一文读懂Axios核心源码思想

Vue中文社区 1周前

阅读完本文，下面的问题会迎刃而解，

- Axios 的适配器原理是什么?
- Axios 是如何实现请求和响应拦截的?
- Axios 取消请求的实现原理?
- CSRF 的原理是什么？Axios 是如何防范客户端 CSRF 攻击?
- 请求和响应数据转换是怎么实现的?

> 全文约两千字，阅读完大约需要 6 分钟，文中 Axios 版本为 0.21.1

我们以特性作为入口，解答上述问题的同时一起感受下 Axios 源码极简封装的艺术。

Features

- 从浏览器创建 XMLHttpRequest
- 从 Node.js 创建 HTTP 请求
- 支持 Promise API
- 拦截请求与响应
- 取消请求
- 自动装换 JSON 数据
- 支持客户端 XSRF 攻击

前两个特性解释了为什么 Axios 可以同时用于浏览器和 Node.js 的原因，简单来说就是通过判断是服务器还是浏览器环境，来决定使用 `XMLHttpRequest` 还是 Node.js 的 HTTP 来创建请求，这个兼容的逻辑被叫做适配器，对应的源码在 `lib/defaults.js` 中，

```
// defaults.js
function getDefaultAdapter() {
  var adapter;
  if (typeof XMLHttpRequest !== 'undefined') {
    // For browsers use XHR adapter
    adapter = require('./adapters/xhr');
  } else if (typeof process !== 'undefined' && Object.prototype.toString.call(process) === '[object process]') {
    // For node use HTTP adapter
    adapter = require('./adapters/http');
  }
  return adapter;
}
```

以上是适配器的判断逻辑，通过侦测当前环境的一些全局变量，决定使用哪个 adapter。其中对于 Node 环境的判断逻辑在我们做 `ssr` 服务端渲染的时候，也可以复用。接下来我们来看一下 Axios 对于适配器的封装。

Adapter xhr

定位到源码文件 `lib/adapters/xhr.js`，先来看下整体结构，

```
module.exports = function xhrAdapter(config) {
```

```
  return new Promise(function dispatchXhrRequest(resolve, reject) {
    // ...
  })
}
```

导出了一个函数，接受一个配置参数，返回一个 Promise。我们把关键的部分提取出来，

```
module.exports = function xhrAdapter(config) {
  return new Promise(function dispatchXhrRequest(resolve, reject) {
    var requestData = config.data;

    var request = new XMLHttpRequest();

    request.open(config.method.toUpperCase(), buildURL(fullPath, config.params, config.paramsSerializer), true);
    request.onreadystatechange = function handleLoad() {}
    request.onabort = function handleAbort() {}
    request.onerror = function handleError() {}
    request.ontimeout = function handleTimeout() {}

    request.send(requestData);
  });
};
```

是不是感觉很熟悉？没错，这就是 `XMLHttpRequest` 的使用姿势呀，先创建了一个 xhr 然后 open 启动请求，监听 xhr 状态，然后 send 发送请求。我们来展开看一下 Axios 对于 onreadystatechange 的处理，

```
request.onreadystatechange = function handleLoad() {
  if (!request || request.readyState !== 4) {
    return;
  }

  // The request errored out and we didn't get a response, this will be
  // handled by onerror instead
  // With one exception: request that using file: protocol, most browsers
  // will return status as 0 even though it's a successful request
  if (request.status === 0 && !(request.responseURL && request.responseURL.indexOf('file:') === 0)) {
    return;
  }

  // Prepare the response
  var responseHeaders = 'getAllResponseHeaders' in request ? parseHeaders(request.getAllResponseHeaders()) : null;
  var responseData = !config.responseType || config.responseType === 'text' ? request.responseText : request.response;
  var response = {
    data: responseData,
    status: request.status,
    statusText: request.statusText,
    headers: responseHeaders,
    config: config,
    request: request
  };

  settle(resolve, reject, response);

  // Clean up request
  request = null;
};
```

首先对状态进行过滤，只有当请求完成时（readyState === 4）才往下处理。需要注意的是，如果 XMLHttpRequest 请求出错，大部分的情况下我们可以通过监听 `onerror` 进行处理，但是也有一个例外：当请求使用文件协议（`file://`）时，尽管请求成功了但是大部分浏览器也会返回 `0` 的状态码。

Axios 针对这个例外情况也做了处理。

请求完成后，就要处理响应了。这里将响应包装成一个标准格式的对象，作为第三个参数传递给了 settle 方法，settle 在 `lib/core/settle.js` 中定义，

```
function settle(resolve, reject, response) {
  var validateStatus = response.config.validateStatus;
  if (!response.status || !validateStatus || validateStatus(response.status)) {
    resolve(response);
  } else {
    reject(createError(
      'Request failed with status code ' + response.status,
      response.config,
      null,
      response.request,
      response
    ));
  }
};
```

settle 对 Promise 的回调进行了简单的封装，确保调用按一定的格式返回。

以上就是 xhrAdapter 的主要逻辑，剩下的是对请求头，支持的一些配置项以及超时，出错，取消请求等回调的简单处理，其中对于 XSRF 攻击的防范是通过请求头实现的。

我们先来简单回顾下什么是 **XSRF** （也叫 **CSRF**，**跨站请求伪造**）。

CSRF

背景：用户登录后，需要存储登录凭证保持登录态，而不用每次请求都发送账号密码。

怎么样保持登录态呢?

目前比较常见的方式是，服务器在收到 HTTP请求后，在响应头里添加 Set-Cookie 选项，将凭证存储在 Cookie 中，浏览器接受到响应后会存储 Cookie，根据浏览器的同源策略，下次向服务器发起请求时，会自动携带 Cookie 配合服务端验证从而保持用户的登录态。

所以如果我们没有判断请求来源的合法性，在登录后通过其他网站向服务器发送了伪造的请求，这时携带登录凭证的 Cookie 就会随着伪造请求发送给服务器，导致安全漏洞，这就是我们说的 CSRF，跨站请求伪造。

所以防范伪造请求的关键就是检查请求来源，refferer 字段虽然可以标识当前站点，但是不够可靠，现在业界比较通用的解决方案还是在每个请求上附带一个 anti-CSRF token，这个的原理是攻击者无法拿到 Cookie，所以我们可以通过对 Cookie 进行加密（比如对 sid 进行加密），然后配合服务端做一些简单的验证，就可以判断当前请求是不是伪造的。

Axios 简单地实现了对特殊 csrf token 的支持，

```
// Add xsrf header
// This is only done if running in a standard browser environment.
// Specifically not if we're in a web worker, or react-native.
if (utils.isStandardBrowserEnv()) {
```

```
  // Add xsrf header
  var xsrfValue = (config.withCredentials || isURLSameOrigin(fullPath)) && config.xsrfCookieName ?
    cookies.read(config.xsrfCookieName) :
    undefined;

  if (xsrfValue) {
    requestHeaders[config.xsrfHeaderName] = xsrfValue;
  }
}
```

Interceptor

拦截器是 Axios 的一个特色 Feature，我们先简单回顾下使用方式，

```
// 拦截器可以拦截请求或响应
// 拦截器的回调将在请求或响应的 then 或 catch 回调前被调用
var instance = axios.create(options);

var requestInterceptor = axios.interceptors.request.use(
  (config) => {
    // do something before request is sent
    return config;
  },
  (err) => {
    // do somthing with request error
    return Promise.reject(err);
  }
);

// 移除已设置的拦截器
axios.interceptors.request.eject(requestInterceptor)
```

那么拦截器是怎么实现的呢?

定位到源码 `lib/core/Axios.js` 第 14 行，

```
function Axios(instanceConfig) {
  this.defaults = instanceConfig;
  this.interceptors = {
    request: new InterceptorManager(),
    response: new InterceptorManager()
  };
}
```

通过 Axios 的构造函数可以看到，拦截器 interceptors 中的 request 和 response 两者都是一个叫做 **InterceptorManager** 的实例，这个 InterceptorManager 是什么?

定位到源码 `lib/core/InterceptorManager.js`，

```
function InterceptorManager() {
  this.handlers = [];
}

InterceptorManager.prototype.use = function use(fulfilled, rejected, options) {
  this.handlers.push({
```

```
    fulfilled: fulfilled,
    rejected: rejected,
    synchronous: options ? options.synchronous : false,
    runWhen: options ? options.runWhen : null
  });
  return this.handlers.length - 1;
};

InterceptorManager.prototype.eject = function eject(id) {
  if (this.handlers[id]) {
    this.handlers[id] = null;
  }
};

InterceptorManager.prototype.forEach = function forEach(fn) {
  utils.forEach(this.handlers, function forEachHandler(h) {
    if (h !== null) {
      fn(h);
    }
  });
};
```

InterceptorManager 是一个简单的事件管理器，实现了对拦截器的管理，

通过 handlers 存储拦截器，然后提供了添加，移除，遍历执行拦截器的实例方法，存储的每一个拦截器对象都包含了作为 Promise 中 resolve 和 reject 的回调以及两个配置项。

值得一提的是，移除方法是通过**直接将拦截器对象设置为 null 实现的**，而不是 splice 剪切数组，遍历方法中也增加了相应的 null 值处理。这样做一方面使得每一项ID保持为项的数组索引不变，另一方面也避免了重新剪切拼接数组的性能损失。

拦截器的回调会在请求或响应的 then 或 catch 回调前被调用，这是怎么实现的呢?

回到源码 `lib/core/Axios.js` 中第 27 行，Axios 实例对象的 request 方法，

我们提取其中的关键逻辑如下，

```
Axios.prototype.request = function request(config) {
  // Get merged config
  // Set config.method
  // ...
  var requestInterceptorChain = [];
  this.interceptors.request.forEach(function unshiftRequestInterceptors(interceptor) {
    requestInterceptorChain.unshift(interceptor.fulfilled, interceptor.rejected);
  });

 var responseInterceptorChain = [];
  this.interceptors.response.forEach(function pushResponseInterceptors(interceptor) {
    responseInterceptorChain.push(interceptor.fulfilled, interceptor.rejected);
  });

  var promise;

  var chain = [dispatchRequest, undefined];

  Array.prototype.unshift.apply(chain, requestInterceptorChain);

  chain.concat(responseInterceptorChain);

  promise = Promise.resolve(config);
```

```
  while (chain.length) {
    promise = promise.then(chain.shift(), chain.shift());
  }

  return promise;
};
```

可以看到，当执行 request 时，实际的请求（dispatchRequest）和拦截器是通过一个叫 chain 的队列来管理的。整个请求的逻辑如下，

1. 首先初始化请求和响应的拦截器队列，将 resolve，reject 回调依次放入队头
2. 然后初始化一个 Promise 用来执行回调，chain 用来存储和管理实际请求和拦截器
3. 将请求拦截器放入 chain 队头，响应拦截器放入 chain 队尾
4. 队列不为空时，通过 Promise.then 的链式调用，依次将请求拦截器，实际请求，响应拦截器出队
5. 最后返回链式调用后的 Promise

这里的实际请求是对适配器的封装，请求和响应数据的转换都在这里完成。

那么数据转换是如何实现的呢?

Transform data

定位到源码 `lib/core/dispatchRequest.js`，

```
function dispatchRequest(config) {
  throwIfCancellationRequested(config);

  // Transform request data
  config.data = transformData(
    config.data,
    config.headers,
    config.transformRequest
  );

  var adapter = config.adapter || defaults.adapter;

  return adapter(config).then(function onAdapterResolution(response) {
    throwIfCancellationRequested(config);

    // Transform response data
    response.data = transformData(
      response.data,
      response.headers,
      config.transformResponse
    );

    return response;
  }, function onAdapterRejection(reason) {
    if (!isCancel(reason)) {
      throwIfCancellationRequested(config);

      // Transform response data
      if (reason && reason.response) {
        reason.response.data = transformData(
          reason.response.data,
          reason.response.headers,
          config.transformResponse
        );
      }
    }

    return Promise.reject(reason);
  });
```

```
};
```

这里的 throwIfCancellationRequested 方法用于取消请求，关于取消请求稍后我们再讨论，可以看到发送请求是通过调用适配器实现的，在调用前和调用后会对请求和响应数据进行转换。

转换通过 transformData 函数实现，它会遍历调用设置的转换函数，转换函数将 headers 作为第二个参数，所以我们可以根据 headers 中的信息来执行一些不同的转换操作，

```
// 源码 core/transformData.js
function transformData(data, headers, fns) {
  utils.forEach(fns, function transform(fn) {
    data = fn(data, headers);
  });

  return data;
};
```

Axios 也提供了两个默认的转换函数，用于对请求和响应数据进行转换。默认情况下，

Axios 会对请求传入的 data 做一些处理，比如请求数据如果是对象，会序列化为 JSON 字符串，响应数据如果是 JSON 字符串，会尝试转换为 JavaScript 对象，这些都是非常实用的功能，

对应的转换器源码可以在 `lib/default.js` 的第 31 行找到，

```
var defaults = {
 // Line 31
  transformRequest: [function transformRequest(data, headers) {
    normalizeHeaderName(headers, 'Accept');
    normalizeHeaderName(headers, 'Content-Type');
    if (utils.isFormData(data) ||
      utils.isArrayBuffer(data) ||
      utils.isBuffer(data) ||
      utils.isStream(data) ||
      utils.isFile(data) ||
      utils.isBlob(data)
    ) {
      return data;
    }
    if (utils.isArrayBufferView(data)) {
      return data.buffer;
    }
    if (utils.isURLSearchParams(data)) {
      setContentTypeIfUnset(headers, 'application/x-www-form-urlencoded;charset=utf-8');
      return data.toString();
    }
    if (utils.isObject(data)) {
      setContentTypeIfUnset(headers, 'application/json;charset=utf-8');
      return JSON.stringify(data);
    }
    return data;
  }],

  transformResponse: [function transformResponse(data) {
    var result = data;
    if (utils.isString(result) && result.length) {
      try {
        result = JSON.parse(result);
```

```
      } catch (e) { /* Ignore */ }
    }
    return result;
  }],
}
```

我们说 Axios 是支持取消请求的，怎么个取消法呢?

CancelToken

其实不管是浏览器端的 xhr 或 Node.js 里 http 模块的 request 对象，都提供了 abort 方法用于取消请求，所以我们只需要在合适的时机调用 abort 就可以实现取消请求了。

那么，什么是合适的时机呢? **控制权交给用户就合适了**。所以这个合适的时机应该由用户决定，也就是说我们需要将取消请求的方法暴露出去，Axios 通过 CancelToken 实现取消请求，我们来一起看下它的姿势。

首先 Axios 提供了两种方式创建 cancel token，

```
const CancelToken = axios.CancelToken;
const source = CancelToken.source();

// 方式一，使用 CancelToken 实例提供的静态属性 source
axios.post("/user/12345", { name: "monch" }, { cancelToken: source.token });
source.cancel();

// 方式二，使用 CancelToken 构造函数自己实例化
let cancel;

axios.post(
  "/user/12345",
  { name: "monch" },
  {
    cancelToken: new CancelToken(function executor(c) {
      cancel = c;
    }),
  }
);

cancel();
```

到底什么是 CancelToken? 定位到源码 `lib/cancel/CancelToken.js` 第 11 行，

```
function CancelToken(executor) {
  if (typeof executor !== "function") {
    throw new TypeError("executor must be a function.");
  }

  var resolvePromise;
  this.promise = new Promise(function promiseExecutor(resolve) {
    resolvePromise = resolve;
  });

  var token = this;
  executor(function cancel(message) {
    if (token.reason) {
      // Cancellation has already been requested
      return;
```

```
    }

    token.reason = new Cancel(message);
    resolvePromise(token.reason);
  });
}
```

CancelToken 就是一个由 promise 控制的极简的状态机，实例化时会在实例上挂载一个 promise，这个 promise 的 resolve 回调暴露给了外部方法 executor，这样一来，我们从外部调用这个 executor方法后就会得到一个状态变为 fulfilled 的 promise，那有了这个 promise 后我们如何取消请求呢?

是不是只要在请求时拿到这个 promise 实例，然后在 then 回调里取消请求就可以了?

定位到适配器的源码 `lib/adapters/xhr.js` 第 158 行,

```
if (config.cancelToken) {
  // Handle cancellation
  config.cancelToken.promise.then(function onCanceled(cancel) {
    if (!request) {
      return;
    }

    request.abort();
    reject(cancel);
    // Clean up request
    request = null;
  });
}
```

以及源码 `lib/adaptors/http.js` 第 291 行,

```
if (config.cancelToken) {
  // Handle cancellation
  config.cancelToken.promise.then(function onCanceled(cancel) {
    if (req.aborted) return;

    req.abort();
    reject(cancel);
  });
}
```

果然如此，在适配器里 CancelToken 实例的 promise 的 then 回调里调用了 xhr 或 http.request 的 abort 方法。试想一下，如果我们没有从外部调用取消 CancelToken 的方法，是不是意味着 resolve 回调不会执行，适配器里的 promise 的 then 回调也不会执行，就不会调用 abort 取消请求了。

小结

Axios 通过适配器的封装，使得它可以在保持同一套接口规范的前提下，同时用在浏览器和 node.js 中。源码中大量使用 Promise 和闭包等特性，实现了一系列的状态控制，其中对于拦截器，取消请求的实现体现了其极简的封装艺术，值得学习和借鉴。

参考链接

- **Axios Docs - axios-http.com**[1]
- **Axios Github Source Code**[2]

- 源码拾遗 Axios —— 极简封装的艺术[3]
- Cross Site Request Forgery - Part III. Web Application Security[4]
- tc39/proposal-cancelable-promises[5]

写在最后

本文首发于我的 博客[6]，才疏学浅，难免有错误，文章有误之处还望不吝指正！

如果有疑问或者发现错误，可以在相应的 issues 进行提问或勘误

如果喜欢或者有所启发，欢迎 star，对作者也是一种鼓励

（完）

> 作者：campcc
> https://github.com/campcc/blog/issues/23

推荐阅读：

50天用react.js重写50个web项目，我学到了什么?

前端潮咖