JavaScript(WebAPI)

WebAPI 背景知识

什么是 WebAPI

前面学习的 JS 分成三个大的部分

- ECMAScript: 基础语法部分
- DOM API: 操作页面结构
- BOM API: 操作浏览器

WebAPI 就包含了 DOM + BOM.

这个是 W3C 组织规定的. (和制定 ECMAScript 标准的大佬们不是一伙人).

> 前面学的 JS 基础语法主要学的是 ECMAScript, 这让我们建立基本的编程思维. 相当于练武需要先扎马步.
>
> 但是真正来写一个更加复杂的有交互式的页面, 还需要 WebAPI 的支持. 相当于各种招式.

什么是 API

API 是一个更广义的概念. 而 WebAPI 是一个更具体的概念, 特指 DOM+BOM

所谓的 API 本质上就是一些现成的函数/对象, 让程序猿拿来就用, 方便开发.

> 相当于一个工具箱. 只不过程序猿用的工具箱数目繁多, 功能复杂.

API 参考文档

https://developer.mozilla.org/zh-CN/docs/Web/API

> 可以在搜索引擎中按照 "MDN + API 关键字" 的方式搜索, 也能快速找到需要的 API 文档.

DOM 基本概念

什么是 DOM

DOM 全称为 Document Object Model.

W3C 标准给我们提供了一系列的函数, 让我们可以操作:

- 网页内容
- 网页结构
- 网页样式

DOM 树

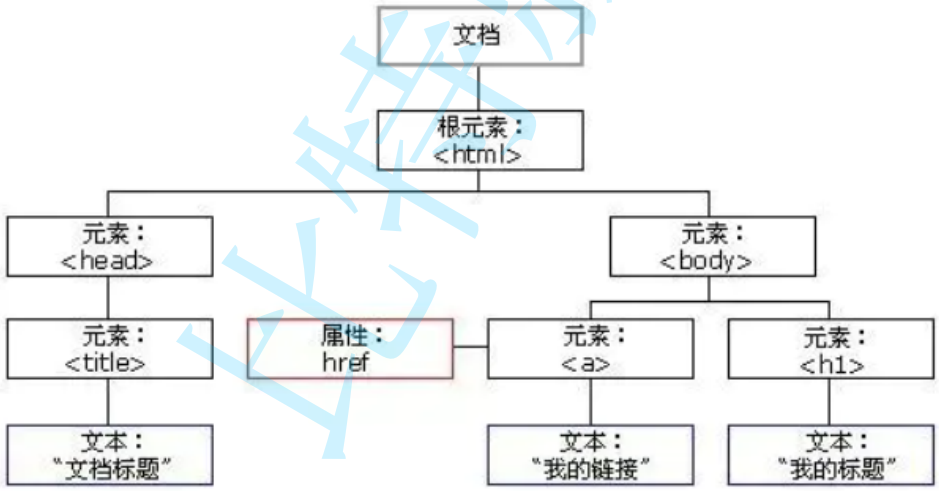
一个页面的结构是一个树形结构, 称为 DOM 树.

> 树形结构在数据结构阶段会介绍. 就可以简单理解成类似于 "家谱" 这种结构

页面结构形如:

```
<html>
<head>
<title>文档标题</title>
</head>
<body>
<a href="" >我的链接</a>
<h1>我的标题</h1>
</body>
</html>
```

DOM 树结构形如



重要概念:

- 文档: 一个页面就是一个 **文档**, 使用 document 表示.
- 元素: 页面中所有的标签都称为 **元素**. 使用 element 表示.
- 节点: 网页中所有的内容都可以称为 **节点**(标签节点, 注释节点, 文本节点, 属性节点等). 使用 node 表示.

这些文档等概念在 JS 代码中就对应一个个的对象.

> 所以才叫 "文档对象模型" .

获取元素

这部分工作类似于 CSS 选择器的功能.

querySelector

> 这个是 HTML5 新增的, IE9 及以上版本才能使用.

前面的几种方式获取元素的时候都比较麻烦. 而使用 querySelector 能够完全复用前面学过的 CSS 选择器知识, 达到更快捷更精准的方式获取到元素对象.

```
var element = document.querySelector(selectors);
```

- selectors 包含一个或多个要匹配的选择器的 DOM字符串 `DOMString`。 该字符串必须是有效的CSS选择器字符串；如果不是，则引发 `SYNTAX_ERR` 异常
- 表示文档中与指定的一组CSS选择器匹配的第一个元素的 html元素 `Element` 对象.
- 如果您需要与指定选择器匹配的所有元素的列表，则应该使用 `querySelectorAll()`
- 可以在任何元素上调用，不仅仅是 document。 调用这个方法的元素将作为本次查找的根元素

> 正因为参数是选择器, 所以一定要通过特殊符号指定是哪种选择器.
>
> 例如 .box 是类选择器, #star 是 id 选择器 等.

```
<div class="box">abc</div>
<div id="id">def</div>
<h3><span><input type="text"></span></h3>
<script>
    var elem1 = document.querySelector('.box');
    console.log(elem1);
    var elem2 = document.querySelector('#id');
    console.log(elem2);
    var elem3 = document.querySelector('h3 span input');
    console.log(elem3);
</script>
```

querySelectorAll

使用 `querySelectorAll` 用法和上面类似.

```
<div class="box">abc</div>
<div id="id">def</div>
<script>
    var elems = document.querySelectorAll('div');
console.log(elems);
</script>
```

事件初识

基本概念

JS 要构建动态页面, 就需要感知到用户的行为.

用户对于页面的一些操作(点击, 选择, 修改等) 操作都会在浏览器中产生一个个事件, 被 JS 获取到, 从而进行更复杂的交互操作.

> 浏览器就是一个哨兵, 在侦查敌情(用户行为). 一旦用户有反应(触发具体动作), 哨兵就会点燃烽火台的狼烟(事件), 后方就可以根据狼烟来决定下一步的对敌策略.

事件三要素

1. 事件源: 哪个元素触发的
2. 事件类型: 是点击, 选中, 还是修改?
3. 事件处理程序: 进一步如何处理. 往往是一个回调函数.

简单示例

```
<button id="btn">点我一下</button>
<script>
    var btn = document.getElementById('btn');
    btn.onclick = function () {
        alert("hello world");
    }
</script>
```

> - btn 按钮就是事件源.
> - 点击就是事件类型
> - function 这个匿名函数就是事件处理程序
> - 其中 `btn.onclick = function()` 这个操作称为 **注册事件/绑定事件**

注意: 这个匿名函数相当于一个回调函数, 这个函数不需要程序猿主动来调用, 而是交给浏览器, 由浏览器自动在合适的时机(触发点击操作时) 进行调用.

操作元素

获取/修改元素内容

1. innerText

`Element.innerText` 属性表示一个节点及其后代的"渲染"文本内容

```
// 读操作
var renderedText = HTMLElement.innerText;
// 写操作
HTMLElement.innerText = string;
```

> 不识别 html 标签. 是非标准的(IE发起的). 读取结果不保留html源码中的 换行 和 空格.

```
<div>
    <span>hello world</span>
    <span>hello world</span>
</div>
<script>
    var div = document.querySelector('div');
    // 读取 div 内部内容
    console.log(div.innerText);
    // 修改 div 内部内容, 界面上就会同步修改
    div.innerText = 'hello js <span>hello js</span>';
</script>
```

可以看到, 通过 innerText 无法获取到 div 内部的 html 结构, 只能得到文本内容.

修改页面的时候也会把 span 标签当成文本进行设置.

2. innerHTML

`Element.innerHTML` 属性设置或获取HTML语法表示的元素的后代.

```
// 读操作
var content = element.innerHTML;
// 写操作
element.innerHTML = htmlString;
```

1. 先获取到事件源的元素
2. 注册事件

> 识别 html 标签. W3C 标准的. 读取结果保留html源码中的 换行 和 空格.

```
<div>
    <span>hello world</span>
    <span>hello world</span>
</div>
<script>
    var div = document.querySelector('div');
    // 读取页面内容
    console.log(div.innerHTML);
    // 修改页面内容
    div.innerHTML = '<span>hello js</span>'
</script>
```

可以看到 innerHTML 不光能获取到页面的 html 结构, 同时也能修改结构. 并且获取到的内容保留的空格和换行.

> innerHTML 用的场景比 innerText 更多.

获取/修改元素属性

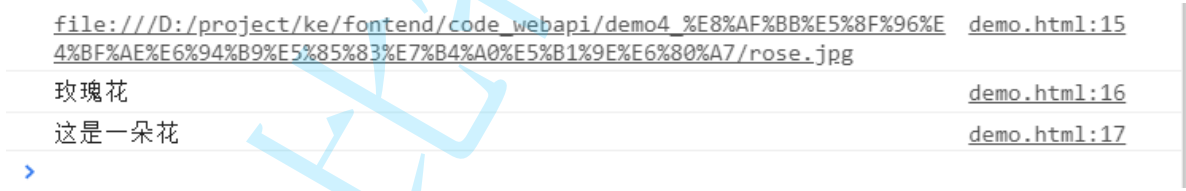
可以通过 Element 对象的属性来直接修改, 就能影响到页面显示效果.

```
<img src="rose.jpg" alt="这是一朵花" title="玫瑰花">
<script>
    var img = document.querySelector('img');
    console.dir(img);
</script>
```

此时可以看到 img 这个 `Element` 对象中有很多属性

我们可以在代码中直接通过这些属性来获取属性的值.

```
<img src="rose.jpg" alt="这是一朵花" title="玫瑰花">
<script>
    var img = document.querySelector('img');
    // console.dir(img);
    console.log(img.src);
    console.log(img.title);
    console.log(img.alt);
</script>
```



还可以直接修改属性

```
<img src="rose.jpg" alt="这是一朵花" title="玫瑰花">
<script>
    var img = document.querySelector('img');
    img.onclick = function () {
        if (img.src.lastIndexOf('rose.jpg') !== -1) {
            img.src = './rose2.png';
        } else {
            img.src = './rose.jpg';
        }
    }
</script>
```

> 此时点击图片就可以切换图片显示状态. (需要提前把两个图片准备好)

获取/修改表单元素属性

表单(主要是指 input 标签)的以下属性都可以通过 DOM 来修改

- value: input 的值.
- disabled: 禁用
- checked: 复选框会使用
- selected: 下拉框会使用
- type: input 的类型(文本, 密码, 按钮, 文件等)

代码示例: 切换按钮的文本.

> 假设这是个播放按钮, 在 "播放" - "暂停" 之间切换.

```
<input type="button" value="播放">
<script>
    var btn = document.querySelector('input');
    btn.onclick = function () {
        if (btn.value === '播放') {
            btn.value = '暂停';
        } else {
            btn.value = '播放';
        }
    }
</script>
```

代码示例: 点击计数

> 使用一个输入框输入初始值(整数). 每次点击按钮, 值 + 1

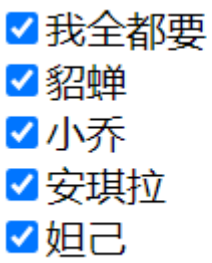
```
<input type="text" id="text" value="0">
<input type="button" id="btn" value='点我+1'>

<script>
    var text = document.querySelector('#text');
    var btn = document.querySelector('#btn');

    btn.onclick = function () {
        var num = +text.value;
        console.log(num);
        num++;
        text.value = num;
    }
</script>
```

- input 具有一个重要的属性 value, 这个 value 决定了表单元素的内容
- 如果是输入框, value 表示输入框的内容, 修改这个值会影响到界面显式; 在界面上修改这个值也会影响到代码中的属性
- 如果是按钮, value 表示按钮的内容. 可以通过这个来实现按钮中文本的替换

代码示例: 全选/取消全选按钮



> 1. 点击全选按钮, 则选中所有选项
> 2. 只要某个选项取消, 则自动取消全选按钮的勾选状态.

```
<input type="checkbox" id="all">我全都要 <br>
<input type="checkbox" class="girl">貂蝉 <br>
<input type="checkbox" class="girl">小乔 <br>
<input type="checkbox" class="girl">安琪拉 <br>
<input type="checkbox" class="girl">妲己 <br>

<script>
    // 1. 获取到元素
    var all = document.querySelector('#all');
    var girls = document.querySelectorAll('.girl');
    // 2. 给 all 注册点击事件, 选中/取消所有选项
    all.onclick = function () {
        for (var i = 0; i < girls.length; i++) {
            girls[i].checked = all.checked;
        }
    }
    // 3. 给 girl 注册点击事件
    for (var i = 0; i < girls.length; i++) {
        girls[i].onclick = function () {
            // 检测当前是不是所有的 girl 都被选中了.
            all.checked = checkGirls(girls);
        }
    }
    // 4. 实现 checkGirls
    function checkGirls(girls) {
        for (var i = 0; i < girls.length; i++) {
            if (!girls[i].checked) {
                // 只要一个 girl 没被选中, 就认为结果是 false(找到了反例)
                return false;
            }
        }
        // 所有 girl 中都没找到反例, 结果就是全选中
        return true;
    }
</script>
```

获取/修改样式属性

CSS 中指定给元素的属性, 都可以通过 JS 来修改.

行内样式操作

```
element.style.[属性名] = [属性值];
element.style.cssText = [属性名+属性值];
```

"行内样式", 通过 style 直接在标签上指定的样式. 优先级很高.

适用于改的样式少的情况

代码示例: 点击文字则放大字体.

> style 中的属性都是使用 **驼峰命名** 的方式和 CSS 属性对应的.
>
> 例如: font-size => fontSize, background-color => backgroundColor 等
>
> 这种方式修改只影响到特定样式, 其他内联样式的值不变.

```
<div style="font-size: 20px; font-weight: 700;">
    哈哈
</div>
<script>
    var div = document.querySelector('div');
    div.onclick = function () {
        var curFontSize = parseInt(this.style.fontSize);
        curFontSize += 10;
        this.style.fontSize = curFontSize + "px";
    }
</script>
```

类名样式操作

```
element.className = [CSS 类名];
```

修改元素的 CSS 类名. 适用于要修改的样式很多的情况.

> 由于 class 是 JS 的保留字, 所以名字叫做 className

代码示例: 开启夜间模式

```
这是一大段话.
这是一大段话.
这是一大段话.
这是一大段话.
```

- 点击页面切换到夜间模式(背景变成黑色)
- 再次点击恢复日间模式(背景变成白色)

```
<div class="container light">
    这是一大段话. <br>
    这是一大段话. <br>
    这是一大段话. <br>
    这是一大段话. <br>
</div>
```

```
* {
    margin: 0;
    padding: 0;
}

html,
body {
    width: 100%;
    height: 100%;
}

.container {
    width: 100%;
    height: 100%;

}

.light {
    background-color: #f3f3f3;
    color: #333;
}

.dark {
    background-color: #333;
    color: #f3f3f3;
}
```

```
var div = document.querySelector('div');
div.onclick = function () {
    console.log(div.className);
    if (div.className.indexOf('light') != -1) {
        div.className = 'container dark';
    } else {
        div.className = 'container light';
    }
}
```

操作节点

新增节点

分成两个步骤

1. 创建元素节点
2. 把元素节点插入到 dom 树中.

> 第一步相当于生了个娃, 第二步相当于给娃上户口.

1. 创建元素节点

使用 createElement 方法来创建一个元素. options 参数暂不关注.

```
var element = document.createElement(tagName[, options]);
```

代码示例:

```
<div class="container">

</div>

<script>
    var div = document.createElement('div');
    div.id = 'mydiv';
    div.className = 'box';
    div.innerHTML = 'hehe';
    console.log(div);
</script>
```

此时发现, 虽然创建出新的 div 了, 但是 div 并没有显示在页面上. 这是因为新创建的节点并没有加入到 DOM 树中.

> 上面介绍的只是创建元素节点, 还可以使用:
>
> - createTextNode 创建文本节点
> - createComment 创建注释节点
> - createAttribute 创建属性节点
>
> 我们以 createElement 为主即可.

2. 插入节点到 dom 树中

1) 使用 appendChild 将节点插入到指定节点的最后一个孩子之后

```
element.appendChild(aChild)
```

```
<div class="container">

</div>

<script>
    var div = document.createElement('div');
    div.id = 'mydiv';
    div.className = 'box';
    div.innerHTML = 'hehe';

    var container = document.querySelector('.container');
    container.appendChild(div);
</script>
```

```
▼<div class="container">
    <div id="mydiv" class="box">hehe</div>
  </div>
```

2) 使用 insertBefore 将节点插入到指定节点之前.

```
var insertedNode = parentNode.insertBefore(newNode, referenceNode);
```

> - `insertedNode` 被插入节点(newNode)
> - `parentNode` 新插入节点的父节点
> - `newNode` 用于插入的节点
> - `referenceNode` `newNode` 将要插在这个节点之前
>
> 如果 `referenceNode` 为 `null` 则 `newNode` 将被插入到子节点的末尾.
>
> 注意: *referenceNode* 引用节点**不是**可选参数.

```
<div class="container">
    <div>11</div>
    <div>22</div>
    <div>33</div>
    <div>44</div>
</div>

<script>
    var newDiv = document.createElement('div');
    newDiv.innerHTML = '我是新的节点';

    var container = document.querySelector('.container');
    console.log(container.children);
    container.insertBefore(newDiv, container.children[0]);
</script>
```

我是新的节点
11
22
33
44

注意1: 如果针对一个节点插入两次, 则只有最后一次生效(相当于把元素移动了)

```
<div class="container">
    <div>11</div>
    <div>22</div>
    <div>33</div>
    <div>44</div>
</div>

<script>
    var newDiv = document.createElement('div');
    newDiv.innerHTML = '我是新的节点';

    var container = document.querySelector('.container');
    console.log(container.children);
    // 此处的 children 里有 4 个元素
    container.insertBefore(newDiv, container.children[0]);
    // 此处的 children 里有 5 个元素(上面新插了一个), 0 号元素是 新节点,
    // 1 号元素是 11, 2号节点是 22, 所以是插入到 22 之前.
    container.insertBefore(newDiv, container.children[2]);
</script>
```

11
我是新的节点
22
33
44

注意2: 一旦一个节点插入完毕, 再针对刚刚的节点对象进行修改, 能够同步影响到 DOM 树中的内容.

```
<div class="container">
    <div>11</div>
    <div>22</div>
    <div>33</div>
    <div>44</div>
</div>

<script>
    var newDiv = document.createElement('div');
    newDiv.innerHTML = '我是新的节点';

    var container = document.querySelector('.container');
    console.log(container.children);
    container.insertBefore(newDiv, container.children[0]);
```

```
    // 插入完毕后再次修改 newDiv 的内容
    newDiv.innerHTML = '我是新节点2';
</script>
```

我是新节点2
11
22
33
44

删除节点

使用 removeChild 删除子节点

```
oldChild = element.removeChild(child);
```

- child 为待删除节点
- element 为 child 的父节点
- 返回值为该被删除节点
- 被删除节点只是从 dom 树被删除了, 但是仍然在内存中, 可以随时加入到 dom 树的其他位置.
- 如果上例中的 `child节点` 不是 `element` 节点的子节点,则该方法会抛出异常.

代码案例: 猜数字

预期效果

重新开始一局游戏
请输入要猜的数字: 猜
已经猜的次数: 0
结果:

代码实现

```
<button type="button" id="reset">重新开始一局游戏</button>
<br>

请输入要猜的数字：<input type="text" id="number">
<button type="button" id="button">猜</button>
<br>
已经猜的次数：<span id="count">0</span>
<br>
结果：<span id="result"></span>
<script>
    var inputE = document.querySelector("#number");
```

```
    var countE = document.querySelector("#count");
    var resultE = document.querySelector("#result");
    var btn = document.querySelector("#button");
    var resetBtn = document.querySelector("#reset");
    // 随机生成一个 1-100 的数字
    var guessNumber = Math.floor(Math.random() * 100) + 1// 0 - 1 之间的数
    var count = 0;
    // on: 当
    // click: 点击
    // 事件驱动（Event-Drive）：只要真正发生了点击事件时，才执行该函数
    btn.onclick = function() {
        count++;
        countE.innerText = count;

        var userGuess = parseInt(inputE.value);
        if (userGuess == guessNumber) {
            resultE.innerText = "猜对了";
            resultE.style = "color: gray;";
        } else if (userGuess < guessNumber) {
            resultE.innerText = "猜小了";
            resultE.style = "color: blue;";
        } else {
            resultE.innerText = "猜大了";
            resultE.style = "color: red;";
        }
    };

    resetBtn.onclick = function() {
        guessNumber = Math.floor(Math.random() * 100) + 1
        count = 0;
        countE.innerText = count;
        resultE.innerText = "";
        inputE.value = "";
    }
</script>
```

代码案例: 表白墙

预期效果

创建页面布局

创建 `表白墙.html`

```
<h1>表白墙</h1>
<p>输入后点击提交，会将信息显示在表格中</p>
<span>谁: </span>
<input type="text">
<span>对谁: </span>
<input type="text">
<span>说什么: </span>
<input type="text">
<input type="button" value="提交">
```

此时效果形如

调整样式

```
<div class="container">
    <h1>表白墙</h1>
    <p>输入后点击提交，会将信息显示在表格中</p>
    <div class="row">
        <span>谁: </span>
        <input class="edit" type="text">
    </div>
    <div class="row">
        <span>对谁: </span>
```

```
        <input class="edit" type="text">
    </div>
    <div class="row">
        <span>说什么: </span>
        <input class="edit" type="text">
    </div>
    <div class="row">
        <input type="button" value="提交" class="submit">
    </div>
</div>

<style>
    * {
        margin: 0;
        padding: 0;
    }

    .container {
        width: 400px;
        margin: 0 auto;
    }

    h1 {
        text-align: center;
        padding: 20px 0;
    }

    p {
        color: #666;
        text-align: center;
        font-size: 14px;
        padding: 10px 0;
    }

    .row {
        height: 40px;
        display: flex;
        justify-content: center;
        align-items: center;
    }

    span {
        width: 100px;
        line-height: 40px;
    }

    .edit {
        width: 200px;
        height: 30px;
    }

    .submit {
        width: 304px;
        height: 40px;
        color: white;
        background-color: orange;
        border: none;
    }
```

```
</style>
```

此时效果形如

实现提交

```
// 给点击按钮注册点击事件
var submit = document.querySelector('.submit');
submit.onclick = function () {
    // 1. 获取到编辑框内容
    var edits = document.querySelectorAll('.edit');
    var from = edits[0].value;
    var to = edits[1].value;
    var message = edits[2].value;
    console.log(from + "," + to + "," + message);
    if (from == '' || to == '' || message == '') {
        return;
    }
    // 2. 构造 html 元素
    var row = document.createElement('div');
    row.className = 'row';
    row.innerHTML = from + '对' + to + '说: ' + message;
    // 3. 把构造好的元素添加进去
    var container = document.querySelector('.container');
    container.appendChild(row);
    // 4. 同时清理之前输入框的内容
    for (var i = 0; i < 3; i++) {
        edits[i].value = '';
    }
}
```

此时效果形如:

实现点击按钮的动画效果(选学)

需要使用伪类选择器.

> 伪类选择器种类有很多. 此处的 :active 表示选中被按下的按钮.

```
.submit:active {
    background-color: #666;
}
```

代码案例: 待办事项

预期效果

创建页面布局

创建 `待办事项.html`

```
<div class="nav">
    <input type="text">
    <button>新建任务</button>
</div>
<div class="container">
    <div class="todo">
        <h3>未完成</h3>
        <div class="row">
            <input type="checkbox">
            <span>吃饭</span>
            <button>删除</button>
        </div>
    </div>
    <div class="done">
        <h3>已完成</h3>
    </div>
</div>
```

实现页面样式

```
<style>
    * {
        margin: 0;
        padding: 0;
        box-sizing: border-box;
    }

    .container {
        width: 800px;
        margin: 0 auto;
        display: flex;
    }

    .todo,
    .done {
        width: 50%;
        height: 100%;
    }

    .container h3 {
        height: 50px;

        text-align: center;
        line-height: 50px;

        background-color: #333;
        color: #fff;
    }

    .nav {
        width: 800px;
        height: 100px;
```

```
        margin: 0 auto;
        display: flex;
        align-items: center;
    }

    .nav input {
        width: 600px;
        height: 50px;
    }

    .nav button {
        width: 200px;
        height: 50px;

        border: none;
        background-color: orange;
        color: #fff;
    }

    .row {
        height: 50px;
        display: flex;
        align-items: center;
    }

    .row input {
        margin: 0 10px;
    }

    .row span {
        width: 300px;
    }

    .row button {
        width: 50px;
        height: 40px;
    }
</style>
```

实现页面行为

实现新增

```
// 实现新增任务
var addTaskButton = document.querySelector('.nav button');
addTaskButton.onclick = function () {
    // 1. 获取到任务内容的输入框
    var input = document.querySelector('.nav input');
    // 2. 获取到输入框内容
    var taskContent = input.value;
    // 3. 根据内容新建一个元素节点
    var row = document.createElement('div');
    row.className = 'row';
    var checkbox = document.createElement('input');
    checkbox.type = 'checkbox';
```

```
    var span = document.createElement('span');
    span.innerHTML = taskContent;
    var button = document.createElement('button');
    button.innerHTML = '删除';

    row.appendChild(checkbox);
    row.appendChild(span);
    row.appendChild(button);
    // 4. 把新节点插入到 todo 中
    var todo = document.querySelector('.todo');
    todo.appendChild(row);
}
```

点击复选框后将元素放到 "已完成"

注意:

- 在事件回调函数中使用 this 能够获取到当前处理事件的元素.
- 通过 this.parentNode 属性能够获取到当前元素的父元素.
- 点击 checkbox 时, 会先修改 value , 再触发点击事件.

```
// 修改 addTaskButton.onclick
addTaskButton.onclick = function () {
    // 上方的部分不变...

    // 5. 给 checkbox 注册点击事件
    checkbox.onclick = function () {
        //
        var row = this.parentNode;
        // 注意! 是先触发 checked 为 true, 然后再调用 onclick 函数
        if (this.checked) {
            var target = document.querySelector('.done');
        } else {
            var target = document.querySelector('.todo');
        }
        target.appendChild(row);
    }
}
```

点击删除按钮删除该任务

```
// 修改 addTaskButton.onclick
addTaskButton.onclick = function () {
    // 上方的部分不变...

    // 6. 给删除按钮注册点击事件
    button.onclick = function () {
        var row = this.parentNode;
        var grandParent = row.parentNode;
        grandParent.removeChild(row);
    }
}
```