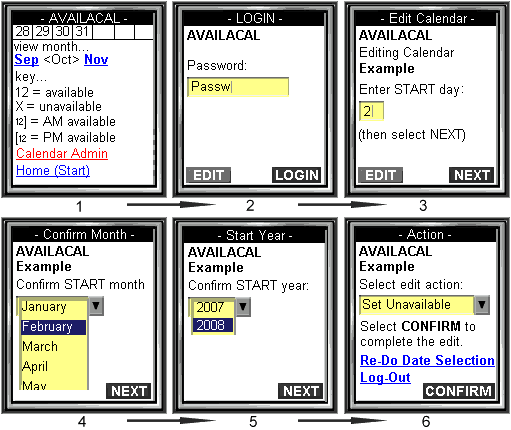
有时像Django这一类，自称为Model-Template-View的框架，更容易让人理解其意图：

{% for blog\_post in blog\_posts.object\_list %}  
{% block blog\_post\_list\_post\_title %}  
<section class="section--center mdl-grid mdl-grid--no-spacing mdl-shadow--2dp mdl-cell--11-col blog-list">  
{% editable blog\_post.title %}  
<div class="mdl-card\_\_title mdl-card--border mdl-card--expand">  
 <h2 class="mdl-card\_\_title-text">  
 <a href="{{ blog\_post.get\_absolute\_url }}" itemprop="headline">{{ blog\_post.title }} › </a>  
 </h2>  
</div>  
{% endeditable %}  
{% endblock %}

作为一个前端人员，我们真正在接触的是View层和Template层，但是MVC并没有说明这些。

### 从桌面版到移动版

Wap出现了，并带来了更多的挑战。随后，分辨率从1024x768变成了176×208，开发人员不得不面临这些挑战。当时所需要做的仅仅是修改View层，而View层随着iPhone的出现又发生了变化。



WAP 网站

这是一个短暂的历史，PO还需要为手机用户制作一个怎样的网站？于是他们把桌面版的网站搬了过去变成了移动版。由于网络的原因，每次都需要重新加载页面，这带来了不佳的用户体验。

幸运的是，人们很快意识到了这个问题，于是就有了SPA。**如果当时的移动网络速度可以更快的话，我想很多SPA框架就不存在了**。

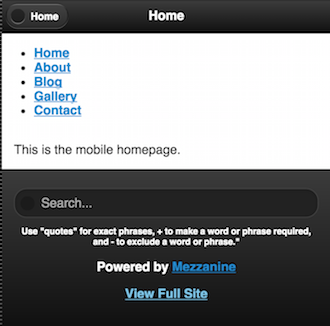
先说说jQuery Mobile，在那之前，先让我们来看看两个不同版本的代码，下面是一个手机版本的blog详情页：

<ul data-role="listview" data-inset="true" data-splittheme="a">  
 {% for blog\_post in blog\_posts.object\_list %}  
 <li>  
 {% editable blog\_post.title blog\_post.publish\_date %}  
 <h2 class="blog-post-title"><a href="{% url "blog\_post\_detail" blog\_post.slug %}">{{ blog\_post.title }}</a></h2>  
 <em class="since">{% blocktrans with sometime=blog\_post.publish\_date|timesince %}{{ sometime }} ago{% endblocktrans %}</em>  
 {% endeditable %}  
 </li>  
 {% endfor %}  
</ul>

而下面是桌面版本的片段：

{% for blog\_post in blog\_posts.object\_list %}  
{% block blog\_post\_list\_post\_title %}  
{% editable blog\_post.title %}  
<h2>  
 <a href="{{ blog\_post.get\_absolute\_url }}">{{ blog\_post.title }}</a>  
</h2>  
{% endeditable %}  
{% endblock %}  
{% block blog\_post\_list\_post\_metainfo %}  
{% editable blog\_post.publish\_date %}  
<h6 class="post-meta">  
 {% trans "Posted by" %}:  
 {% with blog\_post.user as author %}  
 <a href="{% url "blog\_post\_list\_author" author %}">{{ author.get\_full\_name|default:author.username }}</a>  
 {% endwith %}  
 {% with blog\_post.categories.all as categories %}  
 {% if categories %}  
 {% trans "in" %}  
 {% for category in categories %}  
 <a href="{% url "blog\_post\_list\_category" category.slug %}">{{ category }}</a>{% if not forloop.last %}, {% endif %}  
 {% endfor %}  
 {% endif %}  
 {% endwith %}  
 {% blocktrans with sometime=blog\_post.publish\_date|timesince %}{{ sometime }} ago{% endblocktrans %}  
</h6>  
{% endeditable %}  
{% endblock %}

人们所做的只是**重载View层**。这也是一个有效的SEO策略，上面这些代码是我博客过去的代码。对于桌面版和移动版都是不同的模板和不同的JS、CSS。



移动版网页

在这一时期，桌面版和移动版的代码可能在同一个代码库中。他们使用相同的代码，调用相同的逻辑，只是View层不同了。但是，每次改动我们都要维护两份代码。

随后，人们发现了一种更友好的移动版应用——APP。

### APP与过渡期API

这是一个艰难的时刻，过去我们的很多API都是在原来的代码库中构建的，即桌面版和移动版一起。我们已经在这个代码库中开发了越来越多的功能，系统开发变得臃肿。如《Linux/Unix设计思想》中所说，这是一个伟大的系统，但是它臃肿而又缓慢。

我们是选择重新开发一个结合第一和第二系统的最佳特性的第三个系统，还是继续臃肿下去。我想你已经有答案了。随后我们就有了APP API，构建出了博客的APP。



应用

最开始，人们越来越喜欢用APP，因为与移动版网页相比，其响应速度更快，而且更流畅。对于服务器来说，也是一件好事，因为请求变少了。

但是并非所有的人都会下载APP——**有时只想看看上面有没有需要的东西**。对于刚需不强的应用，人们并不会下载，只会访问网站。

有了APP API之后，我们可以向网页提供API，我们就开始设想要有一个好好的移动版。

### 过渡期SPA

Backbone诞生于2010年，和响应式设计出现在同一个年代里，但他们似乎在同一个时代里火了起来。如果CSS3早点流行开来，似乎就没有Backbone啥事了。不过移动网络还是限制了响应式的流行，只是在今天这些都有所变化。

我们用Ajax向后台请求API，然后Mustache Render出来。因为JavaScript在模块化上的缺陷，所以我们就用Require.JS来进行模块化。

下面的代码就是我在尝试对我的博客进行SPA设计时的代码：

define([  
 'zepto',  
 'underscore',  
 'mustache',  
 'js/ProductsView',  
 'json!/configure.json',  
 'text!/templates/blog\_details.html',  
 'js/renderBlog'  
],function($, \_, Mustache, ProductsView, configure, blogDetailsTemplate, GetBlog){  
  
 var BlogDetailsView = Backbone.View.extend ({  
 el: $("#content"),  
  
 initialize: function () {  
 this.params = '#content';  
 },  
  
 getBlog: function(slug) {  
 var getblog = new GetBlog(this.params, configure['blogPostUrl'] + slug, blogDetailsTemplate);  
 getblog.renderBlog();  
 }  
 });  
  
 return BlogDetailsView;  
});

从API获取数据，结合Template来Render出Page。但是这无法改变我们需要Client Side Render和Server Side Render的两种Render方式，除非我们可以像淘宝一样不需要考虑SEO——因为它不那么依靠搜索引擎带来流量。

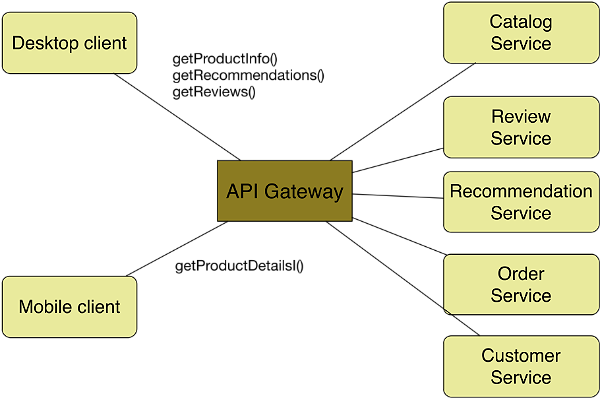
这时，我们还是基于类MVC模式。只是数据的获取方式变成了Ajax，我们就犯了一个错误——将大量的业务逻辑放在前端。这时候我们已经不能再从View层直接访问Model层，从安全的角度来说有点危险。

如果你的View层还可以直接访问Model层，那么说明你的架构还是MVC模式。之前我在Github上构建一个Side Project的时候直接用View层访问了Model层，由于Model层是一个ElasticSearch的搜索引擎，它提供了JSON API，这使得我要在View层处理数据——即业务逻辑。将上述的JSON API放入Controller，尽管会加重这一层的复杂度，但是业务逻辑就不再放置于View层。

如果你在你的View层和Model层总有一层接口，那么你采用的就是MVP模式——MVC模式的衍生（PS：为了区别别的事情，总会有人取个表意的名称）。

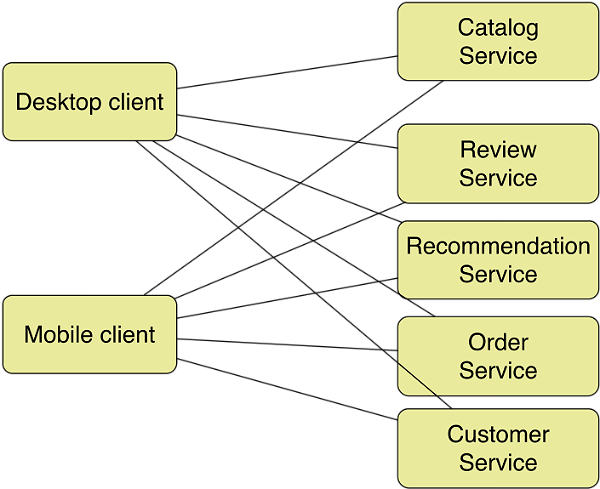
一夜之前，我们又回到了过去。我们离开了JSP，将View层变成了Template与Controller。而原有的Services层并不是只承担其原来的责任，这些Services开始向ViewModel改变。

一些团队便将Services抽成多个Services，美其名为微服务。传统架构下的API从下图



API Gateway

变成了直接调用的微服务：



Micro Services

对于后台开发者来说，这是一件大快人心的大好事，但是对于应用端/前端来说并非如此。调用的服务变多了，在应用程序端进行功能测试变得更复杂，需要Mock的API变多了。

### Hybird与ViewModel

这时候遇到问题的不仅仅只在前端，而在App端，小的团队已经无法承受开发成本。人们更多的注意力放到了Hybird应用上。Hybird应用解决了一些小团队在开发初期遇到的问题，这部分应用便交给了前端开发者。

前端开发人员先熟悉了单纯的JS + CSS + HTML，又熟悉了Router + PageView + API的结构，现在他们又需要做手机APP。这时候只好用熟悉的jQuer Mobile + Cordova。

随后，人们先从Cordova + jQuery Mobile，变成了Cordova + Angular的 Ionic。在那之前，一些团队可能已经用Angular代换了Backbone。他们需要更好的交互，需要data binding。

接着，我们可以直接将我们的Angular代码从前端移到APP，比如下面这种博客APP的代码：

.controller('BlogCtrl', function ($scope, Blog) {  
 $scope.blogs = null;  
 $scope.blogOffset = 0;  
 //  
 $scope.doRefresh = function () {  
 Blog.async('https://www.phodal.com/api/v1/app/?format=json').then(function (results) {  
 $scope.blogs = results.objects;  
 });  
 $scope.$broadcast('scroll.refreshComplete');  
 $scope.$apply()  
 };  
  
 Blog.async('https://www.phodal.com/api/v1/app/?format=json').then(function (results) {  
 $scope.blogs = results.objects;  
 });  
  
 $scope.loadMore = function() {  
 $scope.blogOffset = $scope.blogOffset + 1;  
 Blog.async('https://www.phodal.com/api/v1/app/?limit=10&offset='+ $scope.blogOffset \* 20 + '&format=json').then(function (results) {  
 Array.prototype.push.apply($scope.blogs, results.objects);  
 $scope.$broadcast('scroll.infiniteScrollComplete');  
 })  
 };  
 })

结果**时间轴又错了**，人们总是**超前一个时期做错了一个在未来是正确的决定**。人们遇到了网页版的用户授权问题，于是发明了JWT——Json Web Token。

然而，由于WebView在一些早期的Android手机上出现了性能问题，人们开始考虑替换方案。接着出现了两个不同的解决方案：

1. React Native
2. 新的WebView——Crosswalk

开发人员开始欢呼React Native这样的框架。但是，他们并没有预见到**人们正在厌恶APP**，APP在我们的迭代里更新着，可能是一星期，可能是两星期，又或者是一个月。谁说APP内自更新不是一件坏事，但是APP的提醒无时无刻不在干扰着人们的生活，噪声越来越多。**不要和用户争夺他们手机的使用权**

### 一次构建，跨平台运行

在我们需要学习C语言的时候，GCC就有了这样的跨平台编译。

在我们开发桌面应用的时候，QT就有了这样的跨平台能力。

在我们构建Web应用的时候，Java就有了这样的跨平台能力。

在我们需要开发跨平台应用的时候，Cordova就有了这样的跨平台能力。

现在，React这样的跨平台框架又出现了，而响应式设计也是跨平台式的设计。

响应式设计不得不提到的一个缺点是：**他只是将原本在模板层做的事，放到了样式（CSS）层**。你还是在针对着不同的设备进行设计，两种没有什么多大的不同。复杂度不会消失，也不会凭空产生，它只会从一个物体转移到另一个物体或一种形式转为另一种形式。

React，将一小部分复杂度交由人来消化，将另外一部分交给了React自己来消化。在用Spring MVC之前，也许我们还在用CGI编程，而Spring降低了这部分复杂度，但是这和React一样降低的只是新手的复杂度。在我们不能以某种语言的方式写某相关的代码时，这会带来诸多麻烦。

## RePractise

如果你是一只辛勤的蜜蜂，那么我想你应该都玩过上面那些技术。你是在练习前端的技术，还是在RePractise？如果你不花点时间整理一下过去，顺便预测一下未来，那么你就是在白搭。

前端的演进在这一年特别快，Ruby On Rails也在一个合适的年代里出现，在那个年代里也流行得特别快。RoR开发效率高的优势已然不再突显，语法灵活性的副作用就是运行效率降低，同时后期维护难——每个人元编程了自己。

如果不能把Controller、Model Mapper变成ViewModel，又或者是Micro Services来解耦，那么ES6 + React只是在现在带来更高的开发效率。而所谓的高效率，只是相比较而意淫出来的，因为他只是一层View层。将Model和Controller再加回View层，以后再拆分出来？

现有的结构只是将View层做了View层应该做的事。

首先，你应该考虑的是一种可以让View层解耦于Domain或者Service层。今天，桌面、平板、手机并不是唯一用户设备，虽然你可能在明年统一了这三个平台，现在新的设备的出现又将设备分成两种类型——桌面版和手机版。一开始桌面版和手机版是不同的版本，后来你又需要合并这两个设备。

其次，你可以考虑用混合Micro Services优势的Monolithic Service来分解业务。如果可以举一个成功的例子，那么就是Linux，一个混合内核的“Service”。

最后，Keep Learning。我们总需要在适当的时候做出改变，尽管我们觉得一个Web应用代码库中含桌面版和移动版代码会很不错，但是在那个时候需要做出改变。

对于复杂的应用来说，其架构肯定不是只有纯MVP或者纯MVVM这么简单的。如果一个应用混合了MVVM、MVP和MVC，那么他也变成了MVC——因为他直接访问了Model层。但是如果细分来看，只有访问了Model层的那一部分才是MVC模式。

模式，是人们对于某个解决方案的描述。在一段代码中可能有各种各样的设计模式，更何况是架构。

# 后台与服务篇

尽管在最初我也想去写一篇文章来说说后台的发展史，后来想了想还是让我们把它划分成不同的几部分。以便于我们可以更好的说说这些内容，不过相信这是一个好的开始。

## RESTful与服务化

### 设计RESTful API

REST从资源的角度来观察整个网络，分布在各处的资源由URI确定，而客户端的应用通过URI来获取资源的表征。获得这些表征致使这些应用程序转变了其状态。随着不断获取资源的表征，客户端应用不断地在转变着其状态，所谓表征状态转移。

因为我们需要的是一个Machine到Machine沟通的平台，需要设计一个API。而设计一个API来说，RESTful是很不错的一种选择，也是主流的选择。而设计一个RESTful服务，的首要步骤便是设计资源模型。

### 资源

互联网上的一切信息都可以看作是一种资源。

|  |  |
| --- | --- |
| HTTP Method | Operation Performed |
| GET | Get a resource (Read a resource) |
| POST | Create a resource |
| PUT | Update a resource |
| DELETE | Delete Resource |

设计RESTful API是一个有意思的话题。下面是一些常用的RESTful设计原则:

* 组件间交互的可伸缩性
* 接口的通用性
* 组件的独立部署
* 通过中间组件来减少延迟、实施安全策略和封装已有系统

判断是否是 RESTful的约束条件

* 客户端-服务器分离
* 无状态
* 可缓存
* 多层系统
* 统一接口
* 随需代码（可选）

## 微服务

### 微内核

这只是由微服务与传统架构之间对比而引发的一个思考，让我引一些资料来当参考吧.

单内核：也称为宏内核。将内核从整体上作为一个大过程实现，并同时运行在一个单独的地址空间。所有的内核服务都在一个地址空间运行，相互之间直接调用函数，简单高效。微内核：功能被划分成独立的过程，过程间通过IPC进行通信。模块化程度高，一个服务失效不会影响另外一个服务。Linux是一个单内核结构，同时又吸收了微内核的优点：模块化设计，支持动态装载内核模块。Linux还避免了微内核设计上的缺陷，让一切都运行在内核态，直接调用函数，无需消息传递。

对就的微内核便是:

微内核――在微内核中，大部分内核都作为单独的进程在特权状态下运行，他们通过消息传递进行通讯。在典型情况下，每个概念模块都有一个进程。因此，假如在设计中有一个系统调用模块，那么就必然有一个相应的进程来接收系统调用，并和能够执行系统调用的其他进程（或模块）通讯以完成所需任务。

如果读过《操作系统原理》及其相关书籍的人应该很了解这些，对就的我们就可以一目了然地解决我们当前是的微服务的问题。

文章的来源是James Lewis与Martin Fowler写的[Microservices](http://martinfowler.com/articles/microservices.html)。对就于上面的

* monolithic kernel
* microkernel

与文中的

* monolithic services
* microservices

我们还是将其翻译成微服务与宏服务。

引起原文中对于微服务的解释:

简短地说，微服务架构风格是一种使用一套小服务来开发单个应用的方式途径，每个服务运行在自己的进程中，通过轻量的通讯机制联系，经常是基于HTTP资源API，这些服务基于业务能力构建，能够通过自动化部署方式独立部署，这些服务自己有一些小型集中化管理，可以是使用不同的编程语言编写，正如不同的数据存储技术一样。

原文是:

In short, the microservice architectural style [1] is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare mininum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

而关于微服务的提出是早在2011年的5月份

The term "microservice" was discussed at a workshop of software architects near Venice in May, 2011 to describe what the participants saw as a common architectural style that many of them had been recently exploring.

简单地与微内核作一些对比。微内核，**微内核部分经常只但是是个消息转发站**，而微服务从某种意义上也是如此，他们都有着下面的优点。

* 有助于实现模块间的隔离
* 在不影响系统其他部分的情况下，用更高效的实现代替现有文档系统模块的工作将会更加容易。

对于微服务来说

* 每个服务本身都是很简单的
* 对于每个服务，我们可以选择最好和最合适的工具来开发
* 系统本质上是松耦合的
* 不同的团队可以工作在不同的服务中
* 可以持续发布，而其他部分还是稳定的

从某种意义上来说微服务更适合于大型企业架构，而不是一般的应用，对于一般的应用来说他们的都在同一台主机上。无力于支付更多的系统开销，于是如**微服务不是免费的午餐**一文所说

* 微服务带来很多的开销操作
* 大量的DevOps技能要求
* 隐式接口
* 重复努力
* 分布式系统的复杂性
* 异步性是困难的！
* 可测试性挑战

因而不得不再后面补充一些所知的额外的东西。

针对于同样的话题，开始了解其中的一些问题。当敏捷的思想贯穿于开发过程时，我们不得不面对持续集成与发布这样的问题。我们确实可以在不同的服务下工作，然而当我们需要修改API时，就对我们的集成带来很多的问题。我们需要同时修改两个API！我们也需要同时部署他们！

## 混合微服务

在设计所谓的"Next-Generation CMS"，即Echoes CMS的时候，对于我这种懒得自己写Django App的人来说，通过我会去复制别人的代码，于是我继续在Github上漫游。接着找到了DjangoProject.com的源码，又看了看Mezzanine(ps: 我博客用的就是这个CMS)。于是从DjangoProject复制了Blog的代码，从Mezzanine复制了conf的代码，然后就有了Echoes的codebase。然后，继之前的文章(《微服务的小思考》我想了想， 这不就是我想要的模型么?

微服务与Django

Django 应用架构 Django MVC结构如下如示:



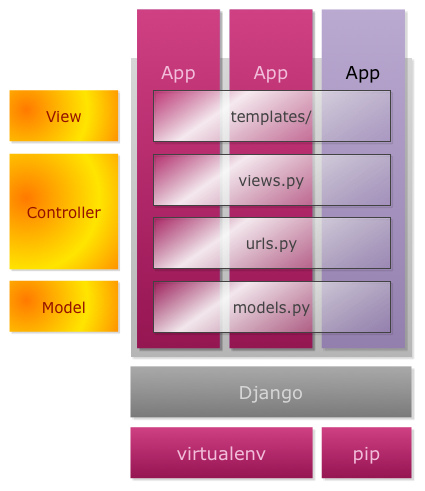
Django MVC

然后，记住这张图，忘记上面的MVC，Django实际上是一个MTV

* Model
* Template
* View

主要是Django中的views.py通常是在做Controller的事。

然而对于一个Django的应用来说，他的架构如下所示:



Django apps architecture

Django的每个App就代表着程序的一个功能。每个App有自己的models、views、urls、templates所以对于一个app来说他的结构如下:

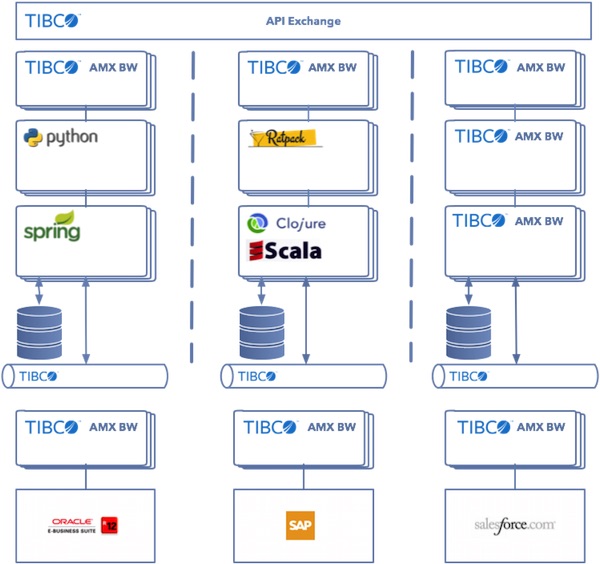
.  
|\_\_\_\_\_\_init\_\_.py  
|\_\_\_\_models.py  
|\_\_\_\_tests.py  
|\_\_\_\_views.py

如果是新版的Django那么它的结构如下:

.  
|\_\_\_\_\_\_init\_\_.py  
|\_\_\_\_admin.py  
|\_\_\_\_migrations  
| |\_\_\_\_\_\_init\_\_.py  
|\_\_\_\_models.py  
|\_\_\_\_tests.py  
|\_\_\_\_views.py

上面少了templates，最后会有一个总的URL，即第一张图的URL Dispatcher。接着，让我们看看微服务是怎样的。

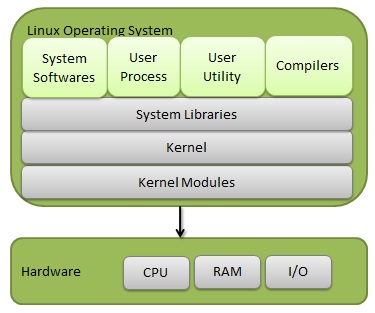
一个典型的微服务如下所示:



microservices architecture

有不同的技术栈python、spring、scala，但是他们看上去和Django应用的图差不多，除了数据库不一样。

与其将复杂的测试、逻辑部分变得不可测，不如把这些部分放置于系统内部。



Linux OS Hybrid

当我们在我们的服务器上部署微服务的时候，也就意味着实现所以的服务都是在我们系统的内部，我们有一个Kernel以及他们的Kernel Moduels，即微服务群们。他们调用DB，或者某些第三方服务。

System Libraries相当于我们的URL Dispatcher。而我们的URL Dispatcher实际上所做的便是将各自调用的服务指向各自的app。

这样我们即可以解决部署的问题，又可以减少内部耦合。

## 其他

我猜，微服务的流行是因为程序员可以欢乐地使用自己的语言，哪怕是Logo。

参考

[Microservices - Not A Free Lunch!](http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html)

[Microservices](http://martinfowler.com/articles/microservices.html)

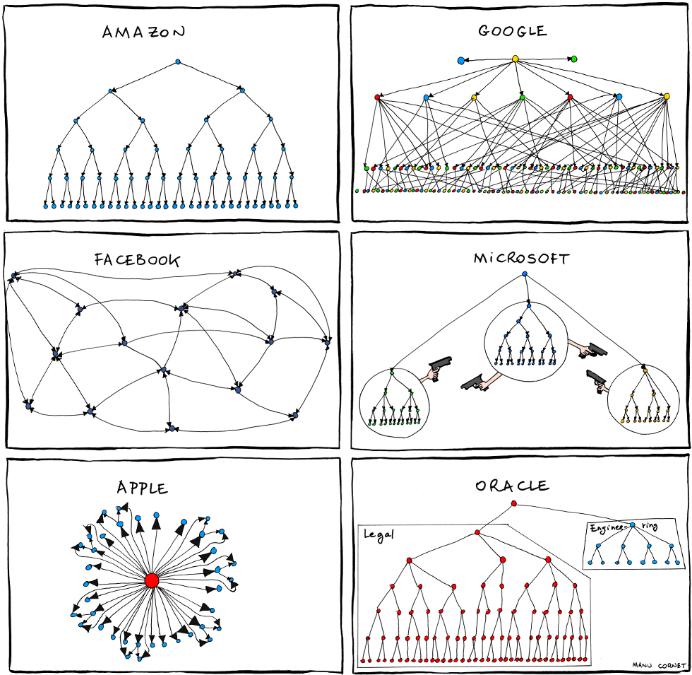
# 前后端篇

## 前后端分离

这是一个很古老的话题，对于大公司来说就是部门大了，需要拆分。因此开始之前，先提一下“康威定律”：

Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations.

换成中文，即：**设计系统的组织，其产生的设计和架构等价于组织间的沟通结构**。上图



Conway

这张图可以解释相当多的软件开发过程中的问题，而我们知道软件开发的主要问题是沟通问题。组织结构影响了我们的沟通结构，进而影响了我们的软件系统结构。好吧，我承认可能离题有点远。不过，我想说的是组织结构可能不允许我们做出一些好的系统架构。

如我们在《[RePractise前端篇: 前端演进史](http://mp.weixin.qq.com/s?src=3&timestamp=1467515714&ver=1&signature=z1onJvKn4TSrUmXm384CQUF1IZBVsLShsQ4DpmumN6xY0Gm5RR9XKdbf6ELzdRqg-mxdtxceTg-4-KrhYHZQC6wiSEWsP64vh0sl2Je4G16hnS6MsuZaD-u01HAENCSKuw3mQL-F2Gc5WYvti9SQlw==)》中提到的那样：我们已经有了一个桌面版网页，然后我们打造了一个APP。然而，总有些客户想在手机上浏览但是又不想下APP，我们就需要一个移动版。为什么会这样？因为用户已经被养成了这样的习惯，大部分的网站提到了桌面版、移动版、APP。要维护这样的三个不同的系统，对于大部分的业务公司来说成本太高了。

于是，大部分公司来说解决方案就是 后台 + 大前端 （桌面前端、移动Web、手机APP）。Angular和React就是为了解决这样的问题，而出现了不同的解决方案——基于Angular.js的混合应用框架Ionic、以及React Native。不过在当前，我对React Native的共用UI还是持观望态度。有人可能会提到Vue和Weex，但是我觉得并没有那么好用。或许是因为我接触React比较早，我觉得Vue的语法四不像。

在这样的情形下，我们只需要几个后台开发人员和几个前端开发人员就可以完成系统的设计了。这种前端开发人员就是最近几年人们“最想要”的。

## 单页面应用后台渲染

我已经想不到一个好的关于前端分享的主题了，于是联想到最近想要做的一件事，就想到了这个标题。或许这是一个好的主题，又或许这不是一个好的主题。但是至少我可以Share一下我的经验：

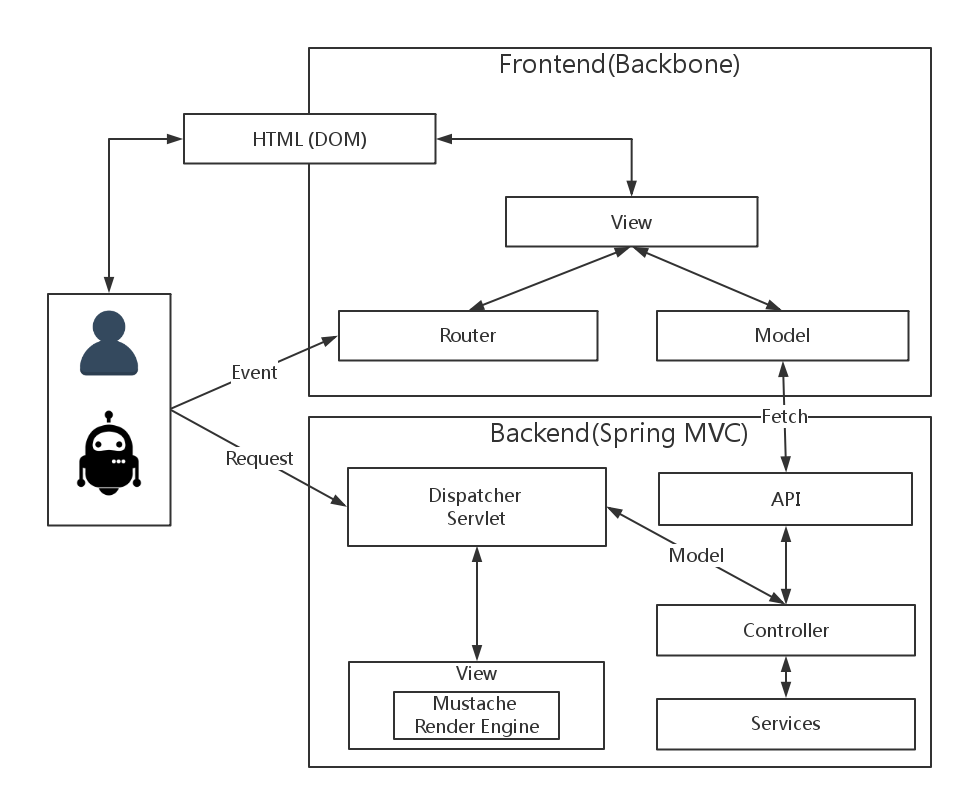
* 基于Mustache模板引擎的前后台渲染。
* 基于PreRender方式的Angular.js应用的后台渲染
* 服务端渲染的React

开始之前，我希望即使你们需要后台渲染，你们也应该前后端分离！由**后台**来提供API数据，前端用自己的后台来渲染页面。听上去有点绕，简单的来说就是不要把大量的业务逻辑放前台来，只把显示逻辑放在前台上。这样一来，即使有一天我们换了新的前端，如移动应用，那么我们的后台也是可用的。。

### 前后台渲染同一模板

我接触的第一个SPA应用是一个基于Spring MVC和Backbone的移动网站，但是它比一般的SPA应该要复杂——由于SEO的缘故，它需要支持后台渲染。

当搜索引擎通过URL访问我们的网站的时候，我们就需要返回相应的HTML。这意味着我们需要在后台有对应的模板引擎来支持，而由于SPA的性质又决定了，这需要使用一个纯前端的模板引擎。因此，我们并不能使用两个模板引擎来做这件事，维护两套模板注定会是一件痛苦的事，并且当时还没有React这种模板引擎在。不过，后来我们发现维护两种不同的渲染方式也是一件痛苦的事。因此，我们就会有了类似于下图的架构：



Spring MVC Backbone

我们在后台使用Spring MVC作为基础架构、Mustache作为模板引擎，和使用JSP作为模板引擎相比没有多大的区别——由Controller去获取对应的Model，再渲染给用户。多数时候搜索引擎都是依据Sitemap来进行索引的，所以我们的后台很容易就可以处理这些请求。同样的当用户访问相应的页面的时候，也返回同样的页面内容。当完成页面渲染的时候，就交由Backbone来处理相应的逻辑了。换句话来说，从这时候它就变成了一个单页面应用。

尽管这是一个三年年前开始的项目，但是在今天看来，这种做法仍然相应地有趣： 大部分的单页面应用只有一个首页，并由HTTP服务器（如Nginx）、Web框架（如Express、Koa）对路由做一些处理，可以让用户通过特定地URL访问特定地页面。而我们需要保证所有的用户访问地都是真实的页面，既然JavaScript没有加载完，用户也能看到完整的页面。

在这个项目里，最大的挑战就是如何保证后台渲染和前台渲染的业务逻辑是一样的。如当我们想要针对不同的产品显示不同的内容时，我们就需要在JavaScript中赋予一些逻辑，我们还需要在Java在有同样的逻辑。相比于在同一个代码里有桌面版、移动版来说，逻辑有更加复杂的趋势——因为在这种情况下，我们只需要维护两个不同的模板即可。而在SPA的情况下，我们要维护**两套逻辑**。后来，这个框架交由下文中的React与响应式设计重写。

在今天你仍然可以使用这样的方式来渲染，JDK 1.8自带了嵌入式JavaScript引擎Nashorn，完成支持ECMAScript 5.1规范以及一些扩展。

### PreRender方式

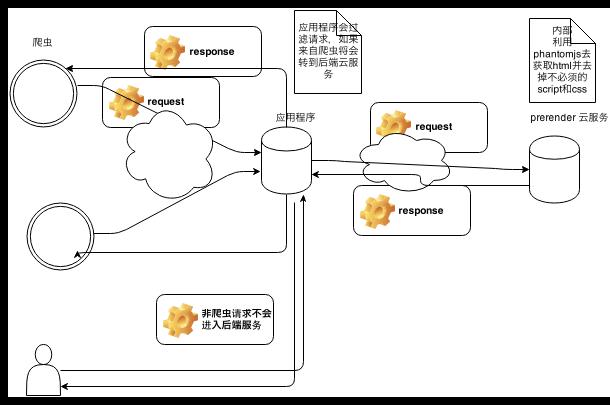
在我们重新设计系统的时候，曾经考虑过类似的做法。将我们的所有页面渲染成静态的HTML，然后用爬虫抓取我们的所有页面，再上传到AWS即可。当时我们咨询了其他小组的做法，其中有一个小组正是采用了这种PreRender的方式——在本地运行起一个Server，由PhantomJS来渲染页面，再保存为对应的HTML。

PreRender就是预先渲染好HTML，并针对于爬虫返回特定的HTML。（PS：不过作为一个很有经验的SEO开发人员，我一点不喜欢这种作法。要知道Google有时候会模拟成真实的用户，不带有爬虫的那些参数和标志，去访问页面。如果你返回给Google的两个页面差异太大——可能是你忘记更新了频率，那么Google可能就会认为你在**作弊**。）



PreRender

对于一般用户来说就不会返回后台渲染的结果了：



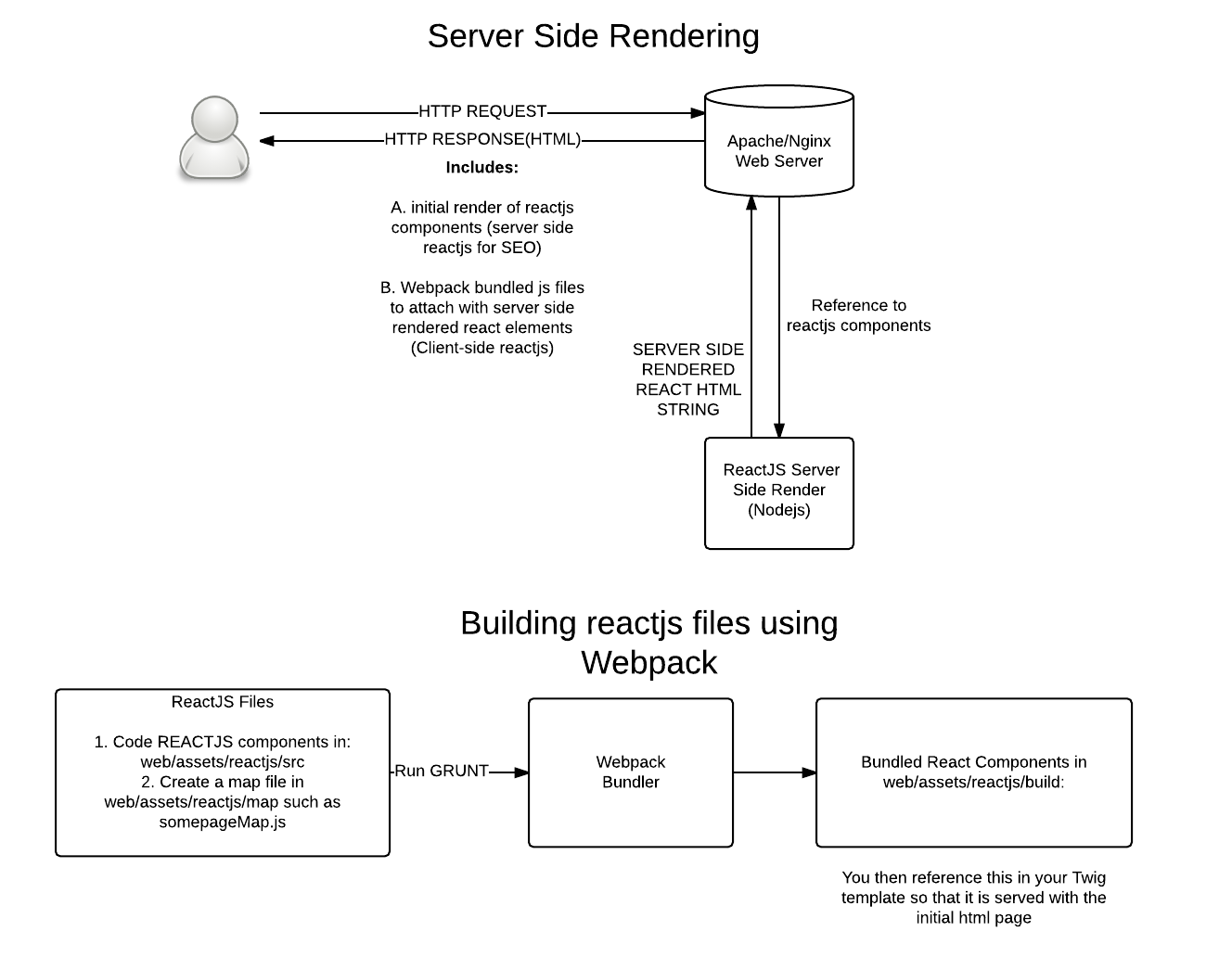
Angular PreRender

和上面的第一种情况相比，这种作法可以大大减少服务器地负担，并且可以直接交由CDN就可以了。这时我们只需要考虑要渲染哪些页面即可，对于数据量比较少的网站来说这是一个不错的做法，但是多了就不一样了。

对于我们来说，有两个问题：一个是速度的问题，他们有上万条数据就需要近一天左右的时间来生成（渲染时间长），而我们有上百万条数据。二是数据的实时问题，我们的产品数据每天都会更新一次。

### React

对于使用React的开发人员来说，要处理后台渲染就是一种更简单的事，毕竟React中提供了一个方法叫 renderToString()。我们所要做的就是用Express或者Koa对路由进行处理，然后返回对应的内容即可：



React Server Side Render

然后，剩下的事都可以交由React来解决，就是这么简单。

因为在这个时候我们在前后台使用的都是JavaScript，我们可以在这个地方直接实现对数据库的操作，就会出现我们在开头说到的前后台分离的问题。这样做并不合理，后台只应该返回我们所需要的数据，并且它可以随时被其他语言替换掉。

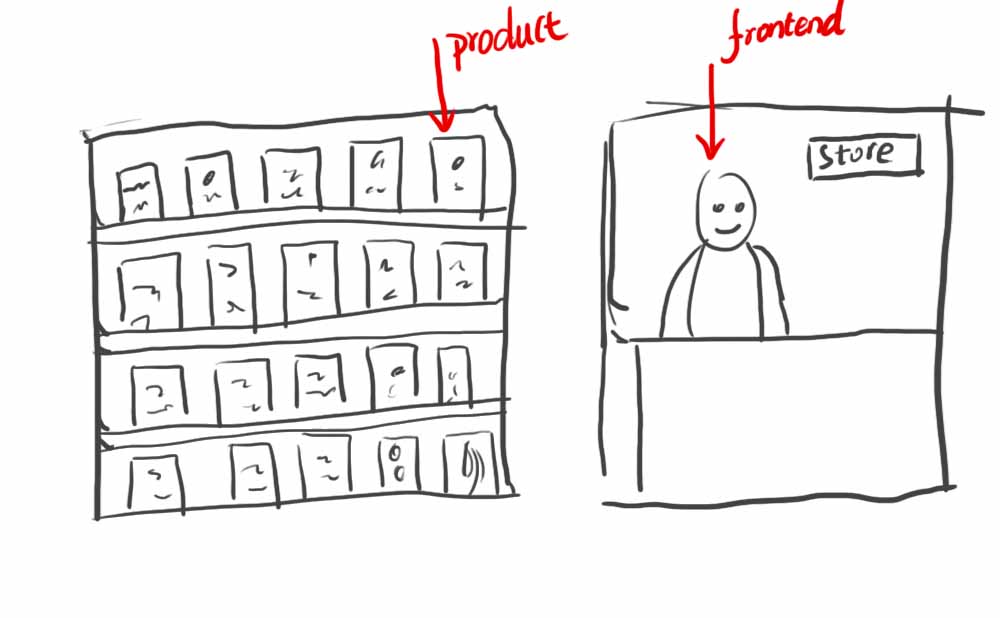
# 从真实世界到前后端

RePractise终于又迎来了新的一篇，要知道上一篇可是在半年前呢——《Repractise前端篇: 前端演进史 》。照RePractise惯例，这又是一篇超长文以及个人的扯淡过程。

当然这也是一个神奇的标题，因为我已经想不到一个好的名字了，不过先这样吧。这篇文章算是我最近两三个月的一篇思考。在上一个项目的打杂生涯里，我开始去学习架构方面的知识，开始去接触DDD的思想。从编码到架构，再回到实际的编码中，总会有很多的灵感闪现。

## 从真实世界到前后端

我们所写的代码在某种程度上都反应了真实世界的模型、行为等等。一个比较常见的模型就是：购物模型。同时， 这也是一个很好的展示前后端分离的模型。



store-model.jpg

（PS: 原谅我的画工）

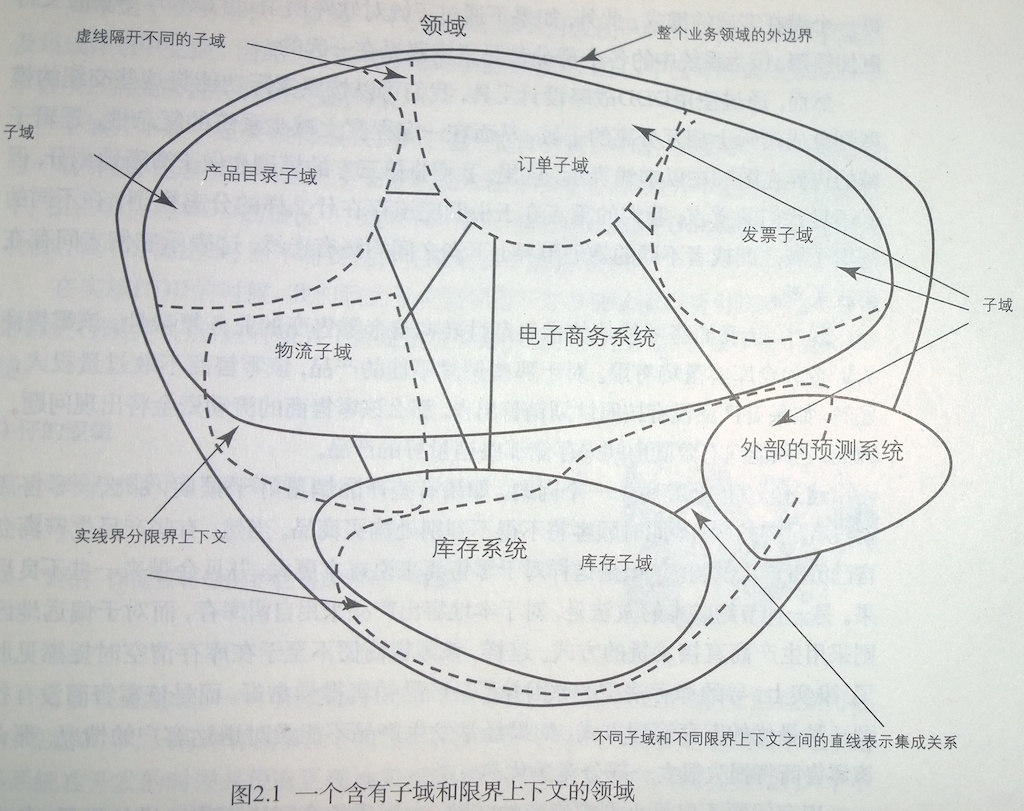
### 便利店与售货员

对于一般的便利店来说，只有一个销售员，ta负责整个商店的一系列事务。从某种意义上来说，ta就是整个系统的核心，负责了系统的业务和事件。

一般来说在一个购买流程里，会有三个主要的人或物：

* 售货员。一般来说，ta只会在最后的结账流程中出以及顾客询问时做出响应。
* 货物。没啥可解释的，就是一堆模型。0
* 顾客 。浏览商店、对比商店、blabla等等。

如果我们要构建这样一个系统，我们只需要区分出系统的各个部分，那么剩下的事情就变得很简单了。



domain.jpg

由于整个系统仍然是相当复杂的，我们在这里只关注于用户购买的过程。

### 模型、领域、抽象

从购买过程来说，顾客所要做的事情就是：

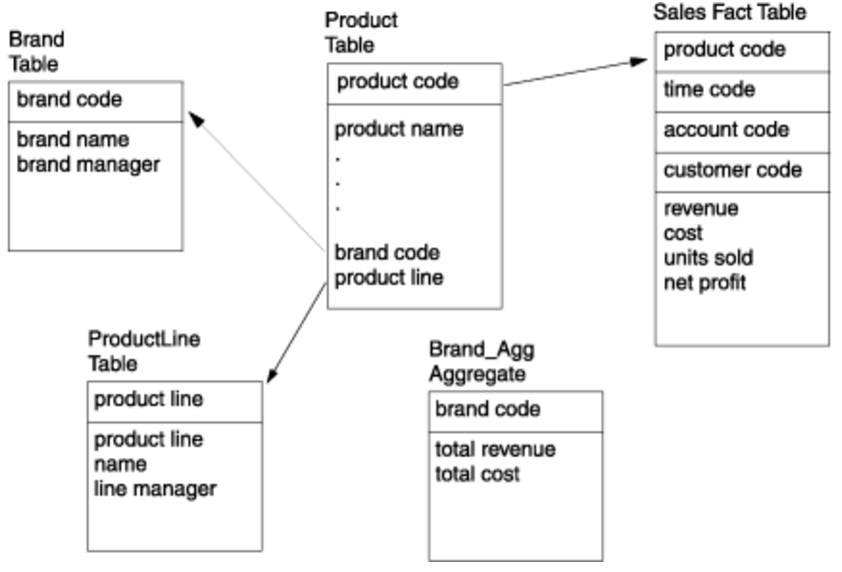
* 浏览、对比商品
* 加到购物车
* 结账、付钱

对应的也就是有模型、领域和抽象几个概念。

#### **模型**

这些商品实现上就是相当于一系列的模型及数据。在用户购买之前，我们只需要一个去获取一个个的数据接口，并展示这些数据。

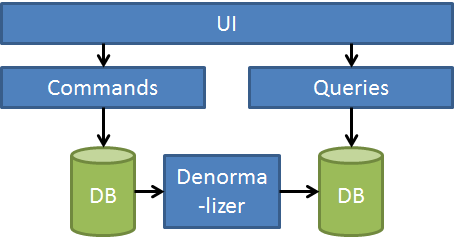
对应于这些商品要建起Schema来是一件容易的事。作为一个商品，他们都拥有着一些共同的元素：price, name, description, location, manufacturer等等的信息。其中一些属性，还会有复杂的对应关系：



Product Schema

这些需要在我们建立数据库的时候，尽可能地明确好这些关系。由于业务本身是难以预料的，你可能和我们之前的项目一样需要一个addtionInfo的字段，来用JSON存储一些额外的字段。当然如果你使用的是NoSQL，那就再好不过了。

最好你还使用了**读写分离架构**，一种比较常见的用法就是CMS网站，人们使用数据库来存储内容，使用静态页面来展示这些内容。比较好的实践还有CQRS(Command Query Responsibility Segregation， 命令查询职责分离模式)，用于CRUD（增、删、改，当然也可以查）的Command，以及Query的查询分开。简单的来说，就是有两个不同的数据持久化中心：



Basic CQRS

这一点特别适合于那些查询、搜索为主的网站，如淘宝。哈哈，我们离题有点远了，总之我们就是在这里提供了数据库的灵气，并对一些字段做一些简单的处理。听上去感觉GraphQL更适合做这样的事。

#### **领域**

而顾客及售货员在整个过程中做的事情就是领域(Domain，《实现领域驱动设计》)——即一个组织所做的事情以及其中所包含的一切。对于顾客和售货员来说，他们在各自的领域里做着不同的事。

对于顾客来说，其整个浏览过程，基本上都是在前端完成：

* 搜索、查找商品 -> 获得商品列表
* 查找商品详细
* 切换到下一个商品

这个场景下就特别适合于上面说到的读写分离架构。在浏览过程中，对用户的数据进行监控，以用于了解用户的行为，改善用户体验。这也是很常见的功能，或者说他们是无处不在的模式：

* 结果页 / 列表页
* 详情页

随后的用户收藏、添加到购物车、购买、交付等流程都需要与后台进行更紧密的交付。而这些都和售货员都有紧密的关系，而这些就不是一种简单的事。

从用户购买完成以后，剩下的就是一堆琐碎的事了，而这些都是由后端来完成的：

* 订单子系统
* 物流系统
* 发票系统
* 支付系统

等等。

对于用户来说，一种最简单的情况就是亚马逊，你只需要按一下“一键下单”即可。不需要关心后面的操作了，同样的这也适合于我们的业务场景。

#### **抽象**

抽象本来不打算写的，但是后来想了想还是写。总的来说整个过程还是需要相对比较好的抽象能力，要不我就很难讲述清楚这里面的过程了。

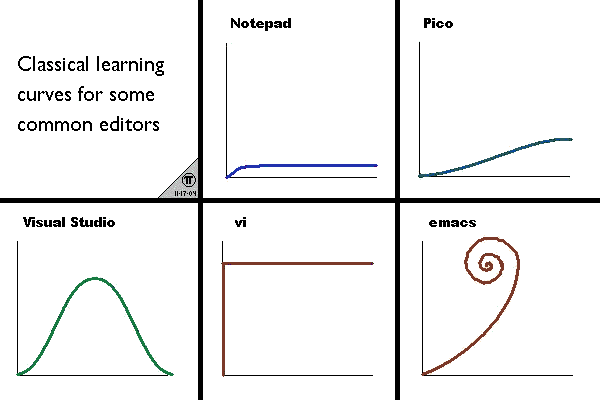
抽象是很神奇的东西，也可以分为几个不同的境界——但是我也不知道有几个境界，简单的来说就是不同的人看上去就有不同的东西。如有的人看到下面的画就是一坨shit——还不如小学生画的呢，有的人就会惊呼大师。



星空

反正，我也很看不懂。这一点倒类似于最初我对设计模型的理解一样：

* 一开始不以为然
* 然后发现很棒
* 接着使用过度
* 最后就和最好的编程器Emacs一样



Editor Learning Curve

这些都在随着编程生涯的展开而发生一些变化，我们不断地抽象出一些概念，以至于到了最后刚进入这个行业的人都看不懂。但是，这些都是一点点在一层层抽象的基础上产生的。

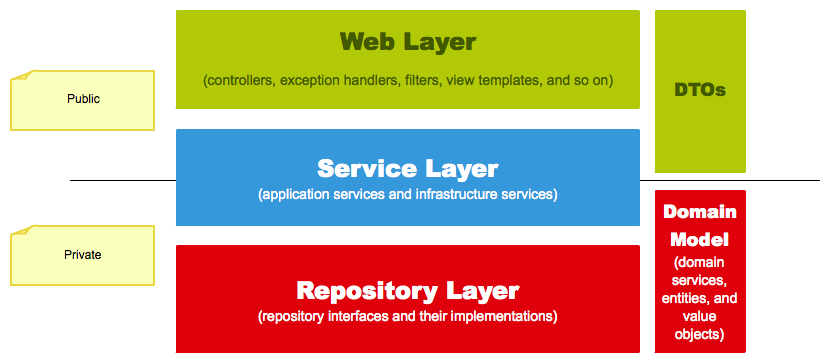


Needs

所以，我就把这一小小节扯完了，反正我是不想说得太抽象了。接着，让我们再扯点技术性的话题。

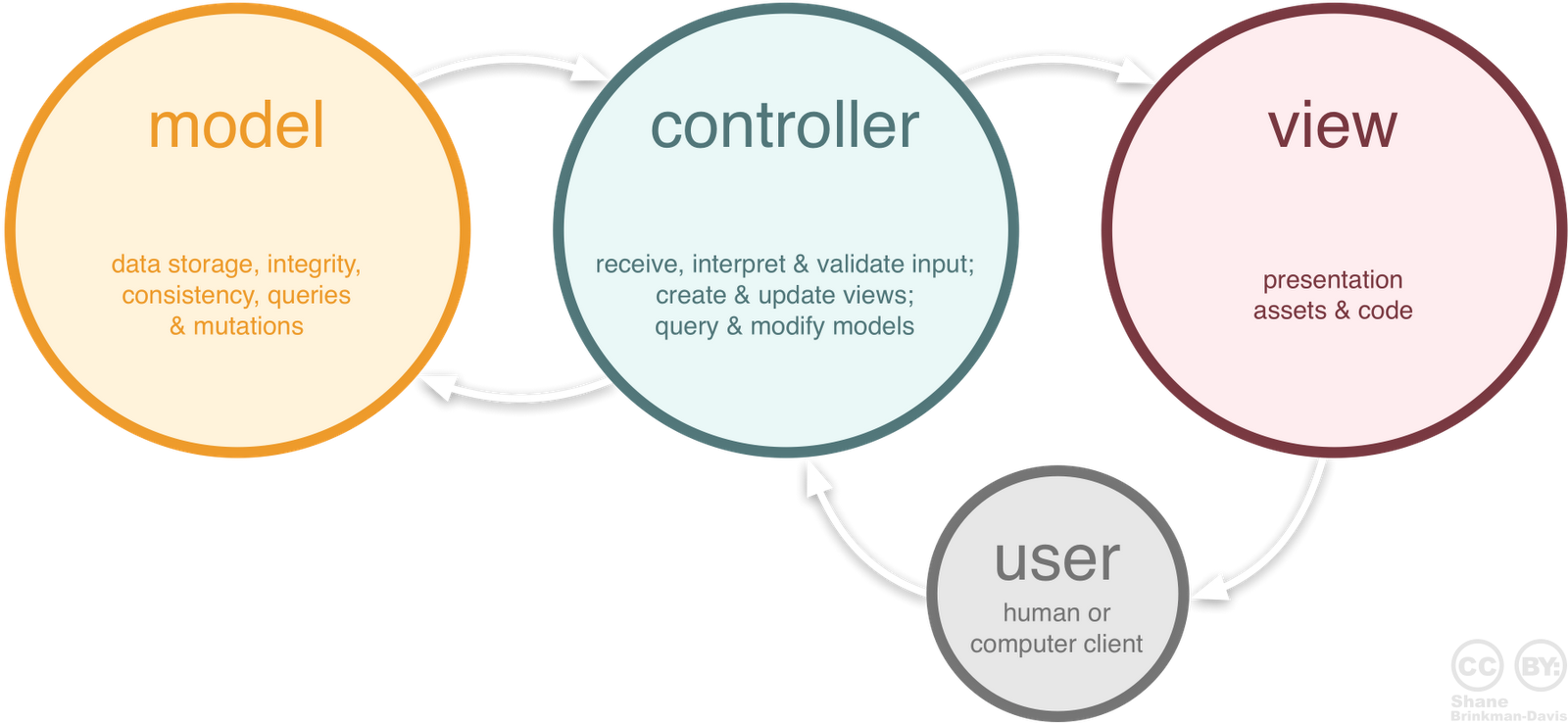
## 前后台分离：后台

典型的Web应用框架就是类似于这样的架构：



Spring Web App Architecture

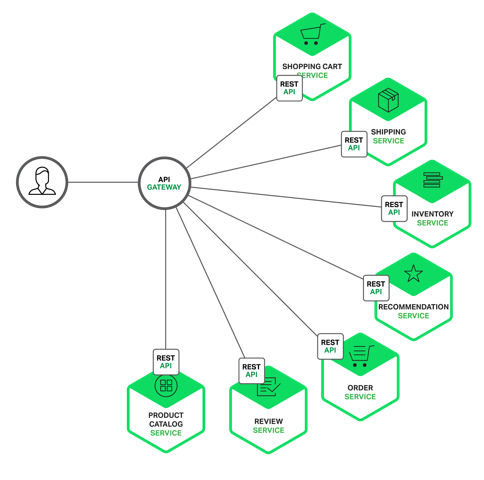
又或者是MVC架构，但是这已经不重要了。我们都前后端分离了，是时候把V层去掉了。



MVC Role Diagram

我们继续以上面的浏览来购买流程来做扯淡，后台除了提高上面的商品信息以外，在购买的时候还需要对用户进行授权。当然注册就是另外一个话题了，另外一个很大的话题。

所有的这些我们都可以向前台提供对应的API即可。理想的情况下，我们对应于不同的模块可以有不同的服务：



MicroServices

但是现实并不总是这么美好的，而在我们当前情况下则可以——毕竟所有的用户都应该能浏览所有的商品，这时就不需要做特殊的处理了。

在这个过程中，我们还有一系列的操作需要在后台来完成。不过，这些都可以在内部中完成的。而复杂的过程，实际上还存在于前端的逻辑当中。

## 前后台分离：前端

开始时，我们需要这样做去获取一个个的商品详情。这些数据也可以在返回页面模板的时候，直接将API数据填充到HTML中——带后台渲染的React都是这样做的。然后在用户浏览的过程中，我们还需要遵循这样的数据流程：

* 获取数据。无论是Ajax，还是新的Fetch API都可以做这样的事。
* 处理数据。依据于业务的需要对数据进行一些特殊的处理，如修改时间、价格的显示格式，描述的长度等等。
* 显示数据。只需要一个简单的模板引擎，过去的JSP、Mustache、今天的React都在做同样的事。

而在进入这个页面之前，我们还需要关注几个基本的元素，虽然这些都是将由框架来解决的。

* 路由。无论你遵不遵循REST的实践，我们只需要给对应的页面一个URL即可。相应的如果我们需要SEO，那么我们也需要在后台有一个对应的URL。理想的情况下，他们应该是由一个URL。
* 模块化。从过去Require.js的火热，到今天的各式各样的框架内建的模块化框架，他们解决都是一个问题：代码度的问题。这一点和后台采用的微服务架构的缘由好像是一样。
* 控制器。我也想不起来为什么控制器非在这里不可，但是我想只有这样，我才能快速地找到这个文件。

作为前端，我们不仅仅要负责的页面的美观，还要对用户的事件做出响应，并且做好用户体验。

* 事件

这就是最近火热的关于DOM的讨论的原因，当顾客想了解一下一些不一样东西的时候，我们就需要对DOM进行操作。

我突然就有一个问题了，你真的有那么多DOM需要操作么？你又不是Facebook，有那么多的Timeline和事件流。还有你的用户体验是不是做得足够好了？顺便再补一刀，如果你使用了React，并且没有进行前后端分离，那就等下一个恶梦。

最后，当用户买下东西的时候，我们也需要这样的交互流程。

## RePractise

因为最近我对DDD又有了一些想法，还在想着如何直接由真实世界来建模。顺便整理了这些思路到一起，但是好似这样的设计更简单。

# 易读

## 简介

### 编程经验

只要我有更多时间，我就会写一封更短的信给你。

从小学算起我的编程年限应该也有十几年了吧，笑~~。只是我过去的多年编程经验对于我现在的工作来说，是多年的无关经验(详见《REWORK》——多年的无关经验)。

高中的时候学习了点游戏编程，也因此学了点C++的皮毛，除了学会面向对象，其他都忘光了。随后在学习Linux内核，当时代码里就各种struct。比起之前学过的Logo和QBASIC简直是特别大的进步，然当时觉得struct与面向对象两者间没啥太大区别。在那个年少的时候，便天真的以为程序语言间的区别不是很大。

大学的时候主要营业范围是各种硬件，也没有发现写出好的代码是特别重要的一件事。也试了试Lisp，尝试过设计模式，然后失败了，GoF写DP的时候一定花了特别长的时间，所以这本书很短。期间出于生活压力(没有钱买硬件)，便开始兼职各种Web前端开发。

在有了所谓的GNU/Linux系统编译经验、写过各种杂七杂八的硬件代码，如Ada、汇编，要保证代码工作是一件很简单的事，从某个项目中引入部分代码，再从某个Demo中引入更多的代码，东拼西凑一下就能工作了。

多年的无关经验只让我写出能工作的代码——在别人看来就是很烂的代码。于是，虽然有着看上去很长的编程经验，但是却比不上实习的时候6个月学到的东西。

只是因为，我们不知道: 我们不知道。

### 代码整洁

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:

1. 确保我们的变量名、函数名是易读的
2. 没有复杂的逻辑判断
3. 没有多层嵌套
4. 减少复杂函数的出现

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。

### 别人的代码很烂?

什么是很烂的代码? 应该会有几种境界吧。

1. 不能工作，不能读懂
2. 不能工作，能读懂
3. 能工作，很难读懂
4. 能工作，能读懂，但是没有意图
5. 能工作，能理解意图，但是读不懂

如果我们能读懂，能理解意图，那么我们还说他烂，可能是因为他并不整洁。这就回到了上面的问题，模式是一种因人而异的东西。

我们在做Code Review的时候，总会尝试问对方说： “这样做的意图是”。

对于代码来说也是如此，如果我们能理解意图的话，那么我们要理解代码相对也比较容易。如果对方是没有意图，那么代码是没救的。

## 变量名

## 函数名

## 小函数

## 测试

# 重构篇

什么是重构?

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。

相似的

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。

## 网站重构

与上述相似的是:在不改变外部行为的前提下，简化结构、添加可读性，而在网站前端保持一致的行为。也就是说是在不改变UI的情况下，对网站进行优化，在扩展的同时保持一致的UI。

过去人们所说的网站重构

把"未采用CSS，大量使用HTML进行定位、布局，或者虽然已经采用CSS，但是未遵循HTML结构化标准的站点"变成"让标记回归标记的原本意义。通过在HTML文档中使用结构化的标记以及用CSS控制页面表现，使页面的实际内容与它们呈现的格式相分离的站点。"的过程就是网站重构(Website Reconstruction)

依照我做过的一些案例，对于传统的网站来说重构通常是

* 表格(table)布局改为DIV+CSS
* 使网站前端兼容于现代浏览器(针对于不合规范的CSS、如对IE6有效的)
* 对于移动平台的优化
* 针对于SEO进行优化

过去的网站重构就是“DIV+CSS”，想法固然极度局限。但也不是另一部分的人认为是“XHTML+CSS”，因为“XHTML+CSS”只是页面重构。

而真正的网站重构

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。

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(如VB)
* 增强用户体验

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(通常是由服务器来解决)
* 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

可以应用的的方面

* [使用Ngx\_pagespeed优化前端](http://www.phodal.com/blog/nginx-with-ngx-pagespeed-module-improve-website-cache/)
* 解耦复杂的模块
* 对缓存进行优化
* 针对于内容创建或预留API
* 需要添加新API，如(weChat等的支持)
* 用新的语言、框架代码旧的框架(如VB.NET，C#.NET)

### 网站重构目的

希望自己的网站

* 成本变得更低
* 运行得更好
* 访问者更多
* 维护愈加简单
* 功能更强

## 代码重构

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。

## 使用工具重构

## 借助工具重构

* 当你写了一大堆代码,你没有意识到里面有一大堆重复。
* 当你写了一大堆测试,却不知道覆盖率有多少。

这就是个问题了，于是偶然间看到了一个叫code climate的网站。

### Code Climate

Code Climate consolidates the results from a suite of static analysis tools into a single, real-time report, giving your team the information it needs to identify hotspots, evaluate new approaches, and improve code quality.

Code Climate整合一组静态分析工具的结果到一个单一的，实时的报告，让您的团队需要识别热点，探讨新的方法，提高代码质量的信息。

简单地来说:

* 对我们的代码评分
* 找出代码中的坏味道

于是，我们先来了个例子

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Rating | Name | Complexity | Duplication | Churn | C/M | Coverage | Smells |
| A | lib/coap/coap\_request\_handler.js | 24 | 0 | 6 | 2.6 | 46.4% | 0 |
| A | lib/coap/coap\_result\_helper.js | 14 | 0 | 2 | 3.4 | 80.0% | 0 |
| A | lib/coap/coap\_server.js | 16 | 0 | 5 | 5.2 | 44.0% | 0 |
| A | lib/database/db\_factory.js | 8 | 0 | 3 | 3.8 | 92.3% | 0 |
| A | lib/database/iot\_db.js | 7 | 0 | 6 | 1.0 | 58.8% | 0 |
| A | lib/database/mongodb\_helper.js | 63 | 0 | 11 | 4.5 | 35.0% | 0 |
| C | lib/database/sqlite\_helper.js | 32 | 86 | 10 | 4.5 | 35.0% | 2 |
| B | lib/rest/rest\_helper.js | 19 | 62 | 3 | 4.7 | 37.5% | 2 |
| A | lib/rest/rest\_server.js | 17 | 0 | 2 | 8.6 | 88.9% | 0 |
| A | lib/url\_handler.js | 9 | 0 | 5 | 2.2 | 94.1% | 0 |

分享得到的最后的结果是:



Coverage

#### 代码的坏味道

于是我们就打开lib/database/sqlite\_helper.js，因为其中有两个坏味道

Similar code found in two :expression\_statement nodes (mass = 86)

在代码的 lib/database/sqlite\_helper.js:58…61 < >

SQLiteHelper.prototype.deleteData = function (url, callback) {  
 'use strict';  
 var sql\_command = "DELETE FROM " + config.table\_name + " where " + URLHandler.getKeyFromURL(url) + "=" + URLHandler.getValueFromURL(url);  
 SQLiteHelper.prototype.basic(sql\_command, callback);

lib/database/sqlite\_helper.js:64…67 < >

与

SQLiteHelper.prototype.getData = function (url, callback) {  
 'use strict';  
 var sql\_command = "SELECT \* FROM " + config.table\_name + " where " + URLHandler.getKeyFromURL(url) + "=" + URLHandler.getValueFromURL(url);  
 SQLiteHelper.prototype.basic(sql\_command, callback);

只是这是之前修改过的重复。。

原来的代码是这样的

SQLiteHelper.prototype.postData = function (block, callback) {  
 'use strict';  
 var db = new sqlite3.Database(config.db\_name);  
 var str = this.parseData(config.keys);  
 var string = this.parseData(block);  
  
 var sql\_command = "insert or replace into " + config.table\_name + " (" + str + ") VALUES (" + string + ");";  
 db.all(sql\_command, function (err) {  
 SQLiteHelper.prototype.errorHandler(err);  
 db.close();  
 callback();  
 });  
};  
  
SQLiteHelper.prototype.deleteData = function (url, callback) {  
 'use strict';  
 var db = new sqlite3.Database(config.db\_name);  
 var sql\_command = "DELETE FROM " + config.table\_name + " where " + URLHandler.getKeyFromURL(url) + "=" + URLHandler.getValueFromURL(url);  
 db.all(sql\_command, function (err) {  
 SQLiteHelper.prototype.errorHandler(err);  
 db.close();  
 callback();  
 });  
};  
  
SQLiteHelper.prototype.getData = function (url, callback) {  
 'use strict';  
 var db = new sqlite3.Database(config.db\_name);  
 var sql\_command = "SELECT \* FROM " + config.table\_name + " where " + URLHandler.getKeyFromURL(url) + "=" + URLHandler.getValueFromURL(url);  
 db.all(sql\_command, function (err, rows) {  
 SQLiteHelper.prototype.errorHandler(err);  
 db.close();  
 callback(JSON.stringify(rows));  
 });  
};

说的也是大量的重复，重构完的代码

SQLiteHelper.prototype.basic = function(sql, db\_callback){  
 'use strict';  
 var db = new sqlite3.Database(config.db\_name);  
 db.all(sql, function (err, rows) {  
 SQLiteHelper.prototype.errorHandler(err);  
 db.close();  
 db\_callback(JSON.stringify(rows));  
 });  
  
};  
  
SQLiteHelper.prototype.postData = function (block, callback) {  
 'use strict';  
 var str = this.parseData(config.keys);  
 var string = this.parseData(block);  
  
 var sql\_command = "insert or replace into " + config.table\_name + " (" + str + ") VALUES (" + string + ");";  
 SQLiteHelper.prototype.basic(sql\_command, callback);  
};  
  
SQLiteHelper.prototype.deleteData = function (url, callback) {  
 'use strict';  
 var sql\_command = "DELETE FROM " + config.table\_name + " where " + URLHandler.getKeyFromURL(url) + "=" + URLHandler.getValueFromURL(url);  
 SQLiteHelper.prototype.basic(sql\_command, callback);  
};  
  
SQLiteHelper.prototype.getData = function (url, callback) {  
 'use strict';  
 var sql\_command = "SELECT \* FROM " + config.table\_name + " where " + URLHandler.getKeyFromURL(url) + "=" + URLHandler.getValueFromURL(url);  
 SQLiteHelper.prototype.basic(sql\_command, callback);  
};

重构完后的代码比原来还长，这似乎是个问题~~

## 测试驱动开发

### 一次测试驱动开发的故事

之前正在重写一个[物联网](http://www.phodal.com/iot)的服务端，主要便是结合CoAP、MQTT、HTTP等协议构成一个物联网的云服务。现在，主要的任务是集中于协议与授权。由于，不同协议间的授权是不一样的，最开始的时候我先写了一个http put授权的功能，而在起先的时候是如何测试的呢?

curl --user root:root -X PUT -d '{ "dream": 1 }' -H "Content-Type: application/json" http://localhost:8899/topics/test

我只要顺利在request中看有无req.headers.authorization，我便可以继续往下，接着给个判断。毕竟，我们对HTTP协议还是蛮清楚的。

if (!req.headers.authorization) {  
 res.statusCode = 401;  
 res.setHeader('WWW-Authenticate', 'Basic realm="Secure Area"');  
 return res.end('Unauthorized');  
}

可是除了HTTP协议，还有MQTT和CoAP。对于MQTT协议来说，那还算好，毕竟自带授权，如:

mosquitto\_pub -u root -P root -h localhost -d -t lettuce -m "Hello, MQTT. This is my first message."

便可以让我们简单地完成这个功能，然而有的协议是没有这样的功能如CoAP协议中是用Option来进行授权的。现在的工具如libcoap只能有如下的简单功能

coap-client -m get coap://127.0.0.1:5683/topics/zero -T

于是，先写了个测试脚本来验证功能。

var coap = require('coap');  
var request = coap.request;  
var req = request({hostname: 'localhost',port:5683,pathname: '',method: 'POST'});  
  
...  
  
req.setHeader("Accept", "application/json");  
req.setOption('Block2', [new Buffer('phodal'), new Buffer('phodal')]);  
  
...  
  
req.end();

写完测试脚本后发现不对了，这个不应该是测试的代码吗? 于是将其放到了spec中，接着发现了上面的全部功能的实现过程为什么不用TDD实现呢？

### 说说测试驱动开发

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。

测试驱动开发的主要过程是:

1. 先写功能的测试
2. 实现功能代码
3. 提交代码(commit -> 保证功能正常)
4. 重构功能代码

而对于这样的一个物联网项目来说，我已经有了几个有利的前提:

1. 已经有了原型
2. 框架设计

### 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?

# 架构篇: CMS的重构与演进

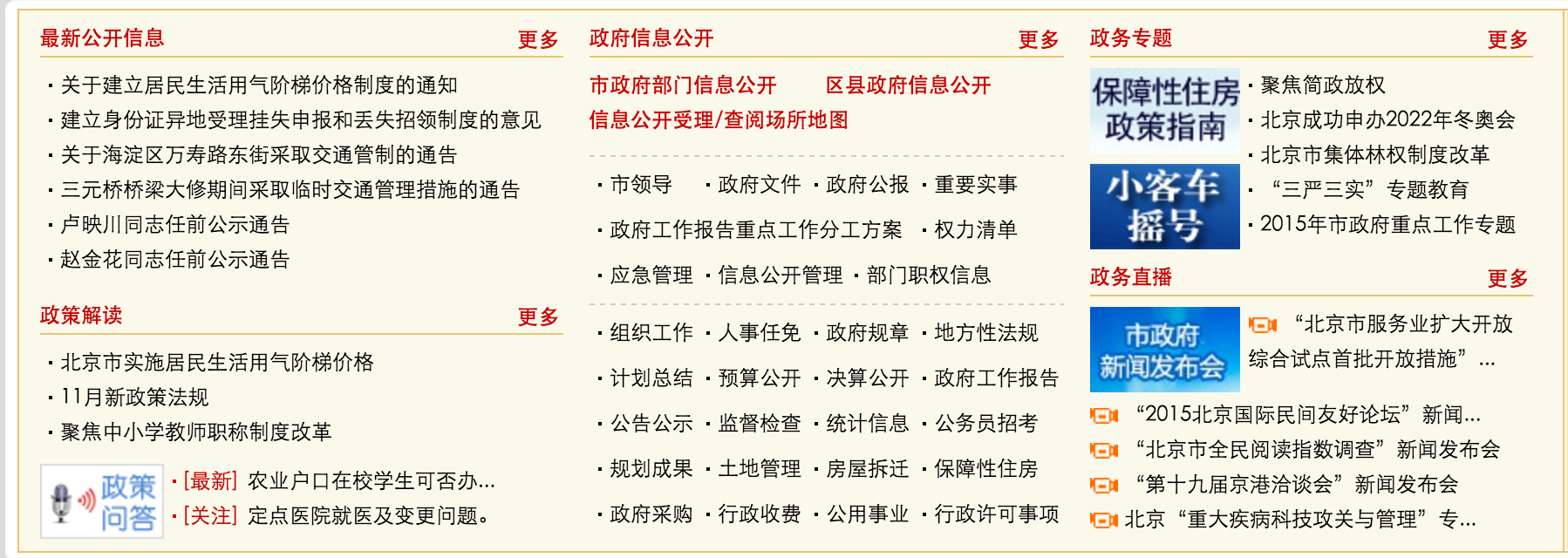
重构系统是一项非常具有挑战性的事情。通常来说，在我们的系统是第二个系统的时候才需要重构，即这个系统本身已经很臃肿。我们花费了太量的时间在代码间的逻辑，开发新的功能变得越来越慢。这不仅仅可能只是因为我们之前的架构没有设计好，而且在我们开发的过程中没有保持着原先设计时的一些原则。如果是这样的情况，那么这就是一个复杂的过程。

还有一种情况是我们发现了一种更符合我们当前业务的框架。

## 动态CMS

### CMS简介

CMS是Content Management System的缩写，意为"内容管理系统".它可以做很多的事情，但是总的来说就是Page和Blog——即我们要创建一些页面可以用于写一些About US、Contact Me，以及持续更新的博客或者新闻，以及其他子系统——通常更新不活跃。通过对这些博客或者新闻进行分类，我们就可以有不同的信息内容，如下图：



不同分类的内容

CMS是政府和企业都需要的系统，他们有很多的信息需要公开，并且需要对其组织进行宣传。在我有限的CMS交付经验里（大学时期），一般第一次交付CMS的时候，已经创建了大部分页面。有时候这些页面可能直接存储在数据库中，后来发现这不是一个好的方案，于是很多页面变成了静态页面。随后，在CMS的生命周期里就是更新内容。

因而，CMS中起其主导的东西还是Content，即内容。而内容是一些持续可变的东西。这也就是为什么WordPress这么流行于CMS界，它是一个博客系统，但是多数时候我们只需要更新内容。除此不得不提及的一个CMS框架是Drupal，两者一对比会发现Drupal比较强大。通常来说，强大的一个负作用就是——复杂。

WordPress和Drupal这一类的系统都属于发布系统，而其后台可以称为编辑系统。

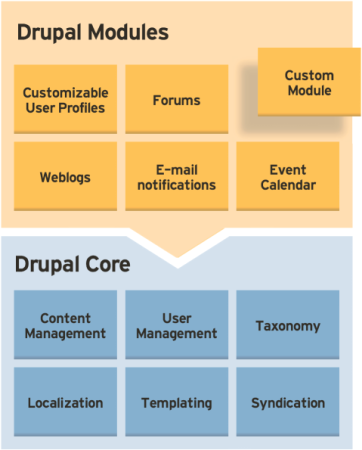
一般来说CMS有下面的特点：

* 支持多用户。
* 角色控制-内容管理。如InfoQ的编辑后台就会有这样的机制，社区编辑负责创建内容，而审核发布则是另外的人做的。
* 插件管理。如WordPress和Drupal在这一方面就很强大，基本可以满足日常的需要。
* 快捷简便地存储内容。简单地来说就是所见即所得编辑器，但是对于开发者来说，Markdown似乎是好的选择。
* 预发布。这是一个很重要的特性，特别是如果你的系统后台没有相对应的预览机制。
* 子系统。由于这属于定制化的系统，并不方便进行总结。
* ...

CMS一直就是这样一个紧耦合的系统。

### CMS架构与Django

说起来，我一直是一个CMS党。主要原因还在于我可以随心所欲地去修改网站的内容，修改网站的架构。好的CMS总的来说都有其架构图，下图似乎是Drupal的模块图



Drupal 框架

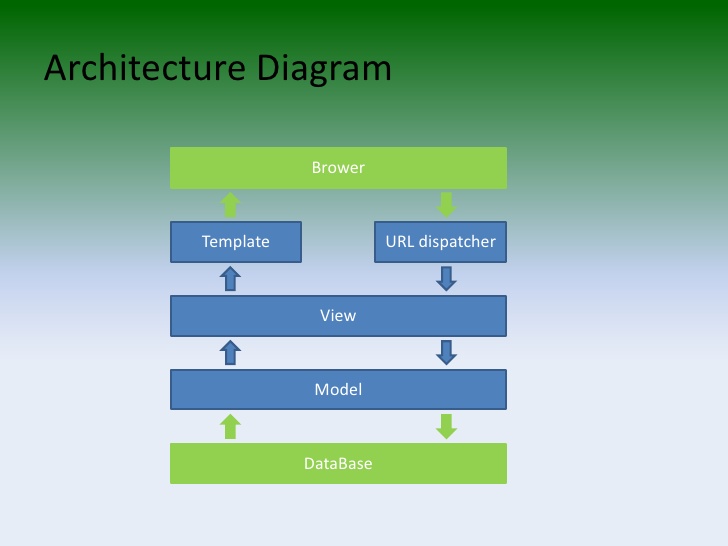
一般来说，其底层都会有：

* ORM
* User Management
* I18n / L10n
* Templates

我一直在使用一个名为Django的Python Web框架，它最初是被开发来用于管理劳伦斯出版集团旗下的一些以新闻内容为主的网站的，即是CMS（内容管理系统）软件。它是一个MTV框架——与多数的框架并没有太大的区别。

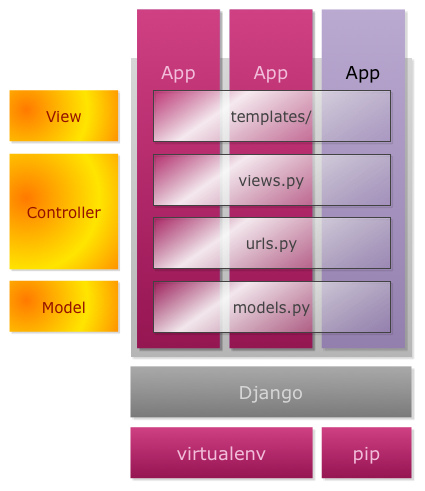
|  |  |
| --- | --- |
| 层次 | 职责 |
| 模型（Model），即数据存取层 | 处理与数据相关的所有事务：如何存取、如何验证有效性、包含哪些行为以及数据之间的关系等。 |
| 模板(Template)，即表现层 | 处理与表现相关的决定： 如何在页面或其他类型文档中进行显示。 |
| 视图（View），即业务逻辑层 | 存取模型及调取恰当模板的相关逻辑。模型与模板之间的桥梁。 |

从框架本身来上看它和别的系统没有太大的区别。



Django Architecture

但是如果我们已经有多外模块（即Django中app的概念），那么系统的架构就有所不同了。

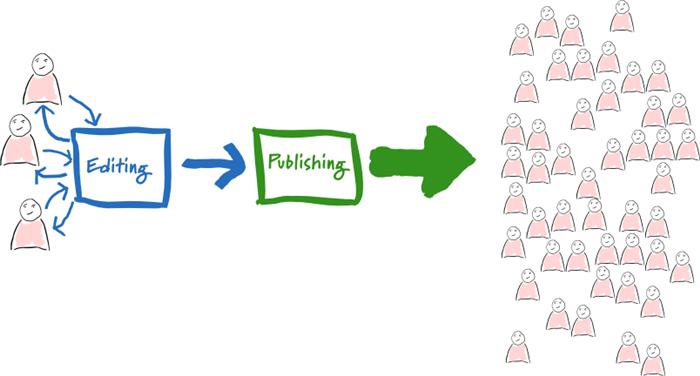


Django App架构

这就是为何我喜欢用这个CMS的原因了，我的每个子系统都以APP的形式提供服务——博客是一个app，sitemap是一个app，api是一个app。系统直接解耦为类似于混合服务的架构，即不像微服务一样多语言化，又不会有宏应用的紧耦合问题。

### 编辑-发布分离

我们的编辑和发布系统在某种意义上紧耦合在一起了，当用户访问量特别大的时候，这样会让我们的应用变得特定慢。有时候编辑甚至发布不了新的东西，如下图引示:



发布-编辑

或者你认识出了上图是源自Martin Folwer的[编辑-发布分离](http://martinfowler.com/bliki/EditingPublishingSeparation.html)

编辑-发布分离是几年前解耦复杂系统游来开来带来的一个成果。今天这个似乎已经很常见了，编辑的时候是在后台进行的，等到发布的时候已经变成了一个静态的HTML。

已经有足够多的CMS支持这样的特性，运行起来似乎特别不错，当然这样的系统也会有缓存的问题。有了APP这后，这个趋势就更加明显了——人们需要提供一个API。到底是在现有的系统里提供一个新的API，还是创建一个新的API。

这时候，我更愿意选择后者——毕竟紧耦合一个系统总会在后期带来足够多的麻烦。而且基于数据库构建一个只读的RESTful API并不是一个复杂的过程，而且也危险。这时候的瓶颈就是数据库，但是似乎数据库都是多数系统的瓶颈。人们想出了各种各样的技术来解决这个瓶颈。

于是之前我试着用Node.js + RESTify将我的博客重构成了一个SPA，当然这个时候CMS还在运行着。出于SEO的原因我并没有在最后采用这个方案，因为[我网站](https://www.phodal.com)的主要流量来源是Google和是百度。但是我在另外的网站里混合了SPA与MPA，其中的性能与应用是相当的，除了第一次加载页面的时候会带来一些延时。

除了Node.js + RESTify，也试了试Python + Falcon（一个高性能的RESTful框架）。这个API理论上也应该可以给APP直接使用，并且可以直接拿来生成静态页面。

#### 编辑-发布-开发分离：静态站点生成

如React一样解决DOM性能的问题就是跳过DOM这个坑，要跳过动态网站的性能问题就是让网站变成静态。

越来越多的开发人员开始在使用Github Pages作为他们的博客，这是一个很有意思的转变。主要的原因是这是免费的，并且基本上可以保证24x7小时是可用的——当且仅当Github发现故障的时候才会不可访问。

在这一类静态站点生成器(Github)里面，比较流行的有下面的内容（数据来源： <http://segmentfault.com/a/1190000002476681>）:

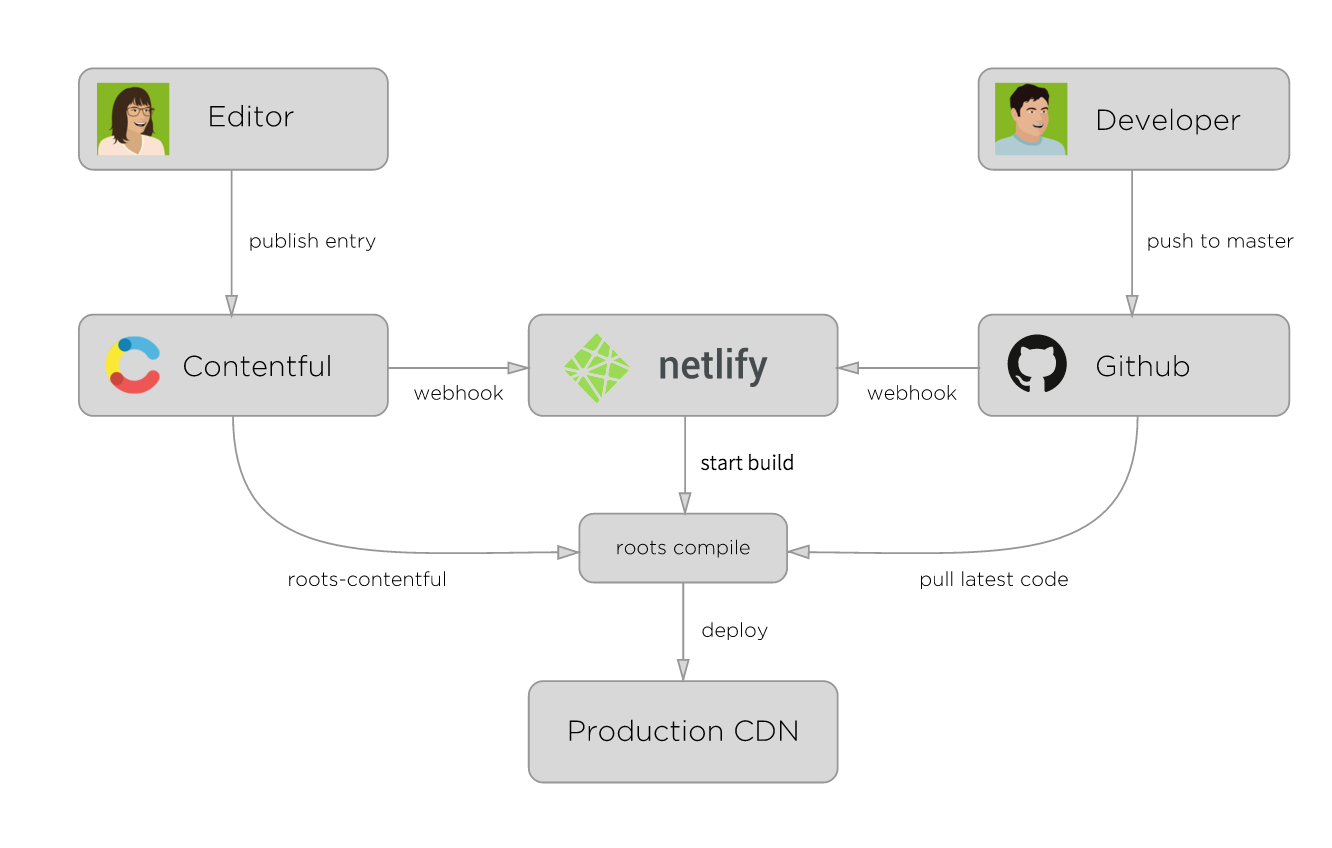
1. Jekyll / OctoPress。Jekyll和OctoPress是最流行的静态博客系统。
2. Hexo。Hexo是NodeJS编写的静态博客系统，其生成速度快，主题数量相对也比较丰富。是OctoPress的优秀替代者。
3. Sculpin。Sculpin是PHP的静态站点系统。Hexo和Octopress专注于博客，而有时候我们的需求不仅仅是博客，而是有类似CMS的页面生成需求。Sculpin是一个泛用途的静态站点生成系统，在支持博客常见的分页、分类tag等同时，也能较好地支持非博客的一般页面生成。
4. Hugo。Hugo是GO语言编写的静态站点系统。其生成速度快，且在较好支持博客和非博客内容的同时提供了比较完备的主题系统。无论是自己写主题还是套用别人的主题都比较顺手。

通常这一类的工具里会有下面的内容：

1. 模板
2. 支持Markdown
3. 元数据

如Hexo这样的框架甚至提供了一键部署的功能。

在我们写了相关的代码之后，随后要做的就是生成HTML。对于个人博客来说，这是一个非常不错的系统，但是对于一些企业级的系统来说，我们的要求就更高了。如下图是Carrot采用的架构：



Editor Develoepr

这与我们在项目上的系统架构目前相似。作为一个博主，通常来说我们修改博客的主题的频率会比较低， 可能是半年一次。如果你经常修改博客的主题，你博客上的文章一定是相当的少。

上图中的编辑者通过一个名为Contentful CMS来创建他们的内容，接着生成RESTful API。而类似的事情，我们也可以用Wordpress + RESTful 插件来完成。如果做得好，那么我想这个API也可以直接给APP使用。

上图中的开发者需要不断地将修改的主题或者类似的东西PUSH到版本管理系统上，接着会有webhook监测到他们的变化，然后编译出新的静态页面。

最后通过Netlify，他们编译到了一起，然后部署到生产环境。除了Netlify，你也可以编写生成脚本，然后用Bamboo、Go这类的CI工具进行编译。

通常来说，生产环境可以使用CDN，如CloudFront服务。与动态网站相比，静态网站很容易直接部署到CDN，并可以直接从离用户近的本地缓存提供服务。除此，直接使用AWS S3的静态网站托管也是一个非常不错的选择。

### 基于Github的编辑-发布-开发分离

尽管我们已经在项目上实施了基于Github的部分内容管理已经有些日子里，但是由于找不到一些相关的资料，便不好透露相关的细节。直到我看到了《[An Incremental Approach to Content Management Using Git 1](https://www.thoughtworks.com/insights/blog/incremental-approach-content-management-using-git)》，我才意识到这似乎已经是一个成熟的技术了。看样子这项技术首先已经应用到了ThoughtWorks的官网上了。

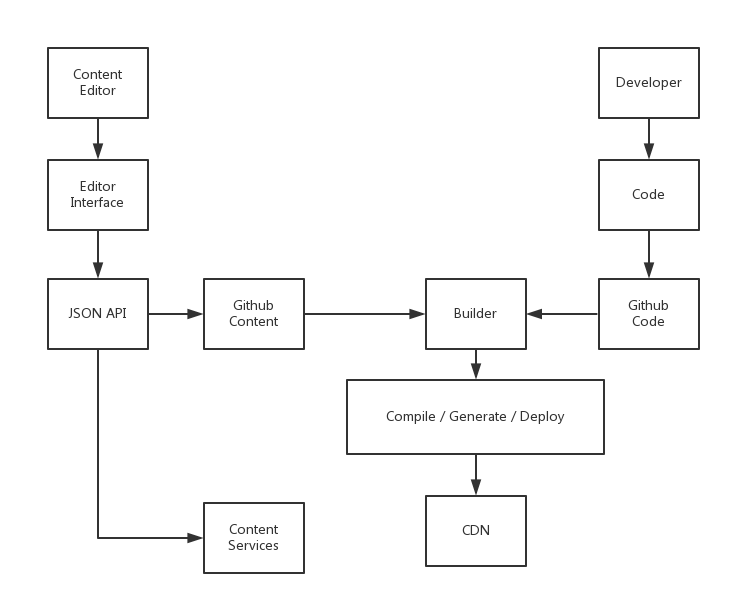
文中提到了使用这种架构的几个点：

1. 快速地开始项目，而不是学习或者配置框架。
2. 需要使用我们信奉的原则，如TDD。而这是大部分CMS所不支持的。
3. 基于服务的架构。
4. 灵活的语言和工具
5. 我们是开发人员。

So，so，这些开发人员做了些什么：

1. 内容存储为静态文件
2. 不是所有的内容都是平等的
3. 引入内容服务
4. 使用Github。所有的content会提交到一个repo里，同时在我们push内容的时候，可以实时更新这些内容。
5. 允许内容通过内容服务更新
6. 使用Github API

于是，有了一个名为[Hacienda](https://github.com/haciendaio/hacienda)的框架用于管理内容，并存储为JSON。这意味着什么？



基于Github的编辑-发布-开发分离

因为使用了Git，我们可以了解到一个文件内容的历史版本，相比于WordPress来说更直观，而且更容易 上手。

开发人员修改完他们的代码后，就可以直接提交，不会影响到Editor使用网站。Editor通过一个编辑器添加内容，在保存后，内容以JSON的形式出现直接提交代码到Github上相应的代码库中。CI或者Builder监测到他们的办法，就会生成新的静态页面。在这时候，我们可以选择有一个预览的平台，并且可以一键部署。那么，事情似乎就完成得差不多了。

如果我们有APP，那么我们就可以使用Content Servies来做这些事情。甚至可以直接拿其搭建一个SPA。

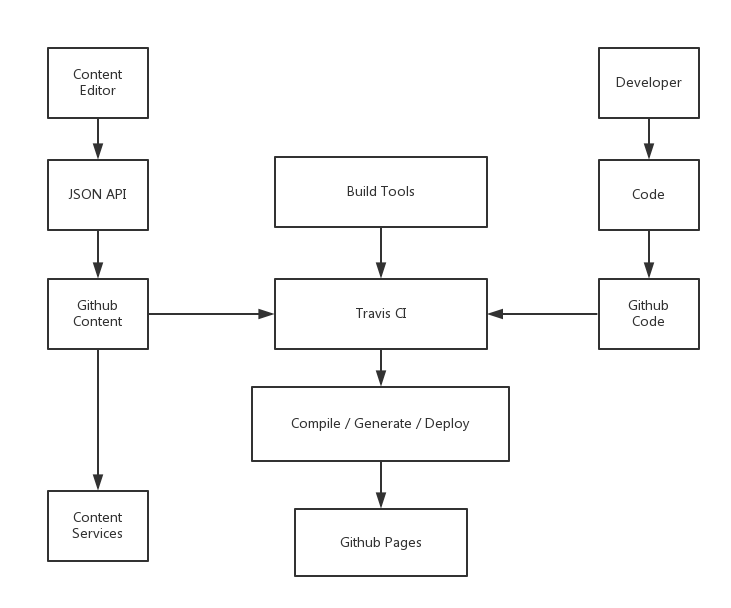
如果我们需要全文搜索功能，也变得很简单。我们已经不需要直接和数据库交互，我们可以直接读取JSON并且构建索引。这时候需要一个简单的Web服务，而且这个服务还是只读的。

在需要的时候，如手机APP，我们可以通过Content Servies来创建博客。

### Repractise

动态网页是下一个要解决的难题。我们从数据库中读取数据，再用动态去渲染出一个静态页面，并且缓存服务器来缓存这个页面。既然我们都可以用Varnish、Squid这样的软件来缓存页面——表明它们可以是静态的，为什么不考虑直接使用静态网页呢？

思考完这些后，我想到了一个符合学习的场景。



基于Travis CI的编辑-发布-开发分离

我们构建的核心都可以基于Travis CI来完成，唯一存在风险的环节是我们似乎需要暴露我们的Key。

#### 其他

参考文章:

1. [静态网站生成器将会成为下一个大热门](http://www.infoq.com/cn/news/2015/11/LAMP-CDN)
2. [EditingPublishingSeparation](http://martinfowler.com/bliki/EditingPublishingSeparation.html)
3. [An Incremental Approach to Content Management Using Git 1](https://www.thoughtworks.com/insights/blog/incremental-approach-content-management-using-git)
4. [Part 2: Implementing Content Management and Publication Using Git](https://www.thoughtworks.com/insights/blog/implementing-content-management-and-publication-using-git)

## 构建基于Git为数据中心的CMS

或许你也用过Hexo / Jekyll / Octopress这样的静态博客，他们的原理都是类似的。我们有一个代码库用于生成静态页面，然后这些静态页面会被PUSH到Github Pages上。

从我们设计系统的角度来说，我们会在Github上有三个主要代码库：

1. Content。用于存放编辑器生成的JSON文件，这样我们就可以GET这些资源，并用Backbone / Angular / React 这些前端框架来搭建SPA。
2. Code。开发者在这里存放他们的代码，如主题、静态文件生成器、资源文件等等。
3. Builder。在这里它是运行于Travis CI上的一些脚本文件，用于Clone代码，并执行Code中的脚本。

以及一些额外的服务，当且仅当你有一些额外的功能需求的时候。

1. Extend Service。当我们需要搜索服务时，我们就需要这样的一些服务。如我正考虑使用Python的whoosh来完成这个功能，这时候我计划用Flask框架，但是只是计划中——因为没有合适的中间件。
2. Editor。相比于前面的那些知识这一步适合更重要，也就是为什么生成的格式是JSON而不是Markdown的原理。对于非程序员来说，要熟练掌握Markdown不是一件容易的事。于是，一个考虑中的方案就是使用 Electron + Node.js来生成API，最后通过GitHub API V3来实现上传。
3. Mobile App。

So，这一个过程是如何进行的。

### 用户场景

整个过程的Pipeline如下所示：

1. 编辑使用他们的编辑器来编辑的内容并点击发布，然后这个内容就可以通过GitHub API上传到Content这个Repo里。
2. 这时候需要有一个WebHooks监测到了Content代码库的变化，便运行Builder这个代码库的Travis CI。
3. 这个Builder脚本首先，会设置一些基本的git配置。然后clone Content和Code的代码，接着运行构建命令，生成新的内容。
4. 然后Builder Commit内容，并PUSH内容。

在这种情形中，编辑能否完成工作就不依赖于网站——脱稿又少了 个借口。这时候网站出错的概率太小了——你不需要一个缓存服务器、HTTP服务器，由于没有动态生成的内容，你也不需要守护进程。这些内容都是静态文件，你可以将他们放在任何可以提供静态文件托管的地方——CloudFront、S3等等。或者你再相信自己的服务器，Nginx可是全球第二好（第一还没出现）的静态文件服务器。

开发人员只在需要的时候去修改网站的一些内容。So，你可能会担心如果这时候修改的东西有问题了怎么办。

1. 使用这种模式就意味着你需要有测试来覆盖这些构建工具、生成工具。
2. 相比于自己的代码，别人的CMS更可靠？

需要注意的是如果你上一次构建成功，你生成的文件都是正常的，那么你只需要回滚开发相关的代码即可。旧的代码仍然可以工作得很好。其次，由于生成的是静态文件，查错的成本就比较低。最后，重新放上之前的静态文件。

## Code: 生成静态页面

Assemble是一个使用Node.js，Grunt.js，Gulp，Yeoman 等来实现的静态网页生成系统。这样的生成器有很多，Zurb Foundation, Zurb Ink, Less.js / lesscss.org, Topcoat, Web Experience Toolkit等组织都使用这个工具来生成。这个工具似乎上个Release在一年多以前，现在正在开始0.6。虽然，这并不重要，但是还是顺便一说。

我们所要做的就是在我们的Gruntfile.js中写相应的生成代码。

assemble: {  
 options: {  
 flatten: true,  
 partials: ['templates/includes/\*.hbs'],  
 layoutdir: 'templates/layouts',  
 data: 'content/blogs.json',  
 layout: 'default.hbs'  
 },  
 site: {  
 files: {'dest/': ['templates/\*.hbs']}  
 },  
 blogs: {  
 options: {  
 flatten: true,  
 layoutdir: 'templates/layouts',  
 data: 'content/\*.json',  
 partials: ['templates/includes/\*.hbs'],  
 pages: pages  
 },  
 files: [  
 { dest: './dest/blog/', src: '!\*' }  
 ]  
 }  
 }

配置中的site用于生成页面相关的内容，blogs则可以根据json文件的文件名生成对就的html文件存储到blog目录中。

生成后的目录结果如下图所示：

.  
├── about.html  
├── blog  
│   ├── blog-posts.html  
│   └── blogs.html  
├── blog.html  
├── css  
│   ├── images  
│   │   └── banner.jpg  
│   └── style.css  
├── index.html  
└── js  
 ├── jquery.min.js  
 └── script.js  
  
7 directories, 30 files

这里的静态文件内容就是最后我们要发布的内容。

还需要做的一件事情就是：

grunt.registerTask('dev', ['default', 'connect:server', 'watch:site']);

用于开发阶段这样的代码就够了，这个和你使用WebPack + React 似乎相差不了多少。

## Builder: 构建生成工具

Github与Travis之间，可以做一个自动部署的工具。相信已经有很多人在Github上玩过这样的东西——先在Github上生成Token，然后用travis加密：

travis encrypt-file ssh\_key --add

加密后的Key就会保存到.travis.yml文件里，然后就可以在Travis CI上push你的代码到Github上了。

接着，你需要创建个deploy脚本，并且在after\_success执行它：

after\_success:  
 - test $TRAVIS\_PULL\_REQUEST == "false" && test $TRAVIS\_BRANCH == "master" && bash deploy.sh

在这个脚本里，你所需要做的就是clone content和code中的代码，并执行code中的生成脚本，生成新的内容后，提交代码。

#!/bin/bash  
  
set -o errexit -o nounset  
  
rev=$(git rev-parse --short HEAD)  
  
cd stage/  
  
git init  
git config user.name "Robot"  
git config user.email "robot@phodal.com"  
  
git remote add upstream "https://$GH\_TOKEN@github.com/phodal-archive/echeveria-deploy.git"  
git fetch upstream  
git reset upstream/gh-pages  
  
git clone https://github.com/phodal-archive/echeveria-deploy code  
git clone https://github.com/phodal-archive/echeveria-content content  
pwd  
cp -a content/contents code/content  
  
cd code  
  
npm install  
npm install grunt-cli -g  
grunt   
mv dest/\* ../  
cd ../  
rm -rf code  
rm -rf content  
  
touch .  
  
if [ ! -f CNAME ]; then  
 echo "deploy.baimizhou.net" > CNAME  
fi  
  
git add -A .  
git commit -m "rebuild pages at ${rev}"  
git push -q upstream HEAD:gh-pages

这就是这个builder做的事情——其中最主要的一个任务是grunt，它所做的就是:

grunt.registerTask('default', ['clean', 'assemble', 'copy']);

## Content：JSON格式

在使用Github和Travis CI完成Content的时候，发现没有一个好的Webhook。虽然我们的Content只能存储一些数据，但是放一个trigger脚本也是可以原谅的。

var Travis = require('travis-ci');  
  
var repo = "phodal-archive/echeveria-deploy";  
  
var travis = new Travis({  
 version: '2.0.0'  
});  
  
travis.authenticate({  
 github\_token: process.env.GH\_TOKEN  
  
}, function (err, res) {  
 if (err) {  
 return console.error(err);  
 }  
  
 travis.repos(repo.split('/')[0], repo.split('/')[1]).builds.get(function (err, res) {  
 if (err) {  
 return console.error(err);  
 }  
  
 travis.requests.post({  
 build\_id: res.builds[0].id  
 }, function (err, res) {  
 if (err) {  
 return console.error(err);  
 }  
 console.log(res.flash[0].notice);  
 });  
 });  
});

这里主要依赖于Travis CI来完成这部分功能，这时候我们还需要数据。

### 从Schema到数据库

我们在我们数据库中定义好了Schema——对一个数据库的结构描述。在《[编辑-发布-开发分离](https://www.phodal.com/blog/editing-publishing-coding-seperate/) 》一文中我们说到了echeveria-content的一个数据文件如下所示：

{  
 "title": "白米粥",  
 "author": "白米粥",  
 "url": "baimizhou",  
 "date": "2015-10-21",  
 "description": "# Blog post \n > This is an example blog post \n Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. ",  
 "blogpost": "# Blog post \n > This is an example blog post \n Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. \n Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum."  
 }

比起之前的直接生成静态页面这里的数据就是更有意思地一步了，我们从数据库读取数据就是为了生成一个JSON文件。何不直接以JSON的形式存储文件呢？

我们都定义了这每篇文章的基本元素:

1. title
2. author
3. date
4. description
5. content
6. url

即使我们使用NoSQL我们也很难逃离这种模式。我们定义这些数据，为了在使用的时候更方便。存储这些数据只是这个过程中的一部分，下部分就是取出这些数据并对他们进行过滤，取出我们需要的数据。

Web的骨架就是这么简单，当然APP也是如此。难的地方在于存储怎样的数据，返回怎样的数据。不同的网站存储着不同的数据，如淘宝存储的是商品的信息，Google存储着各种网站的数据——人们需要不同的方式去存储这些数据，为了更好地存储衍生了更多的数据存储方案——于是有了GFS、Haystack等等。运营型网站想尽办法为最后一公里努力着，成长型的网站一直在想着怎样更好的返回数据，从更好的用户体验到机器学习。而数据则是这个过程中不变的东西。

尽管，我已经想了很多办法去尽可能减少元素——在最开始的版本里只有标题和内容。然而为了满足我们在数据库中定义的结构，不得不造出来这么多对于一般用户不友好的字段。如链接名是为了存储的文件名而存在的，即这个链接名在最后会变成文件名：

repo.write('master', 'contents/' + data.url + '.json', stringifyData, 'Robot: add article ' + data.title, options, function (err, data) {  
 if(data.commit){  
 that.setState({message: "上传成功" + JSON.stringify(data)});  
 that.refs.snackbar.show();  
 that.setState({  
 sending: 0  
 });  
 }  
 });

然后，上面的数据就会变成一个对象存储到“数据库”中。

今天 ，仍然有很多人用Word、Excel来存储数据。因为对于他们来说，这些软件更为直接，他们简单地操作一下就可以对数据进行排序、筛选。数据以怎样的形式存储并不重要，重要的是他们都以文件的形式存储着。

### git作为NoSQL数据库

不同的数据库会以不同的形式存储到文件中去。blob是git中最为基本的存储单位，我们的每个content都是一个blob。redis可以以rdb文件的形式存储到文件系统中。完成一个CMS，我们并不需要那么多的查询功能。

这些上千年的组织机构，只想让人们知道他们想要说的东西。

我们使用NoSQL是因为：

1. 不使用关系模型
2. 在集群中运行良好
3. 开源
4. 无模式
5. 数据交换格式

我想其中只有两点对于我来说是比较重要的集群与数据格式。但是集群和数据格式都不是我们要考虑的问题。。。

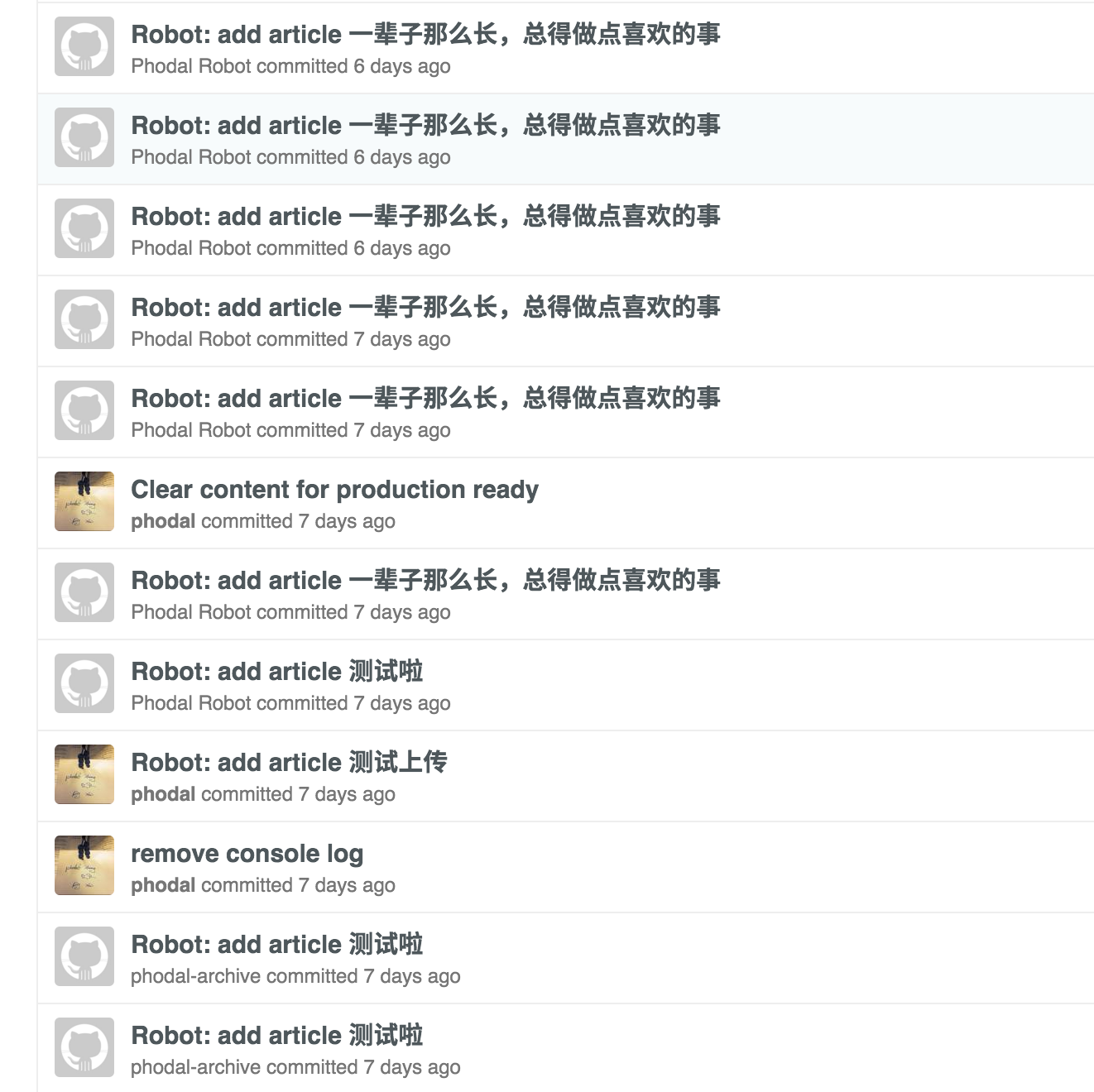
我们也不存在数据格式的问题、开源的问题，什么问题都没有。。除了，我们之前说到的查询——但是这是可以解决的问题，我们甚至可以返回不同的历史版本的。在这一点上git做得很好，他不会像WordPress那样存储多个版本。

JSON文件 + Nginx就可以变成这样一个合理的API，甚至是运行方式。我们可以对其进行增、删、改、查，尽管就当前来说查需要一个额外的软件来执行，但是为了实现一个用得比较少的功能，而去花费大把的时间可能就是在浪费。

git的“API”提供了丰富的增、删、改功能——你需要commit就可以了。我们所要做的就是:

1. git commit
2. git push

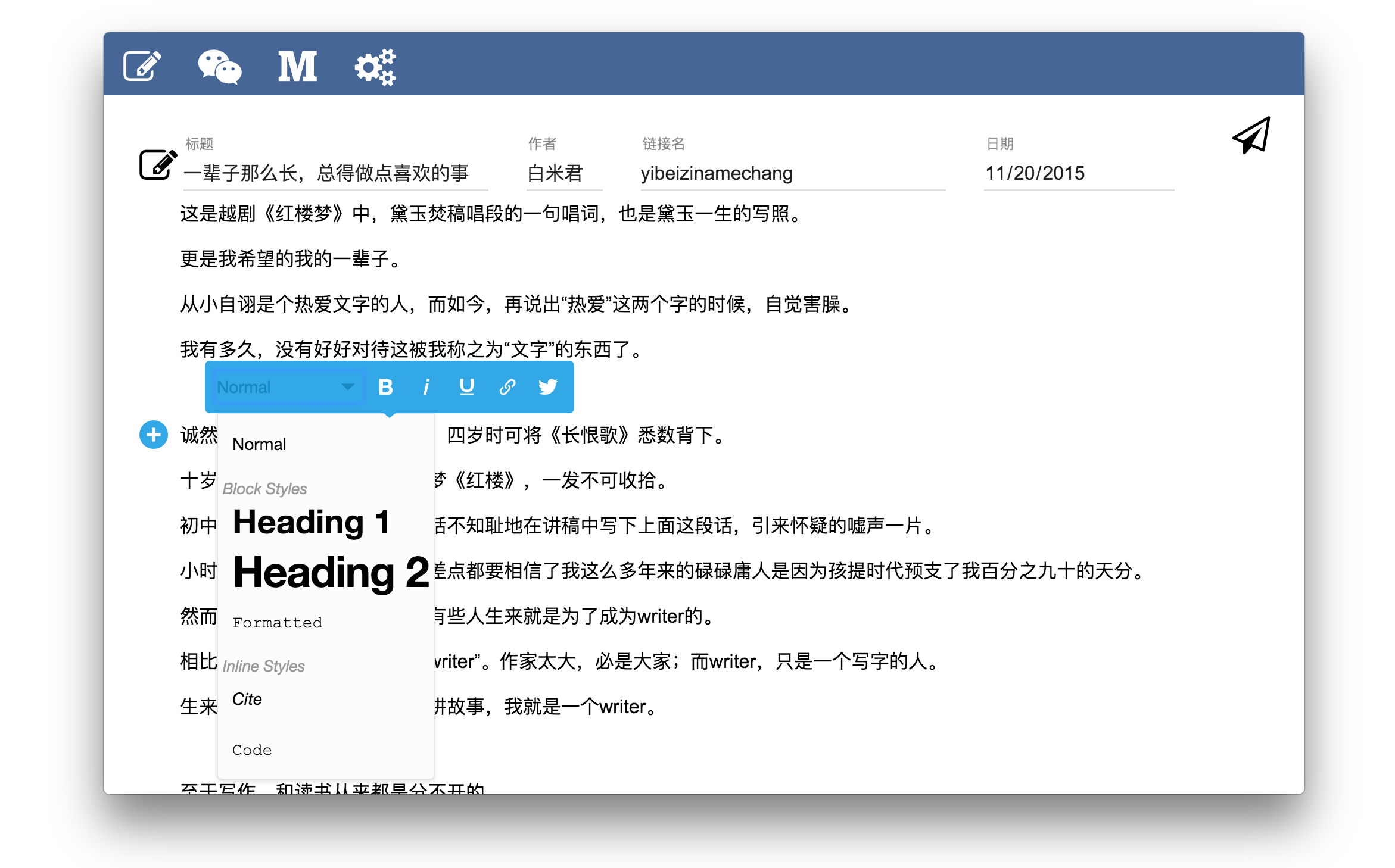
于是，就会有一个很忙的Travis-Github Robot在默默地为你工作。



Robot提交代码

## 一键发布：编辑器

为了实现之前说到的编辑-发布-开发分离的CMS，我还是花了两天的时间打造了一个面向普通用户的编辑器。效果截图如下所示：



编辑器

作为一个普通用户，这是一个很简单的软件。除了Electron + Node.js + React作了一个140M左右的软件，尽管压缩完只有40M左右 ，但是还是会把用户吓跑的。不过作为一个快速构建的原型已经很不错了——构建速度很快、并且运行良好。

* Electron
* React
* Material UI
* Alloy Editor

尽管这个界面看上去还是稍微复杂了一下，还在试着想办法将链接名和日期去掉——问题是为什么会有这两个东西？

Webpack 打包

if (process.env.HOT) {  
 mainWindow.loadUrl('file://' + \_\_dirname + '/app/hot-dev-app.html');  
 } else {  
 mainWindow.loadUrl('file://' + \_\_dirname + '/app/app.html');  
 }

上传代码

repo.write('master', 'content/' + data.url + '.json', stringifyData, 'Robot: add article ' + data.title, options, function (err, data) {  
 if(data.commit){  
 that.setState({message: "上传成功" + JSON.stringify(data)});  
 that.refs.snackbar.show();  
 that.setState({  
 sending: 0  
 });  
 }  
});

当我们点下发送的时侯，这个内容就直接提交到了Content Repo下，如上上图所示。

当我们向Content Push代码的时候，就会运行一下Trigger脚本：

after\_success:  
 - node trigger-build.js

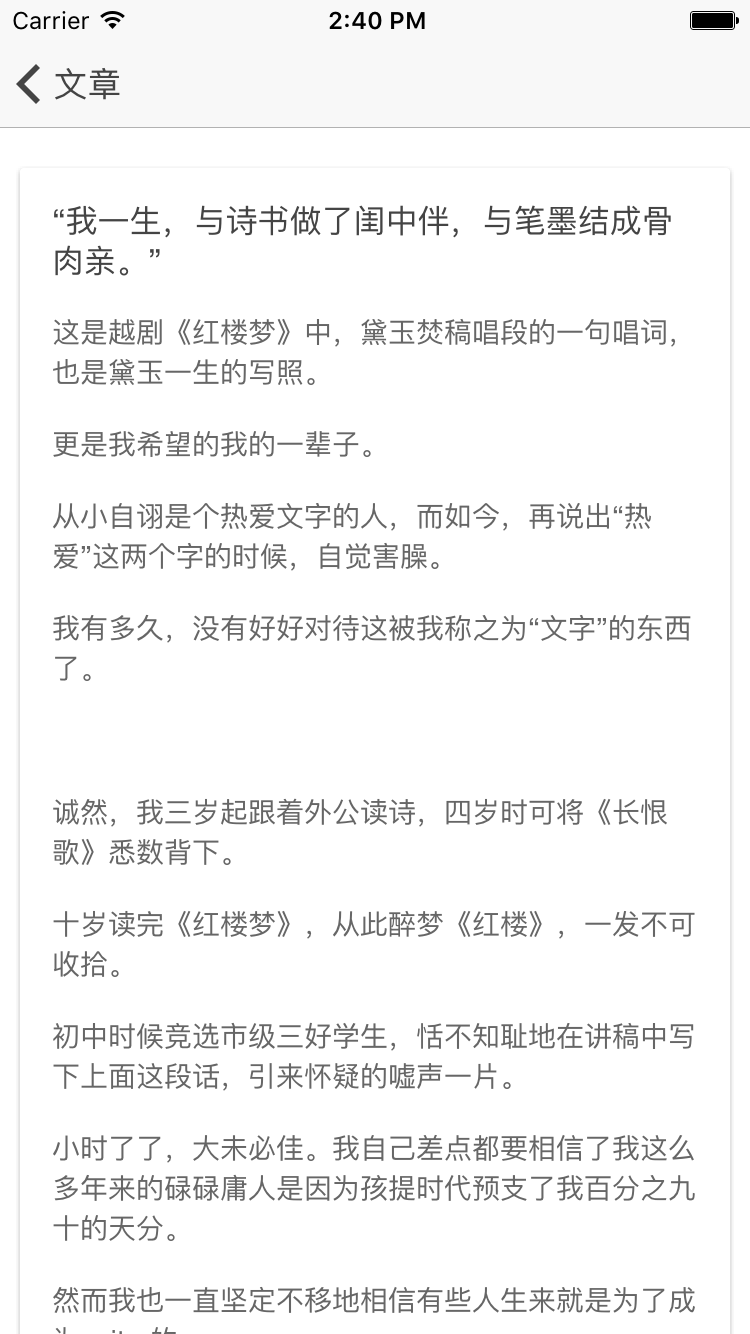
脚本的代码如下所示：

var Travis = require('travis-ci');  
  
var repo = "phodal-archive/echeveria-deploy";  
var travis = new Travis({  
 version: '2.0.0'  
});  
  
travis.authenticate({  
 github\_token: process.env.GH\_TOKEN  
  
}, function (err, res) {  
 if (err) {  
 return console.error(err);  
 }  
 travis.repos(repo.split('/')[0], repo.split('/')[1]).builds.get(function (err, res) {  
 if (err) {  
 return console.error(err);  
 }  
  
 travis.requests.post({  
 build\_id: res.builds[0].id  
 }, function (err, res) {  
 if (err) {  
 return console.error(err);  
 }  
 console.log(res.flash[0].notice);  
 });  
 });  
});

由于，我们在这个过程我们的Content提交的是JSON数据，我们可以直接用这些数据做一个APP。

## 移动应用

为了快速开发，这里我们使用了Ionic + ngCordova来开发 ，最后效果图如下所示：



移动应用

在这个代码库里，主要由两部分组成：

1. 获取全部文章
2. 获取特定文章

为了获取全部文章就意味着，我们在Builder里，需要一个task来合并JSON文件，并删掉其中的一些无用的内容，如articleHTML和article。最后，将生成一个名为articles.json。

if (!grunt.file.exists(src))  
 throw "JSON source file \"" + chalk.red(src) + "\" not found.";  
else {  
 var fragment;  
 grunt.log.debug("reading JSON source file \"" + chalk.green(src) + "\"");  
 try {  
 fragment = grunt.file.readJSON(src);  
 }  
 catch (e) {  
 grunt.fail.warn(e);  
 }  
 fragment.description = sanitizeHtml(fragment.article).substring(0, 200);  
 delete fragment.article;  
 delete fragment.articleHTML;  
 json.push(fragment);  
}

接着，我们就可以获取所有的文章然后显示~~。在这里又顺便加了一个pullToRefresh。

.controller('ArticleListsCtrl', function ($scope, Blog) {  
 $scope.articles = null;  
 $scope.blogOffset = 0;  
 $scope.doRefresh = function () {  
 Blog.async('http://deploy.baimizhou.net/api/blog/articles.json').then(function (results) {  
 $scope.articles = results;  
 });  
 $scope.$broadcast('scroll.refreshComplete');  
 $scope.$apply()  
 };  
 Blog.async('http://deploy.baimizhou.net/api/blog/articles.json').then(function (results) {  
 $scope.articles = results;  
 });  
 })

最后，当我们点击特定的url，将跳转到相应的页面：

<ion-item class="item item-icon-right" ng-repeat="article in articles" type="item-text-wrap" href="#/app/article/{{article.url}}">  
 <h2>{{article.title}}</h2>  
 <i class="icon ion-ios-arrow-right"></i>  
</ion-item>

就会交由相应的Controller来处理。

.controller('ArticleCtrl', function ($scope, $stateParams, $sanitize, $sce, Blog) {  
 $scope.article = {};  
 Blog.async('http://deploy.baimizhou.net/api/' + $stateParams.slug + '.json').then(function (results) {  
 $scope.article = results;  
 $scope.htmlContent = $sce.trustAsHtml($scope.article.articleHTML);  
 });  
  
 });

## 小结

尽管没有一个更成熟的环境可以探索这其中的问题，但是我想对于当前这种情况来说，它是非常棒的解决方案。我们面向的不是那些技术人员，而是一般的用户。他们能熟练使用的是：编辑器和APP。

1. 不会因为后台的升级来困扰他们，也不会受其他组件的影响。
2. 开发人员不需要担心，某个功能影响了编辑器的使用。
3. Ops不再担心网站的性能问题——然后要么转为DevOps、要么被Fire。

### 其他

最后的代码库：

1. Content: <https://github.com/phodal-archive/echeveria-content>
2. Code: <https://github.com/phodal-archive/echeveria-deploy>
3. 移动应用: <https://github.com/phodal-archive/echeveria-mobile>
4. 桌面应用: <https://github.com/phodal/echeveria-editor>
5. Github Pages: <https://github.com/phodal-archive/echeveria-deploy/tree/gh-pages>

# 无栈篇：架构设计

## 博客与技术驱动

### 技术组成

## Lan与架构设计

在设计 lan (Github: <https://github.com/phodal/lan>) 物联网平台的时候，结合之前的一些经验，构建出一个实际应用中的物联网构架模型。

然后像[lan](https://github.com/phodal/lan)这样的应用，在里面刚属于服务层。

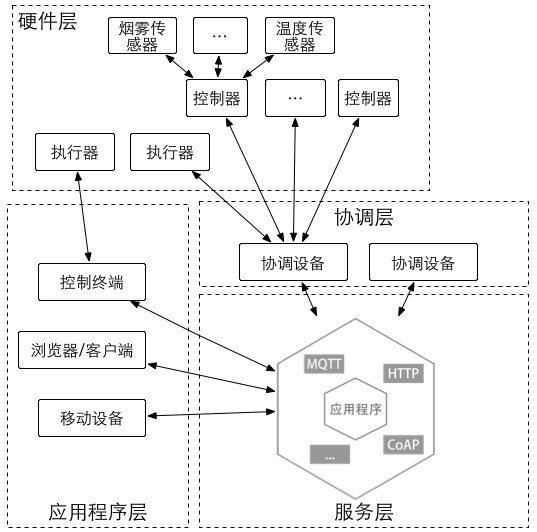
### 物联网层级结构

通常，我们很容易在网上看到如下图所示的三层结构:

[物联网三层结构][1]

从理论上划分这样的层级结构是没有问题的，也是有各种理论依据。然而理论和现实往往是严重脱轨的，如上图所示，图中将网络层单独分为了一层，而并没有独立出应用程序相关的功能。

从实践的角度上，我更愿意用如下的架构来构建我的物联网系统。



物联网层级结构

其功能可以用下表来表示。

|  |  |  |
| --- | --- | --- |
| 层级 | 作用 | 与下一层级的连接方式 |
| 硬件层 | 获取、发送传感器数据，执行指令 | 串口、蓝牙、有线、SPI、WiFi、USB等等 |
| 协调层 | 协调硬件层与服务器的通信，并负责处理部分数据 | 网络连接及硬件层的连接方式 |
| 服务层 | 以视为服务器层 | 网络连接 |
| 应用程序层 | 为用户提供交互功能 | 网络连接 |

硬件层包含了数据众多的传感器、控制器、以及执行器，通常这部份会由硬件人员与硬件开发人员一起协作和开发。而协调层则是充当硬件与服务层通信的桥梁，这是在系统中需要**特别考虑**的部份，一个物联网系统的设计主要**取决于这个层级**。

#### 物联网服务层

而服务层的核心是传统的Web应用程序的结构，只是协议层变成了一些适配器，我们需要支持不同的协议，这导致了我们在这个层需要有一个更好的结构，故而我们建议使用**六边形架构**。而在实际中，用户最后接触到的便是应用程序层，在这一层中需要有很好的用户体验设计及流畅度。

因而在设计[Lan](https://github.com/phodal/lan)物联网平台的时候，参考了之前的[物联网平台](https://github.com/phodal/diaonan)的设计，增加了用户授权以及模块化加载思想。

上图的模型可以让我们脱离具体的框架与实现，关注于业务上逻辑。

### 分层架构

### 六边形架构

# 消息队列

“消息”是在两台计算机间传送的数据单位。消息可以非常简单，例如只包含文本字符串；也可以更复杂，可能包含嵌入对象。

在复杂的系统里，人们使用消息队列不仅是为了解决跨系统的异步通信，也是为了系统间的耦合。

## JMS

## MQ

# 模式篇：设计与架构

设计模式算是在面向对象中比较有趣的东西，特别是对于像我，这样的用得不是很多的。虽然有时候也会用上，但是并不知道用的是怎样的模式。之前学习了一段时间的设计模式，实际上也就是将平常经常用到的一些东西进行了总结，如此而已。学习设计模式的另外一个重要的意义在于，我们使用了设计模式的时候我们会知道自己使用了，并且还会知道用了是怎样的设计模式。

至于设计模式这个东西和有些东西一样，是发现的而不是发明的，换句话说，我们可以将经常合到一起的几种模式用一个新的模式来命名，它是复合模式，但是也可以用别的模式来命名。

设计模式算是简化了我们在面向对象设计时候的诸多不足，这个在系统设计的初期有时候会有一定的作用，不过多数时候对于我来说，会用上他的时候，多半是在重构的时候，因为不是很熟悉。

## 观察者模式

观察者模式又叫做发布-订阅（Publish/Subscribe）模式、模型-视图（Model/View）模式、源-监听器（Source/Listener）模式或从属者（Dependents）模式。

观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，会通知所有观察者对象，使它们能够自动更新自己。

一个软件系统常常要求在某一个对象的状态发生变化的时候，某些其它的对象做出相应的改变。做到这一点的设计方案有很多，但是为了使系统能够易于复用，应该选择低耦合度的设计方案。减少对象之间的耦合有利于系统的复用，但是同时设计师需要使这些低耦合度的对象之间能够维持行动的协调一致，保证高度的协作（Collaboration）。观察者模式是满足这一要求的各种设计方案中最重要的一种。

简单的来说，就是当我们监测到一个元素变化的时候，另外的元素依照此而改变。

### Ruby观察者模式

Ruby中为实现Observer模式提供了名为observer的库，observer库提供了Observer模块。 其API如下所示

方法名 | 功 能 -------|----------------- add\_observer(observer) | 添加观察者 delete\_observer(observer) | 删除特定观察者 delete\_observer | 删除观察者 count\_observer | 观察者的数目 change(state=true) | 设置更新标志为真 changed? | 检查更新标志 notify\_observer(\*arg) | 通知更新，如果更新标志为真，调用观察者带参数arg的方法

#### Ruby观察者简单示例

这里要做的就是获取一个json数据，将这个数据更新出来。

获取json数据，同时解析。

require 'net/http'  
require 'rubygems'  
require 'json'  
  
class GetData  
 attr\_reader:res,:parsed  
  
 def initialize(uri)  
 uri=URI(uri)  
 @res=Net::HTTP.get(uri)  
 @parsed=JSON.parse(res)  
 end  
  
 def id  
 @parsed[0]["id"]  
 end  
  
 def sensors1  
 @parsed[0]["sensors1"].round(2)  
 end  
  
 def sensors2  
 @parsed[0]["sensors2"].round(2)  
 end  
  
 def temperature  
 @parsed[0]["temperature"].round(2)  
 end  
  
 def led1  
 @parsed[0]["led1"]  
 end  
  
end

下面这个也就是重点，和观察者相关的，就是被观察者，由这个获取数据。 通过changed ，同时用notify\_observer方法告诉观察者

require 'rubygems'  
require 'thread'  
require 'observer'  
require 'getdata'  
require 'ledstatus'  
  
class Led   
 include Observable  
   
 attr\_reader:data  
 def initialize  
 @uri='http://www.xianuniversity.com/athome/1'  
 end  
 def getdata  
 loop do   
 changed()  
 data=GetData.new(@uri)  
 changed  
 notify\_observers(data.id,data.sensors1,data.sensors2,data.temperature,data.led1)  
 sleep 1  
 end  
 end  
end

然后让我们新建一个观察者

class LedStatus  
 def update(arg,sensors1,sensors2,temperature,led1)  
 puts "id:#{arg},sensors1:#{sensors1},sensors2:#{sensors2},temperature:#{temperature},led1:#{led1}"  
 end  
end

测试

require 'spec\_helper'  
  
describe LedStatus do  
 let(:ledstatus){LedStatus.new()}  
  
 describe "Observable" do  
 it "Should have a result" do   
 led=Led.new  
 led.add\_observer(ledstatus)  
 led.getdata  
 end  
 end  
end

测试结果如下所示

phodal@linux-dlkp:~/tw/observer&gt; rake  
/usr/bin/ruby1.9 -S rspec ./spec/getdata\_spec.rb ./spec/ledstatus\_spec.rb  
id:1,sensors1:22.0,sensors2:11.0,temperature:10.0,led1:0  
id:1,sensors1:22.0,sensors2:11.0,temperature:10.0,led1:1  
id:1,sensors1:22.0,sensors2:11.0,temperature:10.0,led1:0  
id:1,sensors1:22.0,sensors2:11.0,temperature:10.0,led1:1  
id:1,sensors1:22.0,sensors2:11.0,temperature:10.0,led1:1  
id:1,sensors1:22.0,sensors2:11.0,temperature:10.0,led1:1

使用Ruby自带的Observer库的优点是，让我们可以简化相互之间的依赖性。同时，也能简化程序的结构，相比于自己写observer的情况下。

### PUB/SUB

## 模板方法

原本对于设计模式的写作还不在当前的计划中，然而因为在写TWU作业的时候，觉得代码写得不好，于是慢慢试着一点点重构，重新看着设计模式。也开始记录这一点点的方法，至少这些步骤是必要的。

### 从基本的App说起

对于一个基本的C/C++/Java/Python的Application来说，他只需要有一个Main函数就够了。对于一个好一点的APP来说，他可能是下面的步骤，

main(){  
 init();  
 while(!condition()){  
 do();  
 }  
}

上面的代码是我在学51/AVR等各式嵌入式设备时，经常是按上面的写法写的，对于一个更符合人性的App来说他应该会有一个退出函数。

main(){  
 init();  
 while(!condition()){  
 do();  
 }  
 exit();  
}

于是很幸运地我找到了这样的一个例子。

过去看过Arduino的代码，了解过他是如何工作的，对于一个Arduino的代码来说，必要的两个函数就是。

void setup() {  
}  
  
void loop() {  
}

setup()函数相当于上面的init()，而loop()函数刚相当于上面的do()。似乎这就是我们想要的东西，看看Arduino目录中的Arduino.h就会发现，如下的代码(删减部分代码)

#include <Arduino.h>  
  
int main(void)  
{  
 init();  
 setup();   
 for (;;) {  
 loop();  
 if (serialEventRun) serialEventRun();  
 }  
   
 return 0;  
}

代码中的for(;;)看上去似乎比while(True)容易理解得多，这也就是为什么嵌入式中经常用到的是for(;;)，从某种意义上来说两者是等价的。再有不同的地方，就是gcc规定了,main()函数不能是void。so,两者是差不多的。只是没有，并没有在上面看到模板方法，等等。我们在上面所做的事情，便是创建一个框架。

### Template Method

**模板方法**： 在一方法中定义一个算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。

对于我来说，我就是在基本的App中遇到的情况是一样的，在我的例子中，一开始我的代码是这样写的。

public static void main(String[] args) throws IOException {  
 initLibrary();  
 while(!isQuit){  
 loop();  
 }  
 exit;  
}  
  
protected void initLibrary(); {  
 System.out.println(welcomeMessage);  
}  
  
protected void loop() {  
 String key = "";  
 Scanner sc = new Scanner(System.in);  
 key = sc.nextLine();  
  
 System.out.println(results);  
 if(key.equals("Quit")){  
 setQuit();  
 }  
}  
  
protected void exit() {  
 System.out.println("Quit Library");  
}

只是这样写感觉很是别扭，看上去一点高大上的感觉，也木有。于是，打开书，找找灵感，就在《敏捷软件开发》一书中找到了类似的案例。Template Method模式可以分离能用的算法和具体的上下文，而我们通用的算法便是。

main(){  
 init();  
 while(!condition()){  
 do();  
 }  
 exit();  
}

看上去正好似乎我们当前的案例，于是便照猫画虎地来了一遍。

### Template Method实战

创建了一个名为App的抽象基类，

public abstract class App {  
 private boolean isQuit = false;  
  
 protected abstract void loop();  
 protected abstract void exit();  
  
 private boolean quit() {  
 return isQuit;  
 }  
  
 protected boolean setQuit() {  
 return isQuit = true;  
 }  
  
 protected abstract void init();  
  
 public void run(){  
 init();  
 while(!quit()){  
 loop();  
 }  
 exit();  
 }  
}

而这个也和书中的一样，是一个通用的主循环应用程序。从应用的run函数中，可以看到主循环。而所有的工作也都交付给抽象方法，于是我们的LibraryApp就变成了

public class LibraryApp extends App {  
 private static String welcomeMessage = "Welcome to Biblioteca library";  
  
 public static void main(String[] args) throws IOException {  
 (new LibraryApp()).run();  
 }  
  
 protected void init() {  
 System.out.println(welcomeMessage);  
 }  
  
 protected void loop() {  
 String key = "";  
 Scanner sc = new Scanner(System.in);  
 key = sc.nextLine();  
  
 if(key.equals("Quit")){  
 setQuit();  
 }  
 }  
  
 protected void exit() {  
 System.out.println("Quit Library");  
 }  
}

然而，如书中所说这是一个很好的用于示范TEMPLATE METHOD模式的例子，却不是一个合适的例子。

## Pipe and Filters

继续码点关于架构设计的一些小心得。架构是什么东西并没有那么重要，重要的是知道它存在过。我会面对不同的架构，有一些不同的想法。一个好的项目通常是存在一定的结构，就好像人们在建造房子的时候也都会有结构有一样。

我们看不到的架构，并不意味着这个架构不存在。

### Unix Shell

最出名的Pipe便是Unix中的Shell

**管道（英语：Pipeline）是原始的软件管道：即是一个由标准输入输出链接起来的进程集合，所以每一个进程的输出（stdout）被直接作为下一个进程的输入（stdin）。 每一个链接都由未命名管道实现。过滤程序经常被用于这种设置。**

所以对于这样一个很好的操作便是，统计某种类型的文件的个数:

ls -alh dot | grep .dot | wc -l

在执行

ls -alh dot

的输出便是下一个的输入，直至最后一个输出。

这个过程有点类似于工厂处理废水，

[pipe and filter][1]

上图是一个理想模型~~。

一个明显地步骤是，水中的杂质越来越少。

### Pipe and Filter模式

**Pipe and Filter**适合于处理数据流的系统。每个步骤都封装在一个过滤器组件中，数据通过相邻过滤器之间的管道传输。

* **pipe**: 传输、缓冲数据。
* **filter**: 输入、处理、输出数据。

这个处理过程有点类似于我们对数据库中数据的处理，不过可不会有这么多步骤。

### Fluent API

这个过程也有点类似于Fluent API、链式调用，只是这些都是DSL的一种方式。

流畅接口的初衷是构建可读的API，毕竟代码是写给人看的。

类似的，简单的看一下早先我们是通过方法级联来操作DOM

var btn = document.createElement("BUTTON"); // Create a <button> element  
var t = document.createTextNode("CLICK ME"); // Create a text node  
btn.appendChild(t); // Append the text to <button>  
document.body.appendChild(btn); // Append <button> to <body>

而用jQuery写的话，便是这样子

$('<span>').append("CLICK ME");

等等

于是回我们便可以创建一个简单的示例来展示这个最简单的DSL

Func = (function() {  
 this.add = function(){  
 console.log('1');  
 return this;  
 };  
 this.result = function(){  
 console.log('2');  
 return this;  
 };  
 return this;  
});  
  
var func = new Func();  
func.add().result();

然而这看上去像是表达式生成器。

### DSL 表达式生成器

表达式生成器对象提供一组连贯接口，之后将连贯接口调用转换为对底层命令-查询API的调用。

这样的API，我们可以在一些关于数据库的API中看到:

var query =  
 SQL('select name, desc from widgets')  
 .WHERE('price < ', $(params.max\_price), AND,  
 'clearance = ', $(params.clearance))  
 .ORDERBY('name asc');

链式调用有一个问题就是收尾，同上的代码里面我们没有收尾，这让人很迷惑。。加上一个query和end似乎是一个不错的结果。

### Pipe and Filter模式实战

所以，这个模式实际上更适合处理数据，如用Hadoop处理数据的时候，我们会用类似于如下的方法来处理我们的数据:

A = FOREACH LOGS\_BASE GENERATE ToDate(timestamp, 'dd/MMM/yyyy:HH:mm:ss Z') as date, ip, url,(int)status,(int)bytes,referrer,useragent;  
B = GROUP A BY (timestamp);  
C = FOREACH B GENERATE FLATTEN(group) as (timestamp), COUNT(A) as count;  
D = ORDER C BY timestamp,count desc;

每一次都是在上一次处理完的结果后，再处理的。

参考书目

* 《Head First 设计模式》
* 《设计模式》
* 《敏捷软件开发 原则、模式与实践》
* 《 面向模式的软件架构:模式系统》
* 《Java应用架构设计》

# 数据模型与领域

无论是MVC、MVP或者MVVP，都离不开这些基本的要素：数据、表现、领域。

## 数据

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。

### 数据库

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。

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。在那之间，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' // Will result in an attribute that is firstName when user facing but first\_name in the database  
 },  
 lastName: {  
 type: Sequelize.STRING  
 }  
}, {  
 freezeTableName: true // Model tableName will be the same as the model name  
});  
  
User.sync({force: true}).then(function () {  
 // Table created  
 return User.create({  
 firstName: 'John',  
 lastName: 'Hancock'  
 });  
});

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。

### 数据模型

## 领域

### DDD

### 值对象

### DSL

DSL(domain-specific languages)即领域特定语言，唯一能够确定DSL边界的方法是考虑“一门语言的一种特定用法”和“该语言的设计者或使用者的意图。在试图设计一个DSL的时候，发现了一些有意思的简单的示例。

### DSL示例

#### jQuery 最流行的DSL

jQuery是一个Internal DSL的典型的例子。它是在一门现成语言内实现针对领域问题的描述。

$('.mydiv').addClass('flash').draggable().css('color', 'blue')

这也就是其最出名的**链式方法调用**。

#### Cucumber.js

Cucumber, the popular Behaviour-Driven Development tool, brought to your JavaScript stack。它是使用通用语言描述该领域的问题。

Feature: Example feature  
 As a user of cucumber.js  
 I want to have documentation on cucumber  
 So that I can concentrate on building awesome applications  
  
 Scenario: Reading documentation  
 Given I am on the Cucumber.js GitHub repository  
 When I go to the README file  
 Then I should see "Usage" as the page title

#### CoffeeScript

发明一门全新的语言描述该领域的问题。

math =  
 root: Math.sqrt  
 square: square  
 cube: (x) -> x \* square x

#### JavaScript DSL 示例

所以由上面的结论我们可以知道的是，难度等级应该是

内部DSL < 外部DSL < 语言工作台(这是怎么翻译的)

接着在网上找到了一个高级一点的内部DSL示例，如果我们要做jQuery式的链式方法调用也是简单的，但是似乎没有足够的理由去说服其他人。

原文在: <http://alexyoung.org/2009/10/22/javascript-dsl/>，相当于是一个微测试框架。

var DSLRunner = {  
 run: function(methods) {  
 this.ingredients = [];  
 this.methods = methods;  
  
 this.executeAndRemove('first');  
  
 for (var key in this.methods) {  
 if (key !== 'last' && key.match(/^bake/)) {  
 this.executeAndRemove(key);  
 }  
 }  
  
 this.executeAndRemove('last');  
 },  
  
 addIngredient: function(ingredient) {  
 this.ingredients.push(ingredient);  
 },  
  
 executeAndRemove: function(methodName) {  
 var output = this.methods[methodName]();  
 delete(this.methods[methodName]);  
 return output;  
 }  
};  
  
DSLRunner.run({  
 first: function() {  
 console.log("I happen first");  
 },  
  
 bakeCake: function() {  
 console.log("Commencing cake baking");  
 },  
  
 bakeBread: function() {  
 console.log("Baking bread");  
 },  
 last: function() {  
 console.log("last");  
 }  
});

这个想法，看上去就是定义了一些map，然后执行。

接着，又看到了一个有意思的DSL，作者是在解决表单验证的问题[《JavaScript DSL Because I’m Tired of Writing If.. If…If…》](http://byatool.com/ui/javascript-dsl-because-im-tired-of-writing-if-if-if/)：

var rules =  
 ['Username',  
 ['is not empty', 'Username is required.'],  
 ['is not longer than', 7, 'Username is too long.']],  
 ['Name',  
 ['is not empty', 'Name is required.']],  
 ['Password',  
 ['length is between', 4, 6, 'Password is not acceptable.']]];

有一个map对应了上面的方法

var methods = [  
 ['is not empty', isNotEmpty],  
 ['is not longer than', isNotLongerThan],  
 ['length is between', isBetween]];

原文只给了一部分代码

var methodPair = find(methods, function(method) {  
 return car(method) === car(innerRule);  
});  
  
var methodToUse = peek(methodPair);  
  
return function(obj) {  
 var error = peek(innerRule); //error is the last index  
 var values = sink(cdr(innerRule)); //get everything but the error   
 return methodToUse(obj, propertyName, error, values); //construct the validation call  
};

# 写作

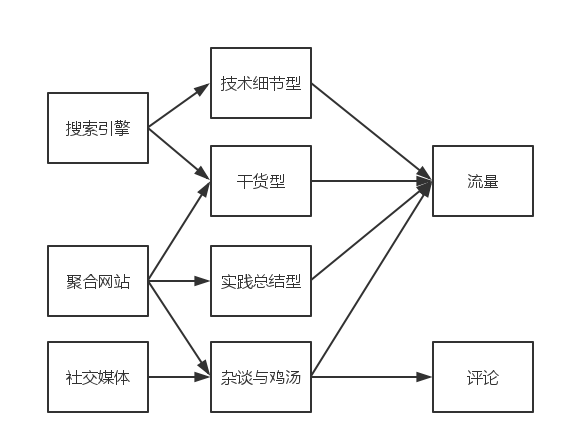
每天有大把的时间刷GitHub，写博客。从我大二的时候，大概六年前开始写技术博客，到现在已经有540+了，大概每年会写一百篇左右。

## 流量来源

首先，我将技术博客分为下面几种类型:

1. 技术细节型。频率: 每天几次，15~30分钟可以写完。
2. 干货型。频率: 每月几次，1~2小时可以搞定。
3. 实践总结型。频率: 每周1～2次，2+小时以上的时间。
4. 杂谈与鸡汤型。频率: 每季度几次，少则几小时，动则上月。

接着，奉献上一章我的博客的用户来源图，分析完大概如下：



traffic-source.png

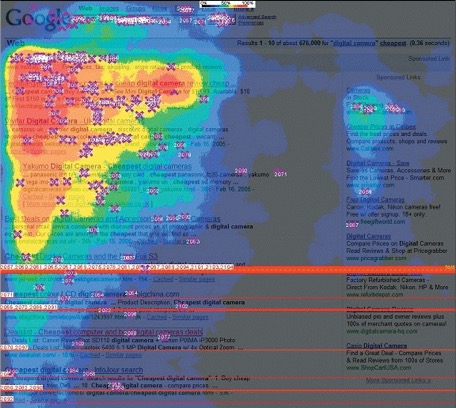
**前三者可以带来很多的流量，后者可以带来大量的评论啦。**

**前三者可以让你在Google上有一个好位置~~，后者可以让你在用户心中有个好位置。**

**通过社交媒体可以让第四种类型的文章，有大量的评论和转载。**

**干货型和实践总结型的文章通过聚合网站来传播的效果最好。**

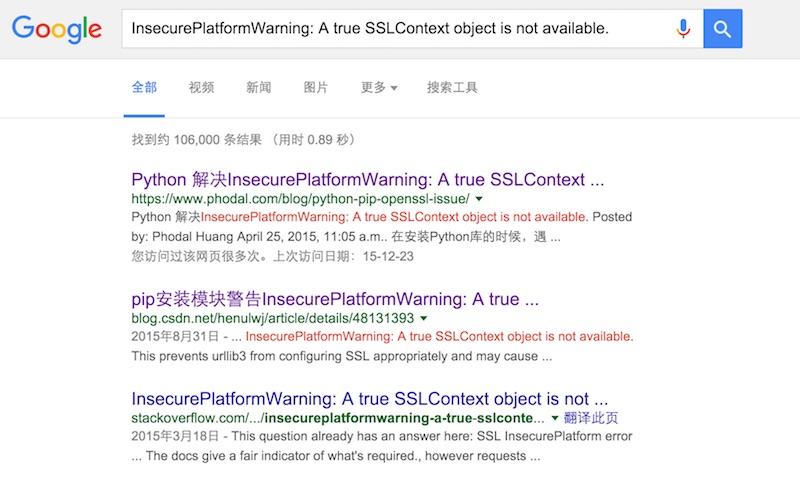
在Google中有个好位置意味着——第一的位置会有50%以上的用户访问，这一点很重要哦~~。再放出神图：



google.png

首先作为一个专业的程序员，我们会用某个关键词去搜索，如“InsecurePlatformWarning: A true SSLContext object is not available.” :如果这时你的标题就是《[Python 解决 InsecurePlatformWarning: A true SSLContext object is not available](https://www.phodal.com/blog/python-pip-openssl-issue/)》，那么你就找到了你的用户了。

**这是我之前用了5分钟不到的文章写的一篇文章，它在半年的时间里带来了6,599个访问量**。它在Google的搜索结果中排第一：



google-search-python-issue.jpg

虽然很不情愿地被排名第二的文章抄袭了——但是我还是第一，这该死的**伪原创**。

而用户不会只访问你的一个页面的:



user-behavior.jpg

**一般来说，前三种类型的用户获取到他们想要的信息就走了，不会留下评论**。至于第四种类型，没有大V的光环，偶尔文章被转企业V转转也能多个几十个粉丝。所以我没有第四种类型的文章太多的经验哈。

## 不同文章类型的写作要点

接着， 让我们再看看上面的几种类型：

### 技术细节型

在我的博客里很大一部分文章都是这一类，主要是原自早先对SEO的研究。对于先这种没有光环的人来说，这是我博客的主要流量来源。这也是让我坚持下去的动力，每天有大概三百个用户来自搜索引擎。

这种类型主要来自于日常工作，但是好像写这一类的人不多。每天我们都会遇到不同的技术问题——如某个第三库更新，某个浏览器bug，如何使用某个技术。因此，我们就会用这样的关键字和词去搜索，反正大部分教程序员使用Google的文章就是这么写的。

既然你想要这样的搜索，那我就直接把搜索结果放前面好了，如：

1. [Mac OS Laravel 安装](https://www.phodal.com/blog/mac-os-install-laravel-4/)
2. [Homebrew versions安装特定版本软件](https://www.phodal.com/blog/use-brew-versions-manage-specif-software-version/)
3. [brew versions gradle安装指定版本gradle](https://www.phodal.com/blog/homebrew-version-manage-gradle-version/)

如果你也擅长用Google的话，我想你就懂了奥秘。标题的前半部分是关键字，后半部分指名意图。

接着，就是贴解决方案和代码咯~~。在这时候需要注意几点：

* 试着去表达一些技术上的细节，也会省去一些技术上的细节。
* 以自己学习的角度去想怎样写可以一步步下来。
* 加入一点点自己的情感，博客和文档是不一样的。（详细见下文）
* 编写更简洁的内容，因为技术博客与此文不同的是，技术博客是面向寻找信息的人。

### 干货型

一般来说，这种类型的文章更类似于GitHub上的那些Awesome类型的Repo。说说几个标题，大概可能就会有点感觉了：

* [每个程序员必知之SEO](https://www.phodal.com/blog/every-programmer-should-know-how-seo/)
* 作为一个前端工程师你了解你的小伙伴么 - chrome
* App 上架相关事宜、解决技巧
* 20 个非常有用的 Java 程序片段
* Java 开发者不容错过的 12 种高效工具

通常来说，这一类型的文章都是作者一段时间对于某一个东西的总结，非常有收藏价值。故而，这种类型的文章会在GitHub或者聚合网站上比较受欢迎。所以，它也更容易传播。

但是写起来的难度比较大，这依赖于你的使用经验。所以，也不是一天、两天就能搞定的。

### 实践总结型

看看标题，可能大家都知道这一类文章要怎么写了：

* JavaScript之闭包与高阶函数（一）
* [编辑-发布-开发分离：git作为NoSQL数据库](https://www.phodal.com/blog/editing-publishing-coding-seperate/)
* [构建基于Javascript的移动CMS——添加滑动](https://www.phodal.com/blog/use-jquery-backbone-mustache-build-mobile-app-cms-add-swipe/)
* Android开发从零开始

这通常是一系列的文章，而这一系列的文章一般是连续写出来的。也因此，我们可以发现很大的书都是由这一类的文章衍生出来的。

这一类的文章更像是干货型和技术细节型的结合，面向特定领域的技术，也属于干货。对于这种类型的文章来说，更依赖于代码——读者需要依据代码一步步往下深入。

所以在这一类型的文章中代码往往比较重要。

### 杂谈与鸡汤型

来几一些标题：

1. 为什么整个互联网行业都缺前端工程师?
2. 程序员职业生涯中应该思考的10个问题
3. 怎么减少编程中的 bug？
4. 为什么大公司看起来都那么糟糕？

BlaBla，简单地来说这一类文章基本上是没有技术的，都是一些以理论为主的概括。同时，写这一类文章的时候，也意料着可能在某一领域有一定的水平——写出来才会有人看。而由于偏向理论，这一类文章看的人往往会更多，因为它并不倾向于使用领域特定的技术。

所以这种类型是受众很广的文章，并且也像鸡汤一样容易传播。同时，这种类型的文章，更依赖于社区媒体及聚合网站的传播。因为大部分情况下，用户不会这么去搜索文章的。

## 如何写博客

对于博客的内容来说，下面几点很重要哦：

1. 标题、小标题。一个好的标题会带来更多的流量的。
2. 内容。以代码为例的文章，最好有代码。代码可以放在GitHub上，一举两得。
3. 图 —— 一图胜千言，如本文第一节中的框架图。，一个框架图可以省去你的很多话语。
4. 排版，又可称为UI设计。基本的语法高亮要有，容易阅读也要有。最简单就是白底黑字。

下面有我之前在《[招聘笔杆子——成为笔杆子](https://www.phodal.com/blog/think-of-rework-be-a-writer/)》一文中说到的一些要点：

### 标题——必须重要，类名

对于写博文的人来说，重点的是如何清楚的去表达他们的想法，标题算是其中之一，这个也就是为什么标题党成为了标题党，而《设计模式》成为了经典。刚开始学编程的时候，更吸引你注意力的可能是《72小时学会Javascript》，而不是《Javascript 权威指南》，兴许让你买前者的原因是因为你能看懂前者，而后者不仅看不懂，而且价格更贵。只是一年以后，《72 小时学会Javascript》被你扔到了垃圾箱，而《Javascript 权威指南》却放在了原来放那本书的位置上。你定义的类难道仅仅应该是class class1么？

### 小标题——地图，method

小标题有点类似于sitemap.xml，只是他就是站点地图，一点就到了相应的地方。他应该直接了解的说这是开始菜单，标题栏，菜单栏，而不应该是简简单单的第一章，如果你真是那样写的话，你写的函数想必是

def fun1  
end

如果你写的是get\_data那么，我想你的函数名应该和你的文章一样，告诉人们，你要的是get\_data。所以不要吝啬你鼠标的一下，它可以承受上百万字的点击。如果因为那样坏了，你可以告诉我，我可以帮你免费换一个欧姆龙的微动，前提是你的鼠标可以换。如果是HTML那么应该是h2 h3，markdown也就是用得比较多的github上的README.md的##或者是###

### 内容——函数体

这里可不是让你用一个让人难以理解的Magic Number，你写得越复杂，别人看的时间就越久，通俗易懂，就是一个很好的开始。你可以把一个个复杂的方法分解出来，或者提炼函数，或者重命名。当你相信你看不懂你的文章，正如你看不懂你写的hello,world我想你是时候去重构你的函数了。

复杂的部分，就用段落来解决，一个函数如同一个段落只应该表达一个思想，太长了就如同这篇文章一样没有多少人会认真去看。你需要给你写的一个精美的代码加一个注释，所以你也需要给你复杂的地方加上个（PS）。

在编写内容的时候需要注意下面的几点：

* 代码。必要的时候贴出代码，会是一种快速有效的方法，工作的时候是为了完成某个任务。
* 列表。列表可以简明扼要的说明你想要表达的内容。
* 图表。一张流程图可以说明程序的步骤，一个表格可以作好一个对比。
* 标题。标题的好处，可以让用户快速定位，让用户找到想要的内容。
* 链接。一个好的URL对于使用英文的人来说，可以利于SEO，对于中文，使用的人可以判断出文章的专业与否。

### 引言——README

我想都会去看的，无论是在破解软件的时候，还是github上面的项目。简单的说说，这篇文章是干什么的，这个程序是干什么的。大家都会，不是么？

没有什么好写的？ 说说你是怎么开始编程吧，然后写在你的博客上，你会发现你会爱不释手的。

## 如何收集写作的灵感

下面是我最近在做的，感觉很不错，推荐一下咯：

一般是先有个论点，然后再论据，最后才写。

第一步，在平时把想到的一些能写的文章用TODO的形式列出来，如我用的是“奇妙清单”。下面就是我最近想写的一些文章的一些标题：



wunderlist

但是有一些还没灵感，只是想到了个大概——所以有的是只是一个标题。这些文章大部分都属于杂谈型的，想到什么就说点什么，但是会有个核心。

第二步，在平时偶然会有点灵感的时候，就用Google Keep创建一个新的记事，稍微写几句，提醒一下自己咯。在这一步，相当于就是找一些论据来支撑自己的观点，等到这些观点差不多的时候就可以写了。



Google Keep

第三步，便是写。写的时候其实没有啥感觉，就如同我之前在《[编程同写作，写代码只是在码字](https://www.phodal.com/blog/write-similar-program-just-typing/)》一文中所说的。不得不再说说这个标题，一看就知道要写什么了~~。

真正的想法都在脑子里，而不在纸上，或者IDE里。

所以，其实写代码的时间很短，真实长的是想出怎么写，文章也是如此。