# 1.浏览器篇

## 1.1 常用那几种浏览器测试？主流浏览器的内核有哪些？

1. **Trident内核**：代表产品为**Internet Explorer**，又称其为IE内核。Trident（又称为MSHTML），是微软开发的一种排版引擎；
2. **Webkit内核**：代表作品有Safari、Chrome。WebKit是一个开源项目，主要用于Mac OS系统，它的特点在于源码结构清晰、渲染速度极快。缺点是对网页代码的兼容性不高，导致一些编写不标准的网页无法正常显示；
3. **Gecko内核**： **Netscape6**开始采用的内核，后来的**Mozilla FireFox** (火狐浏览器) 也采用了该内核，K-Meleon浏览器也是使用这种内核,Gecko是一套开放源代码的、以C++编写的网页排版引擎；
4. **Presto内核**：目前只有**Opera浏览器**采用该内核此外，Presto是由Opera Software开发的浏览器排版引擎，供Opera 7.0及以上使用。

## 1.2 说说你对浏览器内核的理解？

浏览器内核主要分成两个部分：**渲染引擎**和**JS引擎**。  
**渲染引擎**：负责取得网页内容（html，xml和图像等），整理讯息（e.g.css），以及计算网页的显示方式，然后输出到显示器或打印机。浏览器的内核的不同对于网页的语法解释会有不同，因此，渲染效果也不同。所有网页浏览器、电子邮件客户端以及需要编辑显示网络内容的应用程序都需要内核。  
**JS引擎**：解析和执行JavaScript来实现网页的动态效果，JavaScript的渲染速度越快，动态网页的展示也越快。

## 1.3 一个页面从输入 URL 到页面加载显示完成，这个过程中都发生了什么？

1. 浏览器根据请求的URL交给DNS域名解析，找到IP地址；
2. 浏览器根据IP地址向服务器发起TCP连接，三次握手，与服务器建立TCP连接；
   1. 客户端向服务器发送一个建立连接的请求
   2. 服务器接到请求后发送同意连接的信号
   3. 客户端接到同意连接的信号后，再次向服务器发送了确认信号，然后客户端与服务器的连接建立成功
3. 浏览器发送HTTP请求：浏览器根据URL内容生成HTTP请求，请求中包含请求文件的位置、请求文件的方式等等；
4. 服务器处理请求并返回HTTP报文（HTTP响应报文由三部分组成：状态码、响应报头和响应报文）
   1. 服务器接到请求后，根据HTTP请求中的内容来决定如何获取相应的HTML文件；
   2. 服务器将得到的HTML文件发送给浏览器；
   3. 在浏览器还没有完全接收HTML文件时便开始渲染、显示网页；
   4. 在执行HTML中代码时，根据需要，浏览器会继续请求图片、CSS和JavaScript等文件，过程同HTML。
5. 断开连接。

## 1.4 浏览器缓存

1. sessionStorage  
   优点：可以临时存储，关闭页面标签自动回收，不支持跨页面交互  
   缺点：只能作为临时存储，不能存储持久化
2. localStorage  
   优点：用于长久保存整个网站的数据，保存的数据没有过期时间，直到手动去删除。  
   缺点：存在大小限制，IE8以上的IE版本才支持这个属性；目前所有的浏览器中都会把localStorage的值类型限定为string类型，这个在对我们日常比较常见的JSON对象类型需要一些转换
3. cookie  
   优点：兼容性最好，几乎所有的浏览器都支持  
   缺点：大小有限制，而且每次发送请求，请求头里会带着cookie一起发过去，现在基本大多数登录的合法性验证都是用cookie验证的
4. userData  
   优点：出现的时间比sessionStorage要早  
   缺点：IE专门的存储方式，存储大小有限，单个文件的大小限制是128KB，一个域名下总共可以保存1024KB的文件，文件个数应该没有限制。在受限站点里这两个值分别是64KB和640KB

## 1.5. HTTP

### 1.5.1 HTTP状态码++

1. 以2xx开头的都表示请求成功响应

|  |  |
| --- | --- |
| 状态码 | 含义 |
| 200 | 请求成功 |
| 204 | 请求处理成功，但是没有资源可以返回 |
| 206 | 对资源一部分进行响应，由Content-Range指定范围的实体内容 |

1. 以3xx开头的都表示需要进行附加操作以完成请求

|  |  |
| --- | --- |
| 状态码 | 含义 |
| 301 | 永久性重定向，该状态码表示请求的资源已经重新分配URI，以后应该使用资源现有的URI |
| 302 | 临时性重定向。该状态码表示请求的资源已被分配了新的URI，希望用户（本次）能使用新的URI访问。 |
| 303 | 该状态码表示由于请求对应的资源存在着另一个URI，应使用GET方法定向获取请求的资源 |
| 304 | 该状态码表示客户端发送附带条件的请求时，服务器允许请求访问资源，但未满足条件的情况 |
| 307 | 临时重定向。该状态码与302 Found有着相同的含义 |

1. 以4xx的响应结果表明客户端是发生错误的原因所在。

|  |  |
| --- | --- |
| 状态码 | 含义 |
| 400 | 该状态码表示请求报文中存在语法错误。当错误发生时，需修改请求的内容后再次发送请求。 |
| 401 | 该状态码表示发送的请求需要通过有HTTP认证（BASIC认证、DIGEST认证）的认证信息。 |
| 403 | 该状态码表明对请求资源的访问被服务器拒绝了 |
| 404 | 该状态码表明服务器上无法找到请求的资源 |

1. 以5xx为开头的响应标头都表示服务器本身发生错误。

|  |  |
| --- | --- |
| 状态码 | 含义 |
| 500 | 该状态码表明服务器端在执行请求时发生了错误。 |
| 503 | 该状态码表明服务器暂时处于超负载状态或正在停机维护，现在无法处理请求 |

#### 302、303和307的区别

**302：**RFC2616(HTTP 1.1)中说如果客户端发出非GET、HEAD请求后，收到服务端的302状态码，那么就不能自动的向新URI发送重复请求，除非得到用户的确认，**因为第二次POST时，环境可能已经发生变化（嗯，POST方法不是幂等的），POST操作会不符合用户预期**。但是，很多浏览器都把302当作303处理了（注意，303是HTTP1.1才加进来的，其实从HTTP1.0进化到HTTP1.1，浏览器什么都没动），它们获取到HTTP响应报文头部的Location字段信息，并发起一个GET请求。

**303：**303状态码的响应，也就是上边提到的现在浏览器对302状态码的处理：POST重定向为GET。

**307：**307状态码则相当于HTTP1.0文档中的302状态码，当客户端的POST请求收到服务端307状态码响应时，需要跟用户询问是否应该在新URI上发起POST方法，也就是说，307是不会把POST转为GET的。

**浏览器对303状态码的处理跟原来浏览器对HTTP1.0的302状态码的处理方法一样；浏览器对307状态码处理则跟原来HTTP1.0文档里对302的描述一样。**

### 1.5.2 URL和URI有什么区别？

**URI = Universal Resource Identifier** 统一资源标志符，用来标识抽象或物理资源的一个紧凑字符串。  
**URL = Universal Resource Locator** 统一资源定位符，一种定位资源的主要访问机制的字符串，一个标准的URL必须包括：protocol、host、port、path、parameter、anchor。  
**URN = Universal Resource Name** 统一资源名称，通过特定命名空间中的唯一名称或ID来标识资源。

URI 是用来唯一标识资源， URL 提供资源的识别方法并用着各种方法定位资源。

由于互联网上每个文件都有唯一的URL, 所以URL是一种具体的URI， 可以说URL是URI的一种实现方式。

### 1.5.3 HTTP和HTTPS+

#### HTTP

超文本传输协议（HTTP，HyperText Transfer Protocol)是互联网上应用最为广泛的一种网络协议。设计 HTTP 最初的目的是为了提供一种发布和接收 HTML 页面的方法。它可以使浏览器更加高效。HTTP 协议是以明文方式发送信息的，如果黑客截取了 Web 浏览器和服务器之间的传输报文，就可以直接获得其中的信息。

**HTTP原理**

1. 客户端的浏览器首先要通过网络与服务器建立连接，该连接是通过 TCP 来完成的，一般 TCP 连接的端口号是80。 建立连接后，客户机发送一个请求给服务器，请求方式的格式为：统一资源标识符（URL）、协议版本号，后边是 MIME 信息包括请求修饰符、客户机信息和许可内容。
2. 服务器接到请求后，给予相应的响应信息，其格式为一个状态行，包括信息的协议版本号、一个成功或错误的代码，后边是 MIME 信息包括服务器信息、实体信息和可能的内容。

#### HTTPS

是以安全为目标的 HTTP 通道，是 **HTTP 的安全版**。HTTPS 的安全基础是 **SSL**（Secure Socket Layer，安全套接字层）。SSL 协议位于 TCP/IP 协议与各种应用层协议之间，为数据通讯提供安全支持。SSL 协议可分为两层：SSL 记录协议（SSL Record Protocol），它建立在可靠的传输协议（如TCP）之上，为高层协议提供数据封装、压缩、加密等基本功能的支持。SSL 握手协议（SSL Handshake Protocol），它建立在 SSL 记录协议之上，用于在实际的数据传输开始前，通讯双方进行身份认证、协商加密算法、交换加密密钥等。

**HTTPS设计目标：**

1. 数据保密性：保证数据内容在传输的过程中不会被第三方查看。就像快递员传递包裹一样，都进行了封装，别人无法获知里面装了什么 。
2. 数据完整性：及时发现被第三方篡改的传输内容。就像快递员虽然不知道包裹里装了什么东西，但他有可能中途掉包，数据完整性就是指如果被掉包，我们能轻松发现并拒收 。
3. 身份校验安全性：保证数据到达用户期望的目的地。就像我们邮寄包裹时，虽然是一个封装好的未掉包的包裹，但必须确定这个包裹不会送错地方，通过身份校验来确保送对了地方 。

#### HTTP 与 HTTPS的区别

1. HTTPS 协议需要到 CA （Certificate Authority，证书颁发机构）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。(无状态的意思是其数据包的发送、传输和接收都是相互独立的。无连接的意思是指通信双方都不长久的维持对方的任何信息。)

#### HTTPS相对于HTTP的改进

**双向的身份认证**

客户端和服务端在传输数据之前,会通过基于X.509证书对双方进行身份认证 。具体过程如下：

1. 客户端发起 SSL 握手消息给服务端要求连接。
2. 服务端将证书发送给客户端。
3. 客户端检查服务端证书，确认是否由自己信任的证书签发机构签发(客户端内置了所有受信任 CA 的证书)。 如果不是，将是否继续通讯的决定权交给用户选择 ( 注意，这里将是一个安全缺陷 )。如果检查无误或者用户选择继续，则客户端认可服务端的身份。
4. 服务端要求客户端发送证书，并检查是否通过验证。失败则关闭连接，认证成功则从客户端证书中获得客户端的公钥，一般为 1024 位或者 2048 位。到此，服务器客户端双方的身份认证结束，双方确保身份都是真实可靠的。

**数据传输的机密性**

客户端和服务端在开始传输数据之前，会协商传输过程需要使用的加密算法。

**防止重放攻击**

SSL 使用序列号来保护通讯方免受报文重放攻击。

#### HTTPS的优缺点

**优点**

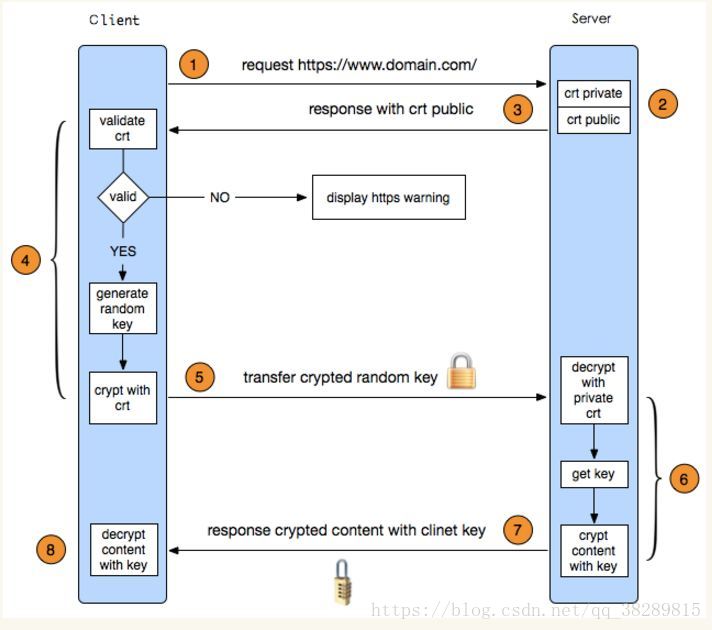
1. 使用 HTTPS 协议可认证用户和服务器，**确保数据发送**到正确的客户机和服务器。
2. HTTPS 协议是由SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，要比 HTTP 协议**安全**，可防止数据在传输过程中不被窃取、修改，确保数据的完整性。
3. HTTPS 是现行架构下最安全的解决方案，虽然不是绝对安全，但它**大幅增加了中间人攻击的成本**。

**缺点**

1. HTTPS 协议**握手阶段比较费时**，会使页面的加载时间延长近。
2. HTTPS **连接缓存不如 HTTP 高效**，会**增加数据开销**，甚至已有的安全措施也会因此而受到影响。
3. HTTPS 协议的**安全是有范围的**，在黑客攻击、拒绝服务攻击和服务器劫持等方面几乎起不到什么作用。
4. **SSL 证书通常需要绑定 IP，不能在同一 IP 上绑定多个域名**，IPv4 资源不可能支撑这个消耗。
5. **成本增加**。部署 HTTPS 后，因为 HTTPS 协议的工作要增加额外的计算资源消耗，例如 SSL 协议加密算法和 SSL 交互次数将占用一定的计算资源和服务器成本。
6. HTTPS 协议的**加密范围也比较有限**。最关键的，SSL 证书的信用链体系并不安全，特别是在某些国家可以控制 CA 根证书的情况下，中间人攻击一样可行。

#### HTTPS握手+

1. 客户端的浏览器向服务器发送请求，并传送客户端 SSL 协议的版本号，加密算法的种类，产生的随机数，以及其他服务器和客户端之间通讯所需要的各种信息。
2. 服务器向客户端传送 SSL 协议的版本号，加密算法的种类，随机数以及其他相关信息，同时服务器还将向客户端**传送自己的证书**。
3. 客户端利用服务器传过来的信息**验证服务器的合法性**，服务器的合法性包括：证书是否过期，发行服务器证书的 CA 是否可靠，发行者证书的公钥能否正确解开服务器证书的 "发行者的数字签名"，服务器证书上的域名是否和服务器的实际域名相匹配。如果合法性验证没有通过，通讯将断开；如果合法性验证通过，将继续进行第四步。
4. 用户端**随机产生一个用于通讯的 "对称密码"**，然后用服务器的公钥（服务器的公钥从步骤2中的服务器的证书中获得）对其加密，然后将加密后的“预主密码”传给服务器。
5. 如果**服务器要求客户的身份认证**（在握手过程中为可选），**用户可以建立一个随机数然后对其进行数据签名**，将这个含有签名的随机数和客户自己的证书以及加密过的密钥一起传给服务器。
6. 如果**服务器要求客户的身份认证**，**服务器必须检验客户证书和签名随机数的合法性**，具体的合法性验证过程包括：客户的证书使用日期是否有效，为客户提供证书的 CA 是否可靠，发行 CA 的公钥能否正确解开客户证书的发行 CA 的数字签名，检查客户的证书是否在证书废止列表（CRL）中。检验如果没有通过，通讯立刻中断；如果验证通过，服务器将用自己的私钥解开加密的私钥，然后执行一系列步骤来产生主通讯密码（客户端也将通过同样的方法产生相同的主通讯密码）。
7. **服务器和客户端用相同的对称加密密钥**，对称密钥用于 SSL 协议的安全数据通讯的加解密通讯。同时在 SSL 通讯过程中还要完成数据通讯的完整性，防止数据通讯中的任何变化。
8. **客户端向服务器端发出信息，指明后面的数据通讯将使用的步骤7中的主密码为对称密钥**，同时**通知服务器客户端的握手过程结束**。
9. **服务器向客户端发出信息，指明后面的数据通讯将使用的步骤7中的主密码为对称密钥**，同时**通知客户端服务器端的握手过程结束**。
10. **SSL 的握手部分结束**，SSL 安全通道的数据通讯开始，客户和服务器开始使用相同的对称密钥进行数据通讯，同时进行通讯完整性的检验。



### 1.5.4 HTTP 2.0 新特性+

#### 新的二进制帧层(二进制传输)

HTTP1.x的解析是基于文本。基于文本协议的格式解析存在天然缺陷，文本的表现形式有多样性，要做到健壮性考虑的场景必然很多，二进制则不同，只认0和1的组合。基于这种考虑HTTP2.0的协议解析决定采用二进制格式，实现方便且健壮。

#### 多路复用

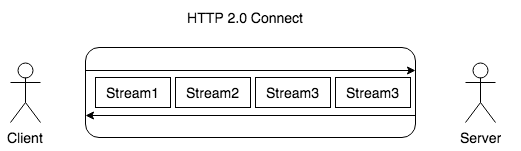
在 HTTP 2.0 中，有两个非常重要的概念，分别是帧（frame）和流（stream）。

**流**：已经建立的连接之间双向流动的字节，它能携带一个至多个消息。  
**消息**：一个完整的帧序列，它映射到逻辑的请求和响应消息。  
**帧**：在HTTP/2通信的最小单元。每个桢包括一个帧头，里面有个很小标志，来区别是属于哪个流。

**帧代表着最小的数据单位**，每个帧会标识出该帧属于哪个流，流也就是多个帧组成的数据流。

所有的通信都建立在一个TCP连接上，可以传递大量的双向流通的流。  
每个流都有**独一无二的标志和优先级**。  
每个消息都是逻辑上的请求和相应消息。由一个或者多个帧组成。  
来自不同流的帧可以通过帧头的标志来关联和组装起来。

**多路复用，就是在一个 TCP 连接中可以存在多条流。**换句话说，**也就是可以发送多个请求，对端可以通过帧中的标识知道属于哪个请求**。通过这个技术，可以避免 HTTP 旧版本中的队头阻塞问题，极大的提高传输性能。



#### 流的优先级

为了能方便流的传输顺序,HTTP/2.0提出,使每个流都有一个权重和依赖.

* 每个流的权重值在1~256之间
* 每个流可以详细给出对其他流的依赖

流权重和依赖的结合使客户端可以构造和通信一个优先级二叉树来表达它更想得到哪种响应.然后服务端可以按权重分配硬件资源(CPU,内存)

在HTTP/2 ,一个流的依赖可以显式用其他流的标志来表达,如果省略了标志,则说明它的依赖是根流.一般来说,父流应该在它的依赖流之前分配资源,例如D应该是C之前被发送.依赖于同一父节点的应该按照他们的权重分配资源.例如A结点的权重为12,它的兄弟结点B的结点的权重为4.然后按比例分资源,A占12/16,B占4/16.如上面所述,流的依赖和权重提供了一种很好的表达式语言来表达资源的优先级.但是我们应该明白,,流的依赖和权重只是提供了一种传输偏好,而不是说一定是这样的比例.

#### header压缩

在 HTTP 1.X 中，我们使用文本的形式传输 header，在 header 携带 cookie 的情况下，可能每次都需要重复传输几百到几千的字节。

在 HTTP 2.0 中，使用了 HPACK 压缩格式对传输的 header 进行编码，减少了 header 的大小。并在两端维护了索引表，用于记录出现过的 header ，后面在传输过程中就可以传输已经记录过的 header 的键名，对端收到数据后就可以通过键名找到对应的值。

#### 服务器端push

在 HTTP 2.0 中，服务端可以在客户端某个请求后，主动推送其他资源。

可以想象以下情况，某些资源客户端是一定会请求的，这时就可以采取服务端 push 的技术，提前给客户端推送必要的资源，这样就可以相对减少一点延迟时间。当然在浏览器兼容的情况下你也可以使用 prefetch 。

#### QUIC

这是一个谷歌出品的基于 UDP 实现的同为传输层的协议，目标很远大，希望替代 TCP 协议。

1、该协议支持多路复用，虽然 HTTP 2.0 也支持多路复用，但是下层仍是 TCP，因为 TCP 的重传机制，只要一个包丢失就得判断丢失包并且重传，导致发生队头阻塞的问题，但是 UDP 没有这个机制

2、实现了自己的加密协议，通过类似 TCP 的 TFO 机制可以实现 0-RTT，当然 TLS 1.3 已经实现了 0-RTT 了

3、支持重传和纠错机制（向前恢复），在只丢失一个包的情况下不需要重传，使用纠错机制恢复丢失的包。**纠错机制**：通过异或的方式，算出发出去的数据的异或值并单独发出一个包，服务端在发现有一个包丢失的情况下，通过其他数据包和异或值包算出丢失包。在丢失两个包或以上的情况就使用重传机制，因为算不出来了。

### 1.5.5 HTTP请求方法

1、OPTIONS

返回服务器针对特定资源所支持的HTTP请求方法，也可以利用向web服务器发送‘\*’的请求来测试服务器的功能性

2、HEAD

向服务器索与GET请求相一致的响应，只不过响应体将不会被返回。这一方法可以再不必传输整个响应内容的情况下，就可以获取包含在响应小消息头中的元信息。

3、GET

向特定的资源发出请求。注意：GET方法不应当被用于产生“副作用”的操作中，例如在Web Application中，其中一个原因是GET可能会被网络蜘蛛等随意访问。Loadrunner中对应get请求函数：web*link和web*url

4、POST

向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的建立和/或已有资源的修改。 Loadrunner中对应POST请求函数：web*submit*data,web*submit*form

5、PUT

向指定资源位置上传其最新内容

6、DELETE

请求服务器删除Request-URL所标识的资源

7、TRACE

回显服务器收到的请求，主要用于测试或诊断

8、CONNECT

HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。

#### PUT和POST的区别+

PUT请求：如果两个请求相同，后一个请求会把第一个请求覆盖掉。（所以PUT用来改资源）

POST请求：后一个请求不会把第一个请求覆盖掉。（所以POST用来增资源）

PUT请求的目的是用来替换整个目标资源。PUT请求具有 **幂等性（idempotent）**。（幂等性的意思在这个RFC的4.2.2里说了“同样的请求，不管发多少次，每次服务器处理完之后的结果，都和只发一次是一样的。”

（缓存性，POST是可以被缓存的，PUT却不行。？）

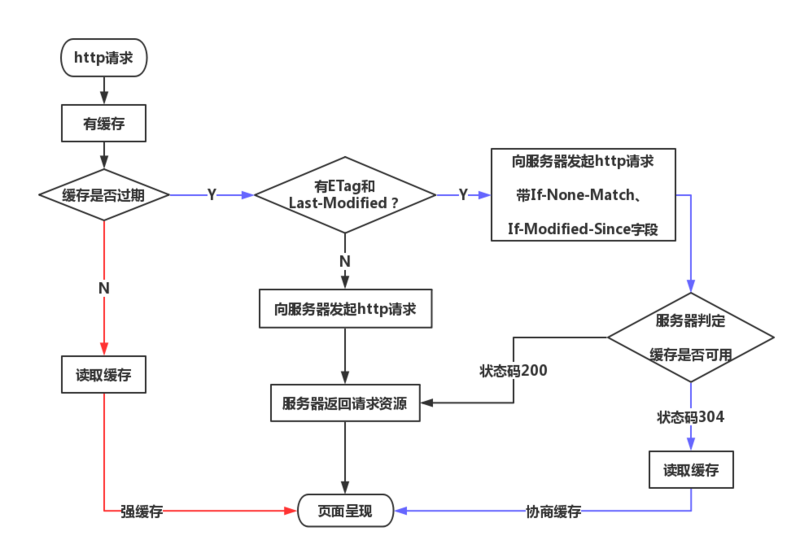
### 1.5.6 HTTP缓存+

Web 缓存大致可以分为：数据库缓存、服务器端缓存（代理服务器缓存、CDN 缓存）、浏览器缓存。

浏览器缓存也包含很多内容： HTTP 缓存、indexDB、cookie、localStorage 等等。

浏览器第一次向一个web服务器发起http请求后，服务器会返回请求的资源，并且在响应头中添加一些有关缓存的字段如：Cache-Control、Expires、Last-Modified、ETag、Date等等。之后浏览器再向该服务器请求该资源就可以视情况使用**强缓存**和**协商缓存**。

* **强缓存**：浏览器直接从本地缓存中获取数据，不与服务器进行交互。
* **协商缓存**：浏览器发送请求到服务器，服务器判定是否可使用本地缓存。
* 联系与区别：两种缓存方式最终使用的都是本地缓存；前者无需与服务器交互，后者需要。



#### 强缓存

如图红线所示的过程代表强缓存。用户发起了一个http请求后，浏览器发现先本地已有所请求资源的缓存，便开始检查缓存是否过期。有两个http头部字段控制缓存的有效期：Expires和Cache-Control，浏览器是根据以下两步来判定缓存是否过期的：

1. 查看缓存是否有Cache-Control的s-maxage或max-age指令，若有，则使用响应报文生成时间Date + s-maxage/max-age获得过期时间，再与当前时间进行对比（s-maxage适用于多用户使用的公共缓存服务器）；
2. 如果没有Cache-Control的s-maxage或max-age指令，则比较Expires中的过期时间与当前时间。Expires是一个绝对时间。

**注意**，在HTTP/1.1中，当首部字段Cache-Control有指定s-maxage或max-age指令，比起首部字段Expires，会优先处理s-maxage或max-age。

另外下面列几个Cache-Control的常用指令：

* no-cache：含义是不使用本地缓存，需要使用协商缓存，也就是先与服务器确认缓存是否可用。
* no-store：禁用缓存。
* public：表明其他用户也可使用缓存，适用于公共缓存服务器的情况。
* private：表明只有特定用户才能使用缓存，适用于公共缓存服务器的情况。

经过上述两步判断后，若缓存未过期，返回状态码为200，则直接从本地读取缓存，这就完成了整个强缓存过程；如果缓存过期，则进入协商缓存或服务器返回新资源过程。

#### 协商缓存

当浏览器发现缓存过期后，缓存并不一定不能使用了，因为服务器端的资源可能仍然没有改变，所以需要与服务器协商，让服务器判断本地缓存是否还能使用。此时浏览器会判断缓存中是否有ETag或Last-Modified字段，如果没有，则发起一个http请求，服务器根据请求返回资源；如果有这两个字段，则在请求头中添加If-None-Match字段（有ETag字段的话添加）、If-Modified-Since字段（有Last-Modified字段的话添加）。**注意：**如果同时发送If-None-Match 、If-Modified-Since字段，服务器只要比较If-None-Match和ETag的内容是否一致即可；如果内容一致，服务器认为缓存仍然可用，则返回状态码304，浏览器直接读取本地缓存，这就完成了协商缓存的过程，也就是图中的蓝线；如果内容不一致，则视情况返回其他状态码，并返回所请求资源。

**详细过程**：

1. **ETag和If-None-Match**：二者的值都是服务器为每份资源分配的唯一标识字符串。
   * 浏览器请求资源，服务器会在响应报文头中加入ETag字段。资源更新时，服务器端的ETag值也随之更新；
   * 浏览器再次请求资源时，会在请求报文头中添加If-None-Match字段，它的值就是上次响应报文中的ETag的值；
   * 服务器会比对ETag与If-None-Match的值是否一致，如果不一致，服务器则接受请求，返回更新后的资源；如果一致，表明资源未更新，则返回状态码为304的响应，可继续使用本地缓存，要注意的是，此时响应头会加上ETag字段，即使它没有变化。
2. **Last-Modified和If-Modified-Since**：二者的值都是GMT格式的时间字符串。
   * 浏览器第一次向服务器请求资源后，服务器会在响应头中加上Last-Modified字段，表明该资源最后一次的修改时间；
   * 浏览器再次请求该资源时，会在请求报文头中添加If-Modified-Since字段，它的值就是上次服务器响应报文中的Last-Modified的值；
   * 服务器会比对Last-Modified与If-Modified-Since的值是否一致，如果不一致，服务器则接受请求，返回更新后的资源；如果一致，表明资源未更新，则返回状态码为304的响应，可继续使用本地缓存，与ETag不同的是：此时响应头中不会再添加Last-Modified字段。
3. **ETag较之Last-Modified的优势**：你可能会觉得使用Last-Modified已经足以让浏览器知道本地的缓存副本是否足够新，为什么还需要ETag呢？HTTP1.1中ETag的出现主要是为了解决几个Last-Modified比较难解决的问题：
   * 一些文件也许会周期性的更改，但是他的内容并不改变(仅仅改变的修改时间)，这个时候我们并不希望客户端认为这个文件被修改了，而重新GET；
   * 某些文件修改非常频繁，比如在秒以下的时间内进行修改，(比方说1s内修改了N次)，If-Modified-Since能检查到的粒度是s级的，这种修改无法判断(或者说UNIX记录MTIME只能精确到秒)；
   * 某些服务器不能精确的得到文件的最后修改时间。

这时，利用ETag能够更加准确的控制缓存，因为ETag是服务器自动生成的资源在服务器端的唯一标识符，资源每次变动，都会生成新的ETag值。Last-Modified与ETag是可以一起使用的，但服务器会优先验证ETag。

## 1.6 TCP和UDP

### 1.6.1 TCP/IP网络模型

OSI七层模型
TCP/IP概念层模型
功能
TCP/IP协议族
应用层
应用层
文件传输，电子邮件，文件服务，虚拟终端
TFTP，HTTP，SNMP，FTP，SMTP，DNS，Telnet
表示层
数据格式化，代码转换，数据加密
没有协议
会话层
解除或建立与别的结点的联系
没有协议
传输层
传输层
提供端对端的接口
TCP，UDP
网络层
网络层
为数据包选择路由
IP，ICMP，RIP，OSPF，BGP，IGMP
数据链路层
链路层
传输有地址的帧以及错误检测功能
SLIP，CSLIP，PPP，ARP，RARP，MTU
物理层
以二进制数据形式在物理媒体上传播数据
ISO2110，IEEE802，IEEE802.2

### 1.6.2 TCP和UDP的区别

TCP全称传输控制协议(Transport Control Protocol)定义了两台计算机之间进行可靠的传输而交换的数据和确认信息的格式，以及计算机为了确保数据的正确到达而采取的措施。

UDP全称用户数据报协议(User Datagram Protocol)是一个简单的面向数据报的传输层协议。提供的是非面向连接的、不可靠的数据流传输。UDP不提供可靠性，也不提供报文到达确认、排序以及流量控制等功能。不能保证数据能到达目的地。但由于UDP在传输数据报前不用在客户和服务器之间建立一个连接，且没有超时重发等机制，故而传输速度很快。

|  |  |  |
| --- | --- | --- |
|  | TCP | UDP |
| 是否连接 | 面向连接 | 无连接 |
| 传输可靠性 | 可靠的 | 不可靠的 |
| 应用场景 | 传输少量数据 | 大量数据 |
| 速度 | 慢 | 快 |

# 2.HTML篇

## 2.1 Doctype

<!DOCTYPE>声明位于位于HTML文档中的第一行，处于<html>标签之前。

### 2.1.1 作用

告知浏览器的解析器用什么文档标准解析这个文档。DOCTYPE**不存在或格式不正确**会导致文档以**兼容模式**呈现。

### 2.1.2 标准模式和兼容模式各有什么区别

标准模式的排版 和JS运作模式都是以该浏览器支持的最高标准运行。  
在兼容模式中，页面以宽松的向后兼容的方式显示,模拟老式浏览器的行为以防止站点无法工作。  
简单的说，就是尽可能的显示能显示的东西给用户看。

**具体的说：**

1. **width不同**  
   在严格模式中 ：width是内容宽度 ，元素真正的宽度 = margin-left + border-left-width + padding-left + width + padding-right + border-right- width + margin-right;  
   在兼容模式中 ：width则是元素的实际宽度 ，内容宽度 = width - ( padding-left + padding-right + border-left-width + border-right-width)
2. 兼容模式下可设置**百分比的高度和行内元素的高宽**  
   在Standards模式下，给span等行内元素设置wdith和height都不会生效，而在兼容模式下，则会生效。  
   在standards模式下，一个元素的高度是由其包含的内容来决定的，如果父元素没有设置高度，子元素设置一个百分比的高度是无效的。
3. 用margin:0 auto设置水平居中在IE下会失效  
   使用margin:0 auto在standards模式下可以使元素水平居中，但在兼容模式下却会失效（用text-align属性解决）  
   body{text-align:center};#content{text-align:left}
4. 兼容模式下Table中的字体属性不能继承上层的设置，white-space:pre会失效，设置图片的padding会失效

### 2.1.3 为什么H5只需要写<!DOCTYPE>

* HTML5 不基于 SGML，因此不需要对DTD进行引用，但是需要doctype来规范浏览器的行为（让浏览器按照它们应该的方式来运行）
* 而HTML4.01基于SGML,所以需要对DTD进行引用，才能告知浏览器文档所使用的文档类型

### 2.1.4 常见的DOCTYPE 声明

**HTML 5**

<!DOCTYPE html>

**HTML 4.01 Strict**  
该 DTD 包含所有 HTML 元素和属性，但不包括展示性的和弃用的元素（比如 font）。不允许框架集（Framesets）。

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">

**HTML 4.01 Transitional**  
该 DTD 包含所有 HTML 元素和属性，包括展示性的和弃用的元素（比如 font）。不允许框架集（Framesets）。

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN""http://www.w3.org/TR/html4/loose.dtd">

**HTML 4.01 Frameset**  
该 DTD 等同于 HTML 4.01 Transitional，但允许框架集内容。

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN""http://www.w3.org/TR/html4/frameset.dtd">

**XHTML 1.0 Strict**

该 DTD 包含所有 HTML 元素和属性，但不包括展示性的和弃用的元素（比如 font）。不允许框架集（Framesets）。必须以格式正确的 XML 来编写标记。

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

**XHTML 1.0 Transitional**

该 DTD 包含所有 HTML 元素和属性，包括展示性的和弃用的元素（比如 font）。不允许框架集（Framesets）。必须以格式正确的 XML 来编写标记。

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

**XHTML 1.0 Frameset**

该 DTD 等同于 XHTML 1.0 Transitional，但允许框架集内容。

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">

**XHTML 1.1**

该 DTD 等同于 XHTML 1.0 Strict，但允许添加模型（例如提供对东亚语系的 ruby 支持）。

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

## 2.2 说说你对语义化的理解？

### 2.2.1 什么是HTML语义化？

根据内容的结构化（内容语义化），选择合适的标签（代码语义化）便于开发者阅读和写出更优雅的代码的同时让浏览器的爬虫和机器很好地解析。

### 2.2.2 为什么要语义化

1. 为了在没有CSS的情况下，页面也能呈现出很好地内容结构及代码结构；
2. 利于SEO搜索引擎优化，和搜索引擎建立良好沟通，有助于爬虫抓取更多的有效信息，爬虫依赖于标签来确定上下文和各个关键字的权重；
3. 方便浏览器和其他设备解析（如屏幕阅读器、盲人阅读器、移动设备）以意义的方式来渲染网页；
4. 提高代码的可维护和可重用性。

### 2.2.3 语义化注意事项

1. 尽可能**少的使用**无语义的标签**div和span**；
2. 在语义不明显时，**既可以使用div或者p时，尽量用p**, 因为p在默认情况下有**上下间距**，对兼容特殊终端有利；
3. **不要使用纯样式标签**，如：b、font、u等，改用css设置。
4. 需要**强调的文本，可以包含在strong或者em标签中**（浏览器预设样式，能用CSS指定就不用他们），strong默认样式是加粗（不要用b），em是斜体（不用i）；
5. 使用**表格时，标题要用caption，表头用thead，主体部分用tbody包围**，尾部用tfoot包围。表头和一般单元格要区分开，表头用th，单元格用td；
6. **表单域要用fieldset**标签包起来，并用**legend标签说明表单的用途**；
7. **每个input标签对应的说明文本都需要使用label标签**，并且通过为input设置id属性，在lable标签中设置for=someld来让说明文本和相对应的input关联起来。

## 2.3 HTML与XHTML有什么区别?

**最主要的不同：**

* XHTML 元素必须被正确地嵌套。
* XHTML 元素必须被关闭。
* 标签名必须用小写字母。
* XHTML 文档必须拥有根元素。
* XHTML对于图片需添加alt属性

## 2.4 页面导入样式时，使用link和@import有什么区别？

### 2.4.1 import的写法

\<style type=”text/css”>  
 @import url（“a.css”）；  
\</style>

### 2.4.2 link的写法

\<link rel="stylesheet" href="index.css">

### 2.4.3 两者区别

1. **来源**：link属于XHTML标签，import是CSS提供的方式。**link方式除了CSS，还可以定义RSS，定义rel连接属性等，而import只能加载CSS**。
2. **顺序**：**link是页面加载时同时执行的，而import是在页面加载完之后，才会执行的**（所以容易导致样式闪烁，即开始页面没有样式，突然会闪烁一下，然后就有了样式，就我个人比较喜欢link引用的方式）
3. **js控制的差别**：当页面需要使用javascript控制dom改变样式的时候，只能使用link标签，因为import不是js操作dom可以控制的。也不支持js操作dom来控制。

## 2.5 HTML5有哪些新特性？+

1. 语义化标签
2. 增强型表单
3. 新增视频和音频标签
4. Canvas&SVG绘图
5. 地理定位API
6. 拖放API
7. Web Worker
8. Web Storage
9. WebSocket

### 2.5.1 h5新语义/结构元素

|  |  |
| --- | --- |
| 标签 | 描述 |
| <article> | 定义文档内的文章。 |
| <aside> | 定义页面内容之外的内容。 |
| <bdi> | 定义与其他文本不同的文本方向。 |
| <details> | 定义用户可查看或隐藏的额外细节。 |
| <dialog> | 定义对话框或窗口。 |
| <figcaption> | 定义 <figure> 元素的标题。 |
| <figure> | 定义自包含内容，比如图示、图表、照片、代码清单等等。 |
| <footer> | 定义文档或节的页脚。 |
| <header> | 定义文档或节的页眉。 |
| <hgroup> | 用于对网页或区段（section）的标题进行组合。 |
| <main> | 定义文档的主内容。 |
| <mark> | 定义重要或强调的内容。 |
| <menuitem> | 定义用户能够从弹出菜单调用的命令/菜单项目。 |
| <meter> | 定义已知范围（尺度）内的标量测量。 |
| <nav> | 定义文档内的导航链接。 |
| <progress> | 定义任务进度。 |
| <rp> | 定义在不支持 ruby 注释的浏览器中显示什么。 |
| <rt> | 定义关于字符的解释/发音（用于东亚字体）。 |
| <ruby> | 定义 ruby 注释（用于东亚字体）。 |
| <section> | 定义文档中的节。 |
| <summary> | 定义 <details> 元素的可见标题。 |
| <time> | 定义日期/时间。 |
| <wbr> | 定义可能的折行（line-break）。 |

总体来说，这些标签就是div标签的语义化转化，过去我们曾用div来包裹某个块或者说组件，现在h5使用这些标签实现语义化，有利于代码可读性和SEO。语义元素在IE6-8的兼容可以使用。

#### H5删除的元素

|  |  |
| --- | --- |
| 删除的元素 | 替代方法 |
| 首字母缩写 |  |
|  |  |
| 页面上默认字体颜色和字号 | css |
| 更大的文本 | css |
| 文本水平居中 | css |
| 目录列表 | css |
| 字体外观，尺寸，颜色 | css |
| 定义子窗口 |  |
| 定义框架集 |  |
| 向浏览器显示无法处理框架的提示文本，位于frameset元素中 |  |
| 文本添加删除线 | css或或 |
| 定义打字机文本 | css |

### 2.5.2 增强型input

#### 新增表单元素

|  |  |
| --- | --- |
| 标签 | 描述 |
| <datalist> | 定义输入控件的预定义选项。 |
| <keygen> | 定义键对生成器字段（用于表单）。 |
| <output> | 定义计算结果。 |

#### 新增输入类型和输入属性

|  |  |  |  |
| --- | --- | --- | --- |
| 新的输入类型 | 说明 | 新的输入属性 | 说明 |
| color | 用于应该包含颜色的输入字段。（颜色选择器） | autocomplete | 属性规定表单或输入字段是否应该自动完成。（on|off） |
| date | 用于应该包含日期的输入字段。 | autofocus | 规定当页面加载时 <input> 元素应该自动获得焦点。 |
| datetime | 允许用户选择日期和时间（有时区）。(Chrome、FF、IE不支持) | form | 规定 <input> 元素所属的一个或多个表单。 （空格分隔） |
| datetime-local | 允许用户选择日期和时间（无时区）。（FF、IE不支持） | formaction | 规定当提交表单时处理该输入控件的文件的 URL。 |
| email | 用于应该包含电子邮件地址的输入字段。 | formenctype | 规定当把表单数据（form-data）提交至服务器时如何对其进行编码（仅针对 method="post" 的表单）。 |
| month | 允许用户选择**月份和年份**。 | formmethod | 定义用以向 action URL 发送表单数据（form-data）的 HTTP 方法。 |
| number | 用于应该包含数字值的输入字段。 | formnovalidate | 规定在提交表单时不对 <input> 元素进行验证。【额外提交】 |
| range | 用于应该包含一定范围内的值的输入字段。（部分浏览器显示为滑块） | formtarget | 规定的名称或关键词指示提交表单后在何处显示接收到的响应。 |
| search | 用于搜索字段（搜索字段的表现类似常规文本字段）。 | height&width | 规定 <input> 元素的高度和宽度。 |
| tel | 用于应该包含电话号码的输入字段。（只有 Safari 8 支持 tel 类型。） | list | list 属性引用的 <datalist> 元素中包含了 <input> 元素的预定义选项。 |
| time | 允许用户选择时间（无时区）。 | min&max | 规定 <input> 元素的最小值和最大值。 |
| url | 用于应该包含 URL 地址的输入字段。自动验证。（IE9及之前不支持） | multiple | 规定允许用户在 <input> 元素中输入一个以上的值。 |
| week | 允许用户选择周和年。 | novalidate | 规定在提交表单时不对表单数据进行验证。 |
|  |  | pattern (regexp) | 规定用于检查 <input> 元素值的正则表达式。 |
|  |  | placeholder | 规定用以描述输入字段预期值的提示（样本值或有关格式的简短描述）。 |
|  |  | required | 规定在提交表单之前必须填写输入字段。 |
|  |  | step | 规定 <input> 元素的合法数字间隔。 |

### 2.5.3 新增视频<video>和音频<audio>标签

|  |  |
| --- | --- |
| **标签** | **描述** |
| **<audio>** | 定义声音或音乐内容。 |
|  | 定义外部应用程序的容器（比如插件）。 |
|  | 定义 和 的来源。 |
|  | 定义 和 的轨道。（主流浏览器都不支持） |
| **<video>** | 定义视频或影片内容。 |

### 2.5.4 新的图形标签

|  |  |
| --- | --- |
| 标签 | 描述 |
|  | 定义使用 JavaScript 的图像绘制。 |
|  | 定义使用 SVG 的图像绘制。 |

#### SVG和Canvas的区别+

* SVG 是一种使用 XML 描述 2D 图形的语言。
* Canvas 通过 JavaScript 来绘制 2D 图形。
* SVG 基于 XML，这意味着 SVG DOM 中的每个元素都是可用的。您可以为某个元素附加 JavaScript 事件处理器。
* 在 SVG 中，每个被绘制的图形均被视为对象。如果 SVG 对象的属性发生变化，那么浏览器能够自动重现图形。
* Canvas是逐像素进行渲染的，在 canvas 中，一旦图形绘制完成，就不会继续被浏览器关注。如果其位置发生变化，那么整个场景也需要重新绘制，包括任何或许已被图形覆盖的对象。而SVG是通过DOM操作来显示的。
* 所以Canvas的文本渲染能力弱，而SVG最适合带有大型渲染区域的应用程序，比如地图。
* 而Canvas 最适合有许多对象要被频繁重绘的图形密集型游戏。
* 而SVG由于DOM操作 在复杂度高的游戏应用中 会减慢渲染速度。所以不适合在游戏应用。

### 2.5.5 地理定位

window.navigator.geolocation {  
 getCurrentPosition: fn 用于获取当前的位置数据  
 watchPosition: fn 监视用户位置的改变  
 clearWatch: fn 清除定位监视  
}

### 2.5.6 拖放API

**拖动的源对象(source)可能触发的事件：**

* dragstart：拖动开始
* drag：拖动中
* dragend：拖动结束

**拖动的目标对象(target)可能触发的事件：**

* dragenter：拖动进入
* dragover：拖动悬停
* drop：松手释放
* dragleave：拖动离开

拖放API事件句柄中所有的事件对象都有一个dataTransfer属性（数据运输对象），用于在源对象和目标对象间传递数据。

**源对象**：event.dataTransfer.setData(key, value)

**目标对象**：var value = event.dataTransfer.getData(key)

### 2.5.7 WebWorker

Chrome浏览器中发起资源请求的有6个线程；但是只有1个线程负责渲染页面——称为UI主线程——浏览器中所有的代码只能由一个线程来执行。若浏览器加载了一个很耗时的JS文件(可能影响DOM树结构)，浏览器必须等待该文件执行完成才会继续执行后续的代码。执行耗时JS任务过程中，会暂停页面中一切内容的渲染以及事件的处理。

**Worker的本质**：就是一个执行指定任务的独立线程；且该线程可以与UI主线程进行消息数据传递。

**HTML文件中：**

var w = new Worker('js/x.js')  
w.postMessage('发送给Worker线程的消息');  
w.onmessage = function(e){  
 e.data; //来自Worker线程的消息  
}

**JS文件中：**

onmessage = function(e){  
 var data = e.data; //接收UI线程的消息  
 //执行耗时任务....  
 postMessage(result); //给UI线程发送消息  
}

**注意：**Worker任务不允许直接操作DOM树，也不允许使用任何的BOM对象！需要的数据只能由UI主线程来传递，处理的结果也必须交由UI线程来显示。

### 2.5.8 WebStorage+

#### Web存储结构

1. **服务器端存储**
   1. 数据库存储，如商品、用户等核心数据
   2. Session/内存存储，如用户的登录信息
2. 客户端存储
   1. Cookie存储，如用户偏好、访问历史，浏览器兼容性好但处理麻烦且容量限制
   2. H5 WebStorage存储，如用户偏好、访问历史等安全要求的数据，老IE不兼容但易使用且容量大

#### H5中的WebStorage

1. sessionStorage：类数组对象，通过key=>value对存储字符串数据——会话级存储
2. localStorage：类数组对象，通过key=>value对存储字符串数据——本地/跨会话级/永久存储

#### cookies、sessionStorage、localStora的区别

|  |  |  |  |
| --- | --- | --- | --- |
| 不同点 | 存储大小 | 有效时间 | 数据与服务器交互方式 |
| cookies | <=4K | 在设置cookie过期之前一直有效（无论窗口浏览器是否关闭） | 正常情况下，cookies数据会自动传到服务器，服务器也可以写cookie到客户端 |
| sessionstorage | 5M | 数据在当前浏览器关闭后删除（sessionStorage与存储数据的顶级窗口或浏览器选项卡具有相同的生命周期。） | 不会发送数据到服务端 |
| localstorage | 5M | 持久存储，浏览器关闭后不会丢失除非主动删除（直到Web应用程序删除它或用户要求浏览器删除它） | 不会发送数据到服务端 |

### 2.5.9 WebSocket

WebSocket 是 HTML5 开始提供的一种在单个 TCP 连接上进行全双工通讯的协议。

WebSocket 使得客户端和服务器之间的数据交换变得更加简单，**允许服务端主动向客户端推送数据**。在 WebSocket API 中，浏览器和服务器只需要完成一次握手，两者之间就直接可以创建持久性的连接，并进行双向数据传输。

## 2.6 iframe的优缺点？

### 2.6.1 优点

1. 能顾原封不动地把嵌入的网页展现出来；
2. 如果有多个网页引用iframe，那么仅需修改iframe的内容，就可以实现调用的每一个页面内容的修改，方便快捷；
3. 网页如果为了统一风格，头部和版本都一样，就可以写成一个页面，用iframe来嵌套，可以增加代码的可重用性；
4. 吐过遇到加载缓慢的第三方内容如图标和广告，那么可以由iframe来解决。

### 2.6.2 缺点

1. 会产生很多页面，不容易管理；
2. iframe框架结构有时会让人感到迷惑，如果框架个数多，可能会出现多个上下、左右滚动条，会分散访问者的注意力，用户体验差；
3. 代码复杂，无法被一些搜素引擎索引到，不能被搜索引擎爬虫处理，不利于SEO。
4. 很多移动设备无法完全显示框架，设备兼容性差。
5. iframe框架会增加服务器的http请求，对于大型网站来说不可取。

## 2.7 img中的alt与title属性

**alt：**在图片无法正确显示的时候起到文本替代的作用

**title：**鼠标滑过时显示的文字提示

## 2.8 meta标签

<meta> 元素可提供有关页面的元信息（meta-information），比如针对搜索引擎和更新频度的描述和关键词。

<meta> 标签位于文档的头部，不包含任何内容。<meta> 标签的属性定义了与文档相关联的名称/值对。

在 HTML 中，<meta>标签没有结束标签。

在 XHTML 中，<meta> 标签必须被正确地关闭。

**注释：**<meta> 标签永远位于 head 元素内部。元数据总是以名称/值的形式被成对传递的。

### 2.8.1 meta标签的属性

**必需**

|  |  |  |
| --- | --- | --- |
| 属性 | 值 | 描述 |
| [content](https://www.w3school.com.cn/tags/tag_meta.asp#meta_prop_content) | some\_text | 定义与 http-equiv 或 name 属性相关的元信息 |

## 可选的属性

|  |  |  |
| --- | --- | --- |
| 属性 | 值 | 描述 |
| [http-equiv](https://www.w3school.com.cn/tags/tag_meta.asp#meta_prop_http-equiv) | content-type<br/>expires<br/>refresh<br/>set-cookie | 把 content 属性关联到 HTTP 头部。 |
| [name](https://www.w3school.com.cn/tags/tag_meta.asp#meta_prop_name) | author<br/>description<br/>keywords<br/>generator<br/>revised<br/>others | 把 content 属性关联到一个名称。 |
| [scheme](https://www.w3school.com.cn/tags/tag_meta.asp#meta_prop_scheme) | some\_text | 定义用于翻译 content 属性值的格式。 |

# 3. CSS篇

## 3.1 行内元素有哪些？块级元素有哪些？

### 3.1.1 行内元素

* 和其他元素都在一行上；
* 高，行高及外边距和内边距不可改变；
* 宽度就是它的文字或图片的宽度，不可改变；
* 内联元素只能容纳文本或者其他内联元素；
* 对行内元素，需要注意如下：
  + 设置宽度width 无效。
  + 设置高度height 无效，可以通过line-height来设置。
  + 设置margin 只有左右margin有效，上下无效。
  + 设置padding 只有左右padding有效，上下则无效。注意元素范围是增大了，但是对元素周围的内容是没影响的。

**常见行内元素**

|  |  |
| --- | --- |
| 标签 | 描述 |
|  | 定义锚。 |
|  | 定义音频。 |
| <br> | 定义简单的折行。 |
|  | 定义按钮 (push button)。 |
|  | 定义计算机代码文本。 |
|  | 定义斜体字。 |
|  | 定义图像。 |
|  | 定义输入控件。 |
|  | 定义 input 元素的标注。 |
|  | 定义选择列表（下拉列表）。 |
|  | 定义文档中的节。 |
|  | 定义多行的文本输入控件。 |
|  | 定义视频。 |
| 其他和文本相关标签 |  |

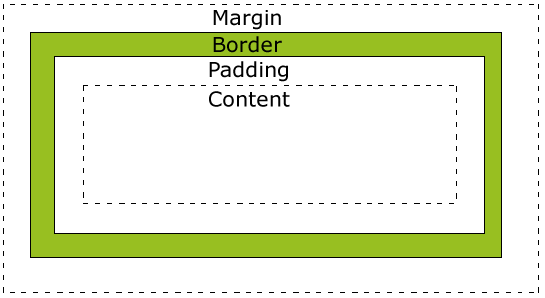
### 3.1.2 块级元素

* 总是在新行上开始；
* 高度，行高以及外边距和内边距都可控制；
* 宽度缺省是它的容器的100%，除非设定一个宽度。
* 它可以容纳内联元素和其他块元素

|  |  |
| --- | --- |
| 标签 | 描述 |
|  | 定义文章。 |
|  | 定义页面内容之外的内容。 |
|  | 定义图形。 |
|  | 定义表格标题。 |
|  | 定义定义列表中项目的描述。 |
|  | 定义文档中的节。 |
|  | 定义定义列表。 |
|  | 定义定义列表中的项目。 |
|  | 定义元素的细节。 |
|  | 定义围绕表单中元素的边框。 |
|  | 定义 figure 元素的标题。 |
|  | 定义媒介内容的分组，以及它们的标题。 |
|  | 定义 section 或 page 的页脚。 |
|  | 定义供用户输入的 HTML 表单。 |
| to | 定义 HTML 标题。 |
|  | 定义 section 或 page 的页眉。 |
|  | 定义水平线。 |
|  | 定义 fieldset 元素的标题。 |
|  | 定义列表的项目。 |
|  | 定义命令的列表或菜单。 |
|  | 定义预定义范围内的度量。 |
|  | 定义导航链接。 |
|  | 定义针对不支持框架的用户的替代内容。 |
|  | 定义针对不支持客户端脚本的用户的替代内容。 |
|  | 定义有序列表。 |
|  | 定义输出的一些类型。 |
|  | 定义段落。 |
|  | 定义预格式文本。 |
|  | 定义 section。 |
|  | 定义表格。 |
|  | 定义表格中的主体内容。 |
|  | 定义表格中的单元。 |
|  | 定义表格中的表注内容（脚注）。 |
|  | 定义表格中的表头单元格。 |
|  | 定义表格中的表头内容。 |
|  | 定义日期/时间。 |
|  | 定义表格中的行。 |
|  | 定义无序列表。 |

## 3.2 CSS盒模型

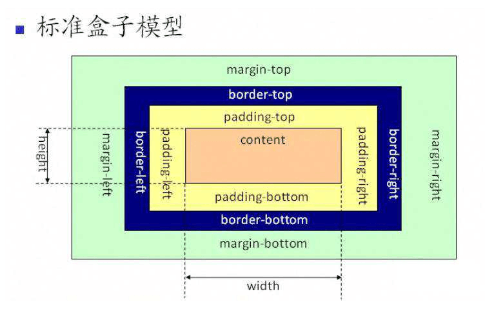
所有HTML元素可以看作盒子，在CSS中，"box model"这一术语是用来设计和布局时使用。CSS盒模型本质上是一个盒子，封装周围的HTML元素，它包括：**边距，边框，填充，和实际内容**。盒模型允许我们在其它元素和周围元素边框之间的空间放置元素。  
下面的图片说明了盒子模型(Box Model)：

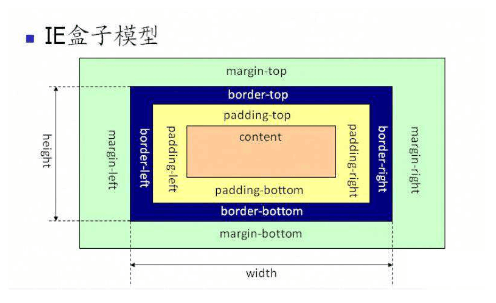


盒模型分为两种：**W3c标准的盒子模型（标准盒模型【content-box】）** 、第二种**IE标准的盒子模型（怪异盒模型【border-box】）**

**标准盒模型盒子大小=**content+border+padding+margin

**怪异盒模型盒子大小=\***\*width(content + border + padding) + margin





定义了完整的doctype的标准文档类型，无论是哪种模型情况，最终都会触发标准模式，如果doctype协议缺失，会由浏览器自己界定，在IE浏览器中IE9以下（IE6.IE7.IE8）的版本触发怪异模式，其他浏览器中会默认为W3c标准模式。

**通过属性box-sizing来设置盒子模型的解析模式**

**content-box**： **默认值**，border和padding不算到width范围内，可以理解为是W3c的标准模型(default)

**border-box**：border和padding划归到width范围内，可以理解为是IE的怪异盒模型

**padding-box：**将padding算入width范围

## 3.3 CSS浮动+

### 3.3.1 产生原因及现象

一个盒子设置了float属性，导致父级对象盒子不能被撑开。

### 3.3.2 浮动产生的副作用

1. **背景不能显示**：由于浮动产生，如果对父级设置了background，而父级不能被撑开，所以导致背景不能显示。
2. **边框不能撑开**
3. **margin、padding设置值不能正确显示**

### 3.3.3 清除浮动的方式+

1. **父级元素定义height**  
   原理：父级div手动定义height，就解决了父级div无法自动获取到高度的问题。  
   优点：简单，代码少，容易掌握  
   缺点：只适合高度固定的布局，要给出精确的高度，如果高度和父级div不一样时，会产生问题  
   建议：不推荐使用，只建议高度固定的布局时使用  
   评分：★★☆☆☆
2. **结尾处加空div标签 clear:both**  
   原理：添加一个空div，利用css提高的clear:both清除浮动，让父级div能自动获取到高度  
   优点：简单，代码少，浏览器支持好，不容易出现怪问题  
   缺点：不少初学者不理解原理；如果页面浮动布局多，就要增加很多空div，让人感觉很不爽  
   建议：不推荐使用，但此方法是以前主要使用的一种清除浮动方法  
   评分：★★★☆☆
3. **父级div定义 伪类:after 和 zoom**  
   原理：IE8以上和非IE浏览器才支持:after，原理和方法2有点类似，zoom(IE转有属性)可解决ie6,ie7浮动问题  
   优点：浏览器支持好，不容易出现怪问题（目前：大型网站都有使用，如：腾迅，网易，新浪等等）  
   缺点：代码多，不少初学者不理解原理，要两句代码结合使用，才能让主流浏览器都支持。  
   建议：推荐使用，建议定义公共类，以减少CSS代码。  
   评分：★★★★☆
4. **父级div定义 overflow:hidden**  
   原理：必须定义width或zoom:1，同时不能定义height，使用overflow:hidden时，浏览器会自动检查浮动区域的高度  
   优点：简单，代码少，浏览器支持好  
   缺点：不能和position配合使用，因为超出的尺寸的会被隐藏。  
   建议：只推荐没有使用position或对overflow:hidden理解比较深的朋友使用。  
   评分：★★★☆☆
5. **父级div定义 overflow:auto**  
   原理：同4，使用overflow:auto时，浏览器会自动检查浮动区域的高度。  
   优点：简单，代码少，浏览器支持好。  
   缺点：内部宽高超过父级div时，会出现滚动条。  
   建议：不推荐使用，如果你需要出现滚动条或者确保你的代码不会出现滚动条就使用吧。

## 3.4 CSS选择符都有哪些？哪些属性可以继承？优先级算法如何计算？

### 3.4.1 选择符有哪些？

#### 通配选择符

\* :{} 匹配所有的元素

#### 类型选择符

p:{} 匹配指定HTML元素

#### 属性选择符

1. input[attr] 选择具有某个属性的元素。
2. input[attr=value] 选择等于具体属性值的元素。
3. input[attr~=value] 根据属性值中的词列表的某个词进行选择。
4. input[attr^="value"] 子串匹配。以value开头。
5. input[attr$="value"] 子串匹配。以value结尾。
6. input[attr\*="value"] 子串匹配。包含value。
7. input[attr|=value] 特定属性选择类型。常用于匹配语言。

* <style>  
   \*[lang|="en"] {color: red;}  
  </style>  
  <p lang="en">Hello!</p> <!--可以匹配-->  
  <p lang="en-us">Greetings!</p> <!--可以匹配-->  
  <p lang="en-au">G'day!</p> <!--可以匹配-->  
  <p lang="fr">Bonjour!</p> <!--无法匹配-->  
  <p lang="cy-en">Jrooana!</p> <!--无法匹配-->

### 3.4.2 哪些属性可以继承

#### 字体系列属性

**font、font-family、font-weight、font-size、font-style**、font-variant、font-stretch、font-size-adjust

#### 文本系列属性

**color、text-align、line-height**、text-indent、word-spacing、letter-spacing、text-transform、direction

#### 元素可见性

visibility

#### 表格布局属性

caption-side、border-collapse、border-spacing、empty-cells、table-layout

#### 列表属性

list-style-type、list-style-image、list-style-position、list-style

#### 生成内容属性

quotes

#### 光标属性

cursor

#### 页面样式属性

page、page-break-inside、windows、orphans

#### 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

## 3.5 CSS3新增伪类有哪些？

1. :nth-child(n)选择父元素下的第n个子元素。n可以接受具体的数值，也可以接受函数（如4n-1）。需要注意的是，n是从1开始计算，而不是0。当函数中的n系数为负时，甚至可以达到选择前n个元素的效果。n为odd和even选择奇数偶数元素。
2. :nth-last-child倒数计算。
3. :last-child最后一个元素。
4. :only-child其父元素的唯一元素。
5. :nth-of-type(n)父元素下第n个elem元素，n接受的格式和nth-child一样。甚至在绝大多数情况下，nth-of-type的效果甚至和nth-child没有区别，那这两个伪类到底是什么区别呢。

* elem:nth-of-type(n)是“选择父元素下第n个elem元素”。
* 而elem:nth-child(n)是“这个伪类选中父元素下的第n个子元素，并且这个子元素的标签名为elem”。

1. elem:first-of-type和elem:last-of-type选中父元素下第1个/最后一个elem元素。
2. elem:only-of-type唯一的elem元素
3. elem:empty选择不包含子元素和内容的elem标签。
4. elem:enabled和elem:disabled可用、禁用状态
5. elem:checked匹配每个选中的input元素（适用于单选框和复选框）

## 3.6 如何居中div？++

<div id="father">  
 <div id="son">  
 </div>  
</div>

### 3.6.1 div行内元素 text-align: center(仅水平)

.father{  
 text-align:center;  
}  
  
.son{  
 display:inline-block;  
}

### 3.6.2 div块级元素 margin: 0 auto（仅水平）

.son{  
 margin:0 auto;  
 display:blck;  
}

### 3.6.3 脱离文档流 绝对定位（水平垂直）

.father{  
 position: relative;  
}  
  
.son{  
 position: absolute;  
 top: 50%;  
 left: 50%;  
   
 width: 10em;  
 height: 10em;  
   
 margin-top: -5em;  
 margin-left: -5em;  
}

使用calc简化

.son{  
 position: absolute;  
 top: calc(50% - 5em);  
 left: calc(50% - 5em);  
 width: 10em;  
 height: 10em;  
}

### 3.6.4 transform变形 （水平垂直）

.son{  
 position: absolute;  
 top:50%;  
 left:50%;  
   
 transform: translate(-50%,-50%);  
}

#### 基于flex布局，全屏（水平垂直）

.father{  
 display:flex;  
 min-height:100vh;  
}

#### 基于flex布局，父元素有宽高

.father{  
 display:flex;  
 align-items:center;  
 justfy-content:center;  
}

#### verticle-align

.son{  
 display:inline-block|table-cell|  
 verticle-align:middle;  
}

#### 隐藏节点

<style>  
 .father{  
 width:600px;  
 height:300px;  
 }  
  
 .hide{  
 width:50%;  
 height:25%;  
 }  
  
 .son{  
 width:50%;  
 height:50%;  
 }  
</style>  
<body>  
 <div class="father">  
 <div class="hide"></div>  
 <div class="son"></div>  
 </div>  
</body>

## 3.7 为什么要初始化CSS?

1. **浏览器差异**

* 因为不同浏览器对标签的默认值不同，如果没对css初始化会出现浏览器之间的页面显示差异

1. **提高编码质量**

* 如果不初始化，整个页面做完会很糟糕，重复的css样式很多

## 3.8 说说你对BFC规范的理解？++

### 3.8.1 何为BFC

在一个Web页面的CSS渲染中，块级格式化上下文 (Block Fromatting Context)是按照块级盒子布局的。W3C对BFC的定义如下：

浮动元素和绝对定位元素，非块级盒子的块级容器（例如 inline-blocks, table-cells, 和 table-captions），以及overflow值不为“visiable”的块级盒子，都会为他们的内容创建新的BFC（块级格式上下文）

### 3.8.2 创建BFC

* float的值不是none
* position 的值不是static或者relative
* display的值是inline-block,table-cell,flex,table-caption或者inline-flex
* overflow的值不是visible

### 3.8.3 BFC的特性

* 内部的盒子会在垂直方向上一个接一个放置
* 于同一个BFC的两个相邻盒子的margin会重叠
* 每个元素的左外边距与包含块的左边界相接触（从左到右），即使浮动元素也是如此
* BFC的区域不会与float的元素区域重叠
* 计算BFC的高度时，浮动子元素也参与计算
* BFC就是页面上的一个隔离的独立容器，容器里面的子元素不会影响到外面的元素，反之亦然

### 3.8.4 BFC在布局中的应用

* 要阻止margin重叠，只要将俩个元素放在不同BFC中即可
* 使得父元素包含子元素，常见的方式是为父元素设置overflow：hidden或者浮动父元素。根本原因在于创建BFC的元素，子浮动元素也会参与其高度计算，即不会产生高度塌陷问题
* 与浮动元素相邻的已生成BFC的元素不能与浮动元素互相覆盖。利用该特性可以作为多栏布局的一种实现方式. 特点在于左右俩栏的宽度固定，中间栏可以根据浏览器宽度自适应

## 3.9 讲讲 position float display 各有哪些取值，它们互相之间会如何影响？

### 3.9.1 position的取值

static(默认)、relative、absolute、fixed、inherit

#### static

始终处于文档流给予的位置。看起来好像没有用，但它可以快速取消定位，让top，right，bottom，left的值失效。在切换的时候可以尝试这个方法。z-index无效（其他才有效)

#### relative和absolute

relative和absolute都可以用于定位，区别在于前者的div还属于正常的文档流，后者已经是脱离了正常文档流，不占据空间位置，不会将父类撑开。定位原点relative是相对于它在正常流中的默认位置偏移，它原本占据的空间仍然保留；absolute相对于第一个position属性值不为static的父类。所以设置了position：absolute，其父类的该属性值要注意，而且overflow：hidden也不能乱设置，因为不属于正常文档流，不会占据父类的高度，也就不会有滚动条。

#### fixed

fixed旧版本IE不支持，却是很有用，定位原点相对于浏览器窗口，而且不能变。常用于header，footer，或者一些固定的悬浮div，随滚动条滚动又稳定又流畅，比JS好多了。fixed可以有很多创造性的布局和作用，兼容性是问题。

#### inherit

position：inherit。规定从父类继承position属性的值，所以这个属性也是有继承性的。IE不支持。

### 3.9.2 float的取值

none（默认）、left、right、inherit。

float：left(或right)，向左（或右）浮动，直到它的边缘碰到包含框或另一个浮动框为止。且脱离普通的文档流，会被正常文档流内的块框忽略。不占据空间，无法将父类元素撑开。

任何元素都可以浮动，浮动元素会生成一个块级框，不论它本身是何种元素。因此，没有必要为浮动元素设置display：block。

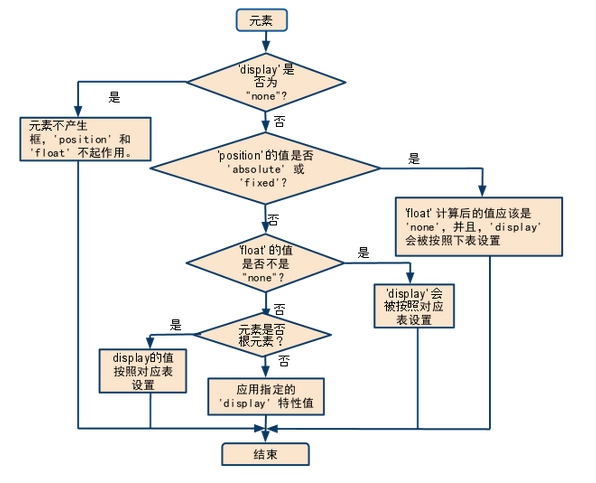
如果浮动非替换元素，则要指定一个明确的width，否则它们会尽可能的窄。（什么叫替换元素？根据元素本身的特点定义的， (X)HTML中的img、input、textarea、select、object都是替换元素，这些元素都没有实际的内容。 (X)HTML的大多数元素是不可替换元素，他们将内容直接告诉浏览器，将其显示出来。）

### 3.9.3 position的取值

**none** | **inline** | **block** | **list-item** | **inline-block** | **table** | **inline-table** | table-caption | table-cell | table-row | table-row-group | table-column | table-column-group | table-footer-group | table-header-group |run-in |box | inline-box | flexbox | inline-flexbox | **flex** | inline-flex | **gird** | inline-grid | **inherit**

**none：隐藏对象。与visibility属性的hidden值不同，其不为被隐藏的对象保留其物理空间**  
**inline：指定对象为内联元素。**  
**block：指定对象为块元素。**  
**list-item：指定对象为列表项目。**  
**inline-block：指定对象为内联块元素。（CSS2）**  
**table：指定对象作为块元素级的表格。类同于html标签<table>（CSS2）**  
inline-table：指定对象作为内联元素级的表格。类同于html标签<table>（CSS2）  
table-caption：指定对象作为表格标题。类同于html标签<caption>（CSS2）  
table-cell：指定对象作为表格单元格。类同于html标签<td>（CSS2）  
table-row：指定对象作为表格行。类同于html标签<tr>（CSS2）  
table-row-group：指定对象作为表格行组。类同于html标签<tbody>（CSS2）  
table-column：指定对象作为表格列。类同于html标签<col>（CSS2）  
table-column-group：指定对象作为表格列组显示。类同于html标签<colgroup>（CSS2）  
table-header-group：指定对象作为表格标题组。类同于html标签<thead>（CSS2）  
table-footer-group：指定对象作为表格脚注组。类同于html标签<tfoot>（CSS2）  
run-in：根据上下文决定对象是内联对象还是块级对象。（CSS3）  
box：将对象作为弹性伸缩盒显示。（伸缩盒最老版本）（CSS3）  
inline-box：将对象作为内联块级弹性伸缩盒显示。（伸缩盒最老版本）（CSS3）  
flexbox：将对象作为弹性伸缩盒显示。（伸缩盒过渡版本）（CSS3）  
inline-flexbox：将对象作为内联块级弹性伸缩盒显示。（伸缩盒过渡版本）（CSS3）  
**flex：将对象作为弹性伸缩盒显示。（伸缩盒最新版本）（CSS3）**  
inline-flex：将对象作为内联块级弹性伸缩盒显示。（伸缩盒最新版本）（CSS3）  
**grid：将网页划分成一个个网格，可以任意组合不同的网格，做出各种各样的布局（CSS3）**

### 3.9.4 相互影响



## 3.10 CSS绘制三角形+

### 3.10.1 border实现



#triangle{  
 width: 0;  
 height: 0;  
 border-left:50px solid transparent;  
 border-right: 50px solid transparent;  
 border-bottom: 100px solid red;  
}

### 3.10.2 clip-path实现

#triangle{  
 width:100px;  
 height:100px;  
 background: red;  
 clip-path:polygon(0,100%,50%,0,100%,100%);  
}

## 3.11 隐藏元素的八种方法

* visibility: hidden; 这个属性只是简单的隐藏某个元素，但是元素占用的空间任然存在
* opacity: 0; CSS3属性，设置0可以使一个元素完全透明
* position: absolute; 设置一个很大的 left 负值定位，使元素定位在可见区域之外
* display: none; 元素会变得不可见，并且不会再占用文档的空间。
* transform: scale(0); 将一个元素设置为缩放无限小，元素将不可见，元素原来所在的位置将被保留
* <div hidden="hidden"> HTML5属性,效果和display:none;相同，但这个属性用于记录一个元素的状态
* height: 0; 将元素高度设为 0 ，并消除边框
* filter: blur(0); CSS3属性，将一个元素的模糊度设置为0，从而使这个元素“消失”在页面中

## 3.12 flex是哪几个属性的缩写

flex: ***flex-grow* *flex-shrink* *flex-basis***|auto|initial|inherit;

|  |  |
| --- | --- |
| 值 | 描述 |
| *flex-grow* | 一个数字，规定项目将相对于其他灵活的项目进行扩展的量。 |
| *flex-shrink* | 一个数字，规定项目将相对于其他灵活的项目进行收缩的量。 |
| *flex-basis* | 项目的长度。合法值："auto"、"inherit" 或一个后跟 "%"、"px"、"em" 或任何其他长度单位的数字。 |
| auto | 与 1 1 auto 相同。 |
| none | 与 0 0 auto 相同。 |
| initial | 设置该属性为它的默认值，即为 0 1 auto。请参阅 [*initial*](https://www.runoob.com/cssref/css-initial.html)。 |
| inherit | 从父元素继承该属性。请参阅 [*inherit*](https://www.runoob.com/cssref/css-inherit.html)。 |

# 4. JS篇

## 4.1 面向对象

### 4.1.1 es5的方法与es6的方法对比

//es5  
function Person(name) {  
 //构造函数里面的方法和属性  
 this.\_name = name;  
 this.getName = function () {  
 console.log(this.\_name);  
 };  
 this.setName = function (name) {  
 this.\_name = name;  
 };  
 }  
   
 let p = new Person("张三");  
 p.getName(); // 张三  
 p.setName("李四");  
 p.getName(); // 李四

//es6  
// 定义类  
class Person {  
 //类的构造函数，实例化的时候执行，new的时候执行  
 constructor(name) {  
 this.\_name = name;  
 }  
 getName() {  
 console.log(this.\_name);  
 }  
 setName(name) {  
 this.\_name = name  
 }  
}  
 let p = new Preson('张三')  
 p.getName(); // 张三  
 p.setName('李四');  
 p.getName(); // 李四

### 4.1.2 面向对象的三大特征

#### 封装

我们平时所用的方法和类都是一种封装，当我们在项目开发中，遇到一段功能的代码在好多地方重复使用的时候，我们可以把他单独封装成一个功能的方法，这样在我们需要使用的地方直接调用就可以了。

#### 继承

继承在我们的项目开发中主要使用为子类继承父类。

**es6继承**

class Father {  
 constructor(name) {  
 this.\_name = name;  
 }　　　//实例方法，通过实例对象调用  
 getName() {  
 console.log(this.\_name);  
 }  
　　// 静态方法不会被继承,并且是通过类名去调用的　　　  
 static hitXiaoMing() {  
 console.log("打小明")  
 }  
}  
   
class Son extends Father {  
 constructor(name, age) {  
 //实例化子类的时候把子类的数据传给父类（这里的super必须有，super里的参数是所继承的父类实例化所需要的数据）  
 super(name);  
 this.\_age = age;  
 }   
}  
   
var DaMing = new Father('大明');  
Father.hitXiaoMing(); //打小明   
DaMing.getName(); //大明   
var XiaoMing = new Son('小明'，15);  
XiaoMing.getName(); //小明

#### 多态

多态的具体表现为方法重载和方法重写：

方法重载：重载是指不同的函数使用相同的函数名，但是函数的参数个数或类型不同。调用的时候根据函数的参数来区别不同的函数

方法重写：重写（也叫覆盖）是指在派生类中重新对基类中的虚函数（注意是虚函数）重新实现。即函数名和参数都一样，只是函数的实现体不一样

class Father {  
　　　constructor(name) {  
 this.\_name = name;  
 }  
　　　//实例方法，通过实例对象调用  
 getName() {  
 console.log(this.\_name);  
 } work() { 　　console.log('我的工作是累死累活，赚钱养家') }  
　　　// 静态方法不会被继承,并且是通过类名去调用的  
　　　static hitXiaoMing() {  
 console.log("打小明")  
 }  
 }  
   
 class Son extends Father {  
 constructor(name, age) {  
 //实例化子类的时候把子类的数据传给父类（这里的super必须有，super里的参数是所继承的父类实例化所需要的数据）  
 super(name);  
 this.\_age = age;  
 } work() { 　　console.log('我的工作是好好学习，天天向上。') }}  
   
 var DaMing = new Father('大明');  
 DaMing.work() // 我的工作是累死累活，赚钱养家。   
var XiaoMing = new Son('小明'，15);   
XiaoMing.work(); // 我的工作是好好学习，天天向上。

#### 三大特征的优点

封装：封装的优势在于定义只可以在类内部进行对属性的操作，外部无法对这些属性指手画脚，要想修改，也只能通过你定义的封装方法；

继承：继承减少了代码的冗余，省略了很多重复代码，开发者可以从父类底层定义所有子类必须有的属性和方法，以达到耦合的目的；

多态：多态实现了方法的个性化，不同的子类根据具体状况可以实现不同的方法，光有父类定义的方法不够灵活，遇见特殊状况就捉襟见肘了

## 4.2 闭包++++

### 4.2.1 闭包的定义

当一个函数的返回值是另外一个函数，而返回的那个函数如果调用了其父函数内部的其它变量，如果返回的这个函数在外部被执行，就产生了闭包。

### 4.2.2 表现形式

使函数外部能够调用函数内部定义的变量。

### 4.2.3 实例

var count=10; //全局作用域 标记为flag1  
function add(){  
 var count=0; //函数全局作用域 标记为flag2  
 return function(){  
 count+=1; //函数的内部作用域  
 alert(count);  
 }  
}  
var s = add()  
s();//输出1  
s();//输出2

### 4.2.4 闭包的作用

闭包可以用在许多地方。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**，不会在调用后被自动清除。

### 4.2.5 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**。

### 4.2.6 闭包的使用场景

#### 1.回调

闭包有用之处在于它可以将一些数据和操作它的函数关联起来。这和面向对象编程明显相似。在面对象编程中，我们可以将某些数据（对象的属性）与一个或者多个方法相关联。

因此，当你想只用一个方法操作一个对象时，可以使用闭包。

#### 2.模拟私有方法

私有方法不仅可以限制代码的访问权限，还提供了管理全局命名空间的强大能力，避免非核心的方法弄乱了代码的公共接口。

下面的代码说明了如何使用闭包定义能访问私有函数和私有变量的公有函数。这种方式也叫做模块模式：

var counter = (function() {  
 var privateCounter = 0;  
 function changeBy(val) {  
 privateCounter += val;  
 }  
 return {  
 increment: function() {  
 changeBy(1);  
 },  
 decrement: function() {  
 changeBy(-1);  
 },  
 value: function() {  
 return privateCounter;  
 }  
 };   
})();  
  
console.log(counter.value()); // logs 0  
counter.increment();  
counter.increment();  
console.log(counter.value()); // logs 2  
counter.decrement();  
console.log(counter.value()); // logs 1

## 4.3 防抖和节流+

### 4.3.1 防抖

对于短时间内连续触发的事件，防抖的含义就是让某个事件期限内，事件处理函数只执行一次。

**非立即执行版**

function debounce(func, wait) {//非立即执行版  
 let timeout;  
 return function () {  
 let context = this;  
 let args = arguments;  
  
 if (timeout) clearTimeout(timeout);  
   
 timeout = setTimeout(() => {  
 func.apply(context, args);  
 }, wait);  
 }  
}  
  
function scrollTop(){  
 console.log("触发了一次");  
}  
  
window.onscroll = debounce(scrollTop,1000);//隔1秒后再触发一次

**立即执行版**

function debounce(func, wait){//立即执行版  
 let timeout;  
 return function(){  
 let context = this;  
 let args = arguments;  
 if(timeout) clearTimeout(timeout);  
   
 let callNow = !timeout;  
 timeout = setTimeout(()=>{  
 timeout = null;  
 }, wait);  
 if(callNow) func.apply(context,args);  
 }  
}  
  
function scrollTop(){  
 console.log("触发了一次");  
}  
  
window.onscroll = debounce(scrollTop,1000);//每隔1秒触发一次

**组合版**：

/\*\*  
 \* @desc 函数防抖  
 \* @param func 函数  
 \* @param wait 延迟执行毫秒数  
 \* @param immediate true 表立即执行，false 表非立即执行  
 \*/  
function debounce(func,wait,immediate) {  
 let timeout;  
  
 return function () {  
 let context = this;  
 let args = arguments;  
  
 if (timeout) clearTimeout(timeout);  
 if (immediate) {  
 var callNow = !timeout;  
 timeout = setTimeout(() => {  
 timeout = null;  
 }, wait)  
 if (callNow) func.apply(context, args)  
 }  
 else {  
 timeout = setTimeout(function(){  
 func.apply(context, args);  
 }, wait);  
 }  
 }  
}

### 4.3.2 节流

**所谓节流，就是指连续触发事件但是在 n 秒中只执行一次函数。**节流会稀释函数的执行频率。对于节流，一般有两种方式可以实现，分别是时间戳版和定时器版。

**时间戳版**：

function throttle(func, wait) {  
 let previous = 0;  
 return function() {  
 let now = Date.now();  
 let context = this;  
 let args = arguments;  
 if (now - previous > wait) {  
 func.apply(context, args);  
 previous = now;  
 }  
 }  
}

**定时器版**：

function throttle(func, wait){  
 let timeout;  
 return function(){  
 let context = this;  
 let args = arguments;  
 if(!timeout){  
 timeout = setTimeout(()=>{  
 timeout = null;  
 func.apply(context, args);  
 }, wait);  
 }  
 }  
}

**组合版**：

/\*\*  
 \* @desc 函数节流  
 \* @param func 函数  
 \* @param wait 延迟执行毫秒数  
 \* @param type 1 表时间戳版，2 表定时器版  
 \*/  
function throttle(func, wait ,type) {  
 if(type===1){  
 let previous = 0;  
 }else if(type===2){  
 let timeout;  
 }  
 return function() {  
 let context = this;  
 let args = arguments;  
 if(type===1){  
 let now = Date.now();  
  
 if (now - previous > wait) {  
 func.apply(context, args);  
 previous = now;  
 }  
 }else if(type===2){  
 if (!timeout) {  
 timeout = setTimeout(() => {  
 timeout = null;  
 func.apply(context, args)  
 }, wait)  
 }  
 }  
 }  
}

## 4.3 继承+++

面向对象的基本特征有：封闭、继承、多态。

在JavaScript中实现继承的方法：

1. 原型链（prototype chaining）
2. call()/apply()
3. 混合方式(prototype和call()/apply()结合)
4. 对象冒充

### 4.3.1 prototype方法

function car(price){  
 this.price = price;  
}  
car.prototype.sayPrice = function(){  
 console.log("Price is "+this.price);  
}  
var oCar = new car("100W");  
oCar.sayPrice();  
   
function toyCar(price){  
 this.price = price;  
}  
toyCar.prototype = new car();  
var oCar2 = new toyCar("10CNY");  
oCar2.sayPrice();

### 4.3.2 call()/apply()方法

function useCall(a,b){  
 this.a = a;  
 this.b = b;  
 this.say = function(){  
 console.log("I'm "+this.a+" You're "+this.b);  
 }  
 }  
 function callThefunction (){  
 var args = arguments;  
 useCall.call(this,args[0],args[1]);  
 // useCall.apply(this,arguments);  
 }  
var testCall1 = new useCall("Not YY","Not TT");  
testCall1.say();  
   
var testCall2 = new callThefunction("YY","TT");  
testCall2.say();  
  
// I'm Not YY You're Not TT  
// I'm YY You're TT

### 4.3.3 混合方法

//line=readline()  
//print(line)  
  
var Person = function(name,age){  
 this.name = name;  
 this.age = age;  
}  
  
var Employee = function(name,age,id){  
 this.id = id;  
 Person.call(this,name,age);  
}  
  
Employee.prototype = new Person();  
  
var employee = new Employee("Jack", 20, 1);  
console.log(employee.name);  
console.log(employee instanceof Employee);  
  
console.log(employee instanceof Person);

### 4.3.4 对象冒充

function Person(name,age){  
 this.name = name;  
 this.age = age;  
 this.show = function(){  
 console.log(this.name+", "+this.age);  
 }  
}  
Person.prototype.sayHi = function(){  
 alert('hi');  
}  
   
function Student(name,age){   
 this.student = Person; //将Person类的构造函数赋值给this.student  
 this.student(name,age); //js中实际上是通过对象冒充来实现继承的  
 delete this.student; //移除对Person的引用  
}  
   
var s = new Student("小明",17);  
s.show();  
   
var p = new Person("小花",18);  
p.show();  
// 小明, 17  
// 小花, 18

### 4.3.5 ES6方法

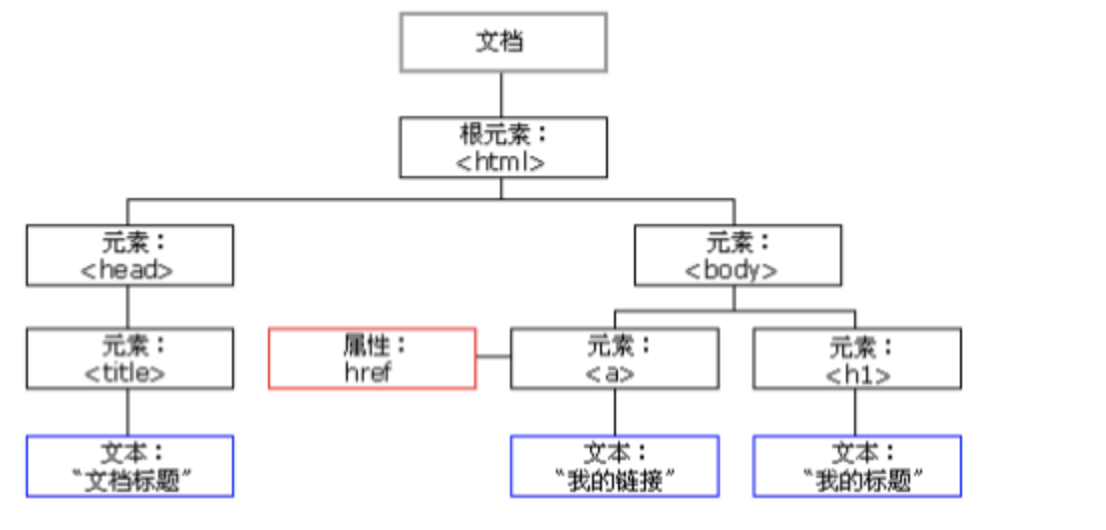
class Father {  
 constructor(name) {  
 this.\_name = name;  
 }　　　//实例方法，通过实例对象调用  
 getName() {  
 console.log(this.\_name);  
 }  
　　// 静态方法不会被继承,并且是通过类名去调用的　　　  
 static hitXiaoMing() {  
 console.log("打小明")  
 }  
}  
   
class Son extends Father {  
 constructor(name, age) {  
 //实例化子类的时候把子类的数据传给父类（这里的super必须有，super里的参数是所继承的父类实例化所需要的数据）  
 super(name);  
 this.\_age = age;  
 }   
}

## 4.4 DOM操作（添加、移除、移动、复制、创建和查找节点）

### 4.4.1 DOM及DOM树

文档对象模型。DOM 为文档提供了结构化表示，并定义了如何通过脚本来访问文档结构。目的其实就是为了能让js操作html元素而制定的一个规范。

DOM就是由节点组成的。



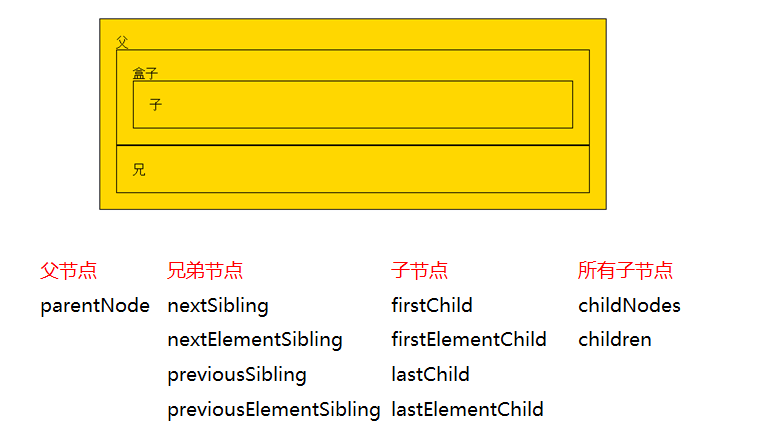
DOM树

* **元素节点**：HMTL标签。
* **文本节点**：标签中的文字（比如标签之间的空格、换行）
* **属性节点**：：标签的属性。

### 4.4.2 查找节点

var div1 = document.getElementById("box1"); //方式一：通过id获取单个标签  
var arr1 = document.getElementsByTagName("div1"); //方式二：通过 标签名 获得 标签数组，所以有s  
var arr2 = document.getElementsByClassName("hehe"); //方式三：通过 类名 获得 标签数组，所以有s  
  
document.getElementsByTagName("div1")[0]; //取数组中的第一个元素  
document.getElementsByClassName("hehe")[0]; //取数组中的第一个元素

#### 查找父节点、子节点及兄弟结点



**childNodes**：标准属性。返回的是指定元素的**子节点**的集合（包括元素节点、所有属性、文本节点）。是W3C的亲儿子。返回一个实时的 NodeList ，表示元素的子节点列表，注意子节点可能包含文本节点、注释节点等。

**children**：非标准属性。返回的是指定元素的**子元素节点**的集合。【重要】返回一个实时的 HTMLCollection ，子节点都是Element。

#### nodeType

这里讲一下nodeType。

* **nodeType == 1 表示的是元素节点**（标签） 。记住：元素就是标签。
* nodeType == 2 表示是属性节点 了解
* nodeType == 3 是文本节点 了解

### 4.4.3 添加

#### 创建元素

let elem = document.createElement("div");  
elem.id = 'newDiv';   
elem.style = 'color: red';   
elem.innerHTML = '我是新创建的测试节点';   
document.body.appendChild(elem);

#### 创建文本节点

var node = document.createTextNode("我是文本节点");   
document.body.appendChild(node);

#### 克隆节点

node.cloneNode(true/false) ，它接收一个bool参数，用来表示是否复制子元素。

var from = document.getElementById("test");   
var clone = from.cloneNode(true);   
clone.id = "test2";   
document.body.appendChild(clone);

### 4.4.4 修改

#### 插入节点（在父节点的最后插入）

var father = document.getElementById("father");   
var son = document.getElementById("son");  
father.appendChild(son);

#### 插入节点（前插）

parentNode.insertBefore(newNode, refNode);

#### 插入临近节点

targetElement.insertAdjacentElement(position, element);

#### 替换节点

parent.replaceChild(newChild, oldChild);

#### 删除节点

removeChild用于删除指定的子节点并返回子节点，语法：

var deletedChild = parent.removeChild(node);

### 4.4.5 getElementByXXX与querySelector的区别

getElementByXXX 获取的是动态集合，querySelector获取的是静态集合（选出的所有元素的数组，不会随着文档操作而改变。）

## 4.5 new 操作符

new操作符新建了一个空对象，这个对象原型指向构造函数的prototype，执行构造函数后返回这个对象

1. 创建一个空的对象

* var obj = new Object();

1. 链接到原型(当调用构造函数创建一个新实例后，该实例的内部将包含一个指针（内部属性），指向构造函数的原型对象)

* obj.\_\_proto\_\_= Func.prototype;

1. 绑定this指向，执行构造函数

* 让Func中的this指向obj，并执行Func的函数体。(创建新的对象之后，将构造函数的作用域赋给新对象[因此this就指向了这个新对象])
* var result =Func.call(obj);

1. 确保返回的是对象

* 判断Func的返回值类型：如果是值类型，返回obj。如果是引用类型，就返回这个引用类型的对象
* if (typeof(result) == "object"){  
   func=result;  
  }  
  else{  
   func=obj;  
  }

默认情况下函数返回值为undefined，即没有显示定义返回值的话，但构造函数例外，new构造函数在没有return的情况下默认返回新创建的对象。

但是，在有显示返回值的情况下，如果返回值为基本数据类型{string，number，null，undefined，Boolean}，返回值仍然是新创建的对象。

只有在显示返回一个非基本数据类型时，函数的返回值才为指定对象。在这种情况下，this所引用的值就会被丢弃了。

var new1 = function(fun){  
 var newObj = Object.create(fun.prototype); //创建一个新对象newObj，并让 newObj.\_\_proto\_\_ 指向 fun，即 newObj.\_\_proto\_\_ === fun 返回true。  
 var returnObj = fun.call(newObj);  
 if(typeof returnObj === 'object'){  
 return returnObj  
 }else{  
 return newObj  
 }  
}

## 4.6 HTML5离线存储

## 4.7 JS的数据类型+

### 4.7.1 基本数据类型

Number，String，Boolean，undefined，null，Symbol（es6新增）

### 4.7.2 引用数据类型

Object，其中包含Function、Array，Date等

### 4.7.3 判断数据类型

#### typeof

返回数据类型，包含这7种： number、boolean、symbol、string、object、undefined、function。

typeof null 返回object

引用类型，除了function返回function类型外，其他均返回object。

#### toString

toString() 是 Object 的原型方法，调用该方法，默认返回当前对象的 [[Class]] 。这是一个内部属性，其格式为 [object Xxx] ，其中 Xxx 就是对象的类型。

Object.prototype.toString.call('') ; // [object String]  
Object.prototype.toString.call(1) ; // [object Number]  
Object.prototype.toString.call(true) ; // [object Boolean]  
Object.prototype.toString.call(Symbol()); //[object Symbol]  
Object.prototype.toString.call(undefined) ; // [object Undefined]  
Object.prototype.toString.call(null) ; // [object Null]  
Object.prototype.toString.call(new Function()) ; // [object Function]  
Object.prototype.toString.call(new Date()) ; // [object Date]  
Object.prototype.toString.call([]) ; // [object Array]  
Object.prototype.toString.call(new RegExp()) ; // [object RegExp]  
Object.prototype.toString.call(new Error()) ; // [object Error]  
Object.prototype.toString.call(document) ; // [object HTMLDocument]  
Object.prototype.toString.call(window) ; //[object global] window 是全局对象 global 的引用

#### constructor

constructor是原型prototype的一个属性，当函数被定义时候，js引擎会为函数添加原型prototype，并且这个prototype中constructor属性指向函数引用， 因此重写prototype会丢失原来的constructor。

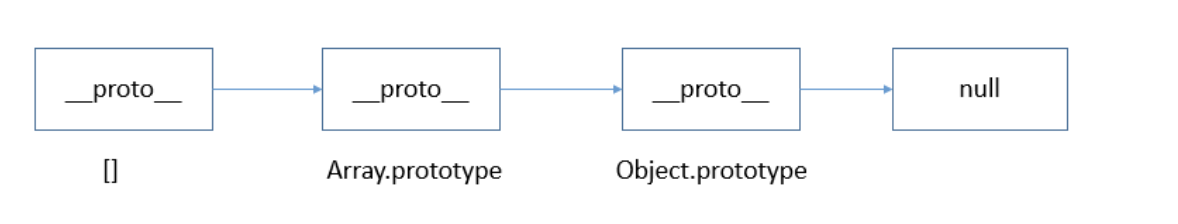
1：null 和 undefined 无constructor，这种方法判断不了。

2：还有，如果自定义对象，开发者重写prototype之后，原有的constructor会丢失，因此，为了规范开发，在重写对象原型时一般都需要重新给 constructor 赋值，以保证对象实例的类型不被篡改。

''.constructor == String //true  
new Number(1).constructor == Number // true  
true.constructor == Boolean //true  
new Function().constructor == Function //true  
new Date().constructor == Date //true  
new Error().constructor == Error //true  
[].constructor == Array //true  
document.constructor == HTMLDocument //true  
window.constructor == Window //true

#### instanceof

instanceof 是用来判断 A 是否为 B 的实例，表达式为：A instanceof B，如果 A 是 B 的实例，则返回 true,否则返回 false。 在这里需要特别注意的是：**instanceof 检测的是原型**



**由上图可以看出[]的原型指向Array.prototype，间接指向Object.prototype, 因此 [] instanceof Array 返回true， [] instanceof Object 也返回true。**

**instanceof 只能用来判断两个对象是否属于实例关系\***\*， 而不能判断一个对象实例具体属于哪种类型。\*\*

## 4.8 null和undefined的区别

console.log(null==undefined)//true  
console.log(null===undefined)//false

## 4.9 call()和apply()的区别+

fun.call(thisobj,arg1,arg2) ;当后面参数个数确定时用call,参数之间用逗号连接

fun.apply(thisobj,[arg1,arg2])；当后面参数个数不确定时用apply，参数通过数组形式输入

**call和apply的作用**：

改变this的指向，第一个参数为你要传入的对象，传入后函数的this就指向了这个对象,后面的参数为你为函数传递的参数值

## 4.10 深拷贝和浅拷贝+

深拷贝和浅拷贝最根本的区别在于是否真正获取一个对象的复制实体，而不是引用。

### 4.10.1 浅拷贝（shallowCopy）

1. 对于**基本数据类型**的成员对象，因为基础数据类型是值传递的，所以是**直接将属性值赋值给新的对象**。基础类型的拷贝，其中一个对象修改该值，不会影响另外一个。
2. 对于**引用类型**，比如数组或者类对象，因为引用类型是引用传递，**所以浅拷贝只是把内存地址赋值给了成员变量，它们指向了同一内存空间**。改变其中一个，会对另外一个也产生影响。

object.assign(target,source)

**Object.assign** 方法只复制源对象中可枚举的属性和对象自身的属性  
 如果目标对象中的属性具有相同的键，则属性将被源中的属性覆盖。后来的源的属性将类似地覆盖早先的属性  
 Object.assign 会跳过那些值为 null的属性

### 4.10.2 深拷贝（deepCopy）

是增加了一个指针并且申请了一个新的内存，使这个增加的指针指向这个新的内存。

#### 深拷贝的方法

1. **JSON.parse(JSON.stringify())**

* 缺点：不可拷贝一些特殊属性，例如正则表达式，undefined和function
* function deepCopy(o) {  
   return JSON.parse(JSON.stringify(o))  
  }  
    
  var c = {  
   age: 1,  
   name: undefined,  
   sex: null,  
   tel: /^1[34578]\d{9}$/,  
   say: () => {  
   console.log('hahha')  
   }  
  }  
  // { age: 1, sex: null, tel: {} }

1. **递归调用，复制所有层级属性**

* var obj = {  
   a:1,  
   arr: [1,2]，  
   nation : '中国',  
   birthplaces:['北京','上海','广州']  
  };  
  var obj2 = {name:'杨'};  
  obj2 = deepCopy(obj,obj2);  
  console.log(obj2);  
  //深复制，要想达到深复制就需要用递归  
  function deepCopy(o, c){  
   var c = c || {};  
   for(var i in o){  
   if(typeof o[i] === 'object'){  
   if(o[i].constructor === Array){  
   //这是数组  
   c[i] = [];  
   }else{  
   //这是对象  
   c[i] = {};  
   }  
   deepCopy(o[i], c[i]);  
   }else{  
   c[i] = o[i];  
   }  
   }  
   return c;  
  }

## 4.11 ajax

AJAX 是一种在无需重新加载整个网页的情况下，能够更新部分网页的技术，它能够让我们前端去请求服务器，然后服务器返回给前端必要的信息，然后我们就可以根据服务器返回的结果进行界面更新等操作，这样就保证我们可以不用去刷新请求整个页面，局部改变Dom界面。

### 4.11.1 创建原生Ajax对象

**XMLHttpRequest** 是 AJAX 的基础，所有现代浏览器（IE7+、Firefox、Chrome、Safari 以及 Opera）均内建 XMLHttpRequest 对象。那么我们下面来创建一个ajax对象

const instance = new XMLHttpRequest();

这样我们就创建了一个用于进行ajax交互的对象。

当然我们这儿提及一下老版本的ie5、ie6的javascript对象都是使用ActiveX来实现的标准，所以没有XMLHttpRequest对象。他们创建一个对象使用这种方式

const instance = new ActiveXObject("Microsoft.XMLHTTP");

### 4.11.2 发送请求

const instance = new XMLHttpRequest();  
instance.open(method,url,async);  
instance.send(string);

### 4.11.3 Ajax通过readState状态变化来接收服务器响应

当请求被发送到服务器时，我们需要执行一些基于响应的任务，每当 readyState 改变时，就会触发 onreadystatechange 事件，readyState 属性存有 XMLHttpRequest 的状态信息。

|  |  |
| --- | --- |
| 属性 | 描述 |
| onreadystatechange | 存储函数（或函数名），每当readyState属性改变时，就会调用该函数 |
| readState | 存有XMLHTTPRequest的状态。取值为0-4。<br />0：请求未初始化；<br />1：服务器连接已建立<br />2：请求已接收<br />3：请求处理中<br />4：请求已完成，且响应已就绪 |
| status | 200：“OK”<br />404：“未找到请求的资源” |

## 4.12 数组去重+

### 4.12.1 使用ES6中的Set方法

let arr = [1,0,0,2,9,8,3,1];  
function unique(arr) {  
 return Array.from(new Set(arr))  
}  
console.log(unique(arr)); // [1,0,2,9,8,3]

## 4.13 this对象+

### 4.13.1 函数的几种调用方式

* 普通函数调用
* 作为方法来调用
* 作为构造函数来调用
* 使用apply/call方法来调用
* Function.prototype.bind方法
* es6箭头函数

谁调用这个函数或方法,this关键字就指向谁。

### 4.13.2 普通函数调用

function person(){  
 this.name="xl";  
 console.log(this);  
 console.log(this.name);  
}  
  
person(); //输出 window xl

在这段代码中person函数作为普通函数调用，实际上person是作为全局对象window的一个方法来进行调用的,即window.person();  
所以这个地方是window对象调用了person方法,那么person函数当中的this即指window,同时window还拥有了另外一个属性name,值为xl.

var name="xl";  
function person(){  
 console.log(this.name);  
}  
  
person(); //输出 xl

这个地方person作为window的方法来调用，在代码的一开始定义了一个全局变量name，值为xl,它相当于window的一个属性,即window.name="xl",又因为在调用person的时候this是指向window的，因此这里会输出xl.

### 4.13.3 作为方法来调用

var name="XL";  
var person = {  
 name:"xl",  
 showName:function(){  
 console.log(this.name);  
 }  
}  
person.showNamr(); //输出 xl  
//这里是person对象调用showName方法，很显然this关键字是指向person对象的，所以会输出name  
  
var showNameA = person.showName;  
shownameA(); //输出 XL  
//这里将person.showName方法赋给showNameA变量，此时showNameA变量相当于window对象的一个属性，因此showNameA()执行的时候相当于window.showNameA(),即window对象调用showNameA这个方法，所以this关键字指向window

另外一种形式

var person = {  
 name:"xl",  
 showName:function(){  
 console.log(this.name);  
 }  
}  
var personB = {  
 name:"XL",  
 sayName:person.showName  
}  
personB.sayName(); // 输出XL  
//虽然showName方法是在person这个对象中定义，但是调用的时候却是在personB这个对象中调用，因此this对象指向

### 4.13.4 作为构造函数调用

function Person(name){  
 this.name = name;  
}  
  
var personA = Person("xl");  
console.log(personA.name); //输出 undefined  
console.log(window.name); //输出 xl  
  
  
var personB = new Person("xl");  
console.log(personB.name); //输出 xl

1. 因此new Person("xl")返回了一个继承了Person.prototype对象上的属性和方法，以及拥有name属性为"xl"的对象，并将它赋给变量personB.
2. 所以console.log(personB.name)会输出"xl"

### 4.13.5 call/apply方法调用

在JS里函数也是对象，因此函数也有方法。从Function.prototype上继承到Function.prototype.call/Function.prototype.apply方法  
call/apply方法最大的作用就是能改变this关键字的指向.

var name="XL";  
var Person = {  
 name:"x1",  
 showName:function(){  
 console.log(this.name);  
 }  
}  
Person.showName.call(); //输出 XL  
//这里call方法里面的第一个参数为空，默认指向window。  
//虽然showName方法定义在Person对象里面，但是使用call方法后，将showName方法里面的this指向了window。因此最后会输出"XL";  
  
function FruitA(n1,n2){  
 this.n1 = n1;  
 this.n2 = n2;  
 this.change = function(x,y){  
 this.n1 = x;  
 this.n2 = y;  
 }  
}  
  
var fruitA = new FruitA("chery","banana");  
var FruitB = {  
 n1:"apple",  
 n2:"orange"  
};  
  
fruitA.change.call(FruitB,"pear","peach");  
  
console.log(FruitB.n1); //输出 pear  
console.log(FruitB.n2); //输出 peach  
  
//FruitB调用fruitA的change方法，将fruitA中的this绑定到对象FruitB上。  
  
fruitA.change.apply(fruitB,["grape","watermelon"]);  
console.log(FruitB.n1); //输出 grape  
console.log(FruitB.n2); //输出 watermelon

### 4.13.6 bind()方法

var name="XL";  
function Person(name){  
 this.name = name;  
 this.sayName = function(){  
 setTimeout(function(){  
 console.log("My name is "+this.name);  
 },50);  
 }  
}  
var person = new Person("xl");  
  
person.sayName(); //输出 My name is XL  
//这里的setTimeout()定时函数,相当于window.setTimeout(),由window这个全局对象对调用,因此this的指向为window, 则this.name则为XL

**使用bind改变this指向**

var name="XL";  
function Person(name){  
 this.name = name;  
 this.sayName = function(){  
 setTimeout(function(){  
 console.log("My name is "+this.name);  
 }.bind(this),50); //注意这个地方使用的bind()方法，绑定setTimeout里面的匿名函数的this一直指向Person对象  
 }  
}  
var person = new Person("xl");  
  
person.sayName(); //输出 My name is xl

setTimeout/setInterval/匿名函数执行的时候，this默认指向window对象，除非手动改变this的指向。在《javascript高级程序设计》当中，写到：“超时调用的代码(setTimeout)都是在全局作用域中执行的，因此函数中的this的值，在非严格模式下是指向window对象，在严格模式下是指向undefined”。本文都是在非严格模式下的情况。

超时调用的代码(setTimeout)都是在全局作用域中执行的，因此函数中的this的值，在非严格模式下是指向window对象，在严格模式下是指向undefined

### 4.13.7 Eval函数

该函数执行的时候，this绑定到当前作用域的对象上。

var name = "XL";  
var person = {  
 name:"xl",  
 showName:function(){  
 eval("console.log(this.name)");  
 }  
}  
  
person.showName(); //输出 "xl"  
  
var a = person.showName;  
a(); //输出 "XL"

### 4.13.8 箭头函数

es6里面this指向固定化，始终指向外部对象，因为箭头函数没有this,因此它自身不能进行new实例化,同时也不能使用call, apply, bind等方法来改变this的指向

function Timer() {  
 this.seconds = 0;  
 setInterval( () => this.seconds ++, 1000);  
}  
  
var timer = new Timer();  
  
setTimeout(()=>console.log(timer.seconds),3100); //输出 3  
// 在构造函数内部的setInterval()内的回调函数，this始终指向实例化的对象，并获取实例化对象的seconds的属性,每1s这个属性的值都会增加1。否则最后在3s后执行setTimeOut()函数执行后输出的是0

## 4.14 eval()

eval() 函数可计算某个字符串，并执行其中的的 JavaScript 代码。

**返回值**

通过计算 string 得到的值（如果有的话）。

**说明**

该方法只接受原始字符串作为参数，如果 string 参数不是原始字符串，那么该方法将不作任何改变地返回。因此请不要为 eval() 函数传递 String 对象来作为参数。

如果试图覆盖 eval 属性或把 eval() 方法赋予另一个属性，并通过该属性调用它，则 ECMAScript 实现允许抛出一个 EvalError 异常。

**抛出**

如果参数中没有合法的表达式和语句，则抛出 SyntaxError 异常。

如果非法调用 eval()，则抛出 EvalError 异常。

如果传递给 eval() 的 Javascript 代码生成了一个异常，eval() 将把该异常传递给调用者。

## 4.15 Generator 生成器

Generator可以理解为一个状态机，内部封装了很多状态，同时返回一个迭代器Iterator对象。可以通过这个迭代器遍历相关的值及状态。  
 Generator的显著特点是**可以多次返回**，每次的返回值作为迭代器的一部分保存下来，可以被我们显式调用。

### 4.15.1 声明

一般的函数使用function声明，return作为回调(没有遇到return，在结尾调用return undefined)，只可以回调一次。而Generator函数使用function\*定义，除了return，还使用yield返回多次。

function\* foo(x) {  
 yield x + 1;  
 yield x + 2;  
 return x + 3;  
}  
const result = foo(0) // foo {<suspended>}  
result.next(); // {value: 1, done: false}  
result.next(); // {value: 2, done: false}  
result.next(); // {value: 3, done: true}  
result.next(); //{value: undefined, done: true}

### 4.15.2 函数方法

#### next/return/throw

**next方式**是按步执行，每次返回一个值,同时也可以每次传入新的值作为计算

function\* foo(x) {  
 let a = yield x + 1;  
 let b= yield a + 2;  
 return x + 3;  
}  
const result = foo(0) // foo {<suspended>}  
result.next(1); // {value: 1, done: false}  
result.next(2); // {value: 4, done: false}  
result.next(3); // {value: 3, done: true}  
result.next(4); //{value: undefined, done: true}

**return**则直接跳过所有步骤，直接返回 {value: undefined, done: true}

**throw**则根据函数中书写try catch返回catch中的内容，如果没有写try，则直接抛出异常

function\* foo(x) {  
 try{  
 yield x+1  
 yield x+2  
 yield x+3  
 yield x+4  
   
 }catch(e){  
 console.log('catch it')  
 }  
}  
const result = foo(0) // foo {<suspended>}  
result.next(); // {value: 1, done: false}  
result.next(); // {value: 2, done: false}  
result.throw(); // catch it {value: undefined, done: true}  
result.next(); //{value: undefined, done: true}

上述代码正常。

function\* foo(x) {  
 yield x+1  
 try{  
 yield x+2  
 }catch(e){  
 console.log('catch it')  
 }  
}  
const result = foo(0) // foo {<suspended>}  
result.next(); // {value: 1, done: false}  
result.throw(); // Uncaught undefined  
result.next(); //{value: undefined, done: true}

执行throw的时候，还没有进入到try语句，所以直接抛错，抛出undefined为throw未传参数，如果传入参数则显示为传入的参数。此状态与未写try的抛错状态一致。

#### 遍历

Generator函数的返回值是一个带有状态的Generator实例。它可以被for of 调用，进行遍历，且只可被for of 调用。此时将返回他的所有状态。

function\* foo(x) {  
console.log('start')  
 yield x+1  
 console.log('state 1')  
 yield x+2  
 console.log('end')  
}  
const result = foo(0) // foo {<suspended>}  
for(let i of result){  
 console.log(i)  
}  
//start  
//1  
//state 1  
//2  
//end  
result.next() //{value: undefined, done: true}  
//调用for of方法后，在后台调用next(),当done属性为true的时候，循环退出。因此Generator函数的实例将顺序执行一遍，再次调用时，状态为已完成

#### 状态存储和改变

function\* foo(x) {  
 let a = yield x + 0;  
 let b= yield a + 2;  
 yield x;  
 yield a   
 yield b  
}  
const result = foo(0)  
result.next() // {value: 0, done: false}  
result.next(2) // {value: 4, done: false}  
result.next(3) // {value: 0, done: false}  
result.next(4) // {value: 2, done: false}  
result.next(5) // {value: 3, done: false}

以上的执行结果中，我们可以看到，在第二步的时候，我们传入2这个参数，foo函数中的a的变量的值0被替换为2，并且在第4次迭代的时候，返回的是2。而第三次迭代的时候，传入的3参数，替换了b的值4，并在第5次迭代的时候返回了3。**所以传入的参数，是替代上一次迭代的生成值**。

#### yield\* 委托

有时需要将多个迭代器的值合在一起，我们可以使用yield \*的形式，将执行委托给另外一个Generator函数

function\* foo1() {  
 yield 1;  
 yield 2;  
 return "foo1 end";  
}  
  
function\* foo2() {  
 yield 3;  
 yield 4;  
}  
  
function\* foo() {  
 yield\* foo1();  
 yield\* foo2();  
 yield 5;  
}  
  
const result = foo();  
  
console.log(iterator.next());// "{ value: 1, done: false }"  
console.log(iterator.next());// "{ value: 2, done: false }"  
console.log(iterator.next());// "{ value: 3, done: false }"  
console.log(iterator.next());// "{ value: 4, done: false }"  
console.log(iterator.next());// "{ value: 5, done: false }"  
console.log(iterator.next());// "{ value: undefined, done: true }"

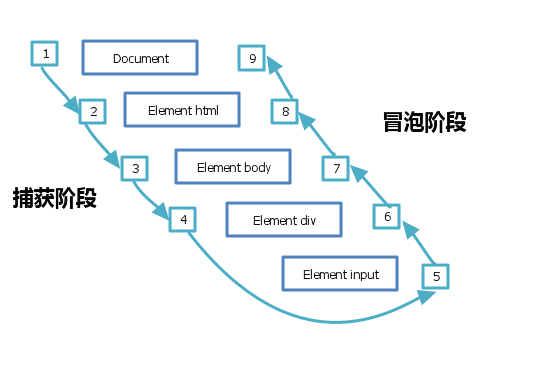
foo在执行的时候，首先委托给了foo1，等foo1执行完毕，再委托给foo2。但是我们发现，”foo1 end” 这一句并没有输出。  
 在整个Generator中，return只能有一次，在委托的时候，所有的yield\*都是以函数表达式的形式出现。return的值是表达式的结果，在委托结束之前其内部都是暂停的，等待到表达式的结果的时候，将结果直接返回给foo。此时foo内部没有接收的变量，所以未打印。  
 如果我们希望捕获这个值，可以使用yield \*foo()的方式进行获取。

## 4.16 什么是事件委托？

事件委托，通俗地来讲，就是把一个元素响应事件（click、focus……）的函数委托到另一个元素；

### 4.16.1 事件模型的三个阶段

* 捕获阶段：在事件冒泡的模型中，捕获阶段不会响应任何事件；
* 目标阶段：目标阶段就是指事件响应到触发事件的最底层元素上；
* 冒泡阶段：冒泡阶段就是事件的触发响应会从最底层目标一层层地向外到最外层（根节点），事件代理即是利用事件冒泡的机制把里层所需要响应的事件绑定到外层；



### 4.16.2 事件委托的优点

#### 减少内存消耗

若果我们有一个列表，列表之中有大量的列表项，我们需要在点击列表项的时候响应一个事件；

<ul id="list">  
 <li>item 1</li>  
 <li>item 2</li>  
 <li>item 3</li>  
 <!-- 多个li -->  
 <li>item n</li>  
</ul>

如果给每个列表项一一都绑定一个函数，那对于内存消耗是非常大的，效率上需要消耗很多性能；

因此，比较好的方法就是把这个点击事件绑定到他的父层，也就是 ul 上，然后在执行事件的时候再去匹配判断目标元素；

所以事件委托可以减少大量的内存消耗，节约效率。

#### 动态事件绑定

在很多时候，我们需要通过 AJAX 或者用户操作动态的增加或者去除列表项元素，那么在每一次改变的时候都需要重新给新增的元素绑定事件，给即将删去的元素解绑事件；

如果用了事件委托就没有这种麻烦了，因为**事件是绑定在父层的，和目标元素的增减是没有关系的**，执行到目标元素是在真正响应执行事件函数的过程中去匹配的；

所以使用事件在动态绑定事件的情况下是可以减少很多重复工作的。

### 4.16.3 jQuery 中的事件委托

jQuery 中的事件委托相信很多人都用过，它主要这几种方法来实现：

* $.on: 基本用法:

$('.parent').on('click', 'a', function () {   
 console.log('click event on tag a');   
})

它是 .parent 元素之下的 a 元素的事件代理到 $('.parent') 之上，只要在这个元素上有点击事件，就会自动寻找到 .parent 元素下的 a 元素，然后响应事件

* $.delegate: 基本用法:

$('.parent').delegate('a', 'click', function () {   
 console.log('click event on tag a');   
})

* 同上，并且还有相对应的 $.delegate 来删除代理的事件；
* $.live: 基本使用方法:
* $('a', $('.parent')).live('click', function () {  
   console.log('click event on tag a');   
  })
* 同上，然而如果没有传入父层元素 $(.parent)，那事件会默认委托到 $(document) 上；(已废除)

## 4.17 promise+++

### 4.17.1 什么是Promise

Promise是一种封装未来值的易于复用的异步任务管理机制。

* 等待（pending）：初始状态；
* 已完成（fulfilled）：操作成功完成；
* 被拒绝（rejected）：操作失败；

这三个状态变更关系需满足以下三个条件：

* 处于等待（pending）状态时，可以转变为已完成（fulfilled）或者被拒绝状态（rejected）；
* 处于已完成状态时，状态不可变，且需要有一个最终值；
* 处于被拒绝状态时，状态不可变，且需要有一个拒绝原因。

### 4.17.2 then方法

一个promise必须提供一个then方法，以供访问其当前状态，或最终值或拒绝原因。

**参数**

该方法接收两个参数，如promise.then(onFulfilled, onRejected):

* 两个参数均为可选，均有默认值，若不传入，则会使用默认值；
* 两个参数必须是函数，否则会被忽略，使用默认函数；
* onFulfilled: 在promise已完成后调用且仅调用一次该方法，该方法接受promise最终值作参数；
* onRejected: 在promise被拒绝后调用且仅调用一次该方法，该方法接受promise拒绝原因作参数；
* 两个函数都是异步事件的回调，符合JavaScript事件循环处理流程

**返回值**

该方法必须返回一个promise

### 4.17.3 实例

var promise = new Promise((resolve, reject) => {  
 setTimeout(function() {  
 resolve('完成');  
 }, 10);  
 });  
 promise.then((msg) => {  
 console.log('first messaeg: ' + msg);  
 })  
 promise.then((msg) => {  
 console.log('second messaeg: ' + msg);  
 });

### 4.17.4 Promise.all(iterable)++

all方法接受一个或多个promsie（以数组方式传递），返回一个新promise，该promise状态取决于传入的参数中的所有promsie的状态：

1. 当所有promise都完成是，返回的promise完成，其最终值为由所有完成promsie的最终值组成的数组；
2. 当某一promise被拒绝时，则返回的promise被拒绝，其拒绝原因为第一个被拒绝promise的拒绝原因；

var p1 = new Promise((resolve, reject) => {  
 setTimeout(function(){  
 console.log('p1决议');  
 resolve('p1');  
 }, 10);  
});  
var p2 = new Promise((resolve, reject) => {  
 setTimeout(function(){  
 console.log('p2决议');  
 resolve('p2');  
 }, 10);  
});  
Promise.all( [p1, p2] )  
 .then((msgs) => {  
 // p1和p2完成并传入最终值  
 console.log(JSON.stringify(msgs));  
})  
 .then((msg) => {  
 console.log( msg );  
});  
  
/\*output:  
p1决议  
p2决议  
["p1","p2"]  
undefined  
\*/

#### 实现promise.all

Promise.all = function(promises) {  
 return new Promise(function(resolve, reject) {  
 var resolvedCounter = 0;  
 var promiseNum = promises.length;  
 var resolvedValues = new Array(promiseNum);  
 for (let i=0;i<promisesLength;i++) {  
 Promise.resolve(promises[i]).then(function(value) {  
 promiseCount++;  
 results[i] = value;  
 // 当所有函数都正确执行了，resolve输出所有返回结果。  
 if (promiseCount === promisesLength) {  
 return resolve(resolvedValues);  
 }  
 }, function(err) {  
 return reject(err);  
 });  
 }  
 });  
};  
  
Promise.race = function(promises){  
 return new Promise(function(resolve, reject) {  
 for (let i = 0; i < promises.length; i++) {  
 Promise.resolve(promises[i]).then(function(value) {  
 return resolve(value)  
 }, function(reason) {  
 return reject(reason)  
 })  
 }  
 })  
}

### 4.17.5 Promise.race(iterable)

race方法返回一个promise,只要传入的诸多promise中的某一个完成或被拒绝，则该promise同样完成或被拒绝，最终值或拒绝原因也与之相同。

## 4.18 window.onload和document.ready的区别？哪一个先执行？+

document.ready方法在DOM树加载完成后就会执行，

而window.onload是在页面资源（比如图片和媒体资源，它们的加载速度远慢于DOM的加载速度）加载完成之后才执行。

也就是说$(document).ready要比window.onload先执行。

## 4.19 var、let和const有什么区别+

### 4.19.1 区别

1. var与let是可以声明变量，const不能声明变量，只能声明只读的常量。
2. var声明的变量不存在块级作用域，他在全局内有效。let与const的声明只在其所在的代码块中有效。
3. let/const不能在同一个作用域中声明相同变量/常量，var可以多次重复声明。
4. var存在[变量提升](https://www.jianshu.com/p/76ab117d63b6),所以var能先使用在声明，但是let const必须先声明再使用。
5. let/const存在暂时性死区。

* var a = 100;  
  if(1){  
   a = 10;  
   // 在当前块作用域中存在a使用let/const声明的情况下，给a赋值10时，只会在当前作用域找变量a，  
   // 而这时，还未到声明时候，所以控制台Error:a is not defined  
   let a = 1;  
  }

1. const声明时必须初始化赋值，一旦声明，其声明赋值的值就不允许改变，更不可以重复声明。const 声明了一个复合类型的常量，其存储的是一个引用地址，不允许改变的是这个地址，而对象本身是可变的。

* const a = 100;   
  const list = [];  
  list[0] = 10;  
  console.log(list);　　  
  // [10]   
  const obj = {a:100};  
  obj.name = 'apple';  
  obj.a = 10000;  
  console.log(obj);　　  
  // {a:10000,name:'apple'}

### 4.19.2 原理

JavaScript中的函数运行在它们被定义的作用域里，而不是它们被执行的作用域里。

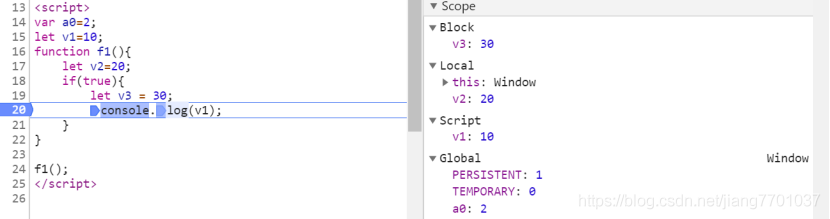
**var**  
会直接在栈内存里预分配内存空间，然后等到实际语句执行的时候，再存储对应的变量，如果传的是引用类型，那么会在堆内存里开辟一个内存空间存储实际内容，栈内存会存储一个指向堆内存的指针。

**let**  
是不会在栈内存里预分配内存空间，而且在栈内存分配变量时，做一个检查，如果已经有相同变量名存在就会报错。

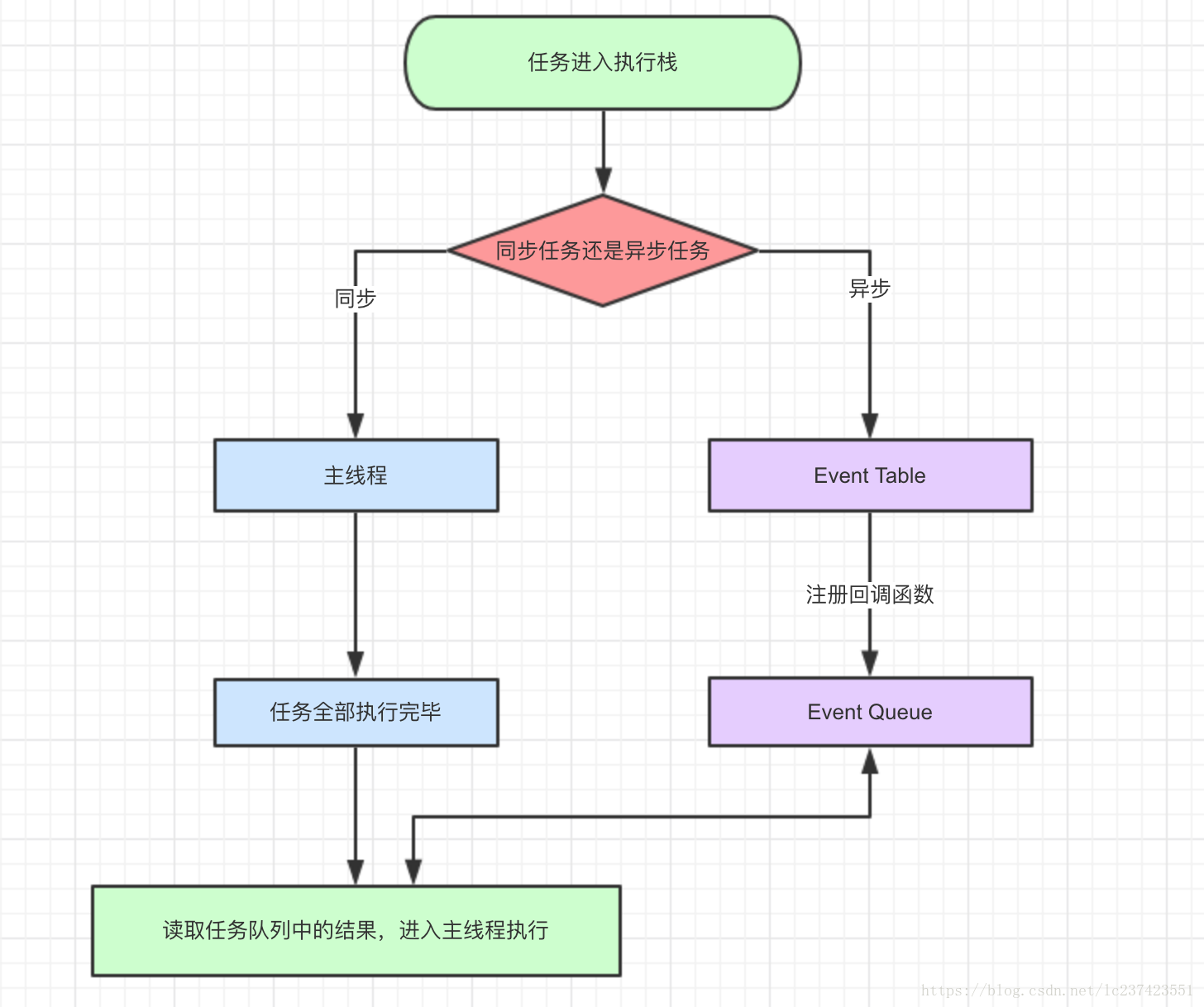
**const**  
也不会预分配内存空间，在栈内存分配变量时也会做同样的检查。不过const存储的变量是不可修改的，对于基本类型来说你无法修改定义的值，对于引用类型来说你无法修改栈内存里分配的指针，但是你可以修改指针指向的对象里面的属性。

### 全局作用域下 let和const不是window对象的属性

在全局作用域中，var声明的变量是在window对象下的，而let和const则是在script语句块的作用域中。



## 4.20 JavaScript事件执行机制？宏任务、微任务+



js事件处理机制

js会将同步和异步任务分别放入不同的执行"场所"，同步的进入主线程(先执行，同步任务实质上是一个宏任务)，异步任务进入Event Table并注册函数。当指定的事情完成时，Event Table会将这个函数移入Event Queue。（指定的事情比如setTimeout的定义的时间完成时）主线程内的任务执行完毕为空，会去Event Queue读取对应的函数，进入主线程执行。上述过程会不断重复，也就是常说的Event Loop(事件循环)。js异步有一个机制，就是遇到宏任务，先执行宏任务，将宏任务放入eventqueue，然后再执行微任务，将微任务放入eventqueue。**值得注意，这两个queue不是一个queue**。当你往外拿的时候先从微任务里拿这个回调函数，然后再从宏任务的queue上拿宏任务的回调函数。异步事件执行的最快时间是4ms。

宏任务 => 微任务的Event Queue => 宏任务的Event Queue

**宏任务和微任务的分类**

macro task（宏任务）: 同步代码(整块script代码)、setImmediate、MessageChannel、setTimeout/setInterval

micro task（微任务）: Promise.then（不是promise，promise会立即执行）、MutationObserver，还有process.nextTick等。

async function async1() {   
 console.log('async1 start'); //4.同步执行，输出  
 await async2(); //5.同步执行，进入函数async2  
 console.log('async1 end');//7.异步，加入事件队列 10.进入事件队列执行，输出  
}   
  
async function async2() {   
 console.log('async2'); //6.同步执行，输出  
}   
// 0.开始同步执行  
console.log('script start'); //1.同步执行，输出  
  
setTimeout(function() {   
 console.log('setTimeout'); //2.当定时器的时间到了之后，加入事件队列，若事件队列已有很多任务，则放置于其之后 12.计时结束，输出  
}, 0);   
  
async1(); //3.执行函数，进入  
  
new Promise(function(resolve) {   
 console.log('promise1'); //7.同步执行，输出  
 resolve();   
}).then(function() {   
 console.log('promise2'); //8.加入异步队列 11.进入事件队列执行，输出  
});   
  
console.log('script end'); //9.同步执行，输出  
  
//执行顺序：  
/\*script start  
async1 start  
async2  
promise1  
script end  
async1 end  
promise2  
setTimeout  
\*/



## 4.21 == 与 === 的区别 +

== 在比较时等号左右两侧数据类型不同时会先转成相同数据类型，再比较。

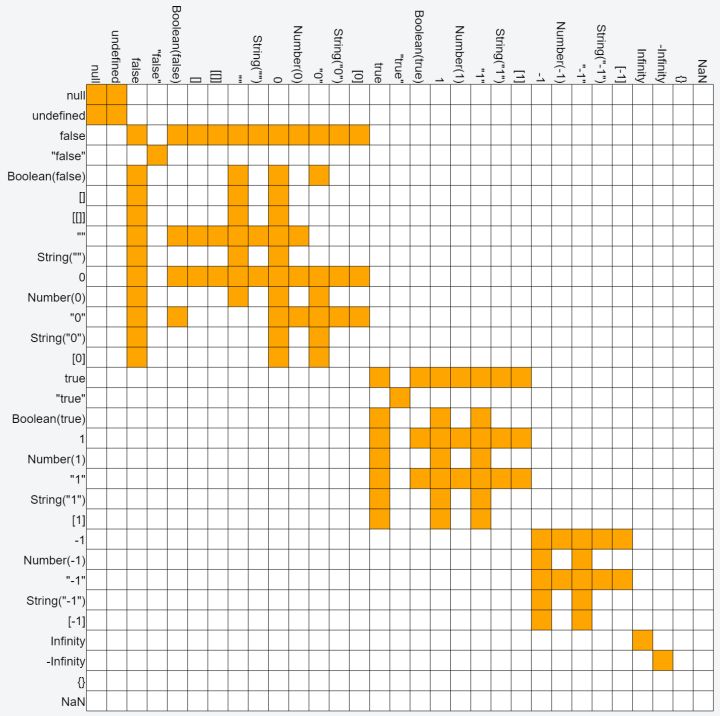
== 是相对比较； === 是绝对比较；

**==比较**

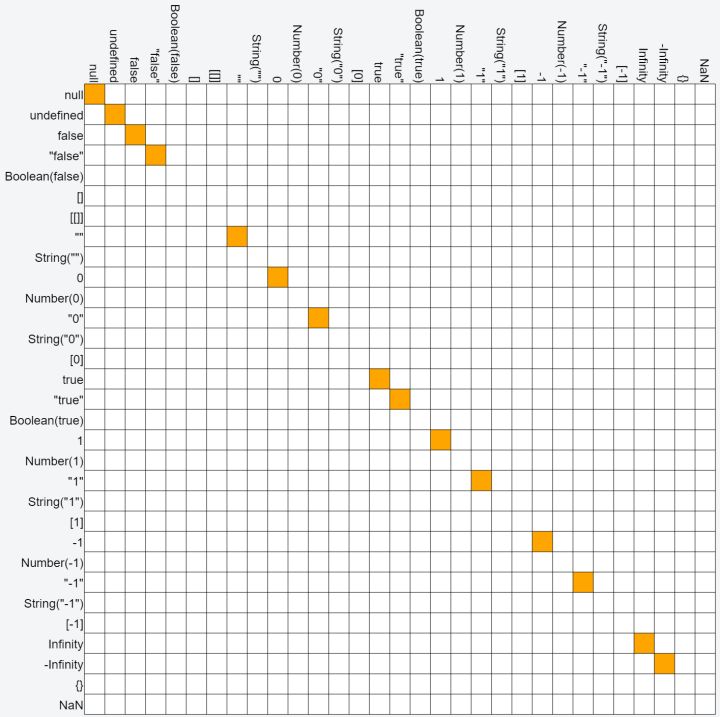
比较方法：

1. 字符串 == 数字 ；字符串转换成数字
2. 布尔值 == 数字; 布尔值转成数字
3. 布尔值 == 字符串; 布尔值转数字，字符串也转成数字，然后进行比较；
4. null == undefined // true, null 和undefined和其他数据类型比较都是false
5. 对象 == 对象; 比较的是空间地址，地址相同返回true
6. 对象 == 字符串; 对象转成字符串，然后和字符串比较
7. 对象 == 布尔值；对象先转成字符串，再转数字，布尔值也转成数字，在比较这两个数字
8. 对象 == 数字；对象先转成字符串，然后再转成数字
9. NaN和NaN永远不相等

{} == '[object Object]' //true  
[] == false //true  
{} == true //false  
{} == 1 //false  
[] == 0 //true  
NaN == NaN //false

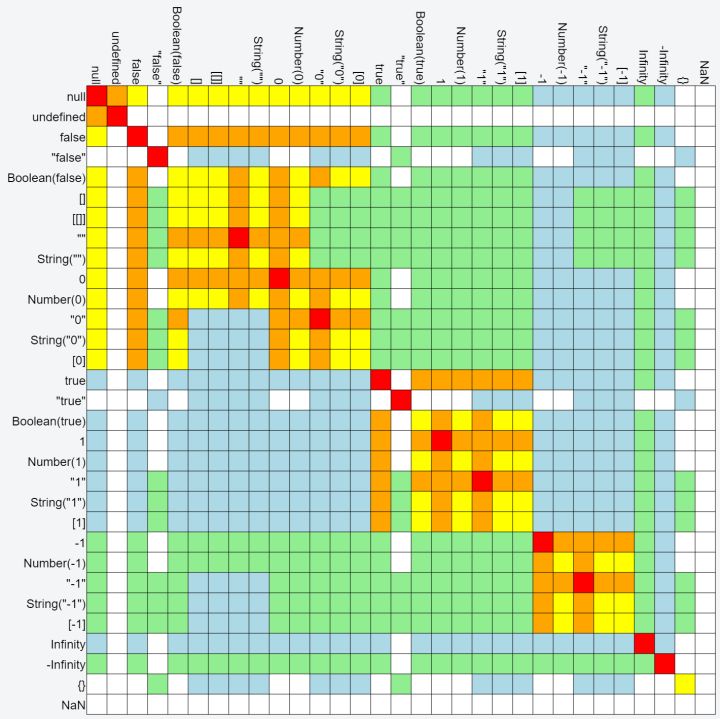


**===比较**



**完整比较图**

* 红色：===
* 橙色：==
* 黄色：<= 和 >= 同时成立，== 不成立
* 蓝色：只有 >=
* 绿色：只有 <=



## 4.22 数组操作函数有哪些？哪些是es6新增的？+

### 4.22.1 数组遍历

for：

普通版：for(let i=0; i<arr.length; i++) { }

优化版：for(let i=0,len=arr.length; i<len; i++) { }

弱化版：for(let i=0; arr[i] != null; i++) { }

forEach： arr.forEach((e, index) => { }) 性能比for要弱

变种：Array.prototype.forEach.call(arr, (el, index) => { }) 实际性能要比普通的foreach弱

for...in... ：for(let i in arr) { } 效率最低

for...of...：for(let i of arr) { } 性能好于forin

### 4.22.2 转换方法

toString: 将 **Array** 的元素转换为字符串。结果字符串由逗号分隔，且连接起来。

toLocaleString: 转为本地化语言字符串

valueOf: 数组的元素被转换为字符串，这些字符串由逗号分隔，连接在一起。其操作与 **Array.toString** 和 **Array.join**方法相同。

### 4.22.3 栈方法

push()：向数组的末尾添加一个或多个元素，并返回新的长度

pop()：删除数组的最后一个元素，数组的长度减1，并且返回删除元素的值(数组为空时，不改变数组，并返回undefined)

### 4.22.4 队列方法

unshift()：向数组的头部添加一个或多个元素，返回新的长度

shift()：删除并返回数组的第一个元素

### 4.22.5 重排序方法

reverse()：颠倒数组中的元素顺序

sort()：对数组元素进行排序（默认按照ASCII）

let arr = [0,1,5,10,15];  
arr.sort();//0,1,10,15,5  
arr.sort((a,b)=>a>b);// 0,1,5,10,15

### 4.22.6 操作方法

concat():链接两个或多个数组

slice(start, end)：从某个已有的数组返回选定的元素(没有end参数时就到数组结尾)

splice(start,count,...insertedValues)：删除元素，并向数组添加新元素

### 4.22.7 位置方法

indexOf(x, start)、lastIndexOf()：查找数组元素的位置，参数：要查的项和起点位置索引

### 4.22.8 迭代方法

map()：arr.map((n) => { }) 实际效率还比不上forEach

every()：数组的每一项都满足给定条件则返回true

some()：数组中只要有一项满足条件就返回true

filter()：返回满足给定条件的数据项组成的新数组，不改变原数组

forEach(): 对数组中的每一项运行给定函数。无返回值。

### 4.22.9 归并方法

reduce()、reduceRight()：迭代所有的项，构建一个最终的返回值。

### 4.22.10 ES6新增数组操作

1. **Array.from()** 把类数组(获取一组元素、arguments…) 对象转成数组

1、字符串转数组(个人感觉有些类似于arr.split(""))：  
let str = "hello";  
console.log(Array.from(str)); //Array(5) ["h", "e", "l", "l", "o"]  
  
2、将类数组对象转换为真正数组：  
let arrList = {  
 0: 'yang',   
 1: '27',  
 2: '男',  
 3: ['wang','ling','huang'],  
 'length': 4  
}  
let arr = Array.from(arrList)  
console.log(arr) // Array(4) ['yang','27','男',['wang','ling','huang']]  
  
PS：对于第二种，如果将最后的length去掉，那将是一个length为0的空数组  
  
如果将代码再改一下：  
let arrList = {  
 'name': 'yang',   
 'age': '27',  
 'sex': '男',  
 'fre': ['wang','ling','huang'],  
 'length': 4  
}  
let arr = Array.from(arrList)  
console.log(arr) // Array(4) [undefined, undefined, undefined, undefined]  
会发现结果是长度为4，元素均为undefined的数组  
  
由此可见，要将一个类数组对象转换为一个真正的数组，必须具备以下条件：  
1、该类数组对象必须具有length属性，用于指定数组的长度。如果没有length属性，那么转换后的数组是一个空数组  
2、该类数组对象的属性名必须为数值型或字符串型的数字

1. **Array.of()** 把一组值,转化数组

let arr = Array.of('apple','banana','orange');

1. **arr.find()** 查找数组内元素，找到第一个符合条件的数组成员，返回该成员的值，如果没有找到，返回undefined
2. **arr.findIndex()** 查找数组内元素，找到第一个符合条件的数组成员，返回该成员的下标（index）， 没找到返回-1
3. **arr.fill()** 填充

array.fill(value, start, end)  
参数 描述  
value 必需。填充的值。  
start 可选。开始填充位置。  
end 可选。停止填充位置 (默认为 Array.length)

1. **arr.copyWithin** 使用数组内部元素替换指定位置元素

//这个东西用的不多，但是很有意思  
//首先，参数有3个  
//1: 被替换的起始位置 2: 选取替换值的起始位置 3: 选取替换值得结束位置  
let arr = [1, 'c', 'd', 'a', 'b']; //假如我想把a，b替换到1的位置  
arr.copyWithin(0, 3, 5) // ["a", "b", "d", "a", "b"]

1. **entries/keys/values**

let arr=['a', 'b', 'c']  
for(let key of arr.keys()){} //取键  
for(let value of arr.values()){} //取值；不能直接使用，有兼容性问题，甚至谷歌  
for(let [key, value] of arr.entries()){} //都取

1. ES2016新增

arr.indexOf() // ES5 查找指定元素是否存在，如果存在，返回下标，如果不存在返回-1  
arr.includes() // 查找指定元素是否存在，如果存在，返回true，如果不存在返回false

## 4.23 ES6的变化

### 4.23.1 定义变量和常量

在es5中我们经常使用var来声明一个变量，es6更改let声明

1. **不能重复声明**变量
2. 不会预处理, **不存在变量提升**
3. 用let声明的变量 如果在块内，会生成 **块作用域**

### 4.23.2 模板字符串

* 模板字符串必须用 `` 包含
* 变化的部分使用${xxx}定义
* 作用:简化字符串pim

//使用的场景  
 //比如:网址字符串的的请求  
  
let url = `http://ezcarry.com?userName=${obj.userName}&psd=${obj.psd}`;

### 4.23.3 解构赋值

### 数组解构赋值

1. 在数组的解构赋值中, 等号左边的格式必须和等号右边的格式一模一样, 才能完全解构

* let [a, b, c] = [1, 3, 5]; // 结果 a=1.b=3,c=5

1. 左边的个数可以和右边数组的个数不一样;就按照下标索引 对应赋值,如果没有对应上 就是 undefined;b,c给定的默认值，其实里面有判断处理，如果对应索引有值，则赋值，如果没有 使用默认值

* let [a, b] = [1, 3, 5]; //a=1,b=3  
  let [a, b, c] = [1]; //a=1,b=undefined,c=undefined  
  let [a, b = 666, c = 888] = [1]; //a=1,b=666,c=888

1. 还可以使用ES6中新增的扩展运算符来打包剩余的数据, 那么**扩展运算符只能写在最后**

* let [a, ...b] = [1, 3, 5]; //a=1,b=[3,5]

#### 对象解构赋值

对象解构赋值是比较严格的，**要和对象的属性值相同**。

let {name,age} = {name:"cc",age:18}; //name="cc",age=18

**允许key有剩余**，就是说声明的变量的个数，不一定要和对象里面key的个数相同**，但是一旦声明**必须和key的名字形同

let {name} = {name:"cc",age:18};

### 4.23.4 扩展运算符

let [a, ...b] = [1, 3, 5]; // a = 1; b = [3, 5];  
  
let arr1 = [1, 3, 5];  
let arr2 = [2, 4, 6];  
let arr = [...arr1, ...arr2]; // arr= [1, 3, 5, 2, 4, 6];

### 4.23.5 箭头函数及函数参数

let test = ()=> {  
 console.log("测试");  
}

#### ES6中函数的扩展运算符

* 扩展运算符在函数的形参列表中的作用，将传递给函数的所有实参打包到一个**数组**中

function test(...value) {//es6  
 console.log(value);  
}  
  
function test() { //es5  
 //arguments 伪数组 只可以 for遍历  
 console.log(arguments);  
}  
  
//value 是一个真实的数组，具备数组的所有属性和方法  
test(1,2,3,4,5);

### 4.23.6 类和对象

* ES6系统提供了一个class关键字 专门定义类
* static 来定义**静态方法**,**不可以**定义**静态属性**

class Student{  
 // 通过new创建的对象的时候，会自动调用constructor  
 constructor(name,age){  
 //属性  
 this.name = name;  
 this.age = age;  
 }  
 //实例方法  
 say = function () {  
 console.log(`你好，我是：${this.name},今年${this.age}`);  
 }  
  
//static 使用在 es6中 规定是 只可以定义静态方法，不可以定义静态属性，有的浏览器不支持  
//静态属性 要用 static 声明  
// static score = 66;  
// 定义静态属性  
  
 //静态方法  
 static run = function () {  
 console.log("走呀，一块跑步")  
 }  
 // 注意 静态方法可以简写  
 // static eat = function () {}  
 static eat(){  
 console.log("吃饭");  
 }  
}  
  
//定义静态属性  
Student.score = 66;

## 4.24 async和await

async用于声明异步方法，await是async wait的缩写，用于等待一个异步方法执行完成。await只能出现在async函数中。async函数返回的是一个Promise对象。

**获取失败的结果**：把await和成功后的操作放到try里，失败的放在catch

function touzi(guess){  
 return new Promise((resolve, reject)=>{  
 let sino = parseInt(Math.random() \* 6 +1)  
 if(sino > 3){  
 if(guess === '大'){  
 resolve(sino)  
 }else{  
 reject(sino)  
 }  
 }else{  
 if(guess === '大'){  
 reject(sino)  
 }else{  
 resolve(sino)  
 }  
 }  
 setTimeout(()=>{  
 resolve(sino)  
 },300)  
 })  
}  
async function test(){  
 try{  
 //把await及获取它的值的操作放在try里  
 let n =await touzi('大')  
 console.log('赢了' + n)  
 }catch(error){  
 //失败的操作放在catch里  
 console.log('输了' + error)  
 }  
}  
test()

使用await使异步代码更像同步代码

await还可以获取多个promise的结果

async function test(){  
 try{  
 let n = await Promise.all([touzi('大'),touzi('大')])  
 console.log(n)  
 }catch(error){  
 console.log(error)  
 }  
}  
test()

async函数会返回一个promise，并且Promise对象的状态值是resolved

1. 如果你没有在async函数中写return，那么Promise对象resolve的值就是是undefined
2. 如果你写了return，那么return的值就会作为你成功的时候传入的值

### 4.24.1 await等到之后做了什么事

#### 不是promise对象

如果不是 promise , await会阻塞后面的代码，先执行async外面的同步代码，同步代码执行完，再回到async内部，把这个非promise的东西，作为 await表达式的结果。

#### 是promise对象

如果它等到的是一个 promise 对象，await 也会暂停async后面的代码，先执行async外面的同步代码，等着 Promise 对象 fulfilled，然后把 resolve 的参数作为 await 表达式的运算结果。

1. async是同步的，那么该函数调用会被同步执行

* async function fn(){  
   console.log('a')  
  }  
  fn()  
  console.log('b')  
    
  /\*输出：  
  a  
  b  
  \*/

1. 如果在await后面接的这个promsie都是同步的，后面的promise会同步执行，但是拿到这个值还是得等待（特别注意：如果promise没有一个成功的值传入，对await来说就算是失败了，下面的代码就不会执行），所以不管await后面的代码是同步还是异步，await总是需要时间，从右向左执行，先执行右侧的代码，执行完后，发现有await关键字，于是让出线程，阻塞代码

* function fn(){  
   return new Promise(resolve=>{  
   console.log(1)  
   })  
  }  
  async function f1(){  
   await fn()  
   console.log(2) //因为fn是属于同步的，所以先打印出1，然后是3，但是因为没有resolve结果，所以await拿不到值，因此不会打印2  
  }  
  f1()  
  console.log(3)  
  //1  
  //3
* function fn(){  
   return new Promise(resolve=>{  
   console.log(1)  
   resolve()  
   })  
  }  
  async function f1(){  
   await fn()  
   console.log(2) //多了个resolve()说明promise成功了，所以await能拿到结果，因此就是1 3 2  
  }  
  f1()  
  console.log(3)  
  //1  
  //3  
  //2

### 4.24.2 async原理

async函数是Generator函数的语法糖。即使用Generator函数+自动执行器来运作的。

async function fn(args){  
 // ...  
}  
  
// 等同于  
  
function fn(args){   
 return spawn(function\*() {  
 // ...  
 });   
}  
//spawn 函数就是自动执行器。

function spawn(generatorFunc) {  
 // 返回的是一个新的函数  
 return function() {  
   
 // 先调用generator函数 生成迭代器  
 // 对应 var gen = testG()  
 const gen = generatorFunc.apply(this, arguments)  
   
 // 返回一个promise 因为外部是用.then的方式 或者await的方式去使用这个函数的返回值的  
 // var test = asyncToGenerator(testG)  
 // test().then(res => console.log(res))  
 return new Promise((resolve, reject) => {  
   
 // 内部定义一个step函数 用来一步一步的跨过yield的阻碍  
 // key有next和throw两种取值，分别对应了gen的next和throw方法  
 // arg参数则是用来把promise resolve出来的值交给下一个yield  
 function step(key, arg) {  
 let generatorResult  
   
 // 这个方法需要包裹在try catch中  
 // 如果报错了 就把promise给reject掉 外部通过.catch可以获取到错误  
 try {  
 generatorResult = gen[key](arg)  
 } catch (error) {  
 return reject(error)  
 }  
   
 // gen.next() 得到的结果是一个 { value, done } 的结构  
 const { value, done } = generatorResult  
   
 if (done) {  
 // 如果已经完成了 就直接resolve这个promise  
 // 这个done是在最后一次调用next后才会为true  
 // 以本文的例子来说 此时的结果是 { done: true, value: 'success' }  
 // 这个value也就是generator函数最后的返回值  
 return resolve(value)  
 } else {  
 // 除了最后结束的时候外，每次调用gen.next()  
 // 其实是返回 { value: Promise, done: false } 的结构，  
 // 这里要注意的是Promise.resolve可以接受一个promise为参数  
 // 并且这个promise参数被resolve的时候，这个then才会被调用  
 return Promise.resolve(  
 // 这个value对应的是yield后面的promise  
 value  
 ).then(  
 // value这个promise被resolve的时候，就会执行next  
 // 并且只要done不是true的时候 就会递归的往下解开promise  
 // 对应gen.next().value.then(value => {  
 // gen.next(value).value.then(value2 => {  
 // gen.next()  
 //  
 // // 此时done为true了 整个promise被resolve了  
 // // 最外部的test().then(res => console.log(res))的then就开始执行了  
 // })  
 // })  
 function onResolve(val) {  
 step("next", val)  
 },  
 // 如果promise被reject了 就再次进入step函数  
 // 不同的是，这次的try catch中调用的是gen.throw(err)  
 // 那么自然就被catch到 然后把promise给reject掉啦  
 function onReject(err) {  
 step("throw", err)  
 },  
 )  
 }  
 }  
 step("next")  
 })  
 }  
}

## 4.25 内存管理机制

内存泄露可以定义为：应用程序不再需要占用内存的时候，由于某些原因，内存没有被操作系统或可用内存池回收。编程语言管理内存的方式各不相同。

### 4.25.1 JavaScript 内存管理

**JavaScript 是一种垃圾回收语言**。垃圾回收语言通过周期性地检查先前分配的内存是否可达，帮助开发者管理内存。换言之，垃圾回收语言减轻了“内存仍可用”及“内存仍可达”的问题。两者的区别是微妙而重要的：仅有开发者了解哪些内存在将来仍会使用，而不可达内存通过算法确定和标记，适时被操作系统回收。

**JS的回收机制分两种：1.标记清除 2.引用计数。各大浏览器常用的是前者。**

#### 什么时候触发垃圾回收

##### 1. IE

垃圾回收器周期性运行，如果分配的内存非常多，那么回收工作也会很艰巨，确定垃圾回收时间间隔就变成了一个值得思考的问题。IE6的垃圾回收是根据内存分配量运行的，当环境中存在256个变量、4096个对象、64k的字符串任意一种情况的时候就会触发垃圾回收器工作，看起来很科学，不用按一段时间就调用一次，有时候会没必要，这样按需调用不是很好吗？但是如果环境中就是有这么多变量等一直存在，现在脚本如此复杂，很正常，那么结果就是垃圾回收器一直在工作，这样浏览器就无法工作了。

微软在 IE7 中做了调整，触发条件不再是固定的，而是动态修改的，初始值和 IE6 相同，如果垃圾回收器回收的内存分配量低于程序占用内存的15%，说明大部分内存不可被回收，设的垃圾回收触发条件过于敏感，这时候把临街条件翻倍，如果回收的内存高于 85%，说明大部分内存早就该清理了，这时候把触发条件置回。这样就使垃圾回收工作职能了很多。

##### 2. Chrome-V8

自动垃圾回收算法的演变过程中出现了很多算法，但是由于不同对象的生存周期不同，没有一种算法适用于所有的情况。所以V8采用了一种分代回收的策 略，将内存分为两个生代：新生代和老生代。新生代的对象为存活时间较短的对象，老生代中的对象为存活时间较长或常驻内存的对象。分别对新生代和老生代使用 不同的垃圾回收算法来提升垃圾回收的效率。对象起初都会被分配到新生代，当新生代中的对象满足某些条件时，会被移动到老生代

**V8的分代内存**

默认情况下，64位环境下的V8引擎的新生代内存大小32MB、老生代内存大小为1400MB，而32位则减半，分别为16MB和700MB。V8内存的最大保留空间分别为1464MB（64位）和732MB（32位）。具体的计算公式是4\*reserved\_semispace\_space\_ + max\_old\_generation\_size\_，新生代由两块reserved\_semispace\_space\_组成，每块16MB（64位）或8MB（32位）

**新生代**

大多数的对象被分配在这里，这个区域很小但是垃圾回特别频繁。在新生代分配内存非常容易，我们只需要保存一个指向内存区的指针，不断根据新对象的大小进行递增即可。**当该指针到达了新生代内存区的末尾，就会有一次清理（仅仅是清理新生代）**。使用Scavenge算法进行回收。在Scavenge算法的实现中，主要采用了Cheney算法。

**老生代**

老生代占用内存较多（64位为1.4GB，32位为700MB），如果使用Scavenge算法，浪费一半空间不说，复制如此大块的内存消耗时间将会相当长。所以Scavenge算法显然不适合。**V8在老生代中的垃圾回收策略采用Mark-Sweep和Mark-Compact相结合**

### 4.25.2 合理的GC方案

#### 基础方案

Javascript 引擎基础GC方案是（simple GC）：mark and sweep（标记清除），即：

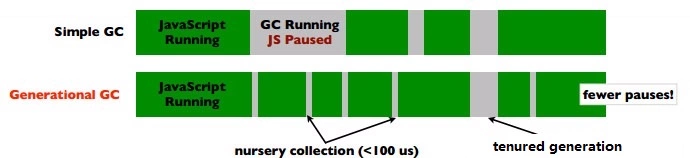
1. 遍历所有可访问的对象。
2. 回收已不可访问的对象。

#### GC的缺陷

和其他语言一样，JS 的 GC 策略也无法避免一个问题：GC时，停止响应其他操作，这是为了安全考虑。而 Javascript 的GC在100ms甚至以上，对一般的应用还好，但对于 JS 游戏，动画对连贯性要求比较高的应用，就麻烦了。这就是新引擎需要优化的点：避免GC造成的长时间停止响应。

#### GC优化策略

1. **分代回收**（Generation GC）  
   这个和Java回收策略思想是一致的，也是V8所主要采用的。目的是通过区分“临时”与“持久”对象；多回收“临时对象”区（young generation），少回收“持久对象”区（tenured generation），减少每次需遍历的对象，从而减少每次GC的耗时。如图：

* 
* 这里需要补充的是：对于tenured generation对象，有额外的开销：把它从young generation迁移到tenured generation，另外，如果被引用了，那引用的指向也需要修改。

1. **增量GC**

* 这个方案的思想很简单，就是“每次处理一点，下次再处理一点，如此类推”。如图：
* 
* 这种方案，虽然耗时短，但中断较多，带来了上下文切换频繁的问题。因为每种方案都其适用场景和缺点，因此在实际应用中，会根据实际情况选择方案。比如：低 (对象/s) 比率时，中断执行GC的频率，simple GC更低些；如果大量对象都是长期“存活”，则分代处理优势也不大。

### 4.25.3 Mark-and-sweep

**大部分垃圾回收语言用的算法称之为 Mark-and-sweep 。**算法由以下几步组成：

1. 垃圾回收器创建了一个“roots”列表。Roots 通常是代码中全局变量的引用。JavaScript 中，“window” 对象是一个全局变量，被当作 root 。window 对象总是存在，因此垃圾回收器可以检查它和它的所有子对象是否存在（即不是垃圾）；
2. 所有的 roots 被检查和标记为激活（即不是垃圾）。所有的子对象也被递归地检查。从 root 开始的所有对象如果是可达的，它就不被当作垃圾。
3. 所有未被标记的内存会被当做垃圾，收集器现在可以释放内存，归还给操作系统了。

现代的垃圾回收器改良了算法，但是本质是相同的：可达内存被标记，其余的被当作垃圾回收。

不需要的引用是指开发者明知内存引用不再需要，却由于某些原因，它仍被留在激活的 root 树中。在 JavaScript 中，不需要的引用是保留在代码中的变量，它不再需要，却指向一块本该被释放的内存。有些人认为这是开发者的错误。

为了理解 JavaScript 中最常见的内存泄露，我们需要了解哪种方式的引用容易被遗忘。

### 4.25.4 常见的JavaScript 内存泄漏+

#### 意外的全局变量

JavaScript 处理未定义变量的方式比较宽松：未定义的变量会在全局对象创建一个新变量。在浏览器中，全局对象是 window 。

function foo(arg) {   
 bar = "this is a hidden global variable";   
 // 实际： window.bar = "this is an explicit global variable";  
}

**另一种意外的全局变量可能由 this 创建：**

function foo() {   
 this.variable = "potential accidental global";   
}   
// Foo 调用自己，this 指向了全局对象（window）   
// 而不是 undefined   
foo();

在 JavaScript 文件头部加上 'use strict'，可以避免此类错误发生。启用严格模式解析 JavaScript ，避免意外的全局变量。

**全局变量注意事项**

尽管我们讨论了一些意外的全局变量，但是仍有一些明确的全局变量产生的垃圾。它们被定义为不可回收（除非定义为空或重新分配）。尤其当全局变量用于临时存储和处理大量信息时，需要多加小心。如果必须使用全局变量存储大量数据时，确保用完以后把它设置为null或者重新定义。与全局变量相关的增加内存消耗的一个主因是缓存。缓存数据是为了重用，缓存必须有一个大小上限才有用。高内存消耗导致缓存突破上限，因为缓存内容无法被回收。

#### 被遗忘的计时器或回调函数

在 JavaScript 中使用 setInterval 非常平常。一段常见的代码：

var someResource = getData();   
setInterval(function() {   
 var node = document.getElementById('Node');   
 if(node) {   
 // 处理 node 和 someResource   
 node.innerHTML = JSON.stringify(someResource));   
 }   
}, 1000);

此例说明了什么：与节点或数据关联的计时器不再需要，node 对象可以删除，整个回调函数也不需要了。可是，计时器回调函数仍然没被回收（计时器停止才会被回收）。同时，someResource 如果存储了大量的数据，也是无法被回收的。

对于观察者（事件监听函数）的例子，一旦它们不再需要（或者关联的对象变成不可达），明确地移除它们非常重要。老的 IE 6 是无法处理循环引用的。如今，即使没有明确移除它们，一旦观察者对象变成不可达，大部分浏览器是可以回收观察者处理函数的。

element.addEventListener('click', onClick);

**对象观察者和循环引用注意事项**

老版本的 IE 是无法检测 DOM 节点与 JavaScript 代码之间的循环引用，会导致内存泄露。如今，现代的浏览器（包括 IE 和 Microsoft Edge）使用了更先进的垃圾回收算法，已经可以正确检测和处理循环引用了。换言之，回收节点内存时，不必非要调用 removeEventListener 了。

#### 脱离DOM的引用

有时，保存 DOM 节点内部数据结构很有用。假如你想快速更新表格的几行内容，把每一行 DOM 存成字典（JSON 键值对）或者数组很有意义。此时，同样的 DOM 元素存在两个引用：一个在 DOM 树中，另一个在字典中。将来你决定删除这些行时，需要把两个引用都清除。

var elements = {   
 button: document.getElementById('button'),   
 image: document.getElementById('image'),   
 text: document.getElementById('text')   
};   
   
function doStuff() {   
 image.src = 'http://some.url/image';   
 button.click();   
 console.log(text.innerHTML);   
 // 更多逻辑   
}   
   
function removeButton() {   
 // 按钮是 body 的后代元素   
 document.body.removeChild(document.getElementById('button'));   
   
 // 此时，仍旧存在一个全局的 #button 的引用   
 // elements 字典。button 元素仍旧在内存中，不能被 GC 回收。   
}

此外还要考虑 DOM 树内部或子节点的引用问题。假如你的 JavaScript 代码中保存了表格某一个 <td> 的引用。将来决定删除整个表格的时候，直觉认为 GC 会回收除了已保存的 <td> 以外的其它节点。实际情况并非如此：此<td> 是表格的子节点，子元素与父元素是引用关系。由于代码保留了 <td> 的引用，导致整个表格仍待在内存中。保存 DOM 元素引用的时候，要小心谨慎。

#### 闭包

var theThing = null;   
var replaceThing = function () {   
 var originalThing = theThing;   
 var unused = function () {   
 if (originalThing)   
 console.log("hi");   
 };   
   
 theThing = {   
 longStr: new Array(1000000).join('\*'),   
 someMethod: function () {   
 console.log(someMessage);   
 }   
 };   
};   
   
setInterval(replaceThing, 1000);

每次调用 replaceThing ，theThing 得到一个包含一个大数组和一个新闭包（someMethod）的新对象。同时，变量 unused 是一个引用 originalThing 的闭包（先前的 replaceThing 又调用了 theThing ）。思绪混乱了吗？最重要的事情是，闭包的作用域一旦创建，它们有同样的父级作用域，作用域是共享的。someMethod 可以通过 theThing 使用，someMethod 与 unused 分享闭包作用域，尽管 unused从未使用，它引用的 originalThing 迫使它保留在内存中（防止被回收）。当这段代码反复运行，就会看到内存占用不断上升，垃圾回收器（GC）并无法降低内存占用。本质上，闭包的链表已经创建，每一个闭包作用域携带一个指向大数组的间接的引用，造成严重的内存泄露。解决办法：在 replaceThing 的最后添加 originalThing = null 。

# 5. 前端安全问题

## 5.1 XSS+

### 5.1.1 什么是XSS攻击

XSS全称是Cross Site Scripting即跨站脚本，当目标网站目标用户浏览器渲染HTML文档的过程中，出现了不被预期的脚本指令并执行时，XSS就发生了。

这里我们主要注意四点：1、目标网站目标用户；2、浏览器；3、不被预期；4、脚本。

### 5.1.2 XSS攻击有什么危害

* 挂马
* 盗取用户Cookie。
* DOS（拒绝服务）客户端浏览器。
* 钓鱼攻击，高级的钓鱼技巧。
* 删除目标文章、恶意篡改数据、嫁祸。
* 劫持用户Web行为，甚至进一步渗透内网。
* 爆发Web2.0蠕虫。
* 蠕虫式的DDoS攻击。
* 蠕虫式挂马攻击、刷广告、刷浏量、破坏网上数据

### 5.1.3 分类

XSS有三类：反射型XSS(非持久型)、存储型XSS(持久型)和DOM XSS。

### 5.1.4 如何防范

1. 注意特殊的 HTML 属性、JavaScript API
2. 根据上下文采用不同的转义规则
3. 预防存储型和反射型 XSS 攻击。
   1. 改成纯前端渲染，把代码和数据分隔开。
   2. 对 HTML 做充分转义。
4. Content Security Policy

* 严格的 CSP 在 XSS 的防范中可以起到以下的作用：
  + 禁止加载外域代码，防止复杂的攻击逻辑。
  + 禁止外域提交，网站被攻击后，用户的数据不会泄露到外域。
  + 禁止内联脚本执行（规则较严格，目前发现 GitHub 使用）。
  + 禁止未授权的脚本执行（新特性，Google Map 移动版在使用）。
  + 合理使用上报可以及时发现 XSS，利于尽快修复问题。

1. 输入内容长度控制
2. HTTP-only Cookie: 禁止 JavaScript 读取某些敏感 Cookie，攻击者完成 XSS 注入后也无法窃取此 Cookie。
3. 验证码：防止脚本冒充用户提交危险操作。

## 5.2 CSRF

**CSRF跨站点请求伪造(Cross—Site Request Forgery)。**攻击者盗用了你的身份，以你的名义发送恶意请求，对服务器来说这个请求是完全合法的，但是却完成了攻击者所期望的一个操作，比如以你的名义发送邮件、发消息，盗取你的账号，添加系统管理员，甚至于购买商品、虚拟货币转账等。

### 5.2.1 防御CSRF

目前防御 CSRF 攻击主要有三种策略：验证 HTTP Referer 字段；在请求地址中添加 token 并验证；在 HTTP 头中自定义属性并验证。

## 5.3 爬虫+

# 6. 跨域

## 6.1 什么是跨域？+

出于浏览器的同源策略限制。即当一个请求url的**协议、域名、端口**三者之间任意一个与当前页面url不同即为跨域。

|  |  |  |  |
| --- | --- | --- | --- |
| **当前页面url** | **被请求页面url** | **是否跨域** | **原因** |
| http://www.test.com/ | http://www.test.com/index.html | 否 | 同源（协议、域名、端口号相同） |
| http://www.test.com/ | https://www.test.com/index.html | 跨域 | 协议不同（http/https） |
| http://www.test.com/ | http://www.baidu.com/ | 跨域 | 主域名不同（test/baidu） |
| http://www.test.com/ | http://blog.test.com/ | 跨域 | 子域名不同（www/blog） |
| http://www.test.com:8080/ | http://www.test.com:7001/ | 跨域 | 端口号不同（8080/7001） |

## 6.2 什么是同源策略？

**同源策略**（Same origin policy）是一种约定，它是浏览器最核心也最基本的安全功能，如果缺少了同源策略，则浏览器的正常功能可能都会受到影响。可以说Web是构建在同源策略基础之上的，浏览器只是针对同源策略的一种实现。同源策略会阻止一个域的javascript脚本和另外一个域的内容进行交互。所谓同源（即指在同一个域）就是两个页面具有相同的协议（protocol），主机（host）和端口号（port）。

## 6.3 非同源限制

1. 无法读取非同源网页的 Cookie、LocalStorage 和 IndexedDB
2. 无法接触非同源网页的 DOM
3. 无法向非同源地址发送 AJAX 请求

## 6.4 不受同源策略限制的操作

1. 页面中的链接，重定向以及表单提交是不会受到同源策略限制的。
2. 跨域资源的引入是可以的。但是js不能读写加载的内容。如嵌入到页面中的<script src="..."></script>，<img>，<link>，<iframe>等。

## 6.5 解决跨域的方法有哪些？+

### 6.5.1 设置document.domain+iframe跨域 【解决无法读取非同源网页的cookie问题】

因为浏览器是通过document.domain属性来检查两个页面是否同源，因此只要通过设置相同的document.domain，两个页面就可以共享Cookie（此方案仅限主域相同，子域不同的跨域应用场景。）

// 两个页面都设置  
document.domain = 'test.com';

1. 父窗口

* <iframe id="iframe" src="http://child.domain.com/b.html"></iframe>  
  <script>  
   document.domain = 'domain.com';  
   var user = 'admin';  
  </script>

1. 子窗口

* <script>  
   document.domain = 'domain.com';  
   alert('get js data from parent -->'+window.parent.user);  
  </script>

### 6.5.2 location.hash +iframe跨域

实现原理： a欲与b跨域相互通信，通过中间页c来实现。 三个页面，不同域之间利用iframe的location.hash传值，相同域之间直接js访问来通信。

具体实现：A域：a.html -> B域：b.html -> A域：c.html，a与b不同域只能通过hash值单向通信，b与c也不同域也只能单向通信，但c与a同域，所以c可通过parent.parent访问a页面所有对象。

1. a.html (http://www.domain1.com/a.html)

* <iframe id="iframe" src="http://www.domain2.com/b.html" style="display:none;"></iframe>  
  <script>  
   var iframe = document.getElementById('iframe');  
    
   // 向b.html传hash值  
   setTimeout(function() {  
   iframe.src = iframe.src + '#user=admin';  
   }, 1000);  
     
   // 开放给同域c.html的回调方法  
   function onCallback(res) {  
   alert('data from c.html ---> ' + res);  
   }  
  </script>

1. b.html (http://www.domain2.com/b.html)

* <iframe id="iframe" src="http://www.domain1.com/c.html" style="display:none;"></iframe>  
  <script>  
   var iframe = document.getElementById('iframe');  
    
   // 监听a.html传来的hash值，再传给c.html  
   window.onhashchange = function () {  
   iframe.src = iframe.src + location.hash;  
   };  
  </script>

1. c.html (http://www.domain1.com/c.html)

* <script>  
   // 监听b.html传来的hash值  
   window.onhashchange = function () {  
   // 再通过操作同域a.html的js回调，将结果传回  
   window.parent.parent.onCallback('hello: ' + location.hash.replace('#user=', ''));  
   };  
  </script>

### 6.5.3 window.name+iframe跨域

window.name属性的独特之处：name值在不同的页面（甚至不同域名）加载后依旧存在，并且可以支持非常长的 name 值（2MB）。

1. a.html：(http://www.domain1.com/a.html)

var proxy = function(url, callback) {  
 var state = 0;  
 var iframe = document.createElement('iframe');  
  
 // 加载跨域页面  
 iframe.src = url;  
  
 // onload事件会触发2次，第1次加载跨域页，并留存数据于window.name  
 iframe.onload = function() {  
 if (state === 1) {  
 // 第2次onload(同域proxy页)成功后，读取同域window.name中数据  
 callback(iframe.contentWindow.name);  
 destoryFrame();  
  
 } else if (state === 0) {  
 // 第1次onload(跨域页)成功后，切换到同域代理页面  
 iframe.contentWindow.location = 'http://www.domain1.com/proxy.html';  
 state = 1;  
 }  
 };  
  
 document.body.appendChild(iframe);  
  
 // 获取数据以后销毁这个iframe，释放内存；这也保证了安全（不被其他域frame js访问）  
 function destoryFrame() {  
 iframe.contentWindow.document.write('');  
 iframe.contentWindow.close();  
 document.body.removeChild(iframe);  
 }  
};  
  
// 请求跨域b页面数据  
proxy('http://www.domain2.com/b.html', function(data){  
 alert(data);  
});

1. proxy.html：([http://www.domain1.com/proxy....](http://www.domain1.com/proxy.html))  
   中间代理页，与a.html同域，内容为空即可。
2. b.html：(http://www.domain2.com/b.html)

<script>  
 window.name = 'This is domain2 data!';  
</script>

### 6.5.4 跨文档通信API：window.postMessage()

调用postMessage方法实现父窗口http://test1.com向子窗口http://test2.com发消息（子窗口同样可以通过该方法发送消息给父窗口）

它可用于解决以下方面的问题：

* 页面和其打开的新窗口的数据传递
* 多窗口之间消息传递
* 页面与嵌套的iframe消息传递
* 上面三个场景的跨域数据传递

用法：postMessage(data,origin)方法接受两个参数

* data： html5规范支持任意基本类型或可复制的对象，但部分浏览器只支持字符串，所以传参时最好用JSON.stringify()序列化。
* origin： 协议+主机+端口号，也可以设置为"\*"，表示可以传递给任意窗口，如果要指定和当前窗口同源的话设置为"/"。

// 父窗口打开一个子窗口  
var openWindow = window.open('http://test2.com', 'title');  
   
// 父窗口向子窗口发消息(第一个参数代表发送的内容，第二个参数代表接收消息窗口的url)  
openWindow.postMessage('Nice to meet you!', 'http://test2.com');  
  
// 监听 message 消息  
window.addEventListener('message', function (e) {  
 console.log(e.source); // e.source 发送消息的窗口  
 console.log(e.origin); // e.origin 消息发向的网址  
 console.log(e.data); // e.data 发送的消息  
},false);

### 6.5.5 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，兼容性好（兼容低版本IE），缺点是只支持get请求，不支持post请求。

核心思想：网页通过添加一个<script>元素，向服务器请求 JSON 数据，服务器收到请求后，将数据放在一个指定名字的回调函数的参数位置传回来。

#### 原生实现

<script src="https://cdn.bootcss.com/jquery/3.3.1/jquery.js"></script>  
<script>  
 function handleCallback(res) {  
 console.log(res);  
 }  
  
 function addScriptTag(src) {  
 var scriptEle = document.createElement("script");  
 $(scriptEle).attr("src", src);  
 $("body").append(scriptEle);  
 $(scriptEle).remove();  
 }  
 $("#b1").click(function () {  
 addScriptTag("http://127.0.0.1:8002/abc/?callback=handleCallback")  
 });  
</script>

#### JQuery实现

//使用getJSON  
$("#b1").click(function () {  
 $.getJSON("http://127.0.0.1:8002/abc/?callback=?", function (res) {  
 console.log(res);  
 })  
 });  
  
//指定回调函数名  
$.ajax({  
 url: 'http://127.0.0.1:8002/abc/',  
 type: 'get',  
 dataType: 'jsonp', // 请求方式为jsonp  
 jsonpCallback: "handleCallback", // 自定义回调函数名  
 data: {}  
});  
  
$.ajax({  
 url: "http://127.0.0.1:8002/abc/",  
 dataType: "jsonp",  
 success: function (res) {  
 console.log(res);  
 }  
});

#### Vue.js实现

this.$http.jsonp('http://www.domain2.com:8080/login', {  
 params: {},  
 jsonp: 'handleCallback'  
}).then((res) => {  
 console.log(res);   
})

6.5.4

### 6.5.6 CORS

CORS 是跨域资源分享（Cross-Origin Resource Sharing）的缩写。它是 W3C 标准，属于跨源 AJAX 请求的根本解决方法。目前，所有浏览器都支持该功能(IE8+：IE8/9需要使用XDomainRequest对象来支持CORS）)，CORS也已经成为主流的跨域解决方案。

1. **普通跨域请求：只需服务器端设置Access-Control-Allow-Origin**
2. **带cookie跨域请求：前后端都需要进行设置**

#### 前端设置

根据xhr.withCredentials字段判断是否带有cookie

1. **原生ajax**

* var xhr = new XMLHttpRequest(); // IE8/9需用window.XDomainRequest兼容  
     
  // 前端设置是否带cookie！！  
  xhr.withCredentials = true;  
     
  xhr.open('post', 'http://www.domain2.com:8080/login', true);  
  xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');  
  xhr.send('user=admin');  
     
  xhr.onreadystatechange = function() {  
   if (xhr.readyState == 4 && xhr.status == 200) {  
   alert(xhr.responseText);  
   }  
  };

1. **JQuery ajax**

* $.ajax({  
   url: 'http://www.test.com:8080/login',  
   type: 'get',  
   data: {},  
   xhrFields: {  
   withCredentials: true // 前端设置是否带cookie  
   },  
   crossDomain: true, // 会让请求头中包含跨域的额外信息，但不会含cookie  
  });

1. **Vue框架**

* Vue.http.options.credentials = true //vue-resource  
  axios.defaults.withCredentials = true //axios

#### 服务端设置

服务器端对于CORS的支持，主要是通过设置Access-Control-Allow-Origin来进行的。如果浏览器检测到相应的设置，就可以允许Ajax进行跨域的访问。

1. Java后台

* /\*  
   \* 导入包：import javax.servlet.http.HttpServletResponse;  
   \* 接口参数中定义：HttpServletResponse response  
   \*/  
    
  // 允许跨域访问的域名：若有端口需写全（协议+域名+端口），若没有端口末尾不用加'/'  
  response.setHeader("Access-Control-Allow-Origin", "http://www.domain1.com");   
    
  // 允许前端带认证cookie：启用此项后，上面的域名不能为'\*'，必须指定具体的域名，否则浏览器会提示  
  response.setHeader("Access-Control-Allow-Credentials", "true");   
    
  // 提示OPTIONS预检时，后端需要设置的两个常用自定义头  
  response.setHeader("Access-Control-Allow-Headers", "Content-Type,X-Requested-With");

1. node.js后台

* var http = require('http');  
  var server = http.createServer();  
  var qs = require('querystring');  
     
  server.on('request', function(req, res) {  
   var postData = '';  
     
   // 数据块接收中  
   req.addListener('data', function(chunk) {  
   postData += chunk;  
   });  
     
   // 数据接收完毕  
   req.addListener('end', function() {  
   postData = qs.parse(postData);  
     
   // 跨域后台设置  
   res.writeHead(200, {  
   'Access-Control-Allow-Credentials': 'true', // 后端允许发送Cookie  
   'Access-Control-Allow-Origin': 'http://www.domain1.com', // 允许访问的域（协议+域名+端口）  
   /\*   
   \* 此处设置的cookie还是domain2的而非domain1，因为后端也不能跨域写cookie(nginx反向代理可以实现)，  
   \* 但只要domain2中写入一次cookie认证，后面的跨域接口都能从domain2中获取cookie，从而实现所有的接口都能跨域访问  
   \*/  
   'Set-Cookie': 'l=a123456;Path=/;Domain=www.domain2.com;HttpOnly' // HttpOnly的作用是让js无法读取cookie  
   });  
     
   res.write(JSON.stringify(postData));  
   res.end();  
   });  
  });  
     
  server.listen('8080');  
  console.log('Server is running at port 8080...');

### 6.5.7 nginx代理跨域

### 6.5.8 Nodejs中间件代理跨域

### 6.5.9 WebSocket协议跨域

## 6.6 CORS详解

### 6.6.1 两种请求

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）。

**简单请求（只要满足以下两大条件）：**

1. 请求方法是以下三种方法之一：HEAD、GET、POST
2. HTTP请求头中仅包含以下几种字段：Accept、Accept-Language、Content-Language、Last-Event-ID、Content-Type：仅限于application/x-www-form-urlencoded、multipart/form-data、text/plain

为了兼容表单form，AJAX 的跨域设计就是，只要表单可以发，AJAX 就可以直接发。凡是不同时满足上面两个条件，就属于非简单请求。

### 6.6.2 简单请求

#### 基本流程

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。用来说明，本次请求来自哪个源（协议 + 域名 + 端口）。

服务器根据这个值，决定是否同意这次请求。

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: http://api.bob.com   
Access-Control-Allow-Credentials: true   
Access-Control-Expose-Headers: FooBar   
Content-Type: text/html; charset=utf-8

**Access-Control-Allow-Origin**：必须。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**：可选。布尔值，表示是否允许发送Cookie。默认情况下，Cookie不包括在CORS请求之中。设为true，即表示服务器明确许可Cookie。

**Access-Control-Expose-Headers**：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。

#### withCredentials属性

允许携带cookie

### 6.6.3 非简单请求

#### 预检请求

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）。一旦服务器通过了**"**预检**"**请求，以后每次浏览器正常的**CORS**请求，就都跟简单请求一样，会有一个**Origin**头信息字段。

**浏览器端**

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。

var url = 'http://api.alice.com/cors';  
var xhr = new XMLHttpRequest();  
xhr.open('PUT', url, true);  
xhr.setRequestHeader('X-Custom-Header', 'value');  
xhr.send();

上面代码中，HTTP请求的方法是PUT，并且发送一个自定义头信息X-Custom-Header。

浏览器发现，这是一个非简单请求，就自动发出一个"预检"请求，要求服务器确认可以这样请求。下面是这个"预检"请求的HTTP头信息。

OPTIONS /cors HTTP/1.1  
Origin: http://api.bob.com  
Access-Control-Request-Method: PUT  
Access-Control-Request-Headers: X-Custom-Header  
Host: api.alice.com  
Accept-Language: en-US  
Connection: keep-alive  
User-Agent: Mozilla/5.0...

"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。

除了Origin字段，"预检"请求的头信息包括两个特殊字段。

**Access-Control-Request-Method**：必须。用来列出浏览器的CORS请求会用到哪些HTTP方法，上例是PUT。

**Access-Control-Request-Headers**：该字段是一个逗号分隔的字符串，指定浏览器CORS请求会额外发送的头信息字段，上例是X-Custom-Header。

**服务器端**

发送该请求后，服务器可以决定是否允许这种类型的请求。服务器通过在响应中发送如下头部与浏览器进行沟通。

Access-Control-Allow-Methods: GET, POST, PUT  
Access-Control-Allow-Headers: X-Custom-Header  
Access-Control-Allow-Credentials: true  
Access-Control-Max-Age: 1728000

**Access-Control-Allow-Origin**：与简单的请求相同。

**Access-Control-Allow-Credentials**：与简单的请求相同。

**Access-Control-Allow-Methods**：必须。它的值是逗号分隔的一个字符串，表明服务器**支持的所有跨域请求**的方法。注意，返回的是所有支持的方法，而不单是浏览器请求的那个方法。**这是为了避免多次"预检"请求。**

**Access-Control-Allow-Headers**：如果浏览器请求包括Access-Control-Request-Headers字段，则Access-Control-Allow-Headers字段是必需的。它也是一个逗号分隔的字符串，表明服务器支持的所有头信息字段，不限于浏览器在"预检"中请求的字段。

**Access-Control-Max-Age**：可选，用来指定本次预检请求的有效期，单位为秒。上面结果中，有效期是20天（1728000秒），即允许缓存该条回应1728000秒（即20天），在此期间，不用发出另一条预检请求。

# 7. 性能优化

## 7.1 雅虎军规35条

### 7.1.1 内容部分

#### 1.尽量减少HTTP请求数

80%的终端用户响应时间都花在了前端上，其中大部分时间都在下载页面上的各种组件：图片，样式表，脚本，Flash等等。减少组件数必然能够减少页面提交的HTTP请求数。这是让页面更快的关键。

减少页面组件数的一种方式是简化页面设计。但有没有一种方法可以在构建复杂的页面同时加快响应时间呢？嗯，确实有鱼和熊掌兼得的办法。

**合并文件**是通过把所有脚本放在一个文件中的方式来减少请求数的，当然，也可以合并所有的CSS。如果各个页面的脚本和样式不一样的话，合并文件就是一项比较麻烦的工作了，但把这个作为站点发布过程的一部分确实可以提高响应时间。

**CSS Sprites**是减少图片请求数量的首选方式。把背景图片都整合到一张图片中，然后用CSS的background-image和background-position属性来定位要显示的部分。

**图像映射**可以把多张图片合并成单张图片，总大小是一样的，但减少了请求数并加速了页面加载。图片映射只有在图像在页面中连续的时候才有用，比如导航条。给image map设置坐标的过程既无聊又容易出错，用image map来做导航也不容易，所以不推荐用这种方式。

**行内图片（Base64编码）**用**data: URL**模式来把图片嵌入页面。这样会增加HTML文件的大小，把行内图片放在（缓存的）样式表中是个好办法，而且成功避免了页面变“重”。但目前主流浏览器并不能很好地支持行内图片。

减少页面的HTTP请求数是个起点，这是提升站点首次访问速度的重要指导原则。

#### 2.减少DNS查找

域名系统建立了主机名和IP地址间的映射，就像电话簿上人名和号码的映射一样。当你在浏览器输入www.yahoo.com的时候，浏览器就会联系DNS解析器返回服务器的IP地址。DNS是有成本的，它需要20到120毫秒去查找给定主机名的IP地址。在DNS查找完成之前，浏览器无法从主机名下载任何东西。

DNS查找被缓存起来更高效，由用户的ISP（网络服务提供商）或者本地网络存在一个特殊的缓存服务器上，但还可以缓存在个人用户的计算机上。DNS信息被保存在操作系统的DNS cache(微软Windows上的”DNS客户端服务”)里。大多数浏览器有独立于操作系统的自己的cache。只要浏览器在自己的cache里还保留着这条记录，它就不会向操作系统查询DNS。

IE默认缓存DNS查找30分钟，写在DnsCacheTimeout注册表设置中。Firefox缓存1分钟，可以用network.dnsCacheExpiration配置项设置。(Fasterfox把缓存时间改成了1小时 P.S. Fasterfox是FF的一个提速插件)

如果客户端的DNS cache是空的（包括浏览器的和操作系统的），DNS查找数等于页面上不同的主机名数，包括页面URL，图片，脚本文件，样式表，Flash对象等等组件中的主机名，减少不同的主机名就可以减少DNS查找。

减少不同主机名的数量同时也减少了页面能够并行下载的组件数量，避免DNS查找削减了响应时间，而减少并行下载数量却增加了响应时间。我的原则是把组件分散在2到4个主机名下，这是同时减少DNS查找和允许高并发下载的折中方案。

#### 3.避免重定向

重定向用301和302状态码，下面是一个有301状态码的HTTP头：

HTTP/1.1 301 Moved Permanently  
 Location: http://example.com/newuri  
 Content-Type: text/html

浏览器会自动跳转到Location域指明的URL。重定向需要的所有信息都在HTTP头部，而响应体一般是空的。其实额外的HTTP头，比如Expires和Cache-Control也表示重定向。除此之外还有别的跳转方式：refresh元标签和JavaScript，但如果你必须得做重定向，最好用标准的3xxHTTP状态码，主要是为了让返回按钮能正常使用。

牢记**重定向会拖慢用户体验**，在用户和HTML文档之间插入重定向会延迟页面上的所有东西，页面无法渲染，组件也无法开始下载，直到HTML文档被送达浏览器。

**有一种常见的极其浪费资源的重定向，而且web开发人员一般都意识不到这一点，就是URL尾部缺少一个斜线的时候。例如，跳转到http://astrology.yahoo.com/astrology会返回一个重定向到http://astrology.yahoo.com/astrology/的301响应（注意添在尾部的斜线）。在Apache中可以用Alias，mod\_rewrite或者DirectorySlash指令来取消不必要的重定向。**

重定向最常见的用途是把旧站点连接到新的站点，还可以连接同一站点的不同部分，针对用户的不同情况（浏览器类型，用户帐号类型等等）做一些处理。用重定向来连接两个网站是最简单的，只需要少量的额外代码。虽然在这些时候使用重定向减少了开发人员的开发复杂度，但降低了用户体验。一种替代方案是用Alias和mod\_rewrite，前提是两个代码路径都在相同的服务器上。如果是因为域名变化而使用了重定向，就可以创建一条CNAME（创建一个指向另一个域名的DNS记录作为别名）结合Alias或者mod\_rewrite指令。

#### 4.Ajax可缓存

Ajax的一个好处是可以给用户提供即时反馈，因为它能够从后台服务器异步请求信息。然而，用了Ajax就无法保证用户在等待异步JavaScript和XML响应返回期间不会非常无聊。在很多应用程序中，用户能够一直等待取决于如何使用Ajax。例如，在基于web的电子邮件客户端中，用户为了寻找符合他们搜索标准的邮件消息，将会保持对Ajax请求返回结果的关注。重要的是，要记得“异步”并不意味着“即时”。

要提高性能，优化这些Ajax响应至关重要。最重要的提高Ajax性能的方法就是**让响应变得可缓存**，就像在添上Expires或者Cache-Control HTTP头中讨论的一样。下面适用于Ajax的其它规则：

* Gzip组件
* 减少DNS查找
* 压缩JavaScript
* 避免重定向
* 配置ETags

我们一起看看例子，一个Web 2.0的电子邮件客户端用了Ajax来下载用户的通讯录，以便实现自动完成功能。如果用户从上一次使用之后再没有修改过她的通讯录，而且Ajax响应是可缓存的，有尚未过期的Expires或者Cache-Control HTTP头，那么之前的通讯录就可以从缓存中读出。必须通知浏览器，应该继续使用之前缓存的通讯录响应，还是去请求一个新的。可以通过给通讯录的Ajax URL里添加一个表明用户通讯录最后修改时间的时间戳来实现，例如&t=1190241612。如果通讯录从上一次下载之后再没有被修改过，时间戳不变，通讯录就将从浏览器缓存中直接读出，从而避免一次额外的HTTP往返消耗。如果用户已经修改了通讯录，时间戳也可以确保新的URL不会匹配缓存的响应，浏览器将请求新的通讯录条目。

即使Ajax响应是动态创建的，而且可能只适用于单用户，它们也可以被缓存，而这样会让你的Web 2.0应用更快。

#### 5.延迟加载组件

什么才是一开始渲染页面所必须的？其余内容都可以等会儿。JavaScript是分隔onload事件之前和之后的一个理想选择。例如，如果有JavaScript代码和支持拖放以及动画的库，这些都可以先等会儿，因为拖放元素是在页面最初渲染之后的。其它可以延迟加载的部分包括隐藏内容（在某个交互动作之后才出现的内容）和折叠的图片。

工具可帮你减轻工作量：YUI Image Loader可以延迟加载折叠的图片，还有YUI Get utility是一种引入JS和CSS的简单方法。

最好让性能目标符合其它web开发最佳实践，比如“渐进增强”。如果客户端支持JavaScript，可以提高用户体验，但必须确保页面在不支持JavaScript时也能正常工作。所以，在确定页面运行正常之后，可以用一些延迟加载脚本增强它，以支持一些拖放和动画之类的华丽效果。

#### 6.预加载组件

你都用预加载可能看起来和延迟加载是相反的，但它其实有不同的目标。通过预加载组件可以充分利用浏览器空闲的时间来请求将来会用到的组件（图片，样式和脚本）。用户访问下一页的时候，大部分组件都已经在缓存里了，所以在用户看来页面会加载得更快。

实际应用中有以下几种预加载的类型：

* 无条件预加载——尽快开始加载，获取一些额外的组件。google.com就是一个sprite图片预加载的好例子，这个sprite图片并不是google.com主页需要的，而是搜索结果页面上的内容。
* 条件性预加载——根据用户操作猜测用户将要跳转到哪里并据此预加载。
* 提前预加载——在推出新设计之前预加载。经常在重新设计之后会听到：“这个新网站不错，但比以前更慢了”，一部分原因是用户访问先前的页面都是有旧缓存的，但新的却是一种空缓存状态下的体验。可以通过在将要推出新设计之前预加载一些组件来减轻这种负面影响，老站可以利用浏览器空闲的时间来请求那些新站需要的图片和脚本。

#### 7.减少DOM元素的数量

一个复杂的页面意味着要下载更多的字节，而且用JavaScript访问DOM也会更慢。举个例子，想要添加一个事件处理器的时候，循环遍历页面上的500个DOM元素和5000个DOM元素是有区别的。

[YUI CSS utilities](https://developer.yahoo.com/yui/)对布局有很大帮助：grids.css针对整体布局，fonts.css和reset.css可以用来去除浏览器的默认格式。这是个开始清理和思考标记的好机会，例如只在语义上有意义的时候使用<div>，而不是因为它能够渲染一个新行。

#### 8.跨域分离组件

分离组件可以最大化并行下载，但要确保只用不超过2-4个域，因为存在DNS查找的代价。例如，可以把HTML和动态内容部署在www.example.org，而把静态组件分离到static1.example.org和static2.example.org。

#### 9.尽量少用iframe

用iframe可以把一个HTML文档插入到父文档里，重要的是明白iframe是如何工作的并高效地使用它。

<iframe>的优点：

* 引入缓慢的第三方内容，比如标志和广告
* 安全沙箱
* 并行下载脚本

<iframe>的缺点：

* 代价高昂，即使是空白的iframe
* 阻塞页面加载
* 非语义

#### 10.杜绝404

HTTP请求代价高昂，完全没有必要用一个HTTP请求去获取一个无用的响应（比如404 Not Found），只会拖慢用户体验而没有任何好处。

有些站点用的是有帮助的404——“你的意思是xxx？”，这样做有利于用户体验，，但也浪费了服务器资源（比如数据库等等）。最糟糕的是链接到的外部JavaScript有错误而且结果是404。首先，这种下载将阻塞并行下载。其次，浏览器会试图解析404响应体，因为它是JavaScript代码，需要找出其中可用的部分。

### 7.1.2 CSS部分

#### 11.避免使用CSS表达式

用CSS表达式动态设置CSS属性，是一种强大又危险的方式。从IE5开始支持，但从IE8起就不推荐使用了。

#### 12.使用<link>舍弃@import

为了实现逐步渲染，CSS应该放在顶部。在IE中用@import与在底部用效果一样，所以最好不要用它。

#### 13.避免使用滤镜

IE专有的AlphaImageLoader滤镜可以用来修复IE7之前的版本中半透明PNG图片的问题。在图片加载过程中，这个滤镜会阻塞渲染，卡住浏览器，还会增加内存消耗而且是被应用到每个元素的，而不是每个图片，所以会存在一大堆问题。

最好的方法是干脆不要用AlphaImageLoader，而优雅地降级到用在IE中支持性很好的PNG8图片来代替。如果非要用AlphaImageLoader，应该用下划线hack：\_filter来避免影响IE7及更高版本的用户。

#### 14.把样式表放在顶部

我们发现把样式表放到文档的HEAD部分能让页面看起来加载地更快。这是因为把样式表放在head里能让页面逐步渲染。

关注性能的前端工程师想让页面逐步渲染。也就是说，我们想让浏览器尽快显示已有内容，这在页面上有一大堆内容或者用户网速很慢时显得尤为重要。给用户显示反馈（比如进度指标）的重要性已经被广泛研究过，并且被记录下来了。在我们的例子中，HTML页面就是进度指标！当浏览器逐渐加载页面头部，导航条，顶部logo等等内容的时候，这些都被正在等待页面加载的用户当作反馈，能够提高整体用户体验。

### 7.1.3 JS部分

#### 15.去除重复脚本

页面含有重复的脚本文件会影响性能，这可能和你想象的不一样。在对美国前10大web站点的评审中，发现只有2个站点含有重复脚本。两个主要原因增加了在单一页面中出现重复脚本的几率：团队大小和脚本数量。在这种情况下，重复脚本会创建不必要的HTTP请求，执行无用的JavaScript代码，而影响页面性能。

IE会产生不必要的HTTP请求，而Firefox不会。在IE中，如果一个不可缓存的外部脚本被页面引入了两次，它会在页面加载时产生两个HTTP请求。即使脚本是可缓存的，在用户重新加载页面时也会产生额外的HTTP请求。

除了产生没有意义的HTTP请求之外，多次对脚本求值也会浪费时间。因为无论脚本是否可缓存，在Firefox和IE中都会执行冗余的JavaScript代码。

避免不小心把相同脚本引入两次的一种方法就是在模版系统中实现脚本管理模块。

#### 16.尽量减少DOM访问

用JavaScript访问DOM元素是很慢的，所以，为了让页面反应更迅速，应该：

* 缓存已访问过的元素的索引
* 先“离线”更新节点，再把它们添到DOM树上
* 避免用JavaScript修复布局问题

#### 17.用智能的事件处理器

有时候感觉页面反映不够灵敏，是因为有太多频繁执行的事件处理器被添加到了DOM树的不同元素上，这就是**推荐使用事件委托**的原因。如果一个div里面有10个按钮，应该只给div容器添加一个事件处理器，而不是给每个按钮都添加一个。事件能够冒泡，所以可以捕获事件并得知哪个按钮是事件源。

#### 18.把脚本放在底部

脚本会阻塞并行下载，HTTP/1.1官方文档建议浏览器每个主机名下并行下载的组件数不要超过两个，如果图片来自多个主机名，并行下载的数量就可以超过两个。如果脚本正在下载，浏览器就不开始任何其它下载任务，即使是在不同主机名下的。

有时候，并不容易把脚本移动到底部。举个例子，如果脚本是用document.write插入到页面内容中的，就没办法再往下移了。还可能存在作用域问题，在多数情况下，这些问题都是可以解决的。

一个常见的建议是**用推迟（deferred）脚本**，有DEFER属性的脚本意味着不能含有document.write，并且提示浏览器告诉他们可以继续渲染。**不幸的是，Firefox不支持DEFER属性。**在IE中，脚本可能被推迟，但不尽如人意。如果脚本可以推迟，我们就可以把它放到页面底部，页面就可以更快地载入。

### 7.1.4 JS和CSS

#### 19.把JavaScript和CSS放到外面

使用外部文件可以让页面更快，因为JavaScript和CSS文件会被缓存在浏览器。HTML文档中的行内JavaScript和CSS在每次请求该HTML文档的时候都会重新下载。这样做减少了所需的HTTP请求数，但增加了HTML文档的大小。另一方面，如果**JavaScript和CSS在外部文件中，并且已经被浏览器缓存起来了，那么我们就成功地把HTML文档变小了，而且还没有增加HTTP请求数。**

#### 20.压缩JavaScript和CSS

压缩具体来说就是从代码中去除不必要的字符以减少大小，从而提升加载速度。代码最小化就是去掉所有注释和不必要的空白字符（空格，换行和tab）。在JavaScript中这样做能够提高响应性能，因为要下载的文件变小了。两个最常用的JavaScript代码压缩工具是JSMin和YUI Compressor，YUI compressor还可以压缩CSS。

混淆是一种可选的源码优化措施，要比压缩更复杂，所以混淆过程也更容易产生bug。在对美国前十的网站调查中，压缩可以缩小21%，而混淆能缩小25%。虽然混淆的缩小程度更高，但比压缩风险更大。

除了压缩外部脚本和样式，行内的<script>和<style>块也可以压缩。即使启用了gzip模块，先进行压缩也能够缩小5%或者更多的大小。JavaScript和CSS的用处越来越多，所以压缩代码会有不错的效果。

### 7.1.5 图片部分

#### 21.优化图片

* 尝试把GIF格式转换成PNG格式，看看是否节省空间。在所有的PNG图片上运行pngcrush（或者其它PNG优化工具）

#### 22.优化CSS Sprite

* 在Sprite图片中横向排列一般都比纵向排列的最终文件小
* 组合Sprite图片中的相似颜色可以保持低色数，最理想的是256色以下PNG8格式
* “对移动端友好”，不要在Sprite图片中留下太大的空隙。虽然不会在很大程度上影响图片文件的大小，但这样做可以节省用户代理把图片解压成像素映射时消耗的内存。100×100的图片是1万个像素，而1000×1000的图片就是100万个像素了。

#### 23.不要用HTML缩放图片

不要因为在HTML中可以设置宽高而使用本不需要的大图。如果需要``

<img width="100" height="100" src="mycat.jpg" alt="My Cat" />

那么图片本身（mycat.jpg）应该是100x100px的，而不是去缩小500x500px的图片。

#### 24.用小的可缓存的favicon.ico

favicon.ico是放在服务器根目录的图片，它会带来一堆麻烦，因为即便你不管它，浏览器也会自动请求它，所以最好不要给一个404 Not Found响应。而且只要在同一个服务器上，每次请求它时都会发送cookie，此外这个图片还会干扰下载顺序，例如在IE中，当你在onload中请求额外组件时，将会先下载favicon。

所以为了缓解favicon.ico的缺点，应该确保：

* 足够小，最好在1K以下
* 设置合适的有效期HTTP头（以后如果想换的话就不能重命名了），把有效期设置为几个月后一般比较安全，可以通过检查当前favicon.ico的最后修改日期来确保变更能让浏览器知道。

#### 25.给cookie减肥

使用cookie的原因有很多，比如授权和个性化。HTTP头中cookie信息在web服务器和浏览器之间交换。重要的是保证cookie尽可能的小，以最小化对用户响应时间的影响。

* 清除不必要的cookie
* 保证cookie尽可能小，以最小化对用户响应时间的影响
* 注意给cookie设置合适的域级别，以免影响其它子域
* 设置合适的有效期，更早的有效期或者none可以更快的删除cookie，提高用户响应时间

#### 26.把组件放在不含cookie的域下

当浏览器发送对静态图像的请求时，cookie也会一起发送，而服务器根本不需要这些cookie。所以它们只会造成没有意义的网络通信量，应该确保对静态组件的请求不含cookie。可以创建一个子域，把所有的静态组件都部署在那儿。

如果域名是www.example.org，可以把静态组件部署到static.example.org。然而，如果已经在顶级域example.org或者www.example.org设置了cookie，那么所有对static.example.org的请求都会含有这些cookie。这时候可以再买一个新域名，把所有的静态组件部署上去，并保持这个新域名不含cookie。Yahoo!用的是yimg.com，YouTube是ytimg.com，Amazon是images-amazon.com等等。

把静态组件部署在不含cookie的域下还有一个好处是有些代理可能会拒绝缓存带cookie的组件。有一点需要注意：如果不知道应该用example.org还是www.example.org作为主页，可以考虑一下cookie的影响。省略www的话，就只能把cookie写到\*.example.org，所以因为性能原因最好用www子域，并且把cookie写到这个子域下。

### 7.1.6 移动端

#### 27.保证所有组件都小于25K

这个限制是因为iPhone不能缓存大于25K的组件，注意这里指的是未压缩的大小。这就是为什么缩减内容本身也很重要，因为单纯的gzip可能不够。

#### 28.把组件打包到一个复合文档里

把各个组件打包成一个像有附件的电子邮件一样的复合文档里，可以用一个HTTP请求获取多个组件（记住一点：HTTP请求是代价高昂的）。用这种方式的时候，要先检查用户代理是否支持（iPhone就不支持）。

### 7.1.7 服务器

#### 29.Gzip组件

前端工程师可以想办法明显地缩短通过网络传输HTTP请求和响应的时间。毫无疑问，终端用户的带宽速度，网络服务商，对等交换点的距离等等，都是开发团队所无法控制的。但还有别的能够影响响应时间的因素，压缩可以通过减少HTTP响应的大小来缩短响应时间。

从HTTP/1.1开始，web客户端就有了支持压缩的Accept-Encoding HTTP请求头。

Accept-Encoding: gzip, deflate

如果web服务器看到这个请求头，它就会用客户端列出的一种方式来压缩响应。web服务器通过Content-Encoding相应头来通知客户端。

Content-Encoding: gzip

尽可能多地用gzip压缩能够给页面减肥，这也是提升用户体验最简单的方法。

#### 30.避免图片src属性为空

Image with empty string **src**属性是空字符串的图片很常见，主要以两种形式出现：

1. straight HTML

<img src=""/>

1. JavaScript

var img = new Image();  
img.src = "";

这两种形式都会引起相同的问题：浏览器会向服务器发送另一个请求。

#### 31.配置ETags

**实体标签（ETags），是服务器和浏览器用来决定浏览器缓存中组件与源服务器中的组件是否匹配的一种机制**（“实体”也就是组件：图片，脚本，样式表等等）。添加ETags可以提供一种实体验证机制，比最后修改日期更加灵活。一个ETag是一个字符串，作为一个组件某一具体版本的唯一标识符。唯一的格式约束是字符串必须用引号括起来，源服务器用相应头中的ETag来指定组件的ETag：

HTTP/1.1 200 OK  
Last-Modified: Tue, 12 Dec 2006 03:03:59 GMT  
ETag: "10c24bc-4ab-457e1c1f"  
Content-Length: 12195

然后，如果浏览器必须验证一个组件，它用If-None-Match请求头来把ETag传回源服务器。如果ETags匹配成功，会返回一个304状态码，这样就减少了12195个字节的响应体。

GET /i/yahoo.gif HTTP/1.1  
Host: us.yimg.com  
If-Modified-Since: Tue, 12 Dec 2006 03:03:59 GMT  
If-None-Match: "10c24bc-4ab-457e1c1f"  
HTTP/1.1 304 Not Modified

#### 32.对Ajax用GET请求

Yahoo!邮箱团队发现使用XMLHttpRequest时，浏览器的POST请求是通过一个两步的过程来实现的：先发送HTTP头，在发送数据。所以最好用GET请求，它只需要发送一个TCP报文（除非cookie特别多）。IE的URL长度最大值是2K，所以如果要发送的数据超过2K就无法使用GET了。

POST请求的一个有趣的副作用是实际上没有发送任何数据，就像GET请求一样。正如HTTP说明文档中描述的，GET请求是用来检索信息的。所以它的语义只是用GET请求来请求数据，而不是用来发送需要存储到服务器的数据。

#### 33.尽早清空缓冲区

当用户请求一个页面时，服务器需要用大约200到500毫秒来组装HTML页面，在这期间，浏览器闲等着数据到达。PHP中有一个[flush()](http://php.net/flush)函数，允许给浏览器发送一部分已经准备完毕的HTML响应，以便浏览器可以在后台准备剩余部分的同时开始获取组件，好处主要体现在很忙的后台或者很“轻”的前端页面上（P.S. 也就是说，响应时耗主要在后台方面时最能体现优势）。

较理想的清空缓冲区的位置是HEAD后面，因为HTML的HEAD部分通常更容易生成，并且允许引入任何CSS和JavaScript文件，这样就可以让浏览器在后台还在处理的时候就开始并行获取组件。

例如：

<head>  
 <!-- css, js -->  
</head>  
<?php flush(); ?>  
<body>  
 <!-- content -->  
</body>

#### 34.使用CDN（内容分发网络）

用户与服务器的物理距离对响应时间也有影响。把内容部署在多个地理位置分散的服务器上能让用户更快地载入页面。但具体要怎么做呢？

实现内容在地理位置上分散的第一步是：不要尝试去重新设计你的web应用程序来适应分布式结构。这取决于应用程序，改变结构可能包括一些让人望而生畏的任务，比如同步会话状态和跨服务器复制数据库事务（翻译可能不准确）。缩短用户和内容之间距离的提议可能被推迟，或者根本不可能通过，就是因为这个难题。

记住终端用户80%到90%的响应时间都花在下载页面组件上了：图片，样式，脚本，Flash等等，这是业绩黄金法则。最好先分散静态内容，而不是一开始就重新设计应用程序结构。这不仅能够大大减少响应时间，还更容易表现出CDN的功劳。

内容分发网络（CDN）是一组分散在不同地理位置的web服务器，用来给用户更高效地发送内容。典型地，选择用来发送内容的服务器是基于网络距离的衡量标准的。例如：选跳数（hop）最少的或者响应时间最快的服务器。

#### 35.添加Expires或者Cache-Control HTTP头

这条规则有两个方面：

* 对于静态组件：通过设置一个遥远的将来时间作为Expires来实现永不失效
* 多余动态组件：用合适的Cache-ControlHTTP头来让浏览器进行条件性的请求

网页设计越来越丰富，这意味着页面里有更多的脚本，图片和Flash。站点的新访客可能还是不得不提交几个HTTP请求，但通过使用有效期能让组件变得可缓存，这避免了在接下来的浏览过程中不必要的HTTP请求。有效期HTTP头通常被用在图片上，但它们应该用在所有组件上，包括脚本、样式和Flash组件。

浏览器（和代理）用缓存来减少HTTP请求的数目和大小，让页面能够更快加载。web服务器通过有效期HTTP响应头来告诉客户端，页面的各个组件应该被缓存多久。用一个遥远的将来时间做有效期，告诉浏览器这个响应在2010年4月15日前不会改变。

Expires: Thu, 15 Apr 2010 20:00:00 GMT

如果你用的是Apache服务器，用ExpiresDefault指令来设置相对于当前日期的有效期。下面的例子设置了从请求时间起10年的有效期：

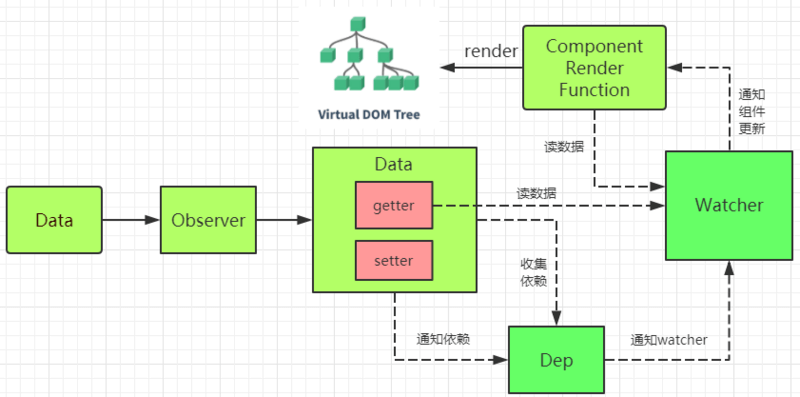
ExpiresDefault "access plus 10 years"

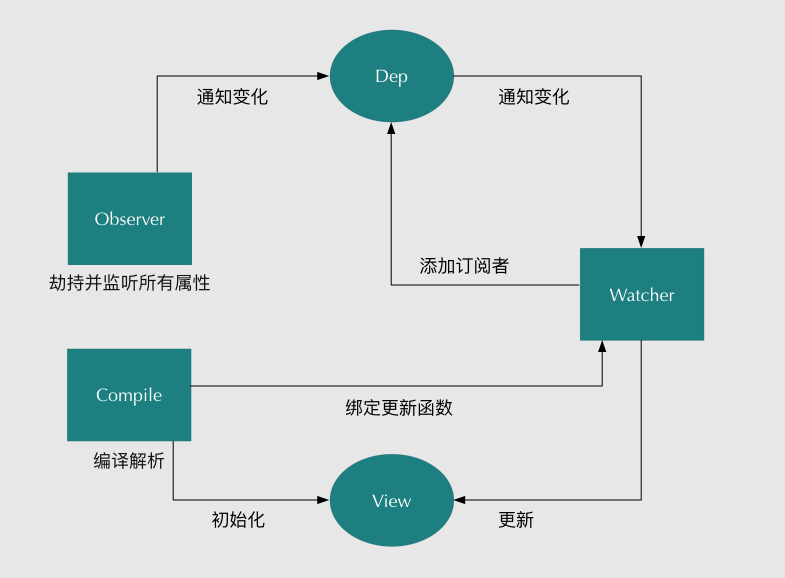
7.2 CSS Sprite是什么，谈谈这个技术的优缺点

# 8. Vue.js

## 8.1 Vue如何实现双向绑定+

Vue采用数据劫持结合发布者-订阅者模式的方法，通过**Object.defineProperty()**来劫持各个属性的setter,getter属性，在数据变动话，通知订阅者，触发更新回调函数，重新渲染视图.





### 8.1.1 核心实现类:

**Observer** : 它的作用是给对象的属性添加 getter 和 setter，用于依赖收集和派发更新

**Dep** : 用于收集当前响应式对象的依赖关系,每个响应式对象包括子对象都拥有一个 Dep 实例（里面 subs 是 Watcher 实例数组）,当数据有变更时,会通过 dep.notify()通知各个 watcher。

**Watcher** : 观察者对象 , 实例分为渲染 watcher (render watcher),计算属性 watcher (computed watcher),侦听器 watcher（user watcher）三种

### 8.1.2 Watcher 和 Dep 的关系

watcher 中实例化了 dep 并向 dep.subs 中添加了订阅者,dep 通过 notify 遍历了 dep.subs 通知每个 watcher 更新。

### 8.1.3 依赖收集

1. initState 时,对 computed 属性初始化时,触发 computed watcher 依赖收集
2. initState 时,对侦听属性初始化时,触发 user watcher 依赖收集
3. render()的过程,触发 render watcher 依赖收集
4. re-render 时,vm.render()再次执行,会移除所有 subs 中的 watcer 的订阅,重新赋值。

### 8.1.4 派发更新

1. 组件中对响应的数据进行了修改,触发 setter 的逻辑
2. 调用 dep.notify()
3. 遍历所有的 subs（Watcher 实例）,调用每一个 watcher 的 update 方法。

### 8.1.5 原理

当创建 Vue 实例时,vue 会遍历 data 选项的属性,利用 Object.defineProperty 为属性添加 getter 和 setter 对数据的读取进行劫持（getter 用来依赖收集,setter 用来派发更新）,并且在内部追踪依赖,在属性被访问和修改时通知变化。

每个组件实例会有相应的 watcher 实例,会在组件渲染的过程中记录依赖的所有数据属性（进行依赖收集,还有 computed watcher,user watcher 实例）,之后依赖项被改动时,setter 方法会通知依赖与此 data 的 watcher 实例重新计算（派发更新）,从而使它关联的组件重新渲染。

一句话总结:

vue.js 采用数据劫持结合发布-订阅模式,通过 Object.defineproperty 来劫持各个属性的 setter,getter,在数据变动时发布消息给订阅者,触发响应的监听回调

### 8.1.1 Vue3中的变化

使用es2015中的Proxy替代Object.defineProperty。Object.defineProperty 只能劫持对象的属性,因此我们需要对每个对象的每个属性进行遍历。Vue 2.x 里,是通过 递归 + 遍历 data 对象来实现对数据的监控的,如果属性值也是对象那么需要深度遍历,显然如果能劫持一个完整的对象是才是更好的选择。Proxy 可以劫持整个对象,并返回一个新的对象。Proxy 不仅可以代理对象,还可以代理数组。还可以代理动态增加的属性。

## 8.2 computed的实现原理

computed 本质是一个惰性求值的观察者。

computed 内部实现了一个惰性的 watcher,也就是 computed watcher,computed watcher 不会立刻求值,同时持有一个 dep 实例。

其内部通过 this.dirty 属性标记计算属性是否需要重新求值。

当 computed 的依赖状态发生改变时,就会通知这个惰性的 watcher,

computed watcher 通过 this.dep.subs.length 判断有没有订阅者,

有的话,会重新计算,然后对比新旧值,如果变化了,会重新渲染。 (**Vue 想确保不仅仅是计算属性依赖的值发生变化，而是当计算属性最终计算的值发生变化时才会触发渲染 watcher 重新渲染，本质上是一种优化。**)

没有的话,仅仅把 this.dirty = true。 (**当计算属性依赖于其他数据时，属性并不会立即重新计算，只有之后其他地方需要读取属性的时候，它才会真正计算，即具备 lazy（懒计算）特性。**)

## 8.3 computed和watch的区别

**computed** 依赖其他的值,且具有缓存,缓存变化才会更新。**计算属性是基于它们的响应式依赖进行缓存的**

* 只有它依赖的属性值发生改变，下一次获取 computed 的值时才会重新计算 进行数值计算，并且依赖于其它数据用它

**watch** 没有缓存 监听某一个值 变化进行一些操作

* 数据变化时执行异步或开销较大的操作时用它，使用 watch 选项允许我们执行异步操作 ( 访问一个 API ),限制我们执行该操作的频率,并在我们得到最终结果前,设置中间状态。这些都是计算属性无法做到的。

## 8.4 vue的diff算法

我们先根据真实DOM生成一颗virtual DOM，当virtual DOM某个节点的数据改变后会生成一个新的Vnode，然后Vnode和oldVnode作对比，发现有不一样的地方就直接修改在真实的DOM上，然后使oldVnode的值为Vnode。

diff的过程就是调用名为patch的函数，比较新旧节点，一边比较一边给**真实的DOM**打补丁。

### diff的比较方式？

在采取diff算法比较新旧节点的时候，比较只会在同层级进行, 不会跨层级比较。

## 8.5 vue中key的作用

key 是给每一个 vnode 的唯一 id,依靠 key,我们的 diff 操作可以更准确、更快速 (对于简单列表页渲染来说 diff 节点也更快,但会产生一些隐藏的副作用,比如可能不会产生过渡效果,或者在某些节点有绑定数据（表单）状态，会出现状态错位。)

diff 算法的过程中,先会进行新旧节点的首尾交叉对比,当无法匹配的时候会用新节点的 key 与旧节点进行比对,从而找到相应旧节点。

**更准确 :** 因为带 key 就不是就地复用了,在 sameNode 函数 a.key === b.key 对比中可以避免就地复用的情况。所以会更加准确,如果不加 key,会导致之前节点的状态被保留下来,会产生一系列的 bug。

**更快速 :** key 的唯一性可以被 Map 数据结构充分利用,相比于遍历查找的时间复杂度 O(n),Map 的时间复杂度仅仅为 O(1)

## 8.6 nextTick的原理

Vue 在更新 DOM 时是**异步**执行的。只要侦听到数据变化，Vue 将开启一个队列，并缓冲在同一事件循环中发生的所有数据变更。

如果同一个 watcher 被多次触发，只会被推入到队列中一次。这种在缓冲时去除重复数据对于避免不必要的计算和 DOM 操作是非常重要的。

然后，在下一个的事件循环“tick”中，Vue 刷新队列并执行实际 (已去重的) 工作。

Vue 在内部对异步队列尝试使用原生的 Promise.then、MutationObserver 和 setImmediate，如果执行环境不支持，则会采用 setTimeout(fn, 0) 代替。

**nextTick 方法的实现原理:**

1. vue 用异步队列的方式来控制 DOM 更新和 nextTick 回调先后执行
2. microtask 因为其高优先级特性，能确保队列中的微任务在一次事件循环前被执行完毕
3. 考虑兼容问题,vue 做了 microtask 向 macrotask 的降级方案。

## 8.7 Vue对数组的监听，如何对数组方法进行变异

Vue 通过原型拦截的方式重写了数组的 7 个方法,首先获取到这个数组的**ob**,也就是它的 Observer 对象,如果有新的值,就调用 observeArray 对新的值进行监听,然后手动调用 notify,通知 render watcher,执行 update

## 8.8 vue组件中data为什么必须是函数

因为组件是可以复用的,JS 里对象是引用关系,如果组件 data 是一个对象,那么子组件中的 data 属性值会互相污染,产生副作用。

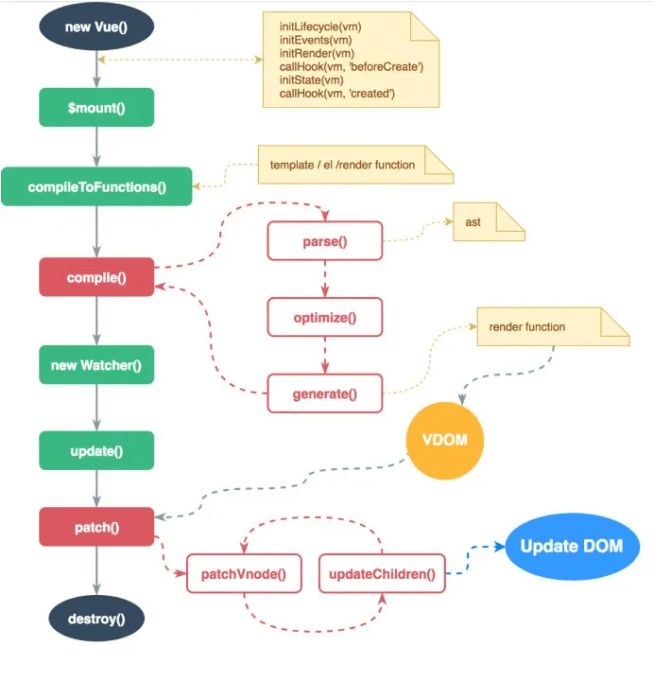
所以一个组件的 data 选项必须是一个函数,因此每个实例可以维护一份被返回对象的独立的拷贝。new Vue 的实例是不会被复用的,因此不存在以上问题。

## 8.9 谈谈vue的事件机制，手写$on,$off,$emit,$once

Vue 事件机制 本质上就是 一个 发布-订阅 模式的实现。

class Vue {  
 constructor() {  
 // 事件通道调度中心  
 this.\_events = Object.create(null);  
 }  
 $on(event, fn) {  
 if (Array.isArray(event)) {  
 event.map(item => {  
 this.$on(item, fn);  
 });  
 } else {  
 (this.\_events[event] || (this.\_events[event] = [])).push(fn);  
 }  
 return this;  
 }  
 $once(event, fn) {  
 function on() {  
 this.$off(event, on);  
 fn.apply(this, arguments);  
 }  
 on.fn = fn;  
 this.$on(event, on);  
 return this;  
 }  
 $off(event, fn) {  
 if (!arguments.length) {  
 this.\_events = Object.create(null);  
 return this;  
 }  
 if (Array.isArray(event)) {  
 event.map(item => {  
 this.$off(item, fn);  
 });  
 return this;  
 }  
 const cbs = this.\_events[event];  
 if (!cbs) {  
 return this;  
 }  
 if (!fn) {  
 this.\_events[event] = null;  
 return this;  
 }  
 let cb;  
 let i = cbs.length;  
 while (i--) {  
 cb = cbs[i];  
 if (cb === fn || cb.fn === fn) {  
 cbs.splice(i, 1);  
 break;  
 }  
 }  
 return this;  
 }  
 $emit(event) {  
 let cbs = this.\_events[event];  
 if (cbs) {  
 const args = [].slice.call(arguments, 1);  
 cbs.map(item => {  
 args ? item.apply(this, args) : item.call(this);  
 });  
 }  
 return this;  
 }  
}

## 8.10 Vue的渲染过程



1. 调用 compile 函数,生成 render 函数字符串 ,编译过程如下:
   1. parse 函数解析 template,生成 ast(抽象语法树)
   2. optimize 函数优化静态节点 (标记不需要每次都更新的内容,diff 算法会直接跳过静态节点,从而减少比较的过程,优化了 patch 的性能)
   3. generate 函数生成 render 函数字符串
2. 调用 new Watcher 函数,监听数据的变化,当数据发生变化时，Render 函数执行生成 vnode 对象
3. 调用 patch 方法,对比新旧 vnode 对象,通过 DOM diff 算法,添加、修改、删除真正的 DOM 元素

## 8.11 keep-alive 的实现原理和缓存策略 ？？

## 8.12 vm.$set()的实现原理

## 8.13 兄弟组件通信

### 8.13.1 props+$emit

<!--父组件-->  
<template>  
 <div id="app">  
 <p>  
 name:{{name}}  
 </p>  
 <Child1 :name="name" @Child1Click="handleParentClick"/>  
 <Child2 :name="name" />  
 </div>  
</template>

<!-- 子组件 -->  
<template>  
<div>  
 <p>  
 name:{{name}}  
 </p>  
 <button @click="handleChild1Click">按钮</button>  
 </div>  
</template>  
  
<script>  
 export default{  
 props:['name'],  
 methods:{  
 handleChild1Click(){  
 this.$emit('Child1Click','ABC');  
 }  
 }  
 }  
</script>

props+$emit,同时会改变父组件的数据

### 8.13.2 eventBus

//main.js  
  
import Vue from 'vue';  
import App from './App';  
  
export const eventBus = new Vue();  
new Vue({  
 render:h=>h(App),  
}).$mount('#app');

<!--Child1.vue-->  
<script>  
 import {eventBus} from '../main';  
 export default{  
 method:{  
 handleChild1Click(){  
 this.myName='ABC';  
 eventBus.$emit('Child1Click','ABC');  
 }  
 }  
 }  
</script>

<!--Child2.vue-->  
<script>  
 import {eventBus} from '../main';  
 export default{  
 created(){  
 eventBus.$on('Child1Click',(name)=>{  
 this.myName = name;  
 })  
 }  
 }  
</script>

eventBus实质上是一个vue实例，数据传输不会经过父组件

### 8.13.3 Vuex

//index.js  
import Vuex from 'vuex';  
Vue.use(Vuex);  
const store = new Vuex.Store({  
 state:{  
 name:"AB",  
 },  
 mutations:{  
 changeName(state){  
 state.name='ABC';  
 }  
 }  
})  
  
export default store;

//main.js  
import store from '../store';  
  
new Vue({  
 render:h=>h(App),  
 store  
}).$mount('#app');

<!-- Child1.vue -->  
<template>  
<div>  
 <p>  
 name:{{store.state.name}}  
 </p>  
 <button @click="handleChild1Click">按钮</button>  
 </div>  
</template>  
  
<script>  
 export default{  
 props:['name'],  
 methods:{  
 handleChild1Click(){  
 this.$store.commit('changeName');  
 }  
 }  
 }  
</script>

# 9. Webpack

## 9.1 核心概念

1. entry：一个可执行模块或者库的入口。
2. chunk：多个文件组成一个代码块。可以将可执行的模块和他所依赖的模块组合成一个chunk，这是打包。
3. loader：文件转换器。例如把es6转为es5，scss转为css等
4. plugin：扩展webpack功能的插件。在webpack构建的生命周期节点上加入扩展hook，添加功能。

## 9.2 webpack构建流程（原理）

从启动构建到输出结果一系列过程：

1. 初始化参数：解析webpack配置参数，合并shell传入和webpack.config.js文件配置的参数，形成最后的配置结果。
2. 开始编译：上一步得到的参数初始化compiler对象，注册所有配置的插件，插件监听webpack构建生命周期的事件节点，做出相应的反应，执行对象的 run 方法开始执行编译。
3. 确定入口：从配置的entry入口，开始解析文件构建AST语法树，找出依赖，递归下去。
4. 编译模块：递归中根据文件类型和loader配置，调用所有配置的loader对文件进行转换，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理。
5. 完成模块编译并输出：递归完事后，得到每个文件结果，包含每个模块以及他们之间的依赖关系，根据entry配置生成代码块chunk。
6. 输出完成：输出所有的chunk到文件系统。

注意：在构建生命周期中有一系列插件在做合适的时机做合适事情，比如UglifyPlugin会在loader转换递归完对结果使用UglifyJs压缩覆盖之前的结果。

# 10. Echarts