Axios基本概念

Axios是一个基于promise网络请求库，作用于node.js和浏览器中。它是isomorphic的（即同一套代码可以运行在浏览器和node.js中）。在服务端它使用原生node.js http模块，而在客户端（浏览器）则使用XMLHttpRequests。

特性

* 从浏览器创建 [XMLHttpRequests](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest)
* 从 node.js 创建 [http](http://nodejs.org/api/http.html) 请求
* 支持 [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) API
* 拦截请求和响应
* 转换请求和响应数据
* 取消请求
* 超时处理
* 查询参数序列化支持嵌套项处理
* 自动将请求体序列化为：
  + JSON (application/json)
  + Multipart / FormData (multipart/form-data)
  + URL encoded form (application/x-www-form-urlencoded)
* 将 HTML Form 转换成 JSON 进行请求
* 自动转换JSON数据
* 获取浏览器和 node.js 的请求进度，并提供额外的信息（速度、剩余时间）
* 为 node.js 设置带宽限制
* 兼容符合规范的 FormData 和 Blob（包括 node.js）
* 客户端支持防御[XSRF](http://en.wikipedia.org/wiki/Cross-site_request_forgery)

安装

* 使用npm:

$ npm install axios

基本用例

用例

发送一个GET请求

const axios =require('axios');

//向给定ID的用户发送请求

axios.get('/user?ID=12345').then(function(response){

    //处理成功的情况

    console.log(response);

})

.catch(function(error){

    //处理错误的情况

    console.log(error);

})

.finally(function(){

    //总会执行

});

axios.get('/user',{

    params:{

        ID:12345

    }

}).then(function(response){

    console.log(response);

})

.catch(function(error){

    console.log(error);

})

.finally(function(){

});

//支持async和await用法

async function getUser() {

    try{

        const response =await axios.get('/user?ID=12345');

        console.log(response);

    }

    catch(error){

        console.log(error);

    }

}

POST请求

发起一个POST请求

axios.post('/user',{

    firstName:'Jet',

    lastName:'Chet'

})

.then(function(response){

    console.log(response);

})

.catch(function(error){

    console.log(error);

})

发起多个并发请求

//定义异步函数通过get请求向指定的URL发送HTTP请求，并返回一个Promise

function getUserAccount(){

    return axios.get('/user/12345');

}

function getUserPermission(){

    return axios.get('/user/12345/permission');

}

//使用解构赋值当所有Promise都fulfil时，返回数组。await等待所有Promise完成，暂停执行，直到结果返回。

const[acct,perm]=await Promise.all([getUserAccount(),getUserPermission]);

//或使用.then的方法

Promise.all([getUserAccount(),getUserPermission()])

.then(function([acct,perm]){

    //...

});

将HTML Form转化成JSON进行请求

const{data}=await axios.post('/user',document.querySelector('#my-form'),

{

    headers:{

        'Content-Type':'application/json'

    }

})

Forms

* Multipart(multipart/form-data)

const {data} = await axios.post('https://httpbin.org/post', {

firstName: 'Fred',

lastName: 'Flintstone',

orders: [1, 2, 3],

photo: document.querySelector('#fileInput').files

}, {

headers: {

'Content-Type': 'multipart/form-data'

}

}

)

* URL encoded form(application/x-www-form-urlencoded)

const {data} = await axios.post('https://httpbin.org/post', {

firstName: 'Fred',

lastName: 'Flintstone',

orders: [1, 2, 3]

}, {

headers: {

'Content-Type': 'application/x-www-form-urlencoded'

}

})

Axios API

可以向axios传递相关配置来创建请求

axios(config)

//发起一个POST请求

axios({

    method:'post',

    url:'/user/12345',

    data:{

        firstName:'Jet',

        lastName:'Chet'

    }

});

//在node.js用GET请求获取远程图片

axios({

    method:'get',

    url:'http://bit.ly/2mTM3nY',

    responseType:'stream'

})

.then(function(response){

    response.data.pipe(fs.createWriteStream('ada\_lovelace.jpg'))

});

axios(url[,config])

// 发起一个 GET 请求 (默认请求方式)

axios('/user/12345');

Axios实例

创建一个实例

axios.create([config])

const instance =axios.create({

    baseURL:'https://some-domain.com/api/',

    timeout:1000,

    headers:{'X-Custom-Header':'footbar'}

});

请求配置

这些是创建请求时可以用的配置选项。至于url是必需的。如果没有指定method，请求将默认使用GET方法。

{

// `url` 是用于请求的服务器 URL

url: '/user',

// `method` 是创建请求时使用的方法

method: 'get', // 默认值

// `baseURL` 将自动加在 `url` 前面，除非 `url` 是一个绝对 URL。

// 它可以通过设置一个 `baseURL` 便于为 axios 实例的方法传递相对 URL

baseURL: 'https://some-domain.com/api/',

// `transformRequest` 允许在向服务器发送前，修改请求数据

// 它只能用于 'PUT', 'POST' 和 'PATCH' 这几个请求方法

// 数组中最后一个函数必须返回一个字符串， 一个Buffer实例，ArrayBuffer，FormData，或 Stream

// 你可以修改请求头。

transformRequest: [function (data, headers) {

// 对发送的 data 进行任意转换处理

return data;

}],

// `transformResponse` 在传递给 then/catch 前，允许修改响应数据

transformResponse: [function (data) {

// 对接收的 data 进行任意转换处理

return data;

}],

// 自定义请求头

headers: {'X-Requested-With': 'XMLHttpRequest'},

// `params` 是与请求一起发送的 URL 参数

// 必须是一个简单对象或 URLSearchParams 对象

params: {

ID: 12345

},

// `paramsSerializer`是可选方法，主要用于序列化`params`

// (e.g. https://www.npmjs.com/package/qs, http://api.jquery.com/jquery.param/)

paramsSerializer: function (params) {

return Qs.stringify(params, {arrayFormat: 'brackets'})

},

// `data` 是作为请求体被发送的数据

// 仅适用 'PUT', 'POST', 'DELETE 和 'PATCH' 请求方法

// 在没有设置 `transformRequest` 时，则必须是以下类型之一:

// - string, plain object, ArrayBuffer, ArrayBufferView, URLSearchParams

// - 浏览器专属: FormData, File, Blob

// - Node 专属: Stream, Buffer

data: {

firstName: 'Fred'

},

// 发送请求体数据的可选语法

// 请求方式 post

// 只有 value 会被发送，key 则不会

data: 'Country=Brasil&City=Belo Horizonte',

// `timeout` 指定请求超时的毫秒数。

// 如果请求时间超过 `timeout` 的值，则请求会被中断

timeout: 1000, // 默认值是 `0` (永不超时)

// `withCredentials` 表示跨域请求时是否需要使用凭证

withCredentials: false, // default

// `adapter` 允许自定义处理请求，这使测试更加容易。

// 返回一个 promise 并提供一个有效的响应 （参见 lib/adapters/README.md）。

adapter: function (config) {

/\* ... \*/

},

// `auth` HTTP Basic Auth

auth: {

username: 'janedoe',

password: 's00pers3cret'

},

// `responseType` 表示浏览器将要响应的数据类型

// 选项包括: 'arraybuffer', 'document', 'json', 'text', 'stream'

// 浏览器专属：'blob'

responseType: 'json', // 默认值

// `responseEncoding` 表示用于解码响应的编码 (Node.js 专属)

// 注意：忽略 `responseType` 的值为 'stream'，或者是客户端请求

// Note: Ignored for `responseType` of 'stream' or client-side requests

responseEncoding: 'utf8', // 默认值

// `xsrfCookieName` 是 xsrf token 的值，被用作 cookie 的名称

xsrfCookieName: 'XSRF-TOKEN', // 默认值

// `xsrfHeaderName` 是带有 xsrf token 值的http 请求头名称

xsrfHeaderName: 'X-XSRF-TOKEN', // 默认值

// `onUploadProgress` 允许为上传处理进度事件

// 浏览器专属

onUploadProgress: function (progressEvent) {

// 处理原生进度事件

},

// `onDownloadProgress` 允许为下载处理进度事件

// 浏览器专属

onDownloadProgress: function (progressEvent) {

// 处理原生进度事件

},

// `maxContentLength` 定义了node.js中允许的HTTP响应内容的最大字节数

maxContentLength: 2000,

// `maxBodyLength`（仅Node）定义允许的http请求内容的最大字节数

maxBodyLength: 2000,

// `validateStatus` 定义了对于给定的 HTTP状态码是 resolve 还是 reject promise。

// 如果 `validateStatus` 返回 `true` (或者设置为 `null` 或 `undefined`)，

// 则promise 将会 resolved，否则是 rejected。

validateStatus: function (status) {

return status >= 200 && status < 300; // 默认值

},

// `maxRedirects` 定义了在node.js中要遵循的最大重定向数。

// 如果设置为0，则不会进行重定向

maxRedirects: 5, // 默认值

// `socketPath` 定义了在node.js中使用的UNIX套接字。

// e.g. '/var/run/docker.sock' 发送请求到 docker 守护进程。

// 只能指定 `socketPath` 或 `proxy` 。

// 若都指定，这使用 `socketPath` 。

socketPath: null, // default

// `httpAgent` and `httpsAgent` define a custom agent to be used when performing http

// and https requests, respectively, in node.js. This allows options to be added like

// `keepAlive` that are not enabled by default.

httpAgent: new http.Agent({ keepAlive: true }),

httpsAgent: new https.Agent({ keepAlive: true }),

// `proxy` 定义了代理服务器的主机名，端口和协议。

// 您可以使用常规的`http\_proxy` 和 `https\_proxy` 环境变量。

// 使用 `false` 可以禁用代理功能，同时环境变量也会被忽略。

// `auth`表示应使用HTTP Basic auth连接到代理，并且提供凭据。

// 这将设置一个 `Proxy-Authorization` 请求头，它会覆盖 `headers` 中已存在的自定义 `Proxy-Authorization` 请求头。

// 如果代理服务器使用 HTTPS，则必须设置 protocol 为`https`

proxy: {

protocol: 'https',

host: '127.0.0.1',

port: 9000,

auth: {

username: 'mikeymike',

password: 'rapunz3l'

}

},

// see https://axios-http.com/zh/docs/cancellation

cancelToken: new CancelToken(function (cancel) {

}),

// `decompress` indicates whether or not the response body should be decompressed

// automatically. If set to `true` will also remove the 'content-encoding' header

// from the responses objects of all decompressed responses

// - Node only (XHR cannot turn off decompression)

decompress: true // 默认值

}

响应结构

一个请求的响应包含以下信息

{

// `data` 由服务器提供的响应

data: {},

// `status` 来自服务器响应的 HTTP 状态码

status: 200,

// `statusText` 来自服务器响应的 HTTP 状态信息

statusText: 'OK',

// `headers` 是服务器响应头

// 所有的 header 名称都是小写，而且可以使用方括号语法访问

// 例如: `response.headers['content-type']`

headers: {},

// `config` 是 `axios` 请求的配置信息

config: {},

// `request` 是生成此响应的请求

// 在node.js中它是最后一个ClientRequest实例 (in redirects)，

// 在浏览器中则是 XMLHttpRequest 实例

request: {}

}

当使用then时，将接收如下响应：

axios.get('/user/12345')

.then(function (response) {

console.log(response.data);

console.log(response.status);

console.log(response.statusText);

console.log(response.headers);

console.log(response.config);

});

当使用catch。或者传递一个rejection callback作为then的第二个参数时，响应可以通过error对象被使用。

默认配置

默认配置将作用于每个请求

全局axios默认值

axios.defaults.baseURL='https://api.example.com';

axios.defaults.headers.common['Authorization']=AUTH\_TOKEN;

axios.defaults.headers.post['Content-Type']='application/x-www-form-urlencoded';

自定义实例默认值

//创建实例时配置默认值

const instance =axios.create({

    baseURL:'https://api.example.com'

});

//创建实例后修改默认值

instance.defaults.headers.common['Authorization']=AUTH\_TOKEN;

配置的优先级

配置将会按优先级进行合并。它的顺序是：在lib/defaults.js中找到的库的默认值，然后实例的defaults属性，最后是请求的config参数。后面的优先级要高于前面的。

//使用库提供的默认配置创建实例

//此时超时配置的默认值是`0`

const instance = axios.create();

//重写库的超时默认值

//现在，所有使用此实例的请求都将等待2.5秒，然后才会超时

instance.defaults.timeout=2500;

//重写此请求的超时时间，因为该请求需要很长时间

instance.get('/longRequest',{

    timeout:5000

});

拦截器

在请求或响应被then或catch处理前拦截它们。

// 添加请求拦截器

axios.interceptors.request.use(function (config) {

// 在发送请求之前做些什么

return config;

}, function (error) {

// 对请求错误做些什么

return Promise.reject(error);

});

// 添加响应拦截器

axios.interceptors.response.use(function (response) {

// 2xx 范围内的状态码都会触发该函数。

// 对响应数据做点什么

return response;

}, function (error) {

// 超出 2xx 范围的状态码都会触发该函数。

// 对响应错误做点什么

return Promise.reject(error);

});

如果稍后需要移除拦截器，可以这样：

const myInterceptor = axios.interceptors.request.use(function () {/\*...\*/});

axios.interceptors.request.eject(myInterceptor);

可以给自定义的axios实例添加拦截器。

const instance = axios.create();

instance.interceptors.request.use(function () {/\*...\*/});

错误处理

axios.get('/user/12345')

.catch(function (error) {

if (error.response) {

// 请求成功发出且服务器也响应了状态码，但状态代码超出了 2xx 的范围

console.log(error.response.data);

console.log(error.response.status);

console.log(error.response.headers);

} else if (error.request) {

// 请求已经成功发起，但没有收到响应

// `error.request` 在浏览器中是 XMLHttpRequest 的实例，

// 而在node.js中是 http.ClientRequest 的实例

console.log(error.request);

} else {

// 发送请求时出了点问题

console.log('Error', error.message);

}

console.log(error.config);

});

使用validateStatus配置选项，可以自定义抛出错误的HTTP code。

axios.get('/user/12345', {

validateStatus: function (status) {

return status < 500; // 处理状态码小于500的情况

}

})

使用toJSON可以获取更多关于HTTP错误的信息。

axios.get('/user/12345')

.catch(function (error) {

console.log(error.toJSON());

});

取消请求

AbortController

const controller = new AbortController();

axios.get('/foo/bar', {

signal: controller.signal

}).then(function(response) {

//...

});

// 取消请求

controller.abort()；

CancelToken deprecated

可以使用CancelToken.source工厂方法创建一个cancel token

const CancelToken = axios.CancelToken;

const source = CancelToken.source();

axios.get('/user/12345', {

cancelToken: source.token

}).catch(function (thrown) {

if (axios.isCancel(thrown)) {

console.log('Request canceled', thrown.message);

} else {

// 处理错误

}

});

axios.post('/user/12345', {

name: 'new name'

}, {

cancelToken: source.token

})

// 取消请求（message 参数是可选的）

source.cancel('Operation canceled by the user.');

也可以通过传递一个executor函数到CancelToken的构造函数来创建一个cancel token：

const CancelToken = axios.CancelToken;

let cancel;

axios.get('/user/12345', {

cancelToken: new CancelToken(function executor(c) {

// executor 函数接收一个 cancel 函数作为参数

cancel = c;

})

});

// 取消请求

cancel();

在过渡期间，可以使用两种取消API，即使是针对同一请求：

const controller = new AbortController();

const CancelToken = axios.CancelToken;

const source = CancelToken.source();

axios.get('/user/12345', {

cancelToken: source.token,

signal: controller.signal

}).catch(function (thrown) {

if (axios.isCancel(thrown)) {

console.log('Request canceled', thrown.message);

} else {

// 处理错误

}

});

axios.post('/user/12345', {

name: 'new name'

}, {

cancelToken: source.token

})

// 取消请求 (message 参数是可选的)

source.cancel('Operation canceled by the user.');

// 或

controller.abort(); // 不支持 message 参数