深拷贝与浅拷贝，

浅拷贝指单层拷贝（并不单指赋值 “=”）

深拷贝：

function deepClone(*source*, *target*) {

for (let key in source) {

if (!source.hasOwnProperty(key)) {

if (source[key] instanceof *Object*) {

if (*Array*.isArray(source[key])) {

target[key] = []

} else {

target[key] = {}

}

deepClone(source[key], target[key])

} else if (source[key] != undefined) {

target[key] = source[key]

}

}

}

}

基本数据类型比较是否相等通过值比较（基本类型的值存放在栈内存中，赋值则在栈中开辟新的空间存放该值，互不干扰）

引用数据类型是否相等通过比较地址所指向的数据（引用类型存放在堆内存中，地址存放在栈内存中，栈内存中的地址指向堆内存的数据）

Bind()方法会创建一个新函数，并不会立即执行，需要立即执行需要在末尾加()

New发生过程：创建一个新的空 对象，将该对象的原型指向构造函数的原型，修改该对象的this指针，有返回值则返回对应值没有则返回改对象

判断变量的类型可以使用instanceof

区分Array和Object可以使用Array.isArray(target)

函数的调用方法：（非箭头函数的this指向为最后的调用者，匿名函数没有函数名无法被其他变量调用，（类似与直接在全局环境中作为函数调用）所以匿名函数的this指向为window，通过函数表达式调用匿名函数）

作为函数直接调用

函数作为方法被调用

作为构造函数调用

作为函数方法在函数里被调用（在js strict环境下，函数的第一个参数将作为this的值，即使这个参数不是对象，在js non strict模式下，如果第一个参数的值是null或者undefined 它将使用全局作对象替代）

箭头函数的this指向：箭头函数的this始终指向函数定义时的this，而非执行时，箭头函数中没有this绑定，必须通过查找作用域链来决定，如果箭头函数被废箭头函数包裹，则this绑定的是最近一层非箭头函数的this，否则，this为undefined

BFC (块级格式化上下文)

通过float overflow不为visible …触发

作用：BFC元素不被浮动元素所覆盖

使得内部元素的margin不重叠

父级设置为BFC会被子float元素所撑开高度

Vue：动态改变数组或者对象数据并不会触发视图的刷新，使用$set（target，key，value）改变可以触发视图刷新

组件命名：html不识别大小写，应该使用-连接，在其他地方都使用单词首字母大写的·命名规则，v-if和v-for不能作用于同一个元素，对循环数据源进行筛选应将数据源通过计算属性筛选后在用于v-for循环

区分对象和数组

Object.prototype.toString().call(target)===”[Object Array]” || ”[Object Object]”

Object.prototype.toString().call(target).slice(8,-1)===”Array” || ”Object”

跨域：产生原因ajax浏览器同源策略

解决方法：script发起jsonp，后台在头部设置安全域名，node.js中间件代理跨域，nginx反向代理，后台服务器上设置cors

Img的title和alt属性：图片正常输出时，不显示alt信息（包括鼠标移入），不正常输出时，显示alt信息，title属性鼠标移入时无论图片是否正常输出都会出现

ES5和ES6继承的区别：

ES5组合继承，缺点，父类实例的属性会被所有子类共享需要借用构造函数重新创建实例覆盖原型中的实例，导致调用两次父类的构造函数（一次父类实例化赋值给子类原型，一次子类构造函数首行调用覆盖父类实例属性：supertype.call(this)）

ES6寄生组合继承：将父类的原型拷贝一份副本，将副本（Object.create(source)）的construct属性指向子类构造函数，将副本赋值给子类原型，子类构造函数行首调用父类构造函数（ es6写法为super() ）

浏览器渲染：js下载也会中断DOM树的更新，影响首屏时间主要因素在于css解析，是否生成renderTree，页面解析流程：构建DOM，构建CSSOM，生成renderTree，layout，panting，浏览器可以显示不完整的renderTree，也就是说可以构建一部分就先显示一部分，所以首屏时间主要受css影响，css解析快则renderTree构建快则显示的快，script

标签的async和defer，async会并行下载并执行，，[MDN](https://link.juejin.im/?target=https%3A%2F%2Fdeveloper.mozilla.org%2Fzh-CN%2Fdocs%2FWeb%2FHTML%2FElement%2Fscript)对此的描述是用来通知浏览器该脚本将在文档完成解析后，触发 DOMContentLoaded 事件前执行，css不会阻塞dom解析，link标签会阻塞script标签，遇到script（没有异步标识符）会促使页面渲染更新

子资源加载

网站通常会使用图像，CSS和JavaScript等外部资源。 这些文件需要从网络或缓存中加载。 主线程可以在解析构建DOM时逐个请求它们，但为了加快速度，“预加载扫描器”会同时运行。 如果HTML文档中存在错误!未指定文件名。或之类的元素 ，则预加载扫描程序会检查由HTML解析器生成的标记，并向网络线程发送请求

Iterate.map(callback())callback默认传递三个参数（currentValue，index，包含该元素的数组对象） parseInt(string，radix)方法 如果string大于radix会返回NaN 多位有效字符串按位进行运算（不传radix，传0或者10 为10进制）

parseInt（1，0） return 1

praseInt（2，1） return NaN

parse Int（11,9）return 10

==和===：==会进行类型转换，当左右都是undefined或者null是为true 当为string和number时，会将string转化为number，当为基本类型时，会向number转换，当存在Object时 会向基本类型转化

DOMContentLoad和Load的区别：DOMContentLoad 初始html文档被解析完成时触发，无需等待样式表，图像和子框架的完成加载（必须等待所属的script之前的样式表加载解析完成才会触发），load用于检测一个完全加载的页面

事件流：stoppropagation 既可以组织事件冒泡 也可以阻止事件捕获，stopImmediatePropagation具有stoppropagation的作用而且可以阻止该注册节点的其余注册函数，如果一个节点同时注册了捕获和冒泡事件，则触发顺序按照注册顺序

CreateDocumentFrame() 文档片段：将需要动态生成的子元素加入到文档片段 ，然后将该文本片段加入到目标父节点，文档片段会在此过程中被其所有子元素一起替换，应为文档片段处于内存中且不再dom树中，所以使用文档片段插入可以减轻回溯，

Typeof实现原理：js存储数据前1-3位机器码用来表示类型

000：对象

010：浮点数

100：字符串

110：布尔

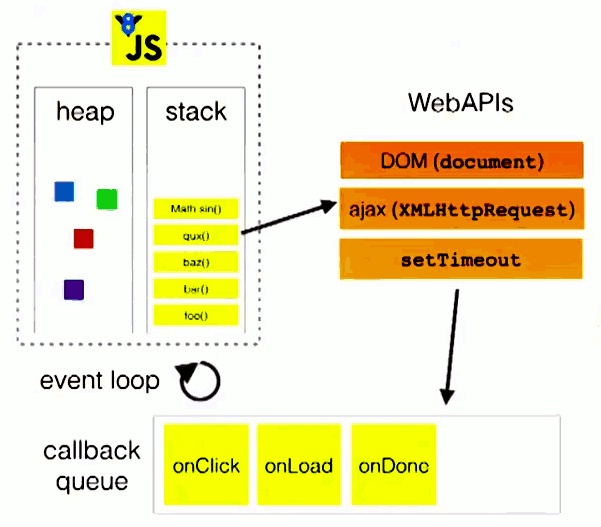
1：整数

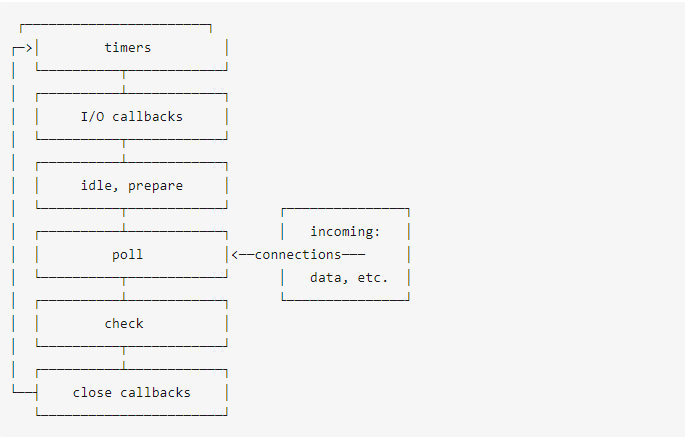
Instanceof实现原理，判断右边的prototype是否在左边的原型链上

requestAnimationFrame：requestAnimationFrame关键的就是他只是请求浏览器在下一次可以获得的机会去展示一帧画面，而不是在一个已经规划好的间隔。也就是说浏览器能够根据页面加载，元素显示，电池的状态来选择requestAnimationFrame的性能，它能够将所有的动画都放到一个浏览器重绘周期里去做，这样能保存你的CPU的循环次数，让你的设备存活时间更长

事件循环：

Browser：



Node：

解释：https://zhuanlan.zhihu.com/p/33058983

http缓存：强制缓存:查找浏览器缓存如果有则直接调用，没有(请求失败)则向服务器发起请求,包含请求参数expires（过期时间）参数cache-control

协商缓存（http0.9，1.0）:第一次请求时从服务器获取资源以及该资源的标识，之后请求时在请求头中携带该标识直接像服务器发请求，如果标识未过期则返回304（not modified）

http1为if-modified-since（如果资源改变则重新发送）if-unmodified-since(如果资源没变则重新发送) （存在资源修改但是并没有实质性变化的情况） http1.1则使用etag：服务器响应请求时，通过该字段告诉浏览器当前资源的唯一标识（由服务器生成）

if-none-match：再次请求包含此字段，后面的值为在缓存中获取的标识。服务器接收到次报文后发现If-None-Match则与被请求资源的唯一标识进行对比。不同，说明资源被改动过，则响应整个资源内容，返回状态码200。相同，说明资源无心修改，则响应header，浏览器直接从缓存中获取数据信息。返回状态码304.但是实际应用中由于Etag的计算是使用算法来得出的，而算法会占用服务端计算的资源，所有服务端的资源都是宝贵的，所以就很少使用Etag了。

使用原生js模拟

call：

*Function*.prototype.\_call = function(*context*) {

var context = context || window

var result

let args = []

let arg\_len = *arguments*.length

for (let i = 1; i < arg\_len; i++) {

args.push('arguments[' + i + ']')

}

context.fn = *this*

result = eval('context.fn(' + args + ')')

delete context.fn

return result

}

apply：

*Function*.prototype.\_apply = function(*context*, *arr*) {

var context = *Object*(context) || window

context.fn = *this*

var result

if (!arr) {

result = context.fn()

} else {

var args = []

for (var i = 0, len = arr.length; i < len; i++) {

args.push('arr[' + i + ']')

}

result = eval('context.fn(' + args + ')')

}

delete context.fn

return result

}

Bind：

*Function*.prototype.bind2 = function(*context*) {

var self = *this*

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

var fNOP = function() {}

var fBound = function() {

var bindArgs = *Array*.prototype.slice.call(*arguments*)

//作为构造函数使用，this指向fNOP则不改变this本身,作为普通函数调用时，//this指向window，将其修改为context

return self.apply(

*this* instanceof *fNOP* ? *this* : context,

args.concat(bindArgs)

)

}

*fNOP*.prototype = *this*.prototype

*fBound*.prototype = **new** *fNOP*()

return fBound

}

在HTTP 0.9和1.0中，[TCP连线](https://zh.wikipedia.org/wiki/%E4%BC%A0%E8%BE%93%E6%8E%A7%E5%88%B6%E5%8D%8F%E8%AE%AE)在每一次请求/回应对之后关闭。在HTTP 1.1中，引入了保持连线的机制，一个连接可以重复在多个请求/回应使用。持续连线的方式可以大大减少[等待时间](https://zh.wikipedia.org/wiki/%E5%BB%B6%E8%BF%9F_(%E5%B7%A5%E7%A8%8B%E5%AD%A6))，因为在发出第一个请求后，双方不需要重新运行[TCP握手程序](https://zh.wikipedia.org/wiki/%E6%8F%A1%E6%89%8B_(%E6%8A%80%E6%9C%AF))。

HTTP 1.1还使改进了HTTP 1.0的带宽。 例如，HTTP 1.1引入了[分块传输编码](https://zh.wikipedia.org/wiki/%E5%88%86%E5%9D%97%E4%BC%A0%E8%BE%93%E7%BC%96%E7%A0%81)，以允许传递内容可以在持续连在线被流传输而不必使用到[缓冲器](https://zh.wikipedia.org/wiki/%E7%B7%A9%E8%A1%9D%E5%99%A8)。HTTP管道允许客户端在收到每个回应之前发送多个请求，进一步减少[用户感受到的](https://zh.wikipedia.org/wiki/%E4%BD%BF%E7%94%A8%E8%80%85%E7%B6%93%E9%A9%97)滞后时间。协议的另一个补充是字节服务，允许客户端请求资源的某一部分，服务器仅回应某资源的指明部分。

http request:请求行（指定方法、资源路径、协议版本），请求头，空行，消息体

http response:响应状态，响应头，空行，响应体

null与undefined的区别：

null:表示没有对象，即该处不应该有值，一个值被定义了，定义为空值，所以将一个值设置为null是合理的

undefined：表示缺少值,即此处应该有值但未定义，该值根本未定义，所以设置一个值为undefined是不合理的

当进入执行上下文时，这时候还没有执行代码，

变量对象会包括：

1. 函数的所有形参 (如果是函数上下文)
   * 由名称和对应值组成的一个变量对象的属性被创建
   * 没有实参，属性值设为 undefined
2. 函数声明
   * 由名称和对应值（函数对象(function-object)）组成一个变量对象的属性被创建
   * 如果变量对象已经存在相同名称的属性，则完全替换这个属性
3. 变量声明
   * 由名称和对应值（undefined）组成一个变量对象的属性被创建；
   * 如果变量名称跟已经声明的形式参数或函数相同，则变量声明不会干扰已经存在的这类属性

变量对象（变量对象是与执行上下文相关的数据作用域，存储了在上下文中定义的变量和函数声明。）的创建过程：

1. 全局上下文的变量对象初始化是全局对象
2. 函数上下文的变量对象初始化只包括 Arguments 对象
3. 在进入执行上下文时会给变量对象添加形参、函数声明、变量声明等初始的属性值
4. 在代码执行阶段，会再次修改变量对象的属性值

Js按值传递：

var obj = 1

function foo(*o*) {

o = 2

*console*.log(o)

}

foo(obj) // 2

*console*.log(obj) // 1

引用传值错误例子：

var obj = {

value: 1

}

function fn(*v*) {

v.value = 2

*console*.log(v.value)//2

}

fn(obj)

*console*.log(obj.value) //2

第三种传递方式：

var obj = {

value: 1

}

function foo(*o*) {

o = 2

*console*.log(o) //2

}

foo(obj)

*console*.log(obj.value) // 1

共享传递，在传递对象时，传递对象的引用的副本，而按引用传递传递的是对象的引用

New操作符：创建一个用户自定义的对象类型的实例，或者具有构造函数的内置对象类型之一

模拟实现new：

如果有返回值，返回值为对象的话，则new出来的实例只能访问返回对象的属性，如果返回值为基本类型值，则只能访问构造函数的私有变量，无法访问返回的基本类型值

function objectFactory() {

var obj = **new** *Object*()

var Constructor = [].shift().call(*arguments*) //获取传入的构造函数

obj.\_\_proto\_\_ = *Constructor*.prototype //改变原型

let result = Constructor.apply(obj, *arguments*) //改变this指向

return typeof result === 'Object' ? result : obj

}

类数组对象：拥有length和若干索引属性的对象

将类数组转化成数组的方法：

// 1. slice

Array.prototype.slice.call(arrayLike); // ["name", "age", "sex"]

// 2. splice

Array.prototype.splice.call(arrayLike, 0); // ["name", "age", "sex"]

// 3. ES6 Array.from

Array.from(arrayLike); // ["name", "age", "sex"]

// 4. apply

Array.prototype.concat.apply([], arrayLike)

arguments 和对应参数的绑定

传入的参数，实参和arguments（通过arguments[index]的方法调用参数）的值会共享，当没有传入时则不共享，严格模式下也不会共享

function foo(name, age, sex, hobbit) {

console.log(name, arguments[0]); // name name

// 改变形参

name = 'new name';

console.log(name, arguments[0]); // new name new name

// 改变arguments

arguments[1] = 'new age';

console.log(age, arguments[1]); // new age new age

// 测试未传入的是否会绑定

console.log(sex); // undefined

sex = 'new sex';

console.log(sex, arguments[2]); // new sex undefined

arguments[3] = 'new hobbit';

console.log(hobbit, arguments[3]); // undefined new hobbit

}

foo('name', 'age')

函数柯里化:

function curry(*fn*, *arg*) {

var length = fn.length

var arg = arg || []

return function() {

var \_args = arg.splice(0)

var arr = []

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

arr[i] = *arguments*[i]

\_args.push(arr[i])

}

if (\_args.length < length) {

return curry.call(*this*, fn, \_args)

} else {

return fn.apply(*this*, \_args)

}

}

}

function fn(*a*, *b*, *c*) {

return a + b + c

}

var init = curry(fn) //function

var one = init(1) //function

var two = one(2) //function

var three = two(3) //3

简洁写法:

var curry = *fn* =>

(judge = (...*args*) =>

args.length === fn.length

? fn(...args)

: *arg* => judge(...args, arg))

***这样理解柯里化*** ：用闭包把参数保存起来，当参数的数量足够执行函数了，就开始执行函数，有没有毛病

***函数组合：***Pointfree 的本质就是使用一些通用的函数，组合出各种复杂运算。上层运算不要直接操作数据，而是通过底层函数去处理。即不使用所要处理的值，只合成运算过程。

pointfree 模式能够帮助我们减少不必要的命名，让代码保持简洁和通用，更符合语义，更容易复用，测试也变得轻而易举

***惰性函数***：解决每次都要进行判断的这个问题，解决原理很简单，重写函数

var foo = function() {

var t = new Date();

foo = function() {

return t;

};

return foo();

};（返回第一次调用函数时的时间）

普通递归和尾递归的区别：

// 尾调用

function f(x){

return g(x);

}

// 非尾调用

function f(x){

return g(x) + 1;

}

为了模拟执行上下文栈的行为，让我们定义执行上下文栈是一个数组：

ECStack = [];

我们模拟下第一个尾调用函数执行时的执行上下文栈变化：

// 伪代码

ECStack.push(<f> functionContext);

ECStack.pop();

ECStack.push(<g> functionContext);

ECStack.pop();

我们再来模拟一下第二个非尾调用函数执行时的执行上下文栈变化：

ECStack.push(<f> functionContext);

ECStack.push(<g> functionContext);

ECStack.pop();

ECStack.pop();

也就说尾调用函数执行时，虽然也调用了一个函数，但是因为原来的的函数执行完毕，执行上下文会被弹出，执行上下文栈中相当于只多压入了一个执行上下文。然而非尾调用函数，就会创建多个执行上下文压入执行上下文栈。

函数调用自身，称为递归。如果尾调用自身，就称为尾递归。

***尾递归实现***：将内部变量都作为参数传入函数