页面中使用组件

```HTML
<div id="app" v-cloak>
    <cat name="my cat">
        <kitten></kitten>
    </cat>
</div>
```

在父组件的 `errorCaptured` 则能够捕获到信息

```JavaScript
cat EC: TypeError: dontexist is not a function
info: render
```

21.2.3. 源码分析

异常处理源码

源码位置：/src/core/util/error.js

JavaScript | 复制代码

```
// Vue 全局配置,也就是上面的Vue.config
import config from '../config'
import { warn } from './debug'
// 判断环境
import { inBrowser, inWeex } from './env'
// 判断是否是Promise，通过val.then === 'function' && val.catch === 'function', val ! === null && val !== undefined
import { isPromise } from 'shared/util'
// 当错误函数处理错误时，停用deps跟踪以避免可能出现的infinite rendering
// 解决以下出现的问题https://github.com/vuejs/vuex/issues/1505的问题
import { pushTarget, popTarget } from '../observer/dep'

export function handleError (err: Error, vm: any, info: string) {
    // Deactivate deps tracking while processing error handler to avoid possible infinite rendering.
    pushTarget()
    try {
        // vm指当前报错的组件实例
        if (vm) {
            let cur = vm
            // 首先获取到报错的组件，之后递归查找当前组件的父组件，依次调用errorCaptured 方法。
            // 在遍历调用完所有 errorCaptured 方法、或 errorCaptured 方法有报错时，调用 globalHandleError 方法
            while ((cur = cur.$parent)) {
                const hooks = cur.$options.errorCaptured
                // 判断是否存在errorCaptured钩子函数
                if (hooks) {
                    // 选项合并的策略，钩子函数会被保存在一个数组中
                    for (let i = 0; i < hooks.length; i++) {
                        // 如果errorCaptured 钩子执行自身抛出了错误，
                        // 则用try{}catch{}捕获错误，将这个新错误和原本被捕获的错误都会发送给全局的config.errorHandler
                        // 调用globalHandleError方法
                        try {
                            // 当前errorCaptured执行，根据返回是否是false值
                            // 是false，capture = true，阻止其它任何会被这个错误唤起的 errorCaptured 钩子和全局的 config.errorHandler
                            // 是true capture = fale，组件的继承或父级从属链路中存在的多个 errorCaptured 钩子，会被相同的错误逐个唤起
                            // 调用对应的钩子函数，处理错误
                            const capture = hooks[i].call(cur, err, vm, info) === false
                            if (capture) return
                        } catch (e) {
```

```
                                globalHandleError(e, cur, 'errorCaptured hook')
                            }
                        }
                    }
                }
            }
            // 除非禁止错误向上传播，否则都会调用全局的错误处理函数
            globalHandleError(err, vm, info)
        } finally {
            popTarget()
        }
    }
    // 异步错误处理函数
    export function invokeWithErrorHandling (
    handler: Function,
     context: any,
     args: null | any[],
        vm: any,
            info: string
            ) {
                let res
                try {
                    // 根据参数选择不同的handle执行方式
                    res = args ? handler.apply(context, args) : handler.call(context)
                    // handle返回结果存在
                    // res._isVue an flag to avoid this being observed，如果传入值的_isVue为ture时(即传入的值是Vue实例本身)不会新建observer实例
                    // isPromise(res) 判断val.then === 'function' && val.catch === 'function', val ！=== null && val !== undefined
                    // !res._handled  _handle是Promise 实例的内部变量之一，默认是false，代表onFulfilled,onRejected是否被处理
                    if (res && !res._isVue && isPromise(res) && !res._handled) {
                        res.catch(e => handleError(e, vm, info + ` (Promise/async)`))
                        // avoid catch triggering multiple times when nested calls
                        // 避免嵌套调用时catch多次的触发
                        res._handled = true
                    }
                } catch (e) {
                    // 处理执行错误
                    handleError(e, vm, info)
                }
                return res
            }
```

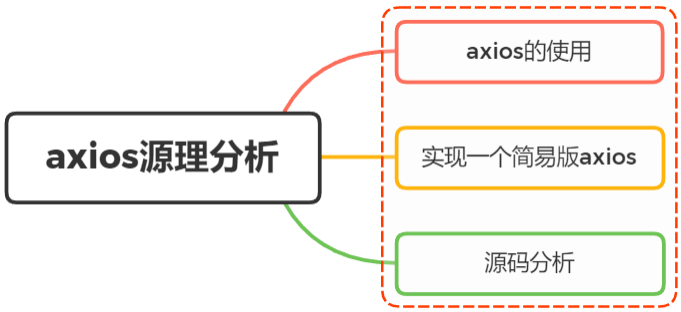
```
78
79
80  //全局错误处理
81  function globalHandleError (err, vm, info) {
82      // 获取全局配置，判断是否设置处理函数，默认undefined
83      // 已配置
84      if (config.errorHandler) {
85          // try{}catch{} 住全局错误处理函数
86          try {
87              // 执行设置的全局错误处理函数，handle error 想干啥就干啥💗
88              return config.errorHandler.call(null, err, vm, info)
89          } catch (e) {
90              // 如果开发者在errorHandler函数中手动抛出同样错误信息throw err
91              // 判断err信息是否相等，避免log两次
92              // 如果抛出新的错误信息throw err Error('你好毒')，将会一起log输出
93              if (e !== err) {
94                  logError(e, null, 'config.errorHandler')
95              }
96          }
97      }
98      // 未配置常规log输出
99      logError(err, vm, info)
100 }
101
102 // 错误输出函数
103 function logError (err, vm, info) {
104     if (process.env.NODE_ENV !== 'production') {
105         warn(`Error in ${info}: "${err.toString()}"`, vm)
106     }
107     /* istanbul ignore else */
108     if ((inBrowser || inWeex) && typeof console !== 'undefined') {
109         console.error(err)
110     } else {
111         throw err
112     }
}
```

21.3. 小结

- `handleError` 在需要捕获异常的地方调用，首先获取到报错的组件，之后递归查找当前组件的父组件，依次调用 `errorCaptured` 方法，在遍历调用完所有 `errorCaptured` 方法或 `errorCaptured` 方法有报错时，调用 `globalHandleError` 方法
- `globalHandleError` 调用全局的 `errorHandler` 方法，再通过 `logError` 判断环境输出错误信息

- `invokeWithErrorHandling` 更好的处理异步错误信息
- `logError` 判断环境，选择不同的抛错方式。非生产环境下，调用 `warn` 方法处理错误

22. 你了解axios的原理吗？有看过它的源码吗？



22.1. axios的使用

关于 `axios` 的基本使用，上篇文章已经有所涉及，这里再稍微回顾下：

发送请求

JavaScript | 复制代码

```
import axios from 'axios';

axios(config) // 直接传入配置
axios(url[, config]) // 传入url和配置
axios[method](url[, option]) // 直接调用请求方式方法，传入url和配置
axios[method](url[, data[, option]]) // 直接调用请求方式方法，传入data、url和配置
axios.request(option) // 调用 request 方法

const axiosInstance = axios.create(config)
// axiosInstance 也具有以上 axios 的能力

axios.all([axiosInstance1, axiosInstance2]).then(axios.spread(response1, response2))
// 调用 all 和传入 spread 回调
```

请求拦截器

```JavaScript
axios.interceptors.request.use(function (config) {
    // 这里写发送请求前处理的代码
    return config;
}, function (error) {
    // 这里写发送请求错误相关的代码
    return Promise.reject(error);
});
```

响应拦截器

```JavaScript
axios.interceptors.response.use(function (response) {
    // 这里写得到响应数据后处理的代码
    return response;
}, function (error) {
    // 这里写得到错误响应处理的代码
    return Promise.reject(error);
});
```

取消请求

```JavaScript
// 方式一
const CancelToken = axios.CancelToken;
const source = CancelToken.source();

axios.get('xxxx', {
  cancelToken: source.token
})
// 取消请求（请求原因是可选的）
source.cancel('主动取消请求');

// 方式二
const CancelToken = axios.CancelToken;
let cancel;

axios.get('xxxx', {
  cancelToken: new CancelToken(function executor(c) {
    cancel = c;
  })
});
cancel('主动取消请求');
```

22.2. 实现一个简易版axios

构建一个 `Axios` 构造函数，核心代码为 `request`

JavaScript 复制代码

```
class Axios {
    constructor() {

    }

    request(config) {
        return new Promise(resolve => {
            const {url = '', method = 'get', data = {}} = config;
            // 发送ajax请求
            const xhr = new XMLHttpRequest();
            xhr.open(method, url, true);
            xhr.onload = function() {
                console.log(xhr.responseText)
                resolve(xhr.responseText);
            }
            xhr.send(data);
        })
    }
}
```

导出 `axios` 实例

JavaScript 复制代码

```
// 最终导出axios的方法，即实例的request方法
function CreateAxiosFn() {
    let axios = new Axios();
    let req = axios.request.bind(axios);
    return req;
}

// 得到最后的全局变量axios
let axios = CreateAxiosFn();
```

上述就已经能够实现 `axios({ })` 这种方式的请求

下面是来实现下 `axios.method()` 这种形式的请求

JavaScript | 复制代码

```
// 定义get,post...方法，挂在到Axios原型上
const methodsArr = ['get', 'delete', 'head', 'options', 'put', 'patch', 'post'];
methodsArr.forEach(met => {
    Axios.prototype[met] = function() {
        console.log('执行'+met+'方法');
        // 处理单个方法
        if (['get', 'delete', 'head', 'options'].includes(met)) { // 2个参数(url[, config])
            return this.request({
                method: met,
                url: arguments[0],
                ...arguments[1] || {}
            })
        } else { // 3个参数(url[,data[,config]])
            return this.request({
                method: met,
                url: arguments[0],
                data: arguments[1] || {},
                ...arguments[2] || {}
            })
        }

    }
})
```

将 `Axios.prototype` 上的方法搬运到 `request` 上

首先实现个工具类，实现将 `b` 方法混入到 `a` ，并且修改 `this` 指向

```javascript
const utils = {
  extend(a,b, context) {
    for(let key in b) {
      if (b.hasOwnProperty(key)) {
        if (typeof b[key] === 'function') {
          a[key] = b[key].bind(context);
        } else {
          a[key] = b[key]
        }
      }

    }
  }
}
```

修改导出的方法

```javascript
function CreateAxiosFn() {
  let axios = new Axios();

  let req = axios.request.bind(axios);
  // 增加代码
  utils.extend(req, Axios.prototype, axios)

  return req;
}
```

构建拦截器的构造函数

```JavaScript
class InterceptorsManage {
  constructor() {
    this.handlers = [];
  }

  use(fullfield, rejected) {
    this.handlers.push({
      fullfield,
      rejected
    })
  }
}
```

实现 `axios.interceptors.response.use` 和 `axios.interceptors.request.use`

```JavaScript
class Axios {
    constructor() {
        // 新增代码
        this.interceptors = {
            request: new InterceptorsManage,
            response: new InterceptorsManage
        }
    }

    request(config) {
    ...
    }
}
```

执行语句 `axios.interceptors.response.use` 和 `axios.interceptors.request.use` 的时候，实现获取 `axios` 实例上的 `interceptors` 对象，然后再获取 `response` 或 `request` 拦截器，再执行对应的拦截器的 `use` 方法

把 `Axios` 上的方法和属性搬到 `request` 过去

```JavaScript
function CreateAxiosFn() {
  let axios = new Axios();

  let req = axios.request.bind(axios);
  // 混入方法， 处理axios的request方法，使之拥有get,post...方法
  utils.extend(req, Axios.prototype, axios)
  // 新增代码
  utils.extend(req, axios)
  return req;
}
```

现在 `request` 也有了 `interceptors` 对象，在发送请求的时候，会先获取 `request` 拦截器的 `handlers` 的方法来执行

首先将执行 `ajax` 的请求封装成一个方法

```JavaScript
request(config) {
    this.sendAjax(config)
}
sendAjax(config){
    return new Promise(resolve => {
        const {url = '', method = 'get', data = {}} = config;
        // 发送ajax请求
        console.log(config);
        const xhr = new XMLHttpRequest();
        xhr.open(method, url, true);
        xhr.onload = function() {
            console.log(xhr.responseText)
            resolve(xhr.responseText);
        };
        xhr.send(data);
    })
}
```

获得 `handlers` 中的回调

```javascript
request(config) {
    // 拦截器和请求组装队列
    let chain = [this.sendAjax.bind(this), undefined] // 成对出现的，失败回调暂时不处理

    // 请求拦截
    this.interceptors.request.handlers.forEach(interceptor => {
        chain.unshift(interceptor.fullfield, interceptor.rejected)
    })

    // 响应拦截
    this.interceptors.response.handlers.forEach(interceptor => {
        chain.push(interceptor.fullfield, interceptor.rejected)
    })

    // 执行队列，每次执行一对，并给promise赋最新的值
    let promise = Promise.resolve(config);
    while(chain.length > 0) {
        promise = promise.then(chain.shift(), chain.shift())
    }
    return promise;
}
```

`chains` 大概是 `['fulfilled1','reject1','fulfilled2','reject2','this.sendAjax','undefined','fulfilled2','reject2','fulfilled1','reject1']` 这种形式

这样就能够成功实现一个简易版

22.3. 源码分析

首先看看目录结构

```
├── /lib/                       # 项目源码目
│ ├── /adapters/                # 定义发送请求的适配器
│ │ ├── http.js                 # node环境http对象
│ │ └── xhr.js                  # 浏览器环境XML对象
│ ├── /cancel/                  # 定义取消功能
│ ├── /helpers/                 # 一些辅助方法
│ ├── /core/                    # 一些核心功能
│ │ ├── Axios.js                # axios实例构造函数
│ │ ├── createError.js          # 抛出错误
│ │ ├── dispatchRequest.js      # 用来调用http请求适配器方法发送请求
│ │ ├── InterceptorManager.js   # 拦截器管理器
│ │ ├── mergeConfig.js          # 合并参数
│ │ ├── settle.js               # 根据http响应状态，改变Promise的状态
│ │ └── transformData.js        # 改变数据格式
│ ├── axios.js                  # 入口，创建构造函数
│ ├── defaults.js               # 默认配置
│ └── utils.js                  # 公用工具
```

`axios` 发送请求有很多实现的方法，实现入口文件为 `axios.js`

JavaScript | 复制代码

```javascript
function createInstance(defaultConfig) {
  var context = new Axios(defaultConfig);

  // instance指向了request方法，且上下文指向context，所以可以直接以 instance(option) 方式调用
  // Axios.prototype.request 内对第一个参数的数据类型判断，使我们能够以 instance(url, option) 方式调用
  var instance = bind(Axios.prototype.request, context);

  // 把Axios.prototype上的方法扩展到instance对象上，
  // 并指定上下文为context，这样执行Axios原型链上的方法时，this会指向context
  utils.extend(instance, Axios.prototype, context);

  // Copy context to instance
  // 把context对象上的自身属性和方法扩展到instance上
  // 注：因为extend内部使用的forEach方法对对象做for in 遍历时，只遍历对象本身的属性，而不会遍历原型链上的属性
  // 这样，instance 就有了  defaults、interceptors 属性。
  utils.extend(instance, context);
  return instance;
}

// Create the default instance to be exported 创建一个由默认配置生成的axios实例
var axios = createInstance(defaults);

// Factory for creating new instances 扩展axios.create工厂函数，内部也是 createInstance
axios.create = function create(instanceConfig) {
  return createInstance(mergeConfig(axios.defaults, instanceConfig));
};

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

axios.spread = function spread(callback) {
  return function wrap(arr) {
    return callback.apply(null, arr);
  };
};
module.exports = axios;
```

主要核心是 `Axios.prototype.request`，各种请求方式的调用实现都是在 `request` 内部实现的， 简单看下 `request` 的逻辑

JavaScript | 复制代码

```javascript
Axios.prototype.request = function request(config) {
  // Allow for axios('example/url'[, config]) a la fetch API
  // 判断 config 参数是否是 字符串，如果是则认为第一个参数是 URL，第二个参数是真正的config
  if (typeof config === 'string') {
    config = arguments[1] || {};
    // 把 url 放置到 config 对象中，便于之后的 mergeConfig
    config.url = arguments[0];
  } else {
    // 如果 config 参数是否是 字符串，则整体都当做config
    config = config || {};
  }
  // 合并默认配置和传入的配置
  config = mergeConfig(this.defaults, config);
  // 设置请求方法
  config.method = config.method ? config.method.toLowerCase() : 'get';
  /*
    something... 此部分会在后续拦截器单独讲述
  */
};

// 在 Axios 原型上挂载 'delete', 'get', 'head', 'options' 且不传参的请求方法，实现内部也是 request
utils.forEach(['delete', 'get', 'head', 'options'], function forEachMethodNoData(method) {
  Axios.prototype[method] = function(url, config) {
    return this.request(utils.merge(config || {}, {
      method: method,
      url: url
    }));
  };
});

// 在 Axios 原型上挂载 'post', 'put', 'patch' 且传参的请求方法，实现内部同样也是 request
utils.forEach(['post', 'put', 'patch'], function forEachMethodWithData(method) {
  Axios.prototype[method] = function(url, data, config) {
    return this.request(utils.merge(config || {}, {
      method: method,
      url: url,
      data: data
    }));
  };
});
```

`request` 入口参数为 `config`，可以说 `config` 贯彻了 `axios` 的一生

`axios` 中的 `config` 主要分布在这几个地方：

- 默认配置 `defaults.js`
- `config.method` 默认为 `get`
- 调用 `createInstance` 方法创建 `axios` 实例，传入的 `config`
- 直接或间接调用 `request` 方法，传入的 `config`

JavaScript | 复制代码

```
// axios.js
// 创建一个由默认配置生成的axios实例
var axios = createInstance(defaults);

// 扩展axios.create工厂函数，内部也是 createInstance
axios.create = function create(instanceConfig) {
  return createInstance(mergeConfig(axios.defaults, instanceConfig));
};

// Axios.js
// 合并默认配置和传入的配置
config = mergeConfig(this.defaults, config);
// 设置请求方法
config.method = config.method ? config.method.toLowerCase() : 'get';
```

从源码中，可以看到优先级：默认配置对象 `default` < `method:get` < `Axios` 的实例属性 `this.default` < `request` 参数

下面重点看看 `request` 方法

JavaScript | 复制代码

```
Axios.prototype.request = function request(config) {
  /*
    先是 mergeConfig ... 等，不再阐述
  */
  // Hook up interceptors middleware 创建拦截器链. dispatchRequest 是重中之重，后续重点
  var chain = [dispatchRequest, undefined];

  // push各个拦截器方法 注意：interceptor.fulfilled 或 interceptor.rejected 是可能为undefined
  this.interceptors.request.forEach(function unshiftRequestInterceptors(interceptor) {
    // 请求拦截器逆序 注意此处的 forEach 是自定义的拦截器的forEach方法
    chain.unshift(interceptor.fulfilled, interceptor.rejected);
  });

  this.interceptors.response.forEach(function pushResponseInterceptors(interceptor) {
    // 响应拦截器顺序 注意此处的 forEach 是自定义的拦截器的forEach方法
    chain.push(interceptor.fulfilled, interceptor.rejected);
  });

  // 初始化一个promise对象，状态为resolved，接收到的参数为已经处理合并过的config对象
  var promise = Promise.resolve(config);

  // 循环拦截器的链
  while (chain.length) {
    promise = promise.then(chain.shift(), chain.shift()); // 每一次向外弹出拦截器
  }
  // 返回 promise
  return promise;
};
```

拦截器 `interceptors` 是在构建 `axios` 实例化的属性

```JavaScript
function Axios(instanceConfig) {
  this.defaults = instanceConfig;
  this.interceptors = {
    request: new InterceptorManager(), // 请求拦截
    response: new InterceptorManager() // 响应拦截
  };
}
```

`InterceptorManager` 构造函数

```JavaScript
// 拦截器的初始化 其实就是一组钩子函数
function InterceptorManager() {
  this.handlers = [];
}

// 调用拦截器实例的use时就是往钩子函数中push方法
InterceptorManager.prototype.use = function use(fulfilled, rejected) {
  this.handlers.push({
    fulfilled: fulfilled,
    rejected: rejected
  });
  return this.handlers.length - 1;
};

// 拦截器是可以取消的，根据use的时候返回的ID，把某一个拦截器方法置为null
// 不能用 splice 或者 slice 的原因是 删除之后 id 就会变化，导致之后的顺序或者是操作不可控
InterceptorManager.prototype.eject = function eject(id) {
  if (this.handlers[id]) {
    this.handlers[id] = null;
  }
};

// 这就是在 Axios的request方法中 中循环拦截器的方法 forEach 循环执行钩子函数
InterceptorManager.prototype.forEach = function forEach(fn) {
  utils.forEach(this.handlers, function forEachHandler(h) {
    if (h !== null) {
      fn(h);
    }
  });
}
```

请求拦截器方法是被 `unshift` 到拦截器中，响应拦截器是被 `push` 到拦截器中的。最终它们会拼接上一个叫 `dispatchRequest` 的方法被后续的 `promise` 顺序执行

JavaScript | 复制代码

```
var utils = require('./../utils');
var transformData = require('./transformData');
var isCancel = require('../cancel/isCancel');
var defaults = require('../defaults');
var isAbsoluteURL = require('./../helpers/isAbsoluteURL');
var combineURLs = require('./../helpers/combineURLs');

// 判断请求是否已被取消，如果已经被取消，抛出已取消
function throwIfCancellationRequested(config) {
  if (config.cancelToken) {
    config.cancelToken.throwIfRequested();
  }
}

module.exports = function dispatchRequest(config) {
  throwIfCancellationRequested(config);

  // 如果包含baseUrl，并且不是config.url绝对路径，组合baseUrl以及config.url
  if (config.baseURL && !isAbsoluteURL(config.url)) {
    // 组合baseURL与url形成完整的请求路径
    config.url = combineURLs(config.baseURL, config.url);
  }

  config.headers = config.headers || {};

  // 使用/lib/defaults.js中的transformRequest方法，对config.headers和config.data进行格式化
  // 比如将headers中的Accept，Content-Type统一处理成大写
  // 比如如果请求正文是一个Object会格式化为JSON字符串，并添加application/json;charset=utf-8的Content-Type
  // 等一系列操作
  config.data = transformData(
    config.data,
    config.headers,
    config.transformRequest
  );

  // 合并不同配置的headers，config.headers的配置优先级更高
  config.headers = utils.merge(
    config.headers.common || {},
    config.headers[config.method] || {},
    config.headers || {}
  );

  // 删除headers中的method属性
```

```
  utils.forEach(
    ['delete', 'get', 'head', 'post', 'put', 'patch', 'common'],
    function cleanHeaderConfig(method) {
      delete config.headers[method];
    }
  );

  // 如果config配置了adapter，使用config中配置adapter的替代默认的请求方法
  var adapter = config.adapter || defaults.adapter;

  // 使用adapter方法发起请求（adapter根据浏览器环境或者Node环境会有不同）
  return adapter(config).then(
    // 请求正确返回的回调
    function onAdapterResolution(response) {
      // 判断是否以及取消了请求，如果取消了请求抛出以取消
      throwIfCancellationRequested(config);

      // 使用/lib/defaults.js中的transformResponse方法，对服务器返回的数据进行格式化
      // 例如，使用JSON.parse对响应正文进行解析
      response.data = transformData(
        response.data,
        response.headers,
        config.transformResponse
      );

      return response;
    },
    // 请求失败的回调
    function onAdapterRejection(reason) {
      if (!isCancel(reason)) {
        throwIfCancellationRequested(config);

        if (reason && reason.response) {
          reason.response.data = transformData(
            reason.response.data,
            reason.response.headers,
            config.transformResponse
          );
        }
      }
      return Promise.reject(reason);
    }
  );
};
```

再来看看 `axios` 是如何实现取消请求的，实现文件在 `CancelToken.js`

JavaScript | 复制代码

```javascript
function CancelToken(executor) {
  if (typeof executor !== 'function') {
    throw new TypeError('executor must be a function.');
  }
  // 在 CancelToken 上定义一个 pending 状态的 promise ，将 resolve 回调赋值给外部变量 resolvePromise
  var resolvePromise;
  this.promise = new Promise(function promiseExecutor(resolve) {
    resolvePromise = resolve;
  });

  var token = this;
  // 立即执行 传入的 executor函数，将真实的 cancel 方法通过参数传递出去。
  // 一旦调用就执行 resolvePromise 即前面的 promise 的 resolve，就更改promise的状态为 resolve。
  // 那么xhr中定义的 CancelToken.promise.then方法就会执行，从而xhr内部会取消请求
  executor(function cancel(message) {
    // 判断请求是否已经取消过，避免多次执行
    if (token.reason) {
      return;
    }
    token.reason = new Cancel(message);
    resolvePromise(token.reason);
  });
}

CancelToken.source = function source() {
  // source 方法就是返回了一个 CancelToken 实例，与直接使用 new CancelToken 是一样的操作
  var cancel;
  var token = new CancelToken(function executor(c) {
    cancel = c;
  });
  // 返回创建的 CancelToken 实例以及取消方法
  return {
    token: token,
    cancel: cancel
  };
};
```

实际上取消请求的操作是在 `xhr.js` 中也有响应的配合的