DAY26 回调函数和AJAX封装

2023年5月23日 18:04

回调函数的概念

一.回调函数的概念

1.函数做参数(传递给另外一个函数，在另外一个函数内部调用)

2.回调函数也叫高阶函数,英文意思是callback

```
function fn(cb) {
  cb();
}
fn(function () {//回调函数
  console.log('回调函数也叫高阶函数,英文意思是callback');
});

window.setTimeout(function () { }, 1000);
arr.forEach(function(){})

function A(B) {
  console.log('A');
  B();
}
function B() {
  console.log('B');
}
A(B)
```

二.为什么需要回调函数

- 当我们执行一个异步的行为的时候，我们需要在一个异步行为执行完毕之后做一些事情
- 那么，我们是没有办法提前预知这个异步行为是什么时候完成的
- 我们就只能以回调函数的形式来进行

```
function fn1(fn2) {
  setTimeout(function () {
    console.log(1);
    fn2();
  }, 1000);
}
function fn2() {
  console.log('2');//包含很多代码
}
fn1(fn2);
```

三.回调函数的注意事项

因为是函数做参数，必须判断传入的必须是函数类型

```
    function fn1(fn2) {
      setTimeout(function () {
        console.log(1);
        fn2 && typeof fn2 === 'function' && fn2();  //注意
      }, 1000);
    }
    fn1(function () {
      console.log(2);
    });
```

```
// 案例:获取异步里面的数据
// 函数外面利用函数获取函数里面的异步数据
function sendData(cb) {
  let arr;
  setTimeout(function () {//异步
    arr = ['zhangsan', 'lisi', 'wangwu'];
    cb(arr);//arr:实参
  }, 0);
  // return arr;//这里是无法输出数据，同步的
}
sendData(function (res) {
  console.log(res);//['zhangsan', 'lisi', 'wangwu']
});
```

```
// 案例：获取ajax返回的数据
function ajax(success) {
  let resData;
  let xhr = new XMLHttpRequest();
  xhr.open('GET', 'http://localhost:8888/test/second');
  xhr.onload = function () {
    if (xhr.status === 200) {
      resData = xhr.responseText;//接收数据
      success(resData);
    }
  };
  xhr.send();
  // return resData;
}
ajax(function (res) {
  console.log(res);
});
```

函数参数的配置

函数的参数不宜超过四个，否则传参比较麻烦

```
// 例如
// var setUserInfo = function (id, name, address, sex, mobile, qq) {
//   console.log('id= ' + id);
//   console.log('name= ' + name);
//   console.log('address= ' + address);
```

```
  //   console.log('sex= ' + sex);
  //   console.log('mobile= ' + mobile);
  //   console.log('qq= ' + qq);
  // };

  // setUserInfo('0001', 'zhangsan', '中国杭州', '男', '13312341234',
'1234567890');
```

进行调整，优化函数的参数

利用对象进行优化，对象的属性是无序的

注意：未来的使用过程中，很多的方法都是采用对象做参数。

```
    var setUserInfo = function (obj) {
      console.log('id= ' + obj.id);
      console.log('name= ' + obj.name);
      console.log('address= ' + obj.address);
      console.log('sex= ' + obj.sex);
      console.log('mobile= ' + obj.mobile);
      console.log('qq= ' + obj.qq);
    };

    setUserInfo({
      qq: '1234567890',
      id: '0001',
      name: 'lisi',
      sex: '女',
      address: '中国杭州',
      mobile: '13312345678'
    });
```

默认参数和配置参数的处理

Object.assign

```
  function fn(options) {//options:配置项
    // 配置默认参数，如果么有配置参数，使用默认参数
    const settings = {
      a: 11,
      b: 33,
      c: 55
    }
    // 利用ES6核心的方法 - 对象合并 - Object.assign
    options = Object.assign({}, settings, options);//将后面的属性合并到第一个参数上
面，如果出现重名，后面覆盖前面。
    console.log(options);// 如果不存在配置，输出默认的参数，如果存在配置项，输出配置项的
参数
  }

  fn({
```

```
    a: 1,
    b: 3,
    c: 5
  })
```

ajax携带参数的转换

本地服务器的ip地址：127.0.0.1

本地服务器主机(本地域名) ： localhost

用在本地测试的

```
  function querystring(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
  }
  // console.log(querystring(123));
  // console.log(querystring(obj));

  submit.onclick = function () {
    // 拼接数据
    const newData = querystring({
      username: username.value,
      password: password.value,
      rpassword: rpassword.value,
      nickname: nickname.value
    })
    let xhr = new XMLHttpRequest();
    xhr.open('POST', 'http://localhost:8888/users/register', true);
    xhr.onload = function () {
      if (xhr.status === 200) {
        console.log(JSON.parse(xhr.responseText));
      }
    };
    xhr.setRequestHeader('content-type', 'application/x-www-form-urlencoded');
    xhr.send(newData);
  }
```

ajax封装

封装函数的注意事项

1.参数的配置 - 对象做参数

2.默认参数和配置参数

3.封装的过程

```
一.封装ajax函数
1.ajax的四步曲，分析参数(将代码可变的部分设为参数)
1.1.参数1：请求类型GET|POST  默认是GET   (限制条件：仅支持GET POST)
1.2.参数2：接口地址，(限制条件：必须填写)
1.3.参数3：是否异步，默认是异步（限制条件：仅支持布尔值)
1.4.参数4：携带参数，如果参数存在，必须携带，可选的
1.5.参数5：是否将获取的数据转换成json对象，可选的
1.6.参数6：如果接口请求成功，利用回调函数获取接口的数据，调用时拼接
1.7.参数7：配置请求头
分析参数的个数，确定用对象进行参数的配置
2.将配置参数放入代码
3.配置限制条件
4.是否将获取的数据转换成json对象，可选的
*/

// 将对象转换成ajax识别的字符串格式
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_callback = 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$/i.test(options.type)) {
    options.url += '?' + querystring(options.data)
  }
  // 4.封装的和ajax相关的核心代码
  let xhr = new XMLHttpRequest();
  xhr.open(options.type, options.url, options.async);
  xhr.onload = function () {
    if (xhr.status === 200) {//请求成功
      // 配置获取的接口数据是否进行对象的转换
      // 如果dataType存在，并且值为json，利用JSON.parse进行转换
      let resData;
      if (options.dataType === 'json') {
        resData = JSON.parse(xhr.responseText)
      } else {
        resData = xhr.responseText
      }
      options.success(resData);
    } else {
      throw new Error('接口地址有误，请检查~~~~~');
    }
  };
  // 数据存在且请求方式是post
  // 约定传输数据的格式
  if (options.headers && options.headers['content-type']) {
    xhr.setRequestHeader('content-type', options.headers['content-type']);
  }
  // 免登录的令牌
  if (options.headers && options.headers['authorization']) {
    xhr.setRequestHeader('authorization', options.headers['authorization']);
  }
  options.data && /^post$/i.test(options.type) ?
xhr.send(querystring(options.data)) : xhr.send();
};
```

```
ajax_callback({
  type: 'post',
  url: 'http://localhost:8888/test/fourth',
  data: {
    name: 'wangwu',
    age: 250
  },
  headers: {//配置请求头
    'content-type': 'application/x-www-form-urlencoded',
    'authorization': '11111111111111111111111'
  },
  dataType: 'json',
  success(res) {
    console.log(res);
  }
})
```

回调函数的弊端:回调地狱

1.回调地狱概念:

其实就是回调函数嵌套过多导致的

2.产生的后果:

造成代码的可读性很差

当代码成为这个结构以后，已经没有维护的可能了

注意：回调函数依然是使用的核心，但是嵌套过多的时候尽量少用。

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

```
// 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
// 获取第一个接口的数据
ajax_callback({
  type: 'GET',
  url: 'http://localhost:8888/test/first',
  success: function (res) {
    first.innerHTML = res;
```

```
    ajax_callback({
      type: 'GET',
      url: 'http://localhost:8888/test/second',
      datType: 'json',
      success: function (res) {
        second.innerHTML = res;
        ajax_callback({
          type: 'GET',
          url: 'http://localhost:8888/test/third',
          datType: 'json',
          data: {
            name: 'zhangsna',
            age: 18
          },
          success: function (res) {
            third.innerHTML = res;
            ajax_callback({
              type: 'POST',
              url: 'http://localhost:8888/test/fourth',
              datType: 'json',
              data: {
                name: 'wangwu',
                age: 180
              },
              headers: {
                'content-type': 'application/x-www-form-urlencoded'
              },
              success: function (res) {
                fourth.innerHTML = res;
              }
            });
          }
        });
      }
    });
  }
});
```