### HTML类

* 1. HTML有什么新特性
     1. 语义特性：HTML5赋予网页更好的意义和结构

1. 文件类型声明（<!DOCTYPE>）仅有一型：<!DOCTYPE HTML>
2. 新的解析顺序：不再基于SGML——一套严谨的文件描述方法
3. 新的元素：section、video、progress、nav、meter、time、aside、canvas、command、datalist、details、embed、figcaption、figure、footer、header、hgroup、keygen、mark、output、rp、rt、ruby、source、summary、wbr
4. input元素的新类型：date、email、url等
5. 新的属性：ping（用于a与area）、charset（用于meta）、async（用于script）
6. 全域属性：id、tabindex、repeat
7. 新的全域属性：contenteditable、contextmenu、draggable、dropzone、hidden、spellcheck
8. 移除元素：acronym、applet、basefont、big、center、dir、font、frame、frameset、isindex、noframes、strike、tt
   * 1. 本地存储特性

HTML5离线存储包含：应用程序缓存（Application Cache）、本地存储、索引数据库、文件接口

1. 应用程序缓存

通过创建cache manifest文件，可以轻松的创建web应用的离线版本，其优势在于：

* 离线浏览——用户可在应用离线时使用他们
* 速度——已缓存静态资源，使加载更快
* 减少服务器负载——浏览器将只存服务器下载更新过或修改过的资源

1. 本地存储

* localStorage——只能手动删除
* SessionStorage——页面关闭就消失

1. 索引数据库（indexed DB）

从本质上说，indexed DB允许用户在浏览器中保存大量的数据。任何需要发送大量数据的应用都可以得益于这个特性，可以把数据存储在用户的浏览器端。当前这只是indexed DB的其中一项功能，indexed DB也提供了强大的基于索引的搜索API功能以获得用户所需要的数据。

indexed DB和以前的存储机制（cookie、session）的区别：

Cookie是最常用的浏览器端保存数据的机制，但其保存数据的大小有限制并且有隐私问题。Cookie并且会在每个请求中来回发送数据，完全没办法发挥客户端数据存储的优势。

LocalStorage在HTML5中有着不错的支持，但就总的存储量而言依然是有所限制。LocalStorage并不提供真正的“检索API”，本地存储的数据只是通过键值对去访问。LocalStorage对于一些特定的需要存储数据的场景是很适合的，如用户的喜好习惯，而indexed DB则更适合存储如广告等数据（它更像一个真正的数据库）。

一般来说，有两种不同类型的数据库：关系型和文档型（也称NoSQL或对象）。关系数据库如SQL、server、MySQL、Oracle的数据库存储在表中。文档数据库如MongoDB、CouchDB、Redis将数据集作为个体对象存储。indexed DB是一个文档数据库，他在完全内置于浏览器中的一个沙盒环境中（将至依照（浏览器）同源策略）。

对数据库的每次操作，描述为通过一个请求打开数据库，访问一个object store。

indexed DB是否适合应用程序的几个关键点

* 用户通过浏览器访问应用程序（浏览器）应支持indexed DB API
* 需要存储大量数据在客户端
* 需要在一个大型的数据集合中快速定位单个数据点
* 架构在客户端需要事务支持

1. 文件接口
   * 1. 设备访问特性

包括地理位置API、媒体访问API、访问联系人及事件、设备方向

* + 1. 连接特性

HTTP是无连接的，一次请求，一次响应。想要实现微信网页版扫一扫登录，网页版微信聊天功能，需要使用轮询的方式达到长连接的效果，轮询的大部分时间是在做无用功，浪费网络资源。现在HTML5为我们带来了更高效的连接方式：web Sockets和Server-Sent Events。

* + 1. 网页多媒体特性

HTML5支持原生的音视频能力：audio、video

* + 1. 三维、图形及特效特性

大致包含SVG、Canvas、WebGL、CSS3 3D

* + 1. 性能与集成特性

性能与集成特性主要包括两个东西 Web Workers和XMLHttpRequest 2

### 前端重构思路

* 1. 搞清楚被重构项目的配置。如：
     1. jQuery：DOM操作，含UI插件
     2. Bootstrap：UI风格
     3. AWF：企业级组件库
     4. Gulp：打包工具
     5. 单页面应用（SPA）还是多页面应用（MPA）
  2. 确定重构方案。如：
     1. Require.js（text+css插件）：模块化加载（我理解是懒加载）
     2. Mockjs：模拟ajax调用
     3. Director.js：路由（vue里面自带vue-router）
     4. Underscore：模块化
     5. jQuery：DOM操作，含UI及插件
     6. Bootstrap：UI风格（建议常用的element-UI）
     7. AWF：企业级组件库（建议常用的element-UI）
     8. 打包：gulp+r.js

### 如何在Vue项目中使用vw实现移动端适配

* 1. <https://www.jianshu.com/p/1f1b23f8348f>

### CSS类

* 1. 视口单位vw、vh实现自适应（带有px，em，rem的简单介绍）
     1. px：像素单位，即图像的基本采样单位。对不同的设备，它的图像基本单位是不同的，如：显示器和打印机。通常来说，显示器分辨率是指的是桌面设定的分辨率，不是物理分辨率。但是为了最佳显示效果，一般显示器分辨率就等于物理分辨率。由此导致：**使用px会严重受到桌面分辨率的影响，需要做自适应**
     2. em：相对长度单位。相对于当前对象内本文的**字体尺寸**（只受字体大小影响，不设即默认等于父元素的字体大小。**即受父元素的影响，**如果父元素没有，会一直往上找，直到body默认字体尺寸，也就是16px，则此时换算就是1em=16px）。如果使用em，建议将body的fon-size设置成62.5%，即16\*62.5%=10px，此时1em=10px，方便后面计算。
     3. rem：相对长度单位，但是**rem始终都是相对html的根元素**，不会受到对象内文本字体尺寸的影响，只需要改变根元素就能改变所有字体大小。兼容IE8以上即其他所有浏览器。
     4. vw、vh：视口（viewport units）单位，即**根据浏览器窗口大小的单位**，不受显示器分辨率的影响。vw（可视窗口的宽度）vh（可视窗口的高度）类似百分比，1vw=可视窗口的百分之一。如：窗口宽度大小是1800px，那么1vw=18px。但是**vw始终相对于可视窗口的宽度，而百分比受到父元素的影响。**vmin和vmax是指选择vm和vh中最小的那个，而vmax是选择最大的那个。**兼容性较差。**vm和vh由于受浏览器窗口影响，所以很轻松完成响应式布局。

5.em举例：

<style>

body {

font-size: 10px;

}

div {

font-size: 3em;

margin-top: 3em;

}

p {

font-size: 2em;

}

span {

font-size: 3em;

}

</style>

</head>

<body>body

<div>div

<p>p<br/>

<span>span</span>

</p>

</div>

</body>结果：

（理解：div元素的font-size =3em \* 10 = 30px margin-top=3em \* 10px=30px

p元素的font-size =2em \* 30px=60px; span元素的font-size=3em \* 60px=180px）

6.rem举例：

<style>

body {

font-size: 10px;

}

div {

font-size: 3rem;

}

p {

font-size: 2rem;

}

span {

font-size: 3rem;

}

</style>

</head>

<body>body

<div>div

<p>p<br/>

<span>span</span>

</p>

</div>

</body>

结果：

（理解：其换算结果都只跟body的10px相关）

1. **JS类**
   1. 简述Event loop/描述场景/执行顺序
      1. JS的事件循环

我们都知道JS是单线程的

原因：作为浏览器脚本语言，JavaScript的主要用途是与用户互动，以及操作DOM。这决定了他只能是单线程，否则会带来很复杂的同步问题。比如，假定JavaScript同时有两个线程，一个线程在某个DOM节点上添加内容，另一个线程删除了这个节点，这时浏览器应该以哪个为准？

所以为了避免复杂性，从一诞生，JavaScript就是单线程

由于JS是单线程，只有当上一个任务完成之后才会继续完成下一个任务，如果前一个任务耗时很长，后一个任务就不得不一直等着。于是，所有任务可以分为两种，一种是同步任务（synchronous），另一种是异步任务（asynchronous）

1. 同步任务

在主线程上排队执行的任务，只有前一个任务执行完毕，才能执行下一个任务。

1. 异步任务

不进入主线程，而是进入到“任务队列”（task queue）的任务，只有“任务队列”通知主线程，某个异步任务可以执行了，该任务才会进入主线程执行。

上面代码中主线程首先打印1，遇到setTimeout时，按照异步处理，1秒之后，setTimeout的回调函数会进入到任务队列，主线程会继续运行，打印3，当主线程中的任务运行完成之后，会运行任务队列中的任务，打印2

流程图：

文字叙述：

* 所有同步任务都在主线程上执行，形成一个执行栈（execution context stack）。
* 主线程之外，还存在一个“任务队列”（task queue）。只要异步任务有了运行结果，就在“任务队列”之中放置一个事件。
* 一旦“执行栈”中的所有同步任务执行完步，系统就会读取“任务队列”，看看里面哪些事件。哪些对应的异步任务，于是结束等待状态，进入执行栈，开始执行。
* 主线程不断重复上面的第三步。

**主线程从任务队列中读取事件，这个过程是不断循环的，所以整个运行机制称为event loop**

* + 1. Macro-task（宏任务）和Micro-task（微任务）

任务还可以认为宏任务和微任务

1. 宏任务：可以理解为每次执行栈执行的代码就是一个宏任务（包括每次从事件队列中获取一个事件回调并放到执行栈中执行，每一个宏任务会从头到尾将这个任务执行完毕，不会执行其他）包括整体代码script、setTimeout、setInterval
2. 微任务：可以理解是在当前task执行结束后立即执行的任务，包括Promise、process.nextTick

上面代码——首先整体代码是一个宏任务，遇到setTimeout，会创建另一个宏任务，接着执行当前的宏任务，Promise新建后会立即执行。所以会首先打印2，then方法是一个微任务，遇到then，添加到微任务队列，代码接着执行会打印4。此时宏任务执行完毕。接着就会检查当前微任务队列是否有微任务，如果有，立即执行微任务（也就是then打印3），当前微任务执行完毕之后，开始执行下一轮宏任务setTimeout，会打印1。

原理图：

面试题：

* 1. 原型与原型链
     1. 构造函数

构造函数与其他函数唯一的区别在于调用方式不同。任何函数只要通过new来调用就可以作为构造函数，它是用来创建特定类型的对象。

下面定义一个构造函数Female：

通过new命令来生成一个person实例：

这里，构造函数Female就是实例对象person1的原型，Female里的this关键字指的是person1这个对象。

New出来的person1对象此时已经与Female再无联系，也就是说每一个new出来的实例都有自己的属性和方法的副本，是完全独立的，修改一个不会影响到另一个，如下面代码所示：

但是，我们希望构造函数中的sex属性是一个共有属性，那么此时用这样的方法，每个实例都有一个相同的sex属性，会造成资源的极大浪费。那么原型对象就由此产生Brendan Eich（一个很牛逼的人）决定给每一个构造函数都设置一个prototype属性，这个属性就指向原型对象。其实原型对象就只是一个普通对象，里面存放着所有实例对象需要共享的属性和方法。所以，我们把需要共享的放到原型对象里，把那些不需要共享的属性和方法存在构造函数里。

故上面代码可改写如下：

可以看出，修改prototype属性会影响他的所有实例的sex的值

实例一旦创建出来就会自动引用prototype对象的属性和方法，所以实例对象的属性一般分为两种：一种是自身的，一种是引用自prototype的。

具体实现：

每当代码读取某个对象的某个属性的时候，都会执行一次搜索。首先从对象实例本身开始，如果在实例中找到了该属性，则返回该属性的值，如果没找到，则会顺着原型链指针向上，到原型对象中去找，如果找到就返回该属性值。

这里要提一点，如果对象实例添加了一个属性与原型中同名，则该属性会屏蔽掉原型中的同名属性，不会去修改它。使用delete可以删除实例中的属性（delete只能删除对象下的属性，不能删除变量和参数）

* + 1. 原型链

事实上，js完全依靠“原型链”模式来实现继承。

\_proto\_：这个就是原型链指针

Prototype：指向原型对象的属性

Constructor：每一个原型对象都包含一个指向构造函数的指针，就是Constructor

* + 1. 继承实现方式：

为了实现继承，\_proto\_会指向上一层的原型对象，而上一层的结构依然类似，那么就利用\_proto\_一致指向object的原型对象上，object.prototype.\_proto\_ = null；表示到达最顶端。如此形成了原型链继承。

图示逻辑：



总结：

Object是作为众多new出来的实例的基类 function object() { // code }

Function是作为众多function出来的函数的基类 function function() { // code }

构造函数的\_proto\_（包括function.prototype和object.prototype）都指向function.prototype

原型对象的\_proto\_（包括function.prototype和object.prototype）都指向object.prototype

object.prototype.\_proto\_指向null

* 1. this的用法

**函数中的this指向与使用场合有关。总的来说，this就是函数运行时所在的环境对象。**

1. 纯粹的函数调用

这是函数最通常用法，属于全局调用，因此this就代表全局对象。



1. 作为对象方法调用

函数还可以作为某个对象的方法调用，这时this就指向这个上级对象



1. 作为构造函数调用

所谓构造函数就是通过这个函数可以生成一个新对象，这时，this就指向这个新对象。



1. Apply调用

Apply()是函数的一个方法，作用是改变函数的调用对象。他的第一个参数就表示改变后的调用这个函数的对象。因此，这时this指的就是这第一个参数。



Apply()的参数为空时，默认调用全局对象。因此，此时的运行结果为2，证明this指的是全局对象。

* 1. 箭头函数和普通函数有什么区别
     1. 箭头函数基本语法

1. 参数：

* 如果箭头函数没有参数，直接写一个空括号即可。
* 如果箭头函数的参数只有一个，也可以省去包裹参数的括号。
* 如果箭头函数有多个参数，将参数一次用逗号（,）分隔，包裹在括号中即可。

1. 函数体：

如果箭头函数的函数体只有一句代码，就是简单返回某个变量或者返回一个简单地JS表达式，可以省去函数体的大括号{}。

如果箭头函数的函数体只有一条语句并且不需要返回值（最常见的是调用一个函数），可以给这条语句前面加一个void关键字

箭头函数最常见的用处就是简化回调函数。



* + 1. 与普通函数区别

1. 语法更简洁、清晰

从上面语法实例中可以看出，箭头函数的定义要比普通函数定义简洁、清晰。

1. 箭头函数不会创建自己的this

MDN上对箭头函数this的解释：箭头函数不会创建自己的this，所以他没有自己的this，他只会从自己的作用域链的上一层**继承**this。

箭头函数没有自己的this，他会在捕获自己在**定义时**（注意：**是定义时，不是调用时**）所处的外层执行环境的this，并继承这个this值。所以箭头函数中this的指向在他被定义的时候就已经确定了，**之后永远不会改变。**

再看例2：

上面这例子，对象obj的方法a使用普通函数定义的，普通函数作为对象的方法调用时，this指向他所属的对象。所以，this.id就是obj.id，即‘Obj’。

但方法b是使用箭头函数定义的，箭头函数中的this实际是继承的它定义时所处的全局执行环境中的this，所以指向window对象，所以输出‘Global’**【注意：定义对象的大括号{}是无法形成一个单独的执行环境的，他依旧是处于全局环境中】**

1. **箭头函数继承而来的this指向永远不变**

上面的例子就完全可以说明箭头函数继承而来的this指向永远不变。对象obj的方法b是使用箭头函数定义的，这个函数中的this就永远指向它定义时所处的全局执行环境中的this，即便这个函数是作为对象obj的方法调用，this依旧指向window对象。

1. .call()/.apply()/.bind()无法改变箭头函数中的this指向

.call()/.apply()/.bind()方法可以用来动态修改函数执行时this的指向，但由于箭头函数的this定义时就已经确定且永远不会改变。所以使用这些方法永远也改变不了箭头函数this的指向（但是代码不会报错）。

1. 箭头函数不能作为构造函数使用

构造函数的new简单工作内容：

1. JS内部首先会先生成一个对象
2. 再把函数中的this指向该对象
3. 然后执行构造函数中的语句
4. 最终返回该实例

因为箭头函数没有自己的this，它的this其实是继承了外层执行环境中的this，且this指向永远不会改变，所以箭头函数不能作为构造函数使用，或者说构造函数不能定义成箭头函数，否则用new调用时会报错。

1. 箭头函数没有自己的arguments

箭头函数没有自己的arguments对象。在箭头函数中访问arguments实际上获得的是外层局部（函数）执行环境中的值。

上面的例2的1、2、3处打印结果如下：

很明显，普通函数的fun内部的箭头函数fun1中的arguments对象，其实是沿用作用链向上访问的外层fun函数的arguments对象。

1. 箭头函数没有原型prototype
2. 箭头函数不能用作Generator函数，不能使用yeild关键字
   1. addEventListener
      1. 定义：addEventListener()方法用于指向指定元素添加事件句柄（提示：使用removeEventListener()方法来移除addEventListener()方法添加的事件句柄）。
      2. 语法：element.addEventListener(event, function, useCapture)
3. event：必须。字符串，指定事件名（**注意：不要使用“on”前缀。例如：写“click”而不是“onclick”**）
4. function：必须。指定要触发的函数。（提示：当事件对象会作为第一个参数传入函数，事件对象的类型取决于特定的事件。例如，“click”事件属于mouseEvent（鼠标事件）对象）
   * 1. **useCapture：可选，布尔值，指定事件是否在捕获或冒泡阶段执行**
   1. **true——事件句柄在捕获阶段执行**
   2. **false——默认，事件句柄在冒泡阶段执行**
      1. useCapture的作用：
5. 假设当前有一个HTML内容：div、p、span元素都有click事件的响应逻辑

<div>

<p>

<span></span>

</p>

</div>

如果此时的第三个参数为false的时候：点击span依次触发——span>p>div

如果此时的第三个参数为true的时候：点击span依次触发——div>p>span

* + 1. **如何阻止冒泡：**

fuction(evt) {

evt.[stopPropagation()](https://www.w3school.com.cn/jsref/event_stoppropagation.asp); // 不会阻止本dom本事件的其他事件处理函数的执行

evt.stopImmediatePropagation(); // 会导致后续绑定在该dom的其他事件处理函数不再执行；

**解析：当我给一个span标签添加两个addEventListener，当我在第一个addEventListener里面添加evt.[stopPropagation()](https://www.w3school.com.cn/jsref/event_stoppropagation.asp)时，停止冒泡，但是第二个addEventListener仍然会继续；而evt.stopImmediatePropagation()则连第二个addEventListener都不会执行了**

}

* + 1. 与on的区别：

以click为例：

1. **on事件会被后面的on的事件覆盖，addEventListener则会全部打印**：

obj.onclick(function() {

alert(‘aaa’);

});

obj.onclick(function() {

alert(‘bbb’);

});

打印结果:‘bbb’

obj.addEventListener(“click”, fn(){

alert(‘aaa’);

});

obj.addEventListener(“click”, fn(){

alert(‘bbb’);

});

打印结果:‘aaa’‘bbb’

* 1. 闭包：
     1. 定义：简单来说就是有权访问另一个函数作用域中变量的函数，MDN上面这么说：闭包是一种特殊的对象。
     2. 理解：**外部函数调用之后，其变量对象本应该被销毁，但是闭包的存在就使得我们仍然可以访问外部函数的变量对象。**
     3. 举例： 
     4. 产生一个闭包：创建一个闭包最常见的方式就是一个函数内部创建另一个函数，右图例子中的closure就是一个闭包：
     5. 作用域：闭包的作用域包含着他自己的作用域，以及包含他的函数的作用域和全局作用域。



* + 1. 注意事项：通常函数的作用域及其所有变量都会在函数执行结束后被销毁。但是，在创建了一个闭包后，这个函数的作用域就会一直保存在闭包不存在为止。举例：

右图所示代码可以看到add5和add10都是闭包。他们共享相同的函数定义，但是保存了不同的环境。在add5的环境中，x为5。而在add10中，x为10。最后通过null释放了add5和add10对闭包的引用。在JavaScript中，如果一个对象不再被引用，那么这个对象就会被垃圾回收机制回收；如果两个对象相互引用，而不再被第三者引用，那么这两个相互引用的对象也会被回收。

* + 1. **闭包只能取得包含函数中的任何变量中的最后一个值**

如下图：arr数组中包含了10个匿名函数，每个匿名函数都能访问到外部函数的变量i。

当arr()执行完毕之后，其作用域被销毁，但他的变量对象仍然保存在内存中，得意被匿名访问，这时候的i值为10。

要想保存在循环过程中的每一个i值，需要在匿名函数外部再套一个匿名函数，在这个匿名函数中定义另一个变量并且立即执行来保存i值。如图的brr()。



* + 1. this对象

如右图所示：在上面这段代码中，obj1.getName()()实际上是在全局作用域中调用了匿名函数，this指向了window（可以理解为：这个匿名函数其实仅仅只是写在obj1里面而已。我把这个匿名函数拿到打印这里执行，其实此时是跟obj1没有任何关系了，这时候的this指向肯定指的是全局的name1而不是obj1里面的name1了）。

这里要清楚：**函数名与函数功能是分割开的，不要认为函数在哪里，其内部的this就指向哪里。**

相比obj2.getName()()，执行函数内部的that是父函数保存的this（因为闭包本身可以保存函数外的变量嘛），所以可以指向obj内部。

obj3.getName()是在对象obj3的环境中执行的，所以this指向obj。而(obj3.getName = obj3.getName)赋值语句返回的是等号右边的值，而右边的值是一个函数且此时的函数还没有执行就已经在全局作用域中返回，所以(obj3.getName = obj3.getName)();的this指向全局。要把函数名和函数功能分隔开。

* + 1. 内存泄漏

闭包会引用包含函数的整个变量对象，如果闭包的作用域中保存着一个HTML元素，那么就意味着该元素无法被销毁。我们有必要在对这个元素的操作完之后主动销毁。

* + 1. 函数内部定时器

当函数内部的定时器引用了外部函数变量对象时，该对象不会被销毁

* + 1. 闭包的应用

应用闭包的主要场合是：**设计私有的方法和变量。**任何在函数中定义的变量，都可以认为是私有变量，因为不能在函数外部访问这些变量，私有变量包括函数的参数、局部变量和函数内定义的其他函数。

把有权访问私有变量的公有方法成为**特权方法**

在这里，需要理解两个概念：

**模块模式：**为单例创建私有变量和方法。

**单例：**指的是只有一个实例的对象。JavaScript一般以对象字面量的方法来创建一个单例对象。

**匿名函数最大的用途就是创建闭包，并且还可以构建命名空间，以减少全局变量的使用。从而使用闭包模块化代码，减少全局变量的污染。**

下面这段代码中的addEvent和removeEvent都是局部变量，但我们可以通过全局变量objEvent使用它，这就大大减少了全局变量的使用，**增强网页的安全性。**

* + 1. **运用闭包的关键**

**闭包引用外部函数变量对象中的值；**

**在外部函数的外部调用闭包。**

* + 1. **闭包的缺陷**

闭包的缺点就是常驻内存会增大内存使用量，并且使用不当很容易造成内存泄漏。

如果不是因为某些特殊任务而需要闭包，在没有必要的情况下，在其他函数中创建函数是不明智的，因为闭包对脚本性能具有负面影响，包括处理速度和内存消耗。

* + 1. 面试题



* 1. Promise：
     1. 含义：本质上，一个promise是某个函数返回的对象，你可以把回调函数绑定在这个对象上，而不是把回调函数当做参数传进函数
     2. 执行顺序原理：**同步优先，异步靠边（setTimeout可算异步）**
     3. 规范：

1. 一个promise可能有三种状态：等待（pending）、已完成（resolved/fulfilled）、已拒绝（rejected）
2. 一个promise的状态只可能从“等待”转到“完成”或“拒绝”。不能逆向转换；同时，“完成”和“拒绝”两种状态不能相互转换
3. Promise必须实现then方法（可以说，then就是promise的核心），而且then必须返回一个promise，同一个promise的then可以调用多次，并且回调的执行顺序跟着它们被定义时的顺序一致
4. Then方法接收两个参数，第一个是成功时候的回调（在resolved状态调用），另一个是失败时候的回调（在rejected状态调用）。
   * 1. promise.resolve()和promise.reject() 两者都是new Promise()的快捷方式。
     2. Promise.all()和Promise.race()
5. promise.all()：接收一个promise对象的数组作为参数，当这个数组的所有Promise对象全部变成resolve的时候，该方法才resolve。

如果其中一个promise对象为reject的话，则该方法为reject。

1. Promise.race()：接收一个promise对象的数组作为参数，此时的目的只是为了拿取到结果即可，不在乎是执行的哪个promise（只要有一个promise执行成功，剩余的都不用执行了）。



* + 1. 关于catch的提问：catch后面还能不能继续catch到错误？Catch后还能不能继续then？then后面的catch还能不能catch到then的错误？

1. 如果异步操作抛出错误，状态就会变成rejected，就会调用catch方法指定的回调函数处理错误。
2. Then方法指定的回调函数在运行中报错也会被catch捕获到。
3. 在resolve()后面抛出的错误会被忽略（如果前面已经是执行了resolve，那么后面throw出来的error就不会被catch）
4. Catch返回的也是一个promise，因此后面可继续then（不管是then方法还是catch方法返回的都是一个新的promise实例，这意味着promise可以**链式调用**then和catch，每一个方法的返回值作为下一个方法的参数）
5. 在异步函数中抛出的错误不会被catch捕获到（例如使用了setTimeOut之类的，抛出来的错误不会被catch到）
6. 有多个catch连写，如果在catch中继续throw出异常，那么后面的catch就会一直执行，直到不再抛出异常。（catch其实是then的语法糖）
7. **vue面试汇总**
   1. 谈谈你对MVVM的理解：
      1. MVVM分为model、view、viewModel三者

1）Model代表数据模型，数据和业务逻辑都在model层中定义；

2）View代表UI视图，负责数据的展示；

3）ViewModel负责监听Model中数据的改变并且控制视图的更新，处理用户交互操作；

4）Model和View并没有直接关联，而是通过ViewModel来进行联系的，Model和ViewModel之间有着双向数据绑定的联系。因此当Model中的数据改变时会触发View层的刷新，View中由于用户交互操作而改变的数据也会在Model中同步。

**这种模式实现了Model和View的数据自动同步，因此开发者只需要专注对数据的维护操作即可，而不需要自己操作dom**

* + 1. MVVM和MVC区别

1. 后端MVC与前端MVC

前端的MVC其实是为了解决前端复杂的JS模块化的问题，从后端MVC的V分出来的MVC，与后端MVC并没有直接的关系。前端的MVC中，M占的比例很低，只指代数据。而后端V的比例很低，只有模板的部分。

1. C：controller——负责监听View的用户事件，得到数据后的controller做一些处理，然后渲染View。但是随着逻辑的复杂，这样处理遇到很难调试的问题的话，由于View一定要在UI环境下，而且Model或者Controller和View强耦合，没办法单独验证应用逻辑的正确性。当出了问题之后，因为各个模块是耦合在一起的，也不能快速判断究竟是哪个模块出现的问题。

个人总结：VM的出现降低了C的存在感，在MVC中，Controller的作用是1、管理自己的生命周期；2、处理Controller之间的跳转；3、实现Controller容器。在MVVM中，VM作用是：处理数据解析。Controller不再像MVC那样直接持有model了，想象Controller是一个boss，数据是一堆文件（model），如果现在是MVC，那么数据解析（比如整理文件）需要Boss亲自完成，然而实际上Boss需要的仅仅是整理好的文件而不是那一堆乱七八糟的整理前的文件。所以Boss招聘了一个秘书，现在Boss就不再需要管理原始数据（即整理那堆乱七八糟的文件）了，他只需要交给秘书即可。故此，秘书就是VM了，并且Controller（Boss）现在只需要直接持有VM而不需要再持有M。如果再进一步理解C、VM、M之间的关系：因为Controller只需要数据解析的结果而不再乎过程，所以就相当于VM把“如何解析Model”给封装起来了，C根本不需要知道M的存在就能把工作做好，前提是它需要持有一个VM。那我们MVVM的持有关系就是C持有VM，VM持有M。一旦在实现Controller的过程中遇到任何跟Model（或者数据）相关的问题，就找VM要答案。

* 1. Vue有哪些指令：

v-html、v-show、v-if、v-for、v-else、v-bind等等

* 1. v-if和v-show的区别：

v-show仅仅是控制元素的显示方式，将display属性在block和none之间来回切换；而v-if会直接控制这个dom节点的存在与否。当我们需要经常切换某个元素的显示/隐藏的时候，使用v-show会更加节省性能上的开销；当只需要一次显示或者隐藏的时候，使用v-if更加合理。

* 1. 简述vue的响应式原理（即双向绑定的原理）

当一个vue实例创建时，vue会遍历data选项的属性，用Object.defineProperty将他们转为getter/setter并在内部追踪相关依赖，在属性被访问和修改时通知变化。

每个组件实例都有相应的watcher程序实例，他会在组件渲染的过程中把属性记录为依赖，之后当依赖项的setter被调用时，会通知watcher重新计算，从而致使它关联的组件得以更新。

* 1. Vue中如何实现在组件内部实现一个双向数据绑定？

父组件通过props传值给子组件，子组件通过$emit来通知父组件修改相应的props值。

* 1. Vue中如何监控某个值的变化？
     1. watch：注意：如果监听某个对象的话，要用深层遍历——deep:true，但仅仅是监听某个对象值的话，直接：

’obj.x’: {

handler(newVal, oldVal) {

// ……

}

}

（2）computed：利用计算属性的特性来实现

* 1. Vue中给data中的对象属性添加一个新的属性时会发生什么，如何解决？

this.$set(obj，key，value);

$set()方法调用时，可以触发页面全部重新渲染。

* 1. delete和vue.delete删除数组的区别

delete只是被删除的元素变成了empty/undefined 其他的元素键值还是不变。

Vue.delete直接删除了数组改变了数组的键值。

* 1. 如何优化单页面（SPA）应用的首屏加载速度慢的问题
     1. 将公用的JS库通过script标签外部引入，减少app.bundel的大小，让浏览器并行下载资源文件，提高下载速度；
     2. 在配置路由时，页面组件也使用懒加载的方式引入，进一步缩小app.bundel的体积，在调用某个组件时再加载对应的js文件；



* + 1. 加一个首屏loading图，提升用户体验；
    2. 补充：**vue本身就是单页面（SPA）应用跟多页面（MPA）应用的区别**：
  1. Vue-router的实现原理
     1. 单页面应用和多页面应用

1. 单页面：

即第一次进入页面的时候会请求一个HTML文件，刷新清除一下。切换到其他的组件，此时路径也相应变化，但是并没有新的HTML文件请求，页面内容也变化了。

**原理**：JS会感知到URL的变化，通过这一点，可以用js动态的将当前页面的内容清除掉，然后将下一个页面的内容挂载到当前页面上，这时候的路由不是由后端来做的了，而是由前端来做，判断页面到底显示哪个组件；清除不需要的；加载需要的组件。这个过程就是单页应用，每次跳转不需要再请求HTML文件。

1. 多页面

即每一次页面跳转的时候，后台服务器都会给返回一个新的HTML文档，这种类型的网站，也叫做多页应用。

原理：传统的页面应用，是用一些超链接来实现页面切换和跳转的

* + 1. vue-router的钩子函数

首页可以控制导航跳转，beforeEach、afterEach等，一般用于页面title的修改。一些需要登录才能调整页面的重定向功能。

beforeEach主要有三个参数：to、from、next：

To：router即将进入的目标路由对象

From：router当前导航正要离开的路由

Next：function一定要调用该方法resolve这个钩子。执行效果依赖于next方法的调用参数。可以控制网站的跳转。

* + 1. $route和$router区别

$route是“路由信息对象”，包括path、params、hash、query、fullPath、matched、name等路由信息参数。而$router是“路由实例对象”，对象包括路由的跳转方法，钩子函数等。

* + 1. 怎么定义vue-router的动态路由？

在router目录下的index.js文件中，对path属性加上/:id，使用router对象的params.id获取。

* + 1. vue-router实现原理核心：**更新视图但是不重新请求页面**

1. 路由模式：
   1. Hash：使用URL hash值来做路由，**默认模式**
   2. History：依赖HTML5 History API和服务器配置。具体查看HTML5 History模式
   3. Abstract：支持所有JavaScript运行环境，如Node.js服务器端
2. Hash模式：

Hash即浏览器的URL中“#”后面的内容，包含“#”。Hash是URL中的锚点，代表的是网页中的一个位置，单单改变“#”后面的部分，浏览器只会加载相应位置的内容，不会重新加载页面。

也就是说：

* 即“#”是用来指导浏览器动作的，对服务器完全无用，HTTP请求中不包含“#”
* 每一次改变“#”后面的部分，都会在浏览器的访问历史中增加一个记录，使用“后退”按钮，就可以回到上一个位置。

所以说hash模式通过锚点值的改变，根据不同的值，渲染指定的dom位置的不同数据。

1. History模式：

HTML5 History API提供了一种功能，能让开发人员在不刷新整个页面的情况下修改站点的URL，就是利用history.pushState API来完成URL跳转而无需重新加载页面；

由于hash模式会在URL中自带“#”，如果不想要很丑的hash，我们可以用路由的history模式，只需要在配置路由规则时，加入：“mode：’history’”，这种模式充分利用history.pushStatus API来完成URL跳转而无需重新加载页面。

有时，history模式下也会出现问题：

Hash模式下：xxx.com/#/id=5请求地址为xxx.com,没有问题。

History模式下：xxx.com/id=5 请求地址为xxx.com/id=5,如果后端没有对应的路由处理，就会返回404错误；

为了应对这种情况，需后端的配置支持：

在服务端增加一个覆盖所有情况的候选资源：如果URL匹配不到任何静态资源，则应该返回一个index.html页面，这个页面就是你APP依赖的页面。

1. Abstract模式：

Abstract模式是使用一个不依赖于浏览器的浏览历史虚拟管理后端。

根据平台差异可以看出，在weex环境中只支持使用abstract模式。不过，在vue-router自身会对环境做校验，如果发现没有浏览器的API，vue-router会自动强制进入abstract模式，所以在使用vue-router时只要不写mode配置即可，默认会在浏览器环境使用hash模式，在移动端原生环境中使用abstract模式。（当然也可以明确指定在所有情况下都使用abstract模式）。

* + 1. Hash模式和history模式实现vue-router跳转API的区别
  1. Vue3.0和2.0双向绑定的区别，这样改动有什么好处

在vue2.0中双向绑定是利用Object.defineProperty来实现双向数据绑定原理，而在vue3.0是利用proxy这个对象实现的。（**IE系列不兼容proxy**）

在vue中，Object.defineProperty无法监控到数组下标的变化，导致直接通过数组的下标给数组设置值，不能实时响应。

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里，是通过递归以及遍历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，不管对操作性还是性能都会有一个很大的提升。

而取代他的Proxy有以下两个优点：

* 可以劫持整个对象，并返回一个新对象
* 有13种劫持操作
  + 1. 什么是Proxy

Proxy是ES6中新增的一个特性，翻译过来就是“代理”，用在这表示由他来“代理”某些操作。Proxy让我们能够以简洁易懂的方式控制外部对对象的访问。其功能非常类似于设计模式中的代理模式。

Proxy可以理解成，在目标对象之前架设一层“拦截”，外界对该对象的访问都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写。

使用Proxy的核心优点是可以交由他来处理一些非核心逻辑（如：读取或设置对象的某些属性前记录日志；设置对象的某些属性值前，需要验证；某些属性的访问控制等）。从而可以让对象只需关注于核心逻辑，达到关注点分离，降低对象的复杂度等目的。

* + 1. 基本用法

Let p = new Proxy(target, handler);

参数：

* target：是用Proxy包装的被代理对象（可以是任何类型的对象，包括原生数组函数，甚至另一个代理）。
* handler：是一个对象，其声明了代理target的一些操作，其属性是当执行一个操作时定义代理行为的函数。

p是Proxy对象，当其他操作对P进行更改的时候，会执行handler对象的方法。Proxy有13种数据劫持的操作，常用的handler处理方法：

* get：读取值
* set：获取值
* has：判断对象是是否拥有该属性
* construct：构造函数

图示第二段代码里把代理器返回的对象代理到this.$data,即this.$data是代理后的对象，外部每次对this.$data进行操作时，实际上执行的是这段代码里handler对象上的方法。（注：reflect属性也是ES6的，详情查reflect属性）

* 1. 虚拟dom为什么效率高？
     1. 什么是虚拟dom？

官方概念：Vue通过虚拟DOM树对真实的DOM发生的变化保持追踪。

一棵真实的DOM树的渲染需要先解析CSS样式和DOM树，然后将其整合形成一棵渲染树，在通过布局算法去计算每个节点在浏览器中的位置，最终输出到显示器上。

而虚拟DOM则可以理解为保存了一棵DOM数被渲染之前所包含的所有信息，而这些信息可以通过对象的形式一直保存在内存中，并通过JavaScript的操作进行维护，提高页面更新速度。

简单来讲：虚拟DOM技术的核心就是**使用JS对象来代替DOM节点。**

* + 1. 优缺点：

1. 优点：

DOM节点在HTML文档中的表现通常如下：

DOM节点也可以表示为一个JavaScript对象，如下：

当我们更新节点时：

如果我们用一个虚拟的DOM，而不是直接调用 .getElementById的方法，这样只操作JavaScript对象，这样是相当方便的。

然后，再把更改的部分更新到真正的DOM，方法如下：

1. 缺点：

尺寸：更多的功能意味着更多的代码。（但是目前VUE 2.x仍然相当小）

内存：虚拟DOM需要在内存中维护一份DOM的副本。在DOM更新速度和使用内存空间之间取得平衡。

* + 1. 应用场景：

虚拟DOM不是适合所有场景的

如果虚拟DOM需要大量修改，这是合适的。

如果是单一的，频繁更新的话，虚拟DOM将会花费更多的时间处理计算的工作。但对于大多数的单页面应用，这应该都会更快。

* 1. 对VUE生命周期的理解
     1. 什么是vue的生命周期

Vue实例从创建到销毁的过程，就是生命周期。从开始创建、初始化数据、编译模板、挂载DOM->渲染、更新->渲染、销毁等一系列过程，成为Vue的生命周期。

* + 1. 生命周期的作用

他的生命周期中有多个事件钩子，让我们在控制整个Vue实例的过程时更容易形成好的逻辑。

* + 1. 简述各阶段的生命周期

Vue的生命周期共分为八个阶段 创建前/后、载入前/后、更新前/后、销毁前/后

创建前/后：即beforeCreated/created，在beforeCreated阶段，VUE实例的挂载元素el还没有。此时可以添加loading事件，在created阶段发起后端请求拿回数据。

载入前/后：在beforeMount阶段，vue实例的$el和data都初始化，但是挂载之前为虚拟的dom节点，data.message还未替换，页面无重新渲染。在mounted阶段，vue实例挂载完成，data.message成功渲染。

更新前/后：当data变化时，会触发beforeUpdate和updated方法。

销毁前/后：在执行destroy方法后，对data的改变不会再触发周期函数，说明此时的vue实例已经解除了事件监听以及dom的绑定，但是dom结构依然存在。

* 1. keep-alive是否与普通组件有一样的生命周期，如果不是，它有哪些钩子？
     1. 当组件外面加了keep-alive，页面第一次进入的时候，钩子触发的顺序是created->mounted->activated

页面退出的时候会触发deactivated，当再次前进或者后退的时候只会触发activated

* + 1. 在vue 2.1.0版本之后，keep-alive新加入两个属性：include（包含的组件缓存）与exclude（排除的组件不缓存，优先级大于include）

Include：字符串或正则表达式，只有名称匹配的组件会被缓存

Exclude：字符串或正则表达式，任何名称匹配的组件都不会被缓存

include和exclude的属性允许组件有条件的缓存。二者都可以用“，”分隔字符串、正则表达式、数组。当使用正则或者数组时，要记得v-bind

* 1. Vue.js的两个核心是什么

数据驱动（靠数据驱动双向绑定，省去操作DOM）、组件系统（组件化，扩展HTML元素，封装可重用代码）

* 1. Vue常用的修饰符
* .prevent：提交事件不再重载页面；
* .stop：阻止单击事件冒泡；
* .self：当事件发生在该元素本身而不是子元素的时候会触发；
* .capture：事件侦听，事件发生的时候会调用
  1. Vue中key值的作用

当Vue.js用v-for正在更新已渲染过的元素列表时，它默认用“就地复用”策略。如果数据项的顺序被改变，Vue将不会移动DOM元素来匹配数据项的顺序，而是简单复用此处每个元素，并确保他在特定的索引下显示已被渲染过的每个元素。**Key的作用主要是为了更高效的更新虚拟DOM。**

* 1. Vue如何实现SEO？

明确三个概念：【后端渲染】指传统的ASP、Java或PHP的渲染机制；【前端渲染】值使用JS来渲染页面大部分内容，代表是现在流行的SPA单页面应用；【同构渲染】指前后端共用JS，首次渲染时使用Node.js来直出HTML。一般来说同构渲染是介于前后端中的共有部分。

* + 1. 网站渲染模式——服务端渲染和客户端渲染

1. 服务端渲染（后端渲染SSR）

后端渲染HTML的情况下，浏览器会直接接收经过服务器计算之后的呈现给用户的最重的HTML字符串，这里的计算就是服务器经过解析存放在服务器端的模板文件来完成的。在这种情况下，浏览器只进行了HTML的解析，以及通过操作系统提供的操纵显示器显示内容的系统调用在显示器上把HTML所代表的图像显示给用户。

* 好处：前端耗时少（只负责将HTML进行展示），利于SEO
* 坏处：网络传输数据量大，占用（部分/少部分）服务器运算资源，response出的数据量会大点，模板改了前端的交互和样式等都要联动修改。

1. 客户端渲染（前端渲染CSR）——SPA（单页面应用）

前端渲染就是指浏览器会从后端得到一些信息，这些信息或许是angular.js的模板文件，亦或是JSON等各种数据交换格式所包装的数据，甚至是直接的合法的HTML字符串。这些形式都不重要，重要的是，将这些信息组织排列形成最终可读的HTML字符串是由浏览器来完成的，在形成了HTML字符串之后，在进行显示。

* 好处：网络传输数据量小（减少了服务器压力）
* 坏处：前端耗时较多，不利于SEO，SPA最低支持IE9
  + 1. 前端渲染遇到的问题——SEO和首屏问题

1. SSR

首先vue2.x直接提供一套完整的构建vue服务端渲染，vue.js是构建客户端应用程序的框架。默认情况下，可以在浏览器中输出Vue组件，进而生成DOM和操作DOM。然而，也可以将同一个组件渲染为服务器端的HTML字符串，将他们直接发送到浏览器，最后将静态标记“混合”为客户端上完全交互的应用程序。服务器渲染的vue.js应用程序也可以被认为是“同构”或“通用”，因为应用程序大部分代码都可以在服务器和客户端上运行。

1. Nuxt.js

Nuxt.js是一个基于Vue的框架，大部分都是Vue的东西，不需要配置webpack那些繁琐的东西，很大程度上让vue-ssr更亲民。

* + 1. 预渲染的方式——Prerender（解决SEO，首屏问题）

由于页面较少，且预渲染相对SSR比较简单，预渲染可以极大的提高网页访问速度。配合一些meta插件（vue-meta-info、Prerender-spa-plugin预渲染），完全可以满足SEO需求。

注：Prerender服务需要有Node.js环境支持，**Prerender需要路由使用history模式，使用hash模式会导致失效。**

通俗的说：预渲染的本质就是在打包的时候，通过插件将JS提前执行一遍，然后再将有内容的HTML放到服务器上，这样爬虫可以抓取到，首页的问题也可以得到解决。

在webpack配置如下：

* + 1. Phantom.js
  1. Computed原理、缓存问题

在模板中放入太多的逻辑会让模板过重且难以维护，在需要对数据进行复杂的处理，且可能多次使用的情况下，尽量采用计算属性的方式。

* + 1. 使用Computed的好处：
* 使得数据处理结构清晰；
* 依赖数据，数据更新，处理结果自动更新；
* 计算属性内部this指向vm实例；
* 在template调用时，直接写计算属性名即可；
* 常用的是getter方法，获取数据，也可以使用set方法改变数据；
* 相较于methods，不管依赖数据变不变methods都会重新计算，但是依赖数据不变时，computed从缓存中获取，不会重新计算。
  + 1. Computed也是响应式的

简单来说，computed本身是有设置get（取值）、set（赋值）函数的，会跟Object.defineProperty关联起来。所以Vue能捕捉到读取computed和赋值computed的操作。

读取computed时，会执行你设置的get函数，但是并没有这么简单，因为还有一层缓存操作；赋值computed时，会执行你设置的set函数，就这个比较简单，会直接把set赋值给Object.defineProperty-set

* + 1. 如何控制缓存

Computed是有缓存的。官方解释：**计算属性是基于他们的依赖进行缓存的。计算属性只有在他相关依赖发生改变时才会重新求值。**

假设我们有一个性能开销比较大的计算属性A，它需要遍历一个巨大的数组并做大量的计算。然后我们可能有其他的计算属性依赖于A，如果没有缓存，我们将不可避免的多次执行A的getter

判断computed是否使用缓存的依据：

首先，computed计算后，会把计算得到的值保存到一个变量中。读取到computed时便直接返回这个变量。当使用缓存时，就直接返回这个变量。当computed更新时，就会重新赋值更新这个变量。

Computed计算就是调用你设置的get函数，然后得到返回值。Computed控制缓存的重要一点是**【脏数据标志位dirty】**，dirty是watcher的一个属性

当dirty为true时，读取computed会重新计算

当dirty为false时，读取computed会使用缓存

一开始每个computed新建自己的watcher时，会设置watcher.dirty = true，以便于computed被使用时，会计算得到值。

当依赖的数据变化了，通知computed时，会设置watcher.dirty = true，以便于其他地方重新渲染，从而重新读取computed重新计算

Computed计算完成后，会设置watcher.dirty = false，以便于其他地方再次读取的时候，使用缓存，免于计算

* + 1. 依赖的data变化，computed怎么更新

1）场景设置

现在 页面A引用 computedB，computedB依赖了dataC。依赖顺序：A->B->C

2）详细流程

dataC开始变化后

1. 通知computedB 的watcher更新，其实只会重置脏数据标志位dirty=true，不会计算值
2. 通知页面A watcher进行更新渲染，进而重新读取computedB，然后computedB开始重新计算。
3. 为什么dataC能通知页面A？

dataC的依赖收集器会同时收集computedB和页面A的watcher

1. 为什么dataC能收集到页面A的watcher？

在页面A在读取computedB的时候，computed趁机把页面A也顺带介绍给dataC认识了，于是页面A的watcher和dataC间接牵在了一起，于是dataC就会收集到页面A的watcher

1. 所以computed如何更新？

**被依赖通知更新后，重置脏数据标志位，页面读取computed时再更新值。**

* + 1. 总结

1. **Computed通过watcher.dirty控制是否读取缓存**
2. **Computed会让【data依赖】收集到【依赖computed的watcher】，从而data变化时，会同时通知computed和依赖computed的地方**
   1. Vuex
      1. 什么是vuex

Vuex是一个专为Vue.js应用程序开发的状态管理模式（我的理解更倾向于全局状态管理）。它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。

* + 1. 使用场景
* 虽然Vuex可以帮助我们管理共享状态，但也附带了更多的概念和框架。这需要对短期和长期的效益进行权衡。
* 如果不打算开发大型单页应用，使用Vuex可能是繁琐冗余的。如果应用很简单，用eventBus即可，就是一个创建事件中心，相当于中转站，可以用它来传递事件和接收事件。（主要用于项目小或者非父子组件间的数据传递，兄弟组件传值）
  1. $nextTick原理

$nextTick主要原理是异步更新，说的直白一点有点类似于setTimeout(fn, 0)；当你处理一个比较耗时的操作的时候，为了保证能够在耗时操作完成的时候做某些事情，一般会用setTimeout(fn, 0)。因为vue的内部就是先收集这些变化的变量，然后处理他们（去重等），处理完之后才会修改DOM上的内容，我们可以把它们理解为一个比较耗时的同步操作，然后$nextTick就是把一些处理方法添加到耗时操作完成之后去执行。

* + 1. 监听同步操作完成的方法：

Promise.resolve().then()

setTimeout(fn, 0)

MutationObserver

* + 1. MutationObserver

1. 概述：它是用来监视DOM变动。DOM的任何变动，比如节点的增减，属性的变动，这个API都可以得到通知。概念上，它很接近事件，可以理解为DOM发生变动就会触发MutationObserver事件。但是，他与事件有一个本质不同：事件是同步触发，也就是说，DOM的变动立刻会触发相应的事件；MutationObserver则是异步触发，DOM的变动并不会马上触发，而是要等到当前所有DOM操作都结束才触发。其目的是为了应付DOM变动频繁的特点。
2. 特点：

* 他等待所有脚本任务完成后才会运行（即异步触发方式）。
* 它把DOM变动记录封装成一个数组进行处理，而不是一条条个别处理DOM变动。
* 它既可以观察DOM的所有类型变动，也可以指定只观察某一类变动。

1. **性能优化**
   1. 前端如何优化网站性能？
      1. 减少HTTP请求数量

在浏览器与服务器通信时，主要是通过HTTP进行通信。浏览器与服务器需要经过三次握手，每次握手都需要花费大量的时间。而且不同的浏览器对资源文件并发请求数量有限（不同浏览器允许的并发数不同），一旦HTTP请求数量达到一定的数量，资源请求就存在等待状态，这是很致命的，因此减少HTTP的请求数量可以很大程度上对网站性能进行优化。

1. CSS Sprites：国内俗称CSS精灵，这是将多张图片合并成一张图片达到减少HTTP请求的一种解决方案，可以通过CSS background属性来访问图片内容，这种方案同时还可以减少图片总字节数。
2. 合并CSS和JS文件：现在前端有很多工程化打包工具，如：grunt、gulp、webpack等。为了减少HTTP请求数量，可以通过这些工具在发布之前将多个CSS或者多个JS文件合并成一个文件。
3. 采用lazyLoad：俗称懒加载，可以控制网页上的内容在一开始无需加载，不需要发请求，等到用户操作真正需要的时候立即加载出内容，这样就控制了网页资源的一次性请求数量。
   * 1. 控制资源文件加载优先级

浏览器在加载HTML内容时，是将HTML内容从上至下依次解析，解析到link或者script标签就会加载herf或者src对应链接内容，为了第一时间展示页面给用户，就需要将CSS提前加载，不要受JS加载影响。

**一般情况下都是CSS在头部，JS在底部。**

1. 利用浏览器缓存

浏览器缓存是将网络资源存储在本地，等待下次请求该资源时，如果已经存在就不需要到服务器重新请求该资源，直接在本地读取该资源。

1. 减少重排（Reflow）

基本原理：重排是DOM的变化影响到了元素的集合属性（宽和高），浏览器会重新计算元素的几何属性，会使渲染树中受到影响的部分失效，浏览器会验证DOM树上的所有其他节点的visibility属性，这也是Reflow低效的原因如果Reflow的过于频繁，CPU使用率就会急剧上升。

**减少Reflow，如果需要在DOM操作时添加样式，尽量使用增加class属性，而不是通过style操作样式。**

1. 减少DOM操作
2. 图标使用iconfont替换
   1. 网页从输入网址到渲染完成经历了哪些过程？

大致可以分为如下7个步骤：

1. 输入网址；
2. 发送到DNS服务器，并获取域名对应的web服务器对应的IP地址；
3. 与web服务器建立的TCP连接；
4. 浏览器向web服务器发送http请求；
5. Web服务器响应请求，并返回指定URL的数据（或错误信息，或重定向的新的URL地址）；
6. 浏览器下载web服务器返回的数据及解析HTML源文件；
7. 生成DOM树，解析CSS和JS，渲染页面，直至显示完成。
   1. jQuery获取的dom对象和原生的dom对象有何区别？

Js原生获取的dom是一个对象，jQuery对象就是一个数组对象，其实就是选择出来的元素的数组集合，所以说他们两者是不同的对象类型不等价。

原生的DOM对象转jQuery对象：（如下左图）

jQuery对象转原生DOM对象：（如上右图）

* 1. jQuery如何扩展自定义方法
  2. 