不知道你有木有听说过一个基于 Javascript 的 Web 页面预处理器，叫做 AbsurdJS。我是它的作者，目前我还在不断地完善它。最初我只是打算写一个 CSS 的预处理器，不过后来扩展到了 CSS 和 HTML，可以用来把 Javascript 代码 转成 CSS 和 HTML 代码。当然，由于可以生成 HTML 代码，你也可以把它当成一个模板引擎，用于在标记语言中填充数据。

于是我又想着能不能写一些简单的代码来完善这个模板引擎，又能与其它现有的逻辑协同工作。AbsurdJS 本身主要是以 NodeJS 的模块的形式发 布的，不过它也会发布客户端版本。考虑到这些，我就不能直接使用现有的引擎了，因为它们大部分都是在 NodeJS 上运行的，而不能跑在浏览器上。我需要的 是一个小巧的，纯粹以 Javascript 编写的东西，能够直接运行在浏览器上。当我某天偶然发现 John Resig 的这篇博客，我惊喜地发现，这不正是我苦苦寻找的东西嘛！我稍稍做了一些修改，代码行数差不多 20 行左右。其中的逻辑非常有意思。在这篇文章中我会一步一步重现编写这个引擎的过程，如果你能一路看下去的话，你就会明白 John 的这个想法是多么犀利！

最初我的想法是这样子的：

```
1.  var TemplateEngine = function(tpl, data) {
2.      // magic here ...
3.  }
4.  var template = '<p>Hello, my name is <%name%>. I\'m <%age%> years old.</p>';
5.  console.log(TemplateEngine(template, {
6.      name: "Krasimir",
7.      age: 29
8.  }));
```

一个简单的函数，输入是我们的模板以及数据对象，输出么估计你也很容易想到，像下面这样子：

```
1.  <p>Hello, my name is Krasimir. I'm 29 years old.</p>
```

其中第一步要做的是寻找里面的模板参数，然后替换成传给引擎的具体数据。我决定使用正则表达式来完成这一步。不过我不是最擅长这个，所以写的不好的话欢迎随时来喷。

```
1.  var re = /<%([^%>]+)?%>/g;
```

这句正则表达式会捕获所有以<%开头，以%>结尾的片段。末尾的参数 *g*（global）表示不只匹配一个，而是匹配所有符合的片段。Javascript 里面有很多种使用正则表达式的方法，我们需要的是根据正则表达式输出一个数组，包含所有的字符串，这正是 *exec* 所做的。

```
var re = /<%([^%>]+)?%>/g;
var match = re.exec(tpl);
```

如果我们用 *console.log* 把变量 *match* 打印出来，我们会看见：

```
[
    "<%name%>",
    " name ",
    index: 21,
    input:
    "<p>Hello, my name is <%name%>. I\'m <%age%> years old.</p>"
]
```

不过我们可以看见，返回的数组仅仅包含第一个匹配项。我们需要用 *while* 循环把上述逻辑包起来，这样才能得到所有的匹配项。

```
var re = /<%([^%>]+)?%>/g;
while(match = re.exec(tpl)) {
    console.log(match);
}
```

如果把上面的代码跑一遍，你就会看见*<%name%>* 和 *<%age%>*都被打印出来了。

下面，有意思的部分来了。识别出模板中的匹配项后，我们要把他们替换成传递给函数的实际数据。最简单的办法就是使用 *replace* 函数。我们可以像这样来写：

```
var TemplateEngine = function(tpl, data) {
    var re = /<%([^%>]+)?%>/g;
    while(match = re.exec(tpl)) {
        tpl = tpl.replace(match[0], data[match[1]])
    }
    return tpl;
}
```

好了，这样就能跑了，但是还不够好。这里我们以 *data["property"]*的方式使用了一个简单对象来传递数据，但是实际情况下我们很可能需要更复杂的嵌套对象。所以我们稍微修改了一下 *data* 对象：

```
{
    name: "Krasimir Tsonev",
    profile: { age: 29 }
}
```

不过直接这样子写的话还不能跑，因为在模板中使用*<%profile.age%>*的话，代码会被替换成 *data[‘profile.age’]*，结果是 *undefined*。这样我们就不能简单地用 *replace* 函数，而是要用别的方法。如果能够在*<%*和*%>*之间直接使用 Javascript 代码就最好了，这样就能对传入的数据直接求值，像下面这样：

```
var template = '<p>Hello, my name is <%this.name%>. I\'m <%this.profile.age%> years old.</p>';
```

你可能会好奇，这是怎么实现的？这里 John 使用了 *new Function* 的语法，根据字符串创建一个函数。我们不妨来看个例子：

```
var fn = new Function("arg", "console.log(arg + 1);");
fn(2); // outputs 3
```

fn 可是一个货真价实的函数。它接受一个参数，函数体是 *console.log(arg + 1);*。上述代码等价于下面的代码：

```
var fn = function(arg) {
    console.log(arg + 1);
}
fn(2); // outputs 3
```

通过这种方法，我们可以根据字符串构造函数，包括它的参数和函数体。这不正是我们想要的嘛！不过先别急，在构造函数之前，我们先来看看函数体是什么样子 的。按照之前的想法，这个模板引擎最终返回的应该是一个编译好的模板。还是用之前的模板字符串作为例子，那么返回的内容应该类似于：

```
return
"<p>Hello, my name is " +
this.name +
". I\'m " +
this.profile.age +
" years old.</p>";
```

当然啦，实际的模板引擎中，我们会把模板切分为小段的文本和有意义的 Javascript 代码。前面你可能看见我使用简单的字符串拼接来达到想要的效果， 不过这并不是 100%符合我们要求的做法。由于使用者很可能会传递更加复杂的 Javascript 代码，所以我们这儿需要再来一个循环，如下：

```
var template =
'My skills:' +
'<%for(var index in this.skills) {%>' +
'<a href=""><%this.skills[index]%></a>' +
'<%}%>';
```

如果使用字符串拼接的话，代码就应该是下面的样子：

```
return
'My skills:' +
for(var index in this.skills) { +
'<a href="">' +
this.skills[index] +
'</a>' +
}
```

当然，这个代码不能直接跑，跑了会出错。于是我用了 John 的文章里写的逻辑，把所有的字符串放在一个数组里，在程序的最后把它们拼接起来。

```
var r = [];
r.push('My skills:');
for(var index in this.skills) {
r.push('<a href="">');
r.push(this.skills[index]);
r.push('</a>');
}
return r.join('');
```

下一步就是收集模板里面不同的代码行，用于生成函数。通过前面介绍的方法，我们可以知道模板中有哪些占位符（译者注：或者说正则表达式的匹配项）以及它们的位置。所以，依靠一个辅助变量（*cursor*，游标），我们就能得到想要的结果。

```
var TemplateEngine = function(tpl, data) {
    var re = /<%([^%>]+)?%>/g,
        code = 'var r=[];\n',
        cursor = 0;
    var add = function(line) {
```

```
            code += 'r.push("' + line.replace(/"/g, '\\"') + '");\n';
        }
        while(match = re.exec(tpl)) {
            add(tpl.slice(cursor, match.index));
            add(match[1]);
            cursor = match.index + match[0].length;
        }
        add(tpl.substr(cursor, tpl.length - cursor));
        code += 'return r.join("");'; // <-- return the result
        console.log(code);
        return tpl;
    }
    var template = '<p>Hello, my name is <%this.name%>. I\'m <%this.profile.age%> years old.</p>';
    console.log(TemplateEngine(template, {
        name: "Krasimir Tsonev",
        profile: { age: 29 }
    }));
```

上述代码中的变量 *code* 保存了函数体。开头的部分定义了一个数组。游标 *cursor* 告诉我们当前解析到了模板中的哪个位置。我们需要依靠它来遍历整个模板字符串。此外还有个函数 *add*，它负责把解析出来的代码行添加到变量 *code* 中去。有一个地方需要特别注意，那就是需要把 *code* 包含的双引号字符进行转义（*escape*）。否则生成的函数代码会出错。如果我们运行上面的代码，我们会在控制台里面看见如下的内容：

```
var r=[];
r.push("<p>Hello, my name is ");
r.push("this.name");
r.push(". I'm ");
r.push("this.profile.age");
return r.join("");
```

等等，貌似不太对啊，*this.name* 和 *this.profile.age* 不应该有引号啊，再来改改。

```
var add = function(line, js) {
    js? code += 'r.push(' + line + ');\n' :
        code += 'r.push("' + line.replace(/"/g, '\\"') + '");\n';
}
while(match = re.exec(tpl)) {
    add(tpl.slice(cursor, match.index));
    add(match[1], true); // <-- say that this is actually valid js
    cursor = match.index + match[0].length;
}
```

占位符的内容和一个布尔值一起作为参数传给 *add* 函数，用作区分。这样就能生成我们想要的函数体了。

```
var r=[];
r.push("<p>Hello, my name is ");
r.push(this.name);
r.push(". I'm ");
r.push(this.profile.age);
return r.join("");
```

剩下来要做的就是创建函数并且执行它。因此，在模板引擎的最后，把原本返回模板字符串的语句替换成如下的内容：

```
return new Function(code.replace(/[\r\t\n]/g, '')).apply(data);
```

我们甚至不需要显式地传参数给这个函数。我们使用 *apply* 方法来调用它。它会自动设定函数执行的上下文。这就是为什么我们能在函数里面使用 *this.name*。这里 *this* 指向 *data* 对象。

模板引擎接近完成了，不过还有一点，我们需要支持更多复杂的语句，比如条件判断和循环。我们接着上面的例子继续写。

```
var template =
'My skills:' +
'<%for(var index in this.skills) {%>' +
'<a href="#"><%this.skills[index]%></a>' +
'<%}%>';
console.log(TemplateEngine(template, {
    skills: ["js", "html", "css"]
}));
```

这里会产生一个异常，*Uncaught SyntaxError: Unexpected token for*。如果我们调试一下，把 *code* 变量打印出来，我们就能发现问题所在。

```
var r=[];
r.push("My skills:");
r.push(for(var index in this.skills) {});
r.push("<a href=\"\">");
r.push(this.skills[index]);
r.push("</a>");
r.push(});
r.push("");
```

```
9.  return r.join("");
```

带有 *for* 循环的那一行不应该被直接放到数组里面，而是应该作为脚本的一部分直接运行。所以我们在把内容添加到 *code* 变量之前还要多做一个判断。

```
1.  var re = /<%([^%>]+)?%>/g,
2.      reExp = /(^( )?(if|for|else|switch|case|break|{|}))(.*)?/g,
3.      code = 'var r=[];\n',
4.      cursor = 0;
5.  var add = function(line, js) {
6.      js? code += line.match(reExp) ? line + '\n' : 'r.push(' + line +
    ');\n' :
7.          code += 'r.push("' + line.replace(/"/g, '\\"') + '");\n';
8.  }
```

这里我们新增加了一个正则表达式。它会判断代码中是否包含 *if*、*for*、*else* 等等关键字。如果有的话就直接添加到脚本代码中去，否则就添加到数组中去。运行结果如下：

```
1.  var r=[];
2.  r.push("My skills:");
3.  for(var index in this.skills) {
4.  r.push("<a href=\"#\">");
5.  r.push(this.skills[index]);
6.  r.push("</a>");
7.  }
8.  r.push("");
9.  return r.join("");
```

当然，编译出来的结果也是对的。

```
1.  My skills:<a href="#">js</a><a href="#">html</a><a href="#">css</a>
```

最后一个改进可以使我们的模板引擎更为强大。我们可以直接在模板中使用复杂逻辑，例如：

```
1.  var template =
2.  'My skills:' +
3.  '<%if(this.showSkills) {%>' +
4.      '<%for(var index in this.skills) {%>' +
5.      '<a href="#"><%this.skills[index]%></a>' +
6.      '<%}%>' +
7.  '<%} else {%>' +
8.      '<p>none</p>' +
```

```
    '<%}%>';
console.log(TemplateEngine(template, {
    skills: ["js", "html", "css"],
    showSkills: true
}));
```

除了上面说的改进，我还对代码本身做了些优化，最终版本如下：

```
 var TemplateEngine = function(html, options) {
    var re = /<%([^%>]+)?%>/g, reExp = /(^( )?(if|for|else|switch|case|break|{|}))(.*)?/g, code = 'var r=[];\n', cursor = 0;
    var add = function(line, js) {
        js? (code += line.match(reExp) ? line + '\n' : 'r.push(' + line + ');\n') :
            (code += line != '' ? 'r.push("' + line.replace(/"/g, '\\"') + '");\n' : '');
        return add;
    }
    while(match = re.exec(html)) {
        add(html.slice(cursor, match.index))(match[1], true);
        cursor = match.index + match[0].length;
    }
    add(html.substr(cursor, html.length - cursor));
    code += 'return r.join("");';
    return new Function(code.replace(/[\r\t\n]/g, '')).apply(options);
}
```

代码比我预想的还要少，只有区区 15 行！