CSSCheckStyle——CSS 的解析、检查、修复和压缩

王致富

2012-12-5

前言

《人人FED CSS编码规范》中，对CSS的编写方式进行了严格的约束和说明，但是如何在实际开发过程中，确保代码能够严格遵循规范的要求，是规范制定者必须要考虑的问题。

人为遵守规范，是规范制定的初衷，也是最终目标，但是在规范没有变成习惯之前，人为遵守终归是“不靠谱”的。因此：“无工具，不规范”，有了规范，必定需要配套的工具来确保规范的执行。

说起来也比较奇怪，同样是前端开发的三大语言之一，CSS的检查工具备受冷落，而JS的代码规范检查工具则层出不穷。JSLint/JSHint/gjslint等神器不断涌现，且检查的严格程度令人瞠目结舌，但是CSS的检查工具却寥寥无几，仅有的一些检查工具，也只对CSS的取值、是否符合W3C规范进行了检查，对于代码风格只字不提，而各大互联网公司的开发规范中，对于代码风格是有严格约束的。因此：

> 一个严格按照规范检查 CSS 代码风格问题的工具是必不可少的

另外，CSS的检查、修复、排序、合并、压缩等功能，是相互关联的，并不是相互孤立的。比如：只有进行了完整的CSS代码修复与合并，才能最大限度的实现压缩。而目前已有的CSS检查、属性排序、压缩等工具，实现方式五花八门，实现语言也是多种多样，不利于把CSS相关的功能有机的整合在一起，也不利于工具的集成和功能的扩展。因此：

> 依据完善的开发规范，实现一个自动修复 CSS 代码（为开发者）、并在修复的基础上压缩代码（为浏览器）的工具，也是很有必要的。

因此，CSSCheckStyle（简称ckstyle）就应运而生了。它自主实现了CSS的解析、检查、修复与压缩，不仅能够检查出CSS的代码风格和取值问题，还能够对CSS代码进行自动修复和压缩，给出符合规范的“开发者视图”代码和高效率的“浏览器视图”代码。此外，它充分利用插件的特性，可以方便灵活的实现功能扩展。

目前此工具已经在github上开源，欢迎大家围观，并加入我们：

> **https://github.com/wangjeaf/CSSCheckStyle**

部分已有 CSS 工具的分析

为了避免重复造轮子，我们先分析一下现有的一些 CSS 相关的自动化工具，这里主要分析以下三个工具：

1、CSS 代码压缩工具： YUICompressor

2、CSS 代码检查工具： CSSLint

3、CSS 属性重排工具： CSSComb

YUICompressor

YUICompressor 是基于 Java 的 CSS 代码压缩工具，主要实现原理是：

1、将 CSS 文件读取并解析成字符串

2、利用正则表达式，替换该字符串中的“应该被替换”的内容

3、替换完成之后的字符串即为 CSS 压缩后的内容

下面是 YUICompressor 源代码的部分截图

```
// Remove the spaces after the things that should not have spaces after them.
css = css.replaceAll("([!{}:;>+\\(\\[,])\\s+", "$1");

// remove unnecessary semicolons
css = css.replaceAll(";+}", "}");

// Replace 0(px,em,%) with 0.
css = css.replaceAll("([\\s:])(0)(px|em|%|in|cm|mm|pc|pt|ex)", "$1$2");

// Replace 0 0 0 0; with 0.
css = css.replaceAll(":0 0 0 0(;|})", ":0$1");
css = css.replaceAll(":0 0 0(;|})", ":0$1");
css = css.replaceAll(":0 0(;|})", ":0$1");
```

由于 YUICompressor 的这种实现方式，导致了它必然存在一些缺陷，例如：

1、代码压缩率有限，不能实现高级的 CSS 压缩，比如下面所示的规则压缩：

```
.test {
    margin-top: 10px;
    margin-right: 10px;
    margin-bottom: 10px;
    margin-left: 10px;
}
➔
.test {margin:10px}
```

2、由于所有规则都已经在源代码中硬编码，所以除非修改核心源码，否则不能很好的实现功能扩展。

CSSLint

CSSLint 是用 JavaScript 实现的，用于检查 CSS 取值和潜在问题的工具，他使用了 Nicholas 大神的 parser-lib 作为 CSS 解析器，并按照 parser-lib 给出的 API 来编写检查规则。例如：检查一个规则是否为空的核心代码如下：

```
//initialization
init: function(parser, reporter){
    var rule = this,
        count = 0;

    parser.addListener("startrule", function(){
        count=0;
    });

    parser.addListener("property", function(){
        count++;
    });

    parser.addListener("endrule", function(event){
        var selectors = event.selectors;
        if (count === 0){
            reporter.report("Rule is empty.", selectors[0].line, selectors[0].col, rule);
        }
    });
}
```

分析其源码可以得知，CSSLint 的每一个规则都是通过监听 parser 给出的事件来进行相应的判断：

1、startrule 为规则开始

2、property 为找到一个属性时的事件

3、endrule 为一个规则结束

一旦规则结束并且没有统计到任何 property，则说明规则为空。

CSSLint 很好的利用了插件机制，实现了检查规则的灵活扩展，构建了一个良性的规则开发和运行生态，但是它也存在一些不足：

1、没有针对代码风格的检查，并且缺失部分检查规则。例如：下面的代码在 CSSLint 检查完成之后认为是没有任何问题的。

```
.test1 ul li a {
width:10px;
color:#ffffff;
      -webkit-border-radius:3px;
-moz-border-radius : 3px;
border-radius:3px
}
```

实际上，依据《人人 FED CSS 编码规范》，上述代码问题还非常多。

2、检查出来的 CSS 问题，没有实现相应的修复方法，比如 width:0px 可以简化成 width:0。需要开发者根据提示，人工修复。

3、插件的编写略显复杂，插件中的核心操作元素是一系列的事件及其监听函数，而不是需要检查的目标（比如文件、规则、属性）本身。

CSSComb

一个 CSS 规则中的属性编写顺序，会影响到浏览器的渲染效率，推荐按照“显示属性、盒模型、文本属性、其他属性”的顺序来编写。

在编写 CSS 时手动调整顺序显得过于繁琐，因此 CSSComb 就有了它的用武之地。它能够按照推荐的 CSS 属性排列顺序，将 CSS 的所有属性重排。

例如：

```
.test {
    width: 100px;       /*盒模型*/
    height: 100px;      /*盒模型*/
    border: none;       /*盒模型*/
    font-size: 16px;    /*文本属性*/
    display: none;      /*显示属性*/
}
```

经过 CSSComb 的重新排列，将会变成符合推荐顺序的排列方式：

```
.test {
    display: none;      /*显示属性*/
    width: 100px;       /*盒模型*/
    height: 100px;      /*盒模型*/
    border: none;       /*盒模型*/
    font-size: 16px;    /*文本属性*/
}
```

值得一提的是，CSSComb 提供了大量的编辑器扩展，通过简单的编辑器命令或操作，即可实现属性的批量重排，使用起来非常方便。

该工具存在的一些问题如下：

1、与 YUICompressor 一样，所有功能集于一个文件，不能很好的实现功能扩展（虽然属性排序也不需要太多扩展哈~）

2、功能相对来说比较单一（也可以说是专注哈~）

总结

不论是代码检查、属性排序，还是代码压缩，对于前端开发来说，这几个工具都是必须要使用的。但是，通过以上分析，以及下面的总结表格，很容易看出，如果要在前端开发环境中集成这些工具，并且对这些工具进行统一灵活的扩展和完善，成本非常高。

名称	功能	实现语言	CSS 解析器
YUICompressor	CSS 代码压缩	Java	无
CSSLint	CSS 代码检查	JavaScript	parser-lib(in JS)
CSSComb	CSS 属性重排	PHP	自带(in PHP)

CSSCheckStyle 需要解决的问题

综合以上的工具分析结论，CSSCheckStyle 需要解决的问题如下：

自主完成 CSS 的解析

为什么不用 parser-lib，而需要自己开发解析器呢？主要原因如下：

1、通过 CSSLint 的插件编写方式可以看出，parser-lib 给出的 API，对于编写检查规则来说相对比较复杂。

2、parser-lib 解析过程中，给出的代码风格方面的 API 不足，导致代码风格检查很难进行。

3、目前人人 FED 的每一位开发者都有 Python 环境，但是并不一定有 Java 或 nodejs，用 Python 自主开发解析器，也能减少工具推广过程中的额外配置。

因此，自主研发的 CSS 解析器，应该具备以下几个方面的特征：

1、解析过程中，要保留原始的代码信息，以便进行代码风格的检查。

2、能够预留非常方便的 API，以供插件规则来检查或修复代码。

3、按照 StyleSheet/RuleSet/Rule 的结构来生成解析内容，便于对规则进行划分，用不同类型的规则来检查不同内容。

建立方便灵活的功能扩展机制

CSSCheckStyle 需要完成的使命光荣而艰巨，并非一朝一夕之功，需要不断收集用户反馈并不断优化完善。

为了把修改和完善的成本降到最低，CSSCheckStyle 采用插件的机制，来扩展工具的功能，每一个插件完成一种或一组规则的检查和修复，这样一来，一旦发现某种规则缺失，或者实现过程中出现问题，添加或修改插件代码即可，无需改动整个工具的框架结构。

目前 CSSCheckStyle 实现的所有检查和修复功能，都是在插件中完成的。

检查 CSS 的代码风格问题

对于代码风格问题的检查，目前并没有比较好的检查工具，因此为了确保规范的严格执行，此功能必须包含。

由于 CSS 解析过程中，保留了代码的原始信息，因此代码风格问题的检查就比较顺利了。

检查并修复 CSS 的代码问题

一般情况下 如果能够检查出一种代码问题 就意味着知道这个问题应该如何修复。在 CSSCheckStyle 的自动修复实现过程中，大多数问题是可以自动修复的，比如：将#FFFFFF 替换为#FFF，但是也有很多问题是无法自动修复的，比如：在 CSS 代码中使用了!important，如果自动修复这个问题，将影响最终的功能实现。

因此，自动修复分为以下两种情况：

1、修复不压缩

修复不压缩，说明最终生成的是“开发者视图”的 CSS，此时，如果遇到不能自动修复的问题，则自动为该代码生成一个特殊注释：/*TOFIX*/，标识此处无法自动修复，请人工修复。

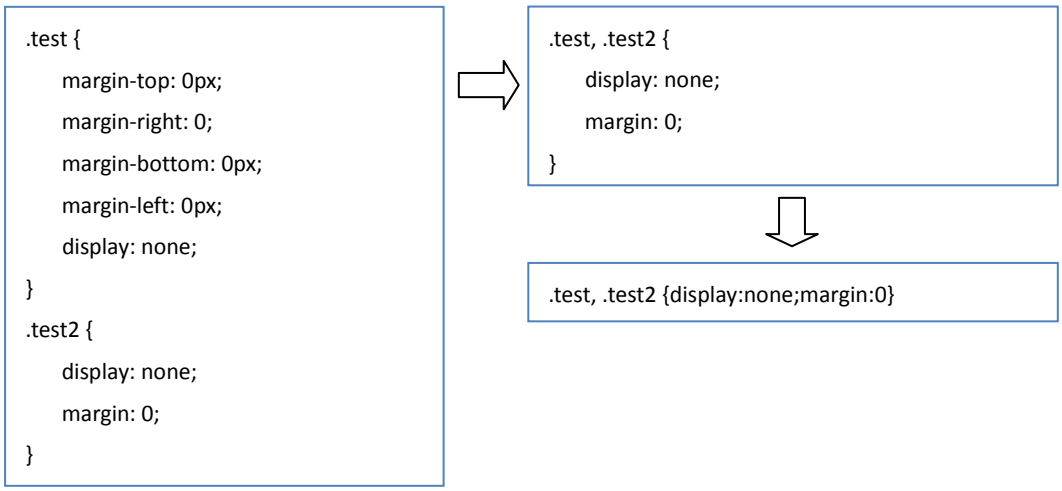
2、修复且压缩

修复且压缩，意味着最终生成的是“浏览器视图”的 CSS，在这种视图中，将按照所有已知的压缩规则，最大限度的压缩 CSS 代码，因此不会生成特殊注释，而是对无法自动修复的问题采取“鸵鸟策略”，对此问题“置之不理”。

在修复的基础上压缩代码

只有代码做到了足够的精简、规范，才能最大限度的实现压缩。

下面是一个典型的示例：



针对以上代码：

1、只有对每一个 Rule 采用“0 后的单位可以省略”的规则，才能确保所有的 margin-*取值一致，便于后续合并；

2、只有对每一个 RuleSet 采用“合并 margin-*为 margin”的规则，才能确保四个 margin-*最终合并成一个。

3、只有对每一个 RuleSet 采用“按照推荐的顺序编写属性”的规则，才能确保 display 和 margin 的顺序符合规范，并且为“合并相同属性的规则”提供便利。

4、只有对整个 StyleSheet 采用“合并相同属性的规则”的规则，才能最大限度减少重复代码。

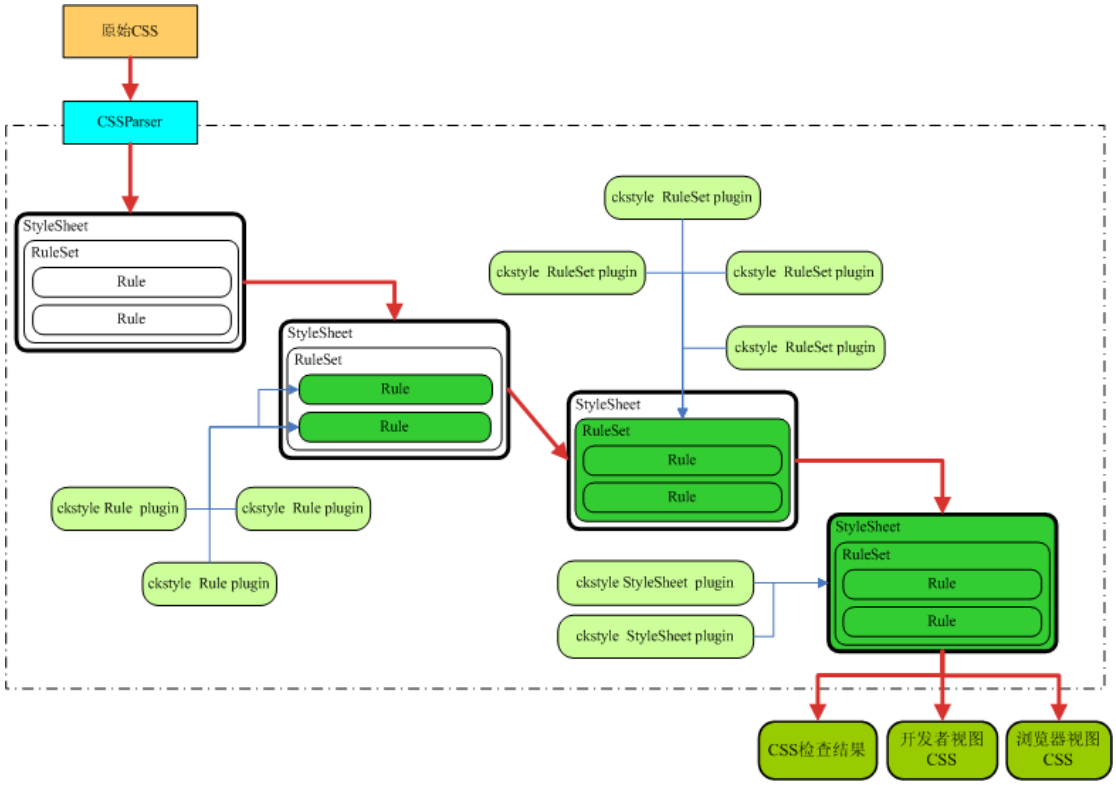
通过一系列规则的自动修复，最终达到上图右下角的合并和压缩效果。

可以看出，

> 只有代码足够精简、规范，才能最大限度的实现压缩

CSSCheckStyle 的处理方案

CSSCheckStyle 的处理流程如下图所示：



说明：

1、CSSParser 通过逐字解析 CSS 文本内容，建立 StyleSheet/RuleSet/Rule 的对象结构，保存解析后的属性值，并完全保留原始代码的缩进、空格、回车等代码风格信息。

2、ckstyle Rule Plugin，包含了针对某一种属性级别的代码规范的检查器实现，它的 check 方法将对每一个 Rule 进行对应的检查，如果是修复或压缩，则将使用其 fix 方法进行对应的修复。

3、其他两种 Plugin 与 ckstyle Rule Plugin 类似，区别在于：ckstyle RuleSet Plugin 针对的是规则级别的，而 ckstyle StyleSheet Plugin 针对的是整个样式表文件级别的。

4、任何与规范、检查、修复相关的内容都是一个 Plugin，每一个 Plugin，都可以包含检查和修复两个部分的功能；代码的压缩和格式化，由每一个 StyleSheet/RuleSet/Rule 实体来自行承担。

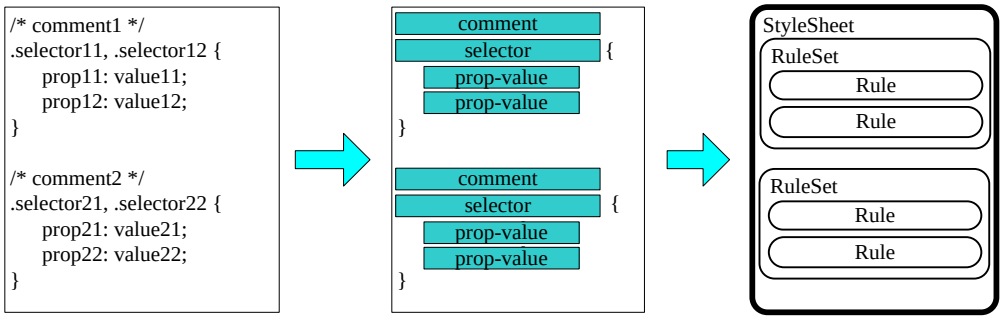
5、所有 Plugin 检查或修复完毕以后，将产生最终的处理结果。如果是检查，则会给出检查结果说明；如果是修复，则会给出修复后并格式化完毕的“开发者视图”代码；如果是压缩，则会在修复后进行最小化输出，给出“浏览器视图”代码。

具体细节将会在后续的章节中进行详细的描述。

CSS 的解析

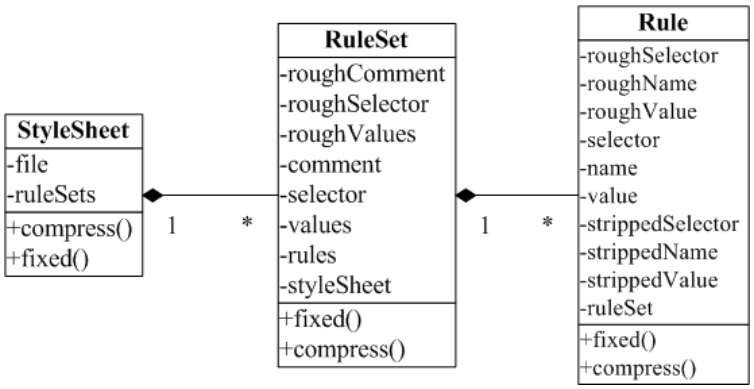
相比于 JavaScript 或 Java 等语言的复杂语法，CSS 的编写方式相对来说比较固定，解析器实现起来也相对简单。因此，本工具实现了一个简单的 CSS 解析器。

一般情况下，CSS 的整体解析和处理流程如下图所示：



在解析完成之后，StyleSheet/RuleSet/Rule 等实体类中保存了 CSS 解析过程中的原始代码和清理后的代码两部分信息，前者用于检查代码风格，后者用于检查取值。

三种实体类的类图及其关系简图如下所示：



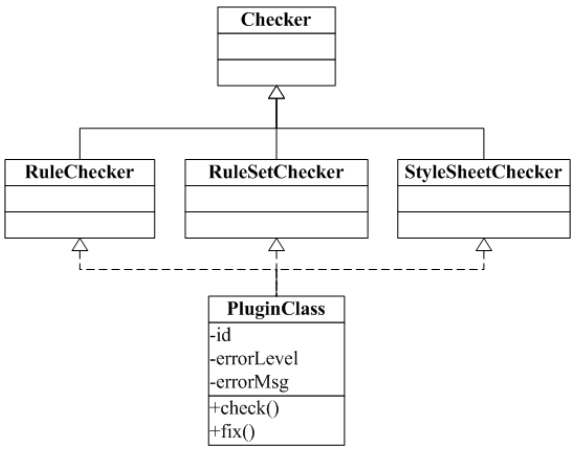
每一个实体类中都包含 compress 和 fixed 两个方法，前者用于生成压缩后的代码，后者用于生成修复后的代码。

基于插件的检查和修复机制

为了最大限度的实现灵活扩展和可插拔，本工具采用插件机制来加载需要的规则检查类，每一个插件类对应一种或一组检查规则，同时，每一个插件类包含 check 和 fix 方法，分别用于检查和自动修复。

插件类的结构和规范

插件类的结构和继承关系简图如下所示：



CSSCheckStyle 的所有与 CSS 检查、修复有关的功能，都将通过插件的形式来完成。插件类 PluginClass 有以下几个约束：

1、插件类文件必须放置在 plugins 目录下

2、插件类的类名必须与插件的文件名相同

3、插件类必须按照检查的类型，继承自 RuleChecker/RuleSetChecker/StyleSheetChecker 当中的一个

4、插件类必须包含 id（规则唯一标识）、errorLevel（错误级别）和 errorMsg（错误消息）三个属性，id 不能与其他插件类重复。

5、插件中可以（并不必须）包含 check 方法（用于检查代码问题）和 fix 方法（用于自动修复），其中：

5.1 check 方法和 fix 方法拥有相同的参数，第一个为 self（Python 类实例），第二个为该插件类需要操作的实体类，例如 rule/ruleSet/styleSheet，第三个为全局配置对象。

5.2 check 方法的必须返回：True（检查通过）/ False（检查不通过）/ Array（错误消息数组）

5.3 为了避免自动修复的相互覆盖，fix 方法中统一针对 fixedXxxx 的属性进行操作，比如 Rule 实体的 fixedName，fixedValue 等。

6、errorMsg 中可以包含${selector}、${name}、${value}，在错误信息解析时将自动替换

以“0 后的单位可以省略”的规则为例，该规则对应的插件代码如下：

```
from Base import *

class FEDNoUnitAfterZero(RuleChecker):
    def __init__(self):
        self.id = 'del-unit-after-zero'
        self.errorLevel = ERROR_LEVEL.WARNING
        self.errorMsg = 'unit should be removed when meet 0 in "${selector}"'

    def check(self, rule, config):
        values = rule.value.split(' ')
        for v in values:
            if self._startsWithZero(v.strip()):
                return False
        return True

    def fix(self, rule, config):
        fixed = rule.fixedValue
        collector = []
        for v in rule.fixedValue.split(' '):
            v = v.strip()
            if self._startsWithZero(v):
                collector.append('0')
            else:
                collector.append(v)
        rule.fixedValue = ' '.join(collector)

    def _startsWithZero(self, value):
        return value.startswith('0') and value != '0' and value[1] != '.'
```

说明：

1、插件的文件名为：FEDNoUnitAfterZero.py，放置在 ckstyle/plugins 目录下

2、该文件中包含一个继承自 RuleChecker 的类，该类名为 FEDNoUnitAfterZero

3、该类包含的 id 为 del-unit-after-zero ,errorLevel 为 WARNING 级别 ,同时包含错误消息设置。

4、此插件类中包含了 check 方法和 fix 方法 ,check 方法在找到类似于“ 0px” “0em” 的情况下，返回 False ,否则返回 True ;fix 方法则针对于 Rule 实体的 fixedValue 属性 ,用“0” 替换“0px” 或“0em” 等值，实现修复。

插件的加载和使用

只有了解了插件的加载和使用过程，才能更好的理解以上插件开发规范的由来和目的。

按照插件的开发规范，所有的插件类都放置在 plugins 目录下，且插件类的具体开发内容也必须遵守上述规范。

有了上述规范，就可以通过以下几个步骤，加载插件并使用插件的有关功能：

1、遍历 plugins 目录下的所有 Python 文件（Base.py 和 helper.py 除外）

2、根据 Python 文件名获取文件内的同名类，该类即为插件类

3、按照配置信息（如命令行参数、配置文件参数），构造插件类的实例，并按照类型分类，保存备用。

4、当需要检查 Rule/RuleSet/StyleSheet 时，使用对应插件的 check 方法来检查代码问题，fix 方法来修复代码问题即可，如果插件不存在 check 或 fix 方法，则不进行对应操作。

插件的开发

了解了插件类的结构和插件的加载使用，对于插件的开发就应该有一定的认识了。如果您想为 CSSCheckStyle 开发一款插件，则您必须：

1、按照插件类的结构和规范，编写插件类文件

2、把此文件放置在 ckstyle/plugins 目录下

3、在 tests/unit 目录下，为您的插件类添加详细的单元测试，单元测试的编写方式请参见“单元测试”小节。

非常期待能够在 plugins 目录下看到您的作品。

现有插件及其说明

按照《人人 FED CSS 编码规范》的要求，目前已有的一些插件及其相关说明如下：

插件 ID	插件类名	插件检查的规范
hexadecimal-color	FEDHexColorShouldUpper	16 进制颜色，大写，并且尽量省略
no-font-family	FEDCanNotSetFontFamily	不允许业务代码设置字体
combine-into-one	FEDCombineInToOne	将可以合并的样式设置合并
comment-length	FEDCommentLengthLessThan80	注释长度不允许超过 80 个字符
css3-with-prefix	FEDCss3PropPrefix	css3 前缀相关检查
css3-prop-spaces	FEDCss3PropSpaces	css3 缩进相关检查
no-style-for-simple-selector	FEDDoNotSetStyleForSimpleSelector	不要为简单选择器设置样式，避免全局覆盖
no-style-for-tag	FEDDoNotSetStyleForTagOnly	不要为 html tag 设置样式
font-unit	FEDFontSizeShouldBePtOrPx	字体的单位必须使用 px 或 pt
hack-prop	FEDHackAttributeInCorrectWay	hack 属性时的检查
hack-ruleset	FEDHackRuleSetInCorrectWay	hack 规则时的检查
high-perf-selector	FEDHighPerformanceSelector	针对低性能的选择器的检查
multi-line-brace	FEDMultiLineBraces	代码多行时的括号检查
multi-line-selector	FEDMultiLineSelectors	代码多行时的选择器检查

multi-line-space	FEDMultiLineSpaces	代码多行时的空格检查
add-author	FEDMustContainAuthorInfo	需要在文件中添加作者信息
no-alpha-image-loader	FEDNoAlphaImageLoader	不要使用 alphaImageLoader
no-appearance-word-in-selector	FEDNoAppearanceNameInSelector	不要在选择器中出现表现相关的词汇
no-comment-in-value	FEDNoCommentInValues	不要在 css 属性中添加注释
no-empty-ruleset	FEDNoEmptyRuleSet	删除空的规则
no-expression	FEDNoExpression	不要使用非一次性表达式
number-in-selector	FEDNoSimpleNumberInSelector	不要在选择器中使用简单数字 1、2、3
no-star-in-selector	FEDNoStarInSelector	不要在选择器中使用星号
del-unit-after-zero	FEDNoUnitAfterZero	删除 0 后面的单位
no-zero-before-dot	FEDNoZeroBeforeDot	删除 0.2 前面的 0
no-border-zero	FEDReplaceBorderZeroWithBorderNone	用 border:none 替换 border:0
no-underline-in-selector	FEDSelectorNoUnderLine	不要在选择器中使用下划线
add-semicolon	FEDSemicolonAfterValue	为每一个属性后添加分号
do-not-use-important	FEDShouldNotUseImportant	不要使用 important
single-line-brace	FEDSingleLineBraces	单行的括号检查
single-line-selector	FEDSingleLineSelector	单行的选择器检查
single-line-space	FEDSingleLineSpaces	单行的空格检查
keep-in-order	FEDStyleShouldInOrder	属性应该按照推荐的顺序编写
no-chn-font-family	FEDTransChnFontFamilyNameIntoEng	不要出现中文的字体设置，改用对应的英文
unknown-css-prop	FEDUnknownCssNameChecker	错误的 css 属性
unknown-html-tag	FEDUnknownHTMLTagName	错误的 html tag
lowercase-prop	FEDUseLowerCaseProp	属性应该用小写
lowercase-selector	FEDUseLowerCaseSelector	选择器用小写字母
single-quotation	FEDUseSingleQuotation	使用单引号
z-index-in-range	FEDZIndexShouldInRange	z-index 取值应该符合范围要求

目前所有的插件都有检查功能（check 方法），有的还没有对应的修复功能（fix 方法），待后续进一步完善。

单元测试

单元测试是开发者能够自行控制的，保证代码正确运行的有效手段。作为开源工具，CSSCheckStyle 开发了一套小型的单元测试框架。其单元测试类型主要包括两类：

1、CSS 版单元测试。以一个 CSS 文件的形式，来编写单元测试的用例和断言

2、Python 版单元测试。在 CSS 文件形式无法很好满足测试要求的时候，Python 版的单元测试可以起到很好的补充作用。

通过 tests/runUnitTests.py 来运行所有的单元测试，测试全部通过才是正常情况。

下面对这两种单元测试类型进行详细的说明。

CSS 版单元测试

CSS 版单元测试，通过编写 CSS 文件来编写用例和断言，主要用于测试“代码检查”的功能。凡是放在 tests/unit 目录下的、不是以_开头的 CSS 文件，都认为是单元测试文件。

每一个单元测试文件包含断言和待测试代码两部分内容：

1、通过@unit-test-expecteds 来定义的断言，断言的 key 为错误优先级，value 为错误信息。

2、按照普通 CSS 的书写风格来编写的待检查的 CSS，所有错误信息都将被收集。

在检查的过程中，单元测试框架收集@unit-test-expecteds 中的断言内容，并且检查剩余的所有 CSS 中的代码问题，通过断言与实际代码问题的对比，确定功能是否正确。对比规则如下：

1、如果断言中存在，而实际检查不存在的问题，则给出[expect but not have]的方式给出错误提示。

2、如果断言中不存在，而实际检查存在的问题，则给出[unexpect but has]的方式给出错误提示。

以 FEDNoUnitAfterZero 的单元测试 CSS 文件为例。其断言为：

```
@unit-test-expecteds {
    1: unit should be removed when meet 0 in ".test-zero-px"
    1: zero should be removed when meet 0.xxx in ".test-zero-dot-one-px"
    1: unit should be removed when meet 0 in ".test-padding"
}
```

而待测试的部分代码如下：

```
.test-zero-px {
    width: 0px;
}
.test-zero {
    width: 0;
}
.test-zero-dot-one-px {
    width: 0.1px;
}
.test-padding {
    padding: 1px 0px;
}
.test-padding-start-with-zero {
    padding: 0 1px 0 1px;
}
```

断言给出的几种错误，确实在待测试的代码中出现了，不多也不少，也只有这样，整个测试用例才算正确通过了。

这种单元测试开发方式的优点是：

1、断言和待测试代码放在同一个文件中，便于快速定位问题

2、断言和待测试代码独立开来，避免两种代码揉在一起，相互混淆

3、保留了 CSS 代码的原始编写风格，无需为了测试做任何额外的改变

4、如果添加的代码没有任何问题，则只需要编写 CSS 代码即可，无须改动断言

以上几个优点，对于测试代码的编写而言，都是非常舒服惬意的。

Python 版单元测试

CSS 版单元测试，主要解决的是代码检查的单元测试，对于代码自动修复、代码压缩等复杂功能的测试，无法很好的总结出一种通用的测试编写模式，因此提供了 Python 版的单元测试编写方法。

为了简化断言的编写，Python 版单元测试提供了类似 QUnit 的断言 API，按照实际情况，目前包括：

1、ok (flag, msg)

2、equal (actural, expected, msg)

3、notEqual (actural, expected, msg)

凡是放置在 tests/unit 目录下的，非 asserts.py、helper.py、__init__.py 的 Python 文件，都认为是单元测试文件。

每一个单元测试文件的编写规则如下：

1、必须导入 asserts.py，从而能够编写断言，并且收集错误信息

2、必须提供 doTest 方法，并将此方法作为测试的执行方法

下面是一个简单的自动修复属性顺序的测试示例，需要说明的是，目前 CSSCheckStyle 的所有 Python 版单元测试目录，都提供了相应的 helper 来封装需要的额外操作，因此示例中只需要引入 helper 的所有成员即可。

```
from helper import *

def doTest():                                                    //doTest 必须有
    fixer, msg = doFix('.test {width:100px; display:none;}', '')  //doFix 由 helper 提供

    styleSheet = fixer.getStyleSheet()
    equal(len(styleSheet.getRuleSets()), 1, 'one ruleset')       //断言 1
    ruleSet = styleSheet.getRuleSets()[0]
    equal(ruleSet.selector, '.test', 'it is the selector that i need')  //断言 2

    rules = ruleSet.getRules()
    equal(rules[0].name, 'display', 'first element is display now')  //断言 3
    equal(rules[1].name, 'width', 'second element is width')     //断言 4

    equal(rules[0].value, 'none', 'first element value is ok')   //断言 5
    equal(rules[1].value, '100px', 'second element value is ok')  //断言 6
```

上面的测试，自动修复了一段 CSS 程序，并且通过一系列的断言，保证了：

· 断言 1：整个样式表只有一个规则，规则个数正确

· 断言 2：确保唯一的规则的 CSS 选择器是正确的

· 断言 3：属性重排以后的第一个规则的 name 为 display，属性重排顺序正确
· 断言 4：属性重排以后的第二个规则的 name 为 width，属性重排顺序正确
· 断言 5：属性重排以后的第一个规则的 value 为 none，取值正确
· 断言 6：属性重排以后的第二个规则的 value 为 100px，取值正确

Python 版的单元测试，弥补了 CSS 版单元测试的不足，其优点是：
1、能够很好的实现不同类型的测试效果
2、通过简单的断言 API，测试代码的编写清晰易懂
3、无需过多考虑测试框架，编写 Python 程序放置在对应目录，即可完成测试，方便快捷

综合运用两种单元测试，将使测试代码的编写更加简洁高效。

工具的安装与使用

目前 CSSCheckStyle 已经加入了 setup 机制，通过 easy_install 即可一键安装，并自动生成命令行工具，方便使用。
安装命令为：

```
easy_install https://github.com/wangjeaf/CSSCheckStyle/archive/master.tar.gz
```

CSSCheckStyle 一共包含三个命令行工具：ckstyle/fixstyle/compress(csscompress)，分别用于代码的检查、自动修复、压缩。具体使用方法、命令行参数、配置方式、使用示例，请移步 CSSCheckStyle 的开源 github 项目：

https://github.com/wangjeaf/CSSCheckStyle

总结

在 CSS 编码规范制定完毕但是检查工具效果差强人意的情况下；
在 CSS 代码风格检查工具缺失导致规范中的代码风格相关规范无法很好保证的情况下；
在 CSS 工具五花八门各司其职但是实现方式迥异导致工具整合困难使用困难扩展困难的情况下；
在 CSS 代码在 Rule/RuleSet/StyleSheet 三个级别持续优化后压缩率能够变得更低 CSS 效率变得更高而没有工具良好支撑这种优化的情况下；
在 CSS 应该自动检查发现问题并且自动分情形修复从而开发者能够无后顾之忧的在不影响开发效率的前提下不断提高编码规范程度并优化产出但是没有工具能够良好支持的情况下；

完全遵循（或者说目标是完全遵循）《人人 FED CSS 编码规范》的 CSS 解析、检查、修复、压缩的自动化工具 CSSCheckStyle 应运而生。

目前该工具在检查、修复、压缩等方面都表现出了强劲的生命力和竞争力。未来，我们会在 CSSCheckStyle 上持续发力，致力于打造一款“多层次全方位立体式”的 CSS 工具集，希望更多的有识之士能够加入我们，加入到这个工具的建设中来。

如果您对此工具很感兴趣，或者发现此工具的任何问题，请在 github 上给我留言。

https://github.com/wangjeaf

【顺便打个广告】人人 FED 长期招聘优秀的前端工程师。如果您想来人人工作，或许，您的留言或贡献，可以为您的求职过程加分不少哦~~