圣诞节，把你的 JavaScript 代码都装扮成圣诞树吧

2015-12-25 07:43　评论: 2 收藏: 1

参考原文：http://f2e.souche.com/blog/sheng-dan-jie-ba-wang-zhan-s...　作者： 小芋头君

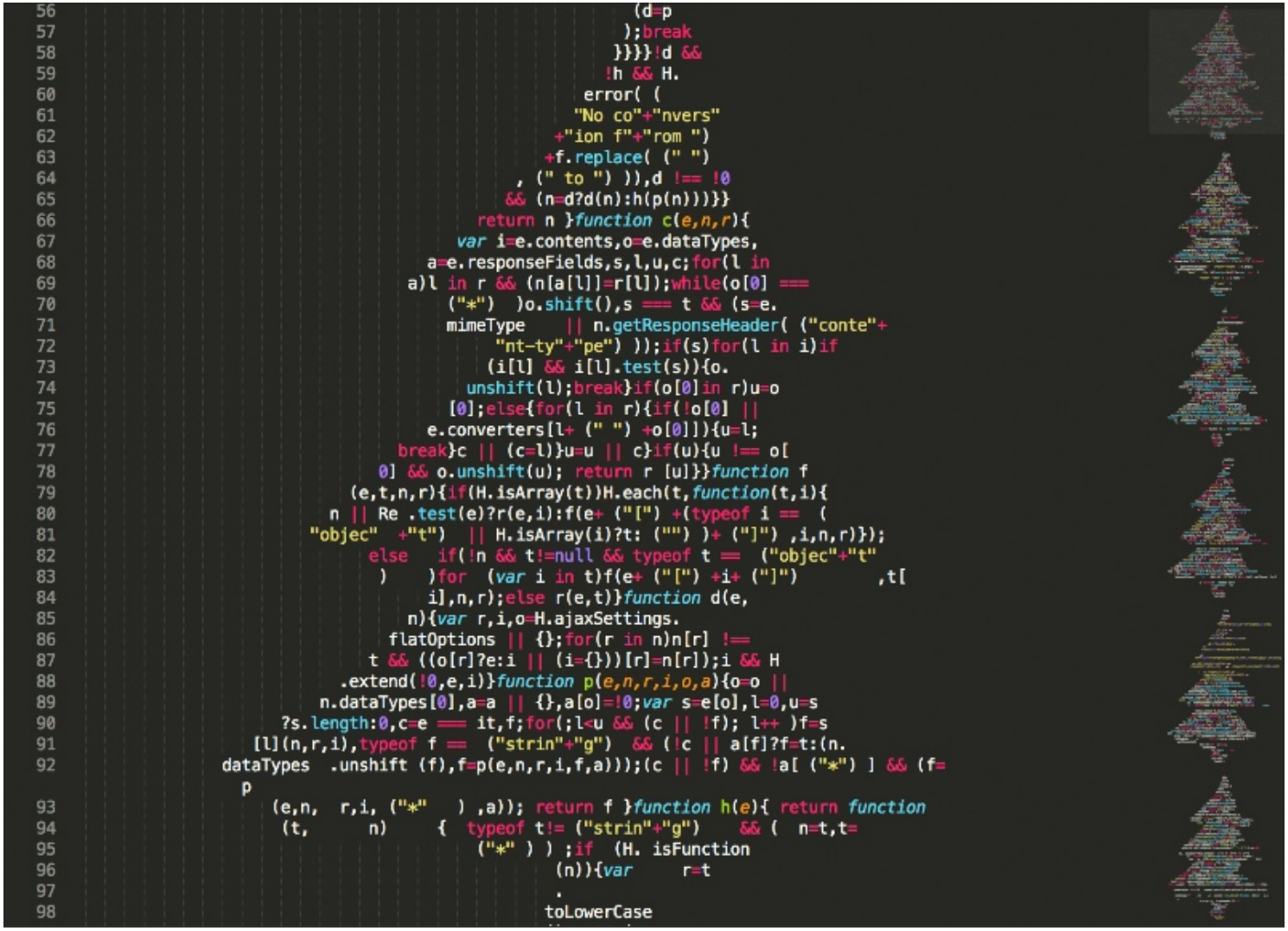
> 用开源项目将你的 JavaScript 变成圣诞树吧！

效果的话，可以去看一下我们公司的官网（**http://www.souche.com <http://www.souche.com/>** ），里面涉及到的js代码在今天大部分被临时替换成了圣诞树，打开每个js代码即可看到效果。

其实也不神奇，我们使用了自己写的一个nodejs库，如果您要实现这样的效果，只需要按照下面第一章节的方法即可。当然您也可以在线压缩代码：**http://f2e.souche.com/cheniu/js2image.html <http://f2e.souche.com/cheniu/js2image.html>**

下面分两章节，分别讲解如何使用js2image这个库 和 js2image这个库的原理。

- github地址：**https://github.com/xinyu198736/js2image <https://github.com/xinyu198736/js2image>** ps:求star
- 在线转换地址：**http://f2e.souche.com/cheniu/js2image.html <http://f2e.souche.com/cheniu/js2image.html>**



js2image使用

github地址：**https://github.com/xinyu198736/js2image <https://github.com/xinyu198736/js2image>** 欢迎送上star或者follow。

js2image主要有两个比较特殊的特性：

1. 将任意js源码 压缩成 用代码堆砌成图形的最终代码。例如圣诞树，圣诞老人，代码和图片都可以自定义。
2. 压缩后的js代码格式虽然被破坏，但是仍然可以运行。这个是关键点！

压缩后的示例可以查看这些js（均来自搜车官网）：

- **http://assets.souche.com/assets/js/souche.js <http://assets.souche.com/assets/js/souche.js>** souche主脚本
- **http://assets.souche.com/assets/js/lib/jquery-1.7.1.min.js <http://assets.souche.com/assets/js/lib/jquery-1.7.1.min.js>** jquery 1.7.1
- **http://assets.souche.com/assets/js/lib/mustache.js <http://assets.souche.com/assets/js/lib/mustache.js>** mustache

使用方式很简单：

```
npm install js2image -g;
```

然后在存在js的文件夹中执行：

```
js2image -s ./resource/jquery.js
```

或者针对某个目录下所有的js执行（慎用），会深度遍历此目录里所有的js文件然后压缩出.xmas.js后缀的结果文件。

```
js2image -s ./resource/
```

即可生成一个对应的 **.xmas.js 的文件。

如果要将js2image集成到gulp或者其他nodes项目中，可以使用用模块的形式：

```
var Js2Image = require("js2image");//获取结果的
codeJs2Image.getCode("./resource/jquery.js","./resource/tree.png",  {}).then(function(code){
    console.log(code);
})
```

更多的信息可以参照github上的文档。

如果只是要使用这个效果，看到这里就ok啦，下面讲解这个库的原理，有些地方可能比较绕。

js2image实现原理

js2image的实现从宏观来说，大体只有3个要点。

1. 从图片生成字符画，这个有现成的库。
2. 把js代码分割成一小块一小块，尽量小，然后用逐行填充的方式分别替换到上一步生成的字符画里去。
3. js代码中有诸多不能分开的语法，分块的时候要把这些语法保留在一个块内。这个是这个库的难点所在，也是代码最多最绕的地方。

稍有想法的同学估计看到这里基本已经明白是怎么回事了，下面一一讲解这3个要点。

① 从图片生成2值得字符画

这里用到了一个现成的npm包：image-to-ascii 。这个库的作用是用指定的字符来还原一个图像。而我们用这个库来生成一个用 🐻字符和空格分别表示黑和白的字符画，然后将字符画的每一行分解成数组的一个元素，供第二步使用，这就是我们中间生成的一个struct，代码见 `utils/image-to-struct.js`

② 分割js源码成尽量小的小块。

这是非常重要的一步，js代码具体可以分解成多细的小块呢？

看下面一段代码：

```
                        !function
                        (e,t
                        ){ (
                       "objec"
                      +"t")  ==
                     typeof
                    module &&  (
                   "objec"+"t")
                  == typeof module
                .exports?module.
              exports=e.document?t(e
            ,!0):function(e){if(!e.
          document) throw new Error (
        ("jQuer"+"y req"+"uires"+" a wi"
    +"ndow "+"with "+"a doc"+"ument") )
; return t (e)}:t(e)}( ("undef"+"ined")
!=typeof    window ?window:this,function(e,t){var
```

这是jQuery开始的一段代码,可以看到，大部分操作符都允许中间插入任意多的空格或者换行，我们正是利用这一特性将js代码解肢，然后拼接成任意形状的图片。

核心代码其实就是一个正则，我们用这个正则把js源码解构成一个数组，然后后续根据每行需要的字符数，从这个数组里不断取片段出来拼

接。

```
//分离代码，以可分割单位拆分成数组。var lines = hold_code.replace(/([^a-zA-Z_0-9=!|&$])/g,"\n$1\n").split("\n");
//有了这个lines数组之后后面就简单了，根据第一步里生成的struct不断遍历从lines抽取代码填充到struct里即可生成最终的代码：
while(lines.length>0){
    //循环往struct里填充代码
    struct.forEach(function(s){
        var chars_arr = s.replace(/ +/g," ");//一行有多组分离的*****
        var r = s;
        chars_arr.split(/ +/).forEach(function(chars){
            if(chars.length == 0){
                return;
            }
            var char_count = chars.length;
            //从lines里取出char_count数量的代码来填充,不一定精准,要确保断行正确
            var l = pickFromLines(lines,char_count);

            r = r.replace(chars,function(){
                return l;
            })
        })
        result += r+"\n"
    })

}
```

③ 保留不可分割的语法

注意：到了这一步，还很早，你分解出来的代码是无法运行的，很多不能换行和加空格的代码都被你分开了，自然会报错，那如何处理这些情况呢？

这一步，我们做的工作就是：

在执行代码分拆之前，提取出代码里所有不可分割的语法，将他们保留在一个对象中，并且在源代码中用占位符替代这些语法，然后让占位符参与上个步骤的分离，因为占位符是一个完整的连字符变量，所以不会被分割。在分割完成之后，我们再把这些占位符替换回来即可。

不过，在js中哪些语法必须是连接在一起才能正常运行的呢？

这里总结下：

1. 字符串不可分割 包括双引号单引号内的内容。
2. 正则表达式绝对不可分割 正则里的转义很难处理，这是这个算法里的难点。
3. 运算操作符 包括2字符的3字符的 例如 以下两种

```
var double_operator = ["==", ">=", "<=", "+=", "-=", "*=", "/=", "%=", "++", "--", "&&", "||", ">>", "<<"]
var three_operator = ['===', '!==']
```

一些固定语法，可以用正则表达，如下：

```
var reg_operator = [
    {
        start:"return",
        reg:/^return[^a-zA-Z_0-1"'][a-zA-Z_0-1.]+/
        // return 0.1 或者 return function 或者return aaabb
```

```
    },
    {
        start:"return\"",
        reg:/^return".*?"/ // return "d" 或者 return ""
    },
    {
        start:"return\'",
        reg:/^return'.*?'/  // return 'd' 或者 return ''
    },
    {
        start:"throw",
        reg:/^throw [a-zA-Z_0-1]+?/ //throw new 或者 throw obj
    }
]
```

小数点语法，例如 0.01 因为之前我们用点号来分割代码的，但是这里的点号不能作为分割符使用，需要保留前后数字跟点号在一行 其他语法，例如 value++ 之类的语法，变量和操作符之间不可分割。 那我们如何从源代码中解析出这些语法，然后做处理呢？

核心代码均在 `utils/keep-line.js` 中

核心算法，事实上是通过一个对字符串的遍历来完成的，然后在遍历每个字符的时候都会判断是否进入某个逻辑来跳跃处理。

例如，判断出当前在双引号内，则进入字符串提取逻辑，一直到字符串结束的时候再继续正常的遍历。

其他操作符和正则表达式的算法也是类似，不过里面很多细节需要处理，例如转义字符之类的。

有些比较特殊的，例如小数点语法的提取，在判断到当前字符是点号之后，需要往前和向后循环查找数字，然后把整个语法找出来。

这里不细讲，在keep-line.js 这个文件中又一大坨代码做这个事情的。

④ 字符串解构

做到这一步的时候，其实效果已经很不错了，也可以保证代码的可运行，但是代码里有些字符串很长，他们总是会被被保留在一行里，这样就造成他会影响一些图案的边缘的准确性（代码分离原则是越细越好，就是为这个考虑）。

我们如何处理呢，那就是将字符串解构，以5个为单位将字符串分离成小块。

这里有两个比较重要的问题需要处理；

1. 字符串内的转义字符如何处理，还有一些特殊字符，例如0x01这样的字符，这些字符不能被分离到不同的字符串里，所以分离的时候要保留这些字符串的完整性。
2. 字符串分离成小字符串，然后用+号拼接起来，不过要注意操作符优先级的问题，所以所有分离后的字符串，都要用括号包起来，让这个+号的优先级永远最高。

   具体算法见 keep-line.js 中的 splitDoubleQuot （分离双引号字符串）。

结语

至此，整个应用就完成了，可以顺利完成从任意js和图像生成图形代码了。

再说一遍项目开源地址：**https://github.com/xinyu198736/js2image <https://github.com/xinyu198736/js2image>** 欢迎star，顺便follow下楼主就更开心了。