DAY27 promise-async-await详解

2023年5月24日 8:47

promise概述

一.promise概述

ES6新增的，异步编程的一种解决方案，比传统的解决方案—回调函数和事件—更合理和更强大。

1.承诺的意思，是一个专门用来解决异步回调地狱的问题

2.promise语法

2.1.利用语法生成promise对象

2.2.promise对象采用一个函数做参数，做参数的函数又有两个参数(resolve，reject)

```
const promise = new Promise((resolve, reject) => {//同步代码

});
```

3.promise的状态

Promise代表一个异步操作，有三种状态：pending进行中、fulfilled(resolve)成功、rejected失败。

状态整合成两种可能

pengding进行中 - resolve成功(解决)

pengding进行中 - reject失败(未解决)

对象的状态不受外界影响，一旦状态设定，就不会再变，承诺

二.promise对象下面的实例方法(原型方法)

1.then的作用是为 Promise 实例添加状态改变时的回调函数。

then方法的第一个参数是resolve状态的回调函数，第二个参数是reject状态的回调函数。

```
const promise = new Promise((resolve, reject) => {//同步代码
  resolve();//成功的状态(这里的参数也是函数，而且可以传递参数)，找promise下面的then，执行then的第一个参数，
  reject();//失败的状态(这里的参数也是函数，而且可以传递参数)，找promise下面的then，执行then的第二个参数，
});
promise.then(() => {//成功的回调
  console.log('成功的回调');
}, () => {//失败的回调
  console.log('失败的回调');
});
```

状态传参

```
const promise = new Promise((resolve, reject) => {//同步代码
  // resolve('我是resolve状态的传参');//这里的参数传递给then的第一个函数参数
  // reject('我失败了或者错了')
});
promise.then((res) => {//res:resolve状态的传参
  console.log(res);
}, (err) => {//err:reject状态的传参
  console.log(err);
});
```

then可以链式调用

```
const promise = new Promise((resolve, reject) => {//同步代码
  resolve();
});
promise
  .then(() => {//如果只有一个函数做参数，说明是成功的回调
    console.log(1);
  })
  .then(() => {//这里的then可以链式调用，因为上面的then默认返回promise对象
    console.log(2);
  })
  .then(() => {//这里的then可以链式调用，因为上面的then默认返回promise对象
    console.log(3);
  })
```

2.catch方法，失败的回调，如果状态变为rejected，就会调用catch()方法指定的回调函数，处理这个错误。

```
const promise = new Promise((resolve, reject) => {//同步代码
  reject('我失败了或者错了')
});
promise.catch((err) => {
  console.log(err);
});
```

```
const promise = new Promise((resolve, reject) => {//同步代码
  reject('数据获取失败')
});
promise
  .then((res) => {
    console.log(res, res);
  })
  .catch((err) => {
    console.log('err', err);
  });
```

3.finally()方法用于指定不管 Promise 对象最后状态如何，都会执行的操作。该方法是ES2018 引入标准的。

```
const promise = new Promise((resolve, reject) => {//同步代码
  reject('数据获取失败')
});
promise
  .then((res) => {
    console.log('res', res);
  })
  .catch((err) => {
    console.log('err', err);
  })
  .finally(() => {//目的就是用来处理收尾的工作
    console.log('不管 Promise 对象最后状态如何，都会执行的操作');
  })
```

利用promise封装ajax

```
const querystring = function (obj) {
  if (Object.prototype.toString.call(obj) === "[object Object]") {
    //出入的参数必须是对象格式
    const arr = [];
    for (let key in obj) {
      arr.push(key + "=" + obj[key]);
    }
    return arr.join("&");
  }
  return obj;
};
const ajax_promise = function (options) {
  //options:配置参数
  // 1.默认参数
  const settings = {
    type: "GET",
    async: true,
  };
  // 2.配置参数和默认参数的合并
  options = Object.assign({}, settings, options); //如果配置了参数使用配置参数，否则使用默认参数
  // 3.配置限制条件
  // 限制1：仅支持GET POST
  if (!/^(get|post)$/i.test(options.type)) {
    throw new Error(
      "目前封装的函数仅支持GET POST请求,其他请求方式敬请期待~~~~~"
    );
  }
```

```
  // 限制2：url必须填写
  if (!options.url || /\s+/.test(options.url)) {
    throw new Error("接口地址必须填写,不能包含空格~~~~~");
  }
  // 限制3：是否异步必须是布尔值
  if (!(typeof options.async === "boolean")) {
    throw new Error("是否异步的值必须是布尔值~~~~~");
  }
  // 限制4：headers的值必须是对象格式
  // 保证options.headers存在
  // 存在的基础上判断是对象格式
  if (options.headers && options.headers.constructor !== Object) {
    throw new Error("headers的值必须是对象Object格式~~~~~");
  }
  // 数据存在且是请求方式是GET
  if (options.data && /^get$/i.test(options.type)) {
    options.url += "?" + querystring(options.data);
  }
  // 4.封装的和ajax相关的核心代码
  let promise = new Promise((resolve, reject) => {
    //创建promise对象
    let xhr = new XMLHttpRequest();
    xhr.open(options.type, options.url, options.async);
    xhr.onload = function () {
      if (/^2\d{2}$/.test(xhr.status)) {
        //成功的
        //请求成功
        // 配置获取的接口数据是否进行对象的转换
        // 如果dataType存在，并且值为json，利用JSON.parse进行转换
        let resData;
        if (options.dataType === "json") {
          resData = JSON.parse(xhr.responseText);
        } else {
          resData = xhr.responseText;
        }
        resolve(resData); //成功的回调，将数据返回出来，利用then方法接收
      } else {
        //失败的
        reject("接口地址有误，请检查~~~~~"); //失败的回调，将失败的信息返回出来，利
用catch方法接收
      }
    };
    // 数据存在且请求方式是post
    // 约定传输数据的格式
    if (options.headers && options.headers["content-type"]) {
```

```
        xhr.setRequestHeader("content-type", options.headers["content-type"]);
      }
      // 免登录的令牌
      if (options.headers && options.headers["authorization"]) {
        xhr.setRequestHeader("authorization", options.headers["authorization"]);
      }
      options.data && /^post$/i.test(options.type)
        ? xhr.send(querystring(options.data))
        : xhr.send();
    });
    // 重要的一步
    return promise;
  };
```

利用promise解决回调地狱

```
<body>
  <p id="first"></p>
  <hr>
  <p id="second"></p>
  <hr>
  <p id="third"></p>
  <hr>
  <p id="fourth"></p>
</body>

<script src="./ajax.js"></script>
<script>
  // 案例：按照顺序分别获取四个接口的数据
  // 如果第一个接口出错，后面的接口都无法显示，依此类推
  // http://localhost:8888/test/first
  // http://localhost:8888/test/second
  // http://localhost:8888/test/third
  // http://localhost:8888/test/fourth
  ajax_promise({
    type: 'GET',
    url: 'http://localhost:8888/test/first',
  })
    .then(res => {
      first.innerHTML = res;
      return ajax_promise({
        type: 'GET',
        url: 'http://localhost:8888/test/second',
        datType: 'json'
      })
    })
    .then(res => {//因为上面的then返回的promise实例使用的
      second.innerHTML = res;
      return ajax_promise({
        type: 'GET',
        url: 'http://localhost:8888/test/third',
        datType: 'json',
```

```
        data: {
          name: 'zhangsan',
          age: 18
        },
      })
    })
    .then(res => {
      third.innerHTML = res;
      return ajax_promise({
        type: 'POST',
        url: 'http://localhost:8888/test/fourth',
        datType: 'json',
        data: {
          name: 'wangwu',
          age: 180
        },
        headers: {
          'content-type': 'application/x-www-form-urlencoded'
        },
      })
    })
    .then(res => {
      fourth.innerHTML = res;
    })
</script>
```

promise下面的静态方法

1.Promise.all - 重点

用于将多个 Promise 实例，包装成一个新的 Promise 实例，接受一个数组作为参数，只有数组里面的每个状态都变成resolve，则新的 Promise 实例状态才会变成resolve.

```
let p1 = new Promise((resolve, reject) => {
  resolve(1);
})
let p2 = new Promise((resolve, reject) => {
  resolve(2);
})
let p3 = new Promise((resolve, reject) => {
  resolve(3);
})
let p4 = new Promise((resolve, reject) => {
  resolve(4);
})
let newPromise = Promise.all([p1, p2, p3, p4]);
newPromise.then((res) => {// 每个状态都变成resolve
  console.log(res);//[1, 2, 3, 4]
}).catch(() => {// 有一个状态是reject
  console.log('有状态是失败的');
})
```

```
// 同时获取多个接口的数据
// let newPromise = Promise.all([
//   ajax_promise({
//     type: 'GET',
//     url: 'http://localhost:8888/test/first',
//   }),
//   ajax_promise({
//     type: 'GET',
//     url: 'http://localhost:8888/test/second',
//     datType: 'json'
//   }),
//   ajax_promise({
//     type: 'GET',
//     url: 'http://localhost:8888/test/third',
//     datType: 'json',
//     data: {
//       name: 'zhangsan',
//       age: 18
//     },
//   }),
//   ajax_promise({
//     type: 'POST',
//     url: 'http://localhost:8888/test/fourth',
//     datType: 'json',
//     data: {
//       name: 'wangwu',
//       age: 180
//     },
//     headers: {
//       'content-type': 'application/x-www-form-urlencoded'
//     },
//   })
// ]);

// newPromise
//   .then(res => {
//     console.log(res);
//   })
//   .catch(() => {// 有一个状态是reject
//   console.log('有状态是失败的');
// })
```

2. Promise.race

方法同样是将多个 Promise 实例，包装成一个新的 Promise 实例，接受一个数组作为参数，只要其中有一个实例率先改变状态，则整个的状态就跟着改变。

那个实例状态改变的快，跟着这个状态。

```
let p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('成功1');
  }, 1500)
```

```
})
let p2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject('失败2')
  }, 500)
})
let p3 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('成功3');
  }, 200)
})
let p4 = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject('失败4');
  }, 1000)
})
let newpro = Promise.race([p1, p2, p3, p4]);
newpro.then(res => console.log(res)).catch(err => console.log(err))
```

3.Promise.resolve - 重点方法

有时需要将现有对象转为 Promise 对象，该方法就起到这个作用

将现有的任何对象转换成成功的状态

```
// 测试promise异步操作
// console.log(1);
// let p1 = new Promise((resolve, reject) => {//同步的
//   console.log(2);
//   resolve(4);
// });
// p1.then(res => {//异步
//   console.log(3);
//   console.log(res);
// }).catch(() => {//异步
//   console.log(4);
// });
// console.log(5);
// 结果: 1,2,5,3,4

let num = 10;
Promise.resolve(num).then(res => console.log(num))//异步
console.log(1);//同步
//结果: 1, 10
```

4.Promise.reject

方法也会返回一个新的 Promise 实例，该实例的状态为rejected。

```
let num = 10;
Promise.reject(num).catch(res => console.log(num))//异步
console.log(1);//同步
//结果: 1, 10
```

5.Promise.allSettled - 重点方法

方法接受一个数组作为参数，数组的每个成员都是一个promise对象，并返回一个新的promise对象，只有等到参数数组的所有promise对象都发生改变(不管成功还是失败)，返回的promise对象才会发生状态改变

```
let p1 = new Promise((resolve, reject) => {
  resolve(1);
})
let p2 = new Promise((resolve, reject) => {
  reject(2);
})
let p3 = new Promise((resolve, reject) => {
  reject(3);
})
let p4 = new Promise((resolve, reject) => {
  resolve(4);
})
Promise.all([p1, p2, p3, p4]).then(res => console.log(res)).catch(err =>
console.log(err)); //2
Promise.allSettled([p1, p2, p3, p4]).then(res => console.log(res)).catch(err =>
console.log(err));
/*
  [
    {status: 'fulfilled', value: 1},
    {status: 'rejected', reason: 2},
    {status: 'rejected', reason: 3},
    {status: 'fulfilled', value: 4},
  ]
*/
```

promise的应用

1.什么是sleep函数

sleep函数的作用是让线程休眠，等到指定的时间再重新唤起，起到延迟的作用。

异步 + 定时器

注意：js中没有sleep函数，需要自己封装

```
function sleep(time) {
  let p1 = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve()
    }, time)
  });
  return p1;
```

```
}
sleep(3000).then(() => {
  console.log('休眠完成后要做的事件');
});
```

优化代码结构

```
// function sleep(time) {
//   return new Promise(resolve => {
//     setTimeout(() => {
//       resolve()
//     }, time)
//   });;
// }
// sleep(3000).then(() => {
//   console.log('休眠完成后要做的事件');
// });

function sleep(time) {
  return new Promise(resolve => setTimeout(resolve, time));
}
sleep(3000).then(() => {
  console.log('休眠完成后要做的事件');
});
```

2.红灯3秒亮一次，绿灯2秒亮一次，黄灯1秒亮一次；如何让三个灯不断交替重复亮灯？（用Promise实现）

```
function sleep(time) {
  return new Promise(resolve => setTimeout(resolve, time));
}
function red() {
  sleep(3000).then(() => {
    console.log('红灯亮');
    green();
  })
}
function green() {
  sleep(2000).then(() => {
    console.log('绿灯亮');
    yellow();
  })
}
function yellow() {
  sleep(1000).then(() => {
    console.log('黄灯亮');
    red();
  })
}
red()
```

3.根据提示代码中的内容，写出这些代码按次序打印出的结果，并在每个结果后面从promise特性的角度阐述为什么会打印出这样的结果

如果参数是 Promise 实例，那么Promise.resolve将不做任何修改、原封不动地返回这个实例。

```
var p1 = Promise.resolve(1);//将数字1转换成promise对象
var p2 = Promise.resolve(p91);//如果参数是 Promise 实例，那么Promise.resolve将不做任何修改、原封不动地返回这个实例。
var p3 = new Promise(function (resolve, reject) {//利用Promise对类生成新的实例对象
  resolve(1);
});
var p4 = new Promise(function (resolve, reject) {
  resolve(p1);
});
console.log(p1 === p2);//true
console.log(p1 === p3);//false    p3是新的实例对象(Promise)    p1对象是利用Promise.resolve生成的
p4.then((value) => {//这里的p4一直最后触发，因为p4实例化的promise对象里面存在promise对象，Promise.resolve是异步的，意味这存在两层异步，先拆开一层，进行下一步输出
  console.log('p4=' + value);
});
p1.then((value) => {
  console.log('p1=' + value);
});
p2.then((value) => {
  console.log('p2=' + value);
});
```

实例方法链式调用

链式调用原理：

this无论在constructor还是方法中表示实例对象

当前的方法中返回this即可。

```
class Person {
  constructor() {
    this.name = 'zhangsan';
  }
  showa() {
    console.log('我是a方法');
    return this;
  }
  showb() {
    console.log('我是b方法');
    return this;
  }
  showc() {
```

```
        console.log('我是c方法');
      }
    }

    let p1 = new Person();
    p1.showa().showb().showc();
```

题目：使用原型或class的方式来实现js的链式调用，对数字进行加减乘除

```
    class Calc {
      constructor(num) {
        this.num = num;
      }
      add(v) {//加
        this.num += v;
        return this;
      }
      reduce(v) {//加
        this.num -= v;
        return this;
      }
      mul(v) {//加
        this.num *= v;
        return this;
      }
      div(v) {//加
        this.num /= v;
        return this.num;
      }
    }
    let c1 = new Calc(9);
    console.log(c1.add(1).reduce(2).mul(3).div(4))  //6
```

try/catch/finally

1.try/catch/finally 语句用于处理代码中可能出现的错误信息

```
console.log(a);//出错,阻止后续代码执行
console.log('这里还有1000行代码执行');
```

2.如果try里面的代码有错误，那么就执行catch里面的代码，同时catch的参数会返回try里面的错误信息。

语法格式

```
    try {
      console.log(a);
    } catch (e) {//e:捕获到try里面的错误信息
```

```
    console.log('这里还有1000行代码执行');
    console.log(e);
  }
```

注意：try...catch...同时存在

trycatch可以实现兼容处理。

```
  try {
    document.addEventListener('click', function () {//标准
      alert('事件触发了')
    });
  } catch (e) {
    document.attachEvent('onclick', function () {//非标准
      alert('IE的事件触发了')
    });
  }
```

3.finally 语句在 try 和 catch 之后无论有无异常都会执行

```
  try {
    console.log('try如果有错,执行catch');
  } catch (e) {
    console.log('catch能够捕获try里面的错误');
  } finally {
    console.log('finally无论有无异常都会执行');
  }
```

async和await

async await 的特点及描述

1.async/await 是一个 es7 的语法

2.async 其实就是promise的语法糖。

3.async异步的意思，作用是申明一个异步函数，函数的返回值是promise 对象

4.函数前面必须加一个async,异步操作的方法前加一个await 关键字，注意await 只能在async函数中执行，否则会报错。

5.await就是promise里面then的语法糖， await直接拿到promise的返回值。

6.匿名函数也可以使用async和await，这个语法是回调地狱的终极解决方案。

一.async

1.async/await 是一个 es7 的语法

2.async 其实就是promise的语法糖。

函数前面必须加一个async,异步操作的方法前加一个await 关键字。

顾名思义，就是让你等一下，执行完了再继续往下走

注意：await 只能在async函数中执行，否则会报错。

3.async:异步的意思，作用是申明一个异步函数，函数的返回值是promise 对象

```
async function fn() {//async函数的返回值是promise实例
  return 'hello';
}
console.log(fn().then(res => console.log(res)));//hello
```

4.await:等待

await就是promise里面then的语法糖。

await直接拿到promise的返回值。

匿名函数也可以使用async和await.

这个语法是回调地狱的终极解决方案

5.重要的特点

可以把异步代码写的看起来像同步代码

```
// 匿名函数演示
(async function () {
  console.log(await ajax_promise({
    url: 'http://localhost:8888/test/first',
  }))
})();

// 箭头函数
(async () => {
  console.log(await ajax_promise({
    url: 'http://localhost:8888/test/first',
  }))
})();

// 这里是解决回调地狱最佳的方案。
async function fn() {
  let first = await ajax_promise({
    url: 'http://localhost:8888/test/first',
  });
  let second = await ajax_promise({
    url: 'http://localhost:8888/test/second',
  });
  let third = await ajax_promise({
    type: 'GET',
    url: 'http://localhost:8888/test/third',
    datType: 'json',
    data: {
      name: 'zhangsan',
      age: 18
    },
  })
```

```
  console.log(first);
  console.log(second);
  console.log(third);
}
fn();

// 案例
let p1 = new Promise(resolve => {
  setTimeout(() => {
    resolve('p1实例的返回值,我2000ms后执行')
  }, 2000);
});
let p2 = new Promise(resolve => {
  setTimeout(() => {
    resolve('p2实例的返回值,我1000ms后执行')
  }, 1000);
});
p1.then(res => console.log(res))
p2.then(res => console.log(res))

// 这样的异步代码写的就看起来像一个同步代码了
async function fn() {
  let a1 = await p1;
  let a2 = await p2;
  console.log(a1);
  console.log(a2);
}
fn();

// 案例：async await 没有捕获错误的机制，利用前面学习的try...catch...可以
async function fn() {
  let first;
  try {
    first = await ajax_promise({
      url: '',
    });
  } catch (e) {
    console.log(e);
    first = await ajax_promise({
      url: 'http://localhost:8888/test/first',
    });
  }
  console.log(first);
}
fn();
```

async应用

1.forEach 只支持同步代码, await是异步代码，await不能使用forEach

```
function fetch(x) {
  return new Promise((resolve, reject) => {
```

```
      resolve(x);
    });
  }

  // async function fn() {
  //   const res = await fetch(10);
  //   console.log(res);
  //   console.log('end');
  // }
  // fn();

  let arr = [1, 2, 3];
  // forEach 只支持同步代码，await是异步代码，await不能使用forEach
  function fn() {
    arr.forEach(async function (item) {
      let res = await fetch(item);
      console.log(res);
    });
    console.log('end');
  }
  fn();//end 1 2 3
```

一般来说解决的办法有2种，for…of和for循环。

```
  async function fn() {
    for (let item of arr) {
      let res = await fetch(item);
      console.log(res);
    }
    console.log('end');
  }
  fn();
```

2.Fetch API介绍

Fetch API 提供了一个获取资源的接口，用于取代传统的XMLHttpRequest的，在JavaScript脚本里面发出HTTP请求。

目前还没有被所有浏览器支持

Fetch API是基于promise的设计，返回的是Promise对象，它是为了取代传统xhr的不合理的写法而生的。

```
console.log(fetch('http://localhost:8888/test/second'));//返回的是Promise对象
```

response.json()是一个异步操作，取出所有内容，并将其转为 JSON 对象

```
fetch('http://localhost:8888/test/second')
  .then(response => response.json())//这里返回的依然是promise对象，继续then
  .then(res => console.log(res))//获取数据成功
```

```
async function fn() {
  let response = await fetch('http://localhost:8888/test/second');
  let res = await response.json();
  console.log(res);
}
fn();
```

promise和async await的区别和优缺点

1.都是处理异步请求的方式

2.promise是ES6 async await是es7的语法

3.async await是基于promise实现的，他和promise都是异步的

4.promise是返回对象，要使用then catch方法去处理和捕获异常，并且书写方式是链式的，容易造成代码重叠，维护不好。

async await 是通过try catch进行异常处理

5.async await 最大的优点是代码看起来像同步一样，只要遇到await就会立刻返回结果，然后再执行后面的操作，而promise是通过then的方式返回，会出现请求还没有返回，就执行了后面的同步操作。