Juicer – 一个 Javascript 模板引擎的实现和优化

2012 年 04 月 23 日 | 前端技术

让我们从一段代码说起，假设有一段这样的 JSON 数据：

```
var json={
        name:"流火",
        blog:"ued.taobao.com"
};
```

我们需要根据这段 JSON 生成这样的 HTML 代码：

```
流火 (blog: ued.taobao.com)
```

传统的 Javascript 代码一定是这个样子：

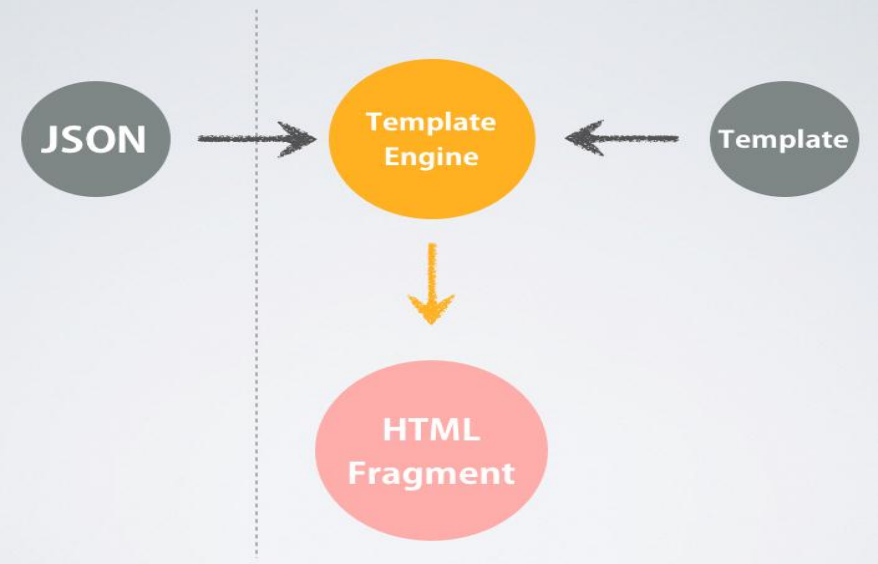
```
var html;
html=''+json.name+' (blog: '+json.blog+')';
```

不言而喻，这样的代码混杂了 html 结构和代码逻辑，而且代码不具可读性，不便于后期维护，于是便有了这样一个函数：

```
function sub(str,data) {
    return str
        .replace(/{(.*?)}/igm,function($,$1) {
            return data[$1]?data[$1]:$;
        });
}
```

有了这个函数，我们拼接字符串的工作就可以简化为：

```
var tpl='{name} (blog: {blog})';
var html=sub(tpl,json);
```



看到这里，不用我多说，我想通过这个例子直观的展现出前端模板引擎的好处所在，这么做能够完全剥离 `html` 和代码逻辑，便于多人协作和后期的代码维护。当然，当我们的业务逻辑需要对数据源进行循环遍历，`if` 判断等的时候，这个简明的函数很显然并不能满足我们的需求，于是便有了如今这市面上众多的模板引擎，诸如 `Mustache`，`jQuery tmpl`，`Kissy template`，`ejs`，`doT`，`nTenjin`，`etc.`

“如无必要，勿增实体。” 这是著名的奥卡姆剃须刀法则，简单的说就是避免重复造轮子。那么就会有童鞋质疑，既然已然有这么多现成的东西可用，为什么还要重新打造一个呢？我个人认为一个完善的模板引擎应该兼顾这几点：

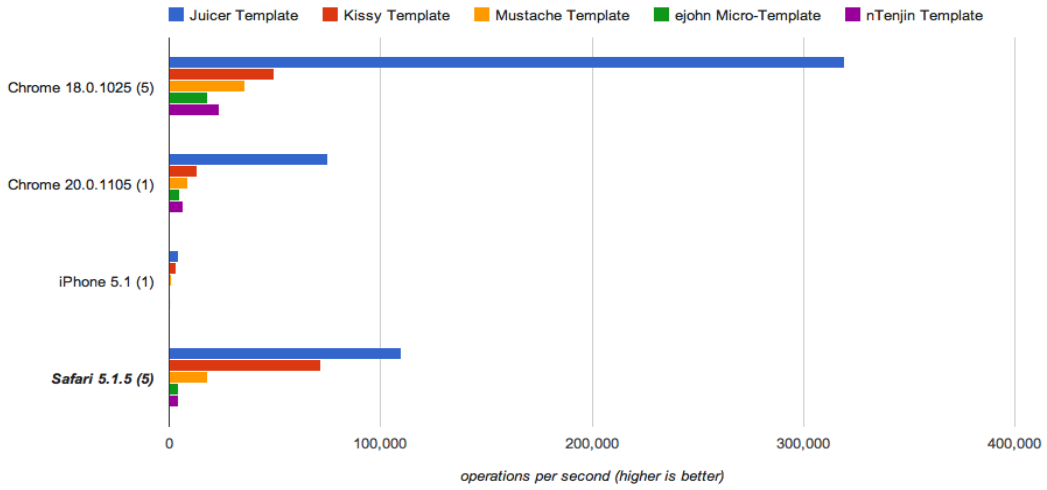
- 语法简明
- 执行效率高
- 安全性
- 错误处理机制
- 多语言通用性

而市面上现有的模板引擎没有做到兼顾以上几点，比如 `Mustache` 支持多种语言，通用性不错，不过性能稍差，而且语法不支持高级特性，例如遍历的时候无法做 `if` 判断，也无法获

得 index 索引值，jQuery tmpl 依赖 jQuery，缺乏可移植性，Kissy template 虽然依赖 Kissy，不过性能和语法都值得推荐，doT/nTenjin 性能和灵活性都很不错，但是语法需要用原生的 js 来写，写好的模板代码可读性稍差。

鱼和熊掌不可兼得，语法的处理，安全性的输出过滤和错误处理机制的引入在一定程度上都会或多或少降低模板引擎的性能，因此就需要我们权衡。Juicer 在实现上首先将性能看做第一个重要的指标，毕竟性能好坏直接影响用户的感知，同时兼顾了安全性和错误处理机制（即便这样会导致性能的略微下降）。

首先来看下 jsperf 上同几个主流模板引擎的性能对比。



前端模板引擎 100 条数据 × 10000 次渲染测试

tmpl	tmpl2	YayaTemplate	ejs	juicer	doT
2005	475	456	391	87	153
1883	481	453	386	82	145
1904	474	458	382	79	144
1898	473	498	384	80	142
1870	478	511	384	78	146

可以看到，性能上比传统模板引擎均有提升，下边的介绍主要从语法、安全性和错误处理，以及如何使用这几个方面介绍下 Juicer.

a. 语法

- 循环 {@each}...{@/each}
- 判断 {@if}...{@else if}...{@else}...{@/if}
- 变量（支持函数）${varname|function}
- 注释 {# comment here}

详细的语法请参考 Juicer Docs.

b．安全性

安全性，简单地说就是对输出数据在输出前进行一次转义过滤，避免 XSS 这样的脚本注入攻击，简单扫下盲，举个 XSS 的例子。

```
var json={
        output:'alert("XSS");'
};
```

如果 JSON 数据是第三方接口返回或者含有用户输入（像 BBS、评价）的内容，我们如果赤裸裸的将 output 写到页面上就会执行恶意的 js 代码，所以 Juicer 默认是对数据输出做了安全转义的，当然如果不想被转义，可以使用$${varname}。

```
juicer.to_html('${output}',json);
//输出：&lt;script&gt;alert("XSS");&lt;/script&gt;

juicer.to_html('$${output}',json);
//输出：<script>alert("XSS");</script>
```

c．错误处理

如果没有错误处理，当模板引擎编译(Compile)或者渲染(Render)出错时候就会引起后续 js 代码停止执行，可想而知，如果因为一个逗号或者 JSON 数据的偶发错误导致整个页面挂掉，是我们不能接受的。但是 Juicer 在遇到这些错误的时候不会影响后续代码的执行，只会在控制台打出一句警告(Warn)告知开发者模板解析出现错误。

```
juicer.to_html('${varname,,,,,,,}',json);
alert('hello, juicer!');
```

执行上边的代码就会看到控制台打出的“Juicer Compile Exception: Unexpected token ,” ，但是不会因为错误导致后续的 alert 被阻塞掉。

实现原理

Juicer 对一个模板的编译和渲染的过程主要有以下几个步骤：

- 1、对模板代码进行语法分析
- 2、分析后生成原生的 Javascript 代码字符串
- 3、将生成的代码转为可重用的 Function (Compiled Template)

```
var json={
```

```
        list:[
                {name:"benben"},
                {name:"liuhuo"}
        ]
};
var tpl='{@each data.list as value,key}$${value.name}{@/each}';
var compiled_tpl=juicer.compile(tpl,{errorhandling:false});
```

我们通过 compiled_tpl.render.toString()看下编译后的代码：

```
function anonymous(data) {
        var data = data || {};
        var out = '';
        out += '';
        for (var i0 = 0, l = data.list.length; i0 < l; i0++) {
                var value = data.list[i0];
                var key = i0;
                out += '';
                out += ((value.name));
                out += '';
        }
        out += '';
        return out;
}
```

是不是已经明白了 Juicer 的原理？这个编译后的函数就会每次帮我们完成从数据到 html 代码的拼装操作。

这里有几点优化的地方值得分享下：

- 1、using += instead of array.push
- 2、avoid using with {}
- 3、cache the compiled template (function)

这几点优化在大数据量循环渲染时候性能提升显著，不过正因为放弃了 with{}语句，所以 JSON 数据外层必须指定“data.”前缀，如果你觉得这点性能的提升不重要，也可以在 options 中指定 loose:true（松散模式），这样就可以省去 data. 前缀。

最后介绍下 Options 配置项，左侧为参数默认值。

```
{
        cache:true/false,
        loose:false/true,
        errorhandling:true/false
```

```
}
```

cache 默认为 true，即同一个模板编译后是否被 juicer 缓存，也就是说如果缓存开启的情况下，同一个模板第一次编译后，为了缩短耗时提升性能，后续不会再次执行编译的操作而是直接从缓存中去取编译好的模板函数。
Juicer 的 API. Juicer 有两种使用方法，一种是通过

```
juicer.to_html(tpl,data,options);
```

直接根据提供的数据将模板转为 html 代码，另一种是通过 compile 方法先将模板编译好，在需要的时候再对模板进行数据的 Render 操作：

```
var compiled_tpl=juicer.compile(tpl,options);
compiled_tpl.render(data);
```

最后附上 Juicer 的项目地址，上边有详细的文档和 Demo 代码。
http://juicer.name