1. **如何实现多标签页之间的通讯。**

调用localStorage

在一个标签页里面使用 localStorage.setItem(key,value)添加（修改、删除）内容；   
在另一个标签页里面监听 storage 事件。   
即可得到 localstorge 存储的值，实现不同标签页之间的通信。

标签页1：

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>localStorage-a</title>

<script src="https://cdn.bootcss.com/jquery/1.12.4/jquery.min.js"></script>

</head>

<body>

<input id="name">

<input type="button" id="btn" value="提交">

<script type="text/javascript">

$(function(){

$("#btn").click(function(){

var name=$("#name").val();

localStorage.setItem("name", name);

});

});

</script>

</body>

</html>

标签页2：

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>localstorage-b</title>

<script src="https://cdn.bootcss.com/jquery/1.12.4/jquery.min.js"></script>

</head>

<body>

<script type="text/javascript">

$(function(){

console.log(localStorage.getItem("name"));

window.addEventListener("storage", function(event){

console.log(event.key + "=" + event.newValue);

});

});

</script>

</body>

</html>

Cookie：

标签页1：

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>cookie-a</title>

<script src="https://cdn.bootcss.com/jquery/1.12.4/jquery.min.js"></script>

</head>

<body>

<input id="name">

<input id="password">

<input type="button" id="btn" value="提交">

<script type="text/javascript">

$(function(){

$("#btn").click(function(){

var name=$("#name").val();

document.cookie="name="+name;

var password=$("#password").val();

document.cookie="password="+password;

});

});

</script>

</body>

</html>

标签页1：

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>cookie-b</title>

<script src="https://cdn.bootcss.com/jquery/1.12.4/jquery.min.js"></script>

</head>

<body>

<script type="text/javascript">

$(function(){

function getCookie() {

//获取cookie字符串

var strCookie=document.cookie;

//将多cookie切割为多个名/值对

var arrCookie=strCookie.split("; ");

var temp = {};

var name;

var password;

//遍历cookie数组，处理每个cookie对

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

var arr=arrCookie[i].split("=");

//找到名称为userId的cookie，并返回它的值

temp[arr[0]] = arr[1];

}

return temp;

}

console.log(getCookie());

setInterval(function(){

console.log(getCookie());

}, 10000);

});

</script>

</body>

</html>

* **sessionStorage 、localStorage 和 cookie 之间的区别**  
  共同点：都是保存在浏览器端，且同源的。
* 区别：cookie数据始终在同源的http请求中携带（即使不需要），即cookie在浏览器和服务器间来回传递；cookie数据还有路径（path）的概念，可以限制cookie只属于某个路径下。存储大小限制也不同，cookie数据不能超过4k，同时因为每次http请求都会携带cookie，所以cookie只适合保存很小的数据，如会话标识。
* 而sessionStorage和localStorage不会自动把数据发给服务器，仅在本地保存。sessionStorage和localStorage 虽然也有存储大小的限制，但比cookie大得多，可以达到5M或更大。
* 数据有效期不同，sessionStorage：仅在当前浏览器窗口关闭前有效，自然也就不可能持久保持；localStorage：始终有效，窗口或浏览器关闭也一直保存，因此用作持久数据；cookie只在设置的cookie过期时间之前一直有效，即使窗口或浏览器关闭。
* 作用域不同，sessionStorage**不在**不同的浏览器窗口中共享，即使是同一个页面；localStorage 在所有同源窗口中都是共享的；cookie也是在所有同源窗口中都是共享的。Web Storage 支持事件通知机制，可以将数据更新的通知发送给监听者。Web Storage 的 api 接口使用更方便。

区别

|  |  |  |  |
| --- | --- | --- | --- |
| 特性 | Cookie | LocalStorage | sessionStorage |
| 数据的声明周期 | 一般由服务器生成，可设置失效时间。如果在浏览器生成，默认是关闭浏览器之后失效 | 除非被清楚，否则永久保存 | 仅在当前会话有效，关闭页面或浏览器后被清除 |
| 存放数据大小 | 4KB | 一般 5MB | 一般 5MB |
| 与服务端通信 | 每次都会携带在HTTP头中，如果使用 cookie 保存过多数据会带来性能问题 | 仅在客户端中保存，不参与和服务器的通信。~~也可有脚本选择性提交到服务器端？~~ | 同 LocalStorage |
| 用途 | 一般由服务器端生成，用于标识用户身份 | 用于浏览器端缓存数据 | 同 LocalStorage |
| 共享 |  |  |  |

**2、简述什么是按需加载，然后实现一个图片按需加载的js。**

按需加载是前端性能优化中的一项重要措施，按需加载是如何定义的呢？顾名思义，指的是当用户触发了动作时才加载对应的功能。触发的动作，是要看具体的业务场景而言，包括但不限于以下几个情况：鼠标点击、输入文字、拉动滚动条，鼠标移动、窗口大小更改等。加载的文件，可以是JS、图片、CSS、HTML等

代码：

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>图片按需加载</title>

</head>

<body id="body">

<div ></div>

<img src="" width="990px" height="100px" data-src="http://dummyimage.com/990x90/37683/fff" alt="">

<img src="" width="990px" height="100px" data-src="http://dummyimage.com/990x90/f7680/fff" alt="">

<img src="" width="990px" height="100px" data-src="http://dummyimage.com/990x90/27689/fff" alt="">

<img src="" width="990px" height="100px" data-src="http://dummyimage.com/990x90/97689/fff" alt="">

<img src="" width="990px" height="100px" data-src="http://dummyimage.com/990x90/47683434/fff" alt="">

<img src="" width="990px" height="100px" data-src="http://dummyimage.com/990x90/344D3C/fff" alt="">

<img src="" width="990px" height="100px" data-src="http://dummyimage.com/990x90/96783/fff" alt="">

<img src="" width="990px" height="100px" data-src="http://dummyimage.com/990x90/96780/fff" alt="">

<img src="" width="990px" height="100px" data-src="http://dummyimage.com/990x90/96789/fff" alt="">

<img src="" width="990px" height="100px" data-src="http://dummyimage.com/990x90/96789/fff" alt="">

<img src="" width="990px" height="100px" data-src="http://dummyimage.com/990x90/96783434/fff" alt="">

<img src="" width="990px" height="100px" data-src="http://dummyimage.com/990x90/96784D3C/fff" alt="">

<img src="" width="990px" height="100px" data-src="http://dummyimage.com/990x90/273459/fff" alt="">

<img src="" width="990px" height="100px" data-src="http://dummyimage.com/990x90/973459/fff" alt="">

<img src="" width="990px" height="100px" data-src="http://dummyimage.com/990x90/473453434/fff" alt="">

<img src="" width="990px" height="100px" data-src="http://dummyimage.com/990x90/343453C/fff" alt="">

<img src="" width="990px" height="100px" data-src="http://dummyimage.com/990x90/963453/fff" alt="">

<img src="" width="990px" height="100px" data-src="http://dummyimage.com/990x90/963450/fff" alt="">

<img src="" width="990px" height="100px" data-src="http://dummyimage.com/990x90/963459/fff" alt="">

<img src="" width="990px" height="100px" data-src="http://dummyimage.com/990x90/963459/fff" alt="">

<img src="" width="990px" height="100px" data-src="http://dummyimage.com/990x90/963453434/fff" alt="">

<img src="" width="990px" height="100px" data-src="http://dummyimage.com/990x90/963454D3C/fff" alt="">

<script type="text/javascript">

(function (){

var imgNum=document.getElementsByTagName('img').length;

var imgObj=document.getElementsByTagName("img");

var l=0;

var lazyLoad = function(){

var seeHeight = document.documentElement.clientHeight;

var scrollTop = document.documentElement.scrollTop || document.body.scrollTop;

for(var i=l;i<imgNum;i++){

if(imgObj[i].offsetTop < seeHeight + scrollTop){

console.log(imgObj[i].getAttribute("src"));

console.log(imgObj[i].src );

if(imgObj[i].getAttribute("src") == ""){

imgObj[i].src = imgObj[i].getAttribute("data-src");

}

}

if(imgObj[i].offsetTop > seeHeight + scrollTop){

l=i;

break;

}

}

}

window.onload = lazyLoad;

window.onscroll = lazyLoad;

})();

</script>

</body>

</body>

</html>

1. **简述下闭包，有什么好处和坏处。**

js的作用域分两种，全局和局部，基于我们所熟悉的作用域链相关知识，我们知道在js作用域环境中访问变量的权利是由内向外的，内部作用域可以获得当前作用域下的变量并且可以获得当前包含当前作用域的外层作用域下的变量，反之则不能，也就是说在外层作用域下无法获取内层作用域下的变量，同样在不同的函数作用域中也是不能相互访问彼此变量的，那么我们想在一个函数内部也有限权访问另一个函数内部的变量该怎么办呢？闭包就是用来解决这一需求的，闭包的本质就是在一个函数内部创建另一个函数。、

**我们首先知道闭包有3个特性：**

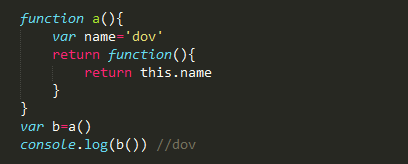
**①函数嵌套函数**

**②函数内部可以引用函数外部的参数和变量**

**③参数和变量不会被垃圾回收机制回收**

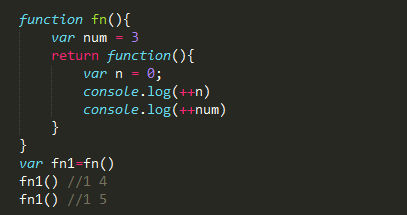
本文我们以闭包两种的主要形式来学习

**①函数作为返回值**



在这段代码中，a()中的返回值是一个匿名函数，这个函数在a()作用域内部，所以它可以获取a()作用域下变量name的值，将这个值作为返回值赋给全局作用域下的变量b,实现了在全局变量下获取到局部变量中的变量的值

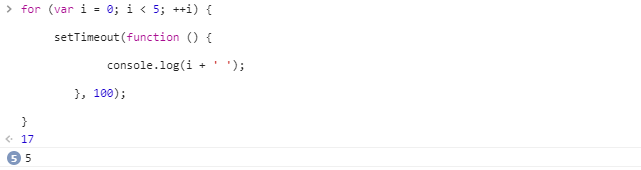
再来看一个闭包的经典例子



一般情况下，在函数fn执行完后，就应该连同它里面的变量一同被销毁，但是在这个例子中，匿名函数作为fn的返回值被赋值给了fn1，这时候相当于fn1=function(){var n = 0 ... }，并且匿名函数内部引用着fn里的变量num，所以变量num无法被销毁，而变量n是每次被调用时新创建的，所以每次fn1执行完后它就把属于自己的变量连同自己一起销毁，于是乎最后就剩下孤零零的num，于是这里就产生了内存消耗的问题

**再来看一个经典例子-定时器与闭包**

写一个for循环，让它按顺序打印出当前循环次数



按照预期它应该依次输出1 2 3 4 5，而结果它输出了五次5，这是为什么呢？原来由于js是单线程的，所以在执行for循环的时候定时器setTimeout被安排到任务队列中排队等待执行，而在等待过程中for循环就已经在执行，等到setTimeout可以执行的时候，for循环已经结束，i的值也已经编程5，所以打印出来五个5，那么我们为了实现预期结果应该怎么改这段代码呢？（ps:如果把for循环里面的var变成let，也能实现预期结果）



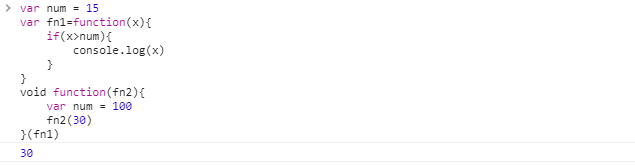
引入闭包来保存变量i，将setTimeout放入立即执行函数中，将for循环中的循环值i作为参数传递，100毫秒后同时打印出1 2 3 4 5

那如果我们想实现每隔100毫秒分别依次输出数字，又该怎么改呢?



在这段代码中，相当于同时启动3个定时器，i\*100是为4个定时器分别设置了不同的时间，同时启动，但是执行时间不同，每个定时器间隔都是100毫秒，实现了每隔100毫秒就执行一次打印的效果。

**②闭包作为参数传递**



在这段代码中，函数fn1作为参数传入立即执行函数中，在执行到fn2(30)的时候，30作为参数传入fn1中，这时候if(x>num)中的num取的并不是立即执行函数中的num，而是取创建函数的作用域中的num这里函数创建的作用域是全局作用域下，所以num取的是全局作用域中的值15，即30>15，打印30

**最后总结一下闭包的好处与坏处**

好处

①保护函数内的变量安全 ，实现封装，防止变量流入其他环境发生命名冲突

②在内存中维持一个变量，可以做缓存（但使用多了同时也是一项缺点，消耗内存）

③匿名自执行函数可以减少内存消耗

坏处

①其中一点上面已经有体现了，就是被引用的私有变量不能被销毁，增大了内存消耗，造成内存泄漏，解决方法是可以在使用完变量后手动为它赋值为null；

②其次由于闭包涉及跨域访问，所以会导致性能损失，我们可以通过把跨作用域变量存储在局部变量中，然后直接访问局部变量，来减轻对执行速度的影响

1. **什么是模块开发，怎么实现一个模块开发。**

一个模块就是实现特定功能的文件，有了模块，我们就可以更方便地使用别人的代码，想要什么功能，就加载什么模块。

实现一个模块开发，有多种方法，这里陈述两种

1. require.js
2. 使用require.js的第一步，是先去官方网站[下载](http://requirejs.org/docs/download.html)最新版本
3. 加载该文件　<script src="js/require.js" **defer async="true"** ></script>
4. 加载主模块<script src="js/require.js" **data-main="js/main"**></script>
5. 写主模块
6. 在主模块头部加载其余子模块
7. equire.js加载的模块，采用AMD规范。也就是说，模块必须按照AMD的规定来写。具体来说，就是模块必须采用特定的define()函数来定义。如果一个模块不依赖其他模块，那么可以直接定义在define()函数之中。
8. es6+webpack

用es6 export，import等写模块，再webpack打包

1. **实现一个多列的等高布局。**

#### 使用正padding和负margin对冲实现多列布局方法（padding补偿法）

padding补偿法。首先把列的padding-bottom设为一个足够大的值，再把列的margin-bottom设一个与前面的padding- bottom的正值相抵消的负值，父容器设置超出隐藏，这样子父容器的高度就还是它里面的列没有设定padding-bottom时的高度，当它里面的任 一列高度增加了，则父容器的高度被撑到它里面最高那列的高度，其他比这列矮的列则会用它们的padding-bottom来补偿这部分高度差。因为背景是 可以用在padding占用的空间里的，而且边框也是跟随padding变化的，所以就成功的完成了一个障眼法。

代码：

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>使用正padding和负margin对冲实现多列布局方法</title>

<style type="text/css">

#container {

margin: 0 auto;

overflow: hidden;

width: 960px;

border: 3px red solid;

}

.column {

background: #ccc;

float: left;

width: 200px;

margin-right: 5px;

margin-bottom: -99999px;

padding-bottom: 99999px;

}

#content {

background: #eee;

}

#right {

float: right;

margin-right: 0;

}

</style>

</head>

<body>

<div id="container">

<div id="left" class="column aside">

<p>Sidebar</p>

</div>

<div id="content" class="column section">

<p>Main content</p>

</div>

<div id="right" class="column aside">

<p>Sidebar</p>

</div>

</div>

</body>

</html>

#### 二、使用flex实现等高

### Flex: align-items属性

stretch（默认值）：如果项目未设置高度或设为auto，将占满整个容器的高度。

所以可多列等高

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>flex实现多列等高</title>

<style type="text/css">

#wrapper {

width: 1000px;

margin: 0 auto;

display: flex;

justify-content: space-between;

}

#left,#right{

width: 300px;

background-color: #ccc;

}

#middle{

width: 330px;

background-color: #eee;

height: 500px;

}

</style>

</head>

<body>

<div id="wrapper">

<div id="left">left</div>

<div id="middle">middle</div>

<div id="right">right</div>

</div>

</body>

</html>

1. **zoom是什么，有什么用？**

Zoom属性是IE浏览器的专有属性， 它可以设置或检索对象的缩放比例。在平常的css编写过程中，zoom:1能够比较神奇地解决ie下比较奇葩的bug。

譬如外边距（margin）的重叠，譬如浮动的清除，譬如触发ie的 haslayout属性等等。

当设置了zoom的值之后，所设置的元素就会就会扩大或者缩小，高度宽度就会重新计算了，这里一旦改变zoom值时其实也会发生重新渲染，运用这个原理，也就解决了ie下子元素浮动时候父元素不随着自动扩大的问题。

放大：

<!DOCTYPE html>

<html lang="zh-cn">

<head>

<meta charset="utf-8" />

<title>zoom</title>

<style>

body{

line-height:1.5;

}

h1{

margin:0;

font-size:16px;

font-family:Arial;

}

.test{

zoom:normal;

}

.test2{

zoom:5;

}

.test3{

zoom:300%;

}

</style>

</head>

<body>

<h1>zoom:normal</h1>

<div class="test">zoom:normal</div>

<h1>zoom:5</h1>

<div class="test2">zoom:5</div>

<h1>zoom:300%</h1>

<div class="test3">zoom:300%</div>

</body>

</html>

**7、怎么优化一个首屏的加载速度，可以从js和css等多方便考虑**

1、JavaScript外联文件引用放在html文档底部；CSS外联文件引用在html文档头部，位于header内；

2、http静态资源尽量用多个子域名；

3、服务器端提供html文档和http静态资源时，尽量开启gzip压缩；

4、在js，css，image等资源响应的http headers里，设置expires，last-modified；

5、尽量减少HTTP Requests的数量；

6、js/css的minify：可统一通过combo handler做到压缩加合并；

7、减少不必要的301/302跳转：别让页面打开时间浪费在302多次跳转上（每次可能几十毫秒）；

8、请大量使用CSS Spirtes：这样做可以大大地减少CSS背景图片的HTTP请求次数；

9、首屏不需要展示的较大尺寸图片，请使用lazyload；

10、图片无损压缩的优化；

11、避免404错误：尽量减少外联js；

12、减少cookies的大小：尽量减少cookies的体积对减少用户获得响应的时间十分重要；

13、使用无cookies的域；

14、减少DOM Elements的数量；

15、引入textarea/script元素做延迟解析异步渲染

**8、简述下new一个对象后，this的变化。**

构造函数的调用需要用new操作符，而普通函数的调用又分很多种，但是都不会用到new操作符。所以，构造函数和普通函数的区别就在这个new操作符里，现在让我们来好好研究一下这个new操作符。  
用new操作符创建对象时发生的事情：  
\*\*第一步: 创建一个Object对象实例。  
第二步: 将构造函数的执行对象赋给新生成的这个实例。  
第三步: 执行构造函数中的代码  
第四步: 返回新生成的对象实例\*\*

注意：原本的构造函数是window对象的方法，如果不用new操作符而直接调用，那么构造函数的执行对象就 是window，即this指向了window。现在用new操作符后，this就指向了新生成的对象。理解这一步至关重要。  
执行构造函数中的代码，看代码：

function Person(){

this.name = "Tiny Colder";

**var** age = 22;

window.age = 22;

}

**var** p = **new** **Person**();

**alert**(p.name)//Tiny Colder;

**alert**(p.age)//undefined;

**alert**(window.age)//22;

当用new操作符创建对象时，先创建了一个对象实例，然后执行代码。所以还在纠结，什么时候构造函数定义的属性会继承给实例对象的，都可以这么来看：

**var** p = **new** Object();

p.name = "Tiny Colder";

这是普通的创建对象，然后给对象添加属性的方法。如果每创建一个对象，都需要这么几行代码，无疑是糟糕的。这个需求就正好跟这一点对应：new操作符，自动执行构造函数里的代码。如此我们便可以省掉添加属性时重复冗余的代码。那么这些属性时如何添加到新生成的对象里的呢？

第二个步骤里已经说了：将构造函数的执行对象赋给新生成的这个实例。再结合上一段里说的，自动执行构造函数里的this.name = "Tiny Colder";时，就相当于是执行p.name = "Tiny Colder";而构造函数里的  
var age = 22;语句，会执行但是对新生成的对象并无影响。window.age = 22;语句，会执行，且会给window对象添加一个属性。alert为证。

或许到这里，你已经理解了new操作符的前三步了，重要的三步。但是这个函数是如何返回对象的呢？我们并没有看到有任何跟return相关的语句。这就是new操作符的最后一步：返回新生成的对象。  
如果被调用的函数没有显式的 return 表达式（仅限于返回对象），则隐式的会返回 this 对象 - 也就是新创建的对象。

所以，在new一个对象的过程中，this会先指向新生成的对象

**9、怎么实现一个对象的继承。**

**1.原型链**

基本思想：利用原型让一个引用类型继承另外一个引用类型的属性和方法。

构造函数，原型，实例之间的关系：每个构造函数都有一个原型对象，原型对象包含一个指向构造函数的指针，而实例都包含一个指向原型对象的内部指针。

例子：

function SuperType() {

this.property = true;

}

SuperType.prototype.getSuperValue = function() {

return this.property;

}

function subType() {

this.property = false;

}

//继承了SuperType

SubType.prototype = new SuperType();

SubType.prototype.getSubValue = function (){

return this.property;

}

var instance = new SubType();

console.log(instance.getSuperValue());//true

**2.借用构造函数**

基本思想：在子类型构造函数的内部调用超类构造函数，通过使用call()和apply()方法可以在新创建的对象上执行构造函数。

例子：

function SuperType() {

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

}

function SubType() {

SuperType.call(this);//继承了SuperType

}

var instance1 = new SubType();

instance1.colors.push("black");

console.log(instance1.colors);//"red","blue","green","black"

var instance2 = new SubType();

console.log(instance2.colors);//"red","blue","green"

**3.组合继承**

基本思想：将原型链和借用构造函数的技术组合在一块，从而发挥两者之长的一种继承模式

例子：

function SuperType(name) {

this.name = name;

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

}

SuperType.prototype.sayName = function() {

console.log(this.name);

}

function SubType(name, age) {

SuperType.call(this,name);//继承属性

this.age = age;

}

//继承方法

SubType.prototype = new SuperType();

Subtype.prototype.constructor = Subtype;

Subtype.prototype.sayAge = function() {

console.log(this.age);

}

var instance1 = new SubType("EvanChen",18);

instance1.colors.push("black");

consol.log(instance1.colors);//"red","blue","green","black"

instance1.sayName();//"EvanChen"

instance1.sayAge();//18

var instance2 = new SubType("EvanChen666",20);

console.log(instance2.colors);//"red","blue","green"

instance2.sayName();//"EvanChen666"

instance2.sayAge();//20

**4.原型式继承**

基本想法：借助原型可以基于已有的对象创建新对象，同时还不必须因此创建自定义的类型。

例子：

var person = {

name:"EvanChen",

friends:["Shelby","Court","Van"];

};

var anotherPerson = object(person);

anotherPerson.name = "Greg";

anotherPerson.friends.push("Rob");

var yetAnotherPerson = object(person);

yetAnotherPerson.name = "Linda";

yetAnotherPerson.friends.push("Barbie");

console.log(person.friends);//"Shelby","Court","Van","Rob","Barbie"

**5.寄生式继承**

基本思想：创建一个仅用于封装继承过程的函数，该函数在内部以某种方式来增强对象，最后再像真正是它做了所有工作一样返回对象。

例子：

function createAnother(original) {

var clone = object(original);

clone.sayHi = function () {

alert("hi");

};

return clone;

}

var person = {

name:"EvanChen",

friends:["Shelby","Court","Van"];

};

var anotherPerson = createAnother(person);

anotherPerson.sayHi();///"hi"

**6.寄生组合式继承**

基本思想：通过借用函数来继承属性，通过原型链的混成形式来继承方法

例子：

function SuperType(name){

this.name = name;

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

}

SuperType.prototype.sayName = function (){

alert(this.name);

};

function SubType(name,age){

SuperType.call(this,name);

this.age = age;

}

inheritProperty(SubType,SuperType);

SubType.prototype.sayAge = function() {

alert(this.age);

}

**10、简述下平时开发用到的技术栈。**

Html+css+js+bootstrap+jquery，会一些Vue