# http

## 1、与https区别

HTTP协议传输的数据都是未加密的，也就是明文的，因此使用HTTP协议传输隐私信息非常不安全，为了保证这些隐私数据能加密传输，于是网景公司设计了SSL（Secure Sockets Layer）协议用于对HTTP协议传输的数据进行加密，从而就诞生了HTTPS。简单来说，HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全。

　　HTTPS和HTTP的区别主要如下：

　　1、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。

　　2、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。

　　3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

4、http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

## 2、http请求

http请求报包含三个部分：

请求行 + 请求头 + 数据体

### 1、请求行

请求行包含三个内容 method + request-URI + http-version

method 包含有 post , get, head,delete, put, connect, options, patch, propfind, propatch, mkcol, copy, move, lock, unlock, trace, head

### 2、请求头

请求头包含：

Accept：指浏览器或其他客户可以接爱的MIME文件格式。Servlet可以根据它判断并返回适当的文件格式。

User-Agent：是客户浏览器名称

Host：对应网址URL中的Web名称和端口号。

Accept-Langeuage：指出浏览器可以接受的语言种类，如en或en-us，指英语。

connection：用来告诉服务器是否可以维持固定的HTTP连接。http是无连接的，HTTP/1.1使用Keep-Alive为默认值，这样，当浏览器需要多个文件时(比如一个HTML文件和相关的图形文件)，不需要每次都建立连接

Cookie：浏览器用这个属性向服务器发送Cookie。Cookie是在浏览器中寄存的小型数据体，它可以记载和服务器相关的用户信息，也可以用来实现会话功能。

Referer：表明产生请求的网页URL。如比从网页/icconcept/index.jsp中点击一个链接到网页/icwork/search，在向服务器发送的GET/icwork/search中的请求中，Referer是http://hostname:8080/icconcept/index.jsp。这个属性可以用来跟踪Web请求是从什么网站来的。

Content-Type：用来表名request的内容类型。可以用HttpServletRequest的getContentType()方法取得。

Accept-Charset：指出浏览器可以接受的字符编码。英文浏览器的默认值是ISO-8859-1.

Accept-Encoding：指出浏览器可以接受的编码方式。编码方式不同于文件格式，它是为了压缩文件并加速文件传递速度。浏览器在接收到Web响应之后先解码，然后再检查文件格式。

### 3、数据体

请求所携带的数据

## 3、http响应

http响应由三个部分组成分别是状态行，响应头，响应正文。

### 1、状态行

状态行是由：HTTP-Version+Status-Code+Reason-Phrase

比如：HTTP/1.1 200 ok

分别表示http版本 + 状态码 + 状态代码的文本描述

### 2、响应头

响应头：包含服务器类型，日期，长度，内容类型等

### 3、响应正文

响应正文响应正文就是服务器返回的HTML页面

## 4、url请求的过程

1、地址栏输入url

2、发送请求到本地网络配置的dns服务器，查找url对应的IP

域名解析过程：

* Chrome浏览器首先检查自己本地是缓存是否有对应的域名，有则直接使用。【查看Chrome浏览器dns缓存地址：chrome://net-internals/#dns】
* 如果浏览器缓存中没有，则查询系统DNS缓存中的域名表，有则直接使用。【windows查看域名表的命令：ipconfig /displaydns】
* 系统缓存中还是没有，则检查hosts文件中的映射表。【windows中hosts文件路径：C:\Windows\System32\drivers\etc】
* 本地实在找不到，则向DNS域名服务器发起请求查询。

3、根据IP建立连接（TCP三次握手）

4、HTTP发起请求

5、服务器处理请求，浏览器接收HTTP响应

6、渲染页面

7、关闭连接（TCP四次握手）

## 5、https加密过程

1. 客户端发起HTTPS请求

这个没什么好说的，就是用户在浏览器里输入一个HTTPS网址，然后连接到服务端的443端口。

1. 服务端的配置

采用HTTPS协议的服务器必须要有一套数字证书，可以自己制作，也可以向组织申请。区别就是自己颁发的证书需要客户端验证通过，才可以继续访问，而使用受信任的公司申请的证书则不会弹出提示页面。这套证书其实就是一对公钥和私钥。如果对公钥不太理解，可以想象成一把钥匙和一个锁头，只是世界上只有你一个人有这把钥匙，你可以把锁头给别人，别人可以用这个锁把重要的东西锁起来，然后发给你，因为只有你一个人有这把钥匙，所以只有你才能看到被这把锁锁起来的东西。

1. 传送证书

这个证书其实就是公钥，只是包含了很多信息，如证书的颁发机构，过期时间等等。

1. 客户端解析证书

这部分工作是由客户端的SSL/TLS来完成的，首先会验证公钥是否有效，比如颁发机构，过期时间等等，如果发现异常，则会弹出一个警示框，提示证书存在的问题。如果证书没有问题，那么就生成一个随机值。然后用证书（也就是公钥）对这个随机值进行加密。就好像上面说的，把随机值用锁头锁起来，这样除非有钥匙，不然看不到被锁住的内容。

1. 传送加密信息

这部分传送的是用证书加密后的随机值，目的是让服务端得到这个随机值，以后客户端和服务端的通信就可以通过这个随机值来进行加密解密了。

1. 服务端解密信息

服务端用私钥解密后，得到了客户端传过来的随机值，然后把内容通过该随机值进行对称加密，将信息和私钥通过某种算法混合在一起，这样除非知道私钥，不然无法获取内容，而正好客户端和服务端都知道这个私钥，所以只要加密算法够彪悍，私钥够复杂，数据就够安全。

1. 传输加密后的信息

这部分信息就是服务端用私钥加密后的信息，可以在客户端用随机值解密还原。

1. 客户端解密信息

客户端用之前生产的私钥解密服务端传过来的信息，于是获取了解密后的内容。整个过程第三方即使监听到了数据，也束手无策。

# XSS、CSRF攻击和防御

xss跨站脚本攻击(Cross Site Scripting)，是一种经常出现在web应用中的计算机安全漏洞，指攻击者在网页中嵌入客户端脚本(例如JavaScript), 当用户浏览此网页时，脚本就会在用户的浏览器上执行，从而达到攻击者的目的。比如获取用户的Cookie，导航到恶意网站,携带木马等。

## xss攻击类型

　　1.非持久型XSS攻击

非持久型XSS漏洞实际上大多数攻击数据是包含在URL中的。需要用户的浏览器访问到这个URL恶意代码才执行，攻击者一般会把URL发给用户让用户通过浏览器去访问。不过URL里面带有稀奇古怪的代码确实有点奇怪，为了掩人耳目，攻击者可以发一个看起来没问题的URL，再通过那个页面跳转到恶意的URL；甚至也可以让一个域名转向到恶意URL，把那个域名发给用户。

2.持久型XSS攻击

持久型XSS（Persistent）又叫做存储XSS（Stored XSS），与非持久型XSS相反，它是指通过提交恶意数据到存储器（比如数据库、文本文件等），Web应用程序输出的时候是从存储器中读出恶意数据输出到页面的一类跨站脚本漏洞。持久型XSS漏洞是把恶意脚本存储到了数据库，访问页面的时候完全没有预兆，所以它的危害也比非持久型XSS略微高一点。

## XSS攻击防御

原则：不相信客户输入的数据  
注意:  攻击代码不一定在<script></script>中

1.使用XSS Filter。

输入过滤，对用户提交的数据进行有效性验证，仅接受指定长度范围内并符合我们期望格式的的内容提交，阻止或者忽略除此外的其他任何数据。输出编码，当需要将一个字符串输出到Web网页时，同时又不确定这个字符串中是否包括XSS特殊字符（如< > &‘”等），为了确保输出内容的完整性和正确性，可以使用编码（HTMLEncode）进行处理。

2.DOM型的XSS攻击防御

把变量输出到页面时要做好相关的编码转义工作，如要输出到 <script>中，可以进行JS编码；要输出到HTML内容或属性，则进行HTML编码处理。根据不同的语境采用不同的编码处理方式。

3.HttpOnly Cookie

将重要的cookie标记为http only,   这样的话当浏览器向Web服务器发起请求的时就会带上cookie字段，但是在脚本中却不能访问这个cookie，这样就避免了XSS攻击利用JavaScript的document.cookie获取cookie。

## CSRF

CSRF（Cross-site request forgery），中文名称：跨站请求伪造。CSRF通过伪装来自受信任用户的请求来利用受信任的网站。

要完成一次CSRF攻击，受害者必须依次完成两个步骤：

　　1.登录受信任网站A，并在本地生成Cookie。

　　2.在不登出A的情况下，访问危险网站B。

## CSRF的防御

### 服务端进行CSRF防御

服务端的CSRF方式方法很多样，但总的思想都是一致的，就是在客户端页面增加伪随机数。

(1).Cookie Hashing(所有表单都包含同一个伪随机值)：这可能是最简单的解决方案了，因为攻击者不能获得第三方的Cookie(理论上)，所以表单中的数据也就构造失败了，然后在服务器端进行Hash值验证

(2).验证码：这个方案的思路是：每次的用户提交都需要用户在表单中填写一个图片上的随机字符串，厄....这个方案可以完全解决CSRF

(3). 验证 HTTP Referer 字段

# js css html加载顺序

正常的网页加载流程是这样的。

1、浏览器一边下载HTML网页，一边开始解析

2、解析过程中，发现<script>标签

3、暂停解析，网页渲染的控制权转交给JavaScript引擎

4、如果<script>标签引用了外部脚本，就下载该脚本，否则就直接执行

5、执行完毕，控制权交还渲染引擎，恢复往下解析HTML网页

# JavaScript知识

## 1、数据类型

**字符串、数字、布尔、数组、对象、Null、Undefined**

### 1、基本数据类型和引用数据类型

　　ECMAScript包括两个不同类型的值：基本数据类型和引用数据类型。

　　基本数据类型指的是简单的数据段，引用数据类型指的是有多个值构成的对象。

　　当我们把变量赋值给一个变量时，解析器首先要确认的就是这个值是基本类型值还是引用类型值。

### 2、常见的基本数据类型：

Number、String 、Boolean、Null和Undefined。基本数据类型是按值访问的，因为可以直接操作保存在变量中的实际值。

### 3、引用类型数据：

也就是对象类型Object type，比如：Object 、Array 、Function 、Data等。

javascript的引用数据类型是保存在堆内存中的对象。

与其他语言的不同是，你不可以直接访问堆内存空间中的位置和操作堆内存空间。只能操作对象在栈内存中的引用地址。

所以，引用类型数据在栈内存中保存的实际上是对象在堆内存中的引用地址。通过这个引用地址可以快速查找到保存中堆内存中的对象。

### 4、总结区别

#### a 声明变量时不同的内存分配：

　　1）原始值：存储在栈（stack）中的简单数据段，也就是说，它们的值直接存储在变量访问的位置。

这是因为这些原始类型占据的空间是固定的，所以可将他们存储在较小的内存区域 – 栈中。这样存储便于迅速查寻变量的值。

　　2）引用值：存储在堆（heap）中的对象，也就是说，存储在变量处的值是一个指针（point），指向存储对象的内存地址。

　　这是因为：引用值的大小会改变，所以不能把它放在栈中，否则会降低变量查寻的速度。相反，放在变量的栈空间中的值是该对象存储在堆中的地址。

　　地址的大小是固定的，所以把它存储在栈中对变量性能无任何负面影响。

#### b 不同的内存分配机制也带来了不同的访问机制

　　1）在javascript中是不允许直接访问保存在堆内存中的对象的，所以在访问一个对象时，首先得到的是这个对象在堆内存中的地址，然后再按照这个地址去获得这个对象中的值，这就是传说中的按引用访问。

　　2）而原始类型的值则是可以直接访问到的。

#### c 复制变量时的不同

　　1）原始值：在将一个保存着原始值的变量复制给另一个变量时，会将原始值的副本赋值给新变量，此后这两个变量是完全独立的，他们只是拥有相同的value而已。

　　2）引用值：在将一个保存着对象内存地址的变量复制给另一个变量时，会把这个内存地址赋值给新变量，也就是说这两个变量都指向了堆内存中的同一个对象，他们中任何一个作出的改变都会反映在另一个身上。（这里要理解的一点就是，复制对象时并不会在堆内存中新生成一个一模一样的对象，只是多了一个保存指向这个对象指针的变量罢了）。多了一个指针

#### d 参数传递的不同（把实参复制给形参的过程）

　　首先我们应该明确一点：ECMAScript中所有函数的参数都是按值来传递的。

　　但是为什么涉及到原始类型与引用类型的值时仍然有区别呢？还不就是因为内存分配时的差别。

　　1）原始值：只是把变量里的值传递给参数，之后参数和这个变量互不影响。

　　2）引用值：对象变量它里面的值是这个对象在堆内存中的内存地址，这一点你要时刻铭记在心！

因此它传递的值也就是这个内存地址，这也就是为什么函数内部对这个参数的修改会体现在外部的原因了，因为它们都指向同一个对象。

## 2、JavaScript 作用域

### 1. 全局作用域（Global Scope）

在代码中任何地方都能访问到的对象拥有全局作用域，一般来说以下几种情形拥有全局作用域：

（1）最外层函数和在最外层函数外面定义的变量拥有全局作用域

（2）所有末定义直接赋值的变量自动声明为拥有全局作用域

（3）所有window对象的属性拥有全局作用域

### 2、局部作用域（Local Scope）

和全局作用域相反，局部作用域一般只在固定的代码片段内可访问到，最常见的例如函数内部，所以在一些地方也会看到有人把这种作用域称为函数作用域

### 作用域链

在JavaScript中，函数也是对象，实际上，JavaScript里一切都是对象。函数对象和其它对象一样，拥有可以通过代码访问的属性和一系列仅供JavaScript引擎访问的内部属性。其中一个内部属性是[[Scope]]，由ECMA-262标准第三版定义，该内部属性包含了函数被创建的作用域中对象的集合，这个集合被称为函数的作用域链，它决定了哪些数据能被函数访问。

### 作用域链和代码优化

从作用域链的结构可以看出，在运行期上下文的作用域链中，标识符所在的位置越深，读写速度就会越慢。如上图所示，因为全局变量总是存在于运行期上下文作用域链的最末端，因此在标识符解析的时候，查找全局变量是最慢的。所以，在编写代码的时候应尽量少使用全局变量，尽可能使用局部变量。一个好的经验法则是：如果一个跨作用域的对象被引用了一次以上，则先把它存储到局部变量里再使用。

## 3、JavaScript事件流和事件代理/委托

当你点击某个元素时，产生了一个点击事件，从而产生了一个事件流。

### 事件流分为两种：

事件冒泡：事件从最具体的节点开始向外传播到最宽泛的节点。这是事件流的默认类型，被绝大多数浏览器所支持。

事件捕获：事件从最宽泛的节点开始向内传播到最具体的节点。这种方式在IE8和更早版本的IE中不被支持。

### 事件代理/委托

大量的元素创建事件监听器会造成页面速度下降，事件流允许你在父元素上监听事件。事件可以影响到容器元素（祖先元素），因此可以将事件处理程序放置在一个元素容器上，然后使用事件对象的target属性找到它的后代中是哪一个发生了事件，因此只需要响应一个元素上的事件（而不是在每一个子元素上分别响应事件）。

## 4、跨域

同源政策：

A网页设置的 Cookie，B网页不能打开，除非这两个网页"同源"。所谓"同源"指的是"三个相同"。协议相同、域名相同、端口相同

同源政策的目的，是为了保证用户信息的安全，防止恶意的网站窃取数据。

提交表单不受同源政策的限制：直接使用action的时候，是直接把请求交给了action里面的域，本身页面不会去管他的请求结果，后面的步骤交给了action里面的域。

限制范围：

* Cookie、LocalStorage 和 IndexDB 无法读取。
* DOM 无法获得。
* AJAX 请求不能发送。

### 1、什么是跨域

js跨域是指通过js在不同的域之间进行数据传输或通信。简单地理解就是因为JavaScript同源策略的限制，a.com 域名下的js无法操作b.com或是c.a.com域名下的对象。

### 2、跨域的实现

#### cookie：

Cookie 是服务器写入浏览器的一小段信息，只有同源的网页才能共享。但是，两个网页一级域名相同，只是二级域名不同，浏览器允许通过设置document.domain共享 Cookie。

注意，这种方法只适用于 Cookie 和 iframe 窗口，LocalStorage 和 IndexDB 无法通过这种方法，规避同源政策。

#### iframe：

如果两个网页不同源，就无法拿到对方的DOM。典型的例子是iframe窗口和window.open方法打开的窗口，它们与父窗口无法通信。

对于完全不同源的网站，目前有三种方法，可以解决跨域窗口的通信问题。

* 片段识别符（fragment identifier）：片段标识符（fragment identifier）指的是，URL的#号后面的部分，比如http://example.com/x.html#fragment的#fragment。如果只是改变片段标识符，页面不会重新刷新。父窗口可以把信息，写入子窗口的片段标识符。子窗口通过监听hashchange事件得到通知。
* window.name：浏览器窗口有window.name属性。这个属性的最大特点是，无论是否同源，只要在同一个窗口里，前一个网页设置了这个属性，后一个网页可以读取它。
* 跨文档通信API（Cross-document messaging）——window.postMessage：这个API为window对象新增了一个window.postMessage方法，允许跨窗口通信，不论这两个窗口是否同源。postMessage方法的第一个参数是具体的信息内容，第二个参数是接收消息的窗口的源（origin），即"协议 + 域名 + 端口"。也可以设为\*，表示不限制域名，向所有窗口发送。窗口和子窗口都可以通过message事件，监听对方的消息。

#### AJAX

同源政策规定，AJAX请求只能发给同源的网址，否则就报错。除了架设服务器代理（浏览器请求同源服务器，再由后者请求外部服务），有三种方法规避这个限制。

* JSONP：

JSONP是服务器与客户端跨源通信的常用方法。最大特点就是简单适用，老式浏览器全部支持，服务器改造非常小。它的基本思想是，网页通过添加一个<script>元素，向服务器请求JSON数据，这种做法不受同源政策限制；服务器收到请求后，将数据放在一个指定名字的回调函数里传回来。

* WebSocket：

WebSocket是一种通信协议，使用ws://（非加密）和wss://（加密）作为协议前缀。该协议不实行同源政策，只要服务器支持，就可以通过它进行跨源通信。浏览器发出的WebSocket请求的头信息有一个字段是Origin，表示该请求的请求源（origin），即发自哪个域名。正是因为有了Origin这个字段，所以WebSocket才没有实行同源政策。因为服务器可以根据这个字段，判断是否许可本次通信。如果该域名在白名单内，服务器就会做出如下回应。

* CORS：

CORS是跨源资源分享（Cross-Origin Resource Sharing）的缩写。

CORS需要浏览器和服务器同时支持。目前，所有浏览器都支持该功能，IE浏览器不能低于IE10。浏览器一旦发现AJAX请求跨源，就会自动添加一些附加的头信息，有时还会多出一次附加的请求，但用户不会有感觉。

浏览器将CORS请求分成两类：简单请求（simple request）和非简单请求（not-so-simple request）。

只要同时满足以下两大条件，就属于简单请求。

（1) 请求方法是以下三种方法之一：

* HEAD
* GET
* POST

（2）HTTP的头信息不超出以下几种字段：

* Accept
* Accept-Language
* Content-Language
* Last-Event-ID
* Content-Type：只限于三个值application/x-www-form-urlencoded、multipart/form-data、text/plain

简单请求：

对于简单请求，浏览器直接发出CORS请求。具体来说，就是在头信息之中，增加一个Origin字段。如果Origin指定的源，不在许可范围内，服务器会返回一个正常的HTTP回应。浏览器发现，这个回应的头信息没有包含Access-Control-Allow-Origin字段（详见下文），就知道出错了，从而抛出一个错误，被XMLHttpRequest的onerror回调函数捕获。注意，这种错误无法通过状态码识别，因为HTTP回应的状态码有可能是200。

如果Origin指定的域名在许可范围内，服务器返回的响应，会多出几个头信息字段。

* Access-Control-Allow-Origin：该字段是必须的。它的值要么是请求时Origin字段的值，要么是一个\*，表示接受任意域名的请求。
* Access-Control-Allow-Credentials：字段可选。它的值是一个布尔值，表示是否允许发送Cookie。
* Access-Control-Expose-Headers：该字段可选。CORS请求时，XMLHttpRequest对象的getResponseHeader()方法只能拿到6个基本字段：Cache-Control、Content-Language、Content-Type、Expires、Last-Modified、Pragma。如果想拿到其他字段，就必须在Access-Control-Expose-Headers里面指定。上面的例子指定，getResponseHeader('FooBar')可以返回FooBar字段的值。

非简单请求：

非简单请求是那种对服务器有特殊要求的请求，比如请求方法是PUT或DELETE，或者Content-Type字段的类型是application/json。非简单请求的CORS请求，会在正式通信之前，增加一次HTTP查询请求，称为"预检"请求（preflight）。浏览器先询问服务器，当前网页所在的域名是否在服务器的许可名单之中，以及可以使用哪些HTTP动词和头信息字段。只有得到肯定答复，浏览器才会发出正式的XMLHttpRequest请求，否则就报错。

"预检"请求用的请求方法是OPTIONS，表示这个请求是用来询问的。头信息里面，关键字段是Origin，表示请求来自哪个源。

1、对于端口和协议的不同前端无能为力，需要通过后台proxy来解决

2、jsonp

3、document.domain用于不同子域的框架间的交互，修改document.domain的方法只适用于不同子域的框架间的交互，不能进行ajax进行数据交互

4、利用location.hash+iframe跨域获取数据

5、使用HTML5中新引进的window.postMessage方法来跨域传送数据

6、CORS: 跨域资源共享(Cross-Origin Resource Sharing)

## 5、localStorage、sessionStorage、Cookie的区别及用法

### localStorage

localStorage生命周期是永久，这意味着除非用户显示在浏览器提供的UI上清除localStorage信息，否则这些信息将永远存在。存放数据大小为一般为5MB,而且它仅在客户端（即浏览器）中保存，不参与和服务器的通信。

### sessionStorage

sessionStorage仅在当前会话下有效，关闭页面或浏览器后被清除。存放数据大小为一般为5MB,而且它仅在客户端（即浏览器）中保存，不参与和服务器的通信。源生接口可以接受，亦可再次封装来对Object和Array有更好的支持。

作用域不同

不同浏览器无法共享localStorage或sessionStorage中的信息。相同浏览器的不同页面间可以共享相同的 localStorage（页面属于相同域名和端口），但是不同页面或标签页间无法共享sessionStorage的信息。这里需要注意的是，页面及标 签页仅指顶级窗口，如果一个标签页包含多个iframe标签且他们属于同源页面，那么他们之间是可以共享sessionStorage的。

### Cookie

生命期为只在设置的cookie过期时间之前一直有效，即使窗口或浏览器关闭。 存放数据大小为4K左右 。有个数限制（各浏览器不同），一般不能超过20个。与服务器端通信：每次都会携带在HTTP头中，如果使用cookie保存过多数据会带来性能问题。

cookie的优点：具有极高的扩展性和可用性

1.通过良好的编程，控制保存在cookie中的session对象的大小。

2.通过加密和安全传输技术，减少cookie被破解的可能性。

3.只有在cookie中存放不敏感的数据，即使被盗取也不会有很大的损失。

4.控制cookie的生命期，使之不会永远有效。这样的话偷盗者很可能拿到的就 是一个过期的cookie。

cookie的缺点：

1.cookie的长度和数量的限制。每个domain最多只能有20条cookie，每个cookie长度不能超过4KB。否则会被截掉。

2.安全性问题。如果cookie被人拦掉了，那个人就可以获取到所有session信息。加密的话也不起什么作用。

3.有些状态不可能保存在客户端。例如，为了防止重复提交表单，我们需要在服务端保存一个计数器。若吧计数器保存在客户端，则起不到什么作用。

localStorage、sessionStorage、Cookie共同点：都是保存在浏览器端，且同源的。

## js的原型链与继承

### 原型链

Javascript还提供了一个instanceof运算符，验证原型对象与实例对象之间的关系。

每个实例对象都有一个hasOwnProperty()方法，用来判断某一个属性到底是本地属性，还是继承自prototype对象的属性。

in运算符可以用来判断，某个实例是否含有某个属性，不管是不是本地属性。

原型prototype，在创建新函数的时候，会自动生成，而prototype中也会有一个constructor，回指创建该prototype的函数对象。

\_\_proto\_\_是对象或者实例中内置的[[prototype]]，其指向的是产生该对象的对象的prototype,在浏览器中提供了\_\_proto\_\_让我们可以访问，通过\_\_proto\_\_的指向形成的一个链条，就称做原型链，原型链的整个链路是：实例对象- ->构造函数的prototype- ->Object的prototype- ->null。

### 继承

function Animal(){

this.species = "动物";

}

function Cat(name,color){

this.name = name;

this.color = color;

}

#### prototype模式

如果"猫"的prototype对象，指向一个Animal的实例，那么所有"猫"的实例，就能继承Animal了。

Cat.prototype = new Animal();

Cat.prototype.constructor = Cat;

代码的第一行，我们将Cat的prototype对象指向一个Animal的实例。它相当于完全删除了prototype 对象原先的值，然后赋予一个新值。但是，第二行又是什么意思呢？原来，任何一个prototype对象都有一个constructor属性，指向它的构造函数。如果没有"Cat.prototype = new Animal();"这一行，Cat.prototype.constructor是指向Cat的；加了这一行以后，Cat.prototype.constructor指向Animal。更重要的是，每一个实例也有一个constructor属性，默认调用prototype对象的constructor属性。因此，在运行"Cat.prototype = new Animal();"这一行之后，cat1.constructor也指向Animal！这显然会导致继承链的紊乱（cat1明明是用构造函数Cat生成的），因此我们必须手动纠正，将Cat.prototype对象的constructor值改为Cat。这就是第二行的意思。

缺点：

* 要想为子类新增属性和方法，必须要在new Animal()这样的语句之后执行，不能放到构造器中；
* 无法实现多继承
* 来自原型对象的所有属性被所有实例共享
* 创建子类实例时，无法向父类构造函数传参

#### 构造函数绑定

第一种方法也是最简单的方法，使用call或apply方法，将父对象的构造函数绑定在子对象上，即在子对象构造函数中加一行：

function Cat(name,color){

　　　　Animal.apply(this, arguments);

　　　　this.name = name;

　　　　this.color = color;

}

特点：

* 解决了1中，子类实例共享父类引用属性的问题
* 创建子类实例时，可以向父类传递参数
* 可以实现多继承（call多个父类对象）

缺点：

* 实例并不是父类的实例，只是子类的实例
* 只能继承父类的实例属性和方法，不能继承原型属性/方法
* 无法实现函数复用，每个子类都有父类实例函数的副本，影响性能

#### 实例继承

核心：为父类实例添加新特性，作为子类实例返回

function Cat(name){

var instance = new Animal();

instance.name = name || 'Tom';

return instance;

}

特点：

* 不限制调用方式，不管是new 子类()还是子类(),返回的对象具有相同的效果

缺点：

* 实例是父类的实例，不是子类的实例
* 不支持多继承

#### 拷贝继承

function Cat(name){

var animal = new Animal();

for(var p in animal){

Cat.prototype[p] = animal[p];

}

Cat.prototype.name = name || 'Tom';

}

特点：

* 支持多继承

缺点：

* 效率较低，内存占用高（因为要拷贝父类的属性）
* 无法获取父类不可枚举的方法（不可枚举方法，不能使用for in 访问到）

#### 组合继承

核心：通过调用父类构造，继承父类的属性并保留传参的优点，然后通过将父类实例作为子类原型，实现函数复用

function Cat(name){

Animal.call(this);

this.name = name || 'Tom';

}

Cat.prototype = new Animal();

Cat.prototype.constructor = Cat;

特点：

* 弥补了方式2的缺陷，可以继承实例属性/方法，也可以继承原型属性/方法
* 既是子类的实例，也是父类的实例
* 不存在引用属性共享问题
* 可传参
* 函数可复用

缺点：

* 调用了两次父类构造函数，生成了两份实例（子类实例将子类原型上的那份屏蔽了）

#### 寄生组合继承

核心：通过寄生方式，砍掉父类的实例属性，这样，在调用两次父类的构造的时候，就不会初始化两次实例方法/属性，避免的组合继承的缺点

function inherit(superType, subType) {

function F() {}

F.prototype = superType.prototype;

var subPrototype = new F();

subPrototype.constructor = subType;

subType.prototype = subPrototype;

}

function Animal(name) {

this.name = name;

}

Animal.prototype.sayName = function() {

console.log(this.name);

}

function Cat(color) {

Animal.call(this, 'cat');

this.color = color;

}

inherit(Animal, Cat);

这样就只调用了一次父类的构造方法

## 7、闭包

闭包就是能够读取其他函数内部变量的函数。

由于在Javascript语言中，只有函数内部的子函数才能读取局部变量，因此可以把闭包简单理解成"定义在一个函数内部的函数"。

例：

function f1(){

var n=999;

function f2(){

alert(n);

}

return f2;

}

var result=f1();

result(); // 999

### 闭包的用途

1. setTimeout/setInterval
2. 回调函数（callback）
3. 事件句柄（event handle）

它的最大用处有两个，一个是前面提到的可以读取函数内部的变量，另一个就是让这些变量的值始终保持在内存中。

为什么会这样呢？原因就在于f1是f2的父函数，而f2被赋给了一个全局变量，这导致f2始终在内存中，而f2的存在依赖于f1，因此f1也始终在内存中，不会在调用结束后，被垃圾回收机制（garbage collection）回收。

### 使用闭包的注意点

1）由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包，否则会造成网页的性能问题，在IE中可能导致内存泄露。解决方法是，在退出函数之前，将不使用的局部变量全部删除。

2）闭包会在父函数外部，改变父函数内部变量的值。所以，如果你把父函数当作对象（object）使用，把闭包当作它的公用方法（Public Method），把内部变量当作它的私有属性（private value），这时一定要小心，不要随便改变父函数内部变量的值。

3）返回函数不要引用任何循环变量，或者后续会发生变化的变量。循环问题，原因就在于返回的函数引用了循环变量i，但它并非立刻执行，等到执行时循环变量i已经发生了改变。

## 8、深拷贝与浅拷贝

### 1、浅拷贝（shallow copy）

只复制指向某个对象的指针，而不复制对象本身，新旧对象共享一块内存；

### 2、深拷贝（deep copy）

复制并创建一个一摸一样的对象，不共享内存，修改新对象，旧对象保持不变。

### 3、深拷贝的实现

1、Object.assign()

Object.assign()是一种可以对非嵌套对象进行深拷贝的方法，如果对象中出现嵌套情况，那么其对被嵌套对象的行为就成了普通的浅拷贝。

2、转成JSON

用JSON.stringify把对象转成字符串，再用JSON.parse把字符串转成新的对象。

但这种方法的缺陷是会破坏原型链，并且无法拷贝属性值为function的属性

3、递归

function deepClone(source) {

var target = Array.isArray(source) ? [] : {};

for (var key in source) {

if (source.hasOwnProperty(key)) {

if (source[key] && typeof source[key] === 'object') {

target[key] = deepClone(source[key]);

} else {

target[key] = source[key];

}

}

}

return target

}

## 9、原生 Ajax

### 1、就绪状态码

* 0：请求未初始化（还没有调用 open()）。
* 1：请求已经建立，但是还没有发送（还没有调用 send()）。
* 2：请求已发送，正在处理中（通常现在可以从响应中获取内容头）。
* 3：请求在处理中；通常响应中已有部分数据可用了，但是服务器还没有完成响应的生成。
* 4：响应已完成；您可以获取并使用服务器的响应了。

### 2、实现

function ajax(options) {

var xhr;

if (window.XMLHttpRequest) {

xhr = new XMLHttpRequest();

} else {

xhr = new ActiveXObject('Microsoft.XMLHTTP');

}

xhr.responseType = options.dataType;

xhr.open(options.type, options.url, options.async);

xhr.setRequestHeader("Content-Type", options.contentType);

xhr.send(options.data);

xhr.onreadystatechange = function() {

if (xhr.readyState == 4) {

if(xhr.status == 200){

options.success(xhr.response)

}else{

options.error()

}

}

}

}

## 10、Object.defineProperty(obj, prop, descriptor)

### 1、参数

* obj：要在其上定义属性的对象。
* prop：要定义或修改的属性的名称。
* descriptor：将被定义或修改的属性描述符。

### 2、属性描述符

对象里目前存在的属性描述符有两种主要形式：数据描述符和存取描述符。数据描述符是一个具有值的属性，该值可能是可写的，也可能不是可写的。存取描述符是由getter-setter函数对描述的属性。描述符必须是这两种形式之一；不能同时是两者。

数据描述符和存取描述符均具有以下可选键值：

* configurable：当且仅当该属性的 configurable 为 true 时，该属性描述符才能够被改变，同时该属性也能从对应的对象上被删除。默认为 false。
* enumerable：当且仅当该属性的enumerable为true时，该属性才能够出现在对象的枚举属性中。默认为 false。

数据描述符同时具有以下可选键值：

* value：该属性对应的值。可以是任何有效的 JavaScript 值（数值，对象，函数等）。默认为 undefined。
* writable： 当且仅当该属性的writable为true时，value才能被赋值运算符改变。默认为 false。

存取描述符同时具有以下可选键值：

* get：一个给属性提供 getter 的方法，如果没有 getter 则为 undefined。当访问该属性时，该方法会被执行，方法执行时没有参数传入，但是会传入this对象（由于继承关系，这里的this并不一定是定义该属性的对象）。默认为 undefined。
* set：一个给属性提供 setter 的方法，如果没有 setter 则为 undefined。当属性值修改时，触发执行该方法。该方法将接受唯一参数，即该属性新的参数值。默认为 undefined。

## 11、bind

bind()方法创建一个新的函数， 当这个新函数被调用时其this置为提供的值，其参数列表前几项置为创建时指定的参数序列。

### 创建绑定函数

bind() 最简单的用法是创建一个函数，使这个函数不论怎么调用都有同样的 this 值。

this.x = 9;

var module = {

x: 81,

getX: function() { return this.x; }

};

module.getX(); // 返回 81

var retrieveX = module.getX;

retrieveX(); // 返回 9, 在这种情况下，"this"指向全局作用域

// 创建一个新函数，将"this"绑定到module对象

// 新手可能会被全局的x变量和module里的属性x所迷惑

var boundGetX = retrieveX.bind(module);

boundGetX(); // 返回 81

### 偏函数

bind()的另一个最简单的用法是使一个函数拥有预设的初始参数。这些参数（如果有的话）作为bind()的第二个参数跟在this（或其他对象）后面，之后它们会被插入到目标函数的参数列表的开始位置，传递给绑定函数的参数会跟在它们的后面。

把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数

例：

function fun1() {

console.log(arguments);

}

var fun2 = fun1.bind(undefined, 1);

fun2(); //Arguments [1, callee: ƒ, Symbol(Symbol.iterator): ƒ]

fun2(2); //Arguments(2) [1, 2, callee: ƒ, Symbol(Symbol.iterator): ƒ]

新的函数fun2就拥有了固定的默认参数1，并且放在函数传入的参数之前

### 配合 setTimeout

在默认情况下，使用 window.setTimeout() 时，this 关键字会指向 window （或全局）对象。

例：

var mod = {

num: 20,

getNum: function() {

console.log(this);

}

}

setTimeout(mod.getNum, 3000); //3秒后打印出this为 window

当使用类的方法时，需要 this 引用类的实例，你可能需要显式地把 this 绑定到回调函数以便继续使用实例。

function LateBloomer() {

this.petalCount = Math.ceil(Math.random() \* 12) + 1;

}

// Declare bloom after a delay of 1 second

LateBloomer.prototype.bloom = function() {

window.setTimeout(this.declare.bind(this), 1000);

};

LateBloomer.prototype.declare = function() {

console.log('I am a beautiful flower with ' +

this.petalCount + ' petals!');

};

var flower = new LateBloomer();

flower.bloom(); // 一秒钟后, 调用'declare'方法

### bind的实现

1. 通过给目标函数指定作用域来简单实现bind()方法：

Function.prototype.bind = function(context){

self = this; //保存this，即调用bind方法的目标函数

return function(){

return self.apply(context,arguments);

};

};

1. 支持在绑定的时候传参

Function.prototype.bind = function(context){

var args = Array.prototype.slice.call(arguments, 1),

self = this;

return function(){

var innerArgs = Array.prototype.slice.call(arguments);

var finalArgs = args.concat(innerArgs);

return self.apply(context,finalArgs);

};

};

1. Javascript的函数还可以作为构造函数，那么绑定后的函数用这种方式调用时，情况就比较微妙了，需要涉及到原型链的传递：

Function.prototype.bind = function(context){

var args = Array.prototype.slice(arguments, 1),

F = function(){},

self = this,

bound = function(){

var innerArgs = Array.prototype.slice.call(arguments);

var finalArgs = args.concat(innerArgs);

return self.apply((this instanceof F ? this : context), finalArgs);

};

F.prototype = self.prototype;

bound.prototype = new F();

retrun bound;

};

或：

if (!Function.prototype.bind) {

Function.prototype.bind = function(oThis) {

if (typeof this !== 'function') {

// closest thing possible to the ECMAScript 5

// internal IsCallable function

throw new TypeError('Function.prototype.bind - what is trying to be bound is not callable');

}

var aArgs = Array.prototype.slice.call(arguments, 1),

fToBind = this,

fNOP = function() {},

fBound = function() {

// this instanceof fNOP === true时,说明返回的fBound被当做new的构造函数调用

return fToBind.apply(this instanceof fNOP

? this

: oThis,

// 获取调用时(fBound)的传参.bind 返回的函数入参往往是这么传递的

aArgs.concat(Array.prototype.slice.call(arguments)));

};

// 维护原型关系

if (this.prototype) {

// Function.prototype doesn't have a prototype property

fNOP.prototype = this.prototype;

}

// 下行的代码使fBound.prototype是fNOP的实例,因此

// 返回的fBound若作为new的构造函数,new生成的新对象作为this传入fBound,新对象的\_\_proto\_\_就是fNOP的实例

fBound.prototype = new fNOP();

return fBound;

};

}

## 12、内存泄漏

不再用到的内存，没有及时释放，就叫做内存泄漏（memory leak）。

### 垃圾回收机制

引用计数：语言引擎有一张"引用表"，保存了内存里面所有的资源（通常是各种值）的引用次数。如果一个值的引用次数是0，就表示这个值不再用到了，因此可以将这块内存释放。

如果一个值不再需要了，引用数却不为0，垃圾回收机制无法释放这块内存，从而导致内存泄漏。

## 13、null和undefined

null是一个表示"无"的对象，转为数值时为0；undefined是一个表示"无"的原始值，转为数值时为NaN。

null表示"没有对象"，即该处不应该有值。典型用法是：

（1） 作为函数的参数，表示该函数的参数不是对象。

（2） 作为对象原型链的终点。

undefined表示"缺少值"，就是此处应该有一个值，但是还没有定义。典型用法是：

（1）变量被声明了，但没有赋值时，就等于undefined。

（2) 调用函数时，应该提供的参数没有提供，该参数等于undefined。

（3）对象没有赋值的属性，该属性的值为undefined。

（4）函数没有返回值时，默认返回undefined。

## 14、正则

1、url：/^http(s)?:\/\/([\w-]+\.)+[\w-]+(\/[\w- ./?%&=]\*)?$/

2、url参数：/([\\?|&])(.+?)=([^&?]\*)/ig

3、手机号：/^1[34578]\d{9}$/g

### 元字符

元字符（Metacharacter）是拥有特殊含义的字符：

| 字符 | 含义 |
| --- | --- |
| [\](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-backslash) | 匹配将依照下列规则：  在非特殊字符之前的反斜杠表示下一个字符是特殊的，不能从字面上解释。例如，没有前面'\'的'b'通常匹配小写'b'，无论它们出现在哪里。如果加了'\',这个字符变成了一个特殊意义的字符，意思是匹配一个[字符边界](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "note" \o "#special-word-boundary)。  反斜杠也可以将其后的特殊字符，转义为字面量。例如，模式 /a\*/ 代表会匹配 0 个或者多个 a。相反，模式 /a\\*/ 将 '\*' 的特殊性移除，从而可以匹配像 "a\*" 这样的字符串。  使用 new RegExp("pattern") 的时候不要忘记将 \ 进行转义，因为 \ 在字符串里面也是一个转义字符。 |
| [^](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-caret) | 匹配输入的开始。如果多行标志被设置为true，那么也匹配换行符后紧跟的位置。  例如，/^A/ 并不会匹配 "an A" 中的 'A'，但是会匹配 "An E" 中的 'A'。  当 '^' 作为第一个字符出现在一个字符集合模式时，它将会有不同的含义。[补充字符集合](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "note) 一节有详细介绍和示例。 |
| [$](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-dollar) | 匹配输入的结束。如果多行标示被设置为true，那么也匹配换行符前的位置。  例如，/t$/ 并不会匹配 "eater" 中的 't'，但是会匹配 "eat" 中的 't'。 |
| [\*](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-asterisk) | 匹配前一个表达式0次或多次。等价于 {0,}。  例如，/bo\*/会匹配 "A ghost boooooed" 中的 'booooo' 和 "A bird warbled" 中的 'b'，但是在 "A goat grunted" 中将不会匹配任何东西。 |
| [+](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-plus) | 匹配前面一个表达式1次或者多次。等价于 {1,}。  例如，/a+/匹配了在 "candy" 中的 'a'，和在 "caaaaaaandy" 中所有的 'a'。 |
| [?](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-questionmark) | 匹配前面一个表达式0次或者1次。等价于 {0,1}。  例如，/e?le?/ 匹配 "angel" 中的 'el'，和 "angle" 中的 'le' 以及"oslo' 中的'l'。  如果紧跟在任何量词 \*、 +、? 或 {} 的后面，将会使量词变为非贪婪的（匹配尽量少的字符），和缺省使用的贪婪模式（匹配尽可能多的字符）正好相反。  例如，对 "123abc" 应用 /\d+/ 将会返回 "123"，如果使用 /\d+?/,那么就只会匹配到 "1"。  还可以运用于先行断言，如本表的 x(?=y) 和 x(?!y) 条目中所述。 |
| [.](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-dot) | （小数点）匹配除换行符之外的任何单个字符。  例如，/.n/将会匹配 "nay, an apple is on the tree" 中的 'an' 和 'on'，但是不会匹配 'nay'。 |
| [(x)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-capturing-parentheses) | 匹配 'x' 并且记住匹配项，就像下面的例子展示的那样。括号被称为 捕获括号。  模式/(foo) (bar) \1 \2/中的 '(foo)' 和 '(bar)' 匹配并记住字符串 "foo bar foo bar" 中前两个单词。模式中的 \1 和 \2 匹配字符串的后两个单词。注意 \1、\2、\n 是用在正则表达式的匹配环节。在正则表达式的替换环节，则要使用像 $1、$2、$n 这样的语法，例如，'bar foo'.replace( /(...) (...)/, '$2 $1' )。 |
| [(?:x)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-non-capturing-parentheses) | 匹配 'x' 但是不记住匹配项。这种叫作非捕获括号，使得你能够定义为与正则表达式运算符一起使用的子表达式。来看示例表达式 /(?:foo){1,2}/。如果表达式是 /foo{1,2}/，{1,2}将只对 ‘foo’ 的最后一个字符 ’o‘ 生效。如果使用非捕获括号，则{1,2}会匹配整个 ‘foo’ 单词。 |
| [x(?=y)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-lookahead) | 匹配'x'仅仅当'x'后面跟着'y'.这种叫做正向肯定查找。  例如，/Jack(?=Sprat)/会匹配到'Jack'仅仅当它后面跟着'Sprat'。/Jack(?=Sprat|Frost)/匹配‘Jack’仅仅当它后面跟着'Sprat'或者是‘Frost’。但是‘Sprat’和‘Frost’都不是匹配结果的一部分。 |
| [x(?!y)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-negated-look-ahead) | 匹配'x'仅仅当'x'后面不跟着'y',这个叫做正向否定查找。  例如，/\d+(?!\.)/匹配一个数字仅仅当这个数字后面没有跟小数点的时候。正则表达式/\d+(?!\.)/.exec("3.141")匹配‘141’但是不是‘3.141’ |
| [x|y](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-or) | 匹配‘x’或者‘y’。  例如，/green|red/匹配“green apple”中的‘green’和“red apple”中的‘red’ |
| [{n}](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-quantifier) | n是一个正整数，匹配了前面一个字符刚好发生了n次。  比如，/a{2}/不会匹配“candy”中的'a',但是会匹配“caandy”中所有的a，以及“caaandy”中的前两个'a'。 |
| [{n,m}](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-quantifier-range) | n 和 m 都是整数。匹配前面的字符至少n次，最多m次。如果 n 或者 m 的值是0， 这个值被忽略。  例如，/a{1, 3}/ 并不匹配“cndy”中的任意字符，匹配“candy”中的a，匹配“caandy”中的前两个a，也匹配“caaaaaaandy”中的前三个a。注意，当匹配”caaaaaaandy“时，匹配的值是“aaa”，即使原始的字符串中有更多的a。 |
| [[xyz]](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-character-set) | 一个字符集合。匹配方括号中的任意字符，包括[转义序列](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Grammar_and_types)。你可以使用破折号（-）来指定一个字符范围。对于点（.）和星号（\*）这样的特殊符号在一个字符集中没有特殊的意义。他们不必进行转义，不过转义也是起作用的。  例如，[abcd] 和[a-d]是一样的。他们都匹配"brisket"中得‘b’,也都匹配“city”中的‘c’。/[a-z.]+/ 和/[\w.]+/都匹配“test.i.ng”中的所有字符。 |
| [[^xyz]](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-negated-character-set) | 一个反向字符集。也就是说， 它匹配任何没有包含在方括号中的字符。你可以使用破折号（-）来指定一个字符范围。任何普通字符在这里都是起作用的。  例如，[^abc] 和 [^a-c] 是一样的。他们匹配"brisket"中的‘r’，也匹配“chop”中的‘h’。 |
| [[\b]](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-backspace) | 匹配一个退格(U+0008)。（不要和\b混淆了。） |
| [\b](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-word-boundary) | 匹配一个词的边界。一个词的边界就是一个词不被另外一个“字”字符跟随的位置或者没有其他“字”字符在其前面的位置。注意，一个匹配的词的边界并不包含在匹配的内容中。换句话说，一个匹配的词的边界的内容的长度是0。（不要和[\b]混淆了）  例子：  /\bm/匹配“moon”中得‘m’；  /oo\b/并不匹配"moon"中得'oo'，因为'oo'被一个“字”字符'n'紧跟着。  /oon\b/匹配"moon"中得'oon'，因为'oon'是这个字符串的结束部分。这样他没有被一个“字”字符紧跟着。  /\w\b\w/将不能匹配任何字符串，因为在一个单词中间的字符永远也不可能同时满足没有“字”字符跟随和有“字”字符跟随两种情况。  注意: JavaScript的正则表达式引擎将[特定的字符集](http://www.ecma-international.org/ecma-262/5.1/" \l "sec-15.10.2.6)定义为“字”字符。不在该集合中的任何字符都被认为是一个断词。这组字符相当有限：它只包括大写和小写的罗马字母，十进制数字和下划线字符。不幸的是，重要的字符，例如“é”或“ü”，被视为断词。 |
| [\B](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-non-word-boundary) | 匹配一个非单词边界。他匹配一个前后字符都是相同类型的位置：都是“字”字符或者都不是“字”字符。一个字符串的开始和结尾都被认为不是“字”字符，或者空字符串。  例如，/\B../匹配"noonday"中的'oo', 而/y\B../匹配"possibly yesterday"中的’yes‘ |
| [\cX](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-control) | 当X是处于A到Z之间的字符的时候，匹配字符串中的一个控制符。  例如，/\cM/ 匹配字符串中的 control-M (U+000D)。 |
| [\d](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-digit) | 匹配一个数字。  等价于[0-9]。  例如， /\d/ 或者 /[0-9]/ 匹配"B2 is the suite number."中的'2'。 |
| [\D](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-non-digit) | 匹配一个非数字字符。  等价于[^0-9]。  例如， /\D/ 或者 /[^0-9]/ 匹配"B2 is the suite number."中的'B' 。 |
| [\f](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-form-feed) | 匹配一个换页符 (U+000C)。 |
| [\n](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-line-feed) | 匹配一个换行符 (U+000A)。 |
| [\r](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-carriage-return) | 匹配一个回车符 (U+000D)。 |
| [\s](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-white-space) | 匹配一个空白字符，包括空格、制表符、换页符和换行符。  等价于[ \f\n\r\t\v\u00a0\u1680\u180e\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff]。  例如, /\s\w\*/ 匹配"foo bar."中的' bar'。 |
| [\S](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-non-white-space) | 匹配一个非空白字符。  等价于[^ \f\n\r\t\v\u00a0\u1680\u180e\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff]。  例如， /\S\w\*/ 匹配"foo bar."中的'foo'。 |
| [\t](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-tab) | 匹配一个水平制表符 (U+0009)。 |
| [\v](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-vertical-tab) | 匹配一个垂直制表符 (U+000B)。 |
| [\w](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-word) | 匹配一个单字字符（字母、数字或者下划线）。  等价于[A-Za-z0-9\_]。  例如, /\w/ 匹配 "apple," 中的 'a'，"$5.28,"中的 '5' 和 "3D." 中的 '3'。 |
| [\W](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-non-word) | 匹配一个非单字字符。  等价于[^A-Za-z0-9\_]。  例如, /\W/ 或者 /[^A-Za-z0-9\_]/ 匹配 "50%." 中的 '%'。 |
| [\n](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-backreference) | 在正则表达式中，它返回最后的第n个子捕获匹配的子字符串(捕获的数目以左括号计数)。  比如 /apple(,)\sorange\1/ 匹配"apple, orange, cherry, peach."中的'apple, orange,' 。 |
| [\0](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-null) | 匹配 NULL (U+0000) 字符， 不要在这后面跟其它小数，因为 \0<digits> 是一个八进制转义序列。 |
| [\xhh](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-hex-escape) | 与代码 hh 匹配字符（两个十六进制数字） |
| [\uhhhh](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-unicode-escape) | 与代码 hhhh 匹配字符（四个十六进制数字）。 |

### 量词

|  |  |
| --- | --- |
| 量词 | 描述 |
| [n+](http://www.w3school.com.cn/jsref/jsref_regexp_onemore.asp) | 匹配任何包含至少一个 n 的字符串。 |
| [n\*](http://www.w3school.com.cn/jsref/jsref_regexp_zeromore.asp) | 匹配任何包含零个或多个 n 的字符串。 |
| [n?](http://www.w3school.com.cn/jsref/jsref_regexp_zeroone.asp) | 匹配任何包含零个或一个 n 的字符串。 |
| [n{X}](http://www.w3school.com.cn/jsref/jsref_regexp_nx.asp) | 匹配包含 X 个 n 的序列的字符串。 |
| [n{X,Y}](http://www.w3school.com.cn/jsref/jsref_regexp_nxy.asp) | 匹配包含 X 至 Y 个 n 的序列的字符串。 |
| [n{X,}](http://www.w3school.com.cn/jsref/jsref_regexp_nxcomma.asp) | 匹配包含至少 X 个 n 的序列的字符串。 |
| [n$](http://www.w3school.com.cn/jsref/jsref_regexp_ndollar.asp) | 匹配任何结尾为 n 的字符串。 |
| [^n](http://www.w3school.com.cn/jsref/jsref_regexp_ncaret.asp) | 匹配任何开头为 n 的字符串。 |
| [?=n](http://www.w3school.com.cn/jsref/jsref_regexp_nfollow.asp) | 匹配任何其后紧接指定字符串 n 的字符串。 |
| [?!n](http://www.w3school.com.cn/jsref/jsref_regexp_nfollow_not.asp) | 匹配任何其后没有紧接指定字符串 n 的字符串。 |

# ES6

1. let、const语句不出现变量提升。let不允许在相同作用域内，重复声明同一个变量。
2. ES6 允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构。 let [a, b, c] = [1, 2, 3]; 如果解构不成功，变量的值就等于undefined。解构不仅可以用于数组，还可以用于对象。let { foo, bar } = { foo: "aaa", bar: "bbb" };可以方便值的交换， [a, b] = [b, a]
3. ES6 为字符串添加了遍历器接口，使得字符串可以被for...of循环遍历。
4. ES6 为字符串又提供了三种新方法：

* includes()：返回布尔值，表示是否找到了参数字符串
* startsWith()：返回布尔值，表示参数字符串是否在原字符串的头部。
* endsWith()：返回布尔值，表示参数字符串是否在原字符串的尾部。

1. 新增加了模板字符串，用反引号（`）标识，模板字符串中嵌入变量，需要将变量名写在${}之中。
2. ES6 在Number对象上，新提供了Number.isFinite()和Number.isNaN()两个方法。Number.isFinite()用来检查一个数值是否为有限的（finite），即不是Infinity。
3. Math.abs(left - right) < Number.EPSILON \* Math.pow(2, 2);判断两个值是否相等。即差是否小于最小误差。
4. ES2016 新增了一个指数运算符（\*\*）。2 \*\* 2 // 4
5. Map 数据结构。它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。

10、async和awiat，async函数返回一个Promise对象。async函数内部return语句返回的值，会成为then方法回调函数的参数。正常情况下，await命令后面是一个 Promise 对象，返回该对象的结果。await命令只能用在async函数之中，如果用在普通函数，就会报错。

11、箭头函数

a) 不绑定this：箭头函数不会创建自己的this,它只会从自己的作用域链的上一层继承this。

b) 箭头函数没有自己的this指针，通过 call() 或 apply() 方法调用一个函数时，只能传递参数（不能绑定this---译者注），他们的第一个参数会被忽略。

c) 不绑定arguments：在大多数情况下，使用剩余参数是相较使用arguments对象的更好选择(...args) => args[0]

d) 箭头函数不能用作构造器，和 new一起用会抛出错误

e) 箭头函数没有prototype属性

12、Iterator(遍历器)：它是一种接口，为各种不同的数据结构提供统一的访问机制。任何数据结构只要部署 Iterator 接口，就可以完成遍历操作（即依次处理该数据结构的所有成员）。

Iterator 的作用有三个：一是为各种数据结构，提供一个统一的、简便的访问接口；二是使得数据结构的成员能够按某种次序排列；三是 ES6 创造了一种新的遍历命令for...of循环，Iterator 接口主要供for...of消费。Iterator有一个next方法，用来移动当前指针所指向的数据项，每一次调用next方法，都会返回数据结构的当前成员的信息。具体来说，就是返回一个包含value和done两个属性的对象。其中，value属性是当前成员的值，done属性是一个布尔值，表示遍历是否结束。

凡是部署了Symbol.iterator属性的数据结构，就称为部署了遍历器接口。调用这个接口，就会返回一个遍历器对象。

原生具备 Iterator 接口的数据结构如下。

* Array
* Map
* Set
* String
* TypedArray
* 函数的 arguments 对象
* NodeList 对象

13、decorator（修饰器）：修饰器是一个对类进行处理的函数。修饰器函数的第一个参数，就是所要修饰的目标类。修饰器只能用于类和类的方法，不能用于函数，因为存在函数提升。

使用：

@decorator

class A {}

修饰器不仅可以修饰类，还可以修饰类的属性。

class Person {

@readonly

name() { return `${this.first} ${this.last}` }

}

上面代码中，修饰器readonly用来修饰“类”的name方法。

修饰器函数readonly一共可以接受三个参数。

function readonly(target, name, descriptor){

// descriptor对象原来的值如下

// {

// value: specifiedFunction,

// enumerable: false,

// configurable: true,

// writable: true

// };

descriptor.writable = false;

return descriptor;

}

# 前端优化

## 一、页面级优化

### 1. 减少 HTTP请求数

减少 HTTP请求数的主要途径包括：

(1). 从设计实现层面简化页面

(2). 合理设置 HTTP缓存

(3). 资源合并与压缩

(4). CSS Sprites

(5). Inline Images

### 2. 将外部脚本置底（将脚本内容在页面信息内容加载后再加载）

## 二、代码级优化

### 1. Javascript

(1). DOM

DOM操作应该是脚本中最耗性能的一类操作

### 2.preload 和 prefetch

#### preload:

提供了一种声明式的命令，让浏览器提前加载指定资源(加载后并不执行)，需要执行时再执行。

这样做的好处在于：

1、将加载和执行分离开，不阻塞渲染和document的onload事件

2、提前加载指定资源，不再出现依赖的font字体隔了一段时间才刷出的情况

使用 preload 后，不管资源是否使用都将提前加载。若不确定资源是必定会加载的，则不要错误使用 preload，以免本末导致，给页面带来更沉重的负担

#### prefetch：

它的作用是告诉浏览器加载下一页面可能会用到的资源，注意，是下一页面，而不是当前页面。因此该方法的加载优先级非常低，也就是说该方式的作用是加速下一个页面的加载速度。

#### 区分：

preload 是告诉浏览器页面必定需要的资源，浏览器一定会加载这些资源

prefetch 是告诉浏览器页面可能需要的资源，浏览器不一定会加载这些资源

在VUE SSR生成的页面中，首页的资源均使用preload，而路由对应的资源，则使用prefetch

preload 和 prefetch 混用的话，并不会复用资源，而是会重复加载

# 算法

排序算法稳定性的简单形式化定义为：如果Ai = Aj，排序前Ai在Aj之前，排序后Ai还在Aj之前，则称这种排序算法是稳定的。通俗地讲就是保证排序前后两个相等的数的相对顺序不变。

## 1、排序算法



### 冒泡排序

它重复地走访过要排序的元素，依次比较相邻两个元素，如果他们的顺序错误就把他们调换过来，直到没有元素再需要交换，排序完成。

时间复杂度 ---- O(n^2)

稳定性 ------------ 稳定

冒泡排序算法的运作如下：

1. 比较相邻的元素，如果前一个比后一个大，就把它们两个调换位置。
2. 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。
3. 针对所有的元素重复以上的步骤，除了最后一个。
4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

例：

function bubbleSort(arr) {

var newArr = JSON.parse(JSON.stringify(arr));

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

for (var j = i + 1; j < newArr.length; j++) {

if (newArr[j] < newArr[i]) {

var temp = newArr[i];

newArr[i] = newArr[j];

newArr[j] = temp;

}

}

}

return newArr;

}

### 鸡尾酒排序

鸡尾酒排序，也叫定向冒泡排序，是冒泡排序的一种改进。此算法与冒泡排序的不同处在于从低到高然后从高到低，而冒泡排序则仅从低到高去比较序列里的每个元素。他可以得到比冒泡排序稍微好一点的效能。

间复杂度 ---- O(n^2)

稳定性 ------------ 稳定

例：

function bubbleHighSort(arr) {

var newArr = JSON.parse(JSON.stringify(arr));

console.log(new Date().getTime());

var left = 0, right = newArr.length - 1;

while(left < right) {

for (var i = left; i < right; i++) {

if (newArr[i] > newArr[i + 1]) {

swapFn(newArr, i, i + 1);

}

}

right--;

for (var i = right; i > left; i--) {

if (newArr[i] < newArr[i - 1]) {

swapFn(newArr, i, i -1);

}

}

left++;

}

console.log(new Date().getTime());

return newArr;

}

### 选择排序

选择排序也是一种简单直观的排序算法。它的工作原理很容易理解：初始时在序列中找到最小（大）元素，放到序列的起始位置作为已排序序列；然后，再从剩余未排序元素中继续寻找最小（大）元素，放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

注意选择排序与冒泡排序的区别：冒泡排序通过依次交换相邻两个顺序不合法的元素位置，从而将当前最小（大）元素放到合适的位置；而选择排序每遍历一次都记住了当前最小（大）元素的位置，最后仅需一次交换操作即可将其放到合适的位置。

时间复杂度 ---- O(n^2)

稳定性 ------------ 不稳定

例：

function selectSort(arr) {

var newArr = JSON.parse(JSON.stringify(arr));

for (var i = 0; i < newArr.length - 1; i++) {

var mini = newArr[i], miniIndex = i;

for (var j = i + 1; j < newArr.length; j++) {

if (newArr[j] < mini) {

mini = newArr[j];

miniIndex = j;

}

}

swapFn(newArr, i, miniIndex);

}

return newArr;

}

比如序列：{ 5, 8, 5, 2, 9 }，一次选择的最小元素是2，然后把2和第一个5进行交换，从而改变了两个元素5的相对次序。

### 插入排序

插入排序是一种简单直观的排序算法。它的工作原理非常类似于我们抓扑克牌

插入排序在实现上，通常采用in-place排序（即只需用到O(1)的额外空间的排序），因而在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。

插入排序不适合对于数据量比较大的排序应用。但是，如果需要排序的数据量很小，比如量级小于千，那么插入排序还是一个不错的选择。

时间复杂度 ---- O(n^2)

稳定性 ------------ 稳定

具体算法描述如下：

1. 从第一个元素开始，该元素可以认为已经被排序
2. 取出下一个元素，在已经排序的元素序列中从后向前扫描
3. 如果该元素（已排序）大于新元素，将该元素移到下一位置
4. 重复步骤3，直到找到已排序的元素小于或者等于新元素的位置
5. 将新元素插入到该位置后
6. 重复步骤2~5

例：

function insertSort(arr) {

var newArr = JSON.parse(JSON.stringify(arr));

for (var i = 1; i < newArr.length; i++) {

var temp = newArr[i];

var j = i - 1;

while(j >= 0 && newArr[j] > temp) {

newArr[j + 1] = newArr[j];

j--;

}

newArr[j + 1] = temp;

}

return newArr;

}

### 快速排序

稳定性 ---------- 不稳定（与所选取的基数有关）

快速排序使用分治策略(Divide and Conquer)来把一个序列分为两个子序列。步骤为：

1. 从序列中挑出一个元素，作为"基准"(pivot).
2. 把所有比基准值小的元素放在基准前面，所有比基准值大的元素放在基准的后面（相同的数可以到任一边），这个称为分区(partition)操作。
3. 对每个分区递归地进行步骤1~2，递归的结束条件是序列的大小是0或1，这时整体已经被排好序了。

例：

function quickSort(arr, left, right) {

var i = left, j = right, base = arr[left];

if (left < right) {

while (i < j) {

while(i < j && arr[j] > base) j--;

if (i < j) {

arr[i] = arr[j];

i++;

}

while(i < j && arr[i] < base) i++;

if (i < j) {

arr[j] = arr[i];

j--;

}

}

arr[i] = base;

quickSort(arr, left, i - 1);

quickSort(arr, i + 1, right);

}

}

比如序列：{ 1, 3, 4, 2, 8, 9, 8, 7, 5 }，基准元素是5，一次划分操作后5要和第一个8进行交换，从而改变了两个元素8的相对次序。

### 各种排序方法比较

简单排序中直接插入最好，快速排序最快，当文件为正序时，直接插入和冒泡均最佳。

### 影响排序效果的因素

因为不同的排序方法适应不同的应用环境和要求，所以选择合适的排序方法应综合考虑下列因素：

　　①待排序的记录数目n；

　　②记录的大小(规模)；

　　③关键字的结构及其初始状态；

　　④对稳定性的要求；

　　⑤语言工具的条件；

　　⑥存储结构；

⑦时间和辅助空间复杂度等。

不同条件下，排序方法的选择

(1)若n较小(如n≤50)，可采用直接插入或直接选择排序。

　当记录规模较小时，直接插入排序较好；否则因为直接选择移动的记录数少于直接插人，应选直接选择排序为宜。

(2)若文件初始状态基本有序(指正序)，则应选用直接插人、冒泡或随机的快速排序为宜；

(3)若n较大，则应采用时间复杂度为O(nlgn)的排序方法：快速排序、堆排序或归并排序。

　快速排序是目前基于比较的内部排序中被认为是最好的方法，当待排序的关键字是随机分布时，快速排序的平均时间最短；

　堆排序所需的辅助空间少于快速排序，并且不会出现快速排序可能出现的最坏情况。这两种排序都是不稳定的。

　若要求排序稳定，则可选用归并排序。但本章介绍的从单个记录起进行两两归并的 排序算法并不值得提倡，通常可以将它和直接插入排序结合在一起使用。先利用直接插入排序求得较长的有序子文件，然后再两两归并之。因为直接插入排序是稳定 的，所以改进后的归并排序仍是稳定的。

## 2、二分查找

有序的，一分为二，和中间值比，如果等于，结束，如果小于比较前半个，如果大于比较后半个，重复此操作。时间复杂度可以表示O(h)=O(log2n)

function binary(find, arr, low, high) {

if (low <= high) {

if (arr[low] == find) {

return low;

}

if (arr[high] == find) {

return high;

}

var mid = Math.ceil((high + low) / 2);

if (arr[mid] == find) {

return mid;

} else if (arr[mid] > find) {

return binary(find, arr, low, mid - 1);

} else {

return binary(find, arr, mid + 1, high);

}

}

return -1;

}

# http status code

HTTP状态码列表

|  |  |  |
| --- | --- | --- |
| 100 | Continue | 继续。[客户端](http://www.dreamdu.com/webbuild/client_vs_server/)应继续其请求 |
| 101 | Switching Protocols | 切换协议。服务器根据客户端的请求切换协议。只能切换到更高级的协议，例如，切换到HTTP的新版本协议 |
|  | |
| 200 | OK | 请求成功。一般用于GET与POST请求 |
| 201 | Created | 已创建。成功请求并创建了新的资源 |
| 202 | Accepted | 已接受。已经接受请求，但未处理完成 |
| 203 | Non-Authoritative Information | 非授权信息。请求成功。但返回的meta信息不在原始的服务器，而是一个副本 |
| 204 | No Content | 无内容。服务器成功处理，但未返回内容。在未更新网页的情况下，可确保浏览器继续显示当前文档 |
| 205 | Reset Content | 重置内容。服务器处理成功，用户终端（例如：浏览器）应重置文档视图。可通过此返回码清除浏览器的表单域 |
| 206 | Partial Content | 部分内容。服务器成功处理了部分GET请求 |
|  | |
| 300 | Multiple Choices | 多种选择。请求的资源可包括多个位置，相应可返回一个资源特征与地址的列表用于用户终端（例如：浏览器）选择 |
| 301 | Moved Permanently | 永久移动。请求的资源已被永久的移动到新URI，返回信息会包括新的URI，浏览器会自动定向到新URI。今后任何新的请求都应使用新的URI代替 |
| 302 | Found | 临时移动。与301类似。但资源只是临时被移动。客户端应继续使用原有URI |
| 303 | See Other | 查看其它地址。与301类似。使用GET和POST请求查看 |
| 304 | Not Modified | 未修改。所请求的资源未修改，服务器返回此状态码时，不会返回任何资源。客户端通常会缓存访问过的资源，通过提供一个头信息指出客户端希望只返回在指定日期之后修改的资源 |
| 305 | Use Proxy | 使用代理。所请求的资源必须通过代理访问 |
| 306 | Unused | 已经被废弃的HTTP状态码 |
| 307 | Temporary Redirect | 临时重定向。与302类似。使用GET请求重定向 |
|  | |
| 400 | Bad Request | 客户端请求的语法错误，服务器无法理解 |
| 401 | Unauthorized | 请求要求用户的身份认证 |
| 402 | Payment Required | 保留，将来使用 |
| 403 | Forbidden | 服务器理解请求客户端的请求，但是拒绝执行此请求 |
| 404 | Not Found | 服务器无法根据客户端的请求找到资源（网页）。通过此代码，网站设计人员可设置"您所请求的资源无法找到"的个性页面 |
| 405 | Method Not Allowed | 客户端请求中的方法被禁止 |
| 406 | Not Acceptable | 服务器无法根据客户端请求的内容特性完成请求 |
| 407 | Proxy Authentication Required | 请求要求代理的身份认证，与401类似，但请求者应当使用代理进行授权 |
| 408 | Request Time-out | 服务器等待客户端发送的请求时间过长，超时 |
| 409 | Conflict | 服务器完成客户端的PUT请求是可能返回此代码，服务器处理请求时发生了冲突 |
| 410 | Gone | 客户端请求的资源已经不存在。410不同于404，如果资源以前有现在被永久删除了可使用410代码，网站设计人员可通过301代码指定资源的新位置 |
| 411 | Length Required | 服务器无法处理客户端发送的不带Content-Length的请求信息 |
| 412 | Precondition Failed | 客户端请求信息的先决条件错误 |
| 413 | Request Entity Too Large | 由于请求的实体过大，服务器无法处理，因此拒绝请求。为防止客户端的连续请求，服务器可能会关闭连接。如果只是服务器暂时无法处理，则会包含一个Retry-After的响应信息 |
| 414 | Request-URI Too Large | 请求的URI过长（URI通常为网址），服务器无法处理 |
| 415 | Unsupported Media Type | 服务器无法处理请求附带的媒体格式 |
| 416 | Requested range not satisfiable | 客户端请求的范围无效 |
| 417 | Expectation Failed | 服务器无法满足Expect的请求头信息 |
|  | |
| 500 | Internal Server Error | 服务器内部错误，无法完成请求 |
| 501 | Not Implemented | 服务器不支持请求的功能，无法完成请求 |
| 502 | Bad Gateway | 充当网关或代理的服务器，从远端服务器接收到了一个无效的请求 |
| 503 | Service Unavailable | 由于超载或系统维护，服务器暂时的无法处理客户端的请求。延时的长度可包含在服务器的Retry-After头信息中 |
| 504 | Gateway Time-out | 充当网关或代理的服务器，未及时从远端服务器获取请求 |
| 505 | HTTP Version not supported | 服务器不支持请求的HTTP协议的版本，无法完成处理 |

# DOM操作

## 1、方法

### 1、获取节点

getElementById(id)根据id获取节点的DOM对象

getElementsByTagName(name)根据name属性名返回相应的节点对象数组（列表）

getElementsByClassName(className)根据class名称获取节点对象数组（列表），在IE9以下不支持

### 2、新增节点

document.createElement()创建节点对象，参数是字符串，也就是html标签

creatTextNode创建文本节点，配合createElement()使用

appendChild(element)把新的节点添加到指定节点下，参数是一个节点对象

insertChild()在指定节点前插入新的子节点

### 3、修改节点

replaceChild()节点交换

setAttribute()设置属性

### 4、删除节点

removeChild(element)删除节点，要先获得父节点然后再删除子节点

## 2、属性

innerHtml节点内容，可以获取或设置

parentNode当前节点的父节点

childNodes获取当前节点的儿子节点

attributes节点的属性

style

# 移动端适配

<meta name="viewport" content="width=device-width, initial-scale=1,user-scalable=no, minimum-scale=1.0, maximum-scale=1.0">

该meta标签的作用是让当前viewport的宽度等于设备的宽度，同时不允许用户手动缩放。

在苹果的规范中，meta viewport 有6个属性(暂且把content中的那些东西称为一个个属性和值)，如下：

width 设置layout viewport 的宽度，为一个正整数，或字符串"width-device"

initial-scale 设置页面的初始缩放值，为一个数字，可以带小数

minimum-scale 允许用户的最小缩放值，为一个数字，可以带小数

maximum-scale 允许用户的最大缩放值，为一个数字，可以带小数

height 设置layout viewport 的高度，这个属性对我们并不重要，很少使用

user-scalable 是否允许用户进行缩放，值为"no"或"yes", no 代表不允许，yes代表允许

## 1、基本概念

1、物理像素

一个物理像素是显示器(手机屏幕)上最小的物理显示单元，在操作系统的调度下，每一个设备像素都有自己的颜色值和亮度值。例如手机375\*667

2、设备独立像素

在javascript中，可以通过window.devicePixelRatio获取到当前设备的dpr。

设备独立像素(也叫密度无关像素)，可以认为是计算机坐标系统中得一个点，这个点代表一个可以由程序使用的虚拟像素(比如: css像素)，然后由相关系统转换为物理像素。例如retina屏 750\*1134

3、设备像素比dpr

设备像素比 = 物理像素 / 设备独立像素 // 在某一方向上，x方向或者y方向

## 2、rem

针对不同手机屏幕尺寸和dpr动态的改变根节点html的font-size大小(基准值)。引入一段js。

rem = document.documentElement.clientWidth \* window.devicePixelRatio / 10

## 3、淘宝flexible

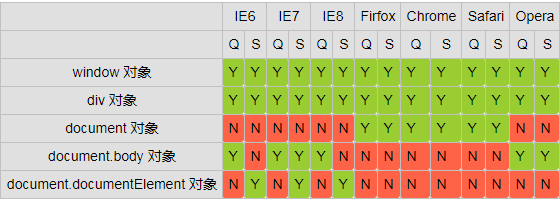
flexible的实质

* 动态改写<meta>标签
* 给<html>元素添加data-dpr属性，并且动态改写data-dpr的值
* 给<html>元素添加font-size属性，并且动态改写font-size的值

# 兼容性

## js兼容

* 在标准的事件绑定中绑定事件的方法函数为 addEventListener,而IE使用的是attachEvent。
* 事件处理中非常有用的event属性获得亦不相同，标准浏览器是作为参数带人，而ie是window.event方式获得，获得目标元素ie为e.srcElement 标准浏览器为e.target；
* 然后ie日期函数处理与其它浏览器不大一致，比如： var year= new Date().getYear(); 在IE中会获得当前年，但是在firefox中则会获得当前年与1900的差值。建议使用getFullYear
* 各浏览器中 Date 对象的 toLocaleString 方法的返回值不一致，Chrome："2018/10/30 上午9:39:59"，IE："2018年10月30日 9:41:02"
* IE6 IE7 IE8(Q) 不支持 JSON 对象。
* 使用 document.getElementById 方法获取页面内的元素时，在 IE6 IE7 IE8(Q) 中的 id 是大小写不敏感的。
* 各浏览器对 document、document.body、document.documentElement 对象的 onscroll 事件支持情况



* 获得DOM节点的方法有所差异，其获得子节点方法不一致。
* IE下,event对象有x,y属性,但是没有pageX,pageY属性; Firefox下,event对象有pageX,pageY属性,但是没有x,y属性. mX = event.x ? event.x : event.pageX;
* 获取父元素，IE：obj.parentElement，firefox：obj.parentNode

## css兼容

# ****promise****

function Promise(executor) {

var self = this

self.status = 'pending'

self.onResolvedCallback = []

self.onRejectedCallback = []

function resolve(value) {

if (value instanceof Promise) {

return value.then(resolve, reject)

}

setTimeout(function() { // 异步执行所有的回调函数

if (self.status === 'pending') {

self.status = 'resolved'

self.data = value

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

self.onResolvedCallback[i](value)

}

}

})

}

function reject(reason) {

setTimeout(function() { // 异步执行所有的回调函数

if (self.status === 'pending') {

self.status = 'rejected'

self.data = reason

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

self.onRejectedCallback[i](reason)

}

}

})

}

try {

executor(resolve, reject)

} catch (reason) {

reject(reason)

}

}

function resolvePromise(promise2, x, resolve, reject) {

var then

var thenCalledOrThrow = false

if (promise2 === x) {

return reject(new TypeError('Chaining cycle detected for promise!'))

}

if (x instanceof Promise) {

if (x.status === 'pending') { //because x could resolved by a Promise Object

x.then(function(v) {

resolvePromise(promise2, v, resolve, reject)

}, reject)

} else { //but if it is resolved, it will never resolved by a Promise Object but a static value;

x.then(resolve, reject)

}

return

}

if ((x !== null) && ((typeof x === 'object') || (typeof x === 'function'))) {

try {

then = x.then //because x.then could be a getter

if (typeof then === 'function') {

then.call(x, function rs(y) {

if (thenCalledOrThrow) return

thenCalledOrThrow = true

return resolvePromise(promise2, y, resolve, reject)

}, function rj(r) {

if (thenCalledOrThrow) return

thenCalledOrThrow = true

return reject(r)

})

} else {

resolve(x)

}

} catch (e) {

if (thenCalledOrThrow) return

thenCalledOrThrow = true

return reject(e)

}

} else {

resolve(x)

}

}

Promise.prototype.then = function(onResolved, onRejected) {

var self = this

var promise2

onResolved = typeof onResolved === 'function' ? onResolved : function(v) {

return v

}

onRejected = typeof onRejected === 'function' ? onRejected : function(r) {

throw r

}

if (self.status === 'resolved') {

return promise2 = new Promise(function(resolve, reject) {

setTimeout(function() { // 异步执行onResolved

try {

var x = onResolved(self.data)

resolvePromise(promise2, x, resolve, reject)

} catch (reason) {

reject(reason)

}

})

})

}

if (self.status === 'rejected') {

return promise2 = new Promise(function(resolve, reject) {

setTimeout(function() { // 异步执行onRejected

try {

var x = onRejected(self.data)

resolvePromise(promise2, x, resolve, reject)

} catch (reason) {

reject(reason)

}

})

})

}

if (self.status === 'pending') {

// 这里之所以没有异步执行，是因为这些函数必然会被resolve或reject调用，而resolve或reject函数里的内容已是异步执行，构造函数里的定义

return promise2 = new Promise(function(resolve, reject) {

self.onResolvedCallback.push(function(value) {

try {

var x = onResolved(value)

resolvePromise(promise2, x, resolve, reject)

} catch (r) {

reject(r)

}

})

self.onRejectedCallback.push(function(reason) {

try {

var x = onRejected(reason)

resolvePromise(promise2, x, resolve, reject)

} catch (r) {

reject(r)

}

})

})

}

}

Promise.prototype.catch = function(onRejected) {

return this.then(null, onRejected)

}

Promise.deferred = Promise.defer = function() {

var dfd = {}

dfd.promise = new Promise(function(resolve, reject) {

dfd.resolve = resolve

dfd.reject = reject

})

return dfd

}

# 缓存

一般针对静态资源如CSS,JS,图片等使用缓存

## 1、为什么要用缓存

* 请求更快：通过将内容缓存在本地浏览器或距离最近的缓存服务器（如CDN），在不影响网站交互的前提下可以大大加快网站加载速度。
* 节省带宽：对于已缓存的文件，可以减少请求带宽甚至无需请求网络。
* 降低服务器压力：在大量用户并发请求的情况下，服务器的性能受到限制，此时将一些静态资源放置在网络的多个节点，可以起到均衡负载的作用，降低服务器的压力。

## 2、缓存分类

缓存分为服务端侧（server side，比如 Nginx、Apache）和客户端侧（client side，比如 web browser）。

常用的服务端缓存有CDN缓存，客户端缓存就是指浏览器缓存。

## 3、浏览器缓存机制

### 缓存类型

浏览器缓存分为强缓存和协商缓存：

* 强缓存：浏览器在加载资源时，先根据这个资源的一些http header判断它是否命中强缓存，强缓存如果命中，浏览器直接从自己的缓存中读取资源，不会发请求到服务器。比如某个css文件，如果浏览器在加载它所在的网页时，这个css文件的缓存配置命中了强缓存，浏览器就直接从缓存中加载这个css，连请求都不会发送到网页所在服务器；
* 协商缓存：当强缓存没有命中的时候，浏览器一定会发送一个请求到服务器，通过服务器端依据资源的另外一些http header验证这个资源是否命中协商缓存，如果协商缓存命中，服务器会将这个请求返回（304），但是不会返回这个资源的数据，而是告诉客户端可以直接从缓存中加载这个资源，于是浏览器就又会从自己的缓存中去加载这个资源；若未命中请求，则将资源返回客户端，并更新本地缓存数据（200）。

强缓存与协商缓存区别：强缓存不发请求到服务器，协商缓存会发请求到服务器。

## 4、如何设置缓存

### 1、HTML Meta标签控制缓存（非HTTP协议定义）

<META HTTP-EQUIV="Pragma" CONTENT="no-cache">

上述代码的作用是告诉浏览器当前页面不被缓存，每次访问都需要去服务器拉取。这种方法使用上很简单，但只有部分浏览器可以支持，而且所有缓存代理服务器都不支持，因为代理不解析HTML内容本身。

### 2、HTTP头信息控制缓存

HTTP头信息控制缓存是通过Expires（强缓存）、Cache-control（强缓存）、Last-Modified/If-Modified-Since（协商缓存）、Etag/If-None-Match（协商缓存）实现，下面详细介绍。

* Expires：是http1.0提出的一个表示资源过期时间的header，它描述的是一个绝对时间，由服务器返回，用GMT格式的字符串表示，如：Expires:Thu, 31 Dec 2016 23:55:55 GMT，读取缓存数据条件：缓存过期时间（服务器的）< 当前时间（客户端的

缺点：Expires是较老的强缓存管理header，由于它是服务器返回的一个绝对时间，这样存在一个问题，如果客户端的时间与服务器的时间相差很大（比如时钟不同步，或者跨时区），那么误差就很大，所以在HTTP 1.1版开始，使用Cache-Control: max-age=秒替代。

* Cache-Control：描述的是一个相对时间，在进行缓存命中的时候，都是利用客户端时间进行判断，所以相比较Expires，Cache-Control的缓存管理更有效，安全一些。读取缓存数据条件：上次缓存时间（客户端的）+max-age < 当前时间（客户端的）

Cache-Control值可以是public、private、no-cache、no- store、no-transform、must-revalidate、proxy-revalidate、max-age

各个消息中的指令含义如下：

1. Public指示响应可被任何缓存区缓存。
2. Private指示对于单个用户的整个或部分响应消息，不能被共享缓存处理。这允许服务器仅仅描述当前用户的部分响应消息，此响应消息对于其他用户的请求无效。
3. no-cache指示请求或响应消息不能缓存，该选项并不是说可以设置”不缓存“，而是需要和服务器确认
4. no-store在请求消息中发送将使得请求和响应消息都不使用缓存，完全不存下來。
5. max-age指示客户机可以接收生存期不大于指定时间（以秒为单位）的响应。上次缓存时间（客户端的）+max-age（64200s）<客户端当前时间
6. min-fresh指示客户机可以接收响应时间小于当前时间加上指定时间的响应。
7. max-stale指示客户机可以接收超出超时期间的响应消息。如果指定max-stale消息的值，那么客户机可以接收超出超时期指定值之内的响应消息。

* Last-Modified/If-Modified-Since：Last-Modified/If-Modified-Since要配合Cache-Control使用。

Last-Modified：标示这个响应资源的最后修改时间。web服务器在响应请求时，告诉浏览器资源的最后修改时间。

If-Modified-Since：当资源过期时（强缓存失效），发现资源具有Last-Modified声明，则再次向web服务器请求时带上头 If-Modified-Since，表示请求时间。web服务器收到请求后发现有头If-Modified-Since 则与被请求资源的最后修改时间进行比对。若最后修改时间较新，说明资源又被改动过，则响应整片资源内容（写在响应消息包体内），HTTP 200；若最后修改时间较旧，说明资源无新修改，则响应HTTP 304 (无需包体，节省浏览)，告知浏览器继续使用所保存的cache。

缺点：

Last-Modified标注的最后修改只能精确到秒级，如果某些文件在1秒钟以内，被修改多次的话，它将不能准确标注文件的修改时间（无法及时更新文件）

如果某些文件会被定期生成，当有时内容并没有任何变化，但Last-Modified却改变了，导致文件没法使用缓存，有可能存在服务器没有准确获取文件修改时间，或者与代理服务器时间不一致等情形（无法使用缓存）。HTTP1.1中Etag解决了上述问题。

* Etag/If-None-Match：Etag/If-None-Match也要配合Cache-Control使用。

1. Etag：web服务器响应请求时，告诉浏览器当前资源在服务器的唯一标识（生成规则由服务器决定）。Apache中，ETag的值，默认是对文件的索引节（INode），大小（Size）和最后修改时间（MTime）进行Hash后得到的。
2. If-None-Match：当资源过期时（使用Cache-Control标识的max-age），发现资源具有Etage声明，则再次向web服务器请求时带上头If-None-Match （Etag的值）。web服务器收到请求后发现有头If-None-Match 则与被请求资源的相应校验串进行比对，决定返回200或304。

Etag是服务器自动生成或者由开发者生成的对应资源在服务器端的唯一标识符，能够更加准确的控制缓存。Last-Modified与ETag一起使用时，服务器会优先验证ETag。

## 5、CDN缓存

CDN缓存属于Cache服务器的一种。

CDN的全称是Content Delivery Network，即内容分发网络。其目的是通过在现有的Internet中增加一层新的网络架构，将网站的内容发布到最接近用户的网络"边缘"，使用户可 以就近取得所需的内容，解决Internet网络拥塞状况，提高用户访问网站的响应速度。从技术上全面解决由于网络带宽小、用户访问量大、网点分布不均等 原因，解决用户访问网站的响应速度慢的根本原因。

使用了CDN缓存后的网站的访问过程为：

1. 用户向浏览器提供要访问的域名；
2. 浏览器调用域名解析库对域名进行解析，由于CDN对域名解析过程进行了调整，所以解析函数库一般得到的是该域名对应的CNAME记录，为了得到实际IP地址，浏览器需要再次对获得的CNAME域名进行解析以得到实际的IP地址；在此过程中，使用的全局负载均衡DNS解析，如根据地理位置信 息解析对应的IP地址，使得用户能就近访问。
3. 此次解析得到CDN缓存服务器的IP地址，浏览器在得到实际的IP地址以后，向缓存服务器发出访问请求；
4. 若请求文件并未修改，返回304（充当服务器的角色）。若当前文件已过期，则缓存服务器根据浏览器提供的要访问的域名，通过Cache内部专用DNS解析得到此域名的实际IP地址，再由缓存服务器向此实际IP地址提交访问请求；
5. 缓存服务器从实际IP地址得得到内容以后，一方面在本地进行保存，以备以后使用，二方面把获取的数据返回给客户端，完成数据服务过程；
6. 客户端得到由缓存服务器返回的数据以后显示出来并完成整个浏览的数据请求过程。

# vue

## 生命周期

* beforeCreate：在实例初始化之后，数据观测 (data observer) 和 event/watcher 事件配置之前被调用。
* created：在实例创建完成后被立即调用。在这一步，实例已完成以下的配置：数据观测 (data observer)，属性和方法的运算，watch/event 事件回调。然而，挂载阶段还没开始，$el 属性目前不可见。
* beforeMount：在挂载开始之前被调用：相关的 render 函数首次被调用。该钩子在服务器端渲染期间不被调用。
* mounted：el 被新创建的 vm.$el 替换，并挂载到实例上去之后调用该钩子。注意 mounted 不会承诺所有的子组件也都一起被挂载。如果你希望等到整个视图都渲染完毕，可以用 vm.$nextTick 替换掉 mounted。
* beforeUpdate：数据更新时调用，发生在虚拟 DOM 打补丁之前。这里适合在更新之前访问现有的 DOM，比如手动移除已添加的事件监听器。
* updated：由于数据更改导致的虚拟 DOM 重新渲染和打补丁，在这之后会调用该钩子。当这个钩子被调用时，组件 DOM 已经更新，所以你现在可以执行依赖于 DOM 的操作。然而在大多数情况下，你应该避免在此期间更改状态。
* activated：keep-alive 组件激活时调用。
* deactivated：keep-alive 组件停用时调用。该钩子在服务器端渲染期间不被调用。
* beforeDestroy：实例销毁之前调用。在这一步，实例仍然完全可用。该钩子在服务器端渲染期间不被调用。
* destroyed：Vue 实例销毁后调用。调用后，Vue 实例指示的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。该钩子在服务器端渲染期间不被调用。

## 双向绑定原理

Vue在观察者模式中结合发布-订阅模式，其中涉及到了三个重要的对象：Observer, Dep, Watcher。

Observer：负责观察目标数据的变化，如果数据变化了，那么通知Dep。

Dep：负责维护一个订阅者列表（收集依赖），当接收到Observer的通知时，他就通知所有订阅者：目标数据更新了。

Watcher：维护一个回调函数，当接收到Dep的通知时，执行回调函数。

* 通过 Object.defineProperty() 替换配置对象属性的 set、get 方法，实现“拦截”
* watcher 在执行 getter 函数时触发数据的 get 方法，从而建立依赖关系
* 写入数据时触发 set 方法，从而借助 dep 发布通知，进而 watcher 进行更新
* 执行 observe 方法，创建 Observer 执行 walk 为对象数据添加setter 和 getter
* 在添加 setter 和 getter 时，创建 Dep，在 getter 方法中执行 dep.depend() 收集 watcher，在 setter 方法中执行 dep.notify() 方法，最终遍历执行 watcher 数组的回调函数。
* Dep 类似于 Watcher 和 Observer 的中间件。
* Watcher 用于监听变化，并执行回调函数。
* 当 Watcher 实例创建时，Watcher 实例会将自身传递给 Dep.target
* Watcher 调用监测数据的 getter方法触发 dep.depend()
* dep.depend()方法将当前 Watcher（Dep.target）传递给Dep的subs（watcher数组）中。
* 当被监测的数据内容发生改变时，执行 setter 方法，触发 dep.notify() 方法，遍历 Dep 中的 subs（watcher数组），执行 Watcher 的回调函数。

## 虚拟DOM的diff算法

patch->patchNode->updateChildren

参考文档：<https://blog.csdn.net/M6i37JK/article/details/78140159>

## vue2.0最大的改进是什么

渲染函数：尽管渲染层全面更新，Vue 2.0 兼容了绝大部分的 1.0 模板语法，仅废弃掉了其中的一小部分。这些模板在背后被编译成了 Virtual DOM 渲染函数，

服务端渲染：vue 2.0 支持服务端渲染 (SSR)，并且是流式的，可以做组件级的缓存，这使得极速渲染成为可能。

## vue3.0改变

* 使用typescript编写，主要进行类型推断，目前版本使用的是flow。
* 数据的监测由原来的Object.defineProperty改为Proxy，从而消除原来的局限性。如：对属性的添加和删除，数组下标的修改和.length的修改、对map、set、的支持等。
* 更精准的变动通知，举个例子：在 2.x 系列中，通过 Vue.set 强制添加一个新的属性，将导致所有依赖于这个对象的 watch 函数都会被执行一次；而在 3.x 中，只有依赖于这个具体属性的 watch 函数会被通知到。
* 更良好的可调试能力：通过使用新的 renderTracked 和 renderTriggered 钩子，我们可以精确地追踪到一个组件发生重渲染的触发时机和完成时机，及其原因。

## mixins

### 选项合并

* 数据对象在内部会进行浅合并 (一层属性深度)，在和组件的数据发生冲突时以组件数据优先。
* 同名钩子函数将混合为一个数组，因此都将被调用。另外，混入对象的钩子将在组件自身钩子之前调用。
* 值为对象的选项，例如 methods, components 和 directives，将被混合为同一个对象。两个对象键名冲突时，取组件对象的键值对。

### 全局混入

一旦使用全局混入对象，将会影响到 所有 之后创建的 Vue 实例(包括第三方模板)。

## vuex

Vuex 应用的核心就是 store（仓库）。“store”基本上就是一个容器，它包含着你的应用中大部分的状态 (state)。有些同学可能会问，那我定义一个全局对象，再去上层封装了一些数据存取的接口不也可以么？

Vuex 和单纯的全局对象有以下两点不同：

* Vuex 的状态存储是响应式的。当 Vue 组件从 store 中读取状态的时候，若 store 中的状态发生变化，那么相应的组件也会相应地得到高效更新。
* 你不能直接改变 store 中的状态。改变 store 中的状态的唯一途径就是显式地提交 (commit) mutation。这样使得我们可以方便地跟踪每一个状态的变化，从而让我们能够实现一些工具帮助我们更好地了解我们的应用。

### dispatch与commit

dispatch分发的是action，action中可以包含异步操作，在action中commit。

commit提交mutation，且必须是同步操作

# 计算机网络

## 1、网络七层协议OSI

OSI是一个开放性的通信系统互连参考模型，他是一个定义得非常好的协议规范。OSI模型有7层结构，每层都可以有几个子层。

### 应用层

应用层向应用进程展示所有的网络服务。当一个应用进程访问网络时，通过该层执行所有的动作。

示例：TELNET，HTTP，FTP，NFS，SMTP等。

### 表示层

这一层的主要功能是定义数据格式及加密。

示例：加密，ASCII等。

### 会话层

它定义了如何开始、控制和结束一个会话，包括对多个双向消息的控制和管理，以便在只完成连续消息的一部分时可以通知应用，从而使表示层看到的数据是连续的，在某些情况下，如果表示层收到了所有的数据，则用数据代表表示层。

示例：RPC，SQL等。

### 传输层

这层的功能包括是否选择差错恢复协议还是无差错恢复协议，及在同一主机上对不同应用的数据流的输入进行复用，还包括对收到的顺序不对的数据包的重新排序功能。

示例：TCP，UDP，SPX。

#### 三次握手-建立TCP/IP连接

建立起一个TCP连接需要经过“三次握手”：

* 第一次握手：客户端发送连接请求报文段syn包(同步序列编号)(syn=j)到服务器，并进入SYN\_SEND状态，等待服务器确认；
* 第二次握手：服务器收到syn包，必须确认客户的SYN（ack=j+1），同时自己也发送一个SYN包（syn=k），即SYN+ACK(确认字符)包，此时服务器进入SYN\_RECV状态；
* 第三次握手：客户端收到服务器的SYN＋ACK包，向服务器发送确认包ACK(ack=k+1)，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。

握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。

另一种说法：

* 首先客户端创建 TCP 头部，头部包含目标服务器的端口号等，同时将头部的 SYN 设为 1，表示开始请求连接。TCP 头部创建好了之后，TCP 模块便将信息传递给 IP 模块并委托它发送，然后信息经过网络到达服务器的 IP 模块再到 TCP 模块，TCP 模块则会根据 TCP 头部的信息找到端口号对应的套接字，套接字则会写入相应的信息，然后将状态改为“正在连接”；
* 服务端的 TCP 模块收到连接请求后就要回应，与客户端一样， 需要在 TCP 头部设置发送方和接收方的端口号，以及将 SYN 设为 1，同时，返回响应时还要将 ACK 设为 1，表示已经接收到相应的包。接着，将信息打包好，发送给客户端；
* 客户端收到消息后，发现 SYN 为 1，则表示连接成功，所以在套接字中写入服务器的端口号，同时将状态改为连接完毕。为了告诉服务器收到消息，客户端也要将 ACK 设为 1，接着发送给服务端。

#### https握手过程：

<https://zhuanlan.zhihu.com/p/28946087>

#### 四次挥手-中断连接

* A的应用进程先向其TCP发出连接释放报文段（FIN=1，序号seq=u），并停止再发送数据，主动关闭TCP连接，进入FIN-WAIT-1（终止等待1）状态，等待B的确认。
* B收到连接释放报文段后即发出确认报文段，（ACK=1，确认号ack=u+1，序号seq=v），B进入CLOSE-WAIT（关闭等待）状态，此时的TCP处于半关闭状态，A到B的连接释放。
* A收到B的确认后，进入FIN-WAIT-2（终止等待2）状态，等待B发出的连接释放报文段。
* B没有要向A发出的数据，B发出连接释放报文段（FIN=1，ACK=1，序号seq=w，确认号ack=u+1），B进入LAST-ACK（最后确认）状态，等待A的确认。
* A收到B的连接释放报文段后，对此发出确认报文段（ACK=1，seq=u+1，ack=w+1），A进入TIME-WAIT（时间等待）状态。此时TCP未释放掉，需要经过时间等待计时器设置的时间2MSL后，A才进入CLOSED状态。

#### 区别

因为当Server端收到Client端的SYN连接请求报文后，可以直接发送SYN+ACK报文。其中ACK报文是用来应答的，SYN报文是用来同步的。但是关闭连接时，当Server端收到FIN报文时，很可能并不会立即关闭SOCKET，所以只能先回复一个ACK报文，告诉Client端，"你发的FIN报文我收到了"。只有等到我Server端所有的报文都发送完了，我才能发送FIN报文，因此不能一起发送。故需要四步握手。

### 网络层

这层对端到端的包传输进行定义，它定义了能够标识所有结点的逻辑地址，还定义了路由实现的方式和学习的方式。为了适应最大传输单元长度小于包长度的传输介质，网络层还定义了如何将一个包分解成更小的包的分段方法。

示例：IP，IPX等。

### 数据链路层

它定义了在单个链路上如何传输数据。

示例：ATM，FDDI等。

### 物理层

OSI的物理层规范是有关传输介质的特这些规范通常也参考了其他组织制定的标准。连接头、帧、帧的使用、电流、编码及光调制等都属于各种物理层规范中的内容。物理层常用多个规范完成对所有细节的定义。

示例：Rj45，802.3等。

# 设计模式

## 设计模式的类型

1. 创建型模式：

这些设计模式提供了一种在创建对象的同时隐藏创建逻辑的方式，而不是使用 new 运算符直接实例化对象。这使得程序在判断针对某个给定实例需要创建哪些对象时更加灵活。

* 工厂模式（Factory Pattern）
* 抽象工厂模式（Abstract Factory Pattern）
* 单例模式（Singleton Pattern）
* 建造者模式（Builder Pattern）
* 原型模式（Prototype Pattern）

1. 结构型模式

这些设计模式关注类和对象的组合。继承的概念被用来组合接口和定义组合对象获得新功能的方式。

* 适配器模式（Adapter Pattern）
* 桥接模式（Bridge Pattern）
* 过滤器模式（Filter、Criteria Pattern）
* 组合模式（Composite Pattern）
* 装饰器模式（Decorator Pattern）
* 外观模式（Facade Pattern）
* 享元模式（Flyweight Pattern）

代理模式（Proxy Pattern）

1. 行为型模式

这些设计模式特别关注对象之间的通信。

* 责任链模式（Chain of Responsibility Pattern）
* 命令模式（Command Pattern）
* 解释器模式（Interpreter Pattern）
* 迭代器模式（Iterator Pattern）
* 中介者模式（Mediator Pattern）
* 备忘录模式（Memento Pattern）
* 观察者模式（Observer Pattern）
* 状态模式（State Pattern）
* 空对象模式（Null Object Pattern）
* 策略模式（Strategy Pattern）
* 模板模式（Template Pattern）
* 访问者模式（Visitor Pattern）

1. J2EE 模式

这些设计模式特别关注表示层。这些模式是由 Sun Java Center 鉴定的。

* MVC 模式（MVC Pattern）
* 业务代表模式（Business Delegate Pattern）
* 组合实体模式（Composite Entity Pattern）
* 数据访问对象模式（Data Access Object Pattern）
* 前端控制器模式（Front Controller Pattern）
* 拦截过滤器模式（Intercepting Filter Pattern）
* 服务定位器模式（Service Locator Pattern）
* 传输对象模式（Transfer Object Pattern）

## 设计模式的六大原则

1. 开闭原则（Open Close Principle）：开闭原则的意思是：对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。简言之，是为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。
2. 里氏代换原则（Liskov Substitution Principle）：里氏代换原则是面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。LSP 是继承复用的基石，只有当派生类可以替换掉基类，且软件单位的功能不受到影响时，基类才能真正被复用，而派生类也能够在基类的基础上增加新的行为。里氏代换原则是对开闭原则的补充。实现开闭原则的关键步骤就是抽象化，而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。
3. 依赖倒转原则（Dependence Inversion Principle）：这个原则是开闭原则的基础，具体内容：针对接口编程，依赖于抽象而不依赖于具体。
4. 接口隔离原则（Interface Segregation Principle）：这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。它还有另外一个意思是：降低类之间的耦合度。由此可见，其实设计模式就是从大型软件架构出发、便于升级和维护的软件设计思想，它强调降低依赖，降低耦合。
5. 迪米特法则，又称最少知道原则（Demeter Principle）：最少知道原则是指：一个实体应当尽量少地与其他实体之间发生相互作用，使得系统功能模块相对独立。
6. 合成复用原则（Composite Reuse Principle）：合成复用原则是指：尽量使用合成/聚合的方式，而不是使用继承。
7. 单一职责原则：即一个类、一个方法应该只完成一件事

## 单利模式

var getSingle = function( fn ){

var result;

return function(){

return result || ( result = fn .apply(this, arguments ) );

}

};

## 策略模式

策略模式的定义是：定义一系列的算法，把它们一个个封装起来，并且使它们可以相互替换。

var strategies = {

"S": function( salary ){

return salary \* 4;

},

"A": function( salary ){

return salary \* 3;

},

"B": function( salary ){

return salary \* 2;

}

};

var calculateBonus = function( level, salary ){

return strategies[ level ]( salary );

};

优点：

* 策略模式利用组合、委托和多态等技术和思想，可以有效地避免多重条件选择语句。
* 策略模式提供了对开放—封闭原则的完美支持，将算法封装在独立的strategy 中，使得它们易于切换，易于理解，易于扩展。
* 策略模式中的算法也可以复用在系统的其他地方，从而避免许多重复的复制粘贴工作。
* 在策略模式中利用组合和委托来让Context 拥有执行算法的能力，这也是继承的一种更轻便的替代方案

缺点：

* 使用策略模式会在程序中增加许多策略类或者策略对象，但实际上这比把它们负责的逻辑堆砌在Context 中要好。
* 要使用策略模式，必须了解所有的strategy，必须了解各个strategy 之间的不同点，这样才能选择一个合适的strategy。

## 代理模式

代理模式是为一个对象提供一个代用品或占位符，以便控制对它的访问。

保护代理：

用于对象应该有不同访问权限的情况

在代理中过执行一些过滤

var Flower = function(){};

var xiaoming = {

sendFlower: function( target){

var flower = new Flower();

target.receiveFlower( flower );

}

};

var B = {

receiveFlower: function( flower ){

A.listenGoodMood(function(){ // 监听A 的好心情

A.receiveFlower( flower );

});

}

};

var A = {

receiveFlower: function( flower ){

console.log( '收到花 ' + flower );

},

listenGoodMood: function( fn ){

setTimeout(function(){ // 假设10 秒之后A 的心情变好

fn();

}, 10000 );

}

};

xiaoming.sendFlower( B );

虚拟代理：

只有在真正使用时才创建对象，减小开销

var B = {

receiveFlower: function( flower ){

A.listenGoodMood(function(){ // 监听A 的好心情

var flower = new Flower(); // 延迟创建flower 对象

A.receiveFlower( flower );

});

}

};

缓存代理：

var mult = function(){

console.log( '开始计算乘积' );

var a = 1;

for ( var i = 0, l = arguments.length; i < l; i++ ){

a = a \* arguments[i];

}

return a;

};

var proxyMult = (function(){

var cache = {};

return function(){

var args = Array.prototype.join.call( arguments, ',' );

if ( args in cache ){

return cache[ args ];

}

return cache[ args ] = mult.apply( this, arguments );

}

})();

proxyMult( 1, 2, 3, 4 );

## 迭代器模式

迭代器模式是指提供一种方法顺序访问一个聚合对象中的各个元素，而又不需要暴露该对象

的内部表示。迭代器模式可以把迭代的过程从业务逻辑中分离出来，在使用迭代器模式之后，即使不关心对象的内部构造，也可以按顺序访问其中的每个元素。

Array.prototype.forEach

## 发布—订阅模式(观察者模式)

它定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都将得到通知。

dom事件

## 享元模式

享元（flyweight）模式是一种用于性能优化的模式，“fly”在这里是苍蝇的意思，意为蝇量级。享元模式的核心是运用共享技术来有效支持大量细粒度的对象。

使用享元模式的关键是如何区别内部状态和外部状态。可以被对象共享的属性通常被划分为内部状态。而外部状态取决于具体的场景，并根据场景而变化。

## 中介者模式

中介者模式的作用就是解除对象与对象之间的紧耦合关系。增加一个中介者对象后，所有的相关对象都通过中介者对象来通信，而不是互相引用，所以当一个对象发生改变时，只需要通知中介者对象即可。中介者使各对象之间耦合松散，而且可以独立地改变它们之间的交互。

表单验证，解除各个表单项目之间的耦合。

## 装饰者模式

装饰者模式可以动态地给某个对象添加一些额外的职责，而不会影响从这个类中派生的其他对象。

## 适配器模式

适配器模式的作用是解决两个软件实体间的接口不兼容的问题。使用适配器模式之后，原本

由于接口不兼容而不能工作的两个软件实体可以一起工作。

适配器模式是一种“亡羊补牢”的模式，没有人会在程序的设计之初就使用它。因为没有人可以完全预料到未来的事情，也许现在好好工作的接口，未来的某天却不再适用于新系统，那么我们可以用适配器模式把旧接口包装成一个新的接口，使它继续保持生命力。

# webpack

## 1、webpack构建流程

* 解析webpack配置参数，合并从shell传入和webpack.config.js文件里配置的参数，生产最后的配置结果。
* 注册所有配置的插件，好让插件监听webpack构建生命周期的事件节点，以做出对应的反应。
* 从配置的entry入口文件开始解析文件构建AST语法树，找出每个文件所依赖的文件，递归下去。
* 在解析文件递归的过程中根据文件类型和loader配置找出合适的loader用来对文件进行转换。
* 递归完后得到每个文件的最终结果，根据entry配置生成代码块chunk。
* 输出所有chunk到文件系统。

## 2、webpack与gulp的区别

Gulp是任务运行工具

简单来说，任务运行程序是一个允许您自动执行可重复活动的应用程序。任务在JS文件中定义，并使用包含任务名称的命令执行。

Gulp侧重于前端开发的整个过程的控制管理（像是流水线），我们可以通过给gulp配置不通的task（通过Gulp中的gulp.task()方法配置，比如启动server、sass/less预编译、文件的合并压缩等等）来让gulp实现不同的功能，从而构建整个前端开发流程。

Webpack不是任务运行工具

相对Gulp来说， Webpack是一个模块构建工具。主要目的是帮助程序模块及其依赖构建静态资源。

Webpack有人也称之为模块打包机，由此也可以看出Webpack更侧重于模块打包，当然我们可以把开发中的所有资源（图片、js文件、css文件等）都可以看成模块，最初Webpack本身就是为前端JS代码打包而设计的，后来被扩展到其他资源的打包处理。Webpack是通过loader（加载器）和plugins（插件）对资源进行处理的。

## 3、webpack常用插件

1）webpack.DefinePlugin：允许创建一个在编译时可以配置的全局常量。定义全局常量

new webpack.DefinePlugin({

'process.env': {

NODE\_ENV: JSON.stringify(env)

}

});

2）ExtractTextPlugin：它会将所有的入口 chunk(entry chunks)中引用的 \*.css，移动到独立分离的 CSS 文件。分离css文件

3）HtmlWebpackPlugin 重构入口html，动态添加<link>和<script>，在以hash命名的文件上非常有用，因为每次编译都会改变

4）webpack.optimize.UglifyJsPlugin： js压缩 new webpack.optimize.UglifyJsPlugin([options])

5）webpack.optimize.CommonsChunkPlugin：通过将公共模块拆出来，最终合成的文件能够在最开始的时候加载一次，便存到缓存中供后续使用。 合并公共模块为单独文件，比如全局通用的js等，长期不会修改的，从而可以从缓存中取，优化网页性能。在v4版本中移除了。

6）ProvidePlugin 定义标识符，当遇到指定标识符的时候，自动加载模块。

new webpack.ProvidePlugin({

$: "jquery"

})

7）CleanWebpackPlugin // new CleanWebpackPlugin (['dist'])每次打包清空dist目录

8）HashedModuleIdsPlugin：插件会根据模块的相对路径生成一个四位数的hash作为模块id, 建议用于生产环境。//防止打包的文件因为解析顺序发生改变而导致文件hash改变

9）webpack.ContextReplacementPlugin：上下文替换插件(ContextReplacementPlugin) 允许你覆盖查找规则

new webpack.ContextReplacementPlugin(

resourceRegExp: RegExp,

newContentResource?: string,

newContentRecursive?: boolean,

newContentRegExp?: RegExp

)

## 4、webpack配置

* entry：起点或是应用程序的起点入口，// string | object | array
* output：打包的文件的位置
* path：打包后的文件存放的地方—**绝对路径**
* filename：打包后输出文件的文件名，每个输出 bundle 的名称。这些 bundle 将写入到 output.path 选项指定的目录下。
* filename: "bundle.js"，//对于单个入口起点，filename 会是一个静态名称。
* filename: "[name].js", // 当通过多个入口起点(entry point)、代码拆分(code splitting)或各种插件(plugin)创建多个 bundle，应该使用以下一种替换方式，来赋予每个 bundle 一个唯一的名称
* filename: '[id].bundle.js',// 使用内部 chunk id
* filename: '[name].[hash].bundle.js',// 使用每次构建过程中，唯一的 hash 生成
* filename: "[chunkhash].js", // 用于长效缓存, 基于每个 chunk 内容的 hash
* publicPath：按需加载或加载外部资源的url的前缀，如果填错，图片等会404
* chunkFilename：非入口(non-entry) chunk 文件的名称

|  |  |
| --- | --- |
| 模板 | 描述 |
| [hash] | 模块标识符(module identifier)的 hash |
| [chunkhash] | chunk 内容的 hash |
| [name] | 模块名称 |
| [id] | 模块标识符(module identifier) |
| [query] | 模块的 query，例如，文件名 ? 后面的字符串 |

[hash] 和 [chunkhash] 的长度可以使用 [hash:16]（默认为20）来指定。或者，通过指定output.hashDigestLength 在全局配置长度。

* module：如何处理项目中的不同类型的模块。
* rules：创建模块时，匹配请求的规则数组。这些规则能够修改模块的创建方式。这些规则能够对模块(module)应用 loader，或者修改解析器(parser)。
* one of：规则数组，当规则匹配时，只使用第一个匹配规则。
* test：条件
* use：使用的loader
* plugins：数组，webpack 插件列表，用于以各种方式自定义 webpack 构建过程。
* optimization：优化相关
  + minimize：压缩，为true时，默认使用UglifyjsWebpackPlugin
  + minimizer：数组，可以修改使用的压缩插件

[new OptimizeCSSAssetsPlugin({ /\* your config \*/ }),

new UglifyJsPlugin({ /\* your config \*/ })]

* + runtimeChunk：’single’ 创建单个运行时 bundle
  + splitChunks：代码分离
    - cacheGroups：将第三方库(library)（例如 lodash 或 react）提取到单独的 vendor chunk 文件中，是比较推荐的做法，这是因为，它们很少像本地的源代码那样频繁修改。因此通过实现以上步骤，利用客户端的长效缓存机制，可以通过命中缓存来消除请求，并减少向服务器获取资源，同时还能保证客户端代码和服务器端代码版本一致。

vendor: {

test: /[\\/]node\_modules[\\/]/,

name: 'vendors',

chunks: 'all'

}

* resolve：这些选项能设置模块如何被解析。
  + alias：创建 import 或 require 的别名，来确保模块引入变得更简单。

// utils: path.join(\_\_dirname, '../src/utils'),

也可以在给定对象的键后的末尾添加 $，以表示精准匹配：

// 'vue$': 'vue/dist/vue.common.js',

* extensions：自动解析确定的扩展。能够使用户在引入模块时不带扩展

// extensions: ['.js', '.vue', '.json'], import File from ‘/src/index’; (index.js)

* modules：告诉 webpack 解析模块时应该搜索的目录。

// modules: ['node\_modules'] 默认值

* plugins：应该使用的额外的解析插件列表。
* externals：配置选项提供了「从输出的 bundle 中排除依赖」的方法。

externals: {

jquery: 'jQuery'

}

# AMD、CMD与commonjs

## 1、commonjs

CommonJS规范加载模块是同步的，也就是说，只有加载完成，才能执行后面的操作。CommonJs 是服务器端模块的规范，Node.js采用了这个规范。根据CommonJS规范，一个单独的文件就是一个模块。加载模块使用require方法，该方法读取一个文件并执行，最后返回文件内部的exports对象。CommonJS 加载模块是同步的，所以只有加载完成才能执行后面的操作。

CommonJS模块的加载机制是，输入的是被输出的值的拷贝。也就是说，一旦输出一个值，模块内部的变化就影响不到这个值。

### require的内部处理流程：

1. 检查 Module.\_cache，是否缓存之中有指定模块

2. 如果缓存之中没有，就创建一个新的Module实例

3. 将它保存到缓存

4. 使用 module.load() 加载指定的模块文件，读取文件内容之后，使用 module.compile() 执行文件代码

5. 如果加载/解析过程报错，就从缓存删除该模块

6. 返回该模块的 module.exports

## 2、AMD

AMD规范则是非同步加载模块，允许指定回调函数。AMD规范采用异步方式加载模块。它的模块支持对象 函数 构造器 字符串 JSON等各种类型的模块。适用AMD规范适用define方法定义模块。

通过数组引入依赖 ，回调函数通过形参传入依赖

define(['someModule1', ‘someModule2’], function (someModule1, someModule2) {

function foo () {

/// someing

someModule1.test();

}

return {foo: foo}

});

## 3、CMD

同AMD一样是异步加载模块，只是AMD是RequireJS 在推广过程中对模块定义的规范化产出，而CMD 是 SeaJS 在推广过程中对模块定义的规范化产出。

## 4、ES6 Module

ES6 在语言标准的层面上，实现了模块功能，而且实现得相当简单，旨在成为浏览器和服务器通用的模块解决方案。其模块功能主要由两个命令构成：export和import。export命令用于规定模块的对外接口，import命令用于输入其他模块提供的功能。

## 5、AMD与CMD区别

1）对于依赖的模，AMD 是提前执行，CMD 是延迟执行。

2）CMD 推崇依赖就近，AMD 推崇依赖前置。

// CMD

define(function(require, exports, module) {

var a = require('./a')

a.doSomething()

// 此处略去 100 行

var b = require('./b') // 依赖可以就近书写

b.doSomething()

// ...

})

// AMD 默认推荐的是

define(['./a', './b'], function(a, b) {

// 依赖必须一开始就写好

a.doSomething()

// 此处略去 100 行

b.doSomething()

...

})

3）AMD加载完模块后，就立马执行该模块；CMD加载完某个模块后没有立即执行而是等到遇到require语句的时再执行。

## 6、ES6模块与Commonjs模块的差异

### 1. CommonJS 模块输出的是一个值的拷贝，ES6 模块输出的是值的引用。

* CommonJS 模块输出的是值的拷贝，也就是说，一旦输出一个值，模块内部的变化就影响不到这个值。
* ES6 模块的运行机制与 CommonJS 不一样。JS 引擎对脚本静态分析的时候，遇到模块加载命令import，就会生成一个只读引用。等到脚本真正执行时，再根据这个只读引用，到被加载的那个模块里面去取值。换句话说，ES6 的import有点像 Unix 系统的“符号连接”，原始值变了，import加载的值也会跟着变。因此，ES6 模块是动态引用，并且不会缓存值，模块里面的变量绑定其所在的模块。

### 2. CommonJS 模块是运行时加载，ES6 模块是编译时输出接口。

* 运行时加载: CommonJS 模块就是对象；即在输入时是先加载整个模块，生成一个对象，然后再从这个对象上面读取方法，这种加载称为“运行时加载”。
* 编译时加载: ES6 模块不是对象，而是通过 export 命令显式指定输出的代码，import时采用静态命令的形式。即在import时可以指定加载某个输出值，而不是加载整个模块，这种加载称为“编译时加载”。

# css

## 1、盒模型

盒子中的区域

一个盒子中主要的属性就5个：width、height、padding、border、margin。如下：

width和height：内容的宽度、高度（不是盒子的宽度、高度）。

padding：内边距。

border：边框。

margin：外边距。

标准盒子模型和IE盒子模型

在 CSS的标准盒子模型中，width 和 height 指的是内容区域的宽度和高度。增加内边距、边框和外边距不会影响内容区域的尺寸，但是会增加元素框的总尺寸。IE盒子模型中，width 和 height 指的是内容区域+border+padding的宽度和高度。

## 2、选择器

### 1、优先级

!important > 行内样式>ID选择器 > 类选择器 > 标签 > 通配符 > 继承 > 浏览器默认属性

### 2、选择器

|  |  |  |  |
| --- | --- | --- | --- |
| 选择器 | 例子 | 例子描述 | CSS |
| [.class](http://www.w3school.com.cn/cssref/selector_class.asp) | .intro | 选择 class="intro" 的所有元素。 | 1 |
| [#id](http://www.w3school.com.cn/cssref/selector_id.asp) | #firstname | 选择 id="firstname" 的所有元素。 | 1 |
| [\*](http://www.w3school.com.cn/cssref/selector_all.asp) | \* | 选择所有元素。 | 2 |
| [element](http://www.w3school.com.cn/cssref/selector_element.asp) | p | 选择所有 <p> 元素。 | 1 |
| [element,element](http://www.w3school.com.cn/cssref/selector_element_comma.asp) | div,p | 选择所有 <div> 元素和所有 <p> 元素。 | 1 |
| [element element](http://www.w3school.com.cn/cssref/selector_element_element.asp) | div p | 选择 <div> 元素内部的所有 <p> 元素。 | 1 |
| [element>element](http://www.w3school.com.cn/cssref/selector_element_gt.asp) | div>p | 选择父元素为 <div> 元素的所有 <p> 元素。 | 2 |
| [element+element](http://www.w3school.com.cn/cssref/selector_element_plus.asp) | div+p | 选择紧接在 <div> 元素之后的所有 <p> 元素。 | 2 |
| [[attribute]](http://www.w3school.com.cn/cssref/selector_attribute.asp) | [target] | 选择带有 target 属性所有元素。 | 2 |
| [[attribute=value]](http://www.w3school.com.cn/cssref/selector_attribute_value.asp) | [target=\_blank] | 选择 target="\_blank" 的所有元素。 | 2 |
| [[attribute~=value]](http://www.w3school.com.cn/cssref/selector_attribute_value_contain.asp) | [title~=flower] | 选择 title 属性包含单词 "flower" 的所有元素。 | 2 |
| [[attribute|=value]](http://www.w3school.com.cn/cssref/selector_attribute_value_start.asp) | [lang|=en] | 选择 lang 属性值以 "en" 开头的所有元素。 | 2 |
| [:link](http://www.w3school.com.cn/cssref/selector_link.asp) | a:link | 选择所有未被访问的链接。 | 1 |
| [:visited](http://www.w3school.com.cn/cssref/selector_visited.asp) | a:visited | 选择所有已被访问的链接。 | 1 |
| [:active](http://www.w3school.com.cn/cssref/selector_active.asp) | a:active | 选择活动链接。 | 1 |
| [:hover](http://www.w3school.com.cn/cssref/selector_hover.asp) | a:hover | 选择鼠标指针位于其上的链接。 | 1 |
| [:focus](http://www.w3school.com.cn/cssref/selector_focus.asp) | input:focus | 选择获得焦点的 input 元素。 | 2 |
| [:first-letter](http://www.w3school.com.cn/cssref/selector_first-letter.asp) | p:first-letter | 选择每个 <p> 元素的首字母。 | 1 |
| [:first-line](http://www.w3school.com.cn/cssref/selector_first-line.asp) | p:first-line | 选择每个 <p> 元素的首行。 | 1 |
| [:first-child](http://www.w3school.com.cn/cssref/selector_first-child.asp) | p:first-child | 选择属于父元素的第一个子元素的每个 <p> 元素。 | 2 |
| [:before](http://www.w3school.com.cn/cssref/selector_before.asp) | p:before | 在每个 <p> 元素的内容之前插入内容。 | 2 |
| [:after](http://www.w3school.com.cn/cssref/selector_after.asp) | p:after | 在每个 <p> 元素的内容之后插入内容。 | 2 |
| [:lang(language)](http://www.w3school.com.cn/cssref/selector_lang.asp) | p:lang(it) | 选择带有以 "it" 开头的 lang 属性值的每个 <p> 元素。 | 2 |
| [element1~element2](http://www.w3school.com.cn/cssref/selector_gen_sibling.asp) | p~ul | 选择前面有 <p> 元素的每个 <ul> 元素。 | 3 |
| [[attribute^=value]](http://www.w3school.com.cn/cssref/selector_attr_begin.asp) | a[src^="https"] | 选择其 src 属性值以 "https" 开头的每个 <a> 元素。 | 3 |
| [[attribute$=value]](http://www.w3school.com.cn/cssref/selector_attr_end.asp) | a[src$=".pdf"] | 选择其 src 属性以 ".pdf" 结尾的所有 <a> 元素。 | 3 |
| [[attribute\*=value]](http://www.w3school.com.cn/cssref/selector_attr_contain.asp) | a[src\*="abc"] | 选择其 src 属性中包含 "abc" 子串的每个 <a> 元素。 | 3 |
| [:first-of-type](http://www.w3school.com.cn/cssref/selector_first-of-type.asp) | p:first-of-type | 选择属于其父元素的首个 <p> 元素的每个 <p> 元素。 | 3 |
| [:last-of-type](http://www.w3school.com.cn/cssref/selector_last-of-type.asp) | p:last-of-type | 选择属于其父元素的最后 <p> 元素的每个 <p> 元素。 | 3 |
| [:only-of-type](http://www.w3school.com.cn/cssref/selector_only-of-type.asp) | p:only-of-type | 选择属于其父元素唯一的 <p> 元素的每个 <p> 元素。 | 3 |
| [:only-child](http://www.w3school.com.cn/cssref/selector_only-child.asp) | p:only-child | 选择属于其父元素的唯一子元素的每个 <p> 元素。 | 3 |
| [:nth-child(n)](http://www.w3school.com.cn/cssref/selector_nth-child.asp) | p:nth-child(2) | 选择属于其父元素的第二个子元素的每个 <p> 元素。 | 3 |
| [:nth-last-child(n)](http://www.w3school.com.cn/cssref/selector_nth-last-child.asp) | p:nth-last-child(2) | 同上，从最后一个子元素开始计数。 | 3 |
| [:nth-of-type(n)](http://www.w3school.com.cn/cssref/selector_nth-of-type.asp) | p:nth-of-type(2) | 选择属于其父元素第二个 <p> 元素的每个 <p> 元素。 | 3 |
| [:nth-last-of-type(n)](http://www.w3school.com.cn/cssref/selector_nth-last-of-type.asp) | p:nth-last-of-type(2) | 同上，但是从最后一个子元素开始计数。 | 3 |
| [:last-child](http://www.w3school.com.cn/cssref/selector_last-child.asp) | p:last-child | 选择属于其父元素最后一个子元素每个 <p> 元素。 | 3 |
| [:root](http://www.w3school.com.cn/cssref/selector_root.asp) | :root | 选择文档的根元素。 | 3 |
| [:empty](http://www.w3school.com.cn/cssref/selector_empty.asp) | p:empty | 选择没有子元素的每个 <p> 元素（包括文本节点）。 | 3 |
| [:target](http://www.w3school.com.cn/cssref/selector_target.asp) | #news:target | 选择当前活动的 #news 元素。 | 3 |
| [:enabled](http://www.w3school.com.cn/cssref/selector_enabled.asp) | input:enabled | 选择每个启用的 <input> 元素。 | 3 |
| [:disabled](http://www.w3school.com.cn/cssref/selector_disabled.asp) | input:disabled | 选择每个禁用的 <input> 元素 | 3 |
| [:checked](http://www.w3school.com.cn/cssref/selector_checked.asp) | input:checked | 选择每个被选中的 <input> 元素。 | 3 |
| [:not(selector)](http://www.w3school.com.cn/cssref/selector_not.asp) | :not(p) | 选择非 <p> 元素的每个元素。 | 3 |
| [::selection](http://www.w3school.com.cn/cssref/selector_selection.asp) | ::selection | 选择被用户选取的元素部分。 | 3 |

## 3、position

|  |  |
| --- | --- |
| 值 | 描述 |
| absolute | 生成绝对定位的元素，相对于 static 定位以外的第一个父元素进行定位。  元素的位置通过 "left", "top", "right" 以及 "bottom" 属性进行规定。 |
| fixed | 生成绝对定位的元素，相对于浏览器窗口进行定位。  元素的位置通过 "left", "top", "right" 以及 "bottom" 属性进行规定。 |
| relative | 生成相对定位的元素，相对于其正常位置进行定位。  因此，"left:20" 会向元素的 LEFT 位置添加 20 像素。 |
| static | 默认值。没有定位，元素出现在正常的流中（忽略 top, bottom, left, right 或者 z-index 声明）。 |
| inherit | 规定应该从父元素继承 position 属性的值。 |

## 4、display

|  |  |
| --- | --- |
| 值 | 描述 |
| none | 此元素不会被显示。 |
| block | 此元素将显示为块级元素，此元素前后会带有换行符。 |
| inline | 默认。此元素会被显示为内联元素，元素前后没有换行符。 |
| inline-block | 行内块元素。（CSS2.1 新增的值） |
| list-item | 此元素会作为列表显示。 |
| run-in | 此元素会根据上下文作为块级元素或内联元素显示。 |
| compact | CSS 中有值 compact，不过由于缺乏广泛支持，已经从 CSS2.1 中删除。 |
| marker | CSS 中有值 marker，不过由于缺乏广泛支持，已经从 CSS2.1 中删除。 |
| table | 此元素会作为块级表格来显示（类似 <table>），表格前后带有换行符。 |
| inline-table | 此元素会作为内联表格来显示（类似 <table>），表格前后没有换行符。 |
| table-row-group | 此元素会作为一个或多个行的分组来显示（类似 <tbody>）。 |
| table-header-group | 此元素会作为一个或多个行的分组来显示（类似 <thead>）。 |
| table-footer-group | 此元素会作为一个或多个行的分组来显示（类似 <tfoot>）。 |
| table-row | 此元素会作为一个表格行显示（类似 <tr>）。 |
| table-column-group | 此元素会作为一个或多个列的分组来显示（类似 <colgroup>）。 |
| table-column | 此元素会作为一个单元格列显示（类似 <col>） |
| table-cell | 此元素会作为一个表格单元格显示（类似 <td> 和 <th>） |
| table-caption | 此元素会作为一个表格标题显示（类似 <caption>） |
| inherit | 规定应该从父元素继承 display 属性的值。 |

## 5、flex

Flex 是 Flexible Box 的缩写，意为"弹性布局"，用来为盒状模型提供最大的灵活性。设为 Flex 布局以后，子元素的float、clear和vertical-align属性将失效。

### 1、基本概念

采用 Flex 布局的元素，称为 Flex 容器（flex container），简称"容器"。它的所有子元素自动成为容器成员，称为 Flex 项目（flex item），简称"项目"。

容器默认存在两根轴：水平的主轴（main axis）和垂直的交叉轴（cross axis）。主轴的开始位置（与边框的交叉点）叫做main start，结束位置叫做main end；交叉轴的开始位置叫做cross start，结束位置叫做cross end。

项目默认沿主轴排列。单个项目占据的主轴空间叫做main size，占据的交叉轴空间叫做cross size。

### 2、容器的属性

以下6个属性设置在容器上。

* flex-direction
* flex-wrap
* flex-flow
* justify-content
* align-items
* align-content

#### flex-direction

flex-direction属性决定主轴的方向（即项目的排列方向）。

flex-direction: row | row-reverse | column | column-reverse;

它可能有4个值。

* row（默认值）：主轴为水平方向，起点在左端。
* row-reverse：主轴为水平方向，起点在右端。
* column：主轴为垂直方向，起点在上沿。
* column-reverse：主轴为垂直方向，起点在下沿。

#### flex-wrap

默认情况下，项目都排在一条线（又称"轴线"）上。flex-wrap属性定义，如果一条轴线排不下，如何换行。

flex-wrap: nowrap | wrap | wrap-reverse;

它可能取三个值。

* nowrap（默认）：不换行。item的大小会自动缩放使显示在一行
* wrap：换行，第一行在上方。
* wrap-reverse：换行，第一行在下方。

#### flex-flow

flex-flow: <flex-direction> || <flex-wrap>;

flex-flow属性是flex-direction属性和flex-wrap属性的简写形式，默认值为row nowrap。

#### justify-content

justify-content属性定义了项目在主轴上的对齐方式。

justify-content: flex-start | flex-end | center | space-between | space-around;

它可能取5个值，具体对齐方式与轴的方向有关。下面假设主轴为从左到右。

* flex-start（默认值）：左对齐
* flex-end：右对齐
* center： 居中
* space-between：两端对齐，项目之间的间隔都相等。两端没有空隙
* space-around：每个项目两侧的间隔相等。所以，项目之间的间隔比项目与边框的间隔大一倍。两端的空隙和项目之间的一样

#### align-items

align-items属性定义项目在交叉轴上如何对齐。

align-items: flex-start | flex-end | center | baseline | stretch;

它可能取5个值。具体的对齐方式与交叉轴的方向有关，下面假设交叉轴从上到下。

* flex-start：交叉轴的起点对齐。
* flex-end：交叉轴的终点对齐。
* center：交叉轴的中点对齐。
* baseline: 项目的第一行文字的基线对齐。
* stretch（默认值）：如果项目未设置高度或设为auto，将占满整个容器的高度。

#### align-content

align-content属性定义了多根轴线的对齐方式。如果项目只有一根轴线，该属性不起作用。

align-content: flex-start | flex-end | center | space-between | space-around | stretch;

该属性可能取6个值。

* flex-start：与交叉轴的起点对齐。
* flex-end：与交叉轴的终点对齐。
* center：与交叉轴的中点对齐。
* space-between：与交叉轴两端对齐，轴线之间的间隔平均分布。
* space-around：每根轴线两侧的间隔都相等。所以，轴线之间的间隔比轴线与边框的间隔大一倍。
* stretch（默认值）：轴线占满整个交叉轴。

### 3、项目的属性

以下6个属性设置在项目上。

* order
* flex-grow
* flex-shrink
* flex-basis
* flex
* align-self

#### order

order属性定义项目的排列顺序。数值越小，排列越靠前，默认为0。

#### flex-grow

flex-grow属性定义项目的放大比例，默认为0，即如果存在剩余空间，也不放大。

如果所有项目的flex-grow属性都为1，则它们将等分剩余空间（如果有的话）。如果一个项目的flex-grow属性为2，其他项目都为1，则前者占据的剩余空间将比其他项多一倍。

#### flex-shrink

flex-shrink属性定义了项目的缩小比例，默认为1，即如果空间不足，该项目将缩小。

如果所有项目的flex-shrink属性都为1，当空间不足时，都将等比例缩小。如果一个项目的flex-shrink属性为0，其他项目都为1，则空间不足时，前者不缩小。

负值对该属性无效。

#### flex-basis

flex-basis属性定义了在分配多余空间之前，项目占据的主轴空间（main size）。浏览器根据这个属性，计算主轴是否有多余空间。它的默认值为auto，即项目的本来大小。

#### flex

flex属性是flex-grow, flex-shrink 和 flex-basis的简写，默认值为0 1 auto。后两个属性可选。

flex: none | [ <'flex-grow'> <'flex-shrink'>? || <'flex-basis'> ]

#### align-self

align-self属性允许单个项目有与其他项目不一样的对齐方式，可覆盖align-items属性。默认值为auto，表示继承父元素的align-items属性，如果没有父元素，则等同于stretch。

## 6、清除浮动

### 1、使用带clear属性的空元素

在浮动元素后使用一个空元素如<div class="clear"></div>，并在CSS中赋予.clear{clear:both;}属性即可清理浮动。亦可使用<br class="clear" />或<hr class="clear" />来进行清理。

优点：简单，代码少，浏览器兼容性好。

缺点：需要添加大量无语义的html元素，代码不够优雅，后期不容易维护。

原理是这个空的div能让父级获取到高度，是因为设置之后这个div的左右都不能有浮动元素，所以这个空的div元素会向下移动，直到换行，而为了让这个div能够换行，父元素至少要包含浮动元素的高度才能提高足够的空间，这样实现清除浮动的影响。

### 2、使用CSS的overflow属性.

给浮动元素的容器添加overflow:hidden;或overflow:auto;可以清除浮动，另外在 IE6 中还需要触发 hasLayout ，例如为父元素设置容器宽高或设置 zoom:1。在添加overflow属性后，浮动元素又回到了容器层，把容器高度撑起，达到了清理浮动的效果。

原理：

定义：BFC（Block Formatting Context）全称是块级格式化上下文，用于对块级元素排版，默认情况下只有根元素（body）一个块级上下文，但是如果一个块级元素设置了float:left,overflow:hidden或position:absolute样式，就会为这个块级元素生产一个独立的块级上下文，使这个块级元素内部的排版完全独立。

作用：独立的块级上下文可以包裹浮动流，全部浮动子元素也不会引起容器高度塌陷，就是说包含块会把浮动元素的高度也计算在内，所以就不用清除浮动来撑起包含块的高度。

### 3、给浮动的元素的容器添加浮动

给浮动元素的容器也添加上浮动属性即可清除内部浮动，但是这样会使其整体浮动，影响布局，不推荐使用。

### 4、使用邻接元素处理

什么都不做，给浮动元素后面的元素添加clear属性。

### 5、使用CSS的:after伪元素

结合 :after 伪元素（注意这不是伪类，而是伪元素，代表一个元素之后最近的元素）和 IEhack ，可以完美兼容当前主流的各大浏览器，这里的 IEhack 指的是触发 hasLayout。

给浮动元素的容器添加一个clearfix的class，然后给这个class添加一个:after伪元素实现元素末尾添加一个看不见的块元素（Block element）清理浮动。

## 7、css3新特性

* border-radius：用于添加圆角效果
* box-shadow：用于添加阴影效果
* box-shadow: h-shadow v-shadow blur spread color inset;
* h-shadow：必需。水平阴影的位置。允许负值。
* v-shadow：必需。垂直阴影的位置。允许负值。
* blur：可选。模糊距离。
* spread：可选。阴影的尺寸。
* color：可选。阴影的颜色
* inset：可选。将外部阴影 (outset) 改为内部阴影。
* background-size：设置背景图片大小
* background-origin：指定背景图片background-image 属性的原点位置的背景相对区域,当使用 background-attachment 为fixed时，该属性将被忽略不起作用
* padding-box：从padding区域（含padding）开始显示背景图像。
* border-box： 从border区域（含border）开始显示背景图像。
* content-box：从content区域开始显示背景图像
* text-shadow：为文字添加阴影
* word-wrap：浏览器是否允许单词中断换行
* @font-face：能够加载服务器端的字体文件，让客户端显示客户端所没有安装的字体。
* animation：检索或设置对象所应用的动画特效。
* @keyframes：用于创建动画
* flex布局：Flex元素是可以让你的布局根据浏览器的大小变化进行自动伸缩。
* box-sizing：用来改变默认的 CSS 盒模型 对元素宽高的计算方式
* transform：旋转,缩放,定位,倾斜
* @media：媒体查询

## 8、动画

### transition—过渡

CSS3 过渡是元素从一种样式逐渐改变为另一种的效果。

|  |  |  |
| --- | --- | --- |
| 属性 | 描述 | CSS |
| [transition](http://www.w3school.com.cn/cssref/pr_transition.asp) | 简写属性，用于在一个属性中设置四个过渡属性。 | 3 |
| [transition-property](http://www.w3school.com.cn/cssref/pr_transition-property.asp) | 规定应用过渡的 CSS 属性的名称。 | 3 |
| [transition-duration](http://www.w3school.com.cn/cssref/pr_transition-duration.asp) | 定义过渡效果花费的时间。默认是 0。 | 3 |
| [transition-timing-function](http://www.w3school.com.cn/cssref/pr_transition-timing-function.asp) | 规定过渡效果的时间曲线。默认是 "ease"。 | 3 |
| [transition-delay](http://www.w3school.com.cn/cssref/pr_transition-delay.asp) | 规定过渡效果何时开始。默认是 0。 | 3 |

transition的局限：

transition的优点在于简单易用，但是它有几个很大的局限。

（1）transition需要事件触发，所以没法在网页加载时自动发生。

（2）transition是一次性的，不能重复发生，除非一再触发。

（3）transition只能定义开始状态和结束状态，不能定义中间状态，也就是说只有两个状态。

（4）一条transition规则，只能定义一个属性的变化，不能涉及多个属性。

### Animation

动画是使元素从一种样式逐渐变化为另一种样式的效果。

您可以改变任意多的样式任意多的次数。

请用百分比来规定变化发生的时间，或用关键词 "from" 和 "to"，等同于 0% 和 100%。

0% 是动画的开始，100% 是动画的完成。

为了得到最佳的浏览器支持，您应该始终定义 0% 和 100% 选择器。

animation属性：

* animation-name：规定 @keyframes 动画的名称。
* animation-duration：规定动画完成一个周期所花费的秒或毫秒。默认是 0。
* animation-timing-function：规定动画的速度曲线。默认是 "ease"。
* linear 动画从头到尾的速度是相同的。 测试
* ease 默认。动画以低速开始，然后加快，在结束前变慢。 测试
* ease-in 动画以低速开始。 测试
* ease-out 动画以低速结束。 测试
* ease-in-out 动画以低速开始和结束。 测试
* cubic-bezier(n,n,n,n) 在 cubic-bezier 函数中自己的值。可能的值是从 0 到 1 的数值。
* animation-delay：规定动画何时开始。默认是 0。
* animation-iteration-count：规定动画被播放的次数。默认是 1。
* n 定义动画播放次数的数值。
* infinite 规定动画应该无限次播放。
* animation-direction：规定动画是否在下一周期逆向地播放。默认是 "normal"。
* normal 默认值。动画应该正常播放。
* alternate 动画应该轮流反向播放。
* animation-play-state：规定动画是否正在运行或暂停。默认是 "running"。
* animation-fill-mode：规定对象动画时间之外的状态。
* none 不改变默认行为。
* forwards 当动画完成后，保持最后一个属性值（在最后一个关键帧中定义）。
* backwards 在 animation-delay 所指定的一段时间内，在动画显示之前，应用开始属性值（在第一个关键帧中定义）。
* both 向前和向后填充模式都被应用。

先，CSS Animation需要指定动画一个周期持续的时间，以及动画效果的名称。

div:hover {

animation: 1s rainbow;

}

@keyframes 定义动画

@keyframes rainbow {

0% { background: #c00; }

50% { background: orange; }

100% { background: yellowgreen; }

}

同transition一样，animation也是一个简写形式。

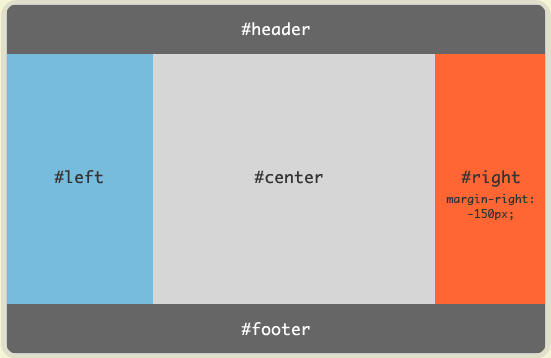
div:hover {

animation: 1s 1s rainbow linear 3 forwards normal;

}

## 9、圣杯布局

圣杯布局（Holy Grail Layout）指的是一种最常见的网站布局。页面从上到下，分成三个部分：头部（header），躯干（body），尾部（footer）。其中躯干又水平分成三栏，从左到右为：导航、主栏、副栏。



### flex实现：

HTML代码如下。

<body class="HolyGrail">

<header>...</header>

<div class="HolyGrail-body">

<main class="HolyGrail-content">...</main>

<nav class="HolyGrail-nav">...</nav>

<aside class="HolyGrail-ads">...</aside>

</div>

<footer>...</footer>

</body>

CSS代码如下。

.HolyGrail {

display: flex;

min-height: 100vh;

flex-direction: column;

}

header,

footer {

flex: 1;

}

.HolyGrail-body {

display: flex;

flex: 1;

}

.HolyGrail-content {

flex: 1;

}

.HolyGrail-nav, .HolyGrail-ads {

/\* 两个边栏的宽度设为12em \*/

flex: 0 0 12em;

}

.HolyGrail-nav {

/\* 导航放到最左边 \*/

order: -1;

}

### 其它实现：

.content-box {

position: relative;

padding: 0 200px;

overflow: hidden;

}

.content {

width: 100%;height: 200px; background-color: deeppink;float:left;

}

.nav-left {

position: relative;

left: -200px;

margin-left: -100%;

width: 200px;

height: 200px;

background-color: blue;

float: left;

}

.nav-right {

position: relative;

right: -200px;

width: 200px;

height: 200px;

background-color: darkorchid;

float:left;

margin-left:-200px;

}

## 属性继承

### 一、无继承性的属性

1、display：规定元素应该生成的框的类型

2、文本属性：

vertical-align：垂直文本对齐

text-decoration：规定添加到文本的装饰

text-shadow：文本阴影效果

white-space：空白符的处理

unicode-bidi：设置文本的方向

3、盒子模型的属性：width、height、margin 、margin-top、margin-right、margin-bottom、margin-left、border、border-style、border-top-style、border-right-style、border-bottom-style、border-left-style、border-width、border-top-width、border-right-right、border-bottom-width、border-left-width、border-color、border-top-color、border-right-color、border-bottom-color、border-left-color、border-top、border-right、border-bottom、border-left、padding、padding-top、padding-right、padding-bottom、padding-left

4、背景属性：background、background-color、background-image、background-repeat、background-position、background-attachment

5、定位属性：float、clear、position、top、right、bottom、left、min-width、min-height、max-width、max-height、overflow、clip、z-index

6、生成内容属性：content、counter-reset、counter-increment

7、轮廓样式属性：outline-style、outline-width、outline-color、outline

8、页面样式属性：size、page-break-before、page-break-after

9、声音样式属性：pause-before、pause-after、pause、cue-before、cue-after、cue、play-during

### 二、有继承性的属性

1、字体系列属性

font：组合字体

font-family：规定元素的字体系列

font-weight：设置字体的粗细

font-size：设置字体的尺寸

font-style：定义字体的风格

font-variant：设置小型大写字母的字体显示文本，这意味着所有的小写字母均会被转换为大写，但是所有使用小型大写字体的字母与其余文本相比，其字体尺寸更小。

font-stretch：对当前的 font-family 进行伸缩变形。所有主流浏览器都不支持。

font-size-adjust：为某个元素规定一个 aspect 值，这样就可以保持首选字体的 x-height。

2、文本系列属性

text-indent：文本缩进

text-align：文本水平对齐

line-height：行高

word-spacing：增加或减少单词间的空白（即字间隔）

letter-spacing：增加或减少字符间的空白（字符间距）

text-transform：控制文本大小写

direction：规定文本的书写方向

color：文本颜色

3、元素可见性：visibility

4、表格布局属性：caption-side、border-collapse、border-spacing、empty-cells、table-layout

5、列表布局属性：list-style-type、list-style-image、list-style-position、list-style

6、生成内容属性：quotes

7、光标属性：cursor

8、页面样式属性：page、page-break-inside、windows、orphans

9、声音样式属性：speak、speak-punctuation、speak-numeral、speak-header、speech-rate、volume、voice-family、pitch、pitch-range、stress、richness、、azimuth、elevation

### 三、所有元素可以继承的属性

1、元素可见性：visibility

2、光标属性：cursor

## bfc-块级格式化上下文

一个块格式化上下文由以下之一创建：

* 根元素或其它包含它的元素
* 浮动元素 (元素的 float 不是 none)
* 绝对定位元素 (元素具有 position 为 absolute 或 fixed)
* 内联块 (元素具有 display: inline-block)
* 表格单元格 (元素具有 display: table-cell，HTML表格单元格默认属性)
* 表格标题 (元素具有 display: table-caption, HTML表格标题默认属性)
* 具有overflow 且值不是 visible 的块元素，
* display: flow-root
* column-span: all 应当总是会创建一个新的格式化上下文，即便具有 column-span: all 的元素并不被包裹在一个多列容器中。
* 一个块格式化上下文包括创建它的元素内部所有内容，除了被包含于创建新的块级格式化上下文的后代元素内的元素。

块格式化上下文对于定位与清除浮动很重要。定位和清除浮动的样式规则只适用于处于同一块格式化上下文内的元素。浮动不会影响其它块格式化上下文中元素的布局，并且清除浮动只能清除同一块格式化上下文中在它前面的元素的浮动。

**BFC 特性(功能)**

使 BFC 内部浮动元素不会到处乱跑；

和浮动元素产生边界

## 层叠上下文

文档中的层叠上下文由满足以下任意一个条件的元素形成：

* 根元素 (HTML),
* z-index 值不为 "auto"的 绝对/相对定位，
* 一个 z-index 值不为 "auto"的 flex 项目 (flex item)，即：父元素 display: flex|inline-flex，
* opacity 属性值小于 1 的元素（参考 the specification for opacity），
* transform 属性值不为 "none"的元素，
* mix-blend-mode 属性值不为 "normal"的元素，
* filter值不为“none”的元素，
* perspective值不为“none”的元素，
* isolation 属性被设置为 "isolate"的元素，
* position: fixed
* 在 will-change 中指定了任意 CSS 属性，即便你没有直接指定这些属性的值（参考 这篇文章）
* -webkit-overflow-scrolling 属性被设置 "touch"的元素

在层叠上下文中，其子元素同样也按照上面解释的规则进行层叠。 特别值得一提的是，其子元素的 z-index 值只在父级层叠上下文中有意义。子级层叠上下文被自动视为父级层叠上下文的一个独立单元。

总结:

* 给一个 HTML 元素定位和 z-index 赋值创建一个层叠上下文，（opacity 值不为 1 的也是相同）。
* 层叠上下文可以包含在其他层叠上下文中，并且一起创建一个有层级的层叠上下文。
* 每个层叠上下文完全独立于它的兄弟元素：当处理层叠时只考虑子元素。
* 每个层叠上下文是自包含的：当元素的内容发生层叠后，整个该元素将会 在父层叠上下文中 按顺序进行层叠。

# meta

元数据（metadata）是关于数据的信息。

标签提供关于 HTML 文档的元数据。元数据不会显示在页面上，但是对于机器是可读的。

典型的情况是，meta 元素被用于规定页面的描述、关键词、文档的作者、最后修改时间以及其他元数据。

标签始终位于 head 元素中。

元数据可用于浏览器（如何显示内容或重新加载页面），搜索引擎（关键词），或其他 web 服务。

# 节流&防抖

1. 给页面注入50万个div怎么做提升性能

让创建节点的工作分批进行，比如把1 秒钟创建1000 个节点，改为每隔200 毫秒创建8 个节点。

var timeChunk = function( ary, fn, count ){

var obj,

t;

var len = ary.length;

var start = function(){

for ( var i = 0; i < Math.min( count || 1, ary.length ); i++ ){

var obj = ary.shift();

fn( obj );

}

};

return function(){

t = setInterval(function(){

if ( ary.length === 0 ){ // 如果全部节点都已经被创建好

return clearInterval( t );

}

start();

}, 200 ); // 分批执行的时间间隔，也可以用参数的形式传入

};

};

var renderFriendList = timeChunk( ary, function( n ){

var div = document.createElement( 'div' );

div.innerHTML = n;

document.body.appendChild( div );

}, 8 );

renderFriendList();

# js 线程、事件循环、任务队列

浏览器的js线程，定时器线程，事件处理线程，http请求线程之间任务分配。 浏览器的队列机制

settimeout时间设为0时，也没有立即执行的原因是，setTimeout有一个最小执行时间，当指定的时间小于该时间时，浏览器会用最小允许的时间作为setTimeout的时间间隔，也就是说即使我们把setTimeout的毫秒数设置为0，被调用的程序也没有马上启动。setTimeout的最小时间间隔和浏览器及操作系统有关。

## 事件循环：

JS 会创建一个类似于 while (true) 的循环，每执行一次循环体的过程称之为 Tick。每次 Tick 的过程就是查看是否有待处理事件，如果有则取出相关事件及回调函数放入执行栈中由主线程执行。待处理的事件会存储在一个任务队列中，也就是每次 Tick 会查看任务队列中是否有需要执行的任务。

## 任务队列：

异步操作会将相关回调添加到任务队列中。而不同的异步操作添加到任务队列的时机也不同，如 onclick, setTimeout, ajax 处理的方式都不同，这些异步操作是由浏览器内核的 webcore 来执行的，webcore 包含上图中的3种 webAPI，分别是 DOM Binding、network、timer模块。

* onclick 由浏览器内核的 DOM Binding 模块来处理，当事件触发的时候，回调函数会立即添加到任务队列中。
* setTimeout 会由浏览器内核的 timer 模块来进行延时处理，当时间到达的时候，才会将回调函数添加到任务队列中。
* ajax 则会由浏览器内核的 network 模块来处理，在网络请求完成返回之后，才将回调添加到任务队列中。

## 主线程：

JS 只有一个线程，称之为主线程。而事件循环是主线程中执行栈里的代码执行完毕之后，才开始执行的。所以，主线程中要执行的代码时间过长，会阻塞事件循环的执行，也就会阻塞异步操作的执行。只有当主线程中执行栈为空的时候（即同步代码执行完后），才会进行事件循环来观察要执行的事件回调，当事件循环检测到任务队列中有事件就取出相关回调放入执行栈中由主线程执行。

## settimeout—>0

所以setTimeout(code, ms)表示的是ms后把code任务添加到队列，而不是说ms后就会执行，就算setTimeout时间设置成0，他也要等主线程上的任务完成才会执行，所以setTimeout的执行时间并不是准确的，往往真正执行的时间要比你设定的时间偏大。

## settimeout队列与promise队列

(function test() {

setTimeout(function() {console.log(4)}, 0);

new Promise(function executor(resolve) {

console.log(1);

for( var i=0 ; i<10000 ; i++ ) {

i == 9999 && resolve();

}

console.log(2);

}).then(function() {

console.log(5);

});

console.log(3);

})()

在这段代码里面，setTimeout和Promise都被称之为任务源，来自不同任务源的回调函数会被放进不同的任务队列里面。

setTimeout的回调函数被放进setTimeout的任务队列之中。而对于Promise,它的回调函数并不是传进去的executer函数，而是其异步执行的then方法里面的参数，被放进Promise的任务队列之中。也就是说Promise的第一个参数并不会被放进Promise的任务队列之中，而会在当前队列就执行。

其中setTimeout和Promise的任务队列叫做macro-task(宏任务)，当然如我们所想，还有micro-task(微任务)。

macro-task包括：script(整体代码), setTimeout, setInterval, setImmediate, I/O, UI rendering。

micro-task包括：process.nextTick, Promises, Object.observe, MutationObserver

事件循环的顺序是从script开始第一次循环，随后全局上下文进入函数调用栈，碰到macro-task就将其交给处理它的模块处理完之后将回调函数放进macro-task的队列之中，碰到micro-task也是将其回调函数放进micro-task的队列之中。直到函数调用栈清空只剩全局执行上下文，然后开始执行所有的micro-task。当所有可执行的micro-task执行完毕之后。循环再次执行macro-task中的一个任务队列，执行完之后再执行所有的micro-task，就这样一直循环。

执行顺序：主队列->micro-task->页面渲染->macro-task

所以micro-task会阻塞页面渲染，而macro-task不会

# 小程序

## 生命周期

onLoad(Object query)：页面加载时触发。一个页面只会调用一次，可以在 onLoad 的参数中获取打开当前页面路径中的参数。

onShow()：页面显示/切入前台时触发。

onReady()：页面初次渲染完成时触发。一个页面只会调用一次，代表页面已经准备妥当，可以和视图层进行交互。

onHide()：页面隐藏/切入后台时触发。 如 navigateTo 或底部 tab 切换到其他页面，小程序切入后台等。

onUnload()：页面卸载时触发。如redirectTo或navigateBack到其他页面时。

# HTML5

## 新特性

1.语意特性,添加<header><header/><nav><nav>等标签

2.多媒体， 用于媒介回放的 video 和 audio 元素

3.图像效果，用于绘画的 canvas 元素，svg元素等

4.离线 & 存储,对本地离线存储的更好的支持,local Store,Cookies等

5.设备兼容特性 ，HTML5提供了前所未有的数据与应用接入开放接口。使外部应用可以直接与浏览器内部的数据直接相连，

6.连接特性，更有效的连接工作效率，使得基于页面的实时聊天，更快速的网页游戏体验，更优化的在线交流得到了实现。HTML5拥有更有效的服务器推送技术，Server-Sent Event和WebSockets就是其中的两个特性，这两个特性能够帮助我们实现服务器将数据“推送”到客户端的功能

7.性能与集成特性，HTML5会通过XMLHttpRequest2等技术，帮助您的Web应用和网站在多样化的环境中更快速的工作

## 兼容

需要在head中加上,当版本低于IE9时，浏览器会加载html5.js脚本，使得支持html5的新功能，也可以将脚本文件下载到本地

<head>

<!--[if lt IE 9]>

<script src='http://apps.bdimg.com/libs/html5shiv/3.7/html5shiv.min.js'>

</script>

<![endif]-->

</head>

## input新增类型

* color,选择颜色
* date 选择日期
* email 用于检测输入的是否为email格式的地址
* month 选择月份
* number 用于应该包含数值的输入域，可以设定对输入值的限定
* range 用于定义一个滑动条，表示范围
* search 用于搜索，比如站点搜索或 Google 搜索
* tel 输入电话号码
* -time 选择时间
* url 输入网址
* week 选择周和年