HTML 知识点

1. html基本结构

- html标签是由<>包围的关键词。
- html标签通常成对出现，分为标签开头和标签结尾。
- 有部分标签是没有结束标签的，为单标签，单标签必须使用 / 结尾。
- 页面所有的内容，都在html标签中。
- html标签分为三部分：标签名称，标签内容，标签属性。
- html标签具有语义化，可通过标签名能够判断出该标签的内容，语义化的作用是网页
- 结构层次更清晰，更容易被搜索引擎收录，更容易让屏幕阅读器读出网页内容。
- 标签的内容是在一对标签内部的内容。
- 标签的内容可以是其他标签。

2. 标签属性

- class属性：用于定义元素的类名
- id属性：用于指定元素的唯一id，该属性的值在整个html文档中具有唯一性
- style属性：用于指定元素的行内样式，使用该属性后将会覆盖任何全局的样式设定
- title属性：用于指定元素的额外信息
- accesskey属性：用于指定激活元素的快捷键
- tabindex属性：用于指定元素在tab键下的次序
- dir属性：用于指定元素中内容的文本方向，属性只有`ltr`或`rtl`两
- lang属性：用于指定元素内容的语言

3. 事件属性

window窗口事件

- onload，在网页加载结束之后触发
- onunload，在用户从网页离开时发生（点击跳转，页面重载，关闭浏览器窗口等）

form表单事件

- onblur，当元素失去焦点时触发
- onchange，在元素的值被改变时触发
- onfocus，当元素获得焦点时触发
- onreset，当表单中的重置按钮被点击时触发
- onselect，在元素中文本被选中后触发
- onsubmit，在提交表单时触发

keyboard键盘事件

onkeydown，在用户按下按键时触发

onkeypress，在用户按下按键后，按着按键时触发。(该属性不会对所有按键生效，不生效的有，alt，ctrl，shift，esc)

mouse鼠标事件

- onclick，当在元素上发生鼠标点击时触发
- onblclick，当在元素上发生鼠标双击时触发
- onmousedown，当元素上按下鼠标按钮时触发
- onmousemove，当鼠标指针移动到元素上时触发
- onmouseout，当元素指针移出元素时触发
- onmouseup，当元素上释放鼠标按钮时触发。Media媒体事件
- onabort，当退出时触发
- onwaiting，当媒体已停止播放但打算继续播放时触发。

4. 文本标签

- 段落标签：`<p></p>`，段落标签用来描述一段文字
- 标题标签：`<hx></hx>` ，标题标签用来描述一个标题，标题标签总共有六个级别，`<h1></h1>` 标签在每个页面中通常只出现一次
- (3)强调语句标签：`<em></em>`，用于强调某些文字的重要性
- (4)更加强调标签：`<strong></strong>` 和 `<em>` 标签一样，用于强调文本，但它强调的程度更强一些
- (5)无语义标签：`<span></span>`，标签是没有语义的
- (6)短文本引用标签：`<q></q>`，简短文字的引用
- (7)长文本引用标签：`<blockquote></blockquote>`，定义长的文本引用
- (8)换行标签：`<br/>`

5. 多媒体标签

- 链接标签：`<a></a>`
- 图片标签：`<img/>`
- 视频标签：`<video></video>`
- 音频标签：`<audio></audio>`

6. 列表标签

- 无序列表标签，ul,li，`<ul></ul>` 列表定义一个无序列表，`<li></li>` 代表无需列表中的每一个元素
- 有序列表，ol,li
- 定义列表，`<dl></dl>`，定义列表通常和 `<dt></dt>` 和 `<dd></dd>` 标签一起使用

7. 表格标签

- 表格标签 `<table></table>`
- 表格的一行 `<tr></tr>`
- 表格的表头 `<th></th>`
- 单元格 `<td></td>`
- 表格合并，同一行内，合并几列colspan="2"，同一列内，合并几行rowspan="3"

8. 表单标签

表单标签 `<form>`

`<form></form>` 表单是可以把浏览者输入的数据传送到服务器端，这样服务器端程序就可以处理表单传过来的数据。

```
<form method="传送方式" action="服务器文件">
```

- action，浏览者输入的数据被传送到的地方
- method，数据传送的方式

输入标签 `<input/>`

name：为文本框命名，用于提交表单，后台接收数据用。

value：为文本输入框设置默认值。

type：通过定义不同的type类型，input的功能有所不同。

- text 单行文本输入框
- password 密码输入框（密码显示为***）
- radio 单选框 （checked属性用于显示选中状态）
- checkbox 复选框（checked属性用于显示选中状态）
- file 上传文件
- button 普通按钮
- reset 重置按钮（点击按钮，会触发form表单的reset事件）
- submit 提交按钮（点击按钮，会吃饭form表单的submit事件）
- email 专门用于输入 e-mail
- url 专门用于输入 url
- number 专门用于number
- range 显示为滑动条，用于输入一定范围内的值
- date 选取日期和时间（还包含：month、week、time、datetime、datetime-local）
  color 选取颜色

button按钮，下拉选择框 `<select></select>`

- `<option value="提交值">` 选项 `</option>` 是下拉选择框里面的每一个选项

文本域： `<textarea></textarea>`

当用户想输入大量文字的时候，使用文本域。cols，多行输入域的列数，rows，多行输入域的行数。

9. 其他语义化标签

- 盒子 `<div></div>`
- 网页头部 `<header></header>`，html5新增语义化标签，定义网页的头部，主要用于布局，分割页面的结构
- 底部信息 `<footer></footer>`，html5新增语义化标签，定义网页的底部，主要用于布局，分割页面的结构
- 导航 `<nav></nav>`，html5新增语义化标签，定义一个导航，主要用于布局，分割页面的结构
- 文章 `<article></article>`，html5新增语义化标签，定义一篇文章，主要用于布局，分割页面的结构

- 侧边栏 `<aside></aside>`，语义化标签，定义主题内容外的信息，主要用于布局，分割页面的结构。
- 时间标签 `<time></time>`，语义化标签，定义一个时间

10. 网页结构

- `<!DOCTYPE html>` 定义文档类型，告知浏览器用哪一种标准解释HTML
- `<html></html>`可告知浏览器其自身是一个 HTML 文档
- `<body></body>` 标签之间的内容是网页的主要内容
- `<head></head>` 标签用于定义文档的头部，它是所有头部元素的容器
- `<title></title>` 元素可定义文档的标题
- `<link>` 标签将css样式文件链接到HTML文件内
- `<meta>` 定义文档的元数据

CSS 知识点

1. CSS 权重及引入方式

CSS权重可以分为四个等级：

- 内联样式，如 style=xxx，权重为 1000
- id选择器，权值为 100
- class、伪类和属性选择器，如 .content, :hover, [attribute]，权值为 10
- 元素选择器和伪元素选择器，权值为 1

需要注意的是：通用选择器(*)，子选择器(>)和相邻同胞选择器(+)并不在这四个等级中，所以它们的权重都为 0。

权重大的选择器优先级也越高，相同权重的优先级又遵循后定义覆盖前面定义的情况。

伪类和伪元素：

一个 : 的为伪类: `:hover`，`:focus`，`:first-of-type`，`:first-child`

两个 :: 为伪元素: `:: before`，`::after`

2. 用CSS画三角形

实现原理：

首先来看一下我们平时为一个盒子添加border的情况：

```
div {
    width: 100px;
    height:100px;
    border: 4px solid blue;
}
```

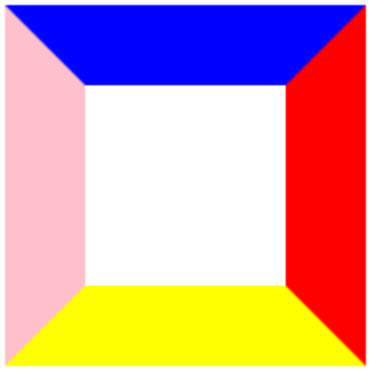
效果如下：

这是我们平时使用 border 最常见的场景——往往只给border一个较小的宽度；然而这样的日常用***让我们对border的形成方式产生误解，即认为border是由四个矩形边框拼接而成的。

然而事实并不是如此。实际上，border是由三角形组合而成的。为了说明这个问题，我们可以增大border的宽度，并为他们设置不同的颜色：

```
div {
    width: 100px;
    height:100px;
    border: 40px solid;
    border-color: blue red yellow pink;
}
```

效果如下：



由此我们可以看出，border确实是由三角形组成的。那么我们把元素的内容尺寸设为0会发生什么呢？

```
div {
    width:0;
    height:0;
    border: 40px solid;
    border-color: blue red yellow pink;
}
```

效果如下：

这样我们就得到了四个三角形，如果想要其中一个，只需要把另外三个的颜色都设为 transparent:

```
div {
    width:0;
    height:0;
    border: 40px solid;
    border-color: transparent red transparent transparent;
}
```

效果如下：

不过其他隐藏的左边框依然占据着空间，因此我们可以把左边框的 border-width 设为 0：

```
div {
    width:0;
    height:0;
    border-style: solid;
    border-width: 40px 40px 40px 0;
    border-color: transparent red transparent transparent;
}
```

3. 元素水平垂直居中的方案

- 元素固定宽高：定位 + 负margin

```
.wrapper {
    height: 900px;
    background-color: gray;
    position: relative;
}
.content {
    width: 400px;
    height: 400px;
    background-color: red;
}
.content {
    position: absolute;
    /* top-left 相对于第一个 position 不为 static 的父元素的 height-width 定位 */
    left: 50%;
    top: 50%;
    /* 相对于元素自己的 height-width 定位 */
    margin-left: -200px;
    margin-top: -200px;
}
<div class="wrapper">
    <div class="content">
    </div>
</div>
```

- 元素不固定宽高

方案一：定位 + transform

```
.wrapper {
    height: 900px;
    background-color: gray;
    position: relative;
}
.content {
    background-color: red;
    width: 400px;
    height: 400px;
}
.content {
    position: absolute;
    left: 50%;
    top: 50%;
    /* 相对于元素的自身宽高进行平移 */
    transform: translate(-50%, -50%);
}
```

方案二：定位 + margin:auto

```
.wrapper {
    height: 900px;
    background-color: gray;
    position: relative;
}
.content {
    background-color: red;
    width: 300px;
    height: 300px;
}
.content {
    position: absolute;
    left: 0;
    right: 0;
    bottom: 0;
    top: 0;
    margin: auto;
}
```

方案三：display:flex

```
.wrapper {
    height: 900px;
    background-color: gray;
    display: flex;
    align-items: center;
    justify-content: center;
}
.content {
    background-color: red;
    width: 300px;
    height: 300px;
}
```

方案四：display:table-cell + vertical-align:middle 需要父元素固定宽高

```
.wrapper {
    width: 900px;
    height: 900px;
    background-color: gray;
    display: table-cell;
    vertical-align: middle;

}
.content {
    background-color: red;
    width: 300px;
    height: 300px;
    margin: 0 auto;
}
```

对于display: tabel-cell的元素，height和padding同时设置会出现问题：

- 当padding与内容高度超过设置的高度时，元素的高度取决于padding和内容的高度，最初为元素设置的高度就不生效了；
- 当padding与内容的高度小于设置的高度时，元素的高度取决于设置的高度；

4. 元素种类的划分

CSS元素可分为行内元素，行内块级元素以及块级元素。

- 行内元素 设置宽高无效，可以设置左右外边距，上下外边距无效。如：span，a，em，label，strong
- 行内块级元素 可以设置宽高，可以设置外边距。如：img，input，video，audio
- 块级元素独占一行。如：div，p，ul，li，h1-h6，table，form

通过 display 可以更改元素的表现形式

5. 盒子模型及其理解

CSS盒模型包含了元素内容区域（content）、边框（border）、内边距（padding）、外边距（margin）。在标准盒模型中，元素的width和height属性为元素的content的宽高。但是在实践中发现，当我们想要一个固定宽高的盒子，并给元素增加padding之后，元素的实际宽高会变大，需要我们自己去手动修改height和width的值，很不方便。因此在 CSS3 中提供了 box-sizing 属性用来改变标准盒

模型。

- box-sizing: content-box：元素的height和width为元素content区域的宽高
- box-sizing: border-box：元素的height和width为元素 content+padding+border的宽高，当设置padding或border时，内容content会自动缩小
- box-sizing: inherit

6. margin塌陷及合并问题

margin塌陷：父子嵌套元素在垂直方向的margin,父子元素是结合在一起的,他们两个的margi会取其中最大的值.

如下所示：

```
.parent {
    width: 300px;
    height: 300px;
    background-color: red;
    margin-top: 100px;
}
.child {
    background-color: blue;
    width: 50px;
    height: 50px;
}
<div class="parent">
    <div class="child"></div>
</div>
```

当给子元素设置 margin-top: 150px 的时候，子元素会带着父元素往下移动50px。

正常情况下,父级元素应该相对浏览器进行定位,子级相对父级定位。但由于margin的塌陷,父级相对浏览器定位.而子级没有相对父级定位,子级相对父级,就像坍塌了一样。

解决方法：

- position:absolute/fixed
- float:left/right
- display:inline-block
- overflow:hidden

这四种方法都能触发BFC，但是使用的时候会带来不同的麻烦，需要根据具体情况解决 margin 塌陷。

margin合并：两个兄弟结构的元素在垂直方向上的 margin 是合并的

如下所示：

```
.div1 {
    height: 200px;
    margin: 20px 0;
    background-color: red;
}
.div2 {
    height: 100px;
    margin: 10px 0;
    background-color: blue;
}
<div class="div1"> </div>
<div class="div2"></div>
```

div1 与 div2 的之间的 margin 取两者的最大值。

7. 浮动模型及清除浮动的方法

- 父元素添加 overflow:hidden
- 在父元素添加最后一个子元素，并设置 clear:both

```
csscss<style>
    .parent p {
      clear:both
    }
    .left {
      float: left;
      height: 200px;
      width: 100px;
      background-color: red;
    }
    .right {
      float: right;
      height: 200px;
      width: 100px;
      background-color: blue;
    }
</style>
<body>
  <div class="parent">
    <div class="left"></div>
    <div class="right"></div>
    <p></p>
  </div>
</body>
```

- 通过伪元素清除浮动

```
<style>
    .parent::after {
      content: "";
      display: block;
      clear: both;
    }
    .left {
```

```
      float: left;
      height: 200px;
      width: 100px;
      background-color: red;
    }
    .right {
      float: right;
      height: 200px;
      width: 100px;
      background-color: blue;
    }
</style>
<body>
  <div class="parent">
    <div class="left"></div>
    <div class="right"></div>
  </div>
</body>
```

- 通过BFC的方式清除浮动
  - position:absolute/fixed，不为 static 或 relative
  - float:left/right
  - display:inline-block，table-cell
  - overflow:hidden，scroll，不为 visible

8. 圣杯布局与双飞翼布局

圣杯布局与双飞翼布局是为了实现一个两侧固定宽度，中间自适应的三栏布局。两者需要遵循以下要点：

- 两侧固定宽度，中间自适应
- 中间部分在DOM结构上优先，以便先行渲染
- 允许三列中的任意一列成为最高列
- 只需要使用一个额外的 div 标签

核心是左边的 `div margin-left:-100%`。

除此之外，两者还可以用 flex 布局实现。

圣杯布局

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <style>
    body {
      min-width: 500px;
      margin: 0;
      padding: 0;
    }
```

```
    .container {
      padding: 0 200px 0 150px;
      overflow: hidden;
    }
    .center {
      background-color: blue;
      height: 800px;
      width: 100%;
    }
    .left {
      background-color: yellow;
      width: 150px;
      height: 200px;
      margin-left: -100%;
      position: relative;
      left: -150px;
    }
    .right {
      background-color: red;
      width: 200px;
      height: 200px;
      margin-right: -200px;

    }
    .container > div {
      float: left;
    }
  </style>
</head>
<body>
  <div class="container">
    <div class="center"></div>
    <div class="left"></div>
    <div class="right"></div>
  </div>
</body>
</html>
```

圣杯布局的flex实现：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>圣杯布局的flex实现</title>
  <style>
    *{
      margin: 0;
      padding: 0;
    }
    .container {
      display: flex;
    }
    .center {
      background-color: blue;
```

```
      height: 800px;
      flex: 1;
    }
    .left {
      background-color: red;
      height: 400px;
      width: 200px;
      order: -1;
    }
    .right {
      background-color: yellow;
      height: 400px;
      width: 150px;
    }
  </style>
</head>
<body>
  <div class="container">
    <div class="center"></div>
    <div class="left"></div>
    <div class="right"></div>
  </div>
</body>
</html>
```

双飞翼布局

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>双飞翼布局</title>
  <style>
    *{
      margin: 0;
      padding: 0;
    }
    body {
      min-width: 500px;
    }
    .left {
      background-color: red;
      width: 150px;
      height: 150px;
      margin-left: -100%;
    }
    .right {
      background-color: yellow;
      width: 200px;
      height: 150px;
      margin-left: -200px;
    }
    .container {
```

```
      width: 100%;

    }
    .center {
      margin-left: 150px;
      margin-right: 200px;
      height: 300px;
      background-color: blue;
    }
    .column {
      float: left;
    }
  </style>
</head>
<body>
  <div class="container column">
    <div class="center"></div>
  </div>
  <div class="left column"></div>
  <div class="right column"></div>
</body>
</html>
```

9. Flex 布局

flex 布局意为“弹性布局”，用来为盒模型提供最大的灵活性。任何一个元素都可以指定为 flex 布局。

注意，设为 flex 布局以后，子元素的 `float`、`clear`、`vertical-align` 属性将失效。

容器的属性

属性	描述
flex-direction	决定主轴的方向，即项目的排列方向。row | row-reverse | column | column-reverse，默认为 row
flex-wrap	默认情况下项目都在一条轴线上，该属性定义，如果一行排不下，该如何换行。wrap | nowrap | wrap-reverse
flex-flow	是 flex-direction 和 flex-wrap 的简写形式，默认为 row nowrap
justify-content	定义项目在主轴上的对齐方式：flex-start | flex-end | center | space-between | space-around
align-items	定义项目在交叉轴上如何对齐：flex-start | flex-end| center | baseline | stretch
align-content	定义了多根轴线的对齐方式。flex-start | flex-end | center | space-between | space-around | stretch;

项目的属性

属性	描述
order	定义项目的排列顺序。默认为0，数值越小，排列越靠前。
flex-grow	定义项目的放大比例，默认为0，即如果存在剩余空间也不放大。
flex-shrink	定义项目的缩小比例，默认为1，即如果空间不足，该项目将缩小。
flex-basis	定义在分配多余空间之前，项目占据的主轴空间。他可以设为跟width一样的属性值，则项目将占据固定空间。
flex	是flex-grow、flex-shrink、flex-basis的简写，默认值为 0 1 auto。
align-self	属性允许单个项目有与其他项目不一样的对齐方式，可覆盖 `align-items` 属性。默认值为 `auto` ，表示继承父元素的 `align-items` 属性。

10. px,em,rem的区别

- px：像素px是相对于显示器屏幕分辨率而言的。
- em：相对于当前父元素的字体大小
- rem：rem相对于根元素设置字体大小

对于只需要适配少部分手机设备，且分辨率对页面影响不大的，使用px即可

对于需要适配各种移动设备，使用rem

11. 媒体查询

```
@media screen and (max-width: 300px) {
    body {
        background-color:lightblue;
    }
}
```

12. HTML5 新特性

音频、视频、地理位置、localStorage、sessionStorage

13. Grid 布局

Flex 布局是轴线布局，只能指定"项目"针对轴线的位置，可以看作是一维布局。

Grid 布局则是将容器划分为 "行" 和 "列"，产生单元格，然后指定项目所在的单元格，可以看作是二维布局。

14. 行内元素的间距怎么解决

- 行内元素写在一行，中间不留空格
- 父元素 font-size 设为 0
- 使用 float

15. 伪类和伪元素有什么不同

伪类用于当已有元素处于的某个状态时，为其添加对应的样式，这个状态是根据用户行为而动态变化的，比如说，当用户悬停在指定的元素时，我们可以用:hover来描述这个元素的状态。虽然它和普通的css类相似，可以为已有的元素添加样式，但是它只有处于dom树无法描述的状态下才能为元素添加样式，所以将其称为伪类。

伪元素不是 DOM 中的真实元素，但是存在于最终的渲染树中，可以为其添加样式，比如说，我们可以通过：before来在一个元素前增加一些文本。

JavaScript 常见知识总结

1. 原始值和引用值类型及区别

原始值，也叫基本类型：如null,undefined,string,number,boolean

引用值，如Object,Function,Date,Array,RegExp

原始值与引用值的区别：

- 原始值存储在栈中，引用值存储在堆中
- 原始值是以值的拷贝方式进行赋值，值是不可变的；引用值是以引用的拷贝方式进行赋值，值是可变的
- 原始值的比较是值的比较，引用值的比较是引用的比较（比较引用的是否为同一对象）

2.判断数据类型的常用方法

typeof

typeof 进行类型判断的返回值有： 'undeinfed', 'string', 'number' , 'boolean' , 'object' , 'symbol' , 'function'

typeof 对 null 返回 'object'，对正则，数组返回 'object'

instanceof

用于检测某个对象的原型链(`__proto__`)上是否存在另一个对象的 `prototype`

```
function instance(target, cons) {
    return cons.prototype.isPrototypeOf(target)
}
```

Object.prototype.toString.call()

在任何值上调用 Object 原生的 toString() 方法，都会返回一个 [object NativeConstructorName] 格式的字符串。每个类在内部都有一个 [[Class]] 属性，这个属性中就指定了上述字符串中的构造函数名。但是它不能检测非原生构造函数的构造函数名。

constructor

指向该对象实例的`__proto__.constructor`。

constructor不能判断undefined和null，并且使用它是不安全的，因为contructor的指向是可以改变的

3. 类数组和数组的区别与转换

类数组对象，如 arguments 对象、NodeList 对象等，类数组对象有length属性，可以通过数组下标取值，但是类数组对象不能调用数组原型上的方法。

怎么把类数组对象转换成数组?

```
Array.prototype.slice.call(NodeList, 0)

Array.from(NodeList)

var new = [...NodeList]
```

4. 数组常见的 API

改变数组本身的方法:

- fill(value, start, end):将数组start到end的值用value填充
- pop
- push
- reverse
- shift
- unshift
- sort
- splice

不改变自身的方法:

- concat
- includes(valueToFind, index)
- join
- slice
- indexOf
- lastIndexOf

遍历方法:

- every
- some
- map
- redeuce
- forEach

- filter
- find:找到第一个满足测试函数的元素并返回那个元素的值
- findIndex

5. bind,call,apply的区别

- 当我们使用一个函数需要改变`this`指向的时候才会用到`call``apply``bind`
- 如果你要传递的参数不多，则可以使用`fn.call(thisObj, arg1, arg2 ...)`
- 如果你要传递的参数很多，则可以用数组将参数整理好调用`fn.apply(thisObj, [arg1, arg2 ...])`
- 如果你想生成一个新的函数长期绑定某个函数给某个对象使用，则可以使用`const newFn = fn.bind(thisObj); newFn(arg1, arg2...)`
- call，apply，bind 不传参数自动绑定在 window

6. new的原理

new 大概会执行以下四个步骤：

- 创建一个空对象
- 将空对象的原型链连接到另一个对象
- 执行构造函数中的代码并绑定 this 到这个对象
- 如函数没有返回值，则返回该对象

自己实现一个 _new(Constructor,...args):

```
function _new() {
    // 参数为 对象A，属性

    // 1.创建一个空对象
    let obj = {}

    // 2.将该空对象的原型链连接到传入的对象
    let [Con, ...args] = arguments
    obj.__proto__ = Con.prototype

    // 3.执行函数并绑定 this
    let res = Con.apply(obj, args)

    // 4.如果函数有返回值并且为object，则返回函数的返回值，否则返回obj
    return res instanceof Object ? res : obj
}

function Person(name, age) {
    this.name = name
    this.age = age
}

Person.prototype.getName = function() {
   return this.name
}

let p = _new(Person, "sillywa", 23)
```

7. 如何正确判断this

this 是 JavaScript 中最复杂的机制之一。它是一个很特别的关键字，被自动定义在所有函数的作用域中。与词法作用域不同，this 是在运行时进行绑定的，并不是在编写时，它的上下文取决于函数调用的各种条件。this的绑定和函数声明的位置没有任何关系，只取决于函数的调用方式。

一，关于 this

关于 this 主要有两种误解，一种是认为 this 指向函数自身，另一种是 this 指向函数的作用域。

指向自身

思考以下代码：

```
function foo(num) {
    console.log("foo: " + num);
    // 记录 foo 被调用的次数
    this.count++;
}
foo.count = 0;

for(var i = 0; i < 5; i++) {
    foo(i);
}
// foo被调用了多少次？
console.log(foo.count); // 0 -- 为什么？
```

执行 foo.count = 0 时，的确向函数对象 foo 添加了一个属性 count。但是函数内部代码 this.count 中的 this 并不是指向那个函数对象，所以虽然属性名相同，跟对象却并不相同。

实际上，如果深入探索的话，就会发现这段代码在无意中创建了一个全局变量 count，它的值为 NaN。

如果要让上面的代码实现我们的功能，我们可以用 foo 来代替 this 来引用函数对象：

```
function foo(num) {
    console.log("foo: " + num);
    // 记录 foo 被调用的次数
    foo.count++;
}
foo.count = 0;

for(var i = 0; i < 5; i++) {
    foo(i);
}

console.log(foo.count);
```

另一种方法是强制 this 指向 foo 函数对象：

```
function foo(num) {
    console.log("foo: " + num);
    // 记录 foo 被调用的次数
    this.count++;
}
foo.count = 0;

for(var i = 0; i < 5; i++) {
    foo.call(foo, i);
}

console.log(foo.count); // 5
```

它的作用域

第二种常见的误解是，this指向函数的作用域。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。

```
function foo() {
    var a = 2;
    this.bar();
}
function bar() {
    console.log(this.a);
}
foo();  //ReferenceError: a in not defined
```

因此在学习 this 之前，我们必须明白，this 既不指向函数自身也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绑定的。

二，调用位置

在理解this的绑定规则之前，首先要理解调用位置，即函数在代码中被调用的位置。最重要的是要分析调用栈，我们关心的调用位置就是当前正在执行的函数的前一个调用中。

```
function baz() {
    // 当前调用栈是： baz
    // 因此调用位置是全局作用域
    console.log("baz");
    bar();  // <-- bar的调用位置
}

function bar() {
    // 当前调用栈是：baz->bar
    // 因此调用位置在 baz 中
    console.log("bar");
    foo();  // <-- foo的调用位置
}

function foo() {
    // 当前调用栈是：baz->bar->foo
    // 因此调用位置在 bar 中
    console.log("foo");
```

```
}
baz();  // <-- baz的调用位置
```

注意我们是如何分析出真正的调用位置的，因为它决定了 this 的绑定。

三，绑定规则

我们首先需要找到调用位置，然后判断需要应用下面四条规则中的哪一条。首先会介绍四条规则，然后说明多条规则都可以使用时的优先级。

默认绑定

默认绑定就是简单的独立函数调用，可以把这条规则看作是无法应用其它规则时的默认规则。

```
function foo() {
    console.log(this.a);
}
var a = 2;
foo();  // 2
```

在代码中，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。在非严格默认下，默认绑定的 this 指向全局对象，严格模式下为 undefined。

隐式绑定

另一条需要考虑的规则是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。

```
function foo() {
    console.log(this.a);
}
var obj = {
    a: 2,
    foo: foo
};
obj.foo();  //2
```

当 foo 被调用时，它前面加上了对 obj 的引用。当函数引用有上下文对象时，隐式绑定的规则会把函数调用中的 this 绑定到这个上下文对象。

对象属性链中只有上一层或者说最后一层在调用位置中起作用。举例来说：

```
function foo() {
    console.log(this.a);
}
var obj2 = {
    a: 42,
    foo: foo
};
var obj1 = {
    a: 2,
    obj2: obj2
};

obj1.obj2.foo();    // 42
```

隐式丢失

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 绑定到全局对象或者 undefined 上。

```
function foo() {
    console.log(this.a);
}
var obj = {
    a: 2,
    foo: foo
};
var bar = obj.foo;  // 函数别名!

var a = "oops,global";

bar();  // "oops,global"
```

虽然 bar 是 obj.foo 的一个引用，但是实际上它引用的是 foo 函数本身，因此此时的 bar() 其实是一个不带任何修饰符的函数调用，因此应用了默认绑定。

一种更微妙、更常见并且更出乎意料的情况发生在传入回调函数时：

```
function foo() {
    console.log(this.a);
}
function doFoo(fn) {
    // fn 其实引用的是 foo
    fn();   // <--调用位置
}
var obj = {
    a: 2,
    foo: foo
};
var a = "oops,global";

doFoo(obj.foo); // "oops,global"
```

传递参数其实就是一种隐式赋值，因此我们传入函数时也会被隐式赋值。

同样把函数传入语言内置的函数结果也是一样的。

```
function foo() {
    console.log(this.a);
}

var obj = {
    a: 2,
    foo: foo
};
var a = "oops,global";

setTimeout(obj.foo, 1000);  // "oops,global"
```

经过上面的分析我们知道，回调函数丢失 this 绑定是非常常见的。

显示绑定

就像我们刚才看到的那样，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 间接绑定到对象上。

如果我们不想在对象内部包含函数的引用，而想在某个对象上强制调用函数，这是我们需要使用函数的 call() 和 apply() 方法。

它们的第一个参数是一个对象，是给 this 准备的，接着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，因此称之为显示绑定。

```
function foo() {
    console.log(this.a);
}
var obj = {
    a:2
};
foo.call(obj);  // 2
```

显示绑定的另一种情况就是硬绑定。

```
function foo() {
    console.log(this.a);
}
var obj = {
    a: 2
};
var bar = function() {
    foo.call(obj);
}
setTimeout(bar, 1000);  // 2

// 硬绑定的 bar 不可能再修改它的 this
bar.call(window);   // 2
```

因为我们把 bar 函数内部调用了 foo，而 foo 的 this 已经被强制绑定在 obj 上，因此无论之后如何调用 bar 函数，它总会手动在 obj 上调用 foo。

硬绑定的另一种应用场景就是创建一个包裹函数，负责接收参数并返回值：

```
function foo(something) {
    console.log(this.a, something);
    return this.a + something;
}
var obj = {
    a: 2
};
var bar = function() {
    return foo.apply(obj, arguments);
};

var b = bar(3); //2 3
console.log(b); // 5
```

另一种方法是创建一个可以重复使用的辅助函数：

```
function foo(something) {
    console.log(this.a, something);
    return this.a + something;
}
var obj = {
    a: 2
};
function bind(fn, obj) {
    return function() {
        return fn.apply(obj, arguments);
    }
}

var bar = bind(foo, obj);
var b = bar(3); // 2 3
console.log(b); // 5
```

ES5 中提供了 Function.prototype.bind 函数，它的用法如下：

```
unction foo(something) {
    console.log(this.a, something);
    return this.a + something;
}
var obj = {
    a: 2
};
var bar = foo.bind(obj);
var b = bar(3); // 2 3
console.log(b); // 5
```

bind() 会返回一个硬编码的新函数，它会把你指定的参数设置为 this 的上下文并调用原始函数。

new绑定

在传统的面向对象的语言中，“构造函数”是类中的一些的特殊方法，使用 new 初始化类时会调用类中的构造函数。Javascript 中也有一个 new 操作符，但是 Javascript 中 new 的机制实际上和面向对象的语言完全不同。在 Javascript 中，构造函数只是一些使用 new 操作符时被调用的函数。它们并不属于某个类，也不会实例化一个类。

使用 new 来调用函数，或者说发生构造函数调用时，会自动执行下面的操作：

- 创建一个全新的对象。
- 这个对象会被执行 [[Prototype]] 连接。
- 这个新对象会被绑定到函数调用的 this。
- 如果函数没有返回其它对象，那么 new 表达式中的函数调用会自动返回这个新对象。

思考下面代码：

```
function foo(a) {
    this.a = a;
}
var bar = new foo(2);
console.log(bar.a); // 2
```

使用 new 来调用 foo() 时，我们会构造一个新对象并把它绑定到 foo() 调用中的 this 上。 new 是最后一种可以影响函数调用时 this 绑定行为的方法，我们称之为 new 绑定。

四，判断 this

学习了上面四条规则，我们可以根据下面的顺序来判断 this 绑定的对象：

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。
2. 函数是否通过 call、apply 显示绑定或者硬绑定？如果是的话，this 绑定的是指定对象。
3. 函数是否在某个上下文中调用（隐式绑定）？如果是的话，this 绑定的是那个上下文对象。
4. 如果都不是，使用默认绑定。严格模式下绑定到 undefined，否则绑定到全局对象。

五，绑定例外

在某些场景下 this 的绑定行为会出乎意料，你认为应该应用其它绑定规则时，实际上应用的可能是默认绑定的规则。

被忽略的 this

如果把 null 或者 undefined 作为 this 的绑定对象传入 call、apply 或者 bind，这些值在调用时会被忽略，实际应用的是默认绑定的规则。

```
function foo() {
    console.log(this.a);
}
var a = 2;
foo.call(null); //2
```

一种常见的做法是使用 apply(...) 来“展开”一个数组，并当作参数传入一个函数。类似地，bind(...)可以对参数进行柯里化，这种方法有时非常有用：

```
function foo(a ,b) {
    console.log("a: " + a + ", b: " + b);
}
// 把数组展开成参数
foo.apply(null, [2, 3]);    //a: 2, b: 3

// 使用 bind 进行柯里化
var bar = foo.bind(null, 2);
bar(3); // a: 2, b: 3
```

间接引用

另一个需要注意的是你可能有意或者无意地创建一个函数的”间接引用“，在这种情况下，调用这个函数会应用默认绑定规则。

间接引用最容易在赋值的时候发生：

```
function foo() {
    console.log(this.a);
}
var a = 2;
var o = {a: 3, foo: foo};
var p = {a: 4};
o.foo();    // 3
(p.foo = o.foo)(); // 2
```

赋值表达式 p.foo = o.foo 的返回值是目标函数的引用，因此调用位置是 foo() 而不是 p.foo() 或者 o.foo()。

六，this 词法

我们之前介绍的四条规则已经可以包含所有的正常函数。但是在 ES6 中介绍了一种无法使用这些规则的特殊类型函数：箭头函数。

箭头函数不使用 this 的四种标准规则，而是根据外层作用域来决定 this。

我们来看看箭头函数的词法作用域：

```
function foo() {
    return a => {
        // this继承自 foo()
        console.log(this.a);
    }
}
var obj1 = {
    a: 2
};
var obj2 = {
    a: 3
};
var bar = foo.call(obj1);
bar.call(obj2); // 2
```

对比正常的函数：

```
function foo() {
    return function() {
        console.log(this.a);
    }
}
var obj1 = {
    a: 2
};
var obj2 = {
    a: 3
};
var bar = foo.call(obj1);
bar.call(obj2); // 3
```

foo() 内部的箭头函数会捕获调用时 foo() 的 this。由于 foo() 的 this 绑定到 obj1，bar 引用箭头函数的 this 也会绑定到 obj1，箭头函数的绑定无法修改。（new 也不行）

箭头函数最常用于回调函数，例如事件处理器或者定时器：

```
function foo() {
    setTimeout(() => {
        // 这里的 this 在词法上继承 foo
        console.log(this.a);
    })
}
var obj = {
    a: 2
};
foo.call(obj);  // 2
```

8. 闭包及其作用

一个函数有权访问另一个函数作用域中的变量，就形成闭包。

闭包可以用来隐藏变量，避免全局污染。也可以用于读取函数内部的变量。

缺点是：导致变量不会被垃圾回收机制回收，造成内存消耗。

9. 原型和原型链

无论什么时候，只要创建了一个函数，就会根据为该函数创建一个 prototype 属性，这个属性指向函数的原型对象。在默认情况下，所有原型对象都会获得一个 constructor，该属性是一个指向 prototype 属性所在函数的指针。

原型链规定了对象如何查找属性，对于一个对象来说，如果它本身没有某个属性，则会沿着原型链一直向上查找，知道找到属性或者查找完整个原型链。

原型链是实现继承的主要方法，其基本思想是利用原型链让一个引用类型继承另一个引用类型的属性和方法。

10. 继承的实现方式及比较

（1）简单的原型继承

```
function SuperType() {
    this.name = "super"
}
function SubType() {}

// 利用原型链实现继承
SubType.prototype = new SuperType()

var instance1 = new SubType()
console.log(instance1.name) // super
```

简单的原型继承存在以下两个问题：

- 包含引用类型值的原型属性会被所有实例共享，在通过原型来实现继承时，原型实际上也会变成另一个类型的实例。于是，原先的实例属性也就变成了现在的原型属性。思考一下代码：

```
function SuperType() {
    this.names = ["sillywa", "xinda"]
}
function SubType() {}

// 利用原型链实现继承
SubType.prototype = new SuperType()

var instance1 = new SubType()
instance1.names.push("hahah")
console.log(instance1.names)    // ["sillywa", "xinda", "hahah"]

var instance2 = new SubType()
console.log(instance2.names)    // ["sillywa", "xinda", "hahah"]
```

这个例子中，SuperType构造函数定义了一个 names 属性，该属性为一个数组（引用类型）。SuperType的每个实例都会有自己的 names 属性。当 SubType 通过原型链继承了 SuperType 之后，SubType.prototype 就变成了 SuperType 的一个实例，因此它也拥有自己的 names 属性——就跟专门创建了一个 SubType.prototype.names 属性一样。但是结果就是 SubType 的所有实例共享一个 names 属性。

- 简单的原型继承的另一个问题是：在创建子类类型的实例时，不能向超类类型的构造函数中传递参数。

因此在继承上我们经常不会单独使用原型继承。

（2）借用构造函数继承（经典继承）

这种继承的思想是在子类的构造函数内部调用超类的构造函数，该方法使用 call() 和 apply() 方法在新创建的对象上执行构造函数。如下所示：

```
function SuperType(age, name) {
    this.colors = ["blue", "red"]
    this.age = age
    this.name = name
}
function SubType() {
    SuperType.call(this, ...arguments)
}

var instance1 = new SubType(23, "sillywa")
instance1.colors.push("yellow")
console.log(instance1.colors, instance1.name)

var instance2 = new SubType(12, "xinda")
console.log(instance2.colors, instance2.name)
```

借用构造函数继承也有一些缺点，比如方法都只能在构造函数中定义，没有办法实现方法的复用。例如：

```
function SuperType(name) {
    this.name = name
    this.sayName = function() {
        return this.name
    }
```

```
}
function SubType(name, age) {
    SuperType.call(this, name)
    this.age = age
}

// 每次实例化一个对象，都会重新实例化 sayName 方法
var instance1 = new SubType("sillywa", 24)
console.log(instance1)
console.log(instance1.sayName())
```

(3) 组合式继承

组合继承结合了原型继承和借用构造函数继承的优点，其背后的思想是，使用原型链实现对原型方法的继承，使用构造函数实现对实例属性的继承。这样，既通过在原型上定义方法实现了函数的服用，又通过构造函数实现了每个实例都有自己的属性。

```
function SuperType(name) {
    this.name = name
    this.colors = ["red", "yellow"]
}
// 方法写在原型上
SuperType.prototype.sayName = function() {
    return this.name
}
function SubType(name, age) {
    // 通过 构造函数继承属性
    SuperType.call(this, name)
    this.age = age
}
// 通过原型继承方法
SubType.prototype = new SuperType()

// 重写了 SubType 的 prototype 属性，因此其 constructor 也被重写了，需要手动修正
SubType.prototype.constructor = SubType

// 定义子类自己的方法
SubType.prototype.sayAge = function() {
    return this.age
}
```

测试案例：

```
var instance1 = new SubType("sillywa", 23)
instance1.colors.push("blue")
console.log(instance1.colors)        //["red", "yellow", "blue"]
console.log(instance1.sayName())    // sillywa
console.log(instance1.sayAge())        // 23

var instance2 = new SubType("xinda", 90)
console.log(instance2.colors)        // ["red", "yellow"]
console.log(instance2.sayName())    // xinda
console.log(instance2.sayAge())        // 90
```

组合继承避免了原型链和借用构造函数的缺陷，融合了它们的优点，成为 JavaScript 中最常用的继承模式。

（4）原型式继承

借助原型可以通过已有的对象创建新对象，同时还不必因此创建自定义类型。为达到这个目的，可以定义如下函数：

```
function create(o) {
    function F(){}
    F.prototype = o
    return new F()
}
```

在 object 函数内部，首先创建了一个临时性构造函数 F，将 F 的 prototype 属性指向传入的对象 o，并返回 F 的一个实例，则该实例继承 o 的所有属性和方法。从本质上讲，create() 对传入的对象执行了一次浅复制。看以下代码：

```
var person = {
    name: "sillywa",
    firends: ["Johe"]
}

var person1 = create(person)
person1.name = "coder"
person1.firends.push("Kobe")

var person2 = create(person)
person2.firends.push("Cury")
console.log(person2.firends)    // ["Johe", "Kobe", "Cury"]
```

ES5 通过新增 Object.create() 方法规范化了原型式继承。这个方法接受两个参数：一个用作新对象原型的对象和（可选的）一个为新对象定义额外属性的对象。在传入一个参数的情况下，Object.create() 与 create() 方法的行为相同。

Object.create() 方法的第二个参数与 Object.defineProterties() 方法的第二个参数格式相同：每个属性都是通过自己的描述符定义的。以这种方式指定的任何属性都会覆盖原型对象上的同名属性。例如：

```
var person = {
    name: "sillywa"
}
var person1 = Object.create(person, {
    name: {
        value: "John"
    }
})
console.log(person1.name)        // John
```

（5）寄生式继承

寄生式继承的思路与继承构造函数和工厂模式类似，即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，最后再像真正地是它做了所有工作一样返回对象。以下是寄生式继承的代码：

```
function createAnother(original) {
    var clone = Object.create(original)
    clone.sayHi = function() {
        console.log("Hi")
    }
    return clone
}
```

（6）组合寄生式继承

前面说过，组合继承是 JavaScript 最常用的继承模式，不过它也有自己的缺点，组合继承最大的问题是，无论什么情况下都会调用两次超类的构造函数。

```
function SuperType(name) {
    this.name = name
    this.colors = []
}
SuperType.prototype.sayName = function() {
    return this.name
}

function SubType(name, age) {
    // 第一次调用父类的构造函数
    SuperType.call(this,name)
    this.age = age
}
// 第二次调用父类的构造函数
SubType.prototype = new SuperType()
SubType.prototype.constructor = SubType
SubType.prototype.sayAge = function() {
    return this.age
}
```

组合寄生式继承就是为了解决这一问题，将第二次调用构造函数改为使用 Object.create() 函数来实现：

```
function SuperType(name) {
    this.name = name
    this.colors = []
}
SuperType.prototype.sayName = function() {
    return this.name
}

function SubType(name, age) {
    // 第一次调用父类的构造函数
    SuperType.call(this,name)
    this.age = age
}
// 关键代码
SubType.prototype = Object.create(SuperType.prototype)
```

```
SubType.prototype.constructor = SubType
SubType.prototype.sayAge = function() {
    return this.age
}
```

11. 对象的深拷贝与浅拷贝

(1) 浅拷贝的实现方法

- 遍历赋值
    - for in

```
function clone(obj) {
    var cloneObj = {}
    // for in 遍历，会遍历原型链里面的属性，所以需要排除原型链
    for(var key in obj) {
        if(obj.hasOwnProperty(key)) {
            cloneObj[key] = obj[key]
        }
    }
    return cloneObj
}
```

    - Object.keys()

```
function clone(obj) {
    var cloneObj = {}
    // Object.keys()不会遍历到原型链中的属性
    for(var key of Object.keys(obj)) {
        cloneObj[key] = obj[key]
    }
    return cloneObj
}
```

    - Object.entries()

```
function clone(obj) {
    var cloneObj = {}
    for(var [key, value] of Object.entries(obj)) {
        cloneObj[key] = value
    }
    return cloneObj
}
```

- Object.assign()

```
function clone(obj) {
    return Object.assign(obj, {})
}
```

(2) 深拷贝的实现方法

- JSON.stringfy() 与 JSON.parse()

```
function deepClone(obj) {
    return JSON.parse(JSON.stringify(obj))
}
```

存在问题：遇到函数，undefined，Symbol，Date对象时会自动忽略，遇到正则时会返回空对象。

- 递归

```
// 三种遍历方式就会有三种递归
// for in
function deepClone(obj) {
    if(!obj || typeof obj != "object") return obj
    if(obj instanceof RegExp) return new RegExp(obj)
    var cloneObj = new obj.constructor
    for(var key in obj) {
        if(obj.hasOwnProperty(key)){
            cloneObj[key] = deepClone(obj[key])
        }
    }
    return cloneObj
}

// Object.keys()
function deepClone(obj) {
    if(!obj || typeof obj != "object") return obj
    if(obj instanceof RegExp) return new RegExp(obj)
    var cloneObj = new obj.constructor
    for(var key of Object.keys(obj)) {
        cloneObj[key] = deepClone(obj[key])
    }
    return cloneObj
}
// Object.entries()
function deepClone(obj) {
    if(!obj || typeof obj != "object") return obj
    if(obj instanceof RegExp) return new RegExp(obj)
    var cloneObj = new obj.constructor
    for(var [key, value] of Object.entries(obj)) {
        cloneObj[key] = deepClone(value)
    }
    return cloneObj
}
```

12. 防抖和节流

（1）防抖

函数在指定时间内只会触发一次，具体实现方法：

- 第一次触发函数的时候，延迟delay时间执行，如果在delay时间段内再次触发该函数，则重新开始计时
- 如果delay时间段内没有触发该函数，则执行该函数

```
function debounce(fn, delay) {
    let timer = null
    return function(){
        if(timer) {
            clearTimeout(timer)
        }
        timer = setTimeout(fn, delay)
    }
}
```

（2）节流

防抖的问题是，在短时间内不断触发事件，回调函数永远不会执行。

节流的思想：在短时间内不断触发事件，回调函数只会在指定间隔时间内执行。

```
function throttle(fn, delay) {
    let timer = null;
    return function() {
        if(timer) return false
        timer = setTimeout(() => {
            fn()
            timer = null
        }, delay)
    }
}
```

13. 作用域和作用域链、执行期上下文

作用域是一套规则，用于确定在何处以及如何查找变量。作用域共有两种主要的工作模式，词法作用域和动态作用域，大多数编程语言采用词法作用域，JavaScript也是基于词法作用域的。词法作用域意味着作用域是由书写代码时函数声明的位置来决定的。

在 JavaScript 每个函数有自己的函数作用域，当执行流进入到一个函数的时候，函数的环境就会被推入到一个环境栈中。而在函数执行之后，栈将其环境弹出。

当代码在一个环境中执行的时候，会为该环境创建一个作用域链，保证对执行环境有权访问的所有变量的有序访问。作用域链的最前端始终是当前执行代码所在环境，在变量查找的过程中，会沿着作用域链一层一层向上查找，直到找到变量或者找不到变量。

14. DOM 常见操作方法

方法名	描述
document.getElementById()	
document.getElementsByClassName()	
document.getElementsByName()	
document.getElementsByTagName()	
Node.hasChildNodes()	
Node.appendChild(newNodes)	
Node.insertBefore(newNode, targetNode)	
Node.replaceChild(newNode, targetNode)	
Node.removeChild(targetNode)	
Node.cloneNode(boolean)	表示是否进行深复制
document.querySelector()	
document.querySelectorAll()	
document.createElement()	
document.createTextNode()	
Element.getAttribute()	
Element.setAttribute()	
Element.removeAttribute()	

15. Ajax请求的过程

```
var xhr = new XMLHttpRequest()

// 请求的类型、请求的 url 以及是否异步发送请求
xhr.open("get", url, true)
// 传入请求的数据
xhr.send(null)

xhr.onreadystatechange = function() {
    if(xhr.readyState == 4) {
        if((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304) {
            console.log(xhr.responseText)
        } else {
            throw new Error("error")
        }
    }
}
```

自己进行原生Ajax封装

```
function Ajax() {
    let handleAjax = function(resolve, reject) {
        if((this.status >= 200 && this.status <300) || this.status == 304) {
            console.log(this)
            resolve(this.response)
        } else {
            reject(this.response)
        }
    }
    this.get = function(URL, params={}) {
        return new Promise((resolve, reject) => {
            let xhr = new XMLHttpRequest()

            let paramsStr = Object.entries(params).map(item => item[0] + "=" +
item[1]).join("&")

            URL = paramsStr ? (URL + "?" + paramsStr) : URL

            xhr.open("GET", URL, true)
            xhr.send(null)
            // xhr.onreadystatechange = function() {
            //   if(xhr.readyState === 4) {
            //     if((this.status >= 200 && this.status <300) || this.status ==
304) {
            //       resolve(this.responseText)
            //     } else {
            //       reject(this.response)
            //     }
            //   }
            // }
            xhr.onload = handleAjax.bind(xhr, resolve, reject)
        })
    }
    this.post = function(URL, data={}) {
        return new Promise((resolve, reject) => {
            let xhr = new XMLHttpRequest()

            let formData = new FormData()

            Object.entries(data).map(item => formData.append(item[0], item[1]))

            xhr.open("POST", URL, true)

            xhr.send(formData)

            xhr.load = handleAjax.bind(xhr, resolve, reject)
        })
    }
}
```

16. JS垃圾回收机制

标记清除

标记清除顾名思义是一种分两阶段对对象进行垃圾回收的算法。

第一阶段：标记。从根结点出发遍历对象，对访问过的对象打上标记，表示该对象可达。

第二阶段：清除。对那些没有标记的对象进行回收，这样使得不能利用的空间能够重新被利用。

这个算法假定设置一个叫做根（root）的对象（在Javascript里，根是全局对象）。垃圾回收器将定期从根开始，找所有从根开始引用的对象，然后找这些对象引用的对象……从根开始，垃圾回收器将找到所有可以获得的对象和收集所有不能获得的对象。

理解什么是根？例如：

- 本地函数的局部变量和参数
- 当前嵌套调用链上的其他函数的变量和参数
- 全局变量
- 还有一些其他的，内部的

具体算法：

- 垃圾回收器获取根并标记它们
- 然后它访问并标记所有来自它们的引用

引用计数

另一种不太常见的垃圾回收策略叫做引用计数。引用计数的含义是跟踪每个值被引用的次数。

17. JS中的String、Array和Math方法

（1）String

- substring
- substr
- slice
- indexOf
- lastIndexOf
- charAt: 以索引方式返回字符
- charCodeAt
- concat:连接字符串
- trim
- toLowerCase
- toUpperCase
- match():接受一个正则，返回匹配到的结果数组
- search():接受一个正则，返回第一个匹配的索引
- replace():第一个参数为字符串或者正则，第二个参数为字符串或者函数
- split()

ES6扩展

- padStart():第一参数指定字符串的最小长度，第二个参数是用来补全的字符串；省略第二个字符串会用空格来补全，两者最常用的 就是为数值补全指定位数或者格式化字符串
- endStart()
- includes()

- startsWith()
- endsWith()
- repeat()

(2) Math

- random() 返回 0-1的随机数
- min，max
- ceil:向上舍入，大于它的最小整数
- floor:向下舍入，小于它的最大整数
- round:四舍五入
- log，abs，exp，pow，sqrt

18. addEventListener 和 onClick() 的区别

普通的事件就是直接触发事件，同一时间只能指向唯一对象，所以会被覆盖掉：

```
btn.onclick = function() {
    alert("111")
}
btn.onclick = function() {
    alert("222")
}
// "222"
```

事件绑定就是对于一个可以绑定的事件对象，进行多次绑定事件都能运行：

```
btn.addEventListener("click", function() {
    alert("111")
})
btn.addEventListener("click", function() {
    alert("222")
})

// "111"  "222"
```

- onclick 同一时间只能指向唯一对象
- addEventListener可以给一个事件注册多个listener
- addEventListener对任何DOM都都是有效的，而onclick仅限于HTML
- addEventListener可以控制事件的触发阶段

19. 事件委托

- 当有很多子元素需要绑定相同的事件的时候，会造成很大的内存消耗，可以用事件委托
- 新增的子元素也自动绑定事件

坏处：

- 事件委托基于冒泡，对于不冒泡的事件不支持（blur，focus，mouseleave，mouseenter，load，scroll，resize）（焦点事件，鼠标移动事件）

20. BOM的location对象

以下是 location 对象的所有属性：

属性名	例子	说明
hash	"#contents"	返回URL中的hash，如果不包含，则返回空字符串
host	"www.sillywa.com:8080"	返回服务器名称号和端口号（如果有）
hostname	"www.sillywa.com"	返回不带端口号的服务器名称
href	"https://www.sillywa.com/index/index.html"	返回当前加载页面的完整URL
pathname	"/index/index.html"	返回URL中的目录或者文件名
port	"8080"	返回URL中指定的端口
protocol	"https:"	返回页面使用的协议
search	"?q=javascript"	返回URL中的查询字符串
origin	"https://www.sillywa.com"	返回源

查询字符串参数，格式化 search

```
function getQueryStringArgs() {
  // 取得 search 并去掉问号
  var qs = (location.search.length > 0) ? location.search.slice(1) : "";
  // 保存对象的数据
  var args = {};
  var items = qs.length ? qs.split("&") : [];
  var item, i, name, value;
  for(i = 0; i < items.length; i++){
    item = items[i].split("=");
    name = decodeURIComponent(item[0]);
    value = decodeURIComponent(item[1]);
    if(name.length) {
      args[name] = value;
    }
  }
  return args;
}
```

位置操作

使用 location.assign() 可以改变浏览器的位置，该方法接受一个 URL，如果是将 location.href 或 window.location 设置为一个 URL值，也会以该值调用 assign() 方法。

每次修改 location 的属性（hash除外），页面都会以新的 URL 加载。

当通过任何一种方式修改 URL 之后，浏览器的历史记录就会生成一条新纪录，因此用户通过单击”后退“按钮都会导航到前一个页面。要禁用这种行为，可以使用 replace() 方法。这个方法只接受一个参数，即要导航到的 URL。结果虽然能导致浏览器位置改变，但不会在浏览器中生成新纪录。在调用 replace() 方法之后，用户不能回到前一个页面。

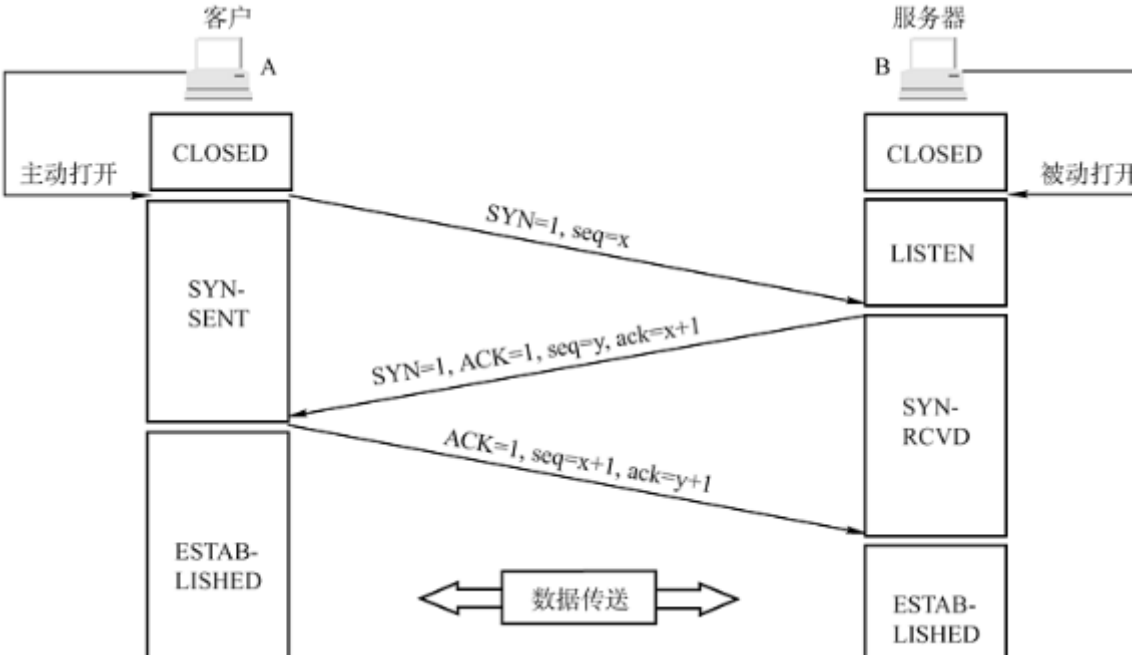
location.reload() 可以重新加载页面，但有可能是从缓存加载。给它传递一个参数 true,就可以强制从服务器加载。

21. 浏览器从输入URL到页面渲染的整个流程

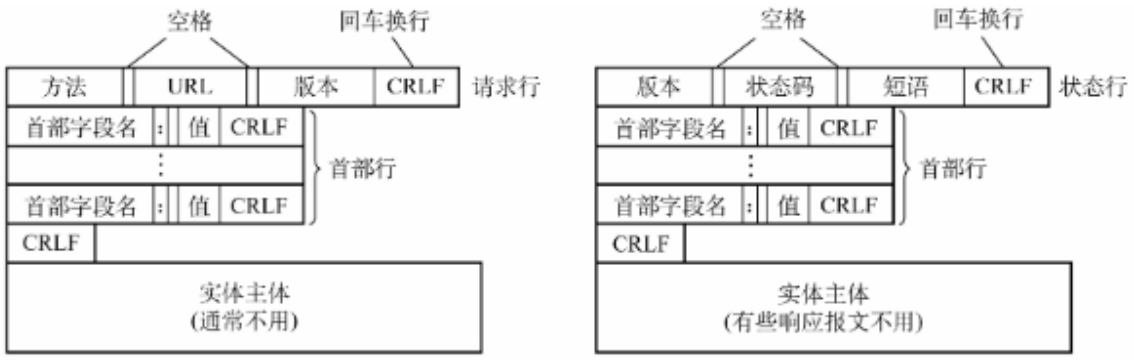
（1）首先对域名进行DNS解析：

- 查询浏览器是否缓存了该域名的IP地址
- 查询hosts文件中是否存在该域名的IP地址
- 查询本地DNS服务器（自己手动设置的DNS服务器）是否存在该域名的IP地址
- 本地DNS服务器向根DNS服务器（13个）发送请求，进行查询

（2）查询到IP地址之后，浏览器开始与服务器建立TCP连接，三次握手：



（3）浏览器发出HTTP请求




```
GET /index.html HTTP/1.1
Host: www.sillywa.com
Connection: Close
Cookie: 123456
```

请求由应用层传入传输层，TCP对发出的数据进行处理，如果数据量较大，对数据进行分包，并给每个包标记序号，加上TCP头部（源端口号、目标端口号），总共大小不得超过1480字节；TCP将TCP包交给网络层进行处理，IP在TCP包头部加上IP头部（源IP地址、目标IP地址），IP头部通常20字节，数据包的大小不得超过1500字节；数据包交给数据链路层进行处理，数据链路层将数据包封装成帧，加上源MAC地址和下一个路由器的MAC地址，添加帧首部标记和帧结束标记，然后交给物理层进行传输。物理层将数据帧转为电信号发送出去

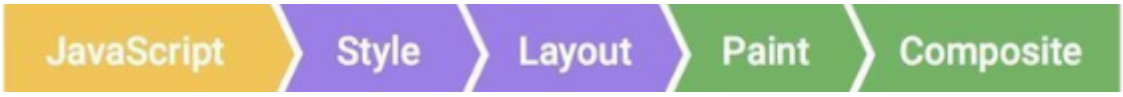
（4）服务端响应

（5）浏览器渲染

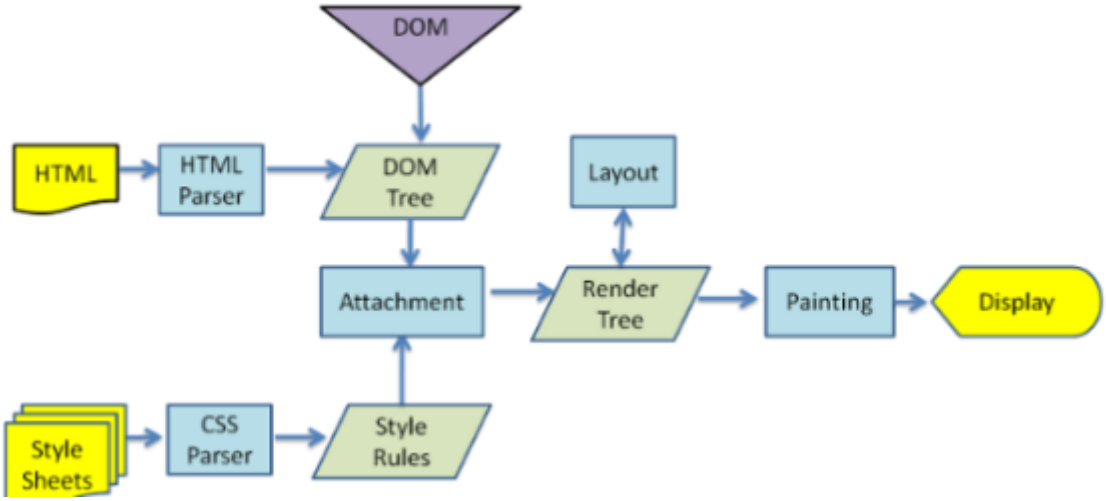
浏览器渲染是指浏览器请求下载资源，然后解析、构建树、渲染布局、绘制，最后呈现给客户界面的整个过程。用户看到页面实际上可分为两个阶段：页面内容加载完成和页面资源加载完成，分别对应`DOMContentLoaded`和`load`。两者的区别如下

- DOMContentLoaded: DOM渲染完成执行，此时样式表、图片、视频还可能没有加载完成
- load：页面全部资源加载完成才会执行，包括图片、视频

浏览器渲染只要包括以下五个步骤：



1. 将HTML解析成DOM树
2. 处理CSS标记，构建层叠样式表模型CSSOM
3. 将DOM和CSSOM合并成渲染树（Render Tree）
4. 计算渲染树中每个DOM元素的坐标和大小，它被称作为布局layout。浏览器使用一种流式处理方法，只需要一次绘制操作就可以布局所有的元素
5. 将渲染树的各个节点绘制到图层上，这一步叫做绘制 painting。本质上就是填充像素的过程。包括绘制文字、颜色、图像、边框和阴影等，也就是一个 DOM 元素所有的可视效果。一般来说，这个绘制过程是在多个层上完成的。
6. 合成：由于DOM元素的绘制是在多个层上进行的，因此在每个层上绘制完成之后，浏览器会将所有层按照合理的顺序合并成一个图层，然后显示在屏幕上，对于有位置重叠的元素，这个过程就比较重要了。（z-index）



浏览器渲染的具体流程：

1. 构建DOM树

   遍历文档节点，生成DOM树，需要注意以下几点：

   - DOM树的构建过程中，可能会被CSS和JS的加载而阻塞执行
   - display:none的元素也会在DOM树中

    - 注释也会在DOM树中
    - script标签也会在DOM树中
2. 构建CSSOM

   每个CSS文件都被拆分成一个 StyleSheet 对象，每个对象都包含CSS规则。CSS规则对象包含对应于CSS语法的选择器和声明对象以及其他对象，在这个过程中需要注意的是：

    - CSS解析可以与DOM解析同时进行
    - CSS解析与script的执行互斥
    - 在Webkit内核中进行了script执行优化，只有在JS访问CSS时才会发生互斥
3. 构建Render Tree

   浏览器从DOM树的根节点开始遍历每一个可见节点，然后对每个可见节点找到适配的CSS样式规则并应用。需要注意的是：

    - Render Tree 和 DOM Tree并不完成对应
    - display:none的元素并不在Render Tree当中
    - visibility:hidden的元素在RenderTree中
4. 渲染树布局

   生成渲染树之后，还是没办法显示在屏幕上，渲染到屏幕上还需要获得个节点的位置信息。这就需要布局的处理了。

   布局会定位DOM元素的坐标和大小。

   布局会从DOM元素的根节点开始遍历，由于渲染树的每一个节点都是一个 Render Object，包含宽高、位置、背景色等信息。所以浏览器就通过这些样式信息来确定每个节点在页面上的确切大小和位置，布局阶段的输出就是盒子模型，它会精确捕获每个元素在屏幕内的确切位置和大小。需要注意的是：

    - float元素，absolute元素，fixed元素会发生位置偏移
    - 我们常说的脱离文档流，其实就是脱离 Render Tree
5. 渲染树绘制

   在绘制阶段，浏览器会遍历渲染树，调用渲染器的`paint()`方法在屏幕上显示其内容。渲染树的绘制工作是由浏览器的UI后端组件完成的。

浏览器渲染过程中遇到JS文件怎么处理

- 渲染过程中，如果遇到 script 标签就会停止渲染，执行里面的JS代码。JS的加载、解析与执行会阻塞DOM的构建（将script放在底部的原因）
- 样式放在head中仅仅是为了更快的解析CSS，保证更快的首次渲染。

回流（重排）和重绘（reflow和repaint）

- 回流：对DOM元素的修改引发DOM元素的几何尺寸发生变化（比如修改元素的宽高，隐藏元素等）时，浏览器需要重新计算元素的几何属性（其他元素的几何位置也会因此受到影响），然后再将计算的结果绘制出来。这个过程就是回流（重排）
- 重绘：当我们对DOM的修改导致了样式的变化（比如修改了颜色或者背景颜色）却并未影响其几何属性，浏览器不需要重新计算元素的几何属性，直接为该元素绘制新的样式（跳过了重排的环节）。

当网页生成的时候，浏览器至少会渲染一次。在用户访问的过程中还会不断重新渲染。重新渲染会导致回流或者重绘。回流必定会引发重绘，但是重绘未必引发回流。不断的重绘和回流会影响页面的性能。回流的成本也远高于重绘

常见引发回流和重绘的操作

会使任何元素的几何属性发生变化的操作（如元素的位置和尺寸），都会触发回流。

- 添加或删除可见的DOM元素
- 元素尺寸的改变：边距、边框、宽高
- 内容变化，比如用户在input中输入文字
- 浏览器窗口尺寸的改变——resize事件
- 计算 offsetWidth 和 offsetHeight 属性

常见引起重绘的操作：color,background,border-style

减少reflow与repaint

- 避免逐个修改节点样式，尽量一次修改：使用 DocumentFragment将需要多次修改的DOM元素缓存之后，最后一次性append到真是的DOM中
- 使用 transform做形变和位移可以减少 reflow
- CSS选择符避免节点层级过多
- 避免多次读取某些属性

提升为合成层

某些特殊的渲染层会被认为是合成层（Compositing Layers），合成层拥有单独的 GraphicsLayer，而其他不是合成层的渲染层，则和其第一个拥有 GraphicsLayer 父层公用一个。

每个 GraphicsLayer 都有一个 GraphicsContext，GraphicsContext 会负责输出该层的位图，位图是存储在共享内存中，作为纹理上传到 GPU 中，最后由 GPU 将多个位图进行合成，然后 draw 到屏幕上，此时，我们的页面也就展现到了屏幕上。

渲染层提升为合成层有一个先决条件，该渲染层必须是 SelfPaintingLayer（基本可认为是 NormalPaintLayer）

NormalPaintLayer:

- 根元素
- 有明确定位的元素（relative、fixed、sticky、absolute）
- 透明的（opacity 小于 1）
- 有 CSS transform 属性（不为 none）

合成层简单来说有以下好处：

- 合成层的位图会交由 GPU 进行处理，比 CPU 处理快
- 当需要 repaint 时，只会 repaint 本身，不会影响其他的层级
- 对于 transform 和 opacity 效果，不会触发 layout 和 paint

合成层的好处是不会影响到其他元素的绘制，因此，为了减少动画元素对其他元素的影响，从而减少 paint，我们需要把动画效果中的元素提升为合成层。

提升合成层的最好方式是使用 CSS 的 will-change 属性。从上一节合成层产生原因中，可以知道 will-change 设置为 opacity、transform、top、left、bottom、right 可以将元素提升为合成层。

对于那些目前还不支持 will-change 属性的浏览器，目前常用的是使用一个 3D transform 属性来强制提升为合成层：

```
#target {
  transform: translateZ(0);
}
```

22. 跨域、同源策略及跨域实现方式和原理

两个不同源的URL在相互访问的时候就会产生跨域，同源的URL必须保证域名相同、端口相同、协议相同。跨域的解决方法：

(1) JSONP

```
// 封装一个 jsonp
function jsonp(url, params={}) {
    return new Promise((resolve, reject) => {
        window.__fn__ = data => resolve(data)

        // 获取参数
        let paramStr = Object.entries(params).map(item =>
`${item[0]}=${item[1]}`).join("&")

        // 动态添加 script 标签
        let script = document.createElement("script")

        script.scr = paramStr ? `${url}?callback=__fn__&${paramStr}` : `${url}?
callback=__fn__`

        script.onerror= () => reject("error")
        document.body.appendChild(script)
        document.body.removeChild(script)
    })
}
```

(2) CORS

CORS是跨域资源共享，它允许浏览器向跨源服务器发出 XMLHttpRequest 请求，从而克服 Ajax 只能同源使用的限制。相比于jsonp只能发送 get 请求， CORS 允许发送任何类型的请求。

整个 CORS 通讯过程都是浏览器自动完成的，不需要用户参与。CORS通讯和同源的Ajax请求没有区别。浏览器一旦发现 Ajax 请求跨域，就会自动添加一些头部信息，有时候还会多一次附加请求。

浏览器将 CORS 请求分为两大类：简单请求和非简单请求

只要同时满足一下两个条件就是简单请求，否则就是非简单请求：

（1）请求方法是下列方法之一：

- `HEAD`
- `GET`
- `POST`

（2）`http`的头信息不超出以下几个字段：

- `Accept`
- `Accept-Language`
- `Content-Language`
- `Last-Event-ID`

- `Content-Type`：只限于三个值`application/x-www-form-urlencoded、multipart/form-data、text/plain`

对于简单请求，浏览器会自动在头部信息里增加一个`Origin`字段，用来表示请求来自与哪个源，服务器根据这个值决定是否同意此次请求。如果`Origin`不在请求范围内，服务器返回一个正常的`http`回应。这个回应的头信息中没有`Access-Control-Allow-Origin`字段，浏览器发现没有这个字段之后就会抛出一个错误。如果`Origin`在请求范围内，服务器返回的响应会多出几个头信息字段，其中一个是`Access-Control-Allow-Origin`，它的值要么是`Origin`的值，要么是*，表示允许任何域名的请求。

对于非简单请求，它会在正式通信之前，增加一次`http`查询请求，称为"预检"请求（`preflight`）。通常是一个`OPTION`请求。这个请求先询问服务器，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`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。

23. JavaScript 中的 arguments

arguments对象是所有（非箭头）函数中都可用的局部变量。将其转化为真正的 Array 的方法：

- Array.prototype.slice.call(arguments)
- [].slice.call(arguments)
- Array.from(arguments)
- [...arguments]

arguments.callee 指向参数所属的当前执行环境。

24. EventLoop事件循环

JS中把所有任务分成了两类，一类是同步任务，一类是异步任务。同步任务指的是在主线程上排队执行的任务，只有前一个任务执行完毕，才能执行后一个任务；异步任务指的是，不进入主线程，而是进入"任务队列"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。

具体来说，异步执行的机制如下：

- 所有同步任务都在主线程上执行，形成一个执行栈
- 主线程之外还有一个任务队列，只要异步任务有个结果，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毕，系统就会读取任务队列，看看里面有哪些事件。那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。

任务队列是一个事件队列，也可以理解成消息队列，I/O设备完成一项任务，就在任务队列中添加一个事件，表示相关的异步任务可以进入执行栈了。主线程读取任务队列，就是读取里面有哪些事件。

任务队列中的事件，还包括一些用户产生的事件（比如鼠标点击，页面滚动等等）。只要指定过回调函数，这些事件发生时就会进入"任务队列"，等待主线程读取。

25. 发布订阅者模式与观察者的实现

> 观察者模式定义了对象之间的一种一对多的关系，当目标对象 Subject 的状态发生改变的时候，所有依赖它的对象 Observe 都会得到通知。

Subject 具有的方法为：添加、删除、通知 Observe

Observe 具有的方法为：接受 Subject 的状态变更并处理

Subject 被称为发布者，即被观察者

Observe 被称为订阅者，即观察者

观察者模式的实现：

```
// 定义被观察者
class Subject {

  constructor() {
    // 观察者数组
    this.observers = []
  }

  add(observer) {
    this.observers.push(observer)
  }
  remove(observer) {
    let index = this.observers.findIndex(item => item == observer)
    if(index != -1) {
      this.observers.splice(index, 1)
    }
  }
  notify() {
    this.observers.map(item => item.update())
  }
}
// 定义观察者
class Observe {
  constructor(name) {
    this.name = name
  }
  update() {
    console.log("我被通知了");
  }
}
```

> 发布订阅者模式基于一个事件通道，希望接受通知的对象订阅一个事件，被激活事件的对象通过发布事件的方式通知各个订阅了该事件的对象。

以下是发布订阅者模式的简单实现

```
function PubSub() {
    this.handles = {}
    // 订阅事件
    // 事件类型以及事件触发时的回调函数
    this.subscribe = function(eventType, callback) {
        // 如果只是一个函数的话，后面绑定的相同事件的回调函数会覆盖之前的回调函数
        // this.handles[eventType] = callback
        // 因此考虑把每个事件类型的回调函数设置成一个数组，触发事件的时候，依次执行数组里面的函数
        if(!this.handles[eventType]) {
            // 初始化
            this.handles[eventType] = []
        }
        this.handles[eventType].push(callback)
```

```
    }

    // 取消事件
    // 取消指定 eventType 事件下的 fn 回调
    this.unsubscribe = function(eventType, fn) {
      if(!this.handles[eventType]) return
      let fnList = this.handles[eventType]
      if(!fn) {
        // 如果不传入指定回调，则清除该 eventType 下的所有回调
        fnList.length = 0
      } else {
        this.handles[eventType] = fnList.filter(item => item != fn)
      }
    }

    // 发布事件
    // 事件类型以及需要传递的参数
    this.publish = function(eventType, ...args) {
        // 触发事件的时候就执行该事件绑定的回调函数
        // this.handles[eventType].apply(null, args)
        // 现在需要遍历执行
        this.handles[eventType].map(fn => fn.apply(null, args))
    }
}
let pubSub = new PubSub()

let fn1 = time => {
  console.log("上班了", time)
}
pubSub.subscribe("onwork", fn1)
pubSub.subscribe("offwork", time => {
  console.log("下班了", time)
})
pubSub.subscribe("onwork", time => {
  console.log("上班了2", time)
})

pubSub.publish("onwork", "12:00")
pubSub.unsubscribe("onwork", fn1)
pubSub.publish("onwork", "12:00")
```

两者的区别：

- 观察者模式维护单一事件对应多个依赖该事件之间的对象关系
- 发布订阅者模式维护多个事件及依赖各事件的对象之间的关系
- 观察者模式是目标对象直接触发通知，观察对象被迫接受通知；发布订阅者模式多了一个事件中心，由其去管理通知广播（只通知订阅对应事件的对象）
- 观察者模式对象间依赖关系较强，发布订阅者模式中对象之间实现真正的解耦

26. 函数柯里化及其通用封装

柯里化（Currying）是把接受多个参数的函数变换成接受一个单一参数的函数，并且返回接受余下的参数且返回结果的新技术。

函数柯里化其含义就是给函数分步传参，每次传递部分参数，并返回一个更具体的函数接受剩余的参数。这中间可能接受多层这样的接受部分参数的函数，直至返回结果。

```
function curry(fn, args) {
    // 获取fn的形参个数
    var arity = fn.length
    // 获取上一次的参数
    var args = args || []
    // 返回一个函数
    return function() {
        // 获取本次的参数并转化为数组
        var _args = Array.prototype.slice.call(arguments)
        // 将本次参数与上次参数合并
        Array.prototype.unshift.apply(_args, args)
        // 判断参数个数是否等于形参个数
        if(_args.length < arity) {
            // 如果小于形参个数，继续收集参数
            return curry(fn,_args)
        }
        // 否则执行函数的返回结果
        return fn.apply(null, _args)
    }
}
```

27. “==”和“===” 以及 Object.is() 的区别

"=="允许在相等比较过程中进行强制类型转换，而“===”不允许。

1. 字符串和数字之间的比较：字符串执行 ToNumber() 转化为数字
2. 其他类型和布尔值之间的比较：布尔值执行 ToNumber() 转化为数字
3. null只==undefined，其他值和 null 或 undefined 比较均不相等
4. 对象和基本类型的比较，对象执行 ToPrimitive() 转化为基本类型

在 === 中， NaN !== NaN, -0 === +0

而在 Object.is 中，Object.is(NaN, NaN) -> true , Object.is(-0, +0) ->false，其余与 === 一致

实现一个 Object.is:

```
Object.prototype.is = function(a, b) {
    if(a === b) {
        // 可能为 -0 +0
        return x !== 0 || 1/x === 1/y
    }
    // x 与 y 都为 NaN
    return x !== x && y !== y
}
```

28. let、const和var的概念与区别

let，const声明的变量只在其所在的代码块内有效。let声明的变量可以进行修改，而 const 用于定义不可修改的常量。

var不存在块级作用域。区别：

- let,const不存在变量提升，var会有变量提升。即let，const一定要先声明，后使用
- 如果区域块内存在 let 和 const命令，则这个区域块内对这些命令声明的变量从一开始就形成了封闭作用域。只要在声明前使用这些变量，就会报错。
- let 不会成为 window 的属性

29. Symbol概念及其作用

ES5的对象属性名都是字符串，这容易造成命名冲突。ES6引入一种新的数据类型：Symbol，用于创建独一无二的值。Symbol 通过 Symbol() 函数生成。

Symbol作为属性名，该属性不会出现在 for in, for of 循环中，也不会被 Object.keys()、Object.getOwnPropertyNames() 返回。但他也不是私有属性，有一个 Object.getOwnPropertySymbols 方法可以获取对象的所有 Symbol 属性。

30. Set 和Map 数据结构

set的方法：add、delete、has

map的方法：set、delete、has、get

set 无序且不重复

Map和WeakMap的区别：

WeakMap 的 key 只能是对象，且不能被遍历。当把 WeakMap 里面的 key 设为 null 之后，可以被垃圾回收机制回收，可以用于存储DOM节点。

```
let a = {name: "sillywa"}
let b = {age: 90}
let map = new Map()
let weakMap = new WeakMap()
map.set(a, "test1")
weakMap.set(b, "test2")

a=null
b=null
map.get(a)
map.get(b)
```

31. XSS 和 CSRF 攻击

CSRF：跨站请求伪造（Cross-site Request forgery）

用户登录 A 网站得到 cookie，然后访问危险的 B 网站，B网站携带A的cookie发送A的请求

XSS：跨域脚本攻击（Cross Site Scripting）

向网站 A 注入 JS代码，然后执行 JS 里的代码，篡改网站A的内容

32. 浏览器进程及重要的线程

- **浏览器主进程**，负责协调和控制各个页面
- **插件进程**：使用插件时会创建
- **GPU进程**：主要负责 UI 渲染
- **浏览器渲染进程**：负责页面渲染，脚本执行，事件处理

浏览器内核，即我们的浏览器渲染进程，有名为 Renderer 进程，我们页面的渲染，js的执行，事件的循环都在这一个进程内进行，也就是说，该进程下面拥有多个线程，靠着这些线程共同完成渲染任务，那么这些线程是什么，如下：

1. 图形用户界面 **GUI 渲染线程**
    - 负责渲染浏览器界面，包括解析 HTML、CSS、构建 DOM 树，Render 树、布局与绘制等
    - 当界面需要重绘的时候或由于某种操作引发回流时，该线程就会执行
2. **JS 引擎线程**
    - JS内核，也称 JS 引擎，负责处理执行 javascript 脚本
    - 等待任务队列的任务的到来，然后加以处理，浏览器无论什么时候都只有一个 JS 引擎在运行 JS 程序
3. **事件触发线程**
    - 听起来像 JS 的执行，但是其实归属于浏览器，而不是JS引擎，用来控制时间循环（可以理解，JS引擎自己都忙不过来，需要浏览器另开线程协助）
    - 当 JS 引擎执行代码块如 setTimeout 时（也可以来自浏览器内核的其他线程，如鼠标点击、AJAX 异步请求等），会将对应任务添加到事件线程中
    - 当对应的事件符合触发条件被触发时，该线程会把事件添加到待处理队列的队尾，等待 JS 引擎的处理

      注意：由于 JS 单线程关系，所以这些待处理队列中的事件都得排队等待 JS 引擎处理（当 JS 引擎空闲时才会去执行）
4. **定时器触发线程**
    - setInterval 和 setTimeout 所在线程
    - 定时器计时并不是由 JS 引擎计时的，因为 JS 引擎是单线程的，如果 JS 引擎处于堵塞状态，那么会影响到计时的准确性
    - 当计时完成被触发，事件会被添加到事件队列，等待 JS 引擎空闲了执行
5. **异步 HTTP 请求线程**
    - 在 XMLHttpRequest 连接后启动的一个线程
    - 线程如果检测到请求的状态变更，如果设置有回调函数，该线程会把回调函数添加到事件队列，等待 JS 引擎空闲了执行

33. 为什么 JS 引擎是单线程的

如果 JS 设计成多线程，那么在进行 DOM 操作的时候就可能存在两个线程同时修改同一个 DOM 节点的情况，对于这种情况，DOM就成为了一种临界资源，我们还需要实现互斥访问才能解决，这样会使得 JS 引擎变得更加复杂。

34. 为什么 GUI 渲染线程与 JS 引擎线程互斥

因为JS引擎可以修改DOM树，那么如果JS引擎在执行修改了DOM结构的同时，GUI线程也在渲染页面，那么这样就会导致渲染线程获取的DOM的元素信息可能与JS引擎操作DOM后的结果不一致。为了防止这种现象，GUI线程与JS线程需要设计为互斥关系，当JS引擎执行的时候，GUI线程需要被冻结。

35. JS 引擎线程与事件触发线程、定时器触发线程、异步 HTTP 请求线程

事件触发线程、定时触发器线程、异步HTTP请求线程三个线程有一个共同点，那就是使用回调函数的形式，当满足了特定的条件，这些回调函数会被执行。这些回调函数被浏览器内核理解成事件，在浏览器内核中拥有一个事件队列，这三个线程当满足了内部特定的条件，会将这些回调函数添加到事件队列中，等待JS引擎空闲执行。

但是，JS引擎对事件队列（宏任务）与JS引擎内的任务（微任务）执行存在着先后循序，当每执行完一个事件队列的时间，JS引擎会检测内部是否有未执行的任务，如果有，将会优先执行（微任务）。

36. 前端常见性能优化

- 资源压缩与合并：减少文件体积，提高加载速度
- 减少 http 请求
- 利用浏览器缓存：提升二次访问的速度
- 使用 CDN

利用浏览器缓存也仅仅只能提高二次访问的速度，对于首次访问的加速，可以从网络层面进行优化，最常见的手段就是 CDN（Content Delivery Network，内容分发网络）加速。

通过将静态资源缓存到离用户很近的相同网络运营商的 CDN 节点上，不但能提升用户的访问速度，还能节省服务器的带宽消耗。

CDN 是怎么做到加速的呢?

首先 CDN 服务商会在全国各个网站部署计算节点，CDN 加速将网站的内容缓存在网络边缘，不同地区的用户就会访问到离自己最近的相同网络线路上的 CDN 节点。当请求到达 CDN 节点之后，节点会判断自己的缓存内容是否有效，如果有效，就会立即响应缓存的内容给用户，从而加快响应速度。如果CDN节点的缓存失效，它会根据服务器的配置去我们的内容源服务器获取最先的资源响应给用户，并将内容缓存下来以便响应后续访问的用户。

因此，**一个地区只要有一个用户先加载资源，在 CDN 中建立了缓存，该地区其他用户访问相同的资源的时候就可以使用缓存了。**

- 将 CSS 文件放在头部， JS 文件放在尾部

  CSS文件放在头部的原因是为了让用户第一时间看到的页面是有样式的。

  另外，JS 文件也不是不可以放在头部，只要给script标签加上 defer 属性就可以异步加载，延迟执行了。
- 图片懒加载

- 减少重绘和重排

37. defer 和 async 的区别

两者都是异步下载，加载文件的时候不阻塞页面的渲染，但是执行时刻不一样。

async 脚本在他下载结束之后立刻执行，同时会在 window.onload 事件之前执行，所以就有可能出现脚本执行顺序被打乱的情况

defer 的脚本都是在页面解析完毕之后，按照原本的顺序执行，同时会在 document.DOMContentLoaded 之前执行

38. Object.defineProperty 与 Proxy 的区别

两者都可以用于做数据劫持，它们的区别如下：

- Proxy 代理整个对象，而 Object.defineProperty 只代理对象上的某个属性
- 如果对象内部要全部递归代理， Proxy 可以只在调用时递归，而 Object.defineProperty 需要在一开始就全部递归，因此 Proxy 的性能较优
- 对象上定义新的属性的时候，Proxy可以监听到，而 Object.defineProperty 监听不到
- 数组新增删除修改时，Proxy可以监听到，Object.defineProperty监听不到。

39. 单页应用的好处

- 单页应用只需要获取一个 HTML 文件，相比传统的网页，http请求会有所减少，用户体验更好。
- 开发模式上可以使用前后端分离

坏处：

- SEO难度较高（爬虫只会爬取源码，不会执行脚本，而SPA，页面的DOM元素都是由js生成的，可供去爬取的内容大大减少）
- 初次加载耗时较多（打包后文件体积比较大，普通客户端渲染加载所有所需文件时间较长）

40. 使用 IntersectionObsever API 监听元素出现在视图

在开发中，常需要知道某个元素是否进入了视图，传统的实现方法是监听页面的 scroll 事件，调用目标的 getBoundingClientRect() 方法，得到它对应于视口左上角的坐标，再判断是否在视口之内。这种方法的缺点就是，由于 scroll 事件比较密集，计算量大，很容易造成性能问题。

目前有一个新的 IntersectionObserver API，可以自动观察元素是否可见，Chrome51+已经支持，IE不支持。

使用方法：

```
let io = new IntersectionObserver(callback, option)
io.observe(document.querySelector(".inner"))

io.unobserve(element)

// 关闭观察器
io.disconnect()
```

其中 callback 是一个函数，接受一个参数：

```
let callback = function(entries) {
    console.log(entries)
    // 可以通过这个判断元素是否进入视图
    if(entries[0].isIntersecting) {
        console.log("进入视图啦");
    }
    // console.log(entries[0].intersectionRatio);

}
```

这个参数为一个数组，里面保存着被监听的每一个元素的一些信息，是一个IntersectionObserverEntry，该对象的属性如下：

```
IntersectionObserverEntry = {
    ` `
    boundingClientRect: DOMRectReadOnly {x: 8, y: 524, width: 283, height: 100,
top: 524, …}
    // 目标的可见比例，已见部分占整个元素的比例 0-1
    intersectionRatio: 0
    // 目标元素与视口（或根元素）的交叉区域的信息
    intersectionRect: DOMRectReadOnly {x: 0, y: 0, width: 0, height: 0, top: 0,
…}
    // 判断元素是否在视图
    isIntersecting: false
    isVisible: false
    // 根元素矩形区域的信息，getBoundingClientRect()方法的返回值
    rootBounds: DOMRectReadOnly {x: 0, y: 0, width: 896, height: 937, top: 0, …}
    // 被观察的目标元素
    target: div.inner
    // 可见性发生变化的事件
    time: 3280.2300000330433
}
```

41. gitflow 工作流

该工作流首先会新建两个分支，一个是 master 分支，用户保存官方的发布历史，还有一个是 develop 分支，用于开发集成各种功能。

所有开发者都应该在 develop 分支上建立自己的分支进行开发，当自己的功能开发完成之后，请求合并到 develop 分支。

一旦 develop 分支聚集了足够多的新功能之后，可以基于 develop 分支创建一个用户产品发布的 release 分支，这个分支意味着一次发布的开始，并且本次发布不再增加新功能。

在一切准备就绪之后，这个 release 分支会被合并到 master 分支，并且用版本号打上标签。

当线上产品出现 bug 的时候，可以新建一个用于维护的分支， hotfix 分支，并且维护之后直接合并入 master 分支。

42. 服务端渲染与浏览器渲染

浏览器渲染：页面上的内容是**加载进来的 js 文件渲染出来的，js 文件运行在浏览器上面**，服务端只返回一个 html 模板。

服务端渲染：页面的内容是通过服务端渲染出来的，浏览器直接显示服务端返回的 html 文件

优势：

- **利于SEO**，服务器返回给客户端的是已经渲染了内容的完整的HTML文件，网络爬虫可以抓取到页面的完整信息
- **利于首屏渲染**，首屏的渲染是服务端发送过来的html字符串，并不依赖于js文件了，这就会使用户更快的看到页面的内容

43. webpack 打包原理

webpack在打包的时候，他会从入口开始，递归地构建一个依赖图，其中包含应用程序对应的一个或多个模块，然后将它们打包成一个或多个 bundle.js

44. CommonJS与ES6模块的差异

- CommonJS 是运行时加载，ES6 模块是编译时输出接口
- CommonJS 输出的是一个值的复制，ES6 输出的是值得引用
- ES6 module 在编译期间会将所有 import 提升到顶部， commonjs 不会提升 require

45. 箭头函数和非箭头函数的区别

- 箭头函数里面的 this 绑定的是它定义时所在的作用域，而不是使用时动态绑定
- 不可以用作构造函数
- 不可以使用 arguments 对象，该对象在函数体内不存在；如果要用，可以使用 rest 参数代替
- 不可以使用 yield 命令，因此箭头函数不能用作 Generator

46. 数组扁平化的几种方式

1. 简单的遍历

```
function flatten(arr) {
    let result = []
    for(let item of arr) {
        if(Array.isArray(item)) {
            result = result.concat(flatten(item))
        } else {
            result.push(item)
        }
    }
    return result
}
```

2. reduce

```
function flatten(arr) {
    arr.reduce((origin, item) => {
      return  Array.isArray(item) ? origin.concat(flatten(item)) :
[...origin, item]
    }, [])
}
```

3. toString

   只针对数字数组

```
function flatten(arr) {
    return arr.toString().split(",").map(item => Number(item))
}
```

4. some 与 扩展运算符

```
function flatten(arr) {
    while(arr.some(item => Array.isArray(item))){
        arr = [].concat(...arr)
    }
    return arr
}
```

5. generator 函数

```
function flatten(arr) {
    function *fn(arr) {
        for(let item of arr) {
            if(Array.isArray(item)) {
                yield *fn(item)
            } else {
                yield item
            }
        }
    }
    return [...fn(arr)]
}
```

47. input change keyup的区别

input：输入了内容就会触发 input 事件

keyup：键盘抬起触发，不抬起不触发

change：当输入框失去焦点并且内容有变化的时候触发

blur：失去焦点触发