CSS命名规范

- 开头注释

```
/*

@name: 文件的名称

@description: 简要的描述这个css 文件功能

@require: 依赖文件，没有就不用写

@author: 作者 最好附带联系方式(邮箱)

*/
```

- 一律小写
- 尽量用英文
- css很多的时候要写注释
- id 统一用驼峰式，不要在CSS里使用id，id是给JS使用的
- class 用单词，每个词之间用 - 分隔，不要用下划线
- 给CSS添加表示状态的前缀，让语义更明了，如 .is- ，例：.is-width-min {'{表示最小宽度}'}

CSS书写顺序

- 位置属性（position, top, right, z-index, display, float等）
- 大小（margin, padding, width, height）
- 文字系列（line-height, font, color, text-align等）
- 背景（background, border等）
- 其它（animation, transition等）

注意：css代码独立，不要样式放在HTML里

使用csscomb自动转换顺序(webstorm)

1. 全局安装csscom `npm i csscom -g`
2. 打开Websotrm -> Prefernces -> Tools -> External Tools
3. 点击 +
4. 输入
    - Name: CSScomb
    - Group：CSS TOOLS
    - Program: 你的全局node_modules目录/bin/csscomb (/Users/apple/.nvm/versions/node/v10.1.0/lib/node_modules/csscomb/bin/csscomb)
    - Parameters: $FilePath$ -v
    - Working directory: $FileDir$
5. 点击apply
6. 添加.csscomb.json 文件(和package.json同级，有了就无视这步)
7. 添加快捷键 Websotrm -> Prefernces -> keymap -> External Tools -> CSScomb
8. 打开css/less文件，运行一下.

JS命名规范

- 如果需要在JS里使用CSS进行批量操作，必须添加一个不包含CSS的classname，例如：J_list_pid，我们统一为大写J加下划线起头
- 必须使用Tab键进行代码缩进，以节约代码大小，建议设置编辑器的Tab为4个空格的宽度
- 接口风格
  - 类 - 驼峰式 - ModuleClass()
  - 公有方法 - 混合式 - getPosition()
  - 公有变量 - 混合式 - frameStyle
  - 常量 - 大写式 - DEFAULT_FRAME_LAYOUT，每个词之前用下划线分隔
- 其它风格
  - jQuery对象：$dom，用$前缀表示这是一个用jQuery包装的dom对象
  - 函数内私有方法和属性：_mixedCase，用_划线标注
  - 方法的参数：mixedCase， 例 ： function method(mixedCase)
  - 语句结束后，必须使用；号结束
- 所有变量尽量使用有意义的英文 ，不可拼音、英文混搭

特殊命名约定

- 所有布尔值前加 is
- 所有字符串前加 str
- 所有数组前加 arr
- 所有数字前加 num
- “obj” 作为局部变量或参数，表示 Object
- “em” 作为局部变量或参数，表示 Element
- “event” 作为局部变量或参数，表示 event
- “err” 作为局部变量或参数，表示 error
- 处理错误的变量，必须在后面跟着“Error”
- 重复变量 使用 i, j, k 等
- 函数的初始化必须用 init
- 操作符用空格间隔

注释规范

- 不打算给其它人使用的函数，添加 @ignore
- 一些相关的功能相关的函数，建议加上 @see Function来对上下文做索引
- 对于一些函数不建议或则需要注意的使用方法，必须加上 @deprecated 作为提醒
- 每个 js 文件的文件头都必须包含 @fileoverview @author 建议加上 @version
- 每个带参数的函数都必须包含 @param
- 每个有返回值的函数都必须包含 @return
- 构造函数必须加上 @constructor
- 继承函数建议加上 @base 表示其继承于哪个类
- 一般变量及局部变量用//方式进行注释，建议在需要做注释的语句的上一行加一行文字描述
- 更多可以参考 jsDoc

其它

- 循环体内的字符串累加

```
var r=[];
for(.....){
   r.push(`文字内容`);
}
var k = r.join('');
```

- switch可以用Object代替

```
var a = { '1': doAction1, '2': doAction2 }
function doAction1(){}
function doAction2(){}
a[1]();
```

- 不可使用eval
- JS代码独立，不要把javaScript代码放在HTML里

react组件规范

- 单个文件超过300行就应该考虑代码设计问题，适当减小组件粒度，对代码进行分割。
- 组件应分为容器组件和视图组件，多个关联子组件可以放在在命名空间中。

```
// Item组件放在Layer命名空间下
Layer.Item = class {};
export default class Layer {};
```

- 组件代码组织顺序：

```
export default class extends React.Component {
    // 1. 静态属性
    static defaultProps = {};

    static propTypes = {};

    static getDerivedStateFromProps(){}

    // 2. 对象属性
    state = {};

    // 3. 自定义方法
    onPullDown = () => {};

    // 4. 生命周期函数
    componentDidMount() {}

    // 5. render放最后
    render() {}
}
```

- state和defaultProps的每一个属性都要有注释
- 公共组件一定要有defaultProps和propTypes
- 使用新的生命周期(getDerivedStateFromProps替换componentWillReceiveProps)

> 注意：react 16.4 在state改变的时候也会触发getDerivedStateFromProps

以前的写法

```
state = {
  name: 0
};

componentWillReceiveProps(nextProps) {
  if (nextProps.name !== this.props.name) {
    if (this.state.loginStatus === null) {
      fakeServerRequest(nextProps.name).then(result => {
        // 实例属性 没有挂载在state中
        this.loginStatus = result;
      });
    }
    this.setState({
      name: nextProps.name
    })
  }
}
```

现在的写法

```
state = {
  loginStatus: null,
  name: 0
};


static getDerivedStateFromProps(nextProps, prevState) {
  if (nextProps.name !== prevState.name) {
     return {
      name: nextProps.name
      loginStatus
    }
  }
  return null;
}

// 将以前放在componenWillReceiveProps中的异步网络请求放在DidUpdate中
componentDidUpdate(prevProps, prevState) {
  const { name } = this.state;
  if (this.state.loginStatus === null) {
    fakeServerRequest(name).then(result => {
      this.setState({ loginStatus: result });
    });
  }
}
```

DOM操作

- 避免使用 ReactDOM.findDOMNode ， 用 ref 。
- 避免使用 document.querySelector、document.getElementById、jquery 直接查找元素。
- 将查找范围限制在组件本身，可以避免误操作其它组件。

```
export default class extends React.Component {
    componentDidMount() {
        const lis = this.elem.querySelectorAll('li');
    }

    render() {
        return (
            <Fragment>
                <ul className={styles.list} ref={e => this.elem =
e}>
                    {this.props.data.map(item => <li key={item.id}>
{item.name}</li>)}
                </ul>
            </Fragment>
        );
    }
}
```

组件划分

> 组件类型

视图组件	容器组件
关注事物的展示	关注事物如何工作
可能包含展示和容器组件，并且一般会有DOM标签和css样式	可能包含展示和容器组件，并且不会有DOM标签和css样式
常常允许通过this.props.children传递	提供数据和行为给容器组件或者视图组件
不要指定数据如何加载和变化	作为数据源，通常采用较高阶的组件，而不是自己写，比如React Redux的connect()
仅通过属性获取数据和回调	
很少有自己的状态，即使有，也是自己的UI状态	
除非他们需要的自己的状态，生命周期，或性能优化才会被写为功能组件	
通常使用无状态组件	

> 准则

- 职责单一原则；每个组件职责应该固定，不应该做过多的事情。
- 高内聚，低耦合；良好的组件不应该过多依赖外部状态，可以单独使用。