
GR

Growth

Growth序
基础知识篇
Windows还是Unix? WebStorm还是Sublime?
WebStorm还是Sublime?
语言也是一种工具
效率与工具
论快捷键
快速启动软件
IDE
终端或命令提示符
包管理
环境搭建
搭建OSX开发环境
搭建Windows开发环境
搭建GNU/Linux开发环境
如何学好一门语言
一次语言学习体验
输出是最好的输入
如何应用一门新的技术
Web编程基础
从浏览器到服务器
从HTML到页面显示
HTML
HTML的hello,world
想用中文？
其他html标记
小结
CSS
关于CSS
代码结构
样式与目标
选择器
更有趣的CSS
JavaScript
Javascript的Hello,world
更JavaScriptful
面向对象
其他
前端与后台
如何选择一门好的后台语言
JavaScript
Python
Java
其他
MVC
后台即服务
数据持久化
数据库
搜索引擎
如何选择前端框架
Angular
React
Vue
jQuery
RESTful、JSON与Ajax
Ajax
JSON
MVVM
编码
一个Web应用的构建过程
构建工具
构建过程
Git与版本管理
Git初入
写代码只是在码字

学习编程只是在学造句
编程是在写作
编程只是在码字
Kick Off
Tasking
如何Tasking一本书
如何编写测试
测试金字塔
测试用例
测试力度
Mock与Stub
Stub
Mock
测试驱动开发
测试优先
功能实现
重构代码
Selenium与自动化测试
可读的代码
命名
函数长度
函数嵌套
重复代码
测试用例
代码重构
Intellij Idea重构
重构之提炼函数
重构之内联函数
重构之以查询取代临时变量
上线
容器
应用容器
Docker
LNMP架构
HTTP服务器
Apache
Nginx
IIS
代理
Web缓存
数据库端缓存
应用层缓存
前端缓存
客户端缓存
HTML5 离线缓存
可配置
Toggle
Spring PropertyPlaceHolder
数据分析
Google Analytics
受众群体
流量获取
SEO
爬虫与索引
什么样的网站需要SEO？
SEO基础知识
内容
Hadoop分析数据
UX
什么是UX
UX需要什么
UX入门
什么是简单？
进阶
用户体验要素
认知设计
持续交付
持续集成
瀑布流式开发
小步前进
自动化构建
持续交付

自动化
DevOps
云基础
遗留系统与修改代码
遗留代码
什么是遗留代码
遗留代码的问题
如何修改代码
测试
重构
修改测试
再次重构
网站重构
速度优化
功能加强
模块重构
回顾与新架构
自省
Retro
Well
Less Well
Suggestion
Action
浮现式设计
意图导向
重构
模式与演进
架构模式
预设计式架构
演进式架构
每个人都是架构师——如何设计一个博客系统
如何构建一个博客系统
相关阅读资料

Growth序

当我到了一个项目时，我发现这是一个遗留系统（没有人知道为什么这里是这样的），尽管我们有足够的测试覆盖率。接着在我们的另外一个项目里，我们不得不选择了基于别的项目组的代码（毕竟是同样的业务），这是一份遗留代码（充满bug、难以维护），并且没有人有兴致去维护好别人留下的代码。随后，我们开始重构现有的系统，使用新的技术、新的架构。尽管如此，我们的新代码却一直徘徊在遗留代码的边缘。

后来，我开始不断地思索这其中的原因。发现不仅仅是我们的项目是这样的，绝大部分的Web项目都是这样的——只要有业务存在。至今还没发现哪个项目逃离出这样的定律。尽管有一些项目在现在运行良好，但是并不代表他会被淘汰。因此，我们在不断地重复这样的循环。

接着，我就想到了Web开发实际上就是七个步骤：

环境搭建篇
编码
上线
数据分析
持续交付
遗留系统
回顾与新架构
也因此，我在本书中添加了一些基础知识的内容，希望大家能有所收获。

基础知识篇

在我们第一次开始写程序的时候，都是以Hello World开始的。或者：

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

又或许：

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

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

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

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

因此在这一章里，我们将讲述几件事情：

使用怎样的操作系统
如何去选择工具
如何搭建相应操作系统上的环境
如何去学习一门语言

Windows还是Unix? WebStorm还是Sublime?

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

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

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

WebStorm还是Sublime?

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

刚开始学习的时候，我们只需要普通的工具，或者我们习惯了的工具去开始我们的工作。我们要的是把主要精力放在学习的东西上，而不是工具。刚开始学习一种新的语言的时候，我们不需要去讨论哪个是最好的开发工具，如Java，有时候可能是Eclipse，有时候可能是Vim，如果我们为的只是去写一个hello,world。在Eclipse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。

工具是为了效率

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

我们用Windows自带的画图就可以完成裁剪的时候，我们就没有运行起GIMP或者Photoshop去完成这个简单的任务。效率在某些时候的重要性，会比你选择的工具有用得多，学习的开始就是要去了解那些大众推崇的东西。

了解、熟悉你的工具

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

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

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

语言也是一种工具

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

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

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

没有合适的工具，要么创造一个，要么选择一个合适的。

小结

学习Django的时候习惯了有一个后台，于是开始使用Laravel的时候，寻找Administartor。需要编译的时候习惯用IDE，不需要的时候用Editor，只是因为有效率，嵌入式的时候IDE会有效率一点。

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

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

等等

等

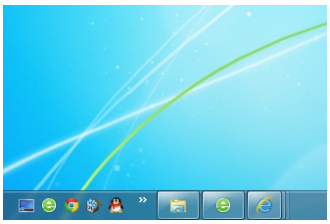
效率与工具

论快捷键

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

快速启动软件

在我还不和道有这样的工具的时候，我都是把图标放在下面的任务栏里：



Windows任务栏

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

Launchy

通过这个软件，我们可以在电脑上通过输入软件名，然后运行相关的软件。

IDE

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

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

一般来说，他们都应该有下面的一些要素：

快捷键

Code HighLight

Auto Complete

Syntax Check

终端或命令提示符

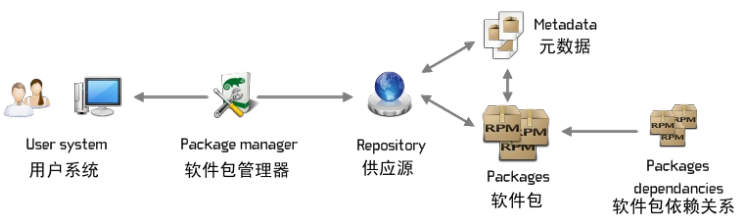
在开始写代码的时候，使用GUI可能是难以戒掉的一个习惯。但是当你习惯了使用终端之后，或者说使用终端的工具，你会发现这是另外一片天空。对于GUI应用上同样的菜单来说，在终端上也会有同样的工具——只是你觉得记住更多的命令。而且不同的工具对于同一实现可能会不同的规范，而GUI应用则会有一致的风格。不过，总的来说使用终端是一个很有益的习惯——从速度、便捷性。忘了提到一点，当你使用Linux服务器的时候，你不得不使用终端。

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

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

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

Dependencies Resolved

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

Linux终端截图

包管理

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



包管理

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

软件包: 软件包不止是一个文件，内含构成软件的所有文件，包括程序本身、共享库、开发包以及使用说明等。

元数据 (metadata) 包含于软件包之中，包含软件正常运行所需要的一些信息。软件包安装之后，其元数据就存储于本地的软件包数据库之中，以用于软件包检索。

依赖关系 (dependencies) 是软件包管理的一个重要方面。实际上每个软件包都会涉及到其他的软件包，软件包里程序的运行需要有一个可执行的环境（要求有其他的程序、库等），软件包依赖关系正是用来描述这种关系的。

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

环境搭建

搭建OSX开发环境

Homebrew

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

Homebrew Cask

> *brew-cask 允许你使用命令行安装 OS X 应用。*

iTerm2

> *iTerm2 是最常用的终端应用，是 Terminal 应用的替代品。*

Zsh

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

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

Oh My Zsh

> *Oh My Zsh 同时提供一套插件和工具，可以简化命令行操作。*

Sublime Text 2

> *强大的文件编辑器。*

MacDown

> *MacDown 是 Markdown 编辑器。*

Vimium

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

Charles Proxy

相关参考：

Mac web developer apps

搭建Windows开发环境

Chocolatey

> *Chocolatey是一个软件包管理工具，类似于Uutu下面的at-get,不过是运行在Widow环境下面。*

Launchy

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

PowerShell

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

WireShark

搭建GNU/Linux开发环境

Zsh

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

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

> *Oh My Zsh 同时提供一套插件和工具，可以简化命令行操作。*

tcpdump

如何学好一门语言

一次语言学习体验

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

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

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

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

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

GET到了5%的知识。

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

表达式和值
函数是一等公民
借贷模式
按名称传递参数
定义类
鸭子类型
柯里化
范型
Traits
…
看上去还行， 又GET到了5%的知识点。接着，依照上面的代码和搭建指南在自己的电脑上安装了Scala的环境：

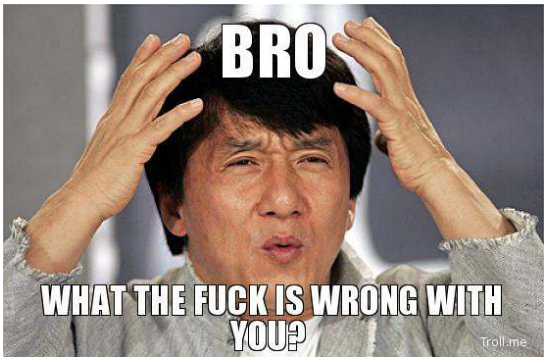
```
brew install scala
```

Windows用户可以用:

```
choco install scala
```

然后开始写一个又一个的Demo，感觉自己GET到了很多特别的知识点。

到了第二天忘了！

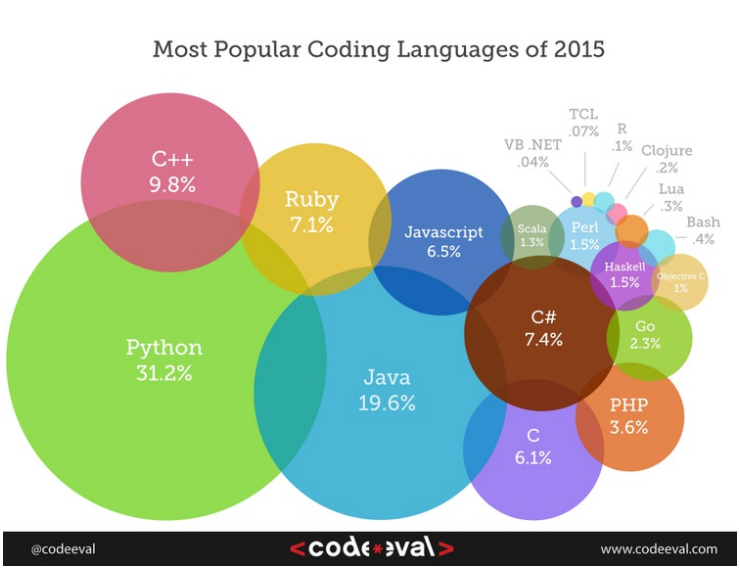


Bro Wrong

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

于是，你说出了Scala这门语言可以：

支持高阶函数。lambda，闭包…
支持偏函数。 match..
mixin，依赖注入..
等等
虽然隔壁的Python小哥赢得了这次辩论，然而你发现你又回想起了Scala的很多特性。



最流行的语言

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

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

先找一Scala的Web框架，Play看上去很不错，就这个了。这是一个MVC框架，原来用的Express也是一个MVC框架。Router写这里，Controller类似这个，就是这样的。
既然已经有PyJS，也会有Scala-js，前端就用这个了。
好了，博客重写了一遍了。

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

『让我想想我有什么杀手锏？』

『这里的知识好像还缺了一点，这个是什么？』

好了，你已经GET到了90%了。如下图所示：

Learn

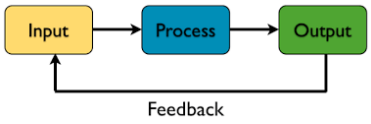
希望你能从这张图上GET到很多点。

输出是最好的输入

上面那张图『学习金字塔』就是在说明——输出是最好的输入。

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

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



Output is Input

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

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

所以，如果下次有人问你如果学一门新语言、技术，那么答案就是写一本书。

如何应用一门新的技术

对于多数人来说，写书不是一件容易的事，而应用新的技术则是一件迫在眉睫的事。

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

而学习一门新的技术的最好实践就是用这门技术对现有的系统行重写。

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

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

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

在这时我们已经实现了前后端分离了，这时候系统实现上变成了这样。

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

或者

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

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

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

系统就是这样一步步演进过来的。

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

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

Web编程基础

从浏览器到服务器

如果你的操作系统带有cURL[^cURL]这个软件(在GNU/Linux、Mac OS都自带这个工具，Windows用户可以从http://curl.haxx.se/download.html下载到)，那么我们可以直接用下面的命令来看这看这个过程[^HTTP2cURL](-v 参数可以显示一次http通信的整个过程)：

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

我们就会看到下面的响应过程：

```
* Rebuilt URL to: https://www.phodal.com/
*   Trying 54.69.23.11...
* Connected to www.phodal.com (54.69.23.11) port 443 (#0)
* TLS 1.2 connection using TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
* Server certificate: www.phodal.com
* Server certificate: COMODO RSA Domain Validation Secure Server CA
* Server certificate: COMODO RSA Certification Authority
* Server certificate: AddTrust External CA Root
> GET / HTTP/1.1
> Host: www.phodal.com
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 403 Forbidden
< Server: phodal/0.19.4
< Date: Tue, 13 Oct 2015 05:32:13 GMT
< Content-Type: text/html; charset=utf-8
< Content-Length: 170
< Connection: keep-alive
<
<html>
<head><title>403 Forbidden</title></head>
<body bgcolor="white">
<center><h1>403 Forbidden</h1></center>
<hr><center>phodal/0.19.4</center>
</body>
</html>
* Connection #0 to host www.phodal.com left intact
```

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

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

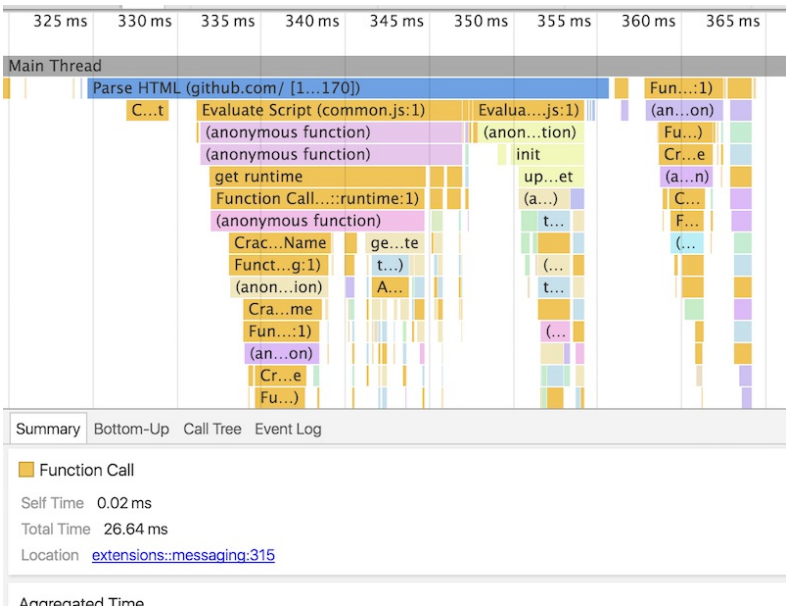
随后以“>”开始的内容，便是向Web服务器发送请求。Host即是我们要访问的主机的域名，GET / 则代表着我们要访问的是根目录，如果我们要访问 https://www.phodal.com/about/页面在这里，便是GET资源文件/about。紧随其后的是HTTP的版本号（HTTP/1.1）。User-Agent通过指向的是使用者行为的软件，通常会加上硬件平台、系统软件、应用软件和用户个人偏好等等的一些信息。Accept则指的是告知服务器发送何种媒体类型。

HTTP协议

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

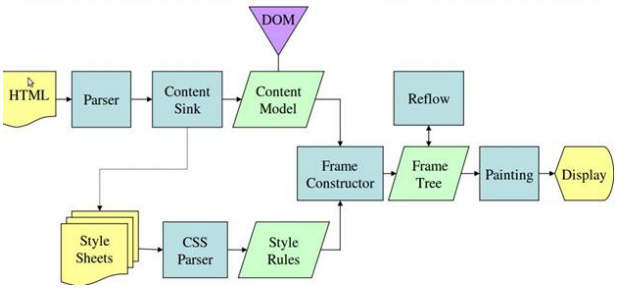
从HTML到页面显示

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



Chrome渲染的Timeline

这个渲染过程如下图所示：



Render HTML

HTML

让我们先从身边的语言下手，也就是现在无处不在的html+javascript+css。

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

HTML的hello,world

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

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

新建文件

输入

```
hello,world
```

保存为->“helloworld.html”，

双击打开这个文件。 正常情况下都应该是用你的默认浏览器打开。只要是一个正常工作的现代浏览器，都应该可以看到上面显示的是“Hello,world”。

这才是最短的hello,world程序，但是呢？在ruby中会是这样子的

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

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

遗憾的是，它可能无法在所有的浏览器上工作，所以我们需要去调试其中的bug。

调试hello,world

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

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

这个才是真正能在大部分浏览器上工作的代码，所以复制它到编辑器里吧。

说说hello,world

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

> *超文本标记语言*

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

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

我们所使用的汉语是人类用智慧创造的，我们所正在学的这门语言同样也是人类创造的。

我们在自己的语言里遵循着**桌子是桌子，凳子是凳子**的原则，很少有人会问为什么。

想用中文？

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

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

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

让我们把上面的代码代替掉原来的代码然后保存，打开浏览器会看到下面的结果

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

更不幸的结果可能是

```
<锘挎<锘挎<鏂囨。澶存<> <鏂囨。娴嬭瘯涓栫晫<鏂囨。娴嬭瘯缁撴潫鏂囨。澶存<鏂囨。娴嬭瘯缁撴潫
```

这是一个编码问题，对中文支持不友好。

我们把上面的代码改为和标记语言一样的结构

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

于是我们看到的结果便是

```
<语言> <头> <身体>你好，世界
```

被chrome浏览器解析成什么样了？

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

以

结尾的是注释，写给人看的代码，不是给机器看的，所以机器不会去理解这些代码。

但是当我们把代码改成

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

浏览器上面显示的内容就变成了

```
你好世界
```

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

结论:

浏览器对中文支持不友好。
浏览器对英文支持友好。
刚开始的时候不要对中文编程有太多的想法，这是很不现实的:

现有的系统都是基于英语语言环境构建的，对中文支持不是很友好。
中文输入的速度在某种程度上来说没有英语快。
我们离开话题已经很远了，但是这里说的都是针对于那些不满于英语的人来说的，只有当我们可以从头构建一个中文系统的时候才是可行的，而这些就要将cpu、软件、硬件都包含在内，甚至我们还需要考虑重新设计cpu的结构，在某种程度上来说会有些不现实。或许，需要一代又一代人的努力。忘记那些吧，师夷长之技以治夷。

其他html标记

添加一个标题，

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

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

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

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

小结

美妙之处

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

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

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

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

更多

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

CSS

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

下面就是我们之前说到的代码，css将Red三个字母变成了红色。

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

只是，

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

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

Red

Red Fonts

关于CSS

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

到目前为止我们没有依赖于任何特殊的硬件或者是软件，对于我们来说我们最基本的需求就是一台电脑，或者可以是你的平板电脑，当然也可以

是你的智能手机，因为他们都有个浏览器，而这些都是能用的，对于我们的CSS来说也不会有例外的。

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

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

代码结构

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

```
<!DOCTYPE html>
<html>
<head>
</head>
<body>
    <p style="font-size: 22px;color:#f00;text-align: center;padding-left: 20px;">如果没有一个好的结构</p>
    <p style="  font-size:44px;color:#3ed;text-indent: 2em;padding-left: 2em;">那么以后可能就是这样子。。。。</p>
</body>
</html>
```

虽然我们看到的还是一样的:

No Style

于是我们就按各种书上的建议重新写了上面的代码

```
<!DOCTYPE html>
<html>
<head>
    <title>CSS example</title>
    <style type="text/css">
        .para{
            font-size: 22px;
            color:#f00;
            text-align: center;
            padding-left: 20px;
        }
        .para2{
            font-size:44px;
            color:#3ed;
            text-indent: 2em;
            padding-left: 2em;
        }
    </style>
</head>
<body>
    <p class="para">如果没有一个好的结构</p>
    <p class="para2">那么以后可能就是这样子。。。。</p>
</body>
</html>
```

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

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

我们需要

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

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

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

这代码和JS的代码有如此多的相似

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

而22px、20px以及#f00都是数值，因此:

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

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

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

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

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

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

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

样式与目标

如下所示，就是我们的样式

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

我们的目标就是

```
如果没有一个好的结构
```

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

选择器

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

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

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

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

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

而通常我们可以通过一个

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

这样的元素选择器来给予所有的p元素一个左对齐。

还有复杂一点的复合型选择器，下面的是HTML文件

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

还有CSS文件

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

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

更有趣的CSS

一个包含了para2以及para_bg的例子

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

我们只是添加了一个黑色的背景

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

重新改变后的网页变得比原来有趣了很多，所谓的继承与合并就是上面的例子。

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

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

JavaScript

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

Javascript的Hello,world

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

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

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

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

按照标准的写法，我们还需要声明这个脚本的类型

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

没有显示hello,world?试试下面的代码

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

更JavaScriptful

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

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

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

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

从数学出发

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

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

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

设计和编程

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

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

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

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

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

函数

记得刚开始学三角函数的时候，我们会写

```
sin 30=0.5
```

而我们的函数也是类似于此，换句话说，因为很多搞计算机的先驱都学好了数学，都把数学世界的规律带到了计算机世界，所以我们的函数也是类似于此，让我们做一个简单的开始。

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

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

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

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

```
sin(30)
```

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

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

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

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

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

重新设计

我们将输出的工作移到函数的外面，

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

接着我们用一种更有意思的方法来写这个问题的解决方案

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

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

object和函数

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

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

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

```
object object
```

所以我们需要去弄清楚什么是object。对象的定义是

> *无序属性的集合，其属性可以包含基本值、对象或者函数。*

创建一个object，然后观察这便是我们接下来要做的

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

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

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

一个包含对象的对象应该是这样子的。

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

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

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

下面代码说的就是这个无序属性集中中的函数。

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

我们定义了一个叫IO的对象，声明对象可以用

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

又或者是

```
var store=new Object();
```

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

复杂一点的对象应该是下面这样的一种情况。

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

而这些会在我们未来的实际编程过程中用得更多。

面向对象

开始之前先让我们简化上面的代码，

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

看上去比上面的简单多了，不过我们还可以简化为下面的代码。。。

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

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

上面的方法显得有点不可取，换句话说和一开始的

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

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

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

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

其他

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

所谓的继承，

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

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

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

完整的Javascript应该由下列三个部分组成:

核心(ECMAScript)——核心语言功能
文档对象模型(DOM)——访问和操作网页内容的方法和接口
浏览器对象模型(BOM)——与浏览器交互的方法和接口
我们在上面讲的都是ECMAScript，也就是语法相关的，但是JS真正强大的，或者说我们最需要的可能就是对DOM的操作，这也就是为什么jQuery等库可以流行的原因之一，而核心语言功能才是真正在哪里都适用的，至于BOM，真正用到的机会很少，因为没有完善的统一的标准。

一个简单的DOM示例，

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

我们需要修改一下helloworld.html添加

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

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

```
var para=document.getElementById("para");
```

```
para.style.color="blue";
```

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

美妙之处

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

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

前端与后台

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？

后台在过去的岁月里起着很重要的作用，当然在未来也是。

就最几年的解耦趋势来看，它在变得更小，变成一系列的服务。并向前台提供很多RESTful API，看上去有点像提供一些辅助性的工作。

如何选择一门好的后台语言

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

JavaScript

在现在看来，JavaScript似乎是一个很不错的选择。毕竟只要是Web就会有前端，有前端就需要有JavaScript。而Node.js对于JavaScript在后台的应用已发挥了很大的作用。

而且:

Electron + Node.js + javascript 做桌面应用
ionic + javascript 做移动应用
node.js + javascript 网站前后台
Javascript + Tessl 做硬件

Python

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

Java

除此呢，我相信Java在目前来说也是一个不错的选择。

在学校的时候，一点儿也不喜欢Java。后来才发现，我从Java上学到的东西比其他语言上学得还多。如果Oracle不毁坏Java，那么他会继续存活很久。我可以用JavaScript造出各种我想要的东西，但是通常我无法保证他们是优雅的实现。过去人们在Java上花费了很多的时间，或在架构上，或在语言上，或在模式上。由于这些投入，都给了人们很多的启发。这些都可以用于新的语言，新的设计，毕竟没有什么技术是独立于旧的技术产生出来的。

其他

个人感觉Go也不错，虽然没怎么用，但是性能应该是相当可以的。

PHP呢，据说是这个『世界上最好的语言』，我服务器上运行着几个不同的WordPress实例。对于这门语言，我还是相当放心的。

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

MVC

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

层次	职责
表现层	提供服务、显示信息、用户请求、HTTP请求和命令行调用。
领域层	逻辑处理，系统中真正的核心。

数据层与数据库、消息系统、事物管理器和其他软件包通讯。
化身于当时最流行的Spring，就是MVC。人们有了iBatis这样的数据持久层框架，即ORM，对象关系映射。于是，你的package就会有这样的几个文件夹：

```
|___mappers
|___model
|___service
|___utils
|___controller
```

在mappers这一层，我们所做的莫过于如下所示的数据库相关查询：

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

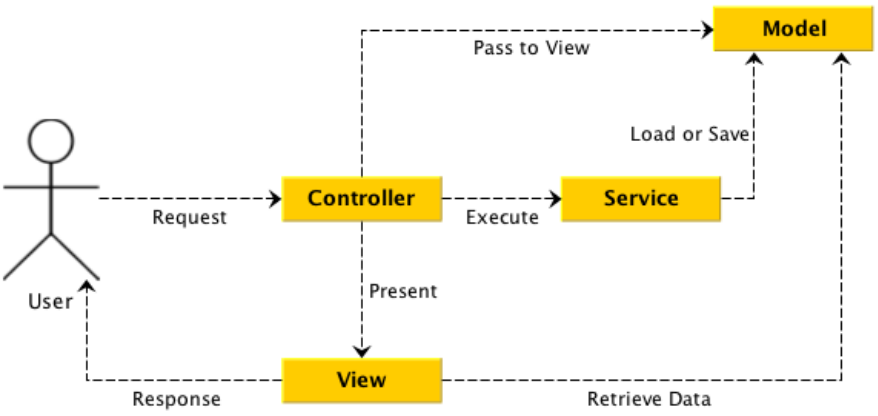
model文件夹和mappers文件夹都是数据层的一部分，只是两者间的职责不同，如：

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

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

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

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

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



Service MVC

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

我们在Controller层应该做的事是：

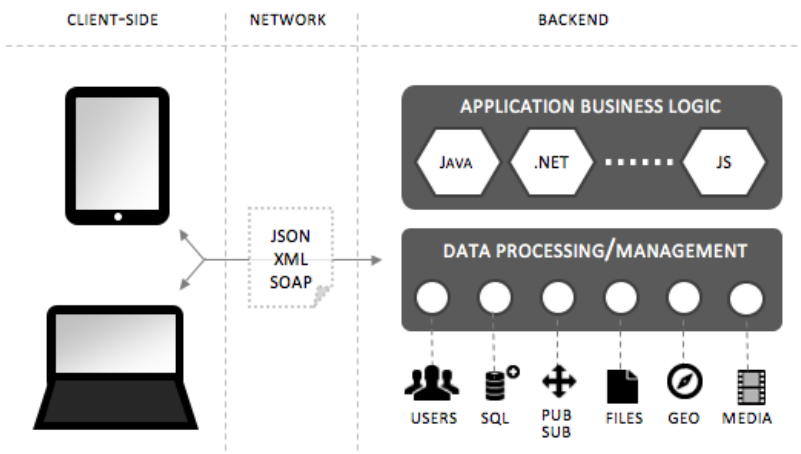
处理请求的参数
渲染和重定向
选择Model和Service
处理Session和Cookies
业务是善变的，昨天我们可能还在和对手竞争谁先推出新功能，但是今天可能已经合并了。我们很难预见业务变化，但是我们应该能预见Controller是不容易变化的。在一些设计里面，这种模式就是Command模式。

后台即服务

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

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

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



Backend As A Service

数据持久化

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

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

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

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。
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。
RESTful。RESTful相当于是CRUD的衍生，只是传输介质变了。
LocalStorage。LocalStorage算是另外一种方式的CRUD。
说了这么多都是废话，他们都是可以用类CRUD的方式操作。

数据库

数据库里存储着大量的数据，在我们对系统建模的时候，也在决定系统的基础模型。

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

```
var User = sequelize.define('user', {
  firstName: {
    type: Sequelize.STRING,
    field: 'first_name' // Will result in an attribute that is firstName when user facing but first_name in the database
  },
  lastName: {
    type: Sequelize.STRING
  }
}, {
  freezeTableName: true // Model tableName will be the same as the model name
});

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

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

搜索引擎

如何选择前端框架

选择前端框架似乎是一件很难的事，然而这件事情并不是看上去那么难。只是有时候你只想追随潮流，或者有一些些偏见。

Angular

Angular.js对于后端人员写前端代码来说，是一个非常不错的选择。它也用于在本应用中写APP，只是不知道它的2.0大坑让多少人没了兴趣。

React

React似乎很受市场欢迎，各种各样的新知识。但是它的发展远不如我的预期的好，理想的情况下应该类似于Ionic可以直接在Web和手机应用上。

Vue

Vue.js轻量级的框架。

jQuery

jQuery还是一个不错的选择，不仅仅对于学习来说，而且对于工作来说也是如此。如果你们不是新起一个项目或者重构旧的项目，那么必然你是没有多少机会去超越DOM。而如果这时候尝试去这样做会付出一定的代价，如我在前端演进史所说的那样——晚点做出选择，可能会好一点。因为谁说jQuery不会去解放DOM，React带来的一些新的思想可能就比不上它的缺点。除此，jQuery耕织几年的生态系统也是不可忽略。

RESTful、JSON与Ajax

Ajax

AJAX即“Asynchronous Javascript And XML”（异步JavaScript和XML），是指一种创建交互式网页应用的网页开发技术。

AJAX = 异步 JavaScript和XML（标准通用标记语言的子集）。

AJAX 是一种用于创建快速动态网页的技术。

通过在后台与服务器进行少量数据交换，AJAX 可以使网页实现异步更新。这意味着可以在不重新加载整个网页的情况下，对网页的某部分进行更新。

传统的网页（不使用 AJAX）如果需要更新内容，必须重载整个网页页面。

JSON

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

JSON WEB Tokens

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

MVVM

MVVM是Model-View-ViewModel的简写。

相对于 MVC 的历史来说，MVVM 是一个相当新的架构，MVVM 最早于 2005 年被微软的 WPF 和 Silverlight 的架构师 John Gossman 提出，并且应用在微软的软件开发中。当时 MVC 已经被提出了 20 多年了，可见两者出现的年代差别有多大。

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

编码

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

如果一件事可以自动化，那么就尽量去自动化，毕竟你是一个程序员。
快捷键！快捷键！快捷键！
使用可以帮助你快速工作的工具——如启动器。
不过不得不提到的一点是：你需要去考虑这个需求是不是一个坑的问题。如果这是个一个坑，那么你应该尽早的去反馈这个问题。沟通越早，成本越低。

一个Web应用的构建过程

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

构建工具

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

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

构建过程

在这里，我们要使用到的一个工具是Grunt，当然对于Gulp也是类似的。

> *Grunt是基于Node.js的项目构建工具。它可以自动运行你所设定的任务。Grunt拥有数量庞大的插件，几乎任何你所要做的事情都可以用Grunt实现。*

语法检测

JSHint，JSLint

自动化测试

Mocha

打包

Webpack

上传

AWS S3

发布

RPM

Git与版本管理

从一般开发者的角度来看，git有以下功能：

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

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

Git初入

如果是第一次使用Git，你需要设置署名和邮箱：

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

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

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

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

向这个本地的代码仓库添加当前目录的所有改动：

```
$ git add .
```

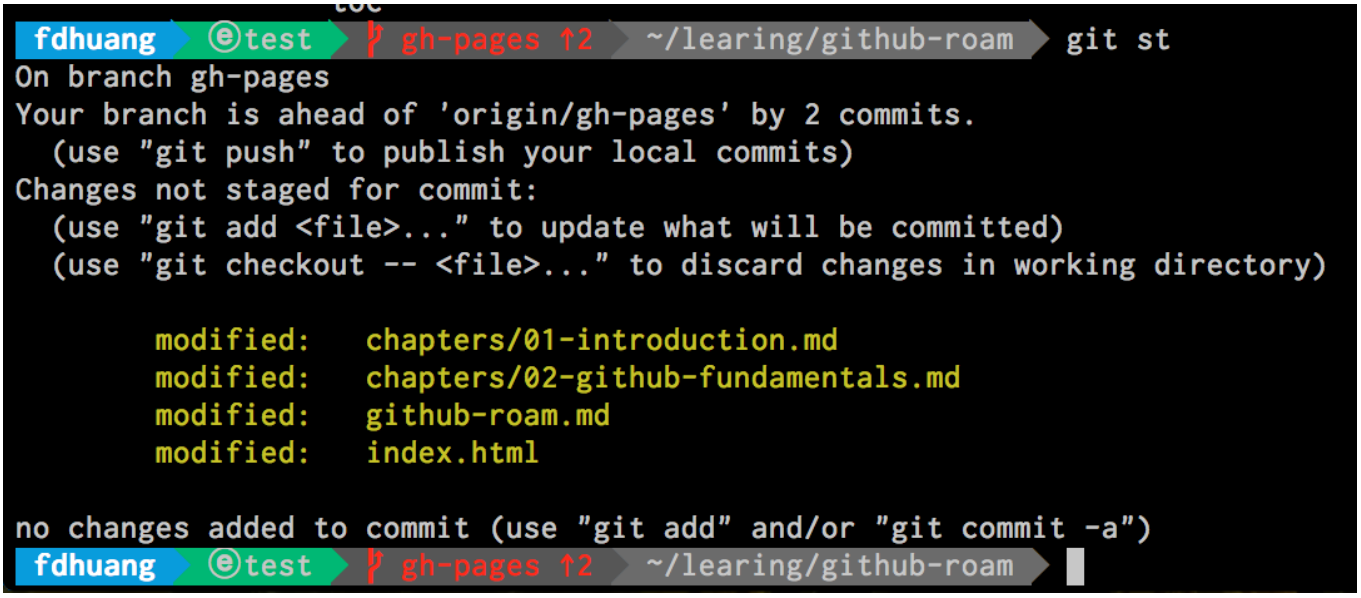
或者只是添加某个文件：

```
$ git add -p
```

我们可以输入

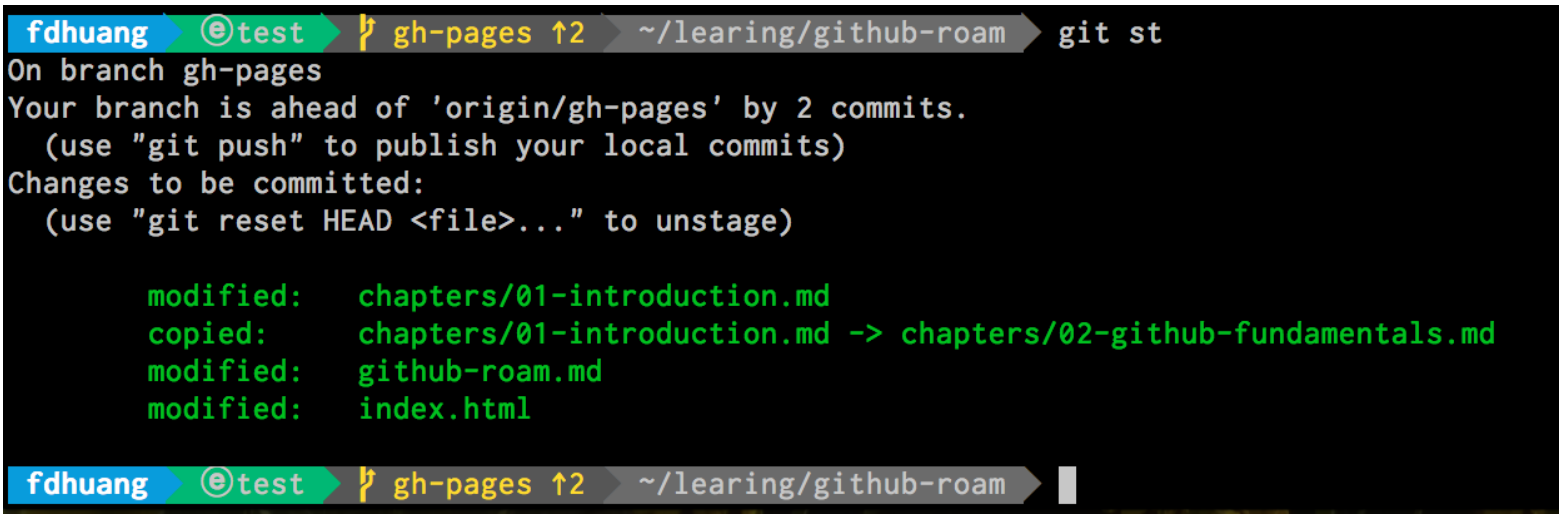
```
$git status
```

来看现在的状态，如下图是添加之前的：



Before add

下面是添加之后 的



After add

可以看到状态的变化是从黄色到绿色，即unstage到add。

写代码只是在码字

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

换个贴切一点的比喻，比如烧菜做饭，对于一个优秀的厨师和一个像我这样的门外汉而言，就算给我们相同的食材、厨具，一段时间后也许一份

是诱人的美食，一份只能喂猪了——即使我模仿着厨师的步骤一步步地来，也许看上去会差不多，但是一吃便吃出差距了。

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

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

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

学习编程只是在学造句

> *?多么无聊的一个标题*

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

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

于是我们开始组词造句，以便最后能拼凑出一整篇文章。

编程是在写作

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

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

那些所谓的写作逻辑对编程的影响

早期的代码是以行数算的，文章是以字数算的
代码是写给人看的，文章也是写给人看的
编程同写作一样，都由想法开始
代码同文章一样都可以堆砌出来(ps:如本文)
写出好的文章不容易，需要反复琢磨，写出好的代码不也是如此么
构造一个类，好比是构造一个人物的性格特点，多一点不行，少一点又不全
代码生成，和生成诗一样，没有情感，过于机械化
。。。
然而好的作家和一般的写作者，区别总是很大，对同一个问题的思考程度也是不同的。从一个作者到一个作家的过程，是一个不断写作不断积累的过程。而从一个普通的程序员到一个优秀的程序员也是如此，需要一个不断编程的过程。

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

> *写作的过程真的就是一次探索之旅，而且它会贯穿人的一生。*

编程只是在码字

> *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。

Kick Off

> *Kick Off也就是项目启动会议。当我们有项目启动的时候，把涉及的相关人员都聚集起来，然后召开一个启动的会议。*

对应于任务，则是任务的启动，开发人员和业务人员一起讨论问题。

Tasking

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

有一个可衡量、可实现、过程可测的目标
Tasking (即对要实现的目标过程进行分解)
一步步实现 (如TDD)
实现目标
现在让我们以这本书的写作过程为例，来看看这个过程是怎么发生的。

如何Tasking一本书

章节规则

Tasking文章

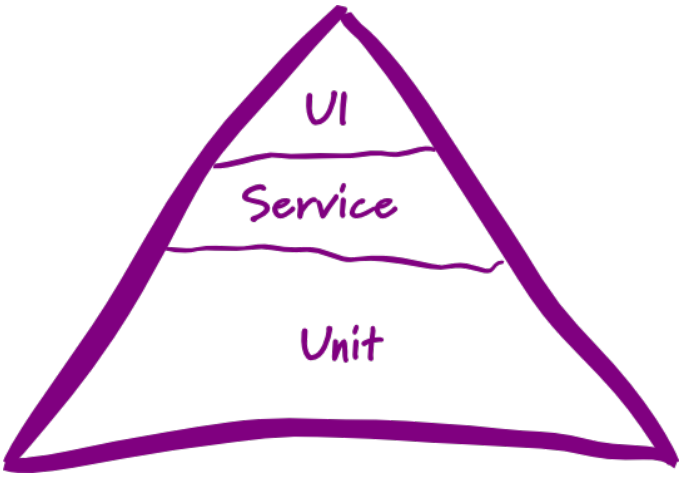
如何编写测试

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

这时，我们需要学会更好地去测试代码——测试金字塔。

测试金字塔

测试金字塔是由Mike Cohn提出的，如下图所示：



测试金字塔

测试用例

测试力度

Mock与Stub

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

Stub

Stub是一种状态确认
Stub用简单的行为来替换复杂的行为
Mock

Mock是一种行为确认
Mock模拟行为
简单的来说：Stub从某种程度上来说，会返回我们一个特定的结果，用代码替换来方法；而Mock只是确保这个方法被调用。

Stub

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

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

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

Mock

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

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

foo.callMe();

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

Mock Server

Mock是一大类的方法，其中还有一个

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:

先写功能的测试
实现功能代码
提交代码(commit -> 保证功能正常)
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:

已经有了原型
框架设计

测试优先

功能实现

重构代码

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?

Selenium与自动化测试

> *Selenium也是一个用于Web应用程序测试的工具。Selenium测试直接运行在浏览器中，就像真正的用户在操作一样。支持的浏览器包括IE(7、8、9)、Mozilla Firefox、Mozilla Suite等。这个工具的主要功能包括：测试与浏览器的兼容性——测试你的应用程序看是否能够很好得工作在不同浏览器和操作系统之上。测试系统功能——创建衰退测试检验软件功能和用户需求。支持自动录制动作和自动生成.Net、Java、Perl等不同语言的测试脚本。*

可读的代码

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

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

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

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

命名

函数长度

函数嵌套

重复代码

测试用例

代码重构

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。*

Intellij Idea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

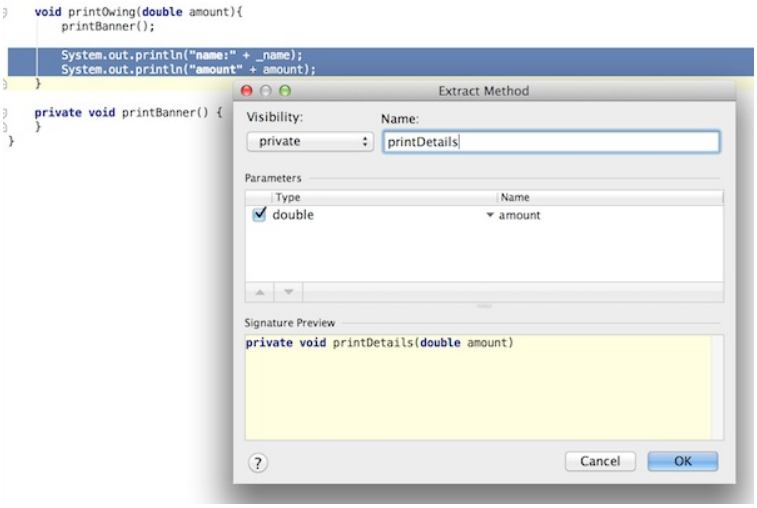
```
public class extract {
    private String _name;

    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);
```

按下上述的快捷键，会弹出下面的对话框



Extrct Method

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 call to extracted method...
```

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`

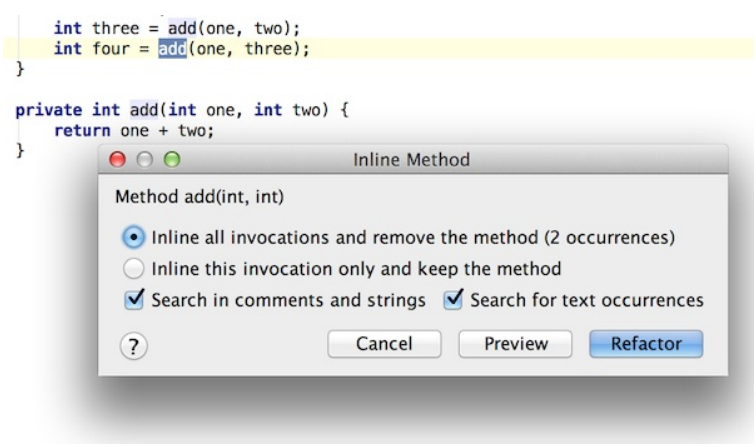
Windows/Linux: `Ctrl`+`Alt`+`M`

鼠标: Refactor | Inline

重构之前

```
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



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

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。

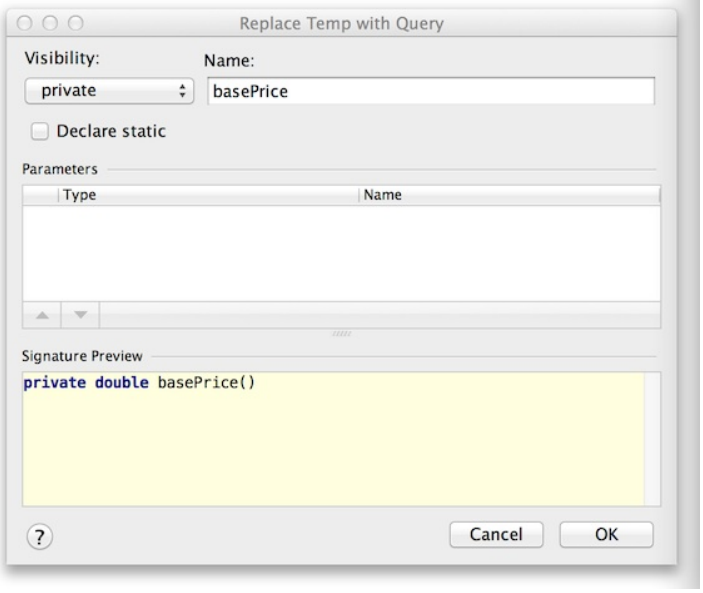
以书中的代码为例

```
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

```
    return basePrice * 0.95;
} else {
    return basePrice * 0.98;
}
```



Replace Temp With Query

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;
        }
    }

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

而实际上我们也可以

选中

_quantity * _itemPrice

对其进行`Extrace Method`

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);
    }

}
```

上线

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

一些简单的Ops技能。
了解服务器的相关软件
搭建运行Web应用的服务器
自动化部署应用
为了即时的完成工作，你是不是放弃了很多东西，比如质量?测试是很重要的一个环节，不仅可以为我们保证代码的质量，而且还可以为我们以后的重构提供基础条件。

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

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

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

容器

> *容器是应用服务器中位于组件和平台之间的接口集合。*

> *容器一般位于应用服务器之内，由应用服务器负责加载和维护。一个容器只能存在于一个应用服务器之内，一个应用服务器可以建立和维护多个容器。*

应用容器

Tomcat 服务器是一个免费的开放源代码的Web 应用服务器，属于轻量级应用服务器，在中小型系统和并发访问用户不是很多的场合下被普遍使用，是开发和调试JSP 程序的首选。对于一个初学者来说，可以这样认为，当在一台机器上配置好Apache 服务器，可利用它响应HTML（标准通用标记语言下的一个应用）页面的访问请求。实际上Tomcat 部分是Apache 服务器的扩展，但它是独立运行的，所以当你运行tomcat 时，它实际上作为一个与Apache 独立的进程单独运行的。

Docker

> *Docker 是一个开源的应用容器引擎，让开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。容器是完全使用沙箱机制，相互之间不会有任何接口（类似 iPhone 的 app）。几乎没有性能开销,可以很容易地在机器和数据中心中运行。最重要的是,他们不依赖于任何语言、框架包括系统。*

LNMP架构

Linux + Nginx + MySQL + PHP

HTTP服务器

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

目前最主流的三个Web服务器是Apache Nginx IIS

Apache

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

Nginx

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

IIS

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

代理

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

Web缓存

Web缓存是显著提高web站点的性能最有效的方法之一。主要有:

数据库端缓存
应用层缓存
前端缓存
客户端缓存

数据库端缓存

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

应用层缓存

应用层缓存这块跟开发人员关系最大，也是平时经常接触的。

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

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

分布式缓存一般可以分为几种:

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

前端缓存

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

客户端缓存

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

HTML5 离线缓存

可配置

让我们写的Web应用可配置是一项很有挑战性，也很实用的技能。

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

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

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

Toggle

很少用Java作为技术栈的我，很少有Java的笔记，记录一下这个简单的feature toggle。

Spring PropertyPlaceHolder

在Stackflow 上有一个关于这个问题的回答。

1.使用bean创建一个properties。(mvc-config.xml)

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

2.注入值

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

这样就可以在`root context`和`mvc context`下工作

3.在jsp中使用

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

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

4.在测试中使用

```
messageToggles = ResourceBundle.getBundle("myProps");
```

数据分析

数据分析是一个很有意思的过程，也是一个非常重要的过程，他们是非常重要的一个循环：

> *想法 -> 构建 -> 产品 -> 衡量 -> 数据 -> 学习 -> 想法*

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

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

我只能借助Google Analytics来收集用户的数据。从这些数据里去学习一些东西，而这些就会变成一个新的想法。

不得不补充一点的是，作为一个开发人员，会讨厌这点异想天开。

Google Analytics

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

受众群体

如下图是Growth应用，最近两星期的数据结果：

Growth GA

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

折线图就是每天的用户数。
下面会有用户数、会话、屏幕浏览量等等的一些信息。
右角的饼图则是回访问用户和新用户的对比。
最下方便是受众的信息——国家、版本等等。
从图中，我们可以读取一些重要的信息，如用户的停留时间、主要面向的用户等等。在浏览器版本会有：

浏览器与操作系统
移动设备
这样的重要数据，如下表是我网站20160104-20160120的访问数据：

浏览器	会话	新会话百分比
Chrome	5048	75.99%
Firefox	694	78.39%

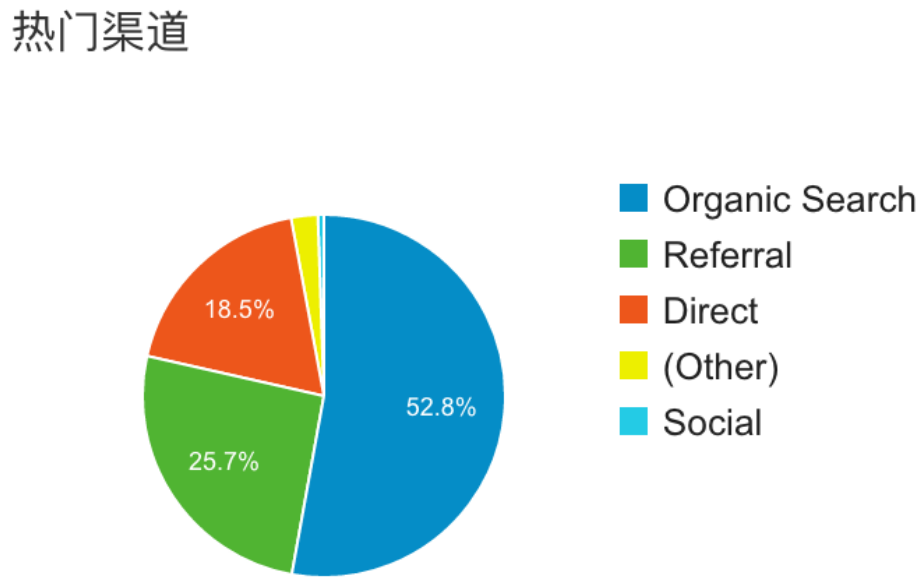
Safari	666	78.68%
Internet Explorer	284	87.68%
Safari (in-app)	92	86.96%
Android Browser	72	87.50%
Edge	63	79.37%
Maxthon	51	68.63%
UC Browser	41	80.49%
Opera	34	64.71%

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

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

流量获取

除此，不得不说的一点就是流量获取，如下图所示是我博客的热门渠道：



Phodal.com Traffic

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

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

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

加快访问速度
更表意的URL
更好的标题
更好的内容
等等等。

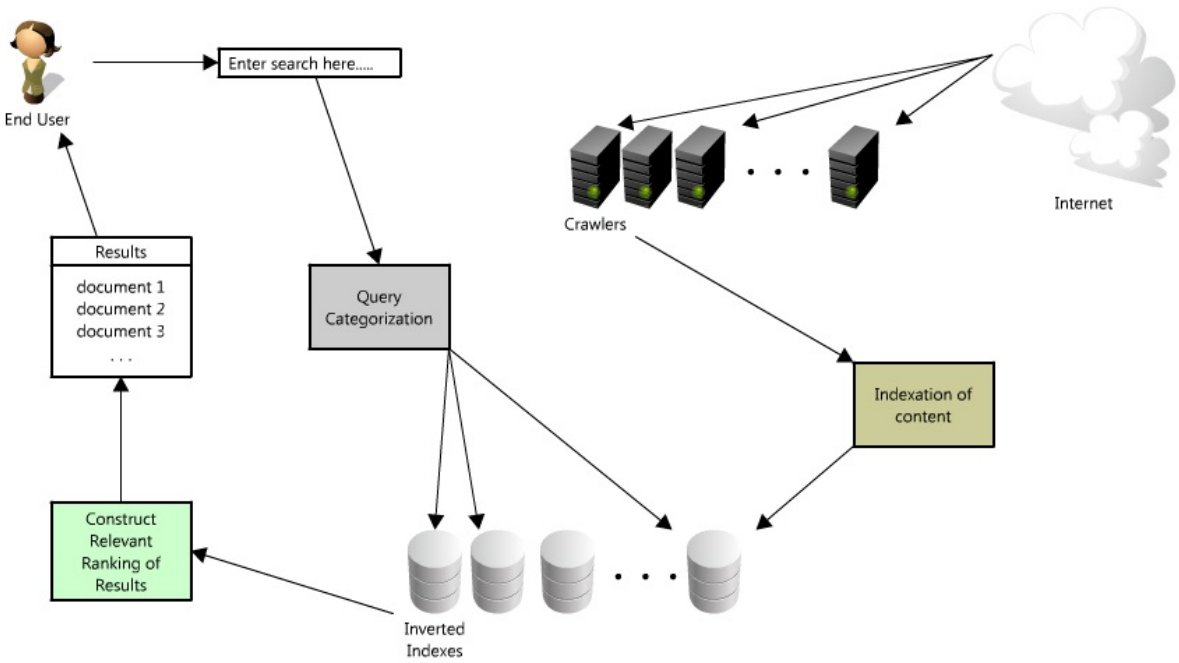
除此，我们可以分析用户的行为，如他们访问的主要网站、URL等等。

SEO

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

搜索时发生什么了？

用户输入查询内容
查询处理以及分词技术
确定搜索意图及返回相关、新鲜的内容



search-engine-arch

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。*

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

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

什么样的网站需要SEO?

下图是我的博客的流量来源

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

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)

Hadoop分析数据

Hadoop是一个由Apache基金会所开发的分布式系统基础架构。

用户可以在不了解分布式底层细节的情况下，开发分布式程序。充分利用集群的威力进行高速运算和存储。

Hadoop实现了一个分布式文件系统（Hadoop Distributed File System），简称HDFS。HDFS有高容错性的特点，并且设计用来部署在低廉的（low-cost）硬件上；而且它提供高吞吐量（high throughput）来访问应用程序的数据，适合那些有着超大数据集（large data set）的应用程序。HDFS放宽了（relax）POSIX的要求，可以以流的形式访问（streaming access）文件系统中的数据。

Hadoop的框架最核心的设计就是：HDFS和MapReduce。HDFS为海量的数据提供了存储，则MapReduce为海量的数据提供了计算。

UX

用户体验设计（英语：User Experience Design），是以用户为中心的一种设计手段，以用户需求为目标而进行的设计。设计过程注重以用户为中心，用户体验的概念从开发的最早期就开始进入整个流程，并贯穿始终。其目的就是保证：

对用户体验有正确的预估
认识用户的真实期望和目的
在功能核心还能够以低廉成本加以修改的时候对设计进行修正
保证功能核心同人机界面之间的协调工作，减少BUG。
关于UX的定义我觉得在知乎上的回答似乎太简单了，于是在网上寻寻觅觅终于找到了一个比较接近于答案的回答。原文是在:Defining UX，这又是一篇不是翻译的翻译。

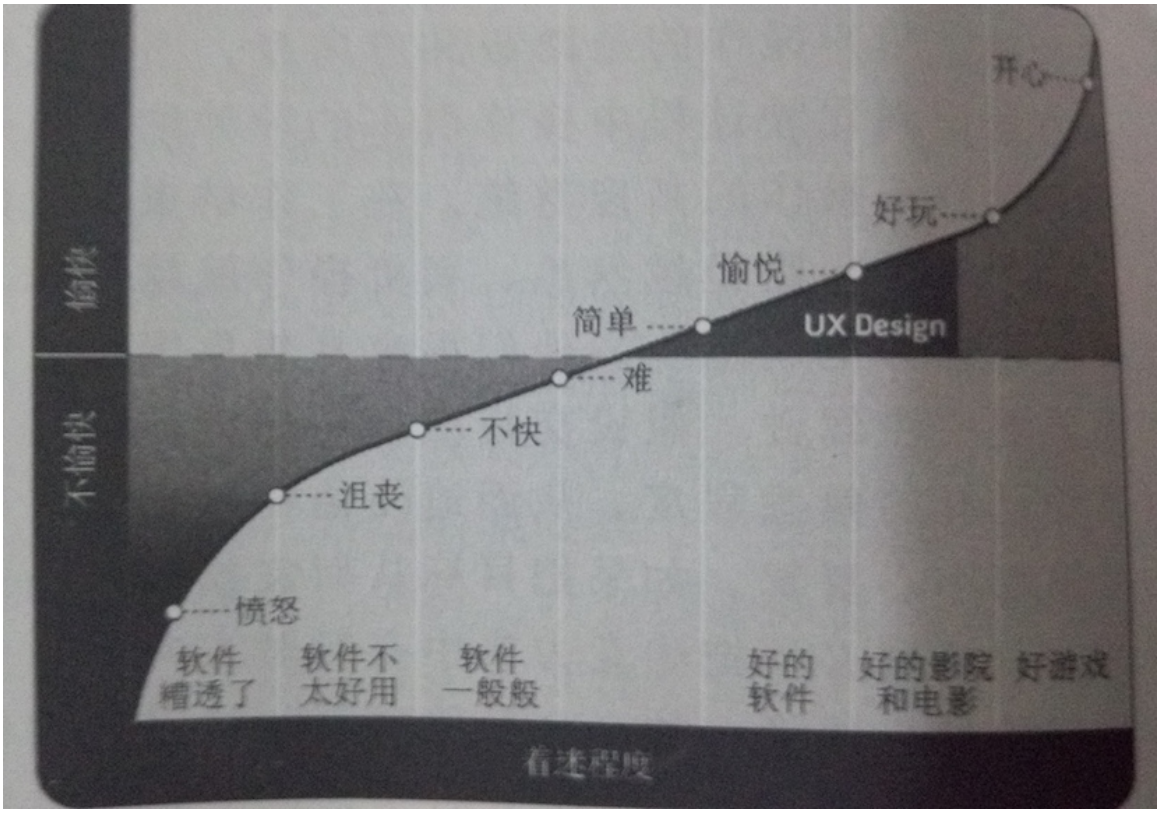
什么是UX

用户体验设计（英语：User Experience Design），是以用户为中心的一种设计手段，以用户需求为目标而进行的设计。设计过程注重以用户为中心，用户体验的概念从开发的最早期就开始进入整个流程，并贯穿始终。其目的就是保证：

对用户体验有正确的预估
认识用户的真实期望和目的
在功能核心还能够以低廉成本加以修改的时候对设计进行修正
保证功能核心同人机界面之间的协调工作，减少BUG。

UX需要什么

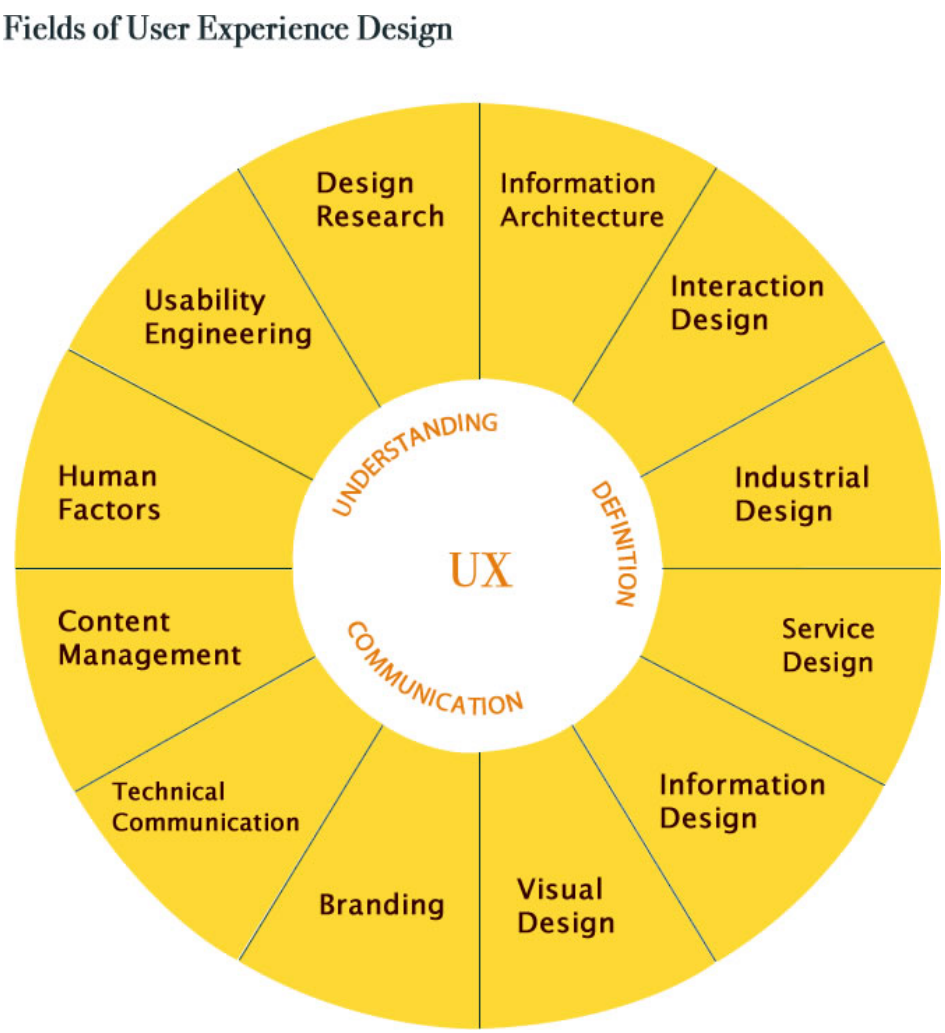
上传一张来自知乎上回答的答案，所需要的东西也差不多是这样子的。



UX

Information Architecture
Architecture
Industrial Design
Human Factors
Sound Design
Human-Computer Interaction
Visual Design
Content(Text,Video,Sound)
即

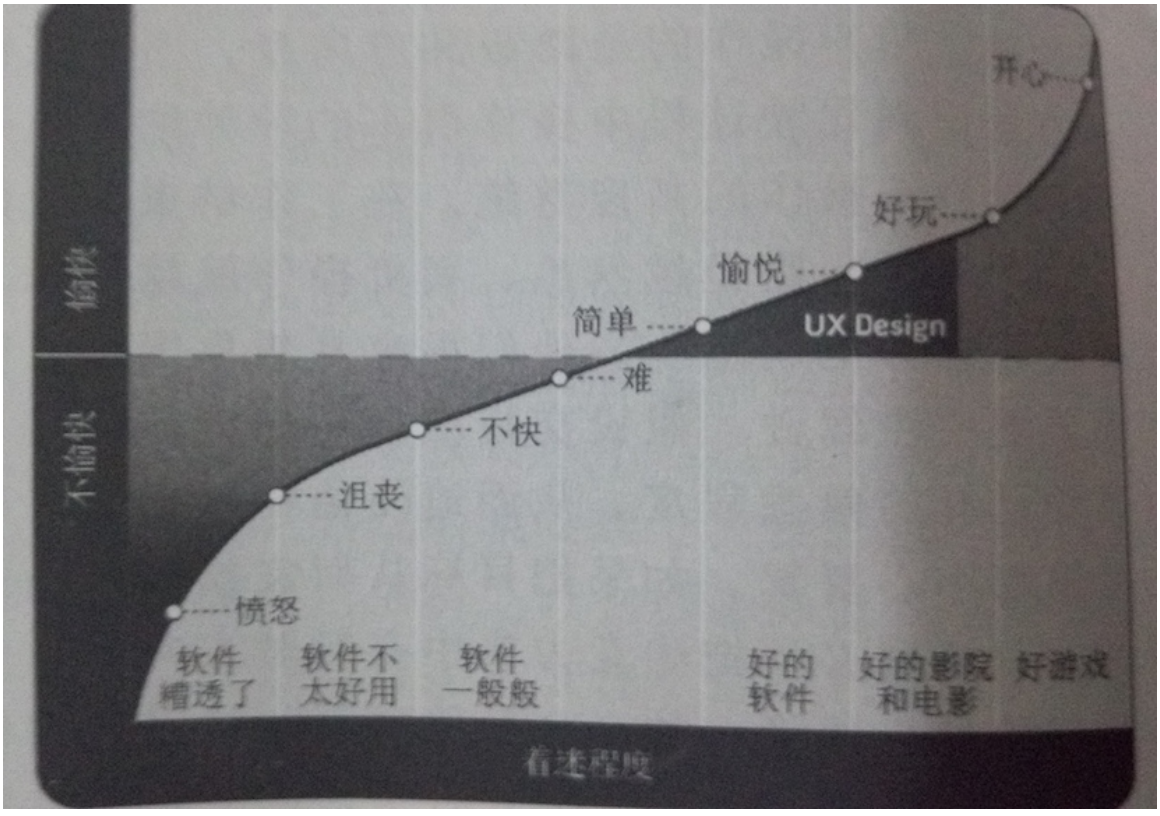
信息构架
构架
工业设计
人为因素 (人因学)
声音设计 (网页设计中比较少)
人机交互
可视化设计
内容 (文字,视频,声音)
交互设计便是用户体验设计的重点。我们再来看看另外的这张图片



Fields Of User Experience Design

UX入门

先让我们来看看用户体验的目的是？



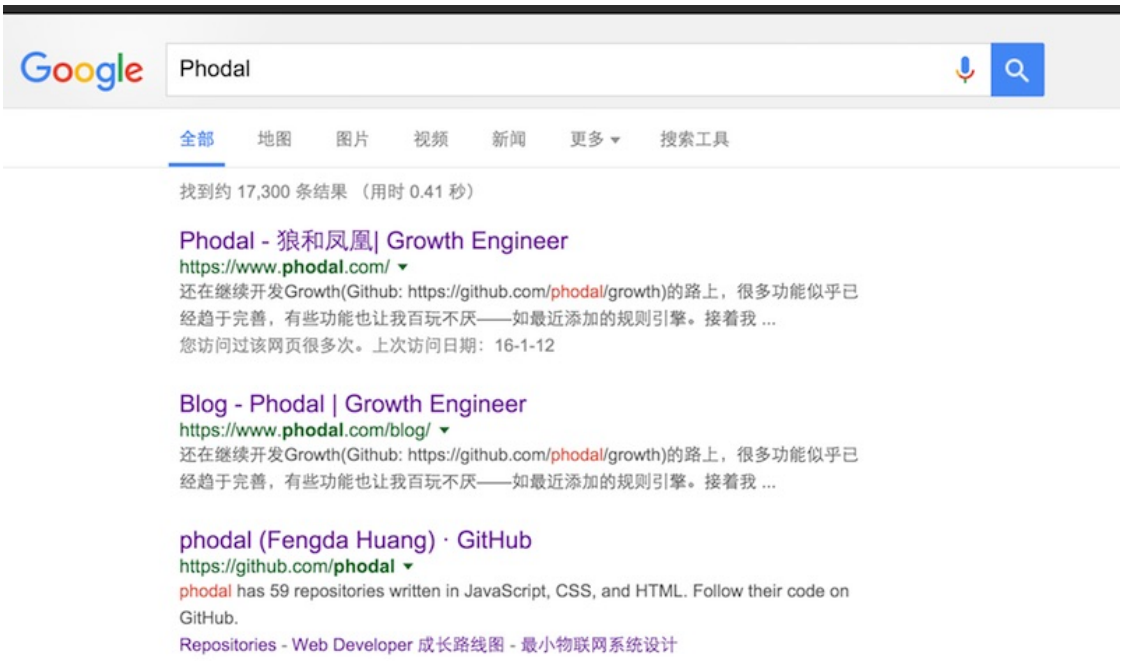
UX Design

图用手机拍得很渣，但是也能看清楚中间的部分——UX Design。从图中可以看到，UX Design的区域很大，而也似乎也决定了一个软件的好坏。
如果我们能将软件的易用程度推向简单的程度，那么我想我要的结果就是这个。

一个好的软件应该是简单的，并且是令人愉快的。

什么是简单？

在不同的UX书籍里，似乎就会说到【简约至上】。简单就是“单纯清楚、不复杂”。而这里的简单并不仅仅只是不复杂那么简单。对于一个软件来说，简单实际上是你一下子就能找到你想要的东西，如：



Search Phodal

而我们并不可能在一开始就得到这样的结果，这需要一个复杂的过程。而在这个过程开始之前，我们一定要知道的一点是：我们的用户到底需要什么？

如果我们无法知道这一点，而只是一直在假想客户需要什么，那么这会变成一条死路。

接着在找寻的过程中，发现了一个有意思的图，即精益画布：

Lean

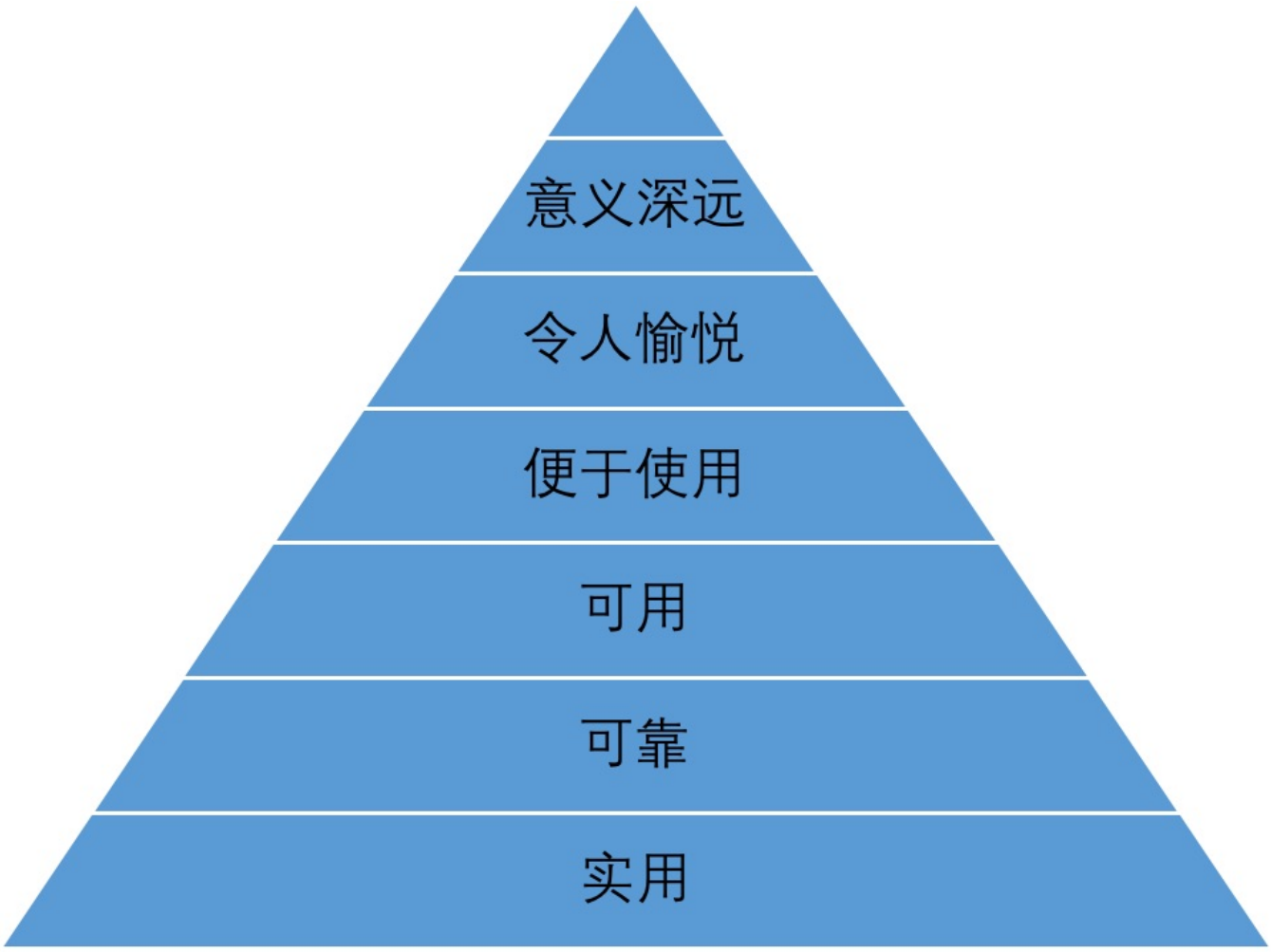
首先，我们需要知道几个用户而存在的问题——即客户最需要解决的三个问题。并且针对三个问题提出对应的解决方案，这也就是产品的主要功能。

那么，这两点结合起来对于用户来说就是简单——这个软件能解决客户存在的主要问题。

如果我们可以完成这部分功能的话，那么这就算得上是一个有用的软件。

进阶

而实际上有用则是位于用户体验的最底层，如下图所示：



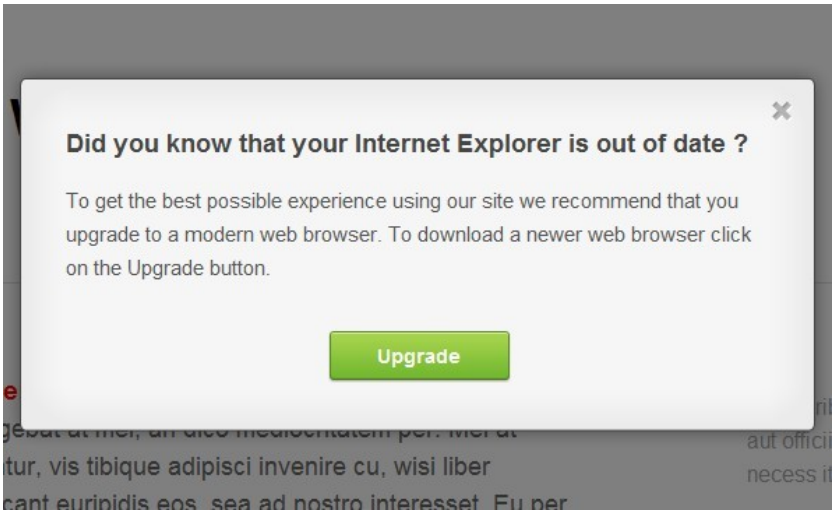
UX

这时候就需要尽量往可用靠拢。怎样对两者进行一个简单的划分？

下图就是实用的软件：

IE Alert

而下图就便好一点了：



jQuery Popup

要达到可用的级别，并不是一件困难的事：

遵循现有软件的模式。

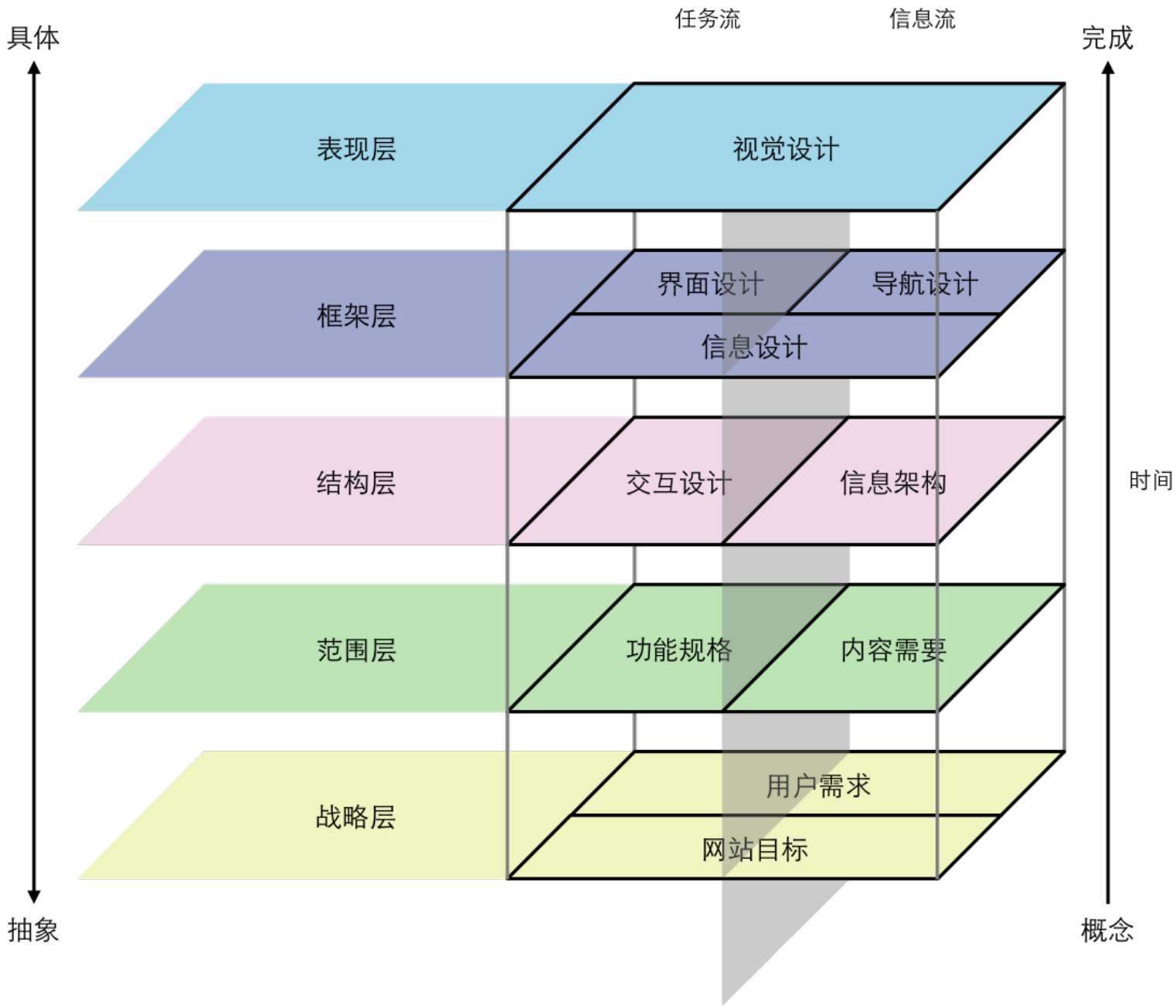
换句话说，这时候你需要的是一本**Cookbook**。这本**Cookbook**上面就会列出诸多现有的设计模式，只需要参考便使用就差不多了。

同样的，我便尝试用《移动应用UI设计模式》一本书对我原有的软件进行了一些设计，发现它真的可以达到这样的地步。

而这也类似于在编程中的设计模式，遵循模式可以创造出不错的软件。

用户体验要素

尽管对于这方面没有非常好的认识，但是还是让我们来看看我们现在可以到哪种程度。如在一开始所说的，我们需要满足用户的需求，这就是我们的目标：



用户体验要素

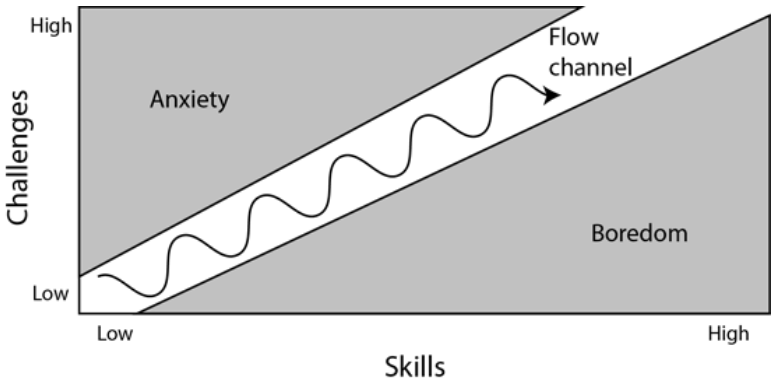
而在最上面的视觉设计则需要更专业的人来设计。

参考目录

《怦然心动——情感化设计交互指南》
《用户体验要素》
《移动应用UI设计模式》

认知设计

第一次意识到这本书很有用的时候，是我在策划一个视频。第二次，则是我在计划写一本书的时候。



持续交付

> *交付管道的建立和自动化是持续交付的基础*

持续集成

更关注代码质量。持续集成是为了确保随着需求变化而变化的代码，在实现功能的同时，质量不受影响。因此，在每一次构建后会运行单元测试，保证代码级的质量。单元测试会针对每一个特定的输入去判断和观察输出的结果，而单元测试的粒度则用来平衡持续集成的质量和速度。

瀑布流式开发

小步前进

自动化构建

自动化构建是一个很大、很广的领域，并且涉及到相当多的知识面。

几十年前，Unix世界就已经有了Make，而Java世界有Ant，Maven，现在则有Grunt和Gulp。

持续交付

自动化
DevOps
云基础设施
以软件为中心的哲学

自动化

DevOps

云基础

遗留系统与修改代码

尽管维基百科上对遗留系统的定义是:

> *一种旧的方法、旧的技术、旧的计算机系统或应用程序。*

但是实际上，当你看到某个网站宣称用新的框架来替换旧的框架的时候，你应该知晓他们原有的系统是遗留系统。人们已经不想在上面工作了，很多代码也不知道是干什么的，也没有人想去深究——毕竟不是自己的代码。判断是否是遗留代码的条件很简单，维护成本是否比开发成本高很多。

在维护这一类系统的过程中，我们可能会遇到一些原因来修改代码。如《修改代码的艺术》的一书中所说，修改软件有四大原因：

增加特性
修复Bug
改善设计
优化
当我们修改代码之后，我们将继续引进新的Bug。

参考阅读

-《修改代码的艺术》

遗留代码

我们生活息息相关的很多软件里满是错误、脆弱，并且难以扩展，这就是我们说的“遗留代码”。

什么是遗留代码

什么是遗留代码？没有自动化测试的代码就是遗留代码，不管它是十年前写的，还是昨天写的。

遗留代码的问题

如何修改代码

> *即使是最训练有素的开发团队，也不能保证始终编写出清晰高效的代码。*

然而，如果我们不去尝试做一些改变，这些代码就会遗留下去——成为遗留代码，再次重构掉。即使说，重构系统是不可避免的一个过程，但是在这个过程中要是能抽象中领域特定的代码、语言也是件不错的事。

So，如何开始修改代码？

测试
重构
修改测试
再次重构
在有测试的情况下重构现有的代码才是安全的。而这些测试用例也是功能的体现，功能首先要得到保证了，然后才能保证一切都可以正常。

。。。

测试

重构

修改测试

再次重构

网站重构

> *应包含结构、行为、表现三层次的分离以及优化，行内分工优化，以及以技术与数据、人文为主导的交互优化等。*

速度优化

通常来说对于速度的优化也包含在重构中

压缩JS、CSS、image等前端资源(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

功能加强

可以应用的的方面

解耦复杂的模块 -> 微服务
对缓存进行优化
针对于内容创建或预留API
需要添加新的API
用新的语言、框架代替旧的框架(如Scala,Node.js,React)

模块重构

深层次的网站重构应该考虑的方面

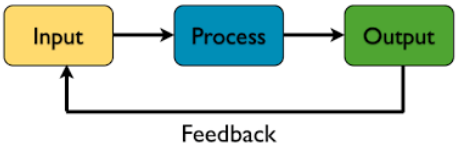
减少代码间的耦合
让代码保持弹性
严格按规范编写代码
设计可扩展的API
代替旧有的框架、语言
增强用户体验

回顾与新架构

在我开始接触架构设计的时候，我对于这个知识点觉得很奇怪。因为架构设计看上去是一个很复杂的话题，然而他是属于设计的一部分。如果你懂得什么是美、什么是丑，那么我想你也是懂得设计的。而设计是一件很有意思的事——刚开始写字时，我们被要求去临摹别人的字体。

自省

总结在某种意义上相当于自己对自己的反馈：



Output is Input

当我们向自己输入更多反馈的时候，我们就可以更好地调整我们的方向。

总结本身属于输出的一部分，而我们也在不断调整我们的输入的时候，我们也在导向更好地输出。

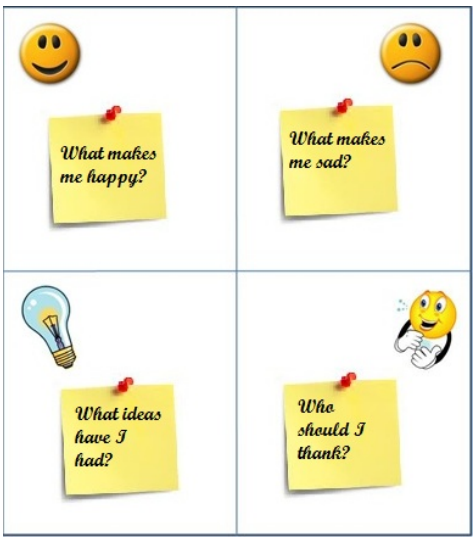
Retro

> *Retro 的目的是对团队的激励。*

> *Retro的模式的特点就是让我们更关注于Less Well。定期，经常，回顾，反思。当我们无法变得更好的时候可以帮助我们反观团队自身，不要变得更差。让破窗效应难以发生。*

Retro四个维度:

Well
Less Well
Suggestion
Action
该模式的特点是会让我们更多的关注less well，关注我们做的不好的那些。



Retro

Well

Less Well

Suggestion

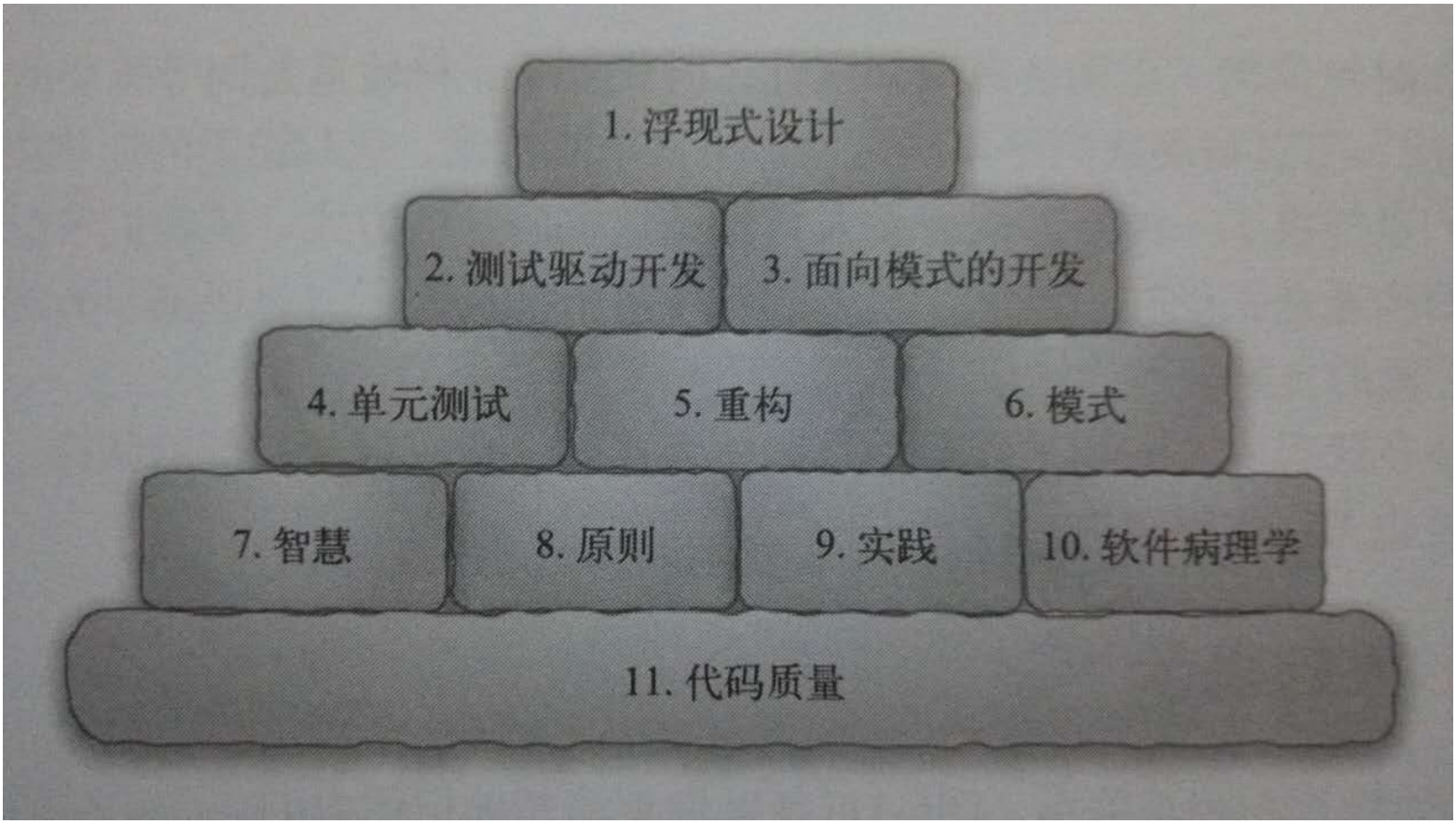
Action

浮现式设计

设计模式不是一开始就有的，好的软件也不是一开始就设计成现在这样的，好的设计亦是如此。

导致我们重构现有系统的原因有很多，但是多数是因为原来的代码变得越来越不可读，并且重构的风险太大了。在实现业务逻辑的时候，我们快速地用代码实现，没有测试，没有好的设计。

而下图算是最近两年来想要的一个答案：



浮现式设计

浮现式设计是一种敏捷技术，强调在开发过程中不断演进。软件本身就不应该是一开始就设计好的，他需要经历一个演化的过程。

意图导向

就和Growth一样在最开始的时候，我不知道我想要的是怎样的——我只有一个想法以及一些相对应的实践。接着我便动手开始做了，这是我的风格。不得不说这是结果导向编程，也是大部分软件开发采用的方法。

所以在一开始的时候，我们就有了下面的代码：

```
if (rating) {
    $scope.showSkillMap = true;
    skillFlareChild[skill.text] = [rating];

    $scope.ratings = $scope.ratings + rating;
    if (rating >= 0) {
      $scope.learnedSkills.push({
        skill: skill.text,
        rating: rating
      });
    }

    if ($scope.ratings > 250) {
      $scope.isInfinite = true;
    }
  }
```

代码在不经意间充斥着各种Code Smell:

Magic Number
超长的类
等等

重构

还好我们在一开始的时候写了一些测试，这让我们可以有足够的可能性来重构代码，而使得其不至于变成遗留代码。而这也是我们推崇的一些基本实践：

> *红 -> 绿 -> 重构*

测试是系统不至于腐烂的一个后勤保障，除此我们还需要保持对于Code Smell的嗅觉。如上代码：

```
if ($scope.ratings > 250) {
  $scope.isInfinite = true;
}
```

上面代码中的“250”指的到底是？这样的数字怎么能保证别人一看代码就知道250到底是什么？

如下的代码就好一些：

```
var MAX_SKILL_POINTS = 250;
if ($scope.ratings > MAX_SKILL_POINTS) {
  $scope.isInfinite = true;
}
```

而在最开始的时候我们想不到这样的结果。最初我们的第一直觉都是一样的，然而只要我们保持着对Code Smell的警惕，情况就会发生更多的变化。

重构是区分普通程序员和专业程序员的一个门槛，而这也是练习得来的一个结果。

模式与演进

如果你还懂得一些设计模式，那么想来，软件开发这件事就变得非常简单——我们只需要理解好需求即可。

从一开始就使用模式，要么你是专家，要么你是在自寻苦恼。模式更多的是一些实现的总结，对于多数的实现来说，他们有着诸多的相似之处，他们可以使用相同的模式。

而在需求变化的过程中，一个设计的模式本身也是在不断的改变。如果我们还固执于原有的模式，那么我们就会犯下一个又一个的错误。

在适当的时候改变原有的模式，进行一些演进变显得更有意义一些。如果我们不能在适当的时候引进一些新的技术来，那么旧有的技术就会不断累积。这些技术债就会不断往下叠加，那么这个系统将会接近于崩塌。而我们在一开始所设定的一些业务逻辑，也会随着系统而逝去，这个公司似乎也要到尽头了。

而如果我们可以不断地演进系统——抽象服务、拆分模块等等。业务在技术不断演进地过程中，得以保留下来。

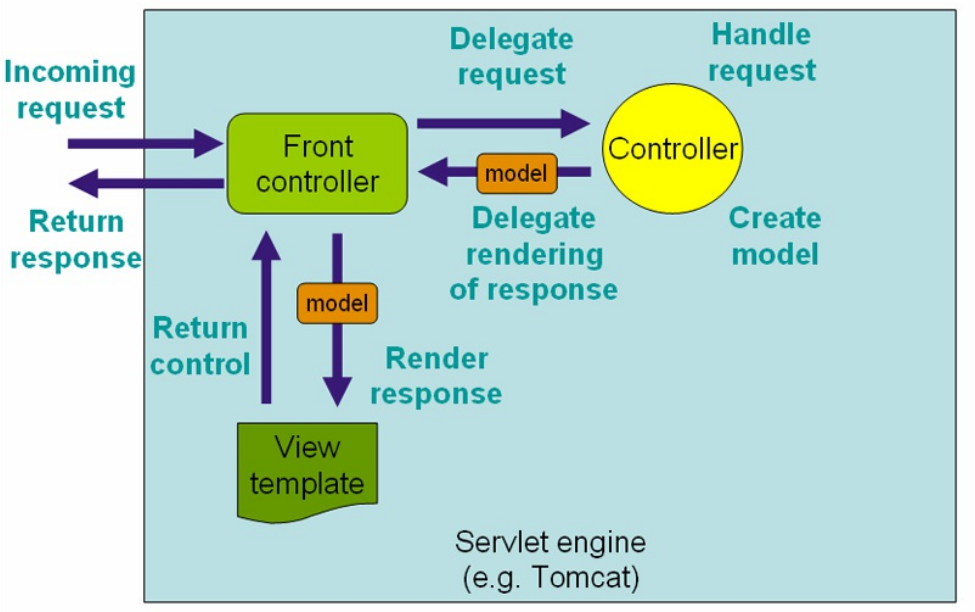
架构模式

> *模式就是最好的架构。*

架构的产生

在我开始接触架构设计的时候，我买了几本书然后我就开始学习了。我发现在这些书中都出现了一些相似的东西，如基本的分层设计、Pipe and Filters模式、MVC模式。然后，我开始意料到这些模式本身就是最好的架构。

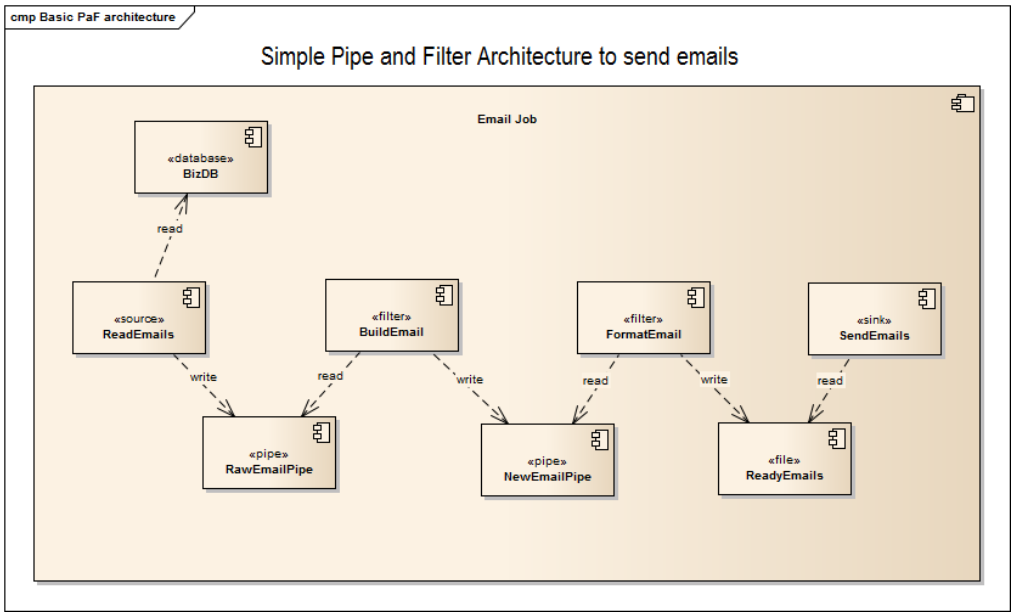
MVC模式本身也是接于分层而设计的，如下图是Spring MVC的请求处理过程：



Spring MVC

而这个框架只是框架本身的架构，这一类也是我们预先设计好的框架。

在框架之上，我们会有自己本身的业务所带来的模式。如下图是我的网上搜罗到的一个简单的发送邮件的架构：

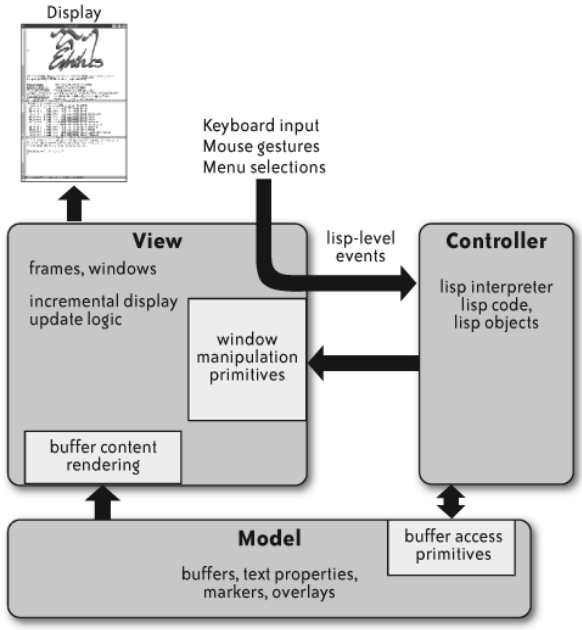


发送邮件中的Pipe and Filters模式

这样的模式则是由业务发展的过程中演进出来的。

预设计式架构

在我们日常使用的框架多数是预先设计的构架，因为这个架构本身的目标是明确的。系统会围绕一定的架构去构建，并且在这个过程中架构会帮助我们更好地理解系统。如下图所示的是Emacs的架构：



Emcas架构

它采用的是交互式应用程序员应用广泛的模型-视图-控制器模式。

演进式架构

演进式架构则是我们日常工作的业务代码库演进出来的。由于业务本身在不断发展，我们不断地演进系统的架构。

每个人都是架构师——如何设计一个博客系统

每一个程序员都是架构师。平时在我们工作的时候，架构师这个Title都被那些非常有经历的开发人员占据着。然而，如果你喜欢刷刷Github，喜欢做一些有意思的东西，那么你也将是一个架构师。

如何构建一个博客系统

如果你需要帮人搭建一个博客你先会想到什么？

先问一个问题，如果要让你搭建一个博客你会想到什么技术解决方案？

静态博客（类似于GitHub Page）
动态博客（可以在线更新，如WordPress）
半动态的静态博客（可以动态更新，但是依赖于后台构建系统）
使用第三方博客
这只是基本的骨架。因此如果只有这点需求，我们无法规划出整体的方案。现在我们又多了一点需求，我们要求是独立的博客，这样我们就把第4个方案去掉了。但是就现在的过程来说，我们还是有三个方案。

接着，我们就需要看看Ta需要怎样的博客，以及他有怎样的更新频率？以及他所能接受的价格？

先说说介格——从价格上来说，静态博客是最便宜的，可以使用AWS S3或者国内的云存储等等。从费用上来说，一个月只需要几块钱，并且快速稳定，可以接受大量的流量访问。而动态博客就贵了很多倍——我们需要一直开着这个服务器，并且如果用户的数量比较大，我们就需要考虑使用缓存。用户数量再增加，我们就需要更多地服务器了。而对于半动态的静态博客来说，需要有一个Hook检测文章的修改，这样的Hook可以是一个客户端。当修改发生的时候，运行服务器，随后生成静态网页。最后，这个网页接部署到静态服务器上。

从操作难度上来说，动态博客是最简单的，静态博客紧随其后，半动态的静态博客是最难的。

整的性价比考虑如下：

x	动态博客	静态博客	半动态的静态博客
价格	几十到几百元	几元	依赖于更新频率 几元~几十元
难度	容易	稍有难度	难度稍大
运维	不容易	容易	容易
数据存储	数据库	无	基于git的数据库

现在，我们已经达到了一定的共识。现在，我们已经有了几个方案可以提用户选择。而这时，我们并不了解进一下的需求，只能等下面的结果。

客户需要可以看到文章的修改变化，这时就去除了静态博客。现在还有第1和第3种方案可以选，考虑到第3种方案实现难度比较大，不易短期内实现。并且第3种方案可以依赖于第1种方案，就采取了动态博客的方案。

但是，问题实现上才刚刚开始。

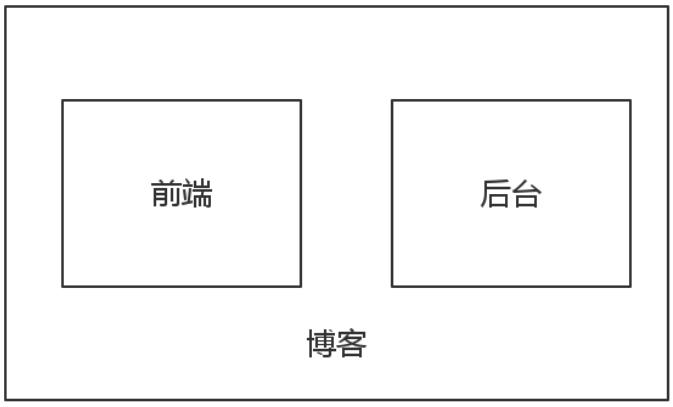
我们使用怎样的技术？

作为一个团队，我们需要优先考虑这个问题。使用怎样的技术解决方案？而这是一个更复杂的问题，这取决于我们团队的技术组成，以及未来的团队组成。

如果在现有的系统中，我们使用的是Java语言。并不意味着，每个人都喜欢使用Java语言。因为随着团队的变动，做这个技术决定的那些人有可能已经不在这个团队里。并且即使那些人还在，也不意味着我们喜欢在未来使用这个语言。当时的技术决策都是在当时的环境下产生的，在现在看来很扯的技术决策，有可能在当时是最好的技术决策。

对于一个优秀的团队来说，不存在一个人对所有的技术栈都擅长的情况——除非这个团队所从事的范围比较小。在一个复杂的系统里，每个人都负责系统的相应的一部分。尽管到目前为止并没有好的机会去构建自己的团队，但是也希望总有一天有这样的机会。在这样的团队里，只需要有一个人负责整个系统的架构。其中的人可以在自己擅长的层级里构建自己的架构。因此，让我们再回到我们的博客中去，现在我们已经决定使用动态的博客。然后呢？

作为一个博客我们至少有前后台，这样我们可能就需要两个开发人员。

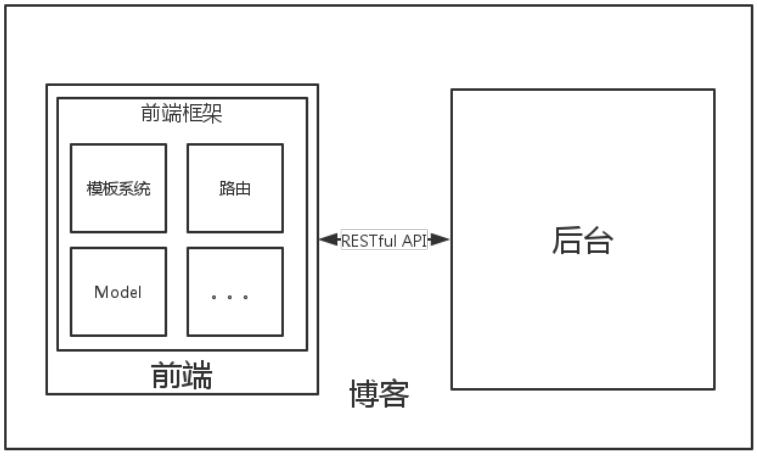


前后台

（PS：当然，我们也可以使用React，但是在这里先让我们忽略掉这个框架，紧耦合会削弱系统的健壮性。）

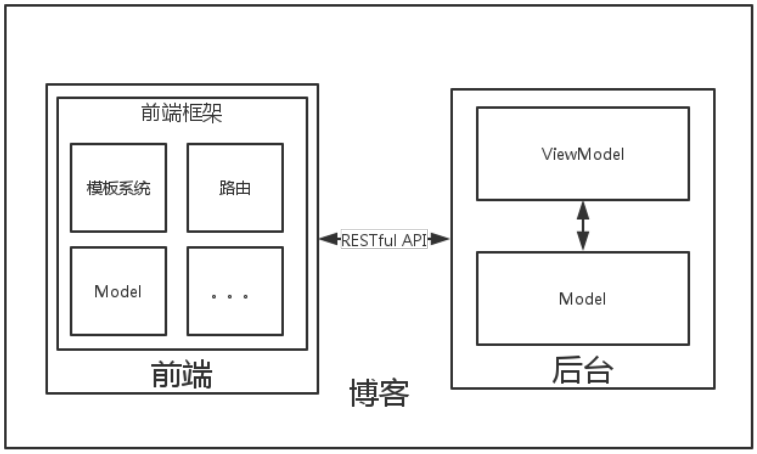
接着，作为一个前端开发人员，我们还需要考虑的两个问题是：

我们的博客系统是否是单页面应用？。
要不要做成响应式设计。
第二个问题不需要和后台开发人员做沟通就可以做决定了。而第一个问题，我们则需要和后台开发人员做决定。单页面应用的天然优势就是：由于系统本身是解耦的，他与后台模板系统脱离。这样在我们更换前端或者后台的时候，我们都不需要去考虑使用何种技术——因为我们使用API作为接口。现在，我们决定做成单页面应用，那么我们就需要定义一个API。而在这时，我们就可以决定在前台使用何种框架：Angular.js、Backbone、Vue.js、jQuery，接着我们的架构可以进一步完善：



含前端的架构

在这时，后台人员也可以自由地选择自己的框架、语言。后台开发人员只需要关注于生成一个RESTful API即可，而他也需要一个好的Model层来与数据库交付。



含前端后台的架构

现在，我们似乎已经完成了大部分的工作？我们还需要：

部署到何处操作系统
使用何处数据库
如何部署
如何去分析数据
如何做测试
。。。
相信看完之前的章节，你也有了一定的经验了，你也可以成为一个架构师了。

相关阅读资料

-《程序员必读之软件架构》