Contents

[HTML 4](#_Toc477521012)

[HTML5权威指南 4](#_Toc477521013)

[基本概念 4](#_Toc477521014)

[CSS样式 5](#_Toc477521015)

[JavaScript 6](#_Toc477521016)

[CSS 10](#_Toc477521017)

[参考http://learn.shayhowe.com/html-css/ 10](#_Toc477521018)

[盒模型The box model 10](#_Toc477521019)

[元素布局 10](#_Toc477521020)

[元素定位 11](#_Toc477521021)

[<Bootstrap Example> 12](#_Toc477521022)

[如何用CSS实现三角形图标？ 12](#_Toc477521023)

[布局：上导航栏固定，左导航栏也固定，剩下空间动态 15](#_Toc477521024)

[登录框样式 16](#_Toc477521025)

[修改bootstrap的类样式？ 17](#_Toc477521026)

[Bootstrap默认值 18](#_Toc477521027)

[固定上导航栏，下脚注？ 20](#_Toc477521028)

[<HTML5与CSS3设计模式> 21](#_Toc477521029)

[简化CSS使用 21](#_Toc477521030)

[HTML设计模式 24](#_Toc477521031)

[CSS选择器与继承 26](#_Toc477521032)

[框模型 27](#_Toc477521033)

[辅助概念 32](#_Toc477521034)

[文字样式 35](#_Toc477521035)

[块级元素 35](#_Toc477521036)

[列表 36](#_Toc477521037)

[图片 37](#_Toc477521038)

[流动布局 37](#_Toc477521039)

[<Scalable and Modular Architecture for CSS> 40](#_Toc477521040)

[Categorizing CSS Rules 40](#_Toc477521041)

[Changing State 43](#_Toc477521042)

[Depth of Applicability 45](#_Toc477521043)

[CSS Preprocessors 46](#_Toc477521044)

[JS 50](#_Toc477521045)

[Best Practice 50](#_Toc477521046)

[函数式编程 50](#_Toc477521047)

[类 51](#_Toc477521048)

[选项对象 52](#_Toc477521049)

[<JavaScript 面向对象编程指南> 52](#_Toc477521050)

[**函数也是数据**？(该数据类型包含的是代码，且可执行) 52](#_Toc477521051)

[**对象** 54](#_Toc477521052)

[**原型** 56](#_Toc477521053)

[**继承** 56](#_Toc477521054)

[惯用法 60](#_Toc477521055)

[<Effective JavaScript> 62](#_Toc477521056)

[全局变量与局部变量 62](#_Toc477521057)

[闭包 62](#_Toc477521058)

[变量声明提升间 63](#_Toc477521059)

[Javascript立即函数表达式 63](#_Toc477521060)

[函数调用，方法调用及构造函数调用之间的区别？ 64](#_Toc477521061)

[高阶函数 65](#_Toc477521062)

[创建可变参数的函数一般提供两个版本 65](#_Toc477521063)

[复用对象方法 66](#_Toc477521064)

[高阶函数惯用法：函数curring 67](#_Toc477521065)

[JavaScript类的本质 67](#_Toc477521066)

[Javascript实现类继承的最佳实践 68](#_Toc477521067)

[Javascript实现存储私有数据 69](#_Toc477521068)

[函数的this变量 70](#_Toc477521069)

[面向函数编程 72](#_Toc477521070)

[在类数组对象上尽量复用通过的数组方法 72](#_Toc477521071)

[undefined: JavaScript无法无法提供具体的值 73](#_Toc477521072)

[选项对象 73](#_Toc477521073)

[无状态API和有状态API 74](#_Toc477521074)

[使用结构类型来实现多态性，从而避免使用继承 74](#_Toc477521075)

[Javascript无法实现重载（函数重名，但参数不同） 74](#_Toc477521076)

[<JavaScript忍者秘籍> 75](#_Toc477521077)

[函数 75](#_Toc477521078)

[闭包 80](#_Toc477521079)

[JS技巧 87](#_Toc477521080)

[ES2015 (ES6) 88](#_Toc477521081)

[SVG 88](#_Toc477521082)

[<SVG Essentials> 89](#_Toc477521083)

[基本概念 89](#_Toc477521084)

[形状元素 89](#_Toc477521085)

[笔画，填充，图案，渐变 91](#_Toc477521086)

[分组和引用对象 92](#_Toc477521087)

[坐标系统变换 92](#_Toc477521088)

[文本 93](#_Toc477521089)

[滤镜 93](#_Toc477521090)

[SVG动画 = SMIL动画 + CSS动画 94](#_Toc477521091)

[交互 (建议直接采用d3) 96](#_Toc477521092)

[SVG DOM 97](#_Toc477521093)

[SVG样式表 （详细见p253） 98](#_Toc477521094)

[推荐使用Javascript库： D3, Raphael, Snap.svg 98](#_Toc477521095)

[Canvas 99](#_Toc477521096)

[<HTML5 Canvas 开发详解> 99](#_Toc477521097)

[基本概念 99](#_Toc477521098)

[几何形状 100](#_Toc477521099)

[填充对象： 基本颜色，渐变色和图案 100](#_Toc477521100)

[画布变换 101](#_Toc477521101)

[清除画布 102](#_Toc477521102)

[物体选择 102](#_Toc477521103)

[文本 102](#_Toc477521104)

[图像 104](#_Toc477521105)

[计算几何 105](#_Toc477521106)

[HTML5视频和音频 107](#_Toc477521107)

[WebGL 109](#_Toc477521108)

[WebGL编程指南 109](#_Toc477521109)

[基本概念 109](#_Toc477521110)

[JavaScript程序与着色器交互 109](#_Toc477521111)

[绘图 110](#_Toc477521112)

# HTML

## HTML5权威指南

### 基本概念

Html开启标记和结束标记之间的文字会在该标记无法解析时直接显示，比如svg, canvas …

HTML5核心规范定义用以标记内容的元素，并明确其含义。CSS可用于控制标记过的内容呈现在用户面前的外貌。JavaScript则可以用来操纵HTML文档的内容以及响应用户的操作

语义元素可以用来说明内容的含义以及内容的不同部分之间的关系

会影响呈现形式的那些元素，通常产生于HTML的早期版本，当时将内容与其呈现形式分开这个理念尚未像如今一样得以严格实施

元素 <code></code>, <hr /> …

元素属性 <a href=”/apples.html”>…</a>

布尔属性 <input diabled>

自定义属性

是对HTML4中“浏览器应当忽略不认识的属性”这种广泛应用的技巧的正式规定。在这类属 性名称之前添加前缀data-是为了避免与HTML的未来版本中可能增加的属性名冲突。自定义属性与CSS和JavaScript结合起来很有用

<input data-creator=”adam” data-purpose=”collection”>

用于处理HTML文档的各种软件有一个共同的名称叫做用户代理(user agent).浏览器是最流行的用户代理，但不是唯一的一种

<!DOCTYPE HTML>

<html>

<head>

<title>…</title>

</head>

<body>

</body>

</html>

文档对象模型DOM也会涉及通过搜索文档树查找文档中某个元素，而文档树正是元素之是关系的一种表述。

HTML实体：实体是浏览器用来替代特殊字符的一种代码, 见p21

**class属性**：用来将元素归类。通常是为了能够找出文档中的某一类元素或某一类元素应用CSS样式

**dir属性**：规定元素中文字的方向

**draggable and dropzone属性**：支持拖放操作

**hidden布尔属性**：浏览器对它的处理方式是隐藏相关元素

**id属性**：用来给元素分配唯一的标识符。通常用来将样式应用到元素上或在JavaScript程序中用来选择元素，id属性还可以用来导航到文档中的特定位置。该URL的末尾部分(#加上元素id值)称为URL片段标识符

**spellcheck属性**：用来表明浏览器是否应该对元素的内容进行拼写检查。这个属性只有用在用户可以编辑的元素上时才有意义。

**style属性**

**title属性**：提供元素的额外信息，浏览器通常用这些东西显示工具提示

### CSS样式

如果样式表中未声明所使用的字符编码，那么浏览器将使用载入该样式表的HTML文档声明的编码。要是HTML文档也没有声明其编码，那么默认情况下使用的将是UTF-8

#### 样式来源

浏览器根据层叠和继承规则确定显示一个元素时各种样式属性采用的值。对于每一个这种属性，浏览器都需要查看一下其所有样式来源，五种：

1. 元素内嵌
2. 文档内嵌
3. 外部样式表
4. 浏览器样式（用户代理样式）：

是元素尚未设置样式时浏览器应用在它身上的默认样式，这些样式因浏览器而略有差异

1. 用户样式：

大多数浏览器都允许用户定义自己的样式表。各种浏览器都有自己管理用户样式的方式。以谷歌的Chrome为例，它会在用户的个人配置信息目录中生成一个名为Default\User StyleSheets\Custom.css的文件。添加到这个文件中的任何样式都会被用于用户访问的所有网站。

#### 样式如何层叠？

1. 元素内嵌样式（用元素的全局属性style定义的样式）
2. 文档内嵌样式（定义在style元素中的样式）
3. 外部样式（用link元素导入的样式）
4. 用户样式（用户定义的样式）
5. 浏览器样式（浏览器应用的默认样式）

#### 样式的具体程度

通过统计三为特征得出：

样式的选择器中id值的数目

选择器中其他属性和伪类的数目

选择器中元素名和伪元素的数目

#### 样式继承

如果浏览器在直接相关的样式中找不到某个属性的值，就会求助于继承机制，使用父元素的这个样式属性值。

并非所有CSS属性均可继承，与元素外观（文字颜色、字体等）相关的样式会被继承。与元素在页面上的布局相关的样式不会被继承。

#### 相对长度

em 与元素字号挂钩

rem 与根元素的字号挂钩

px CSS像素

% 另一属性的值的百分比

对于font-size属性，挂钩的是元素继承到的font-size值；而对于width属性，挂钩的则是元素的包含块的宽度。

### JavaScript

JavaScript: 定义在函数中的变量，只能在该函数范围内使用。直接在script元素中定义的变量称为全局变量，可以在任何地方使用—包括在其他脚本中。

JavaScript是一种弱类型语言，但这不代表它没有类型，而是指不用明确声明变量的类型以及可随心所欲地用同一变量表示不同类型的值。

JavaScript数组会自动调整大小以便容纳所有元素。JavaScript数组可以混合包含各种类型的数据。

try{

} catch ( e ) {

}

若有错误发生，那么try子句中语句的执行将立即停止，控制权转移到catch子句中。发生的错误由一个Error对象描述，它会被传递给catch子句。

HTML5元素选用原则

在语义方面要求严格点不为过，只要有条件，尽量避用那些具有浓重呈现意味或纯粹起呈现作用的元素

呈现与语义分离原则的目的完全是为了让HTML文档更易于程序化处理

少亦可为多

开发在使用元素的时候容易忘乎所有，把文档弄得标记密布，经验法则：问问自己打算如何发挥一个元素的语义作用，如果不能马上答出就不用这个元素

别误用元素

**对内容进行标记时，只宜将元素用于它们原定的用途，不要创造自有语义。如果找不到适合自己所要含义的元素，可以考虑使用通用元素(如span or div)，并且用全局属性class表明其含义。**

具体为佳，一以贯之

用来标记内容的元素应该选择最为具体的那个。如果已有元素能恰当表明内容的类型，就不要使用通过元素。同一个元素的使用在整个页面、网站或Web应用系统上要保持一致。

**文档和元数据元素**： base, body, DOCTYPE, head, html, link, meta, noscript, script, style, title

**文本元素**： a, abbr, b, br, cite, code, del, dfn, em, i, ins, kbd, mark, q, rp, rt, ruby, s, samp, small, span, strong, sub, sup, time, u, var, wbr

**对相关内容分组**： blockquote, dd, div, dl, dt, figcaption, figure, hr, li, ol, p, pre, ul

**划分内容**，让每个概念，观点或主题彼此分隔开： address, article, aside, details, footer, h1-h6, header, hgroup, nav, section, summary

**表格元素**： caption, col, colgroup, table, tbody, td, tfoot, th, thead, tr

**表单元素**： button, datalist, fieldset, form, input, keygen, label, legend, optgroup, option, output, select, textarea

**嵌入内容元素**： area, audio, canvas, embed, iframe, img, map, meter, obect, pram, progress, source, svg, track, video

**文档和元数据元素**

<!DOCTYPE HTML>

告诉浏览器两件事情：第一，它处理的是html文档；第二，用来标记文档内容的html所属的版本。

html元素：是根元素，它表示文档中html部分的开始

head元素：包含文档的元数据。在html中，元数据向浏览器提供了有关文档内容和标记的信息，此外还可以包含脚本和对外部资源比如css样式表的引用

base元素：可用来设置一个基准url,让html文档中的相对链接在此基础上进行解析，相对链接省略了url中的协议、主机和端口部分，需要根据别的url得出其完整形式。如果不用base元素，或不用其href属性设置一个基准url,那么浏览器会将当前文档的url认定为所有相对url的解析基准。

meta元素： 可以用来定义文档的各种元数据。它有多种不同的用法。

指定名/值元数据对，name属性用来表示元数据的类型，content属性用来提供值

<meta name=”author” content=”QiZhong Lin”/>

声明字符编码

<meta charset=”utf-8”/>

style元素：可以出现在html文档中的各个部分。一个文档可包含多个style元素，因此不必把所有样式定义都塞进head部分。在使用模板引擎生成页面的情况下这个特性很有帮助，因为这样一来就可以用页面特有的样式为模板定义的样式提供补充。media属性：可用来表明文档在什么情况下应该使用该元素中定义的样式。规定设备值：all, screen, pring, projection, … 见p110; 使用特性：orientation, aspect-ratio, …见p112

link元素：

<link rel=”stylesheet” type=”text/css” href=”styles.css”/>

可用来在html文档和外部资源建立联系。rel属性：设定的值决定了浏览器对待link元素的方式

<link rel=”shortcut icon” href=”favicon.ico” type=”image/x-icon” />

为页面定义网站标志，定义与页面联系在一起的图标。各种浏览器处理这种图标的方式有所不同，常见的做法是将其显示在相应的标签页标签上或收藏夹中相应的项

如果网站标志文件位于/favicon.ico（即web服务器的根目录），那就不必用到link元素。大多数浏览器在载入页面时都会自动请求这个文件。

script元素：

不使用type属性时，浏览器会假定使用的是JavaScript。

默认情况下，浏览器在页面中一遇到脚本就会执行

为什么推荐将脚本文件引用放到<body>子元素最后？

<script src=”simple2.js”></script>

<body>

<p>I like <code id=”applecode”>apples</code> and oranges.</p>

</body>

simple2.js

document.getElementById(“applecode”).innerText = “Cherries”;

默认情况下，浏览器一遇到script元素就会暂停处理html文档，转而载入脚本文件并执行其中的脚本。在脚本执行完毕之后浏览器才会继承解析html。

所以例子中浏览器载入和执行simple2.js中的语句时还没有解析其余的html文档内容。脚本找不到要找的那个元素所以也就没有作出任何改变。脚本执行完毕之后，浏览器继续解析html,code元素也随之出现。然而对于脚本而言为时已晚。它不会再执行一遍。

解决办法

方法一：将script元素放到文档最后

<body>

<p>I like <code id=”applecode”>apples</code> and oranges.</p>

<script src=”simple2.js”></script>

</body>

方法二：使用defer属性。浏览器在遇到带有defer属性的script元素时，会将脚本的加载和执行推迟到html文档中所有元素都已得到解析之后。

<script defer src=”simple2.js”></script>

<body>

<p>I like <code id=”applecode”>apples</code> and oranges.</p>

</body>

async属性：

浏览器遇到script元素时的默认行为是在加载和执行脚本的同时暂时处理页面。各个script元素依次（即按其定义的次序）同步（即在脚本加载和执行进程中不再管别的事情）执行。

然而有些场景，异步执行脚本可以提高性能。如跟踪脚本 tracking script，这种脚本可以汇报用户的网站访问记录以便广告公司根据用户的浏览习惯定制和投放广告，或者收集网站访问的统计数据以供分析 ，诸如此类。这些脚本自成一体，一般不需要与html文档中的元素互相作用。为等待它们加载然后向自己的服务器发回报告而推迟显示页面没有任何意义。

**文本元素**

文本层面的元素，把这些元素加入文本当中，也就引入了结构和含义。

HTML5规范明确指出：使用元素应该完全从元素的语义出发。首先，如果存在符合需求的专用元素就用这种元素。其次避开那些补了点语义脂粉的呈现性元素（如b元素），把呈现工作交给CSS打理

<a href=””></a>

url不一定都要指向其他网面。尽管URL中用得最多的协议就是http,但浏览器也支持一些其他协议比如https,ftp,mailto…

如果href属性值不是以类似http://这样的已知协议开关，那么浏览器会将该超链接视为相对引用。默认情况下，浏览器会假定目标资源与当前文档位于同一位置。

**超链接也可用来将同一文档中的另一个元素移入视野。为此需要用到形如css中针对目标元素的id选择器的表达式：#id**

**如果该元素不在视野之中，那么浏览器会将文档滚动到能看见它的位置**

target属性的用途是告诉浏览器希望将所链接的资源显示在哪里。默认情况下，浏览器使用的是显示当前文档的窗口、标签页或窗框frame,所以新文档将会取代现在显示的文档,不过还有其他选择。如target=”\_blank”在新容器或标签页中打开文档

em元素：表示对一段文字的强调

em {font-style: italic;}

i元素：表示一段文字与周围内容有本质区别，不过这个元素常用的地方包括外文词语、科技术语甚至某人的想法

i {font-style: italic;}

strong元素：表示一段重要文字

strong { font-weight: bolder;}

u元素：使一段文字从周围内容中凸现出来，但并不表示强调或其重要性有所增加

u { text-decoration: underline; }

small元素：表示小号字体内容，常用于免责声明和澄清声明。

small { font-size: smaller; }

sub and sup元素：分别用于表示下标和上标

sub { vertical-align: sub; font-size:smaller;}

sup { vertical-align: super; font-size: smaller;}

br元素：会引起一次换行。其习惯样式是将后续内容转移到新行上。

wbr元素：是html5新增的，用来表示长度超过当前浏览器窗口的内容适合在些换行，究竟换不换地由浏览器决定。

表示输入和输出

code：表示计算机代码片段 code { font-family: monospace; }

var: 在编程语境中表示变量 var {font-style: italic; }

samp: 表示程序或计算机系统的输出 samp { font-family: monospace; }

kbd: 表示用户输入 kbd { font-family: monospace; }

abbr元素：用来表示缩写，其title属性表示的是该缩写代表的完整词语

dfn元素：表示定义中的术语，也即在用来解释一个词（或短语）的含义的句子中的词或短语

q元素：表示引自他处的内容，q元素的cite属性可以用来指定来源文章的url.

cite元素：表示所引用作品如图书、文章、电影和诗歌的标题

cite { font-style: italic; }

span元素

mark元素：是html5新增的，用来表示因为与某段上下文相关而被突出显示的一段文字。

ins元素 and del元素：分别用来表示文档中添加和删除的文字

该使用什么文本元素要根据它们的含义而不是习惯样式来选择。

# CSS

## 参考http://learn.shayhowe.com/html-css/

### 盒模型The box model



We set a width property value of 400 pixels, but the actual width of our element is 492pixels. By default the box model is additive; thus to determine the actual size of a box we need to take into account padding, borders, and margins for all four sides of the box.

### 元素布局

元素display：block, inline, line-block, none

默认div, p, form, header, footer, section是block元素，span, a, img是inline元素

block, line-block元素有width, height, margin, border, padding属性；inline元素只有margin-left, margin-right, padding

document一般是自然从上到下排列，block占一行，若将一行分为几个block，常见如下方法：

方法1: The float property allow elements to be floated to the left or right of their parent element.

1. 行内的div赋float:left, width:30%, 行后赋clear:both

方法2:

1. 行内的div赋display:line-block,同时div之间加<!----> (为什么？remove the space between inline-block elements)

### 元素定位

static: 默认所有元素position:static,元素将按照它们在标记里出现的先后顺序出现在浏览器窗口里。

relative(相对定位)：若div赋position:relative, top:20px, left:20px,则div相对于本应在文档的位置位移(deltaX=top, deltaY=left)

**absolute(绝对定位)：若div赋position:absolute, top:20px, left:20px,则div相对于最近的具有position:relative的父元素，若该父元素没有，则相对于body元素。把它摆放到容纳它的“容器”的任何位置。**

Fixed(固定定位)：

overflow处理子元素尺寸超出父元素容器尺寸

visible: 全显示子元素

**hidden: 隐藏溢出的内容，只显示在父元素的显示区域，用来对子元素内容进行“裁剪”**

scroll: 显示滚动条

auto: 发生溢出时才显示滚动条

如何实现div占满浏览器视口？

设置子元素的高度为父元素的100%，前提条件父元素得有高度，而浏览器默认元素的height:auto，所以得层层往上直至html设置height:100%

<html style="height: 100%;">

<body style="height: 100%;">

<div style="height: 100%;">

<p>

这样这个div的高度就会100%了

</p>

</div>

</body>

</html>

若有margin, padding,而宽度和高度是内框，从而会出现滚动条，所以需要用上从外往里框设计模式

如何实现div垂直居中？

Table -> table-cell拥有垂直居中属性，可以用display模拟实现

<div class="wrap-table">

<div class="content-table">

<h1>Cover your page.</h1>

<p class="lead">Cover is a one-page template for building simple…</p>

<p><a href="#" class="btn btn-default">Learn more</a> </p>

</div>

</div>

.wrap-table {

height: 100%;

display:table;

text-align: center;

}

.content-table {

display:table-cell;

vertical-align: middle;

}

## <Bootstrap Example>

### 如何用CSS实现三角形图标？

//元素宽度高度为0,只有4px的边框实现为

{

width: 0;

height: 0;

border: 4px solid;

}

//若边框修改为如下，则实现为



{

width: 0;

height: 0;

border-top: 40px solid #000;

border-right: 40px solid #f00;

border-bottom: 40px solid #f00;

border-left: 40px solid #f00;

}

以上样式，若没有下边框，且左右边框透明，则只显示下三角形。当然也可以显示上三角，或右三角，左三角等

//右三角

{

width: 0;

height: 0;

border-top: 40px solid transparent;

border-left: 40px solid #ff0000;

border-bottom: 40px solid transparent;

}

也可左上三角，右上三角，左下三角，左上三角

#triangle-topleft {

width: 0;

height: 0;

border-top: 100px solid red;

border-right: 100px solid transparent;

}

#triangle-topright {

width: 0;

height: 0;

border-top: 100px solid red;

border-left: 100px solid transparent;

}

#triangle-bottomleft {

width: 0;

height: 0;

border-bottom: 100px solid red;

border-right: 100px solid transparent;

}

#triangle-bottomright {

width: 0;

height: 0;

border-bottom: 100px solid red;

border-left: 100px solid transparent;

}

如何实现导航条，选中导航项白字，下三角表示？



<div class="blog-header">

<div class="container">

<nav class="blog-nav" bs-active-link>

<a class="blog-nav-item active" href="#/">Home</a>

<a class="blog-nav-item" href="#/features">New features</a>

<a class="blog-nav-item" href="#/press">Press</a>

<a class="blog-nav-item" href="#/hires">New hires</a>

<a class="blog-nav-item" href="#/about">About</a>

</nav>

</div>

</div>

.blog-header {

background-color: #428bca;

-webkit-box-shadow: inset 0 -2px 5px rgba(0,0,0,.1);

box-shadow: inset 0 -2px 5px rgba(0,0,0,.1);

}

.blog-nav-item {

**position: relative;**

**display: inline-block;**

**padding: 10px;**

color: #cdddeb;

font-weight: 600;

}

**.blog-nav-item:hover,**

**.blog-nav-item:focus {**

**color: #fff;**

**text-decoration: none;**

**}**

**.blog-nav-item.active {**

**color: #fff;**

**}**

**.blog-nav-item.active:after** {

position: absolute;

bottom: 0;

left: 50%;

width: 0;

height: 0;

margin-left: -5px;

vertical-align: middle;

content: " ";

border-right: 5px solid transparent;

border-bottom: 5px solid;

border-left: 5px solid transparent;

}

### 布局：上导航栏固定，左导航栏也固定，剩下空间动态



//上导航条 固定

<nav class="navbar navbar-inverse navbar-fixed-top">

<div class="container-fluid">

...

</div>

</nav>

.navbar-fixed-top {

border: 0;

}

/\* Move down content because we have a fixed navbar that is 50px tall \*/

body {

padding-top: 50px;

}

//左导航栏 固定铺满，注意去除col的左右margin

<div class="container-fluid">

<div class="row">

<div class="col-md-2 **sidebar**">

<ul class="nav **nav-sidebar**">

<li class="active"><a href="#">Overview</a></li>

<li ><a href="#">Reports</a></li>

</ul>

<ul class="nav nav-sidebar">

<li ><a href="#">Nav item</a></li>

<li ><a href="#">Nav item again</a></li>

</ul>

</div>

.sidebar {

position: fixed; //固定

top: 51px; //注意上导航条的高度

bottom: 0; //拉伸左导航栏高度

padding: 20px;

background-color: #f5f5f5;

border-right: 1px solid #eee;

overflow-x: hidden;

overflow-y: auto;

}

.nav-sidebar {

margin-left: -21px; //去除col布局带来的影响

margin-right: -21px;

margin-bottom: 20px;

}

.nav-sidebar > li > a {

padding-left: 20px; //内留空间

padding-right: 20px;

}

.nav-sidebar > .active > a,

.nav-sidebar > .active > a:focus,

.nav-sidebar > .active > a:hover

{

color: #fff; //highligh

background-color: #428bca;

}

### 登录框样式

输入框左上右上有拐角？

checkbox与label对齐？

同时增大字体大小与行高度？



<div class="container">

<form **class="form-signin"**>

<h2 class="form-signin-heading">Please sign in</h2>

<label for="inputEmail" class="sr-only">Email address</label>

<input type="email" id="inputEmail" **class="form-control"** placeholder="Email address" required autofocus>

<label for="inputPassword" class="sr-only">Password</label>

<input type="password" id="inputPassword" class="form-control" placeholder="Password" required>

**<div class="checkbox">**

**<label><input type="checkbox" value="remember-me"> Remember me</label>**

**</div>**

<button type="submit" class="btn btn-lg btn-primary btn-block">Sign in</button>

</form>

</div>

//登录表单限制大小，且居中

.form-signin {

max-width: 340px;

margin: 0 auto;

padding: 15px;

padding-top: 40px;

}

//增大字体大小，且自动增加高度，**先设字体大小，然后设置高度为自动。才会刷新高度**

.form-signin .form-control {

padding: 10px;

font-size: 16px;

height: auto;

}

//去除指定拐角

.form-signin input[type="email"] {

border-bottom-left-radius: 0;

border-bottom-right-right: 0;

}

.form-signin input[type="password"] {

border-top-left-radius: 0;

border-top-right-radius: 0;

}

//checkbox and label包裹div，设置div才能使checkbox and label等高

.form-signin .checkbox {

margin-top: 20px;

margin-bottom: 20px;

font-weight: normal;

}

### 修改bootstrap的类样式？

<div class="jumbotron">

<h1>Marketing stuff!</h1>

<p class="lead">…</p>

<p><a href="#" class="btn btn-success btn-lg">Get started today</a></p>

</div>



自定义为如下：



.jumbotron {

background-color: transparent;

text-align: center;

.btn {

padding: 14px 24px;

font-size: 21px;

}

}

### Bootstrap默认值

font-size: 14px

line-height: 1.428

p { margin-bottom: 10px;}

标题默认值



Marked text: 

You can use the mark tag to <mark>highlight</mark> text.

Deleted text: 

<del>This line of text is meant to be treated as no longer accurate.</del>

Underlined text: 

<u>This line of text will render as underlined</u>

Small text: 85% size of the parent

<small>This line of text is meant to be treated as fine print.</small>

Bold text: 

<strong>rendered as bold text</strong>

Italics text: 

<em>rendered as italicized text</em>

Inline text: 

For example, <code>&lt;section&gt;</code> should be wrapped as inline.

User input text: 

To switch directories, type <kbd>cd</kbd> followed by the name of the directory.<br>

To edit settings, press <kbd><kbd>ctrl</kbd> + <kbd>,</kbd></kbd>

Basic block text: 

<pre>&lt;p&gt;Sample text here...&lt;/p&gt;</pre>

Variables text: 

<var>y</var> = <var>m</var><var>x</var> + <var>b</var>

字体color



<p class="text-muted">...</p>

<p class="text-primary">...</p>

<p class="text-success">...</p>

<p class="text-info">...</p>

<p class="text-warning">...</p>

<p class="text-danger">...</p>

背景color



<p class="bg-primary">...</p>

<p class="bg-success">...</p>

<p class="bg-info">...</p>

<p class="bg-warning">...</p>

<p class="bg-danger">...</p>

### 固定上导航栏，下脚注？



使用navbar-fixed-top and navbar-fixed-bottom, Body padding required

The fixed navbar will overlay your other content, unless you add padding to the top of the <body>. Try out your own values or use our snippet below. Tip: By default, the navbar is 50px high.

body { padding-top: 70px; }

<nav class="navbar navbar-default **navbar-fixed-top**">

<div class="**container**">…</div>

</nav>

<div class="container" >...</div>

<footer class="**navbar-fixed-bottom**">

<div class="**container**">

<p class="text-muted">Place sticky footer content here.</p>

</div>

</footer>

body {

padding-top: 50px;

padding-bottom: 60px;

}

.navbar-fixed-bottom {

background-color: #f5f5f5;

height: 60px;

}

.container .text-muted {

margin-top: 20px;

margin-bottom: 20px;

}

## <HTML5与CSS3设计模式>

### 简化CSS使用

#### CSS语法：

使用空格分隔多个类名

常量值不能加引号，如color:black

CSS注释：/\* ... \*/

当使用多个单词表示一个CSS属性名称或常量属性值时，应该使用连字符分隔单词，如font-family

#### 层叠顺序：

通配选择器

元素选择器

类、属性和伪选择器

ID选择器

#### CSS属性值

常量文本、常量数字、长度、百分数、函数、逗号分隔值和空格分隔值。

##### 常用属性：

display: inline, none, block, inline-block, list-item, table-cell, table, table-row

visibility: visible, hidden

cursor: auto, default, pointer, help, wait, progress, move, crosshair, text, …

background

background-color: transparent, COLOR

background-image: none, url(“file.jpg”)

background-repeat: repeat, repeat-x, repeat-y, no-repeat

background-position: left top, left bottom, right top, right bottom, center center, …

border: WIDTH STYLE COLOR

border-width: medium, LENGTH, thin, thick

border-style: none, hidden, dotted, dashed, solid, double, groove, ridge, inset, outset

border-color: black, COLOR

border-left: WIDTH STYLE COLOR

border-left-width:

border-left-style:

border-left-color:

(same to border-right, border-top, border-bottom, …)

padding: 0.25em; 将padding设置为字号的1/4

padding-left:

padding-right:

padding-top:

padding-bottom:

font: caption, icon, menu, message-box, small-caption, status-bar

font-family: “Century Gotic”, verdana, arial, sans-serif;

font-size: medium, 20px, xx-small, x-small, smaller, small, large, larger, x-large, xx-large

font-style: italic, normal, oblique

font-weight: normal, bold

text-decoration: none, underline, line-through, overline

line-height: 150%; 将line-height设置为font-size值的150%

color: black, …, #rrggbb, #rbg, rgb(100%, 100%, 100%), rgb(255, 255, 255)

##### 布局属性

float: none, left, right 浮动属性适用于除单元格和行以外的所有元素

clear: none, left, right, both 复位属性适用于除行内、行内块、单元格和行以外的所有元素

position: static, relative, absolute, fixed 空位属性适用于除单元格和行以外的所有元素

left: auto, LENGTH, %WidthOfContainingBlock

right:

top:

bottom:

z-index: INTEGER

margin: 0, LENGTH, %WidthOfContainingBlock, auto 除单元格和行以外的所有元素

margin-left:

margin-right:

margin-top:

margin-bottom:

(margin: 1em; 将margin设置为字号的1倍大小)

width：auto, LENGTH, %WidthOfContainingBlock 除行内与行以外的所有元素

height:

min-width:

max-width:

min-height:

max-height:

text-indent: 0, LENGTH, %WidthOfContainingBlock

text-align: left, center, right, justify

overflow: visible, hidde, auto, scroll

##### 专用属性

列表属性仅适用于列表元素

list-style: TYPE POSITION IMAGE

list-style-type: disc, circle, square, none, decimal, …

list-style-position: outside, inside

list-style-image: none, url(“file.jpg”)

表格属性仅适用于表格元素

border-collapse: separate, collapse

table-layout: auto, fixed

单元格属性仅适用于单元格元素

vertical-align: baseline, bottom, middle, top

行内属性仅适用于行内和行内块元素

vertical-align: baseline, LENGTH, %LineHeight, text-bottom, text-top, middle, top, bottom

换页属性仅适用于块和表格元素

page-break-after: auto, always, avoid

page-break-before: auto, always, avoid

##### 尺寸单位

em 指元素设置的font-size值。对于font-size属性，它指的是父元素的font-size,例如5em表示font-size的5倍。如果要相对于文字大小设置元素尺寸，那么非常适合使用em作为度量单位。使用这种方法可以实现根据文本大小自动调整文档布局。使用em可以粗略地设置元素宽度，使之适合显示特定个数的字符。计算方法就是将字符数乘以0.625得到em值。例如如果想要将元素宽度设置为10个字符，那么要将它设置为6.25em

#### CSS排错

如果发现选择器中一些属性有效，一些属性无效，那么无论使用了什么值，往往都可以确定是层叠优先级出现了问题。通常我们可以在属性后添加!important来确认这个问题。

简写属性会将值赋给它所代表的全部属性，哪怕只设置了一个值。例如

background: blue; 等价于

background-color: blue;

background-image: none;

background-repeat: repeat;

background-attachment: scroll;

background-position: 0% 0%

还有简写属性： margin, border, padding, background, font, list-style

#### 为什么要设置样式表的规范化？

由于各个浏览器的默认设置存在差异，我们可能需要在样式表中建立一些规则用以定义每一个元素的基线设置。例如，不同的浏览器会为<h1>元素设置不同的大小和外边距。通过设置<h1>的大小和外边距，便可以统一它在所有浏览器的显示效果

### HTML设计模式

HTML将元素分成三大类：结构、块级和行内元素

结构元素：<html>, <head> and <body>. 文档信息位于<head>元素，文档内容位于<body>元素之内。

Web服务器通过指定MIME内容类型来区分所保存的文档。MIME代表多部分互联网邮件扩展(Multipart Internet MailExtensions). 内容类型由文档的HTTP头信息指定。浏览器根据文档的MIME内容类型来决定文档的处理方式。

#### 块元素

结构块：

<ol>: 有序列表， 集合

<ul>：无序列表， 不排序的集合

<dl>：定义列表， 关联实体，键值

<table>：关系型结构

辅助结构：<li>, <dt>, <dd>, <caption>, <thead>, <tfoot>, <colgroup> and <col>

多功能块：

多功能块可以包含块或内容，但是不能同时包含块和内容。内容即文字与行内元素（图片、对象、控件和语义标签）因为**块在页面上是纵向排列**的，而**内容是横向排列**的，所以浏览器无法同是渲染块和内容

<div>: 可以是**结构型或终止型。除非设置了每一个分区的外边距、边框和内边框，否则用分区创建的块结构是不可见的**

<li>：列表项目。**通常是一个包含内容的终止块，但也可以包含表格和列表等结构，或者标题和段落等终止块**

<dd>:

<td>, <th>: 单元格通常是包含内容的终止块，但是也可以**包含结构块或终止块**

<form>:表单可以包含用于组织表单控件的结构块，或者直接包含行内表单控件。此外，还可以包含标题和段落等终止块

<article>: 表示一篇论坛帖子，一篇杂志或报刊文章，一篇博客文章等

<section>: 表示文档的一个普通节，它可以对内容进行主题分组，一般带有标题

<nav>: 包含导航链接的节。

终止块：

文档块结构的结束节点，**它不能包含其他块，只能包含文字或行内元素**。

<h1>: 标题

<p>： 段落

<blockquote>：块级引用

<dt>：术语

<address>：联系人记录

<caption>：表标题

#### 行内元素

**HTML使用行内元素来确定文字含义，控制文字排列，以及在文档中插入外部内容（如图片和控件）**。行内元素都是内容。行内元素与文字可以混合使用。

语义元素

<span>: 普通文字，表示一般重要性

<em>, <strong>：搜索引擎会使用<em> and <strong>对内容进行排序

<a>,<cite>,<code>,<kbd>,<samp> and <var>: 词组

<abbr>, <dfn>: 单词

<sub>, <sup>: 字符

流控制元素

<br />

<bdo>

替换元素

<img>: 替换成图片

<object>: 替换成视频, flash电影和音频文件

控件：表单中用来填写数据的行内元素

<input>, <textarea>, <select>, <button>

#### 类class

为元素指定用户自定义的语义。Class是扩展HTML元素语义含义的主要机制。在元素的class属性中指定多个类名，就可能给一个元素同时设置多个类。**类应该使用表示语义的名称**，如copyright, date, price, back-to-top, example, figure, listing, illustration, note, result, tip, warning.等

#### ID

应该使用表示语义的名称，如skip-to-main-content, page, preheader, title, header, search, postheader, body, nav, site-map, links, main, section1, section2 news, about-us, services, products等

因为<div> and <span>元素不具有语义所以可以给它们设置任意类，而不会与预定的语义发生冲突。**给<div>设置类，就可以创建出具有自定义语义的自定义文档结构。给<span>设置类，就可以创建自定义的文字含义**。

#### HTML空白字符

浏览器会将多个重复的空白字符合并为一个。因此，标记代码中可以插入额外的空格、制表符、换行符和回车符，以增加其可读性，但不会在文档渲染时显示。

### CSS选择器与继承

选择器分组：要给不同的选择器应用同一组规则，可以使用**逗号将多个选择器串连**一起。

后代元素选择器：要选择后代元素，可以使用**空格将多个选择器串连**到一起

子选择器：要选择子元素，可以使用**大于号将多个选择器串连**在一起

第一个子元素：要选择第一个子元素，可以在**任意选择器之后添加:first-child**

相邻兄弟元素选择器：要选择相邻兄弟元素，可以使用**加号将多个选择器串连**在一起

属性选择器

属性存在性选择器：选择包含指定属性的元素，如p[title]

属性单词选择器：选择在指定属性中包含指定单词的元素，如p[title~=”para”]

属性选择器基于任意子字符串匹配方式。如

[attr^=val] 表示attr属性值以”val”开头的元素

[attr$=val] 表示attr属性值以”val”结束的元素

[attr\*=val] 表示attr属性值包含”val“的元素

伪元素选择器：在选择的类、ID和类型组合使用首字母(first-letter)与首行(first-line)伪元素选择器。 first-letter and first-line选择器仅适用于终止块元素，不适用于行内元素或结构块元素。**first-letter and first-line称为伪元素选择器是因为它们只选择元素的一部分内容，而非元素的所有内容。换言之，它们创建了一个伪元素**

伪类选择器

a:link 选择未访问的超链接

a:visited 选择已访问的超链接

a:hover 选择鼠标正悬停的超链接

a:focus, a:active 选择正聚焦的超链接

**子类选择器**: 创建由类和子类构成的层次规则，和面向对象的编程一样，**子类元素会”继承“基类和子类的规则**。CSS层叠顺序保证子类的规则能够覆盖基类的规则

<ELEMENT class=”class subclass etc”>

\*.class { shared\_base\_styles }

\*.class.subclass.etc { subclass\_styles }

继承： 当浏览器遇到继承属性时，它会自动选择后代行内元素，然后为它们应用继承属性的规则。如果直接设置一个元素的属性，那么它会覆盖所有继承的值

所有元素会继承： visibility, cursor

行内元素会继承： line-height,

color,

font,

font-family,

font-size,

font-style,

font-weight

text-decoration

终止块元素会继承： text-align, text-indent

列表元素会继承： list-style, list-style-type, list-style-position, list-style-image

表格元素会继承： border-collapse

可视化继承：**CSS会自动给元素设置透明背景**。子元素位于父元素之上。如果由于外边距或位置设置使相邻兄弟元素重叠，那么靠后的相邻兄弟元素会覆盖靠前的相邻兄弟元素。对于浮动和设定位置的元素，可以使用z-index属性明确地设置它们的叠放次序。**background-color属性的默认值为transparent, background-image属性的默认值为none**,这些默认值使祖先元素的背景能够透过子元素显示出来。由于子元素嵌套在父元素之内，所以每一个子元素会可视化继承其父元素的边框和内边距。如果将子元素设置为透明背景和无边框，那么最终显示效果就是，父元素的边框和内边距相当于子元素的边框和内边距。

除以上元素，别的不可继承

### 框模型

使用display属性可以将一个元素渲染为另一种类型的框

display: inline 元素 -> 行内元素

display: inline-block 行内元素 -> 行内元素之中的块级元素

display: block 元素 -> 块级元素

display: list-item 元素 -> 列表项目

display: none 隐藏元素

position:absolute或position:fixed 元素 -> 绝对框

float: left或float: right 元素 -> 浮动框

= 行内型、行内块级型、块级型、表格型、绝对型和浮动型, 由display, position and float组合确定。

#### 行内框

**在行内流flow中渲染，它们在横向由左往右排列，然后在超出最近终止块祖先元素宽度时换到新的一行。**

**行内元素不起作用的属性：width, height, overflow. 因为它们总是会收缩以适应内容的宽度和高度。**

font, font-size决定文字的宽度，从而确定元素的宽度 收缩适应

居中偏移的行内元素

margin-left:value使元素向右偏移（或负值向左偏移），还要在上级容器设置text-align:center

垂直居中

将行内元素的line-height:value设置为父元素height属性相同的度量值或百分数，不可以创建垂直居中对齐的行内元素

左右内外边距，行高，左右边框

{

display: inline;

line-height:

margin-left:

margin-right

padding-left

padding-right

border-left

border-right

background

}

#### 行内块级框

**包括可替换元素和显示为行内块的行内元素**

**行内块级框在行内流中与行内框相似，但是具有与块级框相似的内外边距、边框、宽度和高度。**

**行内块级框不能换行。**

**行内块级框会增加行高，以适应其高度、内外边距和边框。**

widith:auto; height:auto 收缩适应

width: 像素,em,百分数 设定大小

width: 100% 拉伸适应

margin-top: +VALUE; 扩大行高 margin-top:-VALUE; 缩小行高

margin-bottom:+VALUE; 抬高元素 margin-bottom:-VALUE; 降低元素

margin-left:+VALUE; 增大元素与前一个元素的距离 margin-left:-VALUE; 接近

margin-right:+VALUE; 增大元素与后一个元素的距离 margin-right:-VALUE; 拉近

border and padding可以增大行内元素的外部尺寸，同时将元素及后续内容往右移动。

{

display: inline-block;

line-height:

overflow:

visibility:

width:

height:

margin:

padding:

border:

background:

}

不需要给要替换元素设置display: inline-block, 因为浏览器会自动将它们显示为行内块。

#### 块级框

块级框是由上至下垂直排列的，这就是所谓的正常块级流。块级框可以包含其他块级框，可以终止块格式化上下文，也可开启一个包含行内框的行内格式化上下文。终止块会在其内部框中创建一个行内格式化上下文，但是它属于其外部框之外的块级格式化上下文

**块可以拉伸至父元素，或设定尺寸，若尺寸大于父元素**，那么它会溢出父元素。overflow属性可以控制器中处理溢出的显示方式。

不能横向收缩适应

width: 像素,em,百分数 设定大小

width:auto; height:auto; 拉伸至父元素

width: 100%也会拉伸到其父元素的宽度，但是元素的水平外边距，边框或内边框会扩大宽度，使之超过父元素的宽度

margin-left, margin-right;可以缩进或凸出拉伸的块级元素。而且它们会使设定尺寸的块在两边产生偏移。

margin-left:auto块对齐到右边

margin-right:auto块对齐到左边

margin-left:auto & margn-right:auto 块居中父元素

margin-top, margin-bottom

静态块级元素对齐

相对左边偏移

position:static;

text-align:left;

width:value;

margin-left:value; 正值使元素向右偏移，负值则向左偏移

margin-right:auto;

相对右边偏移

position:static

text-align:right

width:value

margin-left:auto

margin-right:value 正值使元素向左偏移，负值则向右偏移

居中对齐

position:static

text-align:center

width:value 设定尺寸型 width:auto 拉伸型

margin-left:auto margin-left:value

margin-right:auto margin-right:value

上偏移（若margin-top:0则为上对齐）

使用margin-top:0可使元素对齐到顶边。使用margin-bottom:auto阻止元素对齐到底边

position:static

height: value

margin-top: value 设定尺寸型 margin-top:0 收缩适应型

margin-bottom:auto margin-bottom:0

下偏移

使用margin-bottom:0可使元素对齐到底边。使用margin-top:auto阻止元素对齐到顶边

position:static

height: value

margin-top: auto

margin-bottom:value

border, padding

#### 表格框

#### 绝对框

绝对元素会被从正常流中移除，放入另一个位于正常流之上或之下的布局层。它的位置相对于最近的已定位祖先元素。任何元素都可以采用绝对定位方式。与其化框模型不同，绝对框的位置不会影响其他框的位置。绝对框可以随意重叠。

z-index可以控制确定位置的元素叠放顺序。负值表示将元素置于正常流之下，正值表示将元素置于正常流之上。

绝对元素定位：给绝对元素的left设置度量值，则它的左边以上级容器的左边为准偏移量即设置的正负值. right,top and bottom的效果相似。

绝对元素大小：

若width:auto, left:auto, right:auto 则框为收缩适应型

若width: 像素,em,百分数, left: value, right:auto 则框为设定尺寸型，并且从左边偏移距离, height, top, bottom类似. 百分数是相对于其最近定位祖先元素的宽度和高度

若width:auto, left:value, right:value 则框为拉伸型

若尺寸或收缩适应型，且margin正值或负值 设置其对齐点的偏移量

若拉伸型，且margin正值 缩进元素各边（负值凸排各边）

绝对元素对齐，偏移，居中，外部对齐

使元素相对左边偏移

left:0 and right:auto 元素左对齐

margin-left:value 元素相对左边向右偏移 （负值相对左边向左偏移）

使元素相对于右边偏移

left:auto; right:0; 元素右对齐

margin-right:value 元素相对右边向左偏移 （负值相对右边向右偏移）

使元素相对于上边偏移

top:0; bottom:auto 元素上对齐

margin-top: value 元素相对上边向下偏移 (负值相对上边向上偏移)

使元素相对于下边偏移

top:auto; bottom:0 元素下对齐

margin-bottom: value 元素相对于下边向上偏移 (负责相对下边向下偏移)

使元素水平居中

width:value 元素宽度

left:0; right:0 元素左对齐和右对齐

margin-left:auto; margin-right:auto 使元素居中对齐

使元素垂直居中

height:value 元素高度

top:0; bottom: 0 元素上对齐和下对齐

margin-top:auto; margin-bottom:auto 使元素居中对齐

可以将绝对元素对齐到最近定位祖先元素的任意一边外部。因为100%是指元素容器的宽度，所以将元素一边的偏移值设置为100%，就可以使它对齐到容器另一边的外部。此外，使用margin可以使元素进一步偏移。

position:absolute; right:100%; margin-right:offset 对齐到左边外部的绝对元素

position:absolute; left:100%; margin-left:offset 对齐到右边外部的绝对元素

position:absolute; bottom:100%; margin-bottom:offset 对齐到上边外部的绝对元素

position:absolute; top:100%; margin-top:offset 对齐到下边外部的绝对元素

border, padding

#### 浮动框

使用float:left or float: right，可以使任意元素变成浮动元素。浮动元素会离开正常流，移动到相邻块的边框和背景之上。浮动元素的精确位置是不能预先确定的

width:auto, left:auto, right:auto 则框为收缩适应型

width: 像素,em,百分数 则框为设定尺寸型

width:auto 浮动元素拉伸到所在行的最大宽度

height:auto 浮动元素拉伸到所有子块或行的总高度

margin:value 正值外边距会使浮动元素远离其对齐点，并且将其化浮动元素和行内内容推到更远处。负值外边距会将浮动元素拉到对齐点的另一边，并且拉近浮动元素和行内内容的距离。浮动元素周围的外边距不会合并

border, padding 左浮动元素的左边框和左内边距会使浮动元素向右移动， 而右边框和右内边距会将其他浮动元素和行内内容进一步向右移动。

默认值

width：auto, height: auto

margin: 0

border: none

padding: 0

background: none transparent repeat left top scroll

overflow: visible

若元素边框透明则显示当前元素背景的，若外边距透明则显示父元素的背景和边框

最好避免使用overflow: hidden, scroll or auto,因为截去内容或要求用户使用滚动条、都会影响用户体验。只有在块级元素的尺寸小于显示内容时才需要使用这个属性。

visibility: visible 显示文本 visibility: hidden 隐藏文本

display:none 删除文本 text-decoration: line-through 添加删除线

### 辅助概念

#### 框定位

如何将元素相对于当前位置、最近定位祖先元素或视口偏移一定距离？

position: static 默认值

position: relative 元素相对于常规流的位置偏移一定距离

position: absolute 元素相对于最近定位祖先元素的位置偏移一定的距离

position: fixed 元素相对于视口偏移一定的距离

left: value元素左边相对于参考位置的左边偏移，正值向右偏移，负值向左偏移

right:value 元素右边相对于参考位置的右边偏移，正值向左偏移，负值向右偏移

top:value 元素的上边相对于参考位置的上边偏移，正值向下偏移，负值向上偏移

bottom:value 元素的下边相对于参考位置的下边偏移，正值向上偏移，负值向下偏移

将设定位置的元素相对于最近定位祖先元素进行定位，其主要目的是创建自包含布局。当为自包含布局重新设定位置时它的所有后代元素都会随之移动—包括设定位置和非设定位置的元素。自包含定位布局的嵌套层次可以包含任意深度。这是一个创建可重用布局的强大特性。虽然定位功能非常强大，但是它的最大缺点是要求必须给元素设定尺寸。Position:relative是一个非常好的创建定位祖先元素的方法，因为它不会离开常规流。使用这种方法，能够创建出既保持常规流又实现绝对定位的布局

##### 静态定位 position:static 默认值

元素定位于常规流中，常规流由视口中垂直显示的嵌套组成。在块内部，一个或多个块或行是按垂直向下方式排列的。在一行中，文字和对象是水平排列的。

##### 绝对定位 position:absolute

将元素从常规流中移到独立的定位层，相对于最近定位祖先元素。使用width and height可以设置它的尺寸，百分数是相对于最近定位祖先元素的尺寸而言，而非父元素的尺寸。将元素的left,right,top and bottom设置为特定值，可以使它对齐到最近定位祖先元素的各边。

##### 固定定位 position:fixed

将元素移到独立的定位层，相对于视口的固定位置。元素不会随视口滚动而滚动，其他方面与绝对定位完全相同。使用固定定位元素，可以精确控制元素在视口中的位置。它们不会随着视口的滚动而滚动，很适合显示一些控件，如菜单，工具栏和按钮等。

##### 相对定位 position: relative

设置元素为最近定位祖先元素或者相对于常规流中的位置，使元素偏移，且满足两个要求：一是不改变元素在常规流中的形状，二是元素偏移不会影响其他元素的位置

##### 浮动定位 float:left or right

使元素离开常规流并将它显示在其父元素内边距区域的左边或右边。浮动元素位于块级元素背景之上，紧跟常规流中行内内容之后的独立一层。浮动元素不会影响块级框的位置，而只影响行内内容。浮动元素会影响其他浮动元素的位置，并且可能与左边或右边的浮动元素发生重叠。

任意元素都可以设置为浮动元素。clear适用表格，块级元素和浮动元素。clear不适用于行内，绝对定位或固定定位的元素

#### 堆叠上下文

在设定位置元素中设置数字值z-index，就会得到一个本地自包含的堆叠上下文，元素的所有后代元素都在这个上下文中显示——包括静态定位、浮动定位和设定位置的元素。如果z-index设置为auto,或者给非设定位置元素指定z-index,则不会创建堆叠上下文。**使用以下值都会创建堆叠上下文：z-index: 0 or -1 or 9999. 每一个本地堆叠上下文都会将内部堆叠级别设置为0，然后其后代元素相对它的位置进叠放**。这是非常重要的一点：**z-index不是全局属性，而是相对于设置了数字值z-index的最近定位祖先元素而定。根元素<html>会创建根堆叠上下文**

#### 原子显示 (hasLayout, Grouped)

当浏览器将静态行内元素内容渲染在静态块级元素背景之上的独立分层上。当静态块级元素发生重叠时，它们的背景会重叠，但是它们的行内内容则不会重叠。因为堆叠上下文会先渲染所有块级元素的背景和边框，然后再显示所有浮动元素，最后显示所有行内元素和内容。这样层内块级元素的背景和边框就显示在浮动元素和行内内容之下。这似乎有些奇怪，因为我们一般都会认为行内内容位于所有的块之内，而非位于块之上。但是行内元素位于块之上是有意义的，因为行内容默认是会溢出的。

设定位置（selector{position:relative or absolute or fixed}）的元素是原子显示的，这意味着它的静态后代元素、行内内容和背景之间不可能出现外部元素。

#### 缩进与外凸

先将元素拉伸到容器的各边，然后相对容器各边偏移。

margin:+value 缩进效果 （使各边向中心移动）

margin:-value 外凸效果 （使它们远离中心）

静态行内对齐

position:static; text-align: left; 左对齐

position:static; text-align:center; 居中对齐

position:static; text-align:right; 右对齐

position:static; text-align:justify; 两端对齐

将line-height设置大于内容的高度值，可以使行内内容在垂直方向实现居中对齐。

使用vertical-align:value 可以使行内内容实现垂直对齐。只有在同一行的内容出现不同的高度功设置不同的垂直对齐方式时，垂直对齐效果才会体现。

静态块级对齐与偏移

相对于上级容器对齐和偏移的块级元素必须是尺寸型的。也就是说使用width:value or 百分比，否则无法实现静态块级元素对齐

position:static; width:value; margin-left:value; margin-right:auto 左对齐

position:static; width:value; margin-left:auto; margin-right:auto 居中对齐

position:static; width:value; margin-left:auto; margin-rigth:value; 右对齐

静态元素会扩大并填充整个容器的宽度。如果设置了静态元素的宽度，那么它的宽度就不会填充整个容器。相反，它的外边距会扩大并填充整个容器。使用auto值可以控制外边距的扩大范围。使用margin-left:auto会自动扩大左外边距，使元素实现右对齐。相反，使用margin-right:auto会自动扩大右外边距，使元素实现左对齐。同时使用margin-left:auto and margin-right:auto会自动扩大左右两边，使元素居中对齐。

### 文字样式

font-family:

font-size

em or 百分比，字号设置为父元素字号的相对值

内置常量如xx-small, x-small, small, medium large, x-large, xx-large

像素值

color, background-color

font-style

font-weight

text-transform

text-decoration 设置文字的下划线(underline)，上划线(overline)和删除线(line-through)

text-shadow: COLOR X-OFFSET Y-OFFSET DIFFUSION

由于font-size是继承属性，所以在<body>上设置font-size:small，然后使用百分数或em,就可以根据需要缩放font-size

letter-spacing 设置字母间隔

word-spacing 设置单词间隔

line-height 设置行间距

text-indent 设置终止块级元素的首行缩进

text-align: justify 使文字两端对齐，这样会增大单词间隔

如果内容本身包含空格，则可以使用<pre>标签或者设置css属性保留空白字符。它将会保留空白字符，并将它们视为内容的组成部分

<pre> CONTENT </pre>

SELECTOR {white-space: pre;}

水平对齐内容： text-align: left/right/center/justify

垂直对齐内容：vertical-align:text-top/middle/baseline/text-bottom将行内元素对齐到父元素的4条字体线

### 块级元素

用块级元素确定文档结构

Div或表单中的内容比一般内容的关系要紧密，列表内容之间的关系则更为紧密。而表格内容则关系最紧密

**HTML有两种结构：层次结构和集合结构。使用嵌套元素可以创建层次结构**。将多个元素置于其父元素之内，就形成集合结构。集合结构有两种：有序型和无序型。

表格可用于创建包含行与单元格的层次结构。

列表以层次结构开始，父列表元素包含一组列表项目。

字典列表是一个关联实体，它包含一组关联的无序词汇与定义。

**为了创建统一的外观，我们通常会给同一种类型的块级元素设置标准样式集**。有时候我们需要修改两个特定块级元素之间的间隔。拉近它们的距离，可以突出它们关系的紧密性，而拉远它们的距离，则可以突出它们的差异性。

HTML的section元素可用于划分文档的节。节是一般概念，本身没有具体的含义。节属于文档，包含与具体主题下头或具有特定用途的内容。节通常包含标题，然后是逻辑上相互关联的附表块。节通常嵌套一些子节，用于表示与节主题相关的下级主题。规定每个节的第一个包含内容的子元素必须是标题

<section class=”introduction”>

<h2>Introduction</h2>

<p>This paragraph is about the introduction.</p>

</section>

<section class=”content”>

<h2>Content</h2>

<p>This paragraph is about the content.</p>

<section class=”section example”>

<h3>Subsection Example</h3>

<ul>

<li>This list item relates to the subsection example.</li>

<li>This list item relates to the subsection example.</li>

</ul>

</section>

</section>

### 列表

使用list-style-type属性，可以设置列表项目左边显示的项目符号类型。项目符号包括disc(默认)，circle and square. 使用list-style-type:none可以隐藏项目符号

使用list-style-image,可以在项目符号位置显示图片

使用list-style-position:inside, 可以将项目符号置于列表外边距之内，从而允许后续行显示在项目符号之下

使用display:list-item, 可以将任意块级元素或行内元素显示为列表项目，而浏览器会在其左外边距显示项目符号。

所有主流浏览器都会为列表设置40像素的缩进距离。

行内化

使用CSS规则display:inline可以实现这种效果将块级元素行内化，它就能够从左到右排列，然后在行末换到下一行。

如果父块级元素行内化，那么它的子块级元素也会行内化，否则它们会破坏行内格式化上下文，然后创建新的块级格式化上下文。

合并外边距

**浏览器会将垂直外边距并入相邻块级元素之间较大的上下外边距之中。**

设置父元素的上内边距或上边框，可以防止第一个子元素的上外边距与父元素的上外边距合并。同样，设置父元素的下内边距或下边框，可以防止最后一个子元素的下外边距与父元素的下外边距合并。

相邻兄弟块级元素之间的垂直外边距不能够拆分。如果想要避免相邻元素出现合并效果，可以将外边距设置为0，并且使用透明边框或透明内边距。边框与内边距不会合并。

### 图片

**为什么需要给图片设置width and height?**因为浏览器会单独下载每一张图片，所以它需要知道图片的高度与宽度，这样它才可以在图片下载过程中预留图片显示空间。否则，**只有当各个图片完成下载时，其真实尺寸才明确，这样，浏览器就必须重新调整页面布局，从而影响渲染速度，并且干扰用户浏览。**

图片是行内元素。它会垂直对齐到它所在行的基线。使用vertical-align,可以调整它的垂直对齐方式如果想要将图片变成块级元素，那么应该使用display:block,将图片显示为块级元素。

HTTP和TCP/IP通信协议需要在内容下载之前传输一些握手消息，消息在互联网中传输会占用带宽，而且服务器处理请求之前还会先将它们进行排队。

**浏览器会同时使用3个并发连接进行下载**。

### 流动布局

如何按行与列排列内容，并且随视口宽度变化动态扩大或缩小？

当视口宽度太窄时，如何使列自动转换为行？

如果将列宽设置为随视口的宽度而自动扩大，但是又要控制其宽度，避免出现不具可读性的超宽列？

<div id=”nav”>…</div>

<div id=”main”>…</div>

<div id=”news”>…</div>

#nav { float: left; width:20%; min-width: 170px; }

#main { float:left; width:40%; min-width:170px; }

#news { float:left; width:20%; min-width:170px; }

所有div都设置为浮动到左边，从而显示为列，但是如果视口宽度不足并排显示它们时，浏览器会自动将一个或多个div移动下一行。每一个div都设置了最小宽度，因此它们不会过度缩小而影响可读性。每一个div都设置了百分比宽度，使它们的宽度与一定比例视口宽度相对应

#### 由外而内框

**因为CSS不存在外部宽度属性。width属性是指元素的内部宽度，而外边距、边框和内边距会扩大外部宽度.**那么如何将浮动元素、绝对元素或静态元素的外部宽度设置为指定尺寸或百分比？

不要设置元素的外边距、边框和内边距，而是将它们添加到一个嵌入的元素中。因为外部元素没有外边距、边框和内边距，所以的外部宽度即是内部宽度。从而，使用width就可以设置外部宽度嵌入的元素就可以称为由外而内框。因为它会将外边距、边框和内边距从外部框移动内部。

块级元素或显示为块级元素的行内元素可以创建很好的由外而内框，因为浏览器会自动拉伸其宽度。

在创建布局时，通常需要将子元素的外部宽度设置为其父元素宽度的百分比。例如，将一个容器中两个浮动元素都设置为容器宽度的50%。如果直接在这些浮动元素上设置外边距、边框或内边距，那么它们的外部宽度会扩大并超过50%.在嵌入的由外而内框上设置外边距、边框和内边距，就可以解决这个问题

这个模式很适合用使用百分比设置流动布局的元素排列，因为补偿固定外边距、边框和内边距所需要设置的width百分比值是无法提前确定

<div class=”before float”>

Float

</div>

=>

<div class=”after float”>

<div class=”oi”>Float</div>

</div>

.before { width:100%; margin:5px; padding:5px; border:5px solid black;}

.after { width:100%;}

.after .oi {margin:5px; padding:5px; border:5px solid black;}

**流动布局设计总是由外部到内部。这是因为，我们首先是从视口宽度开始，然后使用百分比、最小宽度和最大宽度将它划分到各个元素。**

这里的问题是，width属性只负责设置元素的内部宽度。内边距、边框和外边距包围元素内部宽度，因此会增大其外部宽度。因为CSS没有外部宽度属性，所以CSS必须采用从内部到外部的设计方法。其结果是外边距、边框和内边距可能会破坏流动布局设计。

按照CSS标准，浏览器会忽略边框所设置的百分比，这意味着必须设置固定尺寸的边框。此外，在外边距和内边距上设置百分比不是常用做法，因为在视口大小变化时，保持外边距和内边距不变，显示效果会更好一些。

每一个浮动元素上嵌入一个由外而内框，设置这个框的外边距、边框、内边距和背景，而不是设置浮动元素的样式。这样就能够轻松且可靠地将浮动元素设置为上级容器尺寸的一定比例，在节上设置min-width,可以防止它的宽度过小。在节上设置max-width,可以防止它的宽度过大。在由外而内框上设置min-height,能够保证内容较少的浮动元素也拥有与其他元素相同的最小高度

<div id=”SECTION\_ID” class=”section”>

<div class=”oi”>

<h2>heading</h2>

<p>content</p>

</div>

</div>

#SECTION\_ID{

float: left;

width: percent;

min-width: value;

max-width: value;

}

#SECTION\_ID .oi {

min-height: value;

margin: value;

border: width style color;

padding:value;

background: styles;

}

浮动元素没有设置外边距、边框、内边距或背景。我们所看到的是每一个浮动元素之内的由外而内框的边框和背景。由外而内框设置了80像素的左外边距，它会在浮动元素之间创建间隔假象，而这些间隔实际上位于第2个浮动元素之内。

#### 浮动分隔区

在水平线规则设计模式上增加clear:both,就可以得到一个空div,它设置了width, height and margin,用于控制它插入的间隔宽度。使用font-size:1px and line-height:1px设置高度，使用border and background可以设置分隔区的线条

<div class=”float-divider”></div>

.float-divider {

clear:both;

font-size: 1px;

line-height:1px;

}

#### 两侧浮动

在一个元素上设置float:left,在另一个同级元素上设置float:right.这样，两个元素会分别移到其父元素的左右两边。这个模式只适用于成对且相邻兄弟元素

在浮动元素之后添加一个浮动分隔区，就可以保证浮动元素之间不会再出现其他内容，并且可以保证父元素会垂直扩大，有足够的空间显示这两个两侧浮动元素。

在每一个浮动元素上设置min-width and max-width,设置它的最小宽度和最大宽度。在左浮动元素上设置margin-left,在右浮动元素上设置margin-right,就可以调整它们的位置

<div id=”SECTION\_ID”>

<ELEMENT id=”ID1”> any content </ELEMENT>

<ELEMENT id=”ID2”> any content </ELEMENT>

<div class=”float-divider”></div>

</div>

#ID1 { float:left; min-width:value; max-width:value}

#ID2 { float:right; min-width:value; max-width:value}

.float-divider { clear:both; display:block; font-size:1px; line-height:1px;}

#### 文字多列布局

<div class=”multi”>

<p>Lorem is ……</p>

</div>

.multi {

column-count: 3 ;

-moz-column-count: 3;

-webkit-column-count: 3;

-ms-column-count:3;

}

## <Scalable and Modular Architecture for CSS>

by Jonathan Snook

Techniques that can keep CSS more organized and more structured, leading to code that is easier to build and easier to maintain.

SMACSS approach has two core goals:

* Increase the semantic value of a section of HTML and content
* Decrease the expectation of a specific HTML structure

### Categorizing CSS Rules

By categorizing CSS rules, we begin to see patterns and can define better practices around each of these patterns. Repetition results in less code, easier maintenance, and greater consistency in the user experience.

Base Rules the defaults

Layout Rules divide the page into sections

Module reusable, modular parts of our design

State Rules describe how our modules or layouts will look when in a particular state

Theme Rules

How to name?

l- or grid- Layout

is- State

.module- Modules

#### Base Rules

Applied to an element using an **element selector, a descendent selector, or a child selector, along with any pseudo-classes**. It doesn’t include any class or ID selectors. It is defining the default styling for how that element show look in all occurrences on the page.

I highly recommended that you specify a body background

CSS Resets: a set of Base styles designed to strip out or **reset the default margin, padding, and other properties**. Its purpose is to define a consistent foundation across browsers to build the site on. Developing your own set of default styles that you consistently use from project to project can also be advantageous.

#### Layout Rules

I refer to the **minor components as Modules**, and **major components as Layout styles**

Layout styles can also be divided into major and minor styles based on reuse. Major Layout styles such as header and footer are traditionally styled using ID selectors but take the time to think about the elements that are common across all components of the page and use class selectors where appropriate.

Generally, a Layout style only has a single selector: **a single ID or class name.**

#### Module Rules

A Module is a more discrete component of the page. **Modules sit inside Layout components Modules can sometimes sit within other Modules,** too. Each Module should be designed to exist as a standalone component.

When defining the rule set for a module, avoid using IDs and element selectors, sticking only to **class names**. A module will likely contain a number of elements and there is likely to be a desire to use descendent or child selectors to target those elements.

Only include a selector that includes semantics. A span or div holds none.

By adding the classes to the elements, we have increased the semantics of what those elements mean and removed any ambiguity when it comes to styling them.

Elements with greater semantics like heading are more likely to appear by themselves within a container and you are more likely able to use an element selector successfully.

**To help battle against specificity, then you can double up on your class names like in the next example.**

Subclassing

.pod.pod-callout {}

<div class=”pod pod-callout”>…</div>

Specific layout components assigned with IDs could be used to provide specialized styling for modules, sub-classing the module will allow the module to be moved to other sections of the site more easily and you will avoid increasing the specificity unnecessarily.

#### State Rules

States are generally applied to the same element as a layout rules or applied to the same element as a base module class.

<div id=”header” class=”**is-collapsed**”>

<form>

<div class=”msg **is-error**”>There is an error!</div>

<label for=”searchbox” class=”**is-hidden**”>Search</lable>

<input type=”search” id=”searchbox”>

</form>

</div>

The header element just has an ID. As such we can assume that any styles, if there are any, on this element are for layout purposes and that the is-collapsed represents a collapsed state.

In this case, we are actually applying the state to a base element and not overriding a layout or module.

What is the difference between sub-module style and state style?

* State styles can apply to layout and module styles;
* State styles indicate a JavaScript dependency.

Sub-module styles are applied to an element at render time and then are never changed again. State styles, however, are applied to elements to indicate a change in state while the page is still running on the client machine.

States should be made to stand alone and are usually build of a single class selector. Since the state will likely need to override the style of a more complex rule set, the use of !important is allowed. Remember, the use of !important should be avoided for all other rule types. Only states should have it. You won’t normally have two states applied to the same module or two states that tend to affect the same set of styles, so specificity conflicts from using !important should be few and far between.

In a case where a state rule is made for a specific module, **the state class name should include the module name in it**. The state rule should also reside with the module rules and not with the rest of the global state rules.

.tab {

background-color: purple;

color: white;

}

.is-tab-active {

background-color: white;

color:black;

}

If you are doing just-in-time loading of your CSS, generic states should be considered part of the base and global styles and loaded on initial page load. The styles for a particular module won’t need to be loaded until that particular module is loaded.

#### Theme Rules

Allow users to customize the color based on a user preference.

Theme can affect any of the primary types. They can override base styles like default link colors. They can change module elements such as color and border. They can affect layout with different arrangements. They can also alter how states look.

For more extensive theming, using a theme- prefix for specific theme components will make it easier to apply them to more elements on the page.

.theme-border {

border-color: purple.

}

.theme-background {

background: linear-gradient(…);

}

Typography: font rules will normally affect base, module and state styles. Font styles won’t normally be specified at the layout level as layouts are intended for positioning and placement. Your site should only have 3 to 6 different font-sizes.

### Changing State

#### Change via Class name

A class name change happens with JavaScript

$(‘.btn-close’).click(function(){

$(this).parents(‘.dialog’).addClass(‘is-hidden’) ;

})

Sub-module naming convention

.btn {color: #333;}

.btn-pressed {color: #000;}

.btn-disabled {opacity: .5; pointer-events:none;}

State naming convention

.btn {color:#333;}

.is-pressed {color: #000;}

.is-disabled {opacity: .5; pointer-events:none;}

A common interface design pattern is that of a button being pressed and displaying a menu. Generally we had **used a naming convention to hook the two together**. The state change for a single item is modified on two different items in two different places via JavaScript.

Button and menu in separate parts of the same document

<div id=”content”><div class=”toolbar”>

<button id=”btn-new” class=”btn” data-action=”menu”>New</button>

</div></div>

<div id=”menu-new” class=”menu”>

<ul>…</ul>

</div>

Loading Menu with jQuery

$(‘btn-new’).click(function(){

var el = $(this) ;

el.addClass(‘is-pressed’) ;

$(‘#menu-‘ + el.id.substr(4)).removeClass(‘is-hidden’) ;

}) ;

Button and menu in the same part of the document

<div id=”content”>

<div class=”toolbar”>

<button id=”btn-new” class=”btn” data-action=”menu”>New</button>

<div id=”menu-new” class=menu”>

<ul>…</ul>

</div>

</div>

</div>

The problem with the above approach is that this HTML structure is now tied together. The improved way is that applying a state to each button is the preferred approach.

#### Handling State Change with Attribute Selectors

.btn[data-state=default] {color: #333;}

.btn[data-state=pressed] {color: #000;}

.btn[data-date=disabled] {opacity: .5; pointer-events:none}

<button class=”btn” data-state=”disabled”>Disabled</button>

Changing State with jQuery

$(‘.btn’).bind(‘click”, function(){

$(this).attr(‘data-state’, ‘pressed’) ;

}) ;

The data-prefix on the attribute is part of the HTML5 specification that allows you to make up attribute names and place them within the data namespace so as not to conflict with future HTML attriute specifications. Changing the state of a button doesn’t require removing classes and adding classes. It just requires changing the value of a single attribute.

#### Change via Pseudo-class

A pseudo-class change is done via any number of pseudo-classes. Pseudo-classes are still limited in that we can only style changes to elements that are descendants or siblings of the element in which the pseudo-class applies.

The default state for a module is normally defined without a pseudo-class. Define the pseudo-classes as a secondary state of the module.

.btn { background-color: #333;}

.btn:hover {background-color: #336;}

.btn:focus {box-shadow: 0 0 3px rgba(48, 48, 96, .3) ;}

Modules, sub-modules, class states and pseudo-class states.

.btn {…}

.btn:hover {…}

.btn:focus {…}

.btn-default {…}

.btn-default:hover {…}

.btn.is-pressed {…}

.btn.is-pressed:hover {…}

.btn-default.is-pressed {…}

.btn-default.is-pressed:hover {…}

#### Media query

Media queries describe how things should by styled under defined criteria, such as different viewport sizes.

Adaptive design and responsive web design use media queries to react to various criteria.

Most examples of media queries define a break point and them throw all styles that pertain to that particular break point and place them indise.

Keeping the module information together (especially in its own CSS file) allows for isolated testing of the module and (depending on how you build your application) allows assets suc as modularized templates and CSS to be loaded after the initial page has been loaded.

### Depth of Applicability

#### Minimize the depth

HTML is like a tree structure of parents and children. The depth of applicability is the number of generations that are afftected by a given rule.

An advantage to using this shallow depth of applicability approach is also the ability to more readily convert these modules into templates for dynamic content.

<div class=”pod”>

<h3>{{heading”}}</h3>

<ul class=”pod-body”>

{{#items}}

<li>{{item}}</li>

</ul>

</div>

#### Selector performance

Google page speed which provides a number of recommendations to improve JavaScript and rendering perfonmance.

Browsers are designed to handle documents like a stream. They begin to receive the document from the server and can render the document before it has completely downloaded. Each node is evaluated and rendered to the viewport as it is received.

Strike a balance between one end of the spectrum (adding classes and identifiers to everything) and the other (using deep selector rules creating tight coupling between HTML and CSS).

I follow three simple guidelines to help limit the number of elements that need to be evaluated:

* Use child selectors
* Avoid tag selectors for common elements
* Use class names as the right-most selector

Class names help describe our content in very specific ways-ways that are more specific than even HTML5 can provide.

Remember two goals: increase semantics and decrease reliance on specific HTML

<nav class=”l-inline”>

<h1>Primary Navigation</h1>

<ul>

<li>About Us

<ul class=”l-stacked”>

<li>Team</li>

<li>Location</li>

</ul>

</li>

</ul>

</nav>

Patterns establish familiarity and encourage re-use.

The web design industry likes reusable components and can be seen in many of the frameworks like Bootstrap and 960.gs

### CSS Preprocessors

Allows you to use a special syntax in your CSS that is then compiled within your project. The two preprocessors that currently lead the market are Sass (or similarly, Compass) and Less

>gem install sass

>gem install compass

>sass –watch before:after //set sass to watch a folder that I am working on

Where before is the folder where all my .scss files are contained and after is where the post-compiled CSS files are places.

Less uses npm, the node package manager, which isn’t pre-installed.

Useful features of a preprocessor

#### Variables

$colr: #369;

body { color: $color; }

#### Operations

#### Mixins

A mixin is a group of styles that can be re-used throughout your CSS. They can take parameters that customize the output of the mixin.

@mixin border-radius($size){

-webkit-border-radius: $size;

-moz-border-radius: $size;

Border-radius: $size;

}

.callout {

@include border-radius(5px) ;

}

#### Nesting

.nav {

>li {

float: left;

>a {

display: block;

}

}

}

#### Functions

The mixins example already looks like al function. There is, however, the ability to do some powerful stuff with calculating values.

Sass comes with a number of handy functions like this and Compass adds even more.

.btn { background-color: $btnColor;}

.btn:hover { background-color: lighten($btnColor, 20%); }

#### Interpolation

#### File importing

#### Extending

.btn {

display: block ;

padding: 5px 10px;

background-color: #003366;

}

.btn-default {

@extend .btn;

background-color: #0066cc;

}

The extension takes the styles from btn and applies them to btn-default.

The extensions are limited to simple selectors. You could not extend #main .btn

#### State-based Media Queries with Nesting

Sass allows media queries to be nested, allowing those state changes to be reflected wherethey belong: with the module.

.nav > li {

width: 100%;

@media screen and (min-width: 320px){

width: 100px;

float: left;

}

@media screen and (min-width: 1200px) {

Width:250px;

}

}

#### Organizing your files

* Preprocessor encourage the separation of concerns that SMACSS recommends
* Place all base rules into their own file.
* Depending on the type of layouts you have, either place all of them into a single file or major layouts into separate files.
* Put each module into its own file.
* Depending on size of project, place sub-modules into their own file.
* Place global states into their own file.
* Place layout and module states, including media queries that affect those layouts and modules, into the module files

#### 反模式

##### Deep Nesting

#sidebar {

width: 300px;

.box {

border-radius: 10px;

background-color: #EEE;

h2 {

font-size: 14px;

font-weight: bold;

}

ul {

margin: 0;

padding: 0;

a { display: block;}

}

}

}

解决方式: minimizes deep nesting via separation of layout and modules also avoids these issues.

#sidebar { width: 300px;}

.box {

border-radius: 10px;

background-color: #EEE;

}

.box-header {

font-size: 14px;

font-weight: bold ;

}

.box-body {

margin: 0;

padding: 0;

a {display: block;}

}

##### Unnecessary extending

.btn {

display: block ;

padding: 5px 10px;

background-color: #003366;

}

.btn-default {

@extend .btn;

background-color: #0066cc;

}

Using the SMACSS approach, extensions are just handled at the HTML level instead of the CSS level by defining the multiple classes in the HTML

<a class=”btn btn-default”>My button</a>

##### Overused Mixins

@mixin them-border {

border: 2px solid #039;

border-radius: 10px;

}

.callout { @include theme-border;}

.summary { @include theme-border; }

Defining a class for common pattern

.them-border{…}

.callout {}

.summary {}

<div class=”callout theme-border”></div>

<div class=”summary theme-border”></div>

# JS

## Best Practice

### 函数式编程

var lists = [‘r’, ‘g’, ‘b’].map( function(item) {

return item.toUpperCase();

});

var specialList = listings.filter( function(item){

return item.price >= min && item.price <= max ;

});

[‘r’, ‘g’, ‘b’].forEach( function(item) {

console.log(item);

});

### 类

function Shape(){

this.name = ‘Shape’;

}

Shape.prototype.toString = function() {

return this.name;

}

Duck type

使用结构类型来实现多态性，从而避免使用继承

编译类型语言使用类和继承来结构化程序。

动态语言可以使用结构类型（或叫鸭子类型）来设计对象的接口，任何对象只要具有预期的结构就属于该类型，从而它不需要编写任何显示的声明，一个调用某个对象方法的函数能够与任何实现了相同接口的对象一起工作。

function Wiki(format){

this.format = format ;

}

Wiki.prototype.displayPage = function(source){

var page = this.format(source) ;

var title = page.getTitle() ;

var author = page.getAuthor() ;

var output = page.toHTML() ;

//...

};

//不需要使用子类化继承，只要类实现相同的接口

Wiki.format.MEDIAWIKI = function(source){

// extract contents from source

//...

return {

getTitle: function() {} ,

getAuthor: function() {},

toHTML: function() {}

};

} ;

var app = new Wiki(Wiki.formats.MEDIAWIKI) ;

选项对象

function Alert(parent, message, opts){

opts = opts || {} ;

//假定0是有效值，所以要测试undefined

this.width = opts.width===undefined ? 320 : opts.width ;

this.height = opts.height===undefined ? 240 : opts.height ;

//假定空字符串是无效值，用逻辑或

this.title = opts.title || "Alert" ;

//使用双重否定模式将参数转换为布尔值

this.modal = !! opts.modal ;

}

基本概念

所有javascript数字均为64位

var txt = "hello world"; 字符串

txt1+txt2, .length, .split(), .substring(), .replace()

var now = new Date(); 日期和时间

now.getHours(), now.getMinutes(), now.getSeconds()

var arr = [] ; 数组

arr1+arr2, .push(), .join('/'), .slice(0, 2), .sort(func), .reverse()

假定data是二维数组

for (let i = 0 ; i < data.length; ++ i) {

for (let j = 0 ; j < data[0].length; ++ j ) {

if (data[i][j] < min) min = data[i][j];

if (data[i][j] > max) max = data[i][j];

}

}

var o = {} ; 对象

var a=new Image(); Image对象, 每遇到<img/>，自动实例化Image对象

img.onload = function() { fun(img); };

img.onerror = function(){ alert("图片加载错误时,发生调用"); };

img.src = "http://www.005.tv/uploads/allimg/161009/1523215548-0.jpg";

<canvas id="myCanvas" width="578" height="400"></canvas>

var canvas = document.getElementById('myCanvas');

var ctx = canvas.getContext('2d');

var imgData=ctx.createImageData(512,512); ImageData对象

for (let nRow = 0 ; nRow < imgData.height; ++ nRow) {

for (let nCol = 0 ; nCol < imgData.width; ++ nCol) {

let idx = nRow\*imgData.width + nCol;

imgData.data[idx\*4 + 0] = 127; // red

imgData.data[idx\*4 + 1] = 127; // green

imgData.data[idx\*4 + 2] = 127; // blue

imgData.data[idx\*4 + 3] = 255; // alpha

}

}

ctx.putImageData(imgData,10,10);

## <JavaScript 面向对象编程指南>

**函数也是数据**？(该数据类型包含的是代码，且可执行)

函数可以被赋给变量，从而被定义，删除，拷贝，或作为参数传给另一函数（作为回调函数）

var callback = function(a, b) {return a+b;}

function fun(a, b, c, callback) {

for (var i=0, ar=[]; i<3; ++i){

ar[i] = callback(arguments[i], 2);

}

return ar;

}

>>>callback （返回函数代码）

>>>delete callback （删除变量，存储函数代码片断）

>>>typeof callback （返回函数类型）

>>>callback(a,b) （执行函数）

浏览器特性探测技术(函数根据其当前所在的浏览器来重定义自己)

var callback = function() {

// 私有函数，用来初始化

function someSetup(){

}

//私有函数

function actualWork(){

}

someSetup() ;

return actualWork ;

}() ;

()表示自动执行，私有函数someSetup()执行初始化工作，然后返回actualWork函数赋给变量callback

Javascript没有{}作用域，有函数作用域，每个函数有自己的作用域链和词法作用域

**函数作用域链**:子函数可以访问自己的作用域，还可以访问父级作用域，还可以访问父级的父有作用域

function f() {

var b = 1 ;

function n() {

var c = 3 \* b ;

}

}

**函数词法作用域**：每个函数在被定义时（而非执行时）都会创建一个属于自己的环境

闭包1:返回子函数的父函数作用域中的变量或子函数

function f(){

var b = “b”;

return function() {

return b;

}

}

闭包2: 子函数赋给全局函数

var n;

function f() {

var b = “b” ;

n = function() {

return b ;

}

}

闭名3: **闭包只是引用局部变量（即拥有的是局部变量的连接，但不会记录局部变量的值**）

function f() {

var a = [] ;

var i ;

for (i = 0 ; i < 3; i ++ ){

a[i] = function() {

return i ;

}

}

}

>>> var a = f() ;

创建三个闭包a[0](), a[1](), a[2](),它们都指向了一个共同的局部变量i, 但是闭包并不会记录它们的值，它们所拥有的只有一个i的连接（即引用），因此只能返回i的当前值。由于循环结束时i的值的为3,所以这三个函数都指向了这一共同值.

闭包的简单应用：

实现getter and setter: 变量在自调函数空间里，子函数赋给全局变量

var getValue, setValue;

(function(){

var secret = 0 ;

getValue = function() {

return secret ;

} ;

setValue = function(v) {

secret = v ;

} ;

})()

实现迭代器:

function setup(x){

var i = 0 ;

return function() {

return x[i++] ;

}

}

>>> var next = setup ([‘a’, ‘b’]) ;

>>>next()

**对象**

数组文本标识法: var a = [] ;

对象文本标识法: var o = {} ;

JavaScript是动态语言，所以它允许我们随时对现存对象的属性和方法进行修改。

* 创建对象的方式：

**通过对象文本标识法直接创建对象: (实际上是由内建构造器Object()函数创建)**

var o = {

name: ‘Rafaelo’,

sayName: function() { … }

} ;

通过构造器函数创建对象:

//定义构造器函数

function Hero(name){

this.name = name ;

this.sayName = function(){…} ;

}

//创建对象

var o = new Hero(‘Rafaelo’) ;

var o2 = new Hero(‘qzlin’) ;

重点：

* 一般将构造器函数首字母大写，以区别一般函数
* 构造器函数作为普通函数直接调用，那么this指的是全局对象window; 利用构造器函数创建对象，那么this指的是这个对象。
* **对象拥有构造器属性**

o.constructor == Hero(name) ; //从而可以通过对象构造器属性创建另一对象

var o3 = new o.constructor(‘tom’) ;

* 工厂函数返回对象:

function factory(name) {

return {

name: name ;

}

}

* 内建对象：

数据封装类对象（Object, Array, Boolean, Number, String）

工具类对象(Math, Date, RegExp)

错误类对象（Error）

* 所有对象都继承自Object对象，并且拥有其所有的方法（如toString(),constructor）
* 函数也是对象，内建构造器是Function(),拥有如下属性:

1. toString() ; //返回函数源码
2. constructor ;
3. length ; //函数的输入参数个数
4. prototype ; //是一个对象，仅和构造器函数配合才有意义,
5. //因为JavaScript没有类概念，采用call or apply复用对象方法？
6. call() ;
7. apply() ;

//某对象

var my\_obj = { name: ‘current object’} ;

var other\_obj = {

name: ‘Ninja’,

say: function(who, role) {

return ‘I am ‘ + who + ‘ with ‘ + role;

}

}

//因为my\_obj没有方法say,不想自己实现，可以复用other\_obj对象的方法say(注意借用的方法say(who)的作用域会改为my\_obj对象)来执行

other\_obj.say.call(my\_obj, ‘Dude’, ‘Doctor’) ;

等同于

other\_obj.say.apply(my\_obj, [‘Dude’, ‘Doctor’]) ;

1. 函数内有arguments变量，也是一个对象，含属性[i], length, callee, 基于callee属性可以实现递归函数

function(count){

if (count <５)　｛

alert(count) ;

arguments.callee(++count) ;

｝

}

**原型**

构造器函数 + 原型

//构造器函数，内赋属性和方法

function Gadget(name, color) {

this.name = name ;

this.color = color ;

}

//原型，添加方法

Gadget.prototype.getInfo = function() {

return …;

}

>>> var newtoy = new Gadget(‘webcam’, ‘black’) ;

>>>newtoy.price ;

分析：对象访问属性或方法时，JavaScript引擎会先遍历该对象的所有属性，如果没找到，JavaScript会查询创建当前对象的构造器函数的原型，如果仍没找到，就会继续搜索其原型的原型，直至最高级的父对象Object.

若重写某对象的prototype时，一定要重置相应的constructor属性. 否则已创建对象无法访问新的prototype，只能新创建的对象能访问。

Gadget.prototype = {

Price: 100 ,

getInfo = function() {…}

}

Gadget.prototype.constructor = Gadget ;

**继承**

* 通过创建父对象，赋给子对象构造器原型 (构造器工作模式—使用原型链模式)

function Shape(){

this.name = ‘shape’ ;

}

Shape.prototype = {

toString: function() { return this.name ; }

};

function Triangle(side, height) {

this.name = ‘Triangle ;

this.side = side ;

this.height = height ;

}

**Triangle.prototype = new Shape() ;**

Triangle.prototype.getArea = function() {..} ;

鸭子类型(duck typing)，在这种风格中，一个对象有效的语义，不是由继承自特定的类或实现特定的接口，而是由当前方法和属性的集合决定的，在鸭子类型中，关注的不是对象的类型本身，而是它是如何使用的。通俗来说**一个对象如果拥有某个对象的方法和属性，就可以看作是这个对象。**

组合继承：

function SuperType(){

this.property=true;

this.colors=['red','blue','green'];

}

SuperType.prototype.getSuperValue=function(){

return this.property;

};

//子类

function SubType(){

this.subProperty=false;

}

SubType.prototype=new SuperType();

SubType.prototype.getSubValue=function(){

return this.subProperty;

}

var sub1=new SubType();

sub1.colors.push('yellow');

var sub2=new SubType();

console.log(sub2.colors);//["red", "blue", "green", "yellow"]

缺点：当父类型中存在引用类型的属性的时候，子类型所有实例都共享这个属性

改进1：子类构造函数调用父类的构造函数

//子类

function SubType(){

this.subProperty=false;

SuperType.call(this); 第一次调用父类的构造函数

}

SubType.prototype=new SuperType(); 第二次调用父类的构造函数

SubType.prototype.getSubValue=function(){

return this.subProperty;

}

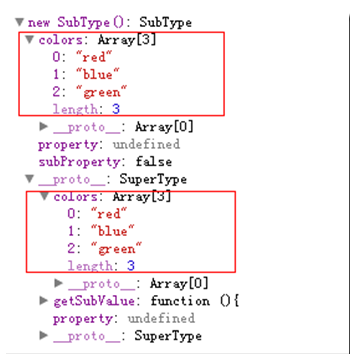
var sub1=new SubType();

sub1.colors.push('yellow');

var sub2=new SubType();

console.log(sub2.colors);//["red", "blue", "green"]

缺点：父类的实例属性和方法在子类的实例中会有两份



问题的根源就是调用两次SuperType的构造函数，其实在第一次调用的时候SubType就已经获得了SuperType的实例属性和方法，第二次调用的时候仅需要SuperType的prototype属性就可以了

替换SubType.prototype=new SuperType(); 第二次调用父类的构造函数

function extend(subType, superType) {

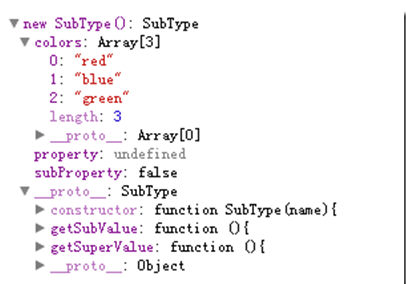
var prototype = new Object(superType.prototype); //得到父类prototype对象副本

prototype.constructor = subType; //contructor属性改为子类自己的

subType.prototype = prototype; //重写子类prototype

}

extend(SubType, SuperType);



* 通过父对象构造器函数的原型，赋给子对象构造器函数的原型 （构造器工作模式—原型拷贝模式, YUI, Extjs采用此方法）

**function extend(Child, Parent) {**

**//避免子对象修改影响父对象（因为共同引用），需要临时构造器**

**var F = function() {} ;**

**F.prototype = Parent.prototype ;**

**Child.prototype = new F() ;**

**Child.prototype.constructor = Child ;**

**Child.uber = Parent.prototype ;**

**}**

**extend(Triangle, Shape) ;**

* 通过对象拷贝 （对象工作模式--属性拷贝模式, 注意深拷贝=值传递与浅拷贝=引用传递）

**function deepCopy(parent, child){**

**var child = child || {} ;**

**for (var i in parent) {**

**if (typeof parent[i] === ‘object’) {**

**child[i] = (parent[i].constructor === Array) ? [] : {} ;**

**deepCopy(parent[i], child[i]) ;**

**} else {**

**child[i] = parent[i] ;**

**}**

**}**

**return child ;**

**}**

**var triangle = deepCopy(new Shape()) ;**

* 原型继承与属性拷贝的混合

**function objectPlus(o, stuff) {**

**//原型继承copy现存对象**

**function F() {}**

**F.prototype = o ;**

**var n = new F() ;**

**n.uber = o ;**

**//对其他对象stuff使用属性copy**

**for (var i in stuff) {**

**n[i] = stuff[i] ;**

**}**

**return n ;**

**}**

**var triangle = objectPlus(new Shape(), {**

**name: ‘Triangle’,**

**getArea: function() {…}**

**side: 0,**

**height: 0**

**}) ;**

* 复用构造器和属性拷贝 (构造器工作模式—原型模式—属性拷贝模式)

**function copyPrototype(Child, Parent) {**

**var p = Parent.prototype ;**

**var c = Child.prototype ;**

**for (var i in p) {**

**C[i] = p[i] ;**

**}**

**c.uber = p ;**

**}**

**function Triangle() {**

**Shape.apply(this, arguments) ;**

**}**

**copyPrototype(Triangle, Shape) ;**

JavaScript中没有类的概念，一般是*将构造器函数当做类*。所以一般选择构造器工作模式来解决问题。

惯用法

* 创建命名空间

var MYAPP = {} ;

MYAPP.namespace = function(name){

var parts = name.split('.') ;

var current = MYAPP ;

for (var i in parts){

var part = parts[i] ;

if (!current[part]) {

current[part] = {} ;

}

current = current[part] ;

}

};

MYAPP.namespace(‘event’) ;

MYAPP.namespace(‘dom.style’) ;

等价于

var MYAPP = {

event: {} ,

dom: {

style: {}

}

}

* 通过配置对象来替代多个参数，传给函数

MYAPP.dom.Button = function(text, type, color, border, font) {…}

new MYAPP.dom.Button(‘push’, null, ‘white’, null, ‘Arial’) ;



MYAPP.dom.Button = function(text, conf) {

var type = conf.type || ‘submit’ ;

var font = conf.font || ‘Verdana’ ;

}

* 私有属性和方法

在构造器中通过使用局部变量和函数的方式来实现类似的权限控制

MYAPP.namespace('dom') ;

MYAPP.dom.Button = function(text, conf){

//私有变量

var styles = {

font: 'Verdana',

border: '1px solid black',

color: 'black',

background: 'grey'

} ;

//私有函数

function \_setStyles(b) {

for (var i in styles){

b.style[i] = conf[i] || styles[i] ;

}

}

conf = conf || {} ;

var b = document.createElement('input') ;

b.type = conf['type'] || 'submit' ;

b.value = text ;

\_setStyles(b) ;

return b ;

};

* 自执行函数: 一般用于创建和返回对象

如果我们创建对象的过程很复杂，并且需要做一些初始化工作，那么我们就可以把第一部分相关的初始化工作设置为一个自执行函数，然后通过它来返回一个对象—它可以访问初始化部分定义的任何私有属性。

var MYAPP = {} ;

MYAPP.dom = function(){

// initialization code

function \_private(){

...

}

return {

getStyle: function(el, prop){

...

\_private() ;

},

setStyle: function(el, prop, value){

...

}

};

}() ;

链式调用：构造器函数（类）当中的方法返回this

## <Effective JavaScript>

全局变量与局部变量

局部变量：函数内定义的变量如

function() {

var sum = 0 ;

}

全局变量： 函数外定义的或函数内定义，但没有var指示的变量如

function() {

sum = 0 ;

}

**在Web浏览器中，全局对象被绑定到全局的window变量，console中访问this.sum或window.sum**

闭包

* Javascript允许你引用在当前函数以外定义的变量

function makeSandwich() {

var magicIngredient = "peanut butter" ;

function make(filling){

return magicIngredient + " and " + filling ;

}

return make("jelly") ;

}

makeSandwich() ; // "peanut butter and jelly"

* 即使外部函数已经返回，当前函数仍然可以引用在外部函数所定义的变量

function makeSandwich() {

var magicIngredient = "peanut butter" ;

function make(filling){

return magicIngredient + " and " + filling ;

}

return make ;

}

var f = makeSandwich() ;

f("jelly") ; // "peanut butter and jelly"

f("bananas") ; // "peanut butter and bananas"

注意：JavaScript的函数值包含了比调用它们时执行所需要的代码还要多的信息。而且Javascript函数值还在内部存储它们可能引用的定义在其封闭作用域的变量。那些在其所涵盖的作用域内跟踪变量的函数被闭包。

* 闭包可以更新外部变量的值

**注意：闭包存储的是外部变量的引用，而不是它们的值的副本**。

function box() {

var val = undefined ;

return {

set: function(newVal) {val = newVal ; },

get: function() {return val ; }

};

}

var b = box() ;

b.set(98.6) ;

b.get(); // 98.6

变量声明提升间

Javascript没有块级作用域，只有函数作用域词法作用域

Javascript隐式地提升声明部分到封闭函数的顶部，而将赋值留在原地。惯用法是所有的var声明放置在函数的顶部。

function f() {

//...

//...

{

//...

var x = /\*...\*?

//...

}

}

function f() {

var x ;

//...

{

//...

x = /\*...\*/

//...

}

}

Javascript立即函数表达式

**没有块级作用域，但可以用立即调用的函数表达式来创建块级作用域**

function wrapElement(a) {

var result = [] ;

var i = 0 ;

for (i = 0, n = a.length ; i < n ; ++ i){

//在函数内部创建局部变量var j,并令该函数立即执行，等价于在函数体内的块级{}作用域定义变量j

(function(){

var j = i ;

result[i] = function() {return a[j] ; } ;

})() ;

}

return result ;

}

以上代码是等效的常见方法

function wrapElement(a) {

var result = [] ;

var i = 0 ;

for (i = 0, n = a.length ; i < n ; ++ i){

(function(j){

result[i] = function() {return a[j] ; } ;

})(i) ;

}

return result ;

}

函数声明与命名函数表达式区别？

//在程序的最顶层，该函数声明将创建一个名为double的全局函数

function double(x) {return x \* 2 ;}

//该命名函数表达式将函数绑定到变量f,而不是变量double

var f = function double(x) {return x \* 2 ;}

命名函数表达会导致很多问题，所以不值得使用。一般采用如下方式：

function find(tree, key){

if (!tree){

return null ;

}

if (tree.key === key){

return tree.value ;

}

return find(tree.left, key) ||

find(tree.right, key) ;

}

var f = find ;

重点：**官方指定函数声明只能出现在其他函数或程序的最外层。编写可移植的函数的最好方式是始终避免将函数声明置于局部块或子语句中。**

函数调用，方法调用及构造函数调用之间的区别？

* //函数调用

function hello(username){

return "hello " + username ;

}

hello("qzlin") ;

* //方法调用

var obj = {

username: "qzlin"

hello: function(){

return "hello " + this.username ;

}

};

obj.hello() ;

//复用对象方法

var obj2 = {

username: "Boo",

hello: obj.hello

};

//在方法调用中是调用表达式自身来确定this变量的绑定，所以obj2.hello方法中的this是obj2对象

obj2.hello() ;

//一个函数里有this变量，一般该函数作为对象方法或构造函数，如果直接作为函数调用，则会将全局对象作为接收者，即this是全局对象

* //构造函数使用

function User(name, passwordHash){

this.name = name ;

this.passwordHash = passwordHash ;

}

var u = new User("qzlin", "000");

(记住构造函数的主要职责是初始化该新对象)

高阶函数

将函数作参数或返回值的函数。将函数作为参数，通常该函数也称为回调函数，因为高阶函数随时调用它。常用于集合

var names = ["tom", "jerry", "puppy"].map(function(name){

return name.toUpperCase() ;

}) ;

创建可变参数的函数一般提供两个版本

//数组作为参数的版本

function averageOfArray(a){

for (var i= 0, sum= 0, n= a.length; i < n ; ++ i){

sum += a[i] ;

}

return sum / n ;

}

averageOfArray([2, 7, 2, 1]) ;

//可变参数的版本

function average(){

return averageOfArray(arguments) ;

}

复用对象方法

仅仅是方法借用，但方法内的this对象没有绑定，因此需要确定，如下：

var buffer = {

entries: [] ,

add: function(s){

this.entries.push(s) ;

},

join: function(){

this.entries.join() ;

}

}

//仅仅借用对象的方法，但没有绑定对象方法的this变量

["021", "-", "24115296"].forEach(buffer.add) ; //error: entries is undefined

// 集合的forEach函数提供绑定回调函数的this变量

["021", "-", "24115296"].forEach(buffer.add, buffer) ;

buffer.join() ;

//若函数没有提供，由需要自己封装一层函数，然后直接调用对象的方法（此时含对象）

["021", "-", "24115296"].forEach(function(s){

buffer.add(s) ;

}) ;

buffer.join() ;

//ES5标准库提供：函数对象的bind方法，第一个参数是作用域this对象，并产生一个以该对象调用方法的方式

["021", "-", "24115296"].forEach(buffer.add.bind(buffer)) ; //等价于上面的封装版本

buffer.join() ;

高阶函数惯用法：函数curring

实现函数委托 = 显示封装 = bind版本

//具体实现函数

function simpleURL(protocol, domain, path){

return protocol + "://" + domain + "/" + path ;

}

// 显示封装

var urls = paths.map(function(path){

return simpleURL("http", "siteDomain", path) ;

}) ;

//使用函数对象的bind来实现函数委托 （注意第一个参数是作用域，其它的是参数）

var urls = paths.map(simpleURL.bind(null, "http", "siteDomain")) ;

JavaScript类的本质

在许多语言中，每个对象是相关类的实例，该类提供在其所有实例间共享代码。相反,JavaScript并没有类的内置概念，对象是从其他对象中继承而来。

**Javascript中的类本质上是一个构造函数与一个用在该类实例间共享方法的原型对象的结合(构造函数+原型)**

将方法存储在实例对象中将创建该函数的多个副本，因为每个实例对象都有一份副本。因此**最佳实践是：构造函数初始化对象，原型存储方法和类静态变量**

function User(name, passwordHash){

this.name = name ;

this.passwordHash = passwordHash ;

}

//构造函数User的prototype属性用来设置新实例的原型关系

User.prototype.toString = function(){

return "[User " + this.name + "]";

} ;

User.prototype.checkPassword = function(password){

return hash(password) === this.passwordHash ;

};

关键点：

* 文档化构造函数期望使用new操作符调用是很重要的，尤其是在跨大型代码中共享构造函数和该构造函数来自一个共享库时。
* 获取原型对象的方法：Object.getPrototypeOf(u) === User.prototype ; //true

C.prototype用于建立由new C()创建的对象原型。

Object.getPrototypeOf(obj)是ES5中用来获取obj对象的原型对象的标准方法（不要使用obj.\_\_proto\_\_）

Javascript实现类继承的最佳实践

子类构造函数调用父类构造函数

子类原型继承父类原型

//父类Actor

function Actor(scene, x, y){

this.scene = scene ;

this.x = x ;

this.y = y ;

scene.register(this) ;

}

Actor.prototype.moveTo = function(x, y){

this.x = x ;

this.y = y ;

this.scene.draw() ;

};

Actor.prototype.exit = function(){

this.scence.unregister(this) ;

this.scene.draw() ;

};

Actor.prototype.draw = function(){

var image = this.scene.images[this.type] ;

this.scene.content.drawImage(image, this.x, this.y) ;

};

//子类SpaceShip

function SpaceShip(scene, x, y){

**Actor.call(this, scene, x, y) ; //子类构造函数调用父类构造函数**

this.points = 0 ;

}

//Object.create以一个原型对象作为参数，并返回一个继承自该原型对象的新对象

**SpaceShip.prototype = Object.create(Actor.prototype) ; //子类继承父类原型**

SpaceShip.prototype.type = "SpaceShip" ;

SpaceShip.prototype.scorePoint = function(){

this.points ++ ;

};

SpaceShip.prototype.left = function(){

this.moveTo(Math.max(this.x-10, 0), this.y) ;

};

SpaceShip.prototype.right = function(){

var maxWidth = this.scene.width - this.width() ;

this.moveTo(Math.min(this.x+10, maxWidth), this.y) ;

};

//游戏场景

function Scene(context, width, height, images){

this.context = context ;

this.width = width ;

this.height = height ;

this.images = images ;

this.actors = [] ;

}

Scene.prototype.register = function(actor){

this.actors.push(actor) ;

} ;

Scene.prototype.unregister = function(actor){

var i = this.actors.indexOf(actor) ;

if (i >= 0){

this.actors.splice(i, 1) ;

}

} ;

Scene.prototype.draw = function(){

this.context.clearRect(0, 0, this.width, this.height) ;

for (var a = this.actors, i = 0, n = a.length ; i < n; ++ i){

a[i].draw() ;

}

}

Javascript实现存储私有数据

和python一样，没有语法支持信息隐藏，如何实现存储私有数据？

编码规范，给私有属性名前置或后置一个下划线

使用闭包，在构造函数中以变量的方式来存储，并使对象方法引用这些变量

function User(name, passwordHash){

var \_role = "" ;

this.name = name ;

this.getRole = function(){

return \_role ;

};

this.setRole = function(role){

\_role = role ;

}

}

分析：在此构造函数中，this引用的变量或方法，都可以被外部直接引用，而构造函数里的变量（其作用域只在构造函数里）如\_role, name, passwordHash(别忘了构造函数输入参数也是函数里的变量)，外部无法引用，只能通过方法接口来引用。

函数的this变量

每个函数都有一个this变量的隐式绑定，该this变量的绑定值是在调用该函数时确定的（注：this变量是隐式绑定到最近的封闭函数）

function CSVReader(separators){

this.seperators = separators || [","] ;

this.regexp = new RegExp(this.seperators.map(function(sep){

return "\\" + sep[0] ;

}).join("|")) ;

}

CSVReader.prototype.read = function(str){

var lines = str.trim().split(/\n/) ;

return lines.map(function(line){

return line.split(this.regexp) ; //wrong this,因为这里的this是指调用函数map的对象lines

});

}

解决方法：

* 数组的map方法引入回调函数的this绑定（即引入作用域），从而改变回调函数作用域的this对象。

CSVReader.prototype.read = function(str){

var lines = str.trim().split(/\n/) ;

return lines.map(function(line){

return line.split(this.regexp) ;

}, this) ;

};

* 使用词法作用域的变量来改变this的混淆

CSVReader.prototype.read = function(str){

var lines = str.trim().split(/\n/) ;

**var self = this ;**

return lines.map(function(line){

return line.split(**self**.regexp) ;

}) ;

};

* 使用回调函数的bind方法改变回调函数内的this对象

CSVReader.prototype.read = function(str){

var lines = str.trim().split(/\n/) ;

return lines.map(function(line){

return line.split(this.regexp) ;

}.bind(this)) ;

};

关键点：

* 不要重用父类的属性名，因为子类和父类，如果属性名相同，那么该属性存储在实例对象上并命名为一个字符串，从而指向的是同一个属性。
* 如果一个程序库提供的对象的属性没有文档化或者明文标为内部属性，对于使用者来说，最好不要干涉那些属性。
* 避免使用轻率的猴子补丁（monkey-patching）：对象共享原型，因此每一个对象都可以增加、删除或修改原型的属性。Monkey-patching一般用来弥补平台的差距。

JavaScript中的对象特点

对象是JavaScript中最万能的数据结构，取决于不同的环境，对象可以表示一个灵活的键值记录，一个继承了方法的面向对象数据抽象，一个密集或稀疏的数组，或一个散列表。

JavaScript的对象操作总是以继承的方式工作，即使是一个空的对象字面量，也继承了Object.prototype的大量属性

var dict = {} ;

"toString" in dict ;

"valueOf" in dict ;

为防止原型污染，可以使用null原型:

var x = Object.create(null) ;

Object.getPrototypeOf(x) === null ; //true

注意与如下区别:

function C() {}

C.prototype = null ;

var x = new C() ;

Object.getPrototypeOf(x) === null ; // false ;

Object.getPrototypeOf(x) === Object.prototype ; // true ;

实现字典类

function Dict(elements){

this.elements = elements || {} ; //simple Object

this.hasSpecialProto = false ; // has "\_\_proto\_\_"" key ?

this.specialProto = undefined ; // "\_\_proto\_\_" element

}

Dict.prototype.has = function(key){

if (key === "\_\_proto\_\_"){

return this.hasSpecialProto ;

}

// own property only

return {}.hasOwnProperty.call(this.elements, key) ;

};

Dict.prototype.get = function(key){

if (key === "\_\_proto\_\_"){

return this.specialProto ;

}

// own property only

return this.has(key) ? this.elements[key] : undefined ;

};

Dict.prototype.set = function(key, val){

if (key === "\_\_proto\_\_"){

this.hasSpecialProto = true ;

this.specialProto = val ;

} else {

this.elements[key] = val ;

}

};

Dict.prototype.remove = function(key){

if (key === "\_\_proto\_\_"){

this.hasSpecialProto = false ;

this.specialProto = undefined ;

} else {

delete this.elements[key] ;

}

};

var dict = new Dict(

{

alice: 34,

bob: 24,

chris: 62

}) ;

面向函数编程

数组迭代优先使用for循环而不是for…in循环，迭代方法优于循环

* Array.prototype.forEach(callback): 数组遍历

var trimmed = [] ;

input.forEach(function(s){

trimmed.push(s.trim()) ;

}) ;

* Array.prototype.map(callback): 映射 实现y = f(x)

var trimmed = input.map(function(s){

return s.trim() ;

});

* Array.prototype.filter(callback): 过滤

var specialList = listings.filter(function(listing){

return listing.price >= min && listing.price <= max ;

});

在类数组对象上尽量复用通过的数组方法

function highlight(){

[].forEach.call(arguments, function(widget){

widget.setBackground("yellow") ;

});

}

var arrayLike = {0: "a", 1: "b", 2:"c", length: 3} ;

var result = Array.prototype.map.call(arrayLike, function(s){

return s.toUpperCase() ;

});

undefined: JavaScript无法无法提供具体的值

//未赋值的变量的初始值

var x ;

x ; //undefined

//访问对象中不存在的属性

var obj = {} ;

obj.x ; // undefined

//一个函数体结局使用未带参数的return语句或未使用return语句都会产生返回值undefined

function g() {}

g()；

选项对象

function Alert(parent, message, opts){

opts = opts || {} ;

//假定0是有效值，所以要测试undefined

this.width = opts.width===undefined ? 320 : opts.width ;

this.height = opts.height===undefined ? 240 : opts.height ;

//假定空字符串是无效值，用逻辑或

this.title = opts.title || "Alert" ;

//使用双重否定模式将参数转换为布尔值

this.modal = !! opts.modal ;

}

一般Javascript库和框架都会提供一个extend函数，该函数接收一个target对象和一个source对象，并将source的属性复制到target对象中。

function extend(target, source){

if (source){

for (var key in source){

var val = source[key] ;

if (typeof val !== "undefined") {

target[key] = val ;

}

}

}

return target ;

}

function Alert(parent, message, opts){

opts = extend({

width: 320,

height: 240,

title: "Alert",

modal: false

}, opts) ;

extend(this, opts) ;

}

无状态API和有状态API

无状态API提供的函数或方法，只取决于输入，而与程序的状态改变无关。

有状态API如Web的Canvas库，它提供了绘制形状和图片到其平面的用户界面元素方法。

使用结构类型来实现多态性，从而避免使用继承

编译类型语言使用类和继承来结构化程序。

动态语言可以使用结构类型（或叫鸭子类型）来设计对象的接口，任何对象只要具有预期的结构就属于该类型，从而它不需要编写任何显示的声明，一个调用某个对象方法的函数能够与任何实现了相同接口的对象一起工作。

function Wiki(format){

this.format = format ;

}

Wiki.prototype.displayPage = function(source){

var page = this.format(source) ;

var title = page.getTitle() ;

var author = page.getAuthor() ;

var output = page.toHTML() ;

//...

};

//不需要使用子类化继承，只要类实现相同的接口

Wiki.format.MEDIAWIKI = function(source){

// extract contents from source

//...

return {

getTitle: function() {} ,

getAuthor: function() {},

toHTML: function() {}

};

} ;

var app = new Wiki(Wiki.formats.MEDIAWIKI) ;

foo.call(this, arg1,arg2,arg3) == foo.apply(this, arguments) == this.foo(arg1, arg2, arg3)

Javascript无法实现重载（函数重名，但参数不同）

因为在调用函数时，所提供的参数数量不一定要和函数的参数数据一致，如果调用函数时比它拥有的参数少些，那么任何你未提供的参数的值都是undefined.若调用函数的参数比它有的多了，那么多出来的会被忽略。

## <JavaScript忍者秘籍>

调试JavaScript两个重要的方法：日志记录和断点

### 函数

#### 函数是第一型对象

就像普通的JavaScript数据类型，函数可能被任意进行引用，或声明成对象字面量，甚至可以将其作为函数参数进行传递

1. 对象在JavaScript中有如下功能：

* 通过字面量进行创建
* 赋值给变量，数组或其他对象的属性
* 作为参数传递给函数
* 作为函数的返回值进行返回
* 拥有动态创建并赋值的属性和方法

1. 所有的函数调用都会传递两个隐式参数：arguments and this
2. 每个函数都有apply() and call()方法，注意函数是对象，拥有属性和方法
3. 构造器函数的特殊性

Javascripty语言最重要的功能之一是可以在代码的任何地方创建函数，只要能用表达式，就能创建函数。除了使代码更加紧凑和易于理解（将函数声明放在使用的地方）外，在一个函数不用多个地方引用的时候，该特性还可以消除使用不必要的名称所带来的全局命名空间污染

函数声明

JavaScript函数是使用函数字面量进行声明从而创建函数值的

#### 函数与变量作用域

**函数内部的作用域与大多数其他语言的作用域不同？**

* 变量的作用域开始于声明处，结束于函数尾部，其会跨域边界块。
* 内部函数在当前函数的任何地方都可用（提升），即使是提前引用。

在JavaScript中，作用域是由function进行声明的，而不是代码块。声明的作用域创建于代码块，但不是终结于代码块（其他语言是终结于代码块的）

函数可以在其作用域范围内提前被引用，但变量不行

函数调用

有四个不同的方式：

* 作为一个函数进行调用
* 作为一个方法进行调用，在对象上进行调用，支持面向对象编程
* 作为构造器进行调用，创建一个新对象
* 通过apply() or call()方法进行调用

所有的函数调用都会传递两个隐式参数：arguments and this。所谓隐式，也就意味着这些参数不会显示列在函数签名里，但是它们默默地传递给函数并存在于函数作用域内

This参数引用了与该函数调用进行隐式关联的一个对象，被称之为函数上下文。

#### 函数上下文

**this参数和Java中的this不同？**

**Java中的this依赖于函数的声明，而Javascript中的this则依赖于函数的调用方式**。基于这个事实，将this称为调用上下文（invocation context）可能更为清晰。其函数上下文会随着函数调用方式的变化而变化，而不是取决于函数是怎样声明的

**不同的调用机制决定了函数上下文的不同**：

* 作为普通函数进行调用时，其上下文是全局对象(window)
* 作为方法进行调用时，其上下文是拥有该方法的对象
* 作为构造器进行调用时，其上下文是拥有该方法的对象
* 通过函数的apply() or call()方法进行调用时，上下文可以设置为任意值

#### 构造器

调用构造器时会发生：

* 创建一个新的空对象
* 传递给构造器的对象是this参数，从而成为构造器的函数上下文
* 如果没有显式的返回值，新创建的对象则作为构造器的返回值进行返回

构造器的目的是通过函数调用初始化创建新的对象

**若将构造函数当作普通函数调用**，产生的结果是？

function Ninja(){

this.skulk = function() { return this;}

}

var whatever = Ninja() ;

则结果：在window对象上创建一个skulk属性，并将返回的window对象赋值给whatever.

注意函数和方法命名以动词开头并且以小写字母开头；构造器的命名是由名词描述且以大写字母开头。

实现面向函数式编程的forEach

function forEach(list, callback) {

for (var n=0; n<list.length; ++n) {

callback.call(list[n], list[n], n) ;

}

}

forEach([1, 2, 3], function(value, index){

…

});

#### 匿名函数

有强类型和面向对象语言开发背景的开发人员可能会认为，函数和方法在使用之前必须要事先进行定义并进行命名，然后才能使用；在函数式语言中，包括JavaScript，函数经常在需要的时候才进行定义 ，然后用完就抛弃。

JavaScript的强大威力依赖于是否将其作为函数式语言进行使用，所以会大量使用匿名函数。**函数式编程专注于：少、通常无副作用，将函数作为程序代码的基础构件块。为了不让不必要的函数名称污染全局命名空间，我们将创建大量的小型函数进行传递，而不是创建包含大量命令语句的大型函数**

#### 递归函数

对函数进行引用的四种方式

1. 通过名称进行引用

function chirp(n) {

return n > 1 ? chirp(n-1) +”-chirp” : “chirp”;

}

1. 作为一个方法进行引用（通过对象的属性名称）

var ninja = {

chirp: function(n) {

return n>1 ? this.chirp(n-1) +”-chirp” : “chirp”;

}

};

1. 通过内联名称进行引用

var ninja = {

chirp: function **signal(n)** {

return n>1 ? signal(n-1) +”-chirp” : “chirp”;

}

};

注意：尽管可以给内联函数进行命名，但这些名称只能在自身函数内部才是可见的，这就是为什么要将全局函数作为window的方法进行创建的原因。不使用window的属性，我们没有办法引用这些函数。虽然给内联函数取名，但它的作用是有限的

1. 通过arguments的callee属性进行引用

var ninja = {

chirp: function(n) {

return n>1 ? **arguments.callee**(n-1) +”-chirp” : “chirp”;

}

};

参数arguments是隐式传递给每一个函数的，arguments有一个callee属性，callee属性引用的是当前所执行的函数。

所有这些不同的函数引用技术，在我们面临大规模复杂程序时提供了很大帮助，它为我们提供各种方式来引用函数，而无需采用硬编码或脆弱的变量/属性名依赖

#### 将函数视为对象

函数可以有属性，也可以有方法，可以分配给变量和属性，也可以享有所有普通对象所拥有的特性，而且还有一个超级特性：它们可以被调用。

在函数属性中保存其他函数，以便稍后进行引用或调用

使用函数属性创建缓存（记忆）

var fn = function(){};

fn.prop = “dada…”;

实现缓存功能

（缓存记忆：能够记住先前计算的结果）

function getElements(name) {

if (!getElements.cache) **getElements.cache** = {};

return getElements.cache[name] =

getElements.cache[name] ||

document.getElementsByTagName(name) ;

}

!!(cache[id] = fn); //!!构造是一个可以将任意JavaScript表达式转化为其等效布尔值的简单方式。

参数列表可变长度与数组无换

var maxValue = Math.max(1, 2, 3) ;

如果Math.max只支持可变参数，如何支持数组参数

var maxValue = Math.max.apply(Math, [1, 2, 3]) ;

最好封装成一个函数

function max(array) {

return Math.max.apply(Math, array) ;

}

var maxValue = max([1, 2, 3]) ;

#### 函数重载

在其他更纯的面向对象语言里，方法重载通常是通过在同名方法（但不同的参数）里声明不同的实现来达到目的的。但在JavaScript中，我们重载函数的时候只用一个实现，只不过这个实现内部是通过传入参数的特性和个数进行相应修改来达到目的的。

在JavaScript中，没有强制函数声明多少个参数就得传入多少参数。函数是否可以成功处理这些参数（或缺少的参数）完全取决于函数本身的定义。

隐式传递给所有函数的内置arguments参数。所有的函数都隐式传递了这一重要参数，这将使得函数有能力处理任意数量的参数。

通过函数的length属性，可以知道**声明了多少命名参数**

通过arguments.length,可以知道在**调用时传入了多少参数**

基于传入的参数，有几种方法判断并进行函数重载

* 根据传入参数的类型执行不同的操作
* 通过某些特定参数是否存在来进行判断
* 通过传入参数的个数进行判断

基本传入参数个数。

最简单的：

var ninja = {

whatever = function() {

switch (arguments.length) {

case 0:

…

break;

case 1:

…

break;

…

}

}

};

利用函数是对象，可以存属性和方法

function addMethod(object, name, fn) {

var old = object[name];

object[name] = function() {

if (fn.length ==arguments.length) return fn.apply(this.arguments);

else if (typeof old == ‘function’) return old.apply(this, arguments) ;

};

}

var ninja = {};

addMethod(ninja, ‘whatever’, function(){…});

addMethod(ninja, ‘whatever’, function(a){…});

addMethod(ninja, ‘whatever’, function(a, b){…});

客户端使用

ninja.find();

ninja.find(a);

ninja.find(a, b);

函数判断

function isFunction(fn) {

return typeof fn == “function”;

}

function isFunction(fn) {

return Object.prototype.toString.call(fn) === “[object Function]”;

}

函数调用时，通过控制传递函数上下文，我们在当前对象上执行该对象没有的方法。利用这种技术，可以利用像Array and Math上已有的方法，在自定义数据上进行操作

### 闭包

可以让函数访问所有的变量和函数，只要这些变量和函数存在于该函数声明时的作用域内就行。使用闭包时，闭包里的信息会一直保存在内存里，直到这些信息确保不再被使用（可以安全进行垃圾回收），或页面卸载时，JavaScript引擎才能清理这些信息

#### 闭包的常见用法：

##### 私有变量

封装一些信息作为私有变量，即限制这些变量的作用域

function Ninja() {

var feints = 0 ;

this.getFeints = function() {

return feints;

};

this.feint = function () {

feints ++ ;

};

}

在构造器内隐藏变量，使其在外部作用域不可访问，但是可以存在于闭包内

##### 回调callback

jQuery(“#testButton”).click(function() {

var elem$ = jQuery(“#testSubject”);

elem$.html(“Loading…”);

jQuery.ajax({

url: “test.html”,

success: function(html) {

elem$.html(html);

}

});

});

##### 计时器timer

var tick = 0 ;

var timer = setInterval(function(){

if (tick < 100) {…}

else {

clearInterval(timer);

}

}, 10);

#### 函数上下文

<button id=”test”>Click Me!></button>

var button = {

clicked: false,

click: function() {

This.clicked = true;

}

};

var elem = document.getElementById(“test”);

elem.addEventListener(“click”, button.click, false) ;

浏览器的事件处理系统认为函数调用的上下文是事件的目标元素，则函数button.click上下文是<button>元素，而不是button对象

通过使用匿名函数，apply()和闭包，我们可以强制让特定的函数在调用时都使用特定所需的上下文。

function bind(context, name) {

return function(){

return context[name].apply(context, arguments);

};

}

elem.addEventListener(“click”, bind(button, “click”), false) ;

#### 缓存记忆

创建一个数据存储（缓存），并且能保存我们之前所有预先计算过的值

Function.prototype.memoized = function(key) {

this.\_values = this.\_values || {};

return this.\_values[key] !== undefined ?

this.\_values[key] :

this.\_values[key] = this.apply(this, arguments) ;

};

Function.prototype.memoize = function() {

var fn = this;

return function() {

return fn.memoized.apply( fn, arguments ) ; //在缓存记忆上封装原始的函数

};

};

var isPrime = (function(num){

var prime = num != 1;

for (var i = 2; i < num; ++i) {

if (num % i == 0) { prime = false; break; }

return prime ;

}

}).memoize();

isPrime(17);

在memoize()方法中，我们构建了一个闭包，通过将上下文复制到一个变量中从而记住需要缓存记忆的原始函数（通过上下文）。这是一种常用的技巧：每个函数都有自己的上下文，所有函数从来都是闭包的一部分。但是可以通过创建一个变量引用到上下文上， 从而可以将上下文变成闭包的一部分。

#### 函数包装

是一种封装逻辑的技巧，用于在单个步骤内重载创建新函数或继承函数。

#### 闭包记住的是变量的引用，而不是闭包创建时该变量的值

通过在for循环内加入即时函数，从而可以将正确的值传递给即时函数（也就是内部函数的闭包），进而让处理程序也得到正确的值

for (var i = 0 ; i < div.length; ++ i) (function(n){

div[n].addEventListener(‘click’, function(){});

})(i);

原型与面向对象

使用原型所定义的属性和功能会自动应用到对象的实例上。一旦进行了定义，原型的属性就会变成实例化对象的属性，从而作为复杂对象创建的概览

原型类似于经典面向对象语言中的类。实际上，JavaScript原型的主要用途就是使用一种类风格的面向对象和继承技术进行编码

所有的函数在初始化的时候都有一个prototype属性，该属性的初始值是一个空对象。只有函数在作为构造器的时候，prototype属性才会挥更大的作用

与面向对象语言Java or C++不同，JavaScript通过构造器执行new操作符来初始化一个新对象，而没有类的定义。而new操作符在构造器函数上应用以后，就会触发创建一个新分配的对象。

原型的更改会影响已创建的对象

function Ninja() {

this.swung = true;

}

var ninja = new Ninja();

Ninja.prototype.swingSword = function() {

return this.swung;

};

对象创建后，在构造器的原型上添加一个方法，对象可以使用新的原型方法。这是因为，原型上的属性并没有复制到其他地方，而是附加到新创建的对象上了，并可以和对象自身的属性引用一起协调运行。

查找过程：

1. 在引用对象的一个属性时，首先检查该对象本身是否拥有该属性，如果有，则直接返回，如果没有…
2. 则，再查看对象的原型prototype,检查该原型上是否有所要的属性，如果有，则直接返回，如果没有…
3. 则，该值是undefined.

JavaScript中的每个对象，都有一个名为constructor的隐式属性，该属性引用的是创建该对象的构造器。由于prototype是构造器的一个属性，所以每个对象都有一种方式可以找到自己的原型

ninja.constructor === function Ninja() {this.swung = true;}

ninja.constructor.prototype.swingSword === ninja. swingSword

这就解释了，为什么对象创建之后原型的更改会生效。原型是实时附加在对象上的可以很好的与该对象的自身属性引协调在一起。引用时，可以随时使用原型。

这些无缝的“实时更新”带给我们令人难以置信的威力和可扩展性，而一般语言都没有这种功能。利用该实时更新特性，我们可以构建一个函数化的框架，用户可以使用该框架进行进一步扩展，即便是在对象被实例化之后。

通过构造器判断对象类型

typeof ninja == “object”

ninja instanceof Ninja

ninja.constructor === Ninja

对象的constructor属性作为创建该对象的原始函数的引用，被添加在所有的实例上。由于该属性只是原始构造器的一个引用，所以我们使用该引号再次实例化一个新Ninja对象

var ninja2 = new ninja.constructor(); 等效于var ninja2 = new Ninja();

我们不用知道原有的构造器函数就可以再次创建一个新实例，即便是原始的构造器在作用域内已经不存在了，我们也完全可以在幕后使用该引用。

继承与原型链

创建这样一个原型链最好的方式是，使用一个对象的实例作为另外一个对象的原型：

SubClass.prototype = new SuperClass();

这将保持原型链，是因为SubClass实例的原型将是SuperClass的一个实例，该实例不仅拥有原型，还持有SuperClass的所有属性，并且该原型指向其自身超类的一个实例

function Person() {}

Person.prototype.dance = function() {};

function Ninja() {}

Ninja.prototype = new Person();

从而Ninja是Person的子类

注意：Ninja.prototype = Person.prototype.这样做的话，Ninja原型上的任何修改都会影响到Person的原型，因为它们是同一个对象，这样做肯定会有不良的副作用

所有原生JavaScript对象构造器（如Object, Array, String, Number, RegExp, Function）都有可以被操作和扩展的原型属性，这是有道理的，因为每个对象构造器自身就是一个函数。利用原型，我们可以扩展这门语言自身的功能，从而给这门语言引用新的特性或丢失的特性。

If (!Array.prototype.forEach) {

Array.prototype.forEach = function(callback, context) {

for (var i = 0 ; i < this.length; ++ i) {

callback.call(context || null, this[i], i, this);

}

};

}

在Array的原型上添加新方法，之后，所有的数组都有这样一个方法了。注意context||null表达式可以防止我们将undefined传递给call().

所有DOM元素都继承于HTMLElement构造器。通过访问HTMLElement的原型，浏览器可以为我们提供扩展任意HTML节点的能力。

HTMLElement.prototype.remove = function(){

if (this.parentNode) this.parentNode.removeChild(this) ;

};

document.getElementById(“b”).remove();

尽管浏览器暴露了其构造器和原型，它们选择性禁用了通过构造器创建元素的能力，所以不可以直接通过其构造器进行实例化如var elem = new HTMLElement();

疑难陷阱

不可以扩展原生Object.prototype

JavaScript提供一个名为hasOwnProperty()的方法，使用该方法可以确定一个属性是在对象实例上定义的，还是从原型里导入的

function(){

var keys = [];

for (var I in this) {

//使用hasOwnProperty()忽略原型对象属性

if (this.hasOwnProperty(i) keys.push(i);

}

return keys;

};

避免在Number的原型上做扩展。

最好的策略是单独实现原生对象的各个功能，而不扩展出子类

function MyArray() {}

MyArray.prototype.length = 0 ;

(function(){

var methods = [‘push’, ‘pop’, ‘shift’, ‘unshift’, ‘slide’, ‘splide’, ‘join’];

for (var I = 0 ; I < methods.length; ++ i) (function(name){

MyArray.prototype[ name] = function() {

return Array.prototype[ name ].apply(this, arguments) ;

};

})(method[i]);

})();

使用即时函数，使用apply()方法，将从Array中选中的方法复制到新类上

判断函数是否是作为构造器进行调用的

function isCalledViaConstructor() {

return this instanceof arguments.callee;

}

函数在作为构造器进行执行的时候，以上表达式结果是true,如果作为普通函数执行，则返回false.

正则表达式

正则表达式是一个拆分字符串并查询相关信息的过程。正则表达式通常被称为一个模式，是一个用简单方式描述或匹配一系列符合某个句法规则的字符串

创建正则表达式：

通过正则表达式字面量

var pattern = /test/;

**像字符串是用引号进行界定的一样，正则字面量是用正斜杠进行界定的**

通过构造RegExp对象的实例

在开发过程中，如果正则是已知的，则优先选择字面量语法，而构造器方式则是用于在运行时，通过动态构建字符串来构建正则表达式

如何使用表达模式？

精确匹配

如果一个字符不是特殊字符或操作符，则表示该字符必须在表达式中出现，如/test/

匹配一类字符

匹配一个有限字符集中的某一个字符 [abc]

匹配一组有限字符集以外的字符 [^abc]

字符集范围 [a-m]

转义 \\

在正则里，使用反斜杠可以对任意字符进行转义，让被转义字符作为字符本身进行匹配

匹配开始与匹配结束

插入符号^作为正则表达式的第一个字符，则表示要从字符串的开头进行匹配，如/^test/

美元符号$表示该模式必须出现在字符串的结尾,如/test$/

重复出现

一次或根本不出现 ?

一次或多次出现 +

0次或多次 \*

指定次数 {5}

指定次数区间 {4, 10}

预定义字符类

. 匹配除了新行\n之外的任意字符

\d 匹配任意数字，等价于[0-9]

\w 匹配包括下划线的任意单词字符，等价于[A-Za-z0-9]

\s 匹配任何空白字符，包括空格、制表符、换页符

分组：用括号进行分组

或操作符： ｜

反向引用：将捕获作为正则表达式中能够成功匹配术语时的候选字符串。在反斜杠后面加一个要引用的捕获数量，如\1, \2, …

/<(\w+)>(.+)<\/\1>/ 注意\1是第一个捕获(\w+)的引用

可以匹配<strong>whatever</strong>

编译正则表达式

编译发生在正则表达式第一次被创建的时候，而执行则是发生在我们使用编译过的正则表达式进行字符串匹配的时候。在编译期间，表达式通过JavaScript引擎进行解析，并转换成其内部表示。解析和转换这个过程，在每个正则表达式创建的时候都会发生。

正则表达式在创建之后都处于编译后的状态。所以正则表达式只编译一次，并将其保存在一个变量中以供后续使用，这是一个重要的优化的过程。

用构造器new RegExp()创建正则表达式的使用。这种技术允许我们，在运行时通过动态创建的字符串构建和编译一个正则表达式。对于构建大量重用的复杂表达式来说，这是非常有用的。

创建带有反斜杠的字面量正则表达时，只需要提供一个反斜杠即可。但是，由于我们在字符串中写反斜杠，所以需要双反斜杠进行转义

String.replace

将正则表达式作为replace()方法的第一个参数时，导致在该模式的匹配元素（全局匹配的主知，就是多个匹配元素）上进行替换，而不是在固定字符串上进行替换

当替换值（第二个参数）是一个函数时，每个匹配都会调用该函数（记住，全局搜索会在源字符串中匹配所有的模式实例）并带有一串参数列表（匹配的完整文本，匹配的捕获，一个捕获对应一个参数，匹配字符在源字符串中的索引，源字符串）

Function upper(all, letter) { return letter.toUpperCase(); }

“boder-bottom-width”.replace(/-(\w)/g, upper) === “borderBottomWidth”

由于全局正则在每一次成功匹配的时候都会调用这样的替换函数。这种技术甚至可以超越简单的机械替换，并且可以作为字符串遍历的一种手段

function trim(str) { return (str || “”).replace(/^\s+|\s+$/g, “”); }

trim(“ #id div.class “) = “#id div.class”

线程和定时器

JavaScript是在一个单线程中执行所有的事件回调。定时器提供了一种让一段代码在一定毫秒之后，再异步执行的能力。

JavaScript定时器的延迟时间是不能保证的，原因和JavaScript线程的本质有很大关系

因为浏览器的所有JavaScript代码都是在单线程中执行的，所以异步事件的处理程序，如用户界面事件和定时器，在线程中没有代码执行的时候才进行执行。

**由于JavaScript的单线程特性，JavaScript在同一时间只能执行一个代码块，这些代码块的执行就阻塞了异步事件的处理。这意味着，当一个异步事件发生时（如，鼠标单击、定时器触发甚至是XMLHttpRequest的完成事件），它就会排队，并且在线程空闲时才进行执行。**

由于Javascript是单线程，在特定的时候间点只能运行一个执行代码，而且我们也不能确定定时器处理程序到底是在什么时候执行的。

定时器回调从来不保证执行的具体时间。它并非像interval定时器那样每隔一段时间触发一次，而是在执行之后重新设置定时器一段时间再次触发。

JavaScript引擎是单线程执行，异步事件必须要排队等待才能执行

如果无法立即执行定时器，该定时器会被推迟到下一个可用的执行时间点上

id = setTimeout(fn, delay);

clearTimeout(id);

id = setInterval(fn, delay);

clearInterval(id);

浏览器不保证我们指定的延迟间隔，虽然可以指定特定的延迟值，但其准确性却并不总是能够保证，尤其是在延迟值很小的时候。

如果一段脚本的运行时间超过5秒，有些浏览器将弹出一个对话框警告用户该脚本无法响应。

## JS技巧

!! ‘foo’ 通过两个取反，可以强制转换为Boolean类型

+’45’ String转为Number

即时函数 防止全局污染

(function (arg) {

// do something

})(arg);

第一组圆括号仅仅是用于划定表达式的范围，而第三个圆括号则是一个操作符

闭包

foo.addEventListener(‘click’, (function() {

var times = 0 ;

return function() {

times ++ ;

console.log(times);

}

})(), false);

常见的即时函数使用方式：简单，自包装功能。各功能所需的变量都保存在闭包内，但对其他地方却都不可见。

在很多编程语言中，作用域是依赖于代码块的。但在JavaScript中，变量的作用域依赖于变量所在的闭包，利用这个简单的构造（即时函数），我们现在可以将作用域限制于代码块、子代码块或各级函数中。

(function($){

$(‘img’).on(‘click’, function(event){});

})(jQuery);

这种技术被很多jQuery插件开发人员所使用，他们在页面中的代码从来不直接使用$.由于假设将$引用jQuery是不安全的，所以这些开发人员在开发插件的时候，都将插件代码放在即时函数内部，从而使用$安全引用jQuery.

有经验的函数式程序员，可以使用即时函数将短名称引用到一个有限的作用域内

(function(v){

Object.extend(v, {

href: v.\_getAttr,

src: v.\_getAttr,

…

});

})(Element.attributeTranslations.read.values);

参数v就是这个长名称数据结构的引用，并存在于即时函数的作用域内

闭包和即时函数尤其有用，它们可以帮助我们让类库尽可能的保持私有，并且可以有选择性的让将一些变量暴露到全局命名空间内。

## ES2015 (ES6)

添加了块作用域，for循环作用域，新的全局let作用域，模块作用域，以及求参数的默认值时使用的附加作用域。

let替代var

let声明的变量拥有块级作用域

let声明的全局变量不是全局对象的属性。这就意味着，你不可以通过window.变量名的方式访问这些变量。它们只存在于一个不可见的块的作用域中，这个块理论上是Web页面中运行的所有JS代码的外层块。

形如for (let x...)的循环在每次迭代时都为x创建新的绑定。

let声明的变量直到控制流到达该变量被定义的代码行时才会被装载，所以在到达之前使用该变量会触发错误

用let重定义变量会抛出一个语法错误（SyntaxError）。如果你的多个脚本中都声明了相同的全局变量，你最好继续用var声明这些变量。如果你换用了let，后加载的脚本都会执行失败并抛出错误。

const

const声明的变量与let声明的变量类似，它们的不同之处在于，const声明的变量只可以在声明时赋值，不可随意修改，否则会导致SyntaxError（语法错误）。

SVG

## <SVG Essentials>

### 基本概念

html5插入SVG，在顶级<svg>元素上设置xmlns=<http://www.w3.org/2000/svg>,HTML解析器会自动辨别<svg>元素和它的子节点都在SVG命名空间内。

For example:

<style>

svg {

display: block;

width: 500px;

height: 500px;

margin: auto;

background-color: lightblue;

}

circle {

fill: lavender;

}

</style>

<svg viewBox="0 0 250 250" xmlns="http://www.w3.org/2000/svg">

<circle cx="125" cy="125" r="100"></circle>

<text x="125" y="125" dy="0.5em" text-anchor="middle">Look Ma, Same Font!</text>

</svg>

svg的width&height是显示在界面上的画布大小，viewBox=”0 0 250 250”指定用户坐标，从而svg里的元素位置相对于0 0, 大小相对于250 250。

若viewBox映射于画布大小，若同比例和，同放大缩小即可

若不同比例，则用preserveAspectRatio=”alignment [meet | slice | none]”指定对齐方式。alignment指定轴和位置.

默认值是xMidYMid meet(即缩小图像以适配可用的空间，并且使它水平和垂直居中)

meet适配画布(视口)，slice剪裁不适合画布的部分，none不考虑等比例缩放，从而使用户坐标适合视口

### 形状元素

<line x0=”x0” y0=”y0” x1=”x1” y1=”y1” />

<rect x=”x-left” y=”y-top” width=”width” height=”height” />

<circle cx=”x-center” cy=”y-center” r=”radius” />

<ellipse cx=”x-center” cy=”y-center” rx=”x-radius” ry=”y-radius” />

<polygon points=”points-list” /> //封闭图形

<polyline points=”points-list” /> //折线段

路径path

<path d=”M60 10 L90 10 L90 30 L60 30 Z”/> //closepath绘制的矩形

大写字母M L Z …表示坐标是绝对坐标，小写字母m l z…表示坐标是相对于当前的画笔位置

M: Moveto L: LineTo Z: closePath H: Horizontal V: Vertical

椭圆弧（7个参数）：椭圆的半径（xRadius, yRadius），椭圆的x轴旋转角度x-aix-rotation，large-arc-flag是否大弧，sweep-flag正角度绘制，终点(x,y)。注意起点由最后绘制点或最后一个moveto命令确定

<path d=”M152 152 A75 60, 0, 1, 1, 302 152”/>

贝塞尔曲线: 想像线是由柔性金属制造，控制点内部是一个磁铁，与控制点越近，吸引力越强

二次贝塞尔曲线:起点，控制点，终点

<path d=”M30 75 Q240 30 300 120”/>

解析曲线画法：直线p1（起点控制点）和直线p2（控制点终点）分别与曲线相切，直线p1,p2的中点连线的中点在曲线上

三次贝塞尔曲线：起点，起点控制点，终点控制点，终点

<path d=”M20 80 C50 20, 160 60, 200 120”/>

解析曲线画法：直线p1（起点控制点1）和直线p2（控制点2终点）分别与曲线相切，直线p1中点,直线p3（控制点1控制点2）中点，p2中点，分别连线的中点再次连线的中点在曲线上 （具体见《SVG精髓》P82）

注意：起点一般是系统的当前点

//裁剪

<defs>

<clipPath id=”curveClip”>

<path id=”curve1” d=”M…” style=”stoke:black; fill:none;”/>

</clipPath>

</defs>

<g id=”shapes”>

<rect …/>

<circle…/>

</g>

<use xlink:href=”#shapes” style=”clip-path: url(#curveClip); “/>

//蒙版

SVG蒙版变换对象的透明度，如果蒙版是不透明的，蒙版可以是任意基本形状，文本，图像或者路径，默认使用用户坐标空间表达

例子：让照片的边缘淡出，使用径向渐变作为蒙版

<radialGradient id=”fade”>

…

</radialGradient>

<mask id=”fademask” maskContentUnits=”objectBoundingBox”>

<rect x=”0” y=”0” width=”1” height=”1” style=”fill: url(#fade);” />

</mask>

<image xlink:href=”\*.jpg” x=”72” y=”72” width=”500” height=”500” style=”mask: url(#fademask);” />

### 笔画，填充，图案，渐变

stoke 笔画颜色,默认值为none

stoke-width 笔画宽度，默认值为1

stoke-dasharray 用一系列的数字来指定虚线和间隙的长度。默认值为none

fill 填充颜色，默认值为black

fill-opacity 默认值为1.0表示完全不透明

图案

每个图案填充对象的一定百分比

<defs>

<pattern id=”tile” x=”0” y=”0” width=”20%” height=”20%” **patternUnits=”objectBoundingBox”**>

<path d=”M0 0 Q5 20 10 10 T20 20” style=”stroke:black; fill:none;”/>

</pattern>

</defs>

<rect x=”20” y=”20” width=”100” height=”100” style=”fill:url(#tile); stroke:black;”/>

以相同大小的图案平铺，不论填充对象的尺寸多大 (标准图形编辑程序)

patternUnits=”userSpaceOnUse”

默认情况下，patternContentUnits=”userSpaceOnUse” 单位表达图案数据。若设置属性值为objectBoundingBox,则路径本身的数据点会基于被填充的对象来确定

<defs>

<pattern id=”tile” x=”0” y=”0” width=”20%” height=”20%”

**patternUnits=”objectBoundingBox” patternContentUnits=”objectBoundingBox”**>

<path d=”M0 0 Q.05 .2 .1 .1 T.2 .2” style=”stroke:black; fill:none;”/>

</pattern>

</defs>

图案可以填充图案

渐变

<defs>

//线性渐变的默认行为是沿着水平线从对象的左侧向右侧过渡

<linearGradient id=”two\_hues”>

<stop offset=”0%” style=”stop-color: #ffcc00;” />

<stop offset=”100%” style=”stop-color:#0099cc;” />

</linearGradient>

//定义渐变起始点和终点,改变渐变方向

<linearGradient id=”down” xlink:href=”#two\_hues” x1=”0%” y1=”0%” x2=”0%” y2=”100%”/>

<linearGradient id=”diagonal” xlink:href=”two\_hues” x1=”0%” y1=”0%” x2=”100%” y2=”100%”/>

//径向渐变： 圆形确定

<radialGradient id=”center\_origin” cx=”0%” cy=”0%” r=”141%”>

<stop offset=”0%” style=”stop-color:#f96;” />

<stop offset=”50%” style=”stop-color:#9c9;” />

<stop offset=”100%” style=”stop-color:#906;” />

</radialGradient>

</defs>

<rect x=”2” y=”2” width=”200” height=”200” style=”fill:url(#two\_hues); stoke:black;”/>

### 分组和引用对象

大多数非抽象的艺术作品都是由一系列命名对象组成的，而这些对象由形状和线条组合而成。<g>元素会将其所有子元素作为一个组合，通常组合还会有一个唯一的id作为名称。每个组合还可以拥有自己的<title> and <desc>来供基于文本的XML应用程序识别

<g>元素可以组合元素，并为它们提供一些注解。 在起始<g>标签中指定的所有样式会应用于组合内的所有子元素。

for example:

<svg width="512" height="512" viewBox="0 0 240 240" xmlns="http://www.w3.org/2000/svg">

<defs>

<g id="house" style="stroke: black;">

<rect x="0" y="41" width="60" height="60" />

<polyline points="0 41, 30 0, 60 41" />

<polyline points="30 101, 30 71, 44 71, 44 101" />

</g>

<g id="twohouse">

<use xlink:href="#house" x="0" y="0" style="fill: #cfc;"/>

<use xlink:href="#house" x="120" y="0" style="fill: #99f;" />

</g>

</defs>

<use xlink:href="#twohouse" x="0" y="0" />

</svg>

### 坐标系统变换

translate(x, y): 获取整个网格，然后把它移动到画布的新位置，而不是移动物体.

scale(xFactor, yFactor): 也是，永远不会改变图形对象的网格坐标或者它的笔画宽度，改变的是对应画布上的坐标系统（网格）的大小

rotate(angle): ，角度的测量是按顺时针增加，水平线的角度为0. 默认的旋转中心点(轴心点)被假定为(0, 0)，被旋转的是整个坐标系统。

rotate(angle, centerX, center): 围绕某个点旋转单个对象：translate(centerX, center) rotate(angle) translate(-centerX, -centerY) 或rotate(angle, centerX, center);这样做的效果是以指定的(x,y)作为原点临时建立一个新的坐标系统执行旋转操作，然后重新建立原始坐标

围绕中心点缩放: translate(-centerX\*(factor-1), -centerY\*(factor-1)) scale(factor), 注意stroke-width也会缩放

matrix( a b c d e f): 变换矩阵

SVG会在计算形状的坐标之前，先对坐标系统应用变换

笛卡尔坐标变换：

<g transform=”translate(0, max(y)) scale(1, -1)”

</g>

### 文本

字符：指带一个数字值的一个或多个字节

符号：指字符的视觉呈现。

一个符号可能由多个字符构成（如一些字体为特定的字母组合），一个字符也可能由几个符号组合而成（如打印符号）

字体：指代表某个字符集合的一组符号。

基线：字体中的所有符号以基线对齐。

em高度(字符总高度) = 上坡度(基线到字符顶部) + 下坡度(基线到字符底部)

<text x=”20” y=”20”>outlined only</text>

(x, y)指定元素内容的第一个字符的基线位置，和所有对象一样，文本的默认样式是黑色填充，没有轮廓

文本的ccs属性:

font-family, font-size, font-weight(bold, normal), font-style(italic, normal), text-decoration(none, underline, overline, line-through), word-spacing, letter-spacing, text-anchor(start, middle, end)

<text x=”10” y=”30”>Switch among<tspan style=”font-style:italic”>italic<tspan>, normal and … </text>

<text>C<tspan style=”baseline-shift:sub;”>12</tspan></text>

<text>6.02 \* 10<tspan baseline-shift=”super”>5</tspan></text>

按指定路径显示文本

文本可以沿任何抽象路径排列，字母会被旋转到与曲线垂直的方向”站立“（即基线是曲线的切线）

<defs>

<path id=”curvePath”>

D=”M30 40 C….”/>

</defs>

<text>

<textPath xlink:href=”#curvePath”>Following a cubic Besizer curve</textPath>

</text>

### 滤镜

使用位图的艺术家通常只关心对象的外观而不是它的几何定义，虽然SVG不是一种位图描述语言，但它仍然允许我们使用一些相同的工具，SVG处理器程序处理一个图形对象时，它会将对象呈现在位图输出设备上；在某一时刻，处理器程序会把对象的描述信息转换为一组对应的像素，然后呈现在输出设备上。

滤镜基元：<filter>之间，每个基元有一个或多个输入，但只有一个输出，输入可以是原始图形(SourceGraphic),图形的Alpha（不透明度）通道(SourceAlpha)，或者前一个滤镜基元的输出

<filter id=”brightness-shadow” filterUnits=”objectBoundingBox”>

//图像作为背景

<feImage xlink:href=”sky.jpg” result=”sky”/>

//各颜色分量线性变换

<feComponentTransfer in=”sky” result=”sky”>

<feFuncR type=”linear” slope=”3” intercept=”0” />

<feFuncG type=”linear” slope=”1.5” intercept=”0.2” />

<feFuncB type=”linear” slope=”2” intercept=”0” />

</feComponentTransfer>

//高斯模糊

<feGaussianBlur in=”SourceAlpha” stdDeviation=”2” result=”blur” />

//平移

<feOffset in=”blur” dx=”4” dy=”4” result=”offsetBlur” />

//组合各滤波为一个滤波

<feMerge>

<feMergeNode in=”sky” />

<feMergeNode in=”offsetBlur” />

<feMergeNode in=”SourceGraphic” />

</feMerge>

</filter>

<g id=”flower” style=”filter: url(#sky-shadow)”>

</g>

还有各种滤镜: <feBlend>图像点运算，<feFlood>提供纯色区域用于合并，<feTile>提取输入信息作为图案，然后平铺填充滤镜指定的区域，<feDiffuseLighting>漫反射照明，<feSpecularLighting>镜面反射照明，<feMorphology>形态滤波，<feConvolveMatrix>卷积, <feTurbulence>生成大理石，云彩等人工纹理

<defs>

<filter id=”blur-background”>

//滤镜对象访问已经渲染到画布上的图片的某一部分作为源

<feGaussianBlur in=”BackgroundImage” stdDeviation=”2” result=”blur” />

</filter>

<g enable-background=”new”>

<rect …/> //作为背景图像

<circle … style=”filter: url(#blur-background);” />

</g>

### SVG动画 = SMIL动画 + CSS动画

#### SMIL动画

<rect x=”10” y=”10” width=”200” height=”200” storke=”black” fill=”none”>

<animate attributeName=”width” from=”200” to=”20” begin=”0s” dur=”5s” />

</rect>

attributeName: 动画属性(width)

attributeType: auto/CSS/XML, 默认值为auto, 首先搜索css属性，然后才是xml属性

属性的起始结束值: from -> to

动画的开始时间和持续时间：begin dur

fill: remove, freeze, 默认为remove, 动画结束后,动画属性width返回原始值

repeatCount=“indefinite”: 无限重复

<rect x=”-10” y=”-10” width=”20” height=”20”>

<animateTransform attributeName=”transform” type=”scale” from=”1” to=”4 2” additive=”sum” begin=”0s” dur=”4s” />

<animateTransform attributeName=”transform” type=”rotate” from=”0” to=”45” additive=”sum” begin=”0s” dur=”4s” />

</rect>

additive=”sum”: 同时进行坐标变换 （动画）

<text x="1800" y="900">Hello World!

<animateMotion from="0, 0" to="-5000, 0" dur="10s" repeatCount="indefinite" />

</text>

animateMotion: 使对象沿着做任意路径运动

<animateMotion path=”…” rotate=”auto” keyPoints=”0; 0.2; 0.8; 1” keyTimes=”0; 0.33; 0.66; 1” calcMode=”linear” dur=”6s” />

rotate=”auto”: 物体沿着路径站立地运动

keyPoints, keyTimes: 路径关键点，控制动画在不同值之间过渡的速度

calcMode: “linear”, “spline”, 插值方法

#### CSS动画：

选择想要运动的元素，然后设置将动画属性作为一个整体进行计算

告诉浏览器改变选中元素的哪个属性以及在动画的什么阶段，这些都定义在@keyframes说明符中

<use id=”star” class=”starStyle” xlink:href=”#starDef”/>

.starStyle {

animation-name: starAnim ; //@keyframes说明符的名称

animation-duration: 2s ; //持续多久

animation-iteration-count: 4 ; //重复几次

animation-direction: alternate;//正向还是反向

animation-timing-function: ease ;//插值函数

animation-play-state: running ;//running/paused

animation-delay: //等待多久才开始动画

animation-fill-mode://动画结束时的属性值，forwards(动画前属性值)，backward(动画后)

}

@keyframes starAnim {

0% {

fill-opacity: 1.0 ;

stroke-width: 0;

}

100%{

fill-opacity: 0;

stroke-width: 6;

}

}

Javascript创建动画：

在动画过程中，不断修改需要动画的属性。

方法一: 肯定不行

使用while(true)，相当于频繁地询问时钟：“现在什么时间了？”这样会使程序没有空闲时间来做其化的事情。在SVG脚本中，则会让计算机没有机会处理其他的任务。因为如果计算机陷入无休止的循环中，则没有时间来重绘屏幕，这样不管你更新属性有多快，你的动画都会变得很慢或者很卡

正常做法： 电影，视频会每秒更新画面30到60次（视频帧率），30-60次已经足够让人眼认为画面是连续的

方法二：setTimeout (周期性调度，不推荐)

setTimeout没有requestAnimationFrame()考虑那么多，不会调整计算机当前正在做的事情，而且无论窗口是否可见都会始终执行动画函数。

方法二：requestAnimationFrame (推荐)

function updateClock() {

fun() ;

window.requestAnimationFrame(updateClock()) ;

}

### 交互 (建议直接采用d3)

<a class="shapes" xlink:href="http:///www.w3.org/SVG/">

<circle cx="50" cy="70" r="20" />

<rect x="75" y="50" width="40" height="40" />

</a>

a.shapes:hover, a.shapes:focus {

stroke: #66f ;

stroke-width: 2;

}

仅<a>的子图形circle, rect会响应，子图形之间不会响应

当鼠标指针在元素上方时，:hover伪类会生效，而当键盘操作聚焦到某个元素上时，:focus伪类会生效

element.getAttribute(attributeName): 以字符串形式返回属性的值

element.setAttribute(name, newValue): 改变属性值，若指定属性不存在，则会创建

element.removeAttribute(name): 删除属性

element.style.getPropertyValue(propertyName): 获取指定样式

element.style.setProperty(propertyName, newValue, priority): 修改属性

element.style.removeProperty(propertyName): 删除属性

element.textContext: 返回节点所有后代的文本拼接后的字符串，修改该属性时，则会用一个文本块替换所有的后代节点

事件类型：

focusIn/focusOut: 元素接受焦点/失去焦点

click

mousedown/mousemove/mouseup

mouseover/mousemove/mouseout

SVGLoad/SVGUnload/SVGAbort/SVGError: 文档加载，移除，中止，错误事件

keydown/keyup

var circle = document.getElementById(“circle”) ;

circle.addEventListener(“click”, function(evt){

var obj = evt.target ;

obj.setAttribute(“r”, “30”) ;

}) ;

//拖拽对象

circle.addEventListener(“mousedown”, startColorDrag, false) ;

circle.addEventListener(“mousemove”, doColorDrag, false) ;

circle.addEventListener(“mouseup”, endColorDrag, false) ;

### SVG DOM

除了DOM标准，SVG还定义自定义对象，使得操作二维图形更容易，利用这些方法很容易精确指定文本或路径元素的位置，控制动画时间，以及在不同坐标系统之间自由转换

SVG定义一些自定义对象来表示不同的几何数据： SVGLength, SVGAngle, SVGRect, SVGPoint, SVGMatrix, SVGTransform, SVGTransformList, SVGXxxList, SVGAnimatedXxx

SVG接口方法：

SVG命名空间中的任何元素：SVGElement.ownerSVGElement(), …

SVG图形元素，<g>或<svg>: SVGLocatable.getBBox(), …

<svg>: SVGSVGElement.createSVGXxx(), …

<path>： SVGPathElement.getTotalLength(), …

SVGTextContentElement, <text>,<tspan>和<textPath>:

据系统语言显示界面

<switch>

<g systemLanguage=”en”>

<text x=”10” y=”30”>English</text>

</g>

<g systemLanguage=”ch”>

<text x=”10” y=”30”>Chinese</text>

</g>

</switch>

### SVG样式表 （详细见p253）

display: inline, block, list-item, …

fill:

fill-opacity:

fill-rule:

filter:

font:

font-family:

font-size:

…

### 推荐使用Javascript库： D3, Raphael, Snap.svg

D3针对SVG写的库，知道SVG中的元素名，如知道circle表示SVGCircleElement

图形设计师通常会使用某些图形软件来生成SVG, 而程序员则使用脚本将原始数据转换为SVG.

XSLT(Extensible Stylesheet Language Transformations可扩展样式表转换语言)，使用XML语法来定义如何将一个XML文件转换为另一个XML文件的方法。（注意SVG是一个XML应用）

将XSLT和XML文件一起交给XLST处理器，会生成SVG文档中对应的元素，并将对应的值填好， 具体见P231

XML处理一组属性就像把一堆东西装进包里，并没有隐含的顺序问题，而处理元素时就像列表上的项目，是存在顺序问题的。

字符转义：

实体引用：

< $lt;

> $gt;

‘ &apos;

“ &quot;

& &amp;

字符编码：

Unicode试图将世界上所有的语言包含在一个字符集中。

欧洲文档使用ISO拉丁字符集，日本文档使用Shite-JIS, 中文文档使用GB2312, Big5

命名空间：

SVG使用<http://www.w3.org/2000/svg>作为它的命名空间。命名空间声明会应用给所在元素包含的所有元素，包括容器元素。SVG程序把XML移交给解析器，然后返回关于XML文档内容的信息。通常解析器通过事件或者DOM做这件事

# Canvas

## <HTML5 Canvas 开发详解>

### 基本概念

Canvas是屏幕上的一个由Javascript控制的即时模式位图区域。即时模式是指在画布上呈现像素，canvas通过javascript调用canvas API,在每一帧中完全重绘屏幕上的位图。**Canvas是一个即时模式的绘图界面，意味着如果什么东西发生了变化就需要即时重新绘制**。

Flash, Silverlight, **SVG是在保留模式下运行，在保留模式下，对象显示列表由图形渲染保存，通过在代码中设置属性控制展示在屏幕上的对象**，这使得程序员可以远离底层操作，但是它弱化了对位图屏幕最终渲染效果的控制。保留模式采用一个绘制界面储存一组对象，并通过一个显示列表操作。

**文档对象模型(DOM)代表了在html页面上的所有对象，它是语言中立且平台中立的。画布元素本身可以通过DOM,在web浏览器中经由canvas 2D环境访问。但是在canvas中创建的单个图形元素是不能通过dom访问的。因为画布工作在即时模式，它并不保存自己的对象**，只是说明在每个单个帧里绘制什么

Javascript事件，当定义的事件发生时，事件从对象发出。其他对象监听事件，这样就可以基于事件进行处理。用javascript可以监听对象的一些常见事件，包括键盘输入，鼠标移动以及加载结束。

在开始标签<canvas>和结束标签</canvas>中间可以添加文本，一旦浏览器执行html页面时不支持canvas,就会显示这些文字。Canvas可以导出到图像theCanvas.toDataURL()

<canvas id=”canvas” width=”500” height=”500”></canvas>

CanvasREnderingContext2D采用画布左上角为原点，坐标轴向右为x,坐标轴向下为y。

当前状态是一个绘制状态的堆栈，这些状态可以应用到整个画布，状态如下：

变换矩阵：缩放、旋转、变换以及平移的方法

裁切区域：通过clip()方法创建

上下文属性：strokeStye, fillStyle, globalAlpha, linewidth, lineCap, line, Join, miterLimit, shadowOffsetX, shadowOffsetY, shadowBlur, shadowColor, global, CompositeOperation, font, textAlign, textBaseline

function canvasApp(){

var theCanvas = document.getElementById(“canvas”) ;

var context = theCanvas.getContext(“2d”) ;

function drawScreen() {

context.save() ;

context.设置状态;

context.画图(基本形状，文本，图像);

context.restore() ;

}

drawScreen() ;

}

动画循环：是一个函数，每隔一定时间就会被一遍又一遍地重复调用它。这个函数被用于清除画布的内容，然后在画布上重新绘制更新后的图像，文字视频和其化绘画对象。

### 几何形状

context.fillRect(x, y, width, height) ; //矩形

context.strokeRect(x, y, width, height) ; //矩形边框

//直线

context.beginPath() ;

context.moveTo(20, 0) ;

context.lineTo(100, 0) ;

context.stroke() ;

context.closePath() ;

context.arc(x, y, radius, startAngle, endAngle, anticlockwise) ; //弧线（含圆形）

//贝塞尔曲线

context.bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y) ;

context.quadraticCurve(cps, cpy, x, y) ;

裁切区域

Context.beginPath() ;

//裁切画布从(0, 0)点至50\*50的正方形

context.rect(0, 0, 50, 50) ;

context.clip() ;

context.绘图;

context.closePath() ;

画布合成

合成方式有Copy, destination-atop, … source-over(默认)； 源图形是指要绘制在画布上的形状，目标图形是指显示在画布上的位图

### 填充对象： 基本颜色，渐变色和图案

context.fillStyle = “rgba(255, 0, 0, 1)” ; //基本颜色

//线性渐变

var gradient = context.createLinearGradient(0, 0, 100, 0) ; //水平

or var gradient = context.createLinearGradient(0, 0, 0, 100) ; //垂直

or var gradient = context.createLinearGradient(0, 0, 100, 100) ; //对角

gradient.addColorStop(0, ‘rgb(255, 0, 0)’) ;

gradient.addColorStop(.5, ‘rgb(0, 255, 0)’) ;

gradient.addColorStop(1, ‘rgb(255, 0, 0)’) ;

context.fillStyle = gradient ;

context.fillRect(0, 0, 100, 100) ;

//径向渐变 (两个圆的圆心和半径)

var gradient = context.createRadialGradient(50, 50, 25, 50, 50, 100) ;

gradient.addColorStop(0, ‘rgb(255, 0, 0)’) ;

gradient.addColorStop(.5, ‘rgb(0, 255, 0)’) ;

gradient.addColorStop(1, ‘rgb(255, 0, 0)’) ;

context.fillStyle = gradient ;

context.fillRect(0, 0, 100, 100) ;

//填充图案

var fillImage = new Image() ;

fillImage.src = “logo.gif” ;

fillImage.onload = function() {

var fillPattern = context.createPattern(fillImage, ‘repeat’) ;

context.fillStyle = fillPattern ;

context.fillRect(0, 0, 200, 200) ;

}

创建阴影

context.shadowOffsetX = -4 ;

context.shadowOffsetY = -4 ;

context.shadowColor = ‘black’ ;

context.shadowBlur = 4 ;

context.fillRect(10, 10, 100, 100) ;

### 画布变换

//围绕物体中心旋转

context.setTransform(1, 0, 0, 1, 0, 0) ;

context.translate(x+.5\*width, y+.5\*height) ;

context.rotate(anguleInRadians) ;

context.fillRect(-.5\*width, -.5\*height, width, height) ;

//从中心点缩放

context.setTransform(1, 0, 0, 1, 0, 0) ;

context.translate(x+.5\*width, y+.5\*height) ;

context.scale(2, 2) ;

context.fillRect(-.5\*width, -.5\*height, width, height) ;

### 清除画布

方法一：背景色填充

context.fillStyle = ‘000000’ ;

context.fillRect(0, 0, theCanvas.width, theCanvas.height) ;

方法二：重置画布大小， 画布内容会被清除

var w = theCanvas.width ;

var h = theCanvas.height ;

theCanvas.width = w ;

theCanvas.height = h ;

方法三：使用清除函数

context.clearRect(0, 0, theCanvas.width, theCanvas.height) ;

### 物体选择

检查一个点是否在当前路径

var isPointInPath = context.isPointInPath(0, 0) ;

### 文本

Canvas使用文本，无法使用css样式

context.font = “font-style font-weight font-size font-face”

eg. context.font=”italic blod 24px serif” ;

font-style: normal/italic/oblique/inherit

font-weight: normal/bold/bolder/lighter/auto/…

font-face:serif/sans-serif/cursive/fantasy/monospace/…

context.fillStyle: CSS颜色，CanvasGradient, CanvasPattern

var gradient = context.createLinearGradient(x0, y0, x1, y2) ;

gradient.addColorStop(0, “color0”) ;

gradient.addColorStop(.5, “color1”) ;

context.fillStyle = gradient ;

or

var patternImage = new Image() ;

patternImage.src = “texture.jpg” ;

patternImage.onload = function() {

var pattern = context.createPattern(patternImage, “repeat”) ;

context.fillStyle = pattern ;

}

//居中文本，把它向左移动至自身宽度的一半，并且将文本的中心放到画布的绝对中心上

//简单将文本定位到画布一半高度的位置上，而不是试图使用字体的大小垂直居中于画布上

var metrics = context.measureText(str) ;

var xPos = (theCanvas.width-metrics.textWidth) / 2 ;

var yPos = theCanvas.height / 2 ;

context.fillText(str, xPos, yPos) ;

context.strokeStyle: CSS颜色，CanvasGradient, CanvasPattern

or context.strokeText(str, xPos, yPos) ;

垂直对齐和水平对齐

context.textBaseLine = “middle” ;

context.textAlign = “center” ;

阴影文本

context.shadowColor

context.shadowOffsetX

context.shadowOffsetY

context.shadowBlur

渐变动画的文本

function gameLoop() {

window.setTimeout(gameLoop, 20) ;

drawScreen() ;

}

function drawScreen() {

var colorStops = [

{color: “red”, stopPercent: 0},

{color:”blue”, stopPercent: .125},

…]

}

var gradient = context.createLinearGradient(canvas.width/2, 0, canvas.width/2, canvas.height) ;

for (var i=0; i<colorStops.length; ++ i){

gradient.addColorStop(colorStops[i].tempStopPercent, colorStops[i].color) ;

//每次渲染颜色往下调

var tempStopPercent = colorStops[i].tempColorStop.stopPercent ;

tempStopPercent += .015 ;

if (tempStopPercent > 1) tempStopPercent = 0 ;

colorStops[i].stopPercent = tempStopPercent ;

}

### 图像

显示图像

<img src=”ship1.png”> //html方式

//javascript方式

var spaceship = new Image() ;

spaceShip.src = “ship.png” ;

spaceShip.load = function() {

context.drawImage(sapceShip, 0, 0) ;

// 源图像的区域 -> 画布上的目标区域

context.drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh) ;

}

图像变换

context.save(); //保存当前环境到堆栈

context.setTransform(1, 0, 0, 1, 0, 0) ; //初始化矩阵

//编码变换算法

context.translate(x+dx, y+dy) ;

context.rorate(angleInRadians) ;

context.drawImage(…) ; //绘制图像

context.restore() ;

像素修改

var imageData = context.createImageData(sw, sh) ; //ImageData对象

imageData.height, imageData.width, imageData.data

var imageData = context.getImageData(sx, sy, sw, sh) ; //从画布上获取ImageData

context.putImageData(imageData, dx, dy) ; //放入图像数据到画布上

重绘图像

画布鼠标事件

function onMouseMove(e ) {

var mouseX = e.clientX – theCanvas.offsetLeft ;

var mouseY = e.clientY – theCanvas.offsetTop ;

}

画布间的复制

<canvas id=”canvas” …/>

<canvas id=”canvas2” …/>

Var theCanvas = document.getElementById(“canvas”) ;

Var theCanvas2 = document.getElementById(“canvas2”) ;

theCanvas2.getContext(“2d”).drawImage(theCanvas, …) ;

### 计算几何

#### 物体碰撞检测

function boundingBoxCollide(object1, object2) {

var left1 = object1.x,

left2 = object2.x,

right1 = object1.x + object1.width,

right2 = object2.x + object2.width,

top1 = object1.y,

top2 = object2.y,

bottom1 = object1.y + object1.height,

bottom2 = object2.y + object2.height ;

if (bottom1<top2 || top1>bottom2 || right1<left2 || left1>right2) return false ;

else return true ;

}

物体重叠后，还得检测物体重叠部分像素的透明度，如果所有重叠像素的透明度为0, 则认为不碰撞

//重叠区域

var xMin = Math.max(object1.x, object2.x) ;

var yMin = Math.max(object1.y, object2.y) ;

var xMax = Math.min(object1.x+object1.width, object2.x+object2.width) ;

var yMax = Math.min(object1.y+object1.height, object2.y+object2.height) ;

for (var j = yMin; j < yMax; ++ j) {

for (var i = xMin; i < xMax; ++ i) {

var alphaIdx1 = ((j-object1.y)\*object1.width + (i-object1.x)) \* 4 + 3 ;

var alphaIdx2 = ((j-object2.y)\*object2.width + (i-object2.x)) \* 4 + 3 ;

if (object1.getImageData().data[alphaIdx1] !== 0 &&

object2.getImageData().data[alphaIdx2] !== 0 ) {

console.log(“pixel collision”) ;

}

}

}

撞墙反弹： 入射角=反射角

#### 曲线运动

形状（特征点表达，如圆=圆心和半径）基于数学表达式变换

圆周运动：围绕点(x0, y0), 围绕半径radius, 围绕角度angle=[0, infinite]

物体中心(x, y)和半径20

function draw() {

x = x0 + Math.cos(angle)\*radius ;

y = y0 + Math.sin(angle)\*radius ;

angle += speed ;

context.beginPath() ;

context.arc(x, y, 20, 0, Math.PI\*2, true) ;

context.closePath() ;

context.fill() ;

}

螺旋运动： 围绕点不变，围绕半径和围绕角度均匀变化

function draw() {

x = x0 + Math.cos(angle)\*radius ;

y = y0 + Math.sin(angle)\*radius ;

angle += speed ;

radius += deltaRadius ;

}

3次贝塞尔曲线运动：

已知：曲线起点p0=(x0, y0), 控制点p1 and p2, 曲线终点p3

求：x(t) = a1\*t^3 + b1\*t^2 + c1\*t + x0 0 <= t <= 1

y(t) = a2\*t^3 + b2\*t^2 + c2\*t + y0

其中[a1 b1 c1 x0; a2 b2 c2 y0]通过解析式求解

c1 = 3\*(x1-x0) ; c2 = 3\*(y1-y0);

b1 = 3\*(x2-x1) – c1 ; b2 = 3\*(y2-y1) – c2 ;

a1 = x3 – x0 – b1 – c1 ; a3 = y3 – y0 – b2 – c2 ;

#### 模拟物理

模拟重力： 对在矢量上移动的对象的y轴速度应用一个重力常数

v\_x = Math.cos(ANGLE) \* SPEED ;

v\_y = Math.sin(ANGLE) \* SPEED ;

function draw(){

x += v\_x ;

y += v\_y ;

v\_y += GRAVITY ;

…

}

模拟摩擦力：减少x轴速度

function draw() {

v\_x = v\_x – v\_x\*FRICTION ;

x += v\_x ;

}

模拟起飞和着陆：对象从一点移动到另一点的过程中，出发时速度快，到达第二点时减速

缓冲结束：找出两个不同的点，将对象在它们之间移动，当对象移近第二个点时，以线性的方式减慢速度。首先计算两点间速度，然后选择一个百分比数值(EASE\_VALUE),用来在那段距离中的每一帧移动对象。如同距离越来越短，需要移动的数量也越来越短，这使得对象看起来好像从开始到结束移动得越来越慢

function draw() {

dx = x\_e – x ;

dy = y\_e – y ;

v\_x = dx \* EASE\_VALUE ;

v\_y = dy \* EASE\_VALUE ;

x += v\_x ;

y += v\_y ;

}

缓冲开始：当动画缓冲开始时，飞机慢慢启动，但是会越来越快。

function draw(){

v\_x = v\_x + v\_x\*EASE\_VALUE ;

v\_y = v\_y + v\_y\*EASE\_VALUE ;

x += v\_x ;

y += v\_y ;

}

物理模型引擎，如three.js, Box2D，3D touch等允许开发都创建一个带有物理属性的世界，如物体具有形状，具有物理属性（密度，摩擦系数和恢复属性）…

### HTML5视频和音频

#### 声明式：

<video id=”theVideo” src=”\*.mp4” autoplay loop volumn …/>

//显示进度

var videoElement = document.getElementById(“theVideo”) ;

videoElement.addEventListener(‘progress’, new function() {

var percentLoaded = videoElement.buffered.end(0)/videoElement.duration \* 100 ;

…

}) ;

#### 代码方式：

var videoDiv = document.createElement(‘div’) ;

document.body.appendChild(videoDiv) ;

var videoElement = document.createElement(“video”) ;

videoDiv.appendChild(videoElement) ;

videoDiv.setAttribute(“style”, “display:none”) ;

videoElement.play() ;

function gameLoop() {

window.setTimeout(gameLoop, 20) ;

drawScreen() ;

}

gameLoop() ;

var context = document.getElementById(‘canvasOne’) .getContext(‘2d’) ;

function drawScreen() {

context.drawImage(videoElement, 0, 0) ;

}

视频处理： 在播放时触发一些事件，在视频上应用形状变换，…

function drawScreen() {

//处理视频图像

context.save();

context.setTransform(1, 0, 0, 1, 0, 0) ;

context.translate, rotate, ….

context.drawImage(videoElement, …) ;

context.restore() ;

//添加内容

If (videoElement.currentTime > message.time) {

context.fillText(message.content, message.x, message.y) ;

}

}

#### Javascript录制视频

接入摄像头，麦克风，将捕获的数据显示在canvas上

navigator.getUserMedia({video: true, audio:true}, mediaSuccess, mediaFail) ;

function mediaSuccess(userMedia) {

videoElement.src = window.URL.createObjectURL(userMedia) ;

}

其他同上

音频内容同视频，也有声明式和代码式

# WebGL

## WebGL编程指南

### 基本概念

JavaScript，接管用户界面、网络通信、多媒体、数据存储，三维图形渲染

HTML -> JavaScript -> canvas -> WebGL

WebGL根植于OpenGL,但它实际上是从OpenGL的一个特殊版本OpenGL ES中派生出来的。后来专用于嵌入式计算机、智能手机、家用游戏机设备。

WebGL页面包含了三种语言：HTML5，JavaScript, GLSL ES

<canvas>标签定义了网页上的绘图区域。<canvas>提供一些简单的绘图函数，用来绘制点、线、矩形、圆等。默认情况下，<canvas>是透明的，如果不用JavaScript在上面画些什么，你是看不到<canvas>的。<canvas>同时支持二维图形和三维图形

WebGL依赖于一种新的称为着色器(shader)的绘图机制。

着色器程序是以字符串的形式嵌入在JavaScript文件中。

WebGL程序的执行流程

1. 获取<canvas>元素
2. 获取WebGL绘图上下文
3. 初始化着色器
4. 设置<canvas>背景色
5. 清除<canvas>
6. 绘图

### JavaScript程序与着色器交互

attribute变量: GLSL ES变量，传输数据从JS传向顶点着色器

uniform变量：传输那些对于所有顶点都相同的数据，从JS传向片元着色器(顶点首色器也可以)

var VSHADER\_SOURCE =

‘ attribute vec4 a\_Position;\n’ +

‘ void main() {\n’ +

‘ gl\_Position = a\_Position;\n’ +

‘ gl\_PositionSize = 10.0;\n’ +

‘}\n’ ;

var FSHADER\_SOURCE =

‘precision medium float;\n’ +

‘uniform vec4 u\_FragColor;\n’ +

‘void main(){\n’ +

‘ gl\_FragColor = u\_FragColor;\n’ +

‘}\n’;

function main() {

var canvas = document.getElementById(‘webgl’) ;

var gl = getWebGLContext(canvas) ;

initShaders(gl, VSHADER\_SOURCE, FSHADER\_SOURCE) ;

var a\_Position = gl.getAttribLocation(gl.program, ‘a\_Position’) ;

gl.vertexAttrib3f(a\_Position, 0.0, 0.0, 0.0) ;

gl.uniform4f(u\_FragColor, 1.0, 0.0, 0.0, 1.0) ;

gl.clearColor(0.0, 0.0, 0.0, 1.0) ;

gl.clear(gl.COLOR\_BUFFER\_BIT) ;

gl.drawArrays(gl.POINTS, 0, 1) ;

}

### 绘图

构成三维模型的基本单位是三角形。不管三维模型的形状多么复杂，其基本组成部分都是三角形，只不过复杂的模型由更多的三角形构成而已。通过创建更细小和更大量的三角形，就可以创建更复杂和更逼真的三维模型。

从球体到立方体，再到游戏中的三维角色，都可以由小的三角形组成。实际上你可以这些最基本的图形来绘制出任何东西

JavaScript通过缓冲区对象向顶点着色器传入多个顶点的数据。

缓冲区对象：WebGL系统中的一块内存区域，我们可以一次性地向缓冲区对象中填充大量的顶点数据，然后将这些数据保存在其中，供顶点着色器使用。

var n = initVertexBuffers(gl) ;

gl.drawArrays(gl.POINTS, 0, n) ; //绘制多个点 或

gl.drawArrays(gl.LINES, 0, n); //绘制多条线

gl.drawArrays(gl.LINE\_STRIP, 0, n);

gl.drawArrays(gl.LINE\_LOOP, 0, n) ;

gl.drawArrays(gl.TRIANGLES, 0, n) ; //绘制三角形

gl.drawArrays(gl.TRIANGLE\_STRIP, 0, n) ;

function initVertexBuffers(gl) {

var vertices = new Float32Array([0.0, 0.5, -0.5, -0.5, 0.5,-0.5]);

var n = 3 ;

var vertexBuffer = gl.createBuffer(); //创建缓冲区对象

gl.bindBuffer(gl.ARRAY\_BUFFER, vertexBuffer) ; //将缓冲区绑定到目标

gl.bufferData(gl.ARRAY\_BUFFER, vertices, gl.STATIC\_DRAW); //向缓冲区对象中写入数据

//将缓冲区对象分配给a\_Position，并连接a\_Position变量与分配给它的缓冲区对象

var a\_Position = gl.getAttribLocation(gl.program, ‘a\_Position’) ;

gl.vertexAttribPointer(a\_Position, 2, gl.FLOAT, false, 0, 0); }

gl.enableVertexAttribArray(a\_Position) ;

return n;

}

JavaScript中通用的数组Array是一种通用的类型，即可以在里面存储数字也可以存储字符串，而并没有对“大量元素都是同一种类型”这种情况（比如vertices）进行优化。为了解决这个问题，WebGL引入了类型化数组，Float32Array就是其中之一。

Int8Array, UInt8Array, Int16Array, UInt16Array, Int32Array, UInt32Array, Float32Array, Float64Array

变换矩阵在三维计算机图形学中应用得如此广泛，以致于着色器本身就实现了矩阵和矢量相乘的功能。

Var VSHADER\_SOURCE =

‘attribute vec4 a\_Position;\n’ +

‘uniform mat4 u\_xformMatrix;\n’ +

‘void main() {\n’ +

‘ gl\_Position = u\_xformMatrix \* a\_Position;\n’ +

‘}\n’ ;

//注意WebGL中矩阵是列主序的

var xformMatrix = new Float32Array([

cosB, sinB, 0.0, 0.0,

-sinB, cosB, 0.0, 0.0,

0.0, 0.0, 1.0, 0.0,

0.0, 0.0, 0.0, 1.0

]);

var u\_xformMatrix = gl.getUniformLocation(gl.program, ‘u\_xformMatrix’) ;

gl.uniformMatrix4fv(u\_xformMatrix, false, xformMatrix) ;

动画

大多数WebGL开发都使用矩阵操作函数库来隐藏矩阵计算的细节，简化与矩阵有关的操作。还记得图形管线：顶点处理器，模型视图投影变换

当你按照顺序快速连续地看到这些图的时候，你的大脑就会不自觉地对影像进行插值，形成流畅的动画。

gl.clearColor(0.0, 0.0, 0.0, 1.0);

var currentAngle = 0.0 ;

var tick = function() {

currentAngle = animate(currentAngle) ; //更新旋转角

draw(currentAngle, …);

requestAnimationFrame(tick) ; //请求浏览器调用tick

}

tick();

//requestAnimationFrame()只是请求浏览器在适当的时机调用参数函数，那么浏览器调用tick()函数的间隔不恒定，那么每次调用时简单地向currentAngle加上一个固定的角度值（度/秒）就会导致不可控制的加速或减速的旋转效果。因此算法：根据本次调用与上次调用之间的时间间隔来决定这一帧的旋转角度比上一帧大出多少。

var g\_last = Date.now();

function animate(angle) {

var now = Date.now();

var elapsed = now – g\_last ;

g\_last = now ;

var newAngle = angle + (ANGLE\_STEP \* elapsed) / 1000.0 ;

return newAngle %= 360 ;

}

requestAnimationFrame(func)与setInterval()区别？

现代的浏览器都支持多个标签页，每个标签页具有单独的JavaScript运行环境，但是自setInterval()函数诞生之初，浏览器还没有开始支持多标签页。所以在现代浏览器中，不管标签页是否被激活，其中的setInterval()函数都会反复调用func,如果标签页比较多，就会增加浏览器的负荷，所以后来，浏览器又引入了requestAnimation()方法，该方法只有当标签页处于激活状态时才会生效。

requestAnimationFrame()更像setTimeOut(), 而不是setInterval()

颜色与纹理

通过为顶点的每种数据建立一个缓冲区，然后分配给对应的attribute变量，你就可以向顶点着色器传递多份逐顶点的数据信息了，如顶上坐标、顶点大小、顶点颜色、顶点纹理坐标、顶点所在平面法向量等

WebGL允许我们把顶点的坐标和尺寸数据打包到同一个缓冲区对象中，并通过某种机制分别访问缓冲区对象中不同种类的数据。比如，可以将顶点的坐标和尺寸数据按照如下方式交错组织(interleaving)，实际上3D建模工具会自动生成数据，所以没有必要手动输入数据，或者显示地检查数据的完整性。

Var verticesSizes = new Float32Array([

//顶点坐标和点的尺寸

0.0, 0.5, 10.0,

-0.5, -0.5, 20.0,

0.5, -0.5, 30.0

]);

var FSIZE = verticesSizes.BYTES\_PER\_SELEMENT;

gl.vertexAttribPointer(a\_Position, 2, gl.FLOAT, false, FSIZE\*3, 0);

gl.vertexAttribPointer(a\_PointSize, 1, gl.FLOAT, false, FSIZE\*3, FSIZE\*2);

顶点着色器会根据stride, offset从缓冲区正确抽取数据，给attribute变量并进行绘制

varying变量：从顶点着色器向片元着色器传输数据。

var VSHADER\_SOURCE =

’attribute vec4 a\_Position;\n’ +

‘attribute vec4 a\_Color;\n’ +

‘varying vec4 v\_Color;\n’ +

‘void main() {\n’ +

‘gl\_Position = a\_Position;\n’ +

‘gl\_PointSize = 10.0;\n’ +

‘v\_Color = a\_Color;\n’ + //将数据传给片元着色器

‘}\n’ ;

var FSHADER\_SOURCE =

varying vec4 v\_Color ;

void main() {

gl\_FragColor = v\_Color ; //内插得到的颜色

}

function initVertexBuffers(gl) {

var verticesColors = new Float32Array([

0.0, 0.5, 1.0, 0.0, 0.0,

-0.5, -0.5, 0.0, 1.0, 0.0,

0.5, -0.5, 0.0, 0.0, 1.0

]);

var n = 3 ;

var vertexColorBuffer = gl.createBuffer() ;

gl.bindBuffer(gl.ARRAY\_BUFFER, vertexColorBuffer) ;

gl.bufferData(gl.ARRAY\_BUFFER, vertexColors, gl.STATIC\_DRAW) ;

var FSIZE = verticesColors.BYTES\_PER\_ELEMENT;

var a\_Position = gl.getAttribLocation(gl.program, ‘a\_Position’) ;

gl.vertexAttribPointer(a\_Position, 2, gl.FLOAT, false, FSIZE\*5.0) ;

gl.enableVertexAttribArray(a\_Position) ;

var a\_Color = gl.getAttribLocation(gl.program, ‘a\_Color’) ;

gl.vertexAttribPointer(a\_Color, 3, gl.FLOAT, false, FSIZE\*5, FSIZE\*2);

gl.enableVertexAttribArray(a\_Color) ;

return n;

}

注意varying变量只能是float(以及相关的vec2, vec3, vec4, mat2, mat3, mat4)类型

在WebGL中，如果顶点着色器与片元着色器中有类型和命名都相同的varying变量，那么顶点着色器赋给该变量的值就会被自动地传入片元着色器

图形管线：顶点坐标 -> 执行顶点着色器 -> 图形装配 -> 光栅化 -> 执行片元着色器

图形装配：将孤立的顶点坐标装配成几何图形。几何图形的类别由gl.drawArrays()函数的第一个参数决定。

光栅化过程：这一步的任务时将装配好的几何图形转化为片元。

一旦光栅化过程结束后，程序就开始逐片元调用处元着色器，对于每个片元，片元着色器计算出该片元的颜色，并写入颜色缓冲区。

光栅化过程生成的片元都是带有坐标信息的。调用片元着色器时这些坐标信息也随着片元传了进去，我们可以通过片元着色器中的内置变量来访问片元的坐标gl\_FragCoord.x, …

顶点着色器中的v\_Color变量在传入片元着色器之前经过了内插过程。

纹理映射：将一张图像映射到一个几何图形的表面上去。这张图片又可以称为纹理图像

纹理映射的作用，就是根据纹理图像，为之前光栅化后的每个片元涂上合适的颜色，组成纹理图像的像素又被称为纹素

在WebGL进行纹理映射步聚：

1. 准备好映射到几何图形上的纹理图像
2. 为几何图形配置纹理映射方式
3. 加载纹理图像，对其进行一些配置，以在WebGL中使用它
4. 在片元着色器中将相应的纹素从纹理中抽取出来，并将纹素的颜色赋给片元

映射方式：”几何图形的某个片元”的颜色如何取决于”纹理图像中哪个或哪几个像素”的问题。利用图形的顶点坐标来确定屏幕上哪部分被纹理图像覆盖，使用纹理坐标来确定纹理图像的哪部分将覆盖到几何图形上。

纹理坐标：笛卡尔坐标系，范围为(0, 0) -> (1, 1)

首先在顶点着色器中为每个顶点指定纹理坐标，然后在片元着色器中根据每个片元的纹理坐标从纹理图像中抽取纹素颜色

var VSHADER\_SOURCE =

attribute vec4 a\_Position;

attribute vec2 a\_TexCoord;

varying vec2 v\_TexCoord ;

void main() {

gl\_Position = a\_Position;

v\_TexCoord = a\_TexCoord ;

}

var FSHADER\_SOURCE =

uniform sampler2D u\_Sampler;

varying vec2 v\_TexCoord ;

void main() {

gl\_FragColor = texture2D(u\_Sampler, v\_TexCoord) ;

}

function main() {

var n = initVertexBuffers(gl) ;

initTextures(gl, n) ;

}

function initVertexBuffers(gl) {

var verticesTexCoords = new Float32Array([

//顶点坐标，纹理坐标

]);

}

Typescript

Basic Types

let isDone: boolean = false;

let decimal: number = 6;//As in JavaScript, all numbers in TypeScript are floating point values

let color: string = "blue";

let list: number[] = [1, 2, 3]; //Array

let x: [string, number]; //tuple

// Initialize it

x = ["hello", 10];

enum Color {Red, Green, Blue} //Enum

let c: Color = Color.Green;

let notSure: any = 4; //Any 动态类型

notSure.ifItExists(); // okay, ifItExists might exist at runtime

notSure.toFixed(); // okay, toFixed exists (but the compiler doesn't check)

let strLength: number = (someValue as string).length; 类型转换

let [first, second] = [1, 2]; 解包

let [first, ...rest] = [1, 2, 3, 4];

let [first] = [1, 2, 3, 4];

let [, second, , fourth] = [1, 2, 3, 4];

let { a, b } = { 解对象

a: "foo",

b: 12,

c: "bar"

};

let first = [1, 2];

let second = [3, 4];

let bothPlus = [0, ...first, ...second, 5]; //Spread

let defaults = { food: "spicy", price: "$$", ambiance: "noisy" };

let search = { ...defaults, food: "rich" };

let block-scoping

const numLivesForCat = 9;

function greeter(person: string) {

return "Hello, " + person;

}

var user = "Jane User";

document.body.innerHTML = greeter(user);

Interfaces

interface Person {

firstName: string;

lastName: string;

}

function greeter(person: Person) {

return "Hello, " + person.firstName + " " + person.lastName;

}

var user = { firstName: "Jane", lastName: "User" };

document.body.innerHTML = greeter(user);

Classes

class Student {

fullName: string;

constructor(public firstName, public middleInitial, public lastName) {

this.fullName = firstName + " " + middleInitial + " " + lastName;

}

}

interface Person {

firstName: string;

lastName: string;

}

function greeter(person : Person) {

return "Hello, " + person.firstName + " " + person.lastName;

}

var user = new Student("Jane", "M.", "User");

document.body.innerHTML = greeter(user);