# axios封装

## 目录



## 背景

目前多个项目中封装的axios存在一些不合理性，比如对正确相应的处理，对正常错误（status === 'fail'）响应的处理，对'no-auth'或'40001'的处理等。这将导致我们在业务代码中写更多的逻辑判断和冗余代码，为了处理该问题，我们需要一套更加合理的axios的解决方案。

## sack的axios

</> TypeScript

```typescript
function isInvalidResponse(response: Response): boolean {
    const {data} = response;
    return !data
        || typeof data !== 'object'
        || !Object.keys(data).length
        || !!(data.status && !tupleResponseStatus.includes(data.status));
}

export function createAjax(config: AxiosRequestConfig) {
    // 创建实例
    const decoratedAjax = _createAjax(config);
    // 设置响应拦截器
    decoratedAjax.interceptors.response.use(
        response => {
            const data = (response.data || {}) as ResponseJSON;
            // status不为'ok'会存在三种情况
            // 1. fail: 此时一般情况需要提示错误
            // 2. no-auth: 没有权限，一般为回到登录页或者是其他处理（根据具体产品）
            // 3. 其他: 其他处理
            if (data.status !== 'ok') {
```

```
                toast.error(data.message || '请求失败');
            }
            if (isInvalidResponse(response)) {
                data.status = 'invalid-response';
            }
            // 返回结果
            return {
                ...data,
                response,
            } as unknown as AjaxInstance;
        },
        (error: AxiosError | Error) => {
            return {
                status: 'request-failed',
                error,
            };
        });

    return decoratedAjax;
}
```

目前Sack中的axios可能存在以下问题：

- status处理的情况覆盖不全，需要覆盖到至少三种情况
  a. fail: 此时一般情况需要提示错误
  b. no-auth: 没有权限，一般为回到登录页或者是其他处理（根据具体产品）
  c. 其他：其他处理
- 无论成功和错误都把结果返回，这会导致我们把最后的处理又交到了业务代码中去处理。

## 期望

我们期望的axios能做什么？

1. 可以对结果进行处理

</> TypeScript

```
// 业务代码
axios.get(url).then(res => {
    this.data = res
})
```

我们期望axios里面的代码只有如此简单，不需要判断status，不需要再编写报错的代码，只需要关注请求成功的处理。

前提就是，我们需要在response interceptor中处理好各个status中的问题。

2. 支持自定义处理

在实际的业务中，并不完全都是非0即1的情况。有些时候，我们需要自定义处理某个请求，比如某个请求在错误的时候不需要提示消息。

所以在一些特殊的情况，我们期望可以深入到内部去处理。

## 封装axios

在期望小节中，我们了解到可以通过interceptor处理响应

</> TypeScript

```typescript
export function createAjax(config: AxiosRequestConfig) {
     // 创建实例
    const decoratedAjax = _createAjax(config);
    // 设置响应拦截器
    decoratedAjax.interceptors.response.use(
        response => {
            const data = (response.data || {}) as ResponseJSON;
            // ok处理
            if (data.status === 'ok') {
                return {
                    ...data,
                    response
                }
            } else if (data.status === 'fail') {
                toast.error(data.message || '请求失败');
                // 把信息代码catch中，不要在then中处理
                return Promise.reject({
                    ...data,
                    response,
                })
            } else if (data.status === 'no-auth') {
                // some code
                return Promise.reject(...)
            } else {
                toast.error(data.message || '请求失败');
                return Promise.reject(...)
            }
        },
        (error: AxiosError | Error) => {
            return {
                status: 'request-failed',
                error,
            };
```

```
        });

    return decoratedAjax;
}
```

- 通过interceptor的处理，我们把status !== 'ok'的情况都过滤掉，这样在then中，我们就可以放心大胆的直接进行业务代码的处理，其他的工作有必要的话在catch中处理即可。
- 如何处理自定义的情况：自定义需要我们的自定义行为在interceptor执行之前，而非之后。比如上面的例子，当status === 'fail'的时候期望不弹框，需要用自定义行为覆盖掉interceptor的行为。下面主要介绍两种实现方式：

  a. axios的 transformResponse API

  axios提供了transformResponse方法，允许我们对响应进行预处理，该方法是在响应拦截器执行前执行的。

图片上传失败，请重新上传

```typescript
axios.post(url, data, {
    transformResponse: function (data, headers) {
        // ...some code
    }
}).then(res => {})
```

b. 通过函数进行包装，具体流程如下：

```typescript
function api (url, data, custom) {
    return axios.post(url, data).then(res => {
```

```
        if (custom) {
            return custom(res)
        } else {
            // deal res
            return res
        }
    }).catch(e => {
        if (custom) {
            return custom(e)
        } else {
            // deal e
            return e
        }
    })
}
```

## Typescript + axios

### 期望

首先想象一下我们如何使用的axios?

TypeScript

```
axios.post('/getList', {
    current: 2,
    size: 10
}).then(res => {
    this.tableData = res.data;
})
```

上面的例子是一个最正常的axios的使用方法，URL永远是string类型，params和res是未知类型，所以我们期望可以让ts对它们进行限制。

### 重写类型

实际上axios提供了一些类型。

重写类型

```typescript
interface ResponseJSON<Data = any> {
    status: ResponseStatus;
    data?: Data;
    message?: string;
    error?: AxiosError | Error;
    response?: Response;
}

export interface AjaxInstance extends AxiosInstance {
    request<T = any, R = ResponseJSON<T>> (config: AxiosRequestConfig): Promise<R>;
    get<T = any, R = ResponseJSON<T>>(url: string, config?: AxiosRequestConfig): Promise<R>;
    delete<T = any, R = ResponseJSON<T>>(url: string, config?: AxiosRequestConfig): Promise<R>;
    head<T = any, R = ResponseJSON<T>>(url: string, config?: AxiosRequestConfig): Promise<R>;
    options<T = any, R = ResponseJSON<T>>(url: string, config?: AxiosRequestConfig): Promise<R>;
    post<T = any, R = ResponseJSON<T>>(url: string, data?: any, config?: AxiosRequestConfig): Promise<R>;
    put<T = any, R = ResponseJSON<T>>(url: string, data?: any, config?: AxiosRequestConfig): Promise<R>;
    patch<T = any, R = ResponseJSON<T>>
    (url: string, data?: any, config?: AxiosRequestConfig): Promise<R>;
}
```

声明完类型，我们便可以使用了