**COOKIE和SESSION**

**会话（Session）**跟踪是Web程序中常用的技术，用来**跟踪用户的整个会话**。常用的会话跟踪技术是Cookie与Session。**Cookie通过在客户端记录信息确定用户身份**，**Session通过在服务器端记录信息确定用户身份**。

　　由于HTTP是一种无状态的协议，服务器单从网络连接上无从知道客户身份。怎么办呢？就给客户端们颁发一个通行证吧，每人一个，无论谁访问都必须携带自己通行证。这样服务器就能从通行证上确认客户身份了。这就是Cookie的工作原理。

Cookie实际上是一小段的文本信息。客户端请求服务器，如果服务器需要记录该用户状态，就使用response向客户端浏览器颁发一个Cookie。客户端浏览器会把Cookie保存起来。当浏览器再请求该网站时，浏览器把请求的网址连同该Cookie一同提交给服务器。服务器检查该Cookie，以此来辨认用户状态。服务器还可以根据需要修改Cookie的内容。

Session是另一种记录客户状态的机制，不同的是Cookie保存在客户端浏览器中，而Session保存在服务器上。客户端浏览器访问服务器的时候，服务器把客户端信息以某种形式记录在服务器上。这就是Session。客户端浏览器再次访问时只需要从该Session中查找该客户的状态就可以了。

　　如果说Cookie机制是通过检查客户身上的“通行证”来确定客户身份的话，那么Session机制就是通过检查服务器上的“客户明细表”来确认客户身份。Session相当于程序在服务器上建立的一份客户档案，客户来访的时候只需要查询客户档案表就可以了。

**cookie和session的区别**

1. cookie数据存放在客户的浏览器上，session数据放在服务器上.

2、cookie不是很安全，别人可以分析存放在本地的COOKIE并进行COOKIE欺骗考虑到安全应当使用session。

3、设置cookie时间可以使cookie过期。但是使用session-destory（），我们将会销毁会话。  
4、session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能考虑到减轻服务器性能方面，应当使用cookie。  
5、单个cookie保存的数据不能超过4K，很多浏览器都限制一个站点最多保存20个cookie。(Session对象没有对存储的数据量的限制，其中可以保存更为复杂的数据类型)

**注意:**

      session很容易失效,用户体验很差;

      虽然cookie不安全,但是可以加密 ;

      cookie也分为永久和暂时存在的;

      浏览器 有禁止cookie功能 ,但一般用户都不会设置;

     一定要设置失效时间,要不然浏览器关闭就消失了;

两者最大的区别在于生存周期，一个是IE启动到IE关闭.(浏览器页面一关 ,session就消失了)，一个是预先设置的生存周期，或永久的保存于本地的文件。(cookie)

**进程和线程的区别**

**进程：**一个在内存中运行的应用程序。每个进程都有自己独立的一块内存空间，一个进程可以有多个线程，比如在Windows系统中，一个运行的xx.exe就是一个进程。

**线程：**进程中的一个执行任务（控制单元），负责当前进程中程序的执行。一个进程至少有一个线程，一个进程可以运行多个线程，多个线程可共享数据。

与进程不同的是同类的多个线程共享进程的堆和方法区资源，但每个线程有自己的程序计数器、虚拟机栈和本地方法栈，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。

**区别：**

**根本区别：**进程是操作系统资源分配的基本单位，而线程是处理器任务调度和执行的基本单位

**资源开销：**每个进程都有独立的代码和数据空间（程序上下文），程序之间的切换会有较大的开销；线程可以看做轻量级的进程，同一类线程共享代码和数据空间，每个线程都有自己独立的运行栈和程序计数器（PC），线程之间切换的开销小。

**包含关系：**如果一个进程内有多个线程，则执行过程不是一条线的，而是多条线（线程）共同完成的；线程是进程的一部分，所以线程也被称为轻权进程或者轻量级进程。

**内存分配：**同一进程的线程共享本进程的地址空间和资源，而进程之间的地址空间和资源是相互独立的

**影响关系：**一个进程崩溃后，在保护模式下不会对其他进程产生影响，但是一个线程崩溃整个进程都死掉。所以多进程要比多线程健壮。

**执行过程：**每个独立的进程有程序运行的入口、顺序执行序列和程序出口。但是线程不能独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制，两者均可并发执行

**TCP和UDP的区别**

**TCP与UDP区别总结**

1、TCP面向连接（如打电话要先拨号建立连接）;UDP是无连接的，即发送数据之前不需要建立连接

2、TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达;UDP尽最大努力交付，即不保证可靠交付。TCP通过校验和，重传控制，序号标识，滑动窗口、确认应答实现可靠传输。如丢包时的重发控制，还可以对次序乱掉的分包进行顺序控制。

3、UDP具有较好的实时性，工作效率比TCP高，适用于对高速传输和实时性有较高的通信或广播通信。

4.每一条TCP连接只能是点到点的；UDP支持一对一、一对多、多对一和多对多的交互通信。

5、TCP对系统资源要求较多，UDP对系统资源要求较少。

**为什么UDP有时比TCP更有优势?**

UDP以其简单、传输快的优势，在越来越多场景下取代了TCP,如实时游戏。

（1）网速的提升给UDP的稳定性提供可靠网络保障，丢包率很低，如果使用应用层重传，能够确保传输的可靠性。

（2）TCP为了实现网络通信的可靠性，使用了复杂的拥塞控制算法，建立了繁琐的握手过程，由于TCP内置的系统协议栈中，极难对其进行改进。

采用TCP，一旦发生丢包，TCP会将后续的包缓存起来，等前面的包重传并接收到后再继续发送，延时会越来越大，基于UDP对实时性要求较为严格的情况下，采用自定义重传机制，能够把丢包产生的延迟降到最低，尽量减少网络问题对游戏性造成影响。

# **[TCP字节流和UDP数据报区别](https://www.cnblogs.com/hoohack/p/4772365.html)**

两者的区别在于TCP接收的是一堆数据，而每次取多少由主机决定;而UDP发的是数据报，客户发送多少就接收多少。

拥有这些区别的原因是由于TCP和UDP的特性不同而决定的。TCP是面向连接的，也就是说，在连接持续的过程中，socket中收到的数据都是由同一台主机发出的，因此，知道保证数据是有序的到达就行了，至于每次读取多少数据自己看着办。 而UDP是无连接的协议，也就是说，只要知道接收端的IP和端口，且网络是可达的，任何主机都可以向接收端发送数据。这时候，如果一次能读取超过一个报文的数据，则会乱套。比如，主机A向发送了报文P1，主机B发送了报文P2，如果能够读取超过一个报文的数据，那么就会将P1和P2的数据合并在了一起，这样的数据是没有意义的。

**作用域链**

函数作用域中的每个对象被称为一个可变对象，每个可变对象都以“键值对”的形式存在。

当一个函数创建后，它的作用域链会被创建此函数的作用域中可访问的数据对象所填充

执行函数时会创建一个称为执行环境（execution context，也称执行上下文）的内部对象。

一个执行环境定义了一个函数执行时的环境，函数每次执行时对应的执行环境都是独一无二的，所以多次调用同一个函数就会导致创建多个执行环境。当函数执行完毕，执行环境就被销毁

每个执行环境都有自己的作用域链，用于解析标识符。当执行环境被创建时，它的作用域链初始化为当前运行函数的[[Scope]]属性中的对象。这些值按照他们出现在函数中的顺序，被复制到执行环境的作用域链中。这个过程一旦完成，一个被称为“活动对象”的新对象就为执行环境创建好了

**[call和apply和bind的区别](https://www.cnblogs.com/lhl66/p/7502794.html)**

在 javascript 中，call 和 apply 都是为了改变某个函数运行时的上下文（context）而存在的，换句话说，就是为了改变函数体内部 this 的指向。

JavaScript 的一大特点是，函数存在「定义时上下文」和「运行时上下文」以及「上下文是可以改变的」。

**apply():** 将函数作为指定对象的方法来调用，传递给它的是指定的参数数组

**bind():** 主要作用：将函数绑定到一个对象，返回一个新函数，通过可选的指定参数，作为指定对象的方法调用该方法

MDN的解释是：bind()方法会创建一个新函数，称为绑定函数，当调用这个绑定函数时，绑定函数会以创建它时传入 bind()方法的第一个参数作为 this，传入 bind() 方法的第二个以及以后的参数加上绑定函数运行时本身的参数按照顺序作为原函数的参数来调用原函数

**call():**类似apply，将函数作为指定对象的方法来调用，传递给它的是指定参数

apply 、 call 、bind 三者都是用来改变函数的this对象的指向的；

apply 、 call 、bind 三者第一个参数都是this要指向的对象，也就是想指定的上下文；

apply 、 call 、bind 三者都可以利用后续参数传参；

bind 是返回对应函数，便于稍后调用；apply 、call 则是立即调用 。

**this的指向**

this并不指向函数自身

需要明确的是,this在任何情况下都不指向函数的词法作用域

this实际上是在函数被调用时发生的绑定，它指向什么完全取决于函数在哪里被调用

当一个函数被调用时，会创建一个活动记录（有时候也称为执行上下文）。这个记录会包含函数在哪里被调用（调用栈）、函数的调用方式、传入的参数等信息。this就是这个记录的一个属性，会在函数执行的过程中用到

寻找调用位置最重要的是要分析调用栈（就是为了到达当前执行位置所调用的所有函数）。调用位置就在当前正在执行的函数的前一个调用

this的四种绑定规则

1.默认绑定：无法应用其他规则时的默认规则

function foo() {

console.log(this.a);

}

var a = 2;

foo(); //2

foo()是直接使用不带任何修饰的函数引用进行调用的，因此只能使用默认绑定（此时this指向全局对象，在严格模式下会指向undefined）

2.隐式绑定：考虑调用位置是否有上下文对象，或者说是否被某个对象拥有或者包含（引用）

在分析隐式绑定时，我们必须在一个对象内部包含一个指向函数的属性，并通过这个属性间接引用函数，从而把this间接（隐式）绑定到这个对象上

function foo() {

console.log(this.a);

}

var obj2 = {

a:42,

foo:foo

};

var obj1 = {

a:2,

obj2:obj2

};

obj1.obj2.foo(); //42

当foo()被调用时，它的前面加上了对obj2的引用。当函数引用有上下文对象时，隐式绑定规则会把函数调用中的this绑定到这个上下文对象

对象属性引用链中只有上一层或者说最后一层在调用位置中起作用

一个最常见的this绑定问题就是被隐式绑定的函数会丢失绑定对象，也就是说它会应用默认绑定，从而把this绑定到全局对象（严格模式下是undefined）

function foo() {

console.log(this.a);

}

var obj = {

a:2,

foo:foo

};

var bar = obj.foo; //函数别名

var a = “test”;

bar(); //test

虽然bar是obj.foo的一个引用，但实际上，它引用的是foo函数本身

更常见的还有在传入回调函数时丢失this绑定的情况

3.显式绑定：使用call()/apply()方法，它们的第一个参数是一个对象，在调用函数时将其绑定到this。因为可以直接指定this的绑定对象，因而称为显式绑定。

如果传入了一个原始值（字符串、布尔或数字）来当做this的绑定对象，这个原始值会被转换成它的对象形式（new String()···）,这通常被称为装箱

硬绑定：

function foo() {

console.log(this.a);

}

var obj = {

a:2

};

var bar = function() {

foo.call(obj);

};

bar(); //2

setTimeout(bar, 100); //2

//硬绑定的bar不可能再修改它的this

bar.call(window); //2

在函数bar()内部手动调用了foo.call(obj)，强制把foo的this绑定到了obj。无论之后如何调用函数bar，它总会手动在obj上调用foo。这种绑定时一种显式的强制绑定，称为硬绑定。

4.new绑定：

function foo(a) {

this.a = a;

}

var bar = new foo(2);

console.log(bar.a); //2

使用new来调用foo()时，会构造一个新对象并把它绑定到foo()调用中的this上。

使用new来调用函数，或者说发生构造函数调用时，会自动执行下面操作：

1. 创建（或者说构造）一个全新的对象
2. 这个新对象会被执行[[Prototype]]连接
3. 这个新对象会绑定到函数调用的this
4. 如果函数没有返回其他对象 ，那么new表达式中的函数调用会自动返回这个新对象

优先级：new绑定>显示绑定>隐式绑定>默认绑定

归纳：判断this

1.函数是否在new中调用（new绑定）？如果是的话this绑定的是新创建的对象

var bar = new foo();

2.函数是否通过call、apply（显式绑定）或者硬绑定调用？如果是的话，this绑定的是指定的对象

var bar = foo.call(obj2);

3.函数是否在某个上下文对象中调用（隐式绑定）？如果是的话，this绑定的是那个上下文对象

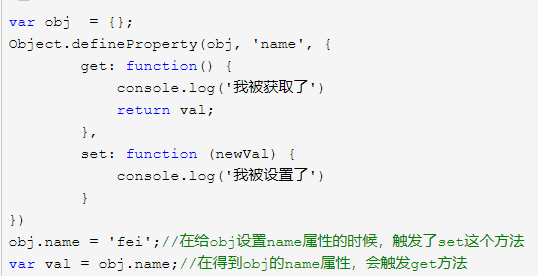
var bar = obj1.foo();

4.如果都不是的话，使用默认绑定。this绑定到全局对象，在严格模式下绑定到undefined

var bar = foo()

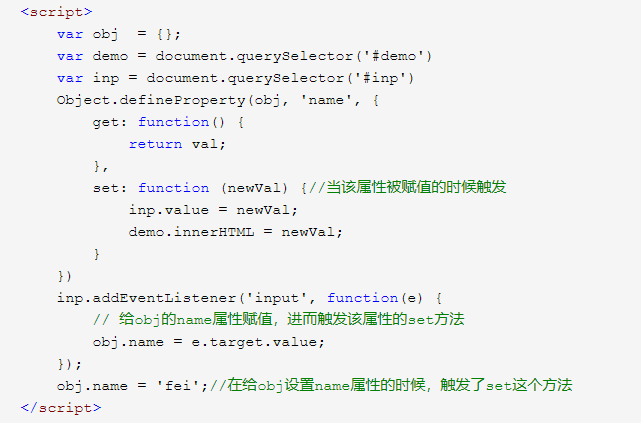
**vue双向绑定原理**

vue.js 则是采用数据劫持结合发布者-订阅者模式的方式，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，触发相应的监听回调。我们先来看Object.defineProperty()这个方法：



已经了解到vue是通过数据劫持的方式来做数据绑定的，其中最核心的方法便是通过Object.defineProperty()来实现对属性的劫持，那么在设置或者获取的时候我们就可以在get或者set方法里假如其他的触发函数，达到监听数据变动的目的，无疑这个方法是本文中最重要、最基础的内容之一。

最简单的双向绑定



## [HTTP缓存机制](https://www.cnblogs.com/ranyonsue/p/8918908.html)

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。这里我们只讨论 HTTP 缓存相关内容。

在具体了解 HTTP 缓存之前先来明确几个术语：

* 缓存命中率：从缓存中得到数据的请求数与所有请求数的比率。理想状态是越高越好。
* 过期内容：超过设置的有效时间，被标记为“陈旧”的内容。通常过期内容不能用于回复客户端的请求，必须重新向源服务器请求新的内容或者验证缓存的内容是否仍然准备。
* 验证：验证缓存中的过期内容是否仍然有效，验证通过的话刷新过期时间。
* 失效：失效就是把内容从缓存中移除。当内容发生改变时就必须移除失效的内容。

浏览器缓存主要是 HTTP 协议定义的缓存机制。HTML meta 标签，例如

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

含义是让浏览器不缓存当前页面。但是代理服务器不解析 HTML 内容，一般应用广泛的是用 HTTP 头信息控制缓存。

**浏览器缓存分类**

浏览器缓存分为强缓存和协商缓存，浏览器加载一个页面的简单流程如下：

1. 浏览器先根据这个资源的http头信息来判断是否命中强缓存。如果命中则直接加在缓存中的资源，并不会将请求发送到服务器。
2. 如果未命中强缓存，则浏览器会将资源加载请求发送到服务器。服务器来判断浏览器本地缓存是否失效。若可以使用，则服务器并不会返回资源信息，浏览器继续从缓存加载资源。
3. 如果未命中协商缓存，则服务器会将完整的资源返回给浏览器，浏览器加载新资源，并更新缓存。

**强缓存**

命中强缓存时，浏览器并不会将请求发送给服务器。在Chrome的开发者工具中看到http的返回码是200，但是在Size列会显示为(from cache)。

# **协商缓存**

若未命中强缓存，则浏览器会将请求发送至服务器。服务器根据http头信息中的Last-Modify/If-Modify-Since或Etag/If-None-Match来判断是否命中协商缓存。如果命中，则http返回码为304，浏览器从缓存中加载资源。

# **HTTP1.0和HTTP1.1和HTTP2.0的区别**

## **1 HTTP1.0和HTTP1.1的区别**

### **1.1 长连接(Persistent Connection)**

       HTTP1.1支持长连接和请求的流水线处理，在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟，在HTTP1.1中默认开启长连接keep-alive，一定程度上弥补了HTTP1.0每次请求都要创建连接的缺点。HTTP1.0需要使用keep-alive参数来告知服务器端要建立一个长连接。

### **1.2 节约带宽**

       HTTP1.0中存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能。HTTP1.1支持只发送header信息（不带任何body信息），如果服务器认为客户端有权限请求服务器，则返回100，客户端接收到100才开始把请求body发送到服务器；如果返回401，客户端就可以不用发送请求body了节约了带宽。

### **1.3 HOST域**

       在HTTP1.0中认为每台服务器都绑定一个唯一的IP地址，因此，请求消息中的URL并没有传递主机名（hostname），HTTP1.0没有host域。随着虚拟主机技术的发展，在一台物理服务器上可以存在多个虚拟主机（Multi-homed Web Servers），并且它们共享一个IP地址。HTTP1.1的请求消息和响应消息都支持host域，且请求消息中如果没有host域会报告一个错误（400 Bad Request）。

### **1.4**缓存处理****

在HTTP1.0中主要使用header里的If-Modified-Since,Expires来做为缓存判断的标准，HTTP1.1则引入了更多的缓存控制策略例如Entity tag，If-Unmodified-Since, If-Match, If-None-Match等更多可供选择的缓存头来控制缓存策略。

### **1.5**错误通知的管理****

       在HTTP1.1中新增了24个错误状态响应码，如409（Conflict）表示请求的资源与资源的当前状态发生冲突；410（Gone）表示服务器上的某个资源被永久性的删除。

## **2 HTTP1.1和HTTP2.0的区别**

### **2.1 多路复用**

       HTTP2.0使用了多路复用的技术，做到同一个连接并发处理多个请求，而且并发请求的数量比HTTP1.1大了好几个数量级。HTTP1.1也可以多建立几个TCP连接，来支持处理更多并发的请求，但是创建TCP连接本身也是有开销的。

### **2.2 头部数据压缩**

       在HTTP1.1中，HTTP请求和响应都是由状态行、请求/响应头部、消息主体三部分组成。一般而言，消息主体都会经过gzip压缩，或者本身传输的就是压缩过后的二进制文件，但状态行和头部却没有经过任何压缩，直接以纯文本传输。随着Web功能越来越复杂，每个页面产生的请求数也越来越多，导致消耗在头部的流量越来越多，尤其是每次都要传输UserAgent、Cookie这类不会频繁变动的内容，完全是一种浪费。

       HTTP1.1不支持header数据的压缩，HTTP2.0使用HPACK算法对header的数据进行压缩，这样数据体积小了，在网络上传输就会更快。

### **2.3 服务器推送**

       服务端推送是一种在客户端请求之前发送数据的机制。网页使用了许多资源：HTML、样式表、脚本、图片等等。在HTTP1.1中这些资源每一个都必须明确地请求。这是一个很慢的过程。浏览器从获取HTML开始，然后在它解析和评估页面的时候，增量地获取更多的资源。因为服务器必须等待浏览器做每一个请求，网络经常是空闲的和未充分使用的。

       为了改善延迟，HTTP2.0引入了server push，它允许服务端推送资源给浏览器，在浏览器明确地请求之前，免得客户端再次创建连接发送请求到服务器端获取。这样客户端可以直接从本地加载这些资源，不用再通过网络。

# [link和@import的区别](https://www.cnblogs.com/passkey/p/10141553.html)

本质上，这两种方式都是为了加载css文件，但还是存在细微的差别。

**1.从属关系区别**  
@import是 CSS 提供的语法规则，只有导入样式表的作用；link是HTML提供的标签，不仅可以加载 CSS 文件，还可以定义 RSS、rel 连接属性等。

**2.加载顺序区别**  
加载页面时，link标签引入的 CSS 被同时加载；@import引入的 CSS 将在页面加载完毕后被加载。

**3.兼容性区别**  
@import是 CSS2.1 才有的语法，故只可在 IE5+ 才能识别；link标签作为 HTML 元素，不存在兼容性问题。

**4.DOM可控性区别**  
可以通过 JS 操作 DOM ，插入link标签来改变样式；由于 DOM 方法是基于文档的，无法使用@import的方式插入样式。

**5.权重区别(该项有争议，下文将详解)**  
link引入的样式权重大于@import引入的样式。

# **HTTP常见状态码（14种）**

****2xx （3种）****

****200 OK：****表示从客户端发送给服务器的请求被正常处理并返回；

****204 No Content：****表示客户端发送给客户端的请求得到了成功处理，但在返回的响应报文中不含实体的主体部分（没有资源可以返回）；

****206 Patial Content：****表示客户端进行了范围请求，并且服务器成功执行了这部分的GET请求，响应报文中包含由Content-Range指定范围的实体内容。

****3xx （5种）****

****301 Moved Permanently：****永久性重定向，表示请求的资源被分配了新的URL，之后应使用更改的URL；

****302 Found：****临时性重定向，表示请求的资源被分配了新的URL，希望本次访问使用新的URL；

       301与302的区别：前者是永久移动，后者是临时移动（之后可能还会更改URL）

****303 See Other：****表示请求的资源被分配了新的URL，应使用GET方法定向获取请求的资源；

      302与303的区别：后者明确表示客户端应当采用GET方式获取资源

****304 Not Modified：****表示客户端发送附带条件（是指采用GET方法的请求报文中包含if-Match、If-Modified-Since、If-None-Match、If-Range、If-Unmodified-Since中任一首部）的请求时，服务器端允许访问资源，但是请求为满足条件的情况下返回改状态码；

****307 Temporary Redirect：****临时重定向，与303有着相同的含义，307会遵照浏览器标准不会从POST变成GET；（不同浏览器可能会出现不同的情况）；

****4xx （4种）****

****400 Bad Request：****表示请求报文中存在语法错误；

****401 Unauthorized：****未经许可，需要通过HTTP认证；

****403 Forbidden：****服务器拒绝该次访问（访问权限出现问题）

****404 Not Found：****表示服务器上无法找到请求的资源，除此之外，也可以在服务器拒绝请求但不想给拒绝原因时使用；

****5xx （2种）****

****500 Inter Server Error：****表示服务器在执行请求时发生了错误，也有可能是web应用存在的bug或某些临时的错误时；

****503 Server Unavailable：****表示服务器暂时处于超负载或正在进行停机维护，无法处理请求；

CSS样式优先级

  行内样式 > id选择器 >  class选择器 > 标签选择器



DNS工作过程（简易）

1. 客户端发请求给本地域名服务器
2. 本地域名服务器查询本地缓存，如果有就直接返回
3. 请求发送到根域名服务器，返回给本地域名服务器一个主域名服务器地址（？）
4. 向域名服务器查询缓存，没有继续查询下级域名服务器地址（？？？）
5. 本地域名服务器返回结果给客户端，并保存到缓存

原型链：

以下三点需要谨记

1.每个对象都具有一个名为\_\_proto\_\_的属性；

2.每个构造函数（构造函数标准为大写开头，如Function()，Object()等等JS中自带的构造函数，以及自己创建的）都具有一个名为prototype的方法（注意：既然是方法，那么就是一个对象（JS中函数同样是对象），所以prototype同样带有\_\_proto\_\_属性）；

3.每个对象的\_\_proto\_\_属性指向自身构造函数的prototype；



# [对防抖和节流的理解及其应用场景](https://www.cnblogs.com/yuwenjing0727/p/13578220.html)

在开发中，我们常常会去监听滚动事件或者用户输入框验证事件，如果事件处理没有频率限制，就会加重浏览器的负担，影响用户的体验感，

因此，我们可以采取**防抖**（debounce）和**节流**（throttle）来处理，减少调用事件的频率，达到较好的用户体验。

**防抖（debounce）：**

在事件被触发n秒后再执行回调，如果在这n秒内又被触发，则重新计时，重新出发定时器。

**相当于就是多次执行，只执行一次。**

**节流（throttle）：**

规定在一个单位时间内，只能触发一次函数。如果这个单位时间内触发多次函数，只有一次生效。

总结，看完了防抖和节流的分别介绍，我们来看看他们的区别：

·函数防抖和函数节流都是防止某一时间频繁出发，但是这两兄弟之间的远离却不一样

**防抖**是将多次执行变为只执行一次，**节流**是将多次执行变为每隔一段时间执行。

### **结合应用场景**

　　防抖(debounce)

　　　　search搜索联想，用户在不断输入值时，用防抖来节约请求资源。

　　　　window触发resize的时候，不断的调整浏览器窗口大小会不断的触发这个事件，用防抖来让其只触发一次

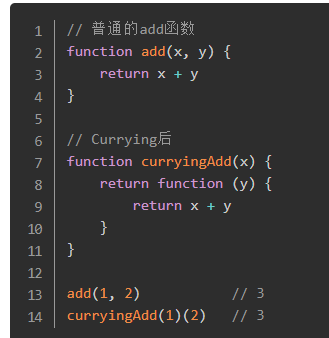
　　节流(throttle)

　　　　鼠标不断点击触发，mousedown(单位时间内只触发一次)

　　　　监听滚动事件，比如是否滑到底部自动加载更多，用throttle来判断

柯里化，英语：Currying(果然是满满的英译中的既视感)，是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数而且返回结果的新函数的技术。

用一句话解释就是，把一个多参数的函数转化为单参数函数的方法。



实际上就是把add函数的x，y两个参数变成了先用一个函数接收x然后返回一个函数去处理y参数。现在思路应该就比较清晰了，就是只传递给函数一部分参数来调用它，让它返回一个函数去处理剩下的参数。

# **script标签中defer和async属性的区别**

**async和defer的相同点：**

都是为了异步加载js，如果不加这两个属性，把script放在body结尾的话，js会等html先解析，才会解析js脚本，但是加了这两个属性，解析html的同时也会解析js脚本。虽然async和defer都是对js脚本异步加载，但是他俩还是有些许不同。

**async和defer的不同点：**

 主要的区别是执行的时机不同，defer会等整个页面都解析完毕之后再运行，延迟到浏览器遇到</html>标签再执行。

async如果解析完了就会执行，不能保证脚本的执行顺序，要看这个js的解析速度了，所以如果和别的脚本有依赖性或者需要操作dom最好不要用async。async一般用于百度统计、谷歌统计那些关联性不强的脚本。

1.defer和async在网络加载过程是一致的，都是异步执行的；

　　2.两者的区别在于脚本加载完成之后何时执行，可以看出defer更符合大多数场景对应用脚本加载和执行的要求；

　　3.如果存在多个有defer属性的脚本，那么它们是按照加载顺序执行脚本的；而对于async，它的加载和执行是紧紧挨着的，无论声明顺序如何，只要加载完成就立刻执行，它对于应用脚本用处不大，因为它完全不考虑依赖。

总结，我们应该尽量使用传统的方法来使用javascript，不要引入defer或者async来改变其执行的时机，防止非预期行为的产生，

# ****HTTP通信过程****

HTTP通信机制是在一次完整的HTTP通信过程中，Web浏览器与Web服务器之间将完成下列**7个步骤**：

### ****2.1 建立TCP连接****

在HTTP工作开始之前，Web浏览器首先要通过网络与Web服务器建立连接，该连接是通过TCP来完成的，该协议与IP协议共同构建Internet，即著名的TCP/IP协议族，因此Internet又被称作是TCP/IP网络。HTTP是比TCP更高层次的应用层协议，根据规则，只有低层协议建立之后才能进行更深层协议的连接，因此，首先要建立TCP连接，一般TCP连接的端口号是**80**。

### ****2.2 Web浏览器向Web服务器发送请求命令****

一旦建立了TCP连接，Web浏览器就会向Web服务器发送请求命令

例如：GET/sample/hello.jsp HTTP/1.1。

### ****2.3 Web浏览器发送请求头信息****

浏览器发送其请求命令之后，还要以头信息的形式向Web服务器发送一些别的信息，之后浏览器发送了一空白行来通知服务器，它已经结束了该头信息的发送。

### ****2.4 Web服务器应答****

客户机向服务器发出请求后，服务器会客户机回送应答，

HTTP/1.1 200 OK

应答的第一部分是协议的版本号和应答状态码。

### ****2.5 Web服务器发送应答头信息****

正如客户端会随同请求发送关于自身的信息一样，服务器也会随同应答向用户发送关于它自己的数据及被请求的文档。

### ****2.6 Web服务器向浏览器发送数据****

Web服务器向浏览器发送头信息后，它会发送一个空白行来表示头信息的发送到此为结束，接着，它就以Content-Type应答头信息所描述的格式发送用户所请求的实际数据。

### ****2.7 Web服务器关闭TCP连接****

一般情况下，一旦Web服务器向浏览器发送了请求数据，它就要关闭TCP连接，然后如果浏览器或者服务器在其头信息加入了这行代码

Connection:keep-alive

TCP连接在发送后将仍然保持打开状态，于是，浏览器可以继续通过相同的连接发送请求。保持连接节省了为每个请求建立新连接所需的时间，还节约了网络带宽。

# **http预请求options**

在有很多情况下,当我们在js里面调用一次ajax请求时,在浏览器那边却会查询到两次请求,第一次的Request Method参数是OPTIONS,还有一次就是我们真正的请求,比如get或是post请求方式

查阅相关的资料之后发现,这是浏览器对复杂跨域请求的一种处理方式,在真正发送请求之前,会先进行一次预请求,就是我们刚刚说到的参数为OPTIONS的第一次请求,他的作用是用于试探性的服务器响应是否正确,即是否能接受真正的请求,如果在options请求之后获取到的响应是拒绝性质的,例如500等http状态,那么它就会停止第二次的真正请求的访问

大致说明一下,有三种方式会导致这种现象:

1:请求的方法不是GET/HEAD/POST

2:POST请求的Content-Type并非application/x-www-form-urlencoded, multipart/form-data, 或text/plain

3:请求设置了自定义的header字段

比如我的我的Content-Type设置为“****application/json;charset=utf-8****”并且自定义了header选项导致了这种情况。

# **[vue和react的diff算法的区别](https://www.cnblogs.com/mengff/p/12828825.html)**

vue和react的diff算法，都是忽略跨级比较，只做同级比较。vue diff时调动patch函数，参数是vnode和oldVnode，分别代表新旧节点。

1. vue比对节点，当节点元素类型相同，但是className不同，认为是不同类型元素，删除重建，而react会认为是同类型节点，只是修改节点属性

2. vue的列表比对，采用从两端到中间的比对方式，而react则采用从左到右依次比对的方式。当一个集合，只是把最后一个节点移动到了第一个，react会把前面的节点依次移动，而vue只会把最后一个节点移动到第一个。总体上，vue的对比方式更高效。

# **[js中new一个对象的过程](https://www.cnblogs.com/kevin2chen/p/6418327.html)**

使用new关键字调用函数（**new** ClassA**(…)**）的具体步骤：

1. 创建空对象；  
　　var obj = {};

2. 设置新对象的constructor属性为构造函数的名称，**设置新对象的\_\_proto\_\_属性指向构造函数的prototype对象；**　　obj.\_\_proto\_\_ = ClassA.prototype;

3. 使用新对象调用函数，函数中的this被指向新实例对象：  
　　ClassA.call(obj);　　//{}.构造函数();

4. 将初始化完毕的新对象地址，保存到等号左边的变量中

注意：若构造函数中返回this或返回值是基本类型（number、string、boolean、null、undefined）的值，则返回新实例对象；若返回值是引用类型的值，则实际返回值为这个引用类型。

# **display:none、visibility:hidden和opacity:0之间的区别**

****一、空间占据****

display:none隐藏后不占据额外空间，它会产生回流和重绘，而visibility:hidden和opacity:0元素虽然隐藏了，但它们仍然占据着空间，它们俩只会引起页面重绘。

****二、子元素继承****

display:none不会被子元素继承，但是父元素都不在了，子元素自然也就不会显示了，皮之不存，毛之安附~~

visibility:hidden 会被子元素继承，可以通过设置子元素visibility:visible 使子元素显示出来

opacity: 0 也会被子元素继承，但是不能通过设置子元素opacity: 0使其重新显示

****三、事件绑定****

display:none 的元素都已经不再页面存在了，因此肯定也无法触发它上面绑定的事件；

visibility:hidden 元素上绑定的事件也无法触发；

opacity: 0元素上面绑定的事件是可以触发的。

****四、过渡动画****

transition对于display肯定是无效的，大家应该都知道；

transition对于visibility也是无效的；

transition对于opacity是有效，大家也是知道的:).

**DOM0事件和DOM2事件**

DOM0级就是通过事件绑定的形式dom元素只能有(绑定)一个事件处理函数，他的特点是同一个元素绑定相同事件， 后面函数会覆盖前面的

分为2个：一是在标签内写onclick事件

　　　　  二是在JS写onlicke=function（）{}函数

DOM2级是通过事件监听的形式绑定， dom元素可以有(绑定)多个事件处理函数

只有一个监听方法，有两个方法用来添加和移除事件处理程序：addEventListener()和removeEventListener()。

position 新增sticky

懒加载典型：用一个占位符占据图片的位置，等到视窗到达图片时瞬间加载目标图片

**继承方式**

**工厂模式**

function createPerson(name, age) {

let o = new Object();

o.name = name;

o.age = age;

return o;

}

let person1 = createPerson(‘jack’, 29);

**构造函数模式**

依照惯例，构造函数名称的首字母都是要大写的，非构造函数则以小写字母开头

function Person(name, age) {

this.name = name;

this.age = age;

this.sayName = function () {

console.log(this.name);

}

}

let person1 = new Person(‘jack’, 29);

与工厂模式的区别

·没有显式地创建对象

·属性和方法直接赋值给了this

·没有return

创建Person实例应使用new操作符，以这种方式调用构造函数会执行如下操作

1. 在内存中创建一个新对象
2. 这个新对象内部的[[Prototype]]特性被赋值为构造函数的prototype属性
3. 构造函数内部的this被赋值为这个新对象（即this指向新对象）
4. 执行构造函数内部的代码（给新对象添加属性）
5. 如果构造函数返回非空对象，则返回该对象；否则，返回刚创建的新对象

构造函数与普通函数唯一的区别就是调用方式不同。除此之外，构造函数也是函数，并没有把某个函数定义为构造函数的特殊语法。任何函数只要使用new操作符调用就是构造函数，而不使用new操作符调用的函数就是普通函数

构造函数的主要问题在于：其定义的方法会在每个实例上都创建一遍

**原型模式**

每个函数都会创建一个prototype属性，这个属性是一个对象，包含应该由特定引用类型的实例共享的属性和方法。实际上，这个对象就是通过调用构造函数创建的对象的原型。使用原型对象的好处是，在它上面定义的属性和方法可以被对象实例共享。原来在构造函数中直接赋给对象实例的值，可以直接赋值给他们的原型

function Person() {

Person.prototype.name = ‘jack’;

Person.prototype.age = 22;

Person.prototype.sayName = function() {

console.log(this.name);

}

}

let person1 = new Person();

使用这种原型模式定义属性和方法是由所有实例共享的，在不同实例访问到的都是相同的属性和方法

无论何时，只要创建一个函数，就会按照特定的规则为这个函数创建一个prototype属性（指向原型对象）。默认情况下，所有原型对象自动获得一个名为constructor的属性，指回与之关联的构造函数。在自定义构造函数时，原型对象默认只会获得constructor属性，其他的所有方法都继承自Object。每次调用构造函数创建一个新实例，这个实例的内部[[Protortpe]]指针就会被赋值为构造函数的原型对象。脚本中没有访问这个[[Prototype]]特性的标准方法

Person.prototype指向原型对象，而Person.prototype.contructor指回Person构造函数，两者循环引用

构造函数、原型对象和实例是3个完全不同的对象。实例与构造函数原型之间有直接的联系，但实例与构造函数之间没有。

这三者的关系：每个构造函数都有一个原型对象，原型有一个属性指回构造函数，而实例有一个内部指针指向原型

JS的Object类型有一个方法叫Object.getPrototypeOf()，返回参数的内部特性[[Prototype]]的值，可以方便地取得一个对象的原型

有两种方式使用in操作符：单独使用和在for-in循环中使用。在单独使用时，in操作符会在可以通过对象访问指定属性时返回true，无论该属性是在实例上还是在原型上

在for-in循环中使用in操作符时，可以通过对象访问且可以被枚举的属性都会返回，包括实例属性和原型属性。遮蔽原型中不可枚举（[[Enumerable]]特性被设置为false）属性的实例属性也会在for-in循环中返回，因为默认情况下开发者定义的属性都是可枚举的

要获得对象上所有可枚举的实例属性，可以使用Object.keys()方法。这个方法接收一个对象作为参数，返回包含该对象所有可枚举属性名称的字符串数组

ES2017新增了两个静态方法，用于将对象内容转换为序列化：Object.values()和Object.entries()。其中非字符串属性会被转换为字符串输出，且这两个方法执行对象的浅复制，符号属性会被忽略

JS把原型链定义为主要继承方式，其基本思想就是通过原型链继承多个引用类型的属性和方法。

原型链：如果实例的原型是另一个类型的实例，这意味着这个原型本身有一个内部指针指向另一个原型，相应地另一个原型也有一个指针指向另一个构造函数。这样在实例和原型之间构造了一条原型链，这就是原型链的基本构想

原型和实例的关系可以通过两种方式来确定。第一种是使用instanceof操作符，如果一个实例的原型链中出现过相应地构造函数，则instanceof返回true

第二种方式是使用isPrototypeOf()方法，原型链中的每个原型都可以调用这个方法，只要原型链中包含这个原型，这个方法就返回true。

**组合继承**

基本的思路是使用原型链继承原型上的属性和方法，而通过盗用构造函数继承实例属性，这样既可以把方法定义在原型上以重现重用，又可以让每个实例都有自己的属性

function SuperType(name) {

this.name = name;

this.colors = [“red”, “blue”];

}

SuperType.prototype.sayName = function() {

console.log(this.name);

}

function SubType(name, age) {

//继承属性

SuperType.call(this, name);

this.age = age;

}

//继承方法

SubType.prototype = new SuperType();

组合继承弥补了原型链和盗用构造函数的不足，是JS中使用最多的继承模式。而且组合继承也保留instanceof操作符和isPrototypeOf()方法识别合成对象的能力

**原型式继承**

出发点是即使不自定义类型也可以通过原型实现对象之间的信息共享

function object(o) {

function F() {}

1. prototype = o;

return new F();

}

本质上，object()是对传入的对象执行了一次浅复制

适用情况：你有一个对象，想在它的基础上再创建新对象。你需要把这个对象先传给object()，然后再对返回的对象进行适当修改

ES5通过增加Object.create()方法将原型式继承的概念规范化了。这个方法接收两个参数：作为新对象原型的对象，以及给新对象定义额外属性的对象（第二个可选）。在只有一个参数时，Object.create()与这里的object()方法效果相同

原型式继承非常适合不需要单独创建构造函数但仍然需要在对象间共享信息的场合。但要记住，属性中包含的引用值始终会在相关对象间共享，跟使用原型模式是一样的

**寄生式继承**

背后的思路类似于寄生构造函数和工厂模式：创建一个实现继承的函数，以某种方式增强对象，然后返回这个对象

function createAnother(original) {

let clone = object(original); //通过调用函数创建一个新对象

clone.sayHi = function() { //以某种方式增强这个对象

console.log(“hi”);

};

return clone;

}

寄生式继承同样适合主要关注对象，而不在乎类型和构造函数的场景。object()函数不是寄生式继承所必须的，任何返回新对象的函数都可以在这里使用

**寄生式组合继承**

组合继承也存在效率问题，最主要的效率问题是父类构造函数始终会被调用两次：一次是在创建子类原型时调用，另一次是在子类构造函数中调用

寄生式组合继承通过盗用构造函数继承属性，但是用混合式原型链继承方法。基本思路是不通过调用父类构造函数给子类原型赋值，而是取得父类原型的一个副本。说到底就是使用寄生式继承来继承父类原型，然后将返回的新对象赋值给子类原型

function inheritPrototype(subType, superType) {

let prototype = object(superType.prototype); //创建对象

prototype.constructor = subType; //增强对象

subType.prototype = prototype; //赋值对象

}

寄生式组合继承可以算是引用类型继承的最佳模式

