接口调用方式

- 原生ajax
- 基于jQuery的ajax
- fetch
- axios

异步

- JavaScript的执行环境是「单线程」
- 所谓单线程，是指JS引擎中负责解释和执行JavaScript代码的线程只有一个，也就是一次只能完成一项任务，这个任务执行完后才能执行下一个，它会「阻塞」其他任务。这个任务可称为主线程
- 异步模式可以一起执行**多个任务**
- JS中常见的异步调用
  - 定时任何
  - ajax
  - 事件函数

promise

- 主要解决异步深层嵌套的问题
- promise 提供了简洁的API 使得异步操作更加容易

```
<script type="text/javascript">
  /*
   1. Promise基本使用
         我们使用new来构建一个Promise  Promise的构造函数接收一个参数，是函数，并且传入两个参数:
resolve, reject,  分别表示异步操作执行成功后的回调函数和异步操作执行失败后的回调函数
  */


  var p = new Promise(function(resolve, reject){
    //2. 这里用于实现异步任务  setTimeout
    setTimeout(function(){
      var flag = false;
      if(flag) {
        //3. 正常情况
        resolve('hello');
      }else{
        //4. 异常情况
        reject('出错了');
      }
    }, 100);
  });
  //  5 Promise实例生成以后，可以用then方法指定resolved状态和reject状态的回调函数
```

```
  //  在then方法中，你也可以直接return数据而不是Promise对象，在后面的then中就可以接收到数据了
  p.then(function(data){
    console.log(data)
  },function(info){
    console.log(info)
  });
</script>
```

基于Promise发送Ajax请求

```
<script type="text/javascript">
  /*
    基于Promise发送Ajax请求
  */
  function queryData(url) {
   #   1.1 创建一个Promise实例
    var p = new Promise(function(resolve, reject){
      var xhr = new XMLHttpRequest();
      xhr.onreadystatechange = function(){
        if(xhr.readyState != 4) return;
        if(xhr.readyState == 4 && xhr.status == 200) {
          # 1.2 处理正常的情况
          resolve(xhr.responseText);
        }else{
          # 1.3 处理异常情况
          reject('服务器错误');
        }
      };
      xhr.open('get', url);
      xhr.send(null);
    });
    return p;
  }
  # 注意:   这里需要开启一个服务
  # 在then方法中，你也可以直接return数据而不是Promise对象，在后面的then中就可以接收到数据了
  queryData('http://localhost:3000/data')
    .then(function(data){
      console.log(data)
      #  1.4 想要继续链式编程下去 需要 return
      return queryData('http://localhost:3000/data1');
    })
    .then(function(data){
      console.log(data);
      return queryData('http://localhost:3000/data2');
    })
    .then(function(data){
      console.log(data)
    });
</script>
```

Promise 基本API

实例方法

.then()

- 得到异步任务正确的结果

.catch()

- 获取异常信息

.finally()

- 成功与否都会执行（不是正式标准）

```
<script type="text/javascript">
  /*
    Promise常用API-实例方法
  */
  // console.dir(Promise);
  function foo() {
    return new Promise(function(resolve, reject){
      setTimeout(function(){
        // resolve(123);
        reject('error');
      }, 100);
    })
  }
  // foo()
  //   .then(function(data){
  //     console.log(data)
  //   })
  //   .catch(function(data){
  //     console.log(data)
  //   })
  //   .finally(function(){
  //     console.log('finished')
  //   });

  // --------------------------
  // 两种写法是等效的
  foo()
    .then(function(data){
      # 得到异步任务正确的结果
      console.log(data)
    },function(data){
      # 获取异常信息
      console.log(data)
    })
    # 成功与否都会执行（不是正式标准）
    .finally(function(){
      console.log('finished')
    });
</script>
```

静态方法

.all()

- `Promise.all`方法接受一个数组作参数，数组中的对象（p1、p2、p3）均为promise实例（如果不是一个promise，该项会被用`Promise.resolve`转换为一个promise)。它的状态由这三个promise实例决定

.race()

- `Promise.race`方法同样接受一个数组作参数。当p1, p2, p3中有一个实例的状态发生改变（变为`fulfilled`或`rejected`），p的状态就跟着改变。并把第一个改变状态的promise的返回值，传给p的回调函数

```
  <script type="text/javascript">
    /*
      Promise常用API-对象方法
    */
    // console.dir(Promise)
    function queryData(url) {
      return new Promise(function(resolve, reject){
        var xhr = new XMLHttpRequest();
        xhr.onreadystatechange = function(){
          if(xhr.readyState != 4) return;
          if(xhr.readyState == 4 && xhr.status == 200) {
            // 处理正常的情况
            resolve(xhr.responseText);
          }else{
            // 处理异常情况
            reject('服务器错误');
          }
        };
        xhr.open('get', url);
        xhr.send(null);
      });
    }

    var p1 = queryData('http://localhost:3000/a1');
    var p2 = queryData('http://localhost:3000/a2');
    var p3 = queryData('http://localhost:3000/a3');
     Promise.all([p1,p2,p3]).then(function(result){
       //   all 中的参数  [p1,p2,p3]   和 返回的结果一 一对应["HELLO TOM", "HELLO JERRY",
"HELLO SPIKE"]
       console.log(result) //["HELLO TOM", "HELLO JERRY", "HELLO SPIKE"]
     })
    Promise.race([p1,p2,p3]).then(function(result){
      // 由于p1执行较快，Promise的then()将获得结果'P1'。p2,p3仍在继续执行，但执行结果将被丢弃。
      console.log(result) // "HELLO TOM"
    })
  </script>
```

fetch

- Fetch API是新的ajax解决方案 Fetch会返回Promise
- **fetch不是ajax的进一步封装，而是原生js，没有使用XMLHttpRequest对象。**

- fetch(url, options).then()

```
<script type="text/javascript">
  /*
    Fetch API 基本用法
      fetch(url).then()
      第一个参数请求的路径    Fetch会返回Promise    所以我们可以使用then 拿到请求成功的结果
  */
  fetch('http://localhost:3000/fdata').then(function(data){
    // text()方法属于fetchAPI的一部分，它返回一个Promise实例对象，用于获取后台返回的数据
    return data.text();
  }).then(function(data){
    //    在这个then里面我们能拿到最终的数据
    console.log(data);
  })
</script>
```

fetch API 中的 HTTP 请求

- fetch(url, options).then()
- HTTP协议，它给我们提供了很多的方法，如POST，GET，DELETE，UPDATE，PATCH和PUT
  - 默认的是 GET 请求
  - 需要在 options 对象中 指定对应的 method method:请求使用的方法
  - post 和 普通 请求的时候 需要在options 中 设置 请求头 headers 和 body

```
 <script type="text/javascript">
      /*
            Fetch API 调用接口传递参数
      */
     #1.1 GET参数传递 - 传统URL  通过url  ?  的形式传参
      fetch('http://localhost:3000/books?id=123', {
              # get 请求可以省略不写 默认的是GET
              method: 'get'
          })
          .then(function(data) {
              # 它返回一个Promise实例对象，用于获取后台返回的数据
              return data.text();
          }).then(function(data) {
              # 在这个then里面我们能拿到最终的数据
              console.log(data)
          });

     #1.2  GET参数传递  restful形式的URL  通过/ 的形式传递参数  即  id = 456 和id后台的配置有关
      fetch('http://localhost:3000/books/456', {
              # get 请求可以省略不写 默认的是GET
              method: 'get'
          })
          .then(function(data) {
              return data.text();
          }).then(function(data) {
              console.log(data)
```

```
    });

#2.1  DELETE请求方式参数传递      删除id  是  id=789
 fetch('http://localhost:3000/books/789', {
        method: 'delete'
    })
    .then(function(data) {
        return data.text();
    }).then(function(data) {
        console.log(data)
    });

#3 POST请求传参
 fetch('http://localhost:3000/books', {
        method: 'post',
        # 3.1  传递数据
        body: 'uname=lisi&pwd=123',
        #  3.2  设置请求头
        headers: {
            'Content-Type': 'application/x-www-form-urlencoded'
        }
    })
    .then(function(data) {
        return data.text();
    }).then(function(data) {
        console.log(data)
    });

# POST请求传参
 fetch('http://localhost:3000/books', {
        method: 'post',
        body: JSON.stringify({
            uname: '张三',
            pwd: '456'
        }),
        headers: {
            'Content-Type': 'application/json'
        }
    })
    .then(function(data) {
        return data.text();
    }).then(function(data) {
        console.log(data)
    });

 # PUT请求传参     修改id 是 123 的
 fetch('http://localhost:3000/books/123', {
        method: 'put',
        body: JSON.stringify({
            uname: '张三',
            pwd: '789'
        }),
        headers: {
```

```
                    'Content-Type': 'application/json'
                }
            })
            .then(function(data) {
                return data.text();
            }).then(function(data) {
                console.log(data)
            });
    </script>
```

fetchAPI 中 响应格式

- 用fetch来获取数据，如果响应正常返回，我们首先看到的是一个response对象，其中包括返回的一堆原始字节，这些字节需要在收到后，需要我们通过调用方法将其转换为相应格式的数据，比如`JSON`，`BLOB`或者`TEXT`等等

```
    /*
      Fetch响应结果的数据格式
    */
    fetch('http://localhost:3000/json').then(function(data){
      // return data.json();    //  将获取到的数据使用 json 转换对象
      return data.text(); //  //  将获取到的数据 转换成字符串
    }).then(function(data){
      // console.log(data.uname)
      // console.log(typeof data)
      var obj = JSON.parse(data);
      console.log(obj.uname,obj.age,obj.gender)
    })
```

axios

- 基于promise用于浏览器和node.js的http客户端
- 支持浏览器和node.js
- 支持promise
- 能拦截请求和响应
- 自动转换JSON数据
- 能转换请求和响应数据

axios基础用法

- get和 delete请求传递参数
    - 通过传统的url 以 ? 的形式传递参数
    - restful 形式传递参数
    - 通过params 形式传递参数
- post 和 put 请求传递参数
    - 通过选项传递参数
    - 通过 URLSearchParams 传递参数

```
# 1. 发送get 请求
axios.get('http://localhost:3000/adata').then(function(ret){
  #  拿到 ret 是一个对象       所有的对象都存在 ret 的data 属性里面
  // 注意data属性是固定的用法，用于获取后台的实际数据
  // console.log(ret.data)
  console.log(ret)
})
# 2.  get 请求传递参数
# 2.1  通过传统的url  以 ? 的形式传递参数
axios.get('http://localhost:3000/axios?id=123').then(function(ret){
  console.log(ret.data)
})
# 2.2  restful 形式传递参数
axios.get('http://localhost:3000/axios/123').then(function(ret){
  console.log(ret.data)
})
# 2.3  通过params  形式传递参数
axios.get('http://localhost:3000/axios', {
  params: {
    id: 789
  }
}).then(function(ret){
  console.log(ret.data)
})
#3 axios delete 请求传参      传参的形式和 get 请求一样
axios.delete('http://localhost:3000/axios', {
  params: {
    id: 111
  }
}).then(function(ret){
  console.log(ret.data)
})

# 4  axios 的 post 请求
# 4.1  通过选项传递参数
axios.post('http://localhost:3000/axios', {
  uname: 'lisi',
  pwd: 123
}).then(function(ret){
  console.log(ret.data)
})
# 4.2  通过 URLSearchParams  传递参数
var params = new URLSearchParams();
params.append('uname', 'zhangsan');
params.append('pwd', '111');
axios.post('http://localhost:3000/axios', params).then(function(ret){
  console.log(ret.data)
})

#5  axios put 请求传参   和 post 请求一样
axios.put('http://localhost:3000/axios/123', {
  uname: 'lisi',
  pwd: 123
```

```
    }).then(function(ret){
      console.log(ret.data)
    })
```

axios 全局配置

```
#  配置公共的请求头
axios.defaults.baseURL = 'https://api.example.com';
#  配置 超时时间
axios.defaults.timeout = 2500;
#  配置公共的请求头
axios.defaults.headers.common['Authorization'] = AUTH_TOKEN;
# 配置公共的 post 的 Content-Type
axios.defaults.headers.post['Content-Type'] = 'application/x-www-form-urlencoded';
```

axios 拦截器

- 请求拦截器
  - 请求拦截器的作用是在请求发送前进行一些操作
    - 例如在每个请求体里加上token，统一做了处理如果以后要改也非常容易
- 响应拦截器
  - 响应拦截器的作用是在接收到响应后进行一些操作
    - 例如在服务器返回登录状态失效，需要重新登录的时候，跳转到登录页

```
# 1. 请求拦截器
axios.interceptors.request.use(function(config) {
  console.log(config.url)
  # 1.1  任何请求都会经过这一步    在发送请求之前做些什么
  config.headers.mytoken = 'nihao';
  # 1.2  这里一定要return    否则配置不成功
  return config;
}, function(err){
   #1.3 对请求错误做点什么
  console.log(err)
})
#2. 响应拦截器
axios.interceptors.response.use(function(res) {
  #2.1  在接收响应做些什么
  var data = res.data;
  return data;
}, function(err){
  #2.2 对响应错误做点什么
  console.log(err)
})
```

async 和 await

- async作为一个关键字放到函数前面
  - 任何一个`async`函数都会隐式返回一个`promise`
- `await`关键字只能在使用`async`定义的函数中使用
  - await后面可以直接跟一个 Promise实例对象
  - await函数不能单独使用
- **async/await 让异步代码看起来、表现起来更像同步代码**

```
# 1.  async 基础用法
# 1.1 async作为一个关键字放到函数前面
async function queryData() {
  # 1.2 await关键字只能在使用async定义的函数中使用       await后面可以直接跟一个 Promise实例对象
  var ret = await new Promise(function(resolve, reject){
    setTimeout(function(){
      resolve('nihao')
    },1000);
  })
  // console.log(ret.data)
  return ret;
}
# 1.3 任何一个async函数都会隐式返回一个promise    我们可以使用then 进行链式编程
queryData().then(function(data){
  console.log(data)
})

#2.  async    函数处理多个异步函数
axios.defaults.baseURL = 'http://localhost:3000';

async function queryData() {
  # 2.1  添加await之后 当前的await 返回结果之后才会执行后面的代码

  var info = await axios.get('async1');
  #2.2  让异步代码看起来、表现起来更像同步代码
  var ret = await axios.get('async2?info=' + info.data);
  return ret.data;
}

queryData().then(function(data){
  console.log(data)
})
```

图书列表案例

1. 基于接口案例-获取图书列表

- 导入axios 用来发送ajax
- 把获取到的数据渲染到页面上

```
<div id="app">
      <div class="grid">
          <table>
              <thead>
```

```
                <tr>
                    <th>编号</th>
                    <th>名称</th>
                    <th>时间</th>
                    <th>操作</th>
                </tr>
            </thead>
            <tbody>
                <!-- 5.  把books  中的数据渲染到页面上   -->
                <tr :key='item.id' v-for='item in books'>
                    <td>{{item.id}}</td>
                    <td>{{item.name}}</td>
                    <td>{{item.date }}</td>
                    <td>
                        <a href="">修改</a>
                        <span>|</span>
                        <a href="">删除</a>
                    </td>
                </tr>
            </tbody>
        </table>
    </div>
</div>
<script type="text/javascript" src="js/vue.js"></script>
1.  导入axios
<script type="text/javascript" src="js/axios.js"></script>
<script type="text/javascript">
    /*
         图书管理-添加图书
     */
    # 2   配置公共的url地址  简化后面的调用方式
    axios.defaults.baseURL = 'http://localhost:3000/';
    axios.interceptors.response.use(function(res) {
        return res.data;
    }, function(error) {
        console.log(error)
    });

    var vm = new Vue({
        el: '#app',
        data: {
            flag: false,
            submitFlag: false,
            id: '',
            name: '',
            books: []
        },
        methods: {
            # 3 定义一个方法 用来发送 ajax
            # 3.1  使用 async  来 让异步的代码  以同步的形式书写
            queryData: async function() {
                // 调用后台接口获取图书列表数据
                // var ret = await axios.get('books');
```

```
                    // this.books = ret.data;
                    # 3.2  发送ajax请求  把拿到的数据放在books 里面
                    this.books = await axios.get('books');
                }
            },

            mounted: function() {
                #  4 mounted  里面 DOM已经加载完毕  在这里调用函数
                this.queryData();
            }
        });
    </script>
```

2 添加图书

- 获取用户输入的数据 发送到后台
- 渲染最新的数据到页面上

```
methods: {
    handle: async function(){
          if(this.flag) {
            // 编辑图书
            // 就是根据当前的ID去更新数组中对应的数据
            this.books.some((item) => {
              if(item.id == this.id) {
                item.name = this.name;
                // 完成更新操作之后，需要终止循环
                return true;
              }
            });
            this.flag = false;
          }else{
            # 1.1  在前面封装好的 handle 方法中  发送ajax请求
            # 1.2  使用async  和 await 简化操作 需要在 function 前面添加 async
            var ret = await axios.post('books', {
              name: this.name
            })
            # 1.3  根据后台返回的状态码判断是否加载数据
            if(ret.status == 200) {
             # 1.4  调用 queryData 这个方法  渲染最新的数据
              this.queryData();
            }
          }
          // 清空表单
          this.id = '';
          this.name = '';
        },
}
```

3 验证图书名称是否存在

- 添加图书之前发送请求验证图示是否已经存在

- 如果不存在 往后台里面添加图书名称
    - 图书存在与否只需要修改submitFlag的值即可

```
watch: {
       name: async function(val) {
         // 验证图书名称是否已经存在
         // var flag = this.books.some(function(item){
         //   return item.name == val;
         // });
         var ret = await axios.get('/books/book/' + this.name);
         if(ret.status == 1) {
           // 图书名称存在
           this.submitFlag = true;
         }else{
           // 图书名称不存在
           this.submitFlag = false;
         }
       }
},
```

4. 编辑图书

- 根据当前书的id 查询需要编辑的书籍
- 需要根据状态位判断是添加还是编辑

```
methods: {
       handle: async function(){
         if(this.flag) {
           #4.3 编辑图书    把用户输入的信息提交到后台
           var ret = await axios.put('books/' + this.id, {
             name: this.name
           });
           if(ret.status == 200){
             #4.4  完成添加后 重新加载列表数据
             this.queryData();
           }
           this.flag = false;
         }else{
           // 添加图书
           var ret = await axios.post('books', {
             name: this.name
           })
           if(ret.status == 200) {
             // 重新加载列表数据
             this.queryData();
           }
         }
         // 清空表单
         this.id = '';
         this.name = '';
       },
       toEdit: async function(id){
```

```
        #4.1  flag状态位用于区分编辑和添加操作
        this.flag = true;
        #4.2  根据id查询出对应的图书信息   页面中可以加载出来最新的信息
        # 调用接口发送ajax 请求
        var ret = await axios.get('books/' + id);
        this.id = ret.id;
        this.name = ret.name;
      },
```

5 删除图书

- 把需要删除的id书籍 通过参数的形式传递到后台

```
deleteBook: async function(id){
       // 删除图书
       var ret = await axios.delete('books/' + id);
       if(ret.status == 200) {
         // 重新加载列表数据
         this.queryData();
       }
}
```