在模板引擎中强制模型-视图严格分离

被推荐为最佳论文

Terence Parr
旧金山大学

译注：部分翻译对照以及说明，参考文章最后的描述
Translated by: Richie

摘要：

每一位富有经验的 web 应用开发者都主张：应当将业务逻辑和显示分离。然而几乎所有的模板引擎都允许违背这一分离原则，这极大的推动了 HTML 模板引擎的发展。这种情况主要是由于对分离缺乏形式的定义，以及担心强制分离将消弱模板的生成能力（generational power）。我将阐述严格的分离是非常有用的设计原则，通过提供有效的模板引擎能够强制分离。我使用 StringTemplate 引擎作为示例，用来建立 jGuru.com 和其它的商业网站，并解决一些重要的生成问题。

我的目的是使对模板引擎的研究形式化，因此提供一个通用的术语，这个术语是一种对模板生成能力分类的方法，是对形式语言理论（formal language theory）感兴趣的补充。类似 Chomsky 的 1..3 型语法分类，我把限制性模板（restricted template）分成三种类型，并且形式的定义分离，包括实现分离的规则。

因为这篇论文提供了一个清晰的模型-视图分离定义，模板引擎设计者也许不再盲目的主张强制分离概念。此外，给出理论论点和经验证明，程序员不再有借口混杂（entangle）模型和视图。

分类和主题描述：

D.3.4 [程序语言]：处理机-代码生成；D2.11 [软件工程]：软件架构-面向领域的架构，模式，语言；D1.1 [编程技术]：应用（功能）编程

常规项：

语言

关键词：

模板引擎，Web应用，模型-视图-控制器

1. 简介

对动态生成web页面的需求，例如Amazon.com上的图书显示页面，导致众多模板引擎的开发趋向于使web应用开发简单化，提高扩展性，降低维护成本，允许编码和HTML的开发并行。这种诱人的好处促进了模板引擎的发展，这完全来自一个原则：将页面的业务逻辑声明（specification）和数据处理声明，与页面怎样显示这些信息的声明分离。通过隔离封装的声明，模板引擎促进了组件复用，可插拔的站点外观，通用组件的单点变化（single-points-of-change），良好的系统清晰度。

我曾经跟很多经验丰富的程序员讨论过分离原则，调查过很多通用的模板引擎，这些模板引擎使用各种语言开发，包括Java、C和Perl。毫不意外，程序员赞成逻辑与显示分离是一种理想的原则。然而实际上，程序员和引擎开发者不太原意强制分离，他们担心满足这个原则会损失生成能力，导致某个关键页面无法生成。相反，他们鼓励而不是强制要求这一原则，给自己留一个退路，避免不完善的页面生成能力。

很不幸，在最终期限的压力下，只要可能，程序员会经常使用这个退路作为权宜之计，导致逻辑和显示混杂。一个负责公司服务器数据模型的程序员告诉我，离最终期限他只有3天多时间，但如果强制他们的程序员采用分离原则，修改那些受影响的多语言页面显示需要10天。为将来的维护考虑做正确的事情，他可能面临被解雇的风险，或者他可以通过数据模型将新的HTML输出到页面，从而保住他的工作，将这种混乱留给以后或者其它程序员。

另一种更普遍的情形是作为一个捷径，程序员将业务逻辑嵌入他们的模板中，避免更新数据模型。提供完全图灵机式（Turing-Complete）的模板编程语言，程序员就趋向于直接在模板中需要的地方添加业务逻辑，而不是在数据模型中完成，从模型中解耦视图。例如几乎每个模板引擎的文档都有描述，如何根据用户权限改变显示，不是简单的向模型询问用户是否特殊，而是在模板中编码逻辑来确定用户是否特殊。如果这个特殊的定义改变了，可能系统中的每个模板都不得不修改，或者程序员可能忘了某个模板，给系统引入一个将来随时可能出现的缺陷。通常这种捷径很快导致完全混杂的声明（entangled specification）。

理想是否可能？就是说我们是否可以强制分离却又不消弱模板引擎的生成能力？限制性引擎导致一些

WWW2004, May 17–20, 2004, New York, New York, USA.
ACM 158113844X/04/0005.

页面无法处理？通常理论跟实践是不一致的。理论表明一个完全图灵机式的模板引擎能够生成任何页面，并且比限制性模板引擎更强大。实际中我建立web服务器的经验，例如jGuru.com（11万行），提供了有力的证据，表明程序员实际上在这两个方面可以获得最好的效果：分离和足够的生成能力。过去五年我在构建站点和打造我的StringTemplate模板引擎过程中，实际上是进行了一项软件设计的试验，在我的模板中严格的避免逻辑和运算*（译注：对上面例子的解释，完全图灵机式的模板引擎= 自由模板引擎，例如JSP、ASP；自由模板引擎理论上强大，但实际中因模型视图混杂而问题很多；限制性引擎理论上能力弱，但作者的实际经验表明能够在强制分离和足够的生成能力两方面获得最好的效果）*。尽管这些微不足道的证据还不能证明我的限制性引擎合格，另一方面就算绝大部分模板引擎的能力通过泰勒级数（Taylor series）接近sin(x)*（译注：意指100分？）*，我不相信一个设计师真的需要。

关键是不提供允许违背分离的结构，而使模板引擎具备足够的生成能力。在研究了我的站点上几百个模板文件之后，我确定只需要四种模板结构：属性引用（attribute references），包括判断属性是否存在的条件模板（conditional template），递归模板引用（recursive template references），以及最重要的类似lambda函数和LISP的map这样对多值属性操作的模板应用（template application）。

到此我提出的观点是严格的强制分离不仅是有用的设计原则，而且通过选择正确的模板结构实际上是可行的。第2节描述模板引擎怎样从之前的策略发展到目前解决各种实际设计问题，第3节明确的阐述分离的好处，第4节说明模型-视图-控制器模式怎样自然的运用到服务器端设计，以及怎样实现逻辑（模型）和显示（视图）的分离。第5和第6节正式的定义模板和三个限制性模板类型，第7和第8节定义严格的模型-视图分离原则，以及怎样避免混杂，最后第9节演示StringTemplate引擎解决一些重要的HTML生成任务。

2. 模板引擎的发展

生成动态页面意味着服务器不再将URL映射到磁盘HTML文件，取而代之的是映射到一堆代码，这些代码丢出正确的HTML，包括内容和相关的显示指令。在这我将描述Java引擎的发展，这些概念同样适用于Perl、VisualBasic等。

Java从Servlets [15]开始支持服务器开发，它调用那些响应HTTP GET和POST命令的方法，通过输出语句生成HTML。例如下面是一个简单servlet的主要处理，它生成一个页面，向URL参数传入的姓名问候“hello”。

```
out.println("<html>");
out.println("<body>");
out.println("<h1>Servlet test</h1>");
String name = request.getParameter("name");
out.println("Hello, "+name+".");
out.println("</body>");
out.println("</html>");
```

问题在于在Java代码中声明HTML很烦杂，是一种错误导向，HTML无法由界面设计师编写。为了改善这种状况，程序员可以尝试将普通的HTML输出元素提取成Java呈现对象（Java rendering objects），例如Table、BulletList，但终究还是在servlets中嵌入HTML方式。

接下来的发展阶段就是引入了JSP [6]，大致看来是前进了一大步。JSP文件本质上就是servlets，将Java代码嵌入在HTML文件中，由服务器自动转化为servlets。前面“hello”页面的JSP版本看起来就是下面这样：

```
<html>
<body>
<h1>JSP test</h1>
Hello, <%=request.getParameter("name")%>.
</body>
</html>
```

也许起初JSP只是作为简单引用数据的HTML文件，例如上面的例子，但很快退化为象Servlets一样完全的将代码和HTML声明混杂在一起，实际上界面设计师仍然无法修改JSP文件。更重要的是，JSP带来不好的面向对象设计，例如include文件就是类继承的一种不好的替代方式。JSP页面不能子类化，使得程序员很难提取通用的代码和HTML。Hunter [5]总结了JSP一些其它的问题，例如不太优雅的循环机制实现列表的显示等。

尽管JSP不是最终方案，但它确实将模板概念带进了程序员的思想中。模板是一个HTML文档，拥有一些填充点，可以使用数据或者是一些简单行为的结果来填充。不幸的是几乎每个模板引擎都在重演JSP的错误，它们提供一个完全图灵机式工具化的（Turing-complete tool-specific）编程语言，嵌入到HTML中，就像JSP一样，然后设计师不得不考虑用程序处理各种突发情况，而不顾及页面模型。

尽管JSP模板引擎解决了一些烦人的问题，很多人仍无法定位JSP不适用于大型系统的主要原因[1]。模板应当只是呈现数据系列的视图，完全与后台数据处理分开，只是显示后台数据处理的结果。如果模板语言功能太强，模板设计者可能就会混杂模板和业务逻辑，下面的章节详细说明为什么应当避免这样的混杂。

[1]尽管jGuru.com的页面URLs以.jsp结束，其实根本就不是JSP文件，去掉JSP后缀后，我们使用页面名字作为向后链接的替代。

3. 分离动机

使用模板引擎的主要目的是在思想和机制上将逻辑和数据处理与显示分离。Web站点开发情况下，这基本意味着代码中不存在HTML，HTML中没有代码。下面是程序员和界面设计师需要这种分离的一些原因：

1. **封装**：站点外观完全包含在模板中，业务逻辑完全位于数据模型里，每一部分都是完整的实体。
2. **清晰度**：模板不是生成HTML页面的程序，它是界面设计师或程序员可以直接阅读的HTML文件。
3. **分工**：编码人员开发时界面设计师可以并行建立模板，可以通过聘请一个界面设计师（通常不会太昂贵）降低程序员的负担，还可以节约沟通成本：界面设计师可以不用与程序员交流直接修改HTML。jGuru.com开发过程中我们不断的验证这种界面设计师和编码人员独立工作的方式。
4. **组件复用**：正如程序员为了提高清晰度和复用能力，将大的方法切分成小的方法一样，界面设计师可以轻松的将模板提取成一系列子模板，例如功能区、导航栏、搜索框、数据列表等。混杂的模板很难提取，并且很难与其它的数据源一起搭配复用。
5. **单点变化**：因为能够提取模板，界面设计师就能够抽象出链接一样小的元素，以及用户记录视图一样较大一些的项目。假如以后需要改变站点中每个用户列表的样式，界面设计师只需简单的修改一个模板文件，这也避免某个行为变化时修改多个地方而引入错误。模型中某个行为变化时只需要修改一个地方也非常重要，需要在模板中避免逻辑，例如“是管理员”这样的逻辑可能会在模板的多个地方重复。
6. **维护性**：更换站点外观只需要修改模板而不是程序，修改程序的风险远远大于修改模板。另外，修改模板不需要重启正在使用的服务器应用，而代码修改通常需要重新编译和重启。
7. **可更换的视图**：将数据模型和显示混杂，页面工程师无法象使用“皮肤”一样轻易的更换新的站点外观。在jGuru.com，每一种站点外观是一个叫做分组的模板集合，分组之间的外观可能完全不一样（针对那些简单的更换颜色、字体而言）。显示页面时，一个简单的指示器告诉页面控制器该使用哪个模板分组。
8. **安全性**：模板应用于页面客制化在博客中是一个通用的功能，不过正如微软Word中的宏一样，自由模板带来严重的安全隐患。Squarespace.com的博客在class loader调用方面经受大量的攻击之后，从Volocity [19]切换到了StringTemplate。人们也会想到一种简单但有效的攻击方式-死循环，正如本文所支持的观点，严格的将模型和视图隔离，禁止视图中的控制指令，等于安全性的提高。

程序员常常认为也不断的争论，严格的操作花费更多的时间。尽管在小规模情况下也许是这样，例如在视图中添加一块新的内容。我的经验是在周期长的项目中，项目进展会更快，具有更灵活、健壮的代码，这都得益于本节中列出的各种突出的优势。

4. 模型 视图 控制器模式

众所周知的模型-视图-控制器模式[9]在web服务器设计方面运用的很好，它提出三个领域，分别属于页面生成过程中不同的处理部分。大致上讲，视图代表页面模板或范本，控制器既代表服务器将URL映射到负责处理的代码片断的分发机制，也代表负责处理的代码片断本身，模型代表应用程序数据（或者状态），绝大部分业务逻辑，以及任何跟模型相关的处理。

不幸的是程序员们不确定哪儿是明确的区分这三个领域的界线，从2002年开始一个标题为“在MVC中将C和V分离”[12]的邮件列表在进行着这些讨论，出现这些最主要是因为控制器和模型的边界模糊不清。相比之下，尽可能的使视图简单，使它从与模型和控制器的混杂中脱离出来，是普遍认同的观点，是值得努力的方向。

以我的经验来看，控制器应当尽可能的轻量级。控制器的作用象是一个事件管理器，指示页面代码什么时候应当执行。不同页面的代码片断也都是控制器的一部分，应当限制使他们仅仅是从模型、会话或者参数列表中提取正确的数据，输出到视图中。如果页面是HTTP POST或者其它请求页面的最终处理者*（译注：例如ASP.NET中有的页面直接是Response.Redirect到其它页面，最终处理者就是它重定向的目标页面了）*，这个页面将触发模型中的行为。页面可能包含一些自己特定的数据处理，但通常最好是从页面中将这些代码提取到模型中，发挥模型的作用或者封装通用的操作。页面中对模型连续一系列的方法调用，也应当提取重构成模型中单一的方法，例如，页面可以完成简单的数据过滤（可能使用模型提供的通用目的过滤机制），但是象“处理论坛登陆请求”这样的操作应当被编码在模型中，处理页面只是简单的调用。

模型包含所有的业务逻辑，运算，以及状态（象数据库之类的持久化状态）。我见过一些学生项目和商业程序员的应用，将SQL语句写在控制器中，更惊讶的是写在视图里面，任何数据库结构的变化使得SQL扩散到的所有页面和视图必须修改。无论如何SQL表很少能成为合适的抽象层次，模型应当将数据库中原始数据处理成对象以及对象间的关系，也要封装象“购买图书”、“注册和发送邮件”这样经常执行的操作。

视图应当只是确定怎样显示模型处理完的数据，怎样向控制器传递数据，这样界面设计师对编写操作视图声明的工作会很合适。视图不应当是程序的一部分，意味着视图不能修改模型也不能处理数据。确定是否良好的隔离了视图，一个比较好的方法是视图是否能运用在其它应用中，例如，如果模板显示一个表中的元素列表，那么这个模板是否能被用在书店应用程序或者是论坛站点上？

模型和控制器之间的分界线有时比较模糊，依赖于具体的应用，但它们与视图之间的界线是明显的。

5. 模板定义

查找文献资料没有模板的定义，Hunter [5]指出了这是模板引擎市场的一个问题。虽然模板的概念很直观，但形式的定义可以提供一个通用术语，一个划分模板引擎和他们生成能力的方法。

输出模板不同于生成输出的程序，模板是一个范本，而后者是程序处理产生输出。模板是嵌入了行为的输出文档，显示模板时由引擎进行处理求值。JSP文件是自由模板的一个示例。

定义 1. *自由模板T是由输出字符t_i和行为表达式e_i交错出现的列表：*
$t_0e_0…t_ie_i…t_{ni}e_{ni}$
任何t_i可能是空字符串，e_i是语法运算上无约束的。如果T中没有e_i，那么T就只是一个简单的字符t_0。

定律 1. *自由模板等同于Chomsky无约束0型语法（unrestricted Chomsky type 0 grammar），因此可以生成递归可枚举语言（recursively enumerable languages）。*

证明. 因为e_i在处理上无约束，实际上可能是一个图灵机（Turing machine），因此可以完全由它自己生成0型语言。

这个结论是自证明的，不是很有意思，但值得明确声明的是自由模板是最强大的，能够生成任何想要的输出（例如图灵机能够处理的）。

在一些L语言中生成文档可能需要多个嵌套模板，因此单个字符t_i和单个模板的输出可能与L或者它的子句不一致。

通常字符被特殊符号表达式分割，例如下面是生成HTML body的一个简单HTML模板[18]：
<p>Hello, [% user %].

这个例子中，t_0 = ″<p>Hello,″，e_0 = ″$user$″，t_1 = ″.″。表达式被嵌入在字符中，词汇分隔符使用[%...%]。

为了与L一致，模板可能使用环境语言的词汇来编码一些表达式，例如XMLC [2]使用HTML的SPAN标签：
<p>Hello <SPAN id="user">Sample name</SPAN>.

自由模板可以修改模型，可以执行任何可调用的函数，也可以使用上下文改变输出。例如，模板自己可以使用近似泰勒级数的sin(i)函数计算table行的颜色。

有意思的是使用这种模板定义，XSL [20]样式表根本就不是模板，因为样式表使用一系列XSLT树状转换，生成的是XML或HTML文档。尽管声明更自然化而不是机器指令式的，XSL样式表更像是servlets一样的程序。

最后值得提出的是因为模板使用文本规格定义（textual specification），这并不限制模板生成文本内容（generating text.）。例如在生成文本前，很多源对源（source-to-source）的转换器可能会执行一系列树状转换。下面是一个模板使用（测试版本的）ANTLR [14]类似LISP树型符号的树型声明。
#(ASSIGN <ID> <expr>)

ASSIGN是拥有两个子表达式<ID>和<expr>的根字符，在构建树之前先处理并填充子表达式的值。

6. 限制性模板（Restricted template）分类

自由模板极为强大，但模板的能力和将模型视图混杂的能力之间有直接关系。模板越强，就越接近图灵机语言，另外，模板的能力与对界面设计师这样非程序员的适用程度存在一个对立关系。

自由模板面临的最大混杂问题是模板会影响模型。如果一个模板可以修改模型，那也是这个问题的一种。因此从视图中分离模型，首先应当限制模板只能以只读方式操作接收到的一系列数据值，这些只读数据值叫做属性*（译注：几乎文中所有提到的属性均指这种只读数据值，这种只读只是相对于模板/视图而言，对模型来说是可修改的）*，它们是由模型处理的。属性可能是像4521这样的单值型，[Jim, Frank, John]这样的多值型，或者是像（name=Tom, ext=5322）这样的聚合型。对未定义/未设置属性的引用将得到空字符串。

一旦限制模板只能操作一系列的属性*（译注：注意上面关于属性的定义，这里的操作指只读形式的操作）*以免产生边际效应（side-effect），那么对生成语法（generational grammar）类似的地方都需要注意。属性是生成语法的终结符（terminals）（词汇）*（译注：终结符是句子中会实际出现的符号，对应的非终结符在句子中不实际出现，仅在推导中起变量作用）*，模板是它的产生式（productions）（规则）。这种关系产生了一些有趣又有用的结果，即通过形式语言（formal language）为工作带来补充完善。例如XML文档类型定义（DTDs）其实是上下文无关语法[8]，因此，具有上下文无关语法能力的模板可以生成任何可使用DTD定义的XML文档。值得欣慰的是一个非常简单的模拟下推自动机（push down automaton）*(译注：下推自动机对应于上下文无关语法)*的模板引擎可以生成XML文档，像第8节介绍的，在设计时就可以强制模型和视图严格分离。

本节定义模板的限制性分类，依据是Chomsky的1..3型生成语法[1]：上下文相关语法（context-sensitive）、上下文无关语法（context-free）、正则语法（regular）。让我们从最弱的3型正则语法[16]模板类型开始。

定义 2. *正则模板由0、1或2个符号确定。第一个符号可以是一个字符t或者是一个属性引用a，第二个符*

号如果存在的话必须是一个正则模板的引用。正则模板可能是空字符串∈。换句话说，正则模板就是t、a、$t\tau$、$a\tau$，或者是∈，其中τ是正则模板的引用。属性和模板引用是边际效应无关的。

定律 2. *正则模板生成正则语言（译注：语言学定律正则语法生成正则语言，正则模板使用的正则语法）。*

证明. 定律3上下文无关模板的相关证明；只是限制了派生树（derivation tree）（子树）只能在右边拥有规则引用和相关模板引用。

更为普遍的正则表达式相当于提供了一个简单的方式来认识正则模板。模板可能重复一系列的属性和字符：a_i和t_i，实际上模板通过对多值属性循环来显示列表。令人惊讶的事实是这些迷你型的模板能够做很多事情，例如下面的模板，用一个虚构的模板引擎伪正则表达式符号生成使用
分隔的用户列表：

```
( $names "<br>" )+
```

这儿(…)+操作符表示“一个或多个”，它应该知道遍历names属性的多个值。大部分模板引擎实际上应该是像下面的Velocity [19]模板一样使用FOR循环类型的结构来处理：

```
#foreach( $n in $names )
$n<br>
#end
```

然而这样为了为列表中的每个姓名应用一个模板*（译注：指$n
部分，需要提取出来作为一个子模板）*，需要程序员在循环中引用另外一个模板，将这个模板应用在列表元素上，这就是说模板不是在右边的位置上被调用。例如假设界面设计师想提取一个叫做item的列表项模板：

```
<li>$attr$</li>
```

其中$attr$是对属性的引用，item会被应用在这个上面（$attr$类似面向对象方法中的this指针）。这样模板可以使用下面StringTemplate符号，将item子模板应用在names的每一个属性值上：

```
<ul>
$names:item()$
</ul>
```

虽然可以使用正则模板表达上面这种特定的输出语言，因为从形式语言[16]中我们知道有些语言本来就是非正则的，因此通常存在正则模板无法生成的输出语言。支持对其它模板全面引用的模板与上下文无关生成语法相关。

定义 3. *上下文无关模板是行为被限制在只能引用属性a_i和模板的一种模板类型。上下文无关模板可以是空，行为是边际效应无关的（side-effect free）。*

定律 3. *上下文无关模板生成上下文无关语言。*

证明. （概要证明）我们可以提供任何上下文无关语法句子的派生树与嵌套模板树结构之间的等价性。上下文无关语法的派生树在叶节点上是终结符引用，在子树根节点上是非终结符，子树对应于一个应用规则，它的下层是终结符或者其它非终结符子树。模板由字符列表、属性引用和模板引用构成，可以把模板的属性和字符看作终结符，把模板引用看作非终结符，这样我们就明白对应于派生树总会存在一个等价的模板，反之亦然。如果上下文无关模板等价于一个上下文无关语法，那么这个模板可以生成上下文无关语言。

为了清楚上下文无关语法的生成能力，回想一下，大多数编程语言的分析器通过LL-和基于LR-分析器生成器可接受的语法实现，他们都是上下文无关语法[17]的子集。因此，上下文无关模板足以生成符合普通编程语言语法的输出。

读者更感兴趣的是上下文无关模板能够生成任何符合某个DTD的XML文件，因为DTDs是上下文无关语法。对于SGML的情况又怎样呢？通过它的DTD例外情况（exceptions），一个SGML DTD可以确定什么情况下某个命名元素会出现或者不会出现[8]。在模板中，如果界面设计师需要基于上下文信息更改输出呢？例如只有在父菜单是活动状态时子菜单才显示的情况。允许上下文指示什么情况下某个模板元素可以出现带来更大的生成能力，这将我们带入到类似上下文相关语法的上下文相关模板分类。

定义 4. *上下文相关模板是上下文无关模板允许断言模板应用（predicated template application）的扩展，确切的说，仅在确定的语法上下文中允许模板引用或者包含子模板。行为指属性引用和模板引用，断言（predicates）作用于a_i和模板树结构本身，行为和断言是边际效应无关的（side-effect free）。*

将断言操作限制在a_i和当前模板树结构上，断言的作用就是测试语法上下文。我猜有人会给出上下文相关模板，与一个在行为上给予适当处理限制的上下文相关语法之间的等价性，但这种模板仅仅是理论研究的方向。实际上，最好是在模型中进行全部处理，将结果通过属性推（push）给模板，避免混杂。在模型中的运算处理是无约束的，因此模板的能力直接从上下文无关的类转移到无约束的类中。Milton和Fischer [11]抱怨类似的结果，他们的断言式LL(k)分析器是完全图灵机式的，这是因为任何断言都是完全图灵机式的。

下一节中，我展示分离的严格规则并阐明不合格的模板结构，然后第8节我指出哪些模板类自然的在设

计上强制分离。

7. 形式分离

程序员们崇尚简洁，但更崇尚强大的功能让人诧异的使用一行代码完成*（one-liners，一行代码主义者）*，这样带来复杂性成本，以Linux的拷贝程序cp为例，拥有29个参数。现有模板在特性和能力上不断突破，一些[10]在视图中有内置的标签让程序员发送邮件，其它一些甚至在视图里提供SQL访问数据库（或者非直接方式）！这是视图揉合了模型的典型情况。然而除了这些，大量表面上无伤大雅的模板特性混杂了视图和模型，还被当作有效的做法。

因为对分离缺乏形式的定义，模板引擎提供让程序员不经意间就违背分离的功能，丧失了前面描述的所有分离的优势，因此模板引擎并没有解决提升他们的开发这个问题。例如，为了支持像“让所有负数变成红色”这样没有坏处的功能，引擎支持属性值测试，这导致像“用户是James并且来历不明”惊人的原则背离*（译注：指在模板中完成这样的逻辑判断）*。尽管很多模板特性都是无坏处的被不恰当的使用，这些特性将不可避免的被用在违背分离原则上，可能是无意的也可能作为权宜之计。

很多引擎只是以学习一门新的编程语言和工具所需的成本来衡量，提供公认为有用并且是方便的动态页面组织方式。没有严格的强制分离，模板引擎就像给坏旧的蛋糕提供可口的糖衣。因此，别相信一个引擎的表现和能力，宁愿考虑它强制分离做的怎样。过去五年我领导他们构建站点和模板引擎的方式，已经抛弃了这个坏旧的蛋糕，在严格的强制分离同时提供足够的能力，尽管有时很烦人。

这节以学术的眼光深入大型商业服务器应用的开发，列出确保视图从模型和控制器分离的规则。然而程序员和界面设计师可以使用任何模板引擎，在不依赖强制手段的情况下遵循这些规则，这要求极强的锻炼和高度的认知感。

在形式化我的观点前，我提出几个简单的规则，在思想上澄清混杂这个概念。是否可以在完全不同的模型中重用这个模板？界面设计师理解这个模板吗？如果它像一个程序，那也许就是混杂了。如果模板能够修改模型，那它就是程序的一部分了。如果模板中存在逻辑处理或者是依赖于模型数据的逻辑，它也是程序的一部分。如果类型很重要，模板就是一个程序。

我定义分离的方式，是先确定通常哪些部分属于模型，哪些部分属于视图，以使他们独立而完整或者说被封装。然后我定义关于模型和视图交互的规则，防止混杂。

定义 5. *如果所有的逻辑和依赖于属性的处理都是在模型中完成，模型就是被封装的。*

注意上面没有提到的部分，例如并不排除视图在模板中计算sin(i)的值来确定第i行的颜色这样的处理。模型封装只是意味着模型必须包含所有与模型数据有关的处理。

定义 6. *如果所有的布局和显示信息都是在视图中声明的，视图就是被封装的。*

模型中不能有任何决定数据怎样被视图显示的处理。

定义 7. *如果模型和视图都是被封装的，那模型和视图就是严格分离的。此外，如果视图不能修改模型，视图就不会对模型数据类型有任何设想。*

这个定义从防止混杂的关键约束中提炼出来，实际上，严格的分离极大的意味着在模型和模板中能做什么，不能做什么。下面是用具体项表示的严格分离的隐含规则：

1. 视图既不能直接修改模型数据对象，也不能调用模型中带来边际效应的方法。就是说模板可以从模型访问数据，调用方法，但这些引用必须是边际效应无关的。提出这个规则部分是因为数据引用必须是顺序无关的，见7.1节。
2. 视图不能进行依赖于属性的运算，因为这些运算以后可能会改变，应当被优雅的封装在模型中。例如视图不能像“$price*.90”这样计算图书销售价格。为了与模型保持中立，视图不能对数据意义进行断言。
3. 视图不能基于属性进行比较，但能测试属性是否存在，或者是多值数据值的长度。像
   $bloodPressure<120
   这样的测试必须转移到模型中，如同医生具有减轻我们心理压力的习惯一样。下面这样测试一个值是否存在的表达式必须被替换掉
   $bloodPressureOk!=null
   模板输出可能取决于模型数据和运算，但这些条件必须由模型决定。就算是使所有负数值红色显示这样简单的测试也应当在模型中完成，其它更高层次的一些抽象是“部门x正在亏损”。
4. 视图不能对数据类型断言。一些类型断言非常明显，例如视图断言数据值是一个日期类型，但出现更多的是一些微妙的断言：如果模板断言$userID是一个整数，不改变模板程序员可能无法在模型中将值修改成非数字值，这个规则禁止像colorCode[$topic]和$name[$ID]这样用ID用作数组索

引。视图也不能使用参数调用方法（静态或者动态的），因为这儿对参数类型有一个断言，除非可以确保模型的方法仅仅将他们当作object类型。此外界面设计师不是程序员，期望他们调用方法知道应当传什么参数是不现实的。

5. 模型中的数据不能包含显示和布局信息。模型不能将任何显示信息伪装成数据值传递给视图，这不包括传递一个应用于其它数据值的模板名称。

对依赖的值，这些规则不仅是指像$user或者$model.getUser()这样直接的模型引用，也指本地模板中逻辑上依赖模型数据而变化的变量，例如，

```
#set interest = 0.08
#if ( $risk < 0.5 )
#set interest = 0.065
#endif
#set interestRate = interest*100
```

这儿变量interest和interestRate依赖于模型数据值$risk，这种依赖性很明显的浮现在你面前。

URLs怎样处理？在模板中硬编码URL是否合适？严格的讲，站点的URL结构属于控制器范畴，不应当在模板中以字符方式引用。

下一节描述普通的模板数据访问策略怎样先天就是不安全的，指出一个界面设计师不留意间就会踩上的隐藏地雷。

7.1 提取策略（Pull Strategy）违背分离

考虑应用程序中一个简单的HTML页面，需要从数据库提取一个姓名列表并显示出来，并且显示列表元素的数量。下面的模板从模型中提取数据：

```
<html>
<body>
Names:
<p>
$model.getNames()
<p>
There were $model.getNumberOfNames() names.
</body>
</html>
```

$model.getNames()调用模型的方法获取姓名列表，并返回给模板，$model.getNumberOfNames()引用让模型计算并返回姓名的数量。因为getNames()先被调用，列表和长度已经可以得到，因此getNumberOfNames()可以简单的返回长度数值。现在如果界面设计师想在页面的<title>中显示姓名的数量会怎样？就是说必须首先调用getNumberOfNames()，这时模型还没有得到列表。最好的情况是模板得到一个无效数字，最坏的情况是发生异常。移除这种顺序依赖性唯一的方法是使getNumberOfNames()也调用getNames()方法。

这样这种简单的数据引用，界面设计师毫不思考的在模板中使用，迫使模型实现上不得不做一些戏剧性的修改。同样在依赖的结构中模型代码的修改也可能破坏以前正确的视图，除非开发团队拥有极度优秀的测试，但对于动态页面站点这非常困难，模型的修改将不可避免、隐蔽的破坏一些站点页面。

为了避免数据引用的顺序依赖性，使用推进策略（Push Strategy）。

定义 8. *模板使用推进策略指在模板处理前，所有模板引用的数据都已经被处理成模板可以使用的一系列属性a_i。*

属性用流标识（stream tokens）这样的方式“推”给模板，对比之下自由模板可能通过函数的方式从数据模型“提取”。

定义 9. *如果模板使用的任何数据都通过调用模型方法按需处理，模板使用的是提取策略。*

定律 4.*安全的提取策略在最坏的情况下将退化成推进方式。*

证明. 模型数据值（后面就称作属性了）经常依赖于其它属性的值，这种依赖性源自于DAG，一个定向无环图。图是非循环的便于清晰的定义，例如属性a_i不能依赖于它自己。为了保持正确性（安全），视图不能违背图的依赖性向模型请求属性。

在最差的情况下，属性依赖于DAG的拓补排序只有唯一的顺序，就是说，DAG形成一个依赖链，a_i依赖于a_{i-1}（$a_{i-1} \rightarrow a_i$），因此间接的a_n依赖于全部$a_1..a_{n-1}$：

$$a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_{n-1} \rightarrow a_n$$

为了处理a_n，意味着为了确保正确性模型必须处理所有其它的属性。因为界面设计师可能第一件事情就是在视图中请求a_n，因此模型必须先处理$a_1..a_{n-1}$，然后再是a_n。如果a_n是模板的第一个表达式e_0，a_n也是在模板生成任何输出前被处理，因此，所有属性先期已经处理完毕，提取策略在这种最差的情况下退化成推进方式。

7.2 混杂索引

评估模板引擎时，根据模板引擎提供的工具所允许的混杂级别进行考核很有帮助。

定义 10. *模板引擎的混杂索引指模板可以违背分离规则的数量。有5个分离规则，因此索引应当是1..5这几个值。最小索引是1，这是因为没有方法可以阻止属性包含显示和布局信息。*

StringTemplate使得设计时不可能违背分离，混杂索引为最小值1。在着手一个模板考察论文时，我发现只有HTML模板[18]和XMLC[2]的混杂索引为1。对比之下下面这些模板引擎混杂指数大于1：Tapestry、WebMacro、Velocity、PTG、UniT、Tea、WebObjects、FreeMarker、ColdFusion、Template Toolkit，以及Mason。

FreeMarker的混杂索引为5，因为它允许对模型进行任意的方法调用，违背规则1和4，它在模板中允许业务逻辑，违背规则3，最后它允许对属性进行的操作，违背规则2。

流行的web应用控制框架Struts没有混杂指数，因为它自己不是模板引擎，它设计为使用其它视图系统，例如XSLT、Velocity、JSP和JSP标签库等，进行工作。Struts-StringTemplate搭配应当是一种有用的组合。

有趣的是好像不存在中间地带，我得到的结论是引擎都两极分化在1和5上面，这暗示出为引擎添加特性具有“斜坡效应”，必须谨慎对待。一旦引擎允许一个违例，它必然会违背其它的，例如为了允许对模型进行方法调用（潜在的违背了边际效应无关原则），引擎必须违背参数类型独立规则。

8. 强制分离

分离的规则很严格、遥不可及，好像将范围限制得极度狭窄，因此有两个问题立即进入脑海中：1）能够强制执行这些规则吗？2）符合规则的模板引擎有用吗？

有一个特例，模板引擎无法监控接收到的属性，确定模型或控制器是否传入了显示信息或到另一个模板的链接，例如属性值中是否存在红色显示某个颜色或是人的姓名。除了这个特例之外，原则上其它规则都是可以强制的。

对于其它四个分离规则，人们可以在自由模板引擎中使用静态或运行时检查的组合方式，去捕获违例，提示程序员。我的直觉是这样会存在许多无法确定的问题，更好的方法是一开始就使用一个非常严格的模板，再逐渐添加符合分离原则的结构特性，这也是StringTemplate走过的发展历程的反映。

基于具有一些可填充点的文本文档，程序员和界面设计师可以将称作属性的具有单个或多个值的只读数据粘贴到这些填充点。模板只是单纯的操作属性的这个断言，允许我们将模板和形式语言语法关联起来，既然数据引用不能具有边际效应，因此可以提供顺序无关的数据引用，确保模板不会修改模型。根据定义，模板不允许处理业务逻辑，属性就像是掉进汤中的石头，使用一个化学类比，数据应当与周围的模板混合但不发生反映。

将大模板分解成子模板很有用并且是安全的，因此引擎可以支持include指令。Include指令更成熟的版本是借鉴宏的方式，因为宏具有命名参数传递给被包含模板。参数自然就是发送给子模板的属性。一个小示例，考虑如下定义一个叫做bold的模板

```
<b>$attr$</b>
```

模板可以像宏一样调用bold：bold(attr=name)，稍微改变一下可以允许“模板应用”型功能，得到如下等价的操作：name:bold()。如果name是多值型的，bold模板会被应用到列表中的每一个元素，这样就得到类似LISP的map一样的属性列表变种版本，这种结构免去了显示的对FOR循环的需求。

添加递归使模板能够直接或间接的调用它自己，模板就达到上下文无关级别。仅仅使用属性引用、模板应用和模板递归，应用程序能够在不违背任何分离原则情况下生成任何具有DTD的XML文档。

我们需要更多能力？就是说我们是否需要上下文相关模板的能力？对于那些需要在登陆、退出之间切换的环境，更重要的是像下面示例那样需要处理条件式include字符（或子模板）任务的情况下，上下文相关模板很有用。

```
$if(title)$
<h1>$title$</h1>
$endif$
```

<h1>标签只在存在title属性时出现。

模板只能够测试属性是否存在而不是使用完全的表达式，因为对属性值的充分使用违背分离。这种受限的IF结构使模板远离上下文无关模板而进入上下文相关范围，但具体多远呢？当然完全的表达式更强大，的确是的，但程序员总能够在模型中做任何他们想要的处理，例如设置属性或者让它为空，如同他们在视图中处理那个表达式一样，可以在模型中实现一个具有同样结果的布尔属性。

在模型中添加一个IF结构，只能用于对模型中无限制的逻辑处理结果进行测试，这样被分离的模型-视图搭配跟自由模板一样强大，因此第二个关于使用性问题的答案是：需要，模型-视图组合完全达到自由模板的能力，并不违背任何分离原则。

8.1 属性呈现（attribute render）

FreeMarker [3]的最初作者Benjamin Geer [4]，给强制分离的引擎提出了一个好像无法解决的难题，例如考虑将字符"<"转义为HTML格式的"<"，需要一个公共的显示操作类来完成类似字符串头尾空格压缩、日期格式化、大小写转换等。不违背分离情况下，模型不能预先转义数据字符串然后发送给模板，因为"<"明显是一种显示信息。模板不能检查接收到的字符串，查找"<"字符做必要的替换，也不能调用模型的方法处理这个字符串转义操作。那么人们怎样完成转义字符串这样一个普通而又必要的操作？

不可避免的事实是在某个地方必须有一些Java代码来完成HTML转义序列，因为这个处理不能在模板中完成。此外Java已经提供强力支持，例如用于处理文本、数字、日期等数据的java.text包。既然模型和控制器都不能包含这个处理，而这些代码必须放在某个地方，这表明在MVC模式工作时应当有一个默默无闻的合作者，适合于这种类型的属性显示描绘操作。

为了揭示这个安静的合作者，考虑对于每一个传递给模板的整数属性，模板引擎使用默认方式隐式的调用属性的toString()方法。所有属性类型的toString()方法都包含一个方面（aspect）[7]，或者建议在针对HTML页面生成时使用一种更全面的设计模式：MVCR（*model-view-controller-renderer模型-视图-控制器-描绘者*），这儿描绘者封装将对象转换为可显示字符串的操作。

在视图和描绘者之间有一个微妙、有争议的区别。模板主要关注页面和属性的布局，以及像颜色、大小这样的文本属性，另一方面，描绘者关注将属性转换（这种情况下或者叫做聚合、委托组件）成可显示字符串。例如描绘者确定是否应当将一个负数显示为(23)而不是-23，而模板确定在页面上什么位置显示这个数字，字体多大，什么颜色，是否是一个链接等等。

这样为了解决HTML字符转义难题，控制器可以简单的在toString()方法中使用一个完成特殊字符转义的对象，包装字符串属性。当模板显示文本时，字符串将被正确的转义，对其它的属性，包装器可以聚合标准的Java库来完成。控制器的作用好像是从模型到视图装配线上的一个基站，为特定对象进行一个描绘工作。标准类型以及所有应用程序用到的包装器对象的toString()方法包含这个描绘者。将包装器对象封装在一个HTMLRenderer类中是有意义的，事实是这个类可以被任何web应用重用，这更说明描绘显示的代码不是特定应用程序模型的一部分。

每个模板可能有它自己的描绘者，控制器可以指定不同的描绘者，以适应不同区域（locales）的页面请求。另外，视图-描绘者配合能够自动执行像字符串转义之类的程序操作，因此消除了控制器不得不格外的包装属性的考量。

在面向对象思想中让属性描绘显示它自己更适合，这样建立了一个边界，将传统视图组件切分成视图和描绘者，产生MVCR模式。描绘者优雅的解决了Geer的难题。正如模板通过测试属性是否存在的解耦机制访问模型中无约束的逻辑，模板也可以在不违背模型-视图分离原则下，隐式的通过toString()方法这个狭窄的渠道，访问由描绘者处理的无约束的显示逻辑。

一些读者会争辩，MVCR模式只是一个聪明的措辞掩饰，它为模型-视图分离打开了潜在的漏洞。描绘者将拥有使用包含显示信息字符串的代码，更坏的是，描绘者可能潜在包含修改应用程序模型的代码。首先应当认识到没有办法阻止对象在toString()方法中描绘他们自己，不管使用模板还是其它方式，不将整数从32位二进制字转换为数字字符串，服务器无法生成web页面。防止滥用这种机制是无法判定的，引擎已经承认了这个弱点，认为最小的混杂索引为1。就算是在描绘者里面，我们也应当考虑将HTML字符提取到细粒度的模板中，完全的在代码中避免HTML。其次，形式化描绘者，将描绘者操作封装到应用程序无关的服务中，这样对于给定的目标语言，例如HTML，允许将描绘者与引擎一起提供。因为描绘者不会成为特定应用的一部分，它无法混杂应用的模型和视图。技术上讲，程序员可以修改HTML描绘者，或者绑定可以修改特定模型数据的"恶意"描绘者，但这种风险已经随处存在：人们可以轻易的修改严格引擎的源代码，以允许违例。

8.2 强制与鼓励

程序员渴望扩展性，他们是否愿意牺牲一些扩展性来强制一些有价值的原则，依赖于他们的经验，以及由这些经验产生的价值观。如果他们了解到自己使用了一种困难的方法，有时就算是天使也会犯错或者是采用权宜之计，那他们在设计时就会努力奋斗使这个机制正确的工作。在一次火箭推进装置测试中，一些安装加速器的人用错了方法，无谓的压坏了测试宇航员的身体，由这件事情想起了墨菲定律“一件事情如果能够被做错，那它一定会被做错”。读者可能还能回想起19世纪80年代的计算机，键盘和鼠标的插口完全一样，很多人因为意外的用错插口而烧坏了键盘。

对于一种好的行为，明智的方法是强制而不仅仅是鼓励，尽管强加的限制有点恼火。程序员第一次转移到新的模板引擎的主要原因，是为了避免原有系统中JSP或者其它系列模板引发的混杂。我认为类似重新实现JSP，在用户手册中添加一些备注，鼓励非JSP风格的行为，这种做法是不充分的。我对待StringTemplate的方法是基于安全而构建起来，在严格的坚持我的原则基础上添加功能特性。

9 StringTemplate

StringTemplate [13]是我和Tom Burns（jGuru.com CEO）在多年构建商业网站，包括jGuru.com、knowspam.net和antlr.org等的经验基础上，认真设计的一个模板库。StringTemplate从简单的填充文档发展成为成熟的模板引擎，令我非常惊讶的是，具有明显函数编程风格。函数编程语言从来不是我的专长，StringTemplate的语言根据我的需要自然发展而来：边际效应无关的表达式、表达式顺序无关性、用于显示数

据对象列表的模板应用*（译注：指StringTemplate中显示多值属性/属性集合的方式）*、模板的嵌套（递归）应用*（译注：指模板可以调用另外一个模板）*，以及简化到使我的界面设计师能够理解。StringTemplate的设计目的是改善强制分离，而不是完全图灵机式的模板，或者具有迷人表现的“一行代码主义者”。StringTemplate这个简单的名字反映出我最简化的目的（它的jar源代码和二进制大约120K）。

StringTemplate强制模型和视图分离，但它无法阻止使用属性传递显示信息这个问题的发生。我们不断的验证，界面设计师和编码人员可以独立工作，一旦我们的界面设计师知道哪些属性她可以填充，哪些属性她可以引用，她就能够完全独立的构建多种站点外观（针对客户而言）。

另外，使用模板表达式语言，StringTemplate极大的促进了组件复用；支持多种站点皮肤（皮肤就是一组模板）；提供各种皮肤之间的模板继承，使得界面设计师能够构建与已有皮肤不同的新站点皮肤；允许界面设计师更新正在使用中的服务器；降低页面控制器手工生成HTML或者子模板的需求。

在讨论模板引擎时，程序员会问模板引擎怎样处理一些有意思的输出任务，下一节演示怎样使用StringTemplate显示一个table，交错变化table行的颜色，以及生成层级菜单。示例演示了两个关键特性，用于减少循环使用的结构，以及其它每一种通用语言都具有的结构：（像LISP的map）用于多值属性的模板应用和模板递归。

9.1 显示一个table

假设我们有一个User类型的对象，从模拟数据库中提取。

```
public class User {
    String name;
    int age;
    ...
}
static User[] list = new User[] {
    new User("Boris", 39),
    ...
};
```

用下面的Java代码将用户列表数据作为users属性推给一个叫做st的模板*（译注：在模板st中使用users名称来访问这个属性）*，

```
st.setAttribute("users", list);
```

为了输出下面这样一个table：

```
<table border=1>
<tr><td>Boris</td><td>39</td></tr>
<tr><td>Natasha</td><td>31</td></tr>
...
</table>
```

为每个用户对象应用一个匿名模板*（译注：匿名模板指{}对以及里面的部分，users:这个地方就是使用users名称访问前面setAttribute方法推给模板的列表类型的属性）*：

```
<table border=1>
$users:{ <tr><td>$attr.name$</td><td>$attr.age$</td></tr>
}$
</table>
```

attr是默认的属性名称，它的值随着StringTemplate在users上循环而自动改变，attr.name通过getName()方法访问当前循环User对象的name属性*（译注：这段话是解释工作方式的，首先users:这个表达式告诉StringTemplate引擎，users是一个列表，需要循环为列表中的每一个元素应用:后面的模板。在{}给出的匿名模板里面，使用attr这个默认的属性名称访问当前循环的User对象）*。其中的匿名模板也可以提取被提取到一个叫做row的模板中：

```
<tr><td>$attr.name$</td><td>$attr.age$</td></tr>
```

主模板缩减到更易于阅读*（译注：下面是使用命名模板的方式，$users:row()$意思就是循环为users中的每一个对象调用row这个模板。模板调用row()类似函数调用方式，了解StringTemplate的话可以知道调用模板时可以传递参数的）*：

```
<table border=1>
$users:row()$
</table>
```

模板应用的结果是另外一个列表，模板可以应用在另一个模板应用的结果之上。例如，每个列表元素先被加粗显示，然后被应用在<ul>项模板中。

```
<ul>
$aList:{<b>$attr$</b>}:{<li>$attr$</li>}$
</ul>
```

（译注：这样来理解上面这个模板的执行。首先使用$aList:{$attr$}$，它的结果是另外一个列表

*{attr1,attr2…attrn}，我用字符串的方式来表示这个结果列表，用逗号将列表中的元素隔开。然后，可以使用bList命名这个结果列表，这样$bList:{
$attr$
}$就很好理解了。例如上面显示用户列表的例子，
users:{$attr.name$}:{
$attr$
}，结果是
Boris
Natasha
…）*

9.2 交错变换table行的颜色

界面设计师经常对列表元素使用一种交替变换的背景颜色：

```
<table border=1>
<tr><td>Boris</td><td>39</td></tr>
<tr><td bgcolor=#F6F6F6>Natasha</td><td>31</td></tr>
<tr><td>Jorge</td><td>25</td></tr>
...
</table>
```

在StringTemplate中，不是借助于嵌套了IF的FOR循环，而是使用逗号分隔的模板列表，以轮循方式对列表元素应用这些模板*（译注：第一个元素使用rowBlue模板，第二个使用rowGreen，第三个使用rowBlue…）*。

```
<table border=1>
$users:rowBlue(), rowGreen()$
</table>
```

9.3 层级菜单

在模板中为什么需要递归式？考虑实现层级菜单的情况，没有递归式，模板需要复杂的嵌套FOR循环和IF结构。层级是一种递归数据结构，需要使用递归遍历这个树，没有递归式的引擎无法自然的处理递归数据结构。为了演示分级菜单，假设菜单中的每个条目都有一个标题，一个url，可能有一个子菜单条目列表，以及某个特定的子菜单是否活动状态的标识。

```
public class TreeMenuItem {
    String title;
    String url;
    boolean active = false;
    List submenu;
    ...
}
```

在整个页面模板环境中将menuItem模板应用到菜单的主选项：

```
$choices:menuItem()$
```

非递归式的模板在仅仅生成最顶层菜单时像下面这样：

```
<a href=$attr.url$>$attr.title$</a>
```

输出可能是：

```
<a href=/>home</a>
<a href=/news>news</a>
...
```

如果一个子菜单是活动状态，为了显示子菜单，StringTemplate只要使模板列举（tail-recursively尾部递归）子菜单列表项。

```
<a href=$attr.url$>$attr.title$</a>
$if(attr.active)$
$attr.submenu:menuItem()$
$endif$
```

显示table的示例仅使用上下文无关模板结构，层级菜单示例需要上下文相关来显示活动状态的子菜单，同时请注意没有任何显示信息需要通过属性传递给模板。

程序员经常问怎样在模板中实现一段复杂算法，层级菜单示例提供了关键的一课，通过正确的数据结构选择，使用简单的状态机引擎和高度互连的状态网映射关系，可以轻松的在模板中避免凌乱的处理和控制流结构。例如，语言分析器的状态机结构不简单，但是遍历结果状态分析句子却很容易，实际上，创建状态的代码使得分析引擎更简单。分析结构类似于控制器-模型组织的数据结构，模板引擎类似于分析引擎。

10 结论

严格的分离模型和视图是极为有意义的目标，因为它提高了弹性，降低维护成本，并允许编码人员和界面设计师可以并行工作。像JSP以及其它混杂模型和视图的模板引擎发展起来，以适应之前的技术，不幸的是这些模板仅仅鼓励分离，仍然支持允许程序员违背分离的结构。程序员会经常使用这些结构作为一种权宜之计，或者是由于缺乏经验，因此现在的引擎提供超越JSP的一些含糊的优势（尽管绝大部分引擎除了模板之外还提供敏捷的框架用于创建web应用）。

因为没有形式的分离定义，也没有规则列表的指导，已有引擎并不强制分离，这也是为了避免在程序员自然的工作习惯上限制功能，担心强制分离意味着被认为是弱引擎。我也不确定是否有引擎设计者有兴趣响

应程序员的呼声而抛弃所有的原则实现模板语言。

我证实了实际上我们能够创建一个模板引擎，在设计上严格的强制遵循所有确定的分离规则。经验证明，StringTemplate用于生成众多像jGuru.com这样复杂的站点是足够强大的，并且，那些页面简单而足够清晰，能够被界面设计师而不是程序员创建。

通过提出限制性模板和生成语法之间的关系，我论述了限制性模板能够生成那些可以被XML DTDs描述的语言种类，达到上下文相关语言和无约束语言程度。

给出清晰的模型视图分离定义，模板引擎设计者可能不再主张没有混杂索引1的强制分离，更重要的，给出理论论点和经验证明，引擎不再有借口支持混杂，担心较弱的生成能力。

我目前正在研究，在下一代的ANTLR分析器中，使模板引擎能够用作强大、可复用的语言转换器。StringTemplate在BSD许可下发布在http://www.stringtemplate.org上。

11 感谢

我将感谢John Witchel促使我形式化关于MVC的分离思想，感谢Monty Zukowski在1995年的时候在我脑海中留下了"meme"模板的映象。我感谢Chris Brooks和David Galles细致的审阅。Matthew Ford为本文的思想以及我的引擎源代码提供了很有价值的回溃。我感谢Benjamin Geer的审阅工作以及指出我论据中的不足之处。

12 引用

[1] N. Chomsky. *Aspects of the Theory of Syntax*. MIT Press, 1965.

[2] Enhydra. XMLC. http://xmlc.enhydra.org/project/aboutProject/index.html.

[3] FreeMarker. http://freemarker.sourceforge.net.

[4] B. Geer. Private communications.

[5] J. Hunter. The problems with JSP. http://www.servlets.com/soapbox/problems-jsp.html.

[6] JSP. http://java.sun.com/products/jsp.

[7] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[8] P. Kilpelainen and D. Wood. SGML and XML document grammars and exceptions. *Information and Computation*, 169(2):230–251, 2001.

[9] G. Krasner and S. Pope. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.

[10] Macromedia. Coldfusion. http://www.macromedia.com/software/coldfusion.

[11] D. R. Milton and C. N. Fischer. LL(k) parsing for attributed grammars. In *International Conference on Automata, Languages, and Programming*, pages 422–430, 1979.

[12] Separating C from V in MVC. http://mathforum.org/epigone/modperl/jilgygland, 2002.

[13] T. Parr. StringTemplate documentation. http://www.antlr.org/stringtemplate/index.tml, September 2003.

[14] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25(7):789–810, 1995.

[15] Servlets. http://java.sun.com/products/servlet/.

[16] S. Sippu and E. Soisalon-Soininen. *Parsing Theory I*. Springer-Verlag, 1988.

[17] S. Sippu and E. Soisalon-Soininen. *Parsing Theory II*. Springer-Verlag, 1988.

[18] S. Tregar. HTML::Template. http://html-template.sourceforge.net.

[19] Velocity. http://jakarta.apache.org/velocity/index.html.

[20] XSL style sheets. http://www.w3.org/Style/XSL/.

译注：
文章很多涉及语言学自动机方面理论，详细请参考相关资料，下表是Chomsky语法分类、自动机、语言对应关系回顾参考：

	语法	自动机	语言
0型	无约束语法 短语结构语法	图灵机	递归可枚举语言
1型	上下文相关语法	线性有界自动机	上下文相关语言
2型	上下文无关语法	下推自动机	上下文无关语言
3型	正则语法	有限自动机	正则语言

部分翻译对照：
Restricted engine/template：限制性引擎/模板，指为了使模型-视图严格分离，在模板引擎中强制加入了限制性规则，通常认为这会给视图的呈现能力，也就是HTML或者说模板的生成能力带来损失。
Unrestricted engine/template：自由引擎/模板，相对于限制性模板、无约束文法而言，例如JSP、ASP都属于自由模板，它们允许在模板的服务器代码中做任何处理，导致模型混杂到视图中，但这种类型的模板对控制HTML生成被认为是最灵活的。
Template application：模板应用，指模板对多值属性、列表属性的一种处理方式，参考9.1或StringTemplate。
Template recursion：模板递归，指模板可以调用另外一个模板，或者叫做模板嵌套，参考9.1或StringTemplate。
Generational power：生成能力，指模板生成最终文档（HTML文档等）的处理能力。
Push strategy：推进策略，指由模型主动给模板设置属性，主动将模板需要的数据推给模板的方式。
pull strategy：提取策略，指模板主动向模型提取需要的数据的方式。