主要内容

- 常见的网络请求模块，以及优缺点对比。
- JSONP的原理和封装
  - JSONP原理回顾
  - JSONP请求封装
- axios的内容详解
  - 认识axios网络模块
  - 发送基本请求
  - axios创建实例
  - axios拦截器的使用

选择什么网络模块?

- Vue中发送网络请求有非常多的方式, 那么, 在开发中, 如何选择呢?

- 选择一: 传统的Ajax是基于XMLHttpRequest(XHR)
- 为什么不用它呢?
  - 非常好解释, 配置和调用方式等非常混乱.
  - 编码起来看起来就非常蛋疼.
  - 所以真实开发中很少直接使用, 而是使用jQuery-Ajax

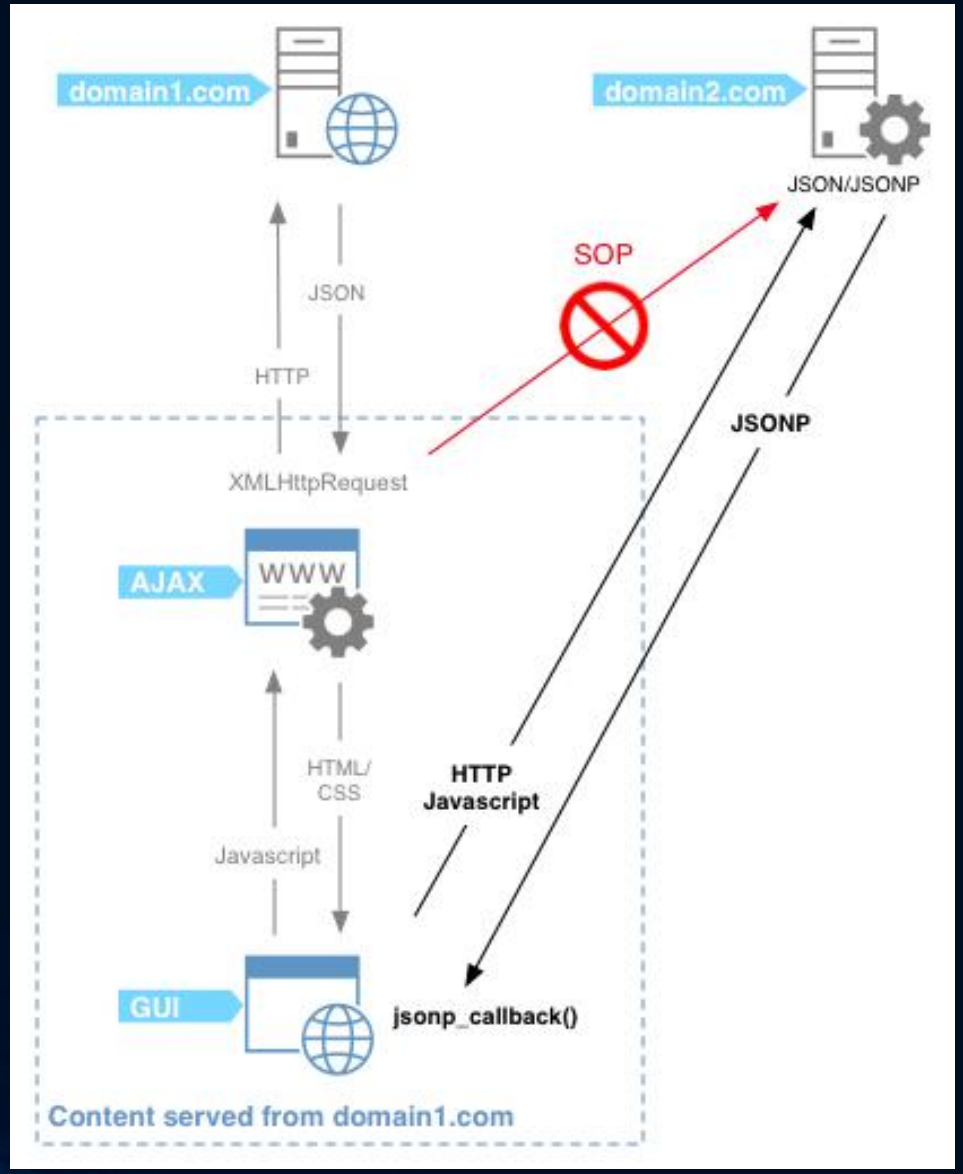
- 选择二: 在前面的学习中, 我们经常会使用jQuery-Ajax
  - 相对于传统的Ajax非常好用.
- 为什么不选择它呢?
  - 首先, 我们先明确一点: 在Vue的整个开发中都是不需要使用jQuery了.
  - 那么, 就意味着为了方便我们进行一个网络请求, 特意引用一个jQuery, 你觉得合理吗?
  - jQuery的代码1w+行.
  - Vue的代码才1w+行.
  - 完全没有必要为了用网络请求就引用这个重量级的框架.

- 选择三: 官方在Vue1.x的时候, 推出了Vue-resource.
  - Vue-resource的体积相对于jQuery小很多.
  - 另外Vue-resource是官方推出的.
- 为什么不选择它呢?
  - 在Vue2.0退出后, Vue作者就在GitHub的Issues中说明了去掉vue-resource, 并且以后也不会再更新.
  - 那么意味着以后vue-reource不再支持新的版本时, 也不会再继续更新和维护.
  - 对以后的项目开发和维护都存在很大的隐患.

- 选择四: 在说明不再继续更新和维护vue-resource的同时, 作者还推荐了一个框架: axios为什么不用它呢?
  - axios有非常多的优点, 并且用起来也非常方便.
  - 稍后, 我们对他详细学习.

jsonp

- 在前端开发中, 我们一种常见的网络请求方式就是JSONP
  - 使用JSONP最主要的原因往往是为了解决跨域访问的问题.
- JSONP的原理是什么呢?
  - JSONP的核心在于通过<script>标签的src来帮助我们请求数据.
  - 原因是我们的项目部署在domain1.com服务器上时, 是不能直接访问domain2.com服务器上的资料的.
  - 这个时候, 我们利用<script>标签的src帮助我们去服务器请求到数据, 将数据当做一个javascript的函数来执行, 并且执行的过程中传入我们需要的json.
  - 所以, 封装jsonp的核心就在于我们监听window上的jsonp进行回调时的名称.
- JSONP如何封装呢?
  - 我们一起自己来封装一个处理JSONP的代码吧.



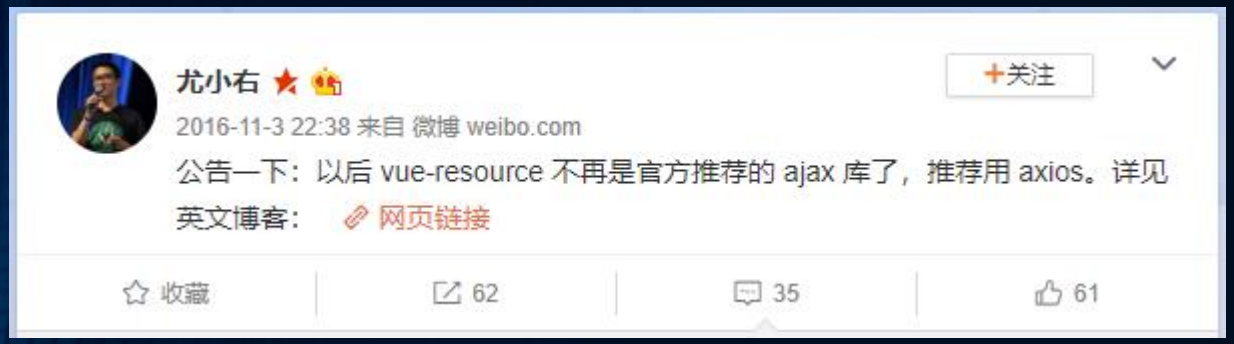
JSONP封装

```
let count = 1
export default function originPJSONP(option) {
  // 1.从传入的option中提取URL
  const url = option.url;

  // 2.在body中添加script标签
  const body = document.getElementsByTagName('body')[0]
  const script = document.createElement('script');

  // 3.内部生产一个不重复的callback
  const callback = 'jsonp' + count++

  // 4.监听window上的jsonp的调用
  return new Promise((resolve, reject) => {
    try {
      window[callback] = function (result) {
        body.removeChild(script);
        resolve(result)
      }
      const params = handleParam(option.data);
      script.src = url + '?callback=' + callback + params;
      body.appendChild(script)
    } catch (e) {
      body.removeChild(script)
      reject(e)
    }
  })
}
```

```
function handleParam(data) {
  let url = ''
  for (let key in data) {
    let value = data[key] !== undefined ? data[key] : ''
    url += `&${key}=${encodeURIComponent(value)}`
  }
  return url
}
```

为什么选择axios?

- 为什么选择axios? 作者推荐和功能特点



- 功能特点:
  - 在浏览器中发送 XMLHttpRequests 请求
  - 在 node.js 中发送 http请求
  - 支持 Promise API
  - 拦截请求和响应
  - 转换请求和响应数据
  - 等等

- 补充: axios名称的由来? 个人理解
  - 没有具体的翻译.
  - axios: ajax i/o system.

axiox请求方式

- 支持多种请求方式:
  - axios(config)
  - axios.request(config)
  - axios.get(url[, config])
  - axios.delete(url[, config])
  - axios.head(url[, config])
  - axios.post(url[, data[, config]])
  - axios.put(url[, data[, config]])
  - axios.patch(url[, data[, config]])
- 如何发送请求呢?
  - 我们看一下左边的案例

发送get请求演示

```
import axios from 'axios'

export default {
  name: 'app',
  created() {
    // 提问: 为什么我这里没有跨域的问题?
    // 1.没有请求参数
    axios.get('http://123.207.32.32:8000/category')
      .then(res => {
      console.log(res);
    }).catch(err => {
      console.log(err);
    })

    // 2.有请求参数
    axios.get('http://123.207.32.32:8000/home/data',
      {params: {type: 'sell', page: 1}})
      .then(res => {
        console.log(res);
      }).catch(err => {
        console.log(err);
    })
  }
}
```

发送并发请求

- 有时候, 我们可能需求同时发送两个请求
  - 使用axios.all, 可以放入多个请求的数组.
  - axios.all([]) 返回的结果是一个数组，使用 axios.spread 可将数组 [res1,res2] 展开为 res1, res2

```
import axios from 'axios'

export default {
  name: 'app',
  created() {
    // 发送并发请求
    axios.all([axios.get('http://123.207.32.32:8000/category'),
               axios.get('http://123.207.32.32:8000/home/data',
                 {params: {type: 'sell', page: 1}})])
      .then(axios.spread((res1, res2) => {
        console.log(res1);
        console.log(res2);
      }))
  }
```

全局配置

- 在上面的示例中, 我们的BaseURL是固定的
  - 事实上, 在开发中可能很多参数都是固定的.
  - 这个时候我们可以进行一些抽取, 也可以利用axiox的全局配置

```
axios.defaults.baseURL = ‘123.207.32.32:8000’
axios.defaults.headers.post[‘Content-Type’] = ‘application/x-www-form-urlencoded’;
```

```
created() {
  // 提取全局的配置
  axios.defaults.baseURL = 'http://123.207.32.32:8000'

  // 发送并发请求
  axios.all([axios.get('/category'),
             axios.get('/home/data',
               {params: {type: 'sell', page: 1}})])
    .then(axios.spread((res1, res2) => {
      console.log(res1);
      console.log(res2);
    }))
}
```

常见的配置选项

- 请求地址
  - url: '/user',
- 请求类型
  - method: 'get',
- 请根路径
  - baseURL: 'http://www.mt.com/api',
- 请求前的数据处理
  - transformRequest:[function(data){}],
- 请求后的数据处理
  - transformResponse: [function(data){}],
- 自定义的请求头
  - headers:{'x-Requested-With':'XMLHttpRequest'},
- URL查询对象
  - params:{ id: 12 },
- 查询对象序列化函数
  - paramsSerializer: function(params){ }
- request body
  - data: { key: 'aa'},
- 超时设置s
  - timeout: 1000,
- 跨域是否带Token
  - withCredentials: false,
- 自定义请求处理
  - adapter: function(resolve, reject, config){},
- 身份验证信息
  - auth: { uname: '', pwd: '12'},
- 响应的数据格式 json / blob /document /arraybuffer / text / stream
  - responseType: 'json',

axios的实例

- 为什么要创建axios的实例呢?
  - 当我们从axios模块中导入对象时, 使用的实例是默认的实例.
  - 当给该实例设置一些默认配置时, 这些配置就被固定下来了.
  - 但是后续开发中, 某些配置可能会不太一样.
  - 比如某些请求需要使用特定的baseURL或者timeout或者content-Type等.
  - 这个时候, 我们就可以创建新的实例, 并且传入属于该实例的配置信息.

```
// 创建新的实例
const axiosInstance = axios.create({
  baseURL: 'http://123.207.32.32:8000',
  timeout: 5000,
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded'
  }
})
```

```
// 发送网络请求
axiosInstance({
  url: '/category',
  method: 'get'
}).then(res => {
  console.log(res);
}).catch(err => {
  console.log(err);
})
```

axios封装

axios.js

```
import originAxios from 'axios'

export default function axios(option) {
  return new Promise((resolve, reject) => {
    // 1.创建axios的实例
    const instance = originAxios.create({
      baseURL: '/api',
      timeout: 5000,
      headers: ''
    });

    // 2.传入对象进行网络请求
    instance(option).then(res => {
      resolve(res)
    }).catch(err => {
      reject(err)
    })
  })
}
```

如何使用拦截器?

- axios提供了拦截器，用于我们在发送每次请求或者得到相应后，进行对应的处理。
- 如何使用拦截器呢?

```
// 配置请求和响应拦截
instance.interceptors.request.use(config => {
  console.log('来到了request拦截success中');
  return config
}, err => {
  console.log('来到了request拦截failure中');
  return err
})

instance.interceptors.response.use(response => {
  console.log('来到了response拦截success中');
  return response.data
}, err => {
  console.log('来到了response拦截failure中');
  return err
})
```

```
axios({
  url: '/home/data',
  method: 'get',
  params: {
    type: 'sell',
    page: 1
  }
}).then(res => {
  console.log(res);
}).catch(err => {
  console.log(err);
})
```

```
来到了request拦截success中
来到了response拦截success中
▶ {data: {…}, status: 200, statusText: "OK", headers: {…}, config: {…}, …}
```

拦截器中都做什么呢?

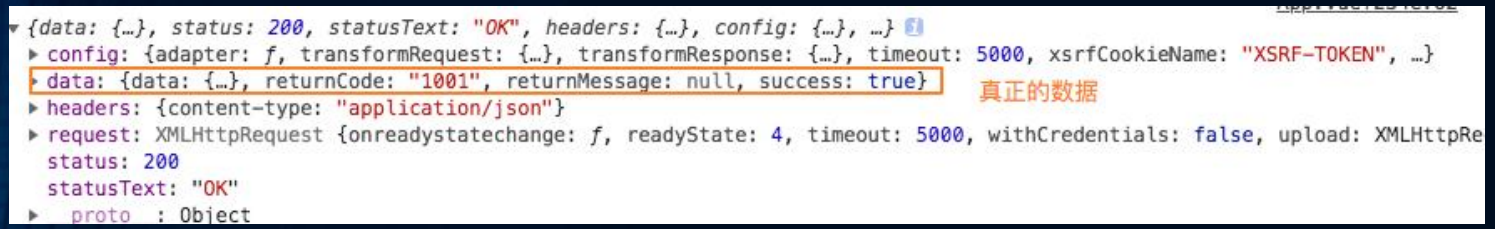
- 请求拦截可以做到的事情:

```
instance.interceptors.request.use(config => {
  console.log('来到了request拦截success中');
  // 1.当发送网络请求时，在页面中添加一个loading组件，作为动画

  // 2.某些请求要求用户必须登录，判断用户是否有token，如果没有token跳转到login页面

  // 3.对请求的参数进行序列化
  config.data = qs.stringify(config.data)
  console.log(config);

  // 4.等等
  return config
```

- 请求拦截中错误拦截较少，通常都是配置相关的拦截
  - 可能的错误比如请求超时，可以将页面跳转到一个错误页面中。

拦截器中都做什么呢?

- 响应拦截中完成的事情:
  - 响应的成功拦截中，主要是对数据进行过滤。




```
instance.interceptors.response.use(response => {
  console.log('来到了response拦截success中');
  return response.data
```

  - 响应的失败拦截中，可以根据status判断报错的错误码，跳转到不同的错误提示页面。

```
}, err => {
  console.log('来到了response拦截failure中');
  if (err && err.response) {
    switch (err.response.status) {
      case 400:
        err.message = '请求错误'
        break
      case 401:
        err.message = '未授权的访问'
        break
    }
  }
  return err
```