目录

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。

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

输了，才需要加倍努力

介绍

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 ali
        <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
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.da
  }

  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 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：

然而对于 Domain 相关的 Service 应该放在哪一层，总会有不同的意见：

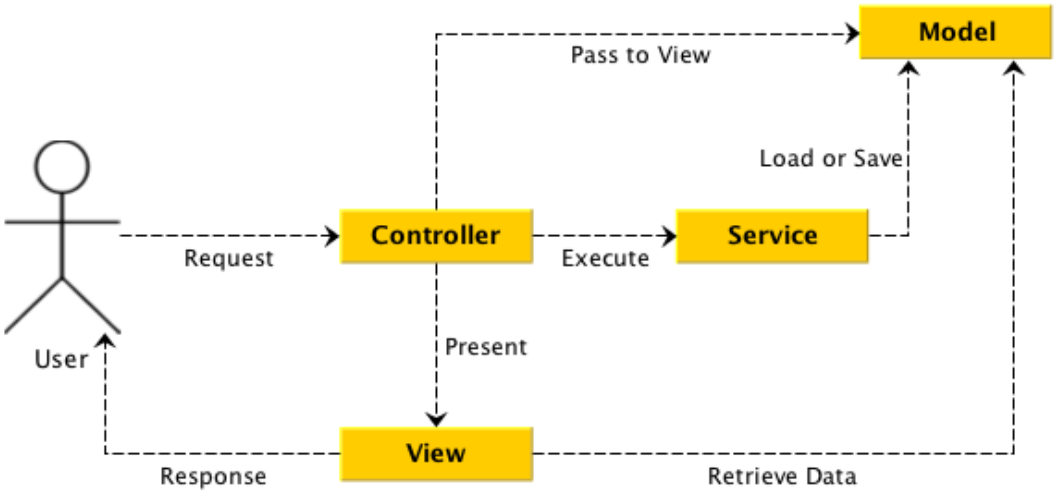

图 1: Service MVC

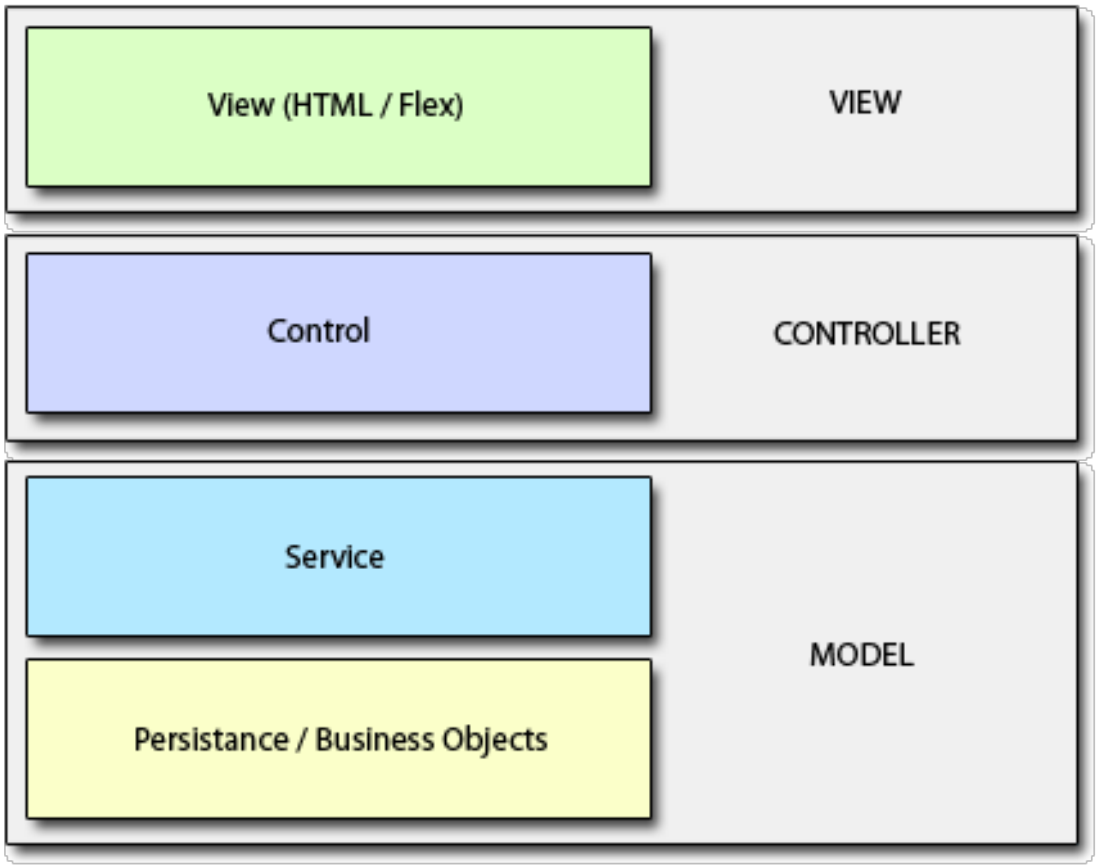

图 2: MVC Player

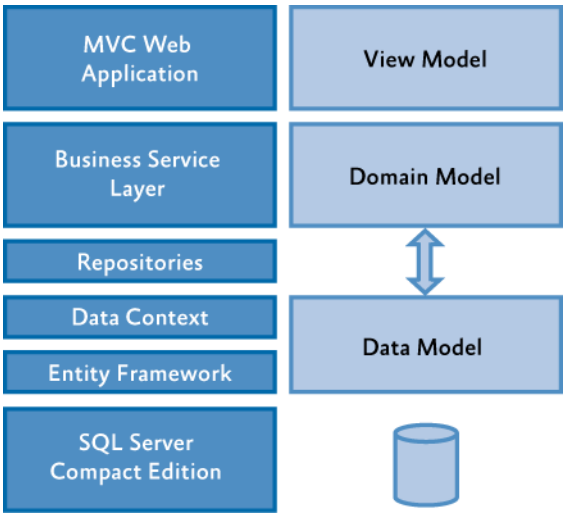

图 3: MS MVC

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.findMainPostByAuthorNameSortedByCreateTi

    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().get
    Authentication authentication = null;
    if(oldAuthentication == null){
        authentication = new UsernamePasswordAuthenticationToken(newUsername,
    }else{
        authentication = new UsernamePasswordAuthenticationToken(newUsername,
    }
    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
    </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:re
{/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
{% 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">{{ bl
    </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。

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。

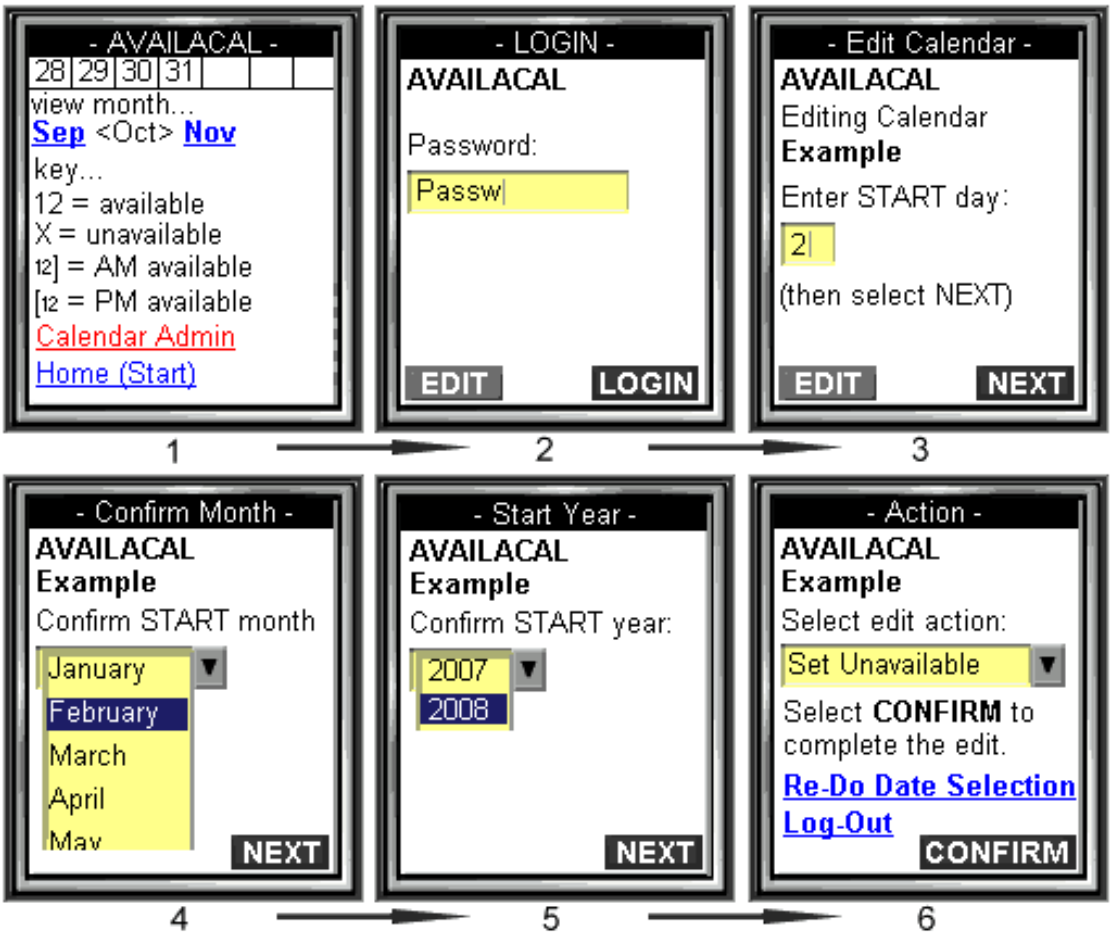
- AVAILACAL -
28 29 30 31
view month...
Sep <Oct> Nov
key...
12 = available
X = unavailable
12] = AM available
[12 = PM available
Calendar Admin
Home (Start)
1
- LOGIN -
AVAILACAL
Password:
Passw
EDIT
LOGIN
2
- Edit Calendar -
AVAILACAL
Editing Calendar
Example
Enter START day:
2
(then select NEXT)
EDIT
NEXT
3
- Confirm Month -
AVAILACAL
Example
Confirm START month
January
February
March
April
May
NEXT
4
- Start Year -
AVAILACAL
Example
Confirm START year:
2007
2008
NEXT
5
- Action -
AVAILACAL
Example
Select edit action:
Set Unavailable
Select CONFIRM to complete the edit.
Re-Do Date Selection
Log-Out
CONFIRM
6

图 4: WAP 网站

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_p
        <em class="since">{% blocktrans with sometime=blog_post.publish_date|
        {% 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_nam
    {% 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 }
    {% endfor %}
```

```
    {% endif %}
    {% endwith %}
    {% blocktrans with sometime=blog_post.publish_date|timesince %}{{ sometime
</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。

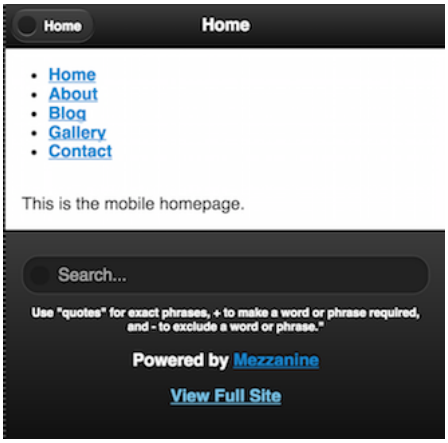

图 5: 移动版网页

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。

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。


21:16
0.03%
80
PHODAL
注销
撰写
事件
搜索
博客
Github 36
After 500:
After 500:
ReThough
ReThough
ReThough
Ionic Auth
Ionic/Corc
Ionic ngM
网站更新
Django 1.

图 6: 应用

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, GetB

    var BlogDetailsView = Backbone.View.extend ({
        el: $("#content"),

        initialize: function () {
            this.params = '#content';
        },

        getBlog: function(slug) {
            var getblog = new GetBlog(this.params, configure['blogPostUrl'] +
            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

变成了直接调用的微服务：

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。

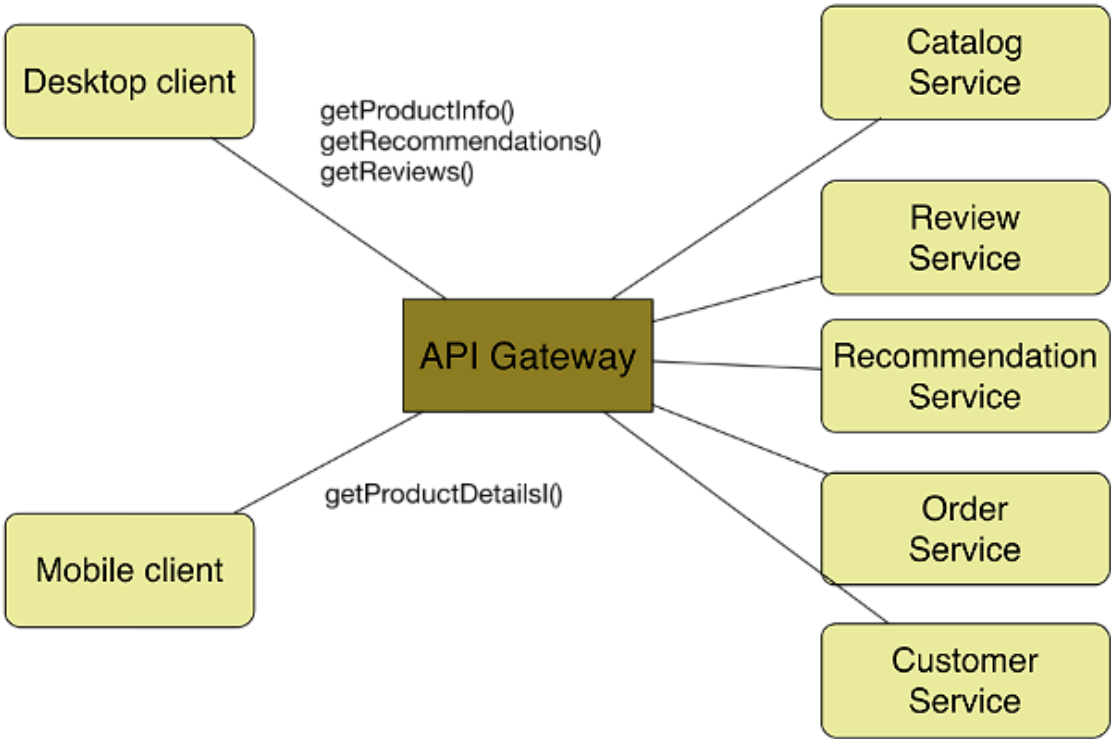

图 7: API Gateway

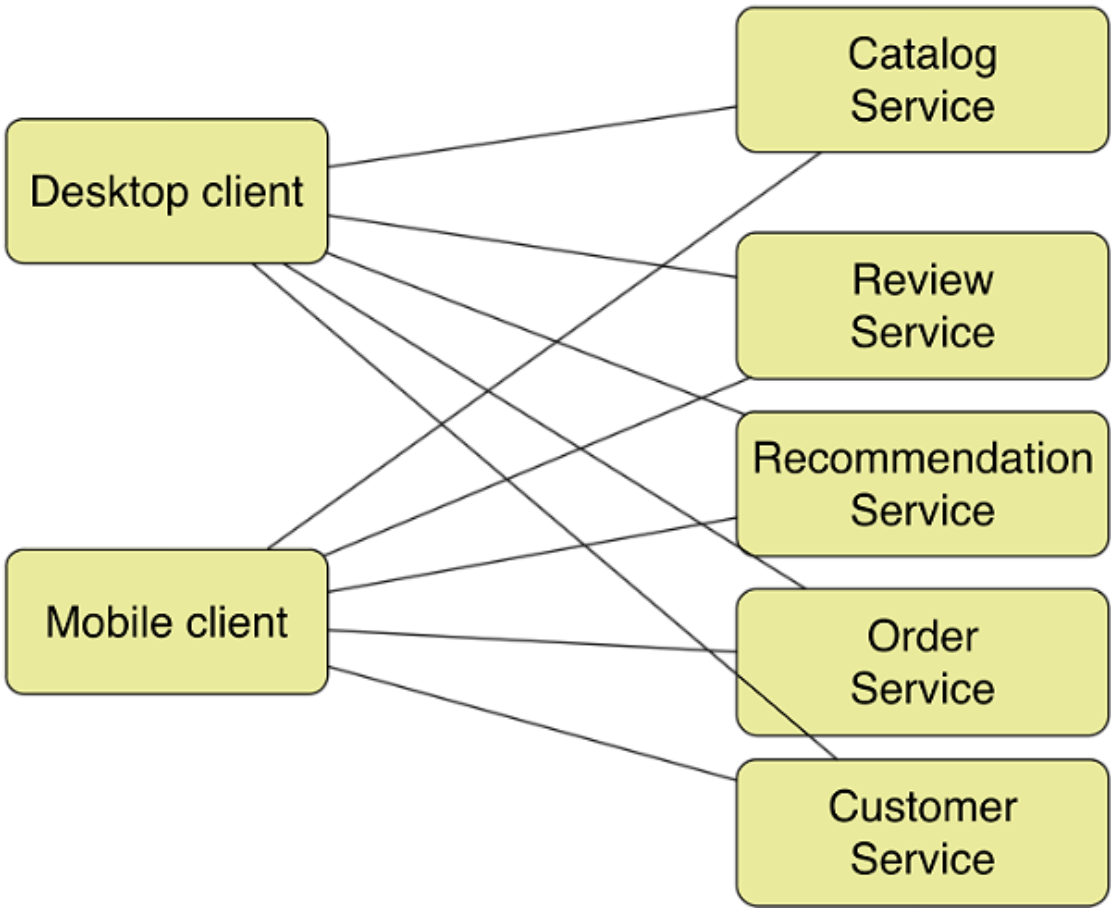

图 8: Micro Services

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(funct
      $scope.blogs = results.objects;
    });
    $scope.$broadcast('scroll.refreshComplete');
    $scope.$apply()
  };

  Blog.async('https://www.phodal.com/api/v1/app/?format=json').then(functio
    $scope.blogs = results.objects;
  });

  $scope.loadMore = function() {
    $scope.blogOffset = $scope.blogOffset + 1;
    Blog.async('https://www.phodal.com/api/v1/app/?limit=10&offset='+ $scop
      Array.prototype.push.apply($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 有就这样的跨平台能力。

在我们构建 Web 应用的时候，Java 有这样的跨平台能力。

在我们需要开发跨平台应用的时候，Cordova 有这样的跨平台能力。

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。

前端篇: 数据-表现-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
  },
  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。

如果是

表现

分离

领域

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。

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...》：

```
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 las
    var values = sink(cdr(innerRule));                        //get everything b
    return methodToUse(obj, propertyName, error, values);     //construct the va
};
```

后台与服务篇

PHP？最好的语言

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。

微服务

文章的来源是 James Lewis 与 Martin Fowler 写的Microservices。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.

微服务思考

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。

微服务与持续集成

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!

微服务与测试　相比较的来说，这也是另外的一个挑战。测试对于项目开发来说是不可缺少的，而当我们的服务一个个隔离的时候，我们的测试不得不去 mock 一个又一个的服务。在有些时候修复这些测试可能比添加这个功能花费的时间还多。

不过他更适合那些喜欢不同技术栈的程序员。

参考

Microservices - Not A Free Lunch!

Microservices

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。

基础网站重构

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

高级网站重构

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
- 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——为了更好的代码。

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

重构之提炼函数

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);
```

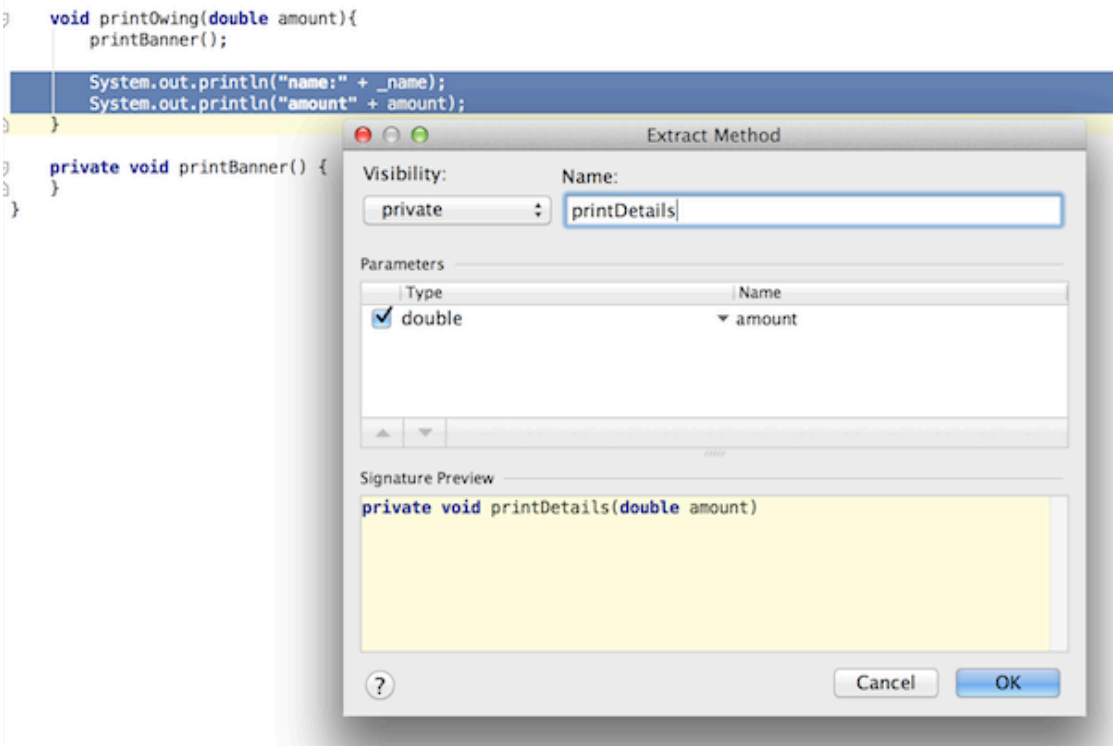

图 9: Extrct Method

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

```
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。

重构之内联函数

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　以之前的提炼函数为例

```
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

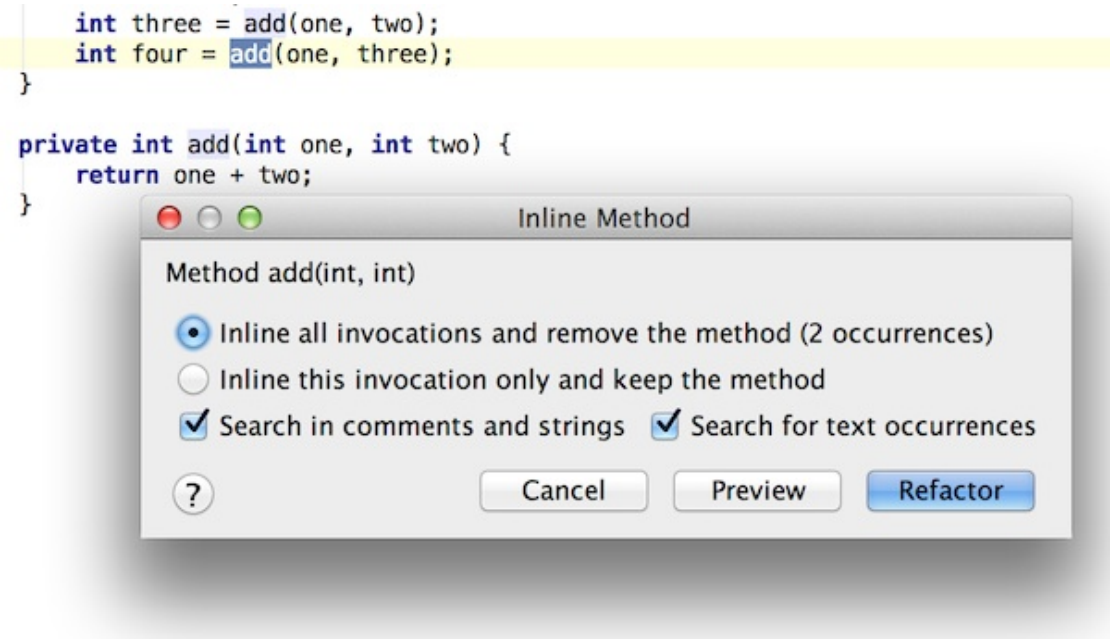

图 10: Inline Method

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);
```

对于 python 之类的语言也是如此

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

重构之以查询取代临时变量

继续看看有木有什么木有试过的功能，从之前的内联函数、提炼函数再到现在的Replace Temp With Query(以查询取代临时变量)

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;

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

便会返回

```
import java.lang.System;

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

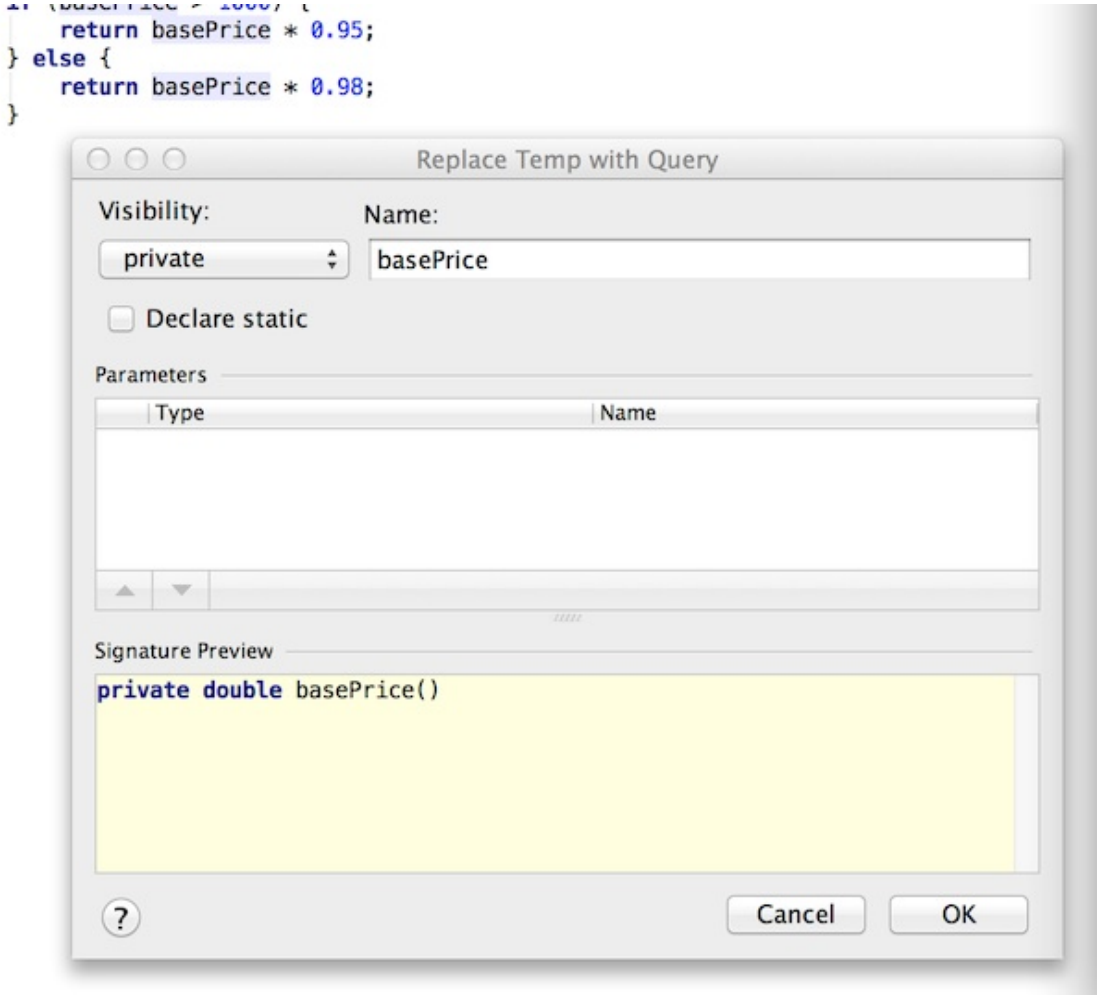

图 11: Replace Temp With Query

```
        }
    }

    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

```
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);
    }

}
```

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	Sm
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

Rating	Name	Complexity	Duplication	Churn	C/M	Coverage	Sm
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:

图 12: 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 " + URLH
    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 " + URI
    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 + " (" + 
    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 " + URLH
    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 " + UR
    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 + " (" + 
    SQLiteHelper.prototype.basic(sql_command, callback);
};

SQLiteHelper.prototype.deleteData = function (url, callback) {
    'use strict';
    var sql_command = "DELETE FROM  " + config.table_name + "  where " + URLH
    SQLiteHelper.prototype.basic(sql_command, callback);
};

SQLiteHelper.prototype.getData = function (url, callback) {
    'use strict';
    var sql_command = "SELECT * FROM  " + config.table_name + "  where " + URI
    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 ~~

一次测试驱动开发

故事

之前正在重写一个物联网的服务端，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?

架构篇一: 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:

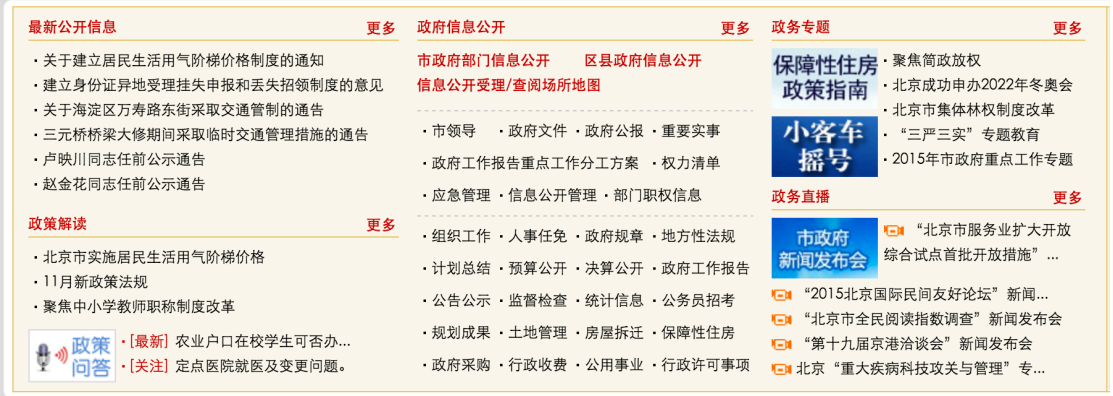

图 13: 不同分类的内容

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

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

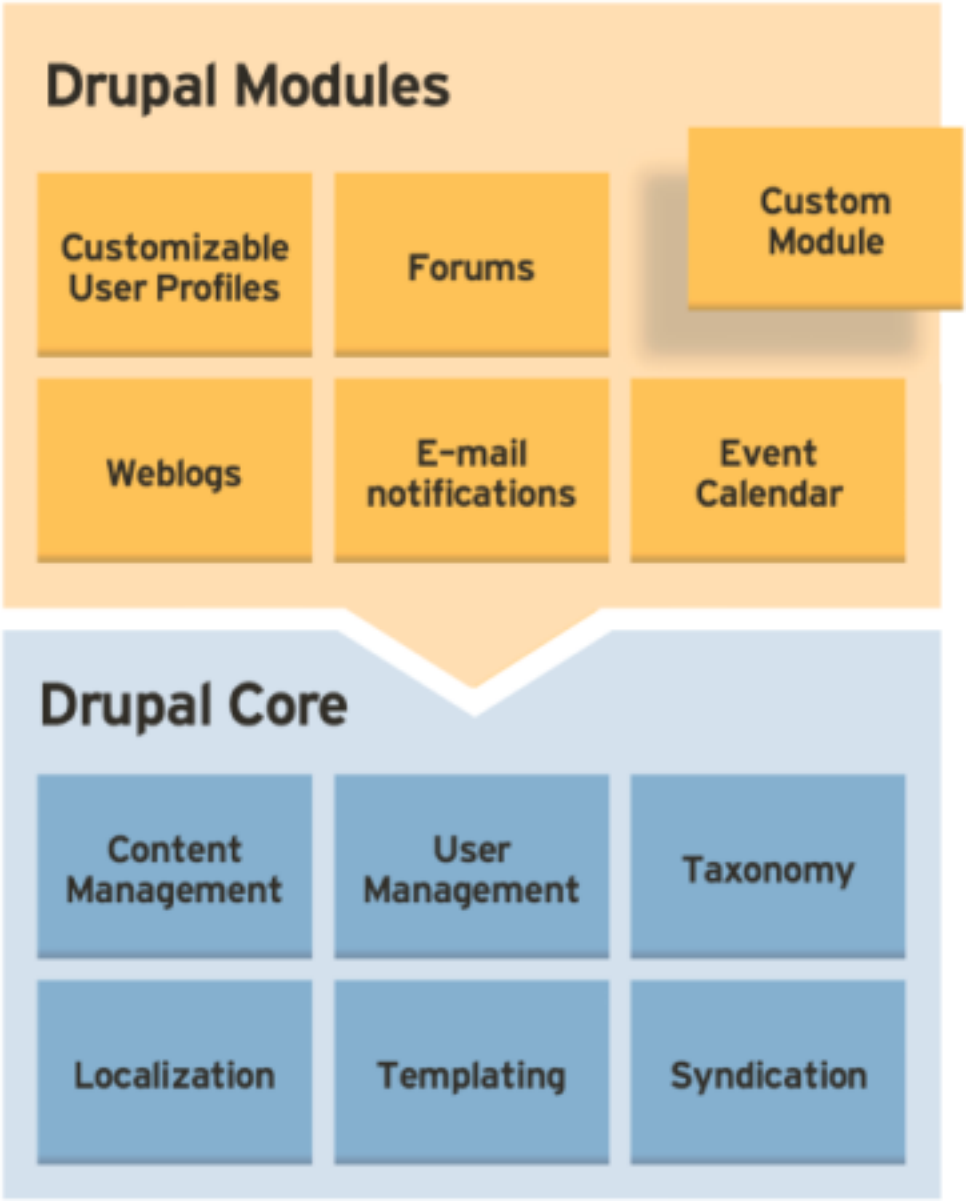

图 14: Drupal 框架

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
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。

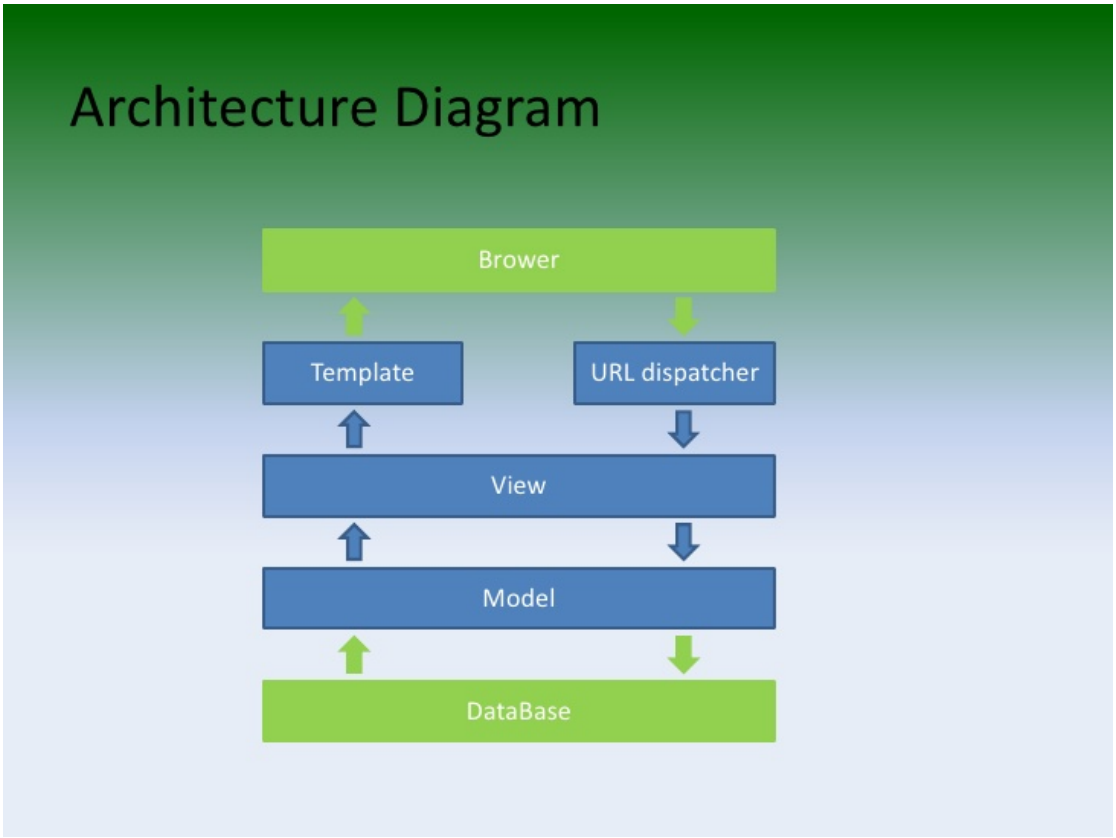

图 15: Django Architecture

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。

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。

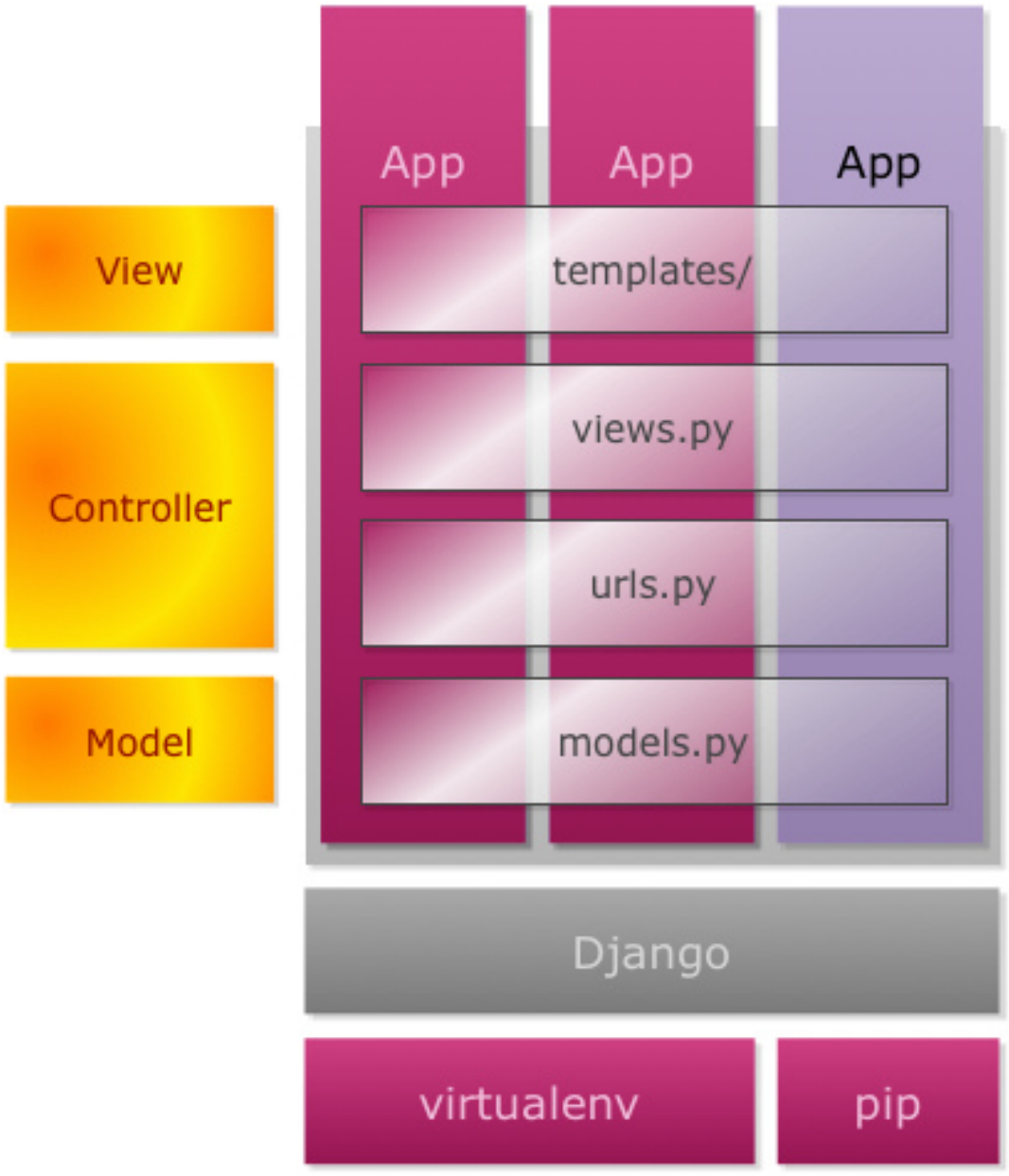

图 16: Django App 架构

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：

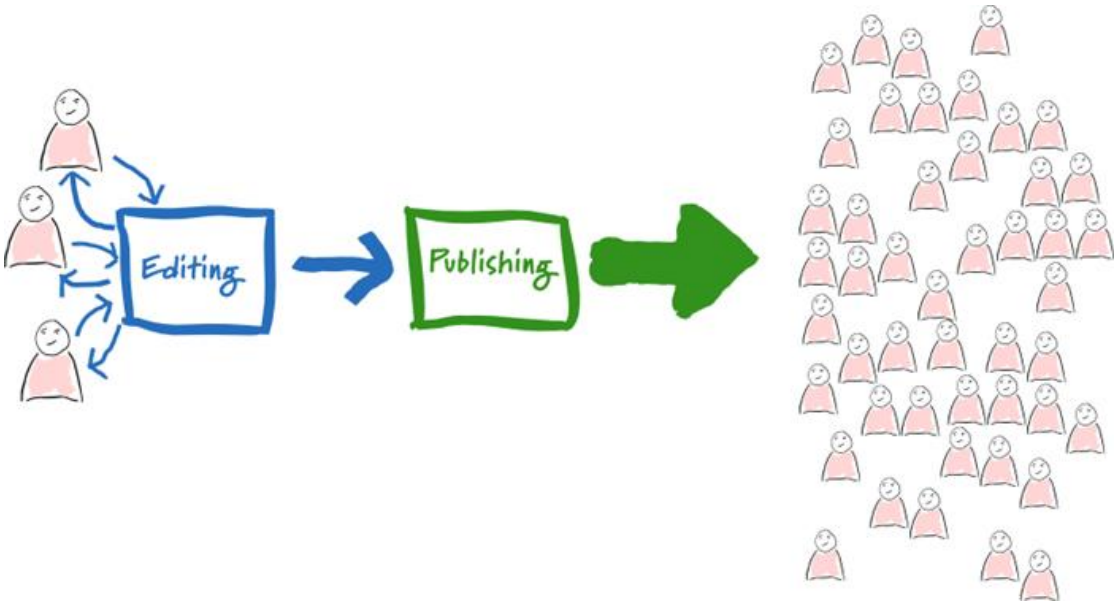

图 17: 发布-编辑

或者你认识出了上图是源自 Martin Folwer 的编辑-发布分离

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：

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

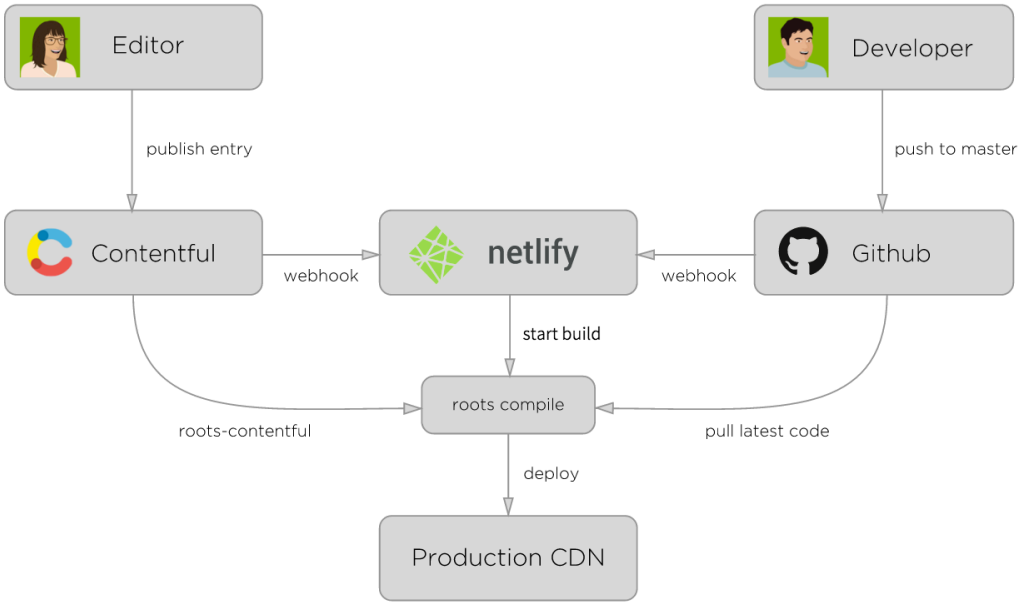

图 18: Editor Develoepr

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》，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的框架用于管理内容，并存储为 JSON。这意味着什么?

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。

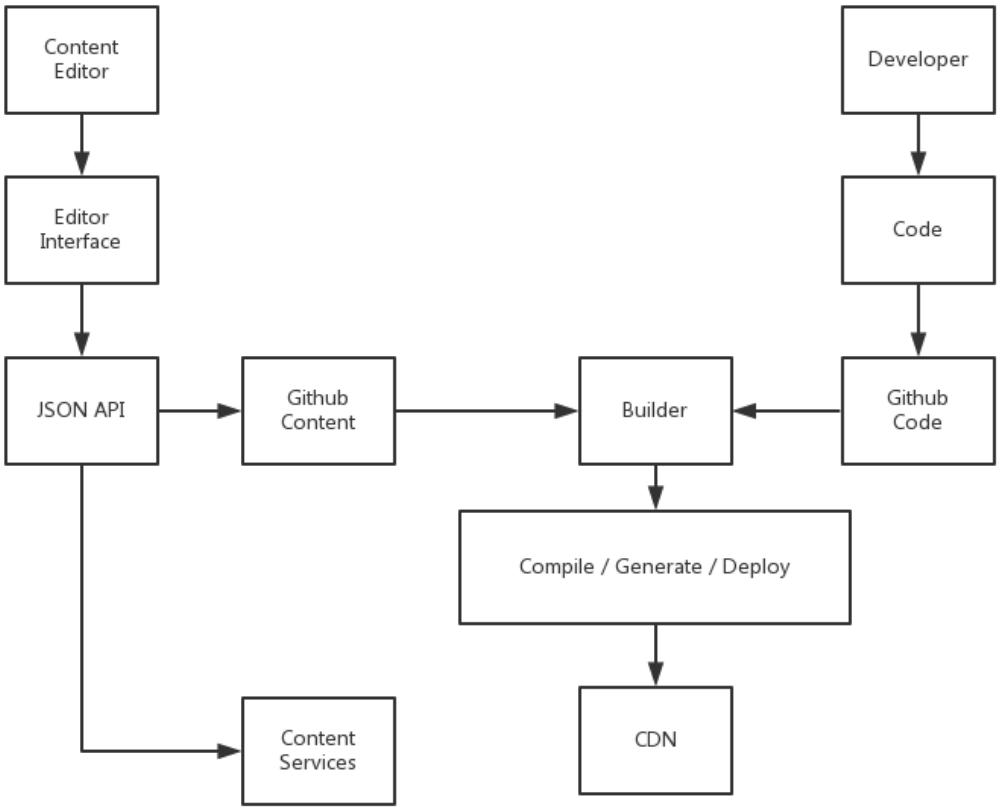

图 19: 基于 Github 的编辑-发布-开发分离

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。

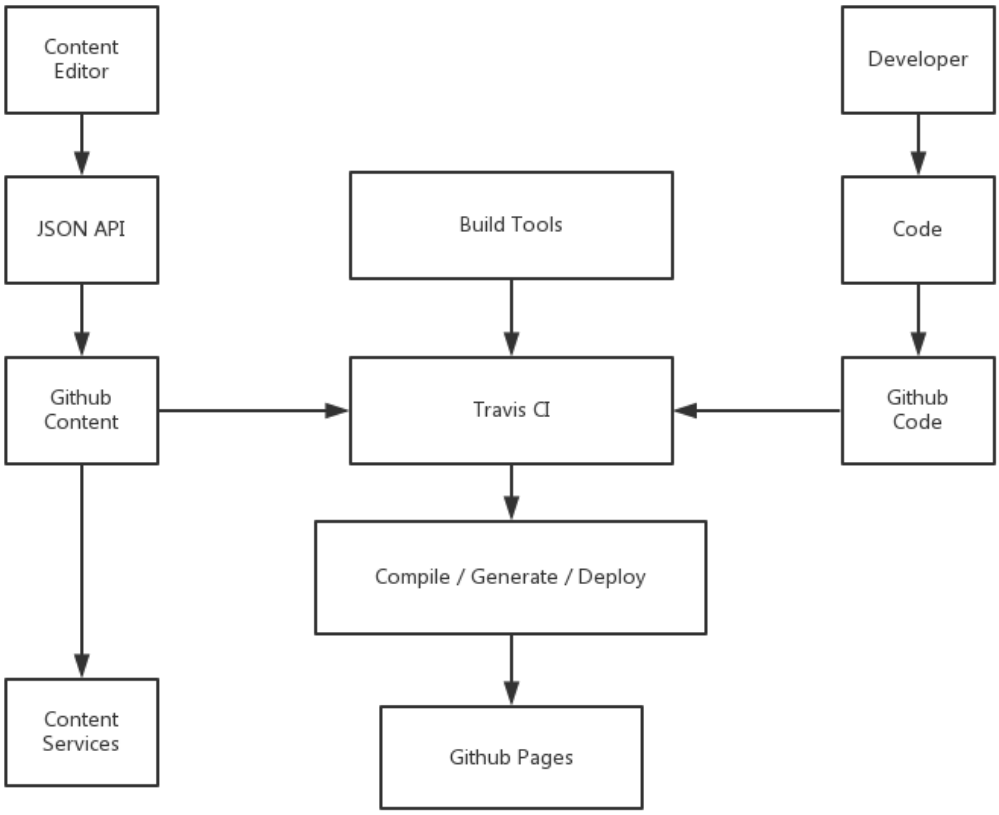

图 20: 基于 Travis CI 的编辑-发布-开发分离

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

2. EditingPublishingSeparation
3. An Incremental Approach to Content Management Using Git 1
4. Part 2: Implementing Content Management and Publication Using Git

架构篇二：构建基于 **Git** 为数据中心的 **CMS**

简介

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" &
```

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
        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》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 i
  "blogpost": "# Blog post \n  > This is an example blog post \n Lorem ipsu
}
```

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:
      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。

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：

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

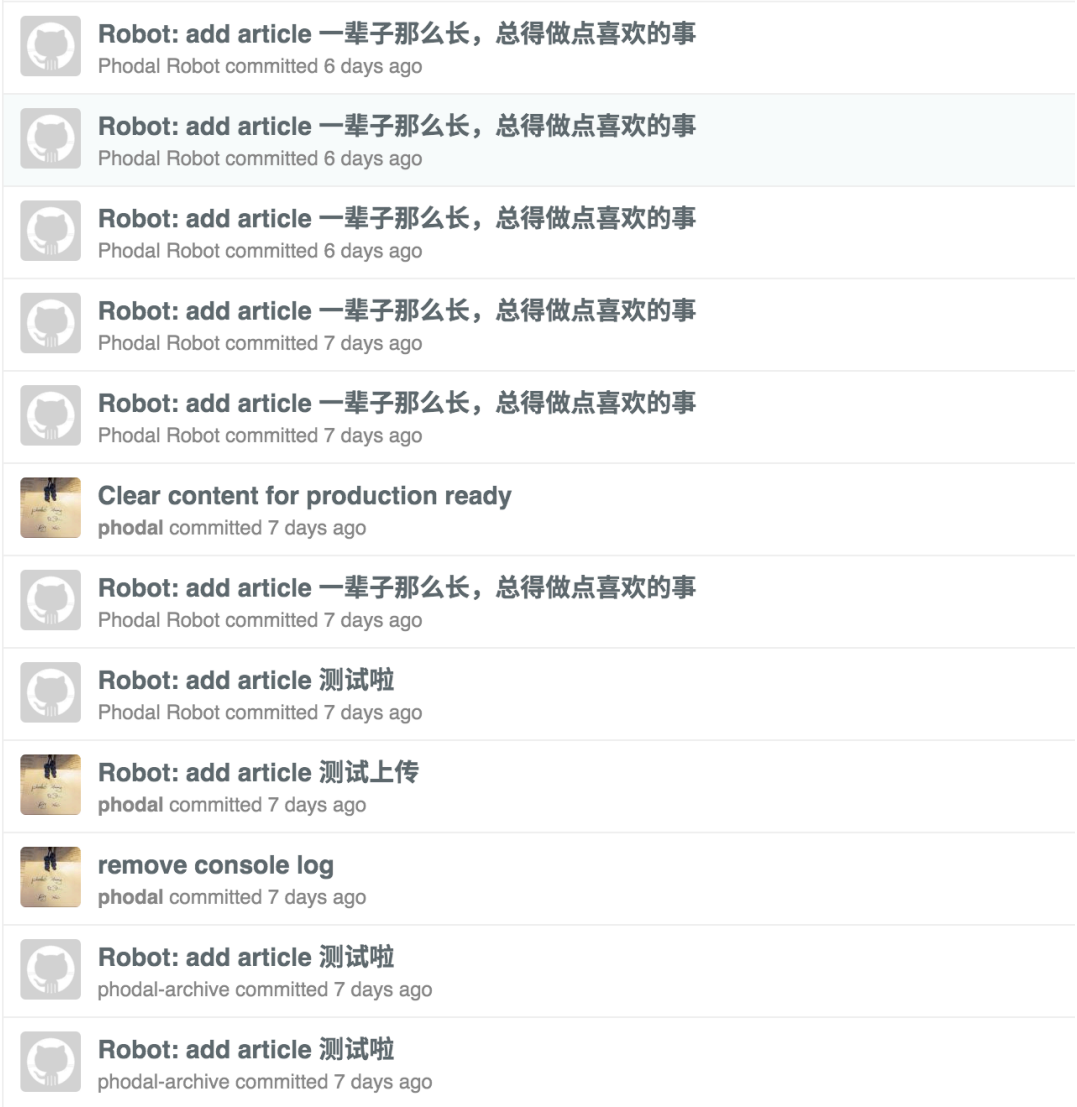

图 21: Robot 提交代码

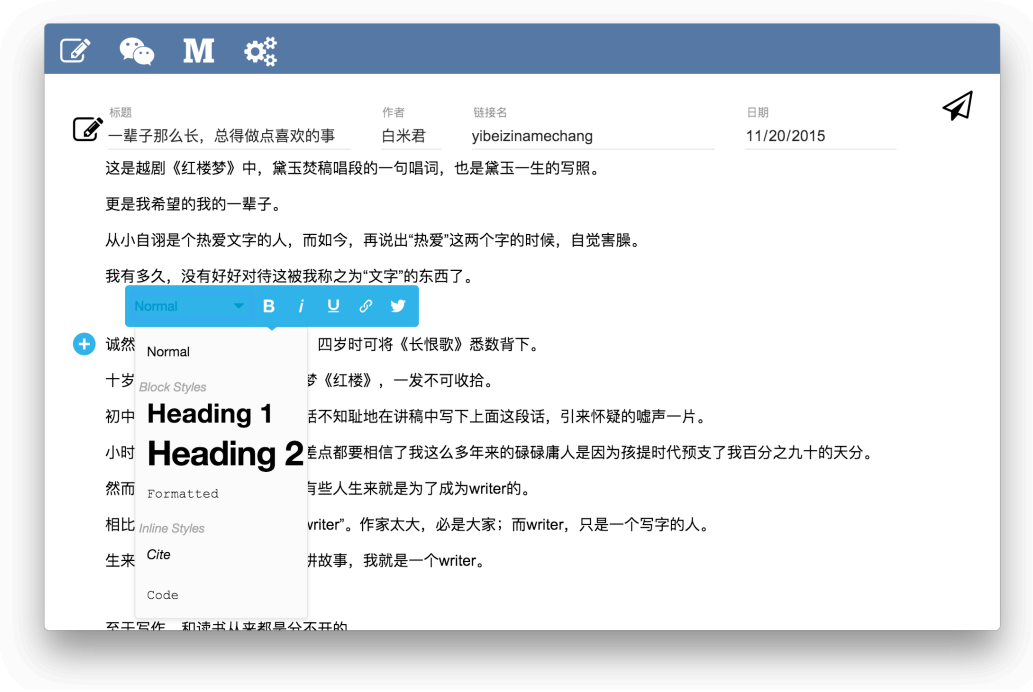

图 22: 编辑器

```
  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:
  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
        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：

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;
```

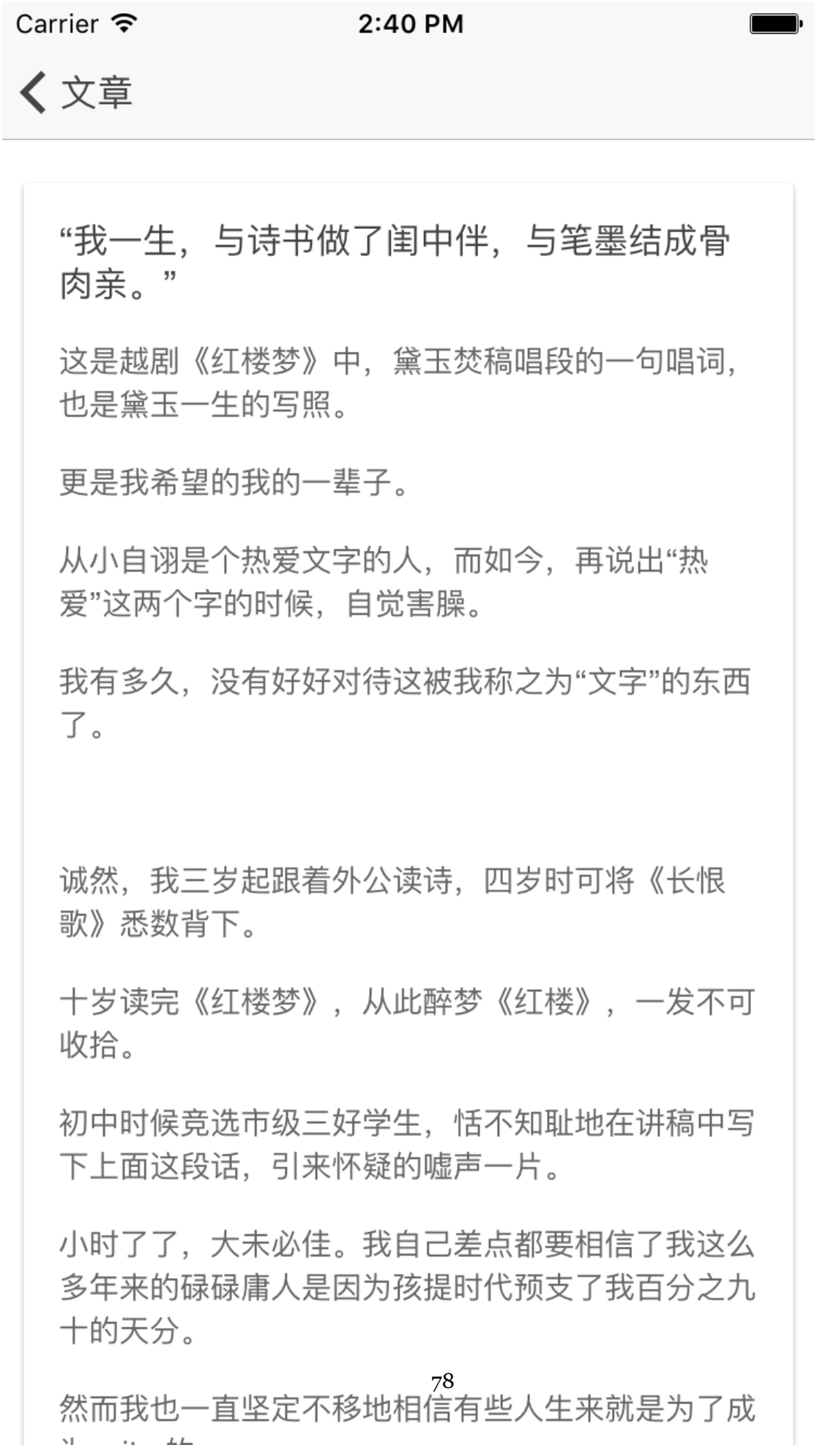
Carrier
2:40 PM
文章
“我一生，与诗书做了闺中伴，与笔墨结成骨肉亲。”
这是越剧《红楼梦》中，黛玉焚稿唱段的一句唱词，也是黛玉一生的写照。
更是我希望的我的一辈子。
从小自诩是个热爱文字的人，而如今，再说出“热爱”这两个字的时候，自觉害臊。
我有多久，没有好好对待这被我称之为“文字”的东西了。
诚然，我三岁起跟着外公读诗，四岁时可将《长恨歌》悉数背下。
十岁读完《红楼梦》，从此醉梦《红楼》，一发不可收拾。
初中时候竞选市级三好学生，恬不知耻地在讲稿中写下上面这段话，引来怀疑的嘘声一片。
小时了了，大未必佳。我自己差点都要相信了我这么多年来的碌碌庸人是因为孩提时代预支了我百分之九十的天分。
然而我也一直坚定不移地相信有些人生来就是为了成

图 23: 移动应用

```
    $scope.blogOffset = 0;
    $scope.doRefresh = function () {
      Blog.async('http://deploy.baimizhou.net/api/blog/articles.json').then(fu
        $scope.articles = results;
      });
      $scope.$broadcast('scroll.refreshComplete');
      $scope.$apply()
    };
    Blog.async('http://deploy.baimizhou.net/api/blog/articles.json').then(func
      $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="
  <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,
    $scope.article = {};
    Blog.async('http://deploy.baimizhou.net/api/' + $stateParams.slug + '.jso
      $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

全栈篇：架构设计

博客

我尚不属于那些技术特别好的人——我只是广度特别广，从拿电烙铁到所谓的大数据。不过相比于所谓的大数据，我想我更擅长于焊电路板，笑 ~~。由于并非毕业于计算机专业，毕业前的实习过程中，我发现在某些特殊领域的技术比不上科班毕业的人，这意味着需要更多的学习。但是后来受益于工作近两年来从没有加班过，朝九晚六的生活带来了大量的学习时间。在这个漫长的追赶过程中，我发现开发博客相关的应用带来了很大的进步。

技术组成

So，在这个博客里会有三个用户来源，Web > 公众号 > App。

在网页上，每天大概会 400 个 PV，其中大部分是来自 Google、百度，接着就是偶尔推送的公众号，最后就是只有我一个人用的 APP。。。

Web 架构

服务器：

1. Nginx(含 Nginx HTTP 2.0、PageSpeed 插件)
2. Gunicorn(2 Workers)
3. New Relic(性能监测)

DevOps:

1. Farbic（自动部署）

Web 应用后台:

1. Mezzaine（基于 Django 的 CMS）
2. REST Framework (API)
3. REST Framework JWT (JSON Web Token)
4. Wechat Python SDK
5. Mezzanine Pagedown （Markdown

Web 应用前台:

1. Material Design Lite (用户)
2. BootStrap (后台)
3. jQuery + jQuery.autocomplete + jquery.githubRepoWidget
4. HighLight.js
5. Angluar.js
6. Backbone (已不维护)

移动端:

1. Ionic
2. Angular + ngCordova
3. Cordova
4. highlightjs
5. showdown.js(Markdown Render)
6. Angular Messages + Angular-elastic

微信端:

1. Wechat-Python-SDK

That’s All...

前后台分离

API

在构建 SPA 的时候，做了一些 API，然后就有了一个 Auto Sugget 的功能：

图 24: Auto Suggest

或者说，它是一个 Auto Complete，可以直接借助于 jQuery AutoComplete 插件。

或许你已经猜到了，既然我们已经有博客详情页和列表页的 API，并且我们也已经有了 Auto Suggestion API。那么，我们就可以有一个 APP 了。

APP

偶然间发现了 Ionic 框架，它等于 = Angluar + Cordova。于是，在测试 Google Indexing 的时候，花了一个晚上做了博客的 APP。

我们可以在上面做搜索，搜索的时候也会有 Auto Suggestion。上面的注销意味着它有登录功能，而 Hybird App 的登录通常可以借用于 JSON Web Token。即在第一次登录的时候生成一个 Token，之后的请求，如发博客、创建事件，都可以用这个 Token 来进行，直到 Token 过期。如果你是第一次在手机上访问，也许你会遇到这个没有节操的广告：

然并卵，作为我的第七个 Hybird 应用，它只发布在 Google Play 上——因为不需要审核。

随后，我意识到了我需要将我的博客推送给读者，但是需要一个渠道。


21:16
0.03
80
PHODAL
注销
撰写
事件
搜索
博客
Github 36
After 500:
After 500:
ReThough
ReThough
ReThough
Ionic Auth
Ionic/Corc
Ionic ngM
网站更新
Django 1.8

图 25: Blog App

图 26: Install Phodal Blog App

微信公众平台

借助于 Wechat-Python-SDK，花了一个下午做了一个基础的公众平台。除了可以查询最新的博客和搜索，它的主要作用就是让我发我的博客了。

对了，如果你用 Python 写代码，可以试试 PyCharm。除了 WebStorm 以外，我最喜欢的 IDE。因为 WebStorm 一直在与时俱进。

中间件篇

JMS

MQ

易读

简介

无关的编程经验

> 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

模式篇：设计与架构

设计模式算是在 OO 中比较有趣的东西，特别是对于如我之类的用得不是很多的，虽然有时候也会用上，但是并不知道用的是怎样的模式。之前了解了几天的设计模式，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.tempera
            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:#{t
  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。

Node.js 简单工厂模式

> 从设计模式的类型上来说，简单工厂模式是属于创建型模式，又叫做静态工厂方法（Static Factory Method）模式，但不属于 23 种 GOF 设计模式之一。简单工厂模式是由一个工厂对象决定创建出哪一种产品类的实例。简单工厂模式是工厂模式家族中最简单实用的模式，可以理解为是不同工厂模式的一个特殊实现，学习了此模式可以为后面的很多中模式打下基础。

当我发现我在代码中重复写了很多个 if 来判断选择那个数据库的时候。于是，我就想着似乎这就可以用这个简单工厂模式来实现 SQLite3 与 MongoDB 的选择。

MongoDB Helper 与 SQLite Helper 类重复

对于我们的类来说是下面这样子的:

```
function MongoDBHelper() {
    'use strict';
    return;
}

MongoDBHelper.deleteData = function (url, callback) {
    'use strict';
    ...
};

MongoDBHelper.getData = function (url, callback) {
    'use strict';
    ...
};

MongoDBHelper.postData = function (block, callback) {
    'use strict';
    ...
```

```
};

MongoDBHelper.init = function () {
    'use strict';
    ...
};

module.exports = MongoDBHelper;
```

然而，我们可以发现的是，对于我们的 SQLiteHelper 来说也是类似的

```
SQLiteHelper.init = function () {
    'use strict';
    ...
};

SQLiteHelper.postData = function (block, callback) {
    'use strict';
    ...
};

SQLiteHelper.deleteData = function (url, callback) {
    'use strict';
    ...
};

SQLiteHelper.getData = function (url, db_callback) {
    'use strict';
    ...
};

module.exports = SQLiteHelper;
```

想来想去觉得写一个父类似乎是没有多大意义的，于是用了简单工厂模式来解决这个问题。

总之，就是我们可以用简单工厂模式来做一个 DB Factory，于是便有了

```
var MongoDBHelper    = require("./mongodb_helper");
var SQLiteHelper     = require("./sqlite_helper");
var config           = require('../../iot').config;

function DB_Factory() {
    'use strict';
    return;
}

DB_Factory.prototype.DBClass = SQLiteHelper;

DB_Factory.prototype.selectDB = function () {
    'use strict';
    if (config.db === 'sqlite3') {
        this.DBClass = SQLiteHelper;
    } else if (config.db === "mongodb") {
        this.DBClass = MongoDBHelper;
    }
    return this.DBClass;
};

module.exports = DB_Factory;
```

这样我们在使用的时候，便可以:

```
var DB_Factory       = require("./lib/database/db_factory");

var db_factory = new DB_Factory();
var database = db_factory.selectDB();
database.init();
```

由于是直接由配置中读取进去的，这里的 selectDB 就不需要参数。

Java Template Method(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。

Hadoop Pipe and Filters 模式

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，

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> eleme
var t = document.createTextNode("CLICK ME");       // Create a text node
btn.appendChild(t);                                // Append the text to <but
document.body.appendChild(btn);                    // Append <button> to <bod
```

而用 jQuery 写的话，便是这样子

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

图 27: pipe and filter

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
(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。

其他

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》

构建篇：**Build**

构建是一个很大的话题，特别是对于传统软件来说，对于 Web 应用也是相当重要的。

在构建上，Ruby 比 Python 会强大些。Ruby 用的是 Rake，Python 兴许是 scons，如果是用于 python 的话可以用 shovel，这个 Python 就没有和一个好的标准，

Rakefile 算是 Ruby 的一个标准。

Rake 简介

> Make 是一个 UNIX® 的本机实用程序，是为管理软件编译过程而设计的。它十分通用，足以用于许多其他环境中，即使它已用于将文档编译成书，维

护 Web 站点以及裁减发行版。但是，make 也有自身的约束。它具有自己的语法，这取决于制表符的（tabbed）和非制表符的（nontabbed）空白空间。许多其他工具已经进行了扩展，可以弥补 make 的一些不足，如 Aegis 和 Ant，但这两者也都具有自己的问题。

Make 以及类似的工具都有改进的余地，但是它们都不可能让 Ruby 黑客十分开心。您从这里要去哪里？幸好，可以使用一些 Ruby 选项。Rant 是一个由 Stefan Lang 编写的工具（请参阅参考资料）。Rant 仍处于开发周期的初级阶段，因此它可能还没有成熟到足以适用于每个人。Jim Weirich 编写的 Rake 是一个在 Ruby 社区中广泛使用的成熟系统。

Rake 是用 Ruby 编写的，并使用 Ruby 作为它的语法，因此学习曲线很短。Rake 使用 Ruby 的元编程功能来扩展语言，使之更利落地适应自动化任务。Rake 附带的 rdoc 中列出了一些优点（请注意，前两个是诸如 make 的其他任务自动化工具所共有的）：

- 用户可以用先决条件指定任务。
- Rake 支持规则模式来合并隐式任务。
- Rake 是轻量级的。它可以用其他项目发布为单个文件。依靠 Rake 的项目不需要在目标系统上安装 Rake。

简单的 **Rakefile**

```
task :default do
  puts "Simple Rakefile Example"
end
```

运行结果

```
Simple Rakefile Example
[Finished in 0.2s]
```

Shovel

官方是这么介绍的

Shovel is like Rake for python. Turn python functions into tasks simply, and access and invoke them from the command line. 'Nuff said. New Shovel also

now has support for invoking the same tasks in the browser you'd normally run from the command line, without any modification to your shovel scripts.

那么就

```
git clone https://github.com/seomoz/shovel.git
cd shovel
python setup.py install
```

与用官方的示例，有一个 foo.py

```
from shovel import task

@task
def howdy(times=1):
    '''Just prints "Howdy" as many times as requests.

    Examples:
        shovel foo.howdy 10
        http://localhost:3000/foo.howdy?15'''
    print('\n'.join(['Howdy'] * int(times)))
```

shovel 一下 shovel foo.howdy 10

构建 **C** 语言的 **Hello,World: Makefile**

C 代码

```
#include

int main(){
    printf("Hello,world\n");
    return 0;
}
```

一个简单的 makefile 示例

```
hello:c
    gcc hello.c -o hello
clean:
    rm hello
```

执行:

```
make
```

就会生成 hello 的可执行文件，再执行

```
make clean
```

清理。

Rakefile

```
task :default =&gt; :make

file 'hello.o' =&gt; 'hello.c' do
    `gcc -c hello.c`
end

task :make =&gt; 'hello.o' do
    `gcc hello.o -o hello`
end

task :clean do
    `rm -f *.o hello`
end
```

再 Rake 一下，似乎 Ruby 中的 Rake 用来作构建工具很强大，当然还有其他语言的也可以，旨在可以替代 Makefile

Scons

新建一个 SConstruct

```
Program('hello.c')
```

Program(‘hello.c’)

```
scons
```

，过程如下

```
phodal@linux-dlkp:~/helloworld&gt; scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
gcc -o hello.o -c hello.c
gcc -o hello hello.o
scons: done building targets.
```

总结

```
Rakefile
```

个人篇：从小工到能手

工作的日子里，每天都会八点多到公司，边点东西边看看 Google Analytics，看看昨天博客有多少访问量，吃完了就写写代码刷刷 Github。到了九点多，人差不多来齐了——我们不打卡，就开始了上午的工作。中午的时候会趁着午休的小间隙翻译点书，或者写点代码，写会文章。晚上吃完饭，走到家里休息会儿就八点了。看看书，写写代码，一天就过去了。

生活似乎变成了流水帐，不会发生什么特别大的变化，没有特别大的故事。日复一日的单调而又不无聊，周末也是码码字、写写代码、看看书，玩局《文明》、看部电影或者陪女朋友出去吃好吃的就过去了。

上即是下，输出即是输入

在你写过了很多的代码之后，你也许也发现了一些神奇的事情——即使你写了再多代码，你的能力并没有多少提升。人们通常称之称为瓶颈。这不禁让人想起经济危机的时候，有的国家发起了战争，有的国家开始变革，有的则无为而治，每况愈下。有时候我们发现不了我们有更好的选择。

编程同写作，写代码只是在码字

看到一篇文章《We Are Typists First, Programmers Second》，日期是 2008 年，好早的一篇文章。当现在打算以一个新人的身份重新进入编程领域，或许是受《重来》的影响，或者只是在重新思考——我们究竟应该如何去编程。

写代码只相当于写字

? 也许这是一篇软文

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。

编程同写作

> ? 编程同写作，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。

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。

写作

写作驱动学习

在软件开发领域有这样的几个名词

- TDD。测试驱动开发（Test-driven development）是极限编程中倡导的程序开发方法，以其倡导先写测试程序，然后编码实现其功能得名
- BDD。行为驱动开发（Behavior-Driven Development）是一种敏捷软件开发的技术，它鼓励软件项目中的开发者、QA 和非技术人员或商业参与者之间的协作。
- DDD。领域驱动设计 (Domain-Driven Design) 的一个核心的原则是使用一种基于模型的语言。

而最近我处在于一个尴尬的位置：写作驱动学习 (`Write-Driven Learing`)。开始之前我找到了一个相似的概念，数据驱动学习 (Data-driven Learning)。

当我来到了一家卓越的软件公司时，开始有了点所谓的骗子综合症 (`imposter syndrome`)（即常常会发现所有一起共事的程序员都比自己聪明、比自己有天份、比自己有才能），幸运的是并没有持续太多的时间。只是已经有了所谓的 `21` 天效应 (在行为心理学中，人们把一个人的新习惯或理念的形成并得以巩固至少需要 **21** 天的现象，称之为 **21** 天效应。这是说，一个人的动作、或想法，如果重复 **21** 天就会变成一个习惯性的动作或想法。)，在刚开始的时候里不断地学习，以减少这种所谓的综合症。然而，在这时已经养成了写作的习惯。

在过去的六个月里：

时间	写作数量
2013.12	**27**
2014.01	**24**
2014.02	**12**
2014.03	**29**
2014.04	**49**
2014.05	**25**

在我在学习更多东西的时候，我玩了更多的东西。而在这时候因为网站流量的关系，我开始写作记录更多的东西。而在那之前的半年里，也就是 **2013.06~2013.11** 的文章数加起来才 **21.**

于是在其他程序员开始写程序的时候我开始写作，我得好好想想今天要什么，正如那些测试人员要开始写测试那样。于是，就有了写作驱动学习 (Write-Driven Learning)。

写作驱动学习的优点

- 对于我们所要学习的东西，我们可以掌握得更深入，因为我们需要去给别人解释。
- 除去收获所学习的东西，我们还可以有副产品——博客。
- 我们在以另外一种方式思问题。
- 我们在不断地对自己知识总结，对自己知识的一个强化。
- 当然了这是我们的笔记。

写作驱动学习的缺点

- 可能会花费更多的时间，在诸如写作这点事情上。
- 有时学习的结果是为了写作。
- 付出不一定有相应的回报。

做为一个独立博客的作者来说我开始在不断地坚持写作，而唯一的动力源是来自于因为有更多的访客。要坚持写博客可能不是一件容易的事，但是现在似乎做到了，转变成一种写作驱动学习。

写博客的动力

对于我来说写博客的动力应该是下面这些:

- 对于 SEO 的学习。
- 成为 Full Stack 的必经之路。
- 更好地去理解 WEB 经营。
- 提高自己的综合能力
- 分享知识与经验。
- 可以认识更多的人。
- 可以被发掘。

阅读

看过了很多的技术书籍，到了今天，似乎对技术书籍失去了盼头。过去，因为是在学校的关系，上课只好带着书过去，现在工作了看书的时间越来越少，但是看的书似乎没有比原来少。只是看书的内容变了，以前看的是技术书籍，现在懂的技术多了，看的类型变成了理论类型的。而我本身是更喜欢实践的类型，于是到了今天技术似乎没有想象中的重要。

业务与技术

在学校的时候，由于接的一些单子数量比较少，三天打鱼两天晒网，可以有更多的时间放在技术的学习上面。或者说，当时的学习是目的性不强的学习。受一些客户的影响，当时我们所要做的一些技术方面的东西，也是被驱动着去学的。

工作的时候，主要的东西也是受业务驱动。换句话说，不论在哪我们都是要以营利为目的去写代码。只有在 Github 上写代码可以自由自在，换句话来说，这也是写博客和写书的区别。

- 作为一本书的作者，我们需要将我们想讲述的东西变得简单一点。有时，我们不得不去重复一些内容，以便让我们的读者读懂我们的思想。这就是我们的主要业务——写作，最好的写作是可以按自己的想法而来，而不是读者想要什么我们给他们什么。然而，我们也需要去考虑读者的感受。毕竟，这时可以给我们带来收入。
- 作为一个博主，我总是自己想写些什么就写什么。我需要写点什么就写点什么，这就是我们的自主。我们可以不需要去重复一些内容。而且相对于写书来说，我们可以不断发布我们地新版本的博客。博客可以是我们的笔记，于是我们想要什么就给自己什么。我们不需要考虑太多读者的感觉。毕竟，这算是一种额外的收入。

于是，我们回到技术与业务上。这也是我们在 Github 上写代码和项目上写代码的区别。

- 工作上写的代码，绝大多数是以业务为驱动的。有时，我们甚至会抗拒某些功能的代码，至少从心里上不会认可这样的行为。然而，在业务上却是受欢迎——比如广告。我们写的代码多数时候是受约束的，这样也就意味着我们可以写出更好地代码。只是当我们有太多优先级比较高的功能时，我们写的代码就会越来越糟。我们没有时间去重构，没有时候去写出更好地代码。正如：如果我有更多的时间，我就能给你写更短的信了。
- Github上的代码。总是可以随心所欲地写，我想有这个框架，用这个数据库。我们可以在自己的项目上选择需要的功能，就像 Linus 一样，可以决定 Linux 内核需要怎样的功能。这时，我们没有业务驱动，我们可以写作更好地代码。我们总会试着去重构我们的代码，因为我们有足够的时间。

只是我们靠工作来赚钱，靠 Github 来保持热情。虽然，多数时候对工作上的代码很有热情，比如某些有挑战性功能。而我们靠 Github 来保持更高地热情，不是么?

说说技术书籍

多数时候我们看书是靠兴趣，少数时候我们是靠技术上的需要。于是，作为一个兴趣广泛的孩子，我有一堆书，多到我不想去统计到底有多少。

有一天，我发现我的书架上摆满了太多关于技术的书籍。

于是有时，我开始向一些非技术类的书籍靠拢。毕竟，了解活着的意义，有时候比更好地活着来得有意义。

如何写技术书籍

这是一个简单而又有趣的话题，如果我们写关于某方面技术的话，我们我想我们要写关于这方面的书籍唯一缺少的就是恒心。因为我们在这上面工作了一些时间，我们有足够的能力去写一本书。当你发现一本书上面的所有知识都是你会的时候，你也有能力写这样的一本书。但是你真的会去写么，如果每天你都想着把睡懒觉的习惯戒掉的话，这可能是一件很能的事情。这需要持之以恒地力量，所以在那之前需要好好地去评估一下这可能性。

听过不少关于技术方面的讲座等等，多数都讲得比较显浅，并不是演讲者的水平问题。而是如果演讲者讲得太深的话，那么多数的人就听不懂，也就不会有兴趣。听上去，你似乎是一个 **NB** 的人物，仅些而已。于是如果我们把一本技术书籍写得复杂的话，那么受众就少了，肯看的人就更少了。很难去了解到底有多少出版社真的愿意去出这样的书，于是多数的技术书籍的主要对象都是那些初学者或者中级。

于是开始慢慢去思考，我们是要写更多的代码，还是要看更多的书。只是有时可以少看一些技术，我们真的需要去了解一些不在用的技术吗? 我们把一门语言学得再好，如果用不到的话，是不是只能用来聊天?

持续

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

持续阅读对于想提升自己的人来说是必须的。

如何成为武林高手　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。

记得 ThoughtWorks 电话面试的时候，有一项就会问Github 账号 (ps: 我的 github 账号是 gmszone。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。

持续学习可以降低危机感　记得在大学期间 (ps: 现在大四实习) 阅读了大量的书籍 (>400)，而之所以会阅读那么多的书籍很大一部分原因来自于危机感。或许对于所谓的重点院校的学生来说，不会有太多的危机感，不幸的是周围的同学也没有危机感。对于一个连二流学校出来的学生来说——学习才是最大的善，记得这句话是犹太人说的。所谓的学习不是指课堂上的学习，中国的课堂上的学习大家都懂的。因为什么也没有，什么也不会，所以就有了危机感，在某种意义上来说有点破釜沉舟的感觉。

当我把图书馆想看的计算机书差不多了解了一遍之后，开始到网上继续汲取知识。所明白的是，好的学校的老师会比一般学校的老师好很多。但是在技术方面来说，正常情况下有能力的都不会想去当老师的 (ps: 这里没有鄙视老师的意思，至少对于我们学校来说是如此)，当然不包括某些有理想的。

对于危机感来说真正会暴发的年代是大四，在工作中或许就是试用期，还有一些工作的其他方面。持续学习可以降低危机感，从心理学上来说，忙碌可以降低焦虑。

个人篇：影响力

影响力，让梦想离你更近。

试想一下，有一天你开发了一个新的语言。它比现有的某某主流软件，运行效率将提高了 50%，开发效率提高了 100%。接着，你在 github 上 release 了 0.1，但是由于出现某个开发难题，你需要别人的帮助。而这时，你找不到有效的途径去找到那些真正会用它的人。接着出现了一个新的语言可以达到一样的效果，而这个项目就死于腹中，我记得mruby刚刚只写了一个 `README.md` 的时候，就获得了上千个 star。

如何提高影响力，为自己代言

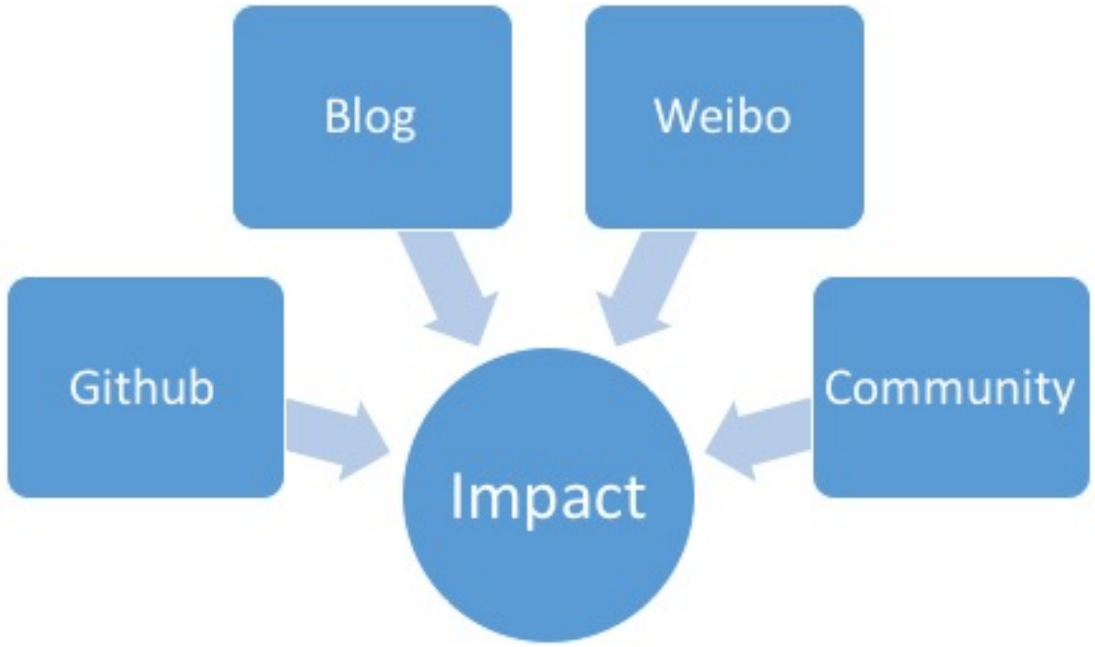

图 28: impact

每个人都可以是一个品牌，对于一个程序员来说，我们的 ID 就是我们的品牌。而构成品牌的有多个要素:

- 博客
- Github
- Weibo(or Twitter)

- StackOverflow(or SegmentFault, Zhihu)

等等。

搭建一个跨平台的平台

连接各个平台的核心是我们的 ID。

第一个平台指的是不同的网站，如我们的博客、Github、知乎等等，第二个平台指的是我们的影响力。

So，在开始的时候我们需要有一个统一的 ID，来标识我们的身份: 我是谁，你在 xx 网站上看到的那个 xx 就是我。刚开始的时候，我在 CSDN、Github 上的 ID(gmszone) 和我的博客的域名 (Phodal) 是不一样的，因为当时的域名 (gmszone.com) 握在别人的手上，于是我便想办法将两个 ID 改了过来（ps: github 提供迁移)。后来，Phodal 就成了我的发声平台:

- http://www.phodal.com/
- http://weibo.com/phodal
- http://www.zhihu.com/people/phodal
- http://github.com/phodal
- http://segmentfault.com/u/phodal
- http://www.douban.com/people/phodal/
- ...

于是，这时就可以开始使用跨平台的平台了。

构建平台

小博客也会有成长的一天。

对于像我这样一个个默默无闻地人来说，用户可能会有下面几种不同的方法来知道我:

- 用户 -> 搜索 {谷歌, 百度, 必应} -> 博客 -> {Weibo,Github}
- 用户 -> 微博 -> {Github, 博客}
- 用户 —> Github -> 博客
- 用户 -> {知乎, SegmentFault} -> {Weibo,Github, 博客}

图 29: live example

博客

刚开始在 CSDN 上写博客的时候，一开始的访问量很少，慢慢地就多了。有一天发现这样的博客不是自己想要的，于是建了自己的博客，一开始的流量是 0。像 CSDN 这样的网站推荐一些文章到首页，如果能这样便是幸运的。在经历大半年的几乎零流量之后，开始慢慢增长了。到了今天，一共有 470 篇博客 (有一些是出于测试 SEO 目的写成多篇文章)。一天的 PageView 大平均有五百左右，主要来源是搜索引擎，百度 200 左右，谷歌 50 左右，必应 10 左右。

用户故事　对于一个程序员来说，必须在某种程度上熟悉怎么搜索自己想要的内容，即关键字。如我们想要知道如何在 OpenWRT OS 上用 Python，那么我们会搜索 `OpenWRT Python`。于是，这个时候我们博客的标题带有 OpenWRT Python，那么我们可能就中奖了。

故事，告诉我们好的标题很重要。重复这个主题也很重要，会有一个更好的排名。至于，如何更好地排到第一，就是 SEO(搜索引擎优化) 的话题了。

笔记　一开始要写一个博客是比较难的，没有流量、没有评论。所以，一个好的切入点是: 笔记。最好是那种网上很少的内容的笔记，虽说很多人不愿意去做这个，但是这是一个很好的方向。

一个技术博客里面的内容应该是两种类型:

- 技术
- 理论

技术型可以带来流量，理论型的可以带来评论。理想的话，两者会相辅相成的，但是在我们刚处于学习期的时候。那么那些 Note，可以给我们带来一些流量，也带来一些信心。如果，只是想着一开始我就只写一些长篇大论的话，那么只是就是拿了 80% 的时间做了 20% 的事。

以用户搜索的过程来说，用户是有目的的进行搜索。换句话说，在我们日常工作的时候，我们只关心和我们工作相关的内容。而在受众来，正常情况下，技术型的博文、笔记可以带来流量的主要原因是: 大部分人都是初学者。

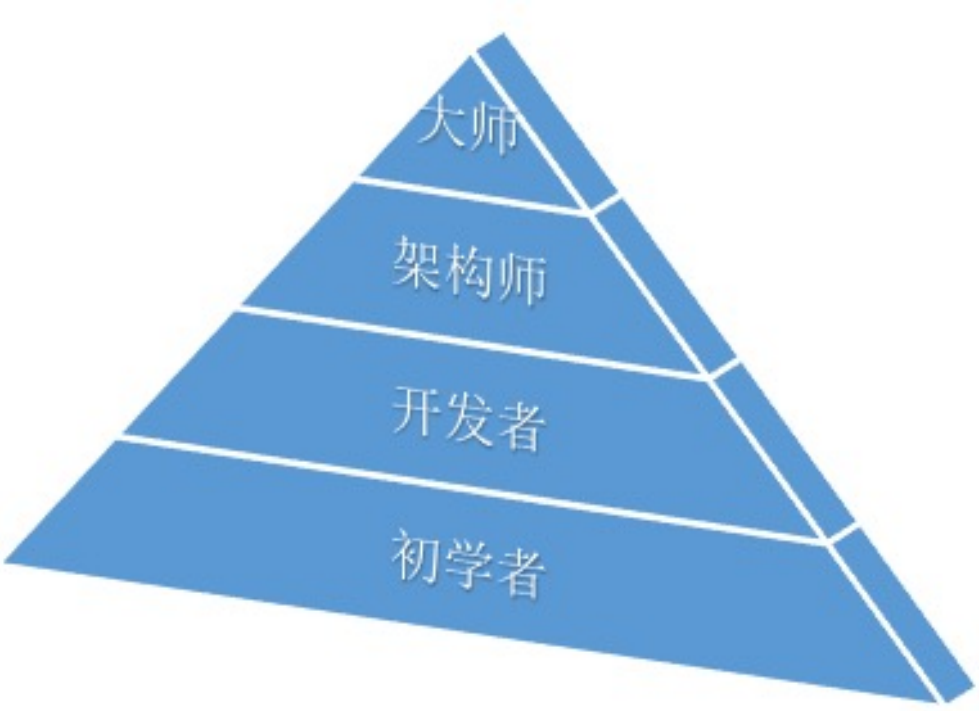

图 30: 70 percent

理论性的内容，更适合更高级别的开发者，这样的受众较少。

上头条　而在今天有其他的平台，可以借用来推销自己的:

- 开发者头条
- 极客头条
- 掘金稀土

- ...

网上的 IT 新闻、博客都是互相 Copy，对于一些软文 (如本文) 来说。这也是期触及率高的原因，通常来说这样可以带来大量的流量。记得在原文中留个原文链接，附张图片 (自己博客的图片) 来保证:Google 把原文指向你的博客，而不是免费为别人打工。

提升 除了是一个很好的展示平台，也是一个很好的测试平台。作为一个 Web Developer，测试过

- Nginx Log 分析 (~600M)
- New Relic
- SEO
- AutoComplete
- 重构网站
- ...

Github

将自己尝试的不同技术栈的内容丢到 Github 上，加上适当地文档、博客，就变成了一个很好的 Demo。然而，不止于此，越来越多地人开始在 Github 寻找人才，因为他们乐于付出，也乐于分离。曾经因为 Github 上的项目:

- 申请 WebStorm 开源 License
- Review 英文版书籍
- ...(有些不方便透露)

而在 Github 上的项目其实不仅仅只有一些库:

- 库和框架: 和 `jQuery`
- 系统: 如 `Linux、hhvm、docker`
- 配置集: 如 `dotfiles`
- 辅助工具: 如 `oh-my-zsh`
- 工具: 如 `Homewbrew` 和 `Bower`
- 资料收集: 如 `free programming books，You-Dont-Know-JS，Font-Awesome`
- 其他: 简历如 `Resume，` 博客

所以，可以尝试不同的切入点使用 Github。

在某种程度上来说，一个人在 Gihub 上的粉丝数量会帮助他的项目在初期获取更多的人气。这样就有助于其下一步开展项目，当然也会在某种程度上更好看 Blabla。

提升　之前写过一篇《如何通过 github 提升自己》中说到了一些细节，大致上还是那些东西，Github 与 Trello、Travis-CI、Coveralls、SauceLabs、Code Climate，配合得很好。已经可以接近于我们在一些项目上用到的功能，因此拿他们搭建一个好的开发环境完全不是问题。

- Travis CI: 支持 Node.js、Python、Ruby、PHP 等二十几种语言，也支持 MySQL、PostgreSQL、MongoDB、Redis 等数据库。
- Coveralls: 可以用来查看代码的测试覆盖率。
- SauceLabs: 可以用来跑 Selenium 功能测试等等
- Code Climate: 查看代码的坏味道。

在我们试着去保证测试覆盖率、代码质量等等的时候，我们就可以学到一些知识，如使用不同语言测试框架的 Mock、Stub、FakeServer 等等。

扩大影响力

上面的两部分属于打造平台部分，而如 Weibo、知乎等则属于扩大影响力。

在某种时候，博客、Github 都更像是个人的平台。如 Weibo、知乎、SegmentFault、CSDN 等等的开发者社区，也可以为自己的博客、Github 带来流量，而这一些都是互相促进的。如果我们在其中的一个网站上表现得很好的话，那么在另外一个网站上我们也很容易聚集同样的粉丝。如，我最常用的一个作法是: 将自己写得相对比较好的一些博客复制到 CSDN、SegemntFault、图灵社区等等，并适当地推到 CSDN 首页、开发者头条等等。

由于写作的开发人员是有限的，所以通常在某某头条上的推荐，会成为某博客园上的新闻，接着会有成群接队的小站开始 Copy。同时，这些文章又会推到他们的微博上，接着还可能是微信平台。毕竟，对于大部分的网络来说，大部分的流量是流向他们的网站的，所以他们不太会在乎其中的外链等等。故而，通常来说: 不是某某东西突然火了，而是突然没有别的新闻了。通常来说一个好的作法是，试着将你的博客分享到微博上，然后 @ 那些愿意帮你分享的平台。这样，你可以有更多的阅读、更多的粉丝，当然好的内容是前提。

其中还有若干其它的好处:

- 更大的曝光率，会带来更多的机会
- 更有机会参与一些未公开互联网产品的试用
- 各种精美小礼物
- 翻译、出版机会

TODO

只需要简单地几步就可以开始提高我们的影响力:

- 在不同的网站上使用一个 ID
- 创建一个博客
- 开始创作内容、提交代码、分享
- 持续 Impact

成为笔杆子

为什么要写博文

写一篇博文意味着要花一定的时间，有时候可能是一个小时，有时候可能会更多，于是人们开始去 copy。在这个 Ctrl+C 越来越盛行的年代，我们还是输了，于是乎在我们的国度里，我们的计算机书算是输给国外的精英了。我们也有优秀的程序员，有优秀的工程师，只是在其中能与大众沟通的又有多少。我们的最终用户可从来不会管你用的是什么技术，他只懂得什么是体验，什么是速度等等。至于你说的东西，他不知道，也不会在乎。

这也就是为什么大师可以成为大师的原因，而菜鸟却还是菜鸟，大师在心里写博文的时候学会了总结，比如，定义算法的集合，将各算法封装，使它们能够交换。利用 Strategy 模式，算法和利用这些算法的客户程序可以分别独立进行修改而不互相影响。这个就是你需要的方法，于是大师就和你说了，“你需要 Strategy 模式”。这就是你要的答案，GoF 分享了他的东西给了你，我们就有了一本《Head First 设计模式》或者是《设计模式解析》。

我们开始走上了成为大师的西天取经，为什么是西天呢，这个领域一直是西方比东方分享得多。《西游记》就这样成为了《西游记》，写下这个过程的到处是吴承恩，还是师徒五人？师徒五人从一个地痞无赖直至成仙成佛。（PS：一直觉得自己写的东西，比较像不是散文的散文，中心似乎一直很明确，只是看懂的仿佛不到。）简单点来说，就是他们写下了自己的那些点点滴滴，我们就知道怎么去“西天”（我的意思不是那个意思，我想你懂的。）

这个过程就是一个个为什么你会看到那么多本优秀的计算机书的原因，大师分享了他们的心得告诉我们如何去成为大师，不过我还不是。只是如果你要成为大师，就要去分享你的过程。至于为什么？简单的说几点：

- 技艺的掌握在于重复。技术和游玩的相同之处在于技术玩得越多，也就越熟悉，当你试着去写一篇博文的时候，你也回顾了过程。游玩的回味可以再次欢乐，博文的书写可以再次熟悉。
- 你的过程正是别人所需要的。不要以为你手上的那点点关于编码的小知识不是别人所需要的，有时候人们就需要像《七周七语言理解多种编程范型》这种书。
- 你的作品有可能因此重构。至于你对于重构是害怕还是享受，我就不得而知的，但是你写出来的时候，也许你会有更好的思路涌现出来。不好的一点是你还需要对这篇文章进行重构，不是么？
- 别人的评价。别人的评价有时候是打击，不过我想更多的时候是一种建议，比如 Linus 在刚写 Linux 的第一个版本的时候，他也遇到了这样的问题。至于宏内核好还是微内核好，这个问题有点类似于先有鸡还是先有蛋，不过我想后者可能科学家会给出答案。至于前者，不同的领域可能是不同的，Python 好还是 C 好？相同的领域也可能是不同的，Ruby 强大，还是 Python 强大？
- 最后一点就是，你想成为大师，不是么？如果你还甘愿……，我就不说了。

Copy 与盗版

当我开始越来越频繁写博客的时候，同学开始复制，于是有一天他的排名对于我来说，已经遥不可及了，于是远远地排在了 CSDN 的前面。一步步的前进着，开始懂得怎样去试着推销自己的博客，这时候渐渐有趣了。又去鼓励另外一个同学去写博客，就如他所说的，“就算是你，写一篇博客也要一个小时”吧，或者对于我的打字速度来说，不算什么，半个小时可以达到三千，五笔加上机械键盘好的手感。

我们总会说别人写的说怎么怎么的烂，但是如果一本书上不是 Copy 过来的，那么他就是不错的，在版权的地位比代课老师还低的天朝。我想你就可以骂这本书烂，因为他是复制的，因为到了最后你没有找到出处。换到博文来说，你搜索到的结果一个个都是一样的，你找不到原版的文章，去问作者一些问题。

一开始的时候我试着去反抗那些复制，你花一个小时写的东西，可能在发布的瞬间就被抓取过去了。有趣的是，渐渐我发现这有利于我们去传播我们的思想。换句话说，这是一个信息时代，你写的东西有可能在一瞬间到了 Obama 的眼前。至于优缺点嘛，补充一句可以借此 SEO。

天朝一直都有天朝特色，无论从哪些方面来说，计算机也是如此，中国特色的免费。

至于付费，我想这就是为什么我不会考虑去做收费软件的原因了。程序员害了程序员，自己害了自己有什么好说的。于是转战到了 openSUSE，都挺好的 fcitx 的五笔很给力，bug 也没有原来多，还有 WPS For Linux 下的此文，因为网络原因。

Copy 对于读者来说，看到的都是千篇一律的东西，只会写的人失去兴趣。盗版对于用户来说，看到的都是免费的东西，只会让开发商失去动力。用户便看到了越来越多的广告，读者便只看一个门户的新闻。

如何去写博文

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）。

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。

小提醒

- 代码，代码有时候会更清晰的表达你思路，太长的代码可能会影响阅读，通常不超过一屏就不算太长
- 图表，耐心的画个 UML 图，或者程序框图也是不错的，很清晰的表达你的思想。
- 美观，要知道 C 上是有 indent，如果我看到别人让我帮他看的代码是一坨。。。WPS 也有段落，如果你没有学好 WORD，找本《72 小时精通 Office》吧，顺便找 PPT、EXLS 也学了。
- 格式，记得好好用好手上的工具，如果你用的是 CSDN 用的 editor，试着一个个探索，CODE 应该要有 CODE 的格式，LIST 应该要有 LIST 的格式

最后，耐心 在 CSDN 上的博文的话，可以按长尾理论来分析，这里说的通常是指——你的东西是原创的，写博文有些时间。SEO 上，以谷歌为例，谷歌对其抓取是比较及时的，同时谷歌会排除掉部分专业的复制网站——就是拉到重复的搜索结果里。文章刚发表的时候的流量有可能会很低，但是有些文章时间一长就显示出来了，比如我写的东西中的《Android 上使用 GCC》算是一个很好的示例。一开始的时候我们写的东西访问量不会很大，特别是我们刚起步的时候，这时候就要一步步慢慢来。只要你写的东西是别人需要的话，那么就会一步步慢慢来。如果你写的刚好是热门的话题、技术的话，那就是好莱坞大篇《速度与激情 1》《指环王 2》《黑客帝国 3》《纳尼亚传奇 4》。。。直到《哈利波特 7 上》，《哈利波特 7 下》。于是作为迪斯尼的你，又推出了下一部分电影.....

总结

现在的我们更多的技术是直接来源于 Google、百度、CSDN 或者其他，搜索得来的，我们并不去考虑别人在其中花费的时间和经历，有时候我们要试着去想我们是不是也应该分享给别人。这算是自己开始写博客的原因，受益于开源社区，我们自然而然的也要回顾给这个社区，只有分享才会使未来更美好。

我们都希望看到有一篇博文够清楚的对我们当前所遇到的那个问题进行好好的解析，问题是也许你解决过的那个问题正是别人所需要的，但是你并没有将它分享出来，仿佛是一个循环一样

```
for(;;){

}
```

于是我们又回到了一个起点，人都是自私的。我们都希望自己能更快的学习好一项技术，一门语言，别人也需要你手上的那项技术，那个语言。都你开始意识到别人需要你手上的东西的时候，你算懂得换位思考了。

写篇分享，写篇心得就是一个好的开始，或许我们已经被高中的语文所吓怕了。但是，是时候从新开始。如果你被 C 的指针吓坏了，被 C++ 的模板吓到了，被 Javascript 的简陋吓到了，而你又需要拾起它，我想是时候重新开始了。

成长篇：实习记

写在前面的话

终于可以在和自己的电脑上写下这些字了，实习的日子也就算到一段落了。时间过得很多，也过得很短。很丑陋地用着没有 UI 的 Fcitx，不过对于我来说已经足够了，五笔是一个不错的中国产物，拼音只能算是一种泊来品。

告别了用了半年的 Macbook Pro，发现原来在 Mac OS 上只能选择某个软件，而现在却可以有不同的选择。在纠结完 openSUSE、Ubuntu Kylin 还是 Mint 之后，我选择了 Mint，毕竟我还是属于追求新软件及一些特性的 “追新族”(主要原因还是显卡不支持了)。将公司配的电脑留在公司里，配置了两个晚上——从 Fcitx 到 bash、vim、Emacs、Texlive。接着我开始计划着写上这些东西，我需要用 Markdown 去写这些东西，需要选择一个编辑器。没有了 Mou，Retext 对于 UTF8 的支持不如想像中的好，最后算是用上了 MdCharm（印象中是一个国人写的）。自己用基于 node-webkit 写的 lumia 写的编辑器似乎性能又不好，有空真应该自己重新写一个。或许当某一天自己正式成为一员的时候，应该抛弃 Mac OS。

如果有一天发现这些不是自己想要的时候又会怎样呢？创造和生活有时就是互相矛盾，一天接着一天的过去了，没有找到最初想要的东西，但是却可以慢慢地掌握想要的能力。实习或许就是这样，然而又没有一种足够的可以燃起兴趣之火。于是，我知道我需要去创造。在创造中学习，在写作中学习。

技术 **x** 思考 **=RETHINK**

> 如果有那么几本书值得一看的话，那么《REWORK》算是不错的一本。

实习之前我只是一个热衷于技术的硬件出身的软件工程师，那本书是在找工作的艰难的日子里买的，慢慢地开始以不同的眼光来看待技术这东西。人会不停地成长，从一个台阶迈向另一个台阶。

很早以前只会去学习不同的技术，没有理解之间的一些奥秘，或许是因为没有多少的项目实践。以硬件为例，当需要一个串口通信来点亮一个 LED 的时候，可能会拿 51 开发板去做，可能会用 MSP430 去做，类似的有时候也需要 K60 去实现，而在刚开始的时候会去研究代码，渐渐地发现我们还是用 Arduino 吧，我们只需要围绕着我们的功能并快速去实现。换句话说，理解那么不同框架的相似之处没有多大的意义，不同框架的不同之处的学习才是有意义的。

```
print "Hello,World"
```

研究这些相似的知识已然没有多大的意义，我们需要的或许是诸如

```
if __name__=='__main__':
```

这种限于某种语言的语法。

慢慢地会发现取决于我们要做的事，我们去选择我们需要的东西。用 Holmes 的话说就是，

> 我们要有选择地将某些东西放进大脑里。

他的意思就是，我们不需要去关心一些无关紧要的东西——如地球是不是围绕着太阳转的。如果我们没有打算对某个领域进行研究的话，那么我们有时候可以放弃这领域的知识，尽管我们可能可以从中学到某些知识。

于是我开始以实用的角度来考虑问题:

- 如何加快网站地载入速度
- 如何提高网站地用户体验
- 如何创造有价值的内容
- ...

什么是我们需要的知识，什么是我们不需要的知识便一目了然，大概这就是所谓的实用主义。以解决问题来学习，而在无聊的时候便是所谓的写作驱动学习，很扯淡却又不扯淡。

某天我突然醒悟过来自己似乎又上了一个台阶，RETHINK 的意义和 REWORK 有着异曲同工之妙。技术之路才算真正的开始，当寻 ta开始有一点不错的势头的时候，我开始在想金字塔中底层的基石是最多的，而社会是金字塔结构。

技术 **x** 工匠 = 手工艺人

> 如果我们还是用学徒制的话，那么中医就不会发展成现在副模样，工匠的技艺也不会失传。

我们学习了外国的太多的东西，在我们开始忘记了我们的/(linux 中是指根) 的时候，我们是否会去追述起源。曾经

- 热衷于古诗词
- 会试着做风筝
- 记录古建筑的结构
- ...

我们接受着越来越多的外来文化，然而我们的文化有传承么？看过一些工匠的故事，我们总在感概别人的设计巧夺天工。总是不喜欢被人叫所谓的工程师，为什么作为一个有所谓五千年文化的国度，我们不能再用工匠而自居呢？

习惯于网上看到所谓的各种程序员、程序猿、码农，便无力吐嘈，没有影响力是一方面，另外一方面没有一个合适的称谓。直至寒假的一次偶然，看到一本书的书名是《巴黎的手工艺人》(或者是诸如此类名字的时候) 才醒悟到这是我想要的名字。

我们是手工艺人，一个可以熟练地掌握着不同技术的工匠，然而与工匠不同的是——可以不停地创作。创作和码字的相同在于他们都是在码字，所以我们没有必要去研究他们的相同之处，不同之处创作更容易让人有自豪感。我们可以创造不同的机器去做不同的事

- 打扫卫生
- 洗碗做饭
- 带你去某地
- ...

然而世界也需要程序员去给他们编码，至于在未来相当长的一段时间内，做不到如此的智能。

于是这个世界会剩下很大一群手工艺人，那就是我们。

熟练 **x** 传承 = 大师

无他，唯手熟尔。

不知还会多少人会记得《卖油翁》中的这句话，在今天这些仍然是适用的。和“某个孩子”聊天的时候，她总会说

那些字写得好的，都是从小练起的。

换成 IT 领域的话，会发现这些仍然是适用的。

当我们把一切不可能的线索都排除时，剩下的就算再不可思议也是事实。

而这也成为了一个事实的存在，当我看着一个又一个所谓的天才程序员的诞生的时候，都会伴随着从某个个位数年纪开始编程。一开始的慢慢成长的一点点积累会为之后的进步带来很重要的影响。当我们开始一点一滴积累的时候，我们就需要一个又一个的导师去帮助我们。

因为只有熟练是不够的，如果我们不知道 C 有指针的时候，C 一点儿也不强大也不危险。虽说唯手熟尔，熟悉只是一部分，更重要的是要知道方法。如果我们不知道怎样更好地将没倒入罐中的话，熟练显得没有多少实际意义。有多少人重复干一些事情多少年，却没有多一点的长进。单纯地只是每天熟练性的复制、替换一些代码，最后也成为不了一个好的程序员。

记得小学时候的一次计算机比赛，仅仅是因为不知道一些 LOGO 语言的特殊指令及用法，就那样输掉了比赛。即使在现在也是记忆犹新，因为我们连所谓的用户手册都没有办法弄到，在我们村上的那个连不上网的计算机室。而很早以前计算机的普及率也没有现在的高，换句话说我们老师学会上网的时候是和我们一起接受培训的时候。

有意识的是又回到我们上面所说的所谓的学徒制，这或许可以为天朝带来一些新的希望。只是所谓的学徒制在当前更多地是限制于一些大型公司的内部，要流传出来怕不是一件容易的事。在这个时代要成为一个独行狭已然不是一件容易的事，而要在一个成功的大型项目内部成为大师也然变得简单了。经验慢慢地积累，或许不一定有用，但是有时又可能起着重要的作用。

似乎实习的日子才刚开始，不过有些事情很有趣，有些东西相当不错，比如说手上的这台电脑。足够大的屏幕让我们可以让思维有更大的空间，这也是这些空间对于我们的好处，不过这也有一个缺点就是写东西的时候，似乎会感觉有一点点。我们想要的不就是这些么，那就足够了，那么我们去想象这一片空间吧。

第一天：新的环境

上班路上

原本预计能早点到公司的，但是既然用原本预计的话，那么结果就是迟到了。我不是故意的，似乎没有预料到公交车会变得那么长，没有预料到上个电梯也会排队。总之，结果就是迟到了十几分钟，没想过这就是迟到的第一天。好在我们公司并不打卡什么的，因为那些东西没有一点实质性的用处，除了限制员工。

所以，迟到的人有很多，只是一开始没注意到，但是发现背后的背包都是 ThoughtWorks 的时候，我意料到了。所以，似乎也没有那么多担心了。而且我去的时候 HR 还没到。我们都不是睡懒觉的，只是公交让我们无奈了，所以不得不感慨明天背着个电脑去挤公交的时候会是怎么样的。所以，还是计划着明天还是早点去看看新闻吧。

MacBook

这部分只所以用这个作标题的原因是，剩下的时间都在折腾这个，所以也就用这个标题了。虽然用过一段时间的 mac os，但是很久没有用之后，还是把这些东西都忘了。只是似乎有足够的时候来学习，也不算是足够，不过光配置密码就让我郁闷了很久。到底是改不了密码，到底还写的都是英语。只是因为没有安排任务，所以也就过得很轻松。

MacBook Pro 的屏幕足够的大，2880x1800，至少比我原来的屏幕大上了一倍，因为原来的是 1440x768 这些参数可能没有那么有用。但是屏幕大了，也就觉得简单了，看的东西也就多了。

配置了一个早上没有什么明显的结果之后，下午就按照自己的意思来搭建自己的开始环境。如 emacs，homebrew 等等，毕竟这些东西都是会用到的。当然还有用于搭建 ThoughtWorks BBS 的环境，gradle,mysql 等等。似乎又烧了一天脑细胞，然后自己似乎应该好好看书。

Mac 的触摸板就是一个有趣的东西，至少要花点时间去学习这个，似乎手势在这上面用得很多，只是都是不熟悉的。可能这些东西原本就存在于自己的电脑上，而没有去使用他们。等等忘了说一点了，公司给配电脑的时候没有配鼠标，所以不得不去用这个来熟悉这些。我们所知道的那么东西又得重新开始了，不同于 openSUSE 或者是

Ubuntu 上面可以直接用命运来安装软件。brew install 虽然也很强大，但是还是没有 apt-get 强大不是么，然后我们又要继续一些东西的学习。

LBS

let's build something。这个也不算是一个题外话，只是有些东西，一个团队来做会比一个人有激情。

实习两周：我所理解的敏捷开发

我所理解的敏捷开发应该是快速的，而后我发现所谓的敏捷并不仅仅只是在开发上面，还包括其他很多东西。或许是之前对于软件开发没有一个系统的认识，敏捷应该是一个过程。

软件开发流程

这样一个开发流程我们可能在不同的书上看到过这样的一个过程，如之前推荐的 Robert,C,Martin 的《敏捷软件开发原则、模式与实践》，英文名《Agile Software Development—Principles,Patterns,and Practices》，似乎这本身将我们之前上过的内容都很好地体现了出来，一本不错的好书，虽然包含了很多书的内容。在有些方面可能不是很具体，但是是个不错的融合性书籍。

传统的开发流程

设计 -> 编码 -> 测试

TDD 的开发流程

测试 -> 编码 -> 重构

两周的时间里和两个前辈（暂且这么说，找不到别的合适的词）pair，也就是结对编程，似乎是两个不同的过程。只是 TDD 在有时候并不是很重要，从理论上来说 TDD 可能会导致效率降低，但是可以保证程序的功能是正确的。做个小的总结吧，TDD 有下面的优点:

- 可以保证程序是正确的
- 是重构的基础保证
- 及时交付，即使现有的程序很 ugly

敏捷开发很重要的一点就是持续交付，而 TDD 正是这样一个不错的保证。TDD 的

过程就是: 红 -> 绿 -> 重构

- 编写测试
- 写出符合功能要求的程序，虽然是臭的
- 通过测试
- 重构代码

设计模式

之所以在这里提到这个是因为感觉这方面很重要，不过如果一开始以设计模式为出发点，可能就变成传统的开发流程，而不是 TDD。也就是先设计，再编程，这个和 TDD 有点距离。两者间的好与坏，可能没有办法那么快区分，没有哪一个是完全正确的，都只是相对的。

设计模式是在设计的过程中一点点累积起来的，这些会体现在过程中。

还有很重要的点是，代码是写给人看的。

实习第三周：结对编程与软件工程

想想来到 ThoughtWorks 西安办公室已经三周了，似乎越来越期待在 TW 的日子，想越早成为一个合格的 TWer。虽然不知道真正的合格的 TWer 要符合怎样怎样的要求，有时候我更需要将目标放空，计划这点东西计划的时候对于我来说会考虑其实现。而且制定一个合理的短期、中期、长期目标似乎对自己来说比较简单，这些在不断地迭代中前进着。

在公司的日子越来越有挑战性，也越来越有意思，只是有时候我在想我是不是投入不够多。简单的工作看上去应该是要有更多的学习时间和机会，不过我想我开始融入工作，好的一点。

Pair programming

这是一个一直在重复的话题，我们的工作天天需要，刚开始的时候我会试着去想这个是不是真正的必要。一开始看的时候觉得只对于编码是有益的，后面渐渐发现对于项目组来说这个是有着重要的意义，对于新人来说：

- 可以快速适应项目组，在边写边学的过程中
- 和不同的人 Pair，可以熟悉不同的人风格，以便适应团队

- 学习不同的技能及能力

对于项目组来说：

- 每个人都不会那么重要
- 当有一个不在时，另外一个人熟悉这个过程
- 其他的，暂时没有想到

迭代与软件工程

对于软件开发算是一个新手，向来对于自己做的东西来说就是自己想加什么功能就加什么功能。不会去考虑测试，不知道怎么去合理的测试。一开始的时候不知道测试真正的用途是干什么，而且如果有那么多的时间去写测试还不如加一个新的功能来得简单。在自己熟悉的嵌入式领域，似乎要写测试也不太可能。

测试在大的工程中是必需的，之前在 coding 的时候一直会想，怎样才能快速定位修改过的函数的代码。软件工程算是一种方法，一些前人总结出来的东西总是很有用的。

迭代有意思的一点在于，切换角色，似乎有一点在于可以保证每个人对产品代码都很熟悉。

工作

在三周的时候里，从前端到后台有了一个大致的理解。刚开始由于做的是底层的东西，对于我来说不是前端都算是底层 ~~。这个算是自己薄弱的环节，在之前做的网站都是只做前台。前台对于我来说比较理想，可以和真实的用户接触，后台就这么地弱了下去。当我试着用 PHP 框架 Laravel 去搭建一个 CMS 的时候，我发现原来这个不是很难，相比处理浏览器的兼容性来说。

更多的还需要慢慢去思考，不过感觉一开始将精力放置于具体的问题有点没必要。

实习记：重构与 **Intellij Idea** 初探

是时候讨论这个 Refactor 利器了，最初看到这个重构的过程是从 ThoughtWorks 郑大晔校开始的，只是之前对于 Java 的另外一个编辑器 Eclipse 的坏感。。这些在目前已经不是很重要了，试试这个公司里面应用广泛的编辑器。

开发的流程大致就是这样子的，测试先行算是推荐的。

编写测试->功能代码->修改测试->重构

上次在和 buddy 聊天的时候，才知道测试在功能简单的时候是后行的，在功能复杂不知道怎么手手的时候是先行的。

Interllij Idea 重构

开始之前请原谅我对于 Java 语言的一些无知，然后，看一下我写的 Main 函数：

```
package com.phodal.learing;

public class Main {

    public static void main(String[] args) {
        int c=new Cal().add(1,2);
        int d=new Cal2().sub(2,1);
        System.out.println("Hello,s");
        System.out.println(c);
        System.out.println(d);
    }
}
```

代码写得还好 (自我感觉)，先不管 Cal 和 Cal2 两个类。大部分都能看懂，除了 c,d 不知道他们表达的是什么意思，于是。

Rename

快捷键**:Shift+F6**

作用**:** 重命名

- 把光标丢到 int c 中的 c，按下 shift+f6，输入 result_add
- 把光标移到 int d 中的 d，按下 shift+f6，输入 result_sub

于是就有

```
package com.phodal.learing;

public class Main {
```

```
    public static void main(String[] args) {
        int result_add=new Cal().add(1,2);
        int result_sub=new Cal2().sub(2,1);
        System.out.println("Hello,s");
        System.out.println(result_add);
        System.out.println(result_sub);
    }
}
```

Extract Method

快捷键**:alt+command+m**

作用**:** 扩展方法

- 选中 System.out.println(result_add);
- 按下 alt+command+m
- 在弹出的窗口中输入 mprint

于是有了

```
public static void main(String[] args) {
    int result_add=new Cal().add(1,2);
    int result_sub=new Cal2().sub(2,1);
    System.out.println("Hello,s");
    mprint(result_add);
    mprint(result_sub);
}

private static void mprint(int result_sub) {
    System.out.println(result_sub);
}
```

似乎我们不应该这样对待 System.out.println，那么让我们内联回去

Inline Method

快捷键**:alt+command+n**

作用**:** 内联方法

- 选中 main 中的 mprint
- alt+command+n
- 选中 Inline all invocations and remove the method(2 occurrences) 点确定

然后我们等于什么也没有做了 ~~:

```
public static void main(String[] args) {
     int result_add=new Cal().add(1,2);
     int result_sub=new Cal2().sub(2,1);
     System.out.println("Hello,s");
     System.out.println(result_add);
     System.out.println(result_sub);
}
```

似乎这个例子不是很好，但是够用来说明了。

Pull Members Up

开始之前让我们先看看 Cal2 类:

```
public class Cal2 extends Cal {

     public int sub(int a,int b){
          return a-b;
          }
}
```

以及 Cal2 的父类 Cal

```
public class Cal {

     public int add(int a,int b){
```

```
        return a+b;
    }

}
```

最后的结果，就是将 Cal2 类中的 sub 方法，提到父类:

```
public class Cal {

    public int add(int a,int b){
        return a+b;
    }

    public int sub(int a,int b){
        return a-b;
    }
}
```

而我们所要做的就是鼠标右键

Refactor->Pull Members Up

Pull Members Down

将方法推迟到子类

Intellij Idea 自带的重构功能似乎真的很强大，其他目前感觉很强大的东西，比如 Live Template，丢到下次讨论。

结束: 不是开始的开始

当我开始实习的时候不得不面对这样的几个问题:

- 不是很熟悉的 Mac OS
- 完成熟悉的 Intellij Idea，需要去熟悉的快捷键
- 项目结构复杂

而在半年之后发现有些东西是一些熟能生巧的东西，有些需要在应用的过程中积累

才能变得熟能生巧。开始实习之前有过一些关于技能的小培训，有些东西会用到，有些不会。

简历 **x?=** 技能

简历是一个有趣的话题，特别是已经在开始的时候提到关于找工作的艰难的日子里的时候。或许我更愿意将此文表达为——将实习过程中的学到的东西，在艰难的日子到来之前好好学习的话，你可以就找到一个好工作的。

于是我们应该去写写我们的简历，看看我们能写下什么东西:

每每写简历时，也是对自我能力评估的时候。于是我试着去写下那些我知道的东西，但是似乎太多了，学太多的东西有时并不是一件好事。只是一些能力似乎也是我们应该去掌握的，我们在课堂上似乎学了太多无用的东西。

所以我试着把我所知道和提升的东西在这里一一罗列出来，也就是实习记的内容了。

内容 **x** 成长 **=** 实习记

实习前掌握了一些知识，而在实习的过程中对一些知识进行了强化，还有一部则被弱化了。也许是不再重要，也许仅仅是因为在当前不再重要。所以实习记的这些内容，实际上也就是:

基础

- HTML
- CSS
- Javascript
- GNU/Linux
- Nginx/Vim
- Git

进化

- Python
- 重构
- 测试
- 重构到模式
- 持续集成

- 敏捷开发

实战

- 物联网毕业设计
- 网站重构
- 用户体验
- story 1
- 博客
- Full Stack

而这些也只是把自己所知道的知识一一罗列了出来，整理而成的。上面的标题更像是内容的缩写，也是核心的知识吧。基础篇或许有些过于基础，进化篇便是实习中学到的知识，实战则是一些平时的游手好闲。

工具论

虽然很不愿意继续回到这个话题，但是不得不回到这里。记得在一篇测试的文章——《每个程序员必知之: 程序员差别的本质》中写到的内容，(ps: 这是一篇软文，只是用来测试标题的作用)，但是并没有另外一篇写得好。

音乐就在你的指尖流淌

> “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。”

DW 还是 notepad++? Adobe Dreamweaver 确实是一个不错的工具，虽然一直用的是和谐版的。这样有两个不好的地点，当我们依赖上这个工具的时候，我们不得不开始去负担他的费用。有时候我们不得不在终端下面编程他们，如果是 windows 服务器自然是无所谓了。DW 在有些时候忽略了太多的东西，曾经试着去用过一段时候，感觉效果还不如 notepad++ 加上 chromium 来得简单呢。

> 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 notepad++ 或者 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 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候完成编译，运行了。

我的意思是在当前应该专注于我们所要做的事情。

找到饭碗，丢掉梦想

我们找到了饭碗，我们是否在这时候丢掉了梦想呢?

关于题目的由来是因为某天在网上看到一个视频上说的，大题也就是题目说的那样——大学是大多数人找到饭碗丢掉梦想的地方。也不肯否认其意义，以及话的意味所在。不过，好在可以庆幸的是，大学对于我就只差那么一个目标了，也就是之前在说说中所说的一样，作为最近一个目标所在，更多的本文是在解释最后一个目标。

大学与梦想

即使一开始是对于大学没有好感，高中便是如此。也是因此而明白了，这是在中国，也就是我们所谓的体制影响下的东西一样。这就是不仅我们的绝大多数高中是一样的，用着相同的方式，对着相同的目标。我们的大学亦是如此，确实大学是有区别的，但是这种区别不是我们想象中的一样。中国大学的很大一部分差别不是来自于学校本身，而是源自学校的收入。也就是绝大多数大学的差异实际上不是，所谓的大师论，而是另外一种论调。确实一所好的大学必须有大师的存在，而只有少数的大师是源自于某种意味的，大部分的所谓的“大师”必然是由另一点开始的也就是投入。

这也是为何如此多的研究者只愿意去国外的原因。事实上，国内大多数大学如同高中本身就是重复的。当然了，传说中的教育资源投入不足是一个原因的，但是更重要的一个原因是我们所谓的体制问题——教育资源浪费。大多数学校不断地重复别的学校做出的一些东西，而很大一部分原因是取决于学校对于研究的开放程度所造成的资源的浪费。这也就是为什么我乐意去看那些国外的书籍而不是国内书籍的原因了。开放体制下的国外学术，有利于发展。

当然这里所谓的解释只是为了说明国内大学的一个事实，也就是如标题所说的一样——-找到工作，丢掉梦想。事实上，我们所知的大多数大学是一样的。也就是大多数

人的大学生活，没有多大区别。也就是考试前开始奋战，而大多数时间都是浪费的。所以，自然而然的这也是我们所知道的结果。

想法与梦想

扯完这一切之后，也就是我来自大学之前的那些想法，差不多逐一实现了，当然了还是只差那么一个很难完成，也许压根无法在大学解决。

至于想法与梦想，这个说法不知道用什么样的论调来说好。当然，第一个完成的自然是社团了，也就是墨颀。所以，这也算是一切的起源了。幸运的是，一开始我们所做的比想象中的简单。只是由于讨厌制度本身，换句话说，这些东西在程序上没有多大的异同。而想法的起源也是来自于云凌斋，当然了，我对于当社长什么的也是没有多大想法。只是，不小心当上了，也只能这么说了。还有一部分是对于某个大学的社团不是很满意，而这个不是我们学校，在这之前遇到的学校的某个群里面的。而一部分工作是在大学开始之前的暑假开始准备的，也因此到大学就相对比较简单了。

因而，在那之后就是，三个有梦想的孩子开始创建社团的过程。能说的不是很多，但是很辛苦。也就是所谓的有时候吃不到饭的那种样子，所以呢。只是随后，社团自然也就转到别的地方去，随后也就不存在了，也就是美食社。人差不多还是那些，只是差不多没有我们两个。所以，也就这样了。虽然没有好的结局，但是实现了自己想做的事情。事实上，多数人忽略的是另外一个事实吧。墨颀虽然不是以社团的形式存在了，但是还是存在的，但是是以另外一种形式存在的。也就是，另外一种形式，意料之外的另一种形式。只是和原来想做的某件事重叠了，也因此呢。算是完成了另外一个想法了，只是这是一个艰辛的过程。

而，随后的就是学习了。当然了，学习的想法只是一开始就想做的，只是学的是自己想学的东西，因为一开始是直接瞄准嵌入式系统，所以核心部分之一的软件体系解决得很早。也就是差不多大一的时候就解决了，所以然后就开始着手解决硬件体系的问题。只是相对于现在的事实来讲，还需要一点点时间问题，也就是需要收入补充对于硬件知识的匮乏吧。软件什么的一开始就相较于硬件成本低，也就是开发本身可以只由计算机解决。同时，也可以由计算机本身来仿真硬件，所以也就没有必要浪费大量的时间和金钱在上面了。

然后，在之前开始统计之前的花费的时候，发现花费的东西已经相对够多了。换句话说，如果只是在学校这边的花费，已经是接近那些计算机书的花费了。事实上，硬件本身是一个很有趣的事情，也就是接触着由不同体系组成的结构是一件相对娱乐的过程。也因此我们可以解开大多数生活中的电子产品是由什么组成的。原理什么的也能因此而解决和了解更多的，随后之后更多的是将时间和精力花在手机以及电脑上，也就是 8086 及手机的一些原理和组成。

因而，随后发现的问题和解决的问题越来越多，也随此了解了之前那些知识积累的必要性了。

这也就是另外一个在大学想做的事情了，也就是了解自己想了解的那些关于计算机的一切。也因此在计算机书的投入算是大学中最多的吧，然后是硬件，后者是与专业相关的，也算是与计算机相关的。这个也就没有必要做大多的解释了，毕竟统计过的话，应该是达到一千了。因为一直是在学习，所以越来越学习得有兴趣了。

以及还有其他什么的，也都差不多了。当然了，因为一开始就规划好了，所以梦想、喜欢、兴趣什么的都结合到了一起了，也就没有多大压力了。可以边做着自己喜欢的事，然后赚钱。也就是，这样以后也可以喜欢自己的生活，过着想要的生活。

未来

至于未来是什么样的已经不是很重要了，当然了人什么的，也是一个很有趣的问题。也就是我们不能有太多的欲望，不要有太多的想法。一次只做一件事，也就够了，虽然经常不是这样子的，但是多数时候还是专注于自己选择的地方。也就因此引出下文吧，主力只用于战争的主要目的消灭敌人。也就是那个还未达成的那个目标吧。其实一开始就已经很明白了，结局是怎样的。也就是人的想法，事实上，按照福尔摩斯的说法，人是可以从外观一下子判断出来的。人们的习惯什么的，可以很容易观察出来的。换句话说，事实也就是这样子的。也就是更多的就是一些归纳问题，经验本身是很有用的，虽然也可能由此造成负影响。但是，很多情况下，都是有用的。只是不承认了那个现实而已，而现实也就在那边了。当排除了其他一切不可能的线索的时候，剩下的就算再不可思议也是现实。而也就是我面对的那个问题了，答案也很明显，也就是如同《孙子兵法》说的一样。而后，结合各种三国游戏得出了的结论，真相就是如此

扩展篇

SEO

似乎因为受这篇文章的影响 http://katemats.com/what-every-programmer-should-know-about-seo/ 于是我也觉得我应该写一个每个程序员必知之 SEO，作为一个擅长前端兼 SEO 的设计师。

搜索引擎是如何工作的

如果你有时间，可以读一下谷歌的框架：

http://infolab.stanford.edu/~backrub/google.html

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。

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。

图 31: search-engine-arch

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

图 32: 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，可以参考一下这篇文章

寻 ta 分析与网站内容

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)

技术的本质

当我开始在阅读《技术的本质》的时候，我就开始在思考这样一个问题，我们在使用技术还是工具。

技术与工具

在某百科上说

一项技术是关于某一领域有效的科学（理论和研究方法）的全部，以及在该领域为实现公共或个体目标而解决设计问题的规则的全部。

对于技术不同的人的理解可能是不同的，和上图中的使用工具类似的是，和工具一


21:16
3.27
80
https://www.phodal.com/blo
4
Blog
《Python Web开发 - 测试驱动方法》阅后感 ›
Posted by: Phodal Huang in Full Stack
Oct. 27, 2015, 8:54 p.m.
这本书的原名是叫《Test-Driven Development with Python》，小标题是 Obey the Testing Goat: Using Django, Selenium, and JavaScript。虽然有点难以理解为何这本书的中文名变成了《Python Web开发 - 测试驱动方法》，总感觉怪怪的，毕竟Kent Beck的那本书名是《测试驱动开发》。
Tags: python django tdd
READ MORE
0 Comments

图 33: 技术的本质

样，技术也在不断地成长和进行。网站的成长史似乎可以简化为下面的过程，可能还会有 ASP.NE 等等，只是因为我接触得比较少。

- 静态的 HTML
- CGI 和 Perl 脚本
- PHP
- J2EE
- Django
- Ruby on Rails
- Nodejs

实际上这是技术的一种演变，然而做为最核心的东西 HTML 似乎还是那样的。作为新技术产生的核心——HTML 也在不断也进化中。然而，没有想象中的那么明显，看上去像是不变的，只是技术在不断地向前前进。对于我们来说这些都是工具，有时我们在用工具创造中新的工具，好比是技术本身，通过结合一些技术创建出新的技术。同进化史一般，我们没有办法从无到有创造出一个新的东西，没有上帝。

而作为一个普通的程序员，我们所做的只是在使用工具，从芯片到语言，从语言到框架，从框架到实现。

编程的秘密

每个人在技术的成长过程中都有不同的经历，对于我来说现在的总结大概是如此 (ps: 有兴趣可以参考过去的那些语言)。

- 当我开始学习第一种语言 `LOGO` 时，我还小觉得很神奇，至少对于计算机还是保持神秘的。
- 当我开始学习 **C++** 时，由于作者对于其优雅的宣称，我觉得 **C++** 确实很优雅、
- 当我开始学习 `Python` 的时候，我发现简单才是我所要追求的。
- 当我开始学习 **Ruby On Rails** 的时候，我发现生成可以很强大，但是因为强大，所以没有意思。
- 当我开始学习 `Django` 的时候，我发现这才是我想要的订制。
- 当我开始写博客的时候，我觉得比于 **HTML** 来说，**Markdown** 才是适合我的。
- 当我开始写下此文时，我开始觉得我应该试着去做点什么。

于是我又回到了原点，开始迷茫我想要的是什么？当我实习半年以后，我学到了更多的东西（实习半年后: 正在变得高效），而我开始的时候我才在偶然的一次机会中才了解到，我们用的都是工具。只是，我们可以用工具创造出工具。

编程只是用来解决问题的工具，优美与否对于解决问题的帮助，取决于是一次性问题还是长期问题。编程的核心是解决问题，正如 SEO 的核心是内容 (详情见:每个程序员必知之 SEO)。于是，

> 我们把一个又一个的迷团解开了，剩下百无聊赖。

当我们在讨论生产率的时候，得知高级语言会比低级语言来得有生产率，但是效率可能会因人而异。高级语言来自于低级语言，这些似乎没有那么重要。人们熟悉了不同的 IDE、不同的语言，相比于那些入门者来说，谙熟语言的人只是更加熟练罢了。同《卖油翁》的我亦无他，唯手熟尔般，对于有些东西只是因为用多了，然后熟悉罢了。事实真的是这样么？如果我们每天写的是 **1+1=2**，我们会知道 **1+2=3** 么。

> 那么所谓的优秀的程序员和普通的程序员的差别在哪?

技术的成长

这里的技术指的不是个人在技术上的成长，而是技术自身的成长。

> 技术在某种程度上一定是来自此前已有技术的新的组合。

一个优秀的框架、软件、系统的产生必然会基于其他的已有技术或者框架，如:

Ubuntu GNU/Linux

- 内核是 Linux
- 编译器 GCC
- 库 GLIBC 等等
- 脚本语言 Python 等
- Bash
- 等

而这其中的一些小命令如 ls、cd、wget 也是这个系统的组成部分之一，我们无法找到一个不依赖于其他系统的软件。如果你自己动手编译过 Linux，或者你会更有体会一个 GNU/Linux 系统需要什么。从一个库来说它是基于其他的基本库，如 C 标准库，而从 C 标准库的编译又依赖于编译器，这些都循环中前进着。

```
gcc4.7编译出了gcc4.8
gcc4.6编译出了gcc4.7
```

gcc4.5编译出了gcc4.6
等等

这是对一个可以自身编译自身的编译器而言，我们无法忽视的是技术背后有许多细节。巨人是站在巨人的肩膀上，过去我们可能一群工程师一个月开发出来的软件，在今天可能可以由一个工程师一天开发出来。因为我们可以基于前人的经验及教训，而这也是所谓的高生产率的程序员和一般的程序员间的区别。

自然而然的优秀的程序员吸收了其他人的经验以及教训，换在今天来说，你今天所在的位置并不是因为你比别人聪明，或者是别人比你聪明，只是因为你吸收了更多的知识及经验。当然，教育不公平不应该这边的讨论范围。