axios API

代码示例

1. 向给定 ID 的用户发起 GET 请求

```
axios.get('/user?ID=12345')
  .then(function (response) {
    // 处理成功情况
    console.log(response);
  })
  .catch(function (error) {
    // 处理错误情况
    console.log(error);
  })
  .finally(function () {
    // 总是会执行
  });
```

```
axios.get('/user', {
    params: {
      ID: 12345
    }
  })
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  })
  .finally(function () {
    // 总是会执行
  });
```

```
async function getUser() {
  try {
    const response = await axios.get('/user?ID=12345');
    console.log(response);
  } catch (error) {
    console.error(error);
  }
}
```

2. 向用户集合发起一个 POST 请求

```
axios.post('/user', {
    firstName: 'Fred',
    lastName: 'Flintstone'
  })
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });
```

3. 发起多个并发请求

```
function getUserAccount() {
  return axios.get('/user/12345');
}

function getUserPermissions() {
  return axios.get('/user/12345/permissions');
}

const [acct, perm] = await Promise.all([getUserAccount(),
getUserPermissions()]);
```

HTML Form 数据的处理方式

1. 将 HTML 表单(Form) 转换成 JSON 进行请求

```
const {data} = await axios.post('/user', document.querySelector('#my-form'),
{
  headers: {
    'Content-Type': 'application/json'
  }
})
```

2. 表单数据的处理方式之 `mutipart/form-data`：此请求头常用于表单数据中需要包含文件上传时

```
const {data} = await axios.post('https://httpbin.org/post', {
    firstName: 'Fred',
    lastName: 'Flintstone',
    orders: [1, 2, 3],
    photo: document.querySelector('#fileInput').files
  }, {
    headers: {
      'Content-Type': 'multipart/form-data'
    }
  }
)
```

3. 表单数据的处理方式之 `applicaton/x-www-form-urlencoded`：此请求头常用于表单数据仅包含简单的文字数据时，会将表单数据编码为 URL 兼容格式，即 `key1=value1&key2=value2` 的形式

```
const {data} = await axios.post('https://httpbin.org/post', {
    firstName: 'Fred',
    lastName: 'Flintstone',
    orders: [1, 2, 3]
  }, {
    headers: {
      'Content-Type': 'application/x-www-form-urlencoded'
    }
})
```

静态方法

`axios(config)`

`axios(url[, config])`

`axios.request(config)`：与 `axios(config)` 相同。

`axios.get(url[, config])`

`axios.delete(url[, config])`

`axios.head(url[, config])`

`axios.options(url[, config])`

`axios.post(url[, data[, config]])`

`axios.put(url[, data[, config]])`

`axios.patch(url[, data[, config]])`

`axios.postForm(url[, data[, config]])`

`axios.putForm(url[, data[, config]])`

`axios.patchForm(url[, data[, config]])`

> 除了 `axios` 函数，剩下的方法称之为**别名方法**，使用时不用在 `config` 中指定 `method`、`url`、`data` 等字段

实例与实例方法

我们可以使用**自定义配置**新建一个**实例** `axios.create([config])`，然后通过**实例方法**发送请求，实例方法的配置将与实例的配置**合并**。

> p.s. 这里创建的实例和 `axios` 类似，也可以作为一个函数使用，即 `axios.create([config])(config)`

`.request(config)`

`.get(url[, config])`

`.delete(url[, config])`

`.head(url[, config])`

`.options(url[, config])`

`.post(url[, data[, config]])`

`.put(url[, data[, config]])`

`.patch(url[, data[, config]])`

`.getUri([config])`

请求配置

请求配置即创建请求时使用的配置选项 `config`，其中只有 `url` 是必需的，`method` 如未指定则默认使用 GET

```
{
    // 第一部分：基础配置选项 (Basic Configuration Options)
    // `url` 请求的服务器 URL，唯一必须指定的字段
    url: '/user',
    // `method` 请求方法，默认值为 'GET'
    method: 'get', // 默认值
    // `baseURL` 将被添加到 url 前面,除非 url 是绝对路径
    // 可以为 axios 实例设置 baseURL`，以便在实例方法中传递相对 URL
    baseURL: 'https: //some-domain.com/api/',
    // 第二部分：请求修改 (Request Transformation)
    // `transformRequest` 在请求数据发送到服务器之前,允许对其进行修改
    // 这只适用于请求方法 'PUT'、'POST'、'PATCH' 和 'DELETE'
    // 数组中的最后一个函数必须返回一个字符串、Buffer 实例、ArrayBuffer、FormData 或
Stream
    // 你可以修改请求头
    transformRequest: [function (data, headers) {
            // 对发送的 data 进行任意转换处理

      return data;
        }
    ],
    // `transformResponse` 在响应数据传递给 then/catch 之前,允许对其进行修改
    transformResponse: [function (data) {
            // 对接收的 data 进行任意转换处理

      return data;
        }
    ],
    // 第三部分：请求头和 URL 参数 (Headers and Params)
    // 自定义请求头
    headers: {'X-Requested-With': 'XMLHttpRequest'
    },
    // `params` 是与请求一起发送的 URL 参数
    // 必须是一个简单对象或 URLSearchParams 对象
    // 补充说明：URLSearchParams 是 JS 内置对象，用于处理 URL 查询字符串 (query
string)，它提供了一组方法，用于解析、操作和构建 URL 查询参数。

    params: {
      ID: 12345
    },
```

```
    // `paramsSerializer`是可选方法，主要用于自定义序列化`params`
    // 通常，当你通过 params 选项将参数传递给 Axios 请求时，Axios 会将这些参数序列化为 URL 查询字符串并添加到请求的 URL 中
    // 这个自定义的序列化函数接受一个包含参数的对象作为参数，并返回一个字符串，这个字符串就是最终的 URL 查询字符串。你可以在这个函数中实现任意的逻辑来对参数进行处理，然后返回符合要求的字符串。
    // 序列化：将数据结构或对象转换为一种特定格式的过程，以便于存储、传输或展示
    // (e.g. https://www.npmjs.com/package/qs, http://api.jquery.com/jquery.param/)
    paramsSerializer: function (params) {
      return Qs.stringify(params,
        {arrayFormat: 'brackets'
        })
    },
    // 第四部分：请求体 (Request Body)
    // `data` 是作为请求体被发送的数据
    // 仅适用 'PUT', 'POST', 'DELETE 和 'PATCH' 请求方法
    // 在没有设置 `transformRequest` 时，则必须是以下类型之一：
    // - string, plain object, ArrayBuffer, ArrayBufferView, URLSearchParams
    // - 浏览器专属: FormData, File, Blob
    // - Node 专属: Stream, Buffer
    data: {
      firstName: 'Fred'
    },
    // 发送请求体数据的可选语法
    // 请求方式 post
    // 只有 value 会被发送，key 则不会
    data: 'Country=Brasil&City=Belo Horizonte',
    // 第五部分：最大请求时间和跨域请求 (Timeout and Cross-Site Requests)
    // `timeout` 指定请求超时的毫秒数。
    // 如果请求时间超过 timeout，请求将被中止。
    timeout: 1000, // 默认值是 `0` (永不超时)
    // `withCredentials` 表示跨域请求时是否需要使用凭证
    // true 则意味着在跨域请求中，浏览器会发送当前域的 cookie 等凭据信息给服务器
    withCredentials: false, // 默认值
    // 第六部分：适配器函数和 HTTP 基本验证 (Adapters and Auth)
    // `adapter` 允许自定义处理请求，这使测试更加容易。
    // 返回一个 promise 并提供一个有效的响应 （参见 lib/adapters/README.md）。
    // 适配器函数是一个能够处理发送 HTTP 请求并返回 Promise 对象的函数，它允许你完全控制 Axios 如何发送请求。
    // 当你提供了一个自定义的适配器函数给 adapter 配置选项时，Axios 将会使用这个函数来处理请求的发送。
    // 这个自定义的适配器函数需要接受一个 Axios 请求配置对象作为参数，然后返回一个 Promise 对象。
    // 通常情况下，你不需要自定义适配器函数，因为 Axios 已经为大多数常见的环境提供了默认的适配器。
    adapter: function (config) {
        /* ... */
    },
    // `auth` 指示应该使用 HTTP 基本认证（HTTP Basic auth），并提供相应的凭据
    // 当你提供了 auth 配置选项时，Axios 将会在请求中添加一个 Authorization 头部，其值为 Basic 加密后的凭据。这个凭据是由提供的用户名和密码经过 Base64 编码得到的。这样，服务器就可以通过解码凭据并验证用户名和密码来进行用户认证。
    // auth 配置选项仅支持 HTTP 基本认证（HTTP Basic auth），并不支持其他类型的认证方式，例如 Bearer token。如果你需要使用其他类型的认证方式，例如 Bearer token，你应该使用自定义的 Authorization 头部来设置。Axios 提供了 headers 配置选项，你可以通过设置 headers.Authorization 来添加自定义的认证头部。
```

```
    auth: {
      username: 'janedoe',
      password: 's00pers3cret'
    },
    // 第七部分：响应类型 (Response Type)
    // `responseType` 表示浏览器将要响应的数据类型
    // 选项包括: 'arraybuffer', 'document', 'json', 'text', 'stream'
    // 浏览器专属: 'blob'
    responseType: 'json', // 默认值
    // `responseEncoding` 表示用于解码响应的编码 (Node.js 专属)
    // 注意：当 `responseType` 的值为 'stream'，或者是客户端请求时忽略
    // Note: Ignored for `responseType` of 'stream' or client-side requests
    // 选项包括：'ascii', 'ASCII', 'ansi', 'ANSI', 'binary', 'BINARY', 'base64',
'BASE64', 'base64url', 'BASE64URL', 'hex', 'HEX', 'latin1', 'LATIN1', 'ucs-2',
'UCS-2', 'ucs2', 'UCS2', 'utf-8', 'UTF-8', 'utf8', 'UTF8', 'utf16le', 'UTF16LE'
    responseEncoding: 'utf8', // 默认值
    // 第八部分：安全 (Security)
    // XSRF(Cross-Site Request Forgery) 译为跨站请求伪造
    // XSRF token 是一种安全措施，用于防止跨站请求伪造攻击。
    // `xsrfCookieName` 是 xsrf token 的值，被用作 cookie 的名称
    xsrfCookieName: 'XSRF-TOKEN', // 默认值
    // `xsrfHeaderName` 是带有 xsrf token 值的 http 请求头名称
    xsrfHeaderName: 'X-XSRF-TOKEN', // 默认值
    // `withXSRFToken` 布尔值或者函数，用于确定是否应该在请求中包含 XSRF token，默认值为
undefined
    // 默认情况下，XSRF token 仅在同源请求中发送。如果你想要在跨源请求中也发送 XSRF token，
你可以将这个配置选项设置为 true。你还可以提供一个函数来动态确定是否应该发送 XSRF token，这个函
数接受请求配置对象作为参数，并返回一个布尔值来表示是否应该发送 XSRF token。
    withXSRFToken: boolean | undefined | ((config: InternalAxiosRequestConfig) =>
boolean | undefined),
    // 第九部分：进度事件 (Progress Event)
    // `onUploadProgress` 用于处理上传进度事件的回调函数
    // 无论是在浏览器环境还是在 Node.js 环境下，当上传进度发生变化时，Axios 将调用这个回调函
数，并将包含上传进度信息的对象传递给它。这个对象包含以下属性：
    // - loaded：已上传的字节数。
    // - total：需要上传的总字节数。
    // - progress：已上传字节数与总字节数的比例，范围从 0 到 1。
    // - bytes：每秒上传的字节数。
    // - estimated：剩余上传时间的估计，单位为秒。
    // - rate：上传速率，单位为字节每秒。
    // - upload：一个布尔值，表示上传是否处于活动状态，默认为 true。
    onUploadProgress: function (progressEvent) {
        // 处理原生进度事件
    },
    // `onDownloadProgress` 用于处理下载进度事件的回调函数。
    // 与 onUploadProgress 类似，当下载进度发生变化时，Axios 将调用这个回调函数，并将包含下
载进度信息的对象传递给它。这个对象包含的属性与 onUploadProgress 中的相同，只是对应下载操作。
    onDownloadProgress: function (progressEvent) {
        // 处理原生进度事件
    },
    // 第十部分：内容长度 (Content Length)
    // `maxContentLength` 定义了 node.js 中允许的 HTTP 响应内容的最大字节数
    maxContentLength: 2000,
    // `maxBodyLength`（仅Node）定义允许的 http 请求内容的最大字节数
    maxBodyLength: 2000,
    // 第十一部分：状态验证 (Status Validation)
```

```
    // `validateStatus` 定义了对于给定的 HTTP 状态码是 resolve 还是 reject promise。
    // 如果 `validateStatus` 返回 `true` (或者设置为 `null` 或 `undefined`)，
    // 则 promise 将会 resolved，否则是 rejected。
    validateStatus: function (status) {
      return status >= 200 && status < 300; // 默认值
    },
    // 第十二部份：重定向 (Redirects)
    // `maxRedirects` 定义在 Node.js 中要遵循的最大重定向数。
    // 如果设置为 0，则不会进行重定向
    maxRedirects: 5, // 默认值
    // `beforeRedirect` 定义一个在重定向之前被调用的函数
    // 用于在重定向时调整请求选项，检查最新的响应头，或通过抛出错误来取消请求。
    // 如果 maxRedirects 设置为 0,则不使用 beforeRedirect。
    beforeRedirect: (options,
    { headers
    }) => {
        if (options.hostname === "example.com") {
          options.auth = "user:password";
        }
    },
    // 第十三部份：网络 (Network)
    // `socketPath` 用于在 Node.js 中定义要使用的 UNIX Socket
    // UNIX Socket 是一种在本地计算机上进行进程间通信的机制
    // 通过指定 socketPath，你可以将请求发送到指定的 UNIX Socket 地址。比如，你可以将请求发送到 Docker 守护进程等。默认情况下，socketPath 为 null
    // e.g. '/var/run/docker.sock' 发送请求到 docker 守护进程
    // 只能指定 socketPath 或 proxy 中的一个。如果都指定，则使用 socketPath。
    socketPath: null, // default
    // `transport`  确定将用于发出请求的传输方法
    // 如果定义了此选项，则将使用指定的传输方法。
    // 否则，如果 maxRedirects 为 0，则将使用默认的 http 或 https 库，具体取决于 protocol 中指定的协议。
    // 否则，将使用 httpFollow 或 httpsFollow 库，这取决于协议，该库可以处理重定向。
    transport: undefined, // default
    // `httpAgent` and `httpsAgent` 定义在 Node.js 中执行 HTTP 和 HTTPS 请求时要使用的自定义代理
    // 这允许添加一些默认情况下未启用的选项，例如 keepAlive。你可以创建自定义的 HTTP 和 HTTPS 代理，并在执行请求时使用它们，以控制请求的行为。
    httpAgent: new http.Agent({ keepAlive: true
    }),
    httpsAgent: new https.Agent({ keepAlive: true
    }),
    // 第十四部分：代理配置 (Proxy Configuration)
    // `proxy` 定义了代理服务器的主机名，端口和协议。
    // 你也可以使用常规的 http_proxy 和 https_proxy 环境变量来定义你的代理。如果你使用环境变量进行代理配置，你还可以定义一个 no_proxy 环境变量，其值为一个以逗号分隔的不应被代理的域的列表。
    // 使用 `false` 可以禁用代理功能，同时环境变量也会被忽略。
    // auth 表示应使用 HTTP Basic auth 连接到代理，并提供凭据。
    // 这将设置一个 Proxy-Authorization 头，覆写掉已经存在的通过 headers 设置的自定义 Proxy-Authorization 头。
    // 如果代理服务器使用 HTTPS，那么必须设置 protocol 为 https。
    proxy: {
        protocol: 'https',
        host: '127.0.0.1',
        port: 9000,
```

```
        auth: {
            username: 'mikeymike',
            password: 'rapunz3l'
        }
    },
    // 第十五部分：请求取消 (Request Cancellation)
    // cancelToken: 指定可用于取消请求的取消 token
    // see https://axios-http.com/zh/docs/cancellation
    cancelToken: new CancelToken(function (cancel) {}),
    // signal 使用 AbortController 取消 Axios 请求（替代方法）
    signal: new AbortController().signal,
    // 第十六部份：其他
    // `decompress` 指示是否应自动解压缩响应体
    // 如果设置为 true，还将从所有解压缩响应的响应对象中删除 'content-encoding' 头。
    // 仅适用于 Node (XHR 无法关闭解压缩)。
    decompress: true // 默认值


      // `insecureHTTPParser` boolean.
    // 指示在何处使用不安全的 HTTP 解析器，该解析器接受无效的 HTTP 头
    // 这可能允许与不符合规范的 HTTP 实现进行互操作
    // 应避免使用不安全的解析器。
    // see options https://nodejs.org/dist/latest-
v12.x/docs/api/http.html#http_http_request_url_options_callback
    // see also https://nodejs.org/en/blog/vulnerability/february-2020-security-
releases/#strict-http-header-parsing-none
      insecureHTTPParser: undefined, // default
    // transitional 用于向后兼容的过渡选项,可能会在新版本中删除。
      transitional: {
        // silentJSONParsing: silent JSON parsing mode，静默 JSON 解析模式。
        // true - 忽略 JSON 解析错误，如果解析失败，将 response.data 设置为 null(旧行为)
        // false - 如果 JSON 解析失败，则抛出 SyntaxError
        // 注意：responseType 必须设置为 'json'
        silentJSONParsing: true, // default value for the current Axios version
        // forcedJSONParsing: 即使 responseType 不是 'json',也要尝试将响应字符串解析为
JSON。
        forcedJSONParsing: true,
        // clarifyTimeoutError: 在请求超时时抛出 ETIMEDOUT 错误而不是通用的
ECONNABORTED。
        clarifyTimeoutError: false,
    },
    // 在不同的环境中自定义一些全局变量或类。
        env: {
        // 指定在自动将请求数据序列化为 FormData 对象时要使用的 FormData 类
            FormData: window?.FormData || global?.FormData
    },
    // 自定义表单数据的序列化方式。
    // 这在发送 POST 请求时特别有用，当请求数据需要以表单形式传输时，可以使用这个选项来控制数据
的序列化方式。
        formSerializer: {
            visitor: (value, key, path, helpers) => {}; // 自定义的访问者函数，用于序
列化表单值。当序列化过程中遍历到每个表单值时，将会调用该函数，并传递当前值、键名、路径和帮助函数作
为参数。你可以在这个函数中对表单值进行定制化的处理。
            dots: boolean; // 布尔值，用于指定是否使用点号（.）而不是传统的方括号格式来表示
对象的属性。默认为 false，表示使用方括号格式
```

```
            metaTokens: boolean; // 布尔值，用于指定是否保留参数键名中的特殊结束符号（比如{}）。默认为 false，表示不保留。
            indexes: boolean; // 枚举值，用于指定数组索引的格式化方式。可以设置为 null（表示不使用方括号）、false（表示使用空方括号）或 true（表示使用带有索引的方括号）。
    },
    // http adapter only (node.js)
    // 限制请求和响应的传输速率，以控制数据的上传和下载速度。
    // maxRate 是一个数组，包含两个元素，分别表示上传和下载的最大速率限制。这两个值的单位是字节每秒（bytes per second）。
    // 这些限制可以帮助控制请求和响应的速度，防止网络拥塞或过度消耗带宽
        maxRate: [
        100 * 1024, // 100KB/s upload limit,
        100 * 1024 // 100KB/s download limit
    ]
}
```

响应结构

`axios` 通过实例方法或静态方法发送请求返回的成功的 Promise 对象的结果就是包装好的响应结果，结构如下

```
{
  // `data` 由服务器提供的响应
  data: {},

  // `status` 来自服务器响应的 HTTP 状态码
  status: 200,

  // `statusText` 来自服务器响应的 HTTP 状态信息
  statusText: 'OK',

  // `headers` 是服务器响应头
  // 所有的 header 名称都是小写，而且可以使用方括号语法访问
  // 例如: `response.headers['content-type']`
  headers: {},

  // `config` 是 `axios` 请求的配置信息
  config: {},

  // `request` 是生成此响应的请求
  // 在 node.js 中它是最后一个 ClientRequest 实例 (in redirects),
  // 在浏览器中则是 XMLHttpRequest 实例
  request: {}
}
```

默认配置

您可以**指定默认配置**，它将作用于每个请求，此外，`axios` 发送请求时传入的 `config` 中的配置选项会覆盖默认配置

1. 全局默认配置

```
axios.defaults.baseURL = 'https://api.example.com';
axios.defaults.headers.common['Authorization'] = AUTH_TOKEN;
axios.defaults.headers.post['Content-Type'] = 'application/x-www-form-urlencoded';
```

2. 实例默认配置

```
// 创建实例时配置默认值
const instance = axios.create({
  baseURL: 'https://api.example.com'
});

// 创建实例后修改默认值
instance.defaults.headers.common['Authorization'] = AUTH_TOKEN;
```

> `instance.defaults.headers.common[请求头] = 值` 表示设置一个在所有类型的请求中都生效的默认请求头
>
> `instance.default.headers.post[请求头] = [请求值]` 表示设置一个仅在 GET 类型的请求中生效的请求头

3. 配置合并优先级：**请求中的配置 > 设置的默认配置 (通过 defaults) > 库中的默认值**

拦截器

拦截器可用于在请求或响应被 then 或 catch 处理前拦截它们。其中多个请求拦截器的调用是逆序的，多个响应拦截器的调用是顺序的，这与 Axios.js 中将请求 (unshift)、响应 (push) 拦截器的函数添加到 chain 数组中的方式有关。

1. 添加请求或响应拦截器（给 `axios`）

```
// 添加请求拦截器
axios.interceptors.request.use(function (config) {
    // 在发送请求之前做些什么
    return config;
  }, function (error) {
    // 对请求错误做些什么
    return Promise.reject(error);
  });

// 添加响应拦截器
axios.interceptors.response.use(function (response) {
    // 2xx 范围内的状态码都会触发该函数。
    // 对响应数据做点什么
    return response;
  }, function (error) {
    // 超出 2xx 范围的状态码都会触发该函数。
    // 对响应错误做点什么
    return Promise.reject(error);
  });
```

2. 添加请求或响应拦截器（给实例）

```
const instance = axios.create();
instance.interceptors.request.use(function () {/*...*/});
```

3. 移除拦截器

```
const myInterceptor = axios.interceptors.request.use(function () {/*...*/});
axios.interceptors.request.eject(myInterceptor);
```

错误处理

1. 默认情况下的错误处理流程：`error.response` => `error.request` => `error.message`

```
axios.get('/user/12345')
  .catch(function (error) {
    if (error.response) {
      // 请求成功发出且服务器也响应了状态码，但状态代码超出了 2xx 的范围
      console.log(error.response.data);
      console.log(error.response.status);
      console.log(error.response.headers);
    } else if (error.request) {
      // 请求已经成功发起，但没有收到响应
      // `error.request` 在浏览器中是 XMLHttpRequest 的实例，
      // 而在node.js中是 http.ClientRequest 的实例
      console.log(error.request);
    } else {
      // 发送请求时出了点问题
      console.log('Error', error.message);
    }
    console.log(error.config);
  });
```

2. 使用 `validateStatus` 配置选项，可以自定义抛出错误的 HTTP code。

```
axios.get('/user/12345', {
  validateStatus: function (status) {
    return status < 500; // 处理状态码小于500的情况
  }
})
```

3. 使用 `toJSON` 可以获取更多关于HTTP错误的信息。

```
axios.get('/user/12345')
  .catch(function (error) {
    console.log(error.toJSON());
  });
```

取消请求

1. 新方式： fetch API

```
const controller = new AbortController(); // step1

axios.get('/foo/bar', {
   signal: controller.signal // step2
}).then(function(response) {
   //...
});
// 取消请求
controller.abort() // step3
```

2. 旧方式：cancel token API

```
const CancelToken = axios.CancelToken; // step1
const source = CancelToken.source(); // step2

axios.get('/user/12345', {
  cancelToken: source.token // step3
}).catch(function (thrown) {
  if (axios.isCancel(thrown)) {
    console.log('Request canceled', 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.'); // step4
```

```
const CancelToken = axios.CancelToken; // step1
let cancel; // step2

axios.get('/user/12345', {
  cancelToken: new CancelToken(function executor(c) {
    // executor 函数接收一个 cancel 函数作为参数
    cancel = c; // step3
  })
});

// 取消请求
cancel(); // step4
```

3. 可以使用同一个 cancel token 或 signal 取消多个请求