Axios

本文参考 axios 官方文档,顺序有变动(我是搬运工小酷)

什么是 axios ?

- axios 是一个基于 Promise 的 HTTP 库,可以用在浏览器和 nodejs 中

特性

- 从浏览器中创建 XMLHttpRequests
- 从 nodejs 中创建 http 请求
- 支持 Promise API
- 拦截请求和响应
- 转换请求数据和响应数据
- 取消请求
- 自动转换 JSON 数据
- 客户端支持防御 XSRF

浏览器支持

					
Latest ✓	Latest ✓	Latest ✓	Latest ✓	Latest ✓	8+ ✓

Browser Matrix

安装

- 使用 npm

```
npm install axios
```

- 使用 bower

```
bower install axios
```

- 使用 cdn

```
<script src="https://unpkg.com/axios/dist/axios.min.js"> </script>
```

axios API

可以通过向 axios 传递相关配置来创建请求
axios(config)

```
// 发送 POST 请求
axios({
  method: "post",
  url: "/user/12345",
  data: {
    firstName: "Fred",
    lastName: "Flintstone"
  }
});
```

```
// 获取远端图片
axios({
  method: "get",
  url: "http://bit.ly/2mTM3nY",
  responseType: "stream"
}).then(function(response) {
  response.data.pipe(fs.createWriteStream("ada_lovelace.jpg"));
});
```

axios(url[, config])

```
// 发送 GET 请求（默认的方法）
axios("/user/12345");
```

创建实例

可以使用自定义配置新建一个 axios 实例

axios.create([config])

```
const instance = axios.create({
  baseURL: "https://some-domain.com/api/",
  timeout: 1000,
  headers: { "X-Custom-Header": "foobar" }
});
```

实例方法

以下是可用的实例方法。指定的配置将与实例的配置合并。

axios#request(config)
axios#get(url[, config])
axios#delete(url[, config])
axios#head(url[, config])
axios#options(url[, config])
axios#post(url[, data[, config]])
axios#put(url[, data[, config]])
axios#patch(url[, data[, config]])

请求配置

这些是创建请求时可以用的配置选项。只有 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', // default

  // `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
  // 只能用在 'PUT', 'POST' 和 'PATCH' 这几个请求方法
  // 后面数组中的函数必须返回一个字符串，或 ArrayBuffer，或 Stream
  transformRequest: [function (data, headers) {
    // 对 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 data 进行任意转换处理
    return data;
  }],

  // `headers` 是即将被发送的自定义请求头
  headers: {'X-Requested-With': 'XMLHttpRequest'},

  // `params` 是即将与请求一起发送的 URL 参数
  // 必须是一个无格式对象(plain object)或 URLSearchParams 对象
  params: {
    ID: 12345
  },

   // `paramsSerializer` 是一个负责 `params` 序列化的函数
  // (e.g. https://www.npmjs.com/package/qs, http://api.jquery.com/jquery.param/)
  paramsSerializer: function(params) {
    return Qs.stringify(params, {arrayFormat: 'brackets'})
  },

  // `data` 是作为请求主体被发送的数据
  // 只适用于这些请求方法 'PUT', 'POST', 和 'PATCH'
  // 在没有设置 `transformRequest` 时，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
  data: {
    firstName: 'Fred'
  },

  // `timeout` 指定请求超时的毫秒数(0 表示无超时时间)
  // 如果请求话费了超过 `timeout` 的时间，请求将被中断
  timeout: 1000,

   // `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，以使测试更轻松
  // 返回一个 promise 并应用一个有效的响应 (查阅 [response docs](#response-api)).
  adapter: function (config) {
    /* ... */
  },

 // `auth` 表示应该使用 HTTP 基础验证，并提供凭据
  // 这将设置一个 `Authorization` 头，覆写掉现有的任意使用 `headers` 设置的自定义
`Authorization`头
  auth: {
    username: 'janedoe',
    password: 's00pers3cret'
```

```
  },

   // `responseType` 表示服务器响应的数据类型，可以是 'arraybuffer', 'blob', 'document', 'json', 'text', 'stream'
  responseType: 'json', // default

  // `responseEncoding` indicates encoding to use for decoding responses
  // Note: Ignored for `responseType` of 'stream' or client-side requests
  responseEncoding: 'utf8', // default

   // `xsrfCookieName` 是用作 xsrf token 的值的cookie的名称
  xsrfCookieName: 'XSRF-TOKEN', // default

  // `xsrfHeaderName` is the name of the http header that carries the xsrf token value
  xsrfHeaderName: 'X-XSRF-TOKEN', // default

   // `onUploadProgress` 允许为上传处理进度事件
  onUploadProgress: function (progressEvent) {
    // Do whatever you want with the native progress event
  },

  // `onDownloadProgress` 允许为下载处理进度事件
  onDownloadProgress: function (progressEvent) {
    // 对原生进度事件的处理
  },

   // `maxContentLength` 定义允许的响应内容的最大尺寸
  maxContentLength: 2000,

  // `validateStatus` 定义对于给定的HTTP 响应状态码是 resolve 或 reject  promise 。如果 `validateStatus` 返回 `true` (或者设置为 `null` 或 `undefined`)，promise 将被 resolve; 否则，promise 将被 rejecte
  validateStatus: function (status) {
    return status >= 200 && status < 300; // default
  },

  // `maxRedirects` 定义在 node.js 中 follow 的最大重定向数目
  // 如果设置为0，将不会 follow 任何重定向
  maxRedirects: 5, // default

  // `socketPath` defines a UNIX Socket to be used in node.js.
  // e.g. '/var/run/docker.sock' to send requests to the docker daemon.
  // Only either `socketPath` or `proxy` can be specified.
  // If both are specified, `socketPath` is used.
  socketPath: null, // default

  // `httpAgent` 和 `httpsAgent` 分别在 node.js 中用于定义在执行 http 和 https 时使用的自定义代理。允许像这样配置选项：
  // `keepAlive` 默认没有启用
```

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

  // 'proxy' 定义代理服务器的主机名称和端口
  // `auth` 表示 HTTP 基础验证应当用于连接代理，并提供凭据
  // 这将会设置一个 `Proxy-Authorization` 头，覆写掉已有的通过使用 `header` 设置的自定义
`Proxy-Authorization` 头。
  proxy: {
    host: '127.0.0.1',
    port: 9000,
    auth: {
      username: 'mikeymike',
      password: 'rapunz3l'
    }
  },

  // `cancelToken` 指定用于取消请求的 cancel token
  // （查看后面的 Cancellation 这节了解更多）
  cancelToken: new CancelToken(function (cancel) {
  })
}
```

响应结构

某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` 服务器响应的头
  headers: {},

   // `config` 是为请求提供的配置信息
  config: {},
 // 'request'
  // `request` is the request that generated this response
  // It is the last ClientRequest instance in node.js (in redirects)
  // and an XMLHttpRequest instance the browser
  request: {}
}
```

使用 then 时，你将接收下面这样的响应：

```
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);
});
```

在使用 catch 时，或传递 rejection callback 作为 then 的第二个参数时，响应可以通过 error 对象可被使用，正如在错误处理这一节所讲

配置默认值

> *你可以指定将被用在各个请求的配置默认值*

- **全局的 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**

```
// Set config defaults when creating the instance
const instance = axios.create({
  baseUrl: "https://api.example.com"
});
// Alter defaults after instance has been created
instance.defaults.headers.common["Authorization"] = AUTH_TOKEN;
```

配置的优先顺序

> *配置会以一个优先顺序进行合并。这个顺序是：在 lib/defaults.js 找到的库的默认值，然后是实例的 defaults 属性，最后是请求的 config 参数。后者将优先于前*

者。这里是一个例子：

```
// 使用由库提供的配置的默认值来创建实例
// 此时超时配置的默认值是 `0`
var instance = axios.create();

// 覆写库的超时默认值
// 现在，在超时前，所有请求都会等待 2.5 秒
instance.defaults.timeout = 2500;

// 为已知需要花费很长时间的请求覆写超时设置
instance.get("/longRequest", {
  timeout: 5000
});
```

拦截器

在请求被 then 或 catch 处理前拦截它们

```
// 添加请求拦截器
axios.interceptors.request.use(
  function(config) {
    // 在发送请求之前做点什么
    return config;
  },
  function(err) {
    // 对请求错误做些什么
    return Promise.reject(err);
  }
);

// 添加响应拦截器
axios.interceptors.response.use(
  function(response) {
    // 对响应数据做点什么
    return response;
  },
  function(err) {
    // 对响应错误做点什么
    return Promise.reject(err);
  }
);
```

如果你想在稍后移除拦截器，可以这样：

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

可以为自定义 axios 实例添加拦截器

```
const instance = axios.create();
instance.interceptors.request.use(function() {
  /* ... */
});
```

案例

执行 GET 请求

```
// 为给定 ID 的 user 创建请求
axios
  .get("/user?ID=12345")
  .then(function(response) {
    console.log(response);
  })
  .catch(function(error) {
    console.log(error);
  });

// 上面的请求也可以这样做
axios
  .get("/user", {
    params: {
      ID: 12345
    }
  })
  .then(function(response) {
    console.log(response);
  })
  .catch(function(error) {
    console.log(error);
  });
```

执行 POST 请求

```
axios
  .post("/user", {
    firstName: "Fred",
    lastName: "Flintstone"
  })
  .then(function(response) {
    console.log(response);
  })
  .catch(function(error) {
    console.log(error);
  });
```

执行多个并发请求

```
function getUserAccount() {
  return axios.get("/user/12345");
}

function getUserPermissions() {
  return axios.get("/user/12345/permissions");
}

axios.all([getUserAccount(), getUserPermissions()]).then(
  axios.spread(function(acct, perms) {
    // 两个请求现在都执行完成
  })
);
```

并发

处理并发请求的助手函数

```
axios.all(iterable);
axios.spread(callback);
```

错误处理

```
axios.get("/user/12345").catch(function(error) {
  if (error.response) {
```

```
    // The request was made and the server responded with a status code
    // that falls out of the range of 2xx
    console.log(error.response.data);
    console.log(error.response.status);
    console.log(error.response.headers);
  } else if (error.request) {
    // The request was made but no response was received
    // `error.request` is an instance of XMLHttpRequest in the browser and an instance
of
    // http.ClientRequest in node.js
    console.log(error.request);
  } else {
    // Something happened in setting up the request that triggered an Error
    console.log("Error", error.message);
  }
  console.log(error.config);
});
```

可以使用 validateStatus 配置选项定义一个自定义 HTTP 状态码的错误范围。

```
axios.get("/user/12345", {
  validateStatus: function(status) {
    return status < 500; // Reject only if the status code is greater than or equal to
500
  }
});
```