axio

axios源码分析 part1 - Axios.js

A

aixos源码入口文件，根目录下面的index.js，内部就只有一行代码，加载axios

```
module.exports = require('./lib/axios');
```

然后打开axios.js 文件，

```
'use strict';

var utils = require('./utils');
var bind = require('./helpers/bind');
var Axios = require('./core/Axios');
var mergeConfig = require('./core/mergeConfig');
var defaults = require('./defaults');

/**
 * Create an instance of Axios
 *
 * @param {Object} defaultConfig The default config for the instance
 * @return {Axios} A new instance of Axios
 */
function createInstance(defaultConfig) {
  var context = new Axios(defaultConfig);
  var instance = bind(Axios.prototype.request, context);

  // Copy axios.prototype to instance
  utils.extend(instance, Axios.prototype, context);

  // Copy context to instance
  utils.extend(instance, context);

  // Factory for creating new instances
  instance.create = function create(instanceConfig) {
    return createInstance(mergeConfig(defaultConfig, instanceConfig));
  };

  return instance;
}

// Create the default instance to be exported
var axios = createInstance(defaults);

// Expose Axios class to allow class inheritance
```

```
axios.Axios = Axios;

// Expose Cancel & CancelToken
axios.CanceledError = require('./cancel/CanceledError');
axios.CancelToken = require('./cancel/CancelToken');
axios.isCancel = require('./cancel/isCancel');
axios.VERSION = require('./env/data').version;
axios.toFormData = require('./helpers/toFormData');

// Expose AxiosError class
axios.AxiosError = require('../lib/core/AxiosError');

// alias for CanceledError for backward compatibility
axios.Cancel = axios.CanceledError;

// Expose all/spread
axios.all = function all(promises) {
  return Promise.all(promises);
};
axios.spread = require('./helpers/spread');

// Expose isAxiosError
axios.isAxiosError = require('./helpers/isAxiosError');

module.exports = axios;

// Allow use of default import syntax in TypeScript
module.exports.default = axios;

```

上来加载了5个文件，如果不能先理解其中的 core/Axios.js ，那么理解起来会比较不容易。所以先把上面的文件放一放，了解一下core/Axios.js 文件做了什么。文件比较长，分开段落慢慢理解。

```
'use strict';

var utils = require('./../utils');
var buildURL = require('../helpers/buildURL');
var InterceptorManager = require('./InterceptorManager');
var dispatchRequest = require('./dispatchRequest');
var mergeConfig = require('./mergeConfig');
var buildFullPath = require('./buildFullPath');
var validator = require('../helpers/validator');


```

1. 加载必要文件

```

function Axios(instanceConfig) {
  this.defaults = instanceConfig;
```

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

1. 函数接受一个参数，参数的作用是配置信息。
2. 函数内部使用defaults 保存配置信息
3. 创建实例属性interceptor ，一个对象2个属性，request和 response。
4. request 与 response都是InterceptorManager对象。
5. interceptorManager 之后章节会单独讲解。
6. 函数第一次调用时机在文件axios.js，axios底层有提供默认的配置，配置文件就在default/index.js。第一次加载axios，会根据这个预置配置文件生成一个实例，然后返回给用户。这里先不展开讲解开，会放在后续章节讲解

```
Axios.prototype.request = function request(configOrUrl, config) {
  /*eslint no-param-reassign:0*/
  // Allow for axios('example/url'[, config]) a la fetch API
  if (typeof configOrUrl === 'string') {
    config = config || {};
    config.url = configOrUrl;
  } else {
    config = configOrUrl || {};
  }

  config = mergeConfig(this.defaults, config);

  // Set config.method
  if (config.method) {
    config.method = config.method.toLowerCase();
  } else if (this.defaults.method) {
    config.method = this.defaults.method.toLowerCase();
  } else {
    config.method = 'get';
  }

  var transitional = config.transitional;

  if (transitional !== undefined) {
    validator.assertOptions(transitional, {
      silentJSONParsing: validators.transitional(validators.boolean),
      forcedJSONParsing: validators.transitional(validators.boolean),
      clarifyTimeoutError: validators.transitional(validators.boolean)
    }, false);
  }

  // filter out skipped interceptors
  var requestInterceptorChain = [];
```

```
var synchronousRequestInterceptors = true;
this.interceptors.request.forEach(function unshiftRequestInterceptors(interceptor) {
  if (typeof interceptor.runWhen === 'function' && interceptor.runWhen(config) === false
    return;
  }

  synchronousRequestInterceptors = synchronousRequestInterceptors && interceptor.synchro

  requestInterceptorChain.unshift(interceptor.fulfilled, interceptor.rejected);
});

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

var promise;

if (!synchronousRequestInterceptors) {
  var chain = [dispatchRequest, undefined];

  Array.prototype.unshift.apply(chain, requestInterceptorChain);
  chain = chain.concat(responseInterceptorChain);

  promise = Promise.resolve(config);
  while (chain.length) {
    promise = promise.then(chain.shift(), chain.shift());
  }

  return promise;
}


var newConfig = config;
while (requestInterceptorChain.length) {
  var onFulfilled = requestInterceptorChain.shift();
  var onRejected = requestInterceptorChain.shift();
  try {
    newConfig = onFulfilled(newConfig);
  } catch (error) {
    onRejected(error);
    break;
  }
}

try {
  promise = dispatchRequest(newConfig);
} catch (error) {
  return Promise.reject(error);
}

while (responseInterceptorChain.length) {
  promise = promise.then(responseInterceptorChain.shift(), responseInterceptorChain.shif
```

```
  }

  return promise;
};
```

1. Axios 是一个函数，函数也是对象，所以可以挂属性在上面。其上面所有添加属性都会在真正网络请求调用作为上下文(this)提供所需的基础功能比如 interceptors。
2. 然后原型上面添加request方法，接受2个参数，第一个参数可以是请求url （string类型）也可以是配置（对象类型）。如果第一个参数url string，那么第二个参数就是对象。如果第一个参数就是配置信息对象，则让第二个参数直接指向第一个参数。
3. 然后融合底层自带的配置信息和用户提供的配置信息，赋值给config。
4. 如果融合之后的config配置参数没有method属性，则使用默认配置的method属性。如果默认配置里面也没有，则直接默认使用get方法。相反，如果提供了method属性，则以提供的method属性为主。
5. 然后从config配置属性里面获取transitional，transitional的作用就是底层帮助转换数据，比如获取的数据自动转换成json等等功能。如果用户没有提供相关配置，则使用默认提供的配置(transitional.js)
6. 请求真正发送之前，如果有定义interceptor , 则会先处理interceptor，interceptor区分成request和response 2个不同类型。具体不在这里展开，在后续会讲解。但是可以先理解一下http request 与interceptor的关系。即为，interceptor request -> http request -> response interceptor
7. 然后处理interceptor request过滤，通过调用interceptor request的 runwhen 方法，如果返回false，则不调用这个interceptor, 否则就收集当前interceptor request 的fulfilled 和 rejected 函数，放入容器requestInterceptorChain
8. 然后处理interceptor response，处理逻辑相同只是，interceptor response 没有runwhen方法也就不需要进行过滤。最后收集fulfilled , rejected 函数，放入responseInterceptorChain
9. 所有request 和 response 的interceptor处理好了，按照之前说的interceptor request -> http request -> response interceptor 顺序处理。此处interceptor request又可以分成2种方法运行，同步或者异步，默认异步使用promise。

```
Axios.prototype.getUri = function getUri(config) {
  config = mergeConfig(this.defaults, config);
  var fullPath = buildFullPath(config.baseURL, config.url);
  return buildURL(fullPath, config.params, config.paramsSerializer);
};
```

1. getUri 函数处理用户的url。接受一个参数。
2. 融合用户传入的配置config 和默认的配置
3. 然后开始组合url，从config 参数获取baseUrl，url。然后调用buildlFullPath方法进行url的处理
4. buildFullPath 首先确认是否url是绝对路径，如果不是绝对路径，就进行和baseUrl的组合，组合之前使用正则进行了url的规范化（buildFullPath.js , conmbineURLs.js）
5. 然后调用buildURL方法（buildURL.js）进行参数param的处理比如序列化。
6. 然后处理好的param与url 进行组合，然后返回

```
// Provide aliases for supported request methods
```

```
utils.forEach(['delete', 'get', 'head', 'options'], function forEachMethodNoData(method) {
  /*eslint func-names:0*/
  Axios.prototype[method] = function(url, config) {
    return this.request(mergeConfig(config || {}, {
      method: method,
      url: url,
      data: (config || {}).data
    }));
  };
});
```

1. 批量给原型上面添加方法，内部都是使用之前讲解的reques方法

```
utils.forEach(['post', 'put', 'patch'], function forEachMethodWithData(method) {
  /*eslint func-names:0*/

  function generateHTTPMethod(isForm) {
    return function httpMethod(url, data, config) {
      return this.request(mergeConfig(config || {}, {
        method: method,
        headers: isForm ? {
          'Content-Type': 'multipart/form-data'
        } : {},
        url: url,
        data: data
      }));
    };
  }
```

1. 批量方法添加，也是调用request，不同点在于对内部param的处理方法，也考虑了form的处理逻辑。

那么，下一章接着讲解。