The Unbearable Madness of Static Blog Generators

http://freemind.pluskid.org/technology/the-unbearable-madness-of-static-blog-generators

posted on Free Mind on June 6, 2012
generated with pandoc on December 3, 2015
category: Technology

tags: Webdev, Tools

或许我就是一个文艺青年，连个吐槽帖子的标题都要想半天然后模仿一下 The Unbearable Lightness of Being，或者至少不是文艺青年也是个可怜的完美主义者：所以才会写出这篇文章来呀。完美主义者很可怜的原因是世间万物都本是不完美的，于是完美主义者就会在永远地最求完美的过程中不断受到打击和挫折。这基本上就是我这段世间尝试各种 static blog generator 的感想了吧！

开始尝试 static blog generator 的理由理所当然地是：原来的 Wordpress 平台的 blog 系统并不是完美的。要数可以数出很多来，最不方便的就是写日志了，由于我的 blog 是放在 dreamhost 上的，从国内连接速度不是很快，而像我这种写一段就喜欢刷新看一下效果（特别是在文章中很多公式的时候）的人来说，基本上大部分时间都花在等待浏览器刷新上了。同样的原因插入图片什么之类的附件也很不方便。此外自己需要的一些扩展功能不太好实现，例如在这篇关于 PAC Learnability 的文章中出现的定义那个 block，还有类似的定理呀、证明呀之类的，我在书写的时候都要写比较复杂的 HTML 代码，如果要自动化的话似乎又得用 PHP 写 Wordpress 插件，又兴师动众了。等等等等，如此这般，反正欲加之罪，何患无辞，于是我就开始尝试 static blog generator 了。

实际上要说静态网页生成的话，我很早就开始尝试了，最开始是为了给自己搞一个主页，出国申请用。当时找到了一个叫做 bonsai 的 Ruby 的简易网页生成工具，现在看来确实很简易，不过那个时候我要求很低，也没见过世面，就用上了。用的过程中发现了一个小 bug：他文档中说子页面的顺序会按照文件名顺序来排，但是实际在代码中根本没有排序，于是我直接改了代码，在某个地方加了一句：

```
1 .sort_by{|p| p.disk_path }
```

（偷偷说其实我在测试新 blog 的代码高亮功能）然后顺便去报告了 bug：首先感谢了作者提供这么好用的工具，然后委婉地说那个地方好像和文档中说的有点不符合，修正如下。作者很快回复说欢迎提交 patch (with a test)。我黑线了一下..... `-_-!!!` 后来作者又说他发现是和文档不太符合，这里应该是返回文件系统默认的顺序，所以我们这里需要把文档修正一下，如果你有时间的话，欢迎提交 patch 来修正文档。此时此刻我心理只想着逃命要紧了！`^_^bb` 虽然这个作者宁愿修改文档都不修改代码让我觉得有点汗，但是他抓人的行为实在是很有既视感：有时候有人问起你的某个 N 久没有更新或维护过的开源项目的时候，你就会巴不得他把一切都搞好比如在 github 上提交个 pull request，你只要点一下 accept 就好了，或者甚至希望他能直接 take over 项目将来的维护.....哈

哈!

第一次 static site generator 之旅就这样结束了，bonsai 果然还只是小工具，虽然能生成网站，但是开发起来不能方便地预览。后来过了很久我有空了，又把自己的主页重新做了一下，主要的改动是使用 twitter-bootstrap 来作为基础 css 让页面更好看一点。顺便呢，把那个 bonsai 给替换掉。

经过一番调研我最后选择了 Hyde 这个用 Python 写的工具。看起来很 promising，似乎也有不少好评。也确实比 bonsai 要强大许多，比如可以直接启动一个 web server 在本地预览呀之类的。不足的地方就是好像没有文档..... =.=，不过最后我也不知道怎么 figure out 了，把主页改造了一番，还用上了一些比较 fancy 的 twitter-bootstrap 里的效果。

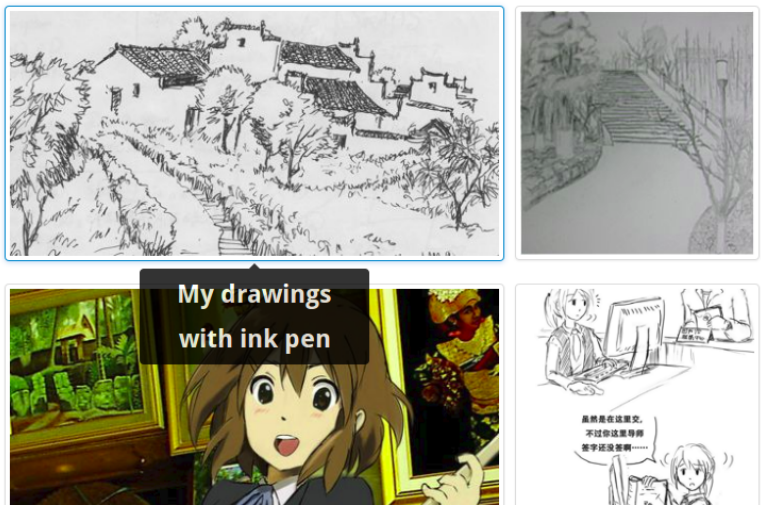

Figure 1: .....

Hyde 的 template 是用的 Jinja，似乎是很受推崇的。发现 Python 派系的 template 和 Ruby 派系有一个比较明显的区别就是：比如 Jinja2 里，在不同的 template 间共享代码的方式是通过继承。例如，你定义一个基本的框架（一个非常简化的例子）：

```
<!DOCTYPE html>
<html lang="en">
  <head>
    {% block head %}
    <meta charset="utf-8">
    {% endblock head %}
  </head>
  <body>
  {% block body %}
  {% endblock body %}
  </body>
```

```
</html>
```

然后在更具体的 template 中，你可以继承这个 template ，并重载其中的某些 block ，例如在 head block 中加入 favicon：

```
{% extends "base.j2" %}

{% block head %}
  {{ super() }}
  <link rel="shortcut icon" href="/favicon.png">
{% endblock head %}
```

就如同 OOP 中的继承一样，还可以调用 super 。不过在 Ruby 社区似乎主要是用另一种方式：他们会把每个不同的功能组件分割开成为一些“partial”，例如 head 、sidebar、navbar 之类的，然后在需要的页面，通过 include 这些 partial 的方式来组装成一个完整的页面 template 。

不过后来开始考虑 blog 也用 static site generator 来搞的时候，我又尝试了 Ruby 派系的，毕竟写网站似乎从 Rails 开始就是用 Ruby 的工具链显得“洒脱”一些。特别是 jekyll 似乎是非常有人气的。不过由于 jekyll 虽然号称是“blog-aware”的 static site generator，但是本身还是一个比较底层的工具，要从头搭建一个 blog 的话，还是比较麻烦的，由于之前对 twitter-boostrap 的好感，自然就找到了 jekyll-bootstrap，而且 jekyll 本身的文档也非常不友好，只有一些零星的 wiki page，虽然作为已经熟悉了系统之后的 reference manual 来用还是挺不错的，但是连一个 introductor 或者 tutorial 都没有的话，对于外行就很不友好了，不过还好 jekyll-bootstrap 那里提供了一个很不错的 jekyll 的 tutorial。

这里可以简单讲一下 jekyll 之类的 blog generator 的工作方式。一个 blog 中的页面通常由两种组成：page 和 post 。page 是比较固定的页面，比如主页呀、About 页面之类的，而 post 则是正常的 blog 了。page 页一般可以比较简单，就将 template 展开成 HTML 即可；而 post 则有一些额外的套路：

- post 通常都是统一 layout 的，因此通常被分为 layout 和 content 两个部分，layout 是普通的 template，里面指定了一个地方放置 content，每一篇或者至少每一类 blog 都是相同的；而 content 则是 blog post 的具体内容，一般用 markdown 或者 textile 之类的标记语言写成。
- post 有一些 meta data，包括发布日期、tags 和 category 等。系统在处理的时候会把这些信息收集起来，比如发布日期会用来对 post 进行排序，tag 和 category 会生成反向映射（例如用于找出某个 tag 对应了哪些 post 等等）。

将 post 生成为 HTML 页面的时候粗略地来讲可以分为以下几个步骤：

1. **收集 payload**：包括整个 site 的信息例如有哪些 tag 、有哪些 post 之

类的；和 post 自己的信息，例如标题、发布日期等。这些 payload 信息可以在 template 中访问到。
2. **渲染内容**：jekyll 的 post 内容是标记语言和模版语言混合的。这样做的好处是添加了灵活性，因为像 markdown 和 textile 之类的标记语言毕竟语法是固定的，如果想要添加什么扩展功能的话，还是添加 template 来得方便一些，例如要添加语法高亮功能，或者是嵌入 video 之类的。
3. **展开模版**：最后 post 对应的 layout 模版会被展开，上一步中生成出来的内容会被插入到模版所指定的位置上。

内容渲染的时候 jekyll 中是先用 Liquid 模版展开，然后把得到的结果传给标记语言的转换器进行转换。具体使用什么转换器依赖于 post 那个文件的扩展名，内置了 markdown 和 textile 等的转换器。这样一来添加的扩展功能就可以抢先被处理，当然由于生成出来的内容会经过（比如）markdown，所以又需要注意不要让你生成出来的内容被 markdown 破坏掉了。markdown 是不会去动 block HTML 里面的内容的，所以如果你生成出来是一个 block HTML 的话，一般没有什么问题，但是如果是嵌入在段落文本中的话就有些麻烦了，标准的 markdown 语法并没有什么有效的 escape 功能，不过如果使用 kramdown 来进行 markdown 转换的话，可以将对应的内容包装在 `{::nomakrdown}` 和 `{:/nomarkdown}` 之间。

实际上我最急需的扩展就是书写数学公式——更确切地说：方便地书写数学公式，我希望可以写 `$e^{i\pi}$` 就可以得到 $e^{i\pi}$，而不需要繁杂的 Liquid tag 语法。所以这个功能不能在 Liquid template 那里添加，不过由于渲染用的 converter 是可以定制的，所以我们可以定制一个 markdown converter，在传入真正的 markdown 渲染之前就把这些公式全部处理好——如果用 MathJax 在网页里动态解析的话，其实只有保持文本原样就可以了，因此这里主要就是将它保护起来，需要保护的原因是公式里的许多标记如上标、下标之类的都是和 markdown 冲突的。

不过后来我发现还有一个东西叫做 textile 之后，就立即放弃 markdown 了，主要原因是 textile 标准就包含了一个保护用的 `<notextile>` 语法。当然 textile 和 markdown 相比也更加强大，或者说目标不同吧：

- markdown 的目标是写出来的文档直接看它原来的纯文本格式也是结构清晰的，顺便还可以方便地转化为 HTML 之类的格式。
- textile 生来就是为了写可以转化为 HTML 的文档的，因此它功能更加强大一些，甚至可以直接控制生成出来的 HTML 元素的各种 property。

于是我果断放弃 markdown 换了 textile，并做了这样的定制 converter。不过最后我用的不是 jekyll-bootstrap，而是同一个作者新搞的一个工具叫做 ruhoh。这个东西比较新一点，据说是作者在开发 jekyll-bootstrap 的过程中总计经验的全新力作，当然是完全重写的，不过野心也比较大，是要做一个 universal static blogging platform，有一套 API，说是要做成 language agnostic 的，目前提供了 Ruby 的版本，好像后面要开发 Python

、PHP 和 node.js 等的版本。当然我对是否 language agnostic 不感兴趣，不过总得来说 ruhoh 还是很不错的。

先说缺点吧，第一就是由于它这个 language agnostic，所以用了一个 language agnostic 的 template 叫做 Mustache，这货语法巨简单，功能有点弱（当然功能弱并不是说它干不了有些事，只是干起来比较麻烦，夸张一点来说就像汇编语言和 Ruby 比较一样），什么 for 啊、if 之类的统统都没有，倒是有一个简单的 loop，可以重复一个列表里的元素进行展开，但是也不支持上下文判断是否是列表第一个、偶数个等功能。比如最简单的我有一个 tags 列表，想要让每个 tag 是到对应 tag 页面的链接然后把这些链接用逗号连接起来，就是除了最后一个 tag 之外其他每个后面都加一个逗号。这个在 Liquid 或者 Jinja2 之类的 template 语言里是很好写的，但是在 Mustache 里就变得非常痛苦。

另外一个缺点就是 ruhoh 还比较新，就在我弄好自己的 blog，并且认认真真写了一篇文章，表示很满意，准备 deploy 的时候，ruhoh 发布了新版本，在 announce blog 中第一句话是 “I Just Broke Everything” ..... `-.-bbb`

这些个缺点促使我还尝试了其他的一些工具，包括自己从裸的 jekyll 开始搭建，以及使用非常受欢迎的基于 jekyll 的 octopress 。总的来说令我非常崩溃.....

先说 jekyll，和 ruhoh 比起来，实在是比较原始。它虽然有一个 auto re-generation 功能，就是启动起来之后监视本地文件，如果改动了就自动重新生成，这样可以使得本地写 blog 预览更加方便一些，但是我从输出来看它似乎每次都把整个网站重新生成一遍，虽然我的 blog 目前就一篇文章和两三张图片，但是对于一个完美主义者来说这简直是不可忍受的。~~万一我以后写了数千万篇的文章了呢？~~并且它似乎把本地预览和 deploy 的时候所生成的文件混在一起用的，原本我在模版里区分了生成的时候的环境是 development 还是 production，如果是前者就载入本地另外用 lighttpd host 的 MathJaX，如果是后者则载入 MathJaX 官方的 CDN 的版本（MathJaX 发布的那个包里的文件实在是太乱了，完美主义者表示不愿意把那个包整个丢到自己的 blog 目录里去）。结果在用 jekyll 的时候就混乱了。其他的什么缺点一时也想不起来了，因为我已经一怒之下把自己折腾半天的那个目录给删掉了。

相比起来，ruhoh 的开发模式就友好得多，本地预览和 deploy 是分开的，并且本地预览是用 rack，我发现这个东西也挺好玩的。最开始是希望能找到一个功能是我在写 blog 的时候，文本编辑器这边一保存，浏览器那边自动就刷新。因为我现在在家里有双显示器，所以这个功能非常实用。Linux 下可以用 inotify 来跟踪文件的改变，可是发现变更之后怎么让浏览器刷新呢？找半天发现 Firefox、Chrome 之类的浏览器都不支持通过命令行发送刷新命令。可能的解决方案有：

- 使用浏览器 REPL，例如 mozrepl，给浏览器添加一个脚本解释器的 server，然后通过 netcat 等工具发送脚本指令到浏览器里去执行。这

样子有点感觉略杀鸡用牛刀。而且我随便尝试了一下没有成功，也没有深究了……

- 使用专用浏览器插件，例如 Firefox 的 Auto Reload 插件，就可以跟踪本地文件，在改变的时候自动刷新，看起来刚好就是为这个任务设计的，不过不知道为什么我就是不喜欢，-.-bb 大概是因为要跑到浏览器里去配置该归本地 blog 目录管的内容吧。而且似乎底下的评论有说新版本不 work 了，于是我就没有尝试。
- 使用 xdotool，人肉发送 `<F5>` 给浏览器。这个方案多少有些迂回，而且需要先把焦点切换到 Firefox，然后再切换回编辑器，反正用起来差强人意。

但是后来我发现一个叫做 rack-livereload 的东西，这货是工作在 server 端的，用法很简单，在 ruhoh 生成的 rack 用的配置 `config.ru` 里加一行变成这样

```
require 'rack'
require 'rack-livereload'
require 'ruhoh'

use Rack::LiveReload

run Ruhoh::Program.preview
```

然后写一个 `Guardfile`，里面写想要 watch 的文件（guard 还可以做很多其他事情，更多例子可以参考它的 readme）

```
guard 'livereload' do
  watch(%r{posts/.+\.textile})
end
```

之后就可以了，运行 `guard` 把文件 watch 起来，然后运行 `rackup` 把本地预览用的 web server 启动起来，然后你修改被 watch 到的文件的时候，浏览器里对应的页面就会自动刷新，神奇吧？它的工作原理其实也不神秘，就是通过一个 rack 的中间件，在最终发送给浏览器的 HTML 页里做了点手脚，插入了一个 livereload.js，这个脚本负责监听服务端的 reload 指令，而 web 服务端那里从 guard 服务器得知文件变动之后就会通知该脚本刷新。用起来很透明也很方便！^_^ 但是如果用 jekyll 的话大概就没法用这个功能了，至少我不知道怎么把它们黏合起来。

不过折腾的结果是对 jekyll 的内部有了更多的了解，于是也能看得懂它那些散布的文档了，顺便在看 octopress 的时候也明白了，其实它就是在 jekyll 的基础上加了一堆 plugin 。所以我也顺便尝试了一下 octopress 。总的来说完成度相当高了，基本直接就可以用，不过由于是基于 jekyll，所以 jekyll 有的缺点它都有。另外它默认的那个主题实在是让人又爱又恨，爱的是看起来确实挺 pp 的，恨的是看起来确实挺 pp 以至于网上已经有无数个基于 octopress 的 blog 直接用了默认主题，最多改了个颜色。

此外 Octopress 虽然默认添加了 SASS 支持，但是它那个 style 文件看起来好像很复杂的样子，不知道是它自己从零写出来的还是基于某个 css 框架搞出来的，要做一些扩展的时候又觉得无从下手了。由于我尝试 Octopress 的直接原因是出于对 ruhoh 的 Mustache 模版的不满，所以自然对 jekyll 的 Liquid 模版满心期待，但是最后也是让我很崩溃，反正我看文档（基本没有这部分的文档）加看代码加运行时调试，最后都没太搞明白在 Liquid 中定义的一个 block tag，它传进来的那些个参数到底是啥.....我定义了一个生成定理的 block，老是出错，后来我找到原因是因为 Octopress 的 RSS 生成插件那里用了非常规的处理方法，由于要将内容嵌入到 XML 中，它将每个 post 的原始内容拿过来直接做了些处理（比如不管三七二十一用 markdown 转换一下 `=.=bb`），然后做了 XML 的 escape，最后再模版自己本身展开的时候有些自定义的 tag 已经被搞得面目全非了，自然就解析出错了。

各种崩溃的最后我还是选择了崩溃点最少的 ruhoh，并更新到最新版。然后做了一些自己的定制和改动，比如可以用页内生成定理、公式编号和引用功能，实在是非常好用呀（比如参见关于 L1 sparsity 的那篇文章）。当然 static blog generator 也有它的许多缺点：

- **搜索**：静态网页是没法提供搜索功能的，最多是基于 javascript 的那种，感觉那种对于文档类的网站有特定的类名呀、函数名之类的需要检索的还有用一些，对于通用的 blog 就不太好用了。要么就只能用 google custom search 了，不过那样也得先等到 blog rank 足够高被 google 给好好索引了之后才会好用。但是其实我是几乎没有用过像 wordpress 之类的平台自己提供的搜索功能的，我想大家要搜索的时候也都是直接去 google 整个 Internet 搜吧，所以这一条不是特别 critical 。
- **评论**：静态网页没法支持评论，现在比较流行的解决办法就是使用 disqus 或者 intensedebate 之类的外部评论系统。关于外部评论系统的优缺点网上已经有很多分析讨论了，甚至还有针对这些系统开不开源的争执。不过我还是主要关心的是 anti-spam 的问题，用这些外部评论工具的话，可以有的控制就少了许多，我目前在 Wordpress 上用的 anti-spam 是要求评论中必须有中文字符出现，实践证明是非常有效

好用的手段，但是如果用 disqus 的话就不成了，不知道到时候体验会如何呢？

- **写作系统**：表面上看 static blog generator 写 blog 似乎变简单了，用一个文本工具就可以了，实际上却变复杂了许多，因为需要有一整套的软件装好了才能生成内容。由于我最近主要是在用 Linux，所以安装配置相关的工具都很轻松（顺便吐槽一下 Linux 下的各种中文输入法都好难用），但要是哪天我想在 windows 或者其他别人的电脑上写个 blog 的话，就痛苦了，相比之下，像 Wordpress 之类的平台就很清爽，只要有一个浏览器和 Internet access 就可以了。

总的来说其实就是一句话：世间没有完美的软件。真是个残酷的事实。唯一的生存之道就是把自己从一个完美主义者强制转换成一个将就主意者。否则走了极端就是大把大把的青春浪费到各种无意义的折腾上了 `=.=`。总之现在的系统先用用看吧。

最后讲一下 rake-pipeline 吧。各种 blog generator 会自动处理 post 的内容转换和渲染，但是有时候我们还有除了 post 之外的其他内容需要转换，例如可能用 SASS 写成的 styles 需要转换为 CSS，而我自己也有一个需求就是有时候画的图（比如用 asymptote ）需要从源码格式编译为图片格式等等。blog generator 一般没有处理这些东西的功能，一方面是因为已经有额外的程序可以比较好地处理这些了。当然用 Make 或者 Rake 手写也是可以的，不过这里我们可以用 rake-pipeline 会更方便一些。

rake-pipeline 的使用方法很简单，写一个 Assetfile （其实是一个 ruby 文件），在里面写规则，然后运行 `rakep build` 即可。例如下面是我用的 Assetfile，包含了编译 asy 文件的规则，由于 2D 的 asy 文件我希望编译为 svg 格式，而 3D 的编译成 png 格式比较好，所以我在 asy 源文件里用注释的方式加 meta 信息来表明输出格式：

```
# vim: filetype=ruby
output "compiled/files"

class AsyFilter < Rake::Pipeline::Filter
    attr_accessor :config

    def initialize
        @output_name_generator = proc { |fn, wrap|
            @config = {'ext' => 'png', 'opt' => ''}

            /^---$(.*?)^---$/m.match(wrap.read) do
                require 'yaml'
                config.merge!(YAML::load($1))
            end
            ext = config['ext']
            fn.sub(/\.[^.]+$/, '.' + ext)
```

```
        }
    end

    def generate_output(inputs, output)
        system("asy #{config['opt']} -f #{config['ext']} -o " +
               File.join(output.root,output.path) + " " +
               File.join(inputs[0].root, inputs[0].path))
    end
end

input "files" do
    match '**/*.asy' do
        filter AsyFilter
    end

    filter Rake::Pipeline::ConcatFilter
end
```

不过我这里由于决定输出文件的文件名的时候需要读取输入文件的内容，所以需要至少 `0.6.0` 版的 rake-pipeline 才好用（低版本的 `output_name_generator` 那个 lambda 的参数只有输入文件名，没有第二个 `wrap` 那个参数，当然在这种情况下要人肉把文件名和目录名合并起来找到原来的文件读取内容也不是不行的）。

为了避免每次都要人肉运行 `rakep`，我们可以写一个 compiler task plugin 来自动调用它。ruhoh 里的 compiler task plugin 就是每次生成 site 的时候都会被运行，里面基本上可以做任意的事情。这里我们很简单地直接调用命令即可，把这个文件放到 `plugins` 目录下就可以了。

```
class Ruhoh
  module Compiler
    module Assets
      def self.run(target, page)
        Ruhoh::Friend.say { green "building files" }
        system("rakep build")
      end
    end
  end
end
```

当然 jekyll 里也有 compiler task plugin，那里被称作 generator plugin 。不过我当时也算吃了点苦头，因为好像文件老是没法生成。最后我发现原来文件其实是生成了，只是 jekyll 管理比较严格，在生成 site 之后做了一些清理工作，把目标目录里未注册的文件全给删了。所以我们还得

多做一点事情，在生成文件之后给它们发放假户口。由于我所以生成的文件都放到 `files` 目录下，所以我就简单地为该目录下的每个文件都注册了一下：

```
module Jekyll
  class FakeStaticFile < StaticFile
    def initialize(site, base, dir, name)
      super
    end

    def write(dest)
    end
  end

  class AssetsGenerator < Generator
    safe true

    def generate(site)
      system("rakep build")

      # register generated files to prevent them from being deleted
      Dir.chdir(site.dest) do
        Dir['files/**/*'].each { |f|
          site.static_files <<
              FakeStaticFile.new(site, site.source, *File.split(f))
        }
      end
    end
  end
end
```

让每个文件都成为一个伪装的 `StaticFile`，然后重写 `write` 方法，让他什么都不干，就可以蒙混过关了，嘿嘿！