深入浅出Axios源码

Axios是什么

Axios 不是一种新的技术，是一个基于 promise 的 HTTP 库，本质上是对原生 XHR 的封装，只不过它是基于 Promise 实现的版本。

有以下特点：

- 在浏览器中创建 XMLHttpRequest 对象获取数据
- 在 node.js 创建 HTTP 请求
- 支持 Promise
- 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

axios 到底是什么

`axios` 是整个 Axios 库对外暴露的对象，通过文档可知，它挂载了很多属性和方法，axios 定义在入口文件 axios.js ，我们看看它到底是什么：

```
function createInstance(defaultConfig) {
  var context = new Axios(defaultConfig);
  var instance = bind(Axios.prototype.request, context);
  utils.extend(instance, Axios.prototype, context);
  utils.extend(instance, context);
  return instance;
}
var axios = createInstance(defaults); // 将要被导出的axios对象
```

这里定义了变量 axios，createInstance 函数执行的返回值赋给了 axios，在 createInstance 函数中，首先创建了一个 Axios 实例，赋给变量 context，再执行 bind 函数，并将返回值赋给变量 instance，然后调用 utils.extend 函数对 instance 做了一些处理，最后 createInstance 函数返回 instance。

我们先看看 *bind* 函数执行返回了什么，为了简洁性，我用 ES6 的写法将 *bind* 函数改写了一下：

```
function bind(fn, thisArg) {
  return function wrap(...args) {
    return fn.apply(thisArg, args);
  };
};
```

结合 `var instance = bind(Axios.prototype.request, context)` 来看：

bind 函数接收 Axios.prototype.request，经查看它是一个函数，还接收 context，一个Axios实例，并且 bind 执行返回一个新的函数wrap，wrap 函数赋给了 instance，所以 instance 指向了 wrap 函数。

wrap 函数执行，即 instance 执行，返回 Axios.prototype.request.apply(context, args)，也就是执行了 Axios.prototype.request 但执行时的 this 指向了 context。可见 bind 函数的实现效果和原生 *bind* 一样，相当于 instance 指向了 `Axios.prototype.request.bind(context)`

```
utils.extend(instance, Axios.prototype, context);
utils.extend(instance, context);
```

继续看接下来两行代码，我们简单看看 extend 函数的实现，它的作用就是把对象b的属性扩展到对象a上，并返回对象a，同时考虑了属性是方法时，this的指向。

```
function extend(a, b, thisArg) {
  forEach(b, function (val, key) { // 遍历对象b的属性，执行回调函数
    if (thisArg && typeof val === 'function') { // 如果指定了this，且屬性值是方法
      a[key] = bind(val, thisArg); // 则在对象a添加改绑 this 后的方法
    } else {
      a[key] = val;
    }
  });
  return a; // 最后返回的是对象a
}
```

所以两次调用 extend 是将 *Axios* 原型上的属性和 Axios 实例上的属性都拷贝到 *instance* 对象上，其中的方法中的 this ，都指向同一个 Axios 实例。

最后 *createInstance* 函数把挂载好的 instance 返回了出来，赋给了 axios 变量，因为 instance 指向了 wrap 函数，所以 axios 也指向了 wrap 函数。

因此 *axios* 执行即 *wrap* 执行，将会执行并返回 *Axios.prototype.request.apply(context, args)* ，执行时 this 指向一个 Axios 的实例，args 是 *axios* 执行时传入的参数组成的数组。因此可以理解为 axios 指向了改变了 this 的 `Axios.prototype.request` 函数，并且这个函数身上挂载了很多属性和方法。

探究Axios构造函数

我们知道 *axios* 实际上指向 Axios.prototype.request 方法，因此 request 方法是整个 axios 库的核心函数，我们到 Axios.js 文件先看看 Axios 构造函数：

```
function Axios(instanceConfig) {
  this.defaults = instanceConfig;
  this.interceptors = {
    request: new InterceptorManager(), // 拦截器管理器实例
    response: new InterceptorManager()
  };
}
Axios.prototype.request = function request(config) {
  // ....
};
```

Axios 的实例挂载了两个属性，一个是 defaults 属性，属性值为 Axios 接收的 instanceConfig ，另一个是 interceptors 属性，属性值为一个拦截器对象，包含 request 和 response 两个属性，属性值都为 InterceptorManager 实例。

在 Axios.js 文件中，接下来还会往 Axios 的原型对象挂载一些方法：

```
utils.forEach(['delete', 'get', 'head', 'options'], function(method) {
  Axios.prototype[method] = function(url, config) {
    return this.request(utils.merge(config || {}, {
      method: method,
      url: url
    }));
  };
});
utils.forEach(['post', 'put', 'patch'], function(method) {
  Axios.prototype[method] = function(url, data, config) {
    return this.request(utils.merge(config || {}, {
      method: method,
      url: url,
      data: data
    }));
  };
});
```

前面提到 Axios.prototype 上的属性已经拷贝了给 axios 对象，所以 axios 对象可以直接调用 get , post , head 等这些挂载到 Axios 原型上的方法，当然也包括 request 方法。

我们注意到 get/post/head 等这些方法的执行，都是返回 Axios.prototype.request 的执行结果，因为 axios 实际指向 Axios.prototype.request 方法，所以 axios 可以直接作为 request 方法调用。

因此我们可以得出，发起请求的调用写法有多种，比如 axios(config) 、 axios.request(config) 、 axios[method](url[, config]) 等，实际都调用的是 Axios.prototype.request 方法。

多种请求写法

Axios有多种请求的写法，但其实核心是执行的是同一个方法，后面将阐述

API 写法	说明
axios(config)	传入相关配置来创建请求
axios(url[, config])	可以只传url，但会默认发送 GET 请求
axios.request(config)	config中url是必须的
axios[method](url[, config]) axios[method](url[, data[, config]])	为了方便，给所有支持的请求方法提供了别名 这种情况下，不用再config中指定url、method、data

axios 接收的参数就很灵活，针对不同的传参它内部整合成一个config对象。

```
Axios.prototype.request = function request(config) {
  if (typeof config === 'string') {
    config = arguments[1] || {};
    config.url = arguments[0];
  } else {
    config = config || {};
  }
  config = mergeConfig(this.defaults, config);
  config.method = config.method ? config.method.toLowerCase() : 'get';
  // ...
  return promise;
};
```

我们可以看到，request方法只设了一个形参config，但用户传入两个参数是有效的，如果 `typeof config === 'string'` 即用户传入的第一个参数是字符串，就把用户传入的第二个参数作为 config ，如果用户没传第二个参数，则 config 取一个空对象，接着把第一个传入的字符串赋给 config.url 属性。

如果用户传的第一个参数不是字符串，那么把它直接赋给 config，如果什么都没传，赋给 config 一个空对象。

然后进行的是默认的配置对象和config的合并，合并后的对象赋给 config 。如果 config对象中method属性值存在，则将它转为小写，如果不存在method属性值，就将 'get' 赋给 config.method

因此，不同的传参方式，都会进行归一化处理成一个config对象。

配置对象config怎么起作用

我们使用 axios ，传入的配置对象是很重要的部分，可以看到传入的 config 对象经历很多层的传递，它在源码里怎么一步步传到需要它的地方，是我们希望了解的。

axios 文档告诉我们，我们可以定义这些配置项：

配置项	说明
`url`	请求的URL
`method`	请求的方法
`baseURL`	加在 `url` 前面，除非 `url` 是绝对URL
`transformRequest`	在向服务器发请求前，修改请求数据
`transformResponse`	在响应数据传递给then/catch前，修改响应数据
`headers`	自定义请求头
`params`	URL参数，必须是plain对象或URLSearchParams对象
`paramsSerializer`	对params序列化的函数
`data`	作为请求主体的数据
`timeout`	指定请求超时的毫秒数，超过就请求中断
`withCredentials`	跨域请求时是否需要使用凭证
`adapter`	允许你自己写处理config的函数
`responseType`	服务器响应的数据类型
`auth` 等.....	.....

axios的默认config

回到 /axios.js 文件中，我们看看 aixos 中默认的 config 对象是怎么被使用的。

```
var defaults = require('./defaults');
var axios = createInstance(defaults);
function createInstance(defaultConfig) {
  var context = new Axios(defaultConfig);
  // ....省略
  return instance;
}
```

我们看到 ./defaults 文件导出一个默认的 config 对象，赋给了变量 defaults ，再传入 createInstance 函数执行，在 createInstance 函数中，传入 new Axios 执行，前面提到 Axios 构造函数会把接收的 config 对象挂载到创建的 Axios 实例上的 defaults 属性：

开发者自己传入的配置对象是怎么处理的？我们知道真正的入口函数 Axios.prototype.request 函数会接收开发者传入的配置对象，我们看看它的具体实现：

```
Axios.prototype.request = function request(config) {
  // ....
  config = mergeConfig(this.defaults, config) // 默认的config和request传入的config合并
  // ....
};
```

在 request 方法中，this 指向 Axios 实例，因此 this.defaults 就是 Axios 实例上的 defaults 属性值，即默认的配置对象。 request 函数中，调用了 mergeConfig 函数，将默认配置对象和开发者传入的 config 对象合并，再覆盖到 config ，具体的合并细节我们看看 mergeConfig 的实现：

我们知道 Axios实例的属性已经被添加到了 axios 对象上了，用户可以通过 `axios.defaults` 访问到 Axios 实例上的 defaults ，可以直接修改默认配置中的配置项，像下面这样伪代码所示：

```
axios.defaults[configName] = value;
```

除了这种修改默认配置的方式之外， axios 对象还对外暴露了一个 create 方法，供开发者传入自定义的配置对象，函数返回出一个新的 axios 对象。像下面这样使用：

```
let newAxiosInstance = axios.create({
  [配置项名称]: [配置项的值]
  // ...
})
```

axios.create 函数的实现只有简单的一句：

```
axios.create = function(instanceConfig) {
  return createInstance(mergeConfig(axios.defaults, instanceConfig));
};
```

axios.create 方法返回 createInstance 函数的执行结果，前面我们知道 createInstance 函数的返回了一个 instance 对象，所以这里 axios.create 返回的一个新的 axios 对象，传入的是 mergeConfig 函数的返回值， mergeConfig 函数将默认配置对象和 axios.create 传入的配置对象合并，返回出一个整合好的对象给 createInstance 函数执行。

可见 axios.create 就是新建一个 axios 对象，由用户配置出一套自定义的通用配置。

总结一下，改动配置对象一共有三种方式

1. *axios(config)* 等通过 Axios.prototype.request 的调用传入配置对象
2. `axios.defaults[name] = value` 直接修改默认的配置对象
3. `axios.create(config)` 另创建一个 axios 对象，配置对象是自定义合并过的

这三种方式叠加使用的话，最后 config 肯定要整合成一个，必然涉及到覆盖，因此存在优先级的问题。

由于 2 和 3 都是针对默认配置进行改动，所以 1 的优先级最高，3 axios.create 接收的配置会合并到默认配置，所以优先级排第 2 ：

1. request方法的的参数config
2. Axios实例属性defaults
3. 默认配置对象defaults（来自/lib/defaults.js)

探究 Axios.prototype.request

前面我们看了 Axios 构造函数，也看了 config 是怎么传递和合并的，现在来看 request 这个核心方法：

```
Axios.prototype.request = function(config) {
  // 省略....
  config = mergeConfig(this.defaults, config);
  var chain = [dispatchRequest, undefined];
  var promise = Promise.resolve(config);
  this.interceptors.request.forEach(function (interceptor) {
    chain.unshift(interceptor.fulfilled, interceptor.rejected);
  });
  this.interceptors.response.forEach(function (interceptor) {
    chain.push(interceptor.fulfilled, interceptor.rejected);
  });
  while (chain.length) {
    promise = promise.then(chain.shift(), chain.shift())
  }
  return promise;
};
```

前面讲过 *config* 整合了默认配置和传入的配置。接着定义一个数组 *chain* ，先放入 *dispatchRequest* 函数和一个 *undefined* 。接着 Promise.resolve(config) 创建一个以 *config* 为实现的 *Promise* 实例，赋给变量 promise 。

接下来是这几行代码：

```
this.interceptors.request.forEach(interceptor => {
  chain.unshift(interceptor.fulfilled, interceptor.rejected);
});
this.interceptors.response.forEach(interceptor => {
  chain.push(interceptor.fulfilled, interceptor.rejected);
});
```

我们前面提到过， *this.interceptors.request* 和 *this.interceptors.response* 是 Axios 实例的 *interceptors* 属性对象的子属性，属性值都为 `new InterceptorManager()`

new InterceptorManager() 具体是什么呢？我们看看 InterceptorManager 这个构造函数和它的 forEach 方法：

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

可见每个 *InterceptorManager* 实例都有自己的 *handlers* 属性，属性值为一个数组。

InterceptorManager 的原型方法 *forEach* 就是遍历实例的 *handlers* 数组，跳过为 null 的项，将数组的每一项传入 *fn* 执行。 *fn* 就是 *forEach* 的回调函数，即：

```
function (interceptor) {
  chain.unshift(interceptor.fulfilled, interceptor.rejected);
}
function (interceptor) {
  chain.push(interceptor.fulfilled, interceptor.rejected);
}
```

因此 *this.interceptors.request* 的 *handlers* 数组中的每个 interceptor 对象，它的 *fulfilled* 属性值和 *rejected* 属性值被添加到 chain 数组的开头。

this.interceptors.response 的 *handlers* 数组中的每个 interceptor 对象，它的 *fulfilled* 属性值和 *rejected* 属性值被添加到 chain 数组的末尾。

注意它们都是被成对地加入到 chain 数组中。问题是， handlers 数组怎么存了这些 interceptor 对象的？其实是用户调用 use 方法注册的：

```
InterceptorManager.prototype.use = function (fulfilled, rejected) {
  this.handlers.push({
    fulfilled,
    rejected
  });
  return this.handlers.length - 1;
};
```

use 是 InterceptorManager 的原型方法， *axios.interceptors.request* 和_axios.interceptors.response_ 都是 InterceptorManager 的实例。用户可以通过调用 *axios.interceptors.request.use* 添加请求拦截器方法，做一些发起请求前的修改请求的 data 或 header 的工作，下面是用户使用 use 的方式：

```
axios.interceptors.request.use(
  config => {
    // 在发送http请求之前做一些事情
    return config; // 有且必须有一个config对象被返回
  }, error => {
    // 请求出错时做一些事情
    return Promise.reject(error);
  }
);
```

use 可以接收用户定义的成功回调 fulfilled 和失败回调 rejected ，然后将它们分别赋给一个对象的 fulfilled 和 rejected 的属性，再将对象推入 handlers 数组中。注意：成功的回调必须返回 config 对象。

这样 axios.interceptors.request.handler 数组，就存放着用户通过 use 注册的请求拦截器的成功回调和失败回调。

同样的，用户调用 axios.interceptors.response.use 添加响应拦截器方法，用于响应数据返回之后的处理工作：

```
axios.interceptors.response.use(
  response => {
      // 针对响应数据做一些事情
    return response;
  }, error => {
      // 对于响应出错做一些事情
    return Promise.reject(error);
  }
);
```

这样 axios.interceptors.response.handler 数组，存放着用户通过 use 注册的响应拦截器的成功回调和失败回调。现在我们知道了 handler 数组中的拦截器对象以及它的两个方法是怎么来的了。

回到 chain 数组，如果用户添加了拦截器方法， chain 数组就会存放成对的拦截器回调和 dispatchRequest 方法，接下来开启一个 while 循环：

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

进入 while 循环之前， promise 是 resolved 状态的 promise 实例，它调用 then ，接收两个从 chain 数组成对 shift 出来的函数作为 then 的成功回调和失败回调。拦截器方法中我们并没有调用 resolve 或 reject ，因此 then 返回的新的 promise 实例的状态是 pending 。

同时，promise.then 的执行将成功和失败的回调推入异步执行的微任务队列中。在 while 循环中， then 返回的 promise 实例覆盖了 promise 变量，然后继续调用 then ，形成了链式调用，直到 chain 数组的元素减少到空，循环结束，这个过程中，chain数组中的函数双双的被推入到异步执行的微任务队列中。

注意，经过 while 循环后的 promise 是一个状态为 pending 的 promise 实例，并且 request 方法会将这个 promise 实例返回。

当同步代码执行完，就开始执行异步的微任务队列，首先执行请求拦截器方法，然后 return config 。依次执行完所有请求拦截器方法，就执行 dispatchRequest 方法，它能接收它上一个 then 的回调返回的 config 。

接下来看看 dispatchRequest 函数，顾名思义，它是真正分发请求的 API ：

dispatchrequest 做了什么

```
function dispatchRequest(config) {
  // ...省略
  var adapter = config.adapter || defaults.adapter;
  return adapter(config).then( /*代码省略*/ );
};
```

我们只关注后面两句，如果用户在 config 对象中定义了 adapter 函数就赋给变量 adapter ，如果没有定义，则使用默认的 defaults.adapter 。然后执行 adapter(config) 并调用 then ，最后 dispatchRequest 将 then 返回的 promise 实例返回。

适配器 adapter 的实现

因为用户一般不会自己定义 adaptor ，我们看看默认的 defaults.adapter 的实现：

```
var defaults = {
  adapter: getDefaultAdapter(),
  // ....
};
function getDefaultAdapter() {
  var adapter;
  if (typeof process !== 'undefined' && Object.prototype.toString.call(process) === '[object prc
    adapter = require('./adapters/http');
  } else if (typeof XMLHttpRequest !== 'undefined') {
    adapter = require('./adapters/xhr');
  }
  return adapter;
}
```

defaults.adaptor 的属性值是 getDefaultAdapter() 的返回值。在 getDefaultAdapter 函数中，根据宿主环境引入不同的 adapter 函数： Node.js 环境下，引入 http.js 模块；浏览器环境下，引入 xhr.js 模块。

http.js 文件中使用 Node.js 内置的 http 模块来实现请求的发送，这里不作具体分析。xhr.js 文件中导出了 xhrAdapter 函数，也就是我们的 defaults.adapter ，我们看看它的实现:

xhrAdapter 的实现

```
function xhrAdapter(config) {
  return new Promise(function(resolve, reject) {
    var request = new XMLHttpRequest(); // 创建 XMLHttpRequest 实例
    // 调用request的实例方法open，发起xhr请求
    request.open(config.method.toUpperCase(), buildURL(config.url, config.params, config.paramsS
    // ...
    // 监听 readyState，设置对应的处理回调函数
    request.onreadystatechange = function() {
      if (!request || request.readyState !== 4) return // readyState没到4会直接返回
      // XMLHttpRequest.status表示服务器响应的HTTP状态码。
      // 如果通信成功为200。請求發出之前，这个属性默认屬性為0。
      if (request.status === 0 && !(request.responseURL && request.responseURL.indexOf('file:')
        return;
      // 准备 response 对象
      var responseHeaders = 'getAllResponseHeaders' in request ? parseHeaders(request.getAllResp
      var responseData = !config.responseType || config.responseType === 'text' ? request.respor
      var response = {
        data: responseData,
        status: request.status,
        statusText: request.statusText,
        headers: responseHeaders,
        config,
        request
      };
      settle(resolve, reject, response);
      request = null;// 清除request对象
    };
    request.send(requestData); // 发送请求
  });
};
```

为了方便阅读，对 xhrAdapter 函数做了一些删减，函数返回出一个 promise 实例，它管控了一套 XMLHTTPRequest 发起 AJAX 请求的流程，这是我们熟悉的。

异步请求成功后，根据返回的响应数据整合出 *response* 对象，将该对象传入 *settle* 方法执行，会根据响应的数据决定是调用 resolve 还是 reject ，我们看看 settle 方法的实现：

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

settle 函数首先获取 validateStatus 函数，调用它对 response.status 值判断，在合并配置项时，即 mergeConfig 函数中，如果用户配置了自己的 validateStatus 函数，就会优先使用用户配置的，否则采用默认配置对象中有 validateStatus 函数。我们看看默认的 defaults.validateStatus ：

```
var defaults = {
  validateStatus (status) { // HTTP状态码必须满足200-300
    return status >= 200 && status < 300;
  }
};
```

validateStatus 函数会对响应的 HTTP 状态码进行验证，如果满足在 [200,300) 内，则返回 true ，将调用 resolve(response) 将 xhrAdapter 函数返回的 promise 实例 resolve ，否则调用 reject 。

因此在 dispatchRequest 函数中， adapter 函数执行返回的 promise 实例，继续调用 then ：

```
function dispatchRequest(config) {
  // ...省略
  var adapter = config.adapter || defaults.adapter;
  return adapter(config).then((response) => {
    throwIfCancellationRequested(config);
    response.data = transformData(// 转换 response data
      response.data,
      response.headers,
      config.transformResponse
    );
    return response;
  }, (reason) => {
    if (!isCancel(reason)) {
      throwIfCancellationRequested(config);
      if (reason && reason.response) { // 转换 response data
        reason.response.data = transformData(
          reason.response.data,
          reason.response.headers,
          config.transformResponse
        );
      }
    }
    return Promise.reject(reason);
  });
};
```

在 then 传入的成功回调和失败回调，会对 adapter 返回的 promise 实例的成功或失败结果，即 response 和 reason 做再次加工，即调用 transformData 函数对 response.data 做处理，最后返回 response 。

因此，如果 HTTP 请求成功， dispatchRequest 函数返回的是以 response 对象为实现的，状态为 resolved 的 promise 实例。

接着就会执行微任务队列中剩下的响应拦截器方法，它们接收 response 对象，用户对 response 对象做一些处理，再 return response ，这样 response 对象就在这些响应拦截器方法中传递，这些操作都是基于微任务的异步的，异步队列执行完后， promise 最后为一个状态为 resolved / rejected 的 promise 实例。

也就是 Axios.prototype.request 最后返回 promise 实例会随着异步任务结束而 resolve 或 reject。用户可以用它调用 then ，在 then 的回调中拿到 response 对象 或 reason 对象。

总结

到目前为止，整个 axios 调用流程就讲完了。核心方法是：Axios.prototype.request 。

如果用户设置了拦截器方法，就将它们推入一个叫 chain 的数组中，chain 数组形成了：[请求拦截器方法 + dispathRequest + 响应拦截器方法] 这样的队列，然后通过链式调用 promise 实例的 then 方法，将 chain 数组中的方法注册为成功和失败的回调，都放入微任务队列中等待异步执行。

config 对象在这个微任务队列中的前半部分传递，到了 dispatchRequest 方法，它执行 adapter 方法（对于浏览器就是 xhrAdapter 方法），而 xhrAdapter 方法就是发起 XHR 请求的流程的用一层 promise 封装，会根据响应的状态决定将该 promise resolve 或 reject 掉。

然后 dispatchRequest 中针对 adapter 的返回值再调用 then ，对响应的数据做再次的处理，再把 response 对象 return 出来。所以在接下来的微任务队列的后半部分，响应拦截器方法接收的是 response ，由对 response 对象做处理， response 相当于在队列中传递。

最后 Axios.prototype.request 经过 then 链式调用的 promise 的状态，随着微任务队列执行的结束而被 settle ，它的状态被确定下来。

意味着，你使用 axios 提供给你的 API 的返回值，再调用 then 就能在回调中拿到 response/reason 对象。

ok，完整的流程就叙述完毕。

mergeConfig 合并的细节

```
function mergeConfig(config1, config2 = {}) {
  var config = {}; // 结果对象

  utils.forEach(['url', 'method', 'params', 'data'], function valueFromConfig2(prop) {
    if (typeof config2[prop] !== 'undefined') { // 如果config2中这四个设置项有定义
      config[prop] = config2[prop]; // 就将其加入到config对象中
    }
  });

  utils.forEach(['headers', 'auth', 'proxy'], function mergeDeepProperties(prop) {
    if (utils.isObject(config2[prop])) {
      // 如果config2中该属性值是对象，就把config1和config2的该属性值深度合并，赋给config
      config[prop] = utils.deepMerge(config1[prop], config2[prop]);
    } else if (typeof config2[prop] !== 'undefined') {
      // config2中該屬性值有定义，但不是对象，就直接加入到config
      config[prop] = config2[prop];
    } else if (utils.isObject(config1[prop])) {
      // config2中该属性未定义，但config1中有定义并且是对象，把config1的该属性值进行内部的深度合并，即
      config[prop] = utils.deepMerge(config1[prop]);
    } else if (typeof config1[prop] !== 'undefined') {
      // 如果config2中該属性未定义，config1该属性有定义，但不是對象，直接加入到config中
      config[prop] = config1[prop];
    }
  });
  utils.forEach(['baseURL', 'transformRequest', 'transformResponse', 'paramsSerializer', 'timeou
    if (typeof config2[prop] !== 'undefined') {
      config[prop] = config2[prop]; // 如果config2中該属性有定义，将其拷贝到config中
    } else if (typeof config1[prop] !== 'undefined') {
      config[prop] = config1[prop]; // config2上該属性未定义，但config1中有定义，则加入config对象中
    }
  });
  return config;
};
```

从调用的方式可知， config1 接收的是 axios 的默认配置对象， config2 接收的是用户定义的配置对象。具体的合并细节见注释。

整体流程介绍完了，还有一些细节我们拾遗一下。比如
我们看看在 Axios.prototype.request 中是怎么实现的。

下面是一些 axios 库的一些补充功能：

取消请求

这部分我们用得比较少，先看看是怎么使用的：

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

axios.get('/user/12345', {
  cancelToken: source.token
}).catch(function(thrown) { // 失败，先看是不是Cancel对象
  if (axios.isCancel(thrown)) {
    console.log('请求取消', 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.');
```

我们发现 要先引用 axios.CancelToken ，然后调用 CancelToken 的 source 方法，返回出一个对象，里面有 cancel 和 token 属性。我们先从 axios.js 中看到

```
axios.Cancel = require('./cancel/Cancel');
axios.CancelToken = require('./cancel/CancelToken');
axios.isCancel = require('./cancel/isCancel');
```

axios 对象挂载了 CancelToken 方法，我们看到它的具体实现：

```
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 构造函数在调用时，传入一个执行器方法 executor ，会在函数内执行 executor 。CancelToken 会给它的实例挂载一个 promise 属性，属性值是一个 promise 对象，值得注意的是，promise 的 resolve 赋给了构造函数内定义的 resolvePromise 变量，resolvePromise 方法在 executor 方法里面调用。

这意味着什么，先看一个简单的例子：

```
let resolveHandle;
new Promise((resolve, reject) => {
  resolveHandle = resolve;
  // resolve('ok')
}).then(res => {
  console.log(res);
});
resolveHandle('ok'); // "ok"
```

我不像正常那样在传入 new Promise 的 执行器函数中调用 resolve 。

而是拿到 resolve 的引用，在外部调用，因为，promise 实例管控的操作，不管是异步还是同步的，都不能从外部决定 promise 实例是成功还是失败的，现在就相当于把控制权交给外部的 resolveHandle ，可以在外部控制这个 promise 成功与否。

```
CancelToken.source = function source() {
  var cancel;
  var token = new CancelToken(function executor(c) {
    cancel = c;
  });
  return {
    token,
    cancel
  };
};
```

CancelToken 函数挂载了一个 source 方法，它返回一个包含 token 和 cancel 的对象，token 的属性值是 CancelToken 的实例。