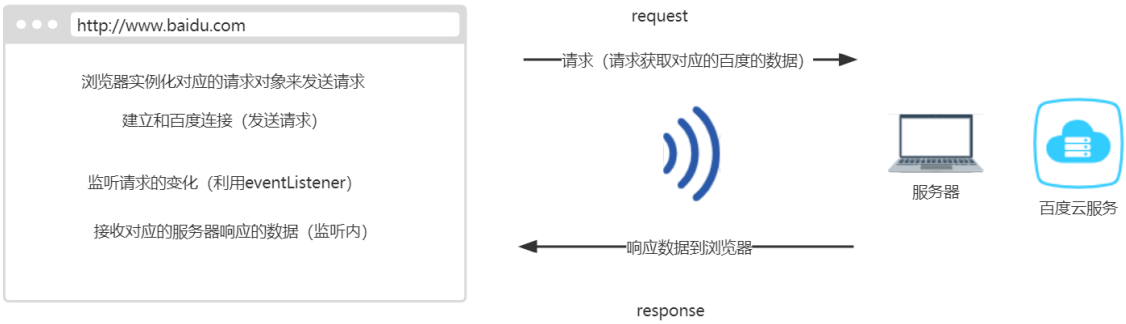
day22 ajax

ajax

概述:

ajax （asynchronous JavaScript and xml）异步的JavaScript和xml（传输格式为xml 但一般为json），它主要是用于请求对应的数据来帮助渲染。

数据请求流程



ajax的基础代码实现

核心对象 （xmlHttpRequest （简称xhr））

```
//实例化请求对象
var xhr = new XMLHttpRequest()
//建立连接 （请求方式  get请求（能看到）  post请求（看不到））
xhr.open(请求方式,地址)
//发送请求
xhr.send()
//监听请求的变化 readystate (状态值 xhr独有的)
xhr.onreadystatechange = function(){
//在监听的处理函数内容接收响应数据
//判断是否有数据成功返回 readystate 0 - 4
//(0 请求未发送 1表示请求准备发送 2表示请求发送成功 3请求成功响应中 4 请求成功 响应完成)
//http状态码 xhr.status (100-599)
if(xhr.readyState == 4 && /^2\d{2}$/.test(xhr.status)){
    console.log(xhr.responseXML) //接收xml
    console.log(xhr.responseText) //接收文本 字符串
}
}
```

xhr （xmlHttpRequest ）的方法及属性

属性

- readyState xhr的状态值
- response 获取响应
- responseText 获取响应文本
- responseType 获取响应类型

- responseURL 获取响应地址
- responseXML 获取响应的xml
- status 获取http的状态码 （后台返回的）
- statusText 获取http的状态文本 （后台返回的）
- timeout 超时时间 （设置的）

方法

- upload （返回一个上传对象 （XMLHttpRequestUpload））
- open 打开一个连接 （传入对应的请求方式和地址地址）
- send 发一个请求 （内部传入的是请求体的内容）
- getAllResponseHeaders 获取所有的响应头 （返回的是字符串）
- getResponseHeader 获取指定响应头 （传入对应的响应头的名字）
- XMLHttpRequest 的 `overrideMimeType` 方法是指定一个 MIME 类型用于替代服务器指定的类型，使服务端响应信息中传输的数据按照该指定 MIME 类型处理。例如强制使流方式处理为"text/xml"类型处理时会被使用到，即使服务器在响应头中并没有这样指定
- setRequestHeader 设置请求头 （里面传入对应的请求头名字和对应的值）

事件

- onreadystateChange readyState的值发生变化的时候调用
- ontimeout 超时之后调用的

后台接口返回具备的三个内容（响应）

- 状态码 （status）
- 状态文本（相关信息 statusText）
- 数据 （data）

请求和响应的组成

请求相关

- 请求头 requestHeader （对应的请求头先被服务器接收 验证后再接收请求体）

```
//常用请求头
//Content-type 请求的内容的类型
//Cookie cookie存在于请求头
//Connection  http.1.1新增的 （keep-alive 缓存当前连接 长连接）
//User-Agent 浏览器相关版本信息
//Referer 对应的请求的详细地址
```

- 请求体 requestBody

```
post 请求将数据添充到请求体 （不可见的）
请求体数据量大 发送的时候是分段发送的（后台获取的时候需要拼接获取）
```

响应相关

响应头（responseHeader）

- Cache-Control 缓存控制器 （强制缓存 1.1新增的）
- expires 缓存控制 （1.0）
- Content-Type 响应内容的类型
- Accpet-Control-Allow-Origin 设置跨域的响应头

- last-modified 文件最后修改的时间 （协商缓存）
- etag 文件标识名 （协商缓存）

响应体 （数据 状态码 状态文本）responseBody

http状态码 (status)

主要取值为100-599内 分别用对应的开头来表示对应的内容

开头状态码	含义	常用状态码
1开头	表示成功但是需要下一步操作	100
2开头	表示成功	200
3开头	表示重定向	304、303
4开头	表示客户端错误	404 、401 、402、403、400
5开头	表示服务端错误	500、501

xhr的readyState

readyState取值	含义
0	请求未发送
1	请求准备发送
2	请求发送成功
3	请求成功响应中
4	请求成功 响应完成

强制缓存和协商缓存

http的缓存

为了节省对应的资源，减少对应的请求。产生了缓存。缓存当于下载也就是说如果缓存了那么就不需要再请求对应的服务器了。

强制缓存 （对应的内容强制被缓存 没有请求的发送）

cache-control 来控制

```
cache-control:no-cache //开启强制缓存 no-storage 不开启强制缓存
cache-control:MaxAge(9000) //规定时间内使用强制缓存
cache-control:public //一定缓存 （第三方代理缓存）
cache-control:private //看客户端的处理
```

协商缓存 （不使用强制缓存 才会采用协商缓存）

- last-modified 最后修改时间
- etag 文件标识名

如果这个文件在访问的时候是a文件 你第二次的时候 我将a文件进行了修改那么对应的 last-modified就会发生更改 那么就会进行数据的重新请求，如果没有进行修改，那么这个时候对应的a文件还是a文件，那么我就需要比对a文件是否是a文件 （etag 来比对），如果是那么不进行缓存，如果不是重新请求。

协商缓存一定要发请求，如果被缓存那么对应的状态码为304 如果没有缓存那么就是一个新的请求（如果成功就是200）

总结

- 优先使用强制缓存 如果没有强制缓存 使用协商缓存
- 如果强制缓存了 那么协商缓存就不会被使用 且不会有请求发送
- 如果没有强制缓存那么对应的协商缓存就会触发 主要比对的是最后修改时间和对应的文件标识如果都比对比成功那么就是使用协商缓存，如果没有比对上那么就不会缓存。
- 如果协商缓存成功那么对应的状态码为304 如果协商缓存失败那么对应的状态码就是一个新请求的状态码（成功就是200）
- 强制缓存使用对应的响应头的cache-control来控制 协商缓存使用响应头的 etag 和 last-modified 来控制
- 强制缓存没有请求 协商缓存必须请求

get请求和post

get请求 （获取数据）

- get请求一般用于获取数据（安全性低 效率高 速度快）
- 默认的请求方式为get请求 （没有设置对应的请求那么就是get请求）
- 传输的参数利用?进行拼接，拼接到对应的url
- get请求会默认缓存对应的参数
- 传输的参数会在请求头中url
- get请求传输的数据大小有限制 （2kb）

post请求 （文件上传 登录 注册...）

- post请求的数据（封装为一个表单对象）在请求体中
- post请求相对get要安全
- post请求的数据量大于get （数据没有大小限制）
- post请求的数据不会被缓存
- post请求必须手动设置为post请求

get请求的封装

回调函数

```
//封装get请求的方法
function get(params = {}, callback) {
  //判断是否传入url
  if (!params.url) {
    throw new Error('必须传入url地址')
  }
  //创建xhr对象
  var xhr = new XMLHttpRequest()
  //参数和baseUrl进行拼接
  let baseURL = params.url
  //遍历参数对象
  for (var key in params) {
    if (key != 'url') {
```

```
      //判断是否存在?
      // 如果没有前面带?
      // 如果有前面带&
      let s = baseURL.includes('?') ? '&' : '?'
      baseURL += `${s+key}=${params[key]}`
    }
  }
  //打开连接
  xhr.open('get', baseURL)
  //发送请求
  xhr.send()
  //监听事件
  xhr.onreadystatechange = () => {
    if (xhr.readyState == 4 && /^2\d{2}$/.test(xhr.status)) {
      //调用回调函数 进行相关操作 传入对应的返回的数据
      callback(JSON.parse(xhr.responseText))
    }
  }
}
```

promise版本

```
//封装get请求的方法 利用promise来解决
export function get(params = {}) {
  //判断是否传入url
  if (!params.url) {
    throw new Error('必须传入url地址')
  }
  //创建xhr对象
  var xhr = new XMLHttpRequest()
  //参数和baseUrl进行拼接
  let baseURL = params.url
  //遍历参数对象
  for (var key in params) {
    if (key != 'url') {
      //判断是否存在?
      // 如果没有前面带?
      // 如果有前面带&
      let s = baseURL.includes('?') ? '&' : '?'
      baseURL += `${s+key}=${params[key]}`
    }
  }
  //打开连接
  xhr.open('get', baseURL)
  //发送请求
  xhr.send()
  //返回一个新的promise
  return new Promise((resolve, reject) => {
    //监听事件
    xhr.onreadystatechange = () => {
      if (xhr.readyState == 4 && /^2\d{2}$/.test(xhr.status)) {
        //调用resolve传递数据 给then
        resolve(JSON.parse(xhr.responseText))
      }
    }
  })
```

```
}
```

post请求

基础post请求

```
//创建xhr
var xhr = new XMLHttpRequest()
//打开连接
xhr.open('post', 'https://jsonplaceholder.typicode.com/posts')
//设置请求头 告诉服务器发送的数据是表单 x-www-form-urlencoded 表示发送的表单数据
xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded')
//发送请求 请求体 name=张三&age=18
xhr.send("name=张三&age=18")
//监听
xhr.onreadystatechange = () => {
    //接收数据处理
    console.log(xhr.responseText)
}
```

- post请求数据发送使用send函数携带 (post请求携带到对应的请求体中)
- 在发送之前要指定对应的请求头 content-type： application/x-www-form-urlencoded （指定数据类型为表单)

回调函数的封装

```
//封装post请求的方法
function post(params = {}, callback) {
  //判断是否传入url
  if (!params.url) {
    throw new Error('必须传入url地址')
  }
  //创建xhr对象
  var xhr = new XMLHttpRequest()
  //参数和baseUrl进行拼接
  let baseURL = params.url
  let paramString = ""
  //遍历参数对象
  for (var key in params) {
    if (key != 'url') {
      let s = paramString?'&':""
      paramString += `${s+key}=${params[key]}`
    }
  }
  //打开连接
  xhr.open('post', baseURL)
  //发送请求
  //设置请求头
  xhr.setRequestHeader('content-type','application/x-www-form-urlencoded')
  xhr.send(paramString)
  //监听事件
  xhr.onreadystatechange = () => {
    if (xhr.readyState == 4 && /^2\d{2}$/.test(xhr.status)) {
      //调用回调函数 进行相关操作 传入对应的返回的数据
      callback(JSON.parse(xhr.responseText))
    }
```

```
  }
}
```

promise的封装

```
//封装post请求的方法
export function post(params = {}) {
  //判断是否传入url
  if (!params.url) {
    throw new Error('必须传入url地址')
  }
  //创建xhr对象
  var xhr = new XMLHttpRequest()
  //参数和baseUrl进行拼接
  let baseURL = params.url
  let paramString = ""
  //遍历参数对象
  for (var key in params) {
    if (key != 'url') {
      let s = paramString?'&':""
      paramString += `${s+key}=${params[key]}`
    }
  }
  //打开连接
  xhr.open('post', baseURL)
  //发送请求
  //设置请求头
  xhr.setRequestHeader('content-type','application/x-www-form-urlencoded')
  xhr.send(paramString)
  //监听事件
  return new Promise((resolve,reject)=>{
    xhr.onreadystatechange = () => {
      if (xhr.readyState == 4 && /^2\d{2}$/.test(xhr.status)) {
        //调用回调函数 进行相关操作 传入对应的返回的数据
        resolve(JSON.parse(xhr.responseText))
      }
    }
  })
}
```

get请求和post请求的联合封装

```
export default class Ajax {
  //传入的是根路径 传入超时时间
  constructor(url, time) {
    this.baseURL = url
    this.timeout = time
  }
  //创建一个请求对象的方法
  static createRequest(option) {
    let defaultOption = {
      baseURL: "",
      timeout: 3000
    }
    //根据对应的option来进行比对 如果option有就使用option的内容来替换对应的defaultOption
    for (var key in option) {
```

```
      defaultOption[key] = option[key]
    }
    //读取对应的配置 返回一个ajax对象
    return new Ajax(defaultOption.baseURL, defaultOption.timeout)
  }
  //准备对应的request方法 {url,contentType,method,data}
  request(requestConfig) {
    //method要传递 url也要传递 data也要传递
    if (requestConfig.url == undefined ||
      (requestConfig.method.toLowerCase() != 'get' &&
requestConfig.method.toLowerCase() != 'post') ||
      !requestConfig.data
    ) {
      throw new Error()
    }
    //默认就是以表单提交
    requestConfig.contentType = requestConfig.contentType ?
requestConfig.contentType : 'application/x-www-form-urlencoded'
    //准备xhr对象
    let xhr = new XMLHttpRequest()
    //设置对应的timeout
    xhr.timeout = this.timeout
    //url
    let url = this.baseURL + requestConfig.url
    //get数据的拼接 拼接到url
    if (requestConfig.method.toLowerCase() == 'get') {
      for (var key in requestConfig.data) {
        if (url.includes('?')) {
          url += `&${key}=${requestConfig.data[key]}`
        } else {
          url += `?${key}=${requestConfig.data[key]}`
        }
      }
    }
    //打开连接
    xhr.open(requestConfig.method, url)
    //设置请求头
    if (requestConfig.method.toLowerCase() == 'post') {
      xhr.setRequestHeader('content-type', requestConfig.contentType)
    }
    //post数据拼接
    if (requestConfig.method.toLowerCase() == 'post') {
      var paramsString = ""
      for (var key in requestConfig.data) {
        if (!paramsString) {
          paramsString += `${key}=${requestConfig.data[key]}`
        } else {
          paramsString += `&${key}=${requestConfig.data[key]}`
        }
      }
    }
    //发送数据
    xhr.send(paramsString)
    //监听
    return new Promise((resolve, reject) => {
      xhr.onreadystatechange = () => {
        //判断
        if(xhr.readyState == 4 && /^2\d{2}$/.test(xhr.status)){
```

```
            resolve(JSON.parse(xhr.responseText))
          }
        }
      })
    }
  }
```