axios-http

Contents

1 ajax-fetch-axios

Ajax、*Fetch* 和 *Axios* 都是用于在 Web 应用程序中进行异步数据请求和处理的技术。虽然它们都可以实现类似的功能，但它们之间也有很多区别。以下是这些技术的一些主要区别：

- Ajax 是一个基于原生 XMLHttpRequest 对象的技术，用于发起 HTTP 请求并处理响应。它需要手动编写较多的代码来设置请求和处理响应，并且通常需要依赖回调函数来处理异步操作。

- Fetch 是一个基于 Promise 的 API，用于通过 Fetch API 发起 HTTP 请求并处理响应。相比于原生的 Ajax 技术，Fetch API 更简洁、更易于使用，并且支持流式响应、跨站资源共享（CORS）等特性。
- Axios 是一个开源的基于 Promise 的 HTTP 客户端库，可用于发送 HTTP 请求并处理响应。与 Fetch 不同，Axios 具有内置的请求取消、错误处理、拦截器、基于浏览器和 Node.js 环境的支持等特性。同时，Axios 也提供了对 Promise 和 async/await 的支持。

综上所述，**Ajax** 以及 **Fetch** 是 **JavaScript** 原生提供的异步请求技术，而 **Axios** 则是一个由第三方库封装的异步请求工具。在选择使用哪种技术时，需要根据实际需求和开发经验进行权衡和选择。

以下是使用 Ajax 发送 GET 请求并处理响应的示例代码：

```
// 创建一个新的 XMLHttpRequest 对象
const xhr = new XMLHttpRequest();

// 设置请求 URL 和方法
xhr.open('GET', 'path/to/data.json');

// 处理响应数据
xhr.onload = function() {
  if (xhr.status === 200) {
    const data = JSON.parse(xhr.responseText);
    console.log(data);
  } else {
    console.error('Request failed. Status code:', xhr.status);
  }
};

// 发送请求
xhr.send();
```

在上面的示例中，我们首先创建了一个新的 XMLHttpRequest 对象，并通过 `.open()` 方法设置请求的 URL 和方法。然后，我们使用 `.onload` 事件处理程序处理响应数据，并将其解析为 JSON 格式。最后，使用 `.send()` 方法发送请求。

以下是使用 Fetch 发送 GET 请求并处理响应的示例代码：

```
fetch('path/to/data.json')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Request failed:', error));
```

在上面的示例中，我们使用 `fetch()` 函数发送 GET 请求，并在 Promise 链中处理响应数据。如果请求成功，则将响应解析为 JSON 格式并打印到控制台上；如果请求失败，则捕获错误并打印到控制台上。

以下是使用 Axios 发送 GET 请求并处理响应的示例代码：

```
axios.get('path/to/data.json')
  .then(response => console.log(response.data))
  .catch(error => console.error('Request failed:', error));
```

在上面的示例中，我们使用 axios.get() 方法发送 GET 请求，并在 Promise 链中处理响应数据。如果请求成功，则将响应解析为 JSON 格式并打印到控制台上；如果请求失败，则捕获错误并打印到控制台上。

需要注意的是，以上示例仅演示了如何使用 Ajax、Fetch 和 Axios 发送 GET 请求并处理响应。实际开发中，您可能需要使用其他 HTTP 方法（例如 POST、PUT 或 DELETE）来与服务器进行交互，并针对具体业务需求进行相关设置和处理。

2 append-appendchild

在 JavaScript 中，append 和 appendChild 是用于向 DOM（文档对象模型）中添加新元素或节点的两个常用方法，它们有一些区别和不同的用途。

1. appendChild 方法：
   - appendChild 是 Node 对象的方法，用于将一个节点对象作为子节点插入到目标节点的最后一个子节点位置。
   - 使用 appendChild 方法时，需要创建一个节点对象，然后将其添加到目标节点中。
   - 示例代码：javascript const parent = document.getElementById('parent'); // 获取父节点 const child = document.createElement('div'); // 创建一个子节点（div） parent.appendChild(child); // 将子节点添加到父节点中
2. append 方法：
   - append 是 Element 对象的方法，用于在目标元素的末尾插入一组指定的节点对象或 DOM 字符串。
   - 使用 append 方法时，可以直接传入一个或多个节点对象、DOM 字符串或其他可以转换为节点的对象。
   - 示例代码：javascript const parent = document.getElementById('parent'); // 获取父节点 const child1 = document.createElement('div'); // 创建子节点 1（div） const child2 = document.createElement('span'); // 创建子节点 2（span） parent.append(child1, child2); // 将多个子节点添加到父节点中

对比：

- appendChild 只能将一个节点对象作为子节点添加到目标节点中，需要先创建节点对象后再使用。
- append 可以将一个或多个节点对象或 DOM 字符串添加到目标元素中，更加灵活方便，且支持链式操作。
- 注意：append 是较新的方法，可能在一些旧版本的浏览器中不被支持，而 appendChild 是较旧的方法，具有更广泛的兼容性。

根据具体的需求和浏览器兼容性要求，您可以选择适合的方法来向 DOM 中添加新的内容。

3 apply-call-bind

call、apply 和 bind 是 JavaScript 中的三个方法，它们都是用于改变函数的 this 指向。

call 和 apply 的作用一样，它们都可以在指定的作用域中调用函数，并将 this 指向第一个参数。区别在于，call 的参数是逐个传入的，而 apply 的参数是以数组的形式传入的。

bind 的作用是创建一个新的函数，新函数的 this 指向第一个参数，后续的参数作为新函数的参数传入。与 call 和 apply 不同的是，bind 方法不会立即执行函数，而是返回一个新函数，需要手动调用才会执行。

下面是这三个方法的用法示例：

```
const obj = {
  name: 'Alice',
  age: 18,
};

function sayHi() {
  console.log(`Hi, my name is ${this.name}, I'm ${this.age} years old.`);
}

// 使用 call 方法调用函数
sayHi.call(obj);

// 使用 apply 方法调用函数
sayHi.apply(obj);

// 使用 bind 方法创建新函数并调用
const newSayHi = sayHi.bind(obj);
newSayHi();
```

在实际开发中，这三个方法经常用于改变函数的 this 指向，例如在回调函数中使用，或者在函数式编程中使用柯里化等技术。

```
const demo = {
  version: 1,
  log: function () {
    console.log(this.version);
  },
};
```

```
const { log } = demo;
// log.apply(demo, []);
// log.bind(demo, null);
const bunLog = log.bind(demo);
bunLog();
```

在这个例子中，demo 对象的 log 方法被定义为一个箭头函数。箭头函数的一个重要特点是它们继承了定义时的上下文，而不是在运行时绑定到一个新的上下文。因此，在这个例子中，箭头函数的 this 值将继承自它的父级上下文，即全局上下文。

在这个例子中，当 log 方法被解构并作为独立函数调用时，它的 this 值将指向全局上下文，因为它的父级上下文是全局上下文。在浏览器环境中，全局上下文通常是 window 对象。由于 window 对象没有名为 version 的属性，因此 console.log(this.version) 表达式将输出 undefined。

因此，箭头函数的 this 值与普通函数的 this 值有很大的区别。普通函数的 this 值在运行时根据调用方式和上下文动态绑定，而箭头函数的 this 值在定义时就已经确定，无法通过 call()、apply() 或 bind() 等方法进行修改。

```
const version = 1;
const demo = {
  log: () => {
    console.log(this.version);
  },
};

const { log } = demo;
log();
```

如果想使箭头函数的 this 值指向 demo 对象,可以将 log 方法改为普通函数,或者使用 Function.prototype.bind() 方法将 log 方法绑定到 demo 对象上，如下所示：

```
const demo = {
  version: 1,
  log: function() {
    console.log(this.version);
  },
};

const { log } = demo;
log.bind(demo)(); // 输出: 1
```

在这个例子中，log 方法被改为普通函数，它的 this 值将根据调用方式和上下文动态绑定。接下来，使用 bind() 方法将 log 方法绑定到 demo 对象上，并立即调用返回的新函数。这样，log 方法就会在 demo 对象的上下文中被调用，从而正确地输出 version 属性。

4 array-from

```
// 虚构数组
const data = Array.from({ length: 10 }, (_, i) => i + 1);

const d1 = [1, 2, 3];
const d2 = [
  {
    id: 1,
    name: 'John',
  },
  {
    id: 2,
    name: 'Jane',
  },
];
const data1 = Array.from(d1, (arr) => arr * 2);
const data2 = Array.from(d2, (obj) => obj.name);
console.log(data, data1, data2);

// data:  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
// data1: [2, 4, 6]
// data2: [('John', 'Jane')]
```

Array.from() 是一个用于将类数组对象或可迭代对象（如字符串、Set、Map 等）转换为数组的方法。以下是如何使用 Array.from() 进行转换的示例：

```
// 将字符串转换为数组
const str = "Hello, World!";
const arr = Array.from(str);
console.log(arr); // 输出: ['H', 'e', 'l', 'l', 'o', ',', ' ', 'W', 'o', 'r', 'l', 'd', '!']

// 将 Set 转换为数组
const mySet = new Set([1, 2, 3, 4, 5]);
const arrFromSet = Array.from(mySet);
console.log(arrFromSet); // 输出: [1, 2, 3, 4, 5]

// 将 Map 转换为数组
const myMap = new Map([['a', 1], ['b', 2], ['c', 3]]);
const arrFromMap = Array.from(myMap);
console.log(arrFromMap); // 输出: [['a', 1], ['b', 2], ['c', 3]]
```

你可以看到，Array.from() 能够将不同类型的可迭代对象转换为数组，并且你还可以提供一个可选的映射函数，用于对数组的每个元素进行转换。例如：

```
// 使用映射函数将数字加倍
const numbers = [1, 2, 3, 4, 5];
const doubled = Array.from(numbers, x => x * 2);
console.log(doubled); // 输出: [2, 4, 6, 8, 10]
```

这种方式可以在将类数组对象或其他可迭代对象转换为数组时非常方便。

使用 Array.from() 方法可以很方便地虚构一个数组。你可以通过传递一个可迭代对象或类数组对象来创建一个新的数组实例。下面是一个使用 Array.from() 方法虚构数组的示例：

```
// 虚构一个包含 1 到 5 的数组
const arr = Array.from({ length: 5 }, (_, index) => index + 1);
console.log(arr); // 输出 [1, 2, 3, 4, 5]

// 虚构一个包含 A 到 E 的数组
const alphabet = Array.from({ length: 5 }, (_, index) => String.fromCharCode(65 + index));
console.log(alphabet); // 输出 ["A", "B", "C", "D", "E"]

// 虚构一个包含随机数的数组
const randomArray = Array.from({ length: 5 }, () => Math.random());
console.log(randomArray); // 输出包含 5 个随机数的数组
```

在上面的示例中，我们使用了 Array.from() 方法来创建了不同类型的数组。通过传递一个具有 length 属性的对象，我们可以指定数组的长度。然后，我们可以使用第二个参数 mapFn 来对数组中的每个元素进行处理，从而虚构出我们想要的数组。

希望这个示例能够帮助你理解如何使用 Array.from() 方法来虚构数组。如果你有任何其他问题，请随时提问！

5 array-prototype-last

```
/**
 * link: https://leetcode.cn/problems/array-prototype-last/
 * Order: 2619
 * Type: Javascript
 * Description:
 *  Write code that enhances all arrays such that you can call the array.last() method on any
 *  请你编写一段代码实现一个数组方法，使任何数组都可以调用 array.last() 方法，这个方法将返回数组最后一个元素
 */

Array.prototype.last = function () {
    if (this.length === 0) { return -1; }
```

```
    else {
        return this[this.length - 1];
    }
};

/**
 * const arr = [1, 2, 3];
 * arr.last(); // 3
 */
```

6 arrow-function

箭头函数（Arrow Function）是 ES6 提出的一种新的函数语法，它可以更简洁地定义函数，并且不会改变 this 的指向。

箭头函数的参数有以下几种形式：

1. 无参数：如果函数没有参数，那么可以在括号中省略参数部分，如下所示：

```
const sayHello = () => {
  console.log('Hello');
};

sayHello(); // 输出 "Hello"
```

1. 单个参数：如果函数只有一个参数，则可以省略参数括号。例如：

```
const double = x => {
  return x * 2;
};

console.log(double(5)); // 输出 10
```

1. 多个参数：如果函数有多个参数，则需要使用圆括号将这些参数括起来。例如：

```
const add = (x, y) => {
  return x + y;
};

console.log(add(2, 3)); // 输出 5
```

注意事项：

- 如果只有一个参数，也可以选择加上括号，如 (x) 或者 (y)，但是不允许只有右侧的括号，如 x)。
- 如果没有参数或者有多个参数，就必须使用圆括号。
- 如果函数体只有一条语句，可以省略花括号和 return 语句，例如：

```
const add = (x, y) => x + y;

console.log(add(2, 3)); // 输出 5
```

- 如果函数体中包含多条语句，就必须使用花括号和 return 语句。

7 arrow-function-brackets

在箭头函数中使用大括号 {} 的有无会影响函数的行为。

如果箭头函数体内只有一条语句，你可以选择省略大括号 {}。这种情况下，该语句的结果将自动成为箭头函数的返回值。例如：

```
const add = (a, b) => a + b;
console.log(add(1, 2)); // 输出: 3
```

上述示例中，箭头函数 add 的函数体只有一条语句 a + b，因此可以省略大括号 {}。这样，函数会直接返回表达式 a + b 的结果。在调用 add(1, 2) 时，输出结果为 3。

但是，如果箭头函数体内有多个语句，则需要使用大括号 {} 将这些语句包裹起来，并使用 return 关键字显式地返回一个值。例如：

```
const multiply = (a, b) => {
  const result = a * b;
  return result;
};
console.log(multiply(2, 3)); // 输出: 6
```

上述示例中，箭头函数 multiply 的函数体有两条语句：定义变量 result 和返回 result。这时就需要使用大括号 {} 将这两条语句包裹起来，并使用 return 关键字明确地返回 result 的值。在调用 multiply(2, 3) 时，输出结果为 6。

因此，是否在箭头函数体内使用大括号 {} 取决于函数体中的语句数量和你是否需要显式地返回一个值。

8 axios-http

是的，axios 可以代替 fetch 进行网络请求。虽然 fetch 在现代浏览器中已经成为了原生 API，但是它的兼容性还不够完善，而且只能基于 Promise 来处理请求结果，使用起来相对比较麻烦。

axios 是一个基于 Promise 的 HTTP 客户端库，提供了丰富的 API 和配置选项，可以使得请求的配置、管理和拦截变得更加方便和灵活。axios 支持多种类型的请求（如 GET、POST、PUT、DELETE 等），支持自定义头部、请求参数、请求体等配置，同时也支持请求的取消、错误处理和超时设置等特性。

以下是使用 axios 发送 GET 请求的示例代码：

```
import axios from 'axios';

axios.get('https://api.github.com/users/octocat')
  .then(response => console.log(response.data))
  .catch(error => console.error(error));
```

上面的代码通过 axios 发送了一个 GET 请求，并在请求成功后打印出响应数据。注意，在 axios 中，响应数据被封装在 response 对象的 data 属性中，我们需要通过 response.data 来获取响应数据。

由于 axios 具有更加丰富的功能和更好的可扩展性，因此在实际开发中，我们通常会优先选择 axios 来进行网络请求。