

# 组件开发文档

## 组件库介绍

组件库独立于可视化开发系统。它的发布路径为 `/huodong.37.com/fe-editor/res/publish/components/`

系统通过读取组件的信息，以可视化编辑的方式呈现。用户对组件进行编辑、组合，完成页面的搭建。

为了让系统可以读取、理解组件，组件的开发需要遵循一定的规范。

下面介绍组件开发的细节。

## 组件结构

首先来看一个 Button（按钮）组件的结构：

```
- Button
  - images
    - 1.jpg
    - 2.jpg
  - js
    - index.js
  - index.html
  - index.css
  - config.json
  - thumb.jpg
```

它包含了：

- images  
目录：存放着组件用到的图片。这些图片是这个组件默认使用到的（比如背景图，hover背景图），通常这些图片在编辑时都会被修改
- js 目录：存放着该组件的 js 代码。里面的主文件以 `index.js` 命名
- index.html：组件的 html 模板
- index.css：组件的 css 样式
- config.json：组件的相关配置
- thumb.jpg：组件的预览图（只能是 .jpg 格式）

以上命名不可更改。

## 组件代码

代码文件包括：

- index.html
- index.css
- index.js
- config.json

下面还是以 Button 组件（按钮）为例，逐个介绍。

### index.html

```
<a href="{{href.value}}" target="_blank" class="link"></a>
```

这里使用了 [Handlebars](#) 这个模板引擎，感兴趣的话可以深入去看一下。现在可以继续往下看，因为实际上组件的开发只用到这个模板引擎很少的一部分功能。

首先，花括号的插值语法是大多数模板引擎的套路。从模板到真实 html 的转换差不多也是类似的写法：

```

/**
 * 如果这里写成 {{href.value}}
 * 那下面的 context 需要改为：
 * var context = {
 *   href: { value: "xxx" }
 * }
 */
var tpl = '<a href="{{href}}" target="_blank" class="link"></a>'
var compiler = Handlebars.compile(tpl)
var context = {
  href: 'http://www.37.com/'
}


// <a href="http://www.37.com/" target="_blank" class="link"></a>
var html = compiler(context)

```

所以，对于一个组件来说，花括号插值的地方，就表示那个是可以被编辑的。

换句话说，如果你希望一个组件的某个地方，是可以被自定义修改的，就把那个地方写成花括号插值的形式。

以这个 Button 组件为例，对于跳转链接的一个按钮来说，它的 html 部分，能够被修改的也就只有链接地址 href 而已。（当然样式也能改，但不属于它的 html 部分）

 按钮的分类不止一种。



这里的 Button 指的是这种：

没有文字，所以它的可编辑项也就只有 href 而已。

如果是那类带有活动字的按钮，那可以编辑的地方除了 href 之外还有活动字的内容。像这样：

```
<a href="{{href.value}}" target="_blank" class="link">{{text.value}}</a>
```

## index.css

```

.link {
  display: block;
  width: 221px;
  height: 54px;
  background-image: url(/images/1.jpg);
  background-repeat: no-repeat;
}
.link:hover {
  background-image: url(/images/2.jpg);
}

```

index.css 这里没有什么特殊的内容，仅仅是普通的 css。

可以看到这里引用了 images 里面的 2 张图片，作为按钮普通状态、鼠标经过状态的样式。





目前不能用 sass 或者 less 之类的。

当然如果真有这个需求，修改下构建流程加上去就好。

个人认为 sass 只有在 css 数量达到一定规模时才有优势。

而对于一个组件来讲，少则几行，多也就几十行的样式，再考虑到团队成员不一定会写 sass，维护不便。

保持简单就好。

## index.js

```
define(["SQ"], function($) {
// 这个模块仅仅返回一个待执行函数
return function(rootId) {
// 预先定义好的函数，用来给 class 或 id 加上前缀
function __(selector) {
    if (!rootId) {
        return selector;
    }
    if (selector.indexOf(".") < 0 && selector.indexOf("#") < 0) {
        return $.map(selector.split(" "), function(clsOrId) {
            return rootId + clsOrId;
        }).join(" ");
    }
    return selector.replace(/(\.|#)/g, function(matched, p1) {
        return p1 + rootId;
    });
}

// 从这里开始写你的代码

}
});
```

index.js 的代码需要详细说一下。

首先，因为开发系统产出的页面用到了 require.js，所以这里需以这种形式来书写模块。

这个模块仅仅返回一个函数，你将要写的所有代码都在这个函数里面。同时，这个函数接收一个参数 rootId：组件的唯一 id。

在返回的这个函数中，除了你自己写的代码之外，还预先定义了一个函数（函数名是两个下划线）：\_\_

这个预先定义的函数的作用是：给 class 或者 id 加上组件的 id 作为前缀。



给 class 加上前缀，是为了解决多个组件之间，类名冲突的问题。

类名冲突，这里先说几种方案：

1. 遵循某种约定，由组件开发人员自主添加前缀。比如以组件名作为前缀，在 Button 组件中：

```

<style>.Button-link {font-size: 14px;}</style>

<a href="xxx" class="Button-link"></a>

<script>
$('.Button-link').on('click', function(){} )
</script>

```

这种方案虽然能解决不同类型组件之间的类名冲突，[但无法解决同类组件的类名冲突](#)。（另外这种代码写起来也特别麻烦）想象页面中存在两个 Button 组件.....

2. 既然第一种方案（手动添加前缀）无效，采用另一种思路，[由系统给每个组件添加唯一的随机 id](#)。

```

<!-- 书写样式的时候，不必关心前缀，按照正常套路来写 -->
<style>.link {font-size: 14px;}</style>
<!-- 最后由系统处理成这样 -->
<style>.OXYZ__link {font-size: 14px;}</style>

<!-- 同样的，html 代码无须关心前缀 -->
<div>
  <a href="xxx" class="link"></a>
  <div id="test"></div>
</div>
<!-- 最后，系统给组件包一层 div，并给它生成一个随机 id。这个 div
仅仅作为标识作用，它没有任何样式，不会影响到组件本身 -->
<!-- 不过，这个包裹的 div 还有一个作用是就是对组件进行绝对定位 -->
<!-- 同时，对 html 中存在的 class 或 id 都会自动加上前缀 -->
<div id="OXYZ__">
  <div>
    <a href="xxx" class="OXYZ__link"></a>
    <div id="OXYZ__test"></div>
  </div>
</div>

```

那 js 代码怎么写？

组件的 id 是系统随机生成的，我们事先没办法知道。但为了获取到对应的 dom 元素，又必须依赖这个 id。

[解决的办法是我们将代码写在一个待执行的函数里面，这个函数接收组件的 id 作为参数](#)。回过头来看看前面 index.js 文件中的内容：

```

define(["SQ"], function($) {
// 这个模块仅仅返回一个待执行函数，它接收组件的唯一 id 作为参数
return function(rootId) {
// 预先定义好的函数，用来给 class(或id)加上前缀
// 它接收原始的 class(或id)，结合组件 id，返回一个处理过后的、加了前缀的 class(或id)
function __(selector) {
    if (!rootId) {
        return selector;
    }

    // 没有点号 "." 或 井号 "#"
    // eg: $dom.addClass(__("test")) --> $dom.addClass("XYZ_test")
    // eg: $dom.removeClass(__("x y")) --> $dom.removeClass("XYZ_x XYZ_y")
    // eg: dom.className = __("a b") --> dom.className = "XYZ_a XYZ_b"
    // eg: $dom.attr("id", __("test")) --> $dom.attr("id", "XYZ_test")
    if (selector.indexOf(".") < 0 && selector.indexOf("#") < 0) {
        return $.map(selector.split(" "), function(clsOrId) {
            return rootId + clsOrId;
        }).join(" ");
    }

    // 带点号 "." 或 井号 "#"
    // eg: $(__(".test")) --> $(".XYZ_test")
    // eg: $(__(".a.b")) --> $(".XYZ_a.XYZ_b")
    // eg: $(__("#test")) --> $("#XYZ_test")
    return selector.replace(/(\.|#)/g, function(matched, p1) {
        return p1 + rootId;
    });
}

// 从这里开始写你的代码

// 注意：凡是跟 class 或 id 相关的操作，都要调用 __ 函数
$(__(".a")).on("click", function(){
    $(__(".b")).toggleClass(__("active"))
    document.body.className = __("hello")
    document.body.id = __("test")
})

// 也可以将结果保存为变量
var activeCls = __("active")
var helloCls = __("hello")

$(__(".a")).on("click", function(){
    $(__(".b")).toggleClass(activeCls)
    document.body.className = helloCls
})
});

```

以上就是解决类名冲突的内容，注意看注释部分。

可能会有疑问，这个模块返回的函数，在哪里调用吗？在 require.js 加载完专题的所有依赖后，自动调用。大致是这样的：

```

// 假设这个专题上有 A / B / C 三个组件需要加载 js
// 又假设 A 类组件在页面上只存在 1 个，id 为 A1
// B 类组件有 2 个，id 为 B1、B2
// C 类组件有 3 个，id 为 C1、C2、C3

// 于是得到下面的依赖信息
var depConfig = {
  tracker: ["A", "B", "C"],
  entities: {
    A: ["A1"],
    B: ["B1", "B2"],
    C: ["C1", "C2", "C3"]
  }
}


// 根据组件名，拼接出依赖模块的路径
// path = ["A/js/index", "B/js/index", "C/js/index"]
var path = depConfig.tracker.map(function(name) {
  return name + "/js/index"
})

// 开始加载依赖
require(path, function() {
  var args = arguments

  depConfig.tracker.forEach(function(name, index) {
    // `name` 类组件对应的组件 id 数据
    var idList = depConfig.entities[name]
    // 开始调用每个模块返回的函数，传入组件 id
    idList.forEach(function(id) {
      args[index](id)
    })
  })
})

```

以上就是加载组件依赖的细节。

 另外，关于解决类名冲突的，再说两点：

- 解决冲突的第 3 种方案：类名本身保持不变，依靠组件外面包裹的 div 的 id 来限制样式的作用范围。

```
<!-- 书写样式的时候，不必关心前缀，按照正常套路来写 -->
<style>.link {font-size: 14px;}</style>
<!-- 最后由系统处理成这样，在每个类前面添加一个 id 来限制样式的作用范围 -->
<style>#OXYZ_ .link {font-size: 14px;}</style>
```

```
<!-- 同样的，html 代码无须关心前缀 -->
<a href="xxx" class="link"></a>
<!-- 最后，系统给组件包一层 div，并给它生成一个随机 id。这个 div
仅仅作为标识作用，它没有任何样式，不会影响到组件本身 -->
<!-- 不过，这个包裹的 div 还有一个作用就是对组件进行绝对定位 -->
<div id="OXYZ_">
  <a href="xxx" class="link"></a>
</div>
```

```
<!-- 这种方案看似可行，但依然存在问题 -->
<!-- 如果在某种情况下，页面引入了外部的一个样式，如下 -->
<style>.link { xxx /* 来自外部的样式 */ }</style>
<!-- 这个外来的样式依然对组件产生了影响，通常这不是我们想要的结果 -->
```

经过以上 3 种方案的对比，最终选择了第 2 种方案，更彻底地解决类名冲突。此外，第 2 种方案还带来一个好处：[开发组件的时候，对于 class 的命名可以做到非常简洁](#)。由于类名是完全不会冲突的，你可以在一个组件的样式里面使用诸如 .name .title .list

这种非常简洁的类名。而由于组件本身是被作为一个很小的单元整体来对待的，简短的类名并不会影响维护，反而减轻了为了避免冲突起类名（比如：.header-title .header-list .main-list）的痛苦.....

- 为了解决类名冲突，引入两个不便之处：[1、样式的冗余](#) [2、操作 class 类名的小繁琐](#)。造成样式冗余的根本原因是：组件之间类名的隔离做得太彻底了，[一个组件的样式只作用于它本身，没办法在同类型组件之间进行复用](#)。至于操作 class 类名的繁琐（涉及到 class 的操作都要通过 \_ 函数），同样也是为了避免类名冲突付出的代价。

只能说有得有舍。当然类名冲突这个问题，肯定存在更好的解决方案。

由于项目开发时间仓促，再考虑到实际的一些客观条件（比如使用require.js，比如没有node后端对文件作进一步的修改、合并等控制），暂时采取了上述方案。

## config.json

下面介绍一个组件最重要的部分。还是以 Button 按钮组件为例，下面是它的配置文件：config.json

```

{
  // 组件的名字
  "name": "Button",

  // 组件所属的分类，这里表示它属于 BUTTON 这个分类
  "ownerType": "BUTTON",

  // 组件的描述，向用户描述这个组件的基本信息
  "description":
    "一个按钮，有普通状态以及鼠标经过的hover状态，可设置跳转链接。如果不要hover状态，可以直接使用【链接】分类里面的【图片链接】组件",

  // 组件的一些基础配置
  "options": {
    "script": false, // 标识这个组件是否需要加载 js 文件，默认为 false
    "draggable": true, // 标识这个组件是否允许在可视化编辑中被拖动，默认为 true（对于那些定位为 fixed 的组件，应该设置为 false）
    "resizableSelector": ".link" // 标识缩放一个组件时，被修改（宽、高）的 css 选择器。默认为空
    "", 表示不能缩放大小
  },

  /**
   * 设置这个组件的可编辑的 html 内容
   * 回想上面提到的 Handlebars，我们将模板转换为真实 html 的时候，需要传入一个对象
   * 下面这个 `html` 对象，就是渲染组件 `html` 时，传进去的对象：
   * var html = compiler(html)
   */
  "html": {
    "href": {
      // label 这个字段，会展示在可视化编辑界面上，它的作用是：告诉用户他当前正在修改的是哪一部分内容
      "label": "跳转链接",
      "value": "http://www.37.com/"
    }
  },

  /**
   * 设置这个组件的可被修改的 css 内容
   * 以一个选择器为 key，一个对象为 value
   * 这个对象里面，同样有一个 label 字段，告诉用户他当前正在修改的是哪一部分样式
   * 此外，还有一个 editable 字段，值类型是一个数组，数组的元素是允许修改的样式
   */
  "css": {
    ".link": {
      "label": "按钮",
      "editable": ["width", "height", "background-image"]
    },
    ".link:hover": {
      "label": "鼠标经过时",
      "editable": ["background-image"]
    }
  }
}

```

## 渲染一个列表



列表是经常会碰到的一种结构。比如，轮播图就是一个图片列表。

下面介绍下列表形式的模板是怎么写的。

假设，我们现在要写一个图片列表组件，它最终的 html 结构是这样的：

```
<ul>
<li> </li>
<li> </li>
<li> </li>
<li> </li>
</ul>
```

对应的，我们可以写出它的数据格式：

```
// config.json

{
  ...

  "html": {
    "list": {
      "label": "图片列表",
      "value": [
        {
          "image": {
            "label": "图片",
            "value": "1.jpg"
          }
        }
      ]
    }
  },
  ...
}
```

显然，列表的数据是用数组来表示的。

下面看下如何用 handlesbar 的语法来遍历这个数组：

```
<ul>
  {{#each list.value}}
    <li><img src={{this.image.value}} /> </li>
  {{/each}}
</ul>
```

each 语法是 handlesbar 内置的，有兴趣可以点这里：[http://handlebarsjs.com/builtin\\_helpers.html](http://handlebarsjs.com/builtin_helpers.html)

在遍历过程中，可以用 `this` 指代当前的列表项。

另外，我们通常会给列表中的第一项，加上一个特定的类名。比如轮播图的第一项，默认是显示的，我们给第一项加一个 `class="show"`：

```
<ul>
  {{#each list.value}}
  {{#if @index}}
    <li><img src={{this.image.value}} /> </li>
  {{else}}
    <li class="show"><img src={{this.image.value}} /> </li>
  {{/if}}
  {{/each}}
</ul>
```

其中，@index 表示当前遍历到的列表项的下标，从 0 开始。

## 富文本编辑项

有时候我们希望某一块内容能以富文本形式进行编辑。比如，[任意设置字体颜色](#)、[添加超链接](#)、[添加图片](#)等等。

这个时候，需要在组件的 config.json 文件中，对指定的字段进行配置：

```
// config.json

{
  ...

  "html": {
    "content": {
      "label": "内容",
      "value": "默认内容",
      "rich": true
    }
  },
  ...
}
```

我们给 content 这个字段添加了一个属性：[rich: true](#)。表明这是富文本编辑的内容。

除此之外，模板也要作相应的小修改：

```
<div>{{{content.value}}}</div>
```

注意这里的花括号变成了 3 个。这同样也是 [handlebars](#) 的语法，三个花括号用来渲染未转义的 html 内容。

通过以上[两个地方](#)的设置，就可以将某个字段变为富文本编辑形式的。



富文本编辑通常只用在[内容块](#)的地方。而对于类似标题、链接文字，这类内容来说，富文本编辑就显得不必要了。

## 提供一个下拉框选项，一键切换组件样式

有时候，你希望对一个组件[设置几套不同的样式](#)，然后有一个下拉选项，可以一键切换组件的样式。

下面介绍这个想法如何实现。

首先，不同的样式，肯定是[通过不同的 class 类名](#)来实现的，这是最简单的途径：

```
<!-- 第一套样式 -->
<style>
.style-1 div { xxx }
.style-1 a { xxx }
</style>

<!-- 第二套样式 -->
<style>
.style-2 div { xxx }
.style-2 a { xxx }
</style>

<!-- html 内容 -->
<div class="base style-1">
  <!-- 任何内容 -->
  ...
</div>
```

我们上面说过，在 [html 模板](#) 中，如果你希望哪部分的内容是可以改变的，就将它转换为花括号插值的形式。

显示在这里，我们希望 `html` 中的 `class` 是可以被修改的。于是将模板修改如下：

```
<div class="base {{style.value}}">
  <!-- 任何内容 -->
  ...
</div>
```

这是第一步。接下来，在组件的 `config.json` 中，对 `style` 这个字段进行配置：

```
// config.json

{
  "html": {
    "style": {
      "label": "选择样式",
      "value": "style-1", // 给它个默认值，默认选中第一套样式

      // options 提供可能的几种选项，每一项的 value 表示 class
      "options": [
        { "label": "样式一", "value": "style-1" },
        { "label": "样式二", "value": "style-2" }
      ]
    }
  }
}
```

到这里，一个一键切换组件样式的功能就完成了。

## 关于组件 `config.json` 的几点总结

### html 部分

对于某个字段的设置，目前有 [4 种](#) 情况。

```
// 第一种情况
{
  "title": {
    "label": "标题",
    "value": "默认标题"
  }
}
```

上面是第一种情况，最简单的，一个 label + 一个 value。

在模板中，通过[两个花括号来取值](#)：{{title.value}}

```
// 第二种情况：富文本编辑
{
  "content": {
    "label": "内容",
    "value": "默认内容",
    "rich": true
  }
}
```

第二种情况：富文本编辑。

在模板中，通过[三个花括号来取值](#)：{{{content.value}}}

```
// 第三种情况：给定一些选项，提供一个下拉框进行选择
{
  "nav": {
    "label": "导航条位置",
    "value": "left",
    "options": [
      { "label": "左", value: "left" },
      { "label": "右", value: "right" }
    ]
  }
}
```

第三种情况，给定一些选项，提供一个下拉框进行选择。通常用在一键更接组件的样式。

```
// 第四种情况：渲染一个列表
{
  "list": {
    "label": "轮播图列表",
    "value": [
      {
        "image": { "label": "图片", "value": "1.jpg" },
        "link": { "label": "跳转地址", "value": "http://www.37.com/" },
        "title": { "label": "图片标题", "value": "xxx" }
      }
    ]
  }
}
```

第四种情况，渲染一个列表内容。通过 handlebars 的 #each 语法来遍历数组。

## css 部分

关于 css 的设置部分，说几点要注意的地方。

1、在 config.json 中设置的选择器，需要在 index.css 文件中存在。

假设 index.css 如下：

```
/* index.css */  
  
.root { xxx }  
.list { xxx }
```

在 config.json 中：

```
// config.json  
  
{  
  ...  
  "css": {  
    // 在 index.css 中找不到 .title  
    ".title": {  
      "label": "xxx",  
      "editable": ["color"]  
    },  
    // 这样可以  
    ".root": { ... }  
  },  
  ...  
}
```

这种情况下，控制台会提醒：组件 xxx 的样式中，不存在 ".title" 选择器。系统会忽略这个设置。

2、任何一个选择器的 editable 列表中的样式类型，同样需要在 index.css 中存在。

假设 index.css 如下：

```
/* index.css */  
  
.root { width: 200px; height: 100px; color: #fff; }
```

在 config.json 中：

```
// config.json

{
  ...
  "css": {
    ".root": {
      "label": "xxx",
      "editable": ["color", "font-size"] // "color" 可以, "font-size" 会被忽略
    }
  },
  ...
}
```

这种情况下, 控制台会提醒: 组件 xxx 的样式中, 选择器 ".root" 不存在 "font-size" 样式。系统会忽略字体的设置。

### 3、复合属性需要分开来写

假设 index.css 如下:

```
/* index.css */

.root { background: url(1.jpg) #fff no-repeat; }
```

在 config.json 中:

```
// config.json

{
  ...
  "css": {
    // 这样可以
    ".root": {
      "label": "xxx",
      "editable": ["background-color"] // 这样不行
    }
  },
  ...
}
```

这种情况下, 控制台会提醒: 组件 xxx 的样式中, 选择器 ".root" 不存在 "background-color" 样式。系统会忽略字体的设置。

这时, 你需要将你希望编辑的样式单独拆出来写:

```
/* index.css */

.root { background: url(1.jpg) no-repeat; background-color: #fff;}
```

另外, 如果你在 editable 的数组中, 直接填写 "background" 也会被忽略掉。因为 "background" 是个复合属性, 系统没办法知道你要修改的是哪一个属性。(其他复合属性同理)

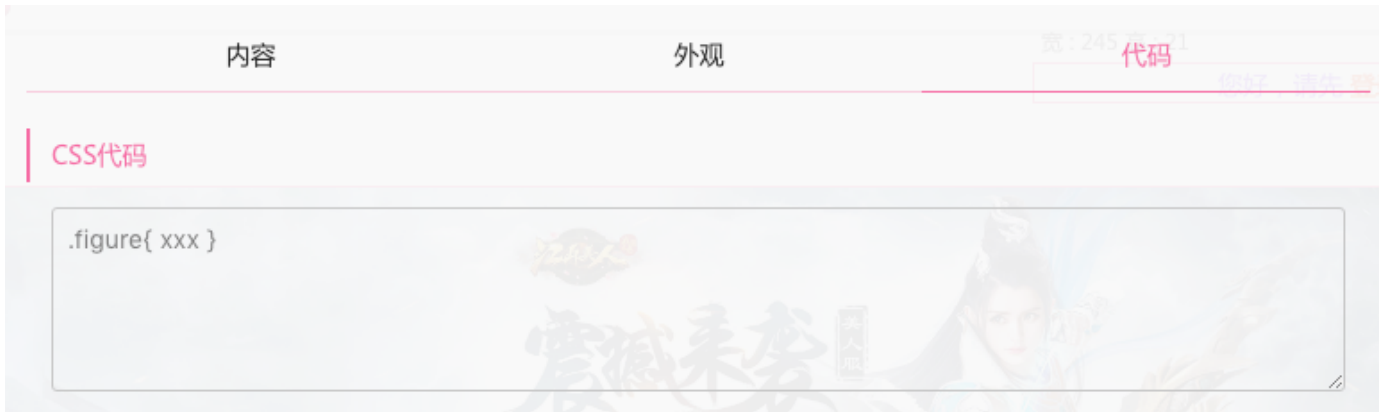
## 关于自定义css代码

在可视化开发系统中, 通过审查组件的元素, 可以看到组件的 class 是带有前缀的:

```
▼ <div id="OCMLX1_" data-role="component-wrapper" data-name="Slide-1" style="position: absolute; left: 2px; top: 88px; z-index: 1; transform: translate3d(0px, 0px, 0px);"> == $0
  ▶ <style>_</style>
  ▼ <tag-will-be-removed>
    ▼ <div class="OCMLX1__root">
      ▼ <ul class="OCMLX1__figure">
        ▶ <li class="OCMLX1__current">_</li>
      </ul>
      <div class="OCMLX1__mask"></div>
      ▶ <ul class="OCMLX1__text">_</ul>
      ▶ <ul class="OCMLX1__navigator">_</ul>
    </div>
  </tag-will-be-removed>
  ▶ <will-be-removed>_</will-be-removed>
  ▶ <will-be-removed>_</will-be-removed>
</div>
```

两个下划线后面的部分，就是原本的 class，在写自定义 css 代码的时候，只需要写这个原本的 class 就行，无需加前缀。

比如，想对 .figure 这个 class 添加样式：



系统会自动加上前缀，并将样式追加到最后，保证优先级：

```
▼ <style>
  .OCMLX1__root{width:1000px;height:280px;position:relative;overflow:hidden;}.OCMLX1__figure{height:100%;position:relative;z-index:1;}.OCMLX1__figure li{.OCMLX1__figure img{width:100%;height:100%;}.OCMLX1__figure li,.OCMLX1__text li{display:none;}.OCMLX1__figure .OCMLX1__current{display:block;}.OCMLX1__mask{position:absolute;width:100%;height:40px;background:#000;opacity:0.6;filter:alpha(opacity=60);left:index:2;}.OCMLX1__text{position:absolute;width:100%;height:40px;line-height:40px;left:0;bottom:0;text-indent:1em;font-size:16px;color:#fff;z-index:3;}.OCMLX1__navigator{position:absolute;height:16px;right:15px;bottom:12px;z-index:4;}.OCMLX1__navigator li{float:left;width:16px;height:16px;text-align:center;}.OCMLX1__navigator .OCMLX1__current{background-color:#389fe9;}.OCMLX1__figure{ xxx }
</style>
▼ <tag-will-be-removed>
  ▼ <div class="OCMLX1__root">
    ▼ <ul class="OCMLX1__figure"> == $0
      ▶ <li class="OCMLX1__current">_</li>
    </ul>
    <div class="OCMLX1__mask"></div>
    ▶ <ul class="OCMLX1__text">_</ul>
    ▶ <ul class="OCMLX1__navigator">_</ul>
  </div>
  </tag-will-be-removed>
  ▶ <will-be-removed>_</will-be-removed>
  ▶ <will-be-removed>_</will-be-removed>
</div>
```

你只能针对组件存在的 class 进行自定义样式的修改，组件不存在的 class 自然不起作用。



填写自定义样式的时候，有一定的规则。

比如，你不能直接填写 `div { xxx }`

因为这样会对全局造成影响，显然不合理。下图中，系统判断到样式没有以 "." 号开头，会提示错误：

### CSS代码

```
div {
```

格式有误，请检查

### CSS代码

```
.div {
```

格式有误，请检查

或者没有以 “}” 结尾：

同时，每一个选择器，都必须以 class 开头。必要情况下，系统会对你的样式进行纠正：

### CSS代码

```
.div { xxx } div span { xxx }
```

### CSS代码

```
.div{ xxx }.div span{ xxx }
```

会被纠正成：

## 编写组件的一个套路

对于一个有 js 逻辑的组件来说，如果 js 需要获取一些可能会被编辑的数据，那通常的做法是：将数据以 `data-xxx="xxx"` 形式保存到组件的 dom 上，然后在组件的 js 中进行获取。



比如，对于一个播放视频的组件：

。因为视频的链接、宽、高必须是可编辑的，将它存在 dom 上：

```
<div id="09SWG3_" data-role="component-wrapper" data-name="Video-1" style="position: absolute; left: 90px; top: 147px; z-index: 1; transform: trans
  ><style>_</style>
  ><tag-will-be-removed>
    ><a href="javascript:;" class="09SWG3_play" data-src="http://video.37wanimg.com/mir/20151026/wjy.flv" data-width="1000" data-height="560"></a>
  </tag-will-be-removed>
  ><will-be-removed>_</will-be-removed>
  ><will-be-removed>_</will-be-removed>
</div>
```



内容	外观	代码
视频链接(格式 : .flv)		<code>http://video.37wanimg.com/mir/20151026/wjy.flv</code>
视频宽度		<code>1000</code>
视频高度		<code>560</code>

然后在组件的 js 中，就可以获取到需要的数据：

```
$(__(".play")).on("click", function(e) {  
    e.preventDefault();  
  
    var $this = $(this);  
  
    var videoSrc = $this.attr("data-src");  
    var videoWidth = parseInt($this.attr("data-width"));  
    var videoHeight = parseInt($this.attr("data-height"));  
  
    if (  
        !videoSrc ||  
        !/^http:\/\/.test(videoSrc) ||  
        !\/\.flv$\/.test(videoSrc) ||  
        isNaN(videoWidth) ||  
        isNaN(videoHeight)  
    ) {  
        return;  
    }  
})
```

任何 js 需要获取的数据，都可以以这种形式来完成。



由于数据完全是可编辑的，也就是说，没办法控制输入是否合法（编辑界面没有正则的判断）。  
推荐在组件内，对获取的数据进行判断，必要时在控制台 warn 一下，有助于排错。

## 组件相关命令

切换到 `huodong.37.com/fe-editor/res/` 目录下：

## 新建一个组件

```
grunt add:components:[componentName]:[author]
```

比如：grunt add:components:Tab:linzerui

这样会在 res/development 目录下，创建一个组件：res/development/Tab

## 组件发布

```
grunt pub:components:[componentName]
```

比如：grunt pub:components:Tab

将 Tab 组件的文件进行压缩、路径替换等，发布到 res/publish/Tab

publish 目录下的文件，需要走发布系统进行发布。