网页示例


```html
<html>
  <head>
    <meta charset="utf-8" />
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1.0, maximum-scale=1.0, user-scalable=no"
    />
    <title>蓝桥知识网</title>
    <link rel="stylesheet" href="./css/style.css" />
  </head>
  <body>
     <!--TODO：请补充代码-->
  <div class="head">
    <ul class="nav">
      <li>蓝桥知识网</li>
      <li>首页</li>
      <li>热门技术</li>
      <li>使用手册</li>
      <li>知识库</li>
      <li>练习题</li>
      <li>联系我们</li>
      <li>更多</li>
    </ul>
    <h1></h1>
    <p class="h1">蓝桥云课</p>
    <p class="h2">随时随地丰富你的技术栈！</p>
```

```
    <div class="box1">
      <p class="h3">加入我们</p>
    </div>
  </div>

  <div class="body">
    <p><span>人工智能</span><br>
      人工智能亦称智械、机器智能，指由人制造出来的机器所表现出来的智能。通常人工智能是指通过普通计算机程序来呈现人类智能的技术。
    </p>
    <p><span>前端开发</span><br>
      前端开发是创建 WEB 页面或 APP 等前端界面呈现给用户的过程，通过 HTML，CSS 及 JavaScript 以及衍生出来的各种技术、框架、解决方案，来实现互联网产品的用户界面交互。
    </p>
    <p><span>后端开发</span><br>
      后端开发是实现页面的交互逻辑，通过使用后端语言来实现页面的操作功能，例如登录、注册等。
    </p>
    <p><span>信息安全</span><br>
      ISO（国际标准化组织）的定义为：为数据处理系统建立和采用的技术、管理上的安全保护，为的是保护计算机硬件、软件、    <br>数据不因偶然和恶意的原因而遭到破坏、更改和泄露。
    </p>
  </div>

  <div class="footer">

    <p>© 蓝桥云课 2022</p>
    <p>京公网安备 11010102005690 号 | 京 ICP 备 2021029920 号</p>

  </div>
  </body>
</html>
```

```
/*
 TODO：请补充代码
*/
.one{
    background-color: #a6b1e1;
    height: 486px;
    display: flex;
    justify-content: center;
}
#one{
    width: 1/*
 TODO：请补充代码
*/
/*
 TODO：请补充代码
*/
* {
    margin: 0;
}

```

```
.head {
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content: center;
    width: auto;
    height: 486px;
    background-color: #a6b1e1;
}

.nav {
    margin-top: -170px;
    width: 1024px;
}


.nav li {
    list-style: none;
    display: inline;
    color: white;
    font-size: 16px;
}

.nav li:nth-child(1) {
    font-size: 18px;
}

.nav li:nth-child(2) {
    margin-left: 365px;
}

.nav li:nth-child(2)~li {
    margin-left: 16px;
}

.h1 {
    margin-top: 30px;
    font-size: 45px;
    color: black
}

.h2 {
    margin-top: 62px;
    color: white;
    font-size: 21px;
    font-weight: 200;
}

.box1 {
    margin-top: 36px;

}

.h3 {
    padding: 10px;
    color: #efbfbf;
```

```
    border: 2px solid #efbfbf;
    border-radius: 2px;
    font-size: 18px;
    box-shadow: inset 0 0 0 2 px #efbfbf;
}

.body {
    display: grid;
    margin-top: 74px;
    width: 1024px;
    height: 302px;
    grid-template-columns: 1fr 1fr;
    grid-template-rows: 1fr 1fr;
    margin-left: 50%;
    transform: translateX(-50%);
}

.body span {
    display: block;
    font-size: 30px;
    font-weight: 200;
    color: black;
}

.body p {
    height: 144px;
    font-size: 18px;
    line-height: 1.4em;
    color: #aaa;
}
.body p:nth-child(2n+1) {
   margin-right: 20px;
}

.footer{
    width: 1024px;
    height: 80px;
    margin-left: 50%;
    transform: translateX(-50%);
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content: center;
}
.footer p{
    font-size: 14px;
    color: #aaa;
}


总像素：1191630
像素差：6%
得分：14024px;
    /* border: 1px gray solid; */
}
nav{
```

```
    margin-top: 13px;
    height: 46px;
    /* border: 1px gray solid; */
}
nav ul{
    /* border: 1px red solid; */
    display: flex;
    color: white;
    /* gap:16px; */

}
nav ul li:first-child{
    /* border: 1px gray solid; */
    font-size: 18px;
    margin-right: 365px;
}
nav ul li:nth-child(n+2){
    font-size: 16px;
    margin-right: 16px;
}
.tit div{
    font-size: 45px;
    color: black;
    /* border: 1px gray solid; */
}
.tit{
    /* border: 1px gray solid; */
    display: flex;
    justify-content: center;
    margin-top: 30px;
}
.des div{
    font-size: 21px;
    color: white;
    font-weight: 200;
    /* border: 1px gray solid; */
}
.des{
    /* border: 1px gray solid; */
    display: flex;
    justify-content: center;
    margin-top: 62px;
}
button{
    background-color: transparent;
    box-shadow: inset 0 0 0 2px #efbfbf;
    border: #efbfbf 0.1px solid;
    border-radius: 2px;
}
button div{
    font-size: 18px;
    background-color: transparent;
    color: #efbfbf;

    padding: 10px;
    /* box-shadow: inset 0 0 0 2px #efbfbf; */
```

```
}
.join{
    display: flex;
    justify-content: center;
    margin-top: 36px;
    /* border: 1px gray solid;  */
}
.two{
    /* border: 1px red solid;  */
    height: 376px;
    display: flex;
    justify-content: center;
}

#two{
    margin-top: 74px;
    width: 1024px;
    /* border: 1px gray solid;  */
}

table{
    height: 144px;
}

th{
    font-size: 30px;
    color: black;
    font-weight: 200;
    text-align: left;
}
td{
    font-size: 18px;
    color: #aaa;
    line-height: 1.4em;
    width: 502px;
    vertical-align: top;
}
tbody tr td:first-child{
    padding-right: 20px;
}
footer{
    height: 80px;
    border-top: 1px #aaa solid;
    display: flex;
    justify-content: center;

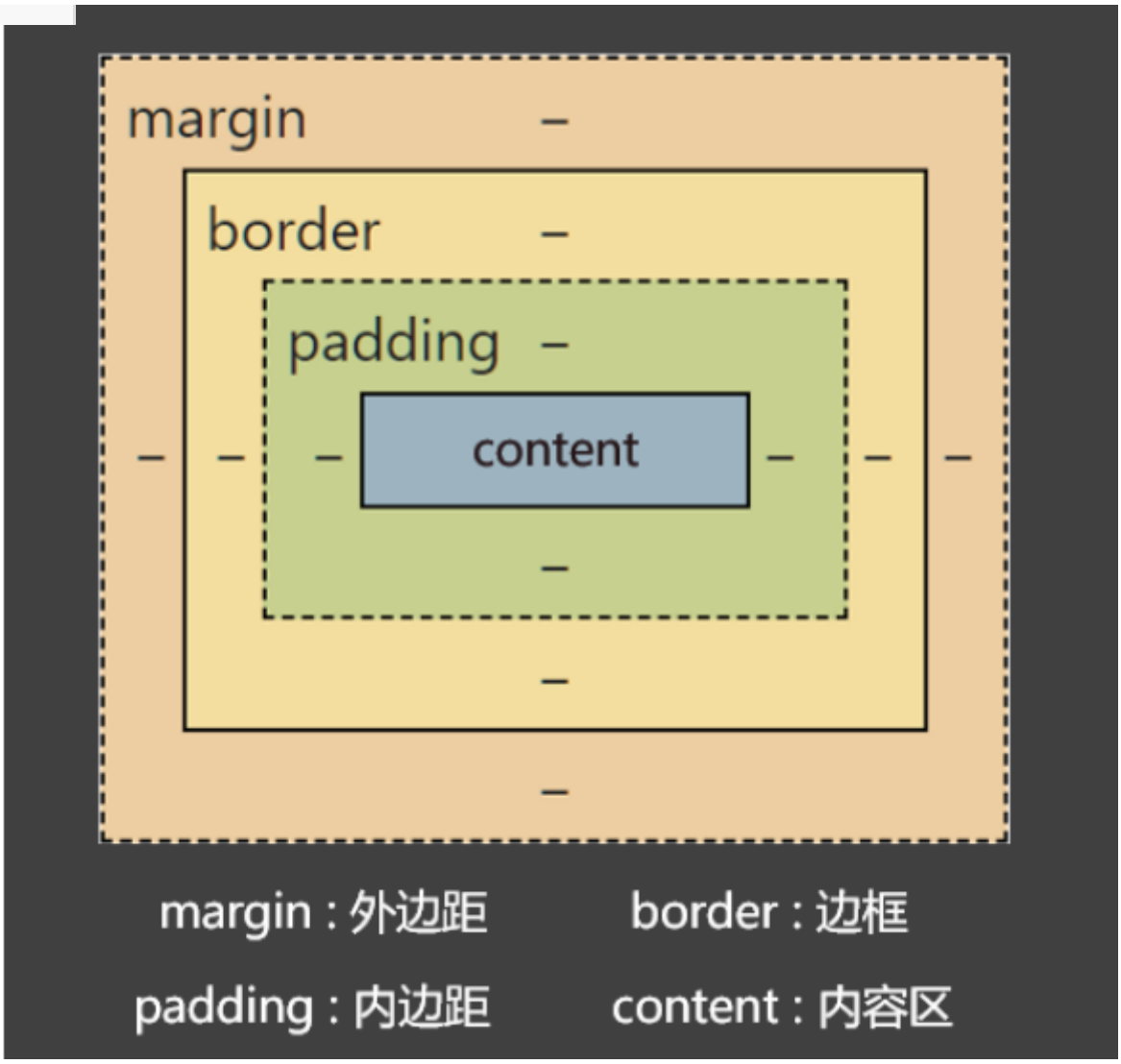
}
#three{
    width: 1024px;
}
.z1 div{
    font-size: 14px;
    color: #aaa;
}
.z1{
    display: flex;
```

```
    justify-content: center;
    margin-top: 30px;
}
.z2 div{
    font-size: 14px;
    color: #aaa;
}
.z2{
    display: flex;
    justify-content: center;
    margin-top: 10px;
}
```

盒子模型

1.组成部分

1. margin（外边距）： 盒子与外界的距离。
2. border（边框）： 盒子的边框。
3. padding（内边距）： 紧贴内容的补白区域。
4. . content（内容）：元素中的文本或后代元素都是它的内容。



盒子的大小 = content + 左右 padding + 左右 border 。

注意：外边距 margin 不会影响盒子的大小，但会影响盒子的位置。

2.盒子内容区（content）

CSS 属性名	功能	属性值
`width`	设置内容区域宽度	长度
`max-width`	设置内容区域的最大宽度	长度
`min-width`	设置内容区域的最小宽度	长度
`height`	设置内容区域的高度	长度
`max-height`	设置内容区域的最大高度	长度
`min-height`	设置内容区域的最小高度	长度

3.盒子内边距（padding）

CSS 属性名	功能	属性值
padding-top	上内边距	长度
padding-right	右内边距	长度
padding-bottom	下内边距	长度
padding-left	左内边距	长度
padding	复合属性	长度，可以设置 1 ~ 4 个值

padding 复合属性的使用规则：

1. padding: 10px; 四个方向内边距都是 10px 。
2. padding: 10px 20px; 上 10px ， 左右 20px 。 （上下、左右）
3. padding: 10px 20px 30px; 上 10px ， 左右 20px ， 下 30px （上、左右、下）
4. padding: 10px 20px 30px 40px; 上 10px ， 右 20px ， 下 30px ， 左 40px 。 （上、右、下、左）

4.盒子边框（border）

CSS 属性名	功能	属性值
`border-style`	边框线风格 复合了四个方向的边框风格	`none`：默认值 `solid`：实线 `dashed`：虚线 `dotted`：点线 `double`：双实线 ……
`border-width`	边框线宽度 复合了四个方向的边框宽度	长度，默认 `3px`
`border-color`	边框线颜色 复合了四个方向的边框颜色	颜色，默认黑色
`border`	复合属性	值没有顺序和数量要求。

`border-left`
`border-left-style`
`border-left-width`
`border-left-color`
 	
`border-right`
`border-right-style`
`border-right-width`
`border-right-color`
 	
`border-top`
`border-top-style`
`border-top-width`
`border-top-color`
 	
`border-bottom`
`border-bottom-style`
`border-bottom-width`
`border-bottom-color`
	分别设置各个方向的边框	同上

5.盒子外边距（margin）

CSS 属性名	功能	属性值
`margin-left`	左外边距	CSS 中的长度值
`margin-right`	右外边距	CSS 中的长度值
`margin-top`	上外边距	CSS 中的长度值
`margin-bottom`	下外边距	CSS 中的长度值
`margin`	复合属性，可以写 1~4 个值，规律同 `padding`（顺时针）	CSS 中的长度值

浮动

1.元素浮动后的特点

1. 脱离文档流。
2. 不管浮动前是什么元素，浮动后：默认宽与高都是被内容撑开（尽可能小），而且可以设置宽高。
3. 不会独占一行，可以与其他元素共用一行。
4. 不会 margin 合并，也不会 margin 塌陷，能够完美的设置四个方向的 margin 和 padding 。
5. 不会像行内块一样被当做文本处理（没有行内块的空白问题）

2.解决浮动产生的影响

2.1元素浮动后会有哪些影响

对兄弟元素的影响： 后面的兄弟元素，会占据浮动元素之前的位置，在浮动元素的下面；

对前面的兄弟 无影响。 对父元素的影响： 不能撑起父元素的高度，导致父元素高度塌陷；但父元素的宽度依然束缚浮动的元素。

2.2 解决浮动产生的影响（清除浮动）

给浮动元素的父元素，设置伪元素，通过伪元素清除浮动

```
.parent::after{
        content : " ";
        display : block;
        clear : both;
}
```

定位

1.相对定位

1.1 如何设置相对定位?

给元素设置 `position:relative` 即可实现相对定位。

可以使用 left 、 right 、 top 、 bottom 四个属性调整位置

1.2 相对定位的参考点在哪里?

相对自己原来的位置

1.3 相对定位的特点:

1. **不会脱离文档流，元素位置的变化，只是视觉效果上的变化，不会对其他元素产生任何影响。**
2. 定位元素的显示层级比普通元素高，无论什么定位，显示层级都是一样的。 默认规则是： 定位的元素会盖在普通元素之上。 都发生定位的两个元素，后写的元素会盖在先写的元素之上。
3. left 不能和 right 一起设置， top 和 bottom 不能一起设置。
4. 相对定位的元素，也能继续浮动，但不推荐这样做。
5. 5. 相对行为的元素，也能通过 margin 调整位置，但不推荐这样做。

注意：**绝大多数情况下，相对定位，会与绝对定位配合使用。（子绝父相）**

2.绝对定位

2.1 如何设置绝对定位?

给元素设置 `position: absolute` 即可实现绝对定位。

可以使用 left 、 right 、 top 、 bottom 四个属性调整位置

2.2绝对定位的参考点在哪里?

参考它的包含块。

什么是包含块?

1. 对于没有脱离文档流的元素：包含块就是父元素； （也就是说父元素有`position：relative`->相对定位不脱离文档流，此时参考位置相对于父元素左上角）
2. 对于脱离文档流的元素：包含块是第一个拥有定位属性的祖先元素（如果所有祖先都没定位，那包含块就是整个页面->即body页面）。

2.3 绝对定位元素的特点：

1. 脱离文档流，会对后面的兄弟元素、父元素有影响。
2. left 不能和 right 一起设置， top 和 bottom 不能一起设置。
3. 绝对定位、浮动不能同时设置，如果同时设置，浮动失效，以定位为主。
4. 绝对定位的元素，也能通过 margin 调整位置，但不推荐这样做。
5. 无论是什么元素（行内、行内块、块级）设置为绝对定位之后，都变成了定位元素。

3.固定定位（页面中的广告）

3.1 如何设置为固定定位?

给元素设置 `position: fixed` 即可实现固定定位。

可以使用 left 、 right 、 top 、 bottom 四个属性调整位置。

3.2 固定定位的参考点在哪里?

参考它的视口

什么是视口？—— 对于 PC 浏览器来说，视口就是我们看网页的那扇“窗户”。

3.3 固定定位元素的特点

1. 脱离文档流，会对后面的兄弟元素、父元素有影响。
2. left 不能和 right 一起设置， top 和 bottom 不能一起设置。
3. 固定定位和浮动不能同时设置，如果同时设置，浮动失效，以固定定位为主。
4. 固定定位的元素，也能通过 margin 调整位置，但不推荐这样做。
5. 无论是什么元素（行内、行内块、块级）设置为固定定位之后，都变成了定位元素。

4.粘性定位

4.1 如何设置为粘性定位?

给元素设置 `position:sticky` 即可实现粘性定位。

可以使用 left 、 right 、 top 、 bottom 四个属性调整位置，不过最常用的是 top 值。

4.2 粘性定位的参考点在哪里?

离它最近的一个拥有“滚动机制”的祖先元素，即便这个祖先不是最近的真实可滚动祖先。

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
```

```
    <meta charset="UTF-8">
    <title>04_粘性定位</title>
    <style>
        * {
            margin: 0;
            padding: 0;
        }
        body {
            height: 2000px;
        }
        .page-header {
            height: 100px;
            text-align: center;
            line-height: 100px;
            background-color: orange;
            font-size: 20px;
        }
        /* .content { */
            /* height: 200px; */
            /* overflow: auto; */
            /* overflow: scroll; */
        /* } */
        .item {
            background-color: gray;
        }
        .first {
            background-color: skyblue;
            font-size: 40px;
            position: sticky;
            top: 0px;
            /* float: right; */
            /* margin-right: 100px; */
        }
    </style>
</head>
<body>
    <!-- 头部 -->
    <div class="page-header">头部</div>
    <!-- 内容区 -->
    <div class="content">
        <!-- 每一项 -->
        <div class="item">
            <div class="first">A</div>
            <h2>A1</h2>
            <h2>A2</h2>
            <h2>A3</h2>
            <h2>A4</h2>
            <h2>A5</h2>
            <h2>A6</h2>
            <h2>A7</h2>
            <h2>A8</h2>
        </div>
        <div class="item">
            <div class="first">B</div>
            <h2>B1</h2>
            <h2>B2</h2>
```

```
            <h2>B3</h2>
            <h2>B4</h2>
            <h2>B5</h2>
            <h2>B6</h2>
            <h2>B7</h2>
            <h2>B8</h2>
        </div>
        <div class="item">
            <div class="first">C</div>
            <h2>C1</h2>
            <h2>C2</h2>
            <h2>C3</h2>
            <h2>C4</h2>
            <h2>C5</h2>
            <h2>C6</h2>
            <h2>C7</h2>
            <h2>C8</h2>
        </div>
    </div>
</body>
</html>
```

4.3粘性定位元素的特点

不会脱离文档流，它是一种专门用于窗口滚动时的新的定位方式。

最常用的值是 top 值。

粘性定位和浮动可以同时设置，但不推荐这样做。

粘性定位的元素，也能通过 margin 调整位置，但不推荐这样做。

注意：粘性定位和相对定位的特点基本一致，不同的是：粘性定位可以在元素到达某个位置时将其固定。

flex布局

1.开启flex

给元素设置: `display:flex` 或 `display:inline-flex` ,该元素就变为了伸缩容器。(display :inline-flex 很少使用，因为可以给多个伸缩容器的父容器，也设置为伸缩容器)

2.主轴与侧轴

主轴： 伸缩项目沿着主轴排列，主轴默认是水平的，默认方向是：从左到右（左边是起点，右边是终点）。

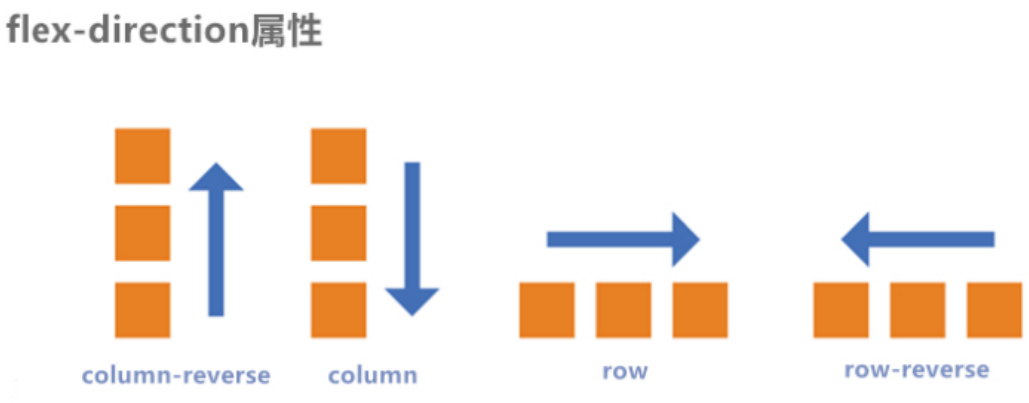
侧轴： 与主轴垂直的就是侧轴，侧轴默认是垂直的，默认方向是：从上到下（上边是起点，下边是终点）。

3.设置主轴方向

属性名： flex-direction

常用值如下：

1. row： 主轴方向水平从左到右 ——默认值
2. row-reverse： 主轴方向水平从右到左。
3. column： 主轴方向垂直从上到下。
4. column-reverse： 主轴方向垂直从下到上。



注意：改变了主轴的方向，侧轴方向也随之改变

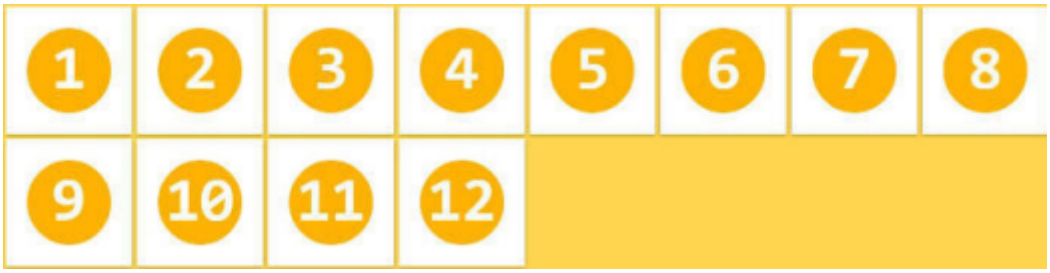
4.设置主轴换行方式

属性名： flex-wrap

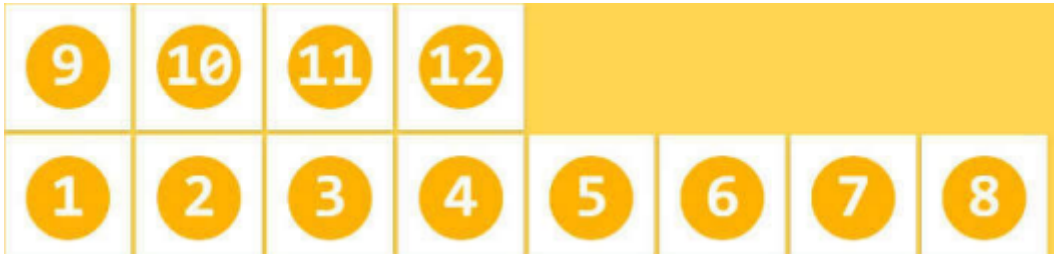
常用值如下：

1.nowrap： 默认值，不换行。

2.wrap： 自动换行，伸缩容器不够自动换行。



3.wrap-reverse： 反向换行



5.flex-flow

flex-flow 是一个复合属性，复合了 flex-direction 和 flex-wrap 两个属性。 值没有顺序要求。

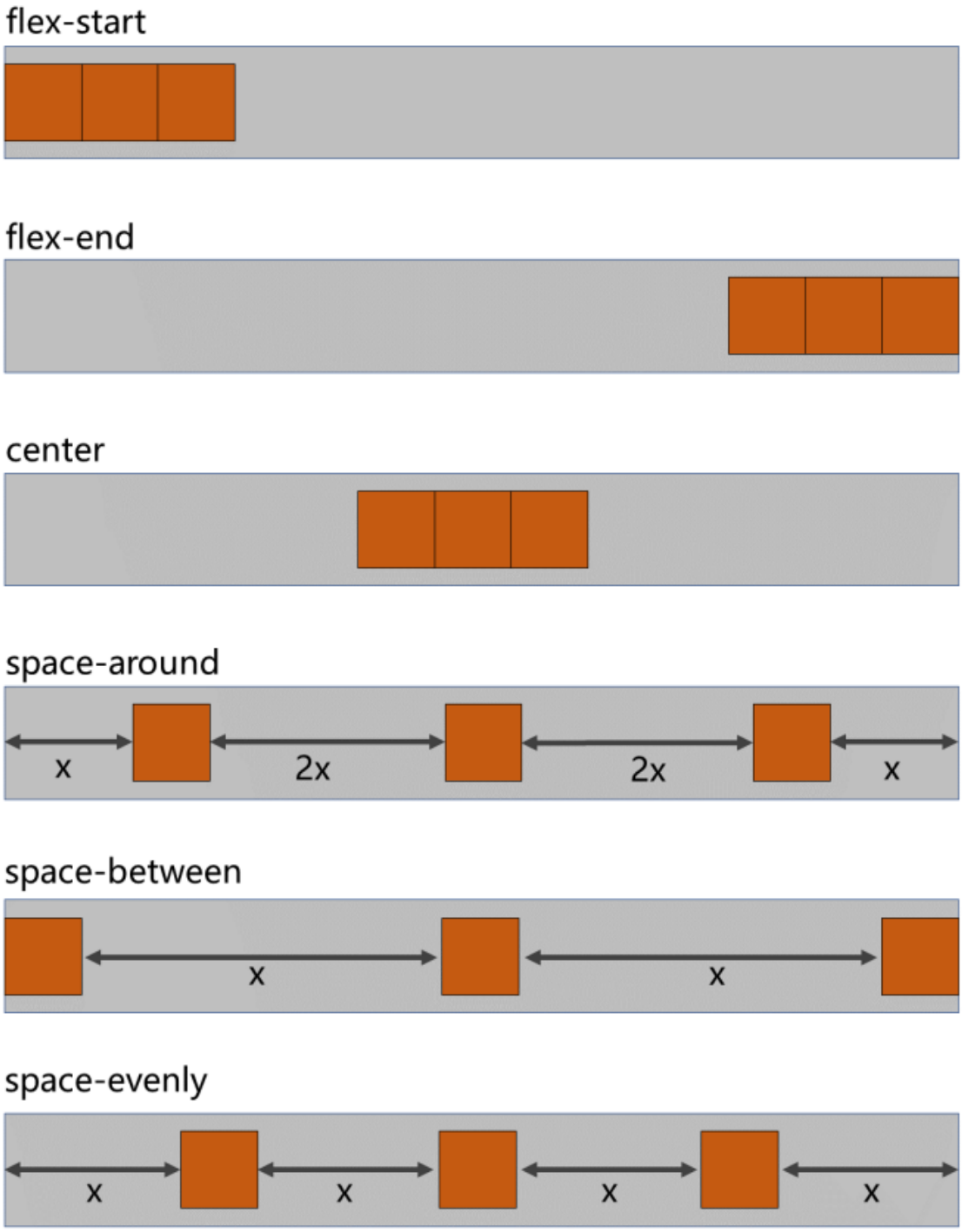
eg：`flex-flow: row wrap;`

6.设置主轴对齐方式

属性名： justify-content

常用值如下：

1. flex-start： 主轴起点对齐。—— 默认值
2. flex-end： 主轴终点对齐。
3. center： 居中对齐
4. space-between： 均匀分布，两端对齐（最常用）。
5. space-around： 均匀分布，两端距离是中间距离的一半。
6. space-evenly： 均匀分布，两端距离与中间距离一致。



7.设置侧轴对齐方式

7.1一行的情况下

所需属性： align-items

常用值如下：

1.flex-start： 侧轴的起点对齐。

2.flex-end： 侧轴的终点对齐。

3.center： 侧轴的中点对齐。

4.baseline : 伸缩项目的第一行文字的基线对齐。

5.stretch： 如果伸缩项目未设置高度，将占满整个容器的高度。—— （默认值）

flex-start

flex-end

center

baseline

stretch

7.2多行的情况下

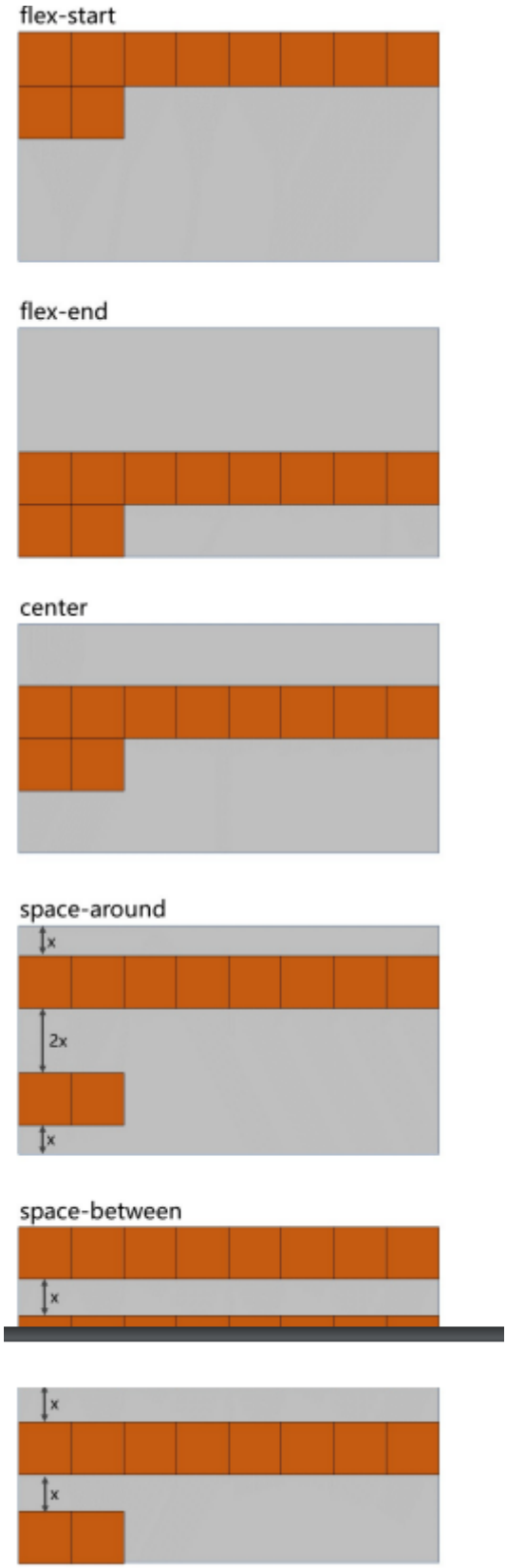
所需属性： align-content

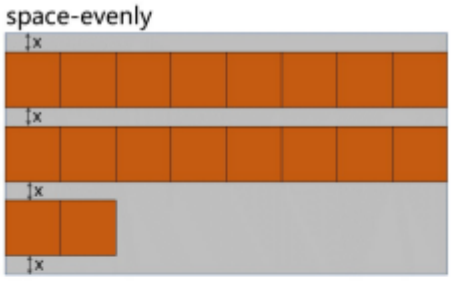
常用值如下：

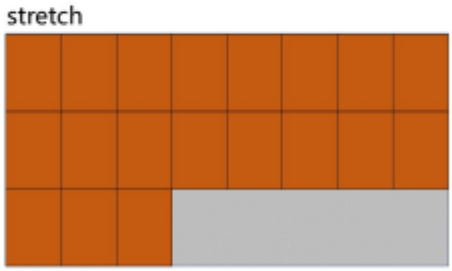
1. flex-start： 与侧轴的起点对齐。
2. flex-end： 与侧轴的终点对齐。
3. center： 与侧轴的中点对齐。
4. space-between： 与侧轴两端对齐，中间平均分布。
5. space-around： 伸缩项目间的距离相等，比距边缘大一

6. space-evenly : 在侧轴上完全平分。

7. stretch： 占满整个侧轴。—— 默认值







8.flex 实现水平垂直居中

方法一：父容器开启 flex 布局，随后使用 justify-content 和 align-items 实现水平垂直居中

```
.outer {
    width: 400px;
    height: 400px;
    background-color: #888;
    display: flex;
    justify-content: center;
    align-items: center;
}
.inner {
    width: 100px;
    height: 100px;
    background-color: orange;
}
```

方法二：父容器开启 flex 布局，随后子元素 margin: auto

```
.outer {
    width: 400px;
    height: 400px;
    background-color: #888;
    display: flex;
}
.inner {
    width: 100px;
    height: 100px;
    background-color: orange;
    margin: auto;
}
```

9.平均分布

- **flex**

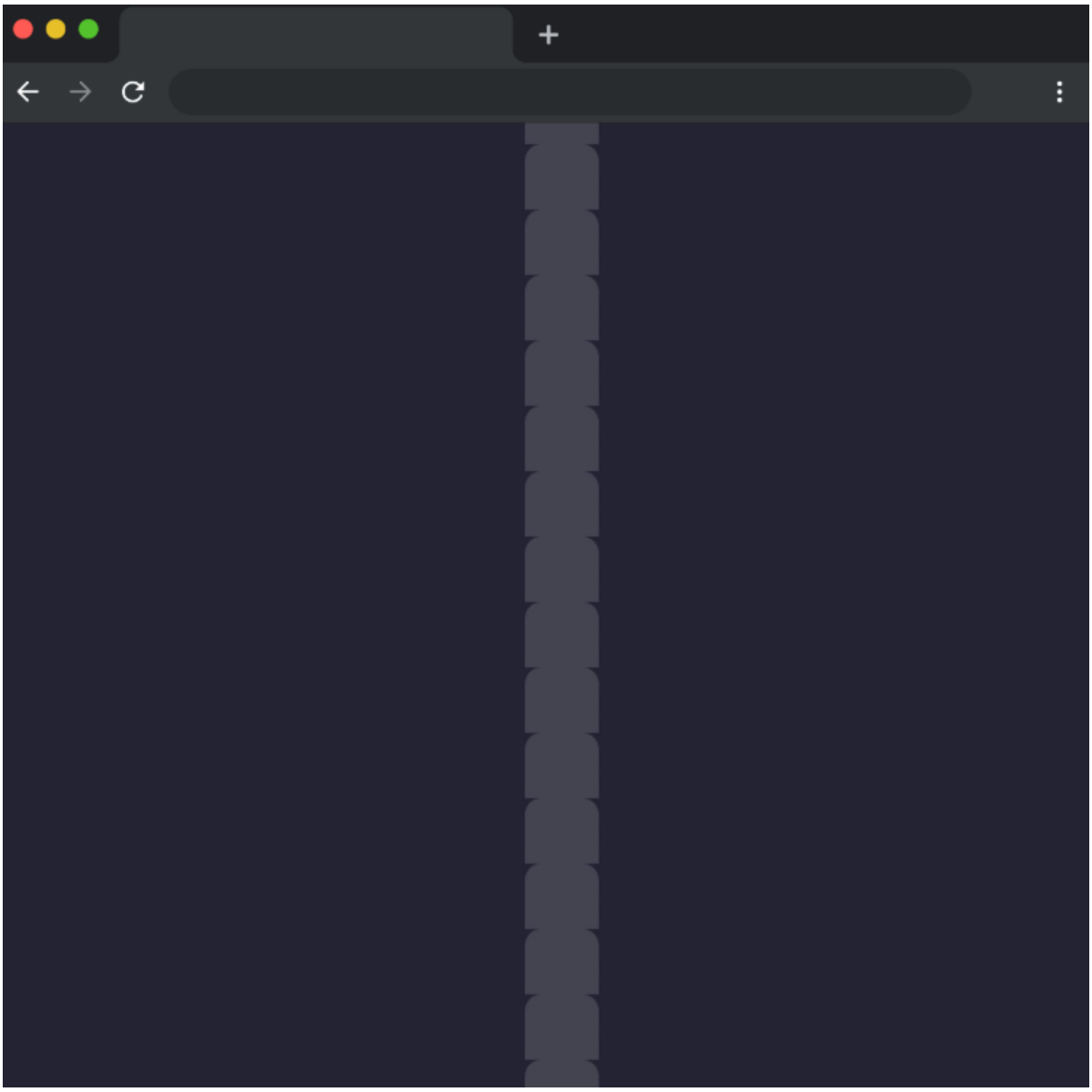
  如果元素在flex容器中，可以设定其`flex`值，最终容器中的元素所占大小即元素之间的`flex`值比例。

  若只有容器中一个元素，则占满；若只有容器中只有一个元素设定了`flex`，则该元素占满剩余空间。

```
<style>
    .layout-flex{
        display: flex;
        height: 100px;
    }
    .flex-1{
        flex: 1;
        background-color: red;
    }
    .flex-2{
        flex: 2;
        background-color: green;
    }
</style>
<body>
    <div class="layout-flex">
        <div class="flex-1"></div>
        <div class="flex-2"></div>
    </div>
</body>
```

10.例题

10.1 电影院排座位

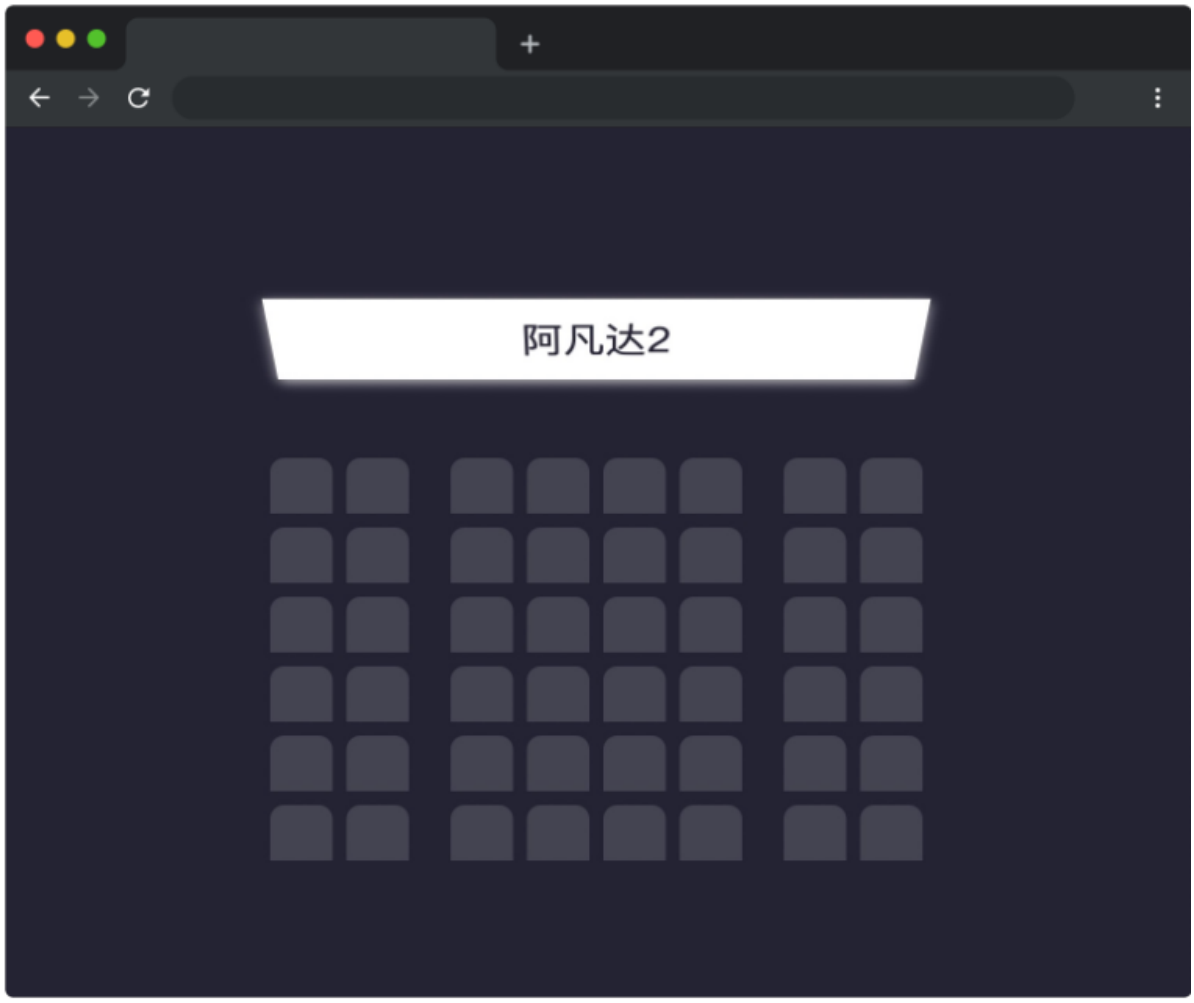


要求：

请在 `css/style.css` 文件中的 TODO 下补全样式代码，最终达到预期布局效果，需要注意：

- 座位区域和荧幕间隔 50px。
- 座位区域每一行包含 8 个座位。
- 第 2 列和第 6 列旁边都是走廊，需要和下一列间隔 30px，其他列都只需要间隔 10px。

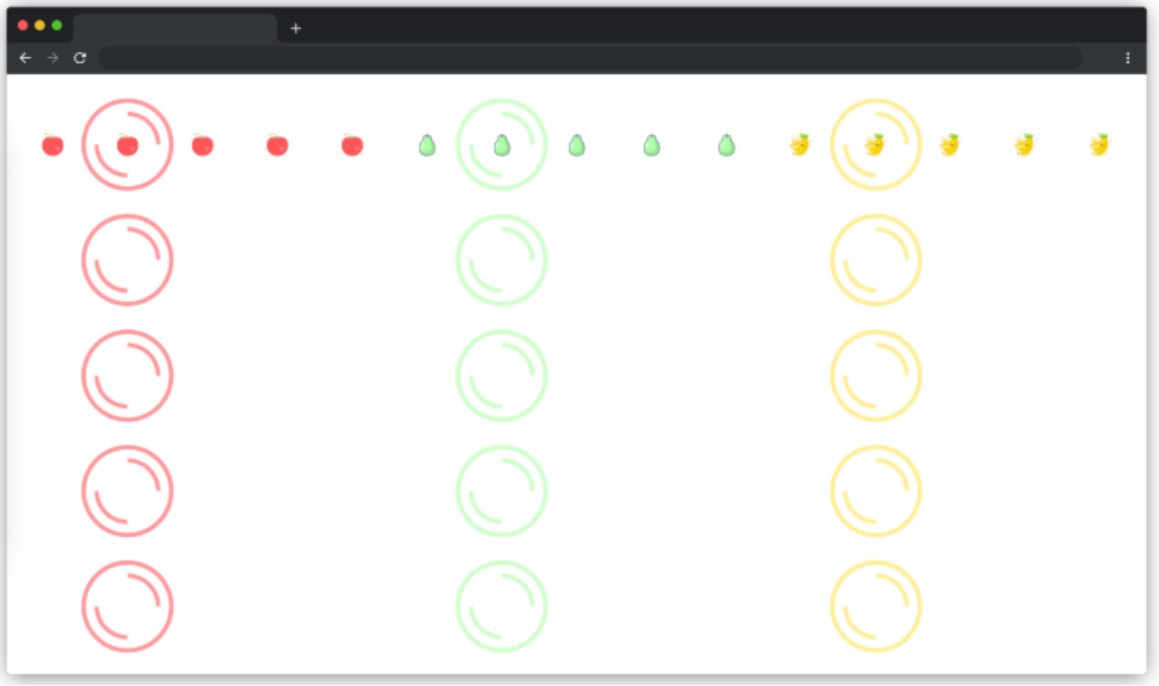
完成后的效果如下：




```
.seat-area{
  margin-top:50px;
  display: flex;
  flex-wrap: wrap; //自动换行
  gap:10px;
}
.seat:nth-children(8n+2){
  margin-right:20px
}
.seat:nth-children(8n+6){
  margin-right:20px
}
```

10.2水果拼盘

在浏览器中预览 `index.html` 页面效果如下：

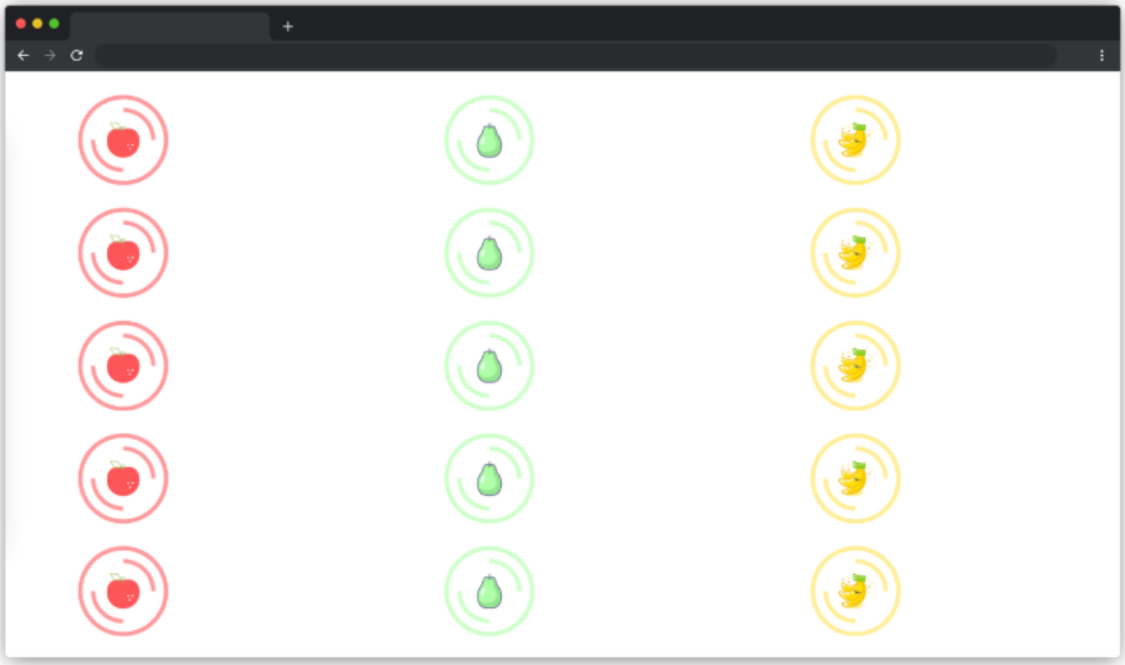


要求：

建议使用 `flex` 相关属性完成 `css/style.css` 中的 TODO 部分。

1. 禁止修改圆盘的位置和图片的大小。
2. 相同颜色的水果放在相同颜色的圆盘正中间（例如：苹果是红色的就放在红色的圆盘里）。

完成后，页面效果如下：



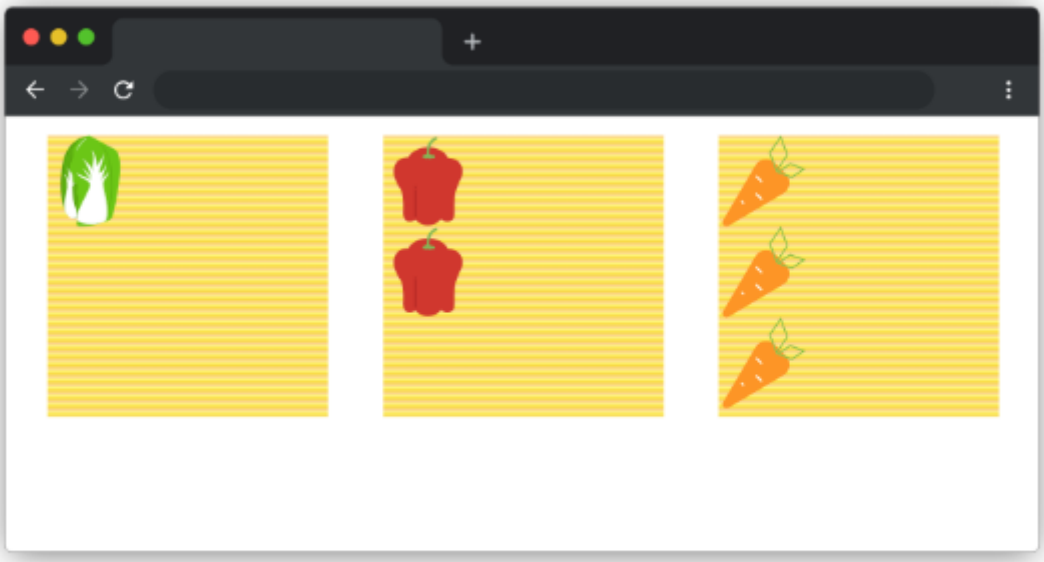
```
#pond{
display:flex;
flex-direction:column;//将主轴方向改为从上到下
flex-wrap：wrap;//设置自动换行
}

//第二种方法：
flex-flow：column wrap；
flex-flow属性相当于flex-direction与flex-warp属性综合使用

```

10.3新鲜蔬菜

在浏览器打开后，效果如下：



目标

完成 `css/style.css` 中的 TODO 部分。所有元素的大小都已给出，无需修改，完成后效果如下（图中灰色线条为布局参考线无需实现）：

```
.box {
  display: flex;
```

```
}
#box1 {
  flex-direction: row;
  align-items: center;
  justify-content: center;
}
#box2 {
  justify-content: space-between;
}
#box2 .item:nth-child(2) {
  align-self: end;
}
#box3 {
  justify-content: space-between;
}
#box3 .item:nth-child(2) {
  align-self: center;
}
#box3 .item:nth-child(3) {
  align-self: end;
}
```

grid布局

1.grid布局与flex布局

Grid 布局与 Flex 布局有一定的相似性，都可以指定容器内部多个项目的位置。但是，它们也存在重大区别。

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成"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。Grid 布局远比 Flex 布局强大。

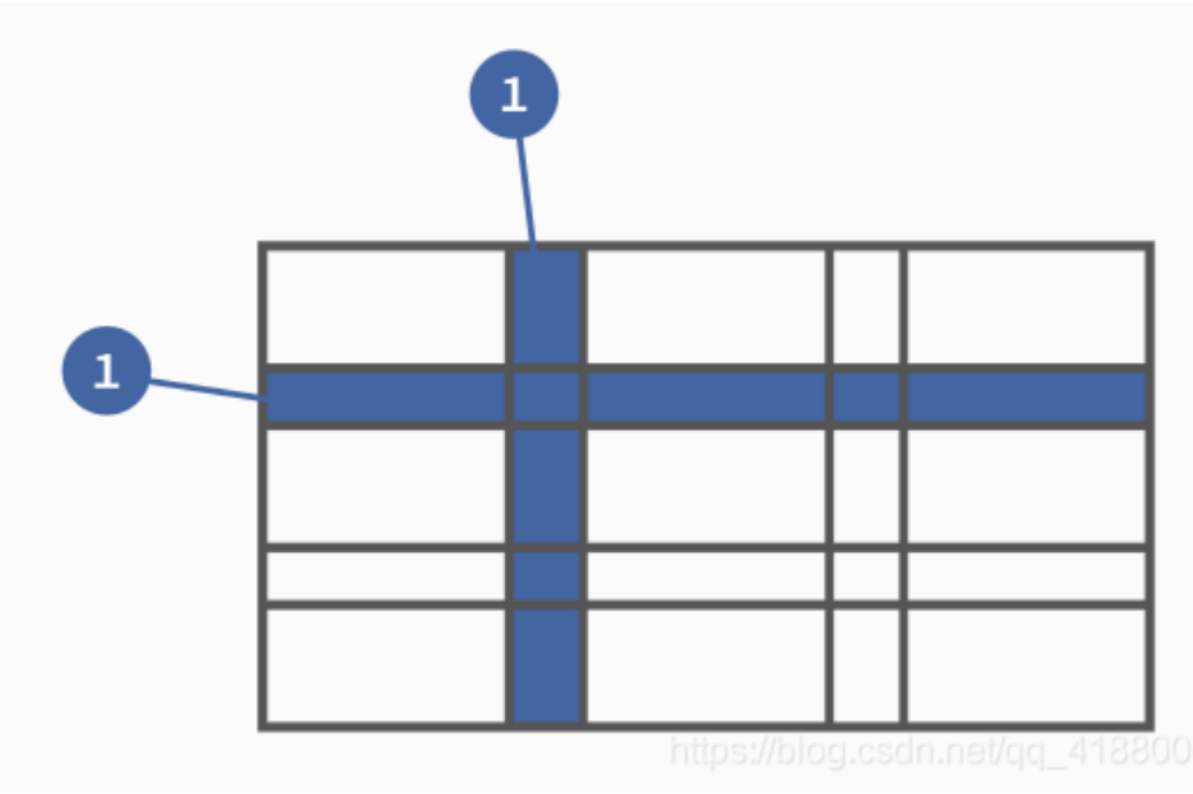
2.相关概念

2.1 容器和项目

采用网格布局的区域，称为"容器"（container）。容器内部采用网格定位的子元素，称为"项目"（item）

```
<div><p>1</p></div>
 <div><p>2</p></div>
 <div><p>3</p></div>
```

2.2 行和列



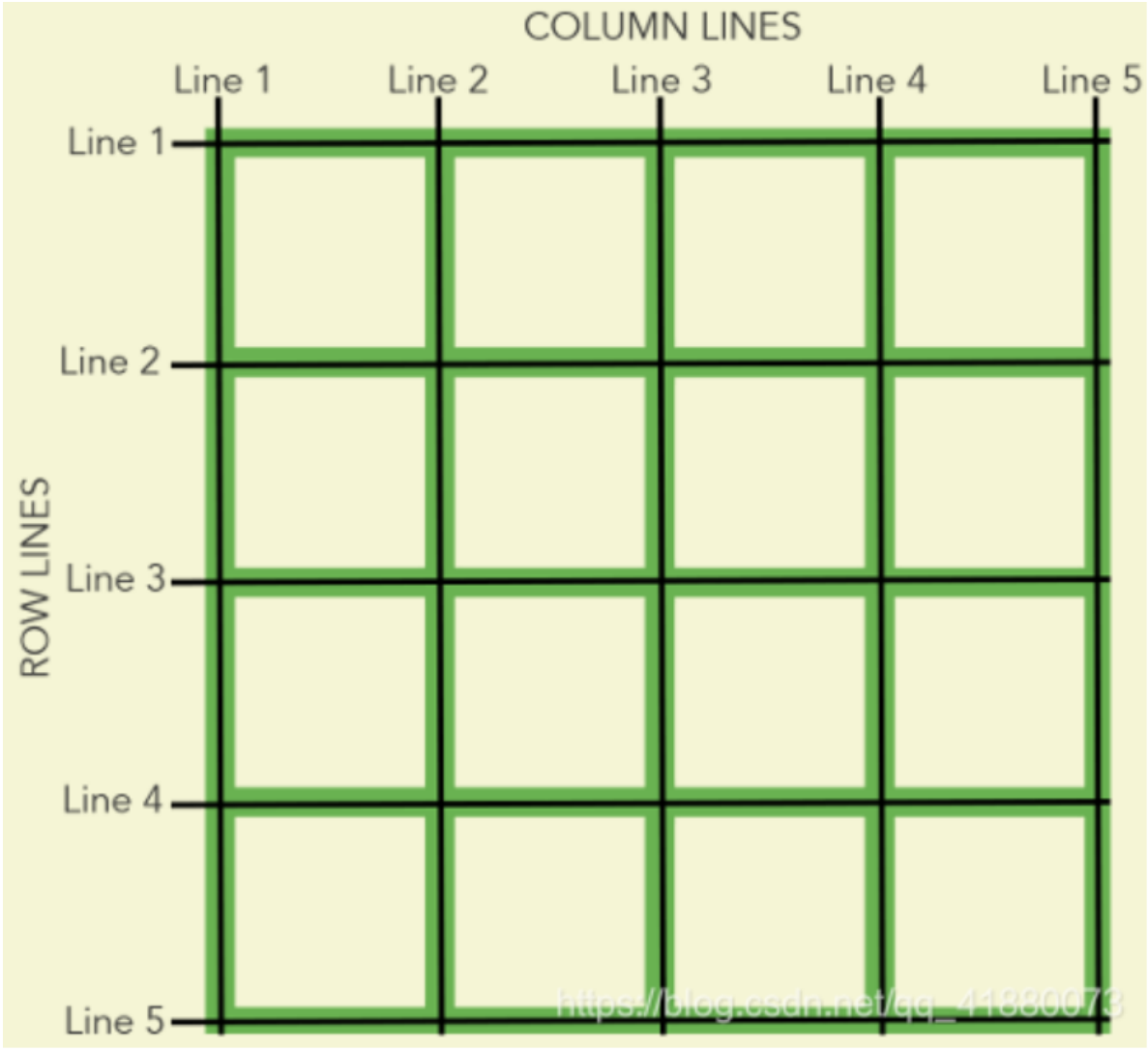
上图中，水平的深色区域就是"行"，垂直的深色区域就是"列"。

2.3单元格

行和列的交叉区域，称为"单元格"（cell）。

正常情况下，n行和m列会产生n x m个单元格。比如，3行3列会产生9个单元格。

2.4网格线



上图是一个 4 x 4 的网格，共有5根水平网格线和5根垂直网格线。

3.开启grid

display: grid指定一个容器采用网格布局。

```
div {

 display: grid;

}
```

列子

```
<!DOCTYPE html>
<html>
```

```
    <head>
        <meta charset="utf-8">
        <title></title>
        <style type="text/css">
            span {
                font-size: 2em;
            }
        #container {
            display: grid;
            grid-template-columns: 200px 200px 200px;
            grid-template-rows: 200px 200px 200px;
        }

        .item {
            font-size: 2em;
            text-align: center;
            border: 1px solid #e5e4e9;
        }

        .item-1 {
            background-color: #ef342a;
        }

        .item-2 {
            background-color: #f68f26;
        }

        .item-3 {
            background-color: #4ba946;
        }

        .item-4 {
            background-color: #0376c2;
        }

        .item-5 {
            background-color: #c077af;
        }

        .item-6 {
            background-color: #f8d29d;
        }

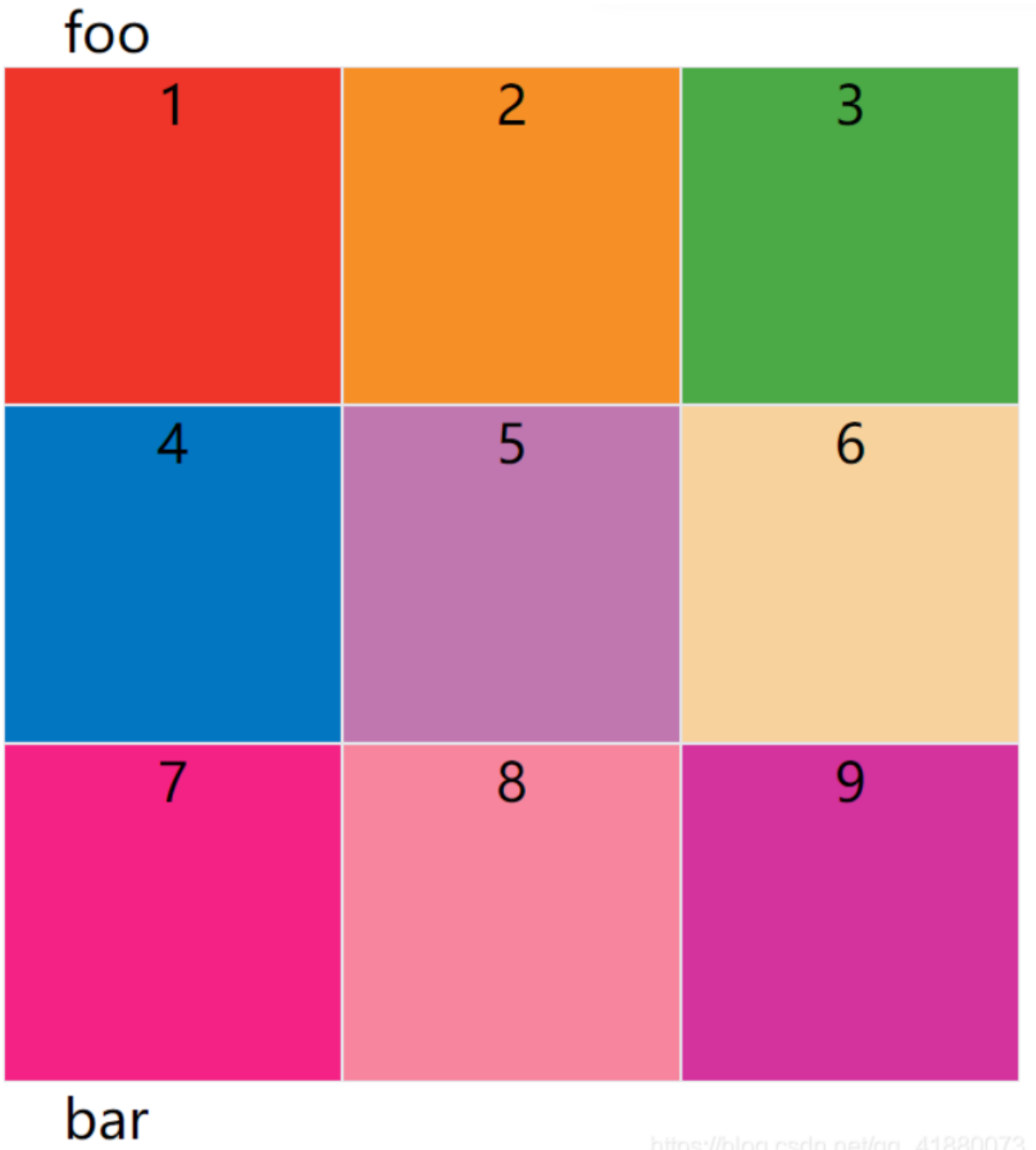
        .item-7 {
            background-color: #f52285;
        }

        .item-8 {
            background-color: #f7859d;
        }

        .item-9 {
            background-color: #d4329d;
        }
    </style>
```

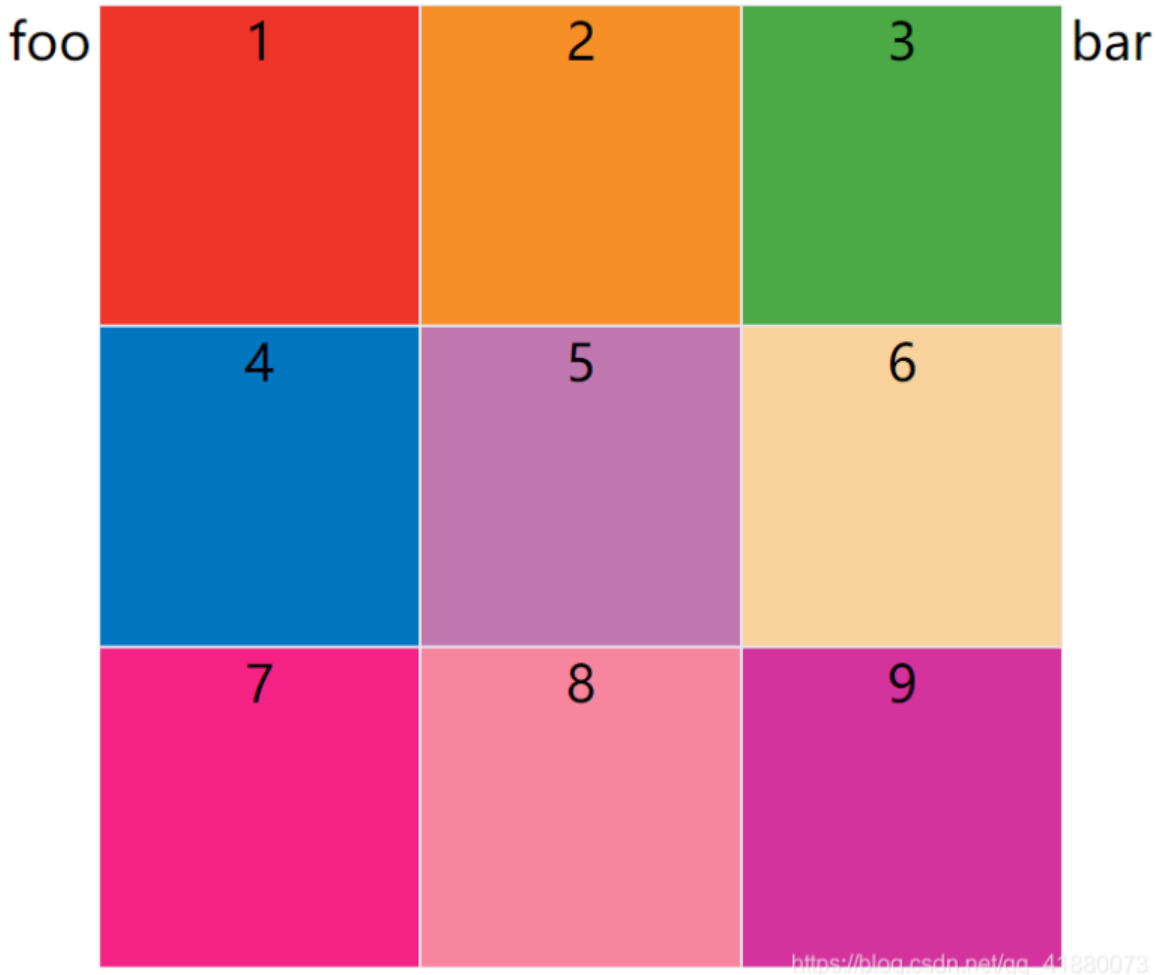
```
</head>
<body>
    <span>foo</span>
    <div id="container">
        <div class="item item-1">1</div>
        <div class="item item-2">2</div>
        <div class="item item-3">3</div>
        <div class="item item-4">4</div>
        <div class="item item-5">5</div>
        <div class="item item-6">6</div>
        <div class="item item-7">7</div>
        <div class="item item-8">8</div>
        <div class="item item-9">9</div>
    </div>
    <span>bar</span>
</body>

</html>`
```



默认情况下，容器元素都是块级元素，但也可以设成行内元素

```
div {
  display: inline-grid;
}
```



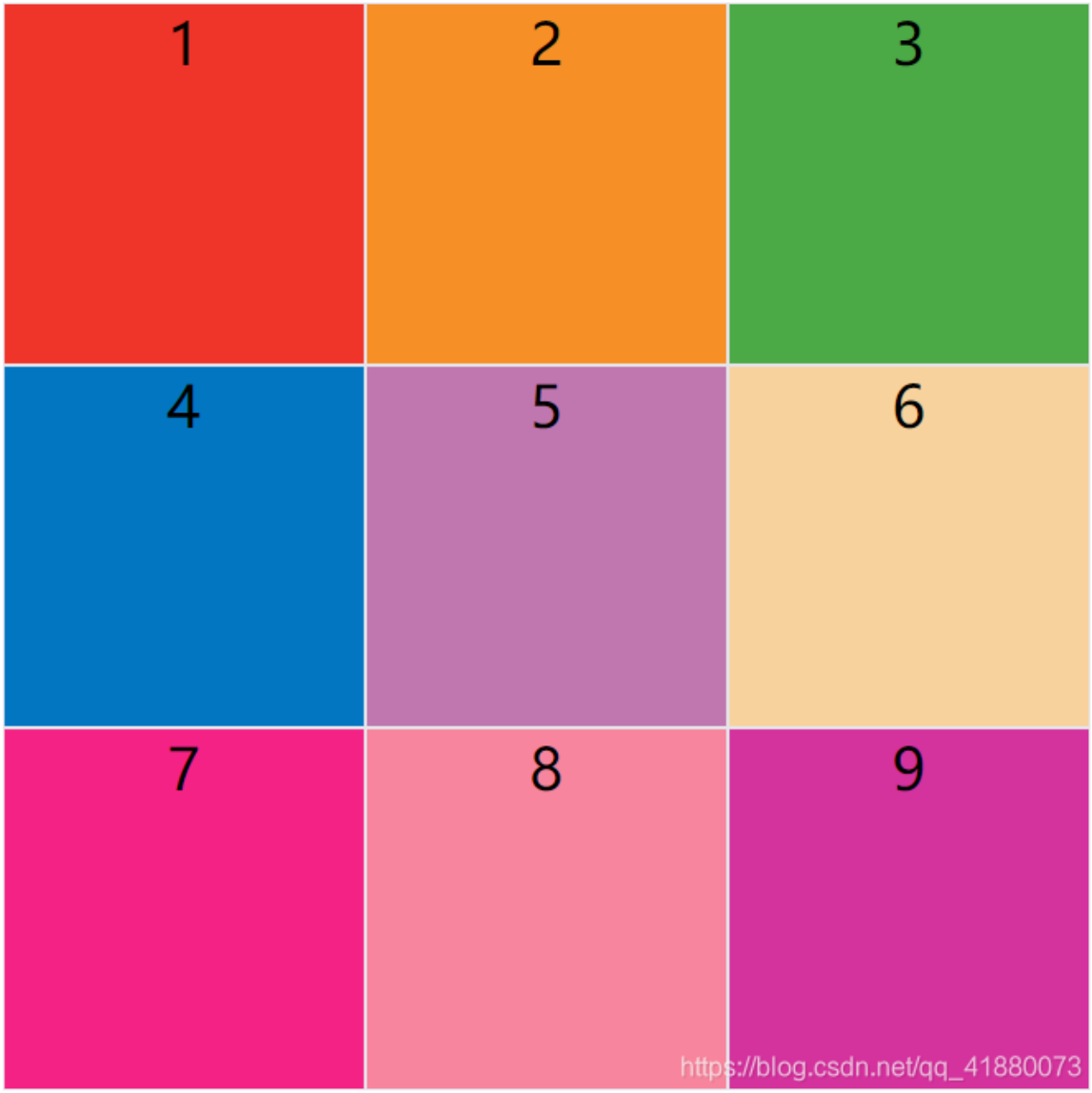
4.grid的属性

4.1grid-template-columns 属性，grid-template-rows 属性

grid-template-columns属性定义每一列的列宽，grid-template-rows属性定义每一行的行高。

```
.container {
  display: grid;
  grid-template-columns: 100px 100px 100px;
  grid-template-rows: 100px 100px 100px;
}
```

上面代码指定了一个三行三列的网格，列宽和行高都是100px。



除了使用绝对单位，也可以使用百分比。

```
#container {
    display: grid;
    width: 300px;
    height: 300px;
    grid-template-columns: 33.3% 33.3% 33.3%;
    grid-template-rows: 33.3% 33.3% 33.3%;
}

```

（1）repeat()

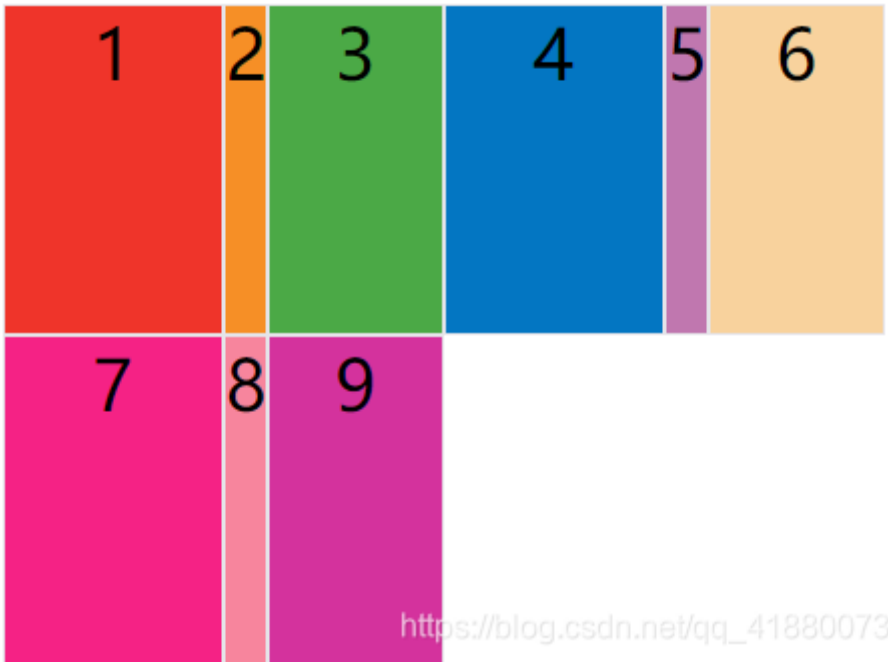
有时候，重复写同样的值非常麻烦，尤其网格很多时。这时，可以使用repeat()函数，简化重复的值。上面的代码用repeat()改写如下。

```
#container {
    display: grid;
    width: 300px;
    height: 300px;
    grid-template-columns: repeat(3, 33.33%);
    grid-template-rows: repeat(3, 33.33%);
}

```

repeat()接受两个参数，第一个参数是重复的次数（上例是3），第二个参数是所要重复的值。

repeat()重复某种模式也是可以的。



（2）auto-fill 关键字

有时，单元格的大小是固定的，但是容器的大小不确定。如果希望每一行（或每一列）容纳尽可能多的单元格，这时可以使用auto-fill关键字表示自动填充。

```
#container {
    display: grid;
    width: 300px;
    height: 300px;
    grid-template-columns: repeat(auto-fill,50px);
}

```

(3) fr 关键字

为了方便表示比例关系，网格布局提供了fr关键字（fraction 的缩写，意为"片段"）。如果两列的宽度分别为1fr和2fr，就表示后者是前者的两倍。

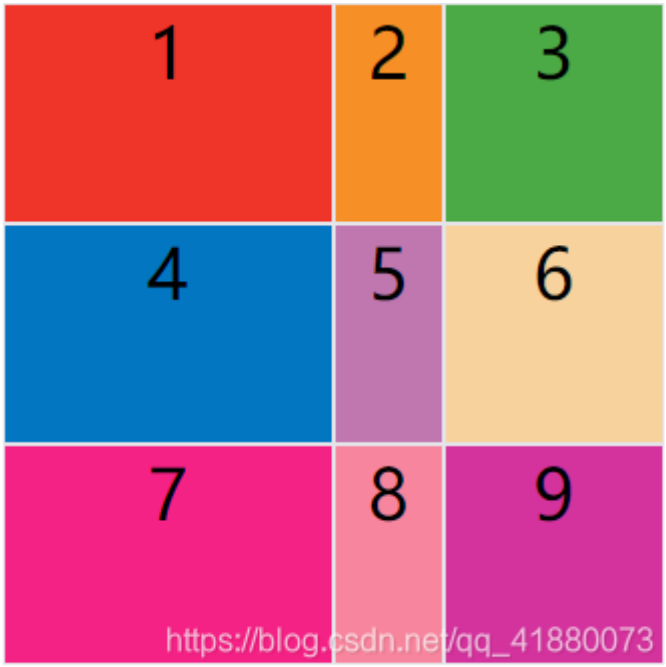
```
#container {
    display: grid;
    width: 300px;
    height: 300px;
    grid-template-columns: 1fr 2fr;
}

```

上面代码表示第二个的宽度是第一个宽度的二倍。
fr可以与绝对长度的单位结合使用，这时会非常方便。

```
#container {
    display: grid;
    width: 300px;
    height: 300px;
    grid-template-columns: 150px 1fr 2fr;
}

```



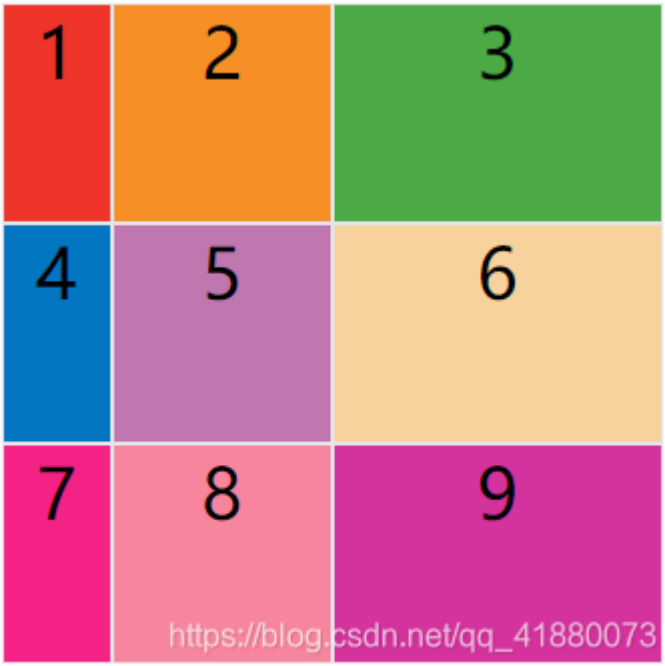
上面代码表示，第一列的宽度为150像素，第二列的宽度是第三列的一半。

(4) minmax()

minmax()函数产生一个长度范围，表示长度就在这个范围之中。它接受两个参数，分别为最小值和最大值。

```
#container {
    display: grid;
    width: 300px;
    height: 300px;
    grid-template-columns: 1fr 2fr minmax(150px , 1fr);
}

```



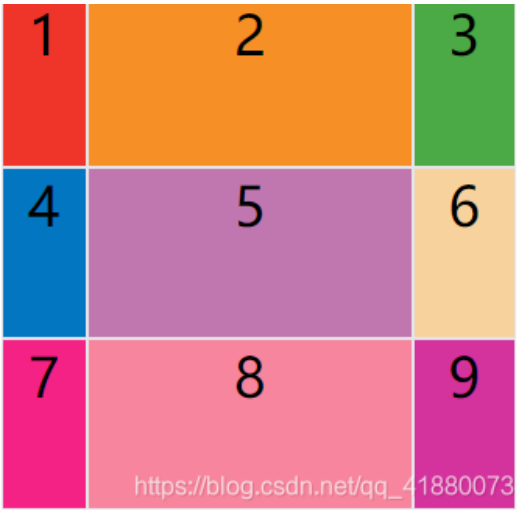
上面代码表示，第二列是第一列的二倍，第三列的宽度最小150px，最大不超过1fr

（5）auto 关键字

auto关键字表示由浏览器自己决定长度。

```
#container {
    display: grid;
    width: 300px;
    height: 300px;
    grid-template-columns: 50px auto 60px;
}

```



上面代码中，第二列的宽度，基本上等于该列单元格的最大宽度，除非单元格内容设置了min-width，且这个值大于最大宽度。

（6）布局实例

grid-template-columns属性对于网页布局非常有用。两栏式布局只需要一行代码

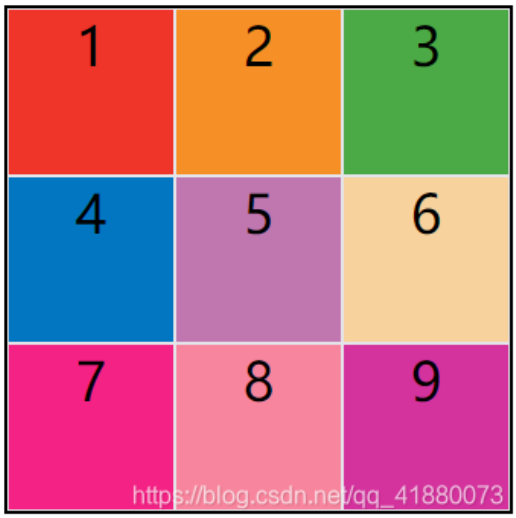
```
#container {
    display: grid;
    width: 300px;
    height: 300px;
    grid-template-columns: 70% 30%;
}

```

上面代码将左边栏设为70%，右边栏设为30%。

5.grid-auto-flow 属性

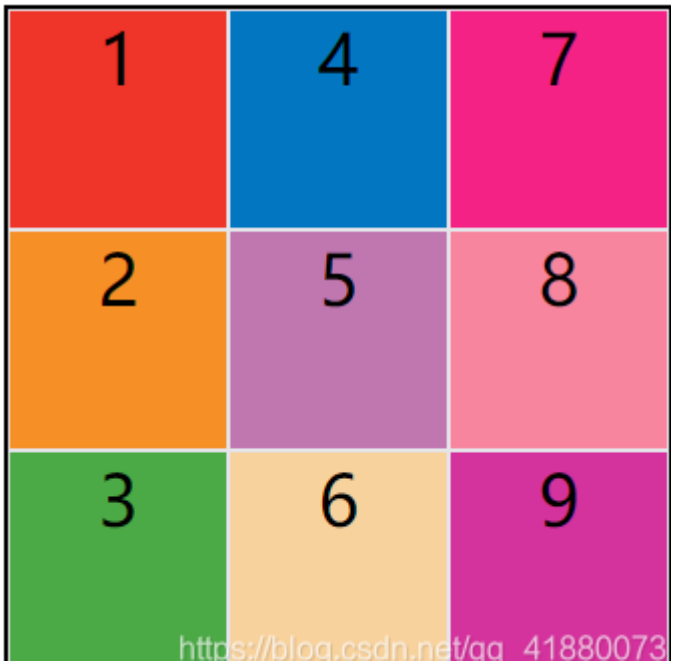
划分网格以后，容器的子元素会按照顺序，自动放置在每一个网格。默认的放置顺序是"先行后列"，即先填满第一行，再开始放入第二行，即下图数字的顺序。



这个顺序由grid-auto-flow属性决定，默认值是row，即"先行后列"。也可以将它设成column，变成"先列后行"。

```
#container {
    display: grid;
    width: 300px;
    height: 300px;
    grid-template-columns: 1fr 1fr 1fr;
    grid-template-rows: 1fr 1fr 1fr;
    grid-auto-flow: column;
    border: 2px solid black;
}

```



grid-auto-flow属性除了设置成row和column，还可以设成row dense和column dense。这两个值主要用于，某些项目指定位置以后，剩下的项目怎么自动放置。

下面的例子让1号项目和2号项目各占据两个单元格，然后在默认的grid-auto-flow: row情况下，会产生下面这样的布局。

```
            #container {
                display: grid;
                width: 300px;
                height: 400px;
                grid-template-columns: repeat(3,100px);
                grid-template-rows: repeat(3,100px);
                grid-auto-flow: row;
                border: 2px solid black;
            }

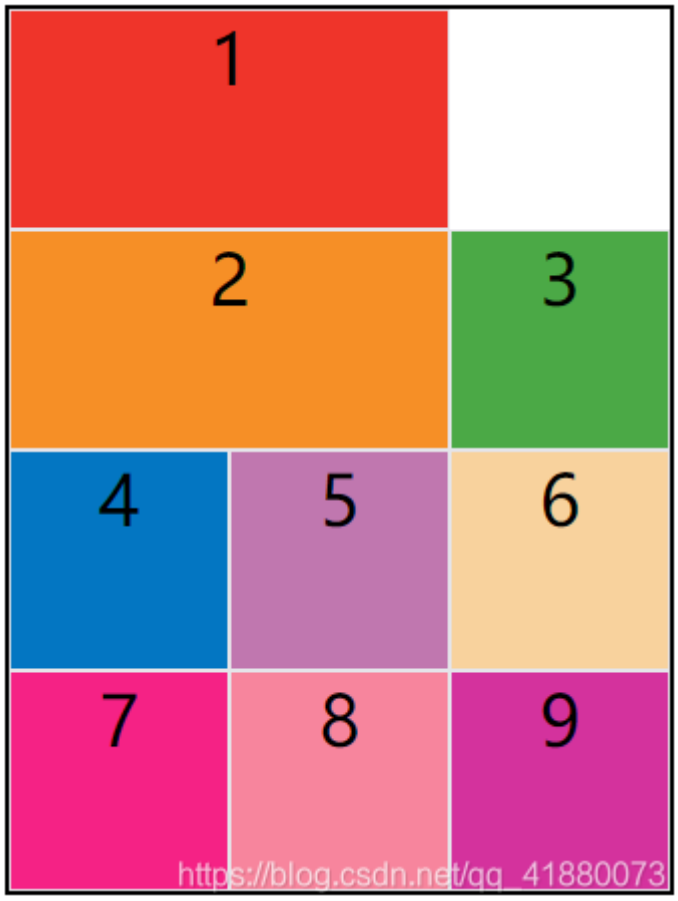
            .item {
                font-size: 2em;
                text-align: center;
```

```
                border: 1px solid #e5e4e9;
            }

            .item-1 {
                background-color: #ef342a;
                grid-column-start: 1;
                grid-column-end: 3;
            }

            .item-2 {
                background-color: #f68f26;
                grid-column-start: 1;
                grid-column-end: 3;
            }

```

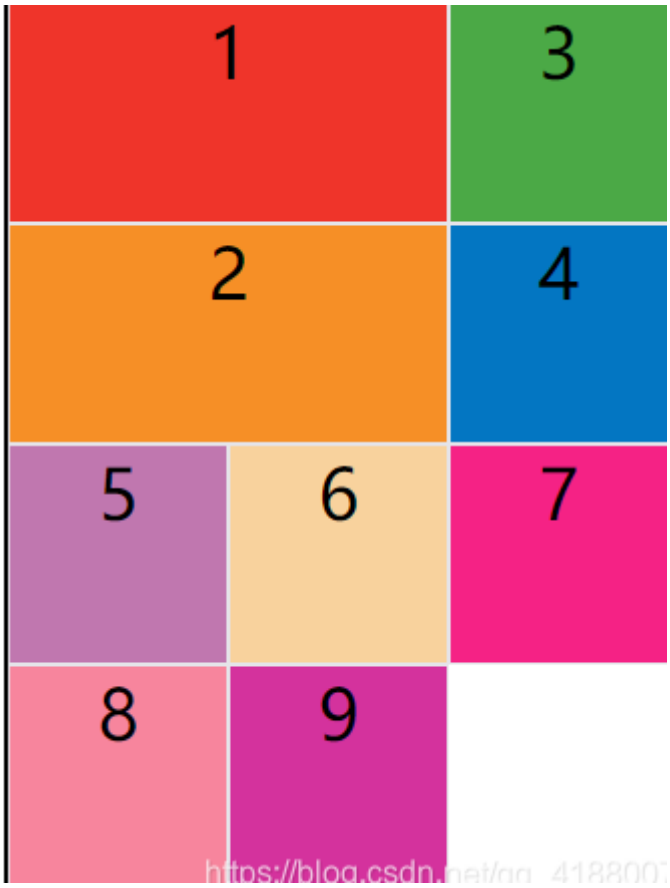


上图中，1号项目后面的位置是空的，这是因为3号项目默认跟着2号项目，所以会排在2号项目后面。

现在修改设置，设为row dense，表示"先行后列"，并且尽可能紧密填满，尽量不出现空格。

```
grid-auto-flow: row dense;

```

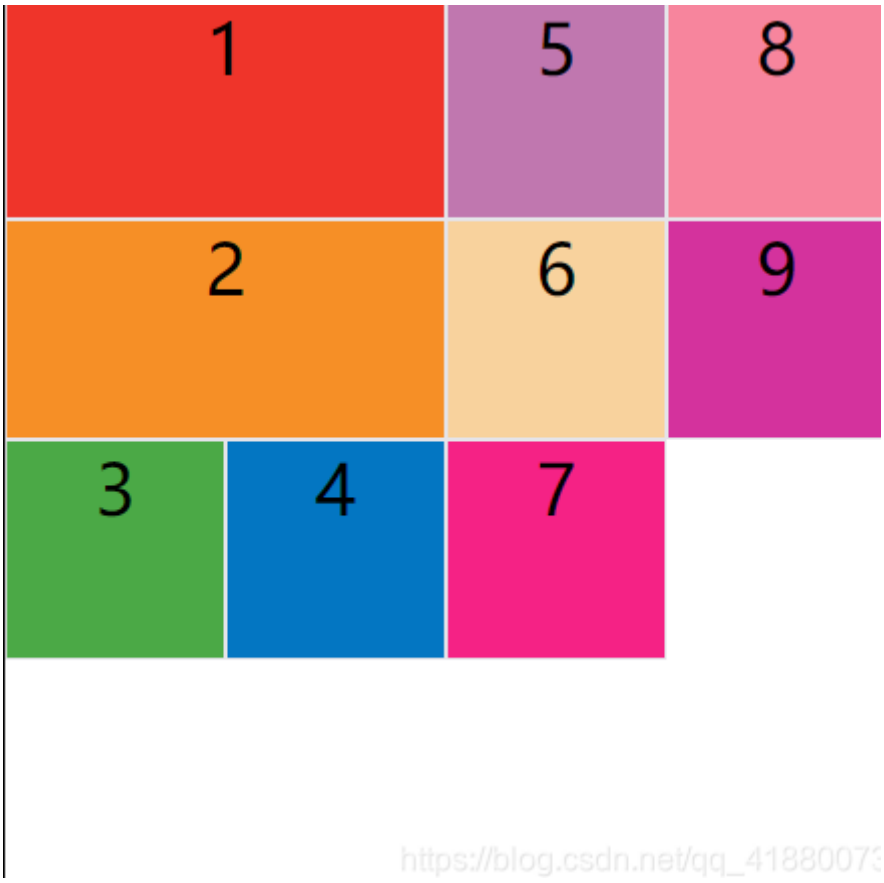


上图会先填满第一行，再填满第二行，所以3号项目就会紧跟在1号项目的后面。8号项目和9号项目就会排到第四行。

如果将设置改为column dense，表示"先列后行"，并且尽量填满空格。

```
grid-auto-flow: column dense;

```



上图会先填满第一列，再填满第2列，所以3号项目在第一列，4号项目在第二列。8号项目和9号项目被挤到了第四列。

6.justify-items 属性，align-items 属性，place-items 属性

justify-items属性设置单元格内容的水平位置（左中右），align-items属性设置单元格内容的垂直位置（上中下）。

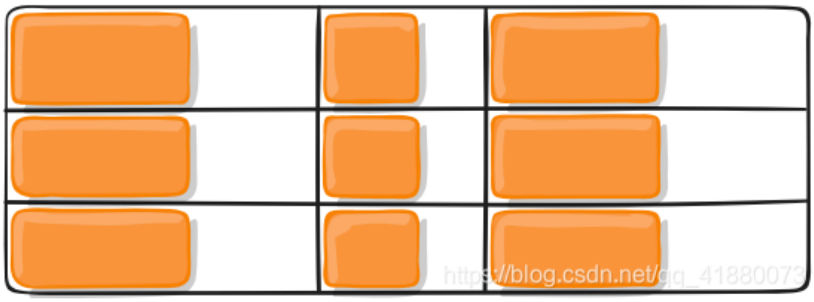
```
.container {
  justify-items: start | end | center | stretch;
  align-items: start | end | center | stretch;
}

```

这两个属性的写法完全相同，都可以取下面这些值。

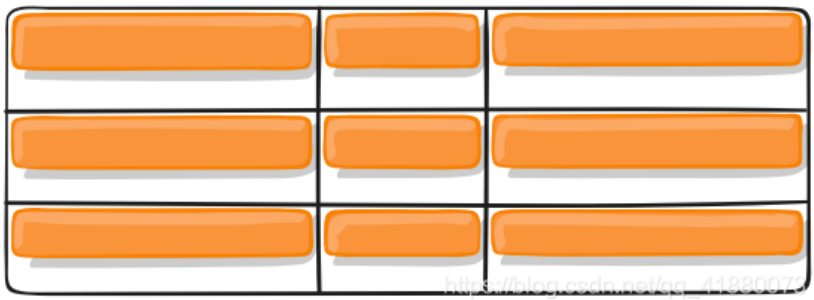
start：对齐单元格的起始边缘。
end：对齐单元格的结束边缘。
center：单元格内部居中。
stretch：拉伸，占满单元格的整个宽度（默认值）。

```
.container {
  justify-items: start;
}
```
复制



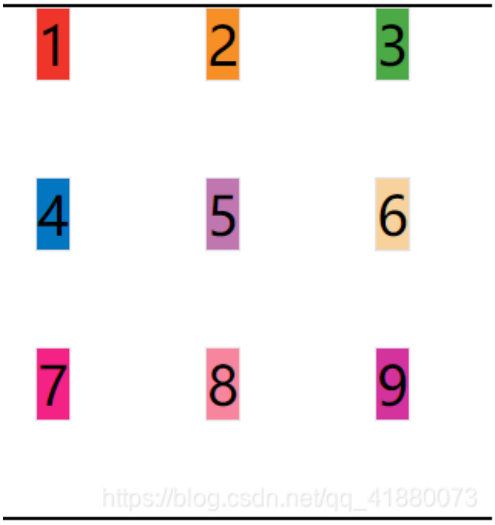

```
.container {
  align-items: start;
}
```
复制

上面代码表示，单元格的内容头部对齐，效果如下图。



place-items属性是justify-items属性和align-items属性的合并简写形式。

```
place-items: start end;
```

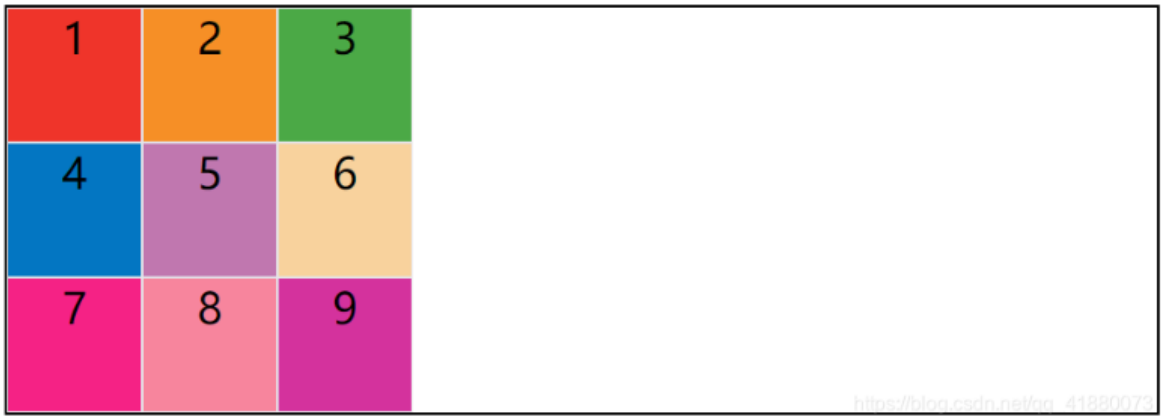


如果省略第二个值，则浏览器认为与第一个值相等。

7、justify-content 属性，align-content 属性，place-content 属性

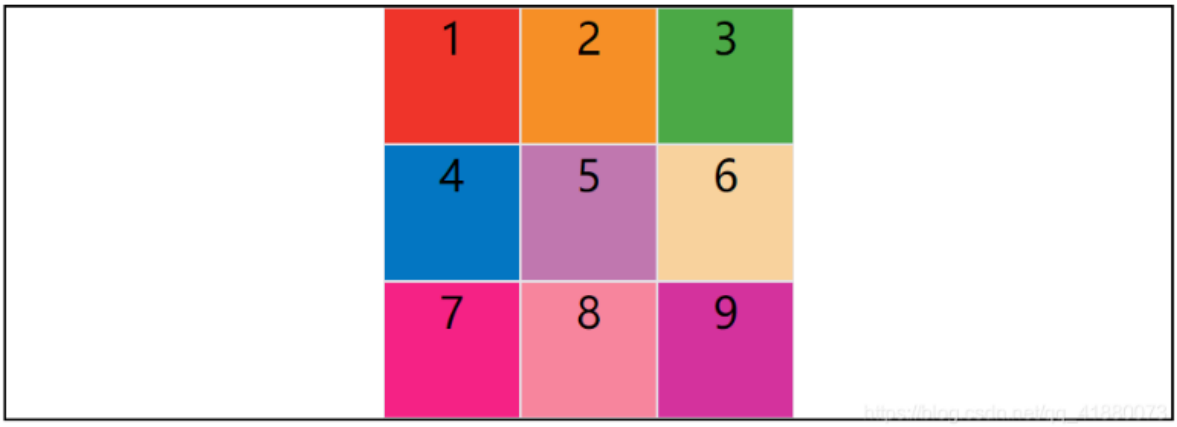
justify-content属性是整个内容区域在容器里面的水平位置（左中右），align-content属性是整个内容区域的垂直位置（上中下）。

start - 对齐容器的起始边框。



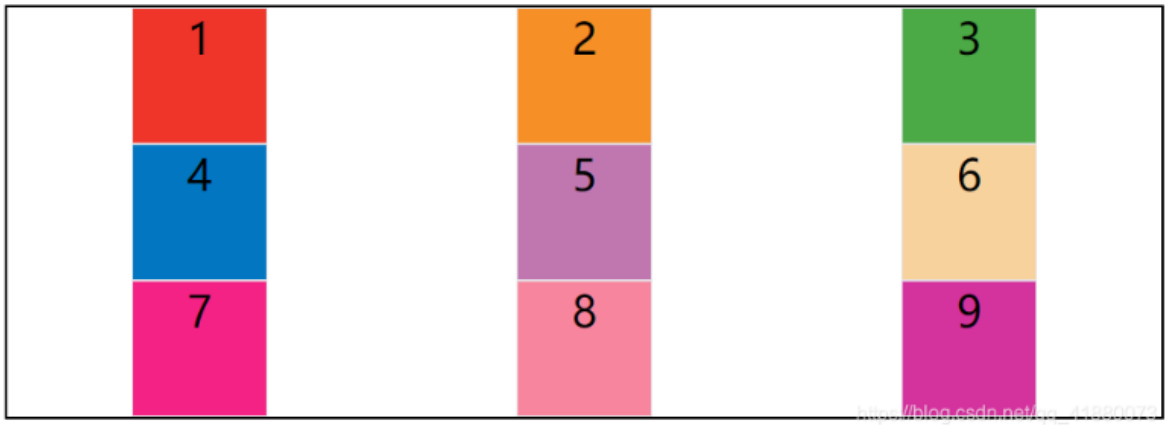
end - 对齐容器的结束边框。

center - 容器内部居中。

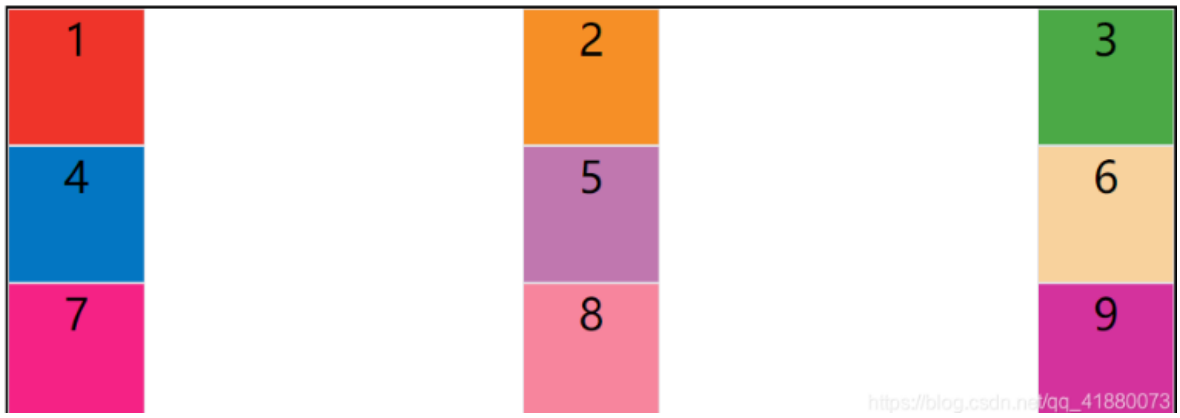


stretch - 项目大小没有指定时，拉伸占据整个网格容器。

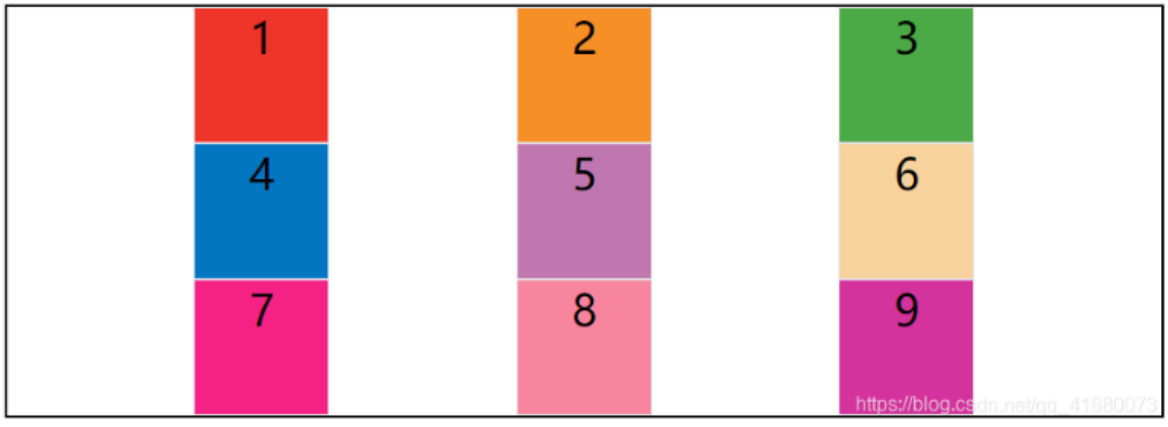
space-around - 每个项目两侧的间隔相等。所以，项目之间的间隔比项目与容器边框的间隔大一倍。



space-between - 项目与项目的间隔相等，项目与容器边框之间没有间隔。



space-evenly - 项目与项目的间隔相等，项目与容器边框之间也是同样长度的间隔。



place-content属性是justify-content属性和align-content属性的合并简写形式。

```
place-content: space-around space-evenly;
```

8.grid-auto-columns 属性，grid-auto-rows 属性

有时候，一些项目的指定位置，在现有网格的外部。比如网格只有3列，但是某一个项目指定在第5行。这时，浏览器会自动生成多余的网格，以便放置项目。

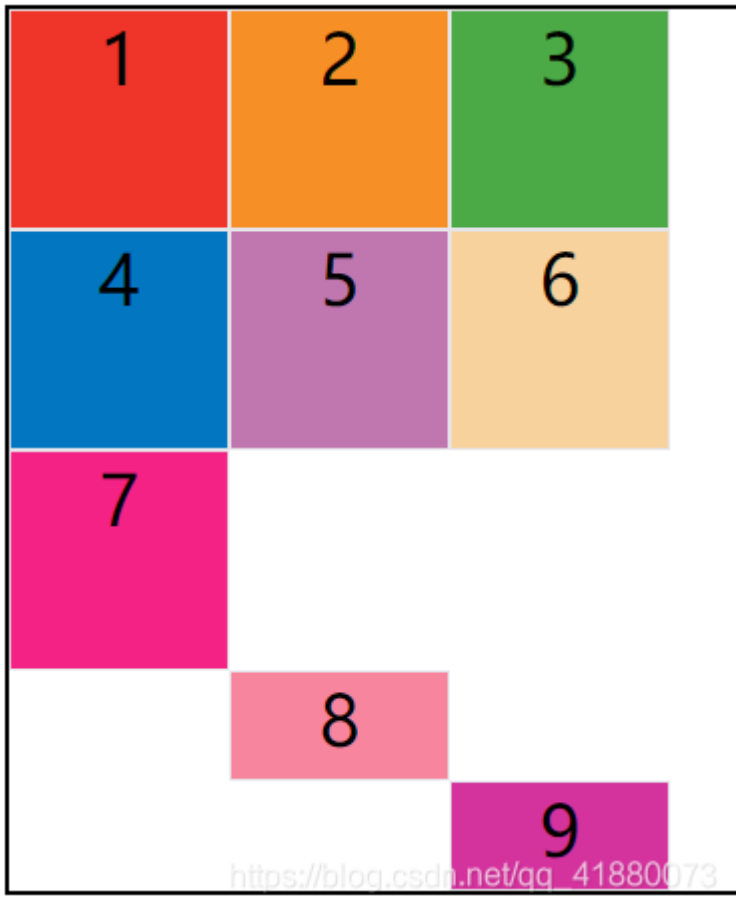
grid-auto-columns属性和grid-auto-rows属性用来设置，浏览器自动创建的多余网格的列宽和行高。它们的写法与grid-template-columns和grid-template-rows完全相同。如果不指定这两个属性，浏览器完全根据单元格内容的大小，决定新增网格的列宽和行高。

下面的例子里面，划分好的网格是3行 x 3列，但是，8号项目指定在第4行，9号项目指定在第5行。

```
            #container {
                display: grid;
                grid-template-columns: repeat(3,100px);
                grid-template-rows: repeat(3,100px);
                border: 2px solid black;
                grid-auto-rows: 50px;
            }
            .item-8 {
                background-color: #f7859d;
                grid-row-start: 4;
                  grid-column-start: 2;
            }

            .item-9 {
                background-color: #d4329d;
                 grid-row-start: 5;
                  grid-column-start: 3;
            }

```



9、grid-template 属性，grid 属性

grid-template属性是grid-template-columns、grid-template-rows和grid-template-areas这三个属性的合并简写形式。

grid属性是grid-template-rows、grid-template-columns、grid-template-areas、 grid-auto-rows、grid-auto-columns、grid-auto-flow这六个属性的合并简写形式。 (不建议合并)

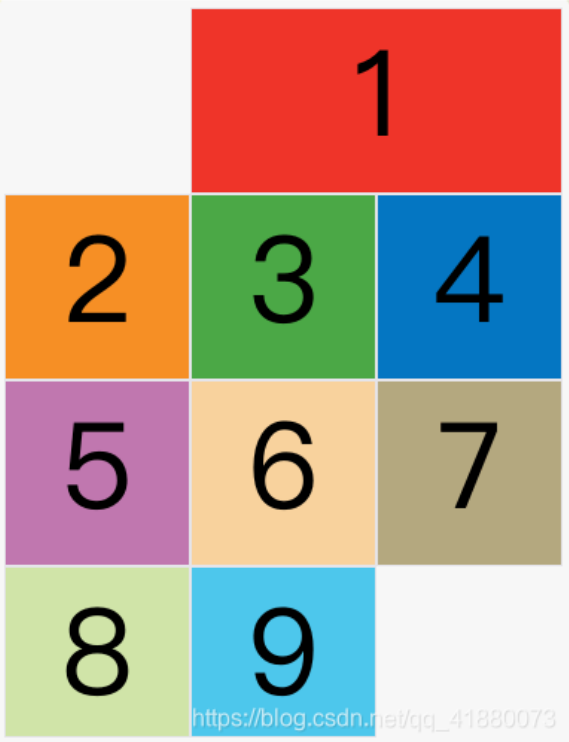
10.grid-column-start 属性，grid-column-end 属性，grid-row-start 属性，grid-row-end 属性

项目的位置是可以指定的，具体方法就是指定项目的四个边框，分别定位在哪根网格线。

grid-column-start属性：左边框所在的垂直网格线
grid-column-end属性：右边框所在的垂直网格线
grid-row-start属性：上边框所在的水平网格线
grid-row-end属性：下边框所在的水平网格线

```
.item-1 {
  grid-column-start: 2;
  grid-column-end: 4;
}
```

复制

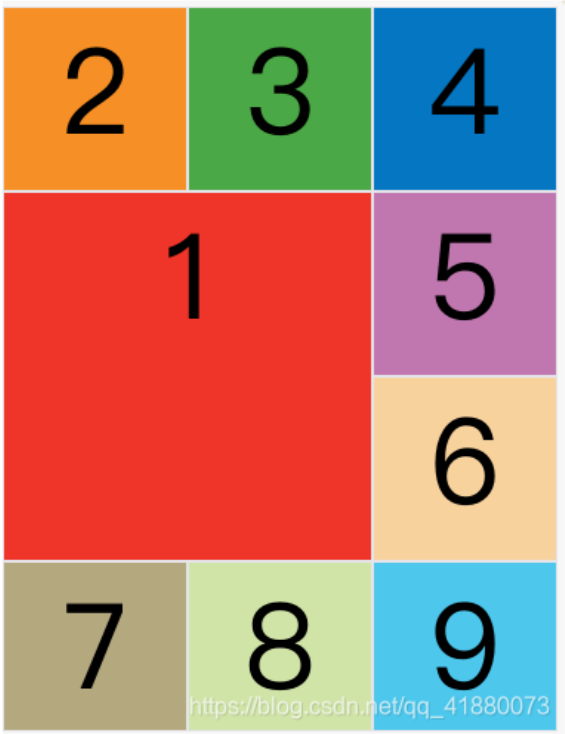


上图中，只指定了1号项目的左右边框（**左边框会对齐到第二根垂直网格线，右边框会对齐到第四根垂直网格线**），没有指定上下边框，所以会采用默认位置，即上边框是第一根水平网格线，下边框是第二根水平网格线。

除了1号项目以外，其他项目都没有指定位置，由浏览器自动布局，这时它们的位置由容器的grid-auto-flow属性决定，这个属性的默认值是row，因此会"先行后列"进行排列。读者可以把这个属性的值分别改成column、row dense和column dense，看看其他项目的位置发生了怎样的变化。

下面的例子是指定四个边框位置的效果。

```
.item-1 {
  grid-column-start: 1;
  grid-column-end: 3;
  grid-row-start: 2;
  grid-row-end: 4;
}
```



这四个属性的值，除了指定为第几个网格线，还可以指定为网格线的名字。

这四个属性的值，除了指定为第几个网格线，还可以指定为网格线的名字。

```
.item-1 {
  grid-column-start: header-start;
  grid-column-end: header-end;
}
```

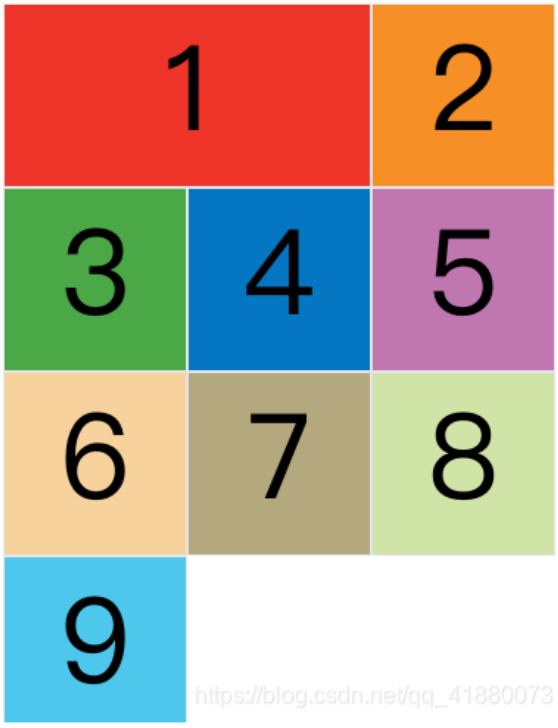
复制

上面代码中，左边框和右边框的位置，都指定为网格线的名字。

这四个属性的值还可以使用span关键字，表示"跨越"，即左右边框（上下边框）之间跨越多少个网格。

```
.item-1 {
  grid-column-start: span 2;
}
```

上面代码表示，1号项目的左边框距离右边框跨越2个网格。



grid-column属性是grid-column-start和grid-column-end的合并简写形式，grid-row属性是grid-row-start属性和grid-row-end的合并简写形式。

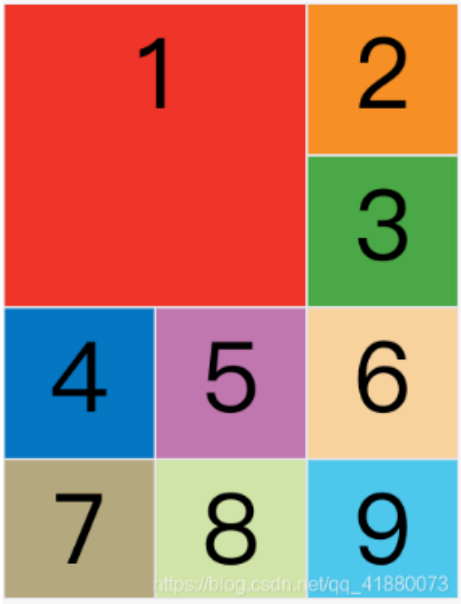
```
.item-1 {
  grid-column: 1 / 3;
  grid-row: 1 / 2;
}
/* 等同于 */
.item-1 {
  grid-column-start: 1;
  grid-column-end: 3;
  grid-row-start: 1;
  grid-row-end: 2;
}

```

这两个属性之中，也可以使用span关键字，表示跨越多少个网格。

```
.item-1 {
  background: #b03532;
  grid-column: 1 / 3;
  grid-row: 1 / 3;
}
/* 等同于 */
.item-1 {
  background: #b03532;
  grid-column: 1 / span 2;
  grid-row: 1 / span 2;
}
```

上面代码中，项目item-1占据的区域，包括第一行 + 第二行、第一列 + 第二列。



grid-area属性还可用作grid-row-start、grid-column-start、grid-row-end、grid-column-end的合并简写形式，直接指定项目的位置。

下面是一个例子。

```
.item-1 {
  grid-area: 1 / 1 / 3 / 3;
}
```

11.justify-self 属性，align-self 属性，place-self 属性

justify-self属性设置单元格内容的水平位置（左中右），跟justify-items属性的用法完全一致，但只作用于单个项目。

align-self属性设置单元格内容的垂直位置（上中下），跟align-items属性的用法完全一致，也是只作用于单个项目。

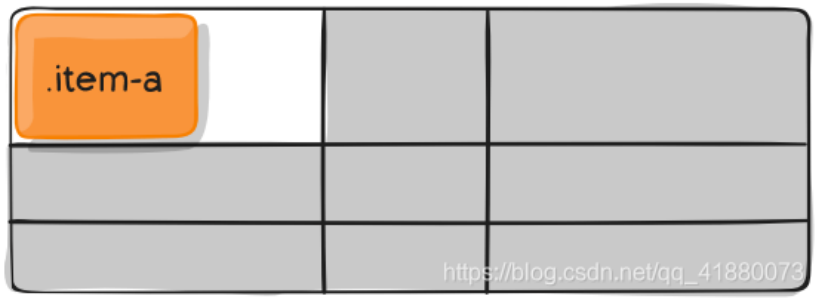
```
.item {
  justify-self: start | end | center | stretch;
  align-self: start | end | center | stretch;
}

```

这两个属性都可以取下面四个值。
start：对齐单元格的起始边缘。
end：对齐单元格的结束边缘。
center：单元格内部居中。
stretch：拉伸，占满单元格的整个宽度（默认值）。

下面是justify-self: start的例子。

```
.item-1  {
  justify-self: start;
}
```



place-self属性是align-self属性和justify-self属性的合并简写形式。
下面是一个例子。

```
place-self: center center;
```

复制

如果省略第二个值，place-self属性会认为这两个值相等。

12.小示例

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title></title>
        <style type="text/css">
            span {
                font-size: 2em;
            }

            #container {

                justify-content: center;
                align-content: center;
                display: grid;
                grid-template-columns: repeat(3,200px);
                grid-template-rows: repeat(3,100px);
                border: 2px solid black;
                gap: 20px;
                box-sizing: border-box;
            }

            .item {
                font-size: 2em;
                text-align: center;
                border: 1px solid #e5e4e9;
            }

            .item-1 {
```

```
                background-color: #ef342a;
                grid-column: 1/3;
                grid-row: 1/3;
            }

            .item-2 {
                background-color: #f68f26;

            }

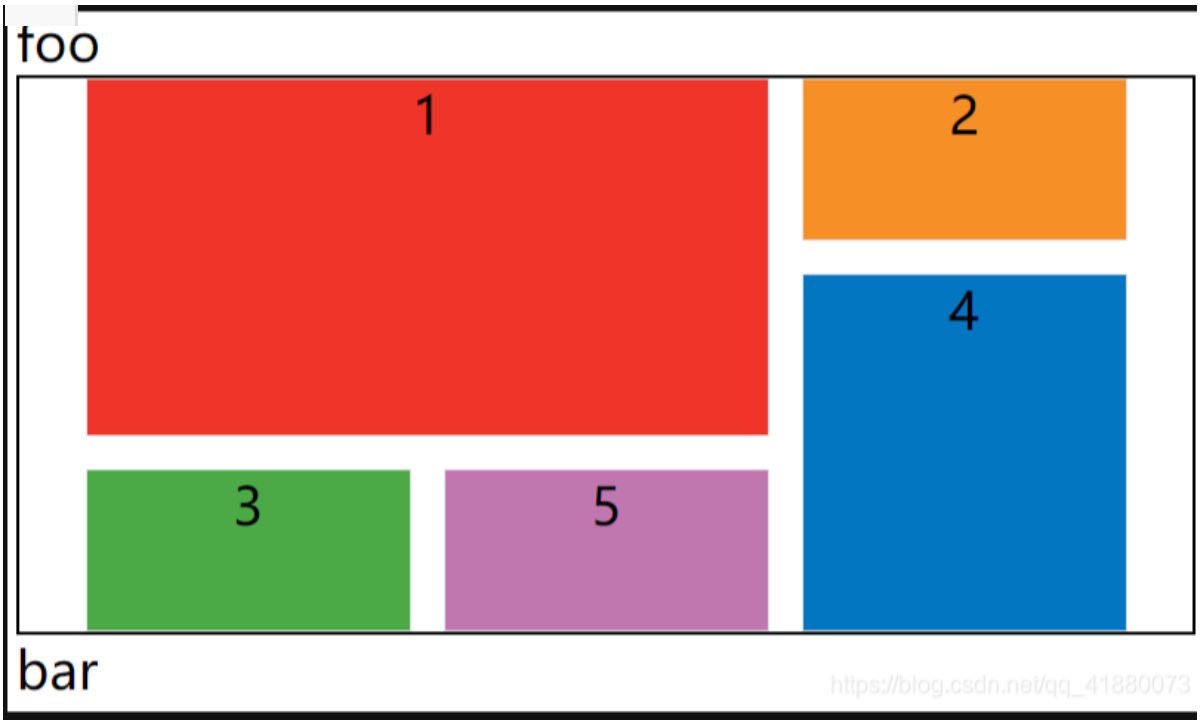
            .item-3 {
                background-color: #4ba946;
            }

            .item-4 {
                background-color: #0376c2;
                grid-column: 3/4;
                grid-row: 2/4;
            }

            .item-5 {
                background-color: #c077af;
            }

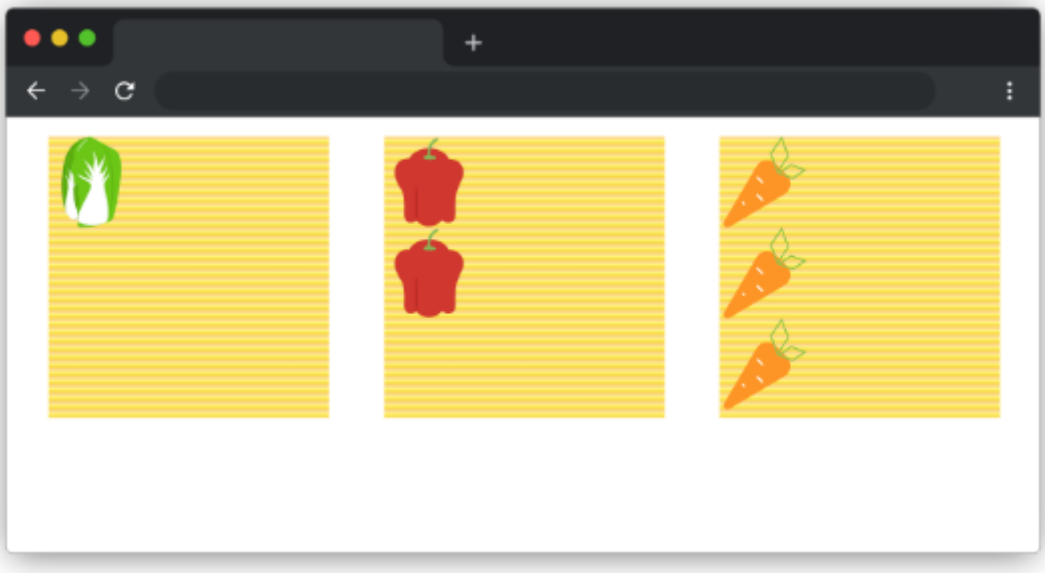
        </style>
    </head>
    <body>
        <span>foo</span>
        <div id="container">
            <div class="item item-1">1</div>
            <div class="item item-2">2</div>
            <div class="item item-3">3</div>
            <div class="item item-4">4</div>
            <div class="item item-5">5</div>
        </div>
        <span>bar</span>
    </body>
</html>


```



13.例题

新鲜蔬菜



完成 `css/style.css` 中的 TODO 部分。所有元素的大小都已给出，无需修改，完成后效果如下（图中灰色线条为布局参考线无需实现）：

```
/* TODO：待补充代码 */
.box {
  display: grid;
  grid-template: repeat(3,1fr) / repeat(3,1fr);//3行3列，fr为网格单位
}
#box1 .item,#box3 .item:nth-child(2){
  grid-area: 2 / 2;
} //2行2列的交叉点
#box2 .item:nth-child(2),#box3 .item:nth-child(3){
  grid-area: 3 / 3;  //3行3列的交叉点
}
```

媒体查询

1.媒体类型

值	含义
`all`	检测所有设备。
`screen`	检测电子屏幕，包括：电脑屏幕、平板屏幕、手机屏幕等。
`print`	检测打印机。

2.媒体特性

1.2 媒体特性

值	含义
width	检测视口**宽度**。
max-width	检测视口**最大宽度**。
min-width	检测视口**最小宽度**。
height	检测视口**高度**。
max-height	检测视口**最大高度**。
min-height	检测视口**最小高度**。
device-width	检测设备**屏幕的宽度**。
max-device-width	检测设备**屏幕的最大宽度**。
min-device-width	检测设备**屏幕的最小宽度**。
orientation	检测**视口的旋转方向**（是否横屏）。 1. portrait：视口处于纵向，即高度大于等于宽度。 2. landscape：视口处于横向，即宽度大于高度。

1. `width` and `height`：这两个特性主要用于描述设备显示区域的宽度和高度，可以用来调整不同设备下的网页布局。例如：

```
@media screen and (max-width: 600px) {
    body {
        background-color: lightblue;
    }
}
```

以上CSS规则表示，当设备的屏幕宽度小于或等于600px时，网页背景颜色变为浅蓝色。

2. `orientation`：描述设备的方向，值可以是`portrait`（竖屏）或`landscape`（横屏）。例如：

```
@media screen and (orientation: portrait) {
    body {
        font-size: 1.2em;
    }
}
```

以上CSS规则表示，当设备处于竖屏状态时，网页字体变为1.2em。

3. `device-width` and `device-height`：这两个特性用于描述设备的物理尺寸。例如：

```
@media screen and (min-device-width: 1200px) {
    body {
        margin: 0 auto;
        width: 1200px;
    }
}
```

以上CSS规则表示，若设备的物理屏幕宽度大于或等于1200px，网页主体将居中显示，并且宽度恒定为1200px。

4. `resolution`：描述设备的分辨率。例如：

```
@media print and (min-resolution: 300dpi) {
    body {
        font-size: 12pt;
    }
}
```

以上CSS规则表示，如果设备的打印分辨率至少为300 dot per inch，网页字体变为12pt。

5. `aspect-ratio` and `device-aspect-ratio`：前者描述的是显示区域的宽高比，后者描述设备物理屏幕的宽高比。例如：

```
@media screen and (aspect-ratio: 16/9) {
    body {
        background: url('widescreen-bg.jpg');
    }
}
```

以上CSS规则表示，如果设备的显示区域的宽高比是16:9，那么网页背景图设为`widescreen-bg.jpg`。

3.示例

1. 在移动端和桌面端显示不同的导航菜单：

```
@media screen and (max-width: 768px) {
  /* 移动端 */
  .menu {
    display: none;
  }

  .mobile-menu {
    display: block;
  }
}
```

```
@media screen and (min-width: 769px) {
  /* 桌面端 */
  .menu {
    display: block;
  }

  .mobile-menu {
    display: none;
  }
}

```

2.在打印时隐藏某些元素：

```
@media print {
  .print-hidden {
    display: none;
  }
}

```

3.根据屏幕大小和朝向应用不同的背景图像：

```
@media screen and (orientation: portrait) {
  .bg-image {
    background-image: url(portrait.jpg);
  }
}

@media screen and (orientation: landscape) {
  .bg-image {
    background-image: url(landscape.jpg);
  }
}

```

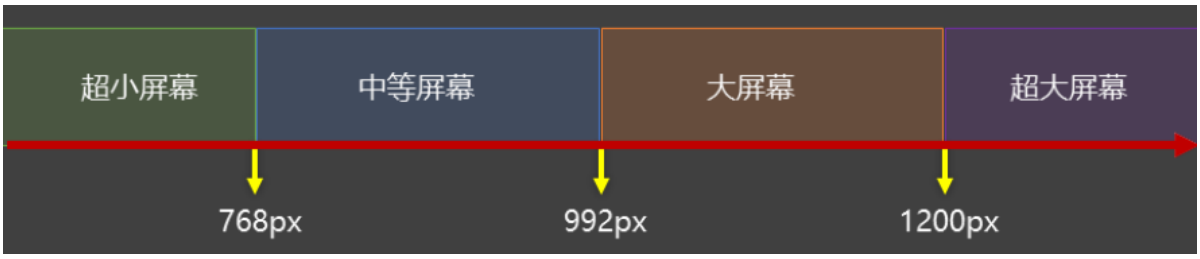
4.link引入

```
<link rel="stylesheet" media="(max-width: 768px)" href="example.css">

```

上述代码的作用是在浏览器视口宽度小于或等于768px时，应用`example.css`样式表。

5.常用阈值



4.例题

- 以 `800px` 为界限，`800px` 以上显示 PC 端布局，否则显示移动端布局，**需要实现移动端布局样式**如下：

- 移动端 **Menu** 由左上侧按钮（即 class 包含 `icon-menu` 的 `label` 标签）控制显隐，按钮大小已经默认提供，无需手动设置大小。且 Menu 按钮展示时，需要浮动在内容卡片上方，不能被遮挡，移动端和 PC 端顶部导航栏高度一致，均为 `54px`。
- 移动端导航栏的菜单项每一项独占一行。
- 显示移动端布局时，卡片描述和对应图片各占一行，且都撑满 `#tutorials` 容器。

移动端页面展开菜单栏效果如下所示：


```
/* TODO: 考生需要完成以下内容 */
/* 当屏幕宽度小于800px时的样式 */
@media (max-width: 800px) {
  /* 设置菜单的高度和内边距 */
  .menu {
    height: 54px;
    padding: 0 20px;
  }

  /* 默认隐藏折叠菜单 */
  .collapse {
    display: none; /* 隐藏折叠菜单 */
    position: absolute; /* 绝对定位，使其脱离文档流 */
    top: 100%; /* 位于菜单下方 */
    left: 0;
    right: 0;
    background: inherit; /* 继承父元素的背景 */
    border-top: 1px #959595 solid; /* 添加顶部边框 */
  }

  /* 折叠菜单中的列表项改为块级显示 */
  .collapse li {
    display: block;
  }

  /* 折叠菜单中的子菜单改为相对定位 */
```

```
  .collapse li ul {
    position: relative;
  }

  /* 显示菜单按钮的样式 */
  label.menu-btn {
    display: inline-block; /* 显示为行内块级元素 */
    color: #999; /* 文字颜色 */
    cursor: pointer; /* 鼠标悬停时显示为手型 */
    line-height: 54px; /* 与菜单高度一致 */
  }

  /* 鼠标悬停时菜单按钮的样式 */
  label.menu-btn:hover {
    color: #fff; /* 鼠标悬停时文字颜色 */
  }

  /* 当菜单按钮被选中时，显示折叠菜单 */
  input.menu-btn:checked ~ .collapse {
    display: block; /* 显示折叠菜单 */
  }

  /* 在小屏幕上，页面内容的布局调整 */
  #tutorials .row {
    grid-template-columns: 1fr; /* 只有一列 */
  }

  /* 图片的外边距调整 */
  #tutorials .row img {
    margin: 0; /* 移除外边距 */
  }
}
```

transform

1.2D位移

2D 位移可以改变元素的位置，具体使用方式如下：

1. 先给元素添加 **转换属性** `transform`
2. 编写 `transform` 的具体值，相关可选值如下：

值	含义
`translateX`	设置水平方向位移，需指定长度值；若指定的是百分比，是参考自身宽度的百分比。
`translateY`	设置垂直方向位移，需指定长度值；若指定的是百分比，是参考自身高度的百分比。
`translate`	一个值代表水平方向，两个值代表：水平和垂直方向。

1. 位移与相对定位很相似，都不脱离文档流，不会影响到其它元素。
2. 与相对定位的区别：相对定位的百分比值，参考的是其父元素；定位的百分比值，参考的是其自身。
3. 浏览器针对位移有优化，与定位相比，浏览器处理位移的效率更高。
4. `transform` 可以链式编写，例如：

```
transform: translateX(30px) translateY(40px);
```

5. 位移对行内元素无效。
6. 位移配合定位，可实现元素水平垂直居中

```
.box {
    position: absolute;
    left: 50%;
    top: 50%;
    transform: translate(-50%, -50%);
}
```

2.2D缩放

`2D` 缩放是指：让元素放大或缩小，具体使用方式如下：

1. 先给元素添加 **转换属性** `transform`
2. 编写 `transform` 的具体值，相关可选值如下：

值	含义
`scaleX`	设置水平方向的缩放比例，值为一个数字，`1` 表示不缩放，大于 `1` 放大，小于 `1` 缩小。
`scaleY`	设置垂直方向的缩放比例，值为一个数字，`1` 表示不缩放，大于 `1` 放大，小于 `1` 缩小。
`scale`	同时设置水平方向、垂直方向的缩放比例，一个值代表同时设置水平和垂直缩放；两个值分别代表：水平缩放、垂直缩放。

3. 注意点：

    1. `scale` 的值，是支持写负数的，但几乎不用，因为容易让人产生误解。
    2. 借助缩放，可实现小于 `12px` 的文字。

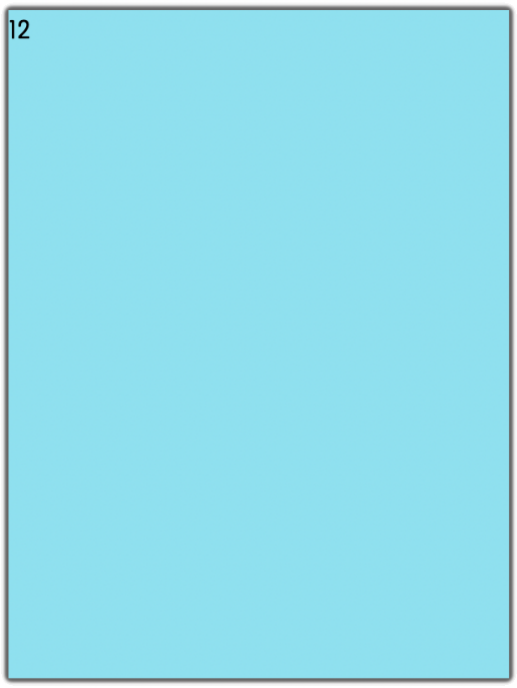
3.2D旋转

`2D` 旋转是指：让元素在二维平面内，顺时针旋转或逆时针旋转，具体使用方式如下：

1. 先给元素添加 **转换属性** `transform`
2. 编写 `transform` 的具体值，相关可选值如下：

值	含义
`rotate`	设置旋转角度，需指定一个角度值(`deg`)，正值顺时针，负值逆时针。

> 注意： `rotateZ(20deg)` 相当于 `rotate(20deg)` ，当然到了 `3D` 变换的时候，还能写：`rotate(x,x,x)`



当鼠标**悬浮**在元素上，元素呈扇形展开，页面效果如下所示：

完成后的效果见文件夹下面的 gif 图，图片名称为 `effect.gif`（提示：可以通过 VS Code 或者浏览器预览 gif 图片）。

具体说明如下：

- 页面上有 12 个相同大小的 `div` 元素。
- 这 12 个 `div` 元素具有不同的背景颜色。
- 前 6 个 `div` 元素（`id="item1"~id="item6"`）均为**逆时针**转动，其**最小**转动的角度为 `10 deg`，相邻元素间的角度差为 `10 deg`。
- 后 6 个 `div` 元素（`id="item7"~id="item12"`）均为**顺时针**转动，其**最小**转动的角度为 `10 deg`，相邻元素间的角度差为 `10 deg`。
- 注意，元素 6（`id="item6"`）和元素 7（`id="item7"`），各自反方向转动 `10 deg`，所以它们之间的角度差为 `20 deg`。

```
#box:hover div:nth-child(6){
  transform: rotate(-10deg);
}
#box:hover div:nth-child(5){
  transform: rotate(-20deg);
}
#box:hover div:nth-child(4){
  transform: rotate(-30deg);
}
#box:hover div:nth-child(3){
  transform: rotate(-40deg);
}
#box:hover div:nth-child(2){
  transform: rotate(-50deg);
}
```

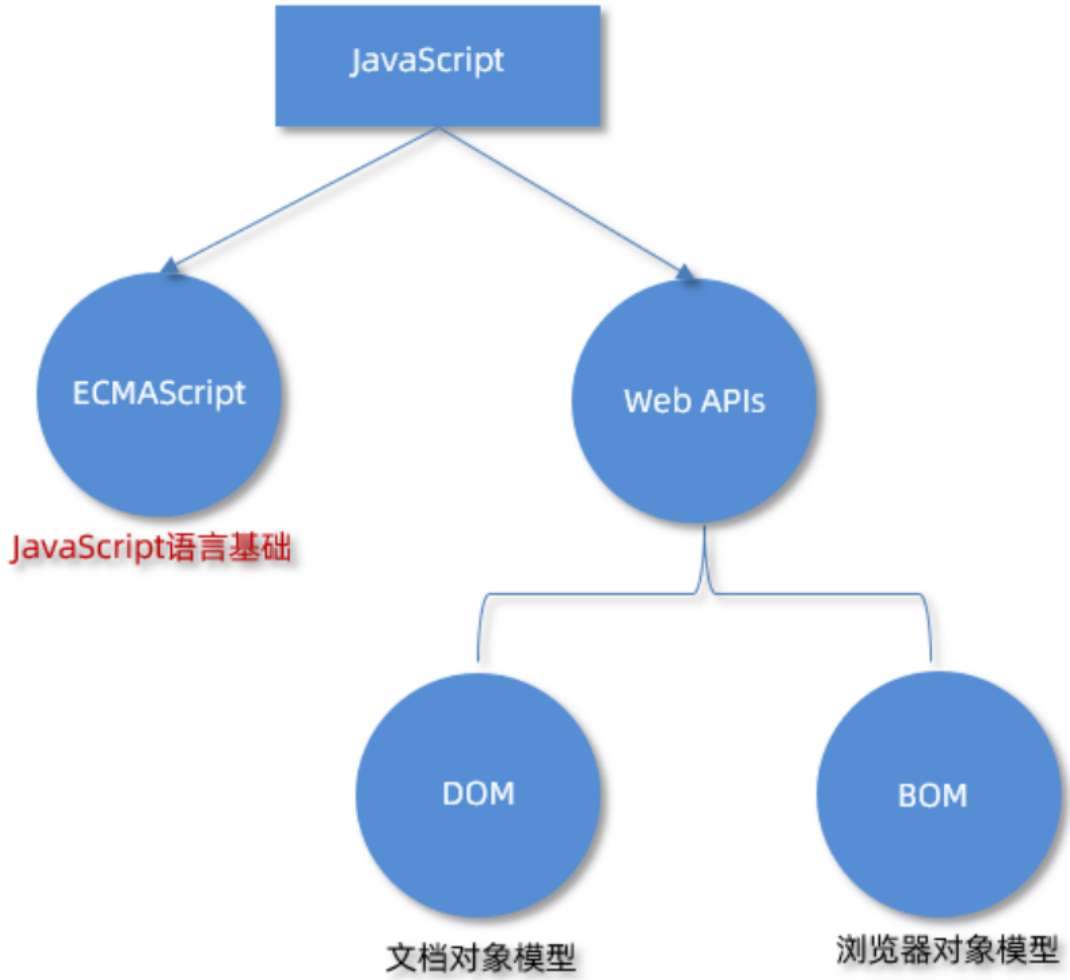
```
#box:hover div:nth-child(1){
  transform: rotate(-60deg);
}
#box:hover div:nth-child(7){
  transform: rotate(10deg);
}
#box:hover div:nth-child(8){
  transform: rotate(20deg);
}
#box:hover div:nth-child(9){
  transform: rotate(30deg);
}
#box:hover div:nth-child(10){
  transform: rotate(40deg);
}
#box:hover div:nth-child(11){
  transform: rotate(50deg);
}
#box:hover div:nth-child(12){
  transform: rotate(60deg);
}
```

JavaScript基础

js的组成

JavaScript的组成

- ECMAScript:
  - 规定了js基础语法核心知识。
  - 比如：变量、分支语句、循环语句、对象等等
- Web APIs :
  - DOM 文档对象模型， 定义了一套操作HTML文档的API
  - BOM 浏览器对象模型，定义了一套操作浏览器窗口的API



1.内联和外联

内联

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>JavaScript 基础 - 引入方式</title>
</head>
<body>
  <!-- 内联形式：通过 script 标签包裹 JavaScript 代码 -->
  <script>
    alert('嗨，欢迎来传智播学习前端技术！')
  </script>
</body>
</html>
```

外联

```
/ demo.js
document.write('嗨，欢迎来传智播学习前端技术！')
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>JavaScript 基础 - 引入方式</title>
</head>
<body>
  <!-- 外部形式：通过 script 的 src 属性引入独立的 .js 文件 -->
  <script src="demo.js"></script>
</body>
</html>
```

2.输入和输出

输出

JavaScript 可以接收用户的输入，然后再将输入的结果输出：

`alert()`、`document.wirte()`

以数字为例，向 `alert()` 或 `document.write()` 输入任意数字，他都会以弹窗形式展示（输出）给用户

输入

向 `prompt()` 输入任意内容会以弹窗形式出现在浏览器中，一般提示用户输入一些内容。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>JavaScript 基础 - 输入输出</title>
</head>
<body>

  <script>
    // 1. 输入的任意数字，都会以弹窗形式展示
    document.write('要输出的内容')
    alert('要输出的内容');

    // 2. 以弹窗形式提示用户输入姓名，注意这里的文字使用英文的引号
    prompt('请输入您的姓名:')
  </script>
</body>
</html>
```

声明

`let` 和 `var` 都是 JavaScript 中的声明变量的关键字，推荐使用 `let` 声明变量

JavaScript 使用专门的关键字 `let` 和 `var` 来声明（定义）变量，在使用时需要注意一些细节：

以下是使用 `let` 时的注意事项：

1. 允许声明和赋值同时进行
2. 不允许重复声明
3. 允许同时声明多个变量并赋值
4. JavaScript 中内置的一些关键字不能被当做变量名

以下是使用 `var` 时的注意事项：

2. 允许声明和赋值同时进行
3. 允许重复声明
4. 允许同时声明多个变量并赋值

大部分情况使用 `let` 和 `var` 区别不大，但是 `let` 相较 `var` 更严谨，因此推荐使用 `let`，后期会更进一步介绍二者间的区别。

3.数据类型和数值类型

通过 typeof 关键字检测数据类型和数值类型

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>JavaScript 基础 - 数据类型</title>
</head>
<body>

  <script>
    // 检测 1 是什么类型数据，结果为 number
    document.write(typeof 1)
  </script>
</body>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>JavaScript 基础 - 数据类型</title>
</head>
<body>

  <script>
    let score = 100 // 正整数
    let price = 12.345 // 小数
```

```
    let temperature = -40 // 负数

    document.write(typeof score) // 结果为 number
    document.write(typeof price) // 结果为 number
    document.write(typeof temperature) // 结果为 number
  </script>
</body>
</html>
```

3.1字符串类型

通过单引号（`''`）、双引号（`""`）或反引号包裹的数据都叫字符串，单引号和双引号没有本质上的区别，推荐使用单引号。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>JavaScript 基础 - 数据类型</title>
</head>
<body>

  <script>
    let user_name = '小明' // 使用单引号
    let gender = "男" // 使用双引号
    let str = '123' // 看上去是数字，但是用引号包裹了就成了字符串了
    let str1 = '' // 这种情况叫空字符串

    documeent.write(typeof user_name) // 结果为 string
    documeent.write(typeof gender) // 结果为 string
    documeent.write(typeof str) // 结果为 string
  </script>
</body>
</html>
```

3.2布尔类型

表示肯定或否定时在计算机中对应的是布尔类型数据，它有两个固定的值 `true` 和 `false`，表示肯定的数据用 `true`，表示否定的数据用 `false`。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>JavaScript 基础 - 数据类型</title>
</head>
<body>

  <script>
    //  pink老师帅不帅？回答 是 或 否
    let isCool = true // 是的，摔死了！
    isCool = false // 不，套马杆的汉子！
```

```
    document.write(typeof isCool) // 结果为 boolean
  </script>
</body>
</html>
```

3.3undefined

未定义是比较特殊的类型，只有一个值 undefined，只声明变量，不赋值的情况下，变量的默认值为 undefined，一般很少【直接】为某个变量赋值为 undefined。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>JavaScript 基础 - 数据类型</title>
</head>
<body>

  <script>
    // 只声明了变量，并未赋值
    let tmp;
    document.write(typeof tmp) // 结果为 undefined
  </script>
</body>
</html>
```

4.类型转换

4.1隐式转换

某些运算符被执行时，系统内部自动将数据类型进行转换，这种转换称为隐式转换。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>JavaScript 基础 - 隐式转换</title>
</head>
<body>
  <script>
    let num = 13 // 数值
    let num2 = '2' // 字符串

    // 结果为 132
    // 原因是将数值 num 转换成了字符串，相当于 '13'
    // 然后 + 将两个字符串拼接到了一起
    console.log(num + num2)

    // 结果为 11
    // 原因是将字符串 num2 转换成了数值，相当于 2
    // 然后数值 13 减去 数值 2
    console.log(num - num2)

```

```
    let a = prompt('请输入一个数字')
    let b = prompt('请再输入一个数字')

    alert(a + b);
  </script>
</body>
</html>
```

4.1显示转换

Number

通过 `Number` 显示转换成数值类型，当转换失败时结果为 `NaN`（Not a Number）即不是一个数字。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>JavaScript 基础 - 隐式转换</title>
</head>
<body>
  <script>
    let t = '12'
    let f = 8

    // 显式将字符串 12 转换成数值 12
    t = Number(t)

    // 检测转换后的类型
    // console.log(typeof t);
    console.log(t + f) // 结果为 20

    // 并不是所有的值都可以被转成数值类型
    let str = 'hello'
    // 将 hello 转成数值是不现实的，当无法转换成
    // 数值时，得到的结果为 NaN （Not a Number）
    console.log(Number(str))
  </script>
</body>
</html>
```

5.运算符

5.1算术运算符

```
// 算术运算符
console.log(1 + 2 * 3 / 2) //  4
let num = 10
console.log(num + 10)  // 20
console.log(num + num)  // 20

// 1. 取模(取余数)  使用场景：  用来判断某个数是否能够被整除
console.log(4 % 2) //  0
console.log(6 % 3) //  0
```

```
console.log(5 % 3) //  2
console.log(3 % 5) //  3

// 2. 注意事项 : 如果我们计算失败，则返回的结果是 NaN (not a number)
console.log('pink老师' - 2)
console.log('pink老师' * 2)
console.log('pink老师' + 2)   // pink老师2
```

5.2赋值运算符

运算符	作用
+=	加法赋值
-+	减法赋值
*=	乘法赋值
/=	除法赋值
%=	取余赋值

```
<script>
let num = 1
// num = num + 1
// 采取赋值运算符
// num += 1
num += 3
console.log(num)
</script>
```

5.3自增自减运算符

```
<script>
    // let num = 10
    // num = num + 1
    // num += 1
    // // 1. 前置自增
    // let i = 1
    // ++i
    // console.log(i) //2

    // let i = 1
    // console.log(++i + 1)  //3
    // 2. 后置自增
    // let i = 1
    // i++
    // console.log(i)//2
    // let i = 1
```

```
    // console.log(i++ + 1)//2

    // 了解
    let i = 1
    console.log(i++ + ++i + i)//1+3+3
  </script>
```

5.4逻辑运算符

符号	名称	日常读法	特点	口诀
&&	逻辑与	并且	符号两边有一个假的结果为假	一假则假
||	逻辑或	或者	符号两边有一个真的结果为真	一真则真
!	逻辑非	取反	true变false false变true	真变假，假变真

A	B	A && B	A || B	!A
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

5.5运算符优先级

逻辑运算符优先级：！> && > ||

6.分支语句

1. if分支语句（重点）
2. 三元运算符
3. switch语句

6.1 if 分支语句

```
if(条件表达式) {
  // 满足条件要执行的语句
}
```

示例：

```
<script>
    // 单分支语句
    // if (false) {
    //   console.log('执行语句')
    // }
    // if (3 > 5) {
    //   console.log('执行语句')
    // }
    // if (2 === '2') {
    //   console.log('执行语句')
    // }
    //  1. 除了0 所有的数字都为真
    //   if (0) {
    //     console.log('执行语句')
    //   }
    // 2.除了 '' 所有的字符串都为真 true
    // if ('pink老师') {
    //   console.log('执行语句')
    // }
    // if ('') {
    //   console.log('执行语句')
    // }
    // // if ('') console.log('执行语句')

    // 1. 用户输入
    let score = +prompt('请输入成绩')
    // 2. 进行判断输出
    if (score >= 700) {
      alert('恭喜考入黑马程序员')
    }
    console.log('------------------')

  </script>
```

```
 <script>
    // 1. 用户输入
    let uname = prompt('请输入用户名:')
    let pwd = prompt('请输入密码:')
    // 2. 判断输出
    if (uname === 'pink' && pwd === '123456') {
      alert('恭喜登录成功')
    } else {
      alert('用户名或者密码错误')
    }
  </script>
```

```
<script>
    // 1. 用户输入
    let score = +prompt('请输入成绩：')
    // 2. 判断输出
    if (score >= 90) {
      alert('成绩优秀，宝贝，你是我的骄傲')
    } else if (score >= 70) {
      alert('成绩良好，宝贝，你要加油哦~~')
    } else if (score >= 60) {
      alert('成绩及格，宝贝，你很危险~')
    } else {
      alert('成绩不及格，宝贝，我不想和你说话，我只想用鞭子和你说话~')
    }
  </script>
```

6.2三元运算符

```
条件 ? 表达式1 ：  表达式2
```

示例:

```
// 三元运算符（三元表达式）
// 1. 语法格式
// 条件 ? 表达式1 : 表达式2

// 2. 执行过程
// 2.1 如果条件为真，则执行表达式1
// 2.2 如果条件为假，则执行表达式2

// 3. 验证
```

```
// 5 > 3 ? '真的' : '假的'
console.log(5 < 3 ? '真的' : '假的')

// let age = 18
// age = age + 1
//  age++

// 1. 用户输入
let num = prompt('请您输入一个数字:')
// 2. 判断输出- 小于10才补0
// num = num < 10 ? 0 + num : num
num = num >= 10 ? num : 0 + num
alert(num)
```

6.3 switch语句

```
// switch分支语句
// 1. 语法
// switch (表达式) {
//   case 值1:
//     代码1
//     break

//   case 值2:
//     代码2
//     break
//   ...
//   default:
//     代码n
// }

<script>
  switch (2) {
    case 1:
    console.log('您选择的是1')
    break  // 退出switch
    case 2:
    console.log('您选择的是2')
    break  // 退出switch
    case 3:
    console.log('您选择的是3')
    break  // 退出switch
    default:
    console.log('没有符合条件的')
  }
</script>
```

7.循环语句

1.while循环

2.for 循环

7.1while循环

```
while (条件表达式) {
   // 循环体
}
```

```
// while循环: 重复执行代码

// 1. 需求: 利用循环重复打印3次 '月薪过万不是梦，毕业时候见英雄'
let i = 1
while (i <= 3) {
  document.write('月薪过万不是梦，毕业时候见英雄~<br>')
  i++   // 这里千万不要忘了变量自增否则造成死循环
}
```

中止循环：`break` 中止整个循环，一般用于结果已经得到, 后续的循环不需要的时候可以使用（提高效率)

`continue` 中止本次循环，一般用于排除或者跳过某一个选项的时候

```
<script>
    // let i = 1
    // while (i <= 5) {
    //   console.log(i)
    //   if (i === 3) {
    //     break  // 退出循环
    //   }
    //   i++

    // }


    let i = 1
    while (i <= 5) {
      if (i === 3) {
        i++
        continue
      }
      console.log(i)
      i++

    }
  </script>
```

无限循环：1.while(true) 来构造“无限”循环，需要使用break退出循环。（常用）

2.for(;;) 也可以来构造“无限”循环，同样需要使用break退出循环。

```
// 无限循环
// 需求： 页面会一直弹窗询问你爱我吗?
// (1). 如果用户输入的是 '爱', 则退出弹窗
// (2). 否则一直弹窗询问

// 1. while(true) 无限循环
// while (true) {
//   let love = prompt('你爱我吗?')
//   if (love === '爱') {
//     break
//   }
// }

// 2. for(;;) 无限循环
for (; ;) {
  let love = prompt('你爱我吗?')
  if (love === '爱') {
    break
  }
}
```

7.2for循环

```
<script>
  // 1. 语法格式
  // for(起始值; 终止条件; 变化量) {
  //   // 要重复执行的代码
  // }

  // 2. 示例：在网页中输入标题标签
  // 起始值为 1
  // 变化量 i++
  // 终止条件 i <= 6
  for(let i = 1; i <= 6; i++) {
    document.write(`<h${i}>循环控制，即重复执行<h${i}>`)
  }
</script>
```

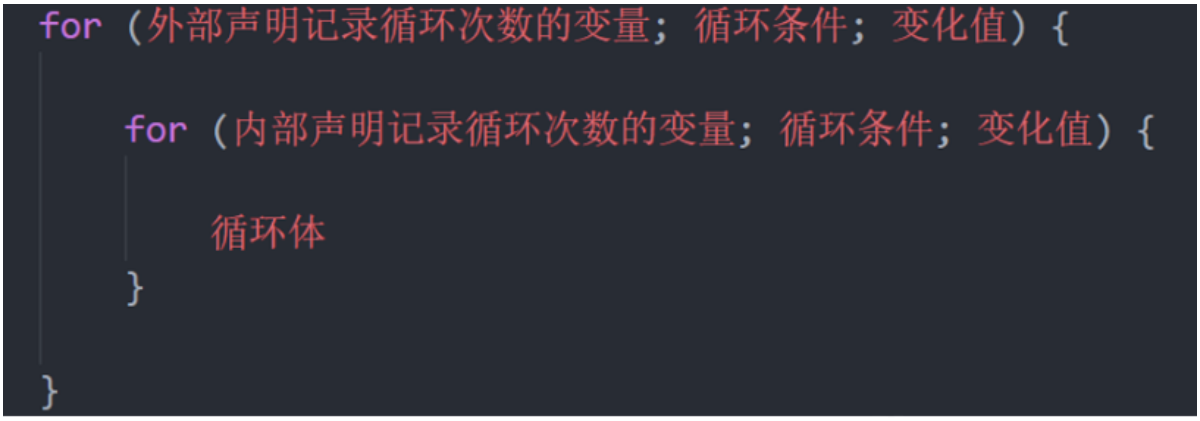
2.变化量和死循环，`for` 循环和 `while` 一样，如果不合理设置增量和终止条件，便会产生死循环。

3.跳出和终止循环

```
<script>
    // 1. continue
    for (let i = 1; i <= 5; i++) {
        if (i === 3) {
            continue  // 结束本次循环，继续下一次循环
        }
        console.log(i)
    }
    // 2. break
```

```
    for (let i = 1; i <= 5; i++) {
        if (i === 3) {
            break  // 退出结束整个循环
        }
        console.log(i)
    }
</script>
```

循环嵌套




```
// 1. 外面的循环 记录第n天
for (let i = 1; i < 4; i++) {
    document.write(`第${i}天 <br>`)
    // 2. 里层的循环记录 几个单词
    for (let j = 1; j < 6; j++) {
        document.write(`记住第${j}个单词<br>`)
    }
}
```

8.数组

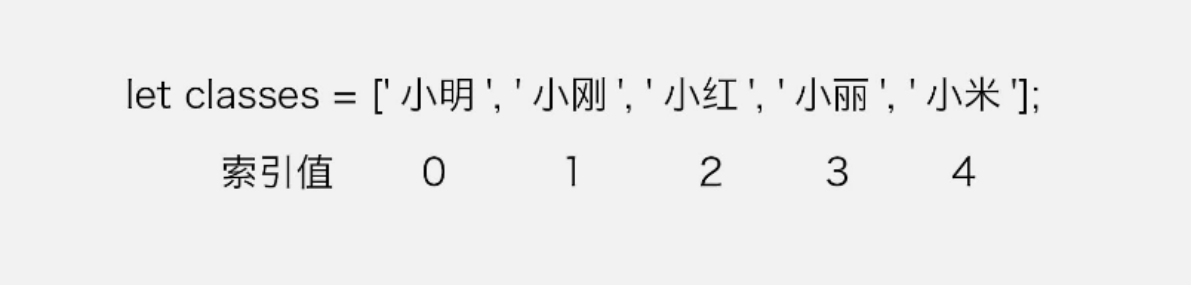
(Array)是一种可以按顺序保存数据的数据类型

定义数组和数组单元

```
<script>
  // 1. 语法，使用 [] 来定义一个空数组
  // 定义一个空数组，然后赋值给变量 classes
  // let classes = [];

  // 2. 定义非空数组
  let classes = ['小明', '小刚', '小红', '小丽', '小米']
</script>
```

```
访问数组和数组索引
```




```
<script>
  let classes = ['小明', '小刚', '小红', '小丽', '小米']

  // 1. 访问数组，语法格式为：变量名[索引值]
  document.write(classes[0]) // 结果为：小明
  document.write(classes[1]) // 结果为：小刚
  document.write(classes[4]) // 结果为：小米

  // 2. 通过索引值还可以为数组单重新赋值
  document.write(classes[3]) // 结果为：小丽
  // 重新为索引值为 3 的单元赋值
  classes[3] = '小小丽'
  document.wirte(classes[3]); // 结果为： 小小丽
</script>
```

8.1操作数组

1. push 动态向数组的尾部添加一个单元
2. unshit 动态向数组头部添加一个单元
3. pop 删除最后一个单元
4. shift 删除第一个单元
5. splice 动态删除任意单元

```
<script>
  // 定义一个数组
  let arr = ['html', 'css', 'javascript']

  // 1. push 动态向数组的尾部添加一个单元
  arr.push('Nodejs')
  console.log(arr)
  arr.push('Vue')

  // 2. unshit 动态向数组头部添加一个单元
  arr.unshift('VS Code')
  console.log(arr)

  // 3. splice 动态删除任意单元
  arr.splice(2, 1) // 从索引值为2的位置开始删除1个单元
  console.log(arr)

  // 4. pop 删除最后一个单元
  arr.pop()
  console.log(arr)

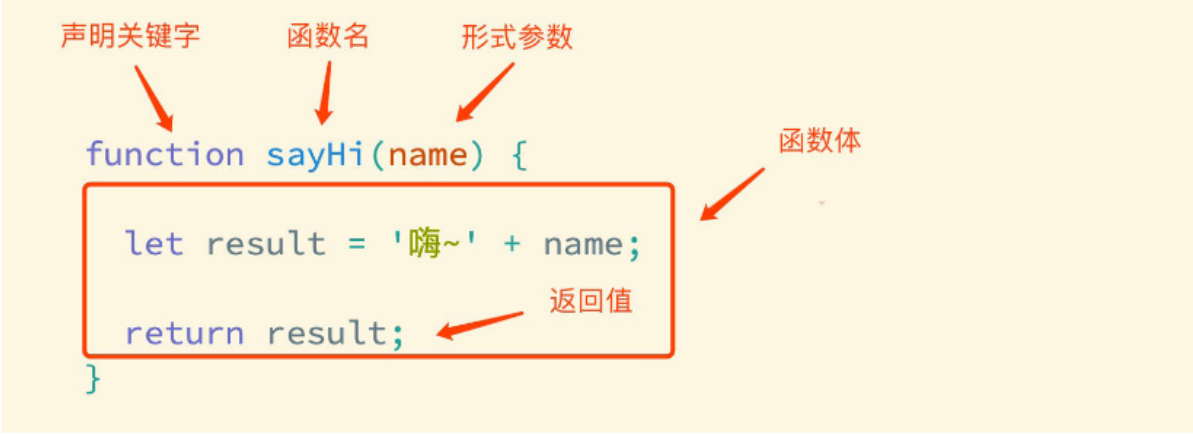
```

```
  // 5. shift 删除第一个单元
  arr.shift()
  console.log(arr)
</script>
```

9.函数

9.1声明定义

声明（定义）一个完整函数包括关键字、函数名、形式参数、函数体、返回值5个部分



调用

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>JavaScript 基础 - 声明和调用</title>
</head>
<body>
  <script>
    // 声明（定义）了最简单的函数，既没有形式参数，也没有返回值
    function sayHi() {
      console.log('嗨~')
    }
    // 函数调用，这些函数体内的代码逻辑会被执行
    // 函数名()

    sayHi()
    // 可以重复被调用，多少次都可以
    sayHi()
  </script>
</body>
</html>
```

9.2参数

通过向函数传递参数，可以让函数更加灵活多变，参数可以理解成是一个变量。

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
  <meta charset="UTF-8">
  <title>JavaScript 基础 - 函数参数</title>
</head>
<body>

  <script>
    // 声明（定义）一个功能为打招呼的函数
    // function sayHi() {
    //   console.log('嗨~')
    // }
    // 调用函数
    // sayHi()


    // 这个函数似乎没有什么价值，除非能够向不同的人打招呼
    // 这就需要借助参数来实现了
    function sayHi(name) {
      // 参数 name 可以被理解成是一个变量
      console.log(name)
      console.log('嗨~' + name)
    }

    // 调用 sayHi 函数，括号中多了 '小明'
    // 这时相当于为参数 name 赋值了
    sayHi('小明')// 结果为 小明

    // 再次调用 sayHi 函数，括号中多了 '小红'
    // 这时相当于为参数 name 赋值了
    sayHi('小红') // 结果为 小红
  </script>
</body>
</html>
```

形参：声明函数时写在函数名右边小括号里的叫形参（形式上的参数）

实参：调用函数时写在函数名右边小括号里的叫实参（实际上的参数）

形参可以理解为是在这个函数内声明的变量（比如 num1 = 10）实参可以理解为是给这个变量赋值

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>JavaScript 基础 - 函数参数</title>
</head>
<body>
  <script>
    // 声明（定义）一个计算任意两数字和的函数
    // 形参 x 和 y 分别表示任意两个数字，它们是两个变量
    function count(x, y) {
      console.log(x + y);
    }
    // 调用函数，传入两个具体的数字做为实参
    // 此时 10 赋值给了形参 x
```

```
    // 此时 5  赋值给了形参 y
    count(10, 5); // 结果为 15
  </script>
</body>
</html>
```

9.3返回值

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>JavaScript 基础 - 函数返回值</title>
</head>
<body>

  <script>
    // 定义求和函数
    function count(a, b) {
      let s = a + b
      // s 即为 a + b 的结果
      // 通过 return 将 s 传递到外部
      return s
    }

    // 调用函数，如果一个函数有返回值
    // 那么可将这个返回值赋值给外部的任意变量
    let total = count(5, 12)
  </script>
</body>
</html>
```

1. 在函数体中使用return 关键字能将内部的执行结果交给函数外部使用
2. 函数内部只能出现1 次 return，并且 return 下一行代码不会再被执行，所以return 后面的数据不要换行写
3. return会立即结束当前函数
4. 函数可以没有return，这种情况默认返回值为 undefined

9.4匿名函数

函数可以分为具名函数和匿名函数

匿名函数：没有名字的函数,无法直接使用。

```
// 声明
let fn = function() {
   console.log('函数表达式')
}
// 调用
fn()
```

10.对象

1.语法

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>JavaScript 基础 - 对象语法</title>
</head>
<body>

  <script>
    // 声明字符串类型变量
    let str = 'hello world!'

    // 声明数值类型变量
    let num = 199

    // 声明对象类型变量，使用一对花括号
    // user 便是一个对象了，目前它是一个空对象
    let user = {}
  </script>
</body>
</html>
```

2.属性和访问

1. 属性都是成 对出现的，包括属性名和值，它们之间使用英文 `:` 分隔
2. 多个属性之间使用英文 `,` 分隔
3. 属性就是依附在对象上的变量
4. 属性名可以使用 `""` 或 `''`，一般情况下省略，除非名称遇到特殊符号如空格、中横线等

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>JavaScript 基础 - 对象语法</title>
</head>
<body>

  <script>
    // 通过对象描述一个人的数据信息
    // person 是一个对象，它包含了一个属性 name
    // 属性都是成对出现的，属性名 和 值，它们之间使用英文 : 分隔
```

```
    let person = {
      name: '小明', // 描述人的姓名
      age: 18, // 描述人的年龄
      stature: 185, // 描述人的身高
      gender: '男', // 描述人的性别
    }
     // 访问人的名字
    console.log(person.name) // 结果为 小明
    // 访问人性别
    console.log(person.gender) // 结果为 男
    // 访问人的身高
    console.log(person['stature']) // 结果为 185
   // 或者
    console.log(person.stature) // 结果同为 185
  </script>
</body>
</html>
```

动态添加:

```
<script>
   // 声明一个空的对象（没有任何属性）
   let user = {}
   // 动态追加属性
   user.name = '小明'
   user['age'] = 18

   // 动态添加与直接定义是一样的，只是语法上更灵活
 </script>
```

3.方法调用

1. 方法是由方法名和函数两部分构成，它们之间使用 : 分隔
2. 多个属性之间使用英文 `,` 分隔
3. 方法是依附在对象中的函数
4. 方法名可以使用 `""` 或 `''`，一般情况下省略，除非名称遇到特殊符号如空格、中横线等

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>JavaScript 基础 - 对象方法</title>
</head>
<body>

  <script>
    // 方法是依附在对象上的函数
    let person = {
      name: '小红',
      age: 18,
      // 方法是由方法名和函数两部分构成，它们之间使用 : 分隔
```

```
      singing: function () {
        console.log('两只老虎，两只老虎，跑的快，跑的快...')
      },
      run: function () {
        console.log('我跑的非常快...')
      }
    }



    // 调用对象中 singing 方法
    person.singing()
    // 调用对象中的 run 方法
    person.run()
  </script>
</body>
</html>
```

动态添加：

```
  <script>
    // 声明一个空的对象（没有任何属性，也没有任何方法）
    let user = {}
    // 动态追加属性
    user.name = '小明'
    user.['age'] = 18

    // 动态添加方法
    user.move = function () {
      console.log('移动一点距离...')
    }

  </script>
```

4.null

null 也是 JavaScript 中数据类型的一种，通常只用它来表示不存在的对象。使用 typeof 检测类型它的类型时，结果为 `object`。

5.遍历对象

```
let obj = {
    uname: 'pink'
}
for(let k in obj) {
    // k 属性名  字符串  带引号    obj.'uname'     k ===  'uname'
    // obj[k]  属性值    obj['uname']   obj[k]
}
```

for in 不提倡遍历数组 因为 k 是 字符串

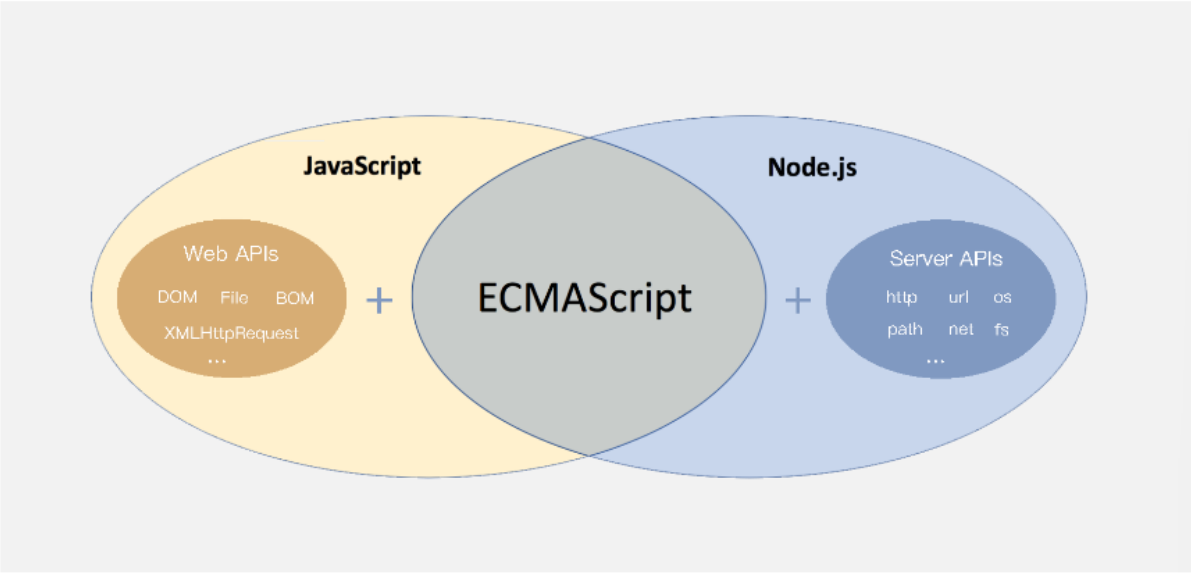
6.内置对象

`Math` 是 JavaScript 中内置的对象，称为数学对象，这个对象下即包含了属性，也包含了许多的方法。

```
// 圆周率
console.log(Math.PI);
// 0 ~ 1 之间的随机数，包含 0 不包含 1
Math.random()
// 舍弃小数部分，整数部分加1，数字向上取整
Math.ceil(3.4)
// 舍弃小数部分，整数部分不变，数字向下取整
Math.floor(4.68)
// 取整，四舍五入原则
Math.round(5.46539)
Math.round(4.849)
// 找出最大值
Math.max(10, 21, 7, 24, 13)
// 找出最小值
Math.min(24, 18, 6, 19, 21)
// 求某个数的多少次方
Math.pow(4, 2) // 求 4 的 2 次方
Math.pow(2, 3) // 求 2 的 3 次方
// 求某数的平方根
Math.sqrt(16)

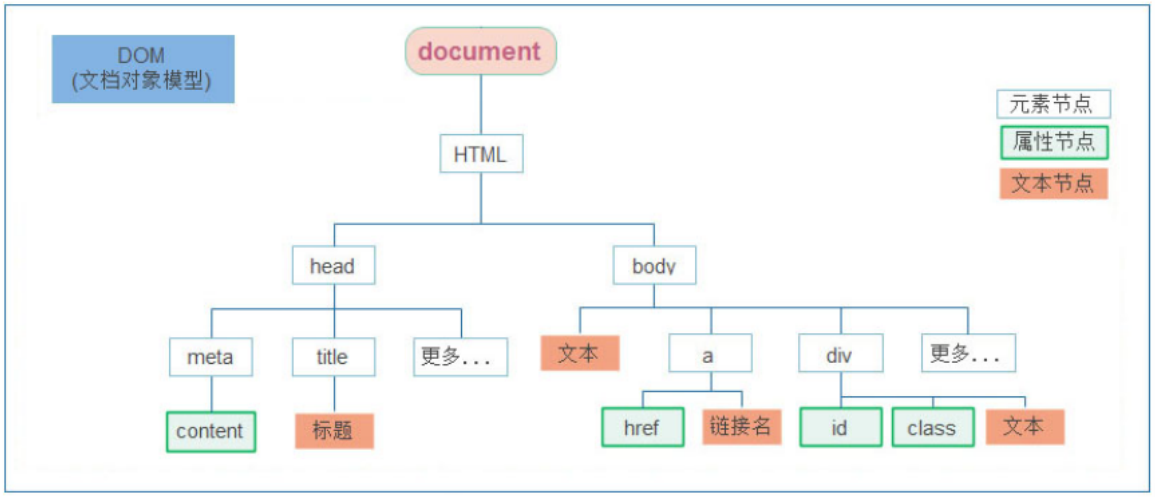
```

WebApis（DOM&BOM）



1.基本概念

如下图所示，将 HTML 文档以树状结构直观的表现出来，我们称之为文档树或 DOM 树，**文档树直观的体现了标签与标签之间的关系。**



DOM 节点

节点是文档树的组成部分，**每一个节点都是一个 DOM 对象**，主要分为元素节点、属性节点、文本节点等。

1. 【元素节点】其实就是 HTML 标签，如上图中 `head`、`div`、`body` 等都属于元素节点。
2. 【属性节点】是指 HTML 标签中的属性，如上图中 `a` 标签的 `href` 属性、`div` 标签的 `class` 属性。
3. 【文本节点】是指 HTML 标签的文字内容，如 `title` 标签中的文字。
4. 【根节点】特指 `html` 标签。
5. 其它...

document

`document` 是 JavaScript 内置的专门用于 DOM 的对象，该对象包含了若干的属性和方法，`document` 是学习 DOM 的核心。

```
<script>
  // document 是内置的对象
  // console.log(typeof document);

  // 1. 通过 document 获取根节点
  console.log(document.documentElement); // 对应 html 标签

  // 2. 通过 document 节取 body 节点
  console.log(document.body); // 对应 body 标签

  // 3. 通过 document.write 方法向网页输出内容
  document.write('Hello World!');
</script>
```

上述列举了 `document` 对象的部分属性和方法，我们先对 `document` 有一个整体的认识。

2.获取DOM对象

1. querySelector 满足条件的第一个元素
2. querySelectorAll 满足条件的元素集合 返回伪数组
3. 了解其他方式
    1. getElementById
    2. getElementsByTagName

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>DOM - 查找节点</title>
</head>
<body>
  <h3>查找元素类型节点</h3>
  <p>从整个 DOM 树中查找 DOM 节点是学习 DOM 的第一个步骤。</p>
  <ul>
      <li>元素</li>
      <li>元素</li>
      <li>元素</li>
```

```
      <li>元素</li>
  </ul>
  <script>
    const p = document.querySelector('p')  // 获取第一个p元素
    const lis = document.querySelectorAll('li')  // 获取第一个p元素
  </script>
</body>
</html>
```

总结：

- document.getElementById 专门获取元素类型节点，根据标签的 `id` 属性查找
- 任意 DOM 对象都包含 nodeType 属性，用来检检测节点类型

3.操作元素内容

通过修改 DOM 的文本内容，动态改变网页的内容。

1.`innerText` 将文本内容添加/更新到任意标签位置，**文本中包含的标签不会被解析。**

```
<script>
  // innerText 将文本内容添加/更新到任意标签位置
  const intro = document.querySelector('.intro')
   intro.innerText = '嗨~ 我叫李雷！'
  // intro.innerText = '<h4>嗨~ 我叫李雷！</h4>'
</script>
```

2.`innerHTML` 将文本内容添加/更新到任意标签位置，**文本中包含的标签会被解析。**

```
<script>
  // innerHTML 将文本内容添加/更新到任意标签位置
  const intro = document.querySelector('.intro')
  intro.innerHTML = '嗨~ 我叫韩梅梅！'
  intro.innerHTML = '<h4>嗨~ 我叫韩梅梅！</h4>'
</script>
```

总结：如果文本内容中包含 `html` 标签时推荐使用 `innerHTML`，否则建议使用 `innerText` 属性。(一个识别标签一个不识别标签)

控制样式属性

1.应用【修改样式】，通过修改行内样式 `style` 属性，实现对样式的动态修改。

通过元素节点获得的 `style` 属性本身的数据类型也是对象，如 `box.style.color`、`box.style.width` 分别用来获取元素节点 CSS 样式的 `color` 和 `width` 的值。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>练习 - 修改样式</title>
```

```
</head>
<body>
  <div class="box">随便一些文本内容</div>
  <script>
    // 获取 DOM 节点
    const box = document.querySelector('.intro')
    box.style.color = 'red'
    box.style.width = '300px'
    // css 属性的 - 连接符与 JavaScript 的 减运算符
    // 冲突，所以要改成驼峰法
    box.style.backgroundColor = 'pink'
  </script>
</body>
</html>
```

任何标签都有 `style` 属性，通过 `style` 属性可以动态更改网页标签的样式，如要遇到 `css` 属性中包含字符 `-` 时，要将 `-` 去掉并将其后面的字母改成大写，如 `background-color` 要写成 `box.style.backgroundColor`

2. 操作类名(className) 操作CSS

   如果修改的样式比较多，直接通过style属性修改比较繁琐，我们可以通过借助于css类名的形式。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>练习 - 修改样式</title>
    <style>
        .pink {
            background: pink;
            color: hotpink;
        }
    </style>
</head>
<body>
  <div class="box">随便一些文本内容</div>
  <script>
    // 获取 DOM 节点
    const box = document.querySelector('.intro')
    box.className = 'pink'
  </script>
</body>
</html>

//注意:
1.由于class是关键字, 所以使用className去代替
2.className是使用新值换旧值, 如果需要添加一个类,需要保留之前的类名
```

3.通过 classList 操作类控制CSS

为了解决className 容易覆盖以前的类名，我们可以通过classList方式追加和删除类名

```
<!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>
        div {
            width: 200px;
            height: 200px;
            background-color: pink;
        }

        .active {
            width: 300px;
            height: 300px;
            background-color: hotpink;
            margin-left: 100px;
        }
    </style>
</head>

<body>

    <div class="one"></div>
    <script>
        // 1.获取元素
        // let box = document.querySelector('css选择器')
        let box = document.querySelector('div')
        // add是个方法 添加  追加
        // box.classList.add('active')
        // remove() 移除 类
        // box.classList.remove('one')
        // 切换类，有就删掉，没有就加上
        box.classList.toggle('one')
    </script>
</body>

</html>
```

操作表单元素属性

表单很多情况，也需要修改属性，比如点击眼睛，可以看到密码，本质是把表单类型转换为文本框

正常的有属性有取值的跟其他的标签属性没有任何区别

获取:DOM对象.属性名

设置:DOM对象.属性名= 新值

```
<!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>

</head>

<body>
    <input type="text" value="请输入">
    <button disabled>按钮</button>
    <input type="checkbox" name="" id="" class="agree">
    <script>
        // 1. 获取元素
        let input = document.querySelector('input')
        // 2. 取值或者设置值  得到input里面的值可以用 value
        // console.log(input.value)
        input.value = '小米手机'
        input.type = 'password'

        // 2. 启用按钮
        let btn = document.querySelector('button')
        // disabled 不可用   =  false  这样可以让按钮启用
        btn.disabled = false
        // 3. 勾选复选框
        let checkbox = document.querySelector('.agree')
        checkbox.checked = false
    </script>
</body>

</html>
```

自定义属性

标准属性: 标签天生自带的属性 比如class id title等, 可以直接使用点语法操作比如： disabled、checked、selected

自定义属性：

在html5中推出来了专门的data-自定义属性

在标签上一律以data-开头

在DOM对象上一律以**dataset**对象方式获取

```
<!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>

```

```
</head>

<body>
   <div data-id="1"> 自定义属性 </div>
    <script>
        // 1. 获取元素
        let div = document.querySelector('div')
        // 2. 获取自定义属性值
         console.log(div.dataset.id)

    </script>
</body>

</html>
```

4.间歇函数

`setInterval` 是 JavaScript 中内置的函数，它的作用是间隔固定的时间自动重复执行另一个函数，也叫定时器函数。

```
<script>
  // 1. 定义一个普通函数
  function repeat() {
    console.log('不知疲倦的执行下去....')
  }

  // 2. 使用 setInterval 调用 repeat 函数
  // 间隔 1000 毫秒，重复调用 repeat
  setInterval(repeat, 1000)
</script>
```

5.随机函数

```
<script>
  // 1. 取到N-M的随机整数
  function getRandom（N,M）{
  return Math.floor（Math.random()*(M-N+1)）+N
  }
  const random = getRandom(1,6)//在1,6中获取随机数


</script>
```

6.事件

事件是编程语言中的术语，它是用来描述程序的行为或状态的，**一旦行为或状态发生改变，便立即调用一个函数。**

6.1事件监听

结合 DOM 使用事件时，需要为 DOM 对象添加事件监听，等待事件发生（触发）时，便立即调用一个函数。

`addEventListener` 是 DOM 对象专门用来添加事件监听的方法，它的两个参数分别为【事件类型】和【事件回调】。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>事件监听</title>
</head>
<body>
  <h3>事件监听</h3>
  <p id="text">为 DOM 元素添加事件监听，等待事件发生，便立即执行一个函数。</p>
  <button id="btn">点击改变文字颜色</button>
  <script>
    // 1. 获取 button 对应的 DOM 对象
    const btn = document.querySelector('#btn')

    // 2. 添加事件监听
    btn.addEventListener('click', function () {
      console.log('等待事件被触发...')
      // 改变 p 标签的文字颜色
      let text = document.getElementById('text')
      text.style.color = 'red'
    })

    // 3. 只要用户点击了按钮，事件便触发了！！！
  </script>
</body>
</html>
```

完成事件监听分成3个步骤：

1. 获取 DOM 元素
2. 通过 `addEventListener` 方法为 DOM 节点添加事件监听
3. 等待事件触发，如用户点击了某个按钮时便会触发 `click` 事件类型
4. 事件触发后，相对应的回调函数会被执行

大白话描述：所谓的事件无非就是找个机会（事件触发）调用一个函数（回调函数）。

6.2事件类型

`click` 译成中文是【点击】的意思，它的含义是监听（等着）用户鼠标的单击操作，除了【单击】还有【双击】`dblclick`

```
<script>
  // 双击事件类型
  btn.addEventListener('dblclick', function () {
    console.log('等待事件被触发...');
    // 改变 p 标签的文字颜色
    const text = document.querySelector('.text')
    text.style.color = 'red'
  })

  // 只要用户双击击了按钮，事件便触发了！！！
</script>
```

结论：【事件类型】决定了事件被触发的方式，如 `click` 代表鼠标单击，`dblclick` 代表鼠标双击。

6.3.事件处理程序

`addEventListener` 的第2个参数是函数，这个函数会在事件被触发时立即被调用，在这个函数中可以编写任意逻辑的代码，如改变 DOM 文本颜色、文本内容等。

```
<script>
  // 双击事件类型
  btn.addEventListener('dblclick', function () {
    console.log('等待事件被触发...')

    const text = document.querySelector('.text')
    // 改变 p 标签的文字颜色
    text.style.color = 'red'
    // 改变 p 标签的文本内容
    text.style.fontSize = '20px'
  })
</script>
```

结论：【事件处理程序】决定了事件触发后应该执行的逻辑。

6.4事件类型

鼠标事件

鼠标事件是指跟鼠标操作相关的事件，如单击、双击、移动等。

1.`mouseenter 监听鼠标是否移入 DOM 元素

```
<body>
  <h3>鼠标事件</h3>
  <p>监听与鼠标相关的操作</p>
  <hr>
  <div class="box"></div>
  <script>
    // 需要事件监听的 DOM 元素
    const box = document.querySelector('.box');

```

```
    // 监听鼠标是移入当前 DOM 元素
    box.addEventListener('mouseenter', function () {
      // 修改文本内容
      this.innerText = '鼠标移入了...';
      // 修改光标的风格
      this.style.cursor = 'move';
    })
  </script>
</body>
```

2.`mouseleave 监听鼠标是否移出 DOM 元素

```
<body>
  <h3>鼠标事件</h3>
  <p>监听与鼠标相关的操作</p>
  <hr>
  <div class="box"></div>
  <script>
    // 需要事件监听的 DOM 元素
    const box = document.querySelector('.box');

    // 监听鼠标是移出当前 DOM 元素
    box.addEventListener('mouseleave', function () {
      // 修改文本内容
      this.innerText = '鼠标移出了...';
    })
  </script>
</body>
```

鼠标经过事件:

mouseover 和 mouseout 会有冒泡效果

mouseenter 和 mouseleave 没有冒泡效果 (推荐)

键盘事件

keydown 键盘按下触发
keyup 键盘抬起触发

焦点事件

focus 获得焦点

blur 失去焦点

文本框输入事件

input

6.5事件对象

任意事件类型被触发时与事件相关的信息会被以对象的形式记录下来，我们称这个对象为事件对象。

```
<body>
  <h3>事件对象</h3>
  <p>任意事件类型被触发时与事件相关的信息会被以对象的形式记录下来，我们称这个对象为事件对象。</p>
  <hr>
  <div class="box"></div>
  <script>
    // 获取 .box 元素
    const box = document.querySelector('.box')

    // 添加事件监听
    box.addEventListener('click', function (e) {
      console.log('任意事件类型被触发后，相关信息会以对象形式被记录下来...');

      // 事件回调函数的第1个参数即所谓的事件对象
      console.log(e)
    })
  </script>
</body>
```

事件回调函数的【第1个参数】即所谓的事件对象，通常习惯性的将这个对数命名为 `event`、`ev` 、`ev` 。

接下来简单看一下事件对象中包含了哪些有用的信息：

1. `ev.type` 当前事件的类型
2. `ev.clientX/Y` 光标相对浏览器窗口的位置
3. `ev.offsetX/Y` 光标相于当前 DOM 元素的位置

注：在事件回调函数内部通过 window.event 同样可以获取事件对象。

6.6环境对象

环境对象指的是函数内部特殊的变量 `this` ，它代表着当前函数运行时所处的环境。

```
<script>
  // 声明函数
  function sayHi() {
    // this 是一个变量
    console.log(this);
  }

  // 声明一个对象
  let user = {
    name: '张三',
    sayHi: sayHi // 此处把 sayHi 函数，赋值给 sayHi 属性
  }
```

```
  let person = {
    name: '李四',
    sayHi: sayHi
  }

  // 直接调用
  sayHi() // window
  window.sayHi() // window

  // 做为对象方法调用
  user.sayHi()// user
    person.sayHi()// person
</script>
```

结论：

1. `this` 本质上是一个变量，数据类型为对象
2. 函数的调用方式不同 `this` 变量的值也不同
3. 【谁调用 `this` 就是谁】是判断 `this` 值的粗略规则
4. 函数直接调用时实际上 `window.sayHi()` 所以 `this` 的值为 `window`

6.7回调函数

如果将函数 A 做为参数传递给函数 B 时，我们称函数 A 为回调函数。

```
<script>
  // 声明 foo 函数
  function foo(arg) {
    console.log(arg);
  }

  // 普通的值做为参数
  foo(10);
  foo('hello world!');
  foo(['html', 'css', 'javascript']);

  function bar() {
    console.log('函数也能当参数...');
  }
  // 函数也可以做为参数！！！！
  foo(bar);
</script>
```

函数 `bar` 做参数传给了 `foo` 函数，`bar` 就是所谓的回调函数了！！！

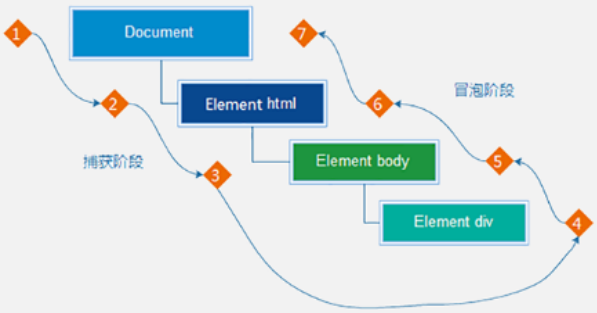
```
<script>
  // 调用定时器，匿名函数做为参数
  setInterval(function () {
    console.log('我是回调函数...');
  }, 1000);
</script>
```

结论：

1. 回调函数本质还是函数，只不过把它当成参数使用
2. 使用匿名函数做为回调函数比较常见

6.8事件流

事件流是对事件执行过程的描述，了解事件的执行过程有助于加深对事件的理解，提升开发实践中对事件运用的灵活度。



如上图所示，任意事件被触发时总会经历两个阶段：【捕获阶段】和【冒泡阶段】。

简言之，捕获阶段是【从父到子】的传导过程，冒泡阶段是【从子向父】的传导过程。

捕获和冒泡

```
<body>
  <h3>事件流</h3>
  <p>事件流是事件在执行时的底层机制，主要体现在父子盒子之间事件的执行上。</p>
  <div class="outer">
    <div class="inner">
      <div class="child"></div>
    </div>
  </div>
  <script>
    // 获取嵌套的3个节点
    const outer = document.querySelector('.outer');
    const inner = document.querySelector('.inner');
    const child = document.querySelector('.child');

    // html 元素添加事件
    document.documentElement.addEventListener('click', function () {
      console.log('html...')
    })

    // body 元素添加事件
    document.body.addEventListener('click', function () {
```

```
      console.log('body...')
    })

    // 外层的盒子添加事件
    outer.addEventListener('click', function () {
      console.log('outer...')
    })

    // 中间的盒子添加事件
    outer.addEventListener('click', function () {
      console.log('inner...')
    })

    // 内层的盒子添加事件
    outer.addEventListener('click', function () {
      console.log('child...')
    })
  </script>
</body>
```

执行上述代码后发现，当单击事件触发时，其祖先元素的单击事件也【相继触发】，这是为什么呢?

结合事件流的特征，我们知道当某个元素的事件被触发时，事件总是会先经过其祖先才能到达当前元素，然后再由当前元素向祖先传递，事件在流动的过程中遇到相同的事件便会被触发。

再来关注一个细节就是事件相继触发的【执行顺序】，事件的执行顺序是可控制的，即可以在捕获阶段被执行，也可以在冒泡阶段被执行。

如果事件是在冒泡阶段执行的，我们称为冒泡模式，它会先执行子盒子事件再去执行父盒子事件，默认是冒泡模式。

如果事件是在捕获阶段执行的，我们称为捕获模式，它会先执行父盒子事件再去执行子盒子事件。

```
<body>
  <h3>事件流</h3>
  <p>事件流是事件在执行时的底层机制，主要体现在父子盒子之间事件的执行上。</p>
  <div class="outer">
    <div class="inner"></div>
  </div>
  <script>
    // 获取嵌套的3个节点
    const outer = document.querySelector('.outer')
    const inner = document.querySelector('.inner')

    // 外层的盒子
    outer.addEventListener('click', function () {
      console.log('outer...')
    }, true) // true 表示在捕获阶段执行事件

    // 中间的盒子
    outer.addEventListener('click', function () {
      console.log('inner...')
    }, true)
  </script>
</body>
```

结论：

1. `addEventListener` 第3个参数决定了事件是在捕获阶段触发还是在冒泡阶段触发
2. `addEventListener` 第3个参数为 `true` 表示捕获阶段触发，`false` 表示冒泡阶段触发，默认值为 `false`
3. 事件流只会在父子元素具有相同事件类型时才会产生影响
4. 绝大部分场景都采用默认的冒泡模式（其中一个原因是早期 IE 不支持捕获）

阻止冒泡

阻止冒泡是指阻断事件的流动，保证事件只在当前元素被执行，而**不再去影响到其对应的祖先元素。**

```
<body>
  <h3>阻止冒泡</h3>
  <p>阻止冒泡是指阻断事件的流动，保证事件只在当前元素被执行，而不再去影响到其对应的祖先元素。</p>
  <div class="outer">
    <div class="inner">
      <div class="child"></div>
    </div>
  </div>
  <script>
    // 获取嵌套的3个节点
    const outer = document.querySelector('.outer')
    const inner = document.querySelector('.inner')
    const child = document.querySelector('.child')

    // 外层的盒子
    outer.addEventListener('click', function () {
      console.log('outer...')
    })

    // 中间的盒子
    inner.addEventListener('click', function (ev) {
      console.log('inner...')

      // 阻止事件冒泡
      ev.stopPropagation()
    })

    // 内层的盒子
    child.addEventListener('click', function (ev) {
      console.log('child...')

      // 借助事件对象，阻止事件向上冒泡
      ev.stopPropagation()
    })
  </script>
</body>
```

结论：事件对象中的 `ev.stopPropagation` 方法，专门用来阻止事件冒泡。

6.9事件委托

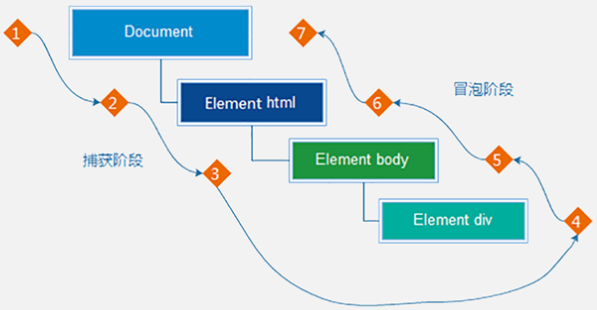
事件委托是利用事件流的特征解决一些现实开发需求的知识技巧，主要的作用是提升程序效率。

利用事件流的特征，可以对上述的代码进行优化，**事件的的冒泡模式总是会将事件流向其父元素的，如果父元素监听了相同的事件类型，那么父元素的事件就会被触发并执行**，正是利用这一特征对上述代码进行优化，如下代码所示：

```
<script>
  // 假设页面中有 10000 个 button 元素
  let buttons = document.querySelectorAll('table button');

  // 假设上述的 10000 个 buttom 元素共同的祖先元素是 table
  let parents = document.querySelector('table');
  parents.addEventListener('click', function () {
    console.log('点击任意子元素都会触发事件...');
  })
</script>
```

我们的最终目的是保证只有点击 button 子元素才去执行事件的回调函数，如何判断用户点击是哪一个子元素呢?



事件对象中的属性 `target` 或 `srcElement` 属性表示真正触发事件的元素，它是一个元素类型的节点。

```
<script>
  // 假设页面中有 10000 个 button 元素
  const buttons = document.querySelectorAll('table button')

  // 假设上述的 10000 个 buttom 元素共同的祖先元素是 table
  const parents = document.querySelector('table')
  parents.addEventListener('click', function (ev) {
    // console.log(ev.target);
    // 只有 button 元素才会真正去执行逻辑
    if(ev.target.tagName === 'BUTTON') {
      // 执行的逻辑
    }
  })
</script>
```

6.10其他事件

页面加载事件

加载外部资源（如图片、外联CSS和JavaScript等）加载完毕时触发的事件

有些时候需要等页面资源全部处理完了做一些事情

事件名：load

监听页面所有资源加载完毕：

```
window.addEventListener('load', function() {
    // xxxxx
})
```

元素滚动事件

滚动条在滚动的时候持续触发的事件

```
window.addEventListener('scroll', function() {
    // xxxxx
})
```

页面尺寸事件

会在窗口尺寸改变的时候触发事件:

```
window.addEventListener('resize', function() {
    // xxxxx
})
```

元素尺寸与位置

获取元素的自身宽高、包含元素自身设置的宽高、padding、border

offsetWidth和offsetHeight

获取出来的是数值,方便计算

注意: 获取的是可视宽高, 如果盒子是隐藏的,获取的结果是0

7.日期对象

掌握 Date 日期对象的使用，动态获取当前计算机的时间。

ECMAScript 中内置了获取系统时间的对象 Date，使用 Date 时与之前学习的内置对象 console 和 Math 不同，它需要借助 new 关键字才能使用。

实例化

```
// 1. 实例化
// const date = new Date(); // 系统默认时间
const date = new Date('2020-05-01') // 指定时间
// date 变量即所谓的时间对象

console.log(typeof date)


```

方法

```
 // 1. 实例化
const date = new Date();
// 2. 调用时间对象方法
// 通过方法分别获取年、月、日，时、分、秒
const year = date.getFullYear(); // 四位年份
const month = date.getMonth(); // 0 ~ 11
```

getFullYear 获取四位年份

getMonth 获取月份，取值为 0 ~ 11

getDate 获取月份中的每一天，不同月份取值也不相同

getDay 获取星期，取值为 0 ~ 6

getHours 获取小时，取值为 0 ~ 23

getMinutes 获取分钟，取值为 0 ~ 59

getSeconds 获取秒，取值为 0 ~ 59

时间戳

时间戳是指1970年01月01日00时00分00秒起至现在的总秒数或毫秒数，它是一种特殊的计量时间的方式。

注：ECMAScript 中时间戳是以毫秒计的。

```
    // 1. 实例化
  const date = new Date()
  // 2. 获取时间戳
  console.log(date.getTime())
// 还有一种获取时间戳的方法
  console.log(+new Date())
  // 还有一种获取时间戳的方法
  console.log(Date.now())

```

获取时间戳的方法，分别为 getTime 和 Date.now 和 +new Date()

8.DOM 节点

> 掌握元素节点创建、复制、插入、删除等操作的方法，能够依据元素节点的结构关系查找节点

回顾之前 DOM 的操作都是针对元素节点的属性或文本的，除此之外也有专门针对元素节点本身的操作，如插入、复制、删除、替换等。

插入节点

在已有的 DOM 节点中插入新的 DOM 节点时，需要关注两个关键因素：首先要得到新的 DOM 节点，其次在哪个位置插入这个节点。

如下代码演示：

```
<body>
  <h3>插入节点</h3>
  <p>在现有 dom 结构基础上插入新的元素节点</p>
  <hr>
  <!-- 普通盒子 -->
  <div class="box"></div>
  <!-- 点击按钮向 box 盒子插入节点 -->
  <button class="btn">插入节点</button>
  <script>
    // 点击按钮，在网页中插入节点
    const btn = document.querySelector('.btn')
    btn.addEventListener('click', function () {
      // 1. 获得一个 DOM 元素节点
      const p = document.createElement('p')
      p.innerText = '创建的新的p标签'
      p.className = 'info'

      // 复制原有的 DOM 节点
      const p2 = document.querySelector('p').cloneNode(true)
      p2.style.color = 'red'

      // 2. 插入盒子 box 盒子
      document.querySelector('.box').appendChild(p)
      document.querySelector('.box').appendChild(p2)
    })
  </script>
</body>
```

结论：

- `createElement` 动态创建任意 DOM 节点
- `cloneNode` 复制现有的 DOM 节点，传入参数 true 会复制所有子节点
- `appendChild` 在末尾（结束标签前）插入节点

再来看另一种情形的代码演示：

```
<body>
  <h3>插入节点</h3>
  <p>在现有 dom 结构基础上插入新的元素节点</p>
    <hr>
  <button class="btn1">在任意节点前插入</button>
  <ul>
    <li>HTML</li>
    <li>CSS</li>
    <li>JavaScript</li>
  </ul>
  <script>
```

```
    // 点击按钮，在已有 DOM 中插入新节点
    const btn1 = document.querySelector('.btn1')
    btn1.addEventListener('click', function () {

      // 第 2 个 li 元素
      const relative = document.querySelector('li:nth-child(2)')

      // 1. 动态创建新的节点
      const li1 = document.createElement('li')
      li1.style.color = 'red'
      li1.innerText = 'Web APIs'

      // 复制现有的节点
      const li2 = document.querySelector('li:first-child').cloneNode(true)
      li2.style.color = 'blue'

      // 2. 在 relative 节点前插入
      document.querySelector('ul').insertBefore(li1, relative)
      document.querySelector('ul').insertBefore(li2, relative)
    })
  </script>
</body>
```

结论：

- `createElement` 动态创建任意 DOM 节点
- `cloneNode` 复制现有的 DOM 节点，传入参数 true 会复制所有子节点
- `insertBefore` 在父节点中任意子节点之前插入新节点

删除节点

删除现有的 DOM 节点，也需要关注两个因素：首先由父节点删除子节点，其次是要删除哪个子节点。

```
<body>
  <!-- 点击按钮删除节点 -->
  <button>删除节点</button>
  <ul>
    <li>HTML</li>
    <li>CSS</li>
    <li>Web APIs</li>
  </ul>

  <script>
    const btn = document.querySelector('button')
    btn.addEventListener('click', function () {
      // 获取 ul 父节点
      let ul = document.querySelector('ul')
      // 待删除的子节点
      let lis = document.querySelectorAll('li')

      // 删除节点
      ul.removeChild(lis[0])
    })
  </script>
```

```
</body>
```

结论：`removeChild` 删除节点时一定是由父子关系。

查找节点

DOM 树中的任意节点都不是孤立存在的，它们要么是父子关系，要么是兄弟关系，不仅如此，我们可以依据节点之间的关系查找节点。

父子关系

```
<body>
  <button class="btn1">所有的子节点</button>
  <!-- 获取 ul 的子节点 -->
  <ul>
    <li>HTML</li>
    <li>CSS</li>
    <li>JavaScript 基础</li>
    <li>Web APIs</li>
  </ul>
  <script>
    const btn1 = document.querySelector('.btn1')
    btn1.addEventListener('click', function () {
      // 父节点
      const ul = document.querySelector('ul')

      // 所有的子节点
      console.log(ul.childNodes)
      // 只包含元素子节点
      console.log(ul.children)
    })
  </script>
</body>
```

结论：

- `childNodes` 获取全部的子节点，回车换行会被认为是空白文本节点
- `children` 只获取元素类型节点

```
<body>
  <table>
    <tr>
      <td width="60">序号</td>
      <td>课程名</td>
      <td>难度</td>
      <td width="80">操作</td>
    </tr>
    <tr>
      <td>1</td>
      <td><span>HTML</span></td>
      <td>初级</td>
      <td><button>变色</button></td>
    </tr>
    <tr>
```

```
      <td>2</td>
      <td><span>CSS</span></td>
      <td>初级</td>
      <td><button>变色</button></td>
    </tr>
    <tr>
      <td>3</td>
      <td><span>Web APIs</span></td>
      <td>中级</td>
      <td><button>变色</button></td>
    </tr>
  </table>
  <script>
    // 获取所有 button 节点，并添加事件监听
    const buttons = document.querySelectorAll('table button')
    for(let i = 0; i < buttons.length; i++) {
      buttons[i].addEventListener('click', function () {
        // console.log(this.parentNode); // 父节点 td
        // console.log(this.parentNode.parentNode); // 爷爷节点 tr
        this.parentNode.parentNode.style.color = 'red'
      })
    }
  </script>
</body>
```

结论：`parentNode` 获取父节点，以相对位置查找节点，实际应用中非常灵活。

兄弟关系

```
<body>
  <ul>
    <li>HTML</li>
    <li>CSS</li>
    <li>JavaScript 基础</li>
    <li>Web APIs</li>
  </ul>
  <script>
    // 获取所有 li 节点
    const lis = document.querySelectorAll('ul li')

    // 对所有的 li 节点添加事件监听
    for(let i = 0; i < lis.length; i++) {
      lis[i].addEventListener('click', function () {
        // 前一个节点
        console.log(this.previousSibling)
        // 下一下节点
        console.log(this.nextSibling)
      })
    }
  </script>
</body>
```

结论：

- `previousSibling` 获取前一个节点，以相对位置查找节点，实际应用中非常灵活。

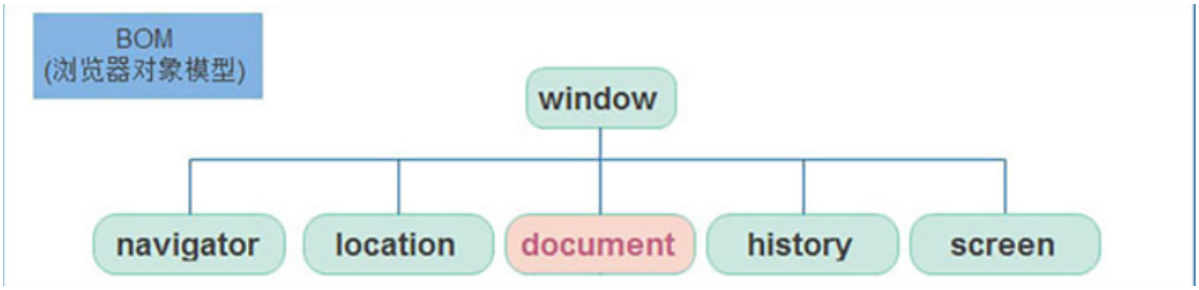
- `nextSibling` 获取后一个节点，以相对位置查找节点，实际应用中非常灵活。

9.BOM

9.1window对象

BOM (Browser Object Model) 是浏览器对象模型

- window对象是一个全局对象，也可以说是JavaScript中的顶级对象
- 像document、alert()、console.log()这些都是window的属性，基本BOM的属性和方法都是window的
- 所有通过var定义在全局作用域中的变量、函数都会变成window对象的属性和方法
- window对象下的属性和方法调用的时候可以省略window



9.2定时器-延迟函数

JavaScript 内置的一个用来让代码延迟执行的函数，叫 setTimeout

语法：

```
setTimeout(回调函数， 延迟时间)
```

setTimeout 仅仅只执行一次，所以可以理解为就是把一段代码延迟执行, 平时省略window

间歇函数 setInterval : 每隔一段时间就执行一次， , 平时省略window

清除延时函数：

```
clearTimeout(timerId)
```

> 注意点
>
> 1. 延时函数需要等待,所以后面的代码先执行
> 2. 返回值是一个正整数，表示定时器的编号

```
<body>
  <script>
    // 定时器之延迟函数

    // 1. 开启延迟函数
    let timerId = setTimeout(function () {
```

```
      console.log('我只执行一次')
    }, 3000)

    // 1.1 延迟函数返回的还是一个正整数数字，表示延迟函数的编号
    console.log(timerId)

    // 1.2 延迟函数需要等待时间，所以下面的代码优先执行

    // 2. 关闭延迟函数
    clearTimeout(timerId)

  </script>
</body>
```

9.3location对象

location (地址) 它拆分并保存了 URL 地址的各个组成部分， 它是一个对象

属性/方法	说明
href	属性，获取完整的 URL 地址，赋值时用于地址的跳转
search	属性，获取地址中携带的参数，符号？ 后面部分
hash	属性，获取地址中的啥希值，符号 # 后面部分
reload()	方法，用来刷新当前页面，传入参数 true 时表示强制刷新

```
<body>
  <form>
    <input type="text" name="search"> <button>搜索</button>
  </form>
  <a href="#/music">音乐</a>
  <a href="#/download">下载</a>

  <button class="reload">刷新页面</button>
  <script>
    // location 对象
    // 1. href属性 （重点） 得到完整地址，赋值则是跳转到新地址
    console.log(location.href)
    // location.href = 'http://www.itcast.cn'

    // 2. search属性  得到 ? 后面的地址
    console.log(location.search)  // ?search=笔记本

```

```
    // 3. hash属性  得到 # 后面的地址
    console.log(location.hash)

    // 4. reload 方法  刷新页面
    const btn = document.querySelector('.reload')
    btn.addEventListener('click', function () {
      // location.reload() // 页面刷新
      location.reload(true) // 强制页面刷新 ctrl+f5
    })
  </script>
</body>
```

9.4navigator对象

navigator是对象，该对象下记录了浏览器自身的相关信息

常用属性和方法:

- 通过 userAgent 检测浏览器的版本及平台
- ```
  // 检测 userAgent（浏览器信息）
  (function () {
    const userAgent = navigator.userAgent
    // 验证是否为Android或iPhone
    const android = userAgent.match(/(Android);?[\s\/]+([\d.]+)?/)
    const iphone = userAgent.match(/(iPhone\sOS)\s([\d_]+)/)
    // 如果是Android或iPhone，则跳转至移动站点
    if (android || iphone) {
      location.href = 'http://m.itcast.cn'
    }})();
  ```

9.5histroy对象

history (历史)是对象，主要管理历史记录， 该对象与浏览器地址栏的操作相对应，如前进、后退等

history对象一般在实际开发中比较少用，但是会在一些OA 办公系统中见到。


  ```

常见方法：

history对象方法	作用
back()	可以后退功能
forward()	前进功能
go(参数)	前进后退功能 参数如果是 1 前进1个页面 如果是-1 后退1个页面

```
<body>
  <button class="back">←后退</button>
  <button class="forward">前进→</button>
  <script>
    // histroy对象

    // 1.前进
    const forward = document.querySelector('.forward')
    forward.addEventListener('click', function () {
      // history.forward()
      history.go(1)
    })
    // 2.后退
    const back = document.querySelector('.back')
    back.addEventListener('click', function () {
      // history.back()
      history.go(-1)
    })
  </script>
</body>

```

9.6本地存储

本地存储：将数据存储在本地浏览器中

常见的使用场景：

https://todomvc.com/examples/vanilla-es6/　页面刷新数据不丢失

好处：

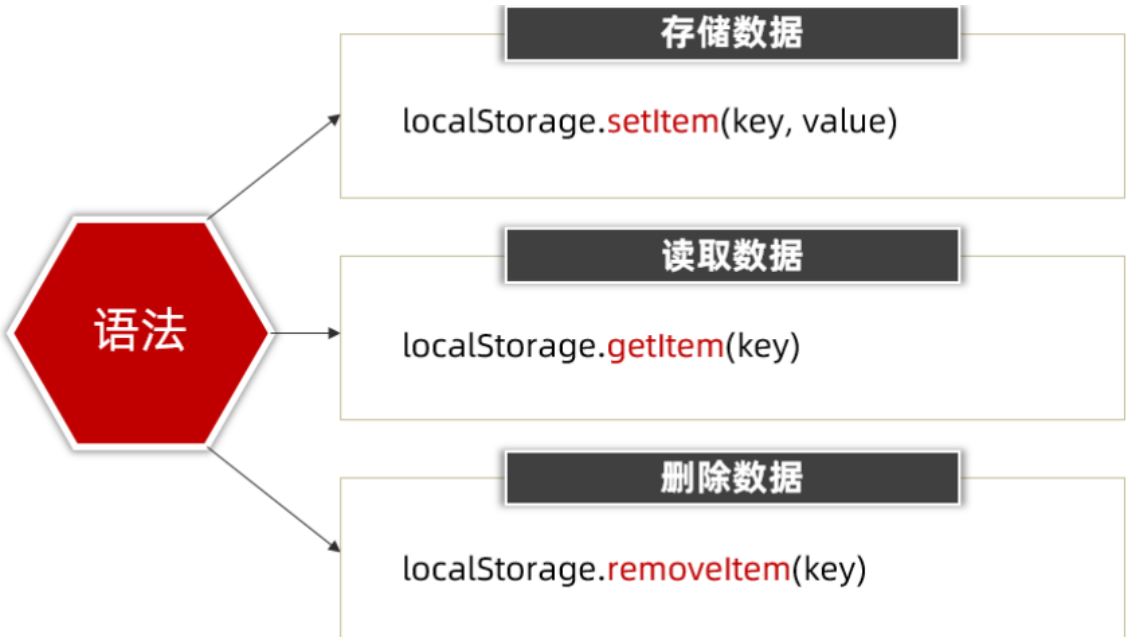
1、页面刷新或者关闭不丢失数据，实现数据持久化

2、容量较大，sessionStorage和 localStorage 约 5M 左右

localStorage（重点）

作用: 数据可以长期保留在本地浏览器中，刷新页面和关闭页面，数据也不会丢失

特性: 以键值对的形式存储，并且存储的是字符串， 省略了window




```html
<!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>本地存储-localstorage</title>
</head>

<body>
  <script>
    // 本地存储 - localstorage 存储的是字符串
    // 1. 存储
    localStorage.setItem('age', 18)

    // 2. 获取
    console.log(typeof localStorage.getItem('age'))

    // 3. 删除
    localStorage.removeItem('age')
  </script>
</body>

</html>
```

sessionStorage（了解）

特性：

- 用法跟localStorage基本相同
- 区别是：当页面浏览器被关闭时，存储在 sessionStorage 的数据会被清除

存储：sessionStorage.setItem(key,value)

获取：sessionStorage.getItem(key)

删除：sessionStorage.removeItem(key)

localStorage 存储复杂数据类型

问题：本地只能存储字符串,无法存储复杂数据类型.

解决：需要将复杂数据类型转换成 JSON字符串,在存储到本地

语法：JSON.stringify(复杂数据类型)

JSON字符串：

- 首先是1个字符串
- 属性名使用双引号引起来，不能单引号
- 属性值如果是字符串型也必须双引号

```
<body>
  <script>
    // 本地存储复杂数据类型
    const goods = {
      name: '小米',
      price: 1999
    }
    // localStorage.setItem('goods', goods)
    // console.log(localStorage.getItem('goods'))

    // 1. 把对象转换为JSON字符串  JSON.stringify
    localStorage.setItem('goods', JSON.stringify(goods))
    // console.log(typeof localStorage.getItem('goods'))

  </script>
</body>
```

问题：因为本地存储里面取出来的是字符串，不是对象，无法直接使用

解决： 把取出来的字符串转换为对象

语法：JSON.parse(JSON字符串)

```
<body>
  <script>
    // 本地存储复杂数据类型
    const goods = {
      name: '小米',
      price: 1999
    }
```

```
    // localStorage.setItem('goods', goods)
    // console.log(localStorage.getItem('goods'))

    // 1. 把对象转换为JSON字符串  JSON.stringify
    localStorage.setItem('goods', JSON.stringify(goods))
    // console.log(typeof localStorage.getItem('goods'))

    // 2. 把JSON字符串转换为对象  JSON.parse
    console.log(JSON.parse(localStorage.getItem('goods')))

  </script>
</body>
```

9.7数组map 方法

使用场景：

map 可以遍历数组处理数据，并且返回新的数组

```
<body>
  <script>
  const arr = ['red', 'blue', 'pink']
  // 1. 数组 map方法 处理数据并且 返回一个数组
   const newArr = arr.map(function (ele, index) {
    // console.log(ele)  // 数组元素
    // console.log(index) // 索引号
    return ele + '颜色'
    })
console.log(newArr)
</script>
</body>
```

map 也称为映射。映射是个术语，指两个元素的集之间元素相互“对应”的关系。

map重点在于有返回值，forEach没有返回值（undefined）

9.8数组join方法

作用：join() 方法用于把数组中的所有元素转换一个字符串

语法：

```
<body>
  <script>
    const arr = ['red', 'blue', 'pink']

    // 1. 数组 map方法 处理数据并且 返回一个数组
    const newArr = arr.map(function (ele, index) {
      // console.log(ele)  // 数组元素
      // console.log(index) // 索引号
      return ele + '颜色'
    })
    console.log(newArr)

```

```
    // 2. 数组join方法  把数组转换为字符串
    // 小括号为空则逗号分割
    console.log(newArr.join())  // red颜色,blue颜色,pink颜色
    // 小括号是空字符串，则元素之间没有分隔符
    console.log(newArr.join(''))  //red颜色blue颜色pink颜色
    console.log(newArr.join('|'))  //red颜色|blue颜色|pink颜色
  </script>
</body>
```

10.正则表达式

正则表达式（Regular Expression）是一种字符串匹配的模式（规则）

使用场景：

- 例如验证表单：手机号表单要求用户只能输入11位的数字 (匹配)
- 过滤掉页面内容中的一些敏感词(替换)，或从字符串中获取我们想要的特定部分(提取)等

10.1正则基本使用

1. 定义规则

```
const reg =  /表达式/
```

   - 其中`/  /`是正则表达式字面量
   - 正则表达式也是`对象`

2. 使用正则

- `test()方法` 用来查看正则表达式与指定的字符串是否匹配
- 如果正则表达式与指定的字符串匹配 ， 返回`true`，否则`false`

```
<body>
  <script>
    // 正则表达式的基本使用
    const str = 'web前端开发'
    // 1. 定义规则
    const reg = /web/

    // 2. 使用正则  test()
    console.log(reg.test(str))  // true  如果符合规则匹配上则返回true
    console.log(reg.test('java开发'))  // false  如果不符合规则匹配上则返回 false
  </script>
</body>
```

10.2元字符

1. **普通字符:**

- 大多数的字符仅能够描述它们本身，这些字符称作普通字符，例如所有的字母和数字。
- 普通字符只能够匹配字符串中与它们相同的字符。
- 比如，规定用户只能输入英文26个英文字母，普通字符的话 /[abcdefghijklmnopqrstuvwxyz]/

2. **元字符(特殊字符)**

- 是一些具有特殊含义的字符，可以极大提高了灵活性和强大的匹配功能。
- 比如，规定用户只能输入英文26个英文字母，换成元字符写法： /[a-z]/

10.3边界符

正则表达式中的边界符（位置符）用来提示字符所处的位置，主要有两个字符

边界符	说明
^	表示匹配行首的文本(以谁开始)
$	表示匹配行尾的文本(以谁结束)

如果 ^ 和 $ 在一起，表示必须是精确匹配

```
<body>
  <script>
    // 元字符之边界符
    // 1. 匹配开头的位置 ^
    const reg = /^web/
    console.log(reg.test('web前端'))  // true
    console.log(reg.test('前端web'))  // false
    console.log(reg.test('前端web学习'))  // false
    console.log(reg.test('we'))  // false

```

```
    // 2. 匹配结束的位置 $
    const reg1 = /web$/
    console.log(reg1.test('web前端'))  //  false
    console.log(reg1.test('前端web'))  // true
    console.log(reg1.test('前端web学习'))  // false
    console.log(reg1.test('we'))  // false

    // 3. 精确匹配 ^ $
    const reg2 = /^web$/
    console.log(reg2.test('web前端'))  //  false
    console.log(reg2.test('前端web'))  // false
    console.log(reg2.test('前端web学习'))  // false
    console.log(reg2.test('we'))  // false
    console.log(reg2.test('web'))  // true
    console.log(reg2.test('webweb'))  // flase
  </script>
</body>
```

10.4量词

量词用来设定某个模式重复次数

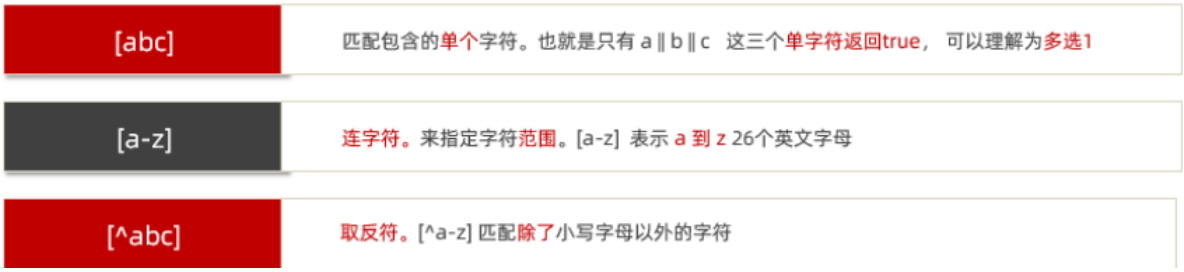
量词	说明
*	重复零次或更多次
+	重复一次或更多次
?	重复零次或一次
{n}	重复n次
{n,}	重复n次或更多次
{n,m}	重复n到m次

```
<body>
  <script>
    // 元字符之量词
    // 1. * 重复次数 >= 0 次
    const reg1 = /^w*$/
    console.log(reg1.test(''))  // true
    console.log(reg1.test('w'))  // true
    console.log(reg1.test('ww'))  // true
    console.log('-----------------------')

    // 2. + 重复次数 >= 1 次
    const reg2 = /^w+$/
    console.log(reg2.test(''))  // false
    console.log(reg2.test('w'))  // true
    console.log(reg2.test('ww'))  // true
    console.log('-----------------------')

    // 3. ? 重复次数  0 || 1
    const reg3 = /^w?$/
    console.log(reg3.test(''))  // true
```

```
    console.log(reg3.test('w'))  // true
    console.log(reg3.test('ww'))  // false
    console.log('-----------------------')


    // 4. {n} 重复 n 次
    const reg4 = /^w{3}$/
    console.log(reg4.test(''))  // false
    console.log(reg4.test('w'))  // flase
    console.log(reg4.test('ww'))  // false
    console.log(reg4.test('www'))  // true
    console.log(reg4.test('wwww'))  // false
    console.log('-----------------------')

    // 5. {n,} 重复次数 >= n
    const reg5 = /^w{2,}$/
    console.log(reg5.test(''))  // false
    console.log(reg5.test('w'))  // false
    console.log(reg5.test('ww'))  // true
    console.log(reg5.test('www'))  // true
    console.log('-----------------------')

    // 6. {n,m}   n =< 重复次数 <= m
    const reg6 = /^w{2,4}$/
    console.log(reg6.test('w'))  // false
    console.log(reg6.test('ww'))  // true
    console.log(reg6.test('www'))  // true
    console.log(reg6.test('wwww'))  // true
    console.log(reg6.test('wwwww'))  // false

    // 7. 注意事项:  逗号两侧千万不要加空格否则会匹配失败

  </script>
```

10.5范围

表示字符的范围，定义的规则限定在某个范围，比如只能是英文字母，或者数字等等，用表示范围




```
<body>
  <script>
    // 元字符之范围  []
    // 1. [abc] 匹配包含的单个字符， 多选1
    const reg1 = /^[abc]$/
    console.log(reg1.test('a'))  // true
    console.log(reg1.test('b'))  // true
```

```
    console.log(reg1.test('c'))  // true
    console.log(reg1.test('d'))  // false
    console.log(reg1.test('ab'))  // false

    // 2. [a-z] 连字符 单个
    const reg2 = /^[a-z]$/
    console.log(reg2.test('a'))  // true
    console.log(reg2.test('p'))  // true
    console.log(reg2.test('0'))  // false
    console.log(reg2.test('A'))  // false
    // 想要包含小写字母，大写字母 ，数字
    const reg3 = /^[a-zA-Z0-9]$/
    console.log(reg3.test('B'))  // true
    console.log(reg3.test('b'))  // true
    console.log(reg3.test(9))  // true
    console.log(reg3.test(','))  // flase

    // 用户名可以输入英文字母，数字，可以加下划线，要求 6~16位
    const reg4 = /^[a-zA-Z0-9_]{6,16}$/
    console.log(reg4.test('abcd1'))  // false
    console.log(reg4.test('abcd12'))  // true
    console.log(reg4.test('ABcd12'))  // true
    console.log(reg4.test('ABcd12_'))  // true

    // 3. [^a-z] 取反符
    const reg5 = /^[^a-z]$/
    console.log(reg5.test('a'))  // false
    console.log(reg5.test('A'))  // true
    console.log(reg5.test(8))  // true

  </script>
</body>
```

10.6字符类

某些常见模式的简写方式，区分字母和数字

字符类	说明
\d	匹配0-9之间的任一数字，相当于[0-9]
\D	匹配所有0-9以外的字符，相当于 [^0-9]
\w	匹配任意的字母、数字和下划线，相当于[A-Za-z0-9_]
\W	除所有字母、数字和下划线以外的字符，相当于 [^A-Za-z0-9_]
\s	匹配空格（包括换行符、制表符、空格符等），相等于[\t\r\n\v\f]
\S	匹配非空格的字符，相当于 [^ \t\r\n\v\f]

替换和修饰符

replace 替换方法，可以完成字符的替换


```
<body>
  <script>
    // 替换和修饰符
    const str = '欢迎大家学习前端，相信大家一定能学好前端，都成为前端大神'
    // 1. 替换  replace  需求：把前端替换为 web
    // 1.1 replace 返回值是替换完毕的字符串
    // const strEnd = str.replace(/前端/, 'web') 只能替换一个
  </script>
</body>
```

修饰符约束正则执行的某些细节行为，如是否区分大小写、是否支持多行匹配等

- i 是单词 ignore 的缩写，正则匹配时字母不区分大小写
- g 是单词 global 的缩写，匹配所有满足正则表达式的结果

```
<body>
  <script>
    // 替换和修饰符
    const str = '欢迎大家学习前端，相信大家一定能学好前端，都成为前端大神'
    // 1. 替换  replace  需求：把前端替换为 web
    // 1.1 replace 返回值是替换完毕的字符串
    // const strEnd = str.replace(/前端/, 'web') 只能替换一个

    // 2. 修饰符 g 全部替换
    const strEnd = str.replace(/前端/g, 'web')
    console.log(strEnd)
  </script>
</body>
```

change 事件

给input注册 change 事件，值被修改并且失去焦点后触发

判断是否有类

```
// 添加类名
元素.classList.add('类名')
// 删除类名
元素.classList.remove('类名')
// 切换类名
元素.classList.toggle('类名')
// 判断是否包含某个类名  有返回true，没有返回false
元素.classList.contains('类名')
```

元素.classList.contains() 看看有没有包含某个类，如果有则返回true，么有则返回false

JavaScript进阶

> 学习作用域、变量提升、闭包等语言特征，加深对 JavaScript 的理解，掌握变量赋值、函数声明的简洁语法，降低代码的冗余度。

- 理解作用域对程序执行的影响
- 能够分析程序执行的作用域范围
- 理解闭包本质，利用闭包创建隔离作用域
- 了解什么变量提升及函数提升
- 掌握箭头函数、解析剩余参数等简洁语法

作用域

> 了解作用域对程序执行的影响及作用域链的查找机制，使用闭包函数创建隔离作用域避免全局变量污染。

作用域（scope）规定了变量能够被访问的“范围”，离开了这个“范围”变量便不能被访问，作用域分为全局作用域和局部作用域。

局部作用域

局部作用域分为函数作用域和块作用域。

函数作用域

在函数内部声明的变量只能在函数内部被访问，外部无法直接访问。

```
<script>
  // 声明 counter 函数
  function counter(x, y) {
    // 函数内部声明的变量
    const s = x + y
    console.log(s) // 18
  }
  // 设用 counter 函数
  counter(10, 8)
  // 访问变量 s
  console.log(s)// 报错
</script>
```

总结：

1. 函数内部声明的变量，在函数外部无法被访问
2. 函数的参数也是函数内部的局部变量
3. 不同函数内部声明的变量无法互相访问
4. 函数执行完毕后，函数内部的变量实际被清空了

块作用域

在 JavaScript 中使用 `{}` 包裹的代码称为代码块，代码块内部声明的变量外部将【有可能】无法被访问。

```
<script>
  {
    // age 只能在该代码块中被访问
    let age = 18;
    console.log(age); // 正常
  }

  // 超出了 age 的作用域
  console.log(age) // 报错

  let flag = true;
  if(flag) {
    // str 只能在该代码块中被访问
    let str = 'hello world!'
    console.log(str); // 正常
  }

  // 超出了 age 的作用域
  console.log(str); // 报错

  for(let t = 1; t <= 6; t++) {
    // t 只能在该代码块中被访问
    console.log(t); // 正常
  }

  // 超出了 t 的作用域
  console.log(t); // 报错
</script>
```

JavaScript 中除了变量外还有常量，常量与变量本质的区别是【常量必须要有值且不允许被重新赋值】，常量值为对象时其属性和方法允许重新赋值。

```
<script>
  // 必须要有值
  const version = '1.0.0';

  // 不能重新赋值
  // version = '1.0.1';

  // 常量值为对象类型
  const user = {
    name: '小明',
    age: 18
  }

  // 不能重新赋值
  user = {};

  // 属性和方法允许被修改
  user.name = '小小明';
  user.gender = '男';
</script>
```

总结：

1. `let` 声明的变量会产生块作用域，`var` 不会产生块作用域
2. `const` 声明的常量也会产生块作用域
3. 不同代码块之间的变量无法互相访问
4. 推荐使用 `let` 或 `const`

注：开发中 `let` 和 `const` 经常不加区分的使用，如果担心某个值会不小被修改时，则只能使用 `const` 声明成常量。

全局作用域

`<script>` 标签和 `.js` 文件的【最外层】就是所谓的全局作用域，在此声明的变量在函数内部也可以被访问。

```
<script>
  // 此处是全局

  function sayHi() {
    // 此处为局部
  }

  // 此处为全局
</script>
```

全局作用域中声明的变量，任何其它作用域都可以被访问，如下代码所示：

```
<script>
    // 全局变量 name
```

```
    const name = '小明'

    // 函数作用域中访问全局
    function sayHi() {
      // 此处为局部
      console.log('你好' + name)
    }

    // 全局变量 flag 和 x
    const flag = true
    let x = 10

    // 块作用域中访问全局
    if(flag) {
      let y = 5
      console.log(x + y) // x 是全局的
    }
</script>
```

总结：

1. 为 `window` 对象动态添加的属性默认也是全局的，不推荐！
2. 函数中未使用任何关键字声明的变量为全局变量，不推荐！！！
3. 尽可能少的声明全局变量，防止全局变量被污染

JavaScript 中的作用域是程序被执行时的底层机制，了解这一机制有助于规范代码书写习惯，避免因作用域导致的语法错误。

作用域链

在解释什么是作用域链前先来看一段代码：

```
<script>
  // 全局作用域
  let a = 1
  let b = 2
  // 局部作用域
  function f() {
    let c
    // 局部作用域
    function g() {
      let d = 'yo'
    }
  }
</script>
```

函数内部允许创建新的函数，`f` 函数内部创建的新函数 `g`，会产生新的函数作用域，由此可知作用域产生了嵌套的关系。

如下图所示，父子关系的作用域关联在一起形成了链状的结构，作用域链的名字也由此而来。

作用域链本质上是底层的变量查找机制，在函数被执行时，会优先查找当前函数作用域中查找变量，如果当前作用域查找不到则会依次逐级查找父级作用域直到全局作用域，如下代码所示：

```
<script>
  // 全局作用域
  let a = 1
  let b = 2

  // 局部作用域
  function f() {
    let c
    // let a = 10;
    console.log(a) // 1 或 10
    console.log(d) // 报错

    // 局部作用域
    function g() {
      let d = 'yo'
      // let b = 20;
      console.log(b) // 2 或 20
    }

    // 调用 g 函数
    g()
  }

  console.log(c) // 报错
  console.log(d) // 报错

  f();
</script>
```

总结：

1. 嵌套关系的作用域串联起来形成了作用域链
2. 相同作用域链中按着从小到大的规则查找变量
3. 子作用域能够访问父作用域，父级作用域无法访问子级作用域

闭包

闭包是一种比较特殊和函数，使用闭包能够访问函数作用域中的变量。从代码形式上看闭包是一个做为返回值的函数，如下代码所示：

```
<body>
  <script>
    // 1. 闭包 : 内层函数 + 外层函数变量
    // function outer() {
    //   const a = 1
    //   function f() {
    //     console.log(a)
    //   }
    //   f()
    // }
    // outer()

    // 2. 闭包的应用：  实现数据的私有。统计函数的调用次数
    // let count = 1
```

```
    // function fn() {
    //   count++
    //   console.log(`函数被调用${count}次`)
    // }

    // 3. 闭包的写法  统计函数的调用次数
    function outer() {
      let count = 1
      function fn() {
        count++
        console.log(`函数被调用${count}次`)
      }
      return fn
    }
    const re = outer()
    // const re = function fn() {
    //   count++
    //   console.log(`函数被调用${count}次`)
    // }
    re()
    re()
    // const fn = function() { }  函数表达式
    // 4. 闭包存在的问题：  可能会造成内存泄漏
  </script>
</body>
```

总结：

1.怎么理解闭包？

- 闭包 = 内层函数 + 外层函数的变量

2.闭包的作用？

- 封闭数据，实现数据私有，外部也可以访问函数内部的变量
- 闭包很有用，因为它允许将函数与其所操作的某些数据（环境）关联起来

3.闭包可能引起的问题？

- 内存泄漏

变量提升

变量提升是 JavaScript 中比较“奇怪”的现象，它允许在变量声明之前即被访问，

```
<script>
  // 访问变量 str
  console.log(str + 'world!');

  // 声明变量 str
  var str = 'hello ';
</script>
```

总结：

1. 变量在未声明即被访问时会报语法错误

2. 变量在声明之前即被访问，变量的值为 `undefined`
3. `let` 声明的变量不存在变量提升，推荐使用 `let`
4. 变量提升出现在相同作用域当中
5. 实际开发中推荐先声明再访问变量

注：关于变量提升的原理分析会涉及较为复杂的词法分析等知识，而开发中使用 `let` 可以轻松规避变量的提升，因此在此不做过多的探讨，有兴趣可查阅资料。

函数

> 知道函数参数默认值、动态参数、剩余参数的使用细节，提升函数应用的灵活度，知道箭头函数的语法及与普通函数的差异。

函数提升

函数提升与变量提升比较类似，是指函数在声明之前即可被调用。

```
<script>
  // 调用函数
  foo()
  // 声明函数
  function foo() {
    console.log('声明之前即被调用...')
  }

  // 不存在提升现象
  bar()  // 错误
  var bar = function () {
    console.log('函数表达式不存在提升现象...')
  }
</script>
```

总结：

1. 函数提升能够使函数的声明调用更灵活
2. 函数表达式不存在提升的现象
3. 函数提升出现在相同作用域当中

函数参数

函数参数的使用细节，能够提升函数应用的灵活度。

默认值

```
<script>
  // 设置参数默认值
  function sayHi(name="小明", age=18) {
    document.write(`<p>大家好，我叫${name}，我今年${age}岁了。</p>`);
  }
  // 调用函数
  sayHi();
  sayHi('小红');
  sayHi('小刚', 21);
</script>
```

总结：

1. 声明函数时为形参赋值即为参数的默认值
2. 如果参数未自定义默认值时，参数的默认值为 `undefined`
3. 调用函数时没有传入对应实参时，参数的默认值被当做实参传入

动态参数

`arguments` 是函数内部内置的伪数组变量，它包含了调用函数时传入的所有实参。

```
<script>
  // 求生函数，计算所有参数的和
  function sum() {
    // console.log(arguments)
    let s = 0
    for(let i = 0; i < arguments.length; i++) {
      s += arguments[i]
    }
    console.log(s)
  }
  // 调用求和函数
  sum(5, 10)// 两个参数
  sum(1, 2, 4) // 两个参数
</script>
```

总结：

1. `arguments` 是一个伪数组
2. `arguments` 的作用是动态获取函数的实参

剩余参数

```
<script>
  function config(baseURL, ...other) {
    console.log(baseURL) // 得到 'http://baidu.com'
    console.log(other)  // other  得到 ['get', 'json']
  }
  // 调用函数
  config('http://baidu.com', 'get', 'json');
</script>
```

总结：

1. `...` 是语法符号，置于最末函数形参之前，用于获取多余的实参
2. 借助 `...` 获取的剩余实参，是个真数组

箭头函数

箭头函数是一种声明函数的简洁语法，它与普通函数并无本质的区别，差异性更多体现在语法格式上。

```
<body>
  <script>
    // const fn = function () {
    //   console.log(123)
    // }
    // 1. 箭头函数 基本语法
    // const fn = () => {
    //   console.log(123)
    // }
    // fn()
    // const fn = (x) => {
    //   console.log(x)
    // }
    // fn(1)
    // 2. 只有一个形参的时候，可以省略小括号
    // const fn = x => {
    //   console.log(x)
    // }
    // fn(1)
    // // 3. 只有一行代码的时候，我们可以省略大括号
    // const fn = x => console.log(x)
    // fn(1)
    // 4. 只有一行代码的时候，可以省略return
    // const fn = x => x + x
    // console.log(fn(1))
    // 5. 箭头函数可以直接返回一个对象
    // const fn = (uname) => ({ uname: uname })
    // console.log(fn('刘德华'))

  </script>
</body>
```

总结：

1. 箭头函数属于表达式函数，因此不存在函数提升
2. 箭头函数只有一个参数时可以省略圆括号 `()`
3. 箭头函数函数体只有一行代码时可以省略花括号 `{}`，并自动做为返回值被返回

箭头函数参数

箭头函数中没有 `arguments`，只能使用 `...` 动态获取实参

```
<body>
  <script>
    // 1. 利用箭头函数来求和
    const getSum = (...arr) => {
      let sum = 0
      for (let i = 0; i < arr.length; i++) {
        sum += arr[i]
      }
      return sum
    }
    const result = getSum(2, 3, 4)
    console.log(result) // 9
  </script>
```

箭头函数 this

箭头函数不会创建自己的this,它只会从自己的作用域链的上一层沿用this。

```
  <script>
    // 以前this的指向：  谁调用的这个函数，this 就指向谁
    // console.log(this)  // window
    // // 普通函数
    // function fn() {
    //   console.log(this)  // window
    // }
    // window.fn()
    // // 对象方法里面的this
    // const obj = {
    //   name: 'andy',
    //   sayHi: function () {
    //     console.log(this)  // obj
    //   }
    // }
    // obj.sayHi()

    // 2. 箭头函数的this  是上一层作用域的this 指向
    // const fn = () => {
    //   console.log(this)  // window
    // }
    // fn()
    // 对象方法箭头函数 this
    // const obj = {
    //   uname: 'pink老师',
    //   sayHi: () => {
    //     console.log(this)  // this 指向谁？ window
    //   }
    // }
    // obj.sayHi()

    const obj = {
      uname: 'pink老师',
      sayHi: function () {
        console.log(this)  // obj
        let i = 10
        const count = () => {
```

```
            console.log(this)  // obj
          }
          count()
        }
      }
      obj.sayHi()

    </script>
```

解构赋值

> 知道解构的语法及分类，使用解构简洁语法快速为变量赋值。

解构赋值是一种快速为变量赋值的简洁语法，本质上仍然是为变量赋值，分为数组解构、对象解构两大类型。

数组解构

数组解构是将数组的单元值快速批量赋值给一系列变量的简洁语法，如下代码所示：

```
<script>
  // 普通的数组
  let arr = [1, 2, 3]
  // 批量声明变量 a b c
  // 同时将数组单元值 1 2 3 依次赋值给变量 a b c
  let [a, b, c] = arr
  console.log(a); // 1
  console.log(b); // 2
  console.log(c); // 3
</script>
```

总结：

1. 赋值运算符 `=` 左侧的 `[]` 用于批量声明变量，右侧数组的单元值将被赋值给左侧的变量
2. 变量的顺序对应数组单元值的位置依次进行赋值操作
3. 变量的数量大于单元值数量时，多余的变量将被赋值为 `undefined`
4. 变量的数量小于单元值数量时，可以通过 `...` 获取剩余单元值，但只能置于最末位
5. 允许初始化变量的默认值，且只有单元值为 `undefined` 时默认值才会生效

注：支持多维解构赋值，比较复杂后续有应用需求时再进一步分析

对象解构

对象解构是将对象属性和方法快速批量赋值给一系列变量的简洁语法，如下代码所示：

```
<script>
  // 普通对象
  const user = {
    name: '小明',
    age: 18
  };
  // 批量声明变量 name age
  // 同时将数组单元值 小明  18 依次赋值给变量 name  age
  const {name, age} = user

  console.log(name) // 小明
  console.log(age) // 18
</script>
```

总结:

1. 赋值运算符 `=` 左侧的 `{}` 用于批量声明变量，右侧对象的属性值将被赋值给左侧的变量
2. 对象属性的值将被赋值给与属性名相同的变量
3. 对象中找不到与变量名一致的属性时变量值为 `undefined`
4. 允许初始化变量的默认值，属性不存在或单元值为 `undefined` 时默认值才会生效

注：支持多维解构赋值

```
<body>
  <script>
    // 1. 这是后台传递过来的数据
    const msg = {
      "code": 200,
      "msg": "获取新闻列表成功",
      "data": [
        {
          "id": 1,
          "title": "5G商用自己，三大运用商收入下降",
          "count": 58
        },
        {
          "id": 2,
          "title": "国际媒体头条速览",
          "count": 56
        },
        {
          "id": 3,
          "title": "乌克兰和俄罗斯持续冲突",
          "count": 1669
        },

      ]
    }

    // 需求1：  请将以上msg对象  采用对象解构的方式 只选出  data 方面后面使用渲染页面
    // const { data } = msg
    // console.log(data)
    // 需求2：  上面msg是后台传递过来的数据，我们需要把data选出当做参数传递给 函数
    // const { data } = msg
```

```
    // msg 虽然很多属性，但是我们利用解构只要 data值
    function render({ data }) {
      // const { data } = arr
      // 我们只要 data 数据
      // 内部处理
      console.log(data)

    }
    render(msg)

    // 需求3， 为了防止msg里面的data名字混淆，要求渲染函数里面的数据名改为 myData
    function render({ data: myData }) {
      // 要求将 获取过来的 data数据 更名为 myData
      // 内部处理
      console.log(myData)

    }
    render(msg)

  </script>
```

综合案例

forEach遍历数组

forEach() 方法用于调用数组的每个元素，并将元素传递给回调函数

> 注意：
>
> 1.forEach 主要是遍历数组
>
> 2.参数当前数组元素是必须要写的， 索引号可选。

```
<body>
  <script>
    // forEach 就是遍历  加强版的for循环  适合于遍历数组对象
    const arr = ['red', 'green', 'pink']
    const result = arr.forEach(function (item, index) {
      console.log(item)  // 数组元素 red  green pink
      console.log(index) // 索引号
    })
    // console.log(result)
  </script>
</body>
```

filter筛选数组

filter() 方法创建一个新的数组，新数组中的元素是通过检查指定数组中符合条件的所有元素

主要使用场景： 筛选数组符合条件的元素，并返回筛选之后元素的新数组

```
<body>
  <script>
    const arr = [10, 20, 30]
    // const newArr = arr.filter(function (item, index) {
```

```
    //   // console.log(item)
    //   // console.log(index)
    //   return item >= 20
    // })
    // 返回的符合条件的新数组

    const newArr = arr.filter(item => item >= 20)
    console.log(newArr)
  </script>
</body>
```

> 了解面向对象编程的基础概念及构造函数的作用，体会 JavaScript 一切皆对象的语言特征，掌握常见的对象属性和方法的使用。

- 了解面向对象编程中的一般概念
- 能够基于构造函数创建对象
- 理解 JavaScript 中一切皆对象的语言特征
- 理解引用对象类型值存储的的特征
- 掌握包装类型对象常见方法的使用

深入对象

构造函数

构造函数是专门用于创建对象的函数，如果一个函数使用 `new` 关键字调用，那么这个函数就是构造函数。

```
<script>
  // 定义函数
  function foo() {
    console.log('通过 new 也能调用函数...');
  }
  // 调用函数
  new foo;
</script>
```

总结：

2. 使用 `new` 关键字调用函数的行为被称为实例化
3. 实例化构造函数时没有参数时可以省略 `()`
4. 构造函数的返回值即为新创建的对象
5. 构造函数内部的 `return` 返回的值无效！

注：实践中为了从视觉上区分构造函数和普通函数，习惯将构造函数的首字母大写。

实例成员

通过构造函数创建的对象称为实例对象，实例对象中的属性和方法称为实例成员。

```
<script>
  // 构造函数
  function Person() {
    // 构造函数内部的 this 就是实例对象
    // 实例对象中动态添加属性
    this.name = '小明'
    // 实例对象动态添加方法
    this.sayHi = function () {
      console.log('大家好~')
    }
  }
  // 实例化，p1 是实例对象
  // p1 实际就是 构造函数内部的 this
  const p1 = new Person()
  console.log(p1)
  console.log(p1.name) // 访问实例属性
  p1.sayHi() // 调用实例方法
</script>
```

总结：

1. 构造函数内部 `this` 实际上就是实例对象，为其动态添加的属性和方法即为实例成员
2. 为构造函数传入参数，动态创建结构相同但值不同的对象

注：构造函数创建的实例对象彼此独立互不影响。

静态成员

在 JavaScript 中底层函数本质上也是对象类型，因此允许直接为函数动态添加属性或方法，构造函数的属性和方法被称为静态成员。

```
<script>
  // 构造函数
  function Person(name, age) {
    // 省略实例成员
  }
  // 静态属性
  Person.eyes = 2
  Person.arms = 2
  // 静态方法
  Person.walk = function () {
    console.log('^_^人都会走路...')
    // this 指向 Person
    console.log(this.eyes)
  }
</script>
```

总结：

1. 静态成员指的是添加到构造函数本身的属性和方法

2. 一般公共特征的属性或方法静态成员设置为静态成员
3. 静态成员方法中的 `this` 指向构造函数本身

内置构造函数

在 JavaScript 中**最主要**的数据类型有 6 种，分别是字符串、数值、布尔、undefined、null 和 对象，常见的对象类型数据包括数组和普通对象。其中字符串、数值、布尔、undefined、null 也被称为简单类型或基础类型，对象也被称为引用类型。

在 JavaScript 内置了一些构造函数，绝大部的数据处理都是基于这些构造函数实现的，JavaScript 基础阶段学习的 `Date` 就是内置的构造函数。

```
<script>
  // 实例化
    let date = new Date();

  // date 即为实例对象
  console.log(date);
</script>
```

甚至字符串、数值、布尔、数组、普通对象也都有专门的构造函数，用于创建对应类型的数据。

Object

`Object` 是内置的构造函数，用于创建普通对象。

```
<script>
  // 通过构造函数创建普通对象
  const user = new Object({name: '小明', age: 15})

  // 这种方式声明的变量称为【字面量】
  let student = {name: '杜子腾', age: 21}

  // 对象语法简写
  let name = '小红';
  let people = {
    // 相当于 name: name
    name,
    // 相当于 walk: function () {}
    walk () {
      console.log('人都要走路...');
    }
  }

  console.log(student.constructor);
  console.log(user.constructor);
  console.log(student instanceof Object);
</script>
```

。

总结：

1. 推荐使用字面量方式声明对象，而不是 `Object` 构造函数

2. `Object.assign` 静态方法创建新的对象
3. `Object.keys` 静态方法获取对象中所有属性
4. `Object.values` 表态方法获取对象中所有属性值

Array

`Array` 是内置的构造函数，用于创建数组。

```
<script>
  // 构造函数创建数组
  let arr = new Array(5, 7, 8);

  // 字面量方式创建数组
  let list = ['html', 'css', 'javascript']

</script>
```

数组赋值后，无论修改哪个变量另一个对象的数据值也会相当发生改变。

总结：

1. 推荐使用字面量方式声明数组，而不是 `Array` 构造函数
2. 实例方法 `forEach` 用于遍历数组，替代 `for` 循环 (重点)
3. 实例方法 `filter` 过滤数组单元值，生成新数组(重点)
4. 实例方法 `map` 迭代原数组，生成新数组(重点)
5. 实例方法 `join` 数组元素拼接为字符串，返回字符串(重点)
6. 实例方法 `find` 查找元素， 返回符合测试条件的第一个数组元素值，如果没有符合条件的则返回 undefined(重点)
7. 实例方法 `every` 检测数组所有元素是否都符合指定条件，如果**所有元素**都通过检测返回 true，否则返回 false(重点)
8. 实例方法 `some` 检测数组中的元素是否满足指定条件 **如果数组中有**元素满足条件返回 true，否则返回 false
9. 实例方法 `concat` 合并两个数组，返回生成新数组
10. 实例方法 `sort` 对原数组单元值排序
11. 实例方法 `splice` 删除或替换原数组单元
12. 实例方法 `reverse` 反转数组
13. 实例方法 `findIndex` 查找元素的索引值

包装类型

在 JavaScript 中的字符串、数值、布尔具有对象的使用特征，如具有属性和方法，如下代码举例：

```
<script>
  // 字符串类型
  const str = 'hello world!'
    // 统计字符的长度（字符数量）
  console.log(str.length)

  // 数值类型
  const price = 12.345
  // 保留两位小数
  price.toFixed(2) // 12.34
</script>
```

之所以具有对象特征的原因是字符串、数值、布尔类型数据是 JavaScript 底层使用 Object 构造函数“包装”来的，被称为包装类型。

String

`String` 是内置的构造函数，用于创建字符串。

```
<script>
  // 使用构造函数创建字符串
  let str = new String('hello world!');

  // 字面量创建字符串
  let str2 = '你好，世界！';

  // 检测是否属于同一个构造函数
  console.log(str.constructor === str2.constructor); // true
  console.log(str instanceof String); // false
</script>
```

总结：

1. 实例属性 `length` 用来获取字符串的度长(重点)
2. 实例方法 `split('分隔符')` 用来将字符串拆分成数组(重点)
3. 实例方法 `substring（需要截取的第一个字符的索引[,结束的索引号]）` 用于字符串截取(重点)
4. 实例方法 `startsWith(检测字符串[, 检测位置索引号])` 检测是否以某字符开头(重点)
5. 实例方法 `includes(搜索的字符串[, 检测位置索引号])` 判断一个字符串是否包含在另一个字符串中，根据情况返回 true 或 false(重点)
6. 实例方法 `toUpperCase` 用于将字母转换成大写
7. 实例方法 `toLowerCase` 用于将就转换成小写
8. 实例方法 `indexOf` 检测是否包含某字符
9. 实例方法 `endsWith` 检测是否以某字符结尾
10. 实例方法 `replace` 用于替换字符串，支持正则匹配
11. 实例方法 `match` 用于查找字符串，支持正则匹配

注：String 也可以当做普通函数使用，这时它的作用是强制转换成字符串数据类型。

Number

`Number` 是内置的构造函数，用于创建数值。

```
<script>
  // 使用构造函数创建数值
  let x = new Number('10')
  let y = new Number(5)

  // 字面量创建数值
  let z = 20

</script>
```

总结：

1. 推荐使用字面量方式声明数值，而不是 `Number` 构造函数
2. 实例方法 `toFixed` 用于设置保留小数位的长度

面向过程

面向过程就是分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候再一个一个的依次

调用就可以了。

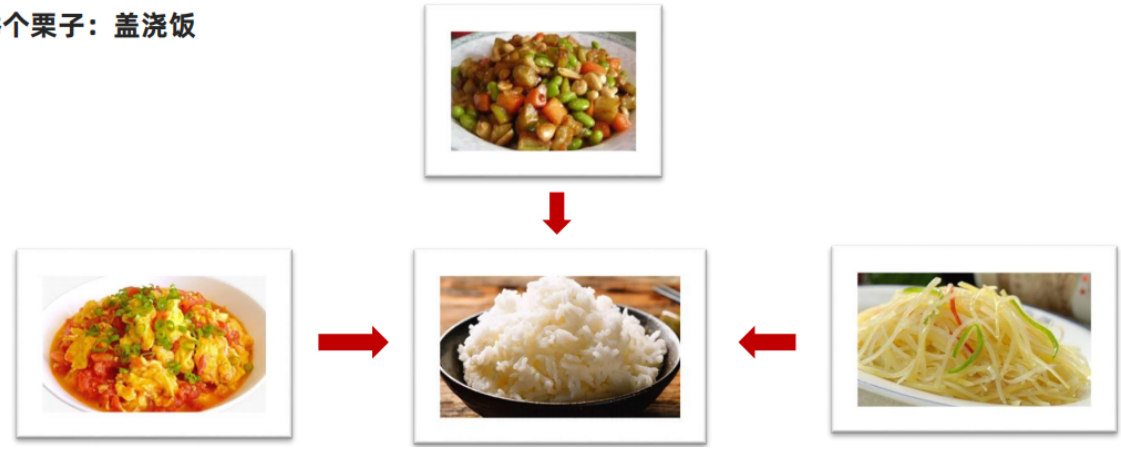
举个栗子：蛋炒饭




面向对象

面向对象是把事务分解成为一个个对象，然后由对象之间分工与合作。

举个栗子：盖浇饭



在面向对象程序开发思想中，每一个对象都是功能中心，具有明确分工。

面向对象编程具有灵活、代码可复用、容易维护和开发的优点，更适合多人合作的大型软件项目。

面向对象的特性：

- 封装性
- 继承性
- 多态性

构造函数

对比以下通过面向对象的构造函数实现的封装：

```
<script>
  function Person() {
    this.name = '佚名'
    // 设置名字
    this.setName = function (name) {
      this.name = name
    }
    // 读取名字
    this.getName = () => {
      console.log(this.name)
    }
  }

  // 实例对像，获得了构造函数中封装的所有逻辑
  let p1 = new Person()
  p1.setName('小明')
  console.log(p1.name)

  // 实例对象
  let p2 = new Person()
  console.log(p2.name)
</script>
```

封装是面向对象思想中比较重要的一部分，js面向对象可以通过构造函数实现的封装。

同样的将变量和函数组合到了一起并能通过 this 实现数据的共享，所不同的是借助构造函数创建出来的实例对象之

间是彼此不影响的

> 总结：
>
> 1. 构造函数体现了面向对象的封装特性
> 2. 构造函数实例创建的对象彼此独立、互不影响

封装是面向对象思想中比较重要的一部分，js面向对象可以通过构造函数实现的封装。

前面我们学过的构造函数方法很好用，但是 存在 `浪费内存` 的问题

原型对象

构造函数通过原型分配的函数是所有对象所 共享的。

- JavaScript 规定，每一个构造函数都有一个 prototype 属性，指向另一个对象，所以我们也称为原型对象
- 这个对象可以挂载函数，对象实例化不会多次创建原型上函数，节约内存
- 我们可以把那些不变的方法，直接定义在 prototype 对象上，这样所有对象的实例就可以共享这些方法。
- 构造函数和原型对象中的this 都指向 实例化的对象

```
<script>
  function Person() {

  }

  // 每个函数都有 prototype 属性
  console.log(Person.prototype)
</script>
```

了解了 JavaScript 中构造函数与原型对象的关系后，再来看原型对象具体的作用，如下代码所示：

```
<script>
  function Person() {
    // 此处未定义任何方法
  }

  // 为构造函数的原型对象添加方法
  Person.prototype.sayHi = function () {
    console.log('Hi~');
  }

  // 实例化
  let p1 = new Person();
  p1.sayHi(); // 输出结果为 Hi~
</script>
```

构造函数 `Person` 中未定义任何方法，这时实例对象调用了原型对象中的方法 `sayHi`，接下来改动一下代码：

```
<script>
  function Person() {
    // 此处定义同名方法 sayHi
    this.sayHi = function () {
      console.log('嗨!');
    }
  }

  // 为构造函数的原型对象添加方法
  Person.prototype.sayHi = function () {
    console.log('Hi~');
  }

```

```
  let p1 = new Person();
  p1.sayHi(); // 输出结果为 嗨!
</script>
```

构造函数 `Person` 中定义与原型对象中相同名称的方法，这时实例对象调用则是构造函中的方法 `sayHi`。

通过以上两个简单示例不难发现 JavaScript 中对象的工作机制：**当访问对象的属性或方法时，先在当前实例对象是查找，然后再去原型对象查找，并且原型对象被所有实例共享。**

```
<script>
    function Person() {
    // 此处定义同名方法 sayHi
    this.sayHi = function () {
      console.log('嗨!' + this.name)
    }
  }

  // 为构造函数的原型对象添加方法
  Person.prototype.sayHi = function () {
    console.log('Hi~' + this.name)
  }
  // 在构造函数的原型对象上添加属性
  Person.prototype.name = '小明'

  let p1 = new Person()
  p1.sayHi(); // 输出结果为 嗨!

  let p2 = new Person()
  p2.sayHi()
</script>
```

总结：**结合构造函数原型的特征，实际开发重往往会将封装的功能函数添加到原型对象中。**

constructor 属性

在哪里？ 每个原型对象里面都有个constructor 属性（constructor 构造函数）

作用：该属性指向该原型对象的构造函数， 简单理解，就是指向我的爸爸，我是有爸爸的孩子

使用场景：

如果有多个对象的方法，我们可以给原型对象采取对象形式赋值.

但是这样就会覆盖构造函数原型对象原来的内容，这样修改后的原型对象 constructor 就不再指向当前构造函数了

此时，我们可以在修改后的原型对象中，添加一个 constructor 指向原来的构造函数。

对象原型

对象都会有一个属性 **proto** 指向构造函数的 prototype 原型对象，之所以我们对象可以使用构造函数 prototype

原型对象的属性和方法，就是因为对象有 **proto** 原型的存在。

注意：

- **proto** 是JS非标准属性
- [[prototype]]和**proto**意义相同
- 用来表明当前实例对象指向哪个原型对象prototype
- **proto**对象原型里面也有一个 constructor属性，指向创建该实例对象的构造函数

原型继承

继承是面向对象编程的另一个特征，通过继承进一步提升代码封装的程度，JavaScript 中大多是借助原型对象实现继承

的特性。

龙生龙、凤生凤、老鼠的儿子会打洞描述的正是继承的含义。

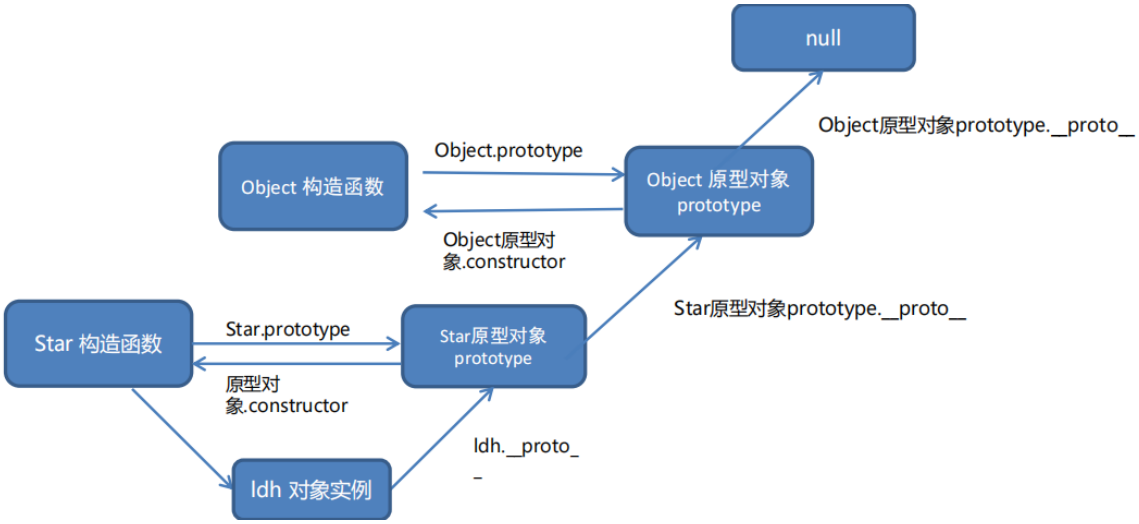
```
<body>
  <script>
    // 继续抽取    公共的部分放到原型上
    // const Person1 = {
    //   eyes: 2,
    //   head: 1
    // }
    // const Person2 = {
    //   eyes: 2,
    //   head: 1
    // }
    // 构造函数  new 出来的对象 结构一样，但是对象不一样
    function Person() {
      this.eyes = 2
      this.head = 1
    }
    // console.log(new Person)
    // 女人  构造函数    继承   想要 继承 Person
    function Woman() {

    }
    // Woman 通过原型来继承 Person
    // 父构造函数（父类）    子构造函数（子类）
    // 子类的原型 =  new 父类
    Woman.prototype = new Person()   // {eyes: 2, head: 1}
    // 指回原来的构造函数
    Woman.prototype.constructor = Woman

    // 给女人添加一个方法   生孩子
    Woman.prototype.baby = function () {
      console.log('宝贝')
    }
    const red = new Woman()
    console.log(red)
    // console.log(Woman.prototype)
    // 男人 构造函数   继承   想要 继承 Person
    function Man() {

    }
```

```
    // 通过 原型继承 Person
    Man.prototype = new Person()
    Man.prototype.constructor = Man
    const pink = new Man()
    console.log(pink)
  </script>
</body>
```

原型链

基于原型对象的继承使得不同构造函数的原型对象关联在一起，并且这种关联的关系是一种链状的结构，我们将原型对

象的链状结构关系称为原型链




```
<body>
  <script>
    // function Objetc() {}
    console.log(Object.prototype)
    console.log(Object.prototype.__proto__)

    function Person() {

    }
    const ldh = new Person()
    // console.log(ldh.__proto__ === Person.prototype)
    // console.log(Person.prototype.__proto__ === Object.prototype)
    console.log(ldh instanceof Person)
    console.log(ldh instanceof Object)
    console.log(ldh instanceof Array)
    console.log([1, 2, 3] instanceof Array)
    console.log(Array instanceof Object)
  </script>
</body>
```

① 当访问一个对象的属性（包括方法）时，首先查找这个对象自身有没有该属性。

② 如果没有就查找它的原型（也就是 **proto**指向的 prototype 原型对象）

③ 如果还没有就查找原型对象的原型（Object的原型对象）

④ 依此类推一直找到 Object 为止（null）

⑤ **proto**对象原型的意义就在于为对象成员查找机制提供一个方向，或者说一条路线

⑥ 可以使用 instanceof 运算符用于检测构造函数的 prototype 属性是否出现在某个实例对象的原型链上

深浅拷贝

浅拷贝

首先浅拷贝和深拷贝只针对引用类型

浅拷贝：拷贝的是地址

常见方法：

1. 拷贝对象：Object.assgin() / 展开运算符 {...obj} 拷贝对象
2. 拷贝数组：Array.prototype.concat() 或者 [...arr]

> 如果是简单数据类型拷贝值，引用数据类型拷贝的是地址 (简单理解： 如果是单层对象，没问题，如果有多层就有问题)

深拷贝

首先浅拷贝和深拷贝只针对引用类型

深拷贝：拷贝的是对象，不是地址

常见方法：

1. 通过递归实现深拷贝
2. lodash/cloneDeep
3. 通过JSON.stringify()实现

递归实现深拷贝

函数递归：

如果一个函数在内部可以调用其本身，那么这个函数就是递归函数

- 简单理解:函数内部自己调用自己, 这个函数就是递归函数
- 递归函数的作用和循环效果类似
- 由于递归很容易发生“栈溢出”错误（stack overflow），所以必须要加退出条件 return

```
<body>
  <script>
    const obj = {
      uname: 'pink',
      age: 18,
      hobby: ['乒乓球', '足球'],
      family: {
        baby: '小pink'
      }
    }
    const o = {}
```

```
    // 拷贝函数
    function deepCopy(newObj, oldObj) {
      debugger
      for (let k in oldObj) {
        // 处理数组的问题  一定先写数组 在写 对象 不能颠倒
        if (oldObj[k] instanceof Array) {
          newObj[k] = []
          //  newObj[k] 接收 []  hobby
          //  oldObj[k]   ['乒乓球', '足球']
          deepCopy(newObj[k], oldObj[k])
        } else if (oldObj[k] instanceof Object) {
          newObj[k] = {}
          deepCopy(newObj[k], oldObj[k])
        }
        else {
          //  k  属性名 uname age    oldObj[k]  属性值  18
          // newObj[k]  === o.uname  给新对象添加属性
          newObj[k] = oldObj[k]
        }
      }
    }
    deepCopy(o, obj) // 函数调用  两个参数 o 新对象  obj 旧对象
    console.log(o)
    o.age = 20
    o.hobby[0] = '篮球'
    o.family.baby = '老pink'
    console.log(obj)
    console.log([1, 23] instanceof Object)
    // 复习
    // const obj = {
    //   uname: 'pink',
    //   age: 18,
    //   hobby: ['乒乓球', '足球']
    // }
    // function deepCopy({ }, oldObj) {
    //   // k 属性名  oldObj[k] 属性值
    //   for (let k in oldObj) {
    //     // 处理数组的问题   k 变量
    //     newObj[k] = oldObj[k]
    //     // o.uname = 'pink'
    //     // newObj.k  = 'pink'
    //   }
    // }
  </script>
</body>
```

js库lodash里面cloneDeep内部实现了深拷贝

```
<body>
  <!-- 先引用 -->
  <script src="./lodash.min.js"></script>
  <script>
    const obj = {
      uname: 'pink',
      age: 18,
```

```
      hobby: ['乒乓球', '足球'],
      family: {
        baby: '小pink'
      }
    }
    const o = _.cloneDeep(obj)
    console.log(o)
    o.family.baby = '老pink'
    console.log(obj)
  </script>
</body>
```

JSON序列化

```
<body>
  <script>
    const obj = {
      uname: 'pink',
      age: 18,
      hobby: ['乒乓球', '足球'],
      family: {
        baby: '小pink'
      }
    }
    // 把对象转换为 JSON 字符串
    // console.log(JSON.stringify(obj))
    const o = JSON.parse(JSON.stringify(obj))
    console.log(o)
    o.family.baby = '123'
    console.log(obj)
  </script>
</body>
```

异常处理

> 了解 JavaScript 中程序异常处理的方法，提升代码运行的健壮性。

throw

异常处理是指预估代码执行过程中可能发生的错误，然后最大程度的避免错误的发生导致整个程序无法继续运行

总结：

1. throw 抛出异常信息，程序也会终止执行
2. throw 后面跟的是错误提示信息
3. Error 对象配合 throw 使用，能够设置更详细的错误信息

```
<script>
  function counter(x, y) {

    if(!x || !y) {
      // throw '参数不能为空!';
      throw new Error('参数不能为空!')
    }

    return x + y
  }

  counter()
</script>
```

总结：

1. `throw` 抛出异常信息，程序也会终止执行
2. `throw` 后面跟的是错误提示信息
3. `Error` 对象配合 `throw` 使用，能够设置更详细的错误信息

try ... catch

```
<script>
   function foo() {
      try {
        // 查找 DOM 节点
        const p = document.querySelector('.p')
        p.style.color = 'red'
      } catch (error) {
        // try 代码段中执行有错误时，会执行 catch 代码段
        // 查看错误信息
        console.log(error.message)
        // 终止代码继续执行
        return

      }
      finally {
          alert('执行')
      }
      console.log('如果出现错误，我的语句不会执行')
    }
    foo()
</script>
```

总结：

1. `try...catch` 用于捕获错误信息
2. 将预估可能发生错误的代码写在 `try` 代码段中
3. 如果 `try` 代码段中出现错误后，会执行 `catch` 代码段，并截获到错误信息

debugger

相当于断点调试

处理this

> 了解函数中 this 在不同场景下的默认值，知道动态指定函数 this 值的方法。

`this` 是 JavaScript 最具“魅惑”的知识点，不同的应用场合 `this` 的取值可能会有意想不到的结果，在此我们对以往学习过的关于【 `this` 默认的取值】情况进行归纳和总结。

普通函数

普通函数的调用方式决定了 `this` 的值，即【谁调用 `this` 的值指向谁】，如下代码所示：

```
<script>
  // 普通函数
  function sayHi() {
    console.log(this)
  }
  // 函数表达式
  const sayHello = function () {
    console.log(this)
  }
  // 函数的调用方式决定了 this 的值
  sayHi() // window
  window.sayHi()


// 普通对象
  const user = {
    name: '小明',
    walk: function () {
      console.log(this)
    }
  }
  // 动态为 user 添加方法
  user.sayHi = sayHi
  uesr.sayHello = sayHello
  // 函数调用方式，决定了 this 的值
  user.sayHi()
  user.sayHello()
</script>
```

注： 普通函数没有明确调用者时 `this` 值为 `window`，严格模式下没有调用者时 `this` 的值为 `undefined`。

箭头函数

箭头函数中的 `this` 与普通函数完全不同，也不受调用方式的影响，事实上箭头函数中并不存在 `this` ！箭头函数中访问的 `this` 不过是箭头函数所在作用域的 `this` 变量。

```
<script>

```

```
  console.log(this) // 此处为 window
  // 箭头函数
  const sayHi = function() {
    console.log(this) // 该箭头函数中的 this 为函数声明环境中 this 一致
  }
  // 普通对象
  const user = {
    name: '小明',
    // 该箭头函数中的 this 为函数声明环境中 this 一致
    walk: () => {
      console.log(this)
    },

    sleep: function () {
      let str = 'hello'
      console.log(this)
      let fn = () => {
        console.log(str)
        console.log(this) // 该箭头函数中的 this 与 sleep 中的 this 一致
      }
      // 调用箭头函数
      fn();
    }
  }

  // 动态添加方法
  user.sayHi = sayHi

  // 函数调用
  user.sayHi()
  user.sleep()
  user.walk()
</script>
```

在开发中【使用箭头函数前需要考虑函数中 `this` 的值】，**事件回调函数**使用箭头函数时，`this` 为全局的 `window`，因此DOM事件回调函数不推荐使用箭头函数，如下代码所示：

```
<script>
  // DOM 节点
  const btn = document.querySelector('.btn')
  // 箭头函数 此时 this 指向了 window
  btn.addEventListener('click', () => {
    console.log(this)
  })
  // 普通函数 此时 this 指向了 DOM 对象
  btn.addEventListener('click', function () {
    console.log(this)
  })
</script>
```

同样由于箭头函数 `this` 的原因，**基于原型的面向对象也不推荐采用箭头函数**，如下代码所示：

```
<script>
  function Person() {
  }
  // 原型对像上添加了箭头函数
  Person.prototype.walk = () => {
    console.log('人都要走路...')
    console.log(this); // window
  }
  const p1 = new Person()
  p1.walk()
</script>
```

改变this指向

以上归纳了普通函数和箭头函数中关于 `this` 默认值的情形，不仅如此 JavaScript 中还允许指定函数中 `this` 的指向，有 3 个方法可以动态指定普通函数中 `this` 的指向：

call

使用 `call` 方法调用函数，同时指定函数中 `this` 的值，使用方法如下代码所示：

```
<script>
  // 普通函数
  function sayHi() {
    console.log(this);
  }

  let user = {
    name: '小明',
    age: 18
  }

  let student = {
    name: '小红',
    age: 16
  }

  // 调用函数并指定 this 的值
  sayHi.call(user); // this 值为 user
  sayHi.call(student); // this 值为 student

  // 求和函数
  function counter(x, y) {
    return x + y;
  }

  // 调用 counter 函数，并传入参数
  let result = counter.call(null, 5, 10);
  console.log(result);
</script>
```

总结：

1. `call` 方法能够在调用函数的同时指定 `this` 的值

2. 使用 `call` 方法调用函数时，第1个参数为 `this` 指定的值
3. `call` 方法的其余参数会依次自动传入函数做为函数的参数

apply

使用 `call` 方法**调用函数**，同时指定函数中 `this` 的值，使用方法如下代码所示：

```
<script>
  // 普通函数
  function sayHi() {
    console.log(this)
  }

  let user = {
    name: '小明',
    age: 18
  }

  let student = {
    name: '小红',
    age: 16
  }

  // 调用函数并指定 this 的值
  sayHi.apply(user) // this 值为 user
  sayHi.apply(student) // this 值为 student

  // 求和函数
  function counter(x, y) {
    return x + y
  }
  // 调用 counter 函数，并传入参数
  let result = counter.apply(null, [5, 10])
  console.log(result)
</script>
```

总结：

1. `apply` 方法能够在调用函数的同时指定 `this` 的值
2. 使用 `apply` 方法调用函数时，第1个参数为 `this` 指定的值
3. `apply` 方法第2个参数为数组，数组的单元值依次自动传入函数做为函数的参数

bind

`bind` 方法并**不会调用函数**，而是创建一个指定了 `this` 值的新函数，使用方法如下代码所示：

```
<script>
  // 普通函数
  function sayHi() {
    console.log(this)
  }
  let user = {
    name: '小明',
    age: 18
```

```
  }
  // 调用 bind 指定 this 的值
  let sayHello = sayHi.bind(user);
  // 调用使用 bind 创建的新函数
  sayHello()
</script>
```

注：`bind` 方法创建新的函数，与原函数的唯一的变化是改变了 `this` 的值。

防抖节流

1. 防抖（debounce）
   所谓防抖，就是指触发事件后在 n 秒内函数只能执行一次，如果在 n 秒内又触发了事件，则会重新计算函数执行时间
2. 节流（throttle）
   所谓节流，就是指连续触发事件但是在 n 秒中只执行一次函数

AjaxDAY1

知识点自测

1. 如下对象取值的方式哪个正确?

```
let obj = {
    name: '黑马'
}
```

A: obj.a

B: obj()a

▶ 答案

2. 哪个赋值会让浏览器解析成标签显示?

```
let ul = document.querySelector('#ul')
let str = `<span>我是span标签</span>`
```

A: ul.innerText = str

B: ul.innerHTML = str

▶ 答案

3. 哪个是获取输入框值的方式?

```
let theInput = document.querySelector('#input')
```

A: theInput.innerHTML

B: theInput.value

▶ 答案

4. 哪个是用于获取标签内容?

```
let theP = document.querySelector('#p')
```

A: theP.innerHTML = '内容'

B: theP.innerHTML

▶ 答案

5. 哪个是数组的映射方法?

A: arr.forEach

B: arr.map

▶ 答案

6. 数组转字符串并指定拼接符的是哪个?

A: arr.join()

B: arr.split()

▶ 答案

7. 函数传参的方式哪个是正确的?

```
function showAlert(msg, className) {}
```

A：showAlert('消息', '类名')

B：showAlert()

▶ 答案

8. 以下哪套代码可以实现对象属性的简写?

A:

```
const username = '老李'
let obj = {
  username: username
}
```

B：

```
const user = '老李'
let obj = {
  username: user
}
```

▶ 答案

9. 以下代码的值是多少？

```
const age = 10
const result = age > 18 ? '成年了' : '未成年'
```

A：‘成年了’

B：‘未成年’

▶ 答案

10. 以下哪个方法可以添加一个额外类名？

A：标签对象.classList.add()

B：标签对象.classList.contains()

▶ 答案

目录

学习目标

1. 掌握 axios 相关参数，从服务器获取并解析展示数据
2. 掌握接口文档的查看和使用
3. 掌握在浏览器的 network 面板中查看请求和响应的内容
4. 了解请求和响应报文的组成部分

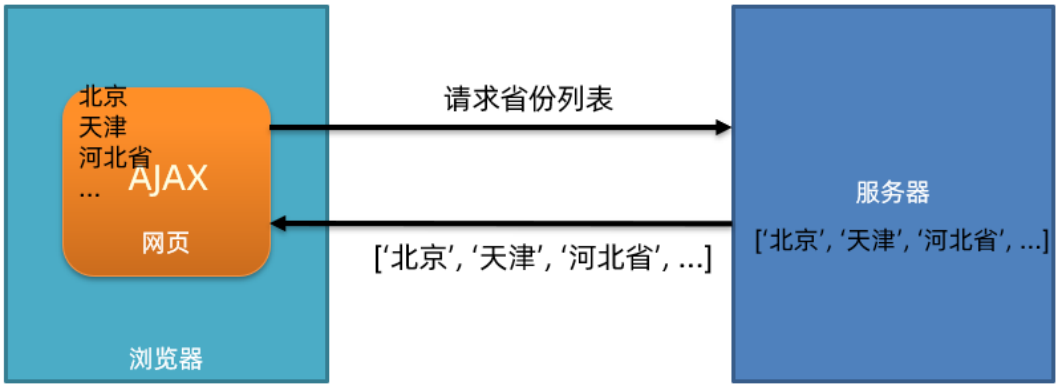
01.AJAX 概念和 axios 使用

目标

了解 AJAX 概念并掌握 axios 库基本使用

讲解

1. 什么是 AJAX ? mdn
   - 使用浏览器的 XMLHttpRequest 对象 与服务器通信
   - 浏览器网页中，使用 AJAX技术（XHR对象）发起获取省份列表数据的请求，服务器代码响应准备好的省份列表数据给前端，前端拿到数据数组以后，展示到网页



2. 什么是服务器?
   - 可以暂时理解为提供数据的一台电脑
3. 为何学 AJAX ?
   - 以前我们的数据都是写在代码里固定的, 无法随时变化
   - 现在我们的数据可以从服务器上进行获取，让数据变活
4. 怎么学 AJAX ?
   - 这里使用一个第三方库叫 axios, 后续在学习 XMLHttpRequest 对象了解 AJAX 底层原理
   - 因为 axios 库语法简单，让我们有更多精力关注在与服务器通信上，而且后续 Vue，React 学习中，也使用 axios 库与服务器通信
5. 需求：从服务器获取省份列表数据，展示到页面上（体验 axios 语法的使用）

   > 获取省份列表数据 - 目标资源地址：http://hmajax.itheima.net/api/province

   - 完成效果：

6. 接下来讲解 axios 语法，步骤：

7. 引入 axios.js 文件到自己的网页中

> axios.js文件链接: https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js

8. 明确axios函数的使用语法

```
axios({
  url: '目标资源地址'
}).then((result) => {
  // 对服务器返回的数据做后续处理
})
```

> 注意：请求的 url 地址, 就是标记资源的网址
>
> 注意：then 方法这里先体验使用，由来后续会讲到

9. 对应代码

```
<!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>AJAX概念和axios使用</title>
</head>

<body>
  <!--
    axios库地址：https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js
    省份数据地址：http://hmajax.itheima.net/api/province

    目标：使用axios库，获取省份列表数据，展示到页面上
    1. 引入axios库
  -->
```

```
  <p class="my-p"></p>
  <script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js">
</script>
  <script>
    // 2. 使用axios函数
    axios({
      url: 'http://hmajax.itheima.net/api/province'
    }).then(result => {
      console.log(result)
      // 好习惯：多打印，确认属性名
      console.log(result.data.list)
      console.log(result.data.list.join('<br>'))
      // 把准备好省份列表，插入到页面
      document.querySelector('.my-p').innerHTML =
result.data.list.join('<br>')
    })
  </script>
</body>

</html>
```

小结

1. AJAX 有什么用?

   ▶ 答案

2. AJAX 如何学:

   ▶ 答案

3. 这一节 axios 体验步骤（语法）?

   ▶ 答案

02.认识 URL

目标

了解 URL 的组成和作用

讲解

1. 为什么要认识 URL ? mdn
   - 虽然是后端给我的一个地址，但是哪部分标记的是服务器电脑，哪部分标记的是资源呢？所以为了和服务器有效沟通我们要认识一下
2. 什么是 URL ?

- 统一资源定位符，简称网址，用于定位网络中的资源（资源指的是：网页，图片，数据，视频，音频等等）

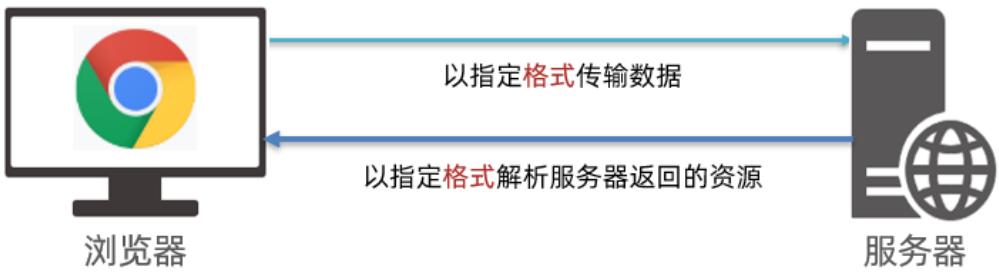
  统一资源定位符（英语：Uniform Resource Locator，缩写：**URL**，或称**统一资源定位器**、**定位地址**、**URL地址**[1]）俗称网页地址，简称**网址**，是因特网上标准的资源的地址（Address），如同在网络上的门牌。它最初是由蒂姆·伯纳斯-李发明用来作为万维网的地址，现在它已经被万维网联盟编制为因特网标准RFC 1738。

3. URL 的组成?

   - 协议，域名，资源路径（URL 组成有很多部分，我们先掌握这3个重要的部分即可）

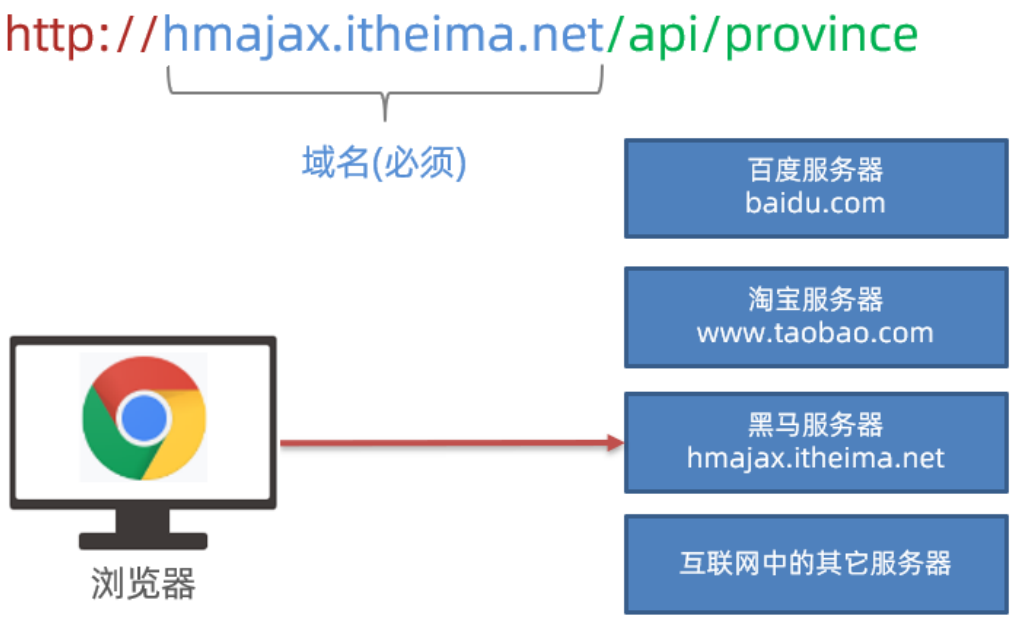
4. 什么是 http 协议？

   - 叫超文本传输协议，规定了浏览器和服务器传递数据的格式（而格式具体有哪些稍后我们就会学到）

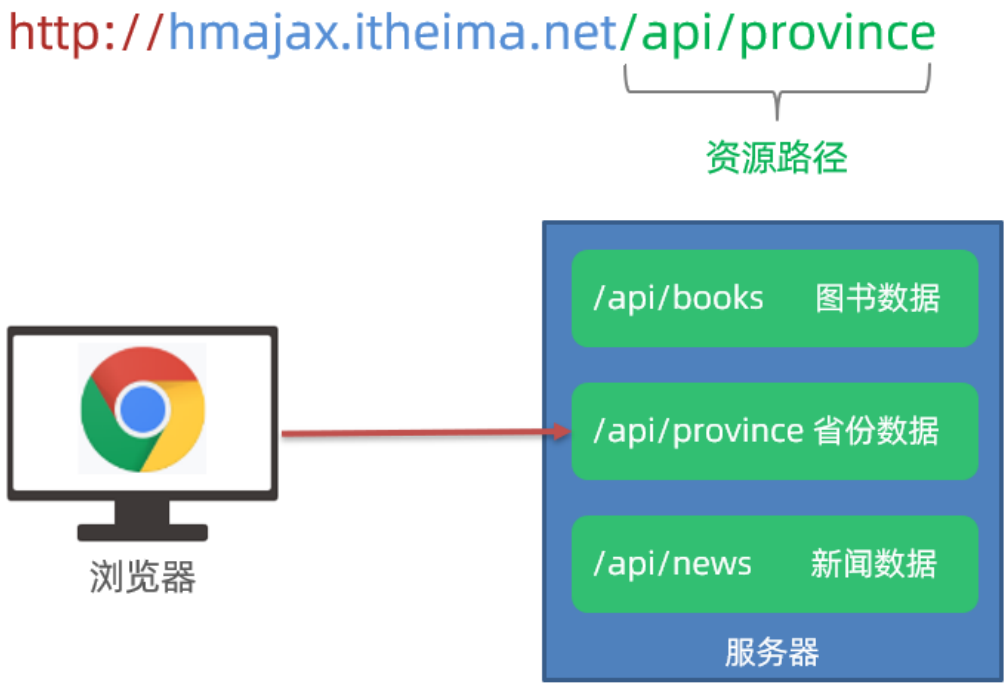


5. 什么是域名？

   - 标记服务器在互联网当中的方位，网络中有很多服务器，你想访问哪一台，就需要知道它的域名才可以



6. 什么是资源路径？
    - 一个服务器内有多个资源，用于标识你要访问的资源具体的位置



7. 接下来做个需求，访问新闻列表的 URL 网址，打印新闻数据
    - 效果图如下：

> 新闻列表数据 URL 网址：http://hmajax.itheima.net/api/news

```
axios({
  url: 'http://hmajax.itheima.net/api/news'
}).then(result => {
  console.log(result)
})
```

> url解释：从黑马服务器使用http协议，访问/api/news路径下的新闻列表资源

小结

1. URL 是什么?

   ▶ 答案

2. 请解释这个 URL，每个部分作用?

   http://hmajax.itheima.net/api/news

   ▶ 答案

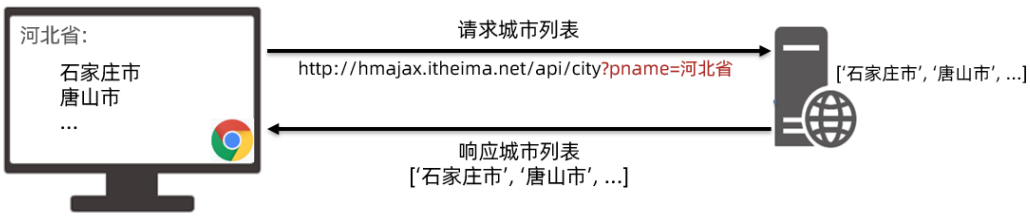
03.URL 查询参数

目标

掌握-通过URL传递查询参数，获取匹配的数据

讲解

1. 什么是查询参数？
   - 携带给服务器额外信息，让服务器返回我想要的某一部分数据而不是全部数据
   - 举例：查询河北省下属的城市列表，需要先把河北省传递给服务器



2. 查询参数的语法？
   - 在 url 网址后面用?拼接格式：http://xxxx.com/xxx/xxx?参数名1=值1&参数名2=值2
   - 参数名一般是后端规定的，值前端看情况传递即可
3. axios 如何携带查询参数?
   - 使用 params 选项即可

```
axios({
  url: '目标资源地址',
  params: {
    参数名: 值
  }
}).then(result => {
  // 对服务器返回的数据做后续处理
})
```

> 查询城市列表的 url地址：http://hmajax.itheima.net/api/city
>
> 参数名：pname （值要携带省份名字）

4. 需求：获取“河北省”下属的城市列表，展示到页面，对应代码：

```
<!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>查询参数</title>
</head>
<body>
  <!--
    城市列表: http://hmajax.itheima.net/api/city
    参数名: pname
    值: 省份名字
  -->
  <p></p>
  <script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js">
</script>
  <script>
    axios({
      url: 'http://hmajax.itheima.net/api/city',
      // 查询参数
      params: {
        pname: '辽宁省'
      }
    }).then(result => {
      console.log(result.data.list)
      document.querySelector('p').innerHTML =
result.data.list.join('<br>')
    })
  </script>
</body>
</html>
```

小结

1. URL 查询参数有什么用?

   ▶ 答案

2. axios 要如何携带查询参数?

   ▶ 答案

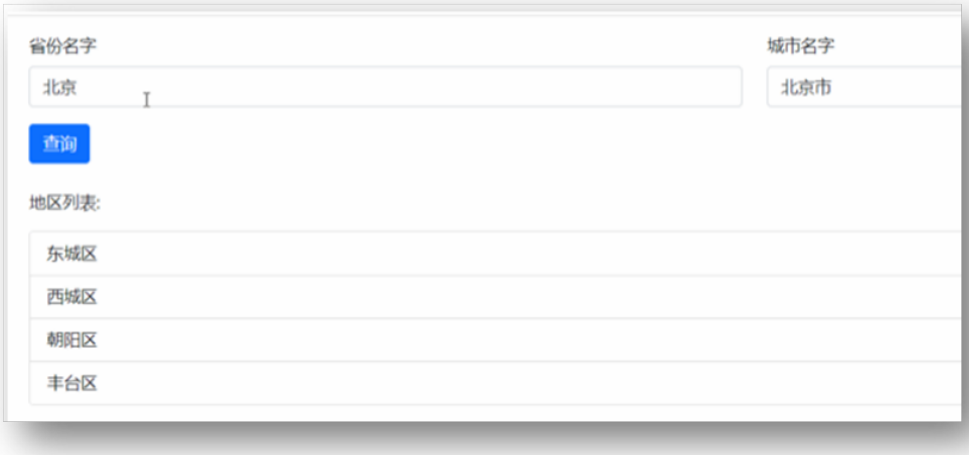
04.案例-查询-地区列表

目标

巩固查询参数的使用，并查看多对查询参数如何传递

讲解

1. 需求：根据输入的省份名字和城市名字，查询下属地区列表
   - 完成效果如下：



   - 相关参数

     > 查询地区: http://hmajax.itheima.net/api/area
     >
     > 参数名：
     >
     > pname：省份名字
     >
     > cname：城市名字

2. 正确代码如下：

```
/*
      获取地区列表: http://hmajax.itheima.net/api/area
      查询参数:
        pname: 省份或直辖市名字
```

```
        cname: 城市名字
    */
// 目标：根据省份和城市名字，查询地区列表
// 1. 查询按钮-点击事件
document.querySelector('.sel-btn').addEventListener('click', () => {
    // 2. 获取省份和城市名字
    let pname = document.querySelector('.province').value
    let cname = document.querySelector('.city').value

    // 3. 基于axios请求地区列表数据
    axios({
        url: 'http://hmajax.itheima.net/api/area',
        params: {
            pname,
            cname
        }
    }).then(result => {
        // console.log(result)
        // 4. 把数据转li标签插入到页面上
        let list = result.data.list
        console.log(list)
        let theLi = list.map(areaName => `<li class="list-group-item">${areaName}</li>`).join('')
        console.log(theLi)
        document.querySelector('.list-group').innerHTML = theLi
    })
})
```

小结

1. ES6 对象属性和值简写的前提是什么?

   ▶ 答案

05.常用请求方法和数据提交

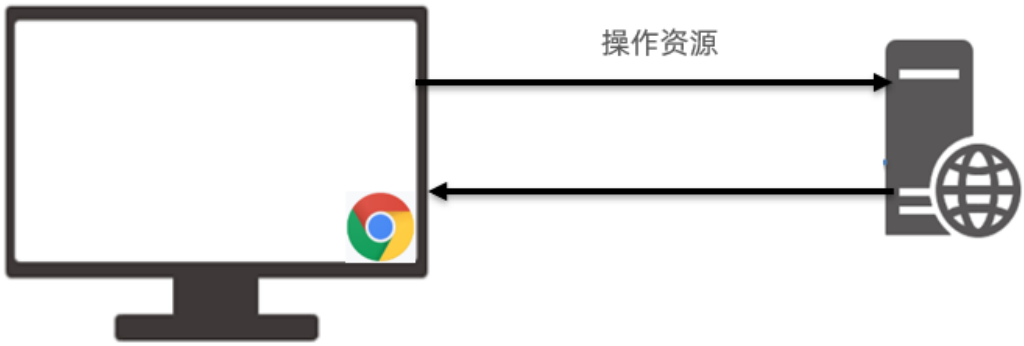
目标

掌握如何向服务器提交数据，而不单单是获取数据

讲解

1. 想要提交数据，先来了解什么是请求方法
   - 请求方法是一些固定单词的英文，例如：GET，POST，PUT，DELETE，PATCH（这些都是http协议规定的），每个单词对应一种对服务器资源要执行的操作

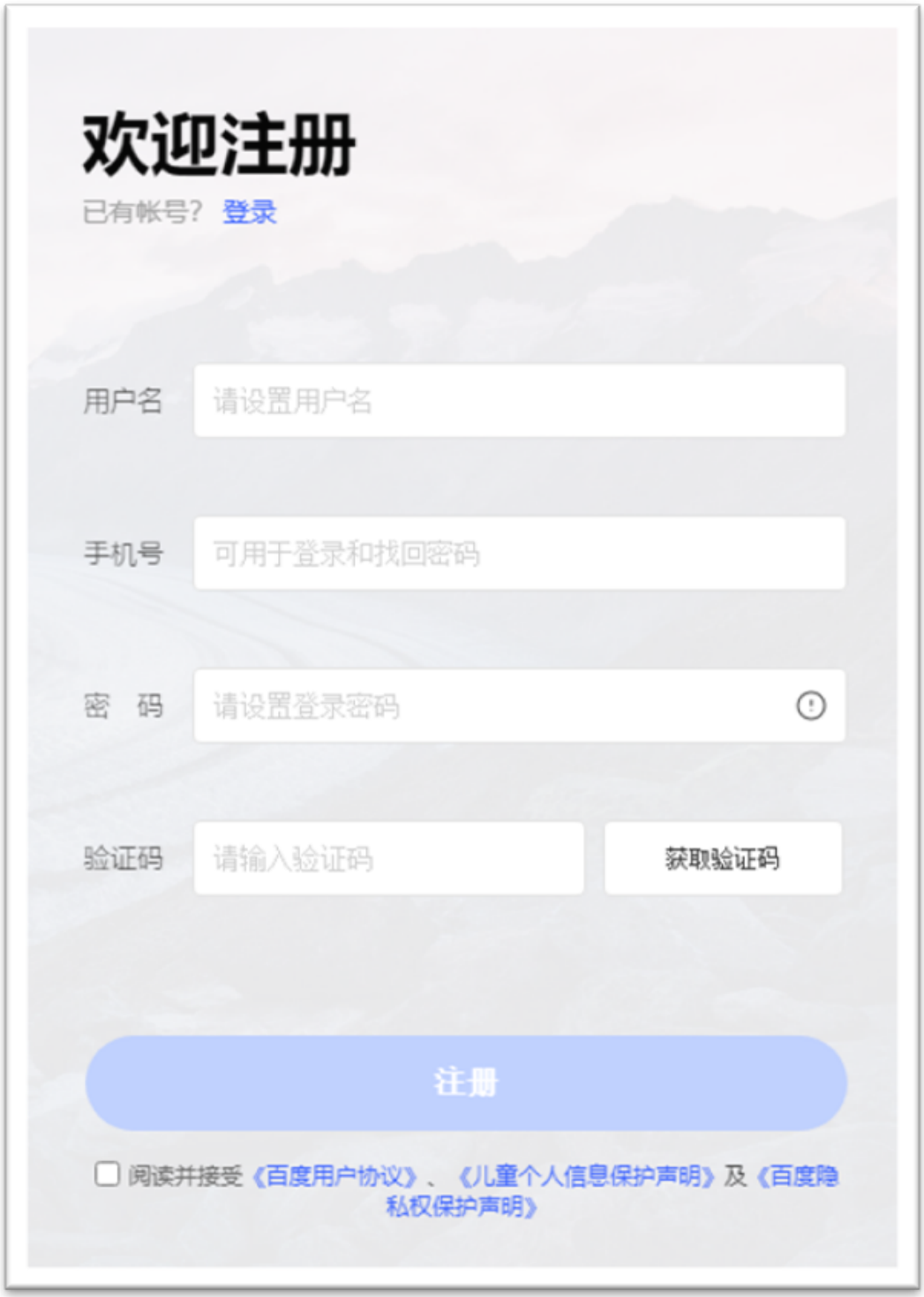
请求方法	操作
GET	获取数据
POST	数据提交
PUT	修改数据（全部）
DELETE	删除数据
PATCH	修改数据（部分）



- 前面我们获取数据其实用的就是GET请求方法，但是axios内部设置了默认请求方法就是GET，我们就没有写
- 但是提交数据需要使用POST请求方法

2. 什么时候进行数据提交呢?

- 例如：多端要查看同一份订单数据，或者使用同一个账号进行登录，那订单/用户名+密码，就需要保存在服务器上，随时随地进行访问

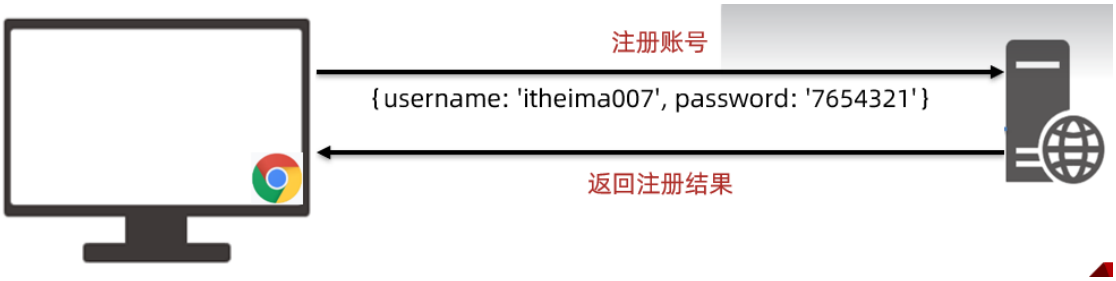


3. axios 如何提交数据到服务器呢?
    - 需要学习，method 和 data 这2个新的选项了（大家不用担心，这2个学完，axios常用的选项就都学完了）

```
axios({
  url: '目标资源地址',
  method: '请求方法',
  data: {
    参数名: 值
  }
}).then(result => {
  // 对服务器返回的数据做后续处理
})
```

4. 需求：注册账号，提交用户名和密码到服务器保存

> 注册用户 URL 网址：http://hmajax.itheima.net/api/register
>
> 请求方法：POST
>
> 参数名：
>
> username：用户名（要求中英文和数字组成，最少8位）
>
> password：密码（最少6位）



5. 正确代码如下：

```
/*
  注册用户：http://hmajax.itheima.net/api/register
  请求方法：POST
  参数名：
    username：用户名（中英文和数字组成，最少8位）
    password：密码  （最少6位）

  目标：点击按钮，通过axios提交用户和密码，完成注册
*/
document.querySelector('.btn').addEventListener('click', () => {
  axios({
    url: 'http://hmajax.itheima.net/api/register',
    method: 'POST',
    data: {
      username: 'itheima007',
      password: '7654321'
    }
  })
})
```

小结

1. 请求方法最常用的是哪2个，分别有什么作用?

▶ 答案

2. axios 的核心配置项?

▶ 答案

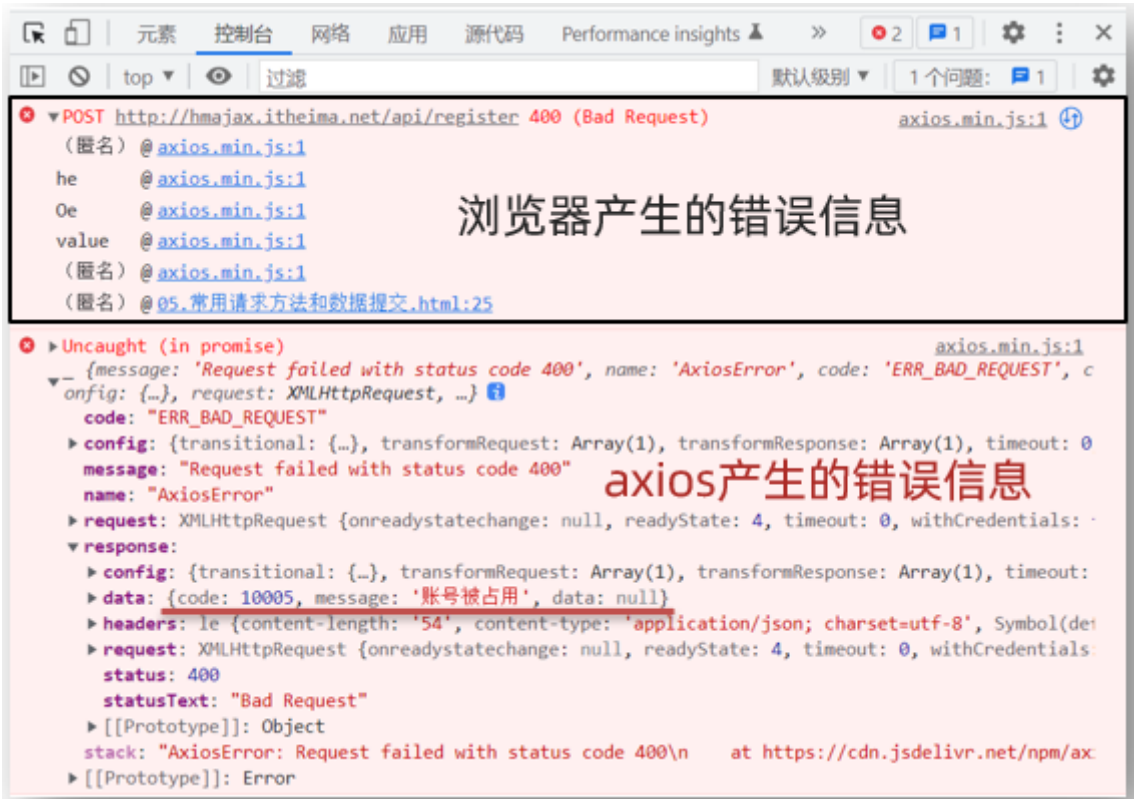
06.axios 错误处理

目标

掌握接收 axios 响应错误信息的处理语法

讲解

1. 如果注册相同的用户名，则会遇到注册失败的请求，也就是 axios 请求响应失败了，你会在控制台看到如图的错误：



2. 在 axios 语法中要如何处理呢?
    - 因为，普通用户不会去控制台里看错误信息，我们要编写代码拿到错误并展示给用户在页面上
3. 使用 axios 的 catch 方法，捕获这次请求响应的错误并做后续处理，语法如下：

```
axios({
  // ...请求选项
}).then(result => {
  // 处理成功数据
}).catch(error => {
  // 处理失败错误
})
```

4. 需求：再次重复注册相同用户名，提示用户注册失败的原因

5. 对应代码

```
document.querySelector('.btn').addEventListener('click', () => {
    axios({
      url: 'http://hmajax.itheima.net/api/register',
      method: 'post',
      data: {
        username: 'itheima007',
        password: '7654321'
      }
    }).then(result => {
      // 成功
      console.log(result)
    }).catch(error => {
      // 失败
      // 处理错误信息
      console.log(error)
      console.log(error.response.data.message)
      alert(error.response.data.message)
    })
})
```

小结

1. axios 如何拿到请求响应失败的信息?

▶ 答案

07.HTTP 协议-请求报文

目标

了解 HTTP 协议中，请求报文的组成和作用

讲解

1. 首先，HTTP 协议规定了浏览器和服务器返回内容的格式
2. 请求报文：是浏览器按照协议规定发送给服务器的内容，例如刚刚注册用户时，发起的请求报文：


```
POST http://hmajax.itheima.net/api/register HTTP/1.1
Host: hmajax.itheima.net
Connection: keep-alive
Content-Length: 46
Accept: application/json, text/plain, */*
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/107.0.0.0
Safari/537.36
Content-Type: application/json
Origin: http://127.0.0.1:5500
Referer: http://127.0.0.1:5500/
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.9

{"username":"itheima007","password":"7654321"}
```

3. 这里的格式包含：
    - 请求行：请求方法，URL，协议
    - 请求头：以键值对的格式携带的附加信息，比如：Content-Type（指定了本次传递的内容类型）
    - 空行：分割请求头，空行之后的是发送给服务器的资源
    - 请求体：发送的资源
4. 我们切换到浏览器中，来看看刚才注册用户发送的这个请求报文以及内容去哪里查看呢
5. 代码：直接在上个代码基础上复制，然后运行查看请求报文对应关系即可

小结

1. 浏览器发送给服务器的内容叫做，请求报文
2. 请求报文的组成是什么?

    ▶ 答案
3. 通过 Chrome 的网络面板如何查看请求体?

08.请求报文-错误排查

目标

了解学习了查看请求报文之后的作用，可以用来辅助错误排查

讲解

1. 学习了查看请求报文有什么用呢?
    - 可以用来确认我们代码发送的请求数据是否真的正确
2. 配套模板代码里，对应 08 标题文件夹里是我同桌的代码，它把登录也写完了，但是无法登录，我们来到模板代码中，找到运行后，在不逐行查看代码的情况下，查看请求报文，看看它登录提交的相关信息对不对，帮他找找问题出现的原因
3. 发现请求体数据有问题，往代码中定位，找到类名写错误了
4. 代码：在配套文件夹素材里，找到需要对应代码，直接运行，根据报错信息，找到错误原因

小结

1. 学会了查看请求报文，对实际开发有什么帮助呢?

    ▶ 答案

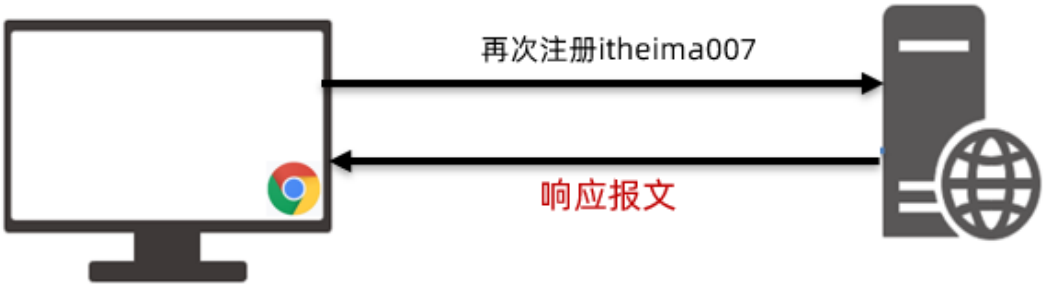
09.HTTP 协议-响应报文

目标

了解响应报文的组成

讲解

1. 响应报文：是服务器按照协议固定的格式，返回给浏览器的内容




```
HTTP/1.1 400 Bad Request
Server: nginx
Date: Wed, 09 Nov 2022 13:26:06 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 54
Connection: keep-alive
Vary: Origin
Access-Control-Allow-Origin: http://127.0.0.1:5500
Access-Control-Allow-Credentials: true
x-frame-options: SAMEORIGIN
x-xss-protection: 1; mode=block
x-content-type-options: nosniff
x-download-options: noopen
x-readtime: 16

{"code":10005,"message":"账号被占用","data":null}
```

2. 响应报文的组成：
    - 响应行（状态行）：协议，HTTP响应状态码，状态信息
    - 响应头：以键值对的格式携带的附加信息，比如：Content-Type（告诉浏览器，本次返回的内容类型）
    - 空行：分割响应头，控制之后的是服务器返回的资源
    - 响应体：返回的资源
3. HTTP 响应状态码：
    - 用来表明请求是否成功完成
    - 例如：404（客户端要找的资源，在服务器上不存在）

状态码	说明
1xx	信息
2xx	成功
3xx	重定向消息
4xx	客户端错误
5xx	服务端错误

小结

1. 响应报文的组成?
   ▶ 答案
2. HTTP 响应状态码是做什么的?
   ▶ 答案

10.接口文档

目标

掌握接口文档的使用，配合 axios 与服务器进行数据交互

讲解

1. 接口文档：描述接口的文章（一般是后端工程师，编写和提供）
2. 接口：指的使用 AJAX 和 服务器通讯时，使用的 URL，请求方法，以及参数，例如：AJAX阶段接口文档
3. 例如：获取城市列表接口样子

获取-城市列表

GET http://hmajax.itheima.net/api/city

获取-城市列表

请求参数

参数名	位置	类型	必填	说明
pname	query	string	是	示例值：辽宁省

4. 需求：打开 AJAX 阶段接口文档，查看登录接口，并编写代码，完成一次登录的效果吧
5. 代码如下：

```
document.querySelector('.btn').addEventListener('click', () => {
  // 用户登录
  axios({
    url: 'http://hmajax.itheima.net/api/login',
    method: 'post',
    data: {
      username: 'itheima007',
      password: '7654321'
    }
  })
})
```

小结

1. 接口文档是什么?

   ▶ 答案
2. 接口文档里包含什么?

   ▶ 答案

11.案例-用户登录-主要业务

目标

尝试通过页面获取用户名和密码，进行登录

讲解

1. 先来到备课代码中，运行完成的页面，查看要完成的登录效果（登录成功和失败）
2. 需求：编写代码，查看接口文档，填写相关信息，完成登录业务
3. 分析实现的步骤
    1. 点击登录，获取并判断用户名和长度
    2. 提交数据和服务器通信
    3. 提示信息，反馈给用户（这节课先来完成前 2 个步骤）

4. 代码如下：

```
// 目标1：点击登录时，用户名和密码长度判断，并提交数据和服务器通信

// 1.1 登录-点击事件
document.querySelector('.btn-login').addEventListener('click', () => {
```

```
  // 1.2 获取用户名和密码
  const username = document.querySelector('.username').value
  const password = document.querySelector('.password').value
  // console.log(username, password)

  // 1.3 判断长度
  if (username.length < 8) {
    console.log('用户名必须大于等于8位')
    return // 阻止代码继续执行
  }
  if (password.length < 6) {
    console.log('密码必须大于等于6位')
    return // 阻止代码继续执行
  }

  // 1.4 基于axios提交用户名和密码
  // console.log('提交数据到服务器')
  axios({
    url: 'http://hmajax.itheima.net/api/login',
    method: 'POST',
    data: {
      username,
      password
    }
  }).then(result => {
    console.log(result)
    console.log(result.data.message)
  }).catch(error => {
    console.log(error)
    console.log(error.response.data.message)
  })
})
```

小结

1. 总结下用户登录案例的思路?

▶ 答案

12.案例-用户登录-提示信息

目标

根据准备好的提示标签和样式，给用户反馈提示

讲解

1. 需求：使用提前准备好的提示框，来把登录成功/失败结果提示给用户

2. 使用提示框，反馈提示消息，因为有4处地方需要提示框，所以封装成函数
    1. 获取提示框
    2. 封装提示框函数，重复调用，满足提示需求

       功能：
        1. 显示提示框
        2. 不同提示文字msg，和成功绿色失败红色isSuccess参数（true成功，false失败）
        3. 过2秒后，让提示框自动消失
3. 对应提示框核心代码：

```
/**
 * 2.2 封装提示框函数，重复调用，满足提示需求
 * 功能：
 * 1. 显示提示框
 * 2. 不同提示文字msg，和成功绿色失败红色isSuccess（true成功，false失败）
 * 3. 过2秒后，让提示框自动消失
*/
function alertFn(msg, isSuccess) {
  // 1> 显示提示框
  myAlert.classList.add('show')
```

```
11
12    // 2> 实现细节
13    myAlert.innerText = msg
14    const bgStyle = isSuccess ? 'alert-success' : 'alert-danger'
15    myAlert.classList.add(bgStyle)
16
17    // 3> 过2秒隐藏
18    setTimeout(() => {
19      myAlert.classList.remove('show')
20      // 提示：避免类名冲突，重置背景色
21      myAlert.classList.remove(bgStyle)
22    }, 2000)
23  }
```

小结

1. 我们什么时候需要封装函数?

   ▶ 答案
2. 如何封装一个函数呢?

   ▶ 答案
3. 我们的提示框是如何控制出现/隐藏的?

   ▶ 答案

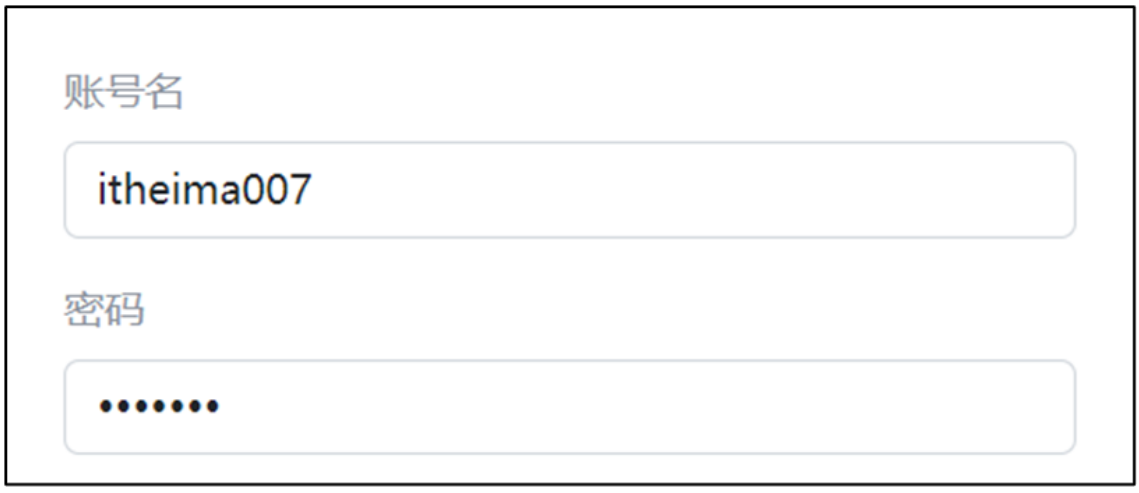
13.form-serialize 插件

目标

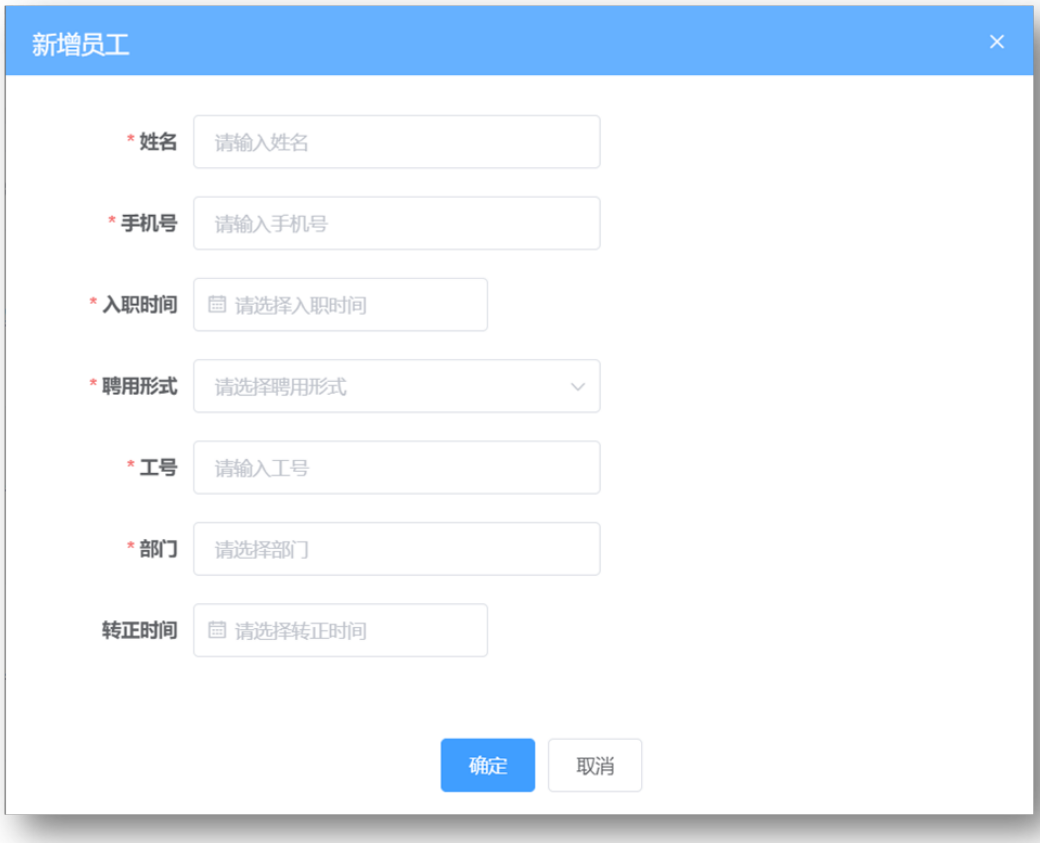
使用 form-serialize 插件，快速收集目标表单范围内表单元素的值

讲解

1. 我们前面收集表单元素的值，是一个个标签获取的



2. 如果一套表单里有很多很多表单元素，如何一次性快速收集出来呢?



3. 使用 form-serialize 插件提供的 serialize 函数就可以办到
4. form-serialize 插件语法：
    1. 引入 form-serialize 插件到自己网页中
    2. 使用 serialize 函数
        - 参数1：要获取的 form 表单标签对象（要求表单元素需要有 name 属性-用来作为收集的数据中属性名）
        - 参数2：配置对象
            - hash：
                - true - 收集出来的是一个 JS 对象结构
                - false - 收集出来的是一个查询字符串格式
            - empty：
                - true - 收集空值
                - false - 不收集空值
5. 需求：收集登录表单里用户名和密码
6. 对应代码：

```
<!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>form-serialize插件使用</title>
</head>
```

```
10
11  <body>
12    <form action="javascript:;" class="example-form">
13      <input type="text" name="username">
14      <br>
15      <input type="text" name="password">
16      <br>
17      <input type="button" class="btn" value="提交">
18    </form>
19    <!--
20      目标：在点击提交时，使用form-serialize插件，快速收集表单元素值
21      1. 把插件引入到自己网页中
22    -->
23    <script src="./lib/form-serialize.js"></script>
24    <script>
25      document.querySelector('.btn').addEventListener('click', () => {
26        /**
27         * 2. 使用serialize函数，快速收集表单元素的值
28         * 参数1：要获取哪个表单的数据
29         *  表单元素设置name属性，值会作为对象的属性名
30         *  建议name属性的值，最好和接口文档参数名一致
31         * 参数2：配置对象
32         *  hash 设置获取数据结构
33         *    - true：JS对象（推荐）一般请求体里提交给服务器
34         *    - false: 查询字符串
35         *  empty 设置是否获取空值
36         *    - true: 获取空值（推荐）数据结构和标签结构一致
37         *    - false：不获取空值
38        */
39        const form = document.querySelector('.example-form')
40        const data = serialize(form, { hash: true, empty: true })
41        // const data = serialize(form, { hash: false, empty: true })
42        // const data = serialize(form, { hash: true, empty: false })
43        console.log(data)
44      })
45    </script>
46  </body>
47
48  </html>
```

小结

1. 我们什么时候使用 form-serialize 插件?

   ▶ 答案

2. 如何使用 form-serialize 插件?

   ▶ 答案

3. 配置对象中 hash 和 empty 有什么用?

   ▶ 答案

14.案例-用户登录-form-serialize

目标

尝试通过 form-serialize 重新修改用户登录案例-收集用户名和密码

讲解

1. 基于模板代码，使用 form-serialize 插件来收集用户名和密码
2. 在原来的代码基础上修改即可
    1. 先引入插件

```
<!-- 3.1 引入插件 -->
<script src="./lib/form-serialize.js"></script>
```

    2. 然后修改代码

```
// 3.2 使用serialize函数，收集登录表单里用户名和密码
const form = document.querySelector('.login-form')
const data = serialize(form, { hash: true, empty: true })
console.log(data)
// {username: 'itheima007', password: '7654321'}
const { username, password } = data
```

小结

1. 如何把一个第三方插件使用在已完成的案例中?

▶ 答案

今日重点(必须会)

1. axios 的配置项有哪几个，作用分别是什么?
2. 接口文档都包含哪些信息?
3. 在浏览器中如何查看查询参数/请求体，以及响应体数据?
4. 请求报文和响应报文由几个部分组成，每个部分的作用?

今日作业(必完成)

参考作业文件夹的md要求

参考文献

1. 客户端->百度百科
2. 浏览器解释->百度百科
3. 服务器解释->百度百科
4. url解释->百度百科
5. http协议->百度百科
6. 主机名->百度百科
7. 端口号->百度百科
8. Ajax解释->百度-懂啦
9. Ajax解释->MDN解释Ajax是与服务器通信而不只是请求
10. axios->百度(可以点击播报听读音)
11. axios(github)地址
12. axios官方推荐官网
13. axios(npmjs)地址
14. GET和POST区别->百度百科
15. 报文讲解->百度百科
16. HTTP状态码->百度百科
17. 接口概念->百度百科

Day02_AJAX综合案例

知识点自测

1. 以下代码运行结果是什么？（考察扩展运算符的使用）

```
const result = {
  name: '老李',
  age: 18
}
const obj = {
  ...result
}
console.log(obj.age)
```

A：报错

B：18

▶ 答案

2. 什么是事件委托?

A：只能把单击事件委托给父元素绑定

B：可以把能冒泡的事件，委托给已存在的向上的任意标签元素绑定

▶ 答案

3. 事件对象e.target作用是什么?

A：获取到这次触发事件相关的信息

B：获取到这次触发事件目标标签元素

▶ 答案

4. 如果获取绑定在标签上自定义属性的值10?

```
<div data-code="10">西游记</div>
```

A：div标签对象.innerHTML

B：div标签对象.dataset.code

C：div标签对象.code

▶ 答案

5. 哪个方法可以判断目标标签是否包含指定的类名?

```
<div class="my-div title info"></div>
```

A: div标签对象.className === 'title'

B: div标签对象.classList.contains('title')

▶ 答案

6. 伪数组取值哪种方式是正确的?

```
let obj = { 0: '老李', 1: '老刘' }
```

A: obj.0

B: obj[0]

▶ 答案

7. 以下哪个选项可以，往本地存储键为'bgImg'，值为图片url网址的代码

A：localStorage.setItem('bgImg')

B：localStorage.getItem('bgImg')

C：localStorage.setItem('bgImg', '图片url网址')

D：localStorage.getItem('bgImg', '图片url网址')

▶ 答案

8. 以下代码运行结果是?

```
const obj = {
  username: '老李',
  age: 18,
  sex: '男'
}
Object.keys(obj)
```

A：代码报错

B：[username, age, sex]

C：["username", "age", "sex"]

D：["老李", 18, "男"]

▶ 答案

9. 下面哪个选项可以把数字字符串转成数字类型?

A：+'10'

B：'10' + 0

▶ 答案

10. 以下代码运行后的结果是什么？（考察逻辑与的短路特性）

```
const age = 18
const result1 = (age || '有年龄')

const sex = ''
const result2 = sex || '没有性别'
```

A：报错，报错

B：18，没有性别

C：有年龄，没有性别

D：18，''

▶ 答案

目录

- 案例-网站换肤
- 案例-个人信息设置

学习目标

> 今天主要就是练，巩固 axios 的使用

1. 完成案例-图书管理系统（增删改查）经典业务
2. 掌握图片上传的思路
3. 完成案例-网站换肤并实现图片地址缓存
4. 完成案例-个人信息设置

01.案例_图书管理-介绍

目标

案例-图书管理-介绍（介绍要完成的效果和练习到的思维）

讲解

1. 打开备课代码运行图书管理案例效果-介绍要完成的增删改查业务效果和 Bootstrap 弹框使用

2. 分析步骤和对应的视频模块

- 先学习 Bootstrap 弹框的使用（因为添加图书和编辑图书需要这个窗口来承载图书表单）
- 先做渲染图书列表（这样做添加和编辑以及删除可以看到数据变化，所以先做渲染）
- 再做新增图书功能
- 再做删除图书功能
- 再做编辑图书功能（注意：编辑和新增图书是2套弹框-后续做项目我们再用同1个弹框）

小结

1. 做完这个案例我们将会有什么收获呢?

   ▶ 答案

02.Bootstrap 弹框_属性控制

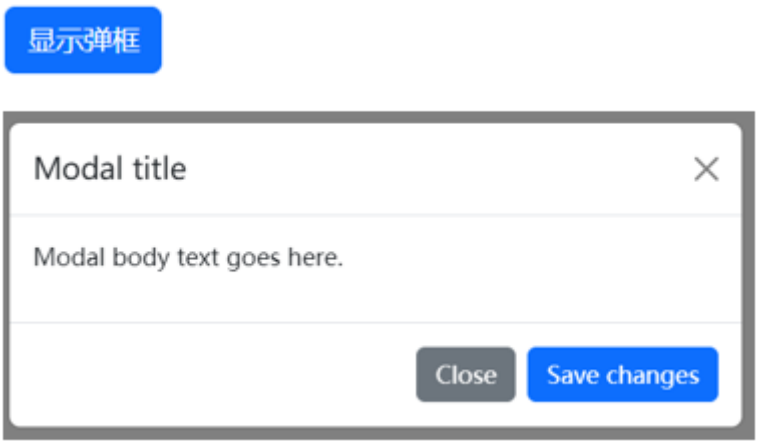
目标

使用属性方式控制 Bootstarp 弹框的显示和隐藏

讲解

1. 什么是 Bootstrap 弹框?
   - 不离开当前页面，显示单独内容，供用户操作

2. 需求：使用 Bootstrap 弹框，先做个简单效果，点击按钮，让弹框出现，点击 X 和 Close 让弹框隐藏



3. 如何使用 Bootstrap 弹框呢?

    1. 先引入 bootstrap.css 和 bootstrap.js 到自己网页中
    2. 准备弹框标签，确认结构（可以从 Bootstrap 官方文档的 Modal 里复制基础例子）- 运行到网页后，逐一对应标签和弹框每个部分对应关系
    3. 通过自定义属性，通知弹框的显示和隐藏，语法如下：

```
<button data-bs-toggle="modal" data-bs-target="css选择器">
  显示弹框
</button>

<button data-bs-dismiss="modal">Close</button>
```

4. 去代码区实现一下

```
<!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>Bootstrap 弹框</title>
  <!-- 引入bootstrap.css -->
  <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/css/bootstrap.mi
n.css" rel="stylesheet">
</head>

<body>
  <!--
    目标：使用Bootstrap弹框
    1. 引入bootstrap.css 和 bootstrap.js
    2. 准备弹框标签，确认结构
    3. 通过自定义属性，控制弹框的显示和隐藏
   -->
  <button type="button" class="btn btn-primary" data-bs-toggle="modal"
data-bs-target=".my-box">
    显示弹框
  </button>

  <!--
    弹框标签
    bootstrap的modal弹框，添加modal类名（默认隐藏）
   -->
  <div class="modal my-box" tabindex="-1">
    <div class="modal-dialog">
      <!-- 弹框-内容 -->
      <div class="modal-content">
        <!-- 弹框-头部 -->
        <div class="modal-header">
          <h5 class="modal-title">Modal title</h5>
```

```
          <button type="button" class="btn-close" data-bs-dismiss="modal" aria-label="Close"></button>
        </div>
        <!-- 弹框-身体 -->
        <div class="modal-body">
          <p>Modal body text goes here.</p>
        </div>
        <!-- 弹框-底部 -->
        <div class="modal-footer">
          <button type="button" class="btn btn-secondary" data-bs-dismiss="modal">Close</button>
          <button type="button" class="btn btn-primary">Save changes</button>
        </div>
      </div>
    </div>
  </div>

  <!-- 引入bootstrap.js -->
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/js/bootstrap.min.js"></script>
</body>

</html>
```

小结

1. 用哪个属性绑定来控制弹框显示呢?

   ▶ 答案

2. 用哪个属性来控制隐藏弹框呢?

   ▶ 答案

03.Bootstrap 弹框_JS控制

目标

使用 JS 方式控制 Bootstarp 弹框的显示和隐藏

讲解

1. 为什么需要 JS 方式控制呢?
   - 当我显示之前，隐藏之前，需要执行一些 JS 逻辑代码，就需要引入 JS 控制弹框显示/隐藏的方式了
   - 例如:
     - 点击编辑姓名按钮，在弹框显示之前，在输入框填入默认姓名
     - 点击保存按钮，在弹框隐藏之前，获取用户填入的名字并打印

2. 所以在现实和隐藏之前，需要执行 JS 代码逻辑，就使用 JS 方式 控制 Bootstrap 弹框显示和隐藏

语法如下：

```
// 创建弹框对象
const modalDom = document.querySelector('css选择器')
const modal = new bootstrap.Modal(modelDom)

// 显示弹框
modal.show()
// 隐藏弹框
modal.hide()
```

3. 去代码区实现一下

```
// 1. 创建弹框对象
const modalDom = document.querySelector('.name-box')
const modal = new bootstrap.Modal(modalDom)

// 编辑姓名->点击->赋予默认姓名->弹框显示
document.querySelector('.edit-btn').addEventListener('click', () => {
  document.querySelector('.username').value = '默认姓名'

  // 2. 显示弹框
  modal.show()
})

// 保存->点击->->获取姓名打印->弹框隐藏
document.querySelector('.save-btn').addEventListener('click', () => {
  const username = document.querySelector('.username').value
  console.log('模拟把姓名保存到服务器上', username)

  // 2. 隐藏弹框
  modal.hide()
})
```

小结

1. 什么时候用属性控制，什么时候用 JS 控制 Bootstrap 弹框的显示/隐藏?

   ▶ 答案

04.案例_图书管理_渲染列表

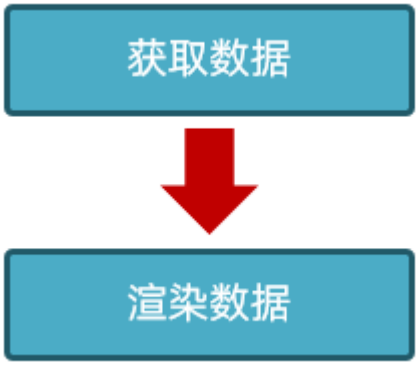
目标

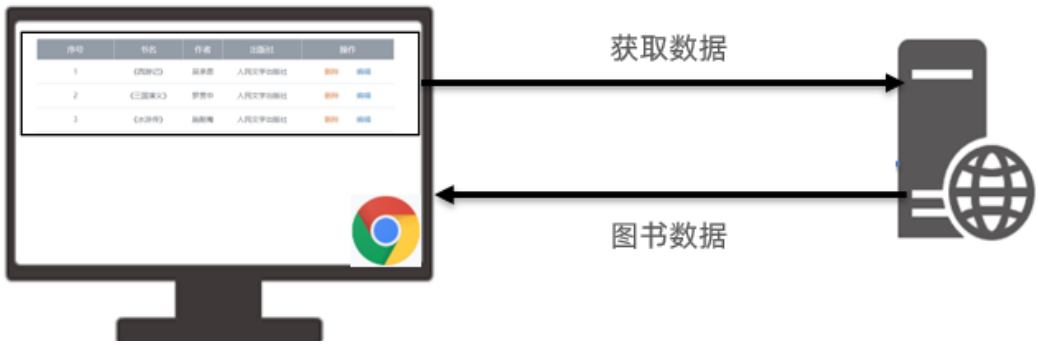
完成图书管理案例-图书列表数据渲染效果

讲解

1. 需求：基于 axios 获取到图书列表数据，并用 JS 代码渲染数据，到准备好的模板标签中

2. 步骤：
   1. 获取数据
   2. 渲染数据





3. 获取数据的时候，需要给自己起一个外号，为什么需要给自己起一个外号呢?

   - 我们所有人数据都来自同一个服务器上，为了区分每个同学不同的数据，需要大家设置一个外号告诉服务器，服务器就会返回你对应的图书数据了

4. 核心代码如下：

> 因为默认展示列表，新增，修改，删除后都要重新获取并刷新列表，所以把获取数据渲染数据的代码封装在一个函数内，方便复用

```
/**
 * 目标1：渲染图书列表
 *  1.1 获取数据
 *  1.2 渲染数据
 */
const creator = '老张'
// 封装-获取并渲染图书列表函数
function getBooksList() {
  // 1.1 获取数据
  axios({
    url: 'http://hmajax.itheima.net/api/books',
    params: {
      // 外号：获取对应数据
      creator
    }
  }).then(result => {
    // console.log(result)
    const bookList = result.data.data
    // console.log(bookList)
    // 1.2 渲染数据
    const htmlStr = bookList.map((item, index) => {
      return `<tr>
      <td>${index + 1}</td>
      <td>${item.bookname}</td>
      <td>${item.author}</td>
      <td>${item.publisher}</td>
      <td data-id=${item.id}>
        <span class="del">删除</span>
        <span class="edit">编辑</span>
      </td>
    </tr>`
    }).join('')
    // console.log(htmlStr)
    document.querySelector('.list').innerHTML = htmlStr
  })
```

```
}
// 网页加载运行，获取并渲染列表一次
getBooksList()
```

小结

1. 渲染数据列表的2个步骤是什么?

   ▶ 答案

05.案例_图书管理_新增图书

目标

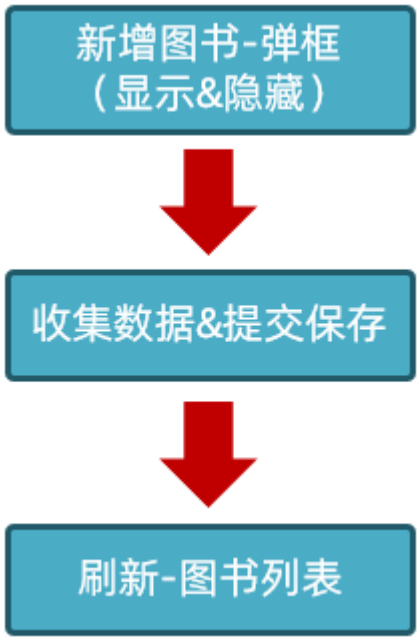
完成图书管理案例-新增图书需求

讲解

1. 需求：点击添加按钮，出现准备好的新增图书弹框，填写图书信息提交到服务器保存，并更新图书列表

2. 步骤：
   1. 新增弹框（控制显示和隐藏）（基于 Bootstrap 弹框和准备好的表单-用属性和 JS 方式控制）
   2. 在点击保存按钮时，收集数据&提交保存

3. 刷新-图书列表）（重新调用下之前封装的获取并渲染列表的函数）



3. 核心代码如下：

```
/**
 * 目标2：新增图书
 *  2.1 新增弹框->显示和隐藏
 *  2.2 收集表单数据，并提交到服务器保存
 *  2.3 刷新图书列表
 */
// 2.1 创建弹框对象
const addModalDom = document.querySelector('.add-modal')
const addModal = new bootstrap.Modal(addModalDom)
// 保存按钮->点击->隐藏弹框
document.querySelector('.add-btn').addEventListener('click', () => {
  // 2.2 收集表单数据，并提交到服务器保存
  const addForm = document.querySelector('.add-form')
  const bookObj = serialize(addForm, { hash: true, empty: true })
  // console.log(bookObj)
  // 提交到服务器
  axios({
    url: 'http://hmajax.itheima.net/api/books',
    method: 'POST',
    data: {
      ...bookObj,
      creator
    }
  }).then(result => {
    // console.log(result)
    // 2.3 添加成功后，重新请求并渲染图书列表
    getBooksList()
    // 重置表单
    addForm.reset()
    // 隐藏弹框
    addModal.hide()
  })
})
```

小结

1. 新增数据的3个步骤是什么?

   ▶ 答案

06.案例_图书管理_删除图书

目标

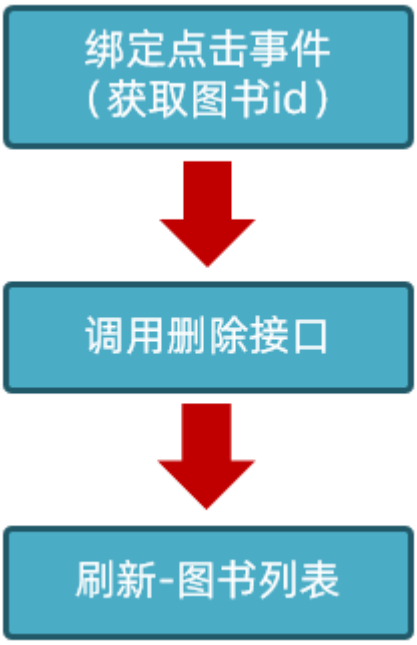
完成图书管理案例-删除图书需求

讲解

1. 需求：点击图书删除元素，删除当前图书数据

2. 步骤：

   1. 给删除元素，绑定点击事件（事件委托方式并判断点击的是删除元素才走删除逻辑代码），并获取到要删除的数据id
   2. 基于 axios 和接口文档，调用删除接口，让服务器删除这条数据
   3. 重新获取并刷新图书列表



3. 核心代码如下：

```
/**
 * 目标3：删除图书
 *  3.1 删除元素绑定点击事件->获取图书id
 *  3.2 调用删除接口
 *  3.3 刷新图书列表
 */
// 3.1 删除元素->点击（事件委托）
document.querySelector('.list').addEventListener('click', e => {
  // 获取触发事件目标元素
  // console.log(e.target)
  // 判断点击的是删除元素
  if (e.target.classList.contains('del')) {
    // console.log('点击删除元素')
    // 获取图书id（自定义属性id）
    const theId = e.target.parentNode.dataset.id
    // console.log(theId)
    // 3.2 调用删除接口
    axios({
      url: `http://hmajax.itheima.net/api/books/${theId}`,
      method: 'DELETE'
    }).then(() => {
      // 3.3 刷新图书列表
      getBooksList()
    })
  }
})
```

小结

1. 删除数据的步骤是什么?

   ▶ 答案

07-09.案例_图书管理_编辑图书

目标

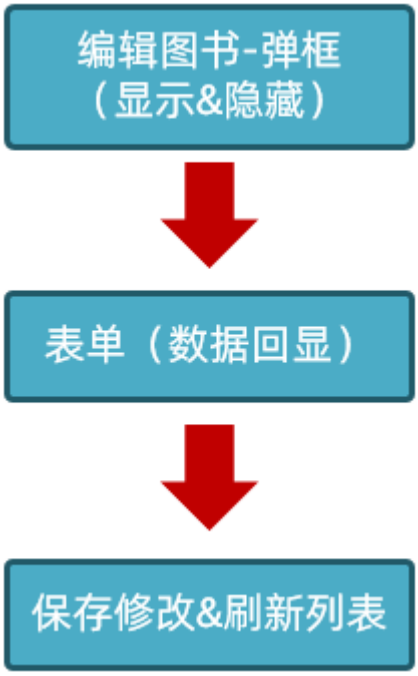
完成图书管理案例-编辑图书需求

讲解

1. 因为编辑图书要做回显等，比较复杂，所以分了3个视频来讲解
2. 需求：完成编辑图书回显当前图书数据到编辑表单，在用户点击修改按钮，收集数据提交到服务器保存，并刷新列表

3. 编辑数据的核心思路：
   1. 给编辑元素，绑定点击事件（事件委托方式并判断点击的是编辑元素才走编辑逻辑代码），并获取到要编辑的数据id
   2. 基于 axios 和接口文档，调用查询图书详情接口，获取正在编辑的图书数据，并回显到表单中（页面上的数据是在用户的浏览器中不够准备，所以只要是查看数据都要从服务器获取）

   3. 收集并提交保存修改数据，并重新从服务器获取列表刷新页面



4. 核心代码如下：

```
/**
 * 目标4：编辑图书
 *  4.1 编辑弹框->显示和隐藏
 *  4.2 获取当前编辑图书数据->回显到编辑表单中
 *  4.3 提交保存修改，并刷新列表
 */
// 4.1 编辑弹框->显示和隐藏
const editDom = document.querySelector('.edit-modal')
const editModal = new bootstrap.Modal(editDom)
// 编辑元素->点击->弹框显示
document.querySelector('.list').addEventListener('click', e => {
  // 判断点击的是否为编辑元素
  if (e.target.classList.contains('edit')) {
    // 4.2 获取当前编辑图书数据->回显到编辑表单中
    const theId = e.target.parentNode.dataset.id
    axios({
      url: `http://hmajax.itheima.net/api/books/${theId}`
    }).then(result => {
      const bookObj = result.data.data
      // document.querySelector('.edit-form .bookname').value = bookObj.bookname
      // document.querySelector('.edit-form .author').value = bookObj.author
      // 数据对象“属性”和标签“类名”一致
      // 遍历数据对象，使用属性去获取对应的标签，快速赋值
      const keys = Object.keys(bookObj) // ['id', 'bookname', 'author', 'publisher']
      keys.forEach(key => {
        document.querySelector(`.edit-form .${key}`).value = bookObj[key]
      })
    })
    editModal.show()
  }
```

```
})
// 修改按钮->点击->隐藏弹框
document.querySelector('.edit-btn').addEventListener('click', () => {
  // 4.3 提交保存修改，并刷新列表
  const editForm = document.querySelector('.edit-form')
  const { id, bookname, author, publisher } = serialize(editForm, {
hash: true, empty: true})
  // 保存正在编辑的图书id，隐藏起来：无需让用户修改
  // <input type="hidden" class="id" name="id" value="84783">
  axios({
    url: `http://hmajax.itheima.net/api/books/${id}`,
    method: 'PUT',
    data: {
      bookname,
      author,
      publisher,
      creator
    }
  }).then(() => {
    // 修改成功以后，重新获取并刷新列表
    getBooksList()

    // 隐藏弹框
    editModal.hide()
  })
})
```

小结

1. 编辑数据的步骤是什么?

   ▶ 答案

10.案例_图书管理_总结

目标

总结下增删改查的核心思路

讲解

1. 因为增删改查的业务在前端实际开发中非常常见，思路是可以通用的，所以总结下思路

> 1.渲染列表（查）
>
> 2.新增图书（增）
>
> 3.删除图书（删）
>
> 4.编辑图书（改）

2. 渲染数据（查）

> 核心思路：获取数据 -> 渲染数据

```
// 1.1 获取数据
axios({...}).then(result => {
  const bookList = result.data.data
  // 1.2 渲染数据
  const htmlStr = bookList.map((item, index) => {
    return `<tr>
    <td>${index + 1}</td>
    <td>${item.bookname}</td>
    <td>${item.author}</td>
    <td>${item.publisher}</td>
    <td data-id=${item.id}>
      <span class="del">删除</span>
      <span class="edit">编辑</span>
    </td>
  </tr>`
  }).join('')
  document.querySelector('.list').innerHTML = htmlStr
})
```

3. 新增数据（增）

> 核心思路：准备页面标签 -> 收集数据提交（必须） -> 刷新页面列表（可选）

```
// 2.1 创建弹框对象
const addModalDom = document.querySelector('.add-modal')
const addModal = new bootstrap.Modal(addModalDom)
document.querySelector('.add-btn').addEventListener('click', () => {
  // 2.2 收集表单数据，并提交到服务器保存
  const addForm = document.querySelector('.add-form')
  const bookObj = serialize(addForm, { hash: true, empty: true })
  axios({...}).then(result => {
    // 2.3 添加成功后，重新请求并渲染图书列表
    getBooksList()
    addForm.reset()
    addModal.hide()
  })
```

```
14 })
```

4. 删除图书（删）

> 核心思路：绑定点击事件（获取要删除的图书唯一标识） -> 调用删除接口（让服务器删除此数据） -> 成功后重新获取并刷新列表

```
// 3.1 删除元素->点击（事件委托）
document.querySelector('.list').addEventListener('click', e => {
  if (e.target.classList.contains('del')) {
    // 获取图书id（自定义属性id）
    const theId = e.target.parentNode.dataset.id
    // 3.2 调用删除接口
    axios({...}).then(() => {
      // 3.3 刷新图书列表
      getBooksList()
    })
  }
})
```

5. 编辑图书（改）

> 核心思路：准备编辑图书表单 -> 表单回显正在编辑的数据 -> 点击修改收集数据 -> 提交到服务器保存 -> 重新获取并刷新列表

```
// 4.1 编辑弹框->显示和隐藏
const editDom = document.querySelector('.edit-modal')
const editModal = new bootstrap.Modal(editDom)
document.querySelector('.list').addEventListener('click', e => {
  if (e.target.classList.contains('edit')) {
    // 4.2 获取当前编辑图书数据->回显到编辑表单中
    const theId = e.target.parentNode.dataset.id
    axios({...}).then(result => {
      const bookObj = result.data.data
      // 遍历数据对象，使用属性去获取对应的标签，快速赋值
      const keys = Object.keys(bookObj)
      keys.forEach(key => {
        document.querySelector(`.edit-form .${key}`).value = bookObj[key]
      })
    })
    editModal.show()
  }
})

document.querySelector('.edit-btn').addEventListener('click', () => {
  // 4.3 提交保存修改，并刷新列表
  const editForm = document.querySelector('.edit-form')
  const { id, bookname, author, publisher } = serialize(editForm, {hash: true, empty: true})
  // 保存正在编辑的图书id，隐藏起来：无需让用户修改
  // <input type="hidden" class="id" name="id" value="84783">
  axios({...}).then(() => {
    getBooksList()
    editModal.hide()
  })
})
```

小结

1. 学完图书管理案例，我们收货了什么?

   ▶ 答案

11.图片上传

目标

把本地图片上传到网页上显示

讲解

1. 什么是图片上传?
   - 就是把本地的图片上传到网页上显示
2. 图片上传怎么做?
   - 先依靠文件选择元素获取用户选择的本地文件，接着提交到服务器保存，服务器会返回图片的 url 网址，然后把网址加载到 img 标签的 src 属性中即可显示
3. 为什么不直接显示到浏览器上，要放到服务器上呢?
   - 因为浏览器保存是临时的，如果你想随时随地访问图片，需要上传到服务器上
4. 图片上传怎么做呢?

1. 先获取图片文件对象
2. 使用 FormData 表单数据对象装入（因为图片是文件而不是以前的数字和字符串了所以传递文件一般需要放入 FormData 以键值对-文件流的数据传递（可以查看请求体-确认请求体结构）

```
const fd = new FormData()
fd.append(参数名, 值)
```

3. 提交表单数据对象，使用服务器返回图片 url 网址

5. 核心代码如下：

```
<!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>图片上传</title>
</head>

<body>
  <!-- 文件选择元素 -->
  <input type="file" class="upload">
  <img src="" alt="" class="my-img">

  <script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js">
</script>
  <script>
    /**
     * 目标：图片上传，显示到网页上
     *  1. 获取图片文件
     *  2. 使用 FormData 携带图片文件
     *  3. 提交到服务器，获取图片url网址使用
    */
    // 文件选择元素->change改变事件
    document.querySelector('.upload').addEventListener('change', e => {
      // 1. 获取图片文件
      console.log(e.target.files[0])
      // 2. 使用 FormData 携带图片文件
      const fd = new FormData()
      fd.append('img', e.target.files[0])
      // 3. 提交到服务器，获取图片url网址使用
      axios({
        url: 'http://hmajax.itheima.net/api/uploadimg',
        method: 'POST',
        data: fd
      }).then(result => {
        console.log(result)
        // 取出图片url网址，用img标签加载显示
        const imgUrl = result.data.data.url
        document.querySelector('.my-img').src = imgUrl
      })
    })
  </script>
```

```
</body>

</html>
```

小结

1. 图片上传的思路是什么?

   ▶ 答案

12.案例_网站-更换背景图

目标

实现更换网站背景图的效果

讲解

1. 需求：先运行备课代码，查看要完成的效果，点击右上角选择本机中提供的素材图片，更换网站背景图

2. 网站更换背景图如何实现呢，并且保证刷新后背景图还在？具体步骤：
   1. 先获取到用户选择的背景图片，上传并把服务器返回的图片 url 网址设置给 body 背景
   2. 上传成功时，保存图片 url 网址到 localStorage 中
   3. 网页运行后，获取 localStorage 中的图片的 url 网址使用（并判断本地有图片 url 网址字符串才设置）
3. 核心代码如下：

```
/**
 * 目标：网站-更换背景
 *  1. 选择图片上传，设置body背景
 *  2. 上传成功时，"保存"图片url网址
```

```
 *  3. 网页运行后，"获取"url网址使用
 * */
document.querySelector('.bg-ipt').addEventListener('change', e => {
  // 1. 选择图片上传，设置body背景
  console.log(e.target.files[0])
  const fd = new FormData()
  fd.append('img', e.target.files[0])
  axios({
    url: 'http://hmajax.itheima.net/api/uploadimg',
    method: 'POST',
    data: fd
  }).then(result => {
    const imgUrl = result.data.data.url
    document.body.style.backgroundImage = `url(${imgUrl})`

    // 2. 上传成功时，"保存"图片url网址
    localStorage.setItem('bgImg', imgUrl)
  })
})

// 3. 网页运行后，"获取"url网址使用
const bgUrl = localStorage.getItem('bgImg')
console.log(bgUrl)
bgUrl && (document.body.style.backgroundImage = `url(${bgUrl})`)
```

小结

1. localStorage 取值和赋值的语法分别是什么?

▶ 答案

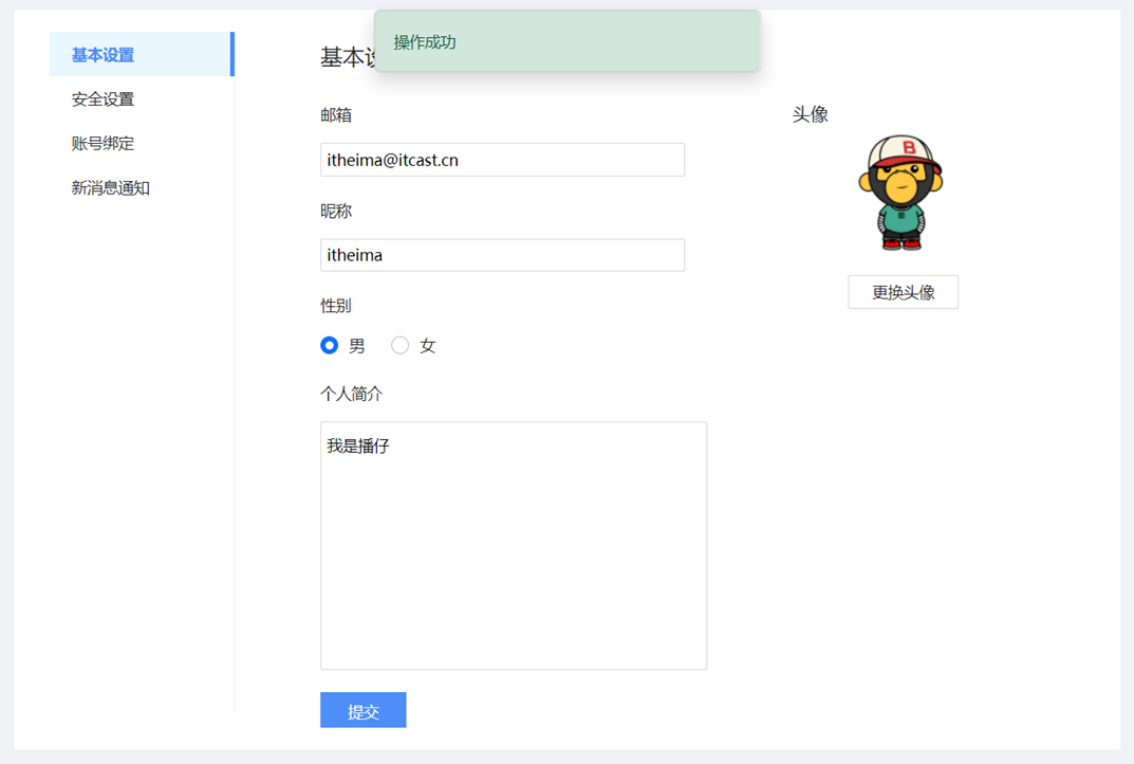
13.案例_个人信息设置-介绍

目标

介绍个人信息设置案例-需要完成哪些效果，分几个视频讲解

讲解

1. 需求：先运行备课代码，查看要完成的效果



2. 本视频分为，信息回显 + 头像修改 + 信息修改+ 提示框反馈 4 部分
    1. 先完成信息回显
    2. 再做头像修改-立刻就更新给此用户
    3. 收集个人信息表单-提交保存
    4. 提交后反馈结果给用户（提示框）

小结

暂无

14.案例_个人信息设置-信息渲染

目标

把外号对应的用户信息渲染到页面上

讲解

1. 需求：把外号对应的个人信息和头像，渲染到页面表单和头像标签上。

2. 注意：还是需要准备一个外号，因为想要查看自己对应的用户信息，不想被别人影响

3. 步骤：

   - 获取数据
   - 渲染数据到页面

4. 代码如下：

```
/**
 * 目标1：信息渲染
 *  1.1 获取用户的数据
 *  1.2 回显数据到标签上
 * */
const creator = '播仔'
// 1.1 获取用户的数据
axios({
  url: 'http://hmajax.itheima.net/api/settings',
  params: {
    creator
  }
}).then(result => {
  const userObj = result.data.data
  // 1.2 回显数据到标签上
  Object.keys(userObj).forEach(key => {
    if (key === 'avatar') {
      // 赋予默认头像
      document.querySelector('.prew').src = userObj[key]
    } else if (key === 'gender') {
      // 赋予默认性别
      // 获取性别单选框：[男radio元素，女radio元素]
      const gRadioList = document.querySelectorAll('.gender')
      // 获取性别数字：0男，1女
      const gNum = userObj[key]
      // 通过性别数字，作为下标，找到对应性别单选框，设置选中状态
      gRadioList[gNum].checked = true
```

```
    } else {
      // 赋予默认内容
      document.querySelector(`.${key}`).value = userObj[key]
    }
  })
})
```

小结

1. 渲染数据和图书列表的渲染思路是否一样呢，是什么？

   ▶ 答案

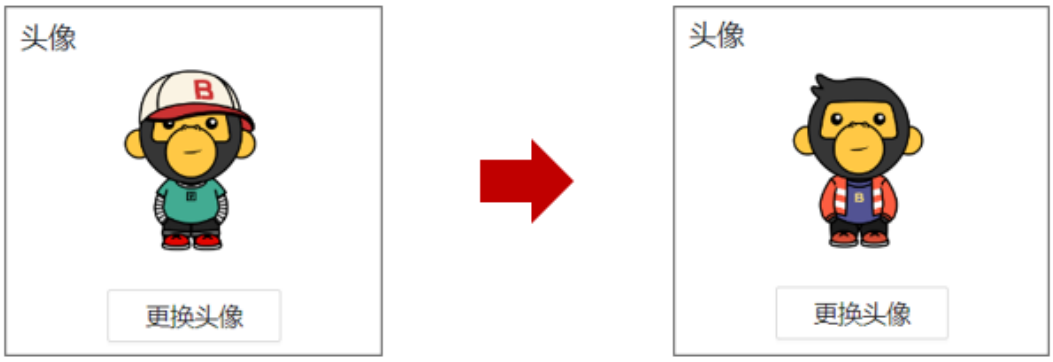
15.案例_个人信息设置-头像修改

目标

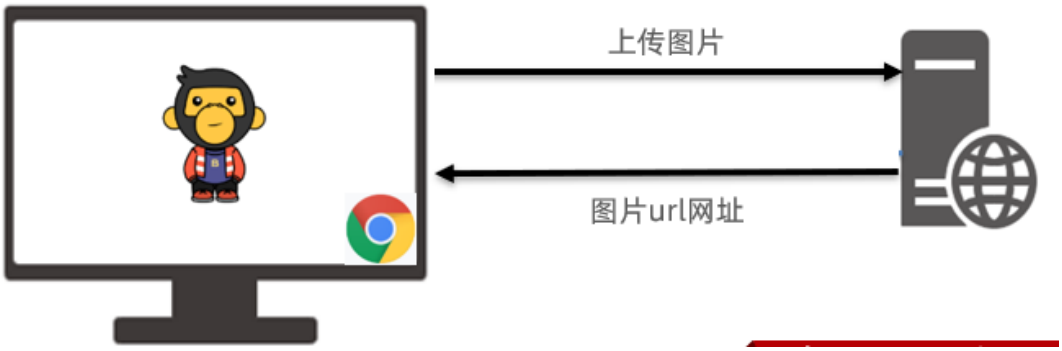
修改用户的头像并立刻生效

讲解

1. 需求：点击修改用户头像



2. 实现步骤如下：
   1. 获取到用户选择的头像文件
   2. 调用头像修改接口，并除了头像文件外，还要在 FormData 表单数据对象中携带外号
   3. 提交到服务器保存此用户对应头像文件，并把返回的头像图片 url 网址设置在页面上



3. 注意：重新刷新重新获取，已经是修改后的头像了（证明服务器那边确实保存成功）

4. 核心代码：

```
/**
 * 目标2：修改头像
 *  2.1 获取头像文件
 *  2.2 提交服务器并更新头像
 * */
// 文件选择元素->change事件
document.querySelector('.upload').addEventListener('change', e => {
  // 2.1 获取头像文件
  console.log(e.target.files[0])
  const fd = new FormData()
  fd.append('avatar', e.target.files[0])
  fd.append('creator', creator)
  // 2.2 提交服务器并更新头像
  axios({
    url: 'http://hmajax.itheima.net/api/avatar',
    method: 'PUT',
    data: fd
  }).then(result => {
    const imgUrl = result.data.data.avatar
    // 把新的头像回显到页面上
    document.querySelector('.prew').src = imgUrl
  })
})
```

小结

1. 为什么这次上传头像，需要携带外号呢?

▶ 答案

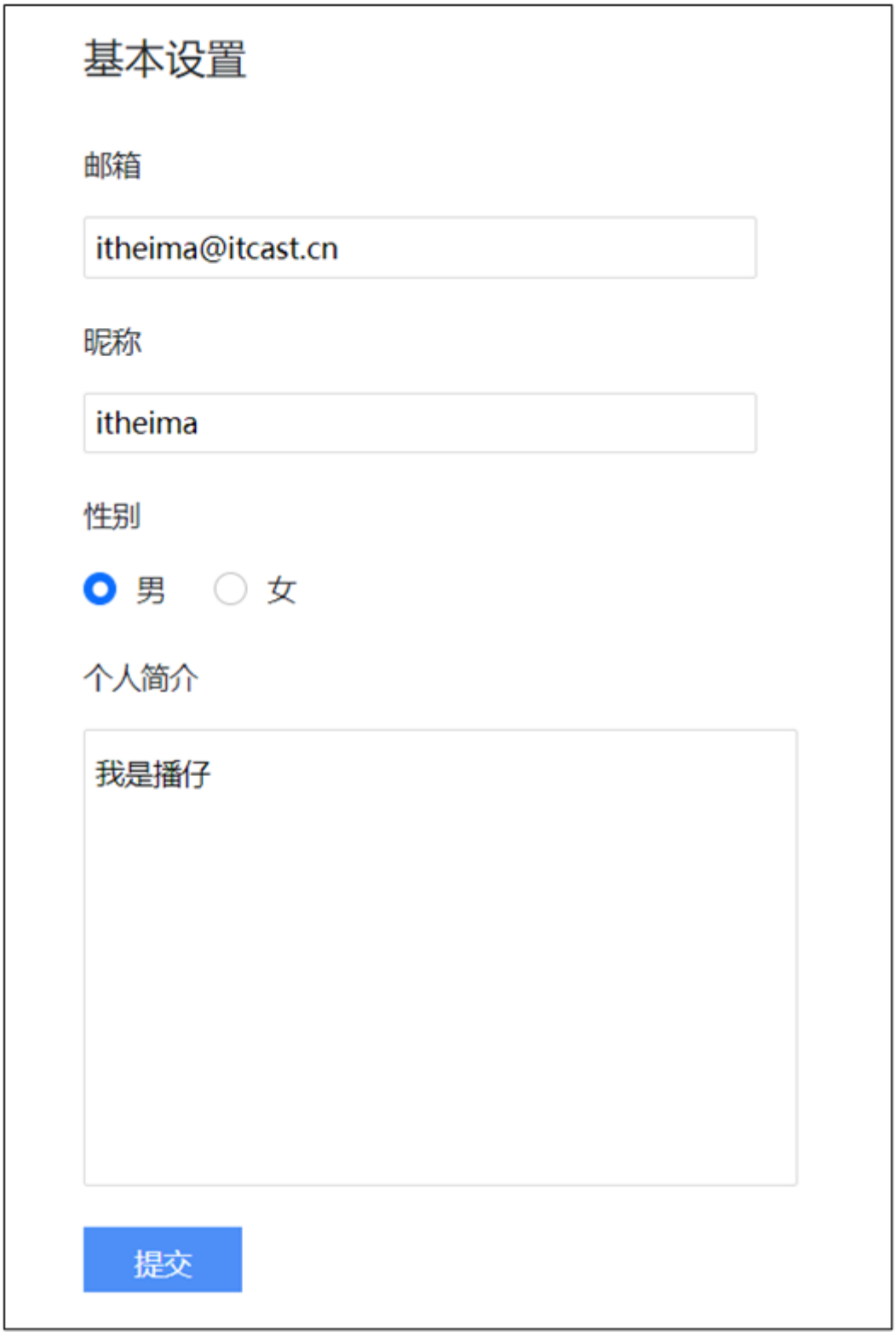
16.案例_个人信息设置-信息修改

目标

把用户修改的信息提交到服务器保存

讲解

1. 需求：点击提交按钮，收集个人信息，提交到服务器保存（无需重新获取刷新，因为页面已经是最新的数据了）
    1. 收集表单数据
    2. 提交到服务器保存-调用用户信息更新接口（注意请求方法是 PUT）代表数据更新的意思



2. 核心代码如下：

```
/**
 * 目标3：提交表单
 *  3.1 收集表单信息
 *  3.2 提交到服务器保存
 */
// 保存修改->点击
document.querySelector('.submit').addEventListener('click', () => {
  // 3.1 收集表单信息
  const userForm = document.querySelector('.user-form')
  const userObj = serialize(userForm, { hash: true, empty: true })
  userObj.creator = creator
  // 性别数字字符串，转成数字类型
```

```
  userObj.gender = +userObj.gender
  console.log(userObj)
  // 3.2 提交到服务器保存
  axios({
    url: 'http://hmajax.itheima.net/api/settings',
    method: 'PUT',
    data: userObj
  }).then(result => {
  })
})
```

小结

1. 信息修改数据和以前增删改查哪个实现的思路比较接近呢?

   ▶ 答案

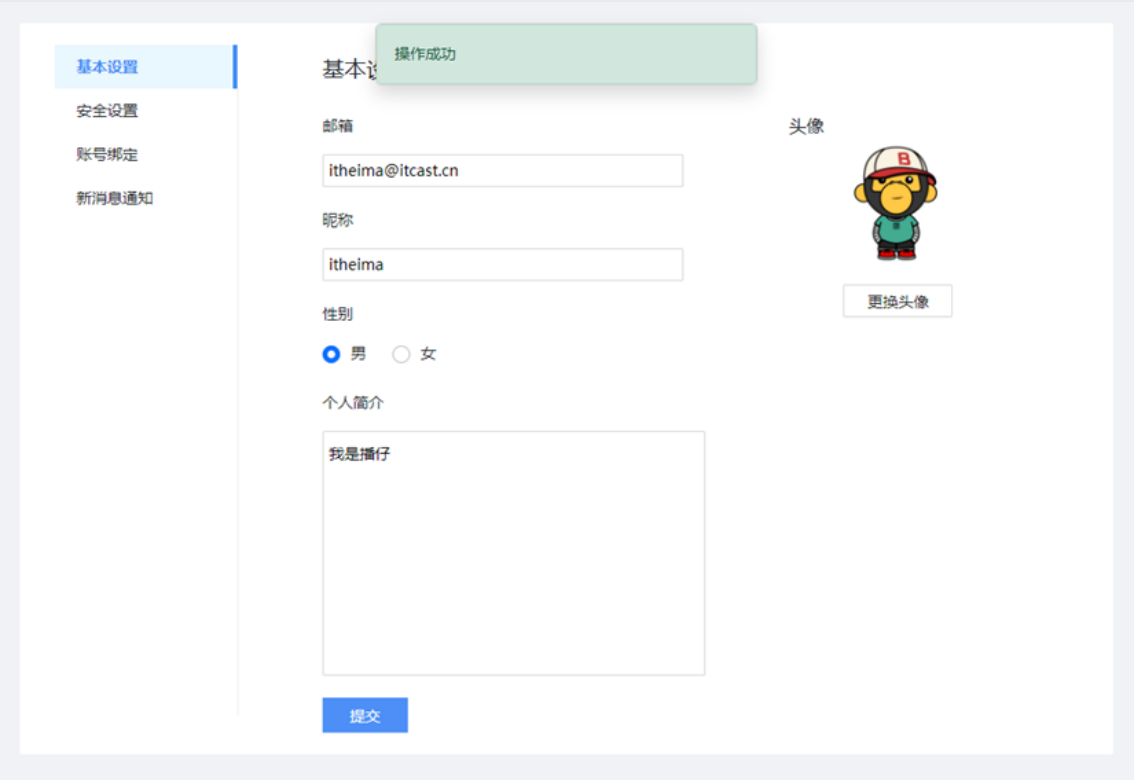
17.案例_个人信息设置-提示框

目标

把用户更新个人信息结果，用提示框反馈给用户

讲解

1. 需求：使用 bootstrap 提示框，提示个人信息设置后的结果



2. bootstrap 的 toast 提示框和 modal 弹框使用很像，语法如下：

1. 先准备对应的标签结构（模板里已有）
2. 设置延迟自动消失的时间

```
<div class="toast" data-bs-delay="1500">
  提示框内容
</div>
```

3. 使用 JS 的方式，在 axios 请求响应成功时，展示结果

```
// 创建提示框对象
const toastDom = document.querySelector('css选择器')
const toast = new bootstrap.Toast(toastDom)

// 显示提示框
toast.show()
```

3. 核心代码：

```
/**
 * 目标3：提交表单
 *  3.1 收集表单信息
 *  3.2 提交到服务器保存
 */
/**
 * 目标4：结果提示
 *  4.1 创建toast对象
 *  4.2 调用show方法->显示提示框
 */
// 保存修改->点击
document.querySelector('.submit').addEventListener('click', () => {
  // 3.1 收集表单信息
  const userForm = document.querySelector('.user-form')
  const userObj = serialize(userForm, { hash: true, empty: true })
  userObj.creator = creator
  // 性别数字字符串，转成数字类型
  userObj.gender = +userObj.gender
  console.log(userObj)
  // 3.2 提交到服务器保存
  axios({
    url: 'http://hmajax.itheima.net/api/settings',
    method: 'PUT',
    data: userObj
  }).then(result => {
    // 4.1 创建toast对象
    const toastDom = document.querySelector('.my-toast')
    const toast = new bootstrap.Toast(toastDom)

    // 4.2 调用show方法->显示提示框
    toast.show()
  })
})
```

小结

1. bootstrap 弹框什么时候用 JS 方式控制显示呢?

   ▶ 答案

今日重点(必须会)

1. 掌握增删改查数据的思路
2. 掌握图片上传的思路和流程
3. 理解调用接口时，携带外号的作用
4. 了解 bootstrap 弹框的使用

今日作业(必完成)

在配套作业文件夹的md内

参考文献

1. 表单概念->百度百科
2. accept属性->mdn
3. accept属性->菜鸟教程
4. FormData->mdn
5. BS的Model文档
6. axios请求方式别名

Day03_AJAX原理

知识点自测

1. 以下哪个方法可以把 JS 数据类型转成 JSON 字符串类型?

   A. JSON.stringify()

   B. JSON.parse()

▶ 答案

2. 以下哪个方法，会延迟一段时间后，再执行函数体，并执行一次就停止?

A. setTimeout(函数体, 毫秒值)

B. setInterval(函数体, 毫秒值)

▶ 答案

3. 下面代码 result 结果是多少?

```
let obj = {
  status: 240
}
const result = obj.status >= 200 && obj.status < 300
```

A. true

B. 大于

C. 240

D. false

▶ 答案

4. 下面代码运行结果是多少?

```
let result = 'http://www.baidu.com'
result += '?a=10'
result += '&b=20'
```

A：'http://www.baidu.com'

B：'?a=10'

C：'&b=20'

D：'http://www.baidu.com?a=10&b=20'

▶ 答案

5. 哪个事件能实时检测到输入框值的变化?

A：input 事件

B：change 事件

▶ 答案

目录

- 封装简易版 axios
- 案例 - 天气预报

学习目标

1. 了解原生 AJAX 语法 - XMLHttpRequest（XHR）
2. 了解 Promise 的概念和使用
3. 了解 axios 内部工作的大概过程（XHR + Promise）
4. 案例 - 天气预报

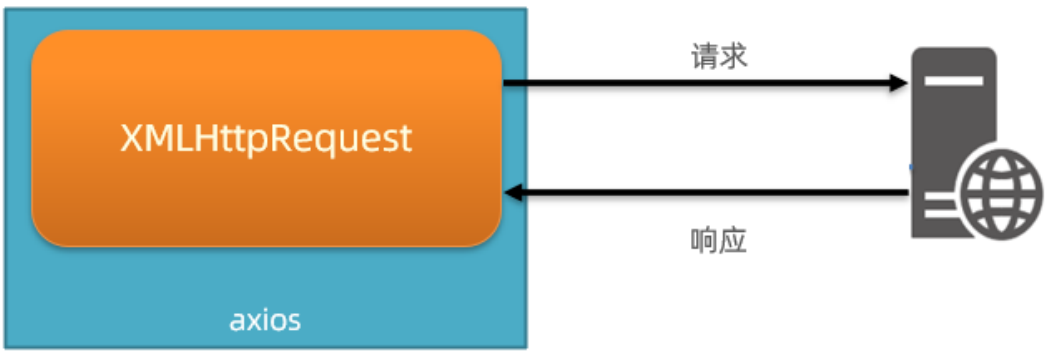
01.XMLHttpRequest - 基础使用

目标

了解 AJAX 原理 XHR 的基础使用

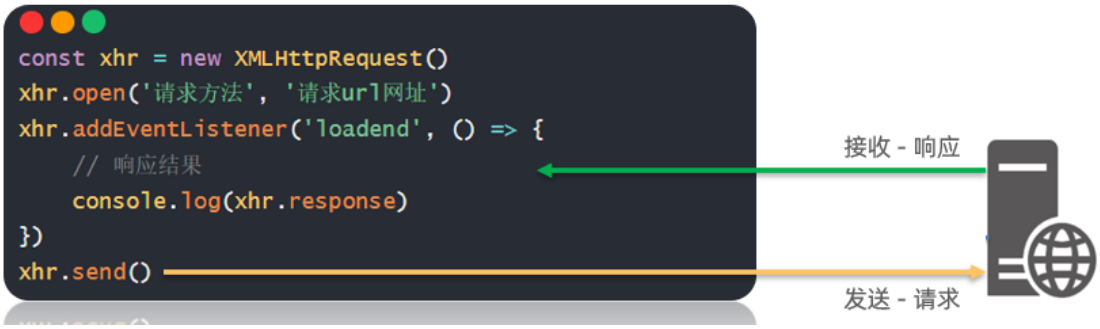
讲解

1. AJAX 是浏览器与服务器通信的技术，采用 XMLHttpRequest 对象相关代码
2. axios 是对 XHR 相关代码进行了封装，让我们只关心传递的接口参数
3. 学习 XHR 也是了解 axios 内部与服务器交互过程的真正原理



4. 语法如下：

```
const xhr = new XMLHttpRequest()
xhr.open('请求方法', '请求url网址')
xhr.addEventListener('loadend', () => {
  // 响应结果
  console.log(xhr.response)
})
xhr.send()
```



5. 需求：以一个需求来体验下原生 XHR 语法，获取所有省份列表并展示到页面上

6. 代码如下：

```
<!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>XMLHttpRequest_基础使用</title>
</head>

<body>
  <p class="my-p"></p>
  <script>
    /**
     * 目标：使用XMLHttpRequest对象与服务器通信
     *  1. 创建 XMLHttpRequest 对象
     *  2. 配置请求方法和请求 url 地址
     *  3. 监听 loadend 事件，接收响应结果
     *  4. 发起请求
     */
    // 1. 创建 XMLHttpRequest 对象
    const xhr = new XMLHttpRequest()

    // 2. 配置请求方法和请求 url 地址
    xhr.open('GET', 'http://hmajax.itheima.net/api/province')

    // 3. 监听 loadend 事件，接收响应结果
    xhr.addEventListener('loadend', () => {
      console.log(xhr.response)
      const data = JSON.parse(xhr.response)
      console.log(data.list.join('<br>'))
      document.querySelector('.my-p').innerHTML = data.list.join('<br>')
    })

    // 4. 发起请求
    xhr.send()
  </script>
</body>

</html>
```

小结

1. AJAX 原理是什么?

   ▶ 答案

2. 为什么学习 XHR ?

   ▶ 答案

3. XHR 使用步骤?

   ▶ 答案

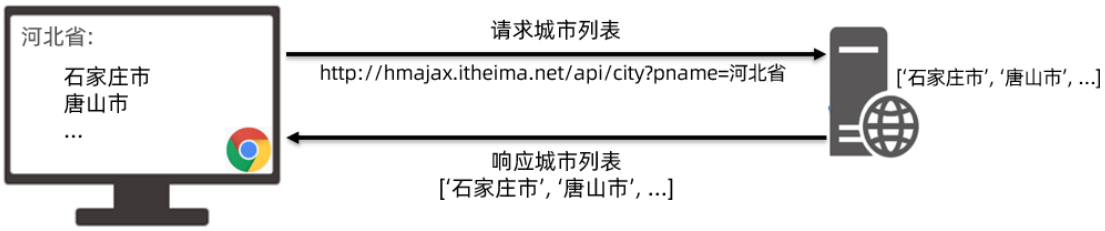
02.XMLHttpRequest - 查询参数

目标

使用 XHR 传递查询参数给服务器，获取匹配数据

讲解

1. 什么是查询参数：携带额外信息给服务器，返回匹配想要的数据
2. 查询参数原理要携带的位置和语法：http://xxxx.com/xxx/xxx?参数名1=值1&参数名2=值2
3. 所以，原生 XHR 需要自己在 url 后面携带查询参数字符串，没有 axios 帮助我们把 params 参数拼接到 url 字符串后面了
4. 需求：查询河北省下属的城市列表



5. 核心代码如下：

```
/**
 * 目标：使用XHR携带查询参数，展示某个省下属的城市列表
*/
const xhr = new XMLHttpRequest()
xhr.open('GET', 'http://hmajax.itheima.net/api/city?pname=辽宁省')
xhr.addEventListener('loadend', () => {
  console.log(xhr.response)
  const data = JSON.parse(xhr.response)
  console.log(data)
  document.querySelector('.city-p').innerHTML = data.list.join('<br>')
})
xhr.send()
```

小结

1. XHR 如何携带查询参数?

   ▶ 答案

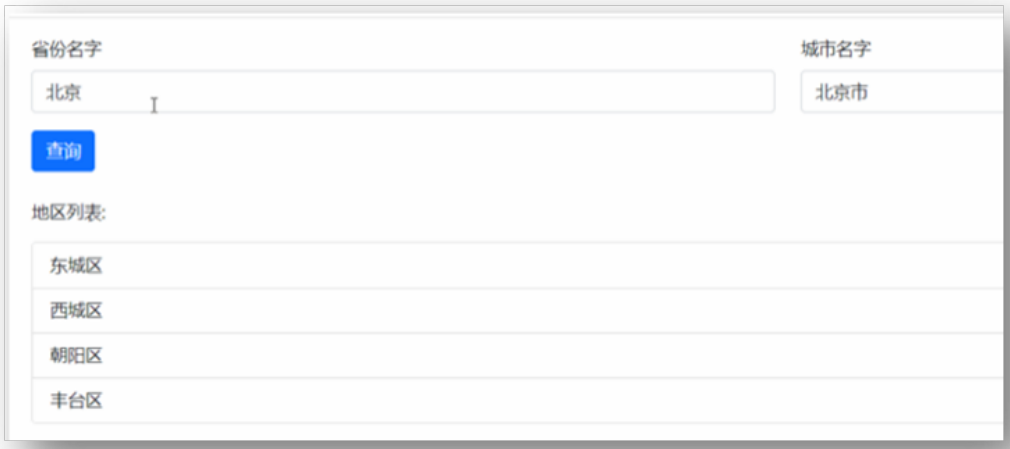
03.案例 - 地区查询

目标

使用 XHR 完成案例地区查询

讲解

1. 需求：和我们之前做的类似，就是不用 axios 而是用 XHR 实现，输入省份和城市名字后，点击查询，传递多对查询参数并获取地区列表的需求



2. 但是多个查询参数，如果自己拼接很麻烦，这里用 URLSearchParams 把参数对象转成“参数名=值&参数名=值”格式的字符串，语法如下：

```
// 1. 创建 URLSearchParams 对象
const paramsObj = new URLSearchParams({
  参数名1: 值1,
  参数名2: 值2
})

// 2. 生成指定格式查询参数字符串
const queryString = paramsObj.toString()
// 结果：参数名1=值1&参数名2=值2
```

小结

1. JS 对象如何转成查询参数格式字符串?

   ▶ 答案

04.XMLHttpRequest - 数据提交

目标

通过 XHR 提交用户名和密码，完成注册功能

讲解

1. 了解原生 XHR 进行数据提交的方式
2. 需求：通过 XHR 完成注册用户功能

3. 步骤和语法：
   1. 注意1：但是这次没有 axios 帮我们了，我们需要自己设置请求头 Content-Type：application/json，来告诉服务器端，我们发过去的内容类型是 JSON 字符串，让他转成对应数据结构取值使用
   2. 注意2：没有 axios 了，我们前端要传递的请求体数据，也没人帮我把 JS 对象转成 JSON 字符串了，需要我们自己转换
   3. 注意3：原生 XHR 需要在 send 方法调用时，传入请求体携带

```
const xhr = new XMLHttpRequest()
xhr.open('请求方法', '请求url网址')
xhr.addEventListener('loadend', () => {
  console.log(xhr.response)
})

// 1. 告诉服务器，我传递的内容类型，是 JSON 字符串
xhr.setRequestHeader('Content-Type', 'application/json')
// 2. 准备数据并转成 JSON 字符串
const user = { username: 'itheima007', password: '7654321' }
const userStr = JSON.stringify(user)
// 3. 发送请求体数据
xhr.send(userStr)
```

4. 核心代码如下：

```
/**
 * 目标：使用xhr进行数据提交-完成注册功能
*/
document.querySelector('.reg-btn').addEventListener('click', () => {
  const xhr = new XMLHttpRequest()
  xhr.open('POST', 'http://hmajax.itheima.net/api/register')
  xhr.addEventListener('loadend', () => {
    console.log(xhr.response)
  })

  // 设置请求头-告诉服务器内容类型（JSON字符串）
  xhr.setRequestHeader('Content-Type', 'application/json')
  // 准备提交的数据
  const userObj = {
    username: 'itheima007',
    password: '7654321'
  }
  const userStr = JSON.stringify(userObj)
  // 设置请求体，发起请求
  xhr.send(userStr)
})
```

小结

1. XHR 如何提交请求体数据?

▶ 答案

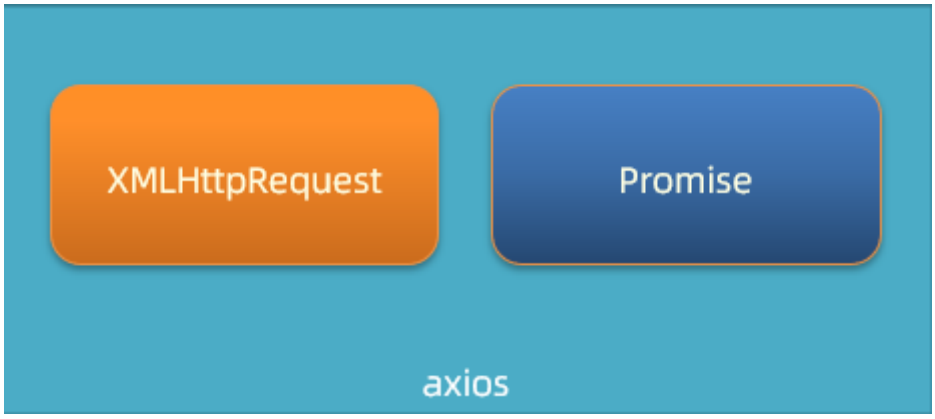
05.认识_Promise

目标

认识 Promise 的作用和好处以及使用步骤

讲解

1. 什么是 Promise ?
   - Promise 对象用于表示一个异步操作的最终完成（或失败）及其结构值
2. Promise 的好处是什么?
   - 逻辑更清晰（成功或失败会关联后续的处理函数）
   - 了解 axios 函数内部运作的机制



   - 能解决回调函数地狱问题（后面会讲到），今天先来看下它的基础使用
3. Promise 管理异步任务，语法怎么用?

```
// 1. 创建 Promise 对象
const p = new Promise((resolve, reject) => {
 // 2. 执行异步任务-并传递结果
 // 成功调用: resolve(值) 触发 then() 执行
 // 失败调用: reject(值) 触发 catch() 执行
})
// 3. 接收结果
p.then(result => {
 // 成功
}).catch(error => {
 // 失败
})
```

4. 示例代码:

```
/**
 * 目标：使用Promise管理异步任务
*/
// 1. 创建Promise对象
const p = new Promise((resolve, reject) => {
  // 2. 执行异步代码
  setTimeout(() => {
    // resolve('模拟AJAX请求-成功结果')
```

```
    reject(new Error('模拟AJAX请求-失败结果'))
  }, 2000)
})

// 3. 获取结果
p.then(result => {
  console.log(result)
}).catch(error => {
  console.log(error)
})
```

小结

1. 什么是 Promise ?

   ▶ 答案
2. 为什么学习 Promise ?

   ▶ 答案
3. Promise 使用步骤?

   ▶ 答案

06.认识_Promise 的状态

目标

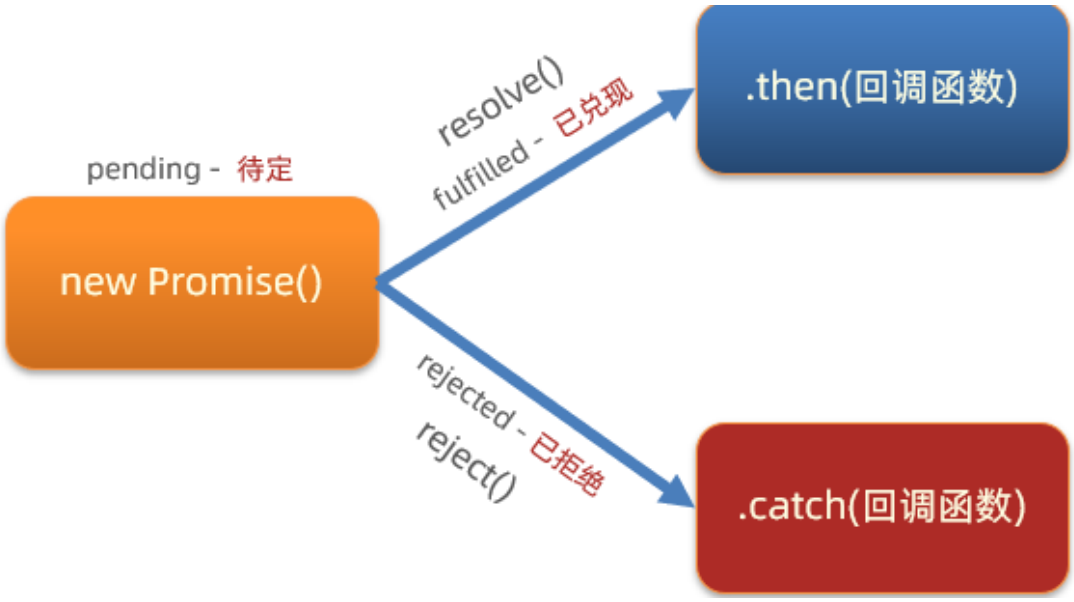
认识 Promise 的三种状态，知道如何关联成功/失败的处理函数

讲解

1. 为什么要了解 Promise 的三种状态 ?
   - 知道 Promise 对象如何关联的处理函数，以及代码的执行顺序
2. Promise 有哪三种状态?

   > 每个 Promise 对象必定处于以下三种状态之一
   >
   > 1. 待定（pending）：初始状态，既没有被兑现，也没有被拒绝
   > 2. 已兑现（fulfilled）：操作成功完成
   > 3. 已拒绝（rejected）：操作失败

   > 状态的英文字符串，可以理解为 Promise 对象内的字符串标识符，用于判断什么时候调用哪一个处理函数
3. Promise 的状态改变有什么用：调用对应函数，改变 Promise 对象状态后，内部触发对应回调函数传参并执行



4. 注意：每个 Promise 对象一旦被兑现/拒绝，那就是已敲定了，状态无法再被改变

小结

1. Promise 对象有哪 3 种状态?
   - ▶ 答案
2. Promise 状态有什么用?
   - ▶ 答案

07.使用 Promise 和 XHR_获取省份列表

目标

尝试用 Promise 管理 XHR 异步任务

讲解

1. Promise 和 XHR 都已经学过基础语法了，我们可以来结合使用一下了
2. 需求：使用 Promise 和 XHR 请求省份列表数据并展示到页面上

3. 步骤：

    1. 创建 Promise 对象
    2. 执行 XHR 异步代码，获取省份列表数据
    3. 关联成功或失败回调函数，做后续的处理

    > 错误情况：用地址错了404演示

4. 核心代码如下：

```
/**
 * 目标：使用Promise管理XHR请求省份列表
 *  1. 创建Promise对象
 *  2. 执行XHR异步代码，获取省份列表
 *  3. 关联成功或失败函数，做后续处理
*/
// 1. 创建Promise对象
const p = new Promise((resolve, reject) => {
  // 2. 执行XHR异步代码，获取省份列表
  const xhr = new XMLHttpRequest()
  xhr.open('GET', 'http://hmajax.itheima.net/api/province')
  xhr.addEventListener('loadend', () => {
    // xhr如何判断响应成功还是失败的？
    // 2xx开头的都是成功响应状态码
    if (xhr.status >= 200 && xhr.status < 300) {
      resolve(JSON.parse(xhr.response))
    } else {
      reject(new Error(xhr.response))
    }
  })
  xhr.send()
})

// 3. 关联成功或失败函数，做后续处理
p.then(result => {
  console.log(result)
```

```
  document.querySelector('.my-p').innerHTML = result.list.join('<br>')
}).catch(error => {
  // 错误对象要用console.dir详细打印
  console.dir(error)
  // 服务器返回错误提示消息，插入到p标签显示
  document.querySelector('.my-p').innerHTML = error.message
})
```

小结

1. AJAX 如何判断是否请求响应成功了?

   ▶ 答案

08.封装_简易axios-获取省份列表

目标

模拟 axios 函数封装，更深入了解 axios 内部运作原理

讲解

1. 需求：基于 Promise 和 XHR 封装 myAxios 函数，获取省份列表展示到页面

2. 核心语法：

```
function myAxios(config) {
  return new Promise((resolve, reject) => {
    // XHR 请求
    // 调用成功/失败的处理程序
  })
}

myAxios({
  url: '目标资源地址'
}).then(result => {

}).catch(error => {
```

```
})
```

3. 步骤：

    1. 定义 myAxios 函数，接收配置对象，返回 Promise 对象
    2. 发起 XHR 请求，默认请求方法为 GET
    3. 调用成功/失败的处理程序
    4. 使用 myAxios 函数，获取省份列表展示

4. 核心代码如下：

```
/**
 * 目标：封装_简易axios函数_获取省份列表
 *  1. 定义myAxios函数，接收配置对象，返回Promise对象
 *  2. 发起XHR请求，默认请求方法为GET
 *  3. 调用成功/失败的处理程序
 *  4. 使用myAxios函数，获取省份列表展示
*/
// 1. 定义myAxios函数，接收配置对象，返回Promise对象
function myAxios(config) {
  return new Promise((resolve, reject) => {
    // 2. 发起XHR请求，默认请求方法为GET
    const xhr = new XMLHttpRequest()
    xhr.open(config.method || 'GET', config.url)
    xhr.addEventListener('loadend', () => {
      // 3. 调用成功/失败的处理程序
      if (xhr.status >= 200 && xhr.status < 300) {
        resolve(JSON.parse(xhr.response))
      } else {
        reject(new Error(xhr.response))
      }
    })
    xhr.send()
  })
}

// 4. 使用myAxios函数，获取省份列表展示
myAxios({
  url: 'http://hmajax.itheima.net/api/province'
}).then(result => {
  console.log(result)
  document.querySelector('.my-p').innerHTML = result.list.join('<br>')
}).catch(error => {
  console.log(error)
  document.querySelector('.my-p').innerHTML = error.message
})
```

小结

1. 自己封装的 myAxios 如何设置默认请求方法 GET?

   ▶ 答案

09.封装_简易axios-获取地区列表

目标

修改 myAxios 函数支持传递查询参数，获取辽宁省，大连市的地区列表

讲解

1. 需求：在上个封装的建议 axios 函数基础上，修改代码支持传递查询参数功能
2. 修改步骤：
   1. myAxios 函数调用后，判断 params 选项
   2. 基于 URLSearchParams 转换查询参数字符串
   3. 使用自己封装的 myAxios 函数显示地区列表
3. 核心代码：

```
function myAxios(config) {
  return new Promise((resolve, reject) => {
    const xhr = new XMLHttpRequest()
    // 1. 判断有params选项，携带查询参数
    if (config.params) {
      // 2. 使用URLSearchParams转换，并携带到url上
      const paramsObj = new URLSearchParams(config.params)
      const queryString = paramsObj.toString()
      // 把查询参数字符串，拼接在url？后面
      config.url += `?${queryString}`
    }

    xhr.open(config.method || 'GET', config.url)
    xhr.addEventListener('loadend', () => {
      if (xhr.status >= 200 && xhr.status < 300) {
        resolve(JSON.parse(xhr.response))
      } else {
        reject(new Error(xhr.response))
      }
    })
    xhr.send()
  })
}

// 3. 使用myAxios函数，获取地区列表
myAxios({
  url: 'http://hmajax.itheima.net/api/area',
  params: {
    pname: '辽宁省',
```

```
    cname: '大连市'
  }
}).then(result => {
  console.log(result)
  document.querySelector('.my-p').innerHTML = result.list.join('<br>')
})
```

小结

1. 外面传入查询参数对象，myAxios 函数内如何转查询参数字符串?

   ▶ 答案

10.封装_简易axios-注册用户

目标

修改 myAxios 函数支持传递请求体数据，完成注册用户

讲解

1. 需求：修改 myAxios 函数支持传递请求体数据，完成注册用户功能
2. 修改步骤：
   1. myAxios 函数调用后，判断 data 选项
   2. 转换数据类型，在 send 方法中发送
   3. 使用自己封装的 myAxios 函数完成注册用户功能
3. 核心代码：

```
function myAxios(config) {
  return new Promise((resolve, reject) => {
    const xhr = new XMLHttpRequest()

    if (config.params) {
      const paramsObj = new URLSearchParams(config.params)
      const queryString = paramsObj.toString()
      config.url += `?${queryString}`
    }
    xhr.open(config.method || 'GET', config.url)

    xhr.addEventListener('loadend', () => {
      if (xhr.status >= 200 && xhr.status < 300) {
        resolve(JSON.parse(xhr.response))
      } else {
        reject(new Error(xhr.response))
      }
    })
    // 1. 判断有data选项，携带请求体
    if (config.data) {
```

```
      // 2. 转换数据类型，在send中发送
      const jsonStr = JSON.stringify(config.data)
      xhr.setRequestHeader('Content-Type', 'application/json')
      xhr.send(jsonStr)
    } else {
      // 如果没有请求体数据，正常的发起请求
      xhr.send()
    }
  })
}

document.querySelector('.reg-btn').addEventListener('click', () => {
  // 3. 使用myAxios函数，完成注册用户
  myAxios({
    url: 'http://hmajax.itheima.net/api/register',
    method: 'POST',
    data: {
      username: 'itheima999',
      password: '666666'
    }
  }).then(result => {
    console.log(result)
  }).catch(error => {
    console.dir(error)
  })
})
```

小结

1. 外面传入 data 选项，myAxios 函数内如何携带请求体参数?

▶ 答案

11-12.案例_天气预报-默认数据

目标

把北京市的数据，填充到页面默认显示

讲解

1. 需求：介绍本项目要完成的效果，和要实现的步骤和分的步骤和视频

2. 步骤
    1. 先获取北京市天气预报，展示
    2. 搜索城市列表，展示
    3. 点击城市，切换显示对应天气数据
3. 本视频先封装函数，获取城市天气并设置页面内容
4. 核心代码如下：

```
/**
 * 目标1：默认显示-北京市天气
 *  1.1 获取北京市天气数据
 *  1.2 数据展示到页面
 */
// 获取并渲染城市天气函数
function getWeather(cityCode) {
  // 1.1 获取北京市天气数据
  myAxios({
    url: 'http://hmajax.itheima.net/api/weather',
    params: {
      city: cityCode
    }
  }).then(result => {
    console.log(result)
    const wObj = result.data
    // 1.2 数据展示到页面
    // 阳历和农历日期
    const dateStr = `<span class="dateShort">${wObj.date}</span>
    <span class="calendar">农历 
      <span class="dateLunar">${wObj.dateLunar}</span>
    </span>`
    document.querySelector('.title').innerHTML = dateStr
    // 城市名字
    document.querySelector('.area').innerHTML = wObj.area
    // 当天气温
    const nowWStr = `<div class="tem-box">
    <span class="temp">
      <span class="temperature">${wObj.temperature}</span>
      <span>°</span>
```

```
      </span>
    </div>
    <div class="climate-box">
      <div class="air">
        <span class="psPm25">${wObj.psPm25}</span>
        <span class="psPm25Level">${wObj.psPm25Level}</span>
      </div>
      <ul class="weather-list">
        <li>
          <img src="${wObj.weatherImg}" class="weatherImg" alt="">
          <span class="weather">${wObj.weather}</span>
        </li>
        <li class="windDirection">${wObj.windDirection}</li>
        <li class="windPower">${wObj.windPower}</li>
      </ul>
    </div>`
      document.querySelector('.weather-box').innerHTML = nowWStr
      // 当天天气
      const twObj = wObj.todayWeather
      const todayWStr = `<div class="range-box">
      <span>今天：</span>
      <span class="range">
        <span class="weather">${twObj.weather}</span>
        <span class="temNight">${twObj.temNight}</span>
        <span>-</span>
        <span class="temDay">${twObj.temDay}</span>
        <span>℃</span>
      </span>
    </div>
    <ul class="sun-list">
      <li>
        <span>紫外线</span>
        <span class="ultraviolet">${twObj.ultraviolet}</span>
      </li>
      <li>
        <span>湿度</span>
        <span class="humidity">${twObj.humidity}</span>%
      </li>
      <li>
        <span>日出</span>
        <span class="sunriseTime">${twObj.sunriseTime}</span>
      </li>
      <li>
        <span>日落</span>
        <span class="sunsetTime">${twObj.sunsetTime}</span>
      </li>
    </ul>`
      document.querySelector('.today-weather').innerHTML = todayWStr

      // 7日天气预报数据展示
      const dayForecast = wObj.dayForecast
      const dayForecastStr = dayForecast.map(item => {
        return `<li class="item">
        <div class="date-box">
          <span class="dateFormat">${item.dateFormat}</span>
          <span class="date">${item.date}</span>
```

```
      </div>
      <img src="${item.weatherImg}" alt="" class="weatherImg">
      <span class="weather">${item.weather}</span>
      <div class="temp">
        <span class="temNight">${item.temNight}</span>-
        <span class="temDay">${item.temDay}</span>
        <span>℃</span>
      </div>
      <div class="wind">
        <span class="windDirection">${item.windDirection}</span>
        <span class="windPower">${item.windPower}</span>
      </div>
    </li>`
    }).join('')
    // console.log(dayForecastStr)
    document.querySelector('.week-wrap').innerHTML = dayForecastStr
  })
}

// 默认进入网页-就要获取天气数据（北京市城市编码：'110100'）
getWeather('110100')
```

小结

1. 做完这个项目会带来什么收货?

   ▶ 答案

13.案例_天气预报-搜索城市列表

目标

根据关键字，展示匹配的城市列表

讲解

1. 介绍本视频要完成的效果：搜索匹配关键字相关城市名字，展示城市列表即可

2. 步骤

    1. 绑定 input 事件，获取关键字
    2. 获取展示城市列表数据

3. 核心代码如下：

```
/**
 * 目标2：搜索城市列表
 *  2.1 绑定input事件，获取关键字
 *  2.2 获取展示城市列表数据
 */
// 2.1 绑定input事件，获取关键字
document.querySelector('.search-city').addEventListener('input', (e) =>
{
  console.log(e.target.value)
  // 2.2 获取展示城市列表数据
  myAxios({
    url: 'http://hmajax.itheima.net/api/weather/city',
    params: {
      city: e.target.value
    }
  }).then(result => {
    console.log(result)
    const liStr = result.data.map(item => {
      return `<li class="city-item" data-
code="${item.code}">${item.name}</li>`
    }).join('')
    console.log(liStr)
    document.querySelector('.search-list').innerHTML = liStr
  })
})
```

小结

1. 监听输入框实时改变的事件是什么?

   ▶ 答案

14.案例_天气预报-展示城市天气

目标

点击搜索框列表城市名字，切换对应城市天气数据

讲解

1. 介绍本视频要完成的效果：点击城市列表名字，切换当前页面天气数据

2. 步骤

   1. 检测搜索列表点击事件，获取城市 code 值
   2. 复用获取展示城市天气函数

3. 核心代码如下：

```
/**
 * 目标3：切换城市天气
 *  3.1 绑定城市点击事件，获取城市code值
 *  3.2 调用获取并展示天气的函数
 */
// 3.1 绑定城市点击事件，获取城市code值
document.querySelector('.search-list').addEventListener('click', e => {
  if (e.target.classList.contains('city-item')) {
    // 只有点击城市li才会走这里
    const cityCode = e.target.dataset.code
    console.log(cityCode)
    // 3.2 调用获取并展示天气的函数
    getWeather(cityCode)
  }
})
```

小结

1. 这次我们获取城市天气，传递的是城市名字还是 code 值?

   ▶ 答案

今日重点(必须会)

1. 了解 AJAX 原理之 XMLHttpRequest（XHR）相关语法
2. 了解 Promise 的作用和三种状态
3. 了解 axios 内部运作的过程
4. 完成案例-天气预报

今日作业(必完成)

参考作业文件夹作用

参考文档

1. Ajax原生-mdn
2. 同步异步-mdn
3. 回调函数-mdn
4. Promise-mdn

Day04_AJAX进阶

知识点自测

1. 看如下标签回答如下问题?

```
<select>
    <option value="北京">北京市</option>
    <option value="南京">南京市</option>
    <option value="天津">天津市</option>
</select>
```

   - 当选中第二个 option 时, JS 中获取下拉菜单 select 标签的 value 属性的值是多少?

     ▶ 答案

- 页面上看到的是北京, 还是北京市等?

  ▶ 答案

2. 我给 select 标签的 value 属性赋予"南京"会有什么效果?

   ▶ 答案

目录

学习目标

1. 区分异步代码，回调函数地狱问题和所有解决防范（Promise 链式调用）
2. 掌握 async 和 await 使用
3. 掌握 EventLoop 的概念
4. 了解 Promise.all 静态方法作用
5. 完成省市区切换效果

01.同步代码和异步代码

目标

能够区分出哪些是异步代码

讲解

1. 同步代码：逐行执行，需原地等待结果后，才继续向下执行
2. 异步代码：调用后耗时，不阻塞代码继续执行（不必原地等待），在将来完成后触发回调函数传递结果
3. 回答代码打印顺序：发现异步代码接收结果，使用的都是回调函数

```js
const result = 0 + 1
console.log(result)
setTimeout(() => {
  console.log(2)
}, 2000)
document.querySelector('.btn').addEventListener('click', () => {
  console.log(3)
})
document.body.style.backgroundColor = 'pink'
console.log(4)
```

> 结果：1， 4， 2
>
> 按钮点击一次打印一次 3

小结

1. 什么是同步代码?
   ▶ 答案
2. 什么是异步代码?
   ▶ 答案
3. JS 中有哪些异步代码?
   ▶ 答案
4. 异步代码如何接收结果?
   ▶ 答案

02.回调函数地狱

目标

了解回调函数地狱的概念和缺点

讲解

1. 需求：展示默认第一个省，第一个城市，第一个地区在下拉菜单中

2. 概念：在回调函数中嵌套回调函数，一直嵌套下去就形成了回调函数地狱
3. 缺点：可读性差，异常无法捕获，耦合性严重，牵一发动全身

```
axios({ url: 'http://hmajax.itheima.net/api/province' }).then(result =>
{
  const pname = result.data.list[0]
  document.querySelector('.province').innerHTML = pname
  // 获取第一个省份默认下属的第一个城市名字
  axios({ url: 'http://hmajax.itheima.net/api/city', params: { pname }
}).then(result => {
    const cname = result.data.list[0]
    document.querySelector('.city').innerHTML = cname
    // 获取第一个城市默认下属第一个地区名字
    axios({ url: 'http://hmajax.itheima.net/api/area', params: { pname,
cname } }).then(result => {
      document.querySelector('.area').innerHTML = result.data.list[0]
    })
  })
})
```

小结

1. 什么是回调函数地狱?

   ▶ 答案
2. 回调函数地狱问题?

   ▶ 答案

03.Promise-链式调用

目标

了解 Promise 链式调用特点和语法

讲解

1. 概念：依靠 then() 方法会返回一个新生成的 Promise 对象特性，继续串联下一环任务，直到结束
2. 细节：then() 回调函数中的返回值，会影响新生成的 Promise 对象最终状态和结果
3. 好处：通过链式调用，解决回调函数嵌套问题

4. 按照图解，编写核心代码：

```
/**
 * 目标：掌握Promise的链式调用
 * 需求：把省市的嵌套结构，改成链式调用的线性结构
*/
// 1. 创建Promise对象-模拟请求省份名字
const p = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('北京市')
  }, 2000)
})

// 2. 获取省份名字
const p2 = p.then(result => {
  console.log(result)
  // 3. 创建Promise对象-模拟请求城市名字
  // return Promise对象最终状态和结果，影响到新的Promise对象
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(result + '--- 北京')
    }, 2000)
  })
})

// 4. 获取城市名字
p2.then(result => {
  console.log(result)
})

// then()原地的结果是一个新的Promise对象
console.log(p2 === p)
```

小结

1. 什么是 Promise 的链式调用?

   ▶ 答案

2. then 回调函数中，return 的值会传给哪里?

   ▶ 答案

3. Promise 链式调用有什么用?

   ▶ 答案

04.Promise-链式调用_解决回调地狱

目标

了解 Promise 链式调用解决回调地狱

讲解

1. 目标：使用 Promise 链式调用，解决回调函数地狱问题
2. 做法：每个 Promise 对象中管理一个异步任务，用 then 返回 Promise 对象，串联起来

3. 按照图解思路，编写核心代码：

```
/**
 * 目标：把回调函数嵌套代码，改成Promise链式调用结构
 * 需求：获取默认第一个省，第一个市，第一个地区并展示在下拉菜单中
*/
let pname = ''
// 1. 得到-获取省份Promise对象
axios({url: 'http://hmajax.itheima.net/api/province'}).then(result => {
  pname = result.data.list[0]
  document.querySelector('.province').innerHTML = pname
  // 2. 得到-获取城市Promise对象
  return axios({url: 'http://hmajax.itheima.net/api/city', params: {
pname }})
}).then(result => {
  const cname = result.data.list[0]
  document.querySelector('.city').innerHTML = cname
  // 3. 得到-获取地区Promise对象
  return axios({url: 'http://hmajax.itheima.net/api/area', params: {
pname, cname }})
}).then(result => {
```

```
  console.log(result)
  const areaName = result.data.list[0]
  document.querySelector('.area').innerHTML = areaName
})
```

小结

1. Promise 链式调用如何解决回调函数地狱?

   ▶ 答案

05.async 函数和 await

目标

掌握 async 和 await 语法来编写简洁的异步代码

讲解

1. 概念：在 async 函数内，使用 await 关键字取代 then 函数，等待获取 Promise 对象成功状态的结果值
2. 做法：使用 async 和 await 解决回调地狱问题
3. 核心代码：

```
/**
 * 目标：掌握async和await语法，解决回调函数地狱
 * 概念：在async函数内，使用await关键字，获取Promise对象"成功状态"结果值
 * 注意：await必须用在async修饰的函数内（await会阻止"异步函数内"代码继续执行，原地等待结果）
*/
// 1. 定义async修饰函数
async function getData() {
  // 2. await等待Promise对象成功的结果
  const pObj = await axios({url: 'http://hmajax.itheima.net/api/province'})
  const pname = pObj.data.list[0]
  const cObj = await axios({url: 'http://hmajax.itheima.net/api/city', params: { pname }})
  const cname = cObj.data.list[0]
  const aObj = await axios({url: 'http://hmajax.itheima.net/api/area', params: { pname, cname }})
  const areaName = aObj.data.list[0]


  document.querySelector('.province').innerHTML = pname
  document.querySelector('.city').innerHTML = cname
  document.querySelector('.area').innerHTML = areaName
}

```

```
getData()
```

> 使用 await 替代 then 的方法

小结

1. await 的作用是什么?

   ▶ 答案

06.async 函数和 await 捕获错误

目标

了解用 try catch 捕获同步流程的错误

讲解

1. try 和 catch 的作用：语句标记要尝试的语句块，并指定一个出现异常时抛出的响应

```
try {
  // 要执行的代码
} catch (error) {
  // error 接收的是，错误消息
  // try 里代码，如果有错误，直接进入这里执行
}
```

> try 里有报错的代码，会立刻跳转到 catch 中

2. 尝试把代码中 url 地址写错，运行观察 try catch 的捕获错误信息能力

```
/**
 * 目标：async和await_错误捕获
*/
async function getData() {
  // 1. try包裹可能产生错误的代码
  try {
    const pObj = await axios({ url:
'http://hmajax.itheima.net/api/province' })
    const pname = pObj.data.list[0]
    const cObj = await axios({ url:
'http://hmajax.itheima.net/api/city', params: { pname } })
    const cname = cObj.data.list[0]
    const aObj = await axios({ url:
'http://hmajax.itheima.net/api/area', params: { pname, cname } })
    const areaName = aObj.data.list[0]

    document.querySelector('.province').innerHTML = pname
    document.querySelector('.city').innerHTML = cname
    document.querySelector('.area').innerHTML = areaName
```

```
  } catch (error) {
    // 2. 接着调用catch块，接收错误信息
    // 如果try里某行代码报错后，try中剩余的代码不会执行了
    console.dir(error)
  }
}

getData()
```

小结

1. try 和 catch 有什么作用?

   ▶ 答案

07.事件循环

目标

掌握事件循环模型是如何执行异步代码的

讲解

1. 事件循环（EventLoop）：掌握后知道 JS 是如何安排和运行代码的

   > 请回答下面 2 段代码打印的结果，并说明原因

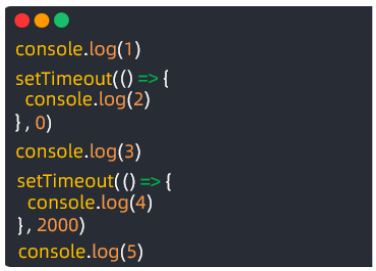
```
console.log(1)
setTimeout(() => {
  console.log(2)
}, 2000)
```

```
console.log(1)
setTimeout(() => {
  console.log(2)
}, 0)
console.log(3)
```

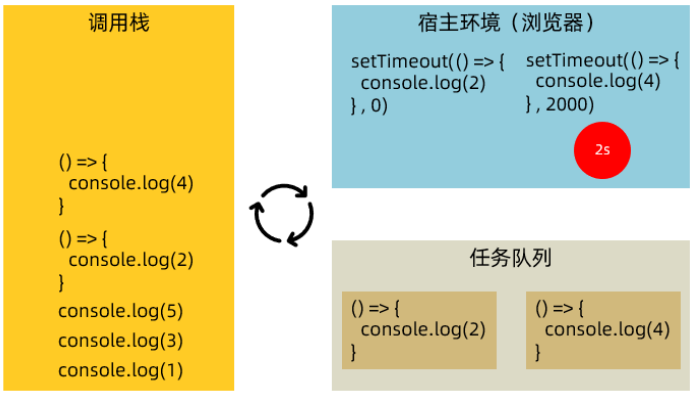
2. 作用：事件循环负责执行代码，收集和处理事件以及执行队列中的子任务
3. 原因：JavaScript 单线程（某一刻只能执行一行代码），为了让耗时代码不阻塞其他代码运行，设计了事件循环模型
4. 概念：执行代码和收集异步任务的模型，在调用栈空闲，反复调用任务队列里回调函数的执行机制，就叫事件循环

```
/**
 * 目标：阅读并回答执行的顺序结果
*/
console.log(1)
setTimeout(() => {
  console.log(2)
}, 0)
console.log(3)
setTimeout(() => {
  console.log(4)
}, 2000)
console.log(5)
```

> 具体运行过程，请参考 PPT 动画和视频讲解



控制台输出：1 3 5 2 4



小结

1. 什么是事件循环?

   ▶ 答案

2. 为什么有事件循环?

   ▶ 答案

3. JavaScript 内代码如何执行?

   ▶ 答案

08.事件循环-练习

目标

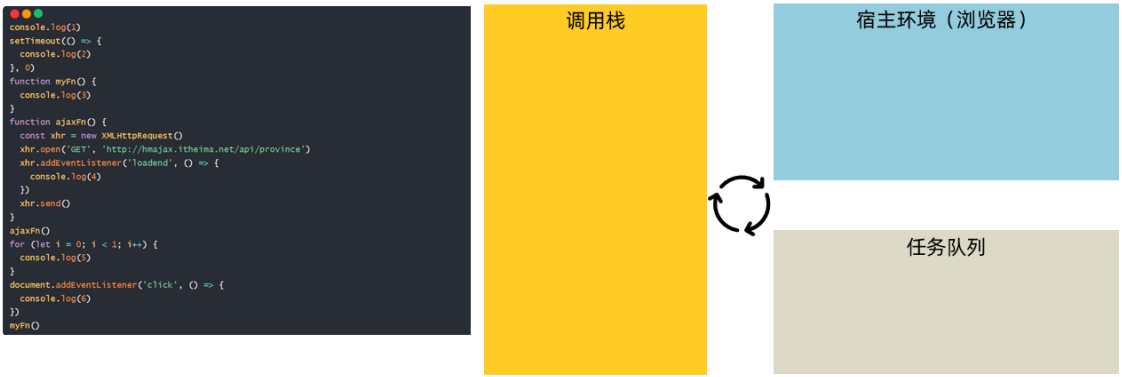
了解事件循环的执行模型

讲解

1. 需求：请根据掌握的事件循环的模型概念，分析代码执行过程

```
/**
 * 目标：阅读并回答执行的顺序结果
```

```
*/
console.log(1)
setTimeout(() => {
  console.log(2)
}, 0)
function myFn() {
  console.log(3)
}
function ajaxFn() {
  const xhr = new XMLHttpRequest()
  xhr.open('GET', 'http://hmajax.itheima.net/api/province')
  xhr.addEventListener('loadend', () => {
    console.log(4)
  })
  xhr.send()
}
for (let i = 0; i < 1; i++) {
  console.log(5)
}
ajaxFn()
document.addEventListener('click', () => {
  console.log(6)
})
myFn()
```



> 结果：1 5 3 2 4 点击一次document就会执行一次打印6

小结

暂无

09.宏任务与微任务

目标

掌握微任务和宏任务的概念和区分

讲解

1. ES6 之后引入了 Promise 对象， 让 JS 引擎也可以发起异步任务
2. 异步任务划分为了
   - 宏任务：由浏览器环境执行的异步代码
   - 微任务：由 JS 引擎环境执行的异步代码
3. 宏任务和微任务具体划分：

任务（代码）	执行所在环境
JS脚本执行事件（script）	浏览器
setTimeout/setInterval	浏览器
AJAX请求完成事件	浏览器
用户交互事件等	浏览器

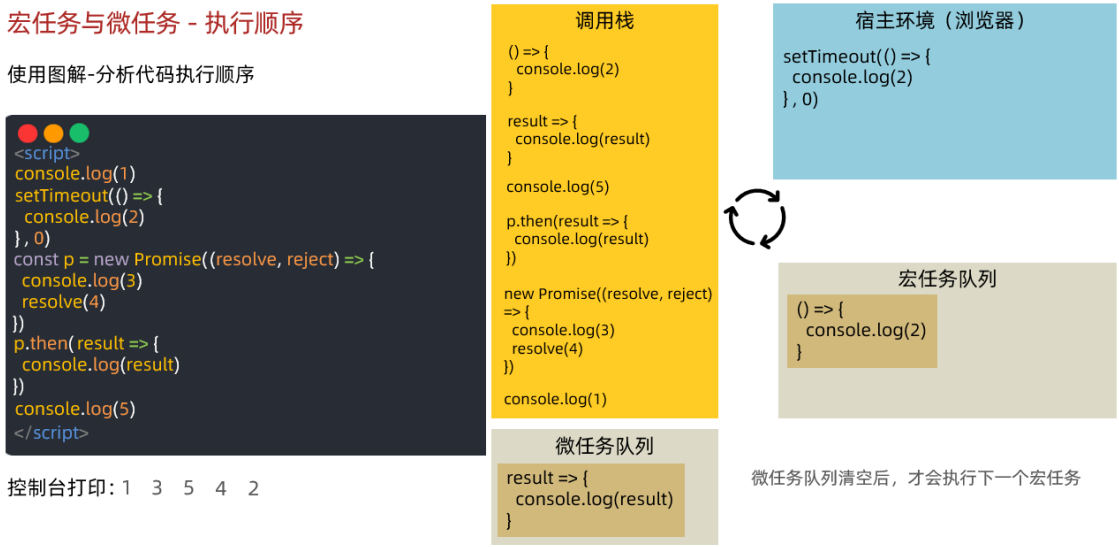
任务（代码）	执行所在环境
Promise对象.then()	JS 引擎

Promise 本身是同步的，而then和catch回调函数是异步的

4. 事件循环模型

> 具体运行效果，参考 PPT 动画或者视频

```
/**
 * 目标：阅读并回答打印的执行顺序
*/
console.log(1)
setTimeout(() => {
  console.log(2)
}, 0)
const p = new Promise((resolve, reject) => {
  resolve(3)
})
p.then(res => {
  console.log(res)
})
console.log(4)
```

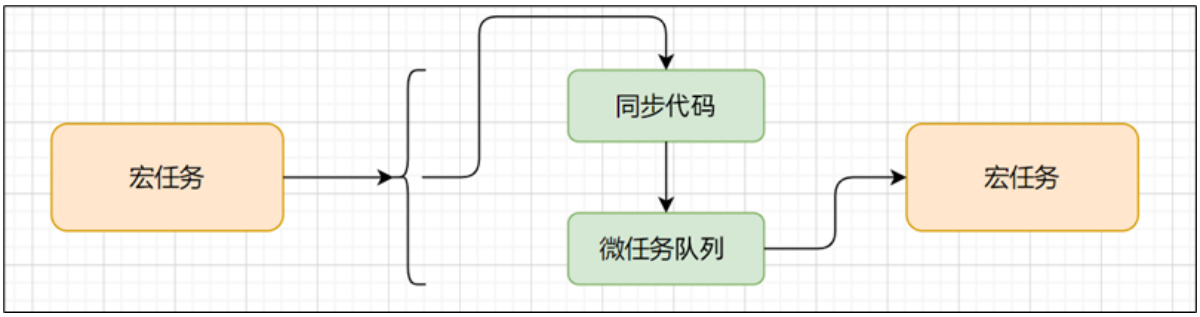


> 注意：宏任务每次在执行同步代码时，产生微任务队列，清空微任务队列任务后，微任务队列空间释放！

> 下一次宏任务执行时，遇到微任务代码，才会再次申请微任务队列空间放入回调函数消息排队
>
> 总结：一个宏任务包含微任务队列，他们之间是包含关系，不是并列关系

小结

1. 什么是宏任务?
   ▶ 答案
2. 什么是微任务?
   ▶ 答案
3. JavaScript 内代码如何执行?
   ▶ 答案



10.事件循环 - 经典面试题

目标

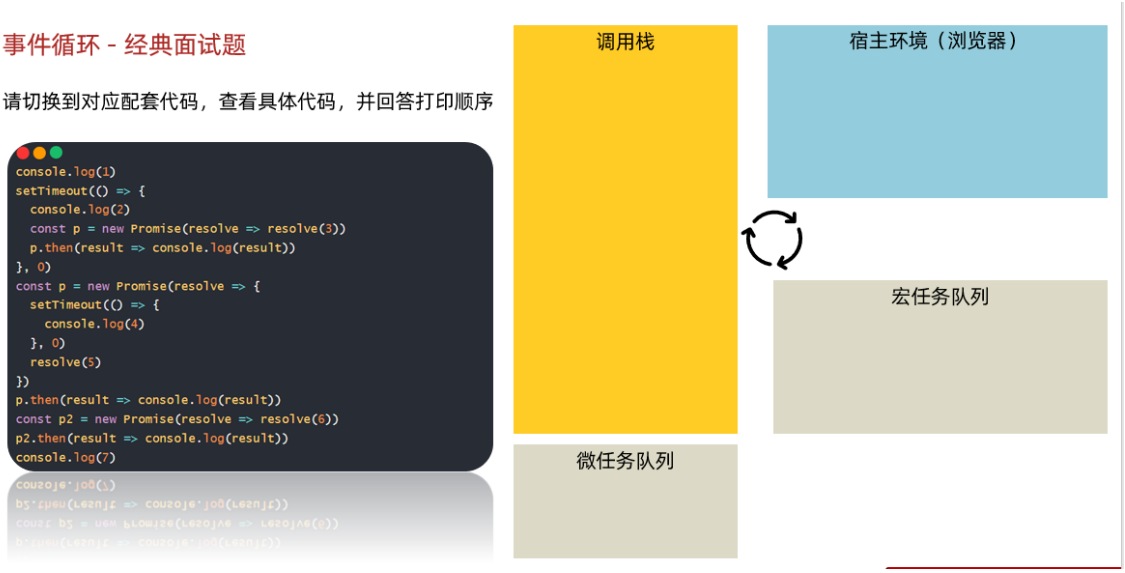
锻炼事件循环模型的使用

讲解

1. 需求：请切换到对应配套代码，查看具体代码，并回答打印顺序（锻炼事件循环概念的理解，阅读代码执行顺序_）

```
// 目标：回答代码执行顺序
console.log(1)
setTimeout(() => {
  console.log(2)
  const p = new Promise(resolve => resolve(3))
  p.then(result => console.log(result))
}, 0)
const p = new Promise(resolve => {
  setTimeout(() => {
    console.log(4)
  }, 0)
  resolve(5)
})
p.then(result => console.log(result))
const p2 = new Promise(resolve => resolve(6))
p2.then(result => console.log(result))
```

```
17 console.log(7)
```



小结

暂无

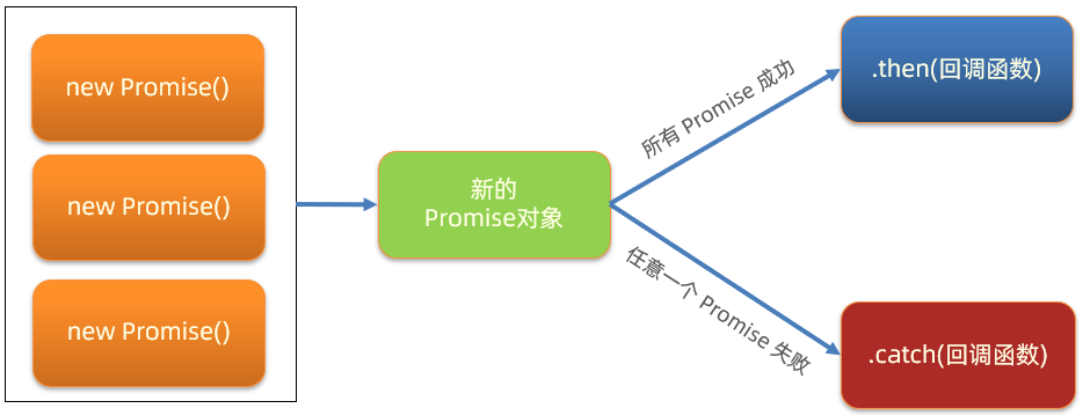
11.Promise.all 静态方法

目标

了解 Promise.all 作用和使用场景

讲解

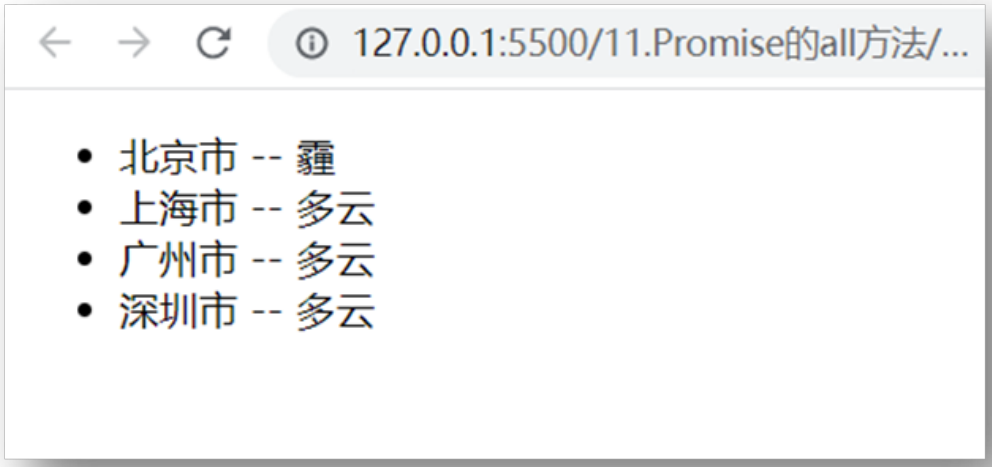
1. 概念：合并多个 Promise 对象，等待所有同时成功完成（或某一个失败），做后续逻辑



2. 语法：

```
const p = Promise.all([Promise对象, Promise对象, ...])
p.then(result => {
  // result 结果: [Promise对象成功结果, Promise对象成功结果, ...]
}).catch(error => {
  // 第一个失败的 Promise 对象，抛出的异常对象
})
```

3. 需求：同时请求“北京”，“上海”，“广州”，“深圳”的天气并在网页尽可能同时显示



4. 核心代码如下：

```
<!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>Promise的all方法</title>
</head>

<body>
  <ul class="my-ul"></ul>
  <script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js">
</script>
  <script>
    /**
     * 目标：掌握Promise的all方法作用，和使用场景
     * 业务：当我需要同一时间显示多个请求的结果时，就要把多请求合并
     * 例如：默认显示"北京", "上海", "广州", "深圳"的天气在首页查看
     * code：
     * 北京-110100
     * 上海-310100
     * 广州-440100
     * 深圳-440300
    */
    // 1. 请求城市天气，得到Promise对象
```

```
    const bjPromise = axios({ url: 'http://hmajax.itheima.net/api/weather', params: { city: '110100' } })
    const shPromise = axios({ url: 'http://hmajax.itheima.net/api/weather', params: { city: '310100' } })
    const gzPromise = axios({ url: 'http://hmajax.itheima.net/api/weather', params: { city: '440100' } })
    const szPromise = axios({ url: 'http://hmajax.itheima.net/api/weather', params: { city: '440300' } })

    // 2. 使用Promise.all，合并多个Promise对象
    const p = Promise.all([bjPromise, shPromise, gzPromise, szPromise])
    p.then(result => {
      // 注意：结果数组顺序和合并时顺序是一致
      console.log(result)
      const htmlStr = result.map(item => {
        return `<li>${item.data.data.area} --- ${item.data.data.weather}</li>`
      }).join('')
      document.querySelector('.my-ul').innerHTML = htmlStr
    }).catch(error => {
      console.dir(error)
    })
  </script>
</body>

</html>
```

小结

1. Promise.all 什么时候使用?

▶ 答案

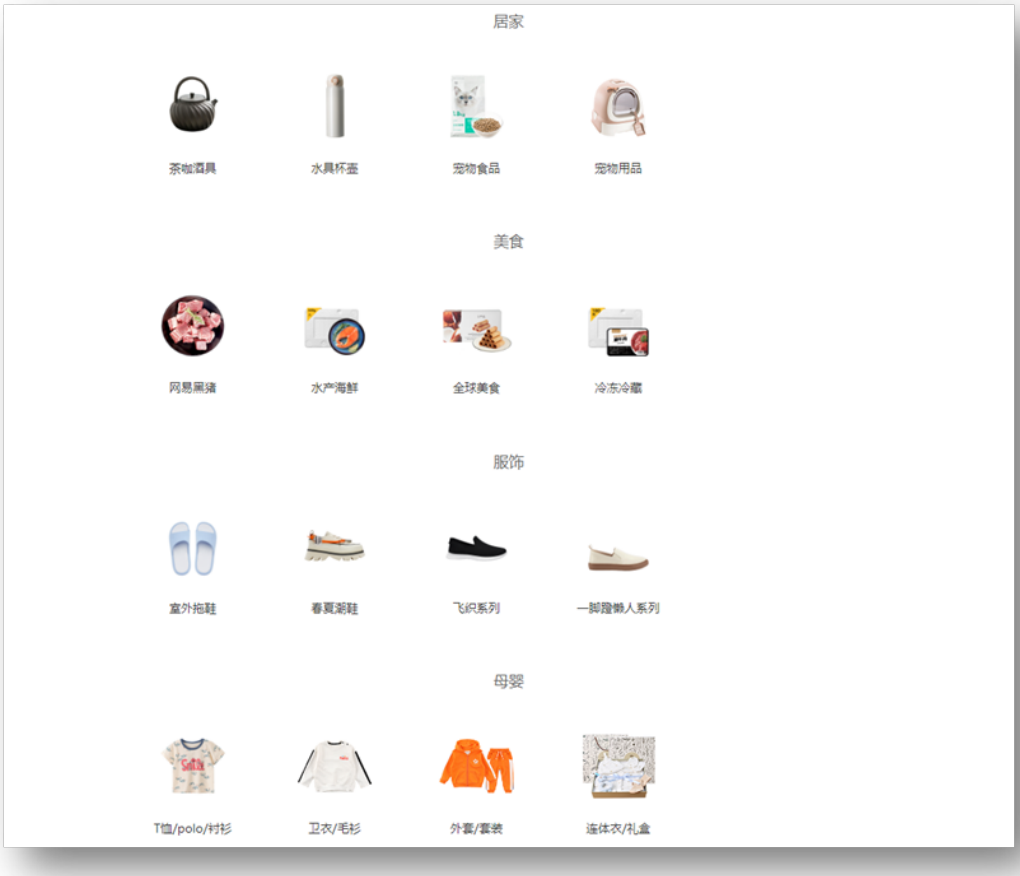
12.案例-商品分类

目标

完成商品分类效果

讲解

1. 需求：尽可能同时展示所有商品分类到页面上



2. 步骤：
    1. 获取所有的一级分类数据
    2. 遍历id，创建获取二级分类请求
    3. 合并所有二级分类Promise对象
    4. 等待同时成功，开始渲染页面
3. 核心代码：

```
/**
 * 目标：把所有商品分类“同时”渲染到页面上
 *  1. 获取所有一级分类数据
 *  2. 遍历id，创建获取二级分类请求
 *  3. 合并所有二级分类Promise对象
 *  4. 等待同时成功后，渲染页面
*/
// 1. 获取所有一级分类数据
axios({
  url: 'http://hmajax.itheima.net/api/category/top'
}).then(result => {
  console.log(result)
  // 2. 遍历id，创建获取二级分类请求
  const secPromiseList = result.data.data.map(item => {
    return axios({
      url: 'http://hmajax.itheima.net/api/category/sub',
      params: {
        id: item.id // 一级分类id
      }
    })
```

```
  })
  console.log(secPromiseList) // [二级分类请求Promise对象，二级分类请求Promise对象，...]
  // 3. 合并所有二级分类Promise对象
  const p = Promise.all(secPromiseList)
  p.then(result => {
    console.log(result)
    // 4. 等待同时成功后，渲染页面
    const htmlStr = result.map(item => {
      const dataObj = item.data.data // 取出关键数据对象
      return `<div class="item">
    <h3>${dataObj.name}</h3>
    <ul>
      ${dataObj.children.map(item => {
        return `<li>
        <a href="javascript:;">
          <img src="${item.picture}">
          <p>${item.name}</p>
        </a>
      </li>`
      }).join('')}
    </ul>
  </div>`
    }).join('')
    console.log(htmlStr)
    document.querySelector('.sub-list').innerHTML = htmlStr
  })
})
```

小结

暂无

13.案例-学习反馈-省市区切换

目标

完成省市区切换效果

讲解

1. 需求：完成省市区切换效果

2. 步骤：

    1. 设置省份数据到下拉菜单
    2. 切换省份，设置城市数据到下拉菜单，并清空地区下拉菜单
    3. 切换城市，设置地区数据到下拉菜单

3. 核心代码：

```
/**
 * 目标1：完成省市区下拉列表切换
 *  1.1 设置省份下拉菜单数据
 *  1.2 切换省份，设置城市下拉菜单数据，清空地区下拉菜单
 *  1.3 切换城市，设置地区下拉菜单数据
 */
// 1.1 设置省份下拉菜单数据
axios({
  url: 'http://hmajax.itheima.net/api/province'
}).then(result => {
  const optionStr = result.data.list.map(pname => `<option value="${pname}">${pname}</option>`).join('')
  document.querySelector('.province').innerHTML = `<option value="">省份</option>` + optionStr
})

// 1.2 切换省份，设置城市下拉菜单数据，清空地区下拉菜单
document.querySelector('.province').addEventListener('change', async e => {
  // 获取用户选择省份名字
  // console.log(e.target.value)
  const result = await axios({ url: 'http://hmajax.itheima.net/api/city', params: { pname: e.target.value } })
```

```
  const optionStr = result.data.list.map(cname => `<option value="${cname}">${cname}</option>`).join('')
  // 把默认城市选项+下属城市数据插入select中
  document.querySelector('.city').innerHTML = `<option value="">城市</option>` + optionStr

  // 清空地区数据
  document.querySelector('.area').innerHTML = `<option value="">地区</option>`
})

// 1.3 切换城市，设置地区下拉菜单数据
document.querySelector('.city').addEventListener('change', async e => {
  console.log(e.target.value)
  const result = await axios({url: 'http://hmajax.itheima.net/api/area', params: {
    pname: document.querySelector('.province').value,
    cname: e.target.value
  }})
  console.log(result)
  const optionStr = result.data.list.map(aname => `<option value="${aname}">${aname}</option>`).join('')
  console.log(optionStr)
  document.querySelector('.area').innerHTML = `<option value="">地区</option>` + optionStr
})
```

小结

暂无

14.案例-学习反馈-数据提交

目标

完成学习反馈数据提交

讲解

1. 需求：收集学习反馈数据，提交保存

2. 步骤：

    1. 监听提交按钮的点击事件
    2. 依靠插件收集表单数据
    3. 基于 axios 提交保存，显示结果

3. 核心代码如下：

```
/**
 * 目标2：收集数据提交保存
 *  2.1 监听提交的点击事件
 *  2.2 依靠插件收集表单数据
 *  2.3 基于axios提交保存，显示结果
 */
// 2.1 监听提交的点击事件
document.querySelector('.submit').addEventListener('click', async () =>
{
  // 2.2 依靠插件收集表单数据
  const form = document.querySelector('.info-form')
  const data = serialize(form, { hash: true, empty: true })
  console.log(data)
  // 2.3 基于axios提交保存，显示结果
  try {
    const result = await axios({
      url: 'http://hmajax.itheima.net/api/feedback',
      method: 'POST',
      data
    })
    console.log(result)
    alert(result.data.message)
  } catch (error) {
    console.dir(error)
```

```
    alert(error.response.data.message)
  }
})
```

小结

暂无

今日重点(必须会)

1. 掌握 async 和 await 的使用
2. 理解 EventLoop 和宏任务微任务执行顺序
3. 了解 Promise.all 的作用和使用场景
4. 完成案例-学习反馈

今日作业(必完成)

参考作业文件夹里md文档的要求

参考文献

1. async和await的mdn讲解

ES6

Vue3