

# 边缘安全加速平台 EO

## 边缘函数

### 产品文档



腾讯云

**【版权声明】**

©2013-2024 腾讯云版权所有

本文档著作权归腾讯云单独所有，未经腾讯云事先书面许可，任何主体不得以任何形式复制、修改、抄袭、传播全部或部分本文档内容。

**【商标声明】**

及其它腾讯云服务相关的商标均为腾讯云计算（北京）有限责任公司及其关联公司所有。本文档涉及的第三方主体的商标，依法由权利人所有。

**【服务声明】**

本文档意在向客户介绍腾讯云全部或部分产品、服务的当时的整体概况，部分产品、服务的内容可能有所调整。您所购买的腾讯云产品、服务的种类、服务标准等应由您与腾讯云之间的商业合同约定，除非双方另有约定，否则，腾讯云对本文档内容不做任何明示或模式的承诺或保证。

# 文档目录

## 边缘函数

概述

快速指引

操作指引

函数管理

触发配置

## Runtime APIs

addEventListener

Cache

Cookies

Encoding

Fetch

FetchEvent

Headers

Request

Response

Streams

ReadableStream

ReadableStreamBYOBReader

ReadableStreamDefaultReader

TransformStream

WritableStream

WritableStreamDefaultWriter

Web Crypto

Web standards

## 示例函数

返回 HTML 页面

返回 JSON

Fetch 远程资源

请求头鉴权

修改响应头

AB 测试

设置 Cookie

基于请求区域重定向

Cache API 使用

缓存 POST 请求

流式响应

合并资源流式响应

防篡改校验

m3u8 改写与鉴权

图片自适应缩放

图片自适应 WebP

自定义 Referer 防盗链

远程鉴权

HMAC 数字签名

自定义下载文件名

获取客户端 IP

最佳实践

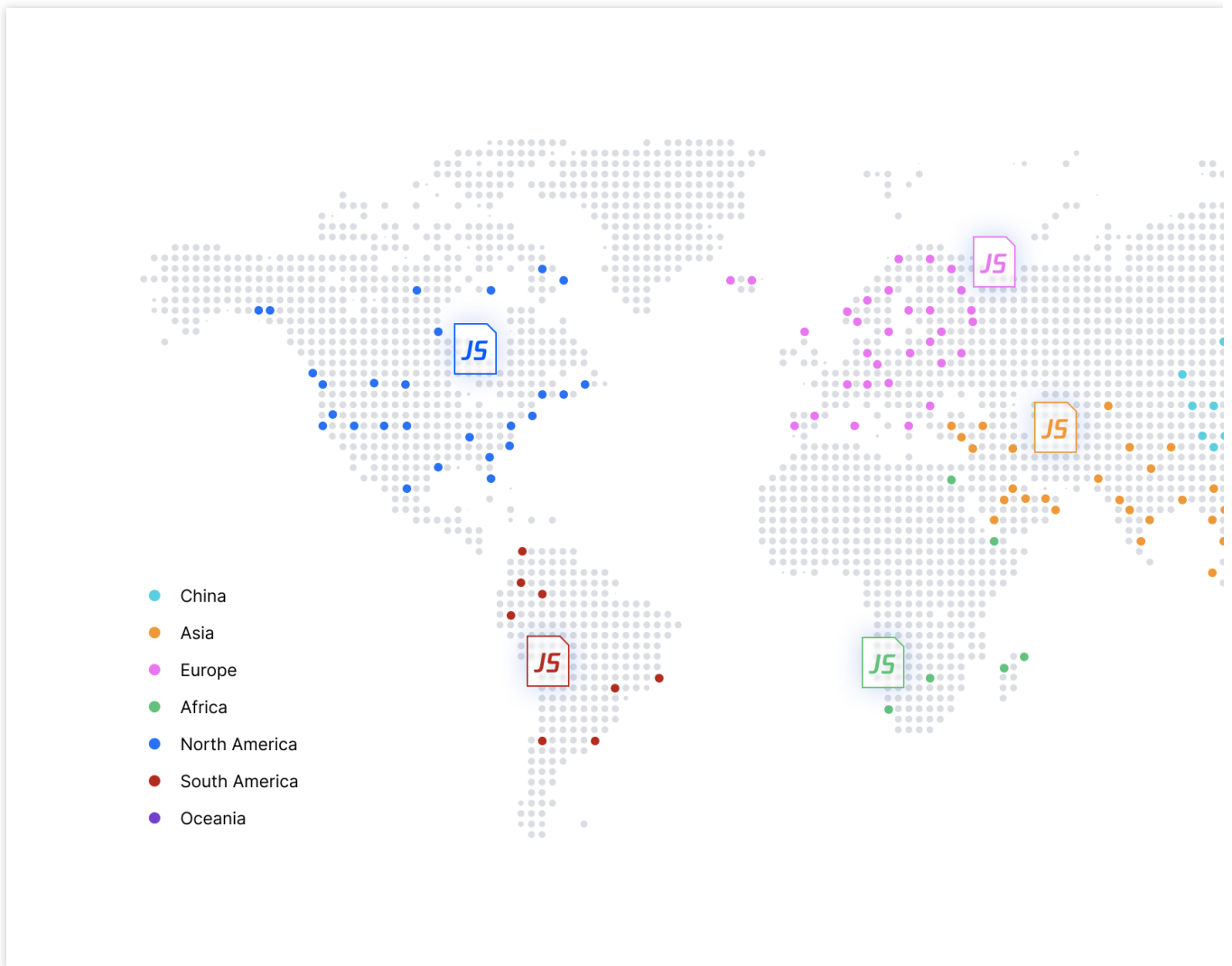
通过边缘函数实现自适应图片格式转换

# 边缘函数

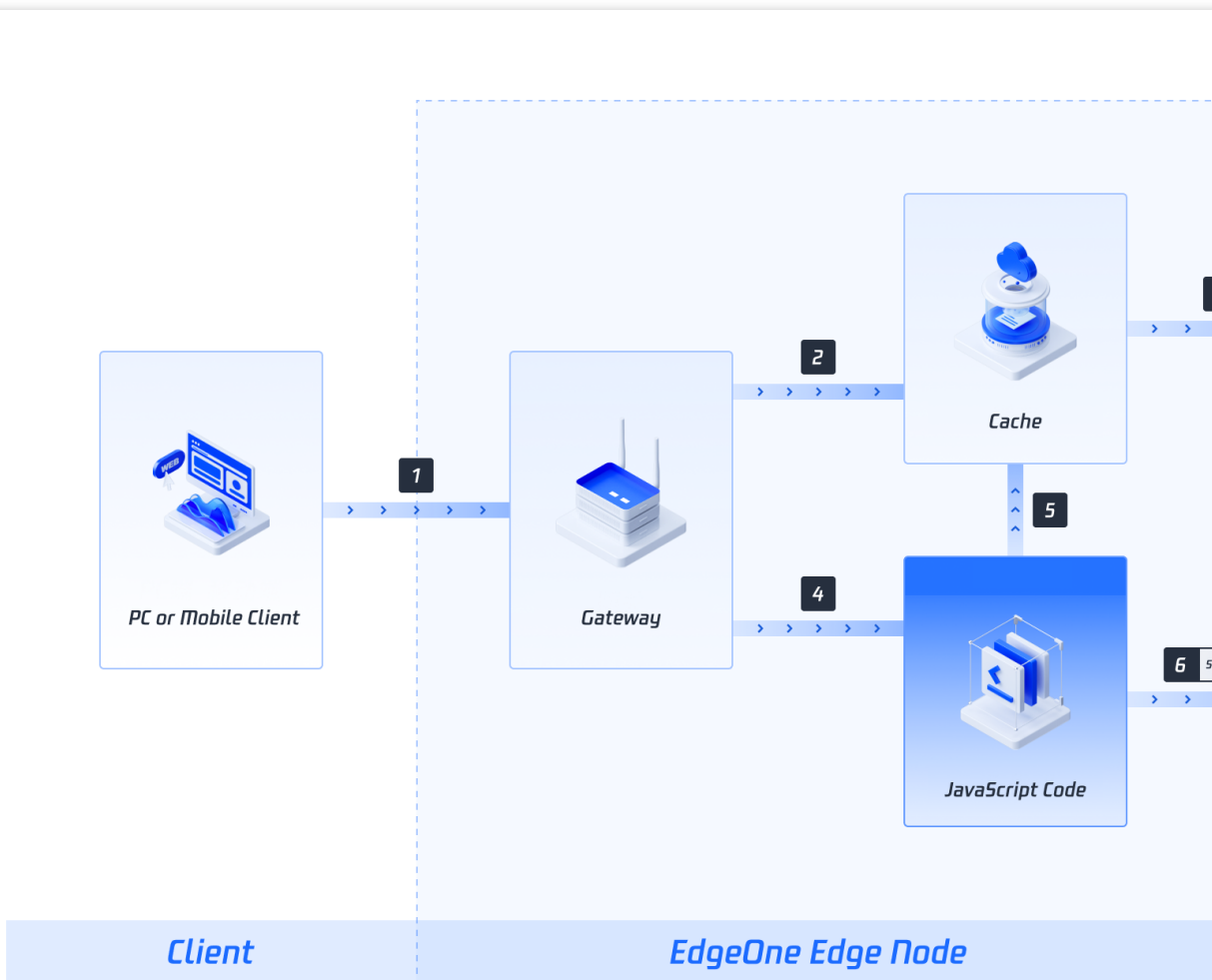
## 概述

最近更新时间：2024-01-15 15:01:15

腾讯云边缘函数（Edge Functions）提供了 EdgeOne 边缘节点的 Serverless 代码执行环境，您只需编写业务函数代码并设置触发规则，无需配置和管理服务器等基础设施，即可在靠近用户的边缘节点上弹性、安全地运行代码。



## 原理简介



您可自行开发并部署您的 JavaScript 函数至 EdgeOne 的边缘节点。

1. 当客户端请求未命中您配置的函数触发规则请求顺序为：

(1) 客户端请求 > 到达 EdgeOne 边缘节点的网关 > (2) 如节点已有缓存则缓存响应 > (3) 如缓存未命中则由源站服务器响应。

2. 当客户端请求命中您配置的函数触发规则请求顺序有如下情况：

(1) 客户端请求 > 到达 EdgeOne 边缘节点的网关 > (4) 边缘函数接管并执行您的 JS 代码 > (5) 子请求访问缓存 > (3) 缓存未命中则由源站服务器响应。

(1) 客户端请求 > 到达 EdgeOne 边缘节点的网关 > (4) 边缘函数接管并执行您的 JS 代码 > (6) 子请求访问公网服务。

## 边缘函数的优势

### 分布式部署

EdgeOne 拥有超过2800+个边缘节点，边缘函数以分布式部署的方式运行在边缘节点。

### 超低延迟

客户端请求将自动被调度至靠近您用户最近的边缘节点上，命中触发规则触发边缘函数对请求进行处理并响应结果给客户端，可显著降低客户端的访问时延。

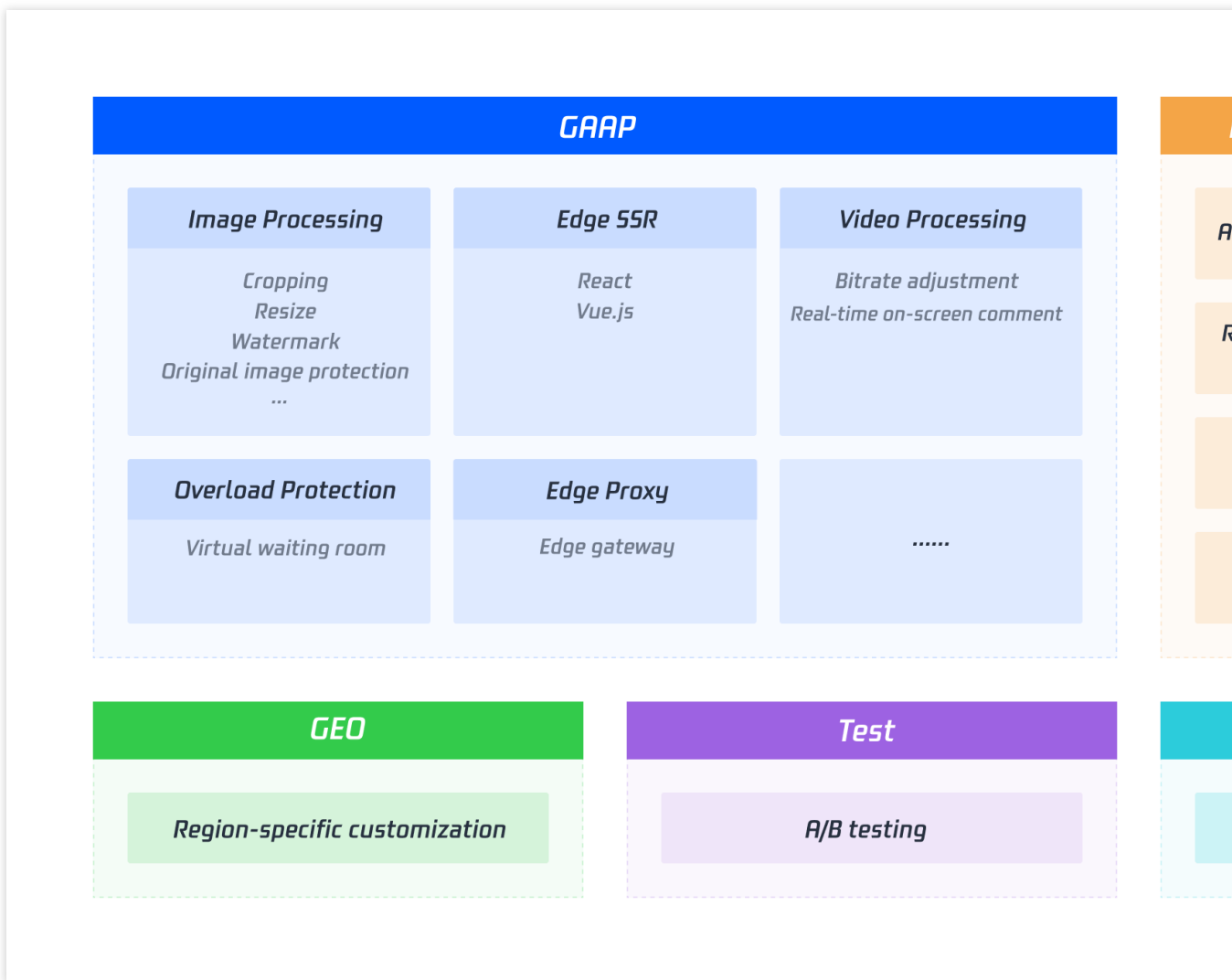
### 弹性扩容

边缘函数可以根据客户端请求数的突增，由近及远的将请求调度至有充足计算资源的边缘节点处理，您无需担忧突峰场景。

### Serverless 架构

您无需再关心和维护底层服务器的内存、CPU、网络和其他基础设施资源，可以挪出精力更专注业务代码的开发。

## 适用场景



# 快速指引

最近更新时间：2024-01-02 10:34:41

本文通过示例创建一个简单的函数，实现请求重定向到其他 URL 并返回自定义响应头，来向您介绍如何使用边缘函数。

## 示例场景

当前已将站点 `example.com` 接入 EdgeOne 服务，在该站点下，需通过自定义域名 `www.example.com` 为用户提供一个自定义的 HTML 活动页面，可通过边缘函数将该页面部署至 EdgeOne 的全球可用区边缘节点内供用户就近访问。

### 说明：

1. 如何接入站点可参考：[从零开始快速接入 EdgeOne](#)。
2. 如何添加加速域名可参考：[添加加速域名](#)。

## 操作步骤

### 步骤1：创建并部署函数

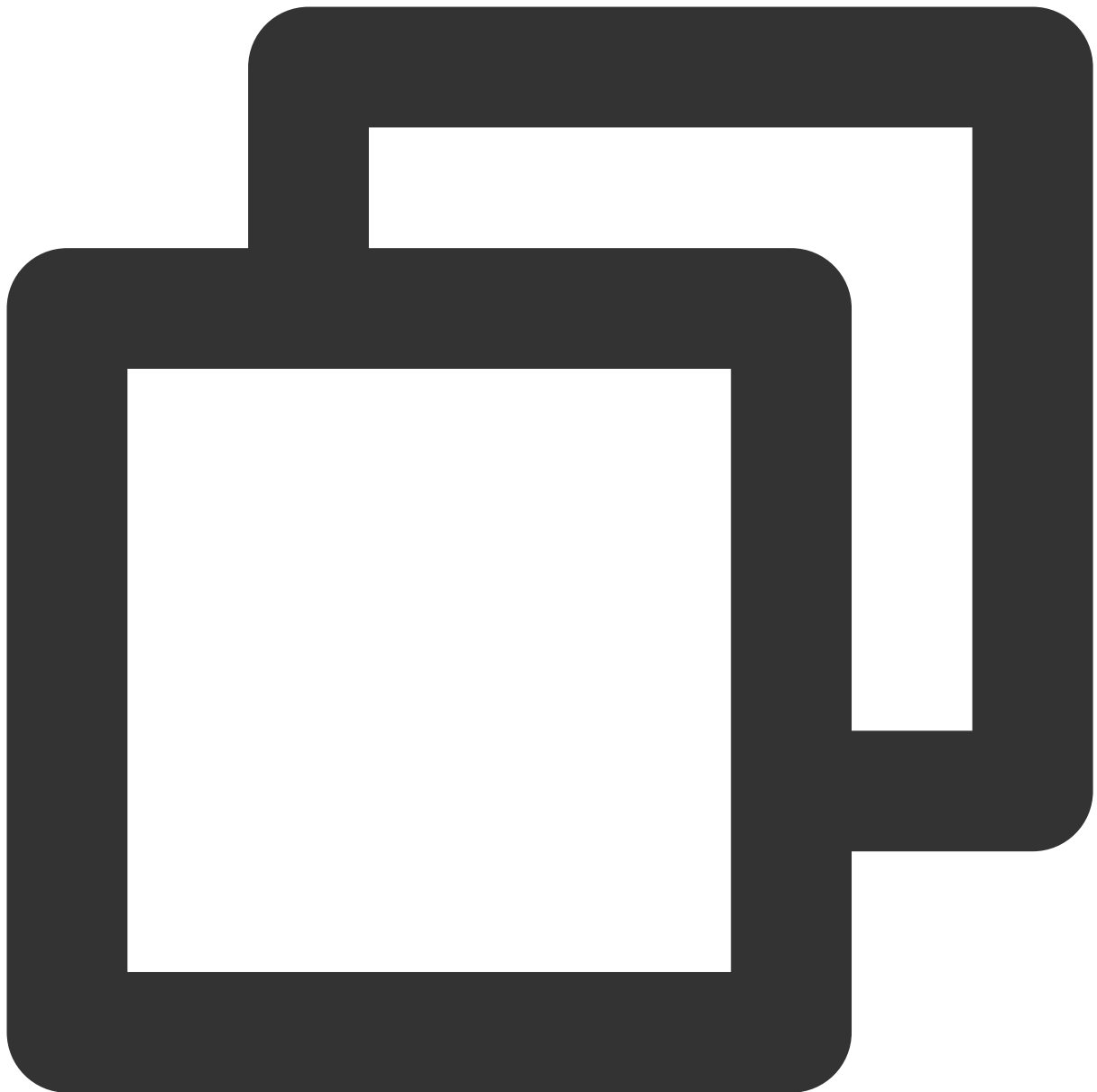
1. 登录 [边缘安全加速平台 EO 控制台](#)，在左侧菜单栏中，单击**站点列表**，在站点列表内单击需配置的**站点**。
2. 在站点详情页面，单击**边缘函数 > 函数管理**。
3. 在边缘函数管理页面，单击**新建函数**，选择使用模板创建函数，在此步骤您可以根据实际业务需求来使用模板创建一个函数。以当前场景为例，可选择使用“创建 Hello World”模板新建。选择模板后，单击**下一步**。
4. 在新建边缘函数页面，配置相关参数，参数说明如下：

函数名称：必填项，只能包含字母、数字、连字符，以字母开头，以数字或字母结尾，2~30个字符；创建后无法修改。如：`test-edgefunctions`。

描述：非必填，最多支持60个字符。如：自定义 HTML 页面和响应头。

函数代码：需响应的边缘函数代码内容。以当前场景为例，可复制如下函数代码，将其粘贴到控制台的代码编辑器中，替换编辑器中的默认代码。





```
const html = `  
  <!DOCTYPE html>  
  <body>  
    <h1>Hello World</h1>  
    <p>This markup was generated by TencentCloud Edge Functions.</p>  
    <a href="https://cloud.tencent.com/product/teo">TencentCloud EdgeOne</a>  
  </body>  
`;  
  
async function handleRequest(request) {  
  return new Response(html, {
```

```
headers: {
  'content-type': 'text/html; charset=UTF-8',
  'x-edgesfunctions-test': 'Welcome to use Edge Functions.',
},
});
}

addEventListener('fetch', event => {
  event.respondWith(handleRequest(event.request));
});
```

## 说明

上述代码的语义为自定义 HTML，并且添加自定义响应头 `x-edgesfunctions-test`：Welcome to use Edge Functions。

5. 单击**创建并部署**，当弹窗如下对话框，即表示部署成功。



### Deployed successfully

Functions are triggered when triggering rules are matched

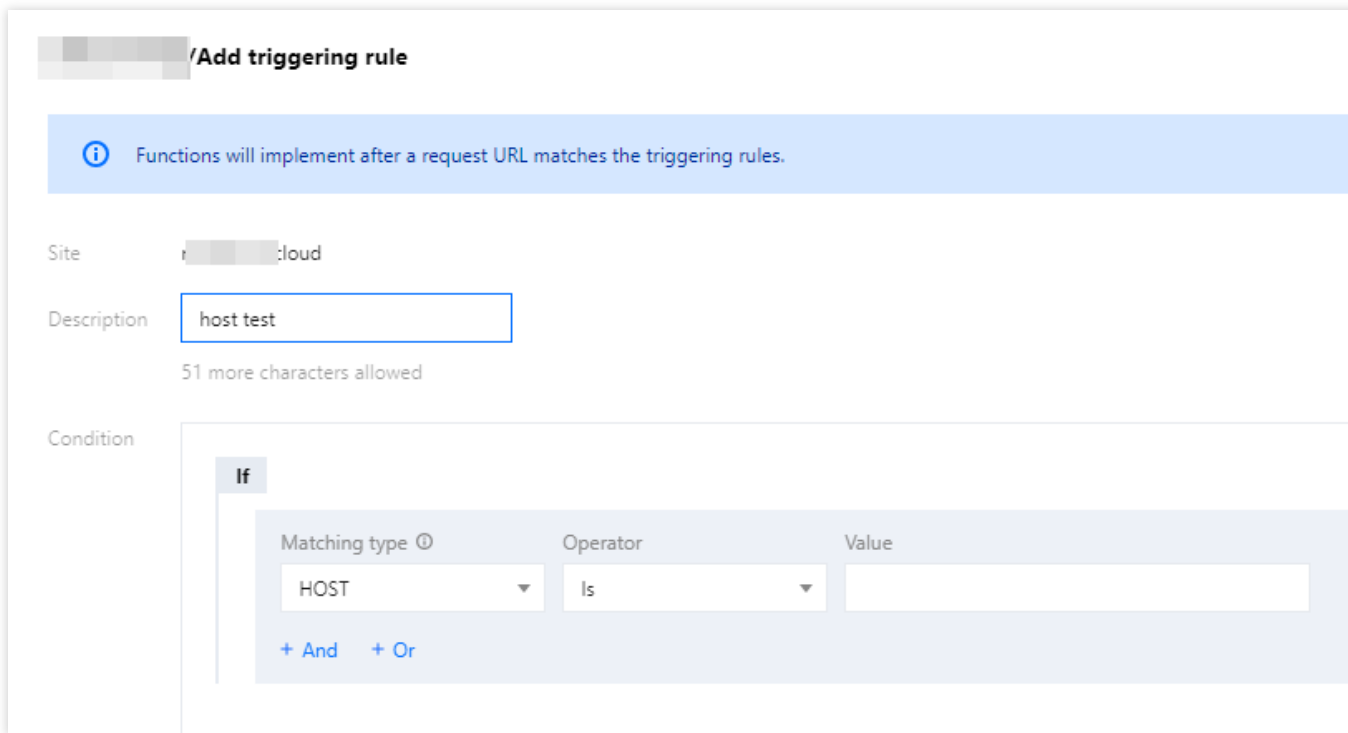
Not now

Add triggering rule

## 步骤2：配置触发规则

如期望通过设置匹配站点的HOST、URL Path或文件后缀等方式触发函数执行，可通过如下2个步骤操作：

1. 创建并部署函数成功后，按照提示单击**新增触发规则**。
2. 在新增触发规则页面，配置匹配条件，以当前场景为例，可选择匹配类型为“HOST”、运算符为“等于”、值为已添加的子域名 `www.example.com`，单击**确定**即可创建触发规则。



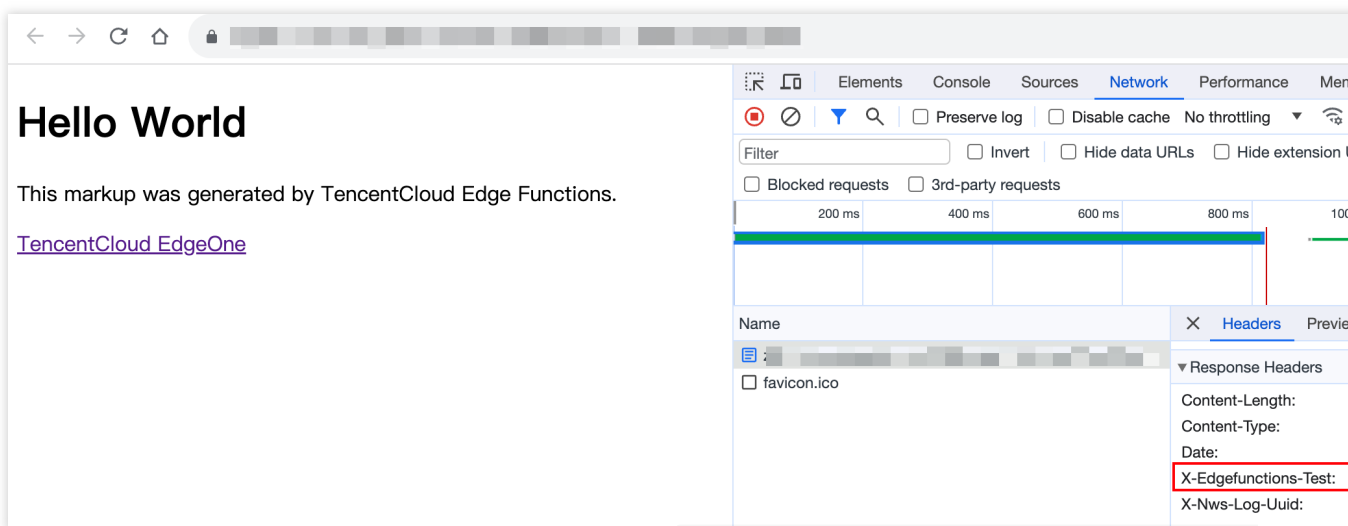
### 步骤3：验证边缘函数

验证函数是否按照预期运行，您可通过在浏览器或 Curl 发起请求测验：

浏览器验证

Curl 验证

在浏览器输入 URL，例如：`https://www.example.com/test-edgefunctions`，该 URL 可匹配已设置的触发条件触发函数的执行，查看响应页面信息：



在 MAC/Linux 终端内，运行 curl 请求命令进行验证，例如：`curl https://www.example.com/test-edgefunctions`，可查看响应如下：

```
→ ~ curl -i https://[redacted] ions
HTTP/2 200
x-edgefunctions-test: Welcome to use Edge Functions.
content-type: text/html; charset=UTF-8
content-length: 238

<!DOCTYPE html>
<body>
  <h1>Hello World</h1>
  <p>This markup was generated by a TencentCloud Edge Functions.</p>
  <a href="https://cloud.tencent.com/product/teo">边缘安全加速平台 (TencentCloud EdgeOne)</a>
</body>
→ ~
```

# 操作指引

## 函数管理


最近更新时间：2024-01-02 10:23:49

### 操作场景

本文介绍如何创建、编辑和删除边缘函数，以及如何配置函数的触发规则。

### 创建并部署函数

1. 登录 [边缘安全加速平台 EO](#) 控制台，在左侧菜单栏中，单击**站点列表**，在站点列表内单击需配置的**站点**。
2. 在站点详情页面，单击**边缘函数 > 函数管理**。
3. 在边缘函数管理页面，单击**新建函数**，选择使用模板创建函数，在此步骤您可以根据实际业务需求来使用模板创建一个函数。
4. 在新建边缘函数页面，配置相关参数，参数说明如下：  
函数名称：必填项，只能包含字母、数字、连字符，以字母开头，以数字或字母结尾，2~30个字符；创建后无法修改。如：`test-edgefunctions`。  
描述：非必填，最多支持60个字符。如：自定义 HTML 页面和响应头。  
函数代码：需响应的边缘函数代码内容。

Function  

2-30 characters ([a-z], [0-9] and [-]). It must start with a letter and end with a digit or letter. Consecutive hyphens (.) are not allowed; cannot be modified after creation.


Description

60 more character allowed

Code [View sample](#)

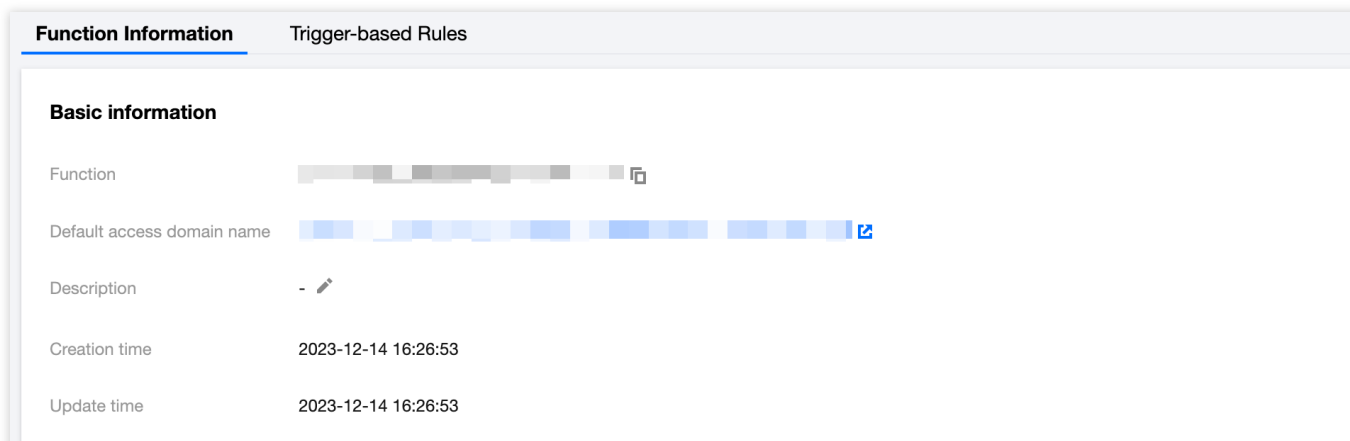
```
1  addEventListener('fetch', e => {
2    const response = new Response('Hello World!');
3    e.respondWith(response);
4  });
```

5. 单击**创建并部署**，当弹窗如下对话框，即表示部署成功。

 **Deployed successfully**

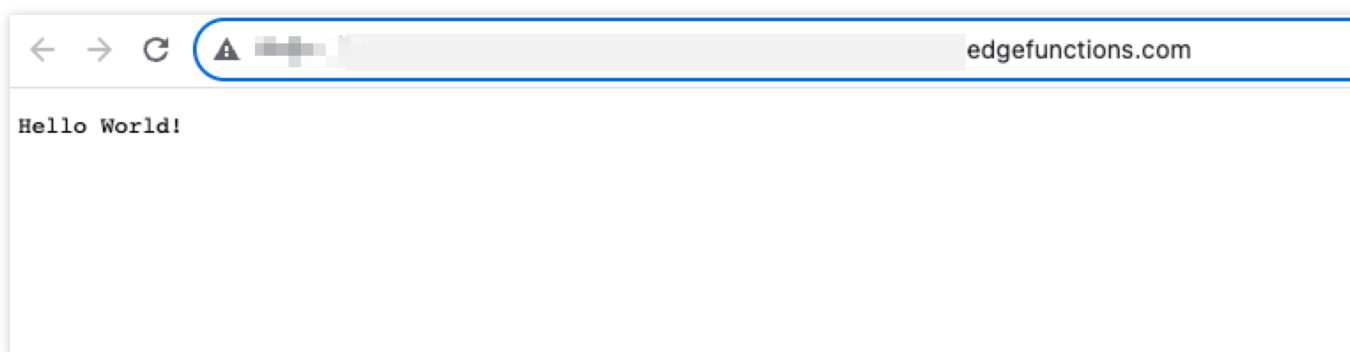
Functions are triggered when triggering rules are matched

部署成功后，可通过单击平台分配的**默认访问域名**触发函数执行可验证是否生效。



Function Information		Trigger-based Rules
<b>Basic information</b>		
Function	[Redacted]	[Copy icon]
Default access domain name	[Redacted]	[Copy icon]
Description	-	[Edit icon]
Creation time	2023-12-14 16:26:53	
Update time	2023-12-14 16:26:53	

如部署默认函数代码后预期生效则如下所示：



← → ↻ [Redacted] edgefunctions.com

Hello World!

## 配置触发规则

如期望通过设置匹配站点的 HOST、URL Path 或文件后缀等方式触发函数执行，可通过如下2个步骤操作：

1. 创建并部署函数成功后，按照提示单击**新增触发规则**。
2. 在新增触发规则页面，按需选择匹配类型、运算符和值。

3/Add triggering rule

Functions will implement after a request URL matches the triggering rules.

Site: [redacted].ss

Description:  (51 more characters allowed)

Condition:

- If
  - Matching type: HOST
  - Operator: Equal to
  - Value: [redacted]

+ And + Or

3. 单击**确定**，即可创建触发规则。

Function information **Triggering rule**

[Add triggering rule](#)

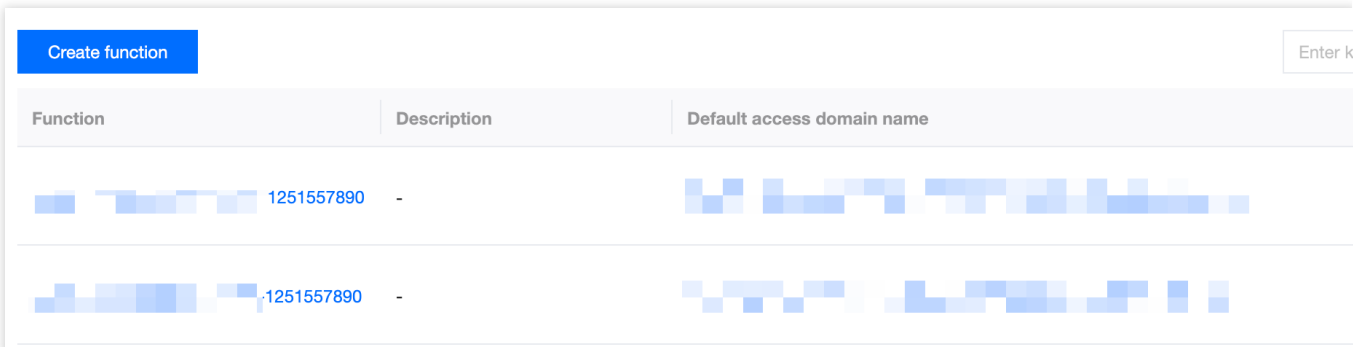
Rule ID	Description	Condition	Creation time	Update time
rule-[redacted]	host test	if [redacted]	2023-01-11 15:05:12	2023-01-11 15:05:12

Total items: 1 10 / page

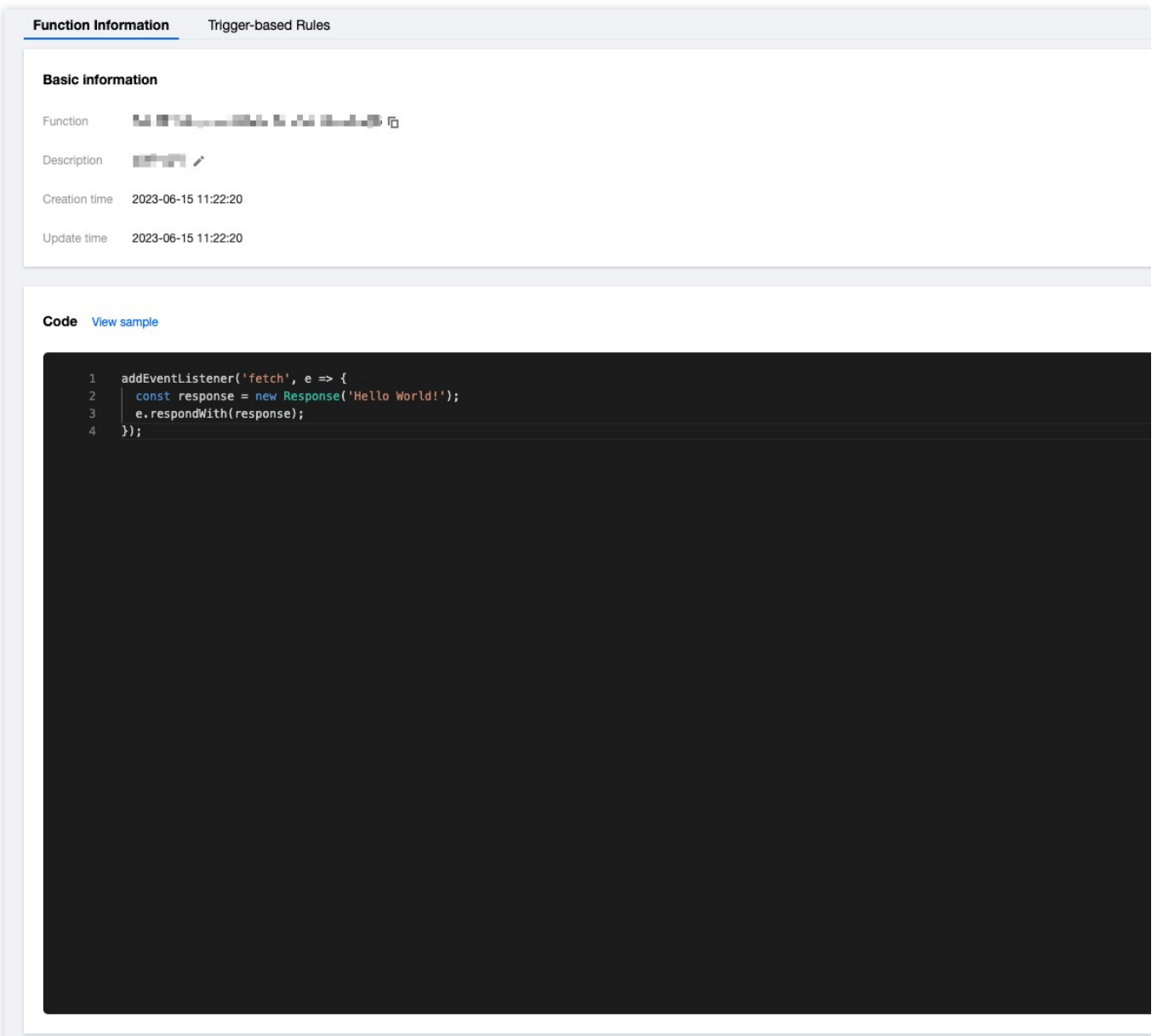
## 编辑边缘函数

1. 在函数管理页面，选择需要修改的函数，单击该**函数名称**。








2. 在函数信息页面，修改函数代码后单击**保存并部署**或**Ctrl + S**。



3. 修改代码后单击**保存并部署**，如此函数已存在触发规则则会提示如下：

 **The following triggering rules will take effect once they're deployed. Check the rules before you deploy.**




Rule ID	Condition	Update time
rule- 	if (  )	2023-01-11 15:20:05

Cancel

Deploy

## 删除边缘函数

1. 如需要删除已新建的函数，您可在函数管理页面，选择需要删除的函数，单击操作列的**删除**。

Function information		Triggering rule			
Rule ID	Description	Condition	Creation time	Update time	
rule- 	host test	if (  )	2023-01-11 15:05:12	2023-01-11 15:20:05	

Total items: 1 10 / page

2. 在确认删除对话框中，单击**确定**，即可完成删除操作。

### 注意：

此函数一旦删除不可恢复，已添加的触发规则会一并删除。

# 触发配置

最近更新时间：2023-12-14 17:47:26

## 操作场景

本功能适用于站点下函数触发规则的如下操作：

支持站点下函数触发规则的增删改查。

支持快速调整触发规则的优先级，适用于请求 URL 匹配到多个触发规则的情况下快速调整执行位置的顺序，位置在前的触发规则将会执行，位置在后的触发规则将不会执行。

## 操作介绍

### 新建触发规则

1. 登录 [边缘安全加速平台](#) 控制台，在左侧菜单栏中，单击**站点列表**，在站点列表内单击需配置的**站点**。
2. 在站点详情页面，单击**边缘函数 > 触发配置**。
3. 在触发配置页面，单击规则列表右侧的



，配置相关参数。

### Add triggering rule

Functions will implement after a request URL matches the triggering rules.

Site: [Redacted]

Description: Optional  
60 more characters allowed

Execute function: Please select [Create now](#)

Condition

**If**

Matching type ⓘ  
Please select

+ And + Or

[OK](#) [Cancel](#)

#### 参数说明：

站点：默认显示当前站点名称。

描述：非必填项，最多可支持60个字符。

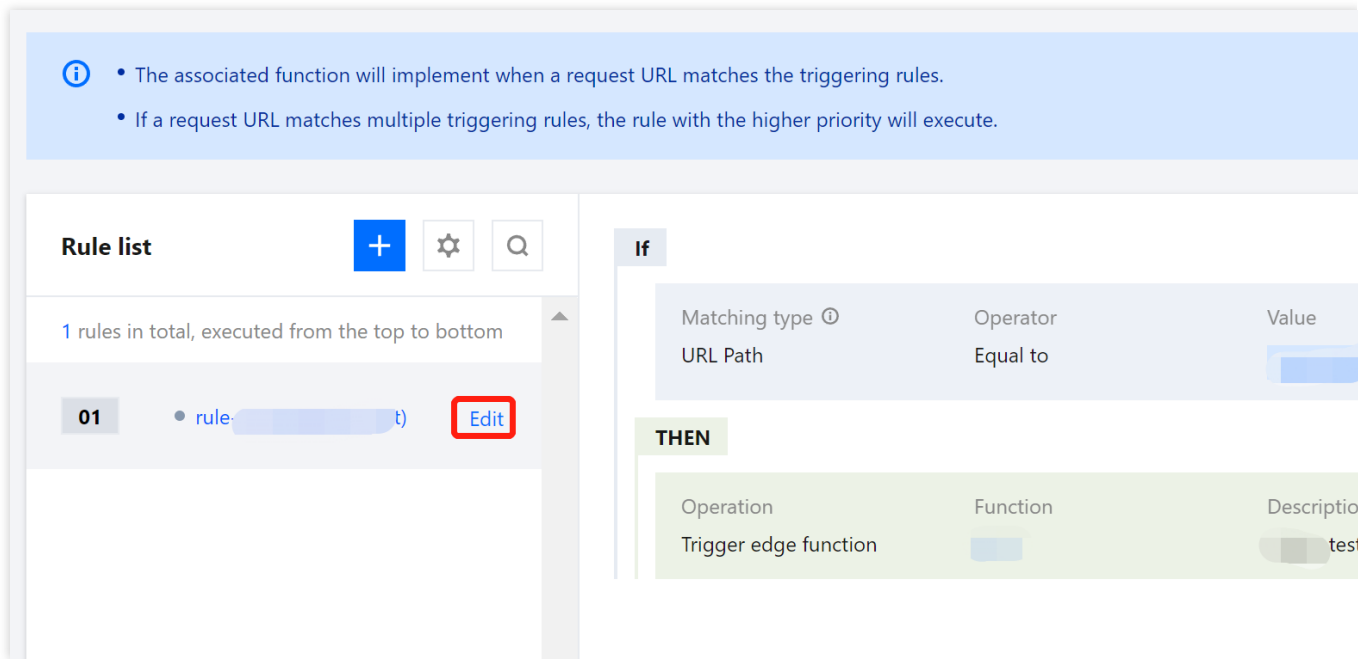
触发条件：按需选择匹配类型、运算符和值，更多参数详情请参见 [规则引擎](#)。

执行函数：下拉选择已创建的函数。

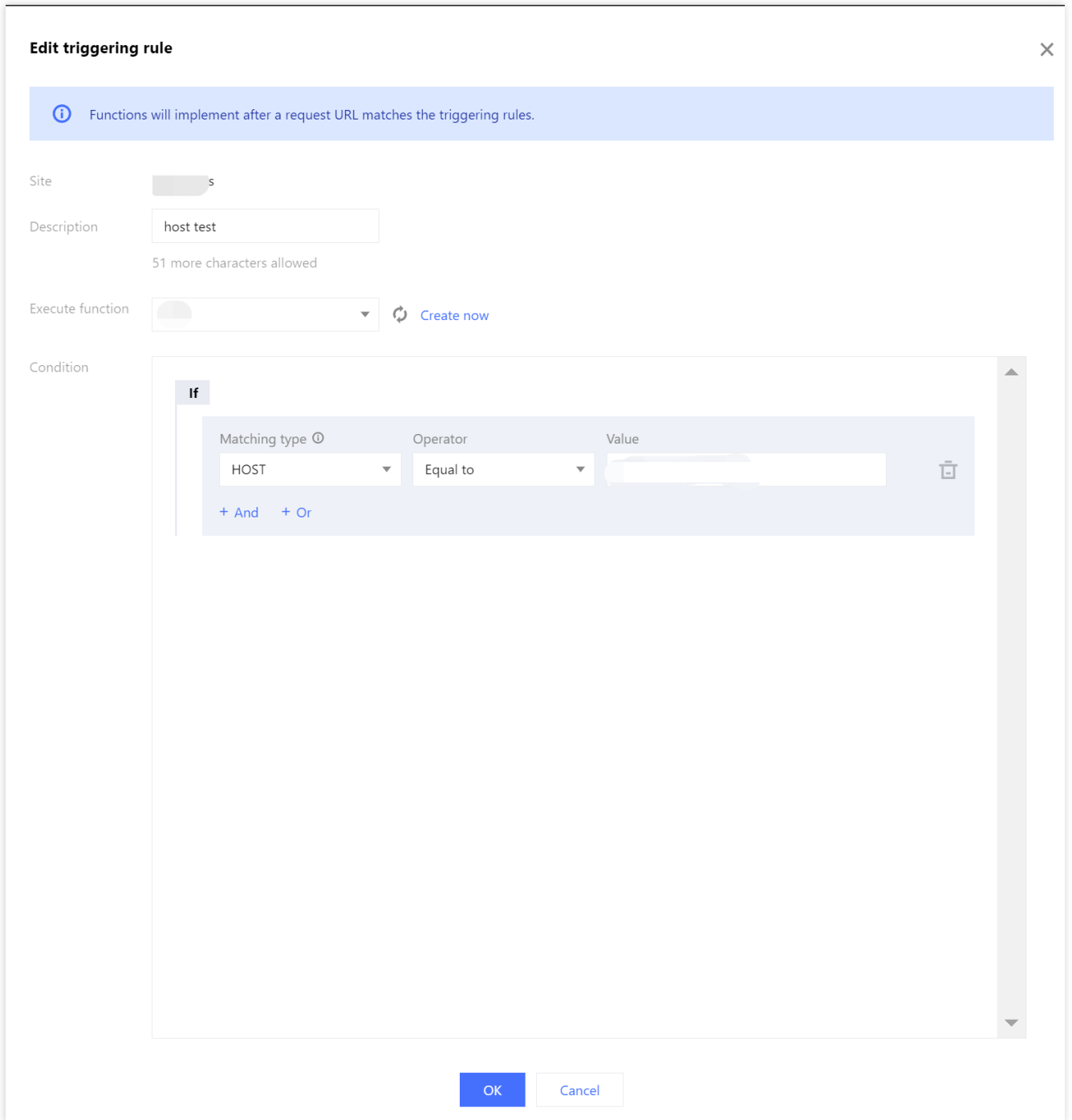
4. 单击**确定**，即可完成触发规则的新建。

#### 编辑触发规则

1. 在触发配置页面，选择需要修改的规则，单击**编辑**。



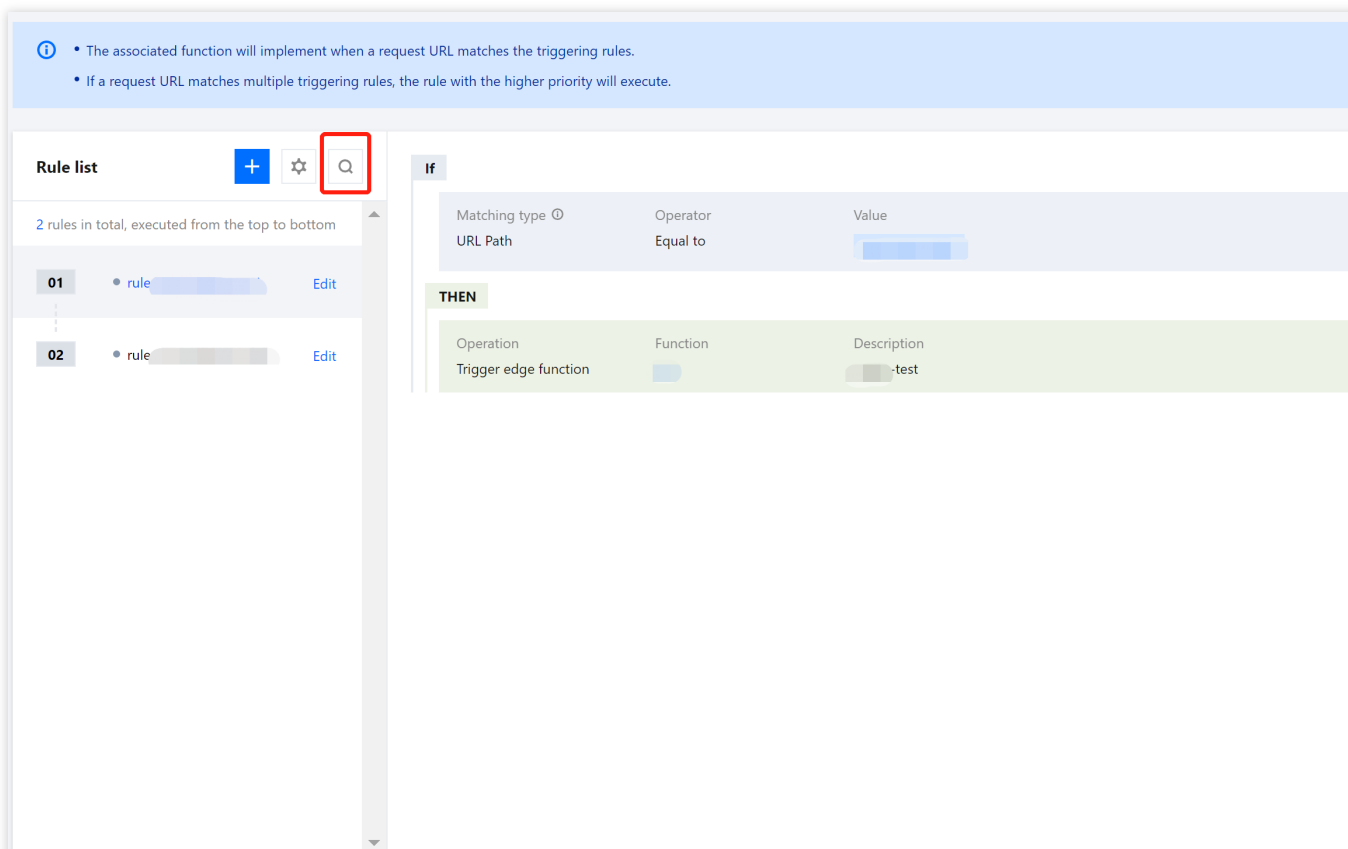
2. 在编辑触发规则对话框中，修改相关参数，单击**确定**即可完成触发规则的编辑。



### 查询触发规则

在触发配置页面，单击规则列表右侧的

，在搜索的输入框中填写规则 ID 的关键词即可完成查询。



### 删除触发规则

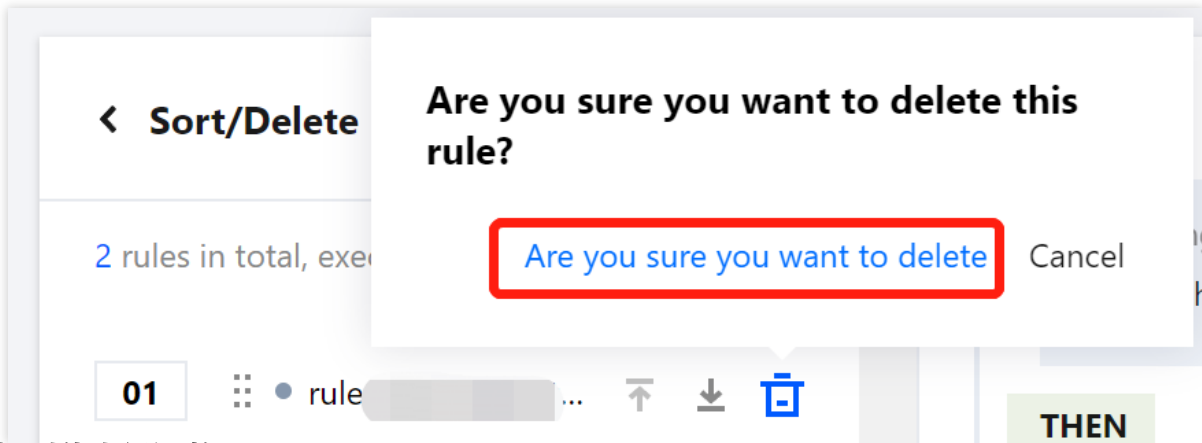
1. 在触发配置页面，单击规则列表右侧的



2. 选择需要删除的规则，单击



3. 在确认删除对话框中，单击**确认删除**，即可完成触发规则的删除。



### 触发规则优先级调整

1. 在触发配置页面，单击规则列表右侧的



图标。

2. 选择需要调整的规则，单击

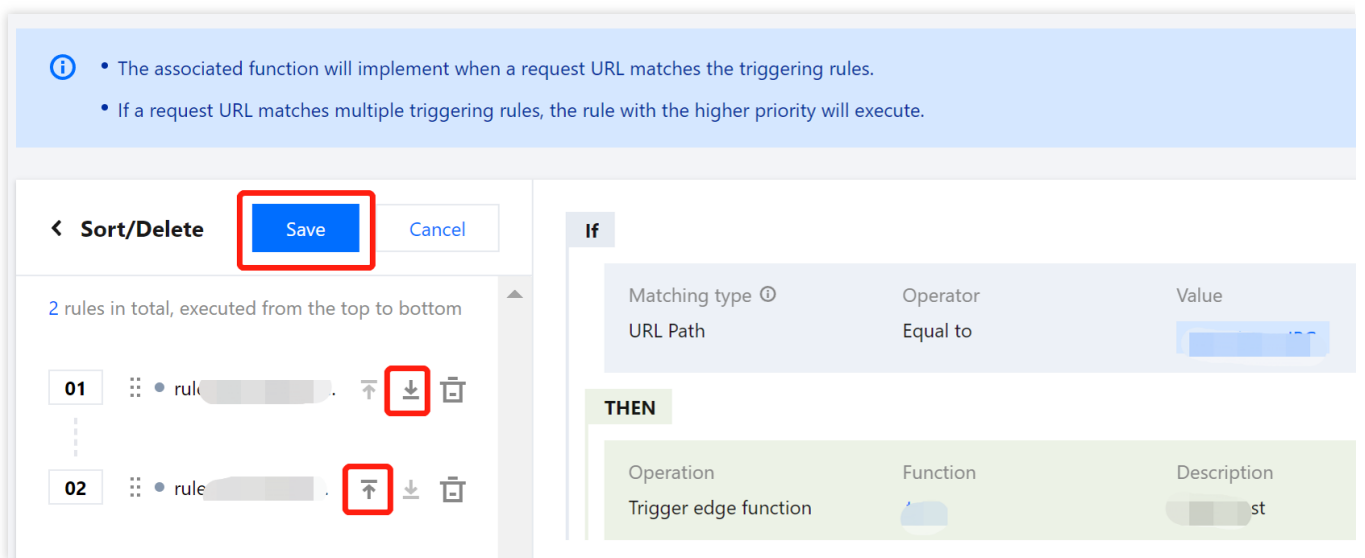


上移规则或

下移规则，单击**保存**即可完成优先级调整。

### 说明

若请求 URL 匹配到多个触发规则的情况下（如下图序号为01和02的触发规则）位置在前的触发规则将会执行（如下图序号01规则），位置在后的触发规则将不会执行（如下图序号02规则）。

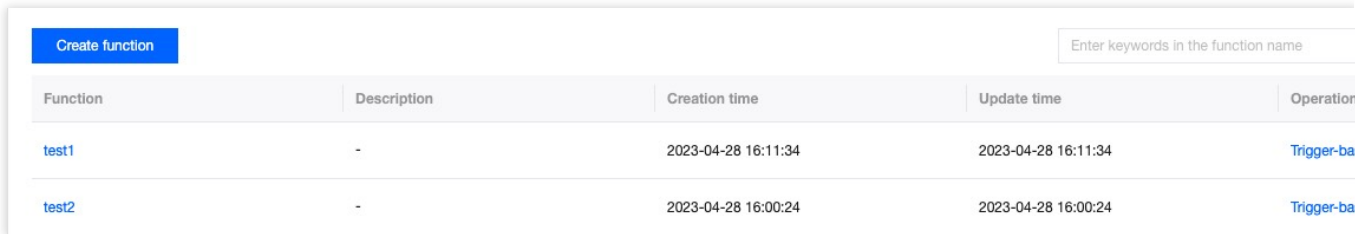


## 案例介绍



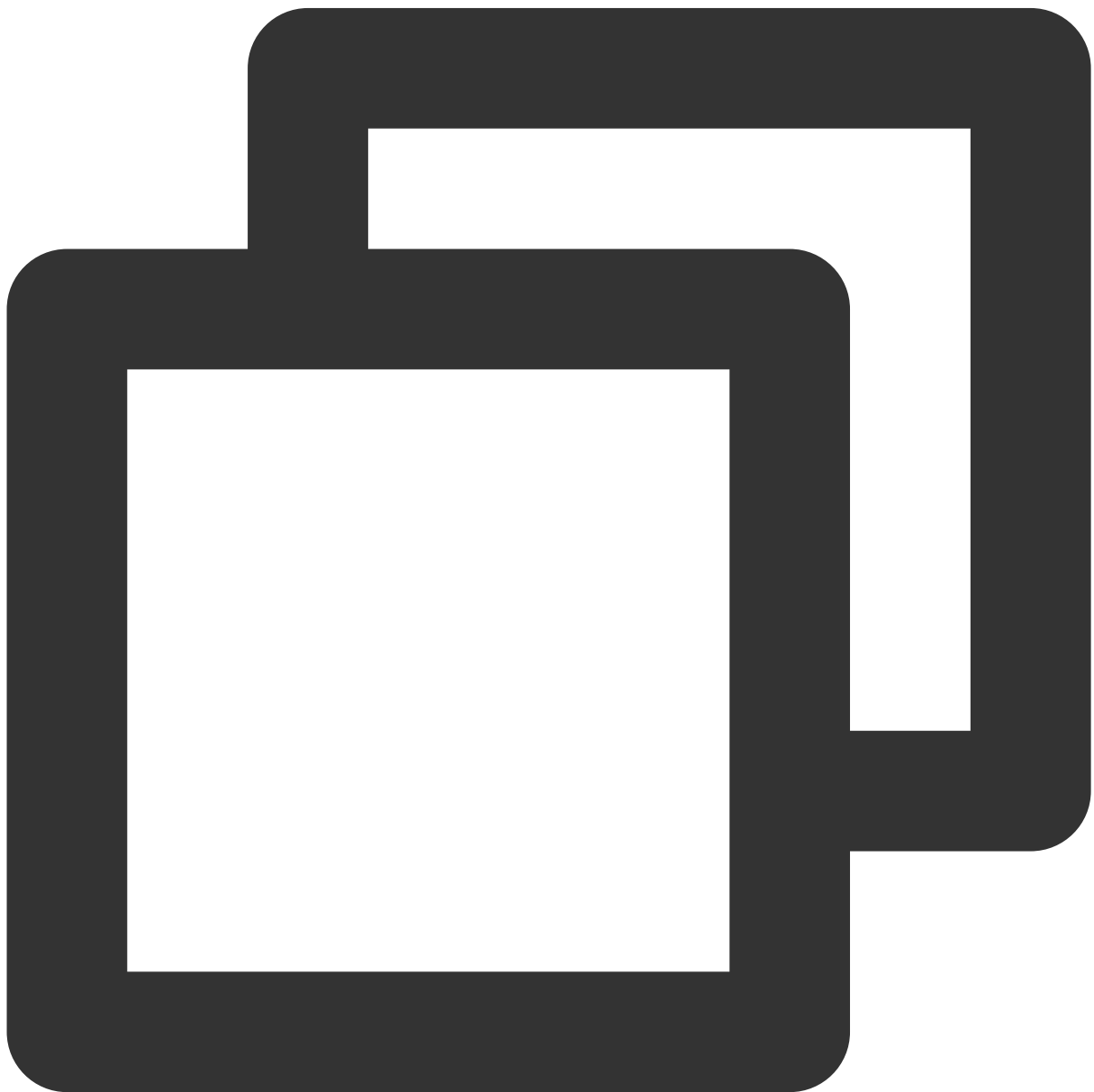
将为您介绍若请求 URL 匹配到多个触发规则的情况下，如何调整触发规则的执行顺序。

1. 在函数管理页面，已新建相同的触发条件的两个不同函数，如下图所示：



Function	Description	Creation time	Update time	Operation
test1	-	2023-04-28 16:11:34	2023-04-28 16:11:34	Trigger-ba
test2	-	2023-04-28 16:00:24	2023-04-28 16:00:24	Trigger-ba

函数 test1 的代码如下：

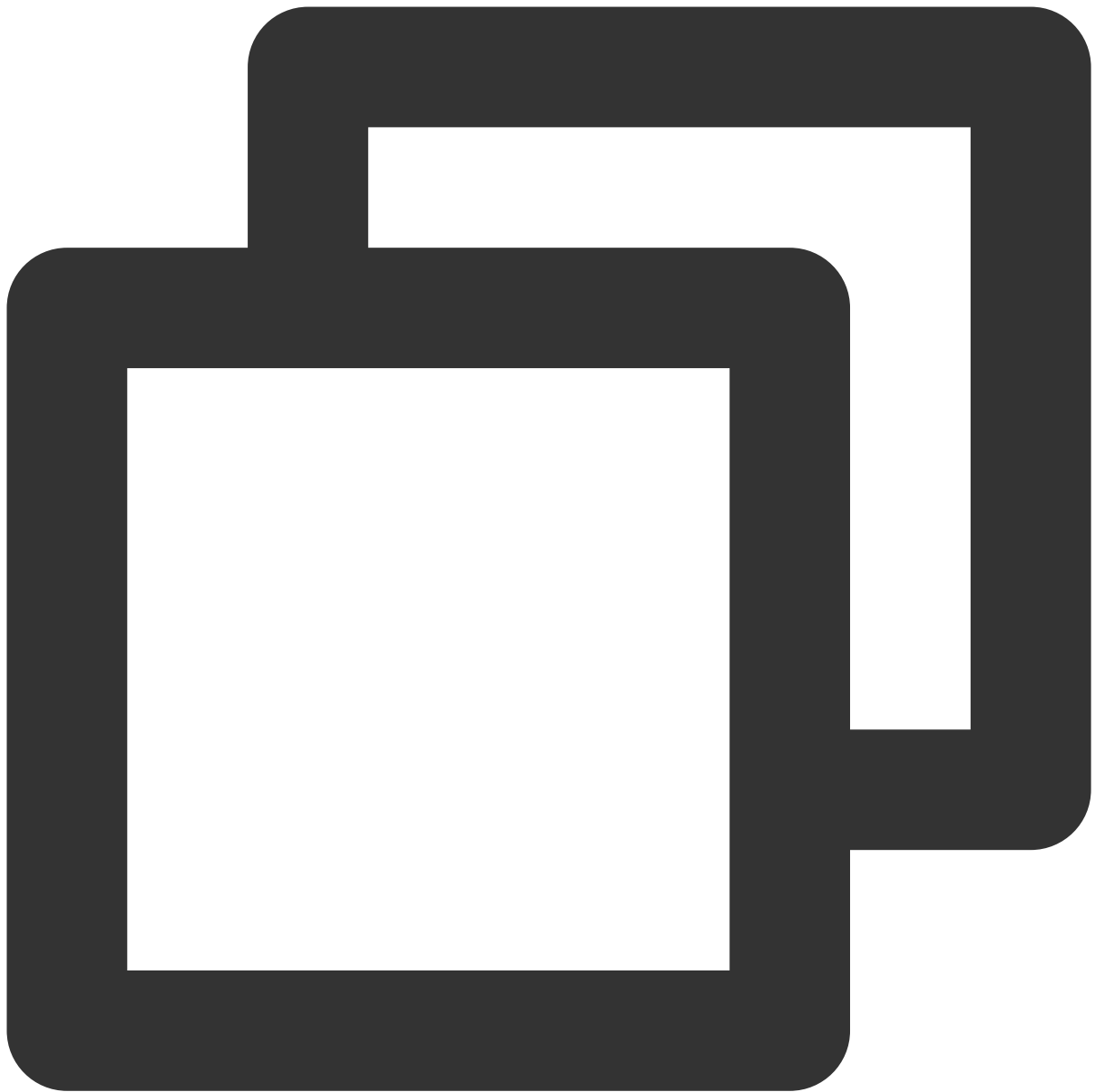


```
const html = `
  <!DOCTYPE html>
  <body>
  <h1>The test 1, Hello World</h1>
  <p>This markup was generated by a TencentCloud Edge Functions.</p>
  <a href="https://cloud.tencent.com/product/teo"> TencentCloud EdgeOne </a>
  </body>
`;

async function handleRequest(request) {
  return new Response(html, {
    headers: {
      'content-type': 'text/html; charset=UTF-8',
      'x-edgefunctions-test': 'Welcome to use Edge Functions.',
    },
  });
}

addEventListener('fetch', event => {
  event.respondWith(handleRequest(event.request));
});
```

函数 `test2` 的代码如下：



```
const html = `  
  <!DOCTYPE html>  
  <body>  
    <h1>The test 2, Hello World</h1>  
    <p>This markup was generated by a TencentCloud Edge Functions.</p>  
    <a href="https://cloud.tencent.com/product/teo"> TencentCloud EdgeOne </a>  
  </body>  
`;  
  
async function handleRequest(request) {  
  return new Response(html, {
```

```
headers: {
  'content-type': 'text/html; charset=UTF-8',
  'x-edgefunctions-test': 'Welcome to use Edge Functions.',
},
});
}

addEventListener('fetch', event => {
  event.respondWith(handleRequest(event.request));
});
```

2. 在触发配置页面，可查看触发规则，如下图所示：



3. 函数 test1 的触发规则的顺序为01，函数 test2的触发规则的顺序为02，如下图所示：



4. 在浏览器中输入：触发规则 url 并按 Enter 键，响应内容如下：

# The test 1, Hello World

This markup was generated by a TencentCloud Edge Functions.

[TencentCloud EdgeOne](#)

5. 函数 test2 单击

后，单击保存。



6. 函数 test2 的触发规则的顺序为 01，函数 test1 的触发规则的顺序为 02；

7. 再次在浏览器中输入：触发规则 url 并按 Enter 键，响应内容如下：

# The test 2, Hello World

This markup was generated by a TencentCloud Edge Functions.

[TencentCloud EdgeOne](#)

以上为若请求 URL 匹配到多个触发规则的情况下，如何调整触发规则的执行顺序的操作过程。

# Runtime APIs

## addEventListener

最近更新时间：2023-03-08 11:26:27

注册事件监听器，是边缘函数的运行入口。`addEventListener` 仅支持注册一个事件监听器。当前仅支持 `fetch` 请求事件，通过注册 `fetch` 事件监听器，生成 HTTP 请求事件 `FetchEvent`，进而实现对 HTTP 请求的处理。

### 描述



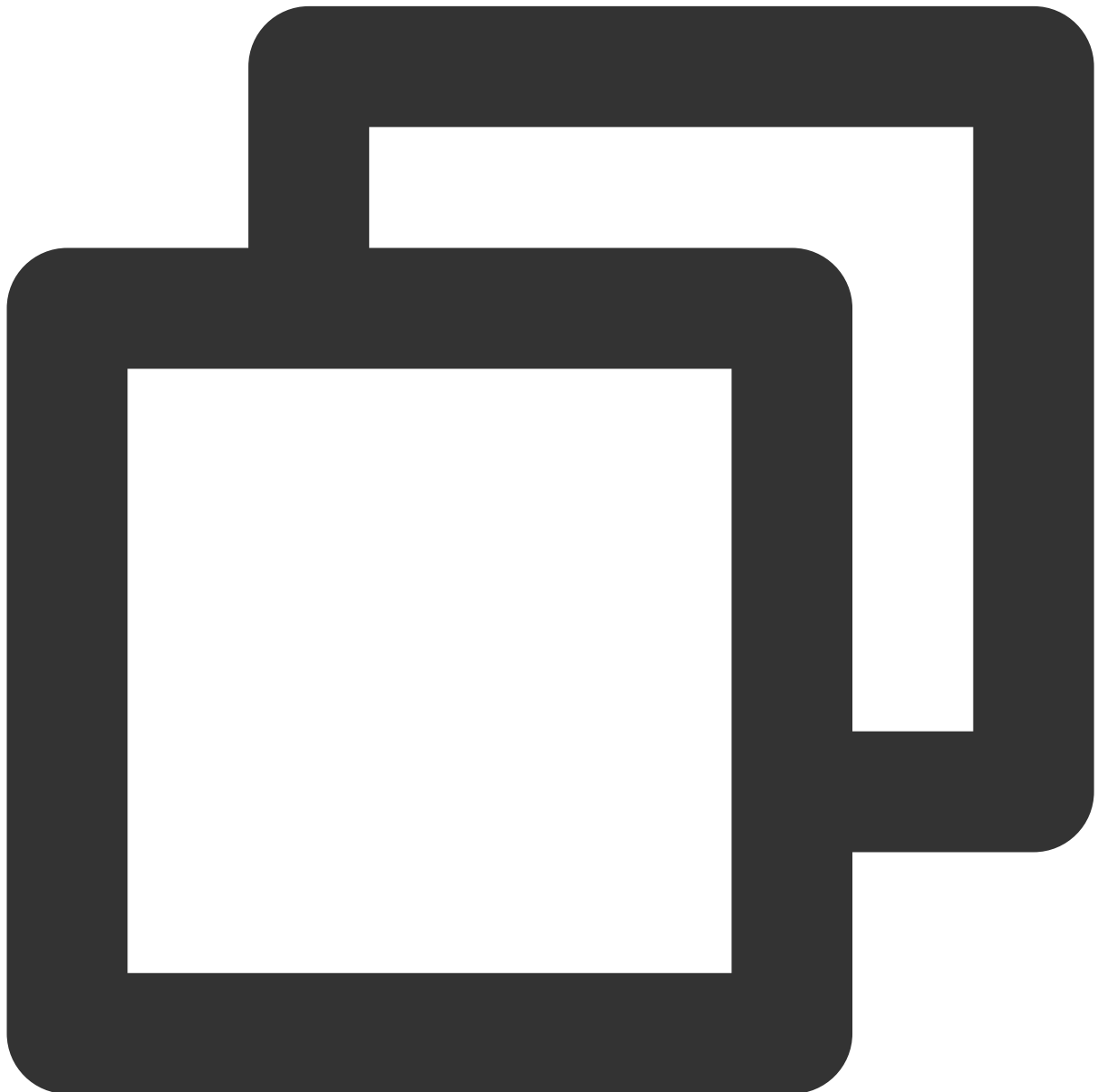
```
function addEventListener(type: string, listener: (event: FetchEvent) => void): voi
```

### 参数

参数名称	类型	必填	说明
type	string	是	事件类型。 当前仅支持 <code>fetch</code> 请求事件。 非 <code>fetch</code> 请求事件，边缘函数引擎会主动抛出 <code>Error</code> 类型的异常。

listener	(event: <a href="#">FetchEvent</a> ) => void	是	事件监听器。用于处理事件回调。 注册 <code>fetch</code> 请求事件生成 <a href="#">FetchEvent</a> 类的事件监听器。
----------	--	---	---

## 示例代码



```
// 注册 fetch 请求事件监听器  
addEventListener('fetch', (event) => {
```



```
// 响应客户端
event.respondWith(new Response('Hello World!'));
});
```

## 相关参考

[MDN 官方文档：addEventListener](#)

示例函数：返回 HTML 页面

示例函数：返回 JSON

# Cache

最近更新时间：2024-01-30 17:39:35

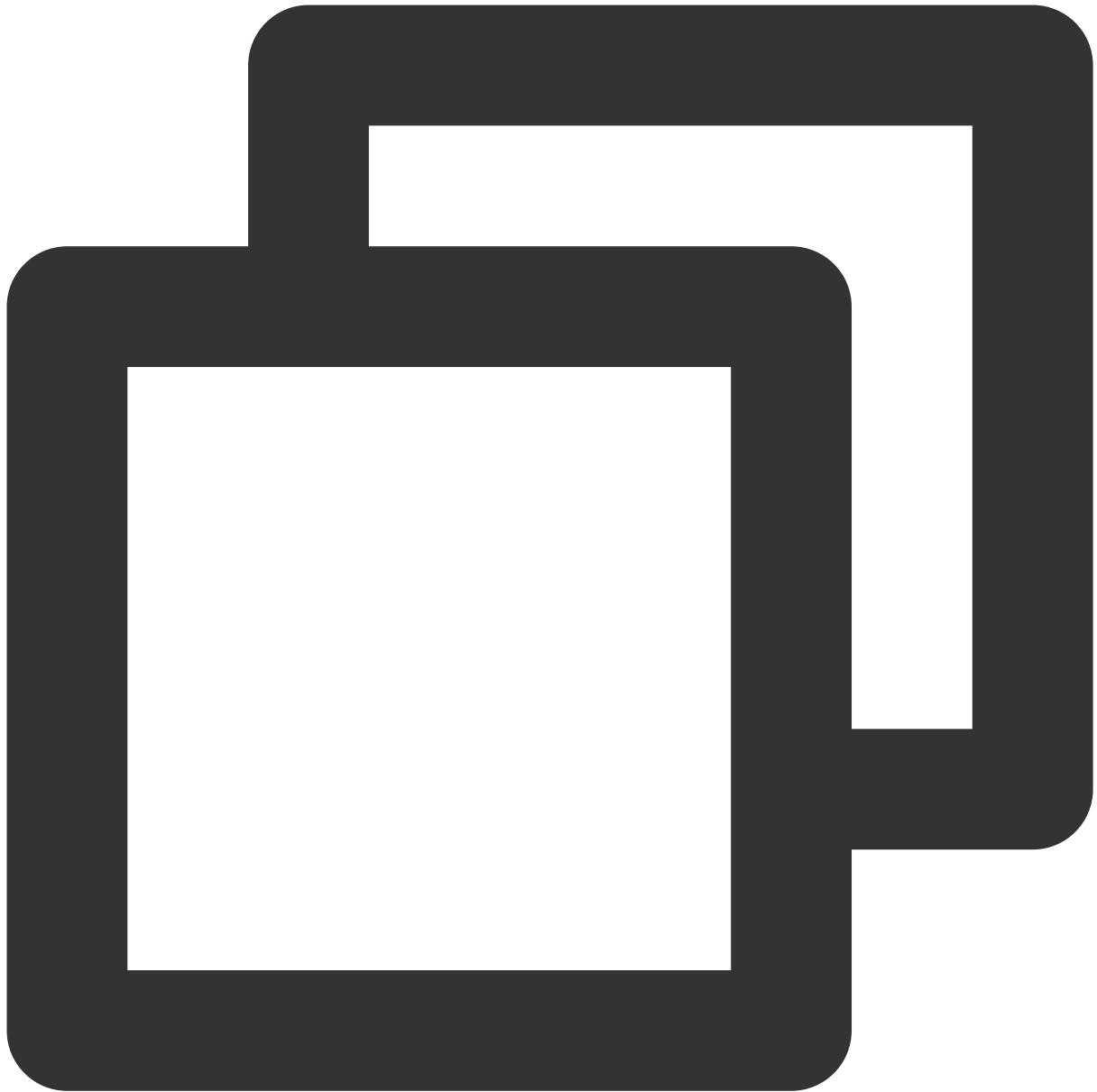
**Cache** 基于 Web APIs 标准 [Cache API](#) 进行设计。边缘函数运行时会在全局注入 `caches` 对象，该对象提供了一组缓存操作接口。

## 说明：

缓存的内容仅在当前数据节点有效，不会自动复制到其他数据节点。

## 构造函数

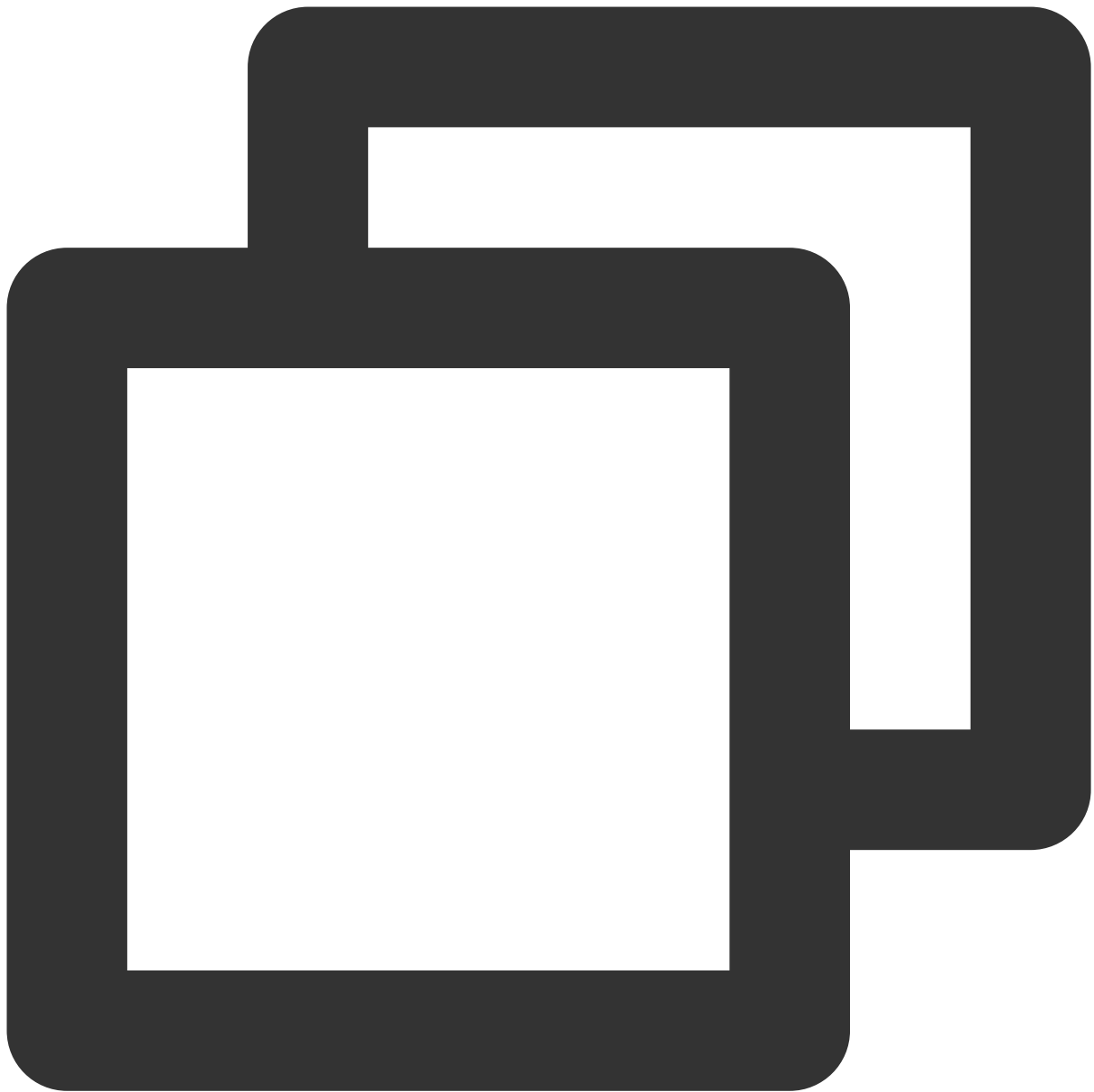
使用 `caches.default` 可以获取默认的 `cache` 实例。



```
// 获取默认 cache 实例
const cache = caches.default;

// 效果等同于 caches.default
await caches.open('default');
```

使用 `caches.open` 创建指定命名空间的 `cache` 实例。



```
// 创建指定命名空间的 cache 实例  
const cache = await caches.open(namespace);
```

## 参数

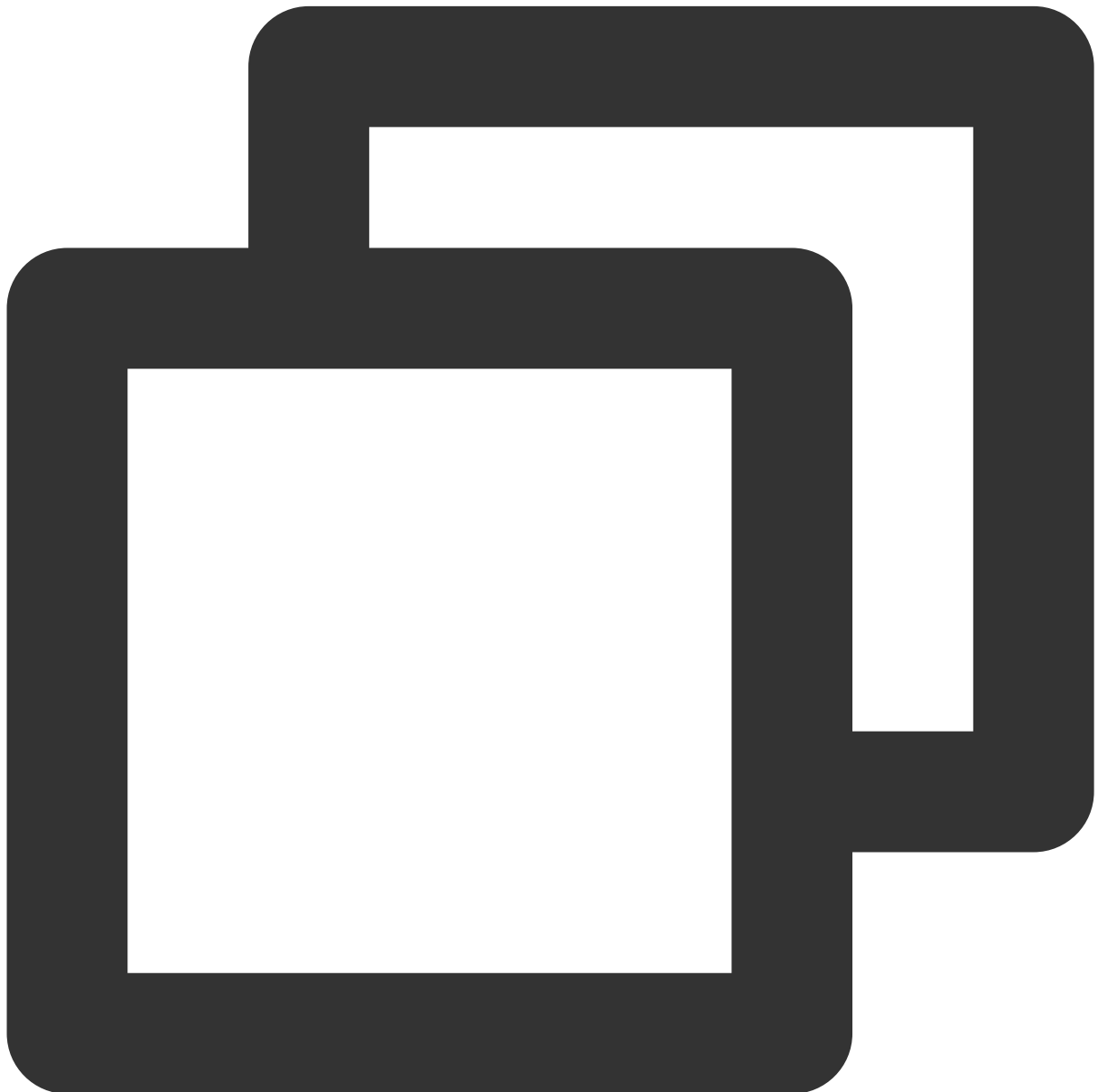
`caches.open(namespace)` 方法参数说明如下。

参数名称	类型	必填	说明
namespace	string	是	缓存命名空间。

		如果该值为 "default" 则表示默认实例，也可直接使用 <code>cache.default</code> 获取默认实例。
--	--	---

## 实例方法

### match



```
cache.match(request: string | Request, options?: MatchOptions): Promise<Response |
```

获取 request 关联的缓存 [Response](#)。返回一个 Promise 对象。如果缓存存在，则包含 Response 对象，反之包含 undefined。

**注意：**

`cache.match` 内部不会主动回源，缓存过期则会抛出 504 错误。

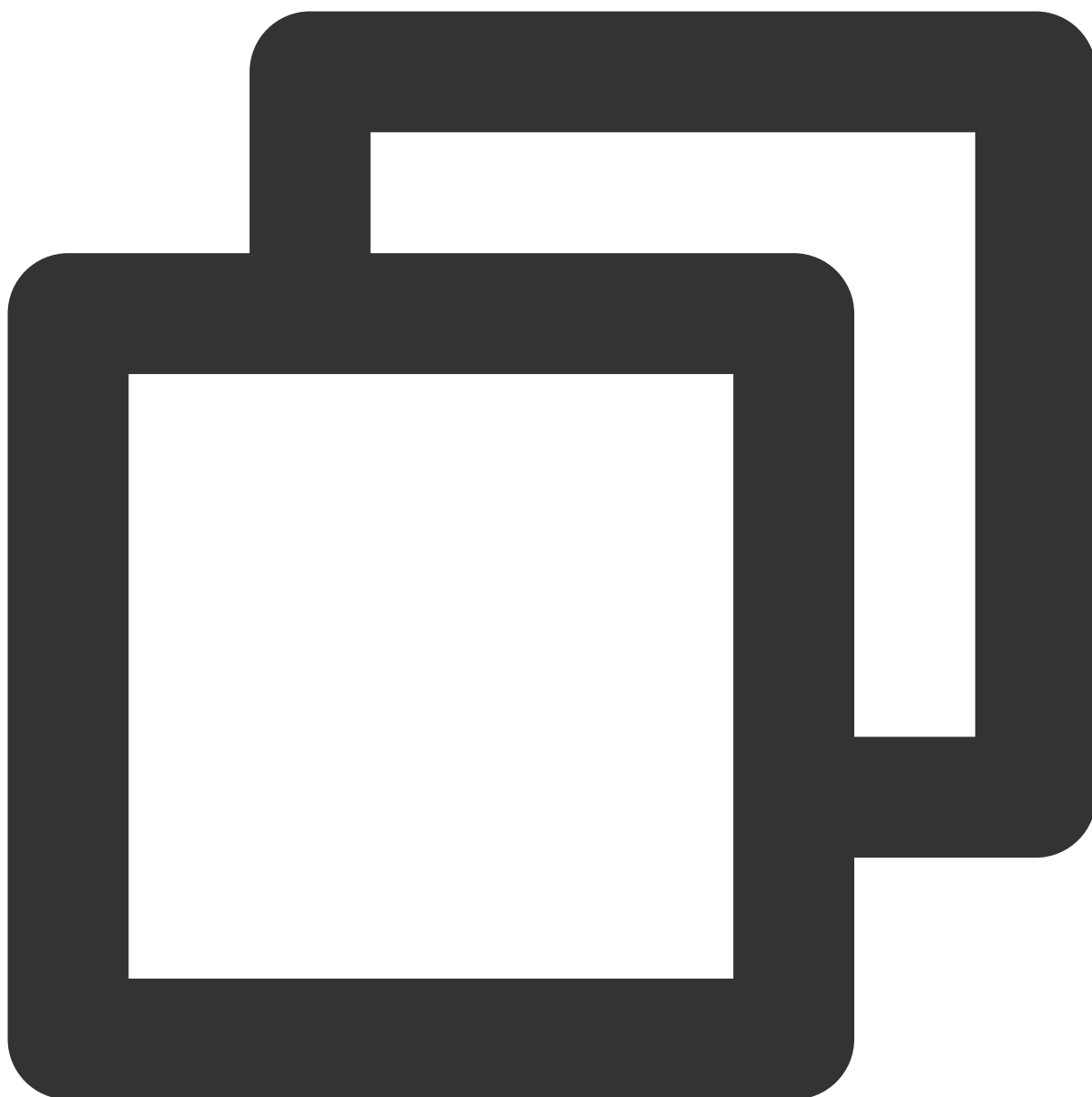
**参数**

参数名称	类型	必填	说明
request	string   <a href="#">Request</a>	是	请求对象，headers 说明如下。 <b>GET</b> request 只支持 GET 方法，当类型为 string 时，将被作为 URL 构造 Request 对象。 <b>Range</b> request 包含 Range 头部时，如果缓存的 Response 能够支持 Range 范围处理，返回 206 响应。 <b>If-Modified-Since</b> request 包含 <a href="#">If-Modified-Since</a> 头部时，如果缓存的 Response 存在 Last-Modified 头部，且 <a href="#">Last-Modified</a> 与 If-Modified-Since 相等，返回 304 响应。 <b>If-None-Match</b> request 包含 <a href="#">If-None-Match</a> 头部时，如果缓存的 Response 存在 ETag 头部，且 <a href="#">ETag</a> 与 If-None-Match 相等，返回 304 响应。
options	<a href="#">MatchOptions</a>	否	选项。

**MatchOptions**

属性名	类型	示例值	说明
ignoreMethod	boolean	true	是否忽略 Request 的 method。为 true 时，会忽略 Request 原来的 method，作为 GET 处理。

**put**



```
cache.put(request: string | Request, response: Response): Promise<undefined>
```

尝试使用给定的 `request` 作为缓存 key，将 `response` 添加到缓存。无论缓存是否成功，均返回

`Promise<undefined>` 对象。

#### 注意：

当参数 `response` 对象的 `Cache-Control` 头部表示不缓存时，抛出 413 错误。

#### 参数

参数名称	类型	必填	说明
------	----	----	----

request	string   <a href="#">Request</a>	是	缓存 key，说明如下。 <b>GET</b> 参数 request 仅支持 GET 方法，其他方法，将抛出参数错误。 <b>string</b> 当参数 request 类型为 string 时，将被作为 URL 构造 <a href="#">Request</a> 对象。
response	<a href="#">Response</a>	是	缓存内容，说明如下。 <b>Cache-Control</b> 支持 s-maxage、max-age、no-store、no-cache、private；其中 no-store、no-cache、private 均表示不缓存，cache.put 将返回 413 错误。 <b>Pragma</b> 当 <a href="#">Cache-Control</a> 未设置，并且 <a href="#">Pragma</a> 为 no-cache。此时表示不缓存。 <b>ETag</b> 当 cache.match 参数 request 包含 <a href="#">If-None-Match</a> 头部时，可关联 <a href="#">ETag</a> 使用。 <b>Last-Modified</b> 当 cache.match 参数 request 包含 <a href="#">If-Modified-Since</a> 头部时，可关联 <a href="#">Last-Modified</a> 使用。 <b>416 Range Not Satisfiable</b> 当参数 response 对象为 <a href="#">416 Range Not Satisfiable</a> 时，暂不缓存。

### 参数限制

`cache.put` 使用以下的参数值，将抛出参数错误：

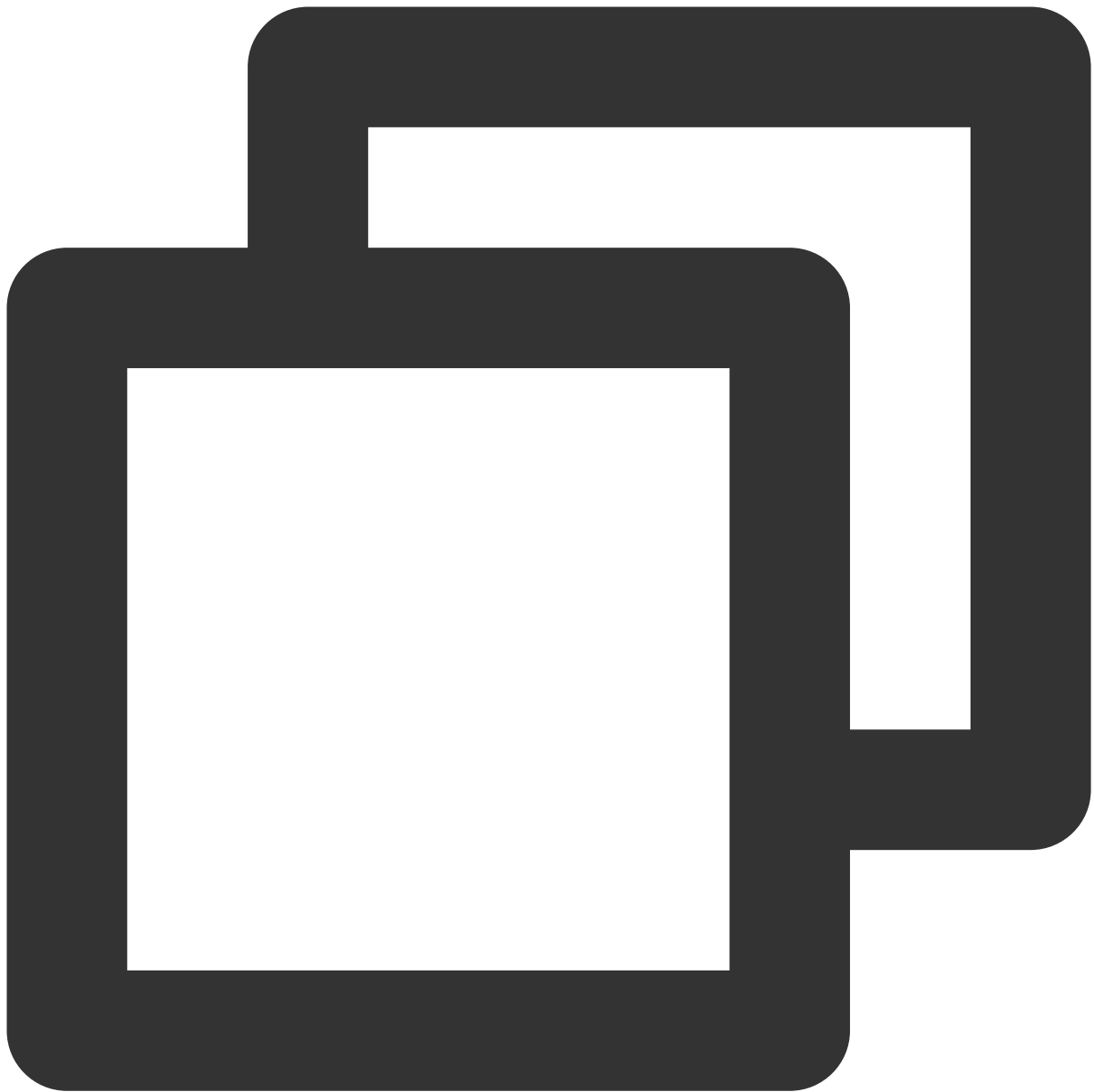
参数 `request` 为 GET 方法之外的其他方法。

参数 `response` 状态码为 [206 Partial Content](#)。

参数 `response` 包含 [Vary: \\*](#) 头部。

### delete





```
cache.delete(request: string | Request, options?: DeleteOptions): Promise<boolean>
```

删除 request 关联的缓存 response。未发生网络错误时,总返回 Promise, 并包含 true, 反之包含 false。

### 参数

参数名称	类型	必填	说明
request	string   <a href="#">Request</a>	是	缓存 key, 说明如下。 GET 参数 request 仅支持 GET 方法

			string 当参数 request 类型为 string 时，将被作为 URL 构造 <a href="#">Request</a> 对象。
options	<a href="#">DeleteOptions</a>	否	配置选项。

### DeleteOptions

属性名	类型	示例值	说明
ignoreMethod	boolean	true	是否忽略 request 的方法名。为 true 时，会忽略 Request 原来的方法，作为 GET 处理

## 相关参考

[MDN 官方文档：Cache](#)

[示例函数：缓存 POST 请求](#)

[示例函数：Cache API 使用](#)

# Cookies

最近更新时间：2024-01-30 17:05:56

**Cookies** 提供了一组 cookie 操作接口。

**注意：**

Cookies 对象以 `name + domain + path` 为唯一 key, 管理 Cookie 对象集。

## 构造函数



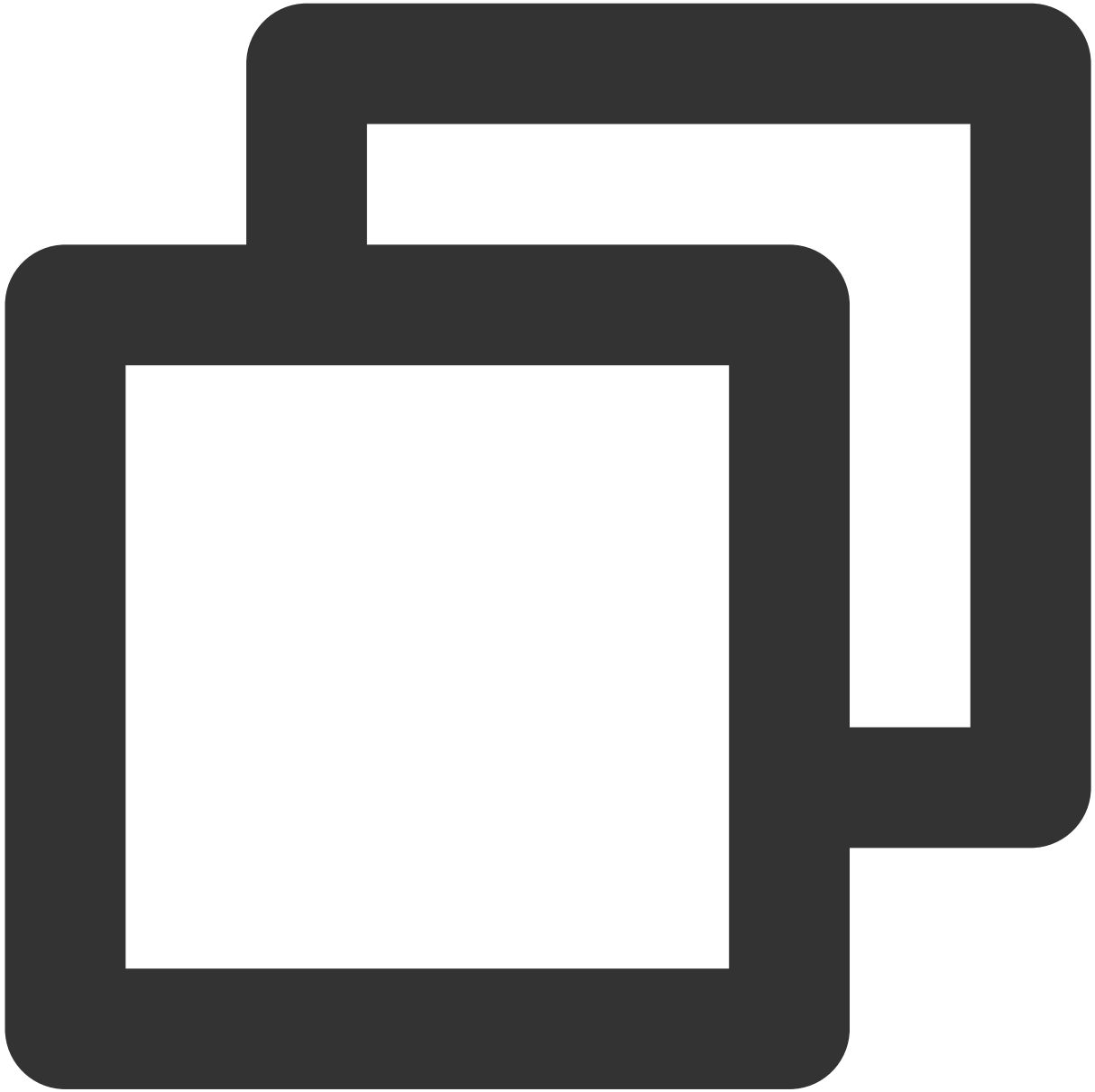
```
const cookies = new Cookies(cookieStr?: string, isSetCookie?: boolean);
```

## 参数

参数名称	类型	必填	说明
cookieStr	string	否	Cookie 字符串或者 <a href="#">Set-Cookie</a> 字符串。
isSetCookie	boolean	否	参数 cookieStr 是否是 <a href="#">Set-Cookie</a> 字符串，默认为 false。

## 方法

get



```
cookies.get(name?: string): null | Cookie | Array<Cookie>;
```

获取指定名称的 [Cookie](#) 对象。存在多个 `name` 匹配时，返回 [Cookie](#) 数组。

### 参数

--	--	--	--	--

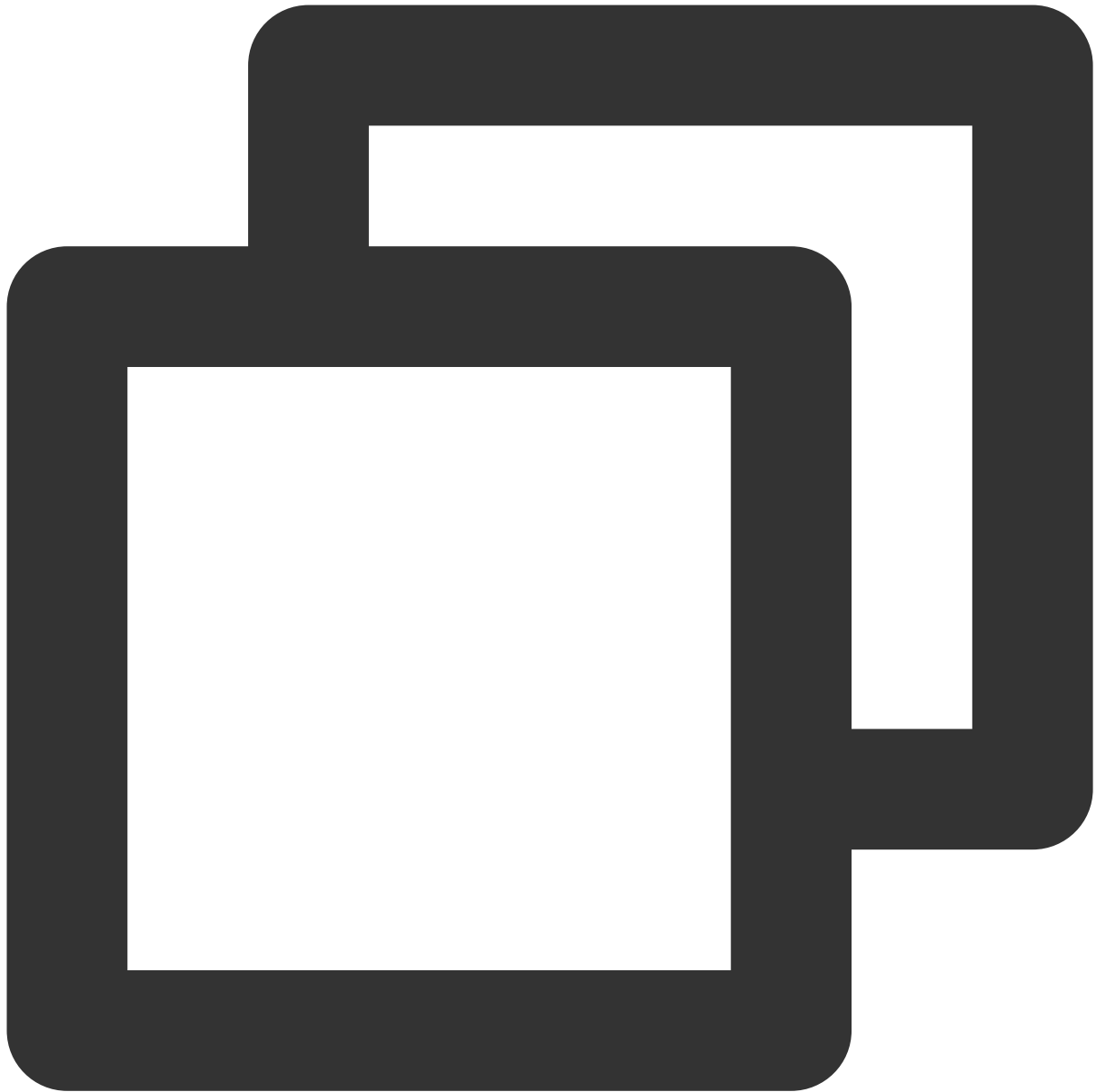
参数名称	类型	必填	说明
name	string	否	Cookie 名称，取值说明如下。 缺省 name 表示获取所有 Cookie 对象。 指定 name 表示获取指定 name 的 Cookie 对象，存在多个匹配时，返回 <a href="#">Cookie</a> 数组。

## Cookie

`Cookie` 对象属性如下，详细参见 [MDN 官方文档 Set-Cookie](#)。

属性名	类型	只读	说明
name	string	是	Cookie 名称。
value	string	是	Cookie 值。
domain	string	是	Cookie 的作用域名。
path	string	是	Cookie 的作用路径。
expires	string	是	Cookie 最长有效时间, 取值符合 <a href="#">HTTP Date</a> 首部标准。
max_age	string	是	Cookie 经过 max_age 秒失效，单位秒 (s)。
samesite	string	是	控制 Cookie 跨站点请求伪造攻击 (CSRF) 的保护。
httponly	boolean	是	禁止 JavaScript 访问 Cookie，仅限 HTTP 请求携带。
secure	boolean	是	Cookie 仅限 HTTPS 请求协议携带。

## set



```
cookies.set(name: string, value: string, options?: Cookie): boolean;
```

覆盖添加 Cookie。返回 true，表示添加成功，返回 false，表示添加失败（超过了 cookies 数量限制，详细参见 [cookies 大小限制](#)）。

#### 注意：

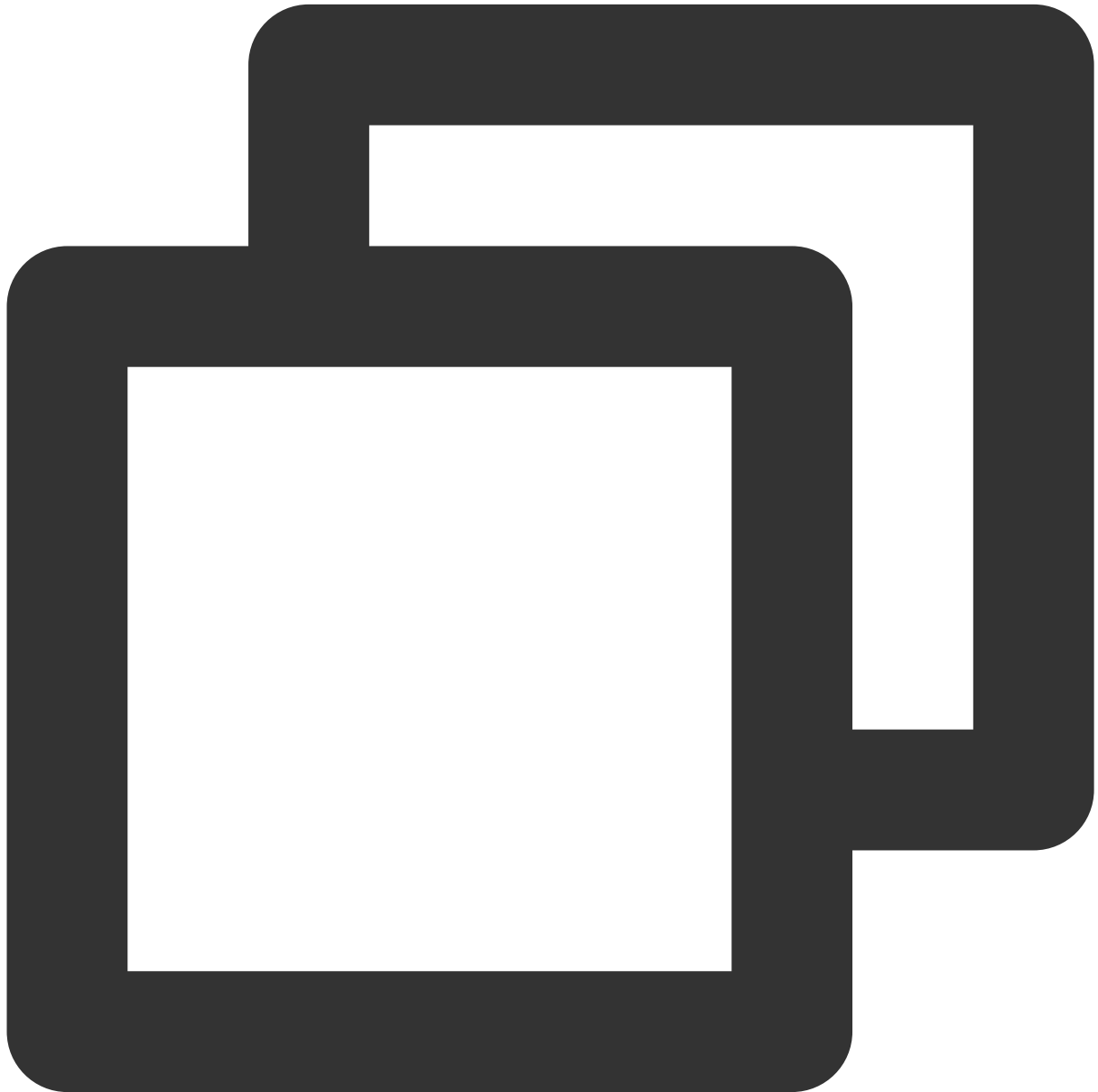
以 `name + domain + path` 为唯一 key，覆盖添加 Cookie。

#### 参数

参数名称	类型	必填	说明
------	----	----	----

name	string	是	Cookie 名称。
value	string	是	Cookie 值。
Cookie	string	否	<a href="#">Cookie</a> 属性配置项。

## append



```
cookies.append(name: string, value: string, options?: Cookie): boolean;
```

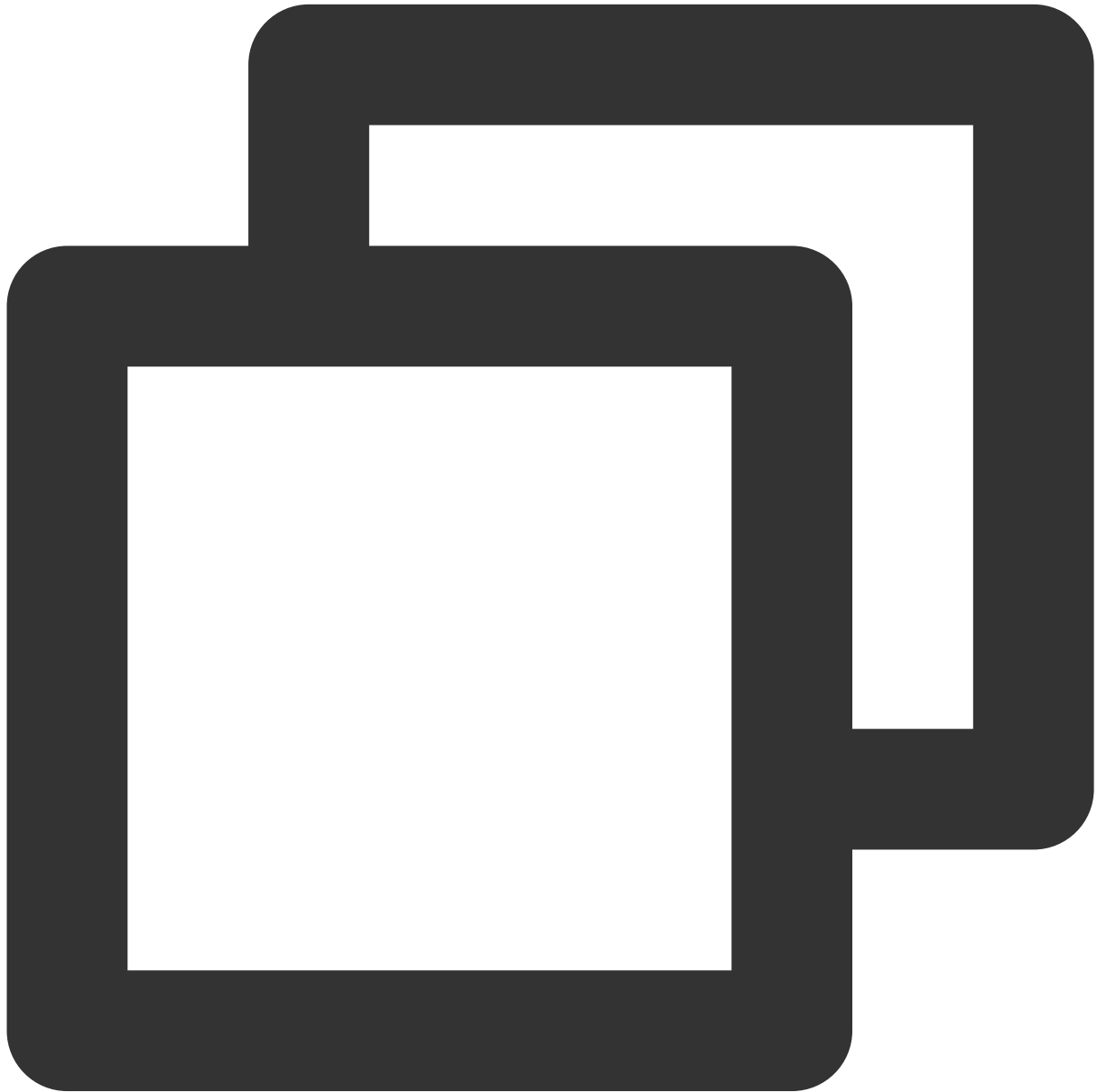


追加 Cookie，用于相同 name, 多个 value 的场景。返回 true，表示添加成功，返回 false，表示添加失败（value 重复或超过了 cookies 数量限制，详细参见 [cookies 大小限制](#)）。

**注意：**

以 `name + domain + path` 为唯一 key 追加 Cookie。

## remove



```
cookies.remove(name: string, options?: Cookie): boolean;
```

删除 Cookie。

**注意：**

以 `name + domain + path` 为唯一 key 删除 Cookie。

**参数**

参数名称	类型	必填	说明
name	string	是	Cookie 名称。
options	Cookie	是	Cookie 属性配置项，其中属性 <code>domain</code> 和 <code>path</code> 可支持 <code>*</code> ，表示匹配所有。

## 使用限制

### 特殊字符自动转义

name 值包含字符 `" ( ) , / : ; ? < = > ? @ [ ] \ { }`，`0x00~0x1F`，`0x7F~0xFF` 将被自动转义。

value 值包含字符 `, , ; " \`，`0x00~0x1F`，`0x7F~0xFF` 将被自动转义。

### cookies 大小限制

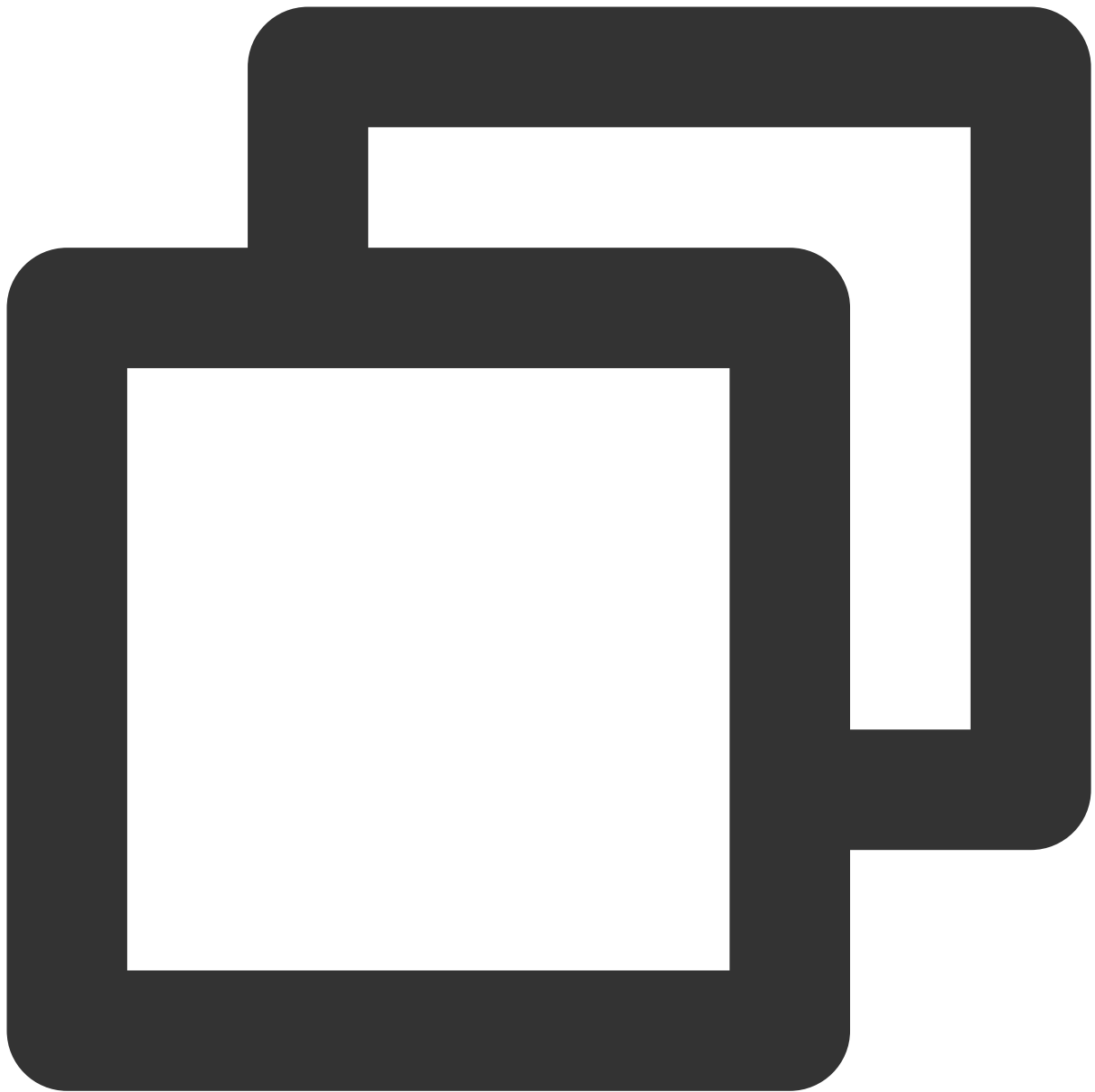
Cookie 属性 name 大小不超过 64B。

Cookie 属性 `value, domain, path, expires, max_age, samesite` 累计大小不超过 1KB。

cookies 转义后所有字段总长度不超过 4KB。

cookies 中包含的 Cookie 对象总数不超过 64个。

## 示例代码



```
function handleEvent(event) {  
  const response = new Response('hello world');  
  
  // 生成 cookies 对象  
  const cookies = new Cookies('ssid=helloworld; expires=Sun, 10-Dec-2023 03:10:01 G  
  
  // 设置响应头 Set-Cookie  
  response.setCookies(cookies);  
  
  return response;  
}
```

```
addEventListener('fetch', (event) => {  
  event.respondWith(handleEvent(event));  
});
```

## 相关参考

[MDN 官方文档：Set-Cookie](#)

[示例函数：AB测试](#)

[示例函数：设置 Cookie](#)

# Encoding

最近更新时间：2024-01-30 16:53:08

基于 Web APIs 标准 [TextEncoder](#)、[TextDecoder](#) 进行设计，实现了编码器与解码器。

## TextEncoder

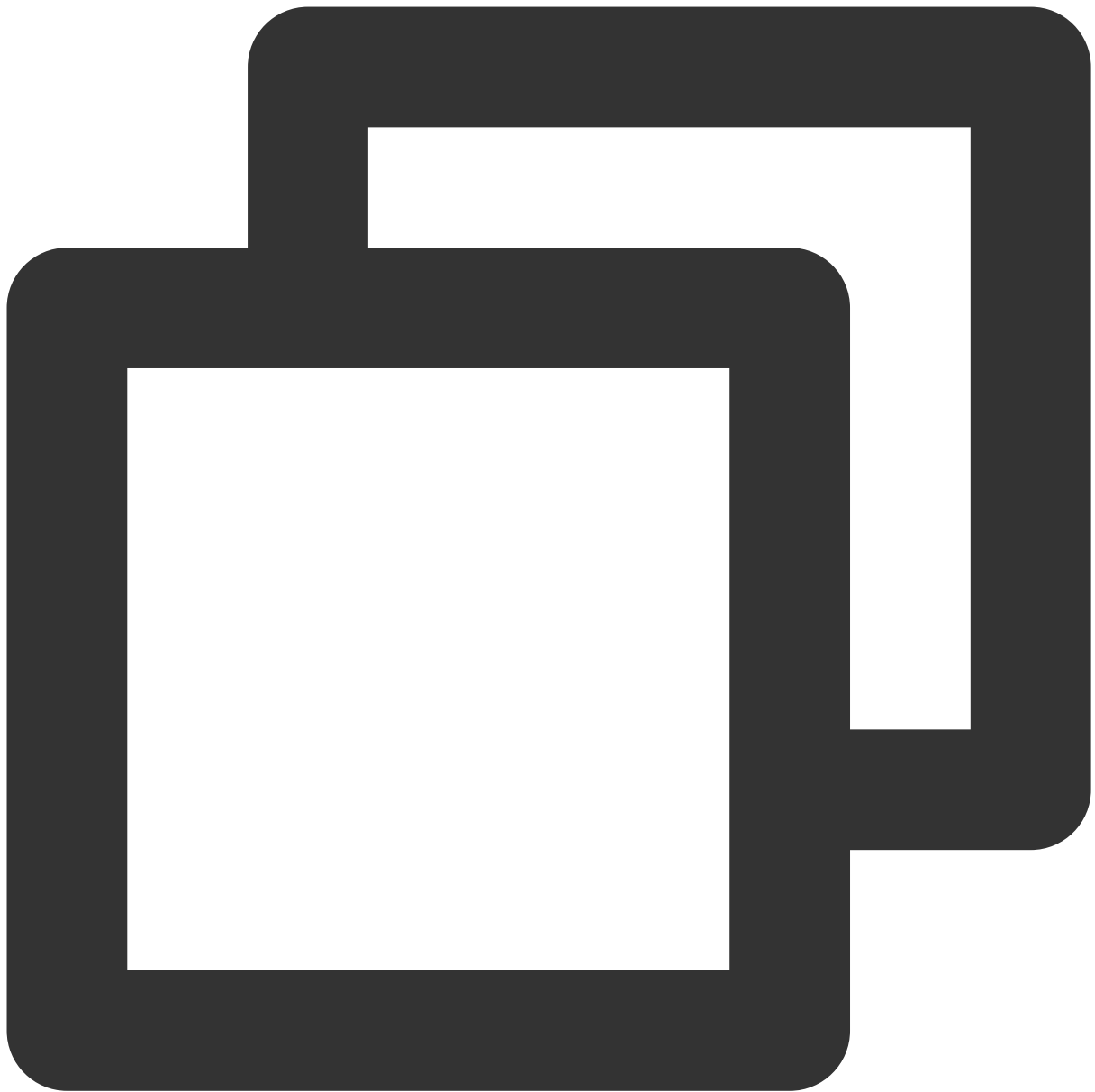
编码器，接受码点流作为输入，并输出 `UTF-8` 字节流。请参考 MDN 官方文档 [TextEncoder](#)。

### 构造函数



```
// TextEncoder 构造函数，不接受任何参数。  
const encoder = new TextEncoder();
```

## 属性

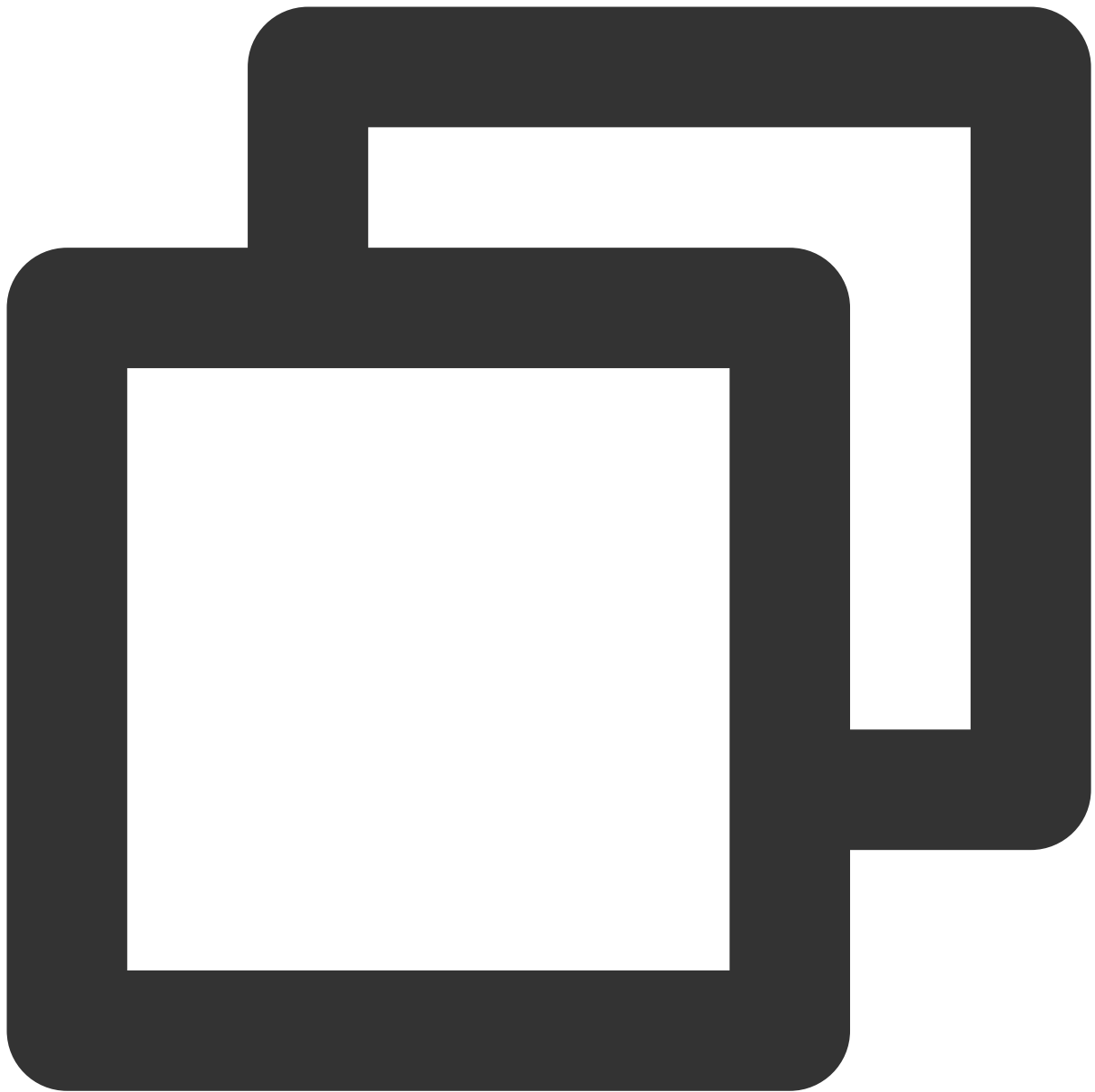


```
// encoder.encoding  
readonly encoding: string;
```

编码器的编码类型，当前值仅为 `UTF-8` 。

## 方法

**encode**



```
encoder.encode(input?: string | undefined): Uint8Array
```

接受码点流作为输入，并输出 `UTF-8` 字节流。

**注意：**

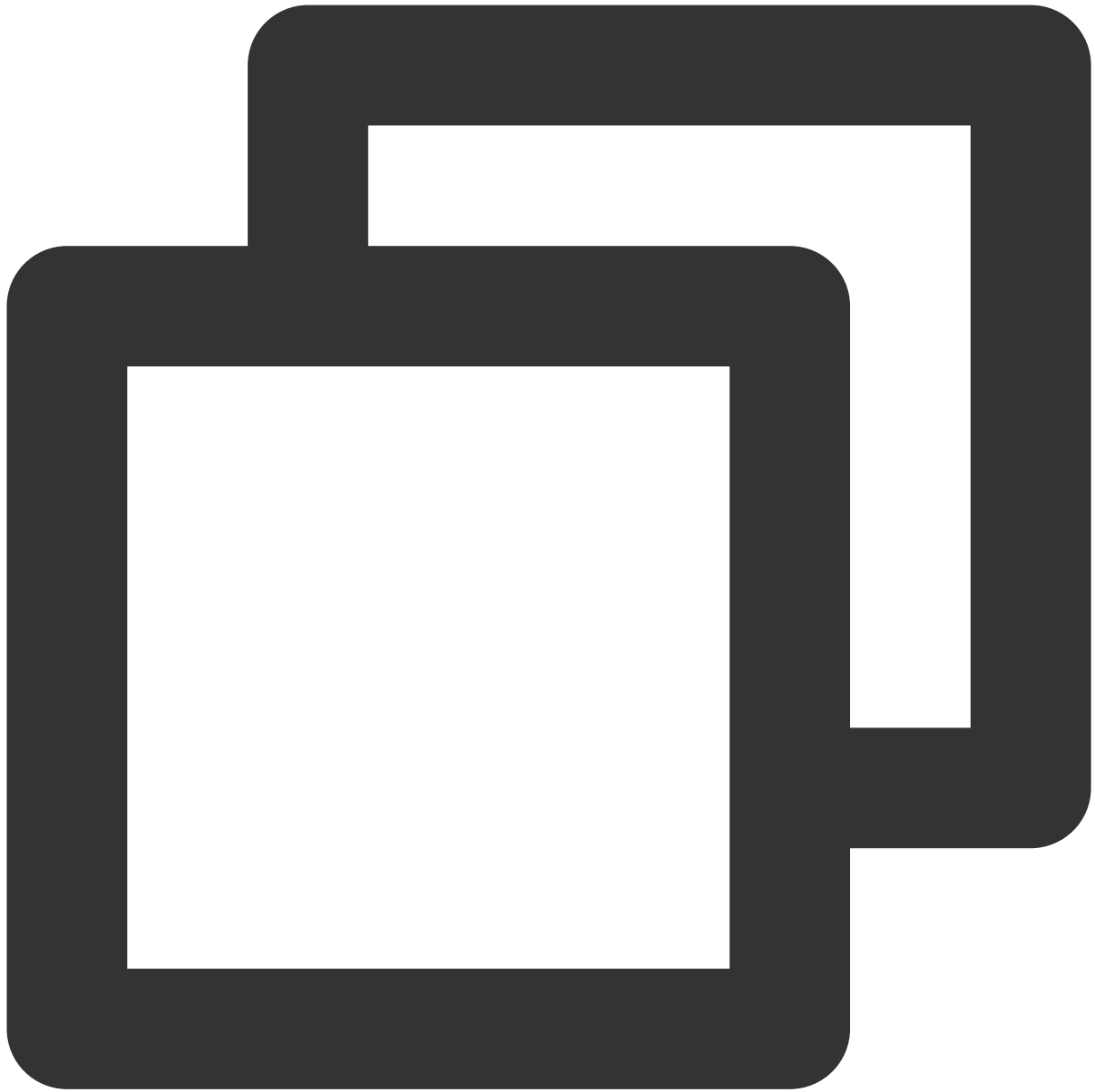
**input 最大长度为 300M**，超出长度会抛出异常。

`encoder.encode` 参数

参数名称	类型	必填	说明
input	string   undefined	否	编码器输入值。



## encodeInto



```
encoder.encodeInto(input: string, destination: Uint8Array): EncodeIntoResult;
```

接受码点流作为输入，输出 `UTF-8` 字节流，并写入到参数 `destination` 字节数组中。

## 参数

参数名称	类型	必填	说明
input	string	是	编码器输入值。
destination	<a href="#">Uint8Array</a>	是	编码器输出值。

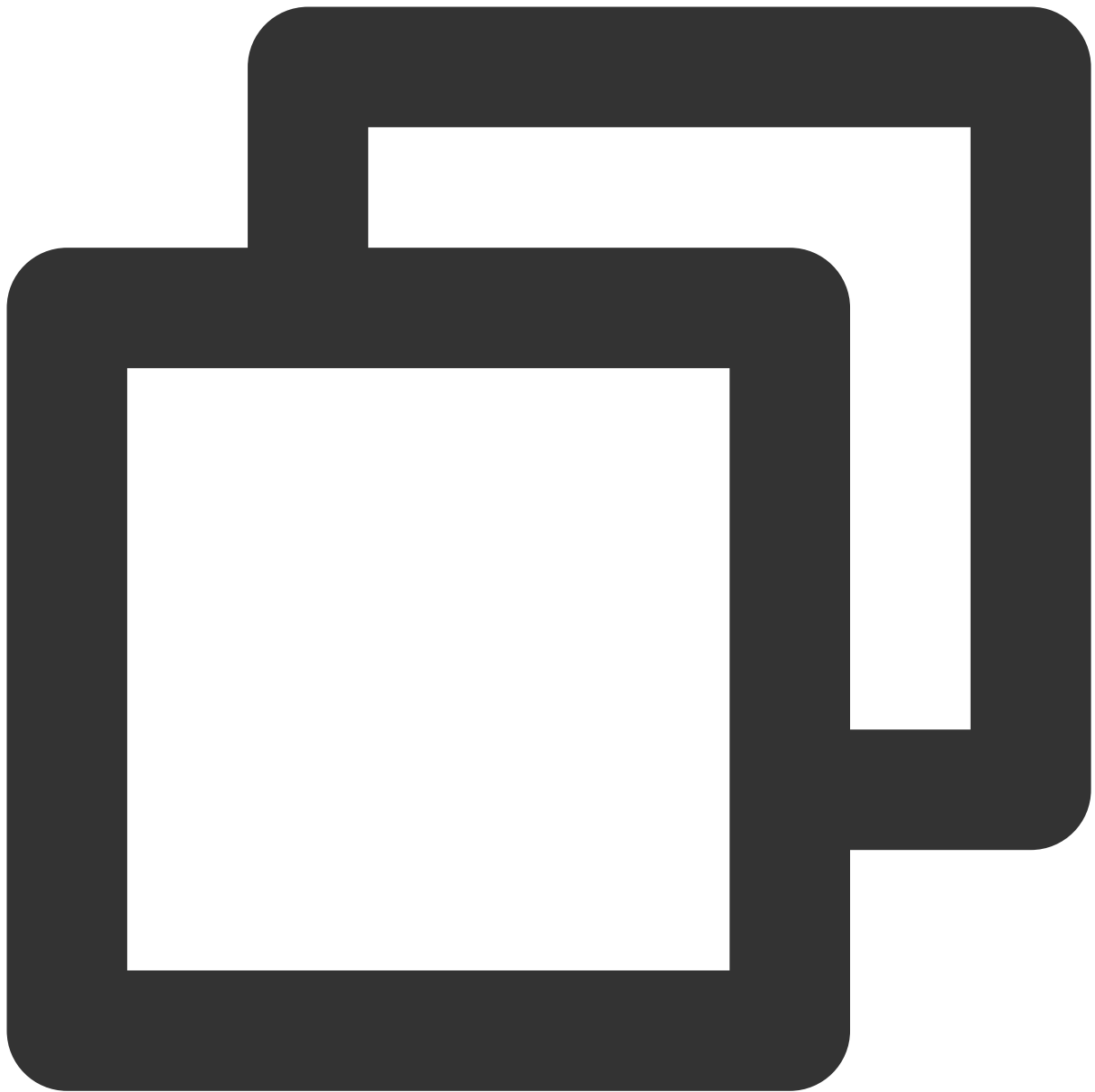
返回值 `EncodeIntoResult`

属性名	类型	说明
<code>read</code>	<code>number</code>	已转换为 UTF-8 的 UTF-16 单元数。
<code>written</code>	<code>number</code>	目标 <code>Uint8Array</code> 中修改的字节数。

## TextDecoder

解码器。将字节流作为输入，并提供码点流作为输出。请参考 MDN 官方文档 [TextDecoder](#)。

### 构造方法



```
const decoder = new TextDecoder(label?: string | undefined, options?: DecoderOption
```

## 参数

### 注意：

参数 `label`，下述的值暂不支持：

iso-8859-16。

hz-gb-2312。

csiso2022kr, iso-2022-kr。

参数名称	类型	必填	说明
label	string   undefined	否	解码器类型，默认值为 UTF-8。可选的 label 值参考 <a href="#">MDN 官方文档</a>
options	<a href="#">DecoderOptions</a>   undefined	否	解码器配置项

## DecoderOptions

解码器配置项如下所示。

属性名	类型	默认值	说明
fatal	boolean	false	标识解码失败时是否抛出异常
ignoreBOM	boolean	false	标识是否忽略 <a href="#">byte-order marker</a>

## 属性

### encoding



```
// decoder.encoding  
readonly encoding: string;
```

解码器类型。

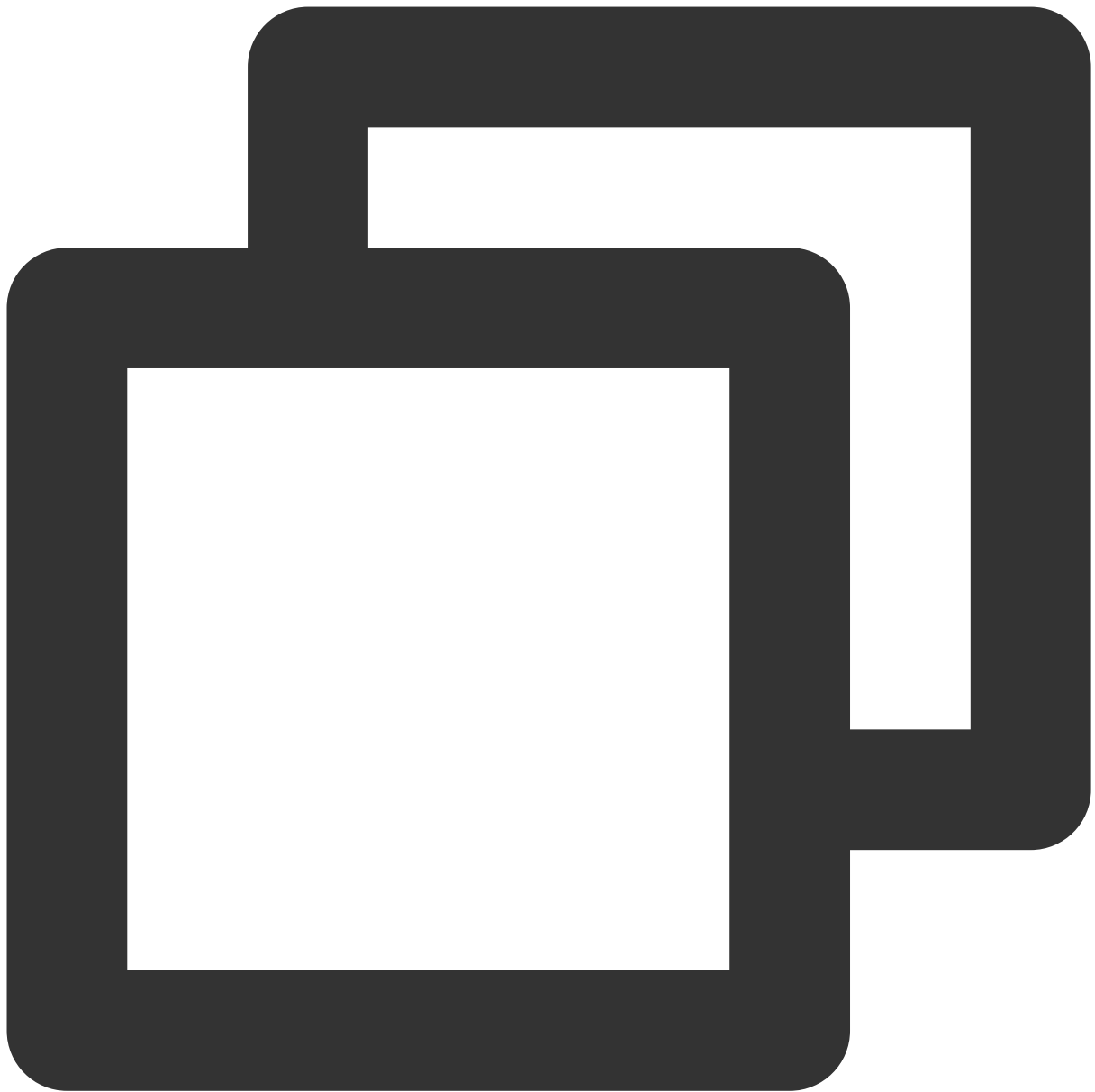
**fatal**



```
// decoder.fatal  
readonly fatal: boolean;
```

当解码失败，标识是否抛出异常。

### **ignoreBOM**

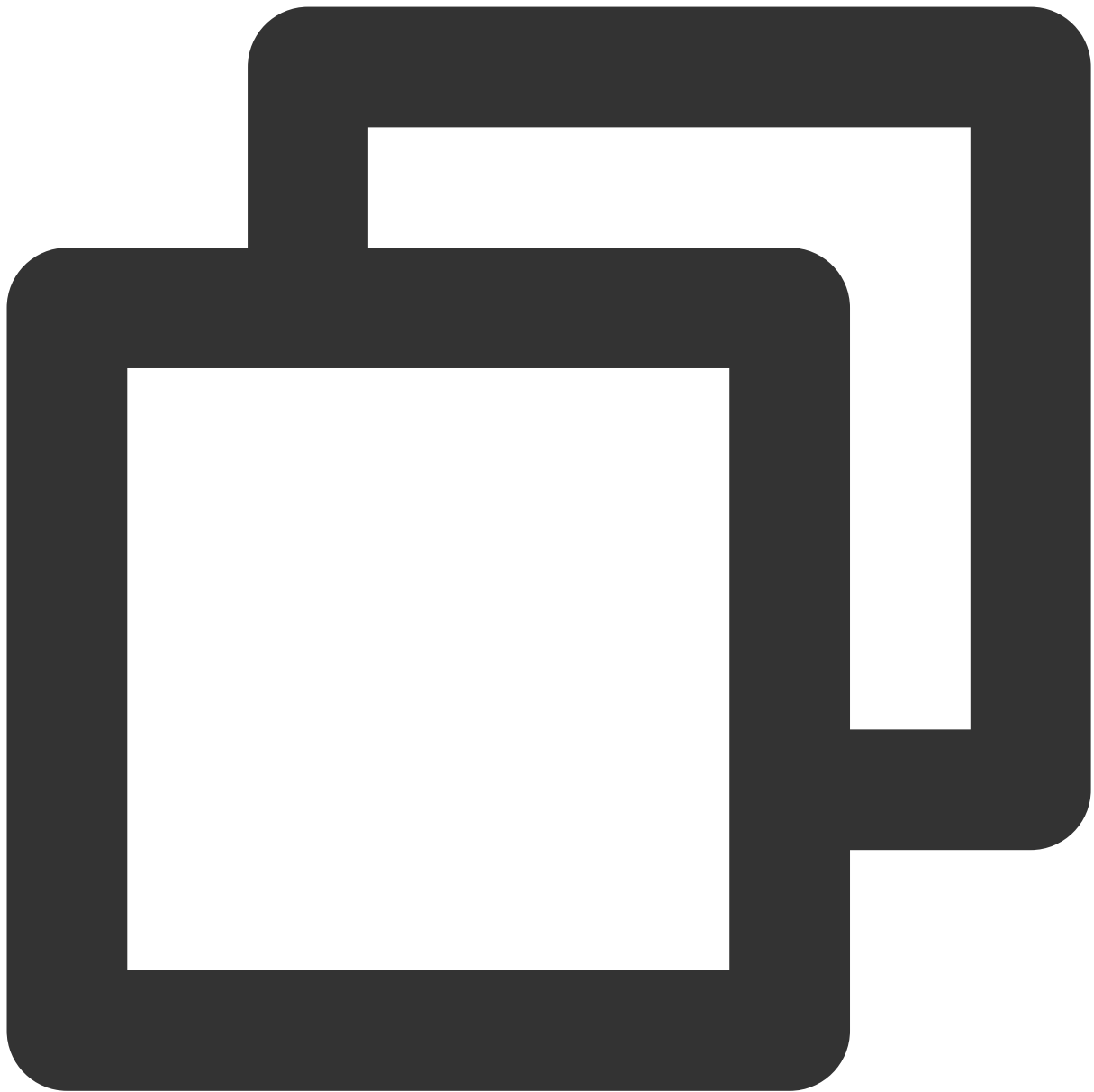


```
// decoder.ignoreBOM  
readonly ignoreBOM: boolean;
```

标识是否忽略 [byte-order marker](#)。

## 方法

**decode**



```
const result = decoder.decode(buffer?: ArrayBuffer | ArrayBufferView | undefined, o
```

**注意：**

参数 `buffer` 最大长度为 100M，超出长度会抛出异常。

参数名称	类型	必填	说明
buffer	<a href="#">ArrayBuffer</a>   <a href="#">ArrayBufferView</a>   undefined	否	待解码的字节流。 buffer 最大长度为 100M，超出长度会抛出异常。



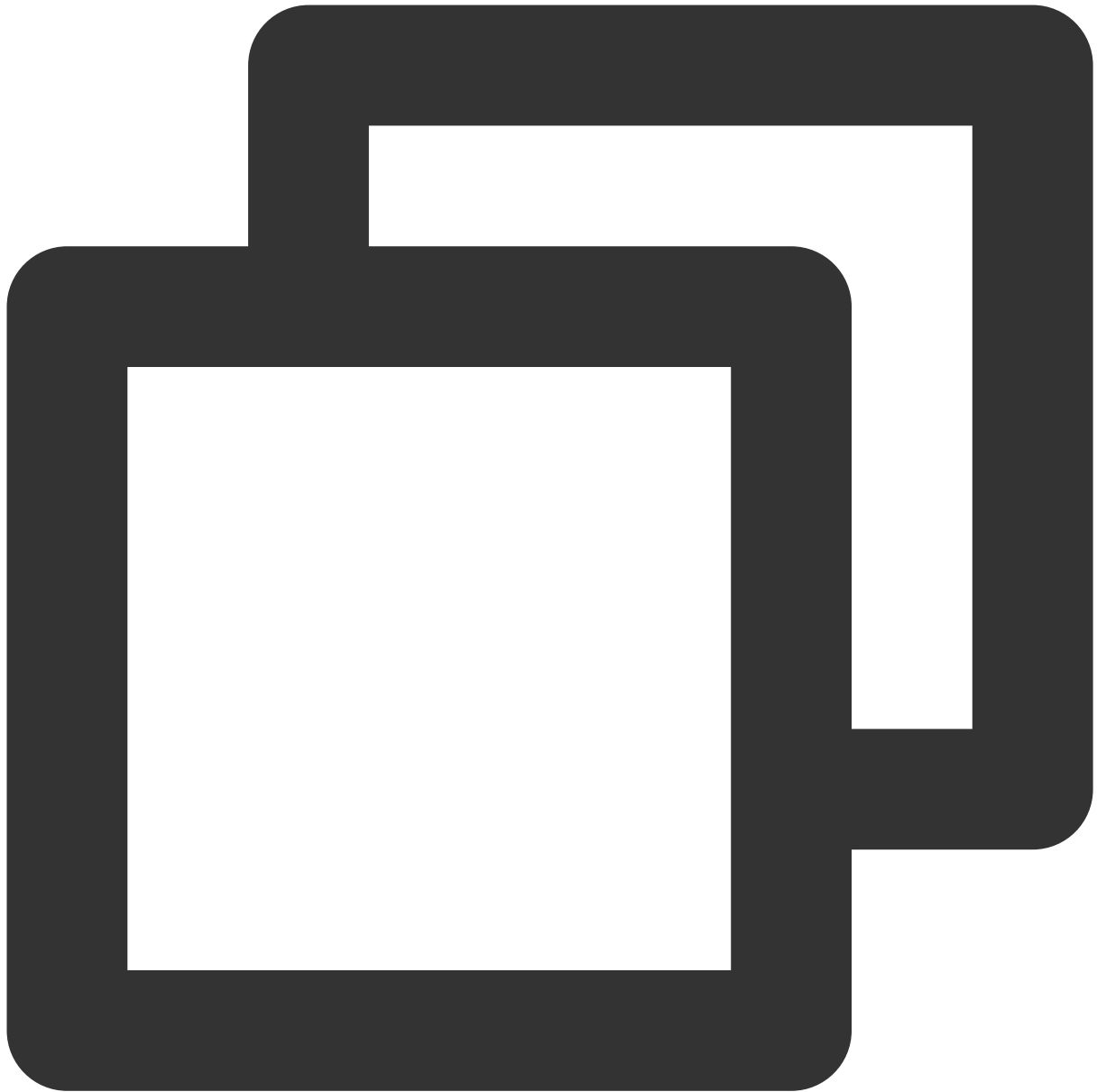
options	<a href="#">DecodeOptions</a>	否	执行解码配置项。
---------	-------------------------------	---	----------

### DecodeOptions

执行解码配置项如下所示。

属性名	类型	默认值	说明
stream	boolean	false	设置流式解码，默认为 <b>false</b> ，取值说明如下。 <b>true</b> 表示以 <b>chunk</b> 的方式处理数据，即流式解码。 <b>false</b> 表示 <b>chunk</b> 已结束或未使用 <b>chunk</b> 处理数据，即非流式解码。

### 示例代码



```
function handleEvent(event) {  
  // 编码器  
  const encoder = new TextEncoder();  
  const encodeText = encoder.encode('hello world');  
  
  // 解码器  
  const decoder = new TextDecoder();  
  const decodeText = decoder.decode(encodeText);  
  
  // 客户端响应内容  
  const response = new Response(JSON.stringify({
```

```
    encodeText: encodeText.toString(),
    decodeText,
  }));

  return response;
}

addEventListener('fetch', (event) => {
  event.respondWith(handleEvent(event));
});
```

## 相关参考

[MDN 官方文档：TextEncoder](#)

[MDN 官方文档：TextDecoder](#)

[MDN 官方文档：Encoding Label](#)

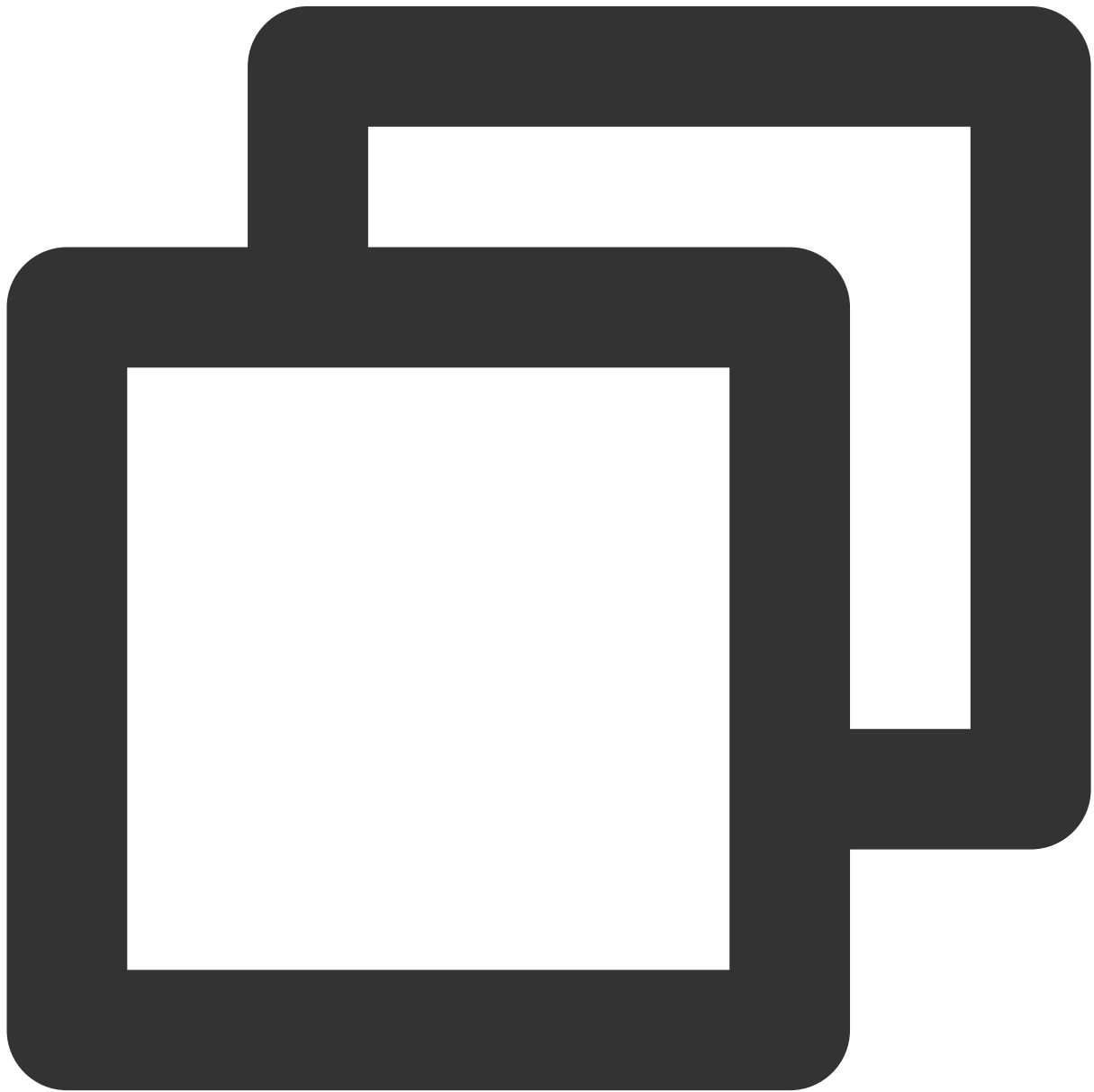
[示例函数：防篡改校验](#)

# Fetch

最近更新时间：2023-09-14 16:42:44

基于 Web APIs 标准 [Fetch API](#) 进行设计。边缘函数运行时可使用 `fetch` 发起异步请求，获取远程资源。

## 描述



```
function fetch(request: string | Request, requestInit?: RequestInit): Promise<Respo
```

## 参数

参数名称	类型	必填	说明
request	string   <a href="#">Request</a>	是	指定将要获取的请求资源。
requestInit	<a href="#">RequestInit</a>	否	请求对象的初始化配置项。详情请参见 <a href="#">RequestInit</a> 。

## 高级功能

使用 `fetch` 时，可以通过传入特定参数实现更精细的控制和定制化逻辑。主要包含访问 EdgeOne 节点缓存或回源，图片处理，重定向。

### 访问 EdgeOne 节点缓存或回源

当客户端访问某个已接入 EdgeOne 站点的[加速域名](#)时（如：`www.example.com`），同时该请求触发了边缘函数执行，此时在该边缘函数中实现 `fetch(www.example.com)` 请求，该请求将访问 EdgeOne 节点缓存，若不存在缓存，则进行回源。

**说明：** `fetch` 访问 EdgeOne 节点缓存与回源，需满足以下条件：

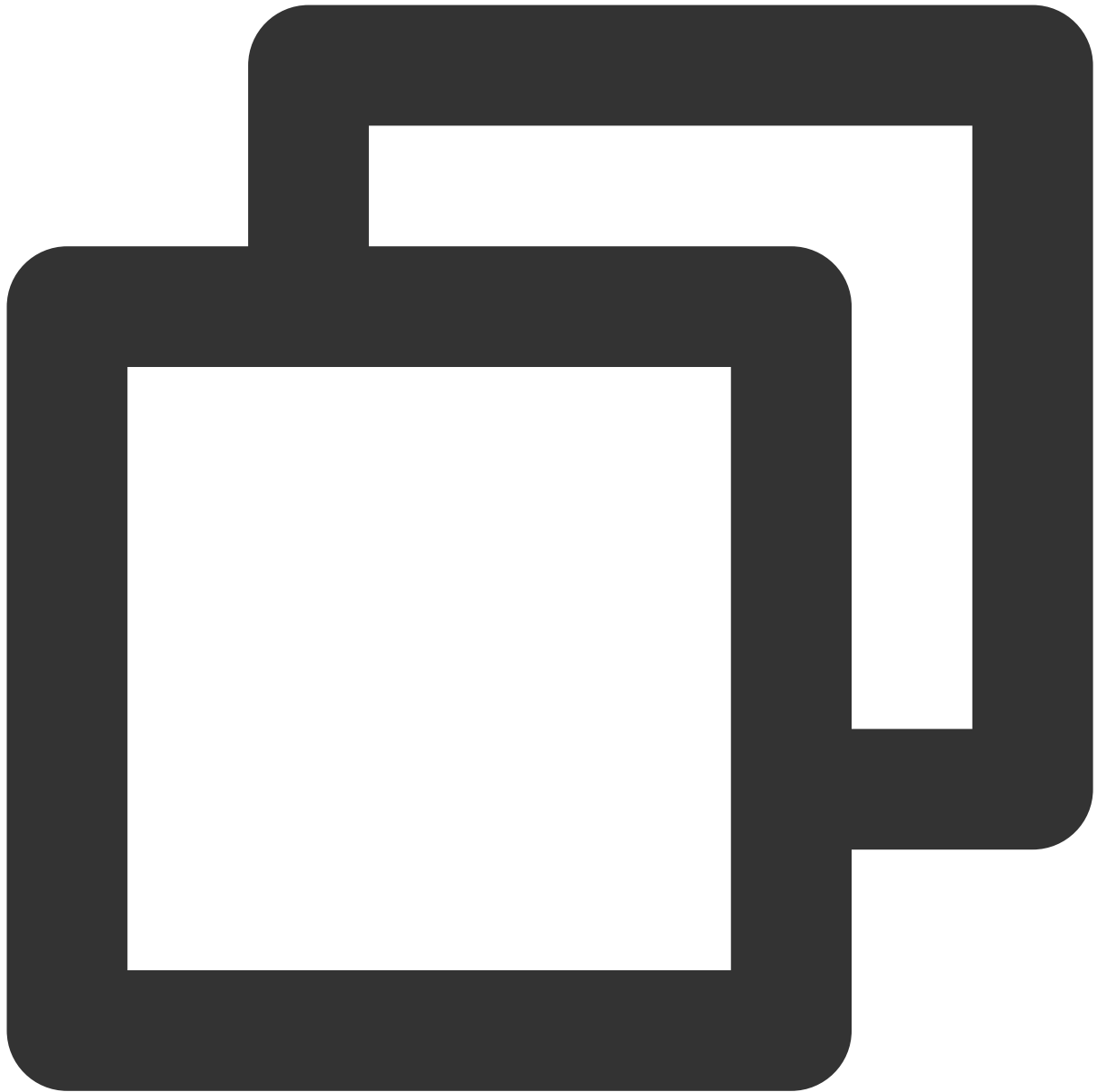
1. 客户端访问 EdgeOne 接入站点的加速域名，同时该请求触发了边缘函数执行。
2. `fetch(request)` 指定的 `request.url` 中的 HOST 和客户端请求 URL 中的 HOST 相同。
3. `fetch(request)` 指定的 `request.headers.host` 和客户端请求头 HOST 值相同。

`fetch(event.request)` 获取 EdgeOne 缓存与回源。



```
addEventListener('fetch', (event) => {  
  // fetch(event.request) 获取 EdgeOne CDN 缓存与回源。  
  const response = fetch(event.request);  
  
  event.respondWith(response);  
});
```

`fetch(url)` 获取 EdgeOne 缓存与回源。



```
addEventListener('fetch', (event) => {
  event.respondWith(handleEvent(event));
});

async function handleEvent(event) {
  const { request } = event;
  const urlInfo = new URL(request.url);
  // 回源 URL 改写
  const url = `${urlInfo.origin}/h5/${urlInfo.pathname}`;

  // fetch(url) 获取 EdgeOne CDN 缓存与回源。
```

```
const response = await fetch(url);
return response;
}
```

## 图片处理

`fetch` 支持传入参数 `requestInit.eo.image` 对图片进行缩放或格式转换，详情参见图片处理的参数配置项 [ImageProperties](#)。

### 说明：

使用 `fetch(request, requestInit)` 实现图片处理时，需要同时满足 `fetch` 获取 EdgeOne 节点缓存与回源的条件。

## 重定向

`fetch` 支持 `3xx` 重定向状态码。可使用第二个参数 `requestInit.redirect` 属性进行设置，更多重定向配置，请查看 [RequestInit](#)。

重定向规则遵从 Web APIs 标准 [fetch API](#)，针对不同状态码有不同的跟随规则：

状态码	重定向规则
301、302	POST 方法被转为 GET 方法。
303	除 HEAD / GET 外的所有方法都被转为 GET 方法。
307、308	保留原始方法。

### 注意

重定向的地址来源于响应头 `Location`，若无该响应头，则不会重定向。

响应头 `Location` 值可以是绝对 URL 或者相对 URL，详情参见 [RFC-3986: URI Reference](#)。

## 运行时限制

边缘函数中使用 `fetch` 发起请求，存在以下限制：

次数限制：边缘函数单次运行中可发起的 `fetch` 总次数为 64，超过该限制的 `fetch` 请求会请求失败，并抛出异常。

并发限制：边缘函数单次运行中允许发起 `fetch` 最大并发数为 8，超过该限制的 `fetch` 请求会被延迟发起，直到某个正在运行着的 `fetch` 被 `resolve`。

### 注意

每一次重定向都会计入请求次数，且其优先级高于新发起的 `fetch` 请求。



---

## 相关参考

[MDN 官方文档：fetch](#)

示例函数：获取远程资源返回

# FetchEvent

最近更新时间：2024-01-30 16:47:27

**FetchEvent** 代表任何传入的 HTTP 请求事件，边缘函数通过注册 `fetch` 事件监听器实现对 HTTP 请求的处理。

## 描述

在边缘函数中，使用 `addEventListener` 注册 `fetch` 事件监听器，生成 HTTP 请求事件 `FetchEvent`，进而实现对 HTTP 请求的处理。

### 注意：

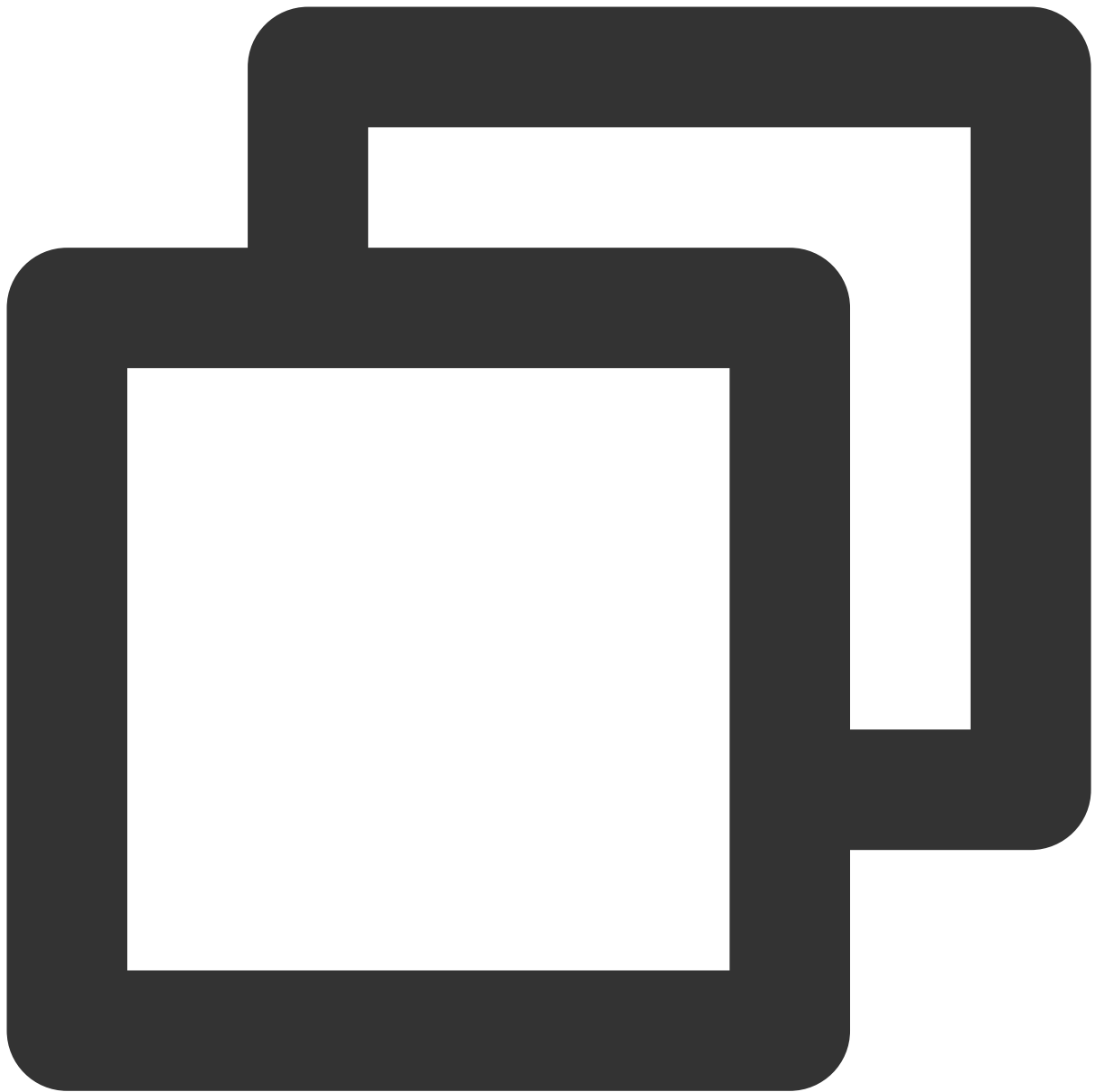
不支持直接构造 `FetchEvent` 对象，使用 `addEventListener` 注册 `fetch` 事件获取 `event` 对象。



```
// event 为 FetchEvent 对象
addEventListener('fetch', (event) => {
  event.respondWith(new Response('hello world!'));
});
```

## 属性

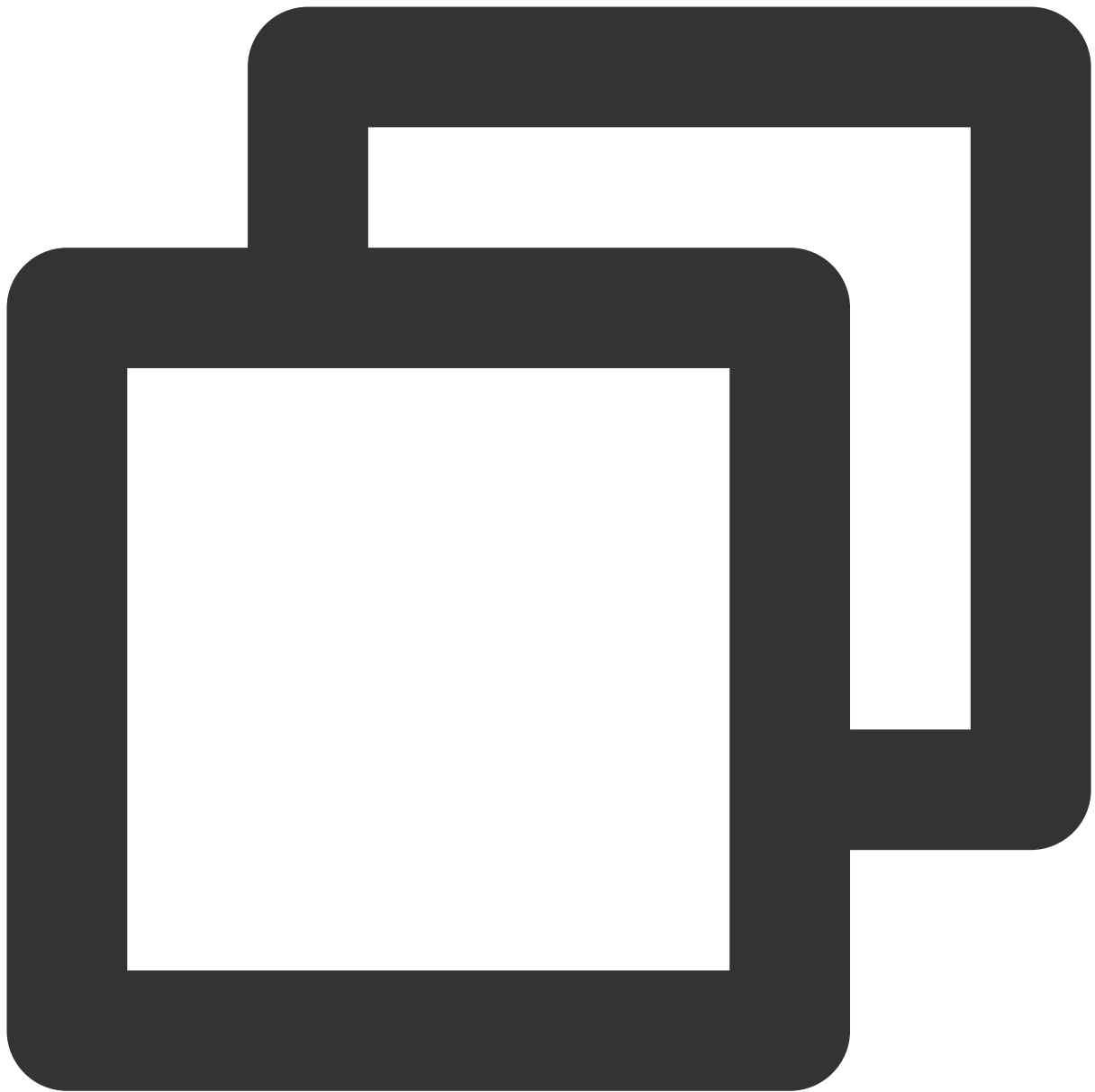
**clientId**



```
// event.clientId  
readonly clientId: string;
```

边缘函数为每一个请求分配的 id 标识。

**request**

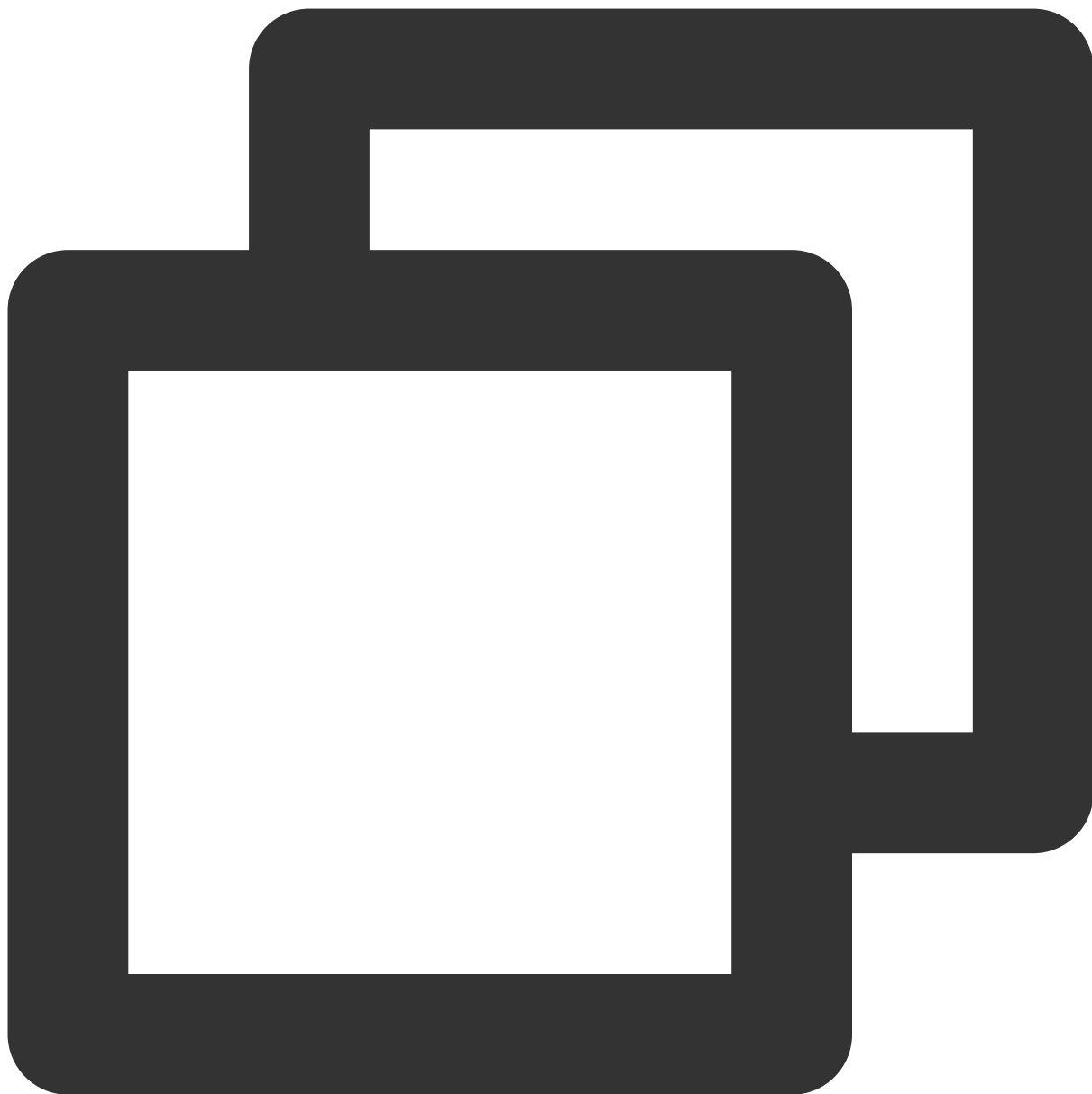


```
// event.request  
readonly request: Request;
```

客户端发起的 HTTP 请求对象，详情参见 [Request](#)。

## 方法

**respondWith**



```
event.respondWith(response: Response | Promise<Response>): void;
```

边缘函数接管客户端的请求，并使用该方法，返回自定义响应内容。

#### 注意：

事件监听器 `addEventListener` 的 `fetch` 事件回调中，需要调用接口 `event.respondWith()` 响应客户端，若未调用该接口，边缘函数会将当前请求转发回源站。

#### 参数

参数名称	类型	必填	说明
------	----	----	----

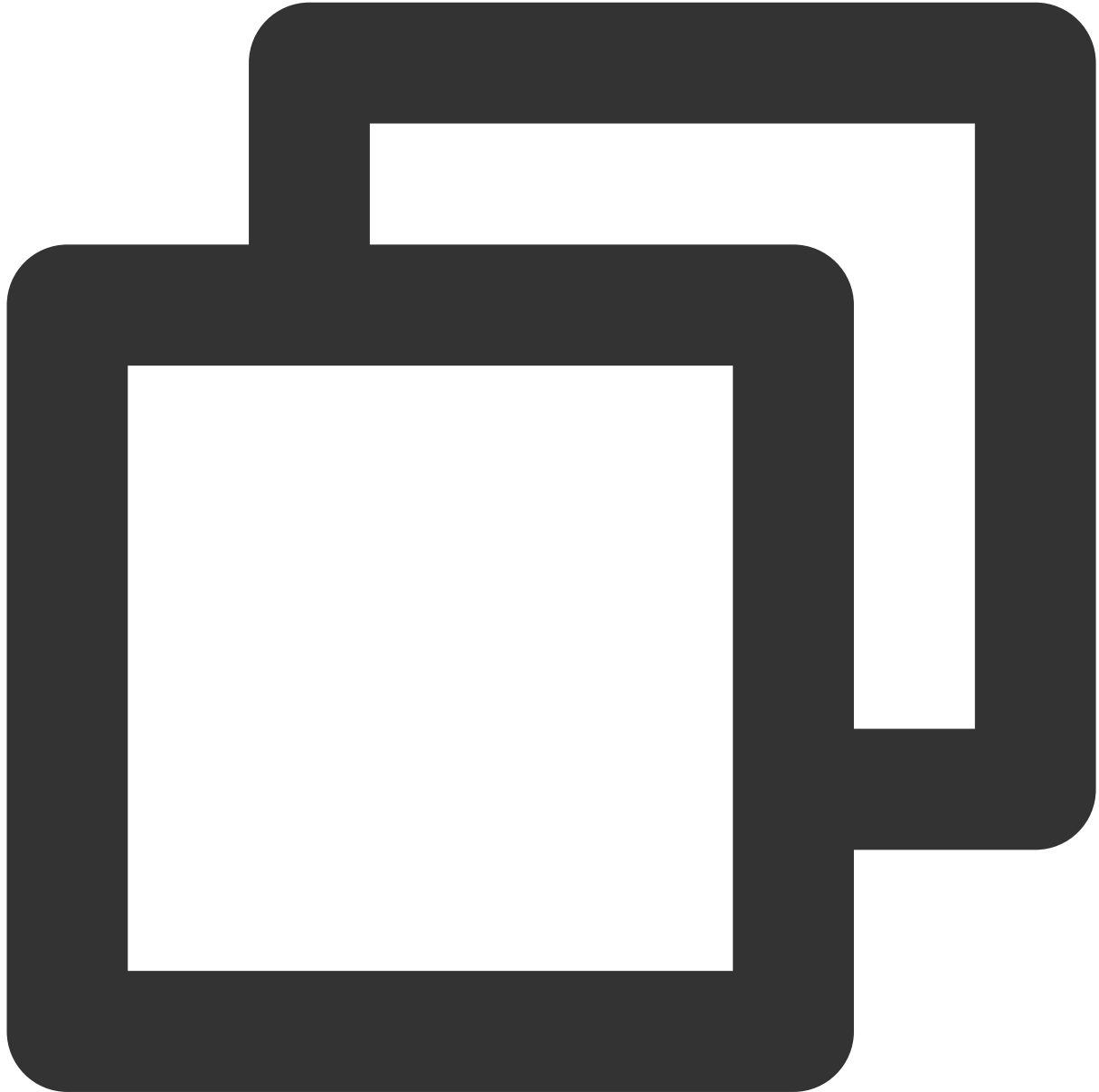
response

[Response](#) |Promise<[Response](#)>

是

客户端 HTTP 请求的响应。

## waitUntil



```
event.waitUntil(task: Promise<any>): void;
```

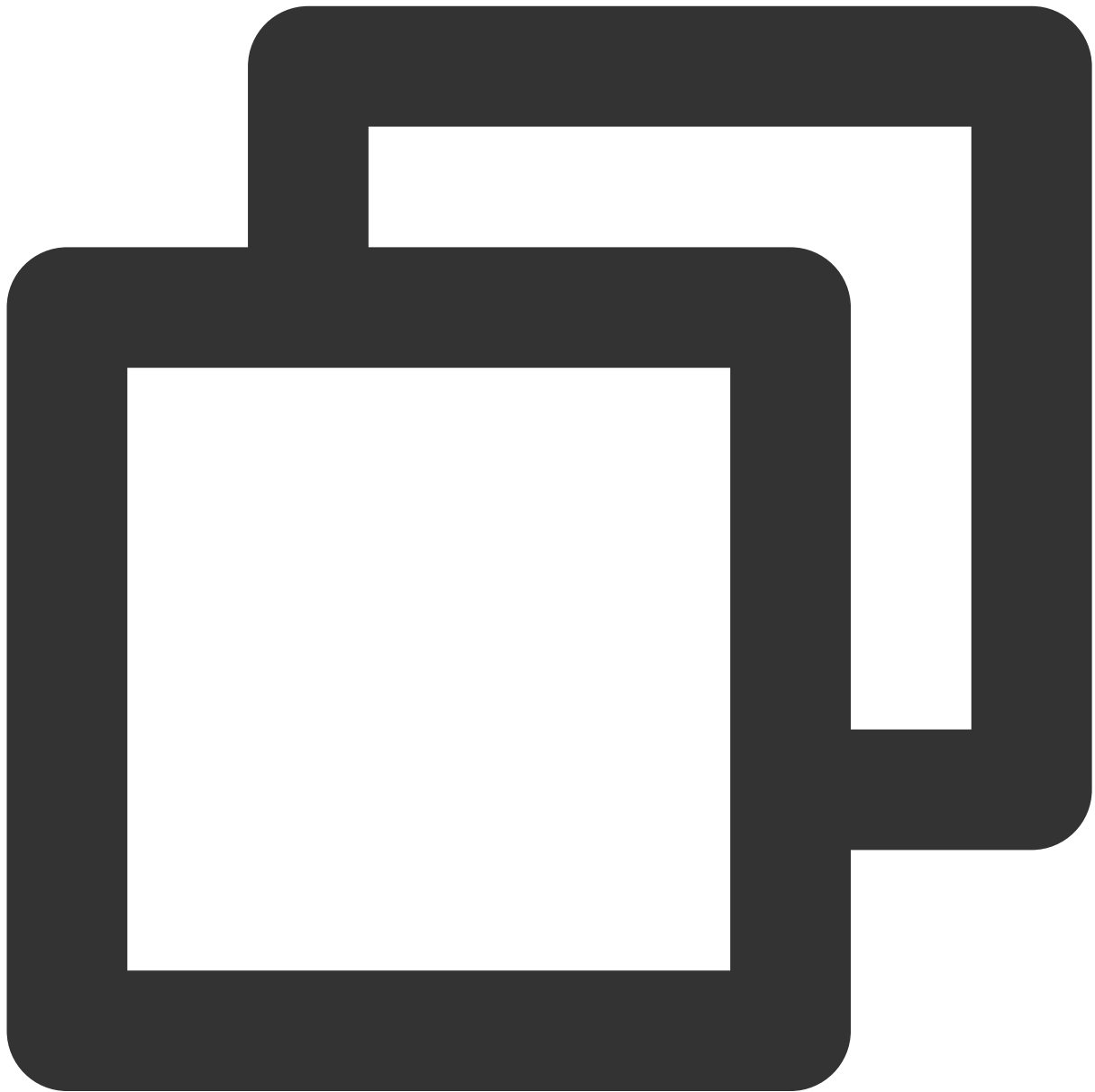
用于通知边缘函数等待 `Promise` 完成，可延长事件处理的生命周期。

### 参数

--	--	--	--

参数名称	类型	必填	说明
task	Promise< <a href="#">Response</a> >	是	等待完成的 Promise 任务。

## passThroughOnException



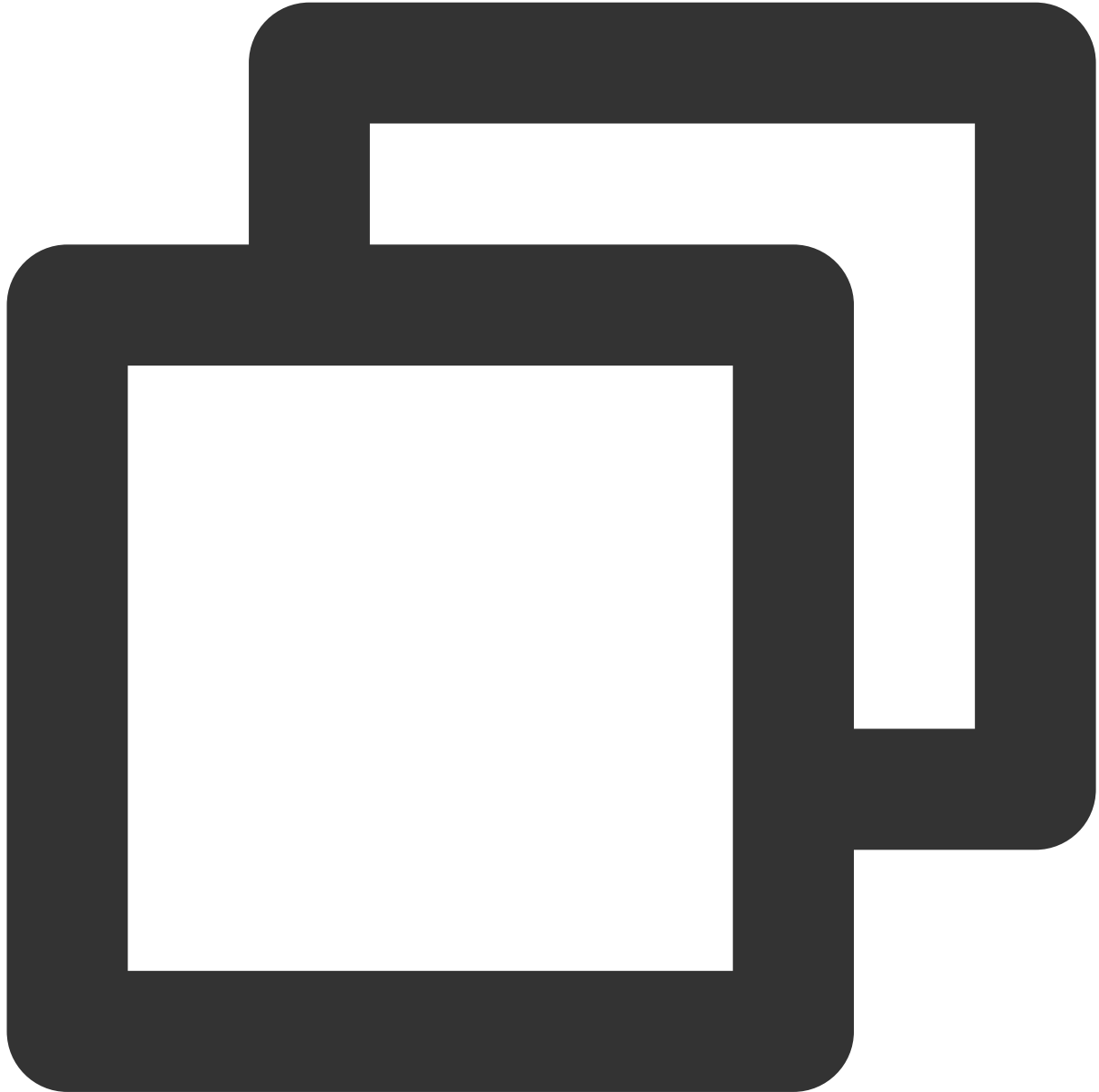
```
event.passThroughOnException(): void;
```

用于防止运行时响应异常信息。当函数代码抛出未处理的异常时，边缘函数会将此请求转发回源站，进而增强服务的可用性。



## 示例代码

未调用接口 `event.respondWith` ，边缘函数将当前请求转发回源站。

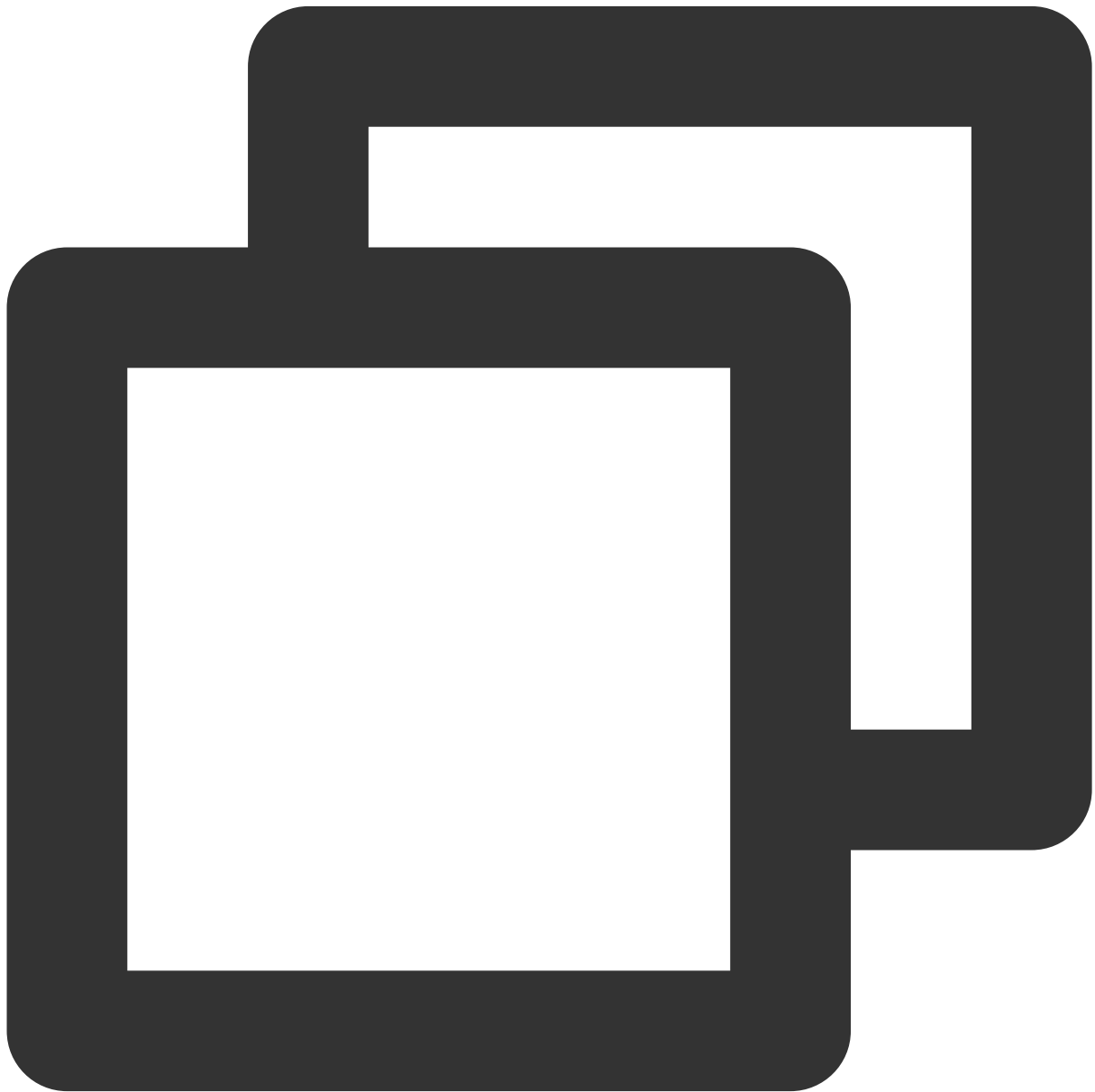


```
function handleRequest(request) {
  return new Response('Edge Functions, Hello World!');
}

addEventListener('fetch', event => {
  const request = event.request;
  // 请求 url 包含字符串 /ignore/ ，边缘函数会将当前请求转发回源站。
});
```

```
if (request.url.indexOf('/ignore/') !== -1) {  
  // 未调用接口 event.respondWith  
  return;  
}  
  
// 在边缘函数中, 自定义内容响应客户端  
event.respondWith(handleRequest(request));  
});
```

当函数代码抛出未处理的异常时, 边缘函数会将此请求转发回源站。



```
addEventListener('fetch', event => {
```

```
// 当函数代码抛出未处理的异常时，边缘函数会将此请求转发回源站
event.passThroughOnException();
throw new Error('Throw error');
});
```

## 相关参考

[MDN 官方文档：FetchEvent](#)

[示例函数：返回 HTML 页面](#)

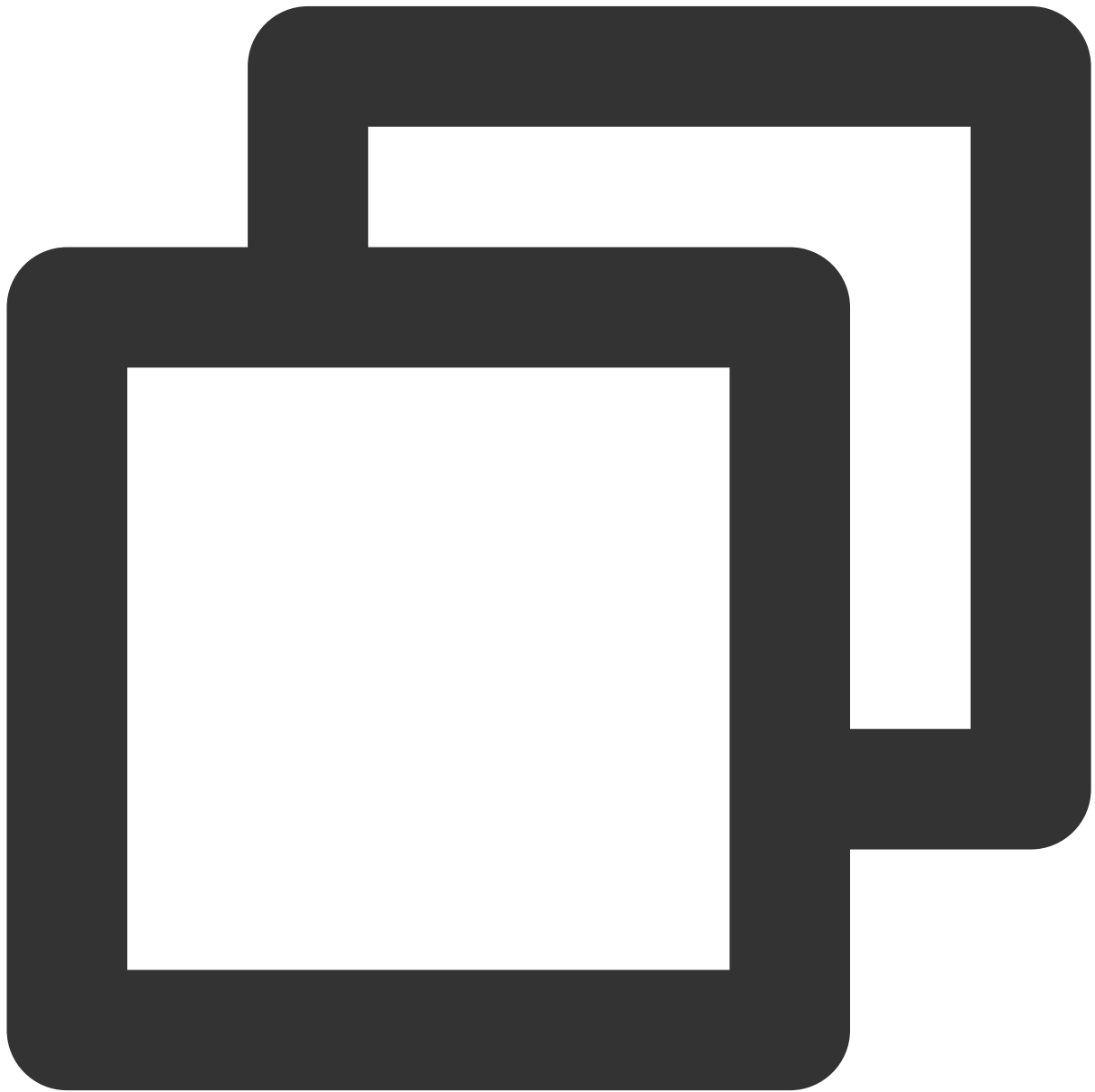
[示例函数：Cache API 使用](#)

# Headers

最近更新时间：2023-09-12 09:50:50

**Headers** 基于 Web APIs 标准 [Headers](#) 进行设计。可用于 HTTP request 和 response 的头部操作。

## 构造函数



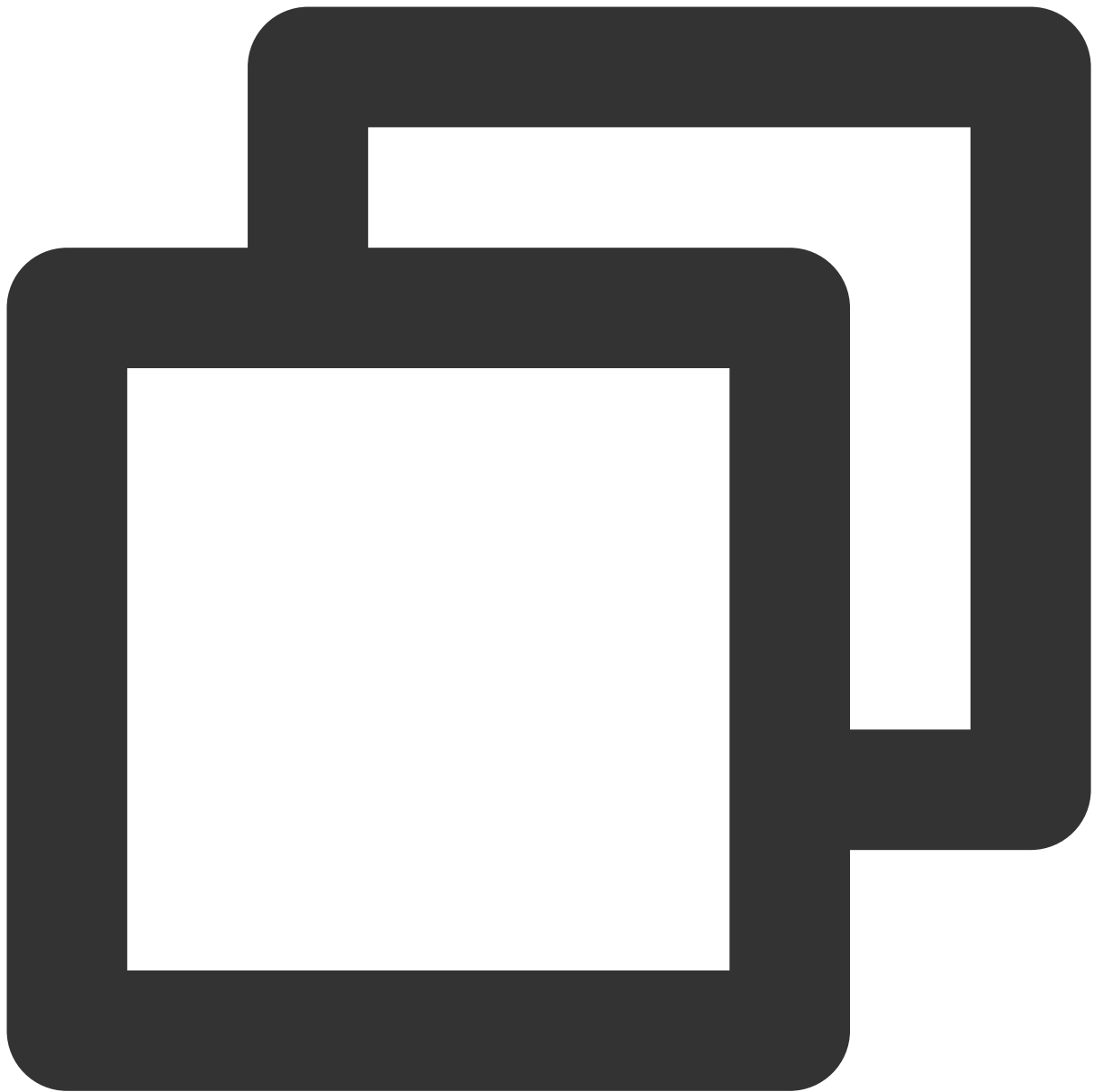
```
const headers = new Headers(init?: object | Array<[string, string]> | Headers);
```

## 参数

参数名称	类型	必填	说明
init	object   Array<[string, string]>   Headers	否	<p>初始化 Headers 对象，参数类型说明如下：</p> <p><b>object</b></p> <p>构造函数将会枚举 Object 包含的所有可枚举属性，并初始化到新的 Headers 对象中。</p> <p><b>Array&lt;[string, string]&gt;</b></p> <p>数组的每一个元素为 key/value 的键值对（如：[key, value]），构造函数遍历数组，并初始化到新的 Headers 对象中。</p> <p><b>Headers</b></p> <p>拷贝 Headers 对象，并把所有字段初始化到新的 Headers 对象中。</p>

## 方法

### append



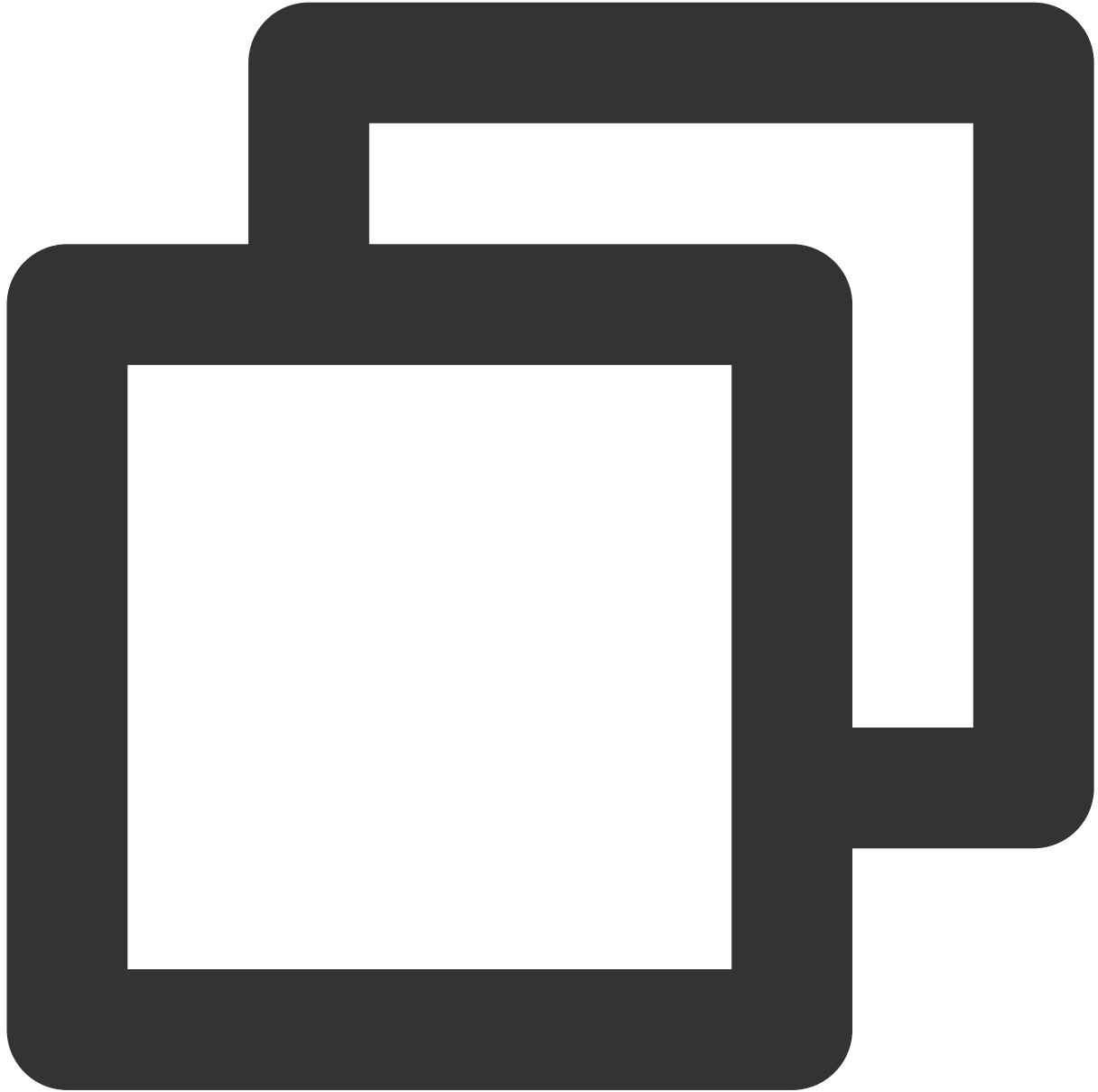
```
headers.append(name: string, value: string): void;
```

在 `headers` 对象指定的 `header` 上追加一个新值，若 `header` 不存在，则直接添加。

### 参数

属性名	类型	必填	说明
name	string	是	header 名
value	string	是	追加的新值

## delete



```
headers.delete(name: string): void;
```

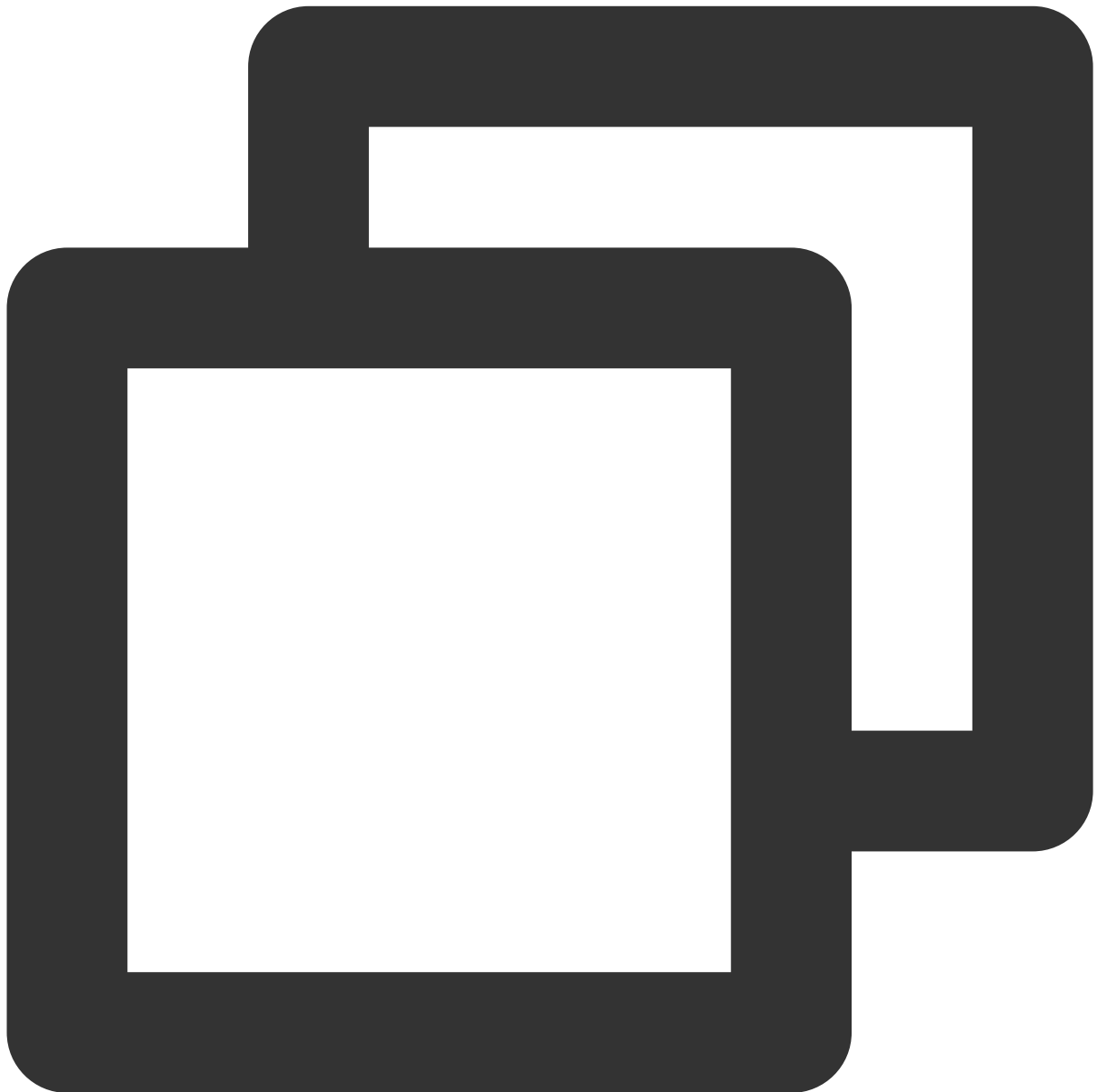
从 `headers` 对象中删除指定 `header`。

## 参数

属性名	类型	必填	说明
-----	----	----	----

name	string	是	header 名
------	--------	---	----------

## entries

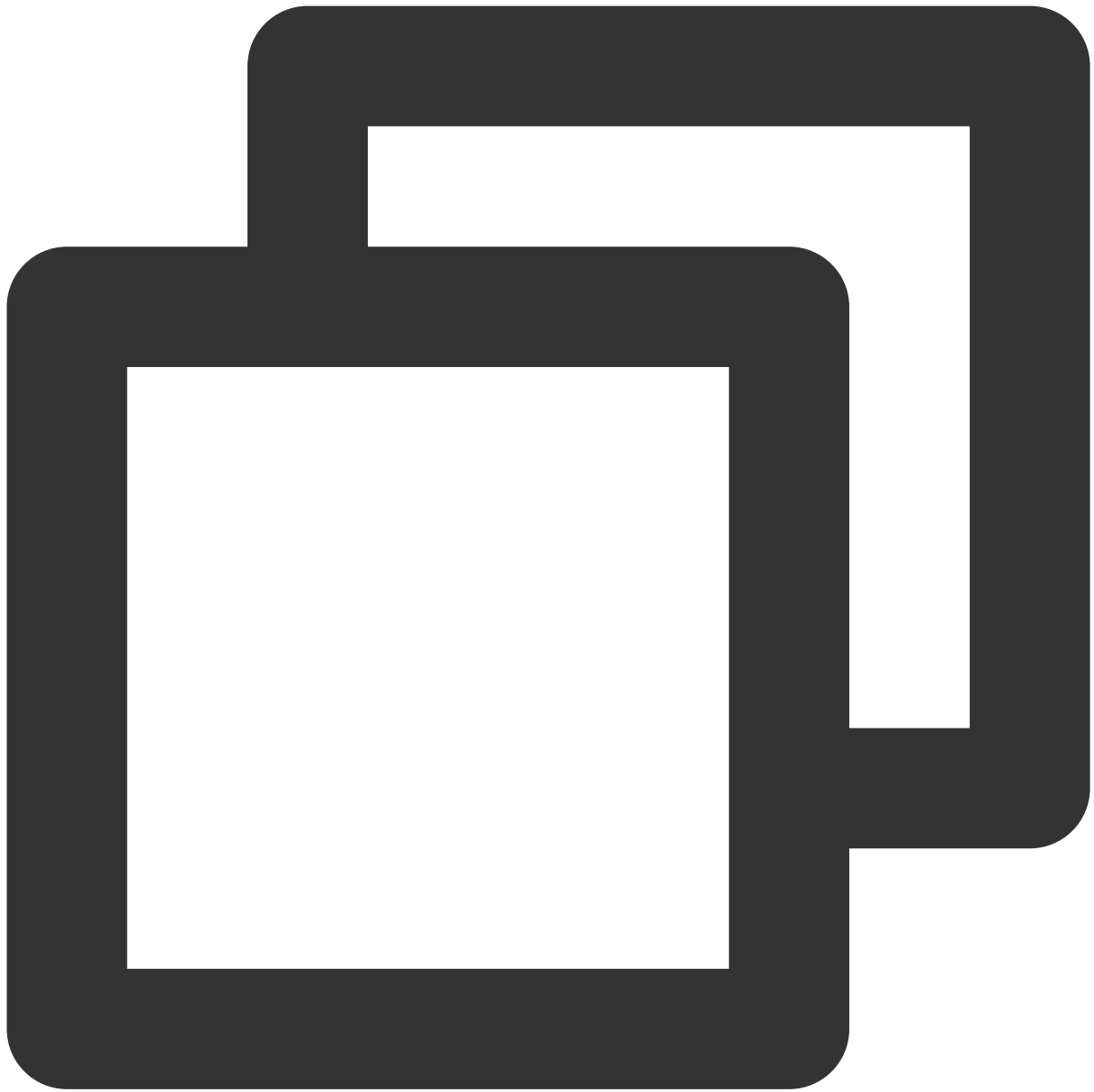


```
headers.entries(): iterator;
```

获取 `headers` 对象所有的键值对 (`[name, value]`) 数组，返回值参考 [MDN 官方文档 : iterator](#)。



## forEach



```
headers.forEach(callback: (name: string, value: string) => void | number): void;
```

遍历 `headers` 对象所有的 `header`。若 `callback` 返回非零值，表示终止遍历。

**注意**

`forEach` 为非 Web APIs 标准方法。为了提供高效遍历 `headers` 的方式，边缘函数基于 Web APIs 标准进行了扩展实现。

**get**



```
headers.get(name: string): string;
```

从 `headers` 对象中获取指定 header 的值。

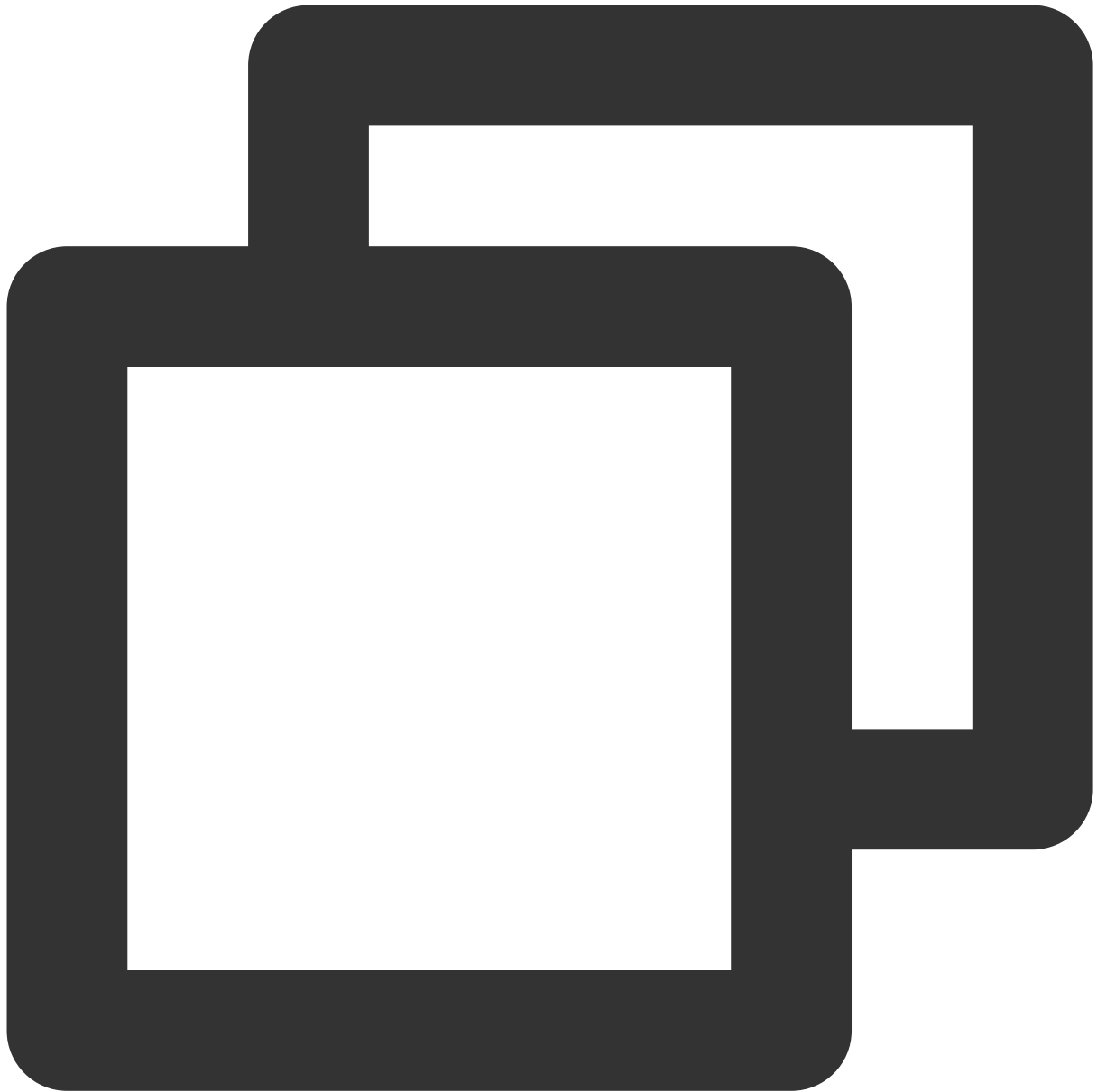
## **getSetCookie**



```
headers.getSetCookie(): Array<string>
```

该方法返回一个数组，包含 [Set-Cookie](#) 头部的所有值。

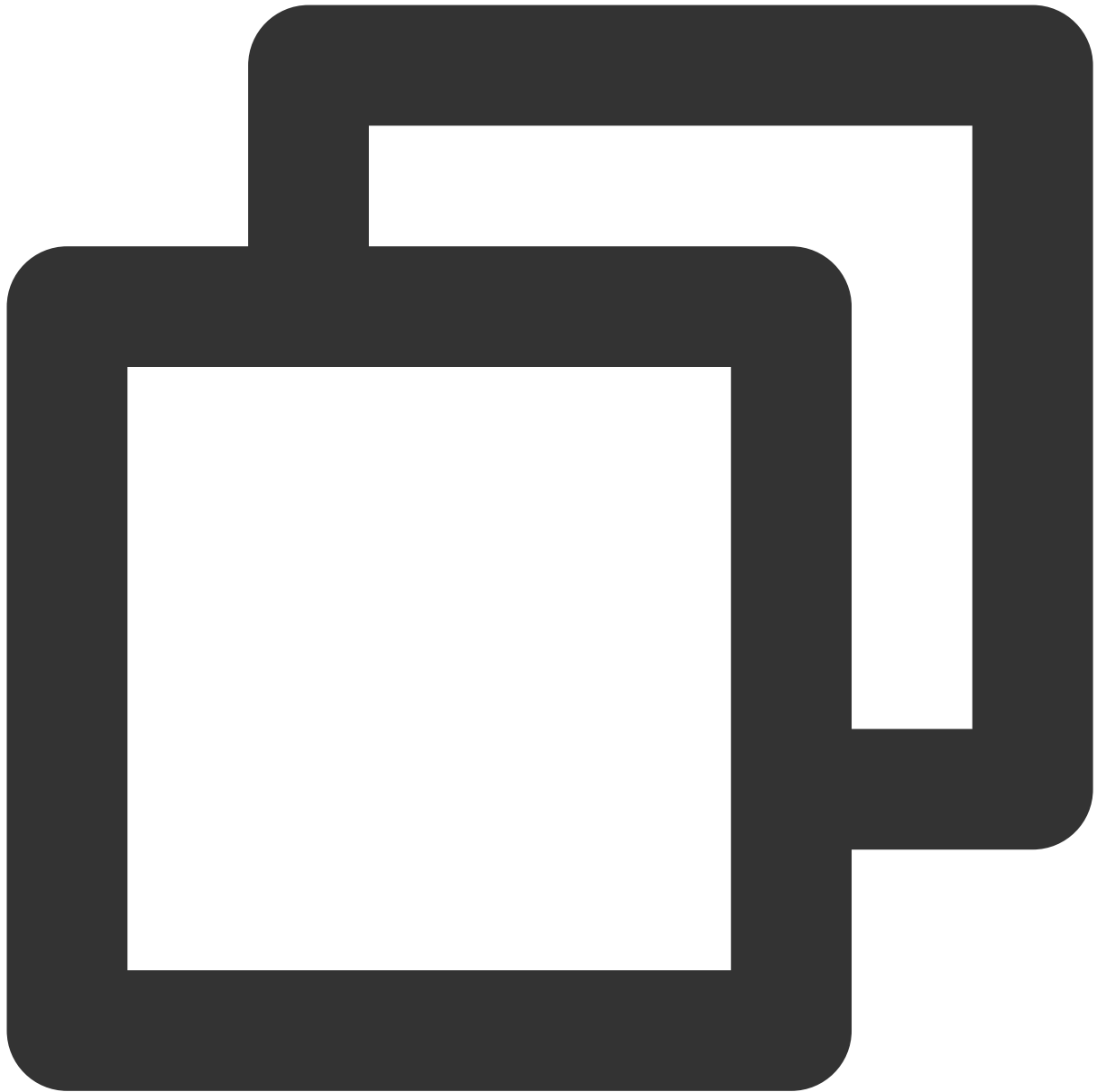
**has**



```
headers.has(name: string): boolean;
```

判断 `headers` 对象是否包含该指定 header。

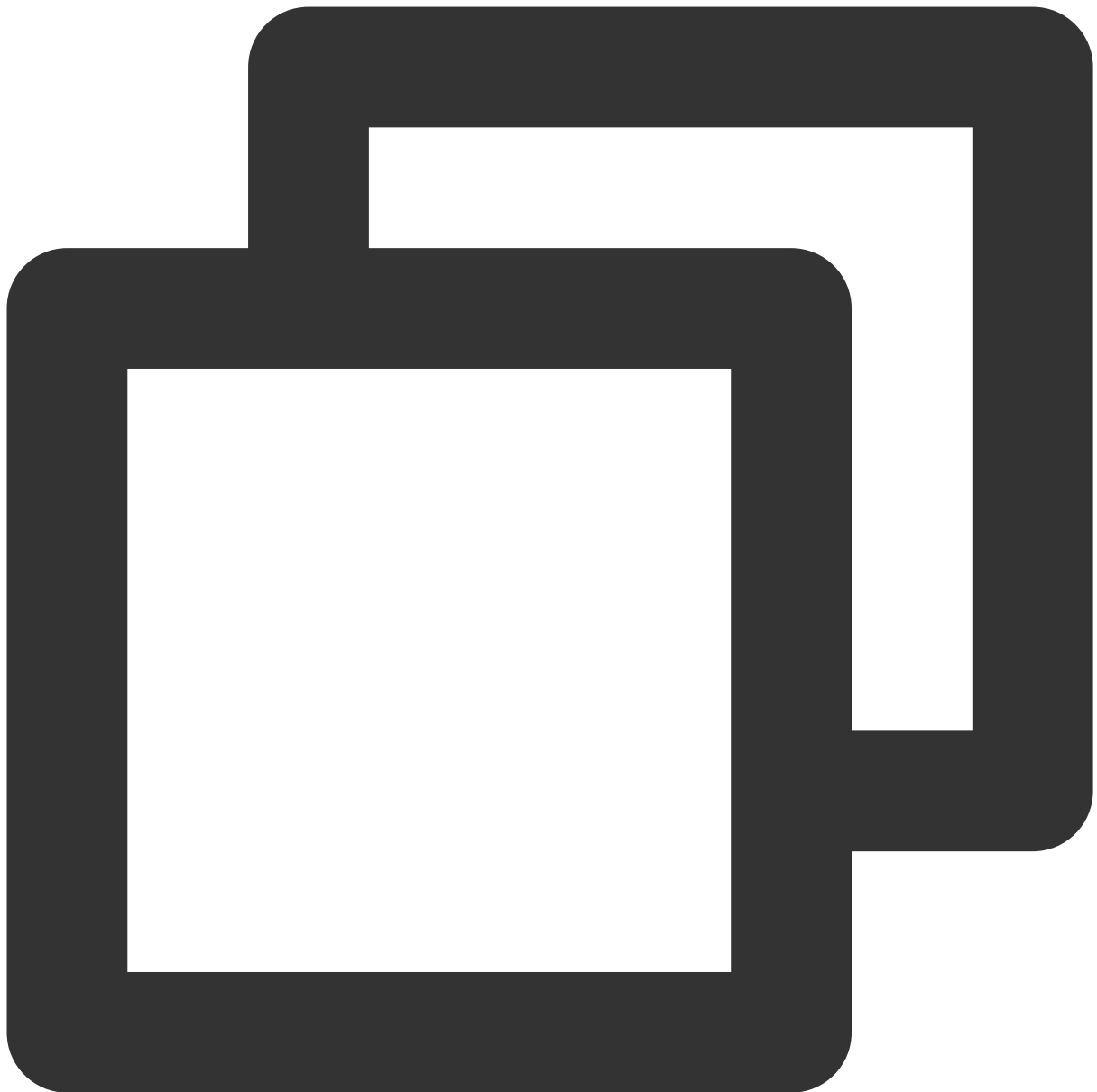
## keys



```
headers.keys(): iterator;
```

获取 `headers` 对象包含的所有 key，返回值参考 [MDN 官方文档：iterator](#)。

**set**



```
headers.set(name: string, value: string): void;
```

设置 `headers` 对象的指定 `header` 值，若该 `header` 不存在，则添加一个新的 `key/value` 键值对。

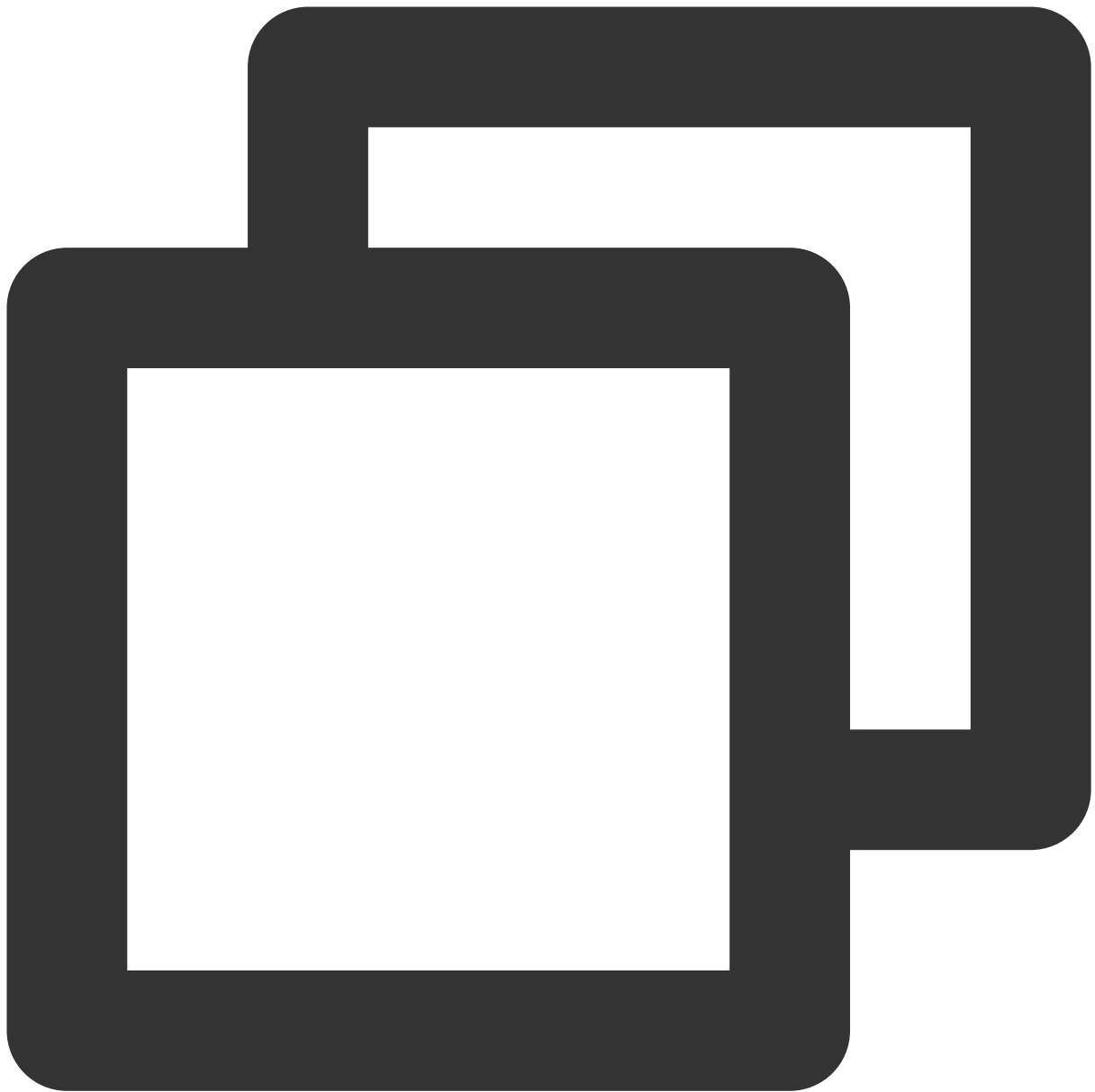
## values



```
headers.values(): iterator;
```

获取 `headers` 对象包含的所有 `value`，返回值参考 [MDN 官方文档：iterator](#)。

## 示例代码



```
function handleEvent() {
  const headers = new Headers({
    'my-header-x': 'hello world',
  });

  const response = new Response('hello world', {
    headers,
  });
  return response;
}
```



```
addEventListener('fetch', (event) => {  
  event.respondWith(handleEvent(event));  
});
```

## 相关参考

[MDN 官方文档：Headers](#)

[示例函数：防篡改校验](#)

[示例函数：请求头鉴权](#)

[示例函数：修改响应头](#)

# Request

最近更新时间：2023-11-24 15:12:04

**Request** 代表 HTTP 请求对象，基于 Web APIs 标准 [Request](#) 进行设计。

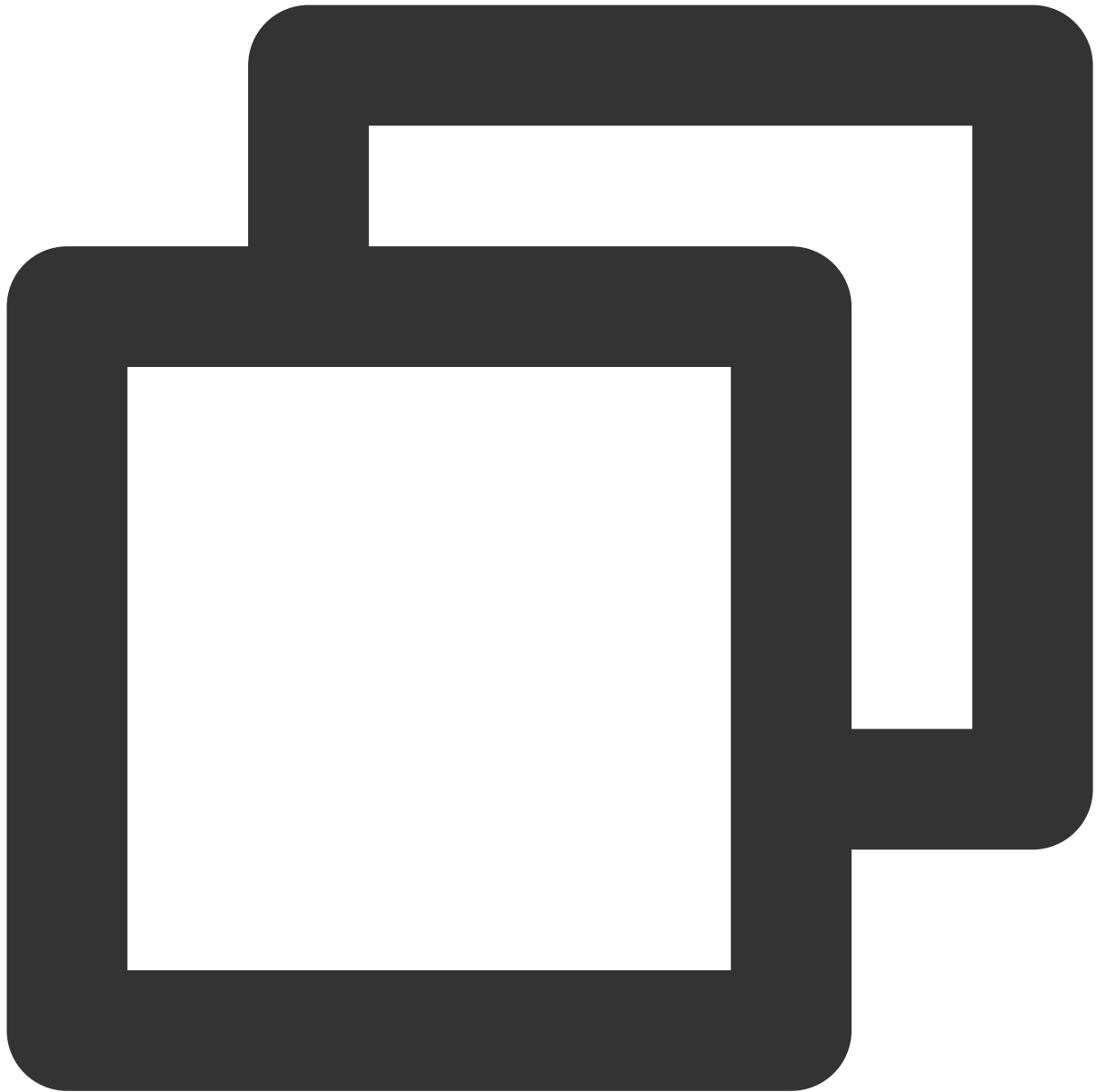
## 说明

边缘函数中，可通过两种方式获得 `Request` 对象：

使用 `Request` 构造函数创建一个 `Request` 对象，用于 Fetch API 的操作。

使用 `FetchEvent` 对象 `event.request`，获得当前请求的 `Request` 对象。

## 构造函数



```
const request = new Request(input: string | Request, init?: RequestInit)
```

## 参数

参数名称	类型	必填	说明
input	string   <a href="#">Request</a>	是	URL 字符串或 <a href="#">Request</a> 对象。
options	<a href="#">RequestInit</a>	否	<a href="#">Request</a> 对象初始化配置项。

## RequestInit

初始化 Request 对象的属性值选项。

属性名	类型	必填	默认值	说明
method	string	否	GET	请求方法 ( GET 、 POST 等)。
headers	<a href="#">Headers</a>	否	-	请求头部信息。
body	string   <a href="#">Blob</a>   <a href="#">ArrayBuffer</a>   <a href="#">ArrayBufferView</a>   <a href="#">ReadableStream</a>	否	-	请求体。
redirect	string	否	follow	重定向策略，支持 <code>manual</code> 、 <code>error</code> 和 <code>follow</code> 。
maxFollow	number	否	12	最大可重定向次数。
version	string	否	HTTP/1.1	HTTP 版本，支持 <code>HTTP/1.0</code> 、 <code>HTTP/1.1</code> 和 <code>HTTP/2.0</code> 。
copyHeaders	boolean	否	-	<b>非 Web APIs 标准选项</b> ，表示是否拷贝传入的 Request 对象的 headers。
eo	<a href="#">RequestInitEoProperties</a>	否	-	<b>非 Web APIs 标准选项</b> ，用于控制边缘函数处理该请求的行为。

## RequestInitEoProperties

非 Web APIs 标准选项，用于控制边缘函数处理该请求的行为。

参数名称	类型	必填	说明
resolveOverride	string	否	用于 <code>fetch</code> 请求下覆盖原有的域名解析，支持指定域名或者 IP 地址。更多说明如下： IP 不允许带 <code>scheme</code> 以及端口号。 IPv6 无需使用方括号包裹。
image	<a href="#">ImageProperties</a>	否	图片处理参数配置项。
cacheEverything	boolean	否	源站响应头是否缓存。
cacheKey	string	否	用于指定自定义的缓存键。
cacheTtl	number	否	用于指定缓存时长（单位秒），小于等于0表示不缓存。
cacheTtlByStatus	{[key: string]:	否	根据状态码指定缓存时长（单位秒），小于等于0表示不缓

	number}		存。 示例：{'200-299': 3600, '404': 10, '500-599': 0}
--	---------	--	---

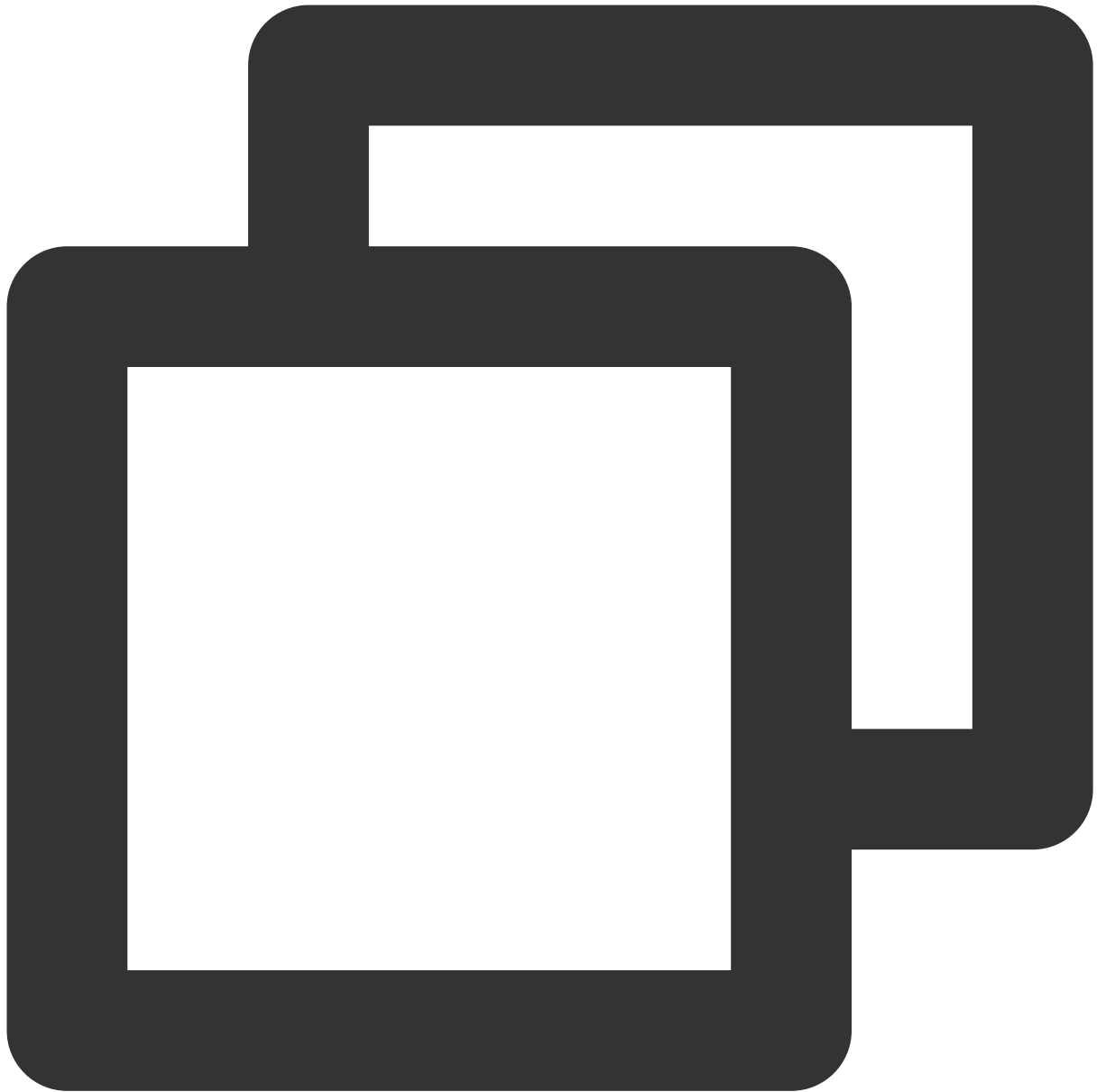
## ImageProperties

`fetch` 支持图片处理功能，参数配置项说明如下。该图片处理参数配置项与站点加速图片处理参数能力相同，您可以参考 [图片处理](#) 了解站点加速内的图片处理能力。

参数名称	类型	必填	说明
format	string	否	将原图转换为指定格式，支持 <code>jpg</code> 、 <code>gif</code> 、 <code>png</code> 、 <code>bmp</code> 、 <code>webp</code> 、 <code>avif</code> 、 <code>jp2</code> 、 <code>jxr</code> 、 <code>heif</code>
long	number	否	指定长边，短边未指定时，短边自适应。
short	number	否	指定短边，长边未指定时，长边自适应。
width	number	否	指定宽度，高度未指定时，高度自适应。
height	number	否	指定高度，宽度未指定时，宽度自适应。

## 实例属性

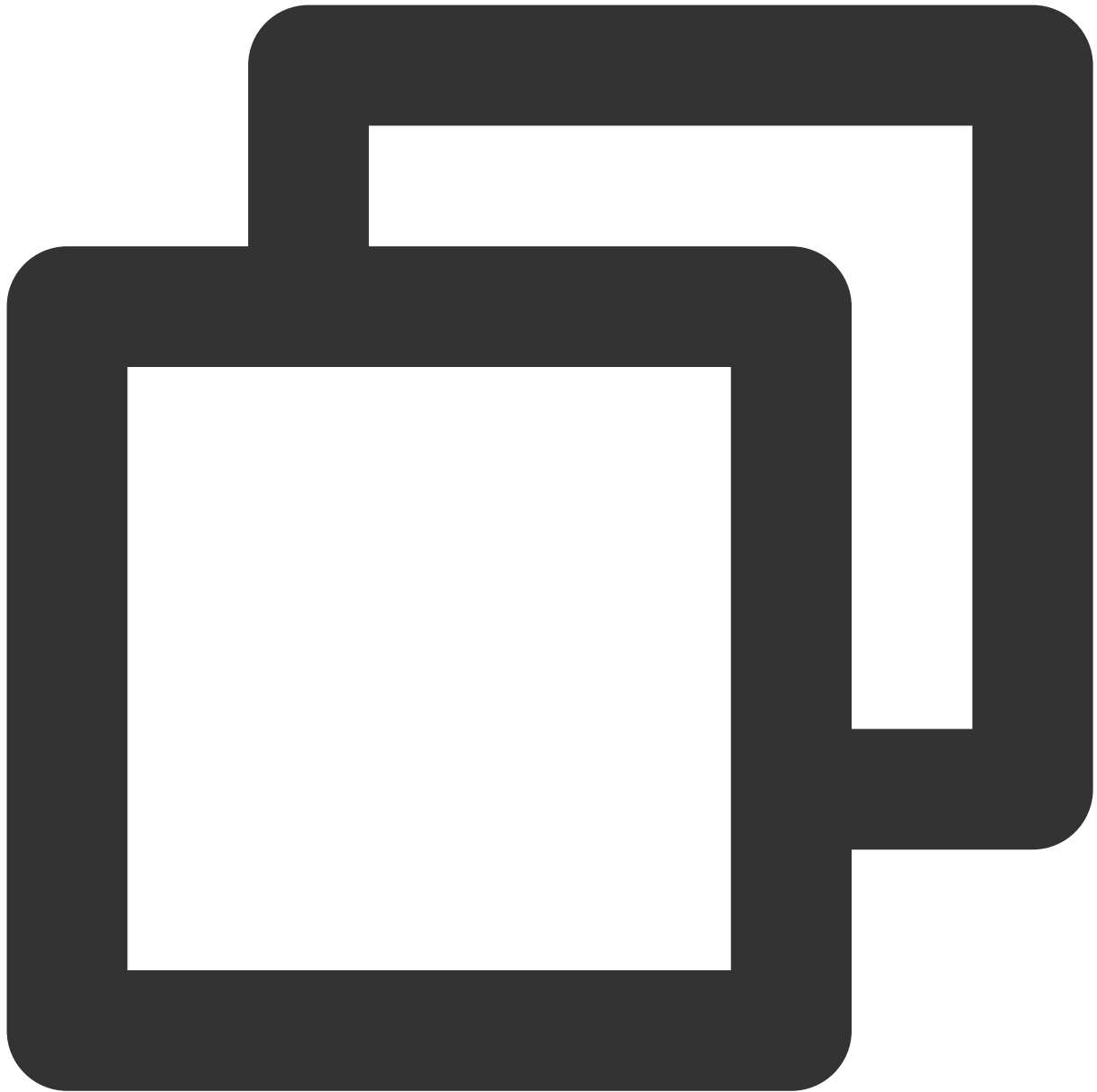
### body



```
// request.body  
readonly body: ReadableStream;
```

请求体，详情参见 [ReadableStream](#)。

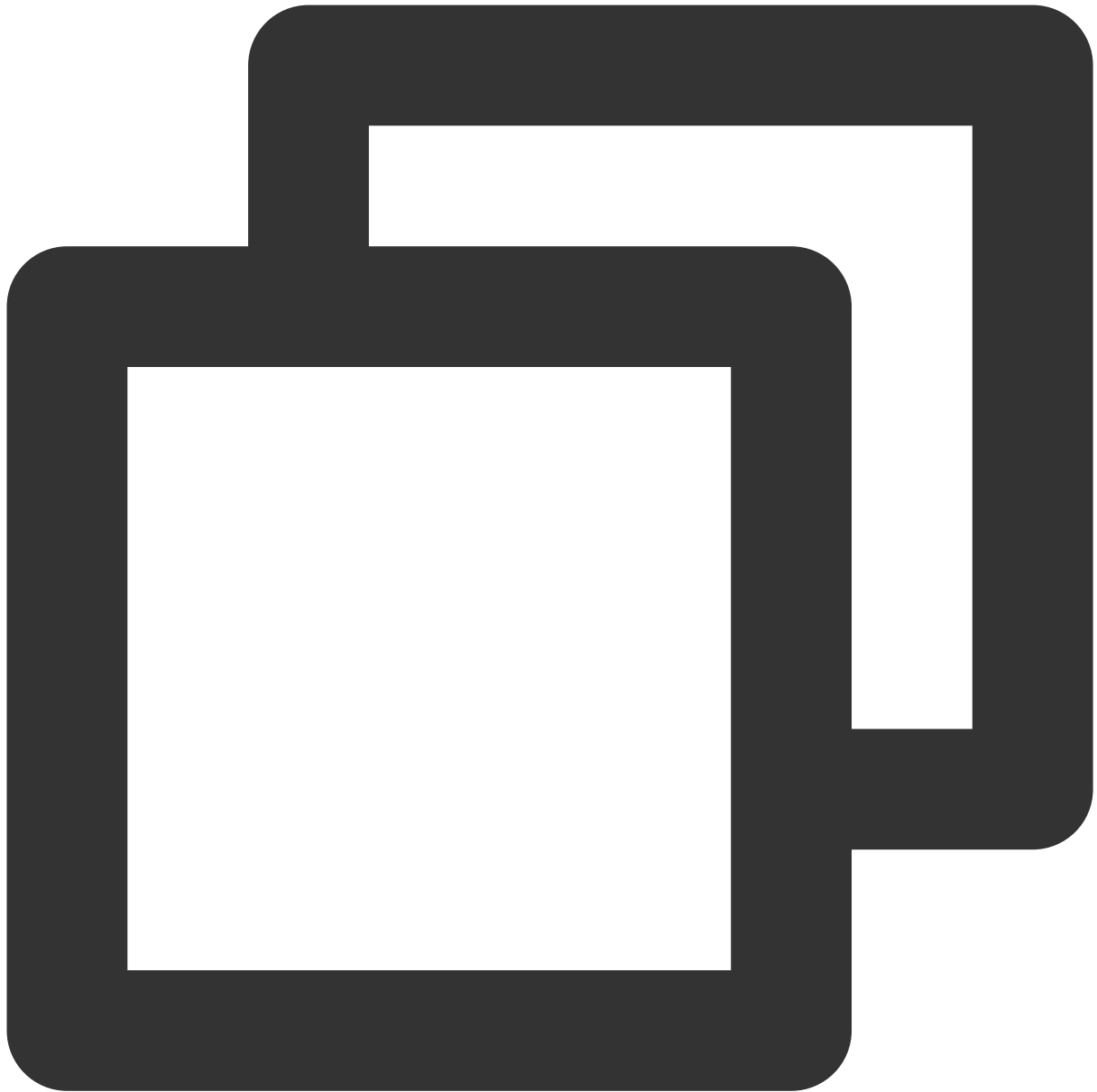
## bodyUsed



```
// request.bodyUsed  
readonly bodyUsed: boolean;
```

标识请求体是否已读取。

## headers

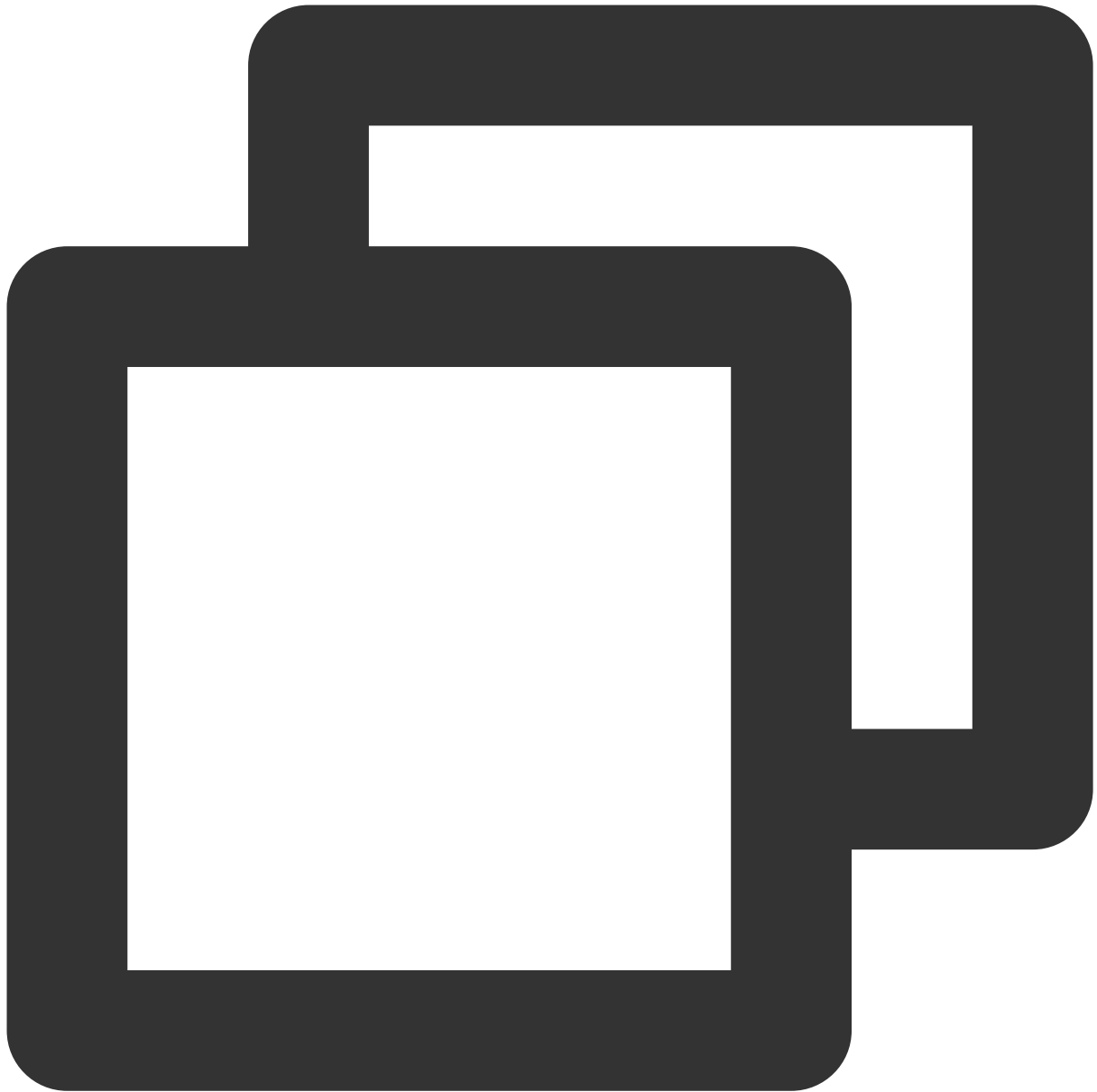


```
// request.headers  
readonly headers: Headers;
```

请求头部，详情参见 [Headers](#)。

## method

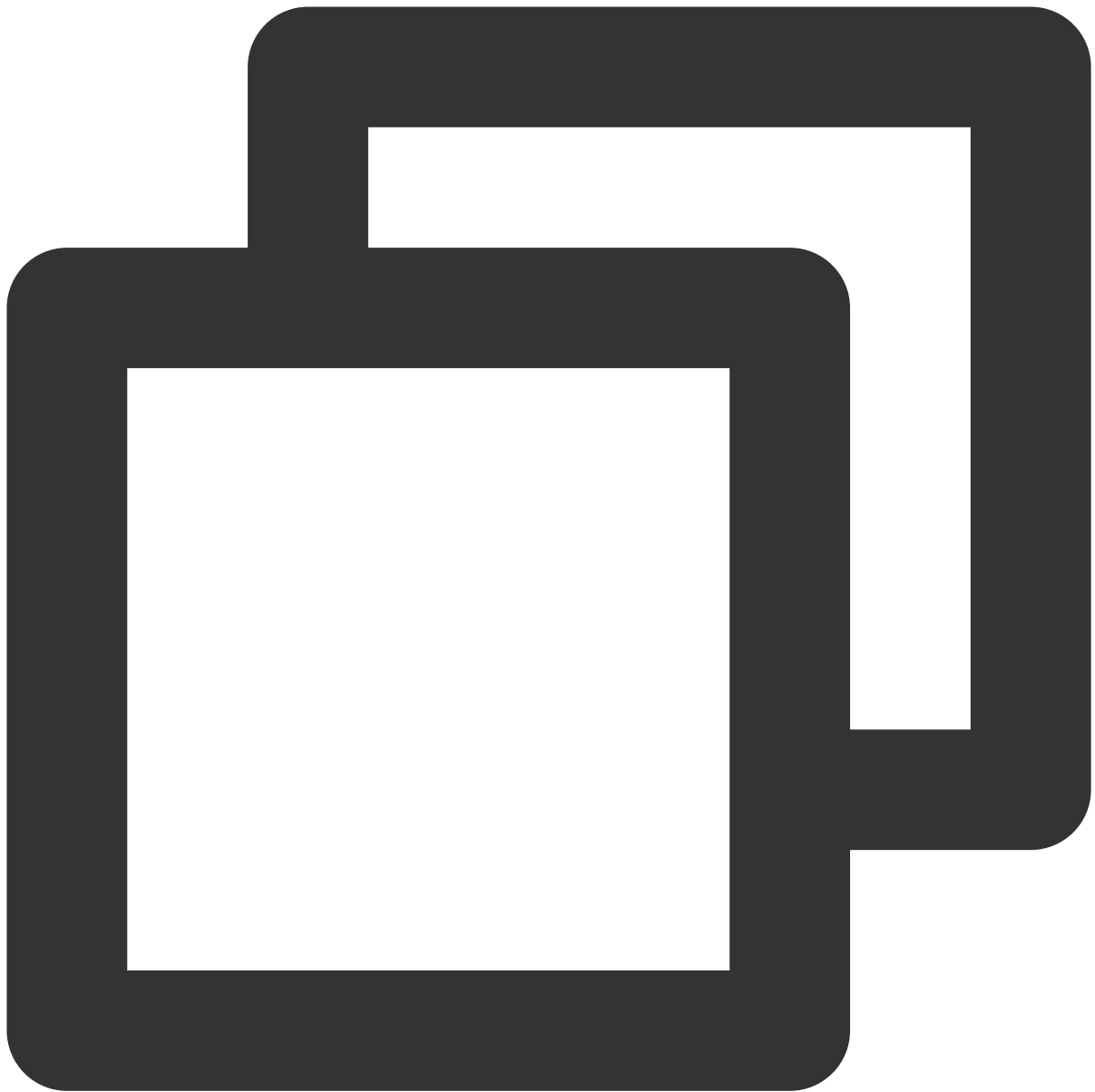




```
// request.method  
readonly method: string;
```

请求方法，默认值为 `GET`。

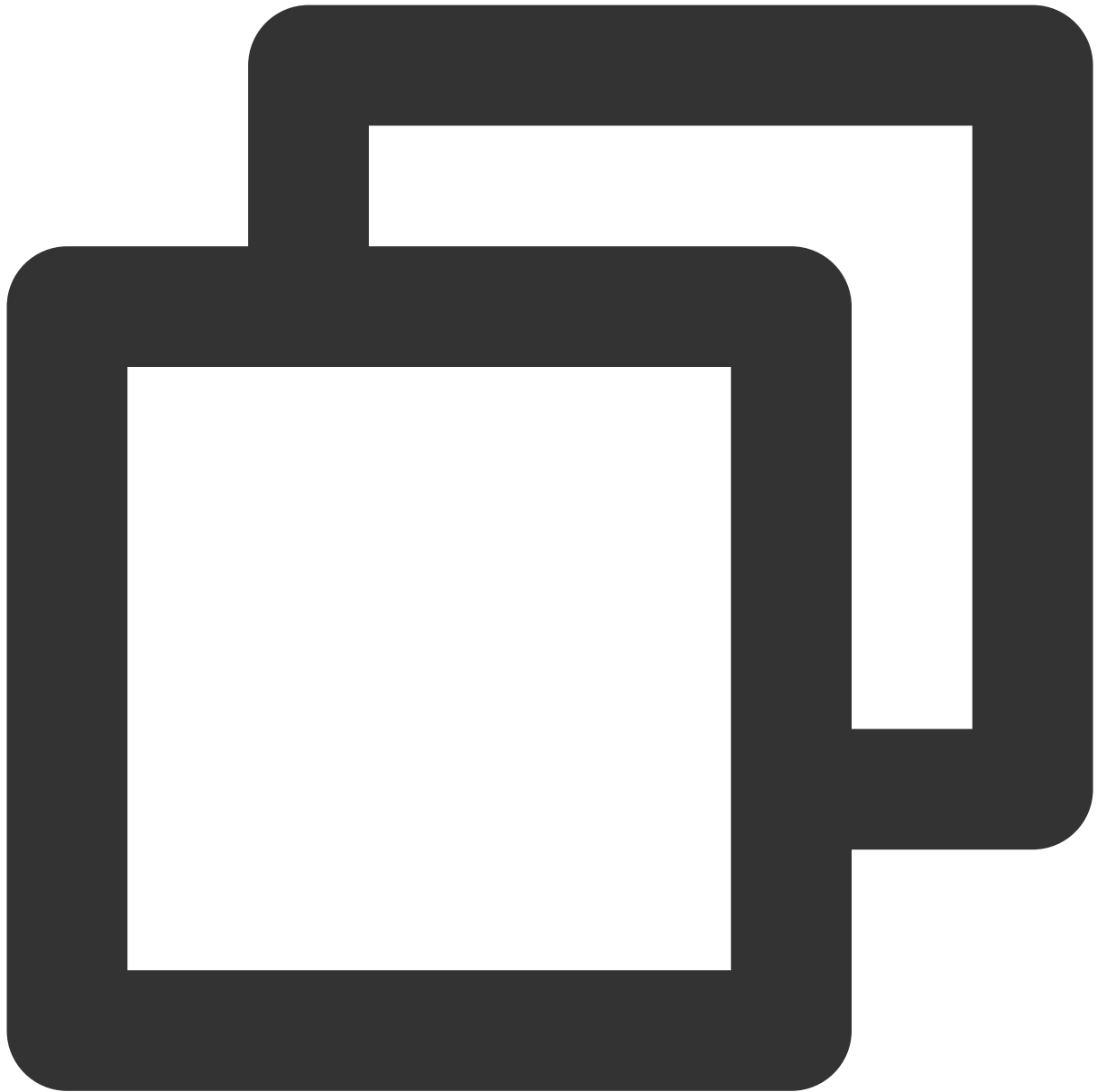
## **redirect**



```
// request.redirect  
readonly redirect: string;
```

请求重定向策略，可取值有：`follow`、`error`、`manual`，默认为 `manual`。

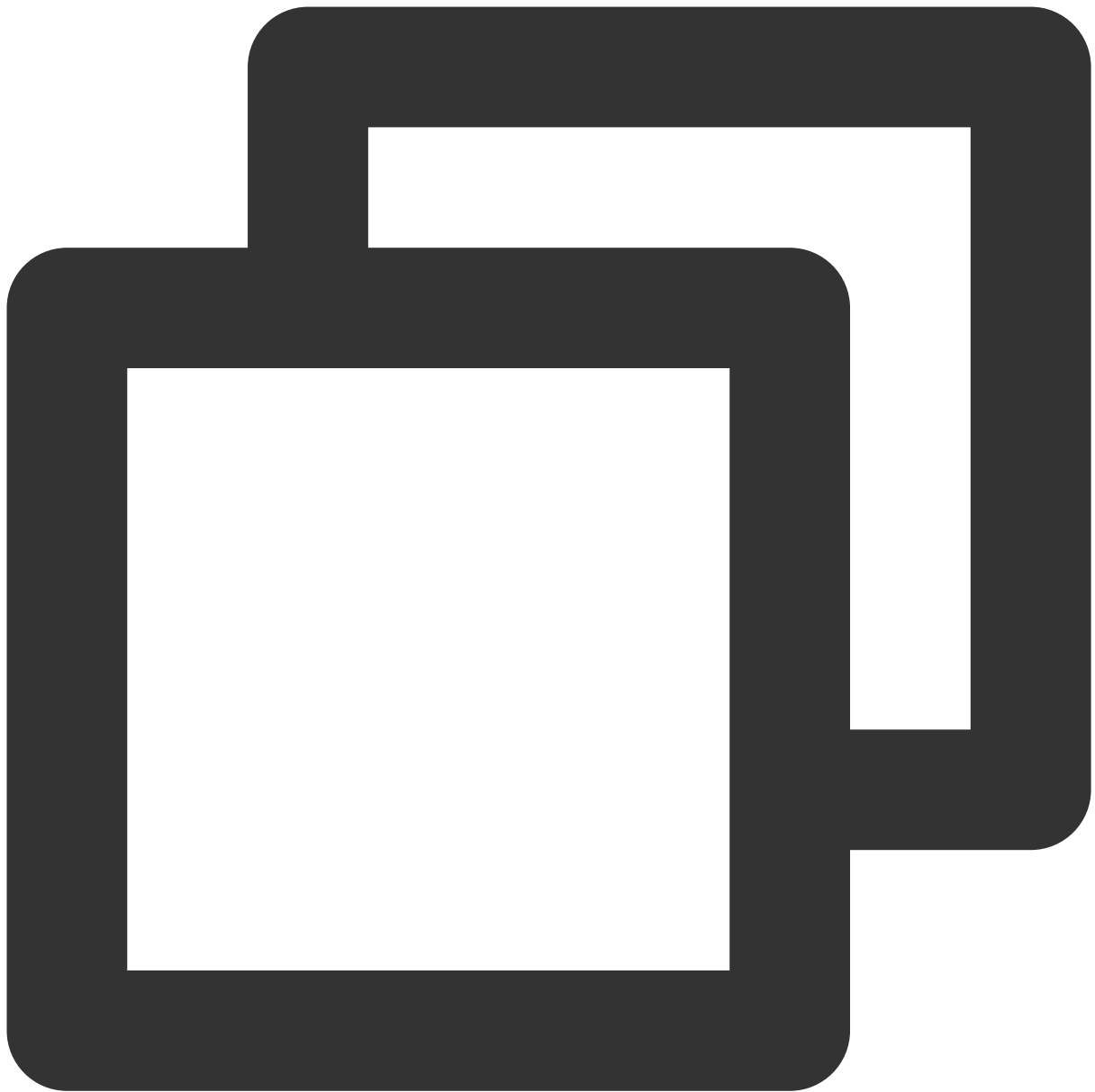
### **maxFollow**



```
// request.maxFollow  
readonly maxFollow: number;
```

请求最大重定向次数。

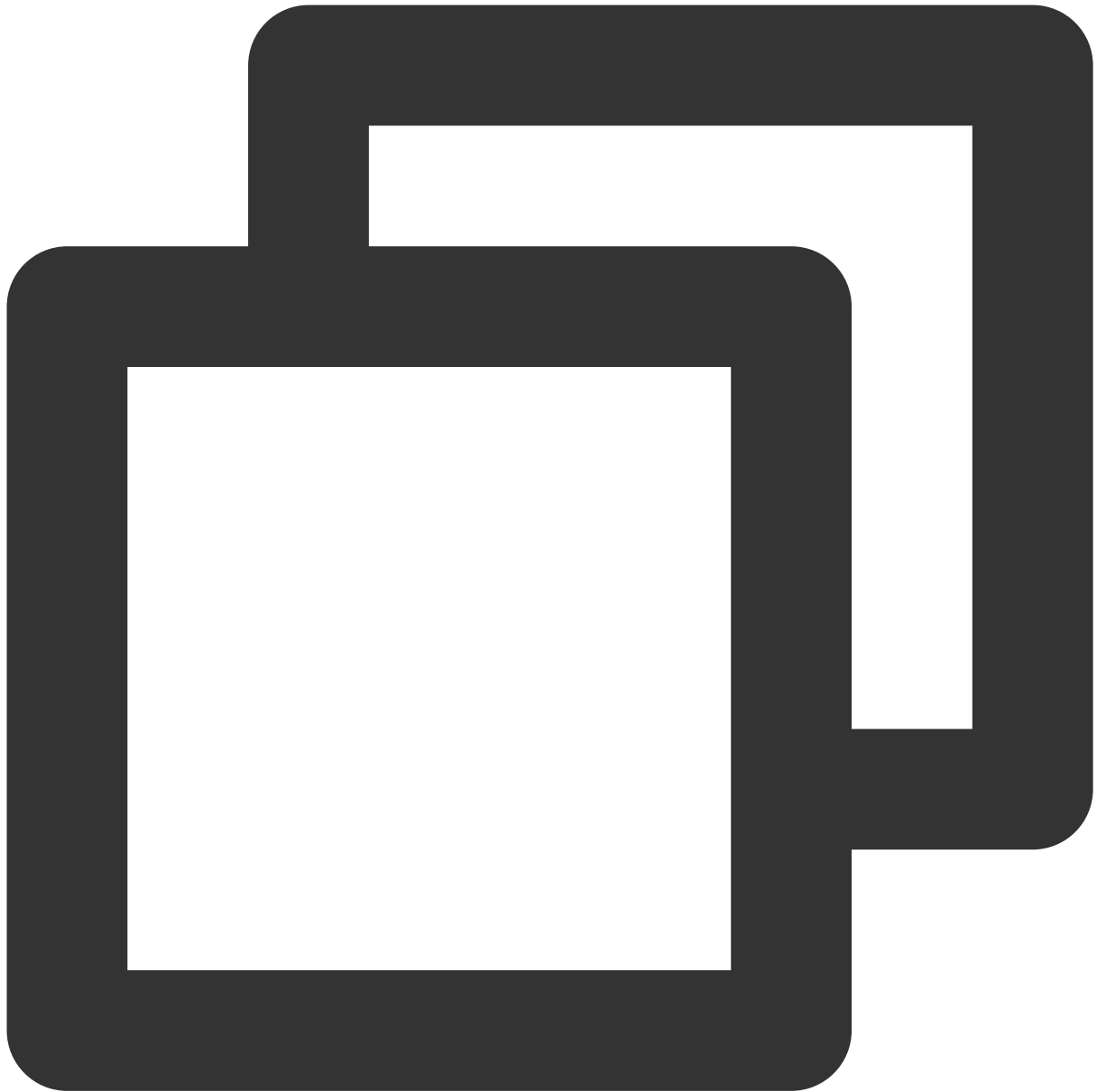
**url**



```
// request.url  
readonly url: string;
```

请求 url。

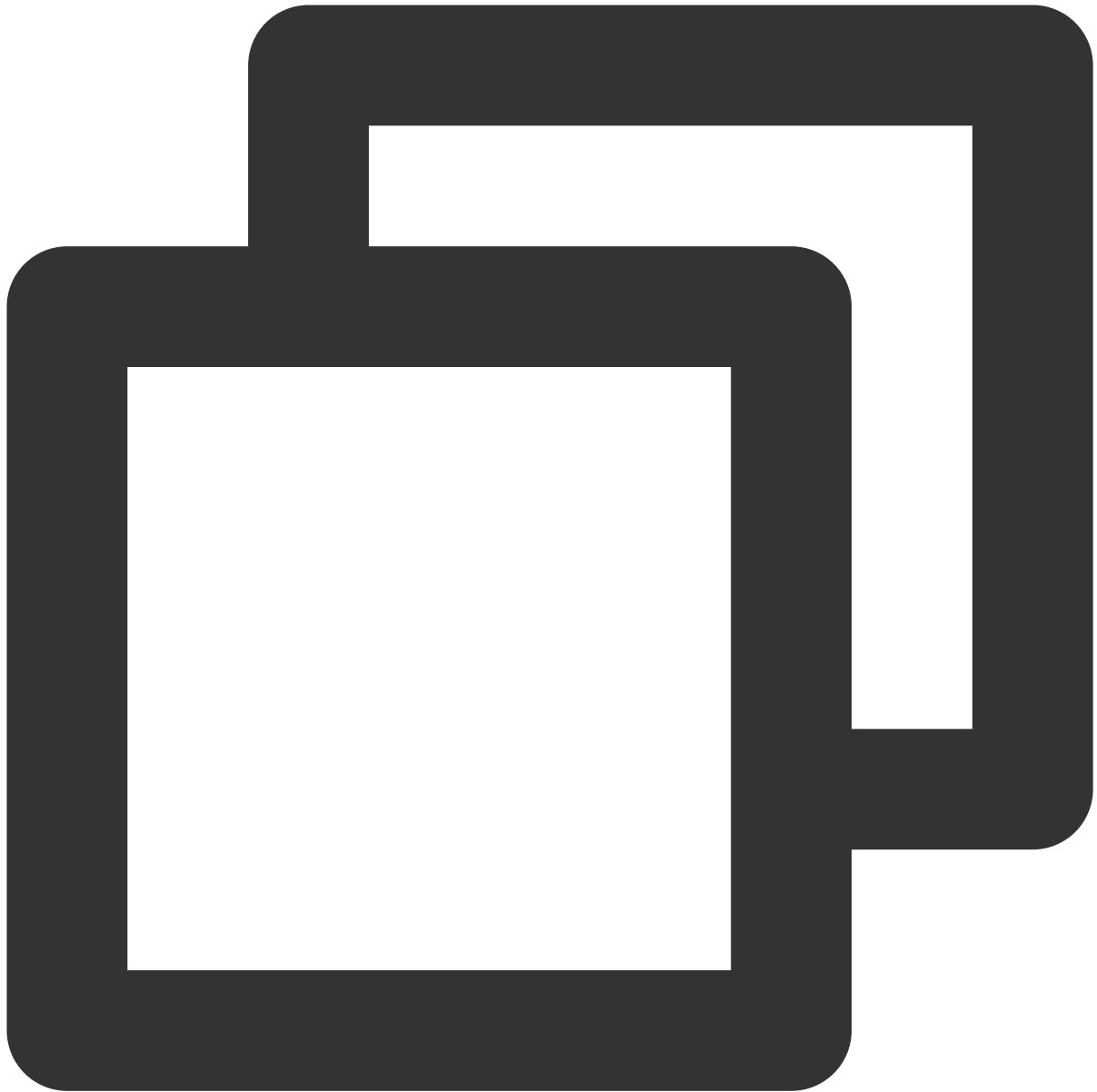
**version**



```
// request.version  
readonly version: string;
```

请求使用的 HTTP 协议版本。

**eo**



```
// request.version  
readonly eo: IncomingRequestEoProperties;
```

边缘函数提供的与当前请求相关的一些其他信息，详情参见 [IncomingRequestEoProperties](#)。

### IncomingRequestEoProperties

客户端请求 `event.request` 对象包含 `eo` 属性，其信息如下：

属性名	类型	说明	示例值
geo	<a href="#">GeoProperties</a>	描述客户请求的位置信息。	-

## GeoProperties

描述客户请求的位置信息。

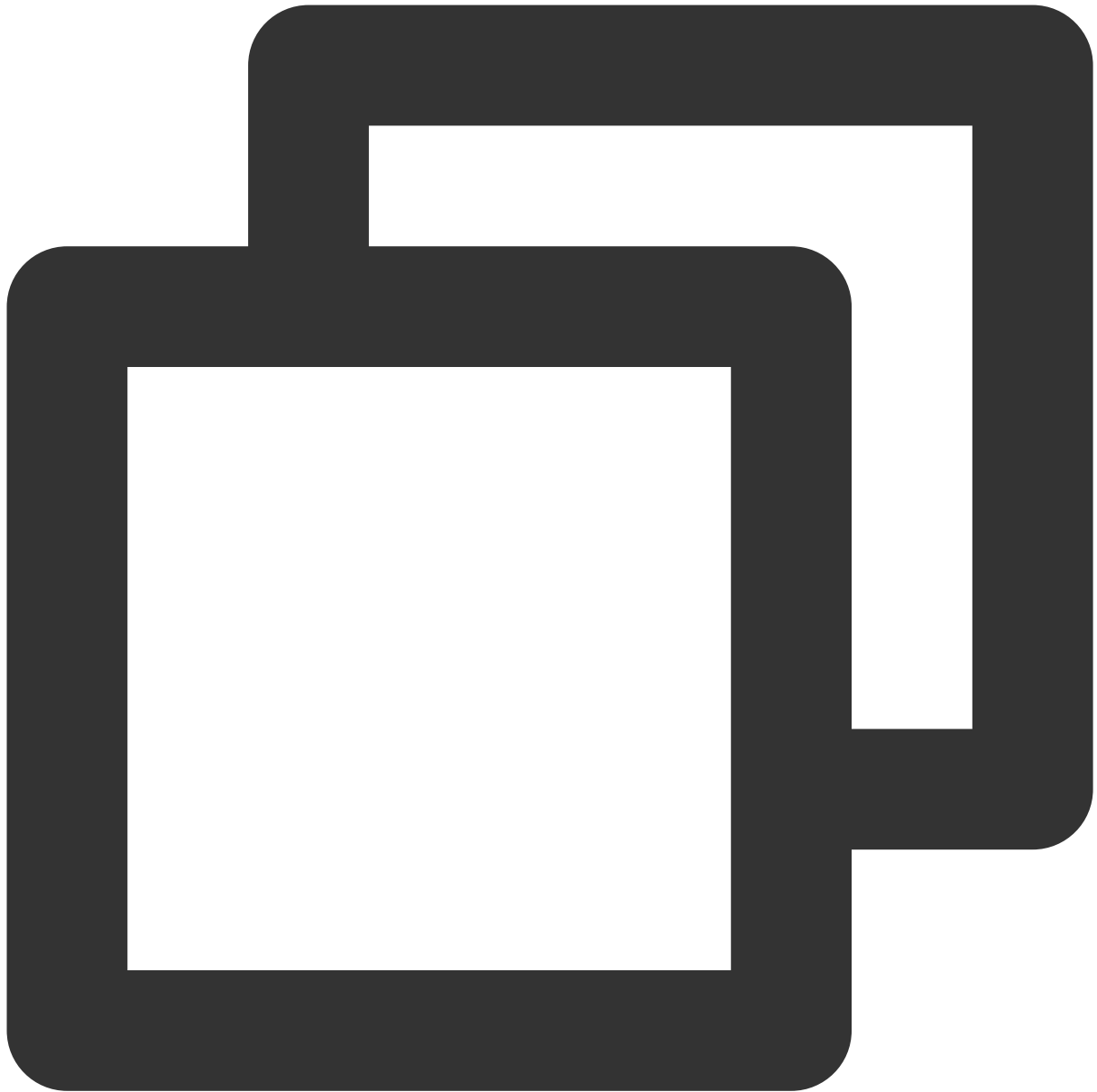
属性名	类型	说明	示例值
asn	number	<a href="#">ASN</a>	132203
countryName	string	国家名	Singapore
countryCodeAlpha2	string	国家的 <a href="#">ISO-3611 alpha2</a> 代码	SG
countryCodeAlpha3	string	国家的 <a href="#">ISO-3611 alpha3</a> 代码	SGP
countryCodeNumeric	string	国家的 <a href="#">ISO-3611 numeric</a> 代码	702
regionName	string	区域名	-
regionCode	string	区域代码	AA-AA
cityName	string	城市名	singapore
latitude	number	纬度	1.29027
longitude	number	经度	103.851959

## 实例方法

### 注意

获取请求体方法，接收 `HTTP body` 最大字节数为 1M，超出大小会抛出 `OverSize` 异常。超出大小时推荐使用 `request.body` 流式读取，详情参见 [ReadableStream](#)。

### arrayBuffer

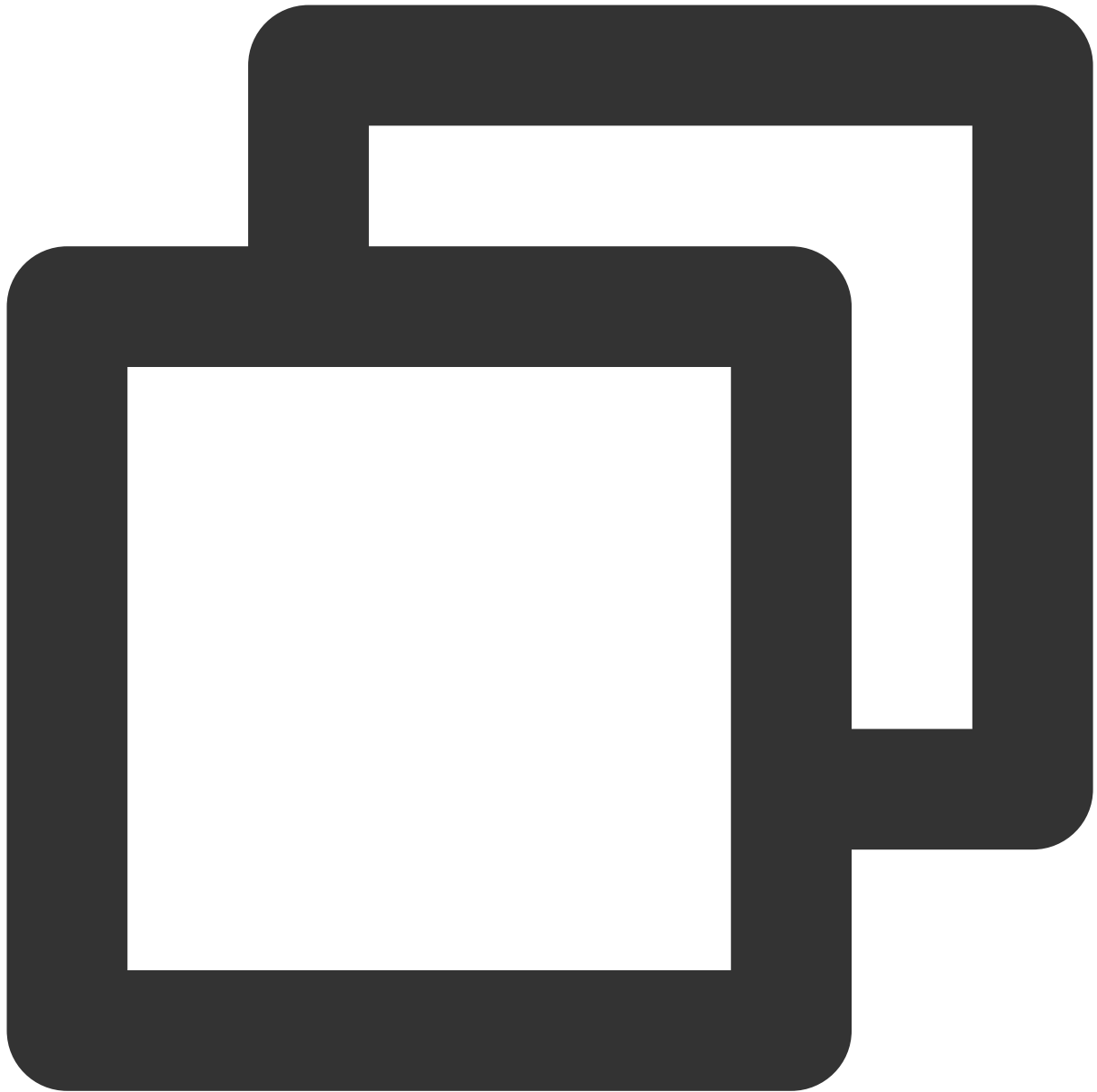


```
request.arrayBuffer(): Promise<ArrayBuffer>;
```

获取请求体，解析结果为 [ArrayBuffer](#)。

## blob

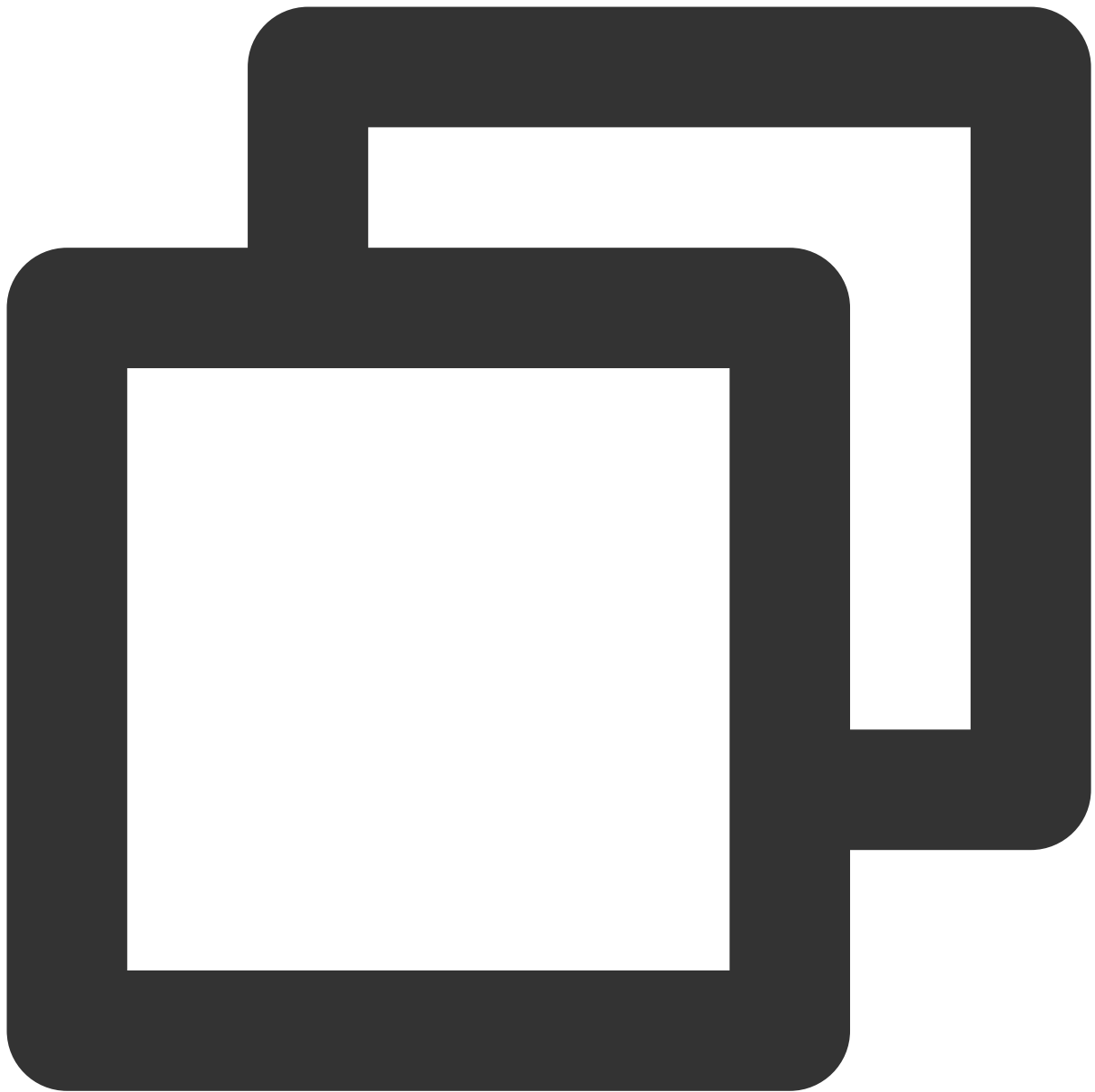




```
request.blob(): Promise<Blob>;
```

获取请求体，解析结果为 [Blob](#)。

## clone



```
request.clone(copyHeaders?: boolean): Request;
```

创建请求对象的副本。

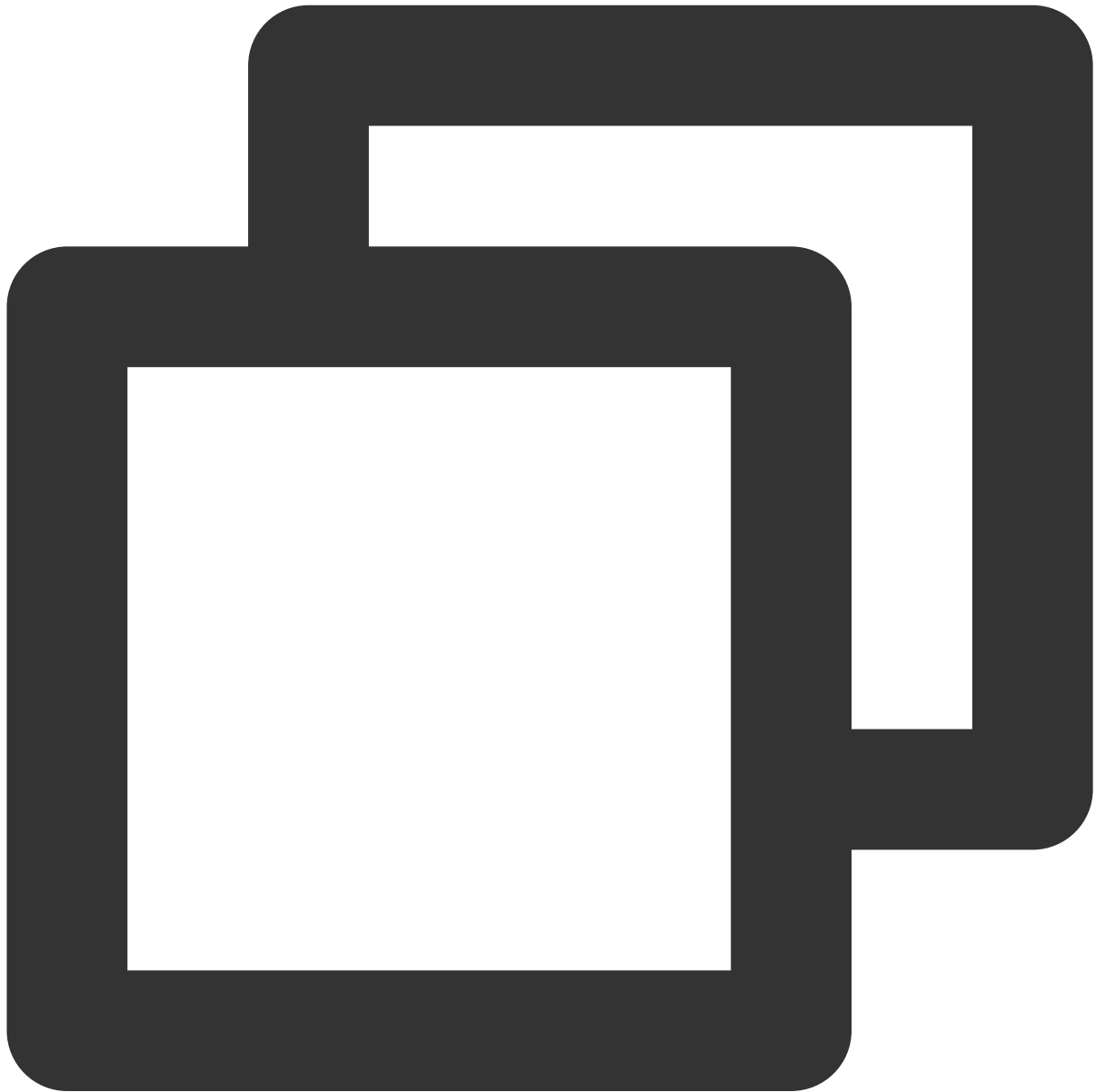
### 参数

参数名称	类型	必填	说明
copyHeaders	boolean	否	开启复制请求头，默认值为 <code>false</code> ，取值说明如下。 <code>true</code> 复制原对象的请求头。

false

引用原对象的请求头。

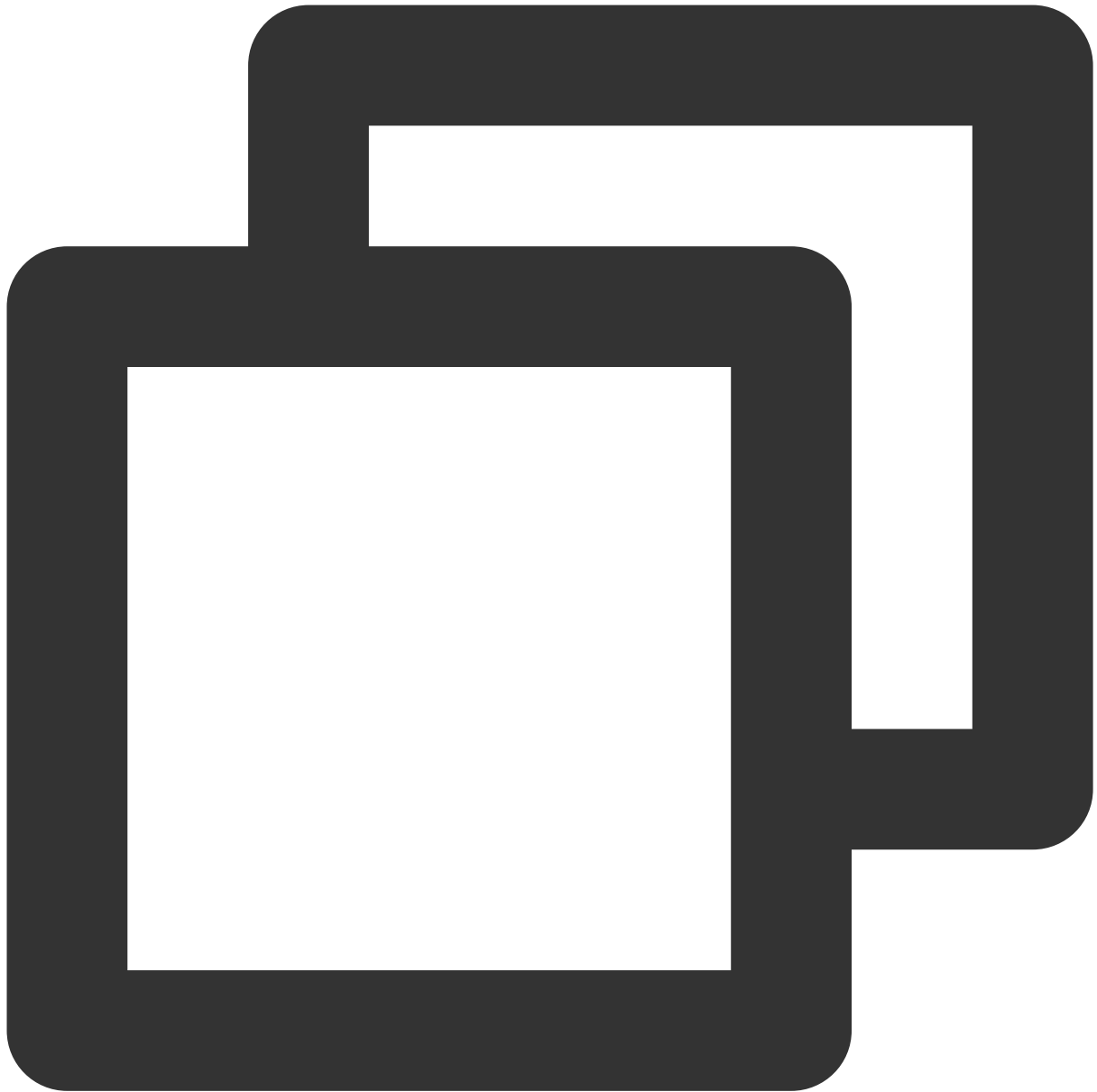
**json**



```
request.json(): Promise<object>;
```

获取请求体，解析结果为 `json`。

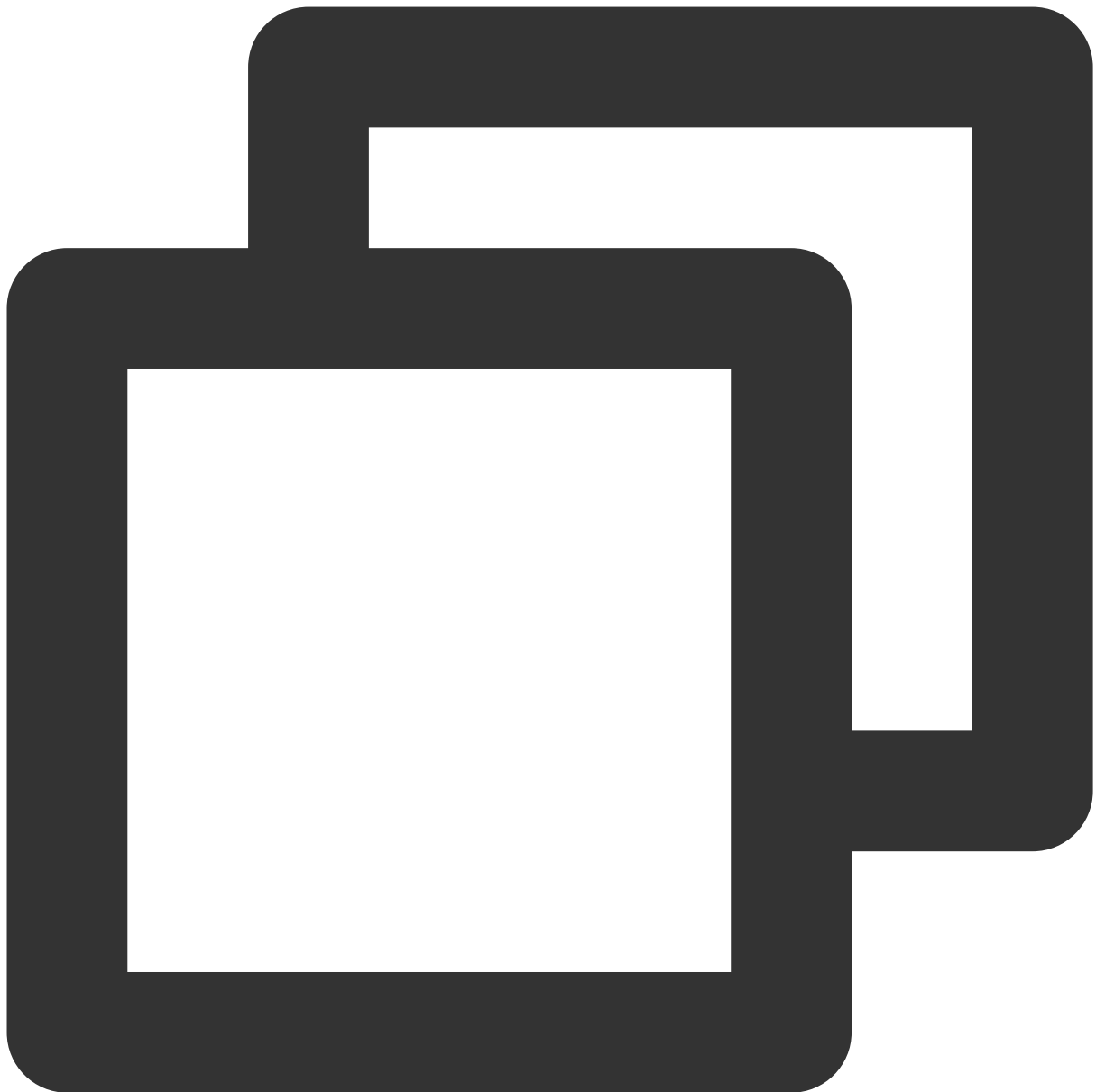
**text**



```
request.text(): Promise<string>;
```

获取请求体，解析结果为文本字符串。

## formData



```
request.formData(): Promise<FormData>;
```

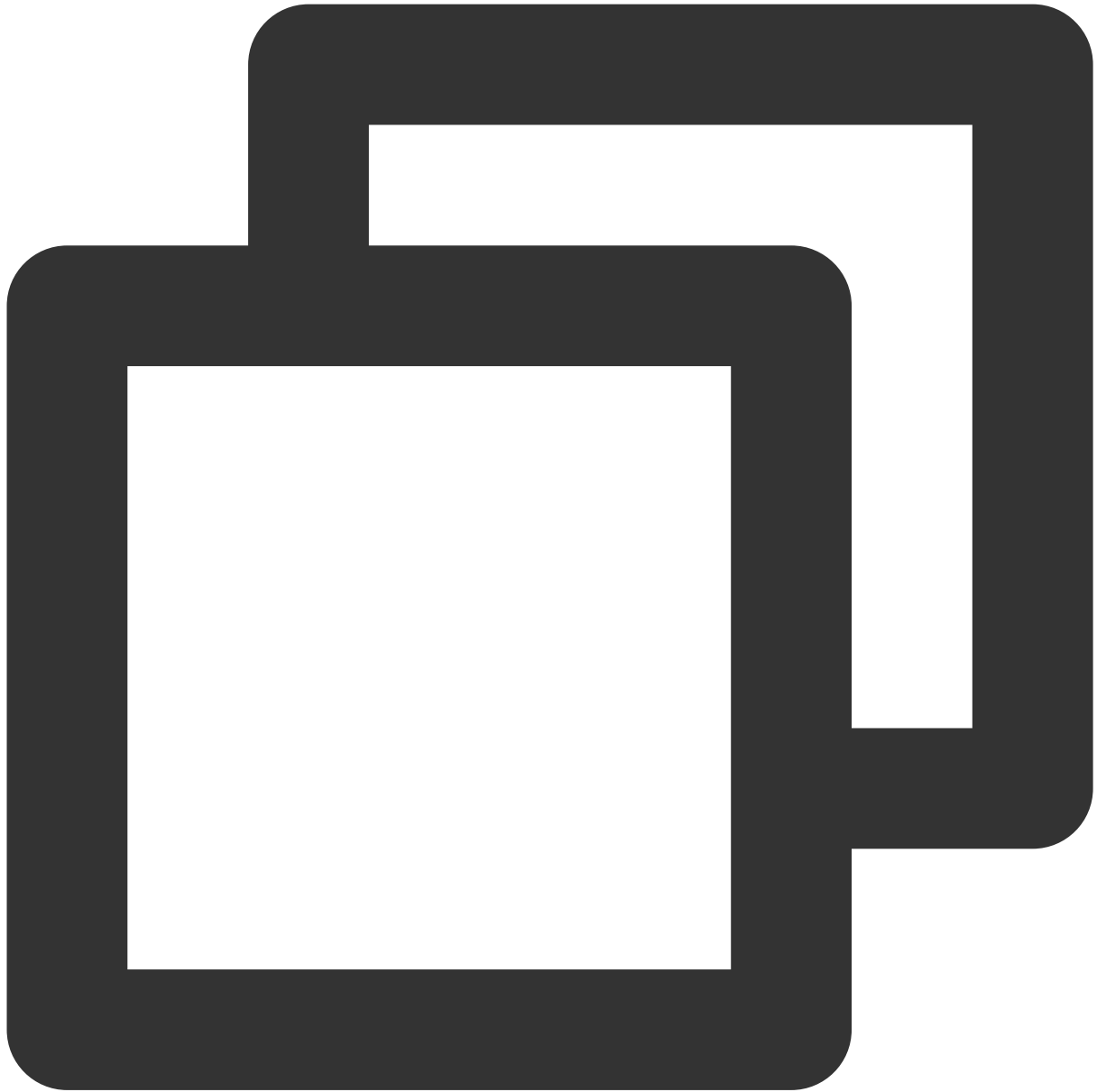
获取响应体，解析结果为 [FormData](#)。

kie 值。

#### 参数

参数名称	类型	必填	说明
cookies	<a href="#">Cookies</a>	否	新的 Cookies 对象。

## 示例代码



```
async function handleRequest() {
  const request = new Request('https://www.tencentcloud.com/');
  const response = await fetch(request);
  return response;
}

addEventListener('fetch', (event) => {
  event.respondWith(handleRequest());
});
```

```
});
```

## 相关参考

[MDN 官方文档：Request](#)

[示例函数：Cache API 使用](#)

[示例函数：基于请求区域重定向](#)

# Response

最近更新时间：2023-09-11 17:53:29

**Response** 代表 HTTP 响应，基于 Web APIs 标准 [Response](#) 进行设计。

## 说明

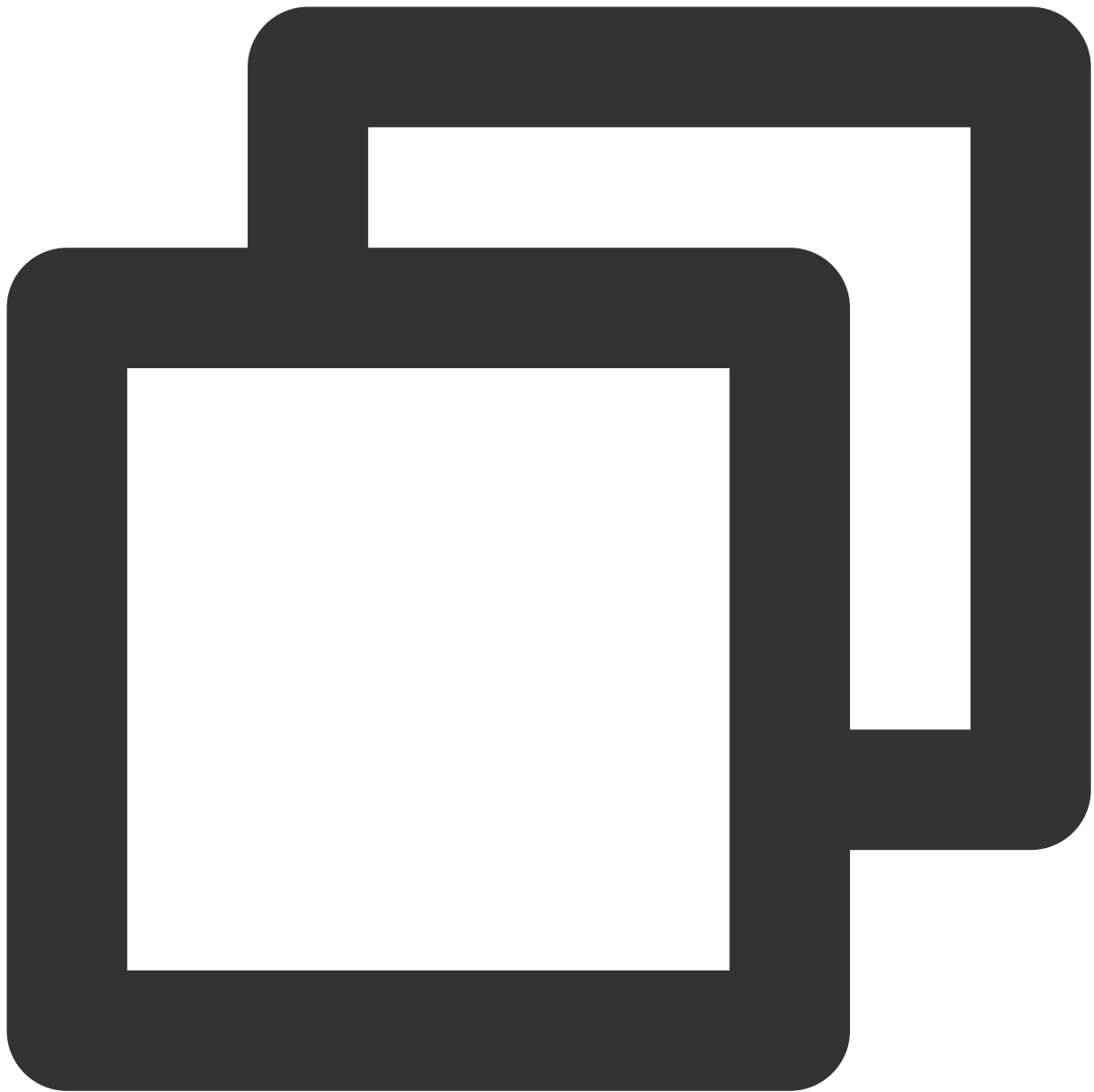
边缘函数中，可通过两种方式获得 `Response` 对象：

使用 `Response` 构造函数创建一个 `Response` 对象，用于 `event.respondWith` 响应。

使用 `fetch` 获取请求响应 `Response` 对象。

## 构造函数





```
const response = new Response(body?: string | ArrayBuffer | Blob | ReadableStream |
```

## 参数

参数名称	类型	必填	说明
body	string   <a href="#">ArrayBuffer</a>   <a href="#">Blob</a>   <a href="#">ReadableStream</a>   null   undefined	是	<code>Response</code> 对象的 body 内容。

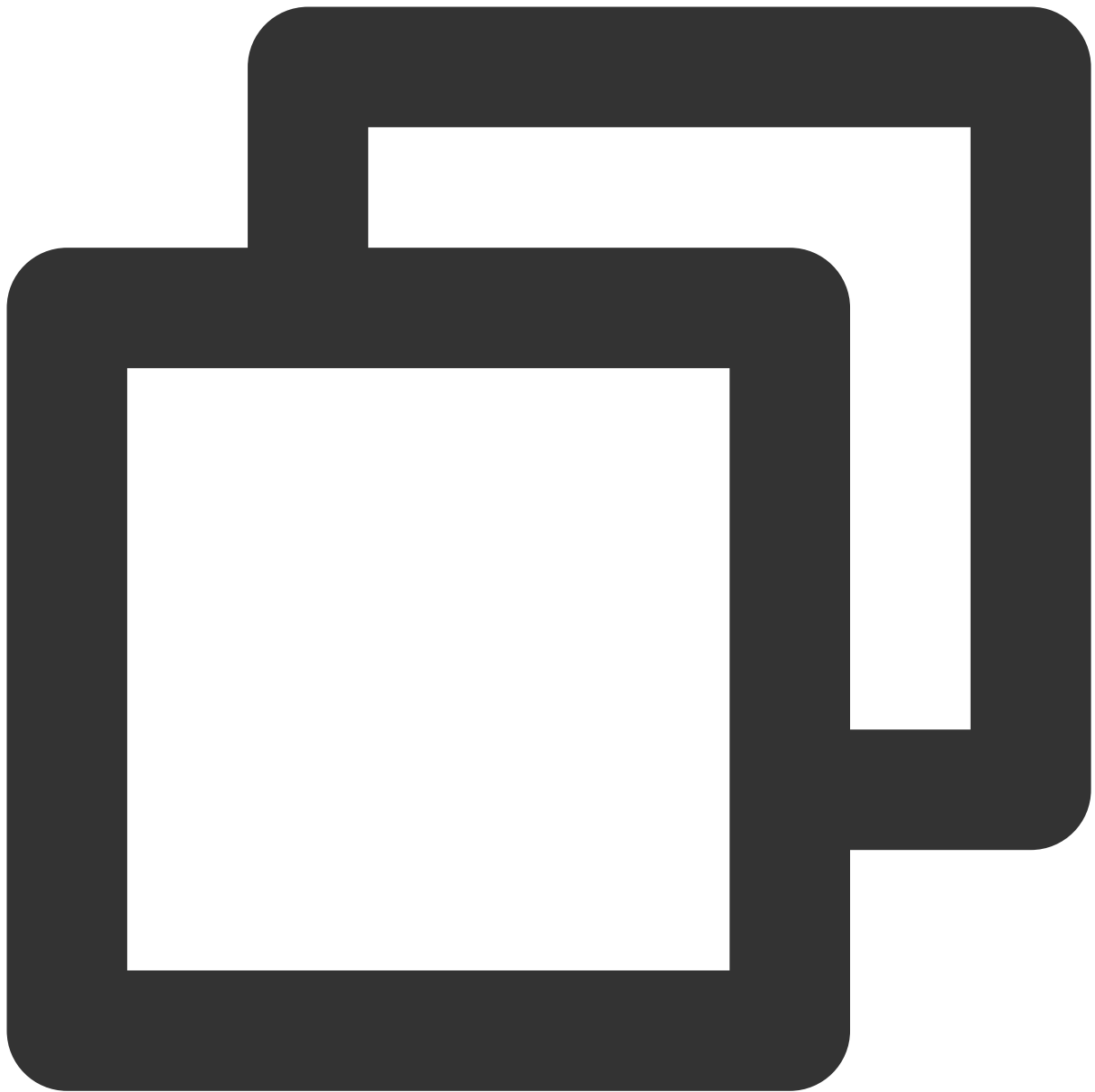
init	<a href="#">ResponseInit</a>	否	<code>Response</code> 对象的初始化配置项。
------	------------------------------	---	----------------------------------

### ResponseInit

参数名称	类型	必填	说明
status	number	否	响应的状态码。
statusText	string	否	响应的状态消息，最大长度为 4095，超出长度会被截断。
headers	<a href="#">Headers</a>	否	响应的头部信息。

## 实例属性

### body



```
// response.body  
readonly body: ReadableStream;
```

响应体，详情参见 [ReadableStream](#)。

## **bodyUsed**



```
// response.bodyUsed  
readonly bodyUsed: boolean;
```

标识响应体是否已读取。

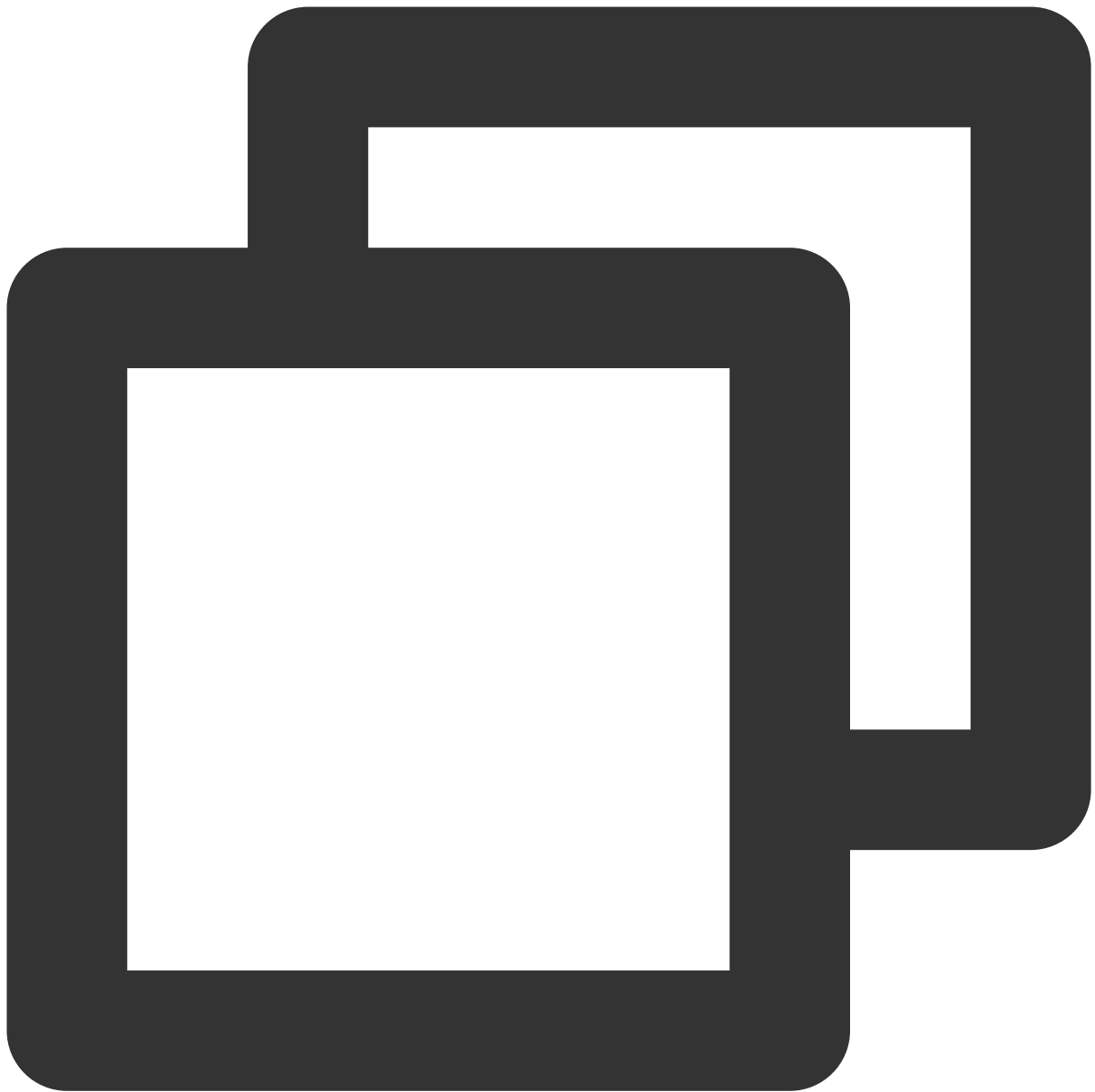
## headers



```
// response.headers  
readonly headers: Headers;
```

响应头部，详情参见 [Headers](#)。

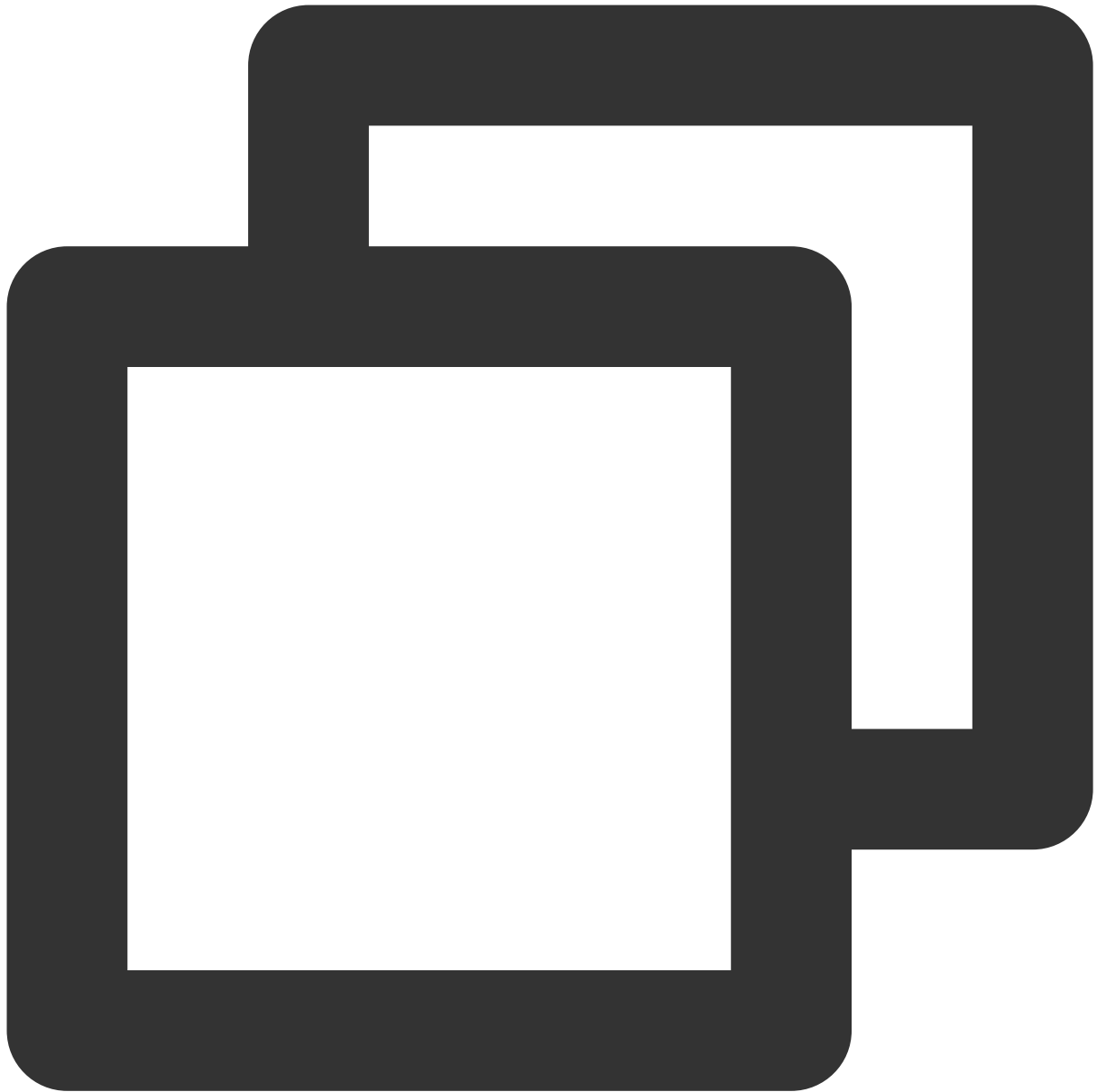
**ok**



```
// response.ok  
readonly ok: boolean;
```

标识响应是否成功（状态码在 200-299 范围内）。

**status**



```
// resposne.status  
readonly status: number;
```

响应状态代码。

**statusText**



```
// resposne.statusText  
readonly statusText: string;
```

响应的状态消息。

**url**

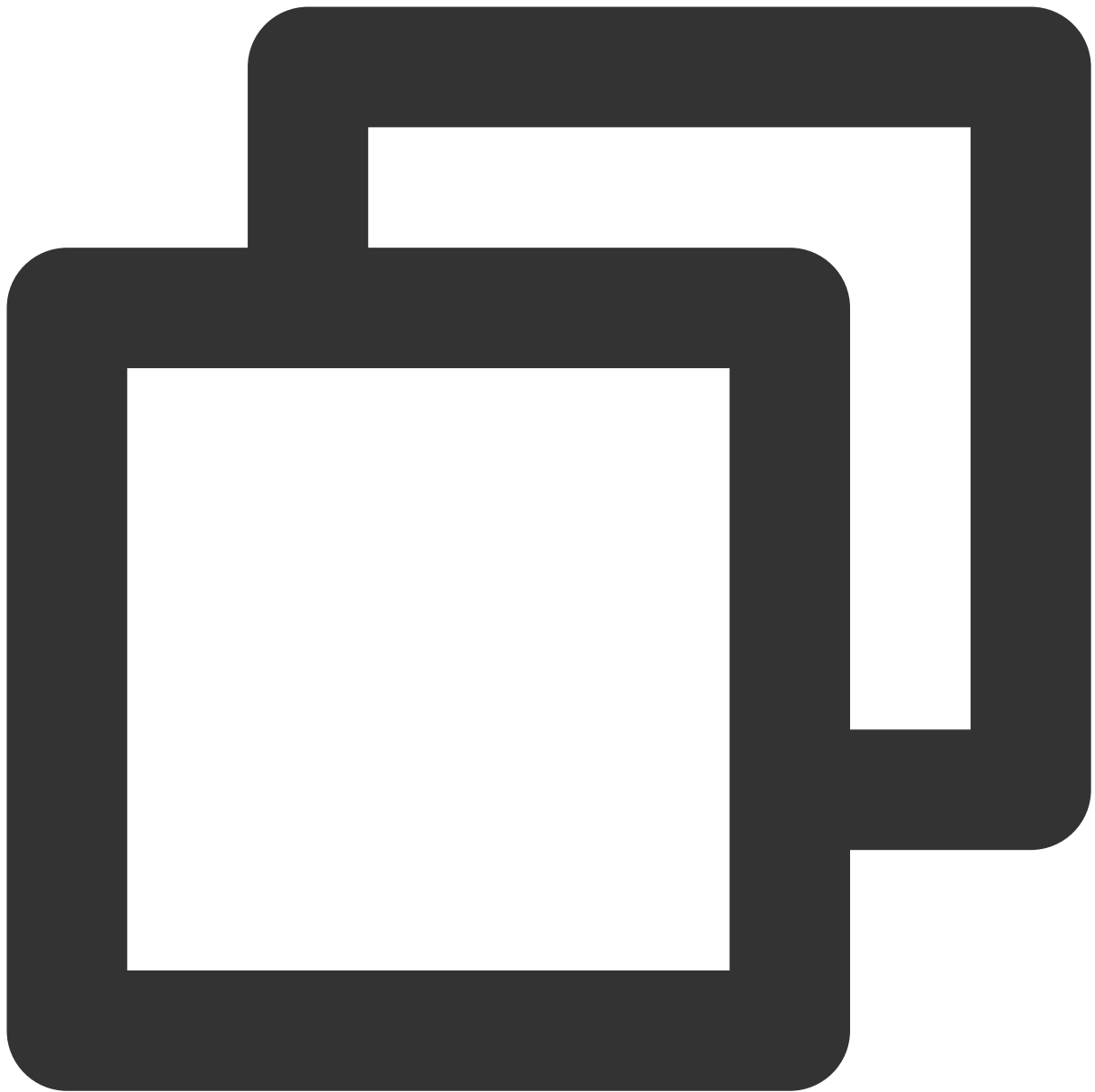




```
// response.url  
readonly url: string;
```

响应的 url。

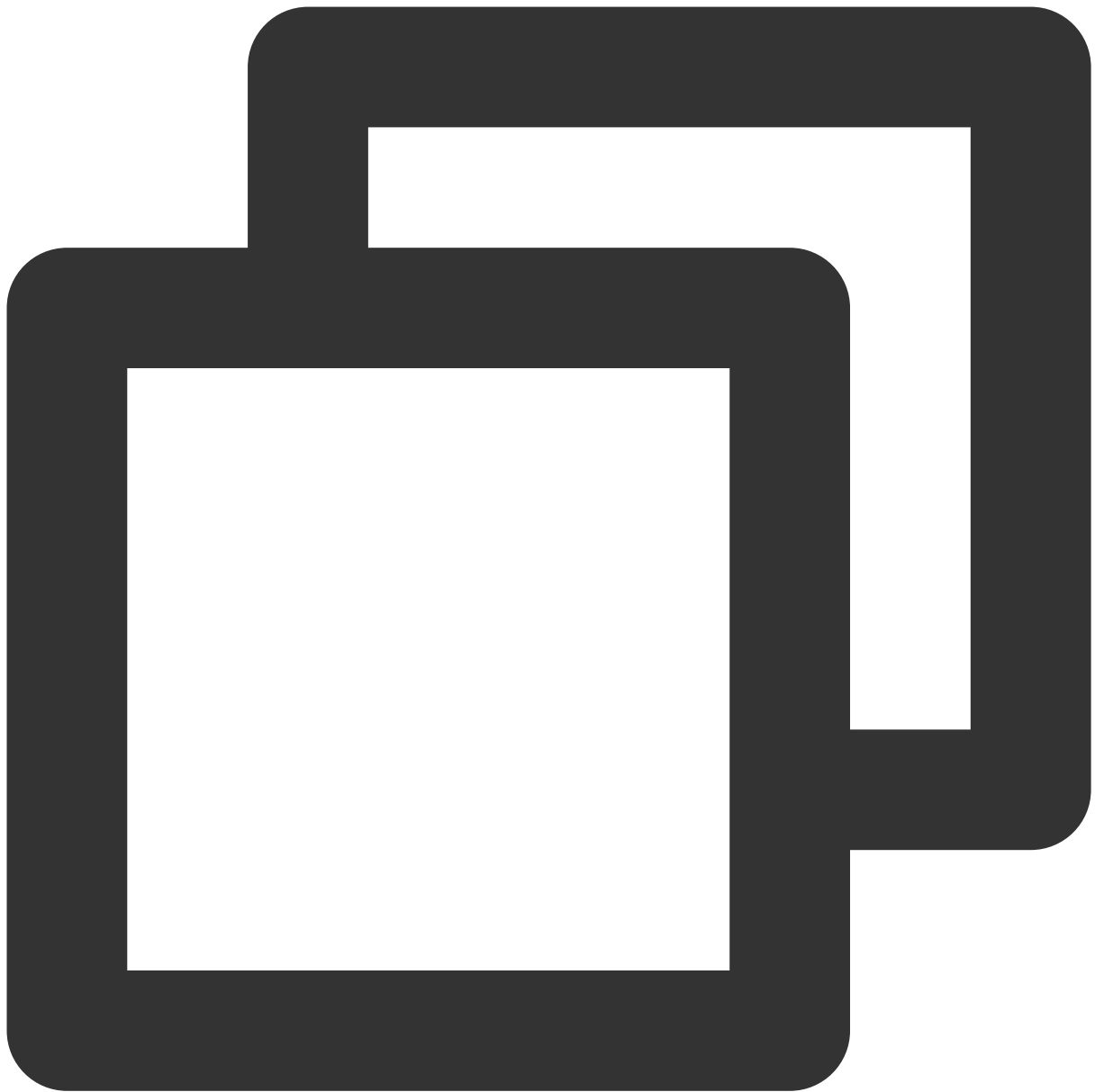
**redirected**



```
// response.redirected  
readonly redirected: boolean;
```

标识响应是否为重定向的结果。

## **redirectUrls**



```
// response.redirectUrls  
readonly redirectUrls: Array<String>
```

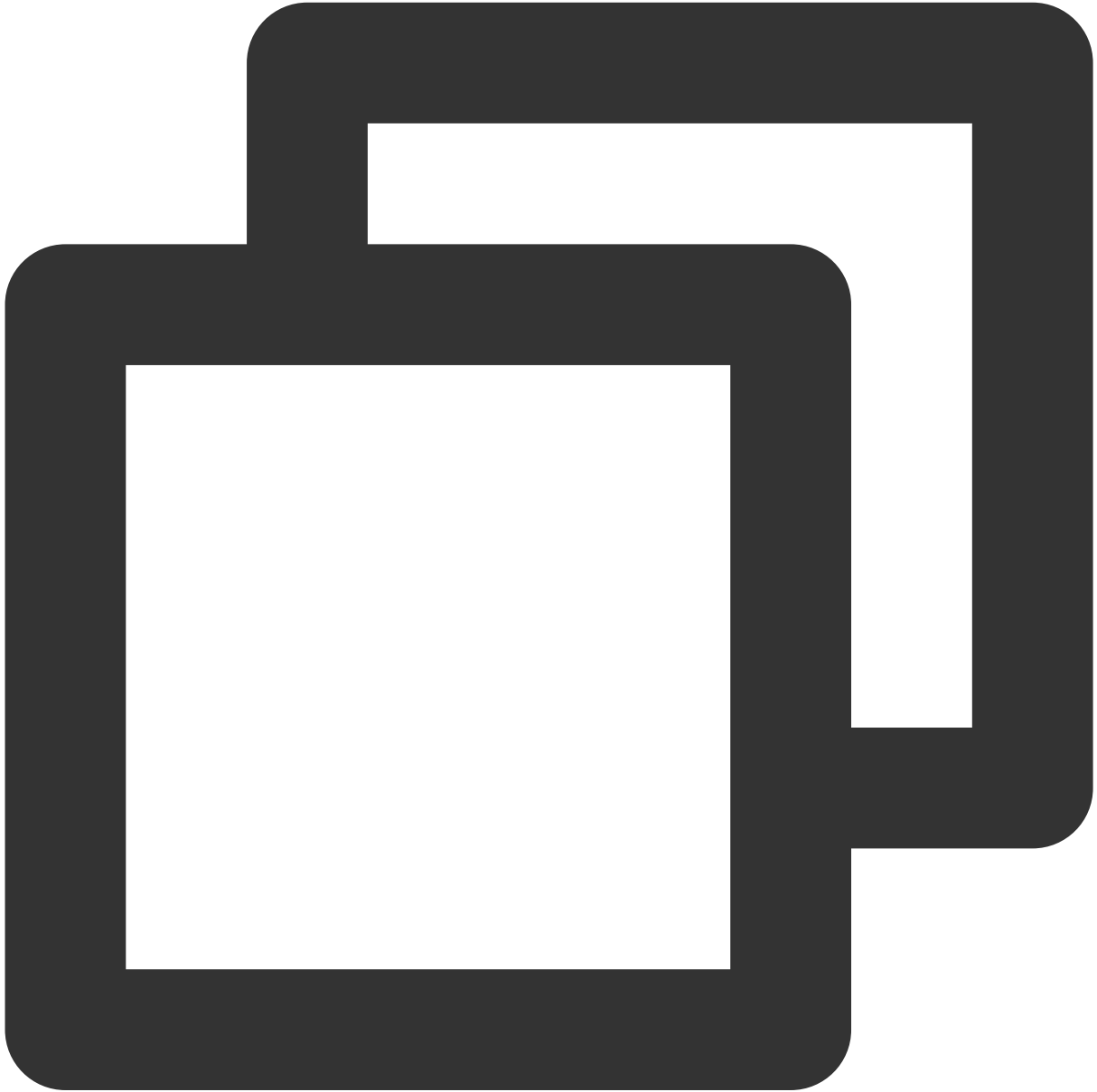
所有重定向 URL。

## 实例方法

### 注意

获取响应体方法，接收 `HTTP body` 最大字节数为 1M，超出大小会抛出 `OverSize` 异常。超出大小时推荐使用 `response.body` 流式读取，详情参见 [ReadableStream](#)。

## arrayBuffer



```
response.arrayBuffer(): Promise<ArrayBuffer>;
```

获取响应体，解析结果为 [ArrayBuffer](#)。

## blob



```
response.blob(): Promise<Blob>;
```

获取响应体，解析结果为 [Blob](#)。

**clone**



```
response.clone(copyHeaders?: boolean): Request;
```

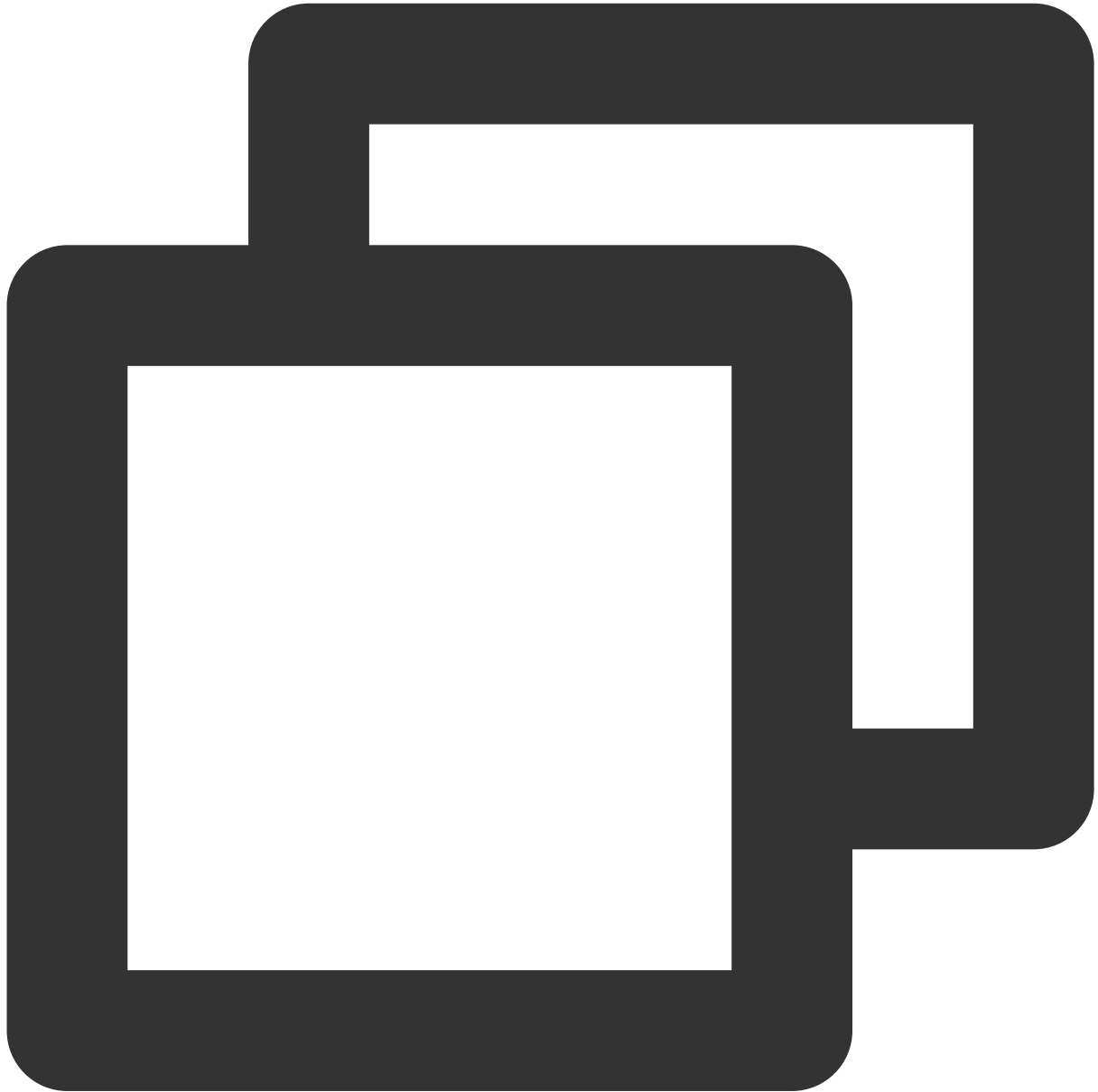
创建响应对象的副本。

### 参数

属性名	类型	必填	说明
copyHeaders	boolean	否	开启复制响应头，默认值为 <code>false</code> ，取值说明如下。 <code>true</code> 复制原对象的响应头。

			<code>false</code> 引用原对象的响应头。
--	--	--	----------------------------------

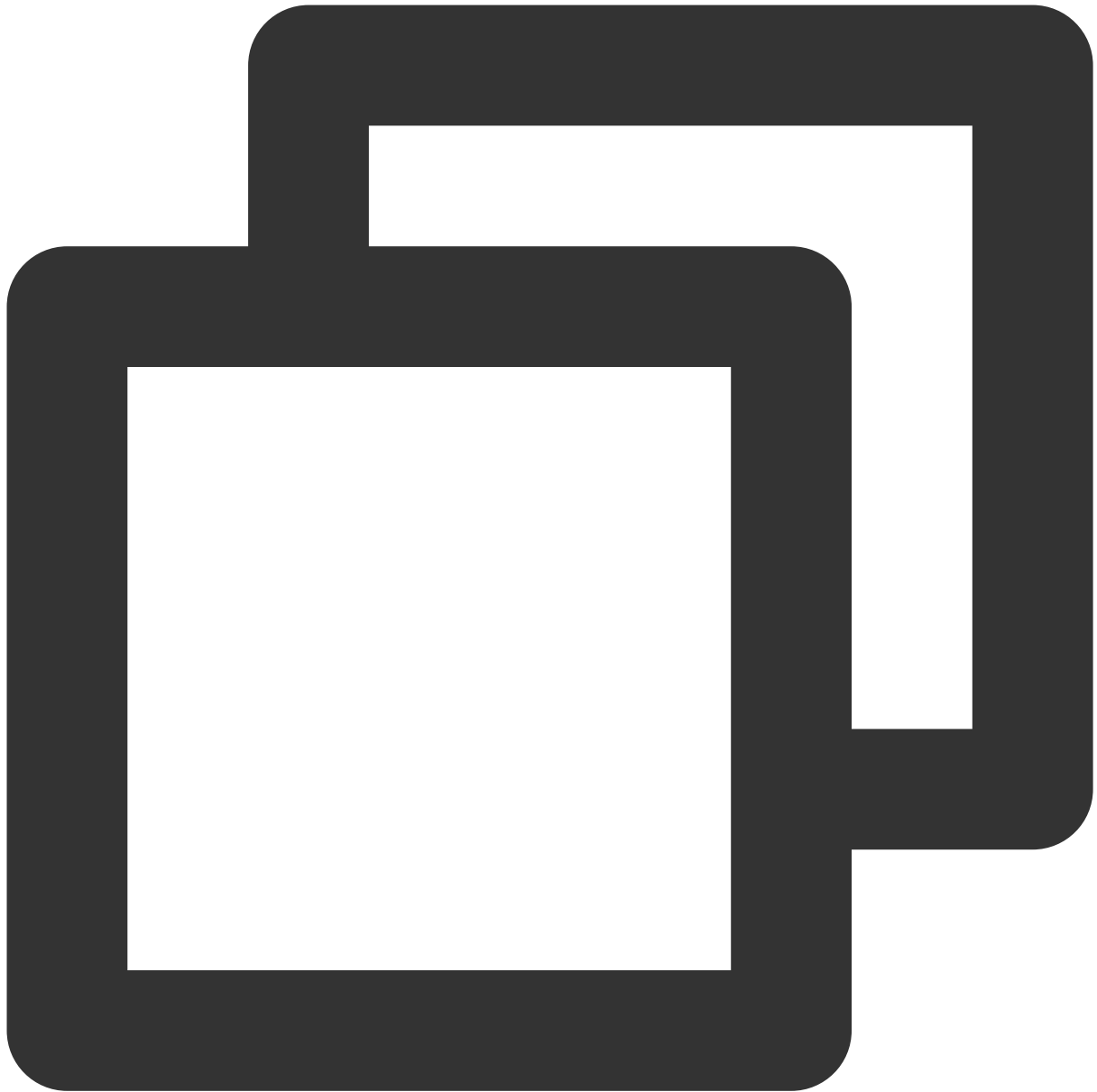
**json**



```
response.json(): Promise<object>;
```

获取响应体，解析结果为 `json`。

**text**

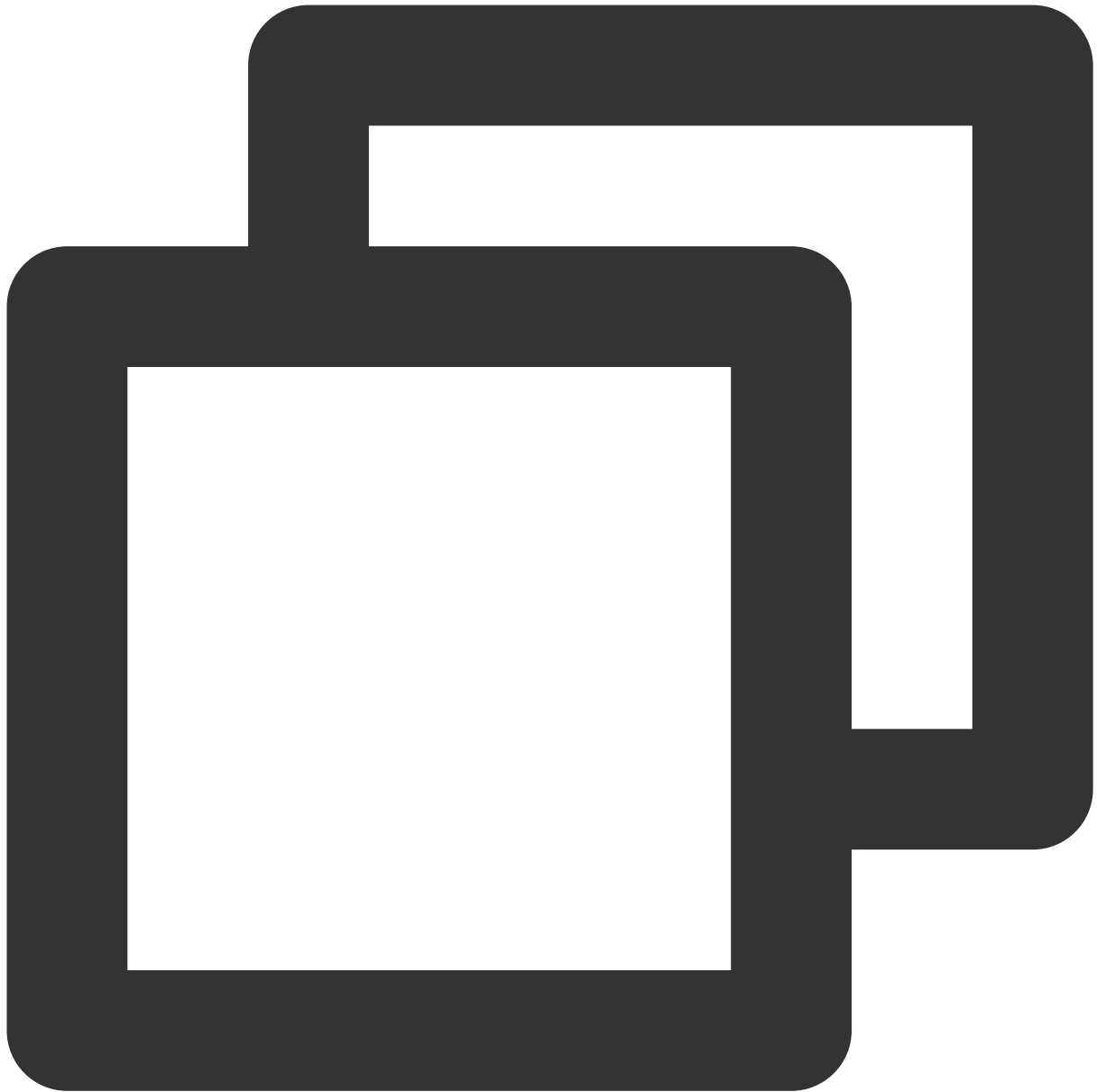


```
response.text(): Promise<string>;
```

获取响应体，解析结果为文本字符串。

## formData



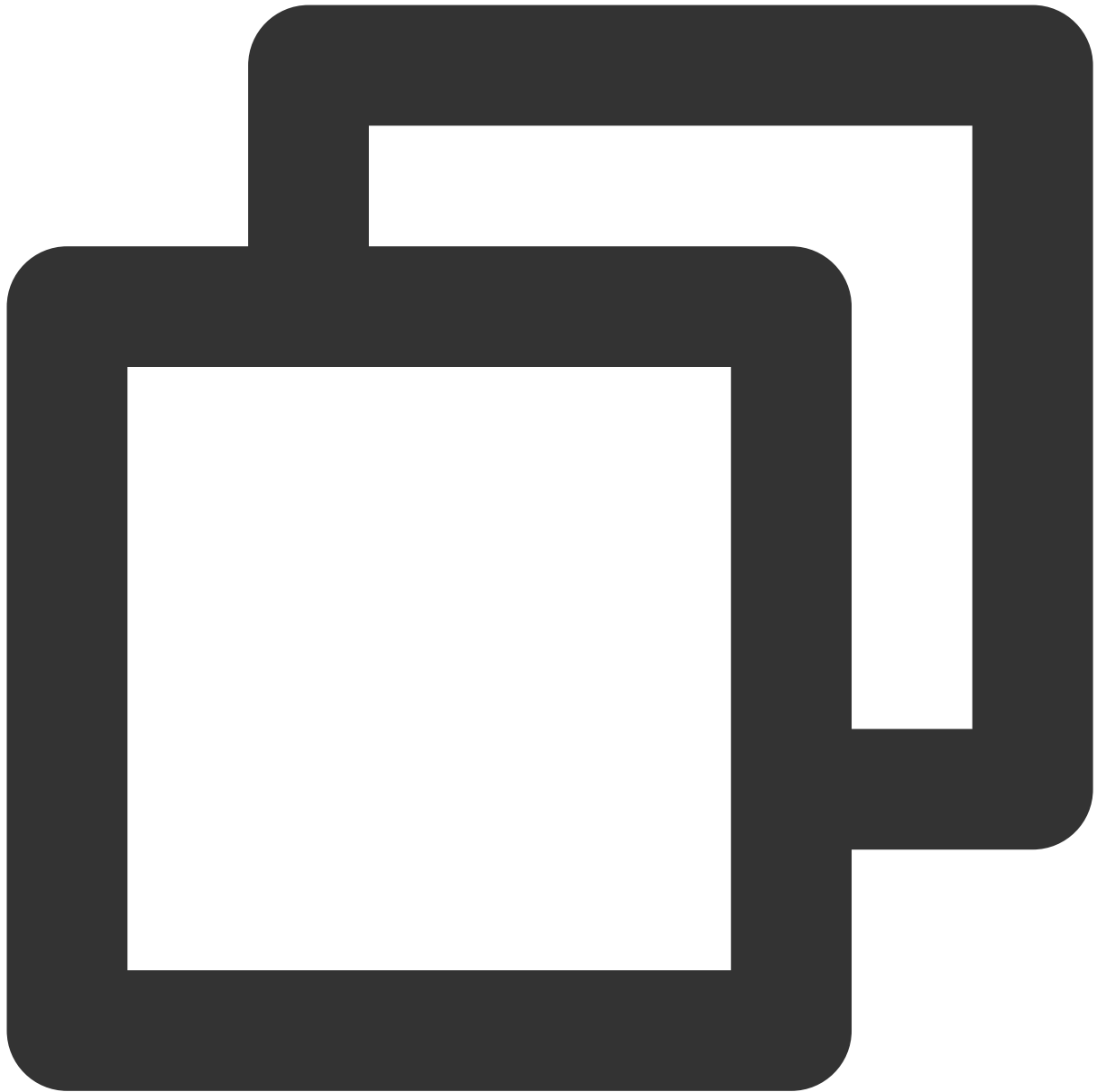


```
response.formData(): Promise<FormData>;
```

获取响应体，解析结果为 [FormData](#)。

## 静态方法

**error**



```
Response.error(): Response;
```

`error()` 方法返回一个包含网络错误相关信息的新 [Response](#) 对象。

## **redirect**



```
Response.redirect(url: string | URL, status?: number): Response;
```

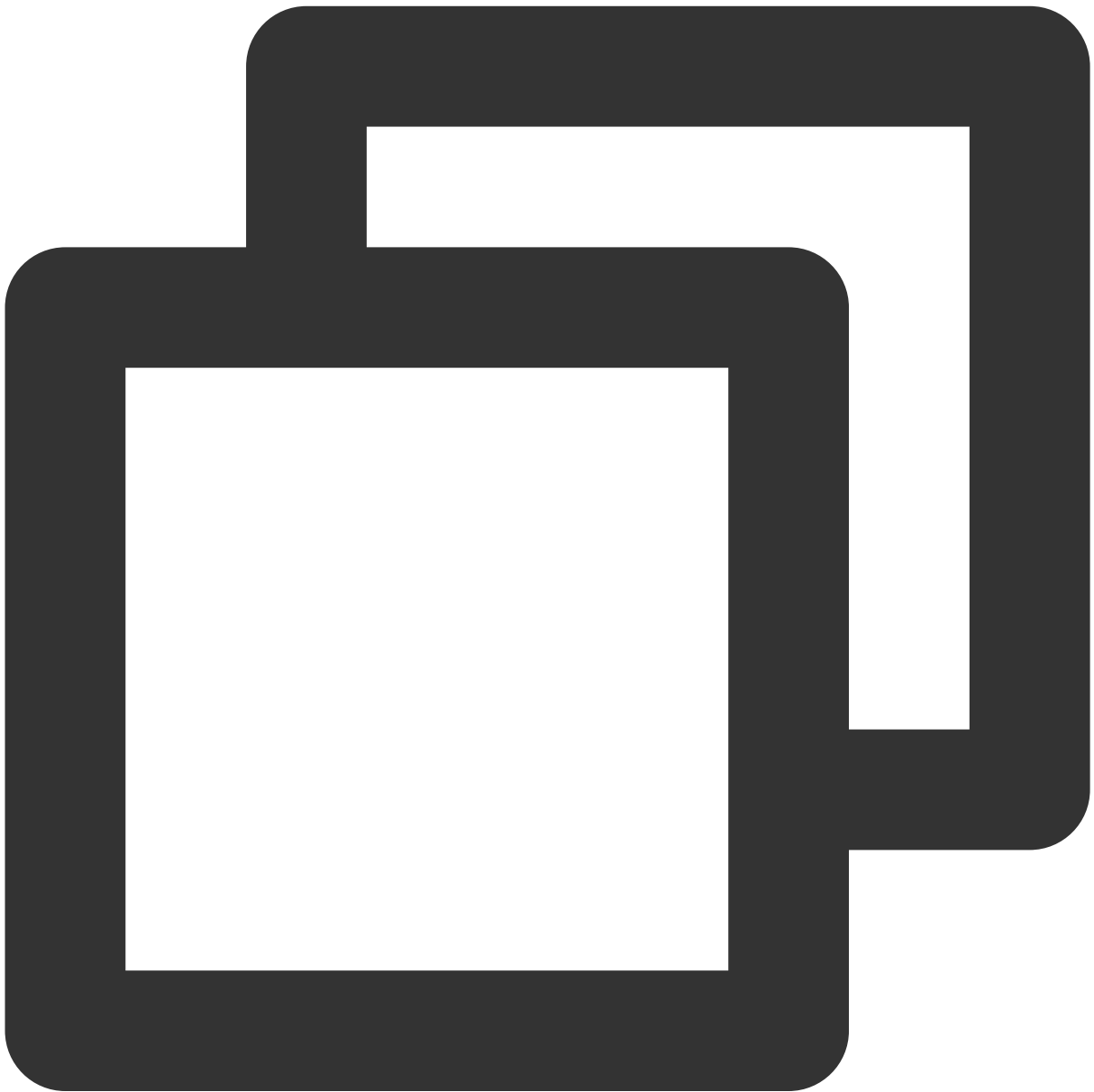
`redirect()` 方法返回一个可以重定向到指定 URL 的 [Response](#) 对象。

### 参数

属性名	类型	必填	说明
url	string	是	重定向地址
status	number	否	用于 response 的可选的状态码, 允许 301/302/303/307/308,

		默认 302
--	--	--------

## 示例代码



```
addEventListener('fetch', (event) => {  
  const response = new Response('hello world');  
  event.respondWith(response);  
});
```

---

## 相关参考

[MDN 官方文档：Response](#)

[示例函数：返回 HTML 页面](#)

[示例函数：修改响应头](#)

[示例函数：AB测试](#)

# Streams

## ReadableStream

最近更新时间：2023-02-07 17:59:18

**ReadableStream** 可读流，也称为可读端，基于 Web APIs 标准 [ReadableStream](#) 进行设计。

注意：

不支持直接构造 `ReadableStream` 对象，使用 [TransformStream](#) 构造得到。

## 描述

```
// 使用 TransformStream 构造得到 ReadableStream 对象
const { readable } = new TransformStream();
```

## 属性

```
// readable.locked
readonly locked: boolean;
```

标识流是否锁定。

说明：

流处于锁定状态的情况有：

- 一个流最多有一个激活的 `reader`，在 `reader` 调用 `releaseLock()` 方法前，该流均处于锁定状态。
- 流处于管道传输中，会处于锁定状态，直至管道传输结束。

## 方法

注意：

使用下述所有方法，要求当前流处于非锁定状态，否则会抛出异常。

## getReader

```
readable.getReader(options?: ReaderOptions): ReadableStreamDefaultReader | ReadableStreamBYOBReader;
```

创建一个 `Reader`，并锁定当前流，直至 `Reader` 调用 `releaseLock()` 释放锁。

### 参数

参数名称	类型	必填	说明
options	<a href="#">ReaderOptions</a>	是	生成 <code>Reader</code> 的配置项。

## ReaderOptions

`ReaderOptions` 对象属性如下所示。

属性名	类型	必填	说明
mode	string	否	<p><code>Reader</code> 类型，默认值为 <code>undefined</code>，取值说明如下。</p> <ul style="list-style-type: none"> <li><code>undefined</code> 创建 <a href="#">ReadableStreamDefaultReader</a> 类型的 <code>Reader</code>。</li> <li><code>byob</code> 创建 <a href="#">ReadableStreamBYOBReader</a> 类型的 <code>Reader</code>。</li> </ul>

## pipeThrough

```
readable.pipeThrough(transformStream: TransformStream, options?: PipeToOptions): ReadableStream;
```

流的管道处理。将当前可读流数据传输到参数 `transformStream` 的 `writable` 端，并返回 `transformStream` 的 `readable` 端。

注意：

在管道传输过程中，会对当前流 `writable` 端进行锁定。

## 参数

参数名称	类型	必填	说明
transfromStream	<a href="#">TransfromStream</a>	是	当前流传输到的目标对象。
options	<a href="#">PipeToOptions</a>	是	流处理配置项。

## PipeToOptions

流处理配置项如下所示：

属性名	类型	必填	说明
preventClose	boolean	否	取值 <code>true</code> 时，表示可读流的关闭，不会导致可写流关闭。
preventAbort	boolean	否	取值 <code>true</code> 时，表示可读流发生错误，不会导致可写流中止。
preventCancel	boolean	否	取值 <code>true</code> 时，表示可写流的错误，不会导致结束可读流。
signal	<a href="#">AbortSignal</a>	否	当 <code>signal</code> 被 <code>abort</code> 时，将会中止正在进行的传输。

## pipeTo

```
readable.pipeTo(destination: WritableStream, options?: PipeToOptions): Promise<void>;
```

流的管道处理，将当前可读流传输到 `destination` 可写流。

注意：

在管道传输过程中，会对当前流 `destination` 进行锁定。

## 参数

参数名称	类型	必填	说明
destination	<a href="#">WritableStream</a>	是	可写流。
options	<a href="#">PipeToOptions</a>	是	流处理配置项。

## tee

```
readable.tee(): [ReadableStream, ReadableStream];
```



---

将当前流派发出两个独立的可读流。

## cancel

```
readable.cancel(reason?: string): Promise<string>;
```

结束当前流。

## 相关参考

- [MDN 官方文档：ReadableStream](#)
- [示例函数：合并资源流式响应](#)
- [示例函数：m3u8 改写与鉴权](#)

# ReadableStreamBYOBReader

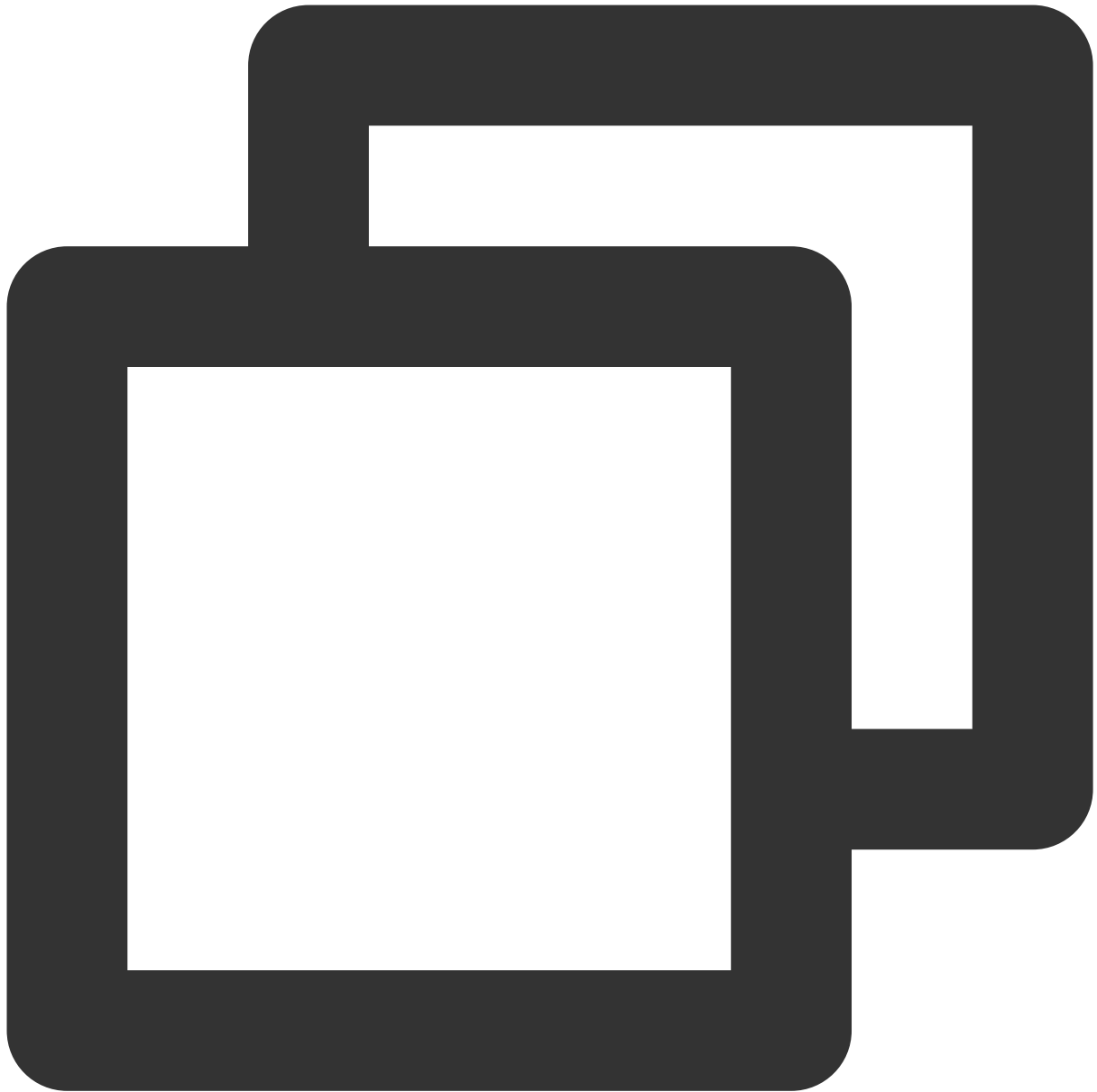
最近更新时间：2024-01-30 15:04:50

**ReadableStreamBYOBReader** 用于可读流操作，基于 Web APIs 标准 [ReadableStreamBYOBReader](#) 进行设计。`BYOB`（bring your own buffer），表示允许从流读取数据到缓冲区，从而最大限度的减少副本。

## 注意：

不支持直接构造 `ReadableStreamBYOBReader` 对象，使用 [ReadableStream.getReader](#) 方法得到。

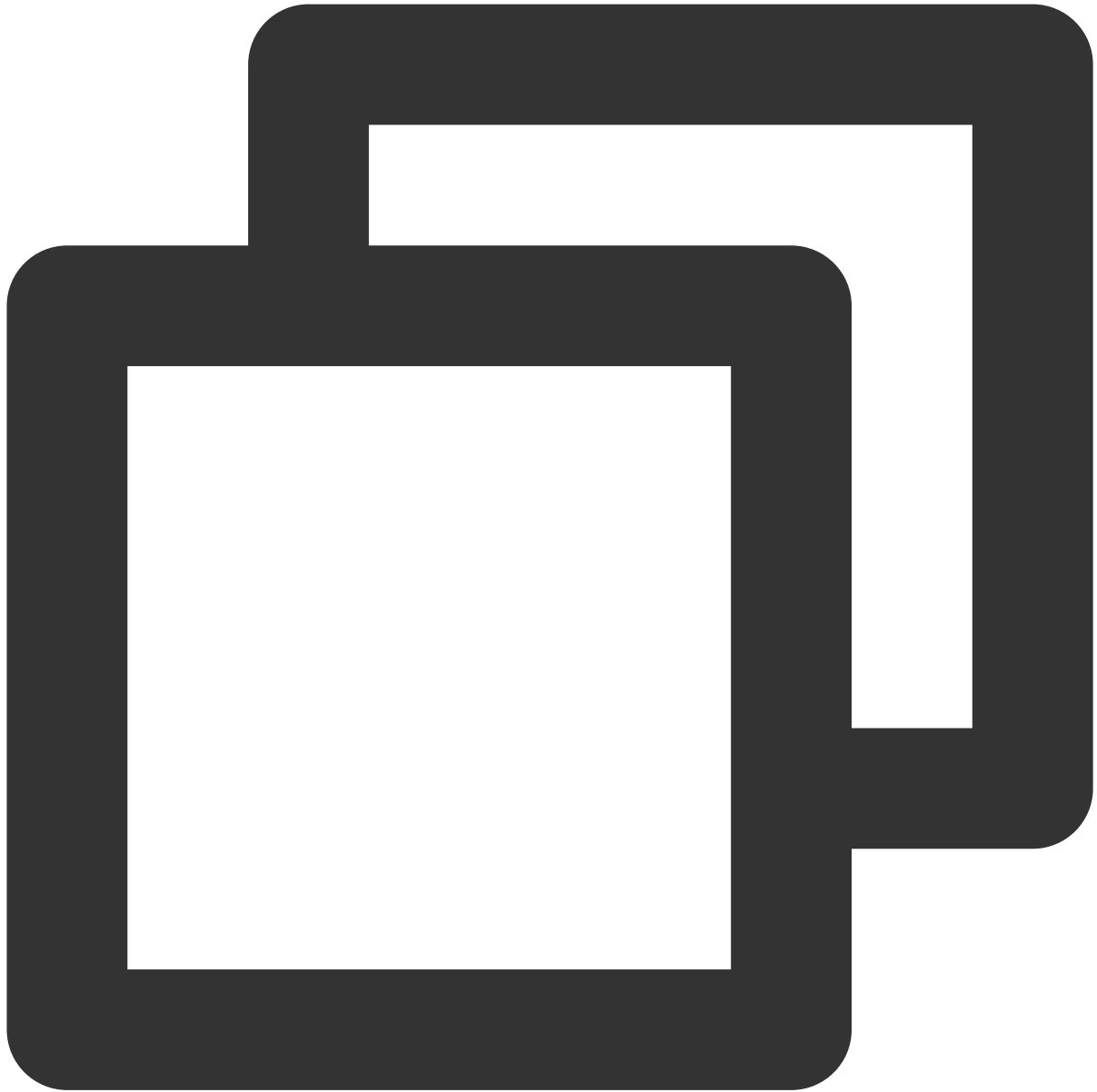
## 描述



```
// 使用 TransformStream 构造得到 ReadableStream 对象
const { readable } = new TransformStream();

// 使用 ReadableStream 对象获取 reader
const reader = readable.getReader({
  mode: 'byob',
});
```

## 属性

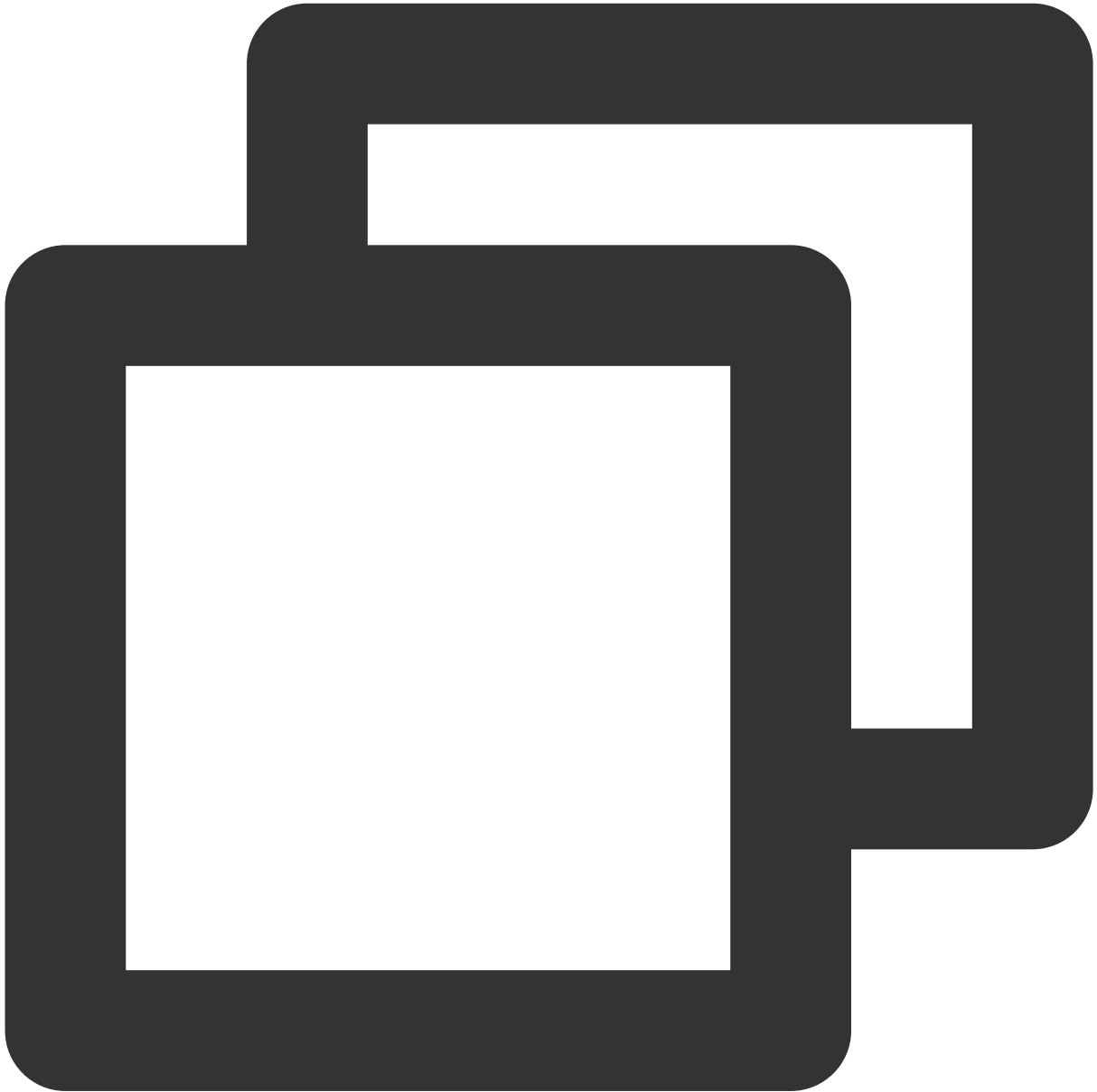


```
// readable.locked  
readonly locked: boolean;
```

返回 Promise 对象，如果流已关闭，Promise 状态为 `fulfilled`，如果流发生错误或读端锁已释放，Promise 状态为 `rejected`。

## 方法

### read



```
reader.read(bufferView: ArrayBufferView): Promise<{value: ArrayBufferView, done: bo
```

从流中读取数据到缓冲区 `bufferView` 。

#### 注意：

不允许前一个流读取操作结束前，调用 `read` 方法发起下一个流读取操作。

## 返回值

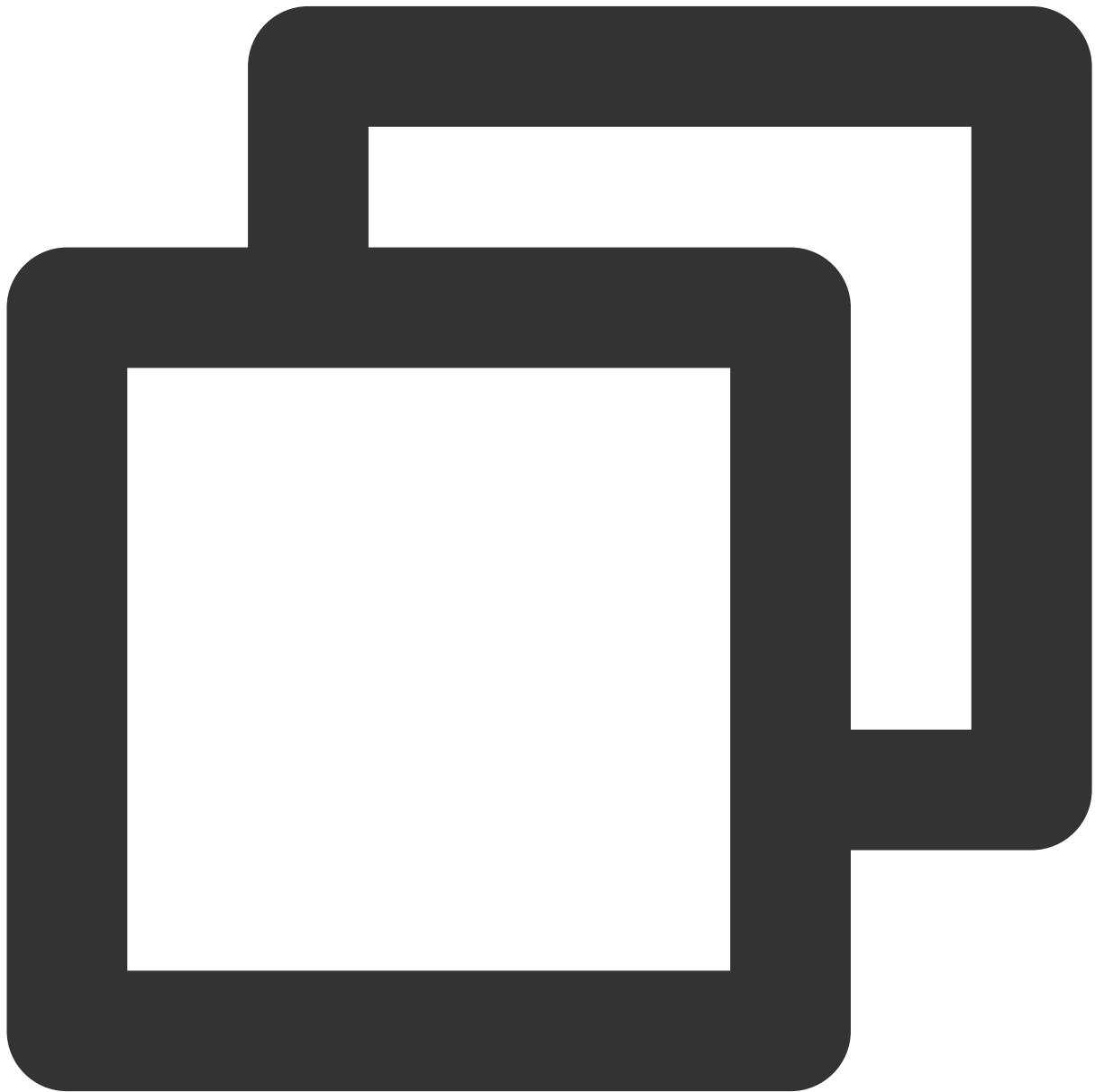
`reader.read` 返回 Promise 包含读取的数据与读取状态，说明如下：

如果有一个 chunk 是可用的，Promise 为 `fulfilled` 状态，包含 `{ value: theChunk, done: false }` 格式的对象。

如果流被关闭，Promise 将转为 `fulfilled` 状态，包含 `{ value: theChunk, done: true }` 格式的对象。

如果流出错，Promise 为 `rejected` 状态，并包含相关错误信息。

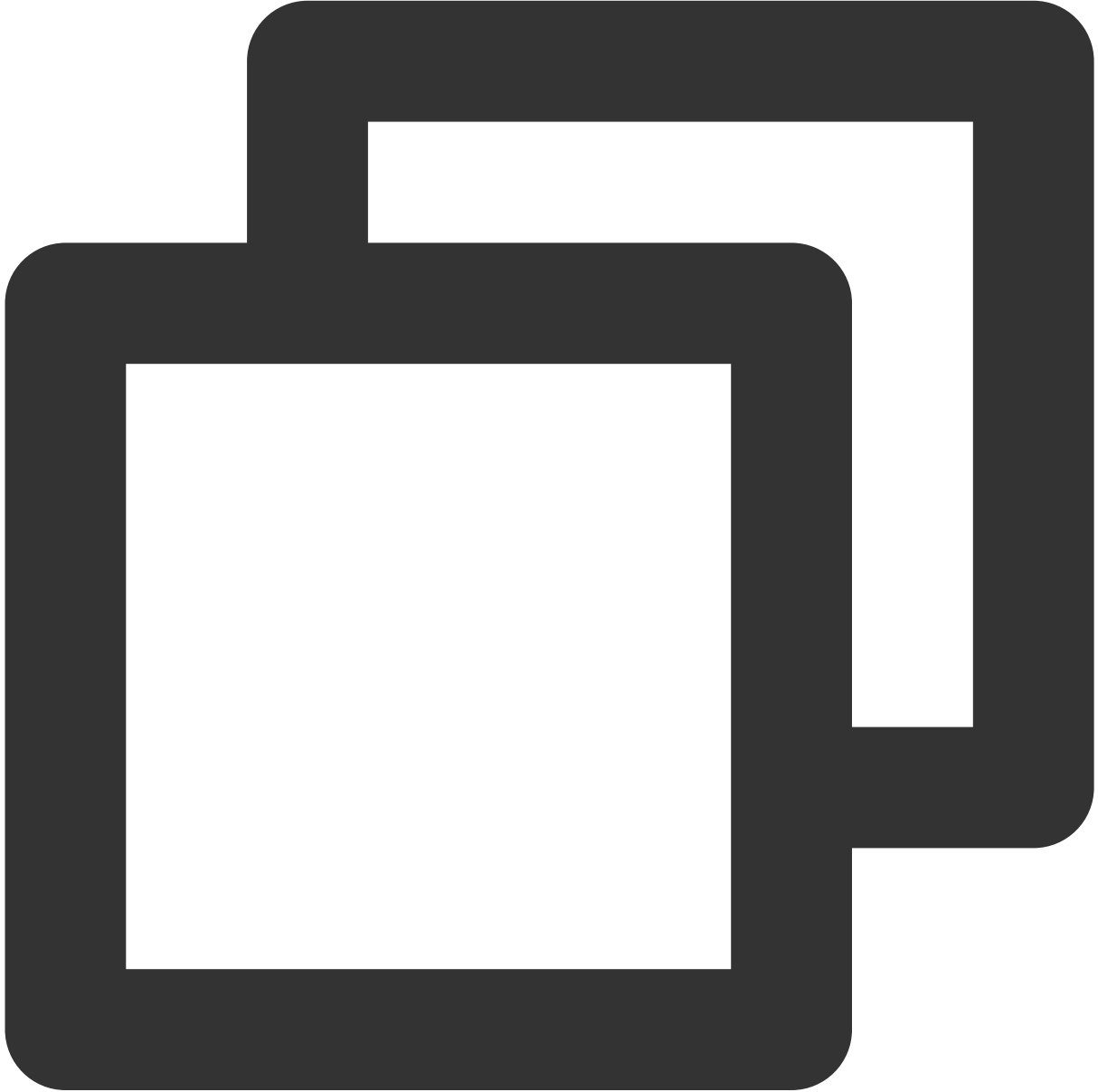
## cancel



```
reader.cancel(reason?: string): Promise<string>;
```

关闭流并结束读取操作。

## releaseLock



```
reader.releaseLock(): void;
```

取消与流的关联，并释放流的锁定。

---

## 相关参考

[MDN 官方文档：ReadableStreamBYOBReader](#)



# ReadableStreamDefaultReader

最近更新时间：2024-01-30 15:07:06

**ReadableStreamDefaultReader** 用于可读流操作，基于 Web APIs 标准 [ReadableStreamDefaultReader](#) 进行设计。

## 注意：

不支持直接构造 `ReadableStreamDefaultReader` 对象，使用 [ReadableStream.getReader](#) 方法得到。

## 描述

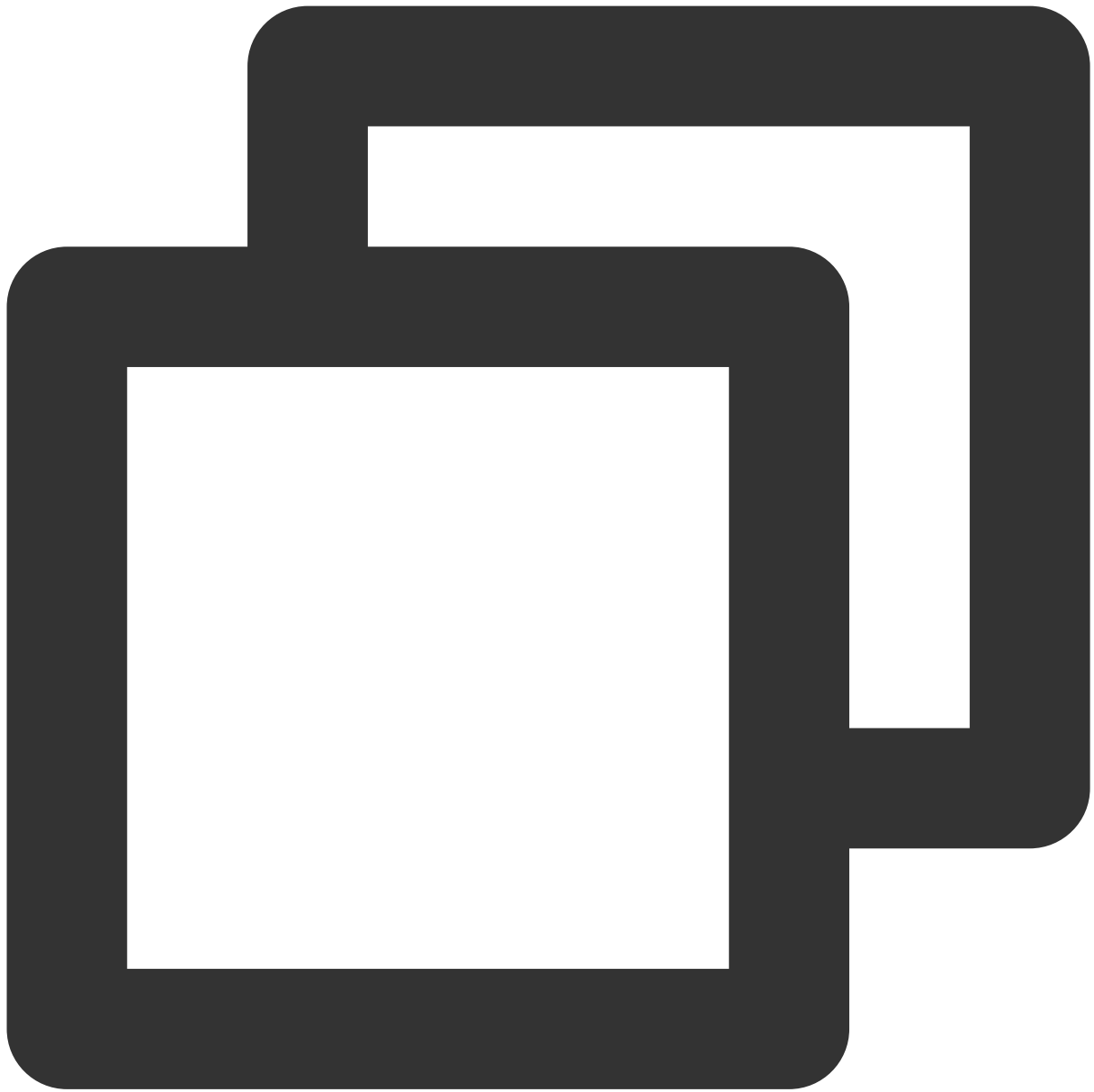


```
// 使用 TransformStream 构造得到 ReadableStream 对象
const { readable } = new TransformStream();

// 使用 ReadableStream 对象获取 reader
const reader = readable.getReader();
```

## 属性

closed



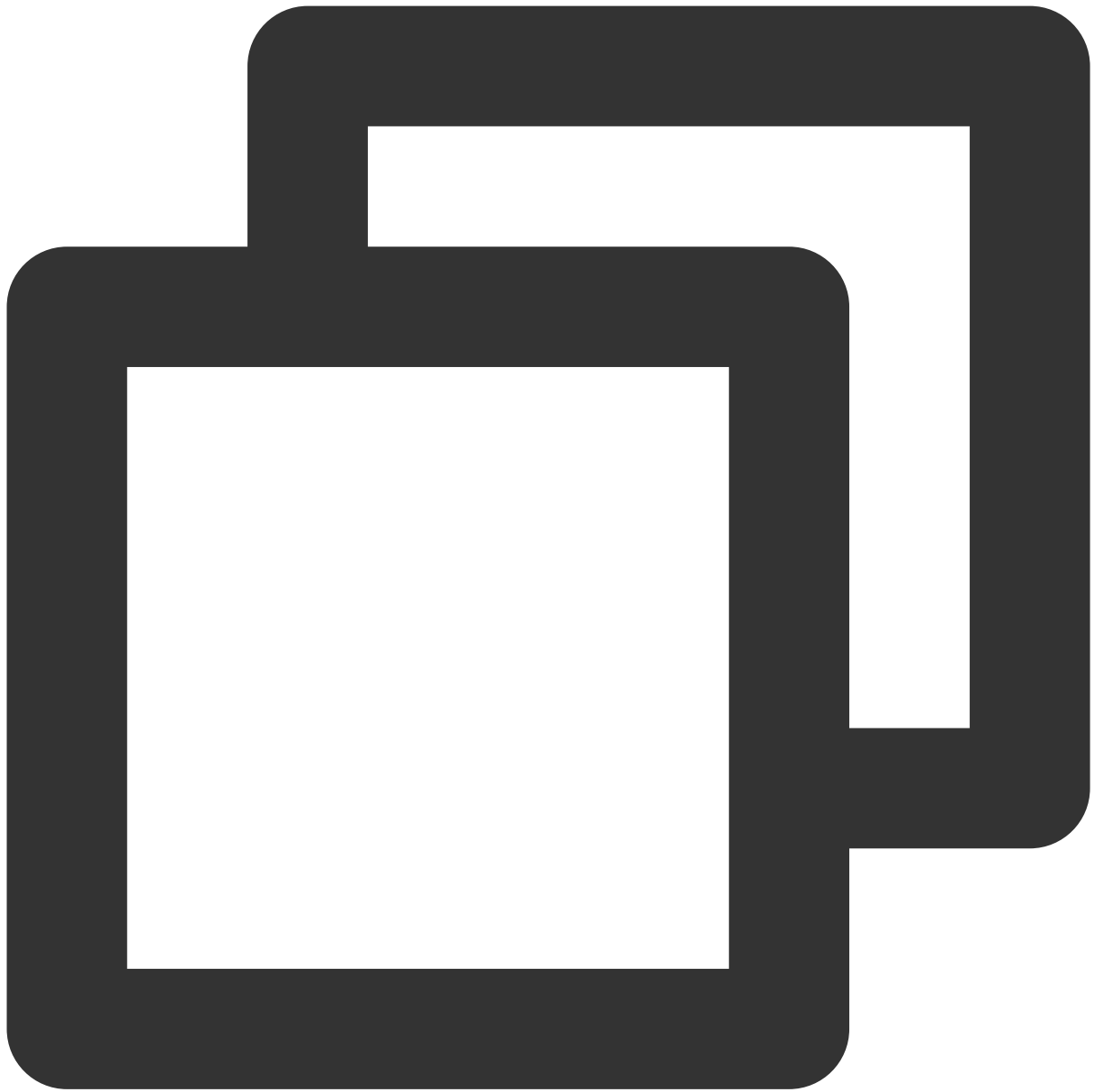
```
// reader.closed  
readonly closed: Promise<void>;
```

返回 **Promise** 对象，如果流已关闭，**Promise** 状态为 `fulfilled`，如果流发生错误或读端锁已释放，**Promise** 状态为 `rejected`。

## 方法

---

## read



```
reader.read(): Promise<{value: Chunk, done: boolean}>;
```

从流中读取数据。

### 注意：

不允许前一个流读取操作结束前，调用 `read` 方法发起下一个流读取操作。

### 返回值

`reader.read` 返回 Promise 包含读取的数据（[Chunk](#)）与读取状态，说明如下：

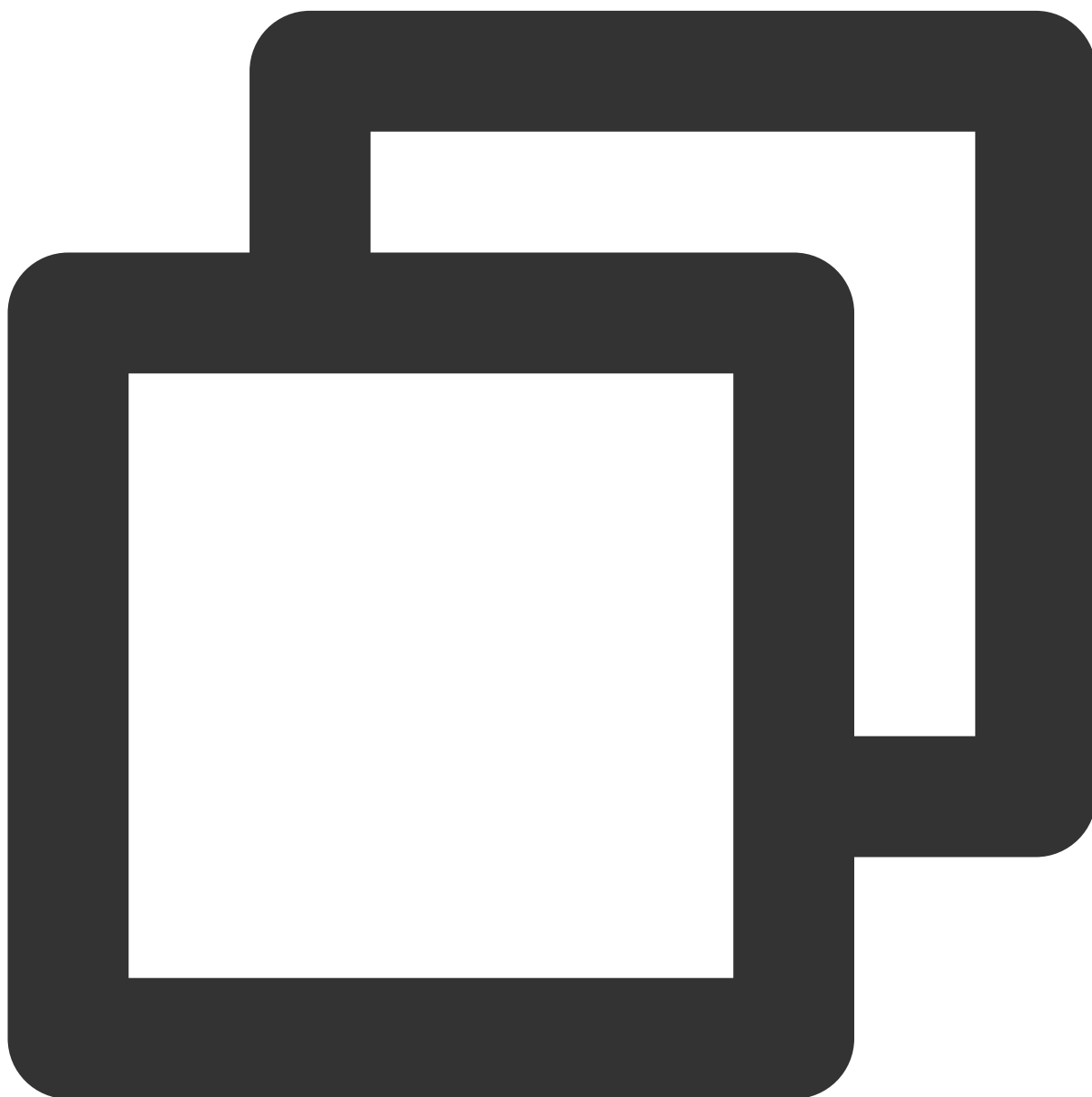
如果有一个 chunk 可用, Promise 为 `fulfilled` 状态, 包含 `{ value: theChunk, done: false }` 格式的对象。

如果流被关闭, Promise 为 `fulfilled` 状态, 包含 `{ value: undefined, done: true }` 格式的对象。

如果流出错, Promise 为 `rejected` 状态, 并包含相关错误信息。

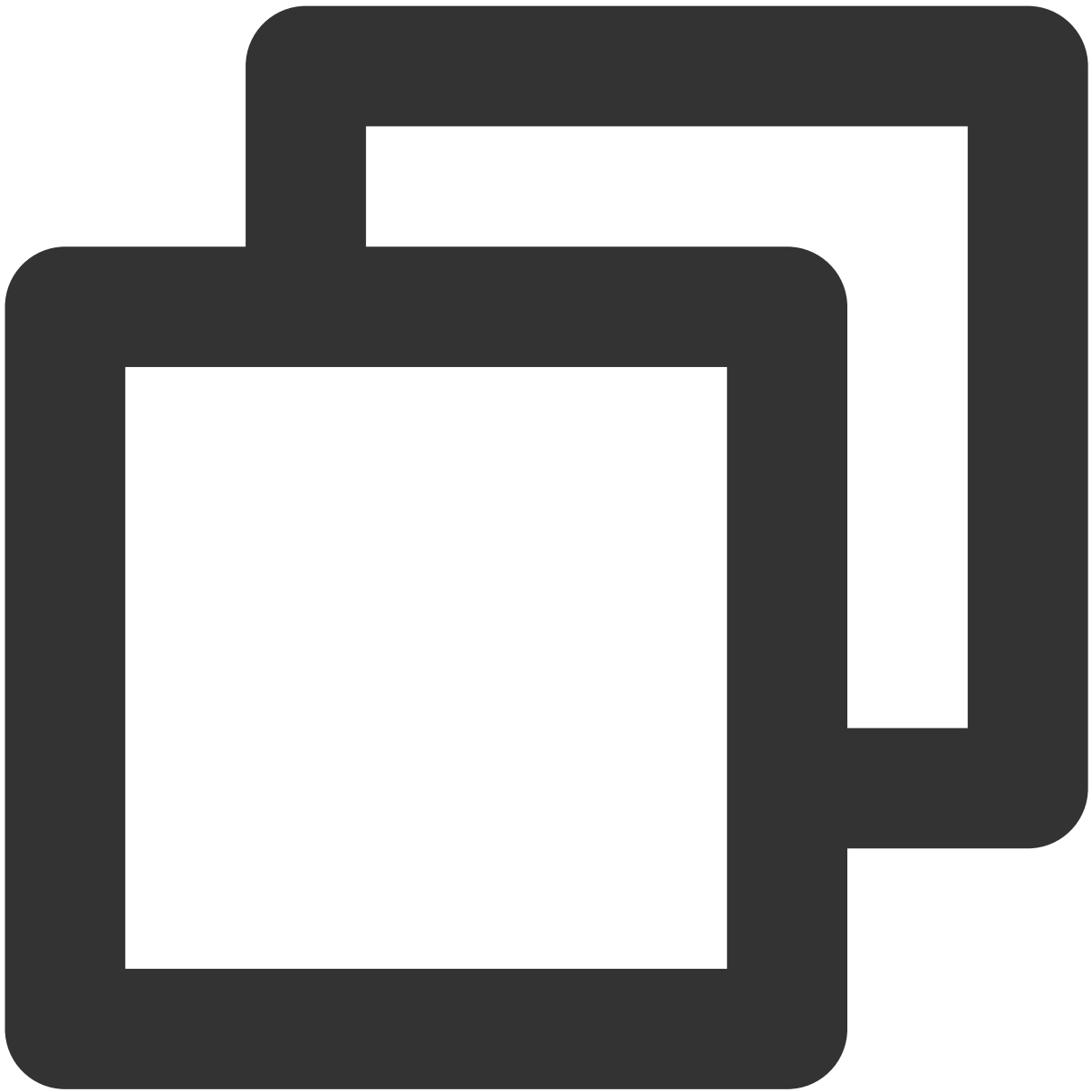
## Chunk

从流中读取的数据 `Chunk`, 描述如下:



```
type Chunk = string | ArrayBuffer | ArrayBufferView;
```

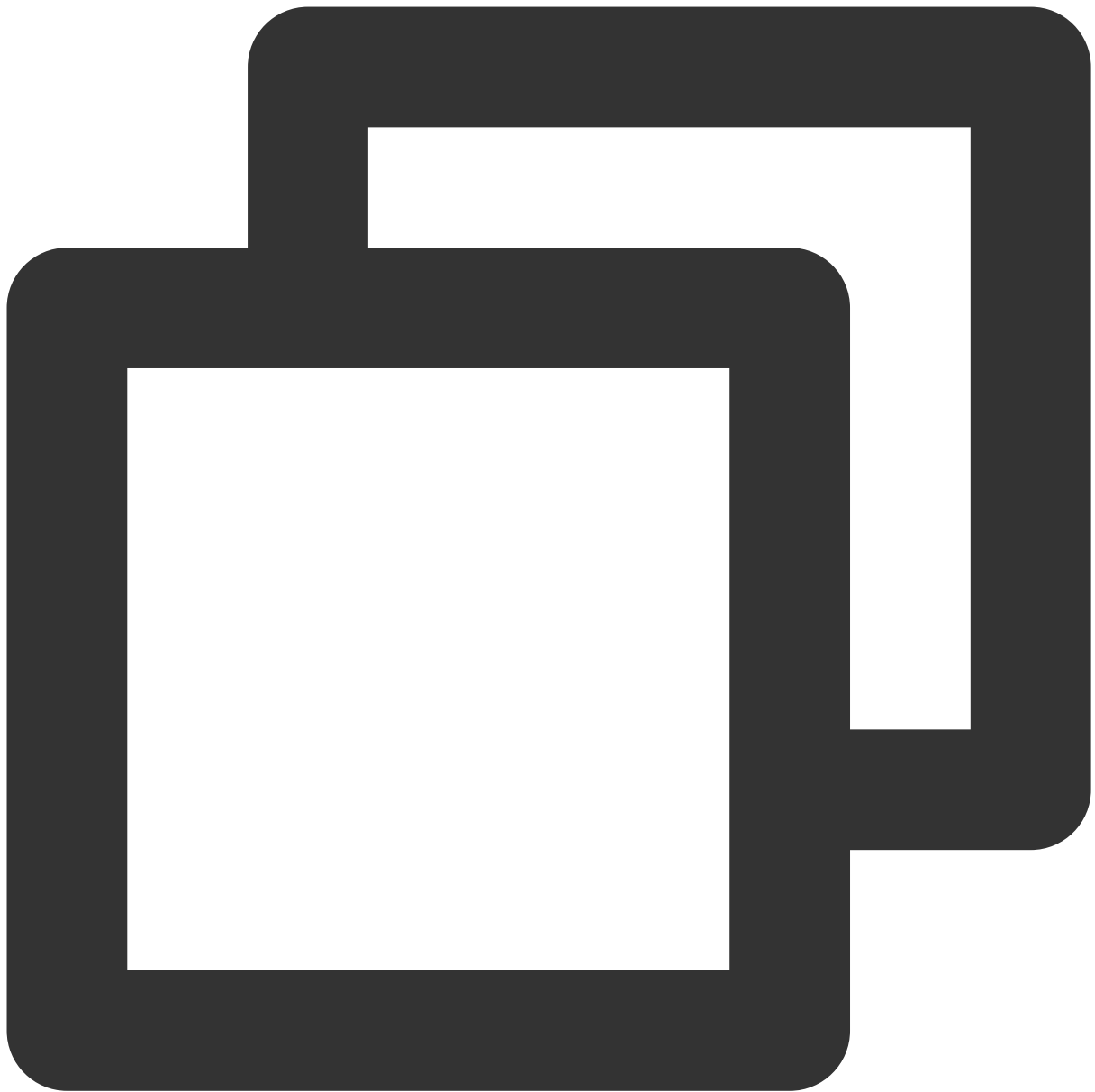
cancel



```
reader.cancel(reason?: string): Promise<string>;
```

关闭流并结束读取操作。

**releaseLock**



```
reader.releaseLock(): void;
```

取消与流的关联，并释放流的锁定。

## 相关参考

[MDN 官方文档：ReadableStreamDefaultReader](#)

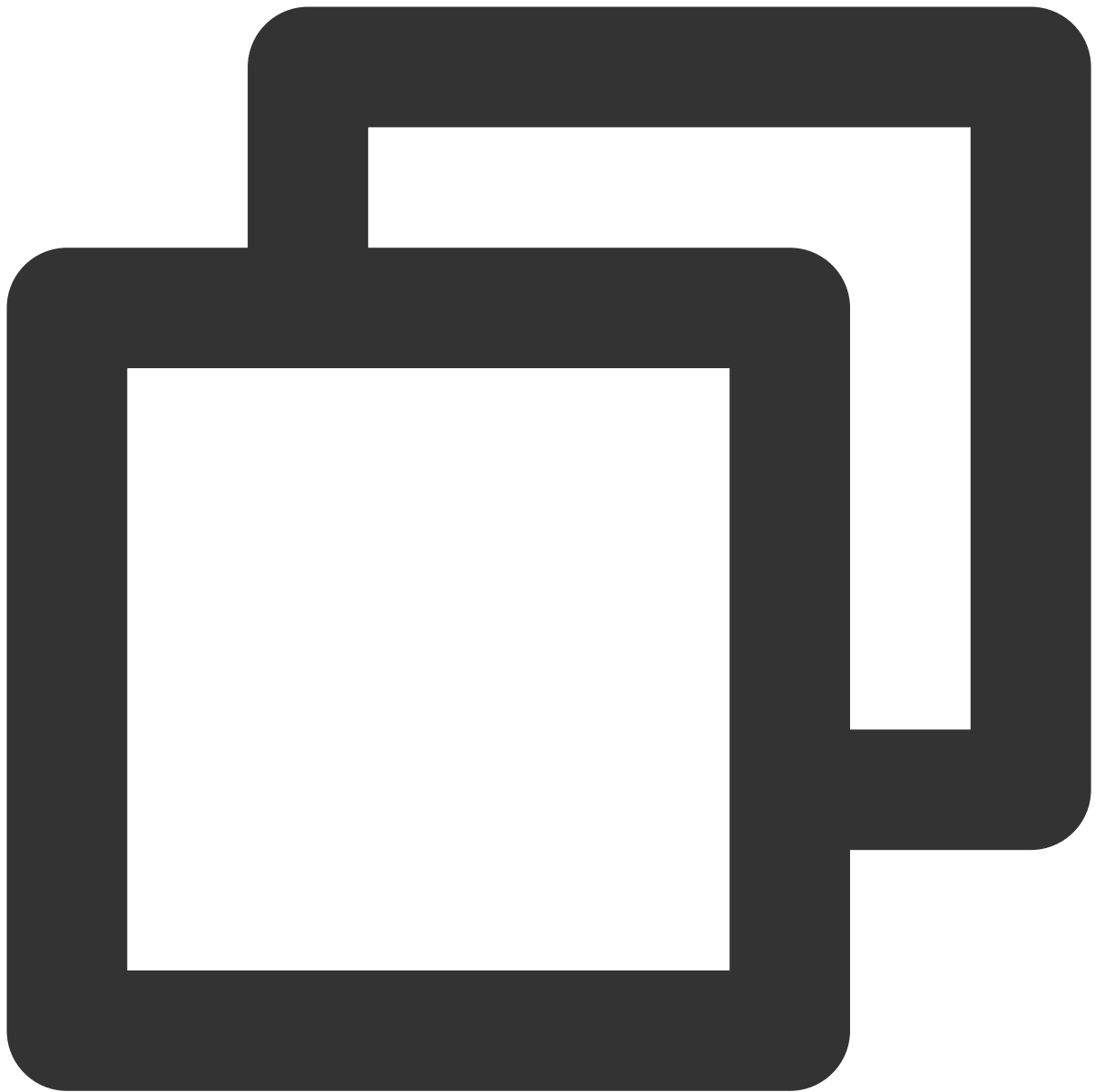
# TransformStream

最近更新时间：2024-01-30 15:33:56

**TransformStream** 由一对流组成，一个可读流，称为可读端，一个可写流，称为可写端。基于 Web APIs 标准 [TransformStream](#) 进行设计。

## 构造函数





```
const { readable, writable } = new TransformStream(transformer?: any, writableStrat
```

### 参数

参数名称	类型	必填	说明
transformer	any	否	暂不支持，传值不生效，自动忽略该参数。
writableStrategy	<a href="#">WritableStrategy</a>	否	可写端策略配置。

**WritableStrategy**

属性名	类型	必填	说明
highWaterMark	number	是	可写端缓冲区大小，以字节为单位，默认值为 32K, 最大值为 256K, 超过最大值则会自动调整为 256K。

## 属性

**readable**



```
readonly readable: ReadableStream;
```

可读端，详情参见 [ReadableStream](#)。

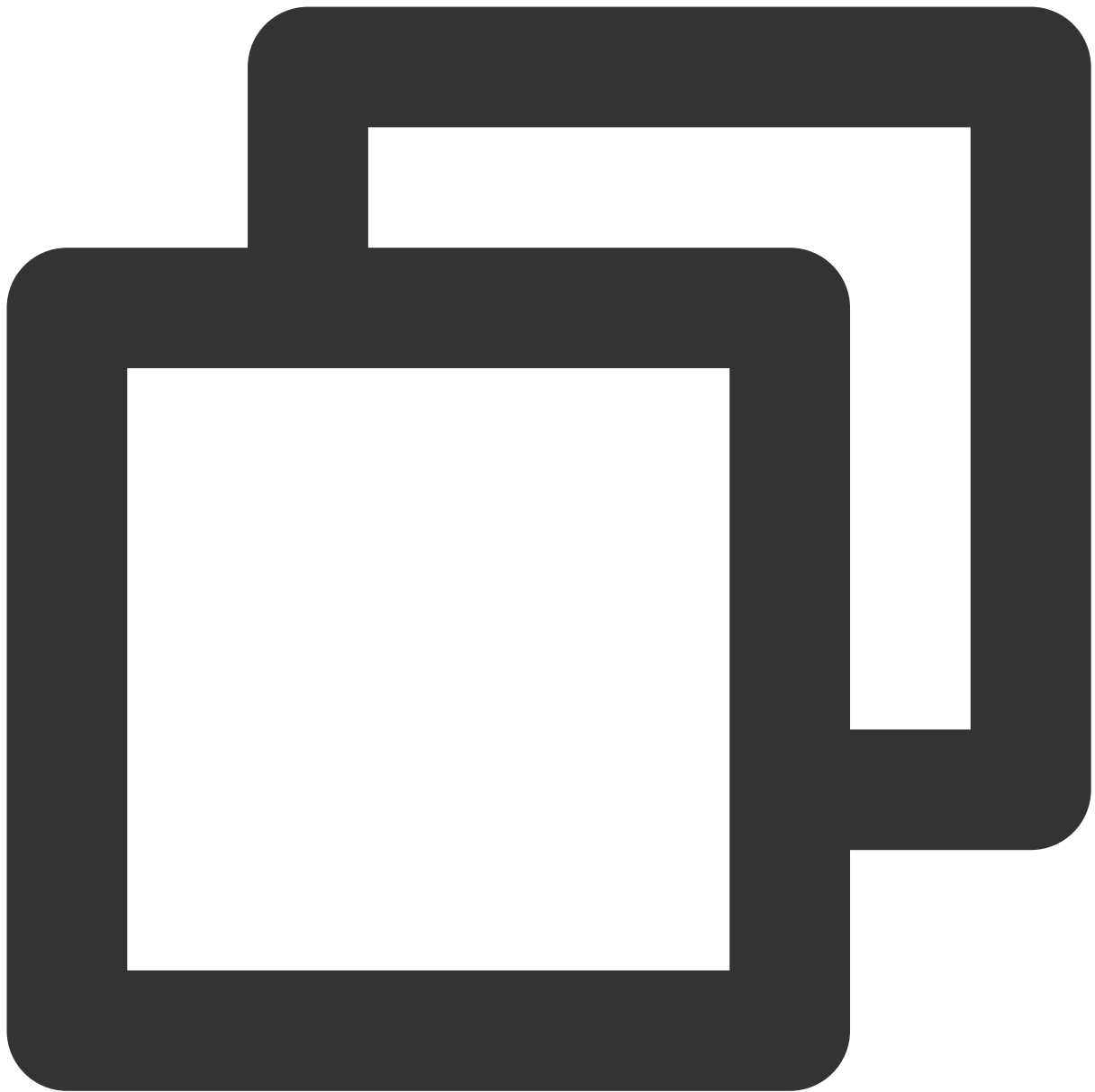
## writable



```
readonly writable: WritableStream;
```

可写端，详情参见 [WritableStream](#)。

## 示例代码



```
async function handleEvent(event) {  
  // 生成可读端与可写端  
  const { readable, writable } = new TransformStream();  
  // 获取远程资源  
  const response = await fetch('https://www.tencentcloud.com/');  
  // 流式响应客户端  
  response.body.pipeTo(writable);  
  
  return new Response(readable, response);  
}
```

```
addEventListener('fetch', (event) => {  
  event.respondWith(handleEvent(event));  
});
```

## 相关参考

[MDN 官方文档：TransformStream](#)

示例函数：[合并资源流式响应](#)

示例函数：[m3u8 改写与鉴权](#)

# WritableStream

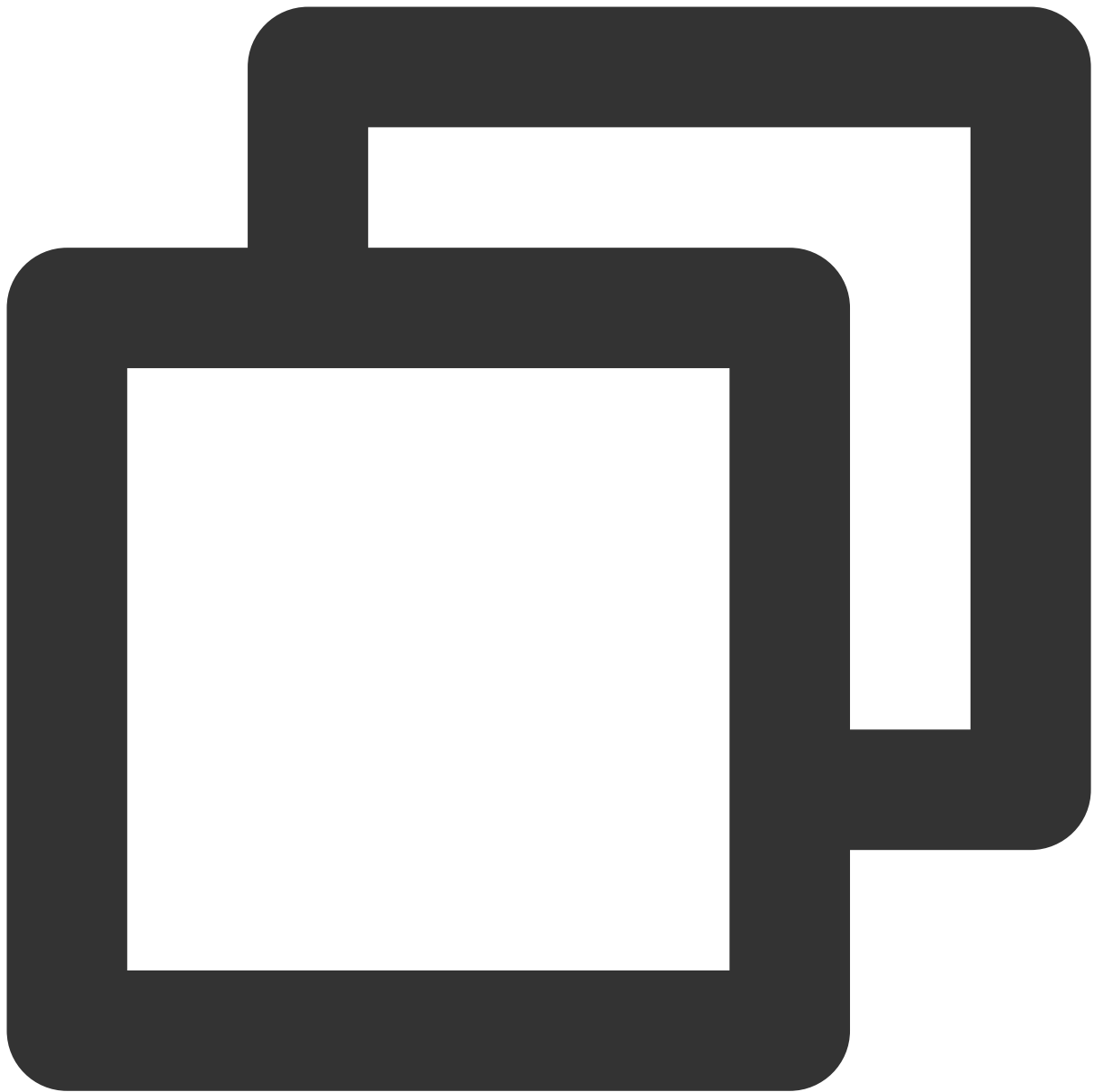
最近更新时间：2024-01-30 15:59:48

**WritableStream** 可写流，也称为可写端。基于 Web APIs 标准 [WritableStream](#) 进行设计。

## 注意：

不支持直接构造 `WritableStream` 对象，使用 [TransformStream](#) 构造得到。

## 描述



```
// 使用 TransformStream 构造得到 WritableStream 对象  
const { writable } = new TransformStream();
```

## 属性

**locked**





```
// writable.locked  
readonly locked: boolean;
```

标识流是否已锁定。

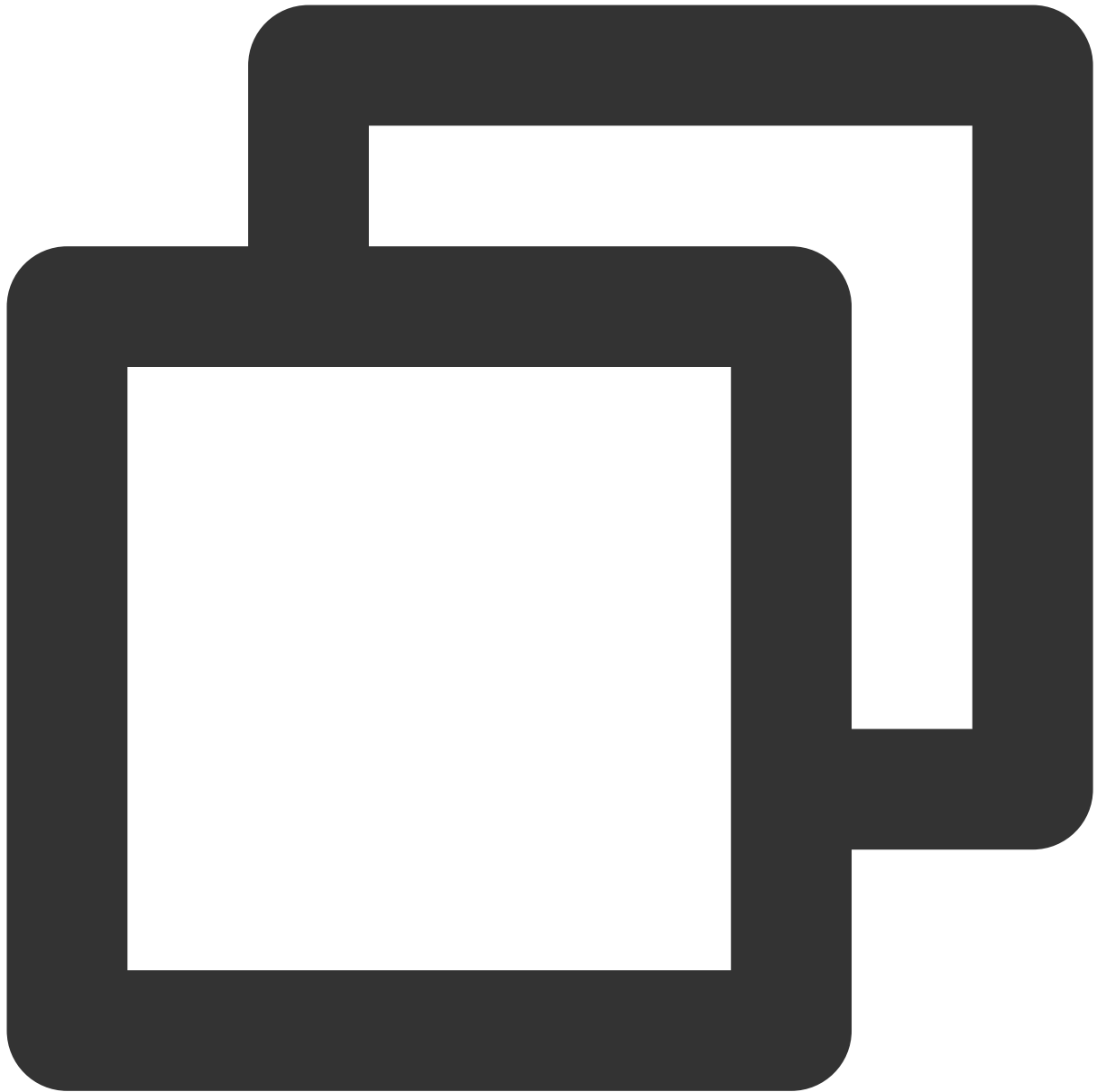
#### 说明：

流处于锁定状态的情况有：

一个流最多有一个激活的 `writer`，在 `writer` 调用 `releaseLock()` 方法前，该流均处于锁定状态。

流处于管道传输中，会处于锁定状态，直至管道传输结束。

#### highWaterMark



```
// writable.highWaterMark  
readonly highWaterMark: number;
```

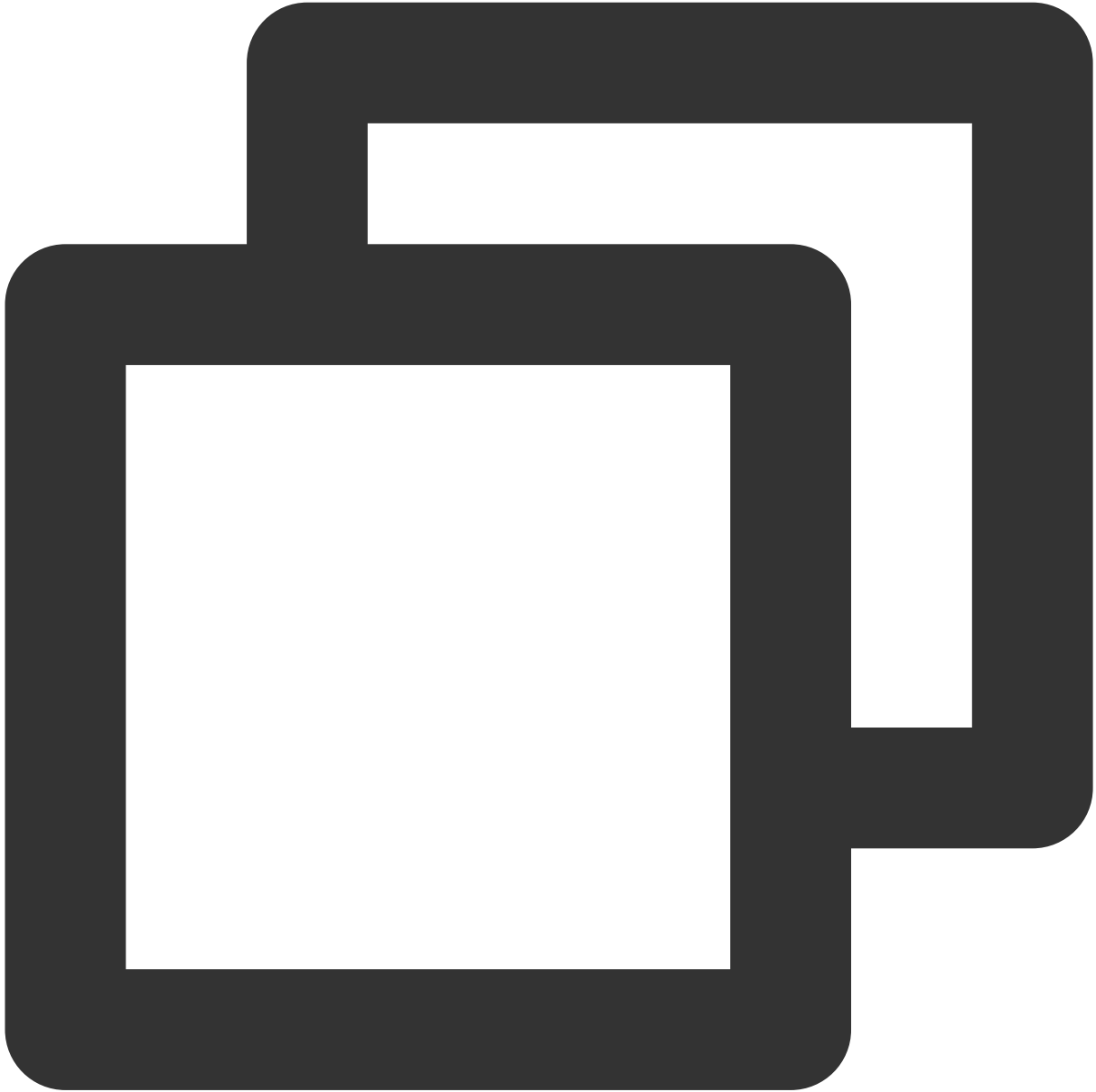
可写端缓冲区大小，以字节为单位，默认值为 32K，最大值为 256K，超过最大值则会自动调整为 256K。

## 方法

**注意：**

使用下述所有方法，要求当前流处于非锁定状态，否则会抛出异常。

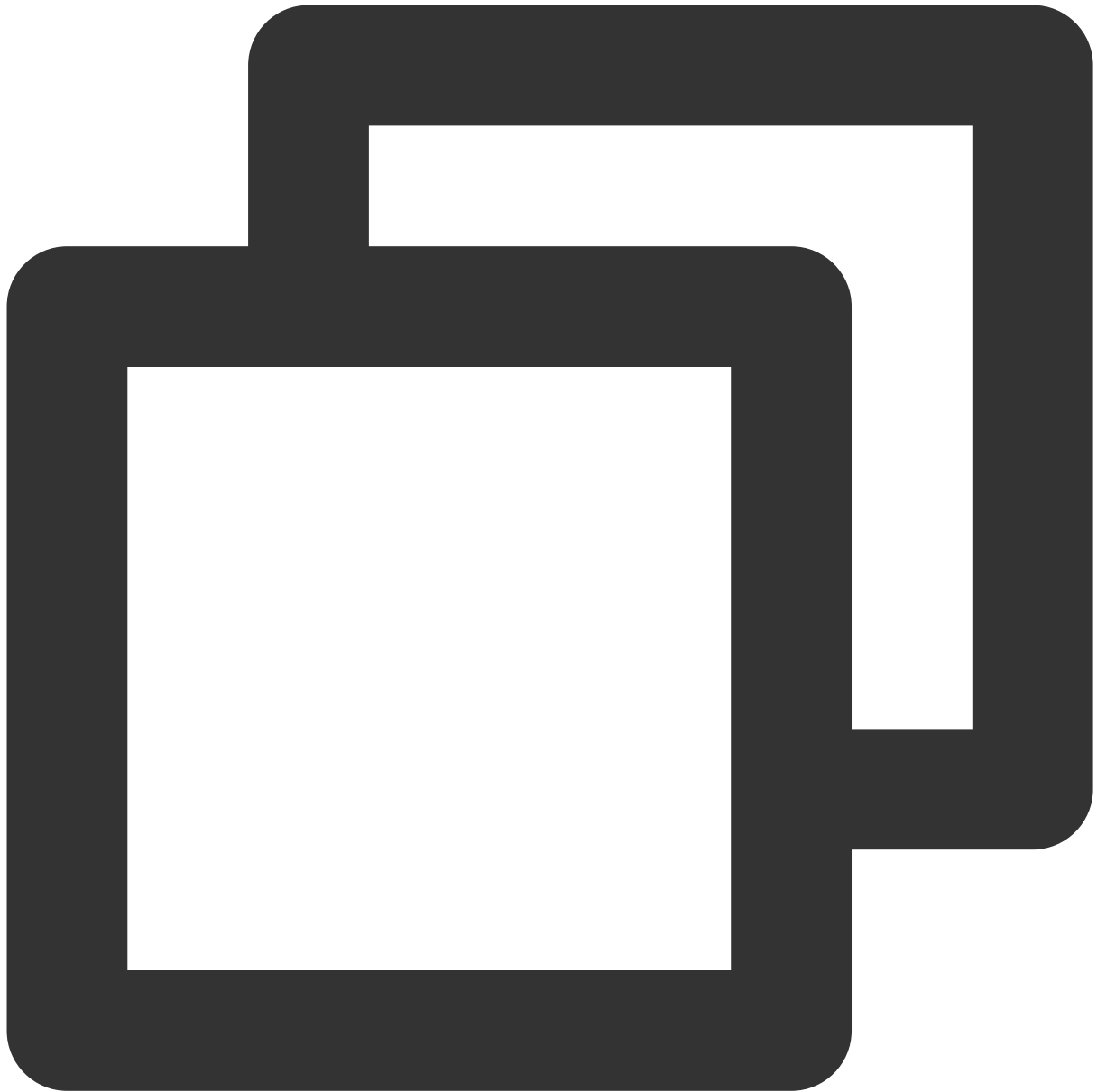
## getWriter



```
writable.getWriter(): WritableStreamDefaultWriter;
```

创建一个 `writer`，并锁定当前流，直至 `writer` 调用 `releaseLock()` 方法释放锁。返回值参见 [WritableStreamDefaultWriter](#)。

## close



```
writable.close(): Promise<void>;
```

关闭当前流。

## abort



```
writable.abort(reason?: string): Promise<string>;
```

中止当前流。

## 相关参考

[MDN 官方文档：WritableStream](#)

[示例函数：合并资源流式响应](#)

---

示例函数：[m3u8 改写与鉴权](#)

# WritableStreamDefaultWriter

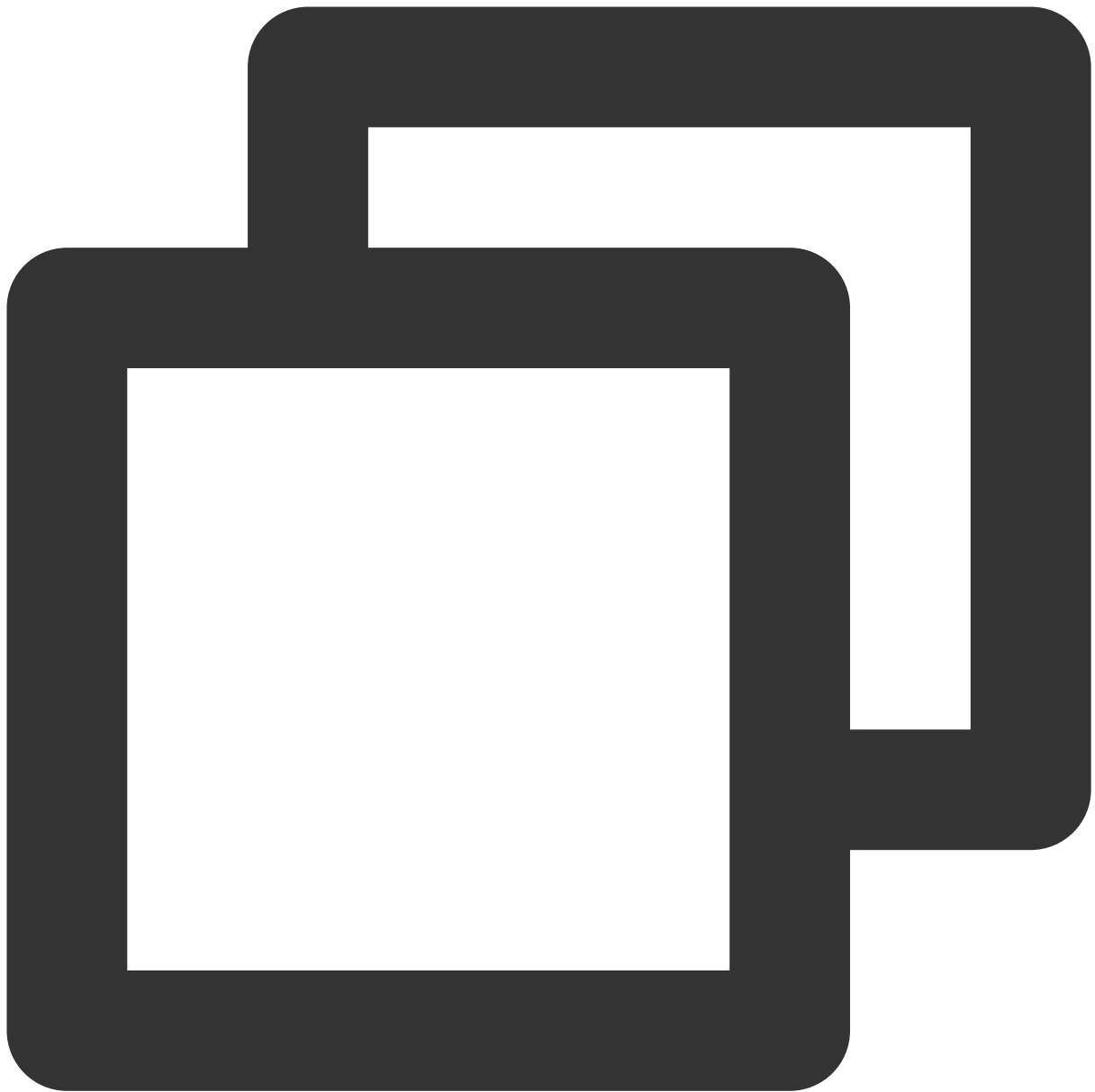
最近更新时间：2024-01-30 16:01:56

**WritableStreamDefaultWriter** 用于可写流的操作。基于 Web APIs 标准 [WritableStreamDefaultWriter](#) 进行设计。

## 注意：

不支持直接构造 `WritableStreamDefaultWriter` 对象，使用 [WritableStream.getWriter](#) 方法得到。

## 描述



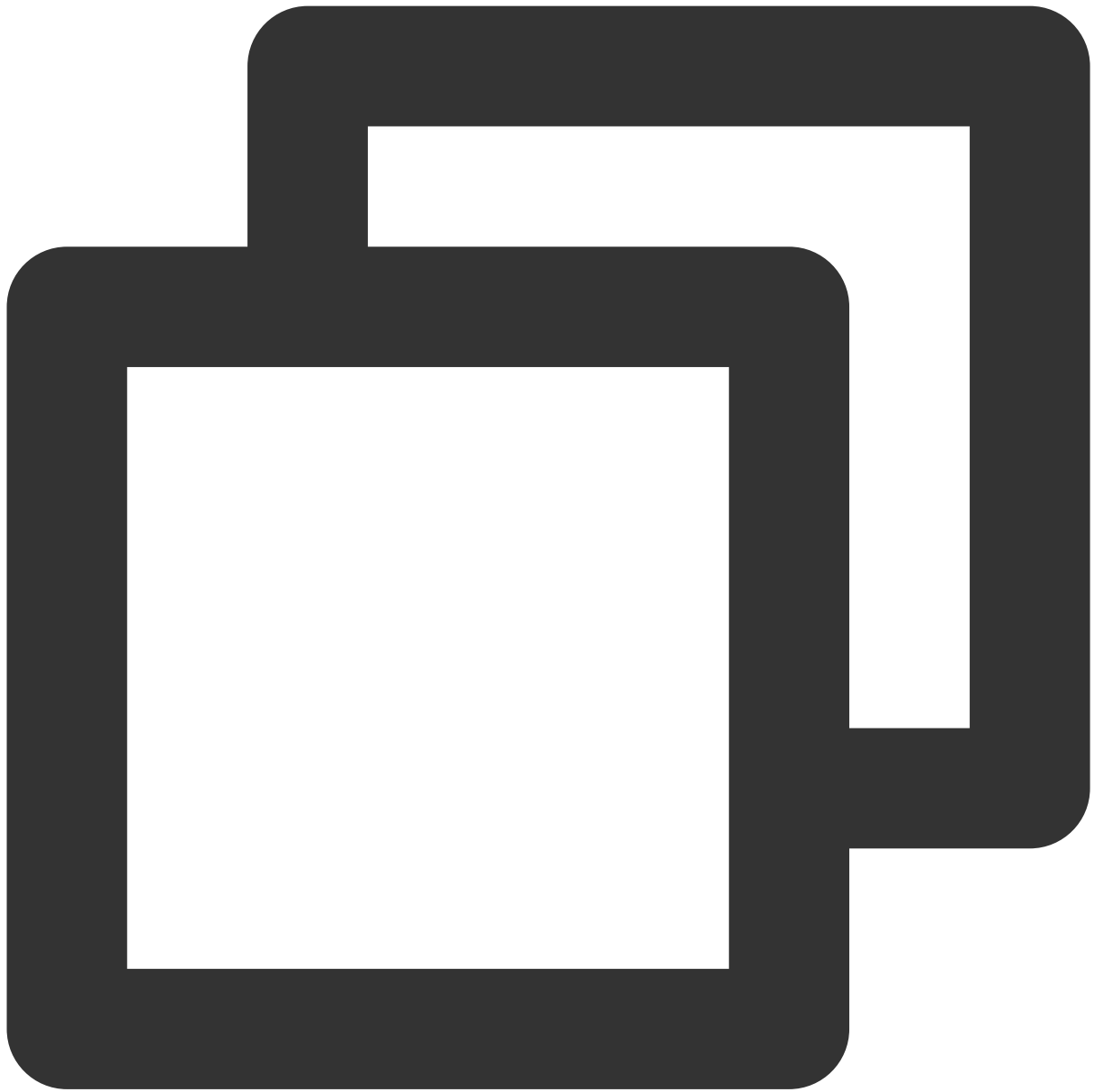
```
// 使用 TransformStream 构造得到 WritableStream 对象
const { writable } = new TransformStream();

// 使用 WritableStream 对象获取 writer
const writer = writable.getWriter();
```

## 属性



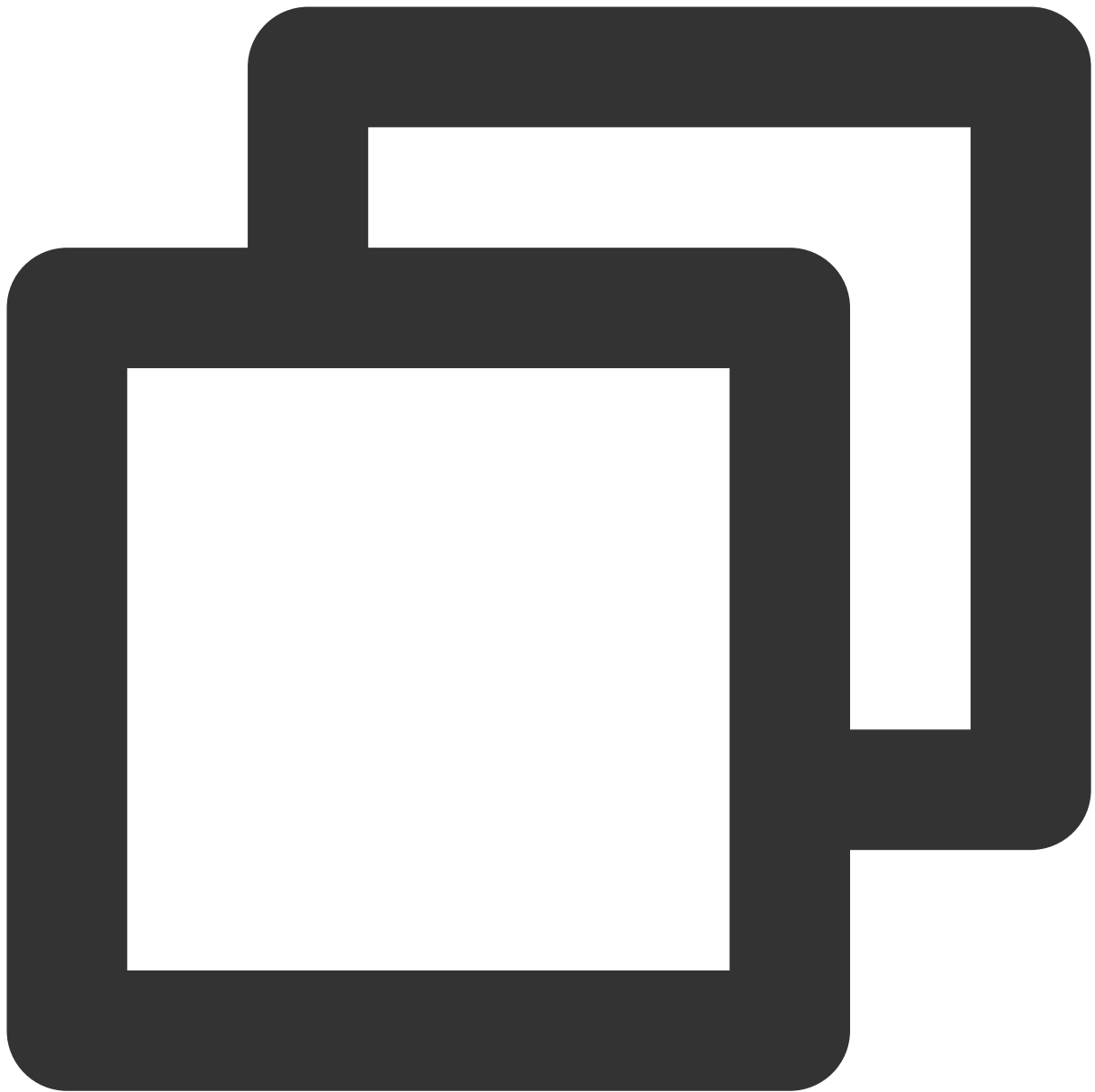
closed



```
// writer.closed  
readonly closed: Promise<void>;
```

返回 **Promise** 对象，如果流已关闭，**Promise** 状态为 `fulfilled`，如果流发生错误或写端锁已释放，**Promise** 状态为 `rejected`。

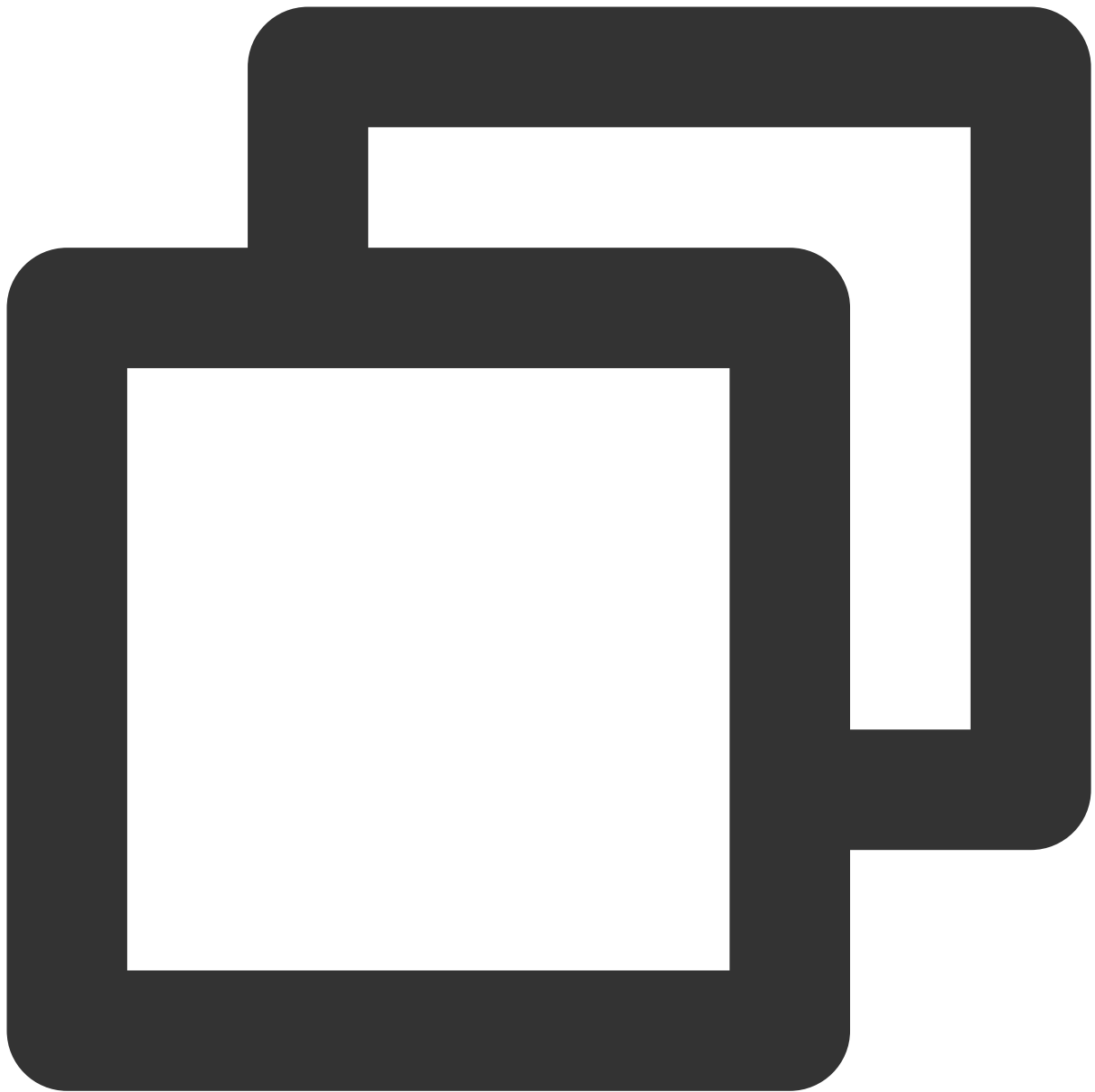
**ready**



```
// writer.ready  
readonly ready: Promise<void>;
```

返回 Promise 对象，当流的内部队列的所需大小从负变为正时，该 Promise 状态为 fulfilled，表示它不再施加背压。

## desiredSize

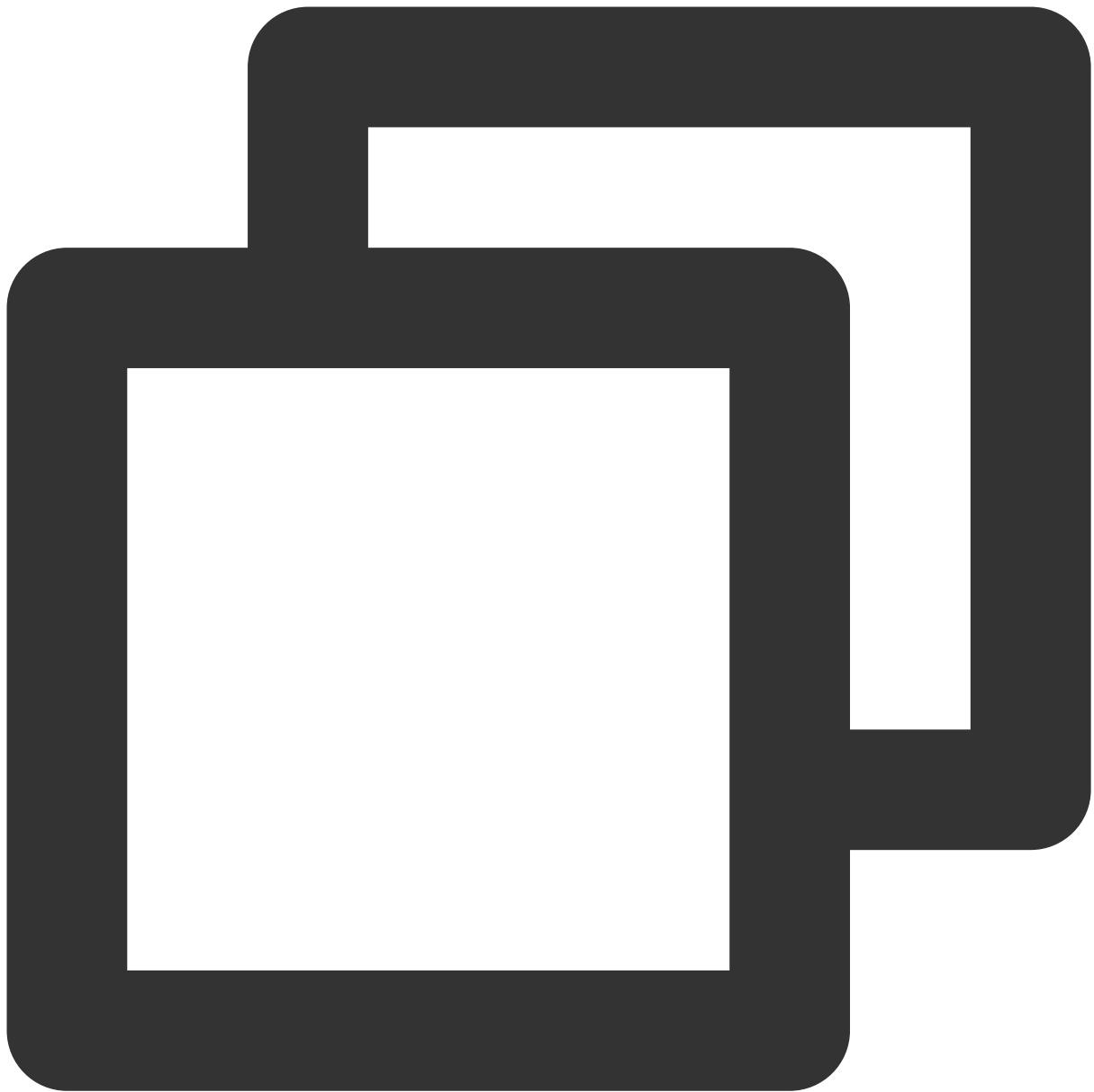


```
// writer.desiredSize  
readonly desiredSize: number;
```

返回填充流的内部队列所需的大小。

## 方法

**write**



```
writer.write(chunk: Chunk): Promise<void>;
```

把 `chunk` 数据写入流中。

#### 注意：

不允许前一个写流操作结束前，调用 `write` 方法发起下一个写流操作。

#### 参数

参数名称	类型	必填	说明

chunk

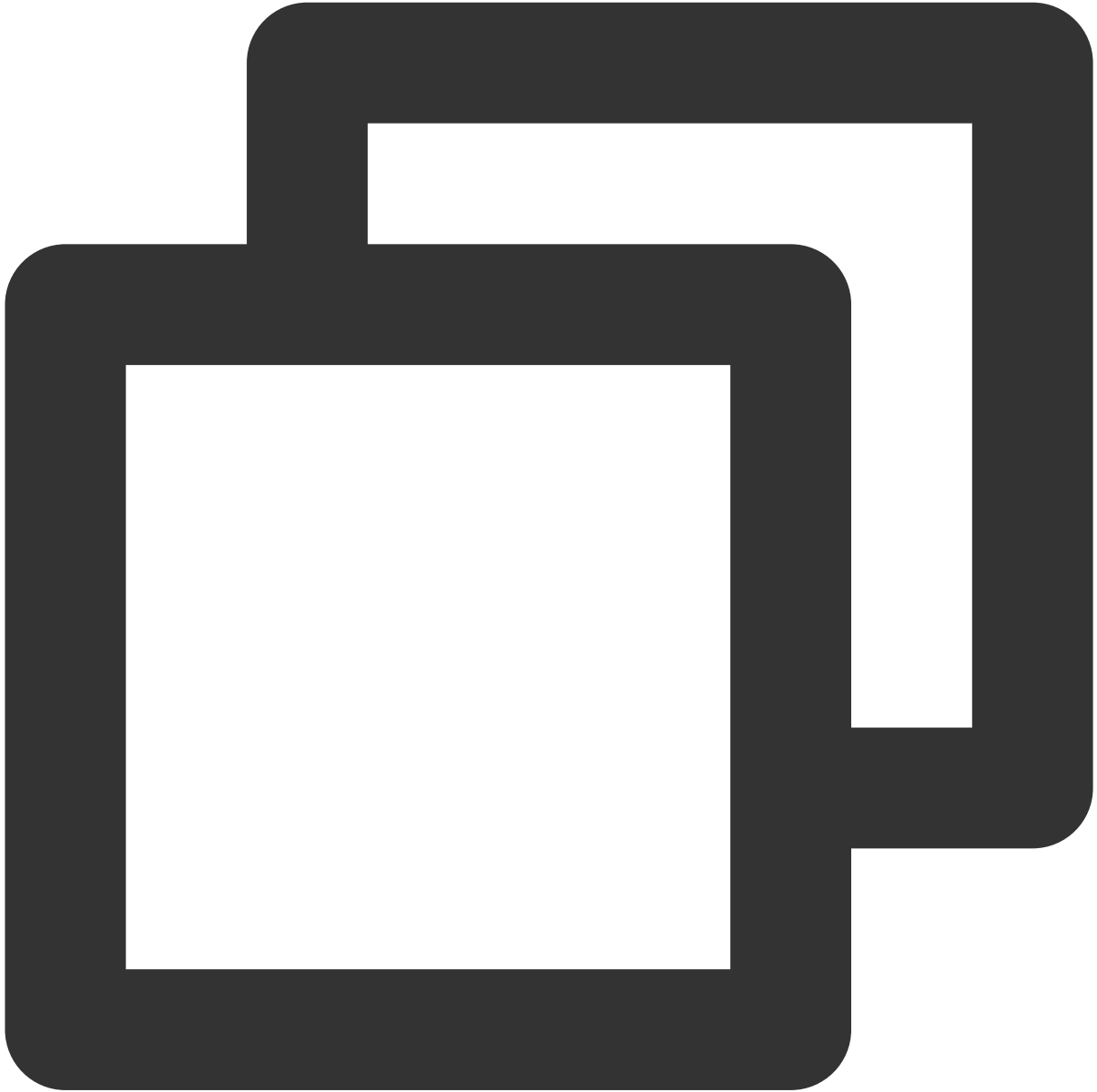
Chunk

是

待写入流的 chunk 数据。

## Chunk

从流中读取的数据 `Chunk`，描述如下：



```
type Chunk = string | ArrayBuffer | ArrayBufferView;
```

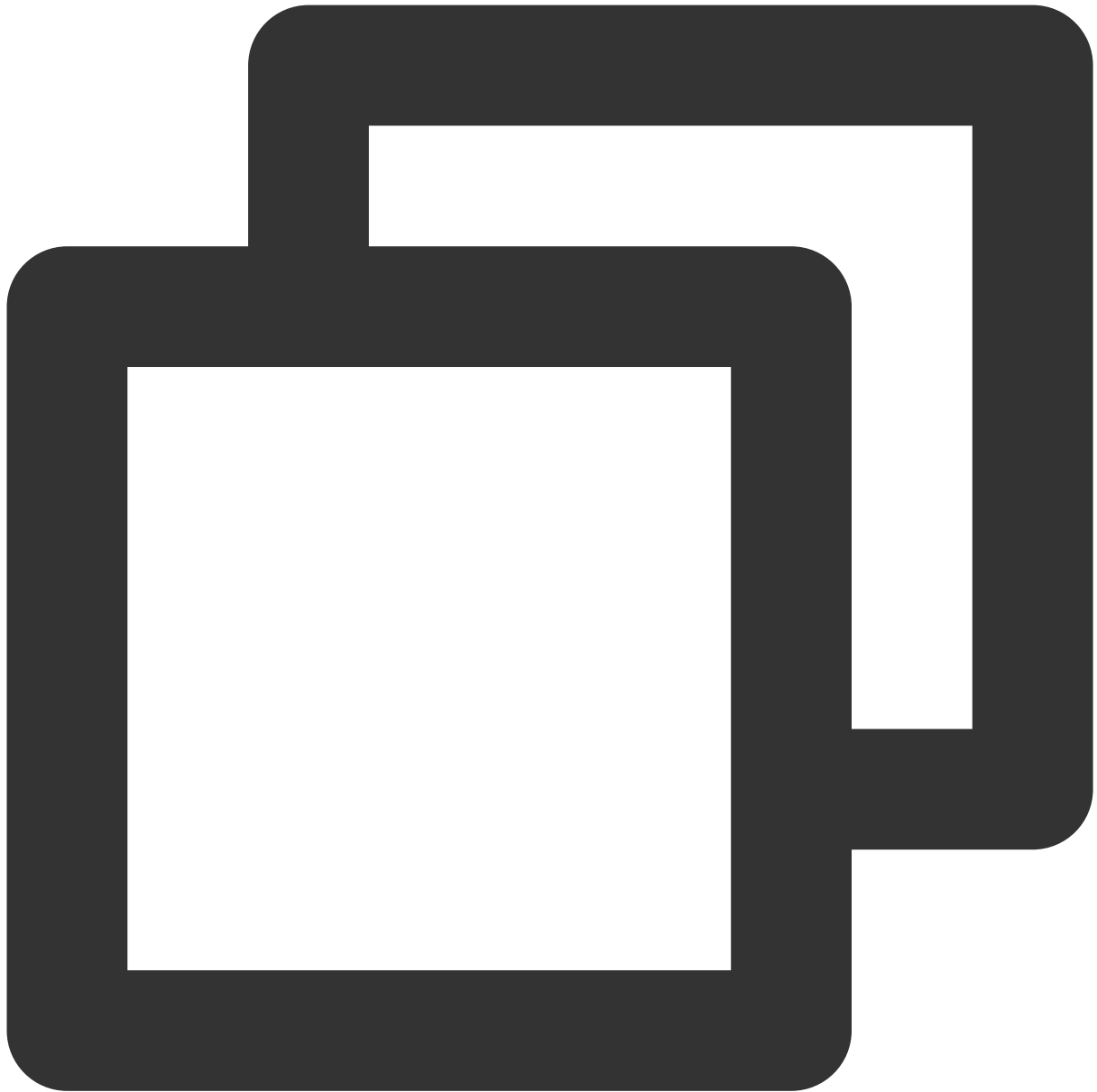
## close



```
writer.close(): Promise<void>;
```

关闭当前流。

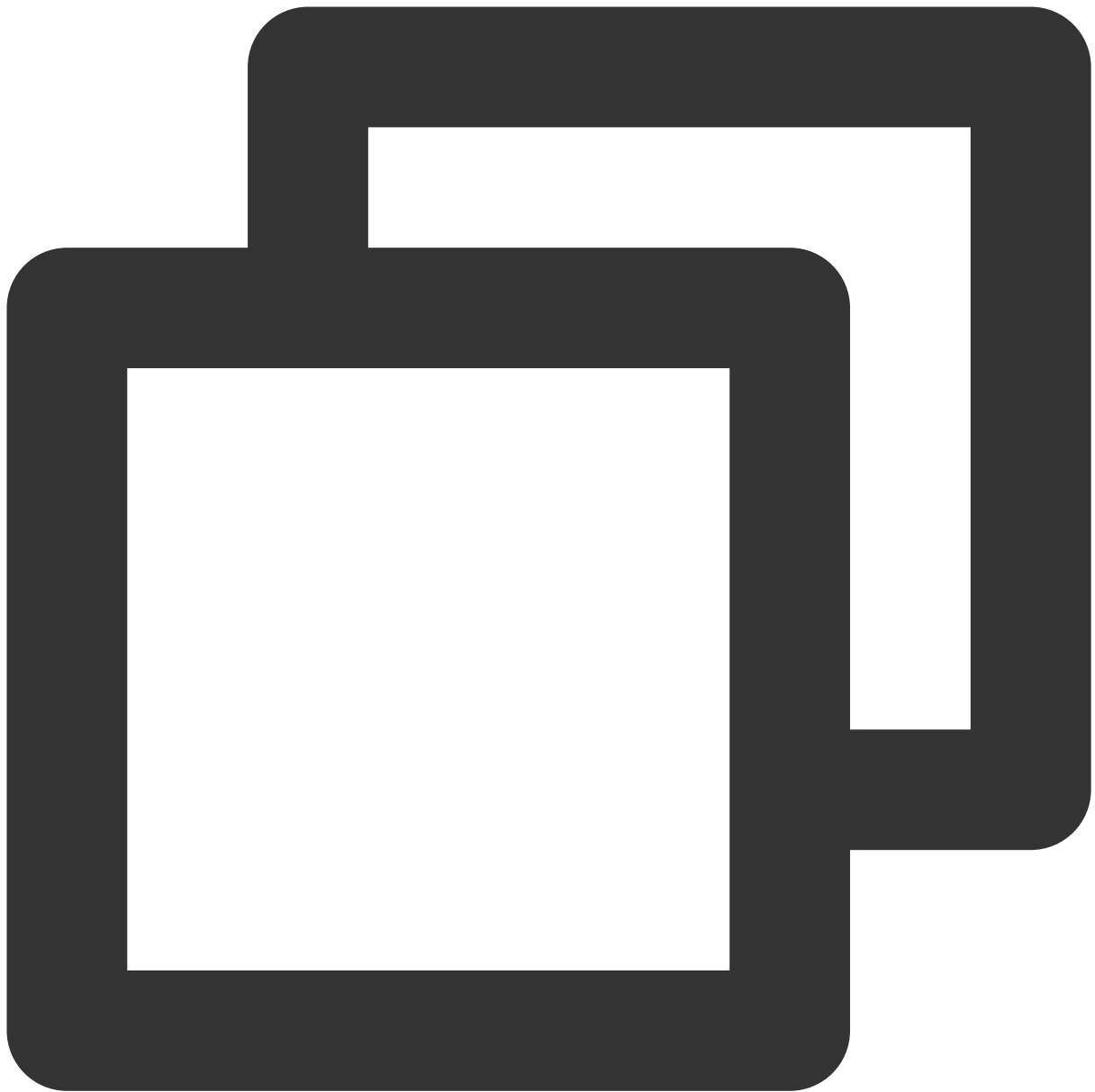
## **abort**



```
writer.abort(reason?: string): Promise<string>;
```

终止当前流。

**releaseLock**



```
writer.releaseLock(): void;
```

取消与流的关联，并释放流的锁定。

## 相关参考

[MDN 官方文档：WritableStreamDefaultWriter](#)



# Web Crypto

最近更新时间：2023-02-13 16:32:14

**Web Crypto API** 基于 Web APIs 标准 [Web Crypto API](#) 进行设计。提供了一组常见的加密操作接口，相比纯 JavaScript 实现的加密接口，`Web Crypto API` 的性能更高。

注意：

不支持直接构造 `Crypto` 对象，边缘函数运行时会在全局注入，直接使用全局 `crypto` 实例即可。

## 描述

```
// 编码
const encodeContent = new TextEncoder().encode('hello world');
// 使用 crypto, 生成 SHA-256 哈希值 Promise<ArrayBuffer>
const sha256Content = await crypto.subtle.digest(
  { name: 'SHA-256' },
  encodeContent
);
const result = new Uint8Array(sha256Content);
```

## 属性

```
// crypto.subtle
readonly subtle: SubtleCrypto;
```

提供常见的加密操作，例如：哈希、签名/验签、加解密等，详情参见 [SubtleCrypto](#)。

## 方法

### getRandomValues

```
crypto.getRandomValues(buffer: TypedArray): TypedArray;
```

生成随机数填充 `buffer`，并返回 `buffer`。

## 参数

属性名	类型	必填	说明
buffer	<a href="#">Int8Array</a>   <a href="#">Uint8Array</a>   <a href="#">Uint8ClampedArray</a>   <a href="#">Int16Array</a>   <a href="#">Uint16Array</a>   <a href="#">Int32Array</a>   <a href="#">Uint32Array</a>   <a href="#">BigInt64Array</a>   <a href="#">BigUint64Array</a>	是	随机数缓冲区，不超过 65536 字节。详情参见 <a href="#">TypedArray</a> 。

## randomUUID

```
crypto.randomUUID(): string;
```

返回随机 UUID(v4)。

## SubtleCrypto

提供常见的加密操作，例如：哈希、签名/验签、加解密等，详情参见 [MDN 官方文档：SubtleCrypto](#)。

说明：

**SubtleCrypto** 加密接口按功能分为两类：

- **加密功能**，包含 `encrypt/decrypt`、`sign/verify`、`digest`，可以用来实现隐私和身份验证等安全功能。
- **密钥管理功能**，包含 `generateKey`、`deriveKey`、`importKey/exportKey`，可以用来管理密钥。

## digest

```
crypto.subtle.digest(algorithm: string | object, data: ArrayBuffer): Promise<ArrayBuffer>;
```

返回 Promise 对象，包含生成的数据摘要（hash），详情参见 [MDN 官方文档：SubtleCrypto.digest](#)。

## encrypt

```
crypto.subtle.encrypt(algorithm: object, key: CryptoKey, data: ArrayBuffer): Promise<ArrayBuffer>;
```

返回 Promise 对象，包含加密数据，详情参见 [MDN 官方文档：SubtleCrypto.encrypt](#)。

## decrypt

```
crypto.subtle.decrypt(algorithm: object, key: CryptoKey, data: ArrayBuffer): Promise<ArrayBuffer>;
```

返回 Promise 对象，包含解密数据，详情参见 [MDN 官方文档：SubtleCrypto.decrypt](#)。

## sign

```
crypto.subtle.sign(algorithm: string | object, key: CryptoKey, data: ArrayBuffer): Promise<ArrayBuffer>;
```

返回 Promise 对象，包含数据签名，详情参见 [MDN 官方文档：SubtleCrypto.sign](#)。

## verify

```
crypto.subtle.verify(algorithm: string | object, key: CryptoKey, signature: BufferSource, data: ArrayBuffer): Promise<boolean>;
```

返回 Promise 对象，包含签名验证结果，详情参见 [MDN 官方文档：SubtleCrypto.verify](#)。

## generateKey

```
crypto.subtle.generateKey(algorithm: object, extractable: boolean, keyUsages: Array<string>): Promise<CryptoKey | CryptoKeyPair>;
```

返回 Promise 对象，包含密钥 `CryptoKey` 或密钥对 `CryptoKeyPair`，详情参见 [MDN 官方文档：SubtleCrypto.generateKey](#)。

## deriveKey

```
crypto.subtle.deriveKey(algorithm: object, baseKey: CryptoKey, derivedKeyAlgorithm: object, extractable: boolean, keyUsages: Array<string>): Promise<CryptoKey>;
```

返回 Promise 对象，包含密钥 `CryptoKey`，详情参见 [MDN 官方文档：SubtleCrypto.deriveKey](#)。

## importKey

```
crypto.subtle.importKey(format: string, keyData: BufferSource, algorithm: string | object, extractable: boolean, keyUsages: Array<string>): Promise<CryptoKey>;
```

返回 Promise 对象，包含密钥 CryptoKey，详情参见 [MDN 官方文档：SubtleCrypto.importKey](#)。

## exportKey

```
crypto.subtle.exportKey(format: string, key: CryptoKey): Promise<ArrayBuffer>;
```

返回 Promise 对象，包含导出密钥 ArrayBuffer，详情参见 [MDN 官方文档：SubtleCrypto.exportKey](#)。

## deriveBits

```
crypto.subtle.deriveBits(algorithm: object, baseKey: CryptoKey, length: integer): Promise<ArrayBuffer>;
```

返回 Promise 对象，包含伪随机字节 ArrayBuffer，详情参见 [MDN 官方文档：SubtleCrypto.deriveBits](#)。

## wrapKey

```
crypto.subtle.wrapKey(format: string, key: CryptoKey, wrappingKey: CryptoKey, wrapAlgo: string | object): Promise<ArrayBuffer>;;
```

返回 Promise 对象，包含封装密钥 ArrayBuffer，详情参见 [MDN 官方文档：SubtleCrypto.wrapKey](#)。

## unwrapKey

```
crypto.subtle.unwrapKey(format: string, wrappedKey: ArrayBuffer, unwrappingKey: CryptoKey, unwrapAlgo: string | object, unwrappedKeyAlgo: string | object, extractable: boolean, keyUsages: Array<string>): Promise<CryptoKey>;
```

返回 Promise 对象，包含解封密钥 CryptoKey，详情参见 [MDN 官方文档：SubtleCrypto.unwrapKey](#)。

# CryptoKey

`CryptoKey` 表示用加密算法生成的密钥，详细参见 [MDN 官方文档 CryptoKey](#)。不支持直接构造 `CryptoKey` 对象，使用下述接口生成密钥：

- [crypto.subtle.generateKey](#)
- [crypto.subtle.importKey](#)
- [crypto.subtle.deriveKey](#)
- [crypto.subtle.unwrapKey](#)

`CryptoKey` 属性描述如下。

属性名	类型	只读	说明
<code>type</code>	<code>string</code>	是	密钥类型。
<code>extractable</code>	<code>boolean</code>	是	密钥是否可导出。
<code>algorithm</code>	<code>object</code>	是	算法相关, 包含算法需要的字段。
<code>usages</code>	<code>Array&lt;string&gt;</code>	是	密钥的用途。

## CryptoKeyPair

`CryptoKeyPair` 表示用加密算法生成的密钥对, 详细参见 [MDN 官方文档 : CryptoKeyPair](#)。不支持直接构造 `CryptoKeyPair` 对象, 使用下述接口生成密钥对:

- [crypto.subtle.generateKey](#)

`CryptoKeyPair` 属性描述如下。

属性名	类型	只读	说明
<code>privateKey</code>	<a href="#">CryptoKey</a>	是	对于加解密算法, 私钥用于解密。对于签名算法, 私钥用于签名。
<code>publicKey</code>	<a href="#">CryptoKey</a>	是	对于加解密算法, 公钥用于加密。对于签名算法, 公钥用于验签。

## 支持算法

边缘函数支持 Web APIs 标准 [WebCrypto](#) 定义的所有算法, 详细如下表所示。

Algorithm	encrypt() decrypt()	sign() verify()	wrapKey() unwrapKey()	deriveKey() deriveBits()	generateKey()	importKey()	exportKey()
RSASSA-PKCS1-v1_5	-	✓	-	-	✓	✓	✓
RSA-PSS	-	✓	-	-	✓	✓	✓
RSA-OAEP	✓	-	✓	-	✓	✓	✓

Igorithm	encrypt() decrypt()	sign() verify()	wrapKey() unwrapKey()	deriveKey() deriveBits()	generateKey()	importKey()	exportKey()
ECDSA	-	✓	-	-	✓	✓	✓
ECDH	-	-	-	✓	✓	✓	✓
HMAC	-	✓	-	-	✓	✓	✓
AES-CTR	✓	-	✓	-	✓	✓	✓
AES-CBC	✓	-	✓	-	✓	✓	✓
AES-GCM	✓	-	✓	-	✓	✓	✓
AES-KW	-	-	✓	-	✓	✓	✓
HKDF	-	-	-	✓	-	✓	-
PBKDF2	-	-	-	✓	-	✓	-
SHA-1	-	-	-	-	-	-	-
SHA-256	-	-	-	-	-	-	-
SHA-384	-	-	-	-	-	-	-
SHA-512	-	-	-	-	-	-	-
MD5	-	-	-	-	-	-	-

## 示例代码

```
function uint8ArrayToHex(arr) {
    return Array.prototype.map.call(arr, (x) => ((`0${x.toString(16)}`).slice(-2))).join('');
}

async function handleEvent(event) {
    const encodeArr = TextEncoder().encode('hello world');
    // 执行 md5
    const md5Buffer = await crypto.subtle.digest({ name: 'MD5' }, encodeArr);
    // 输出十六进制字符串
    const md5Str = uint8ArrayToHex(new Uint8Array(md5Buffer));
}
```

```
const response = new Response(md5Str);
return response;
}

addEventListener('fetch', async (event) => {
  event.respondWith(handleEvent(event));
});
```

## 相关参考

- [MDN 官方文档：Web Crypto API](#)
- [MDN 官方文档：SubtleCrypto](#)
- [MDN 官方文档：CryptoKey](#)
- [MDN 官方文档：CryptoKeyPair](#)
- [示例函数：防篡改校验](#)
- [示例函数：m3u8 改写与鉴权](#)
- [示例函数：缓存 POST 请求](#)

---

# Web standards

最近更新时间：2023-12-13 11:13:09

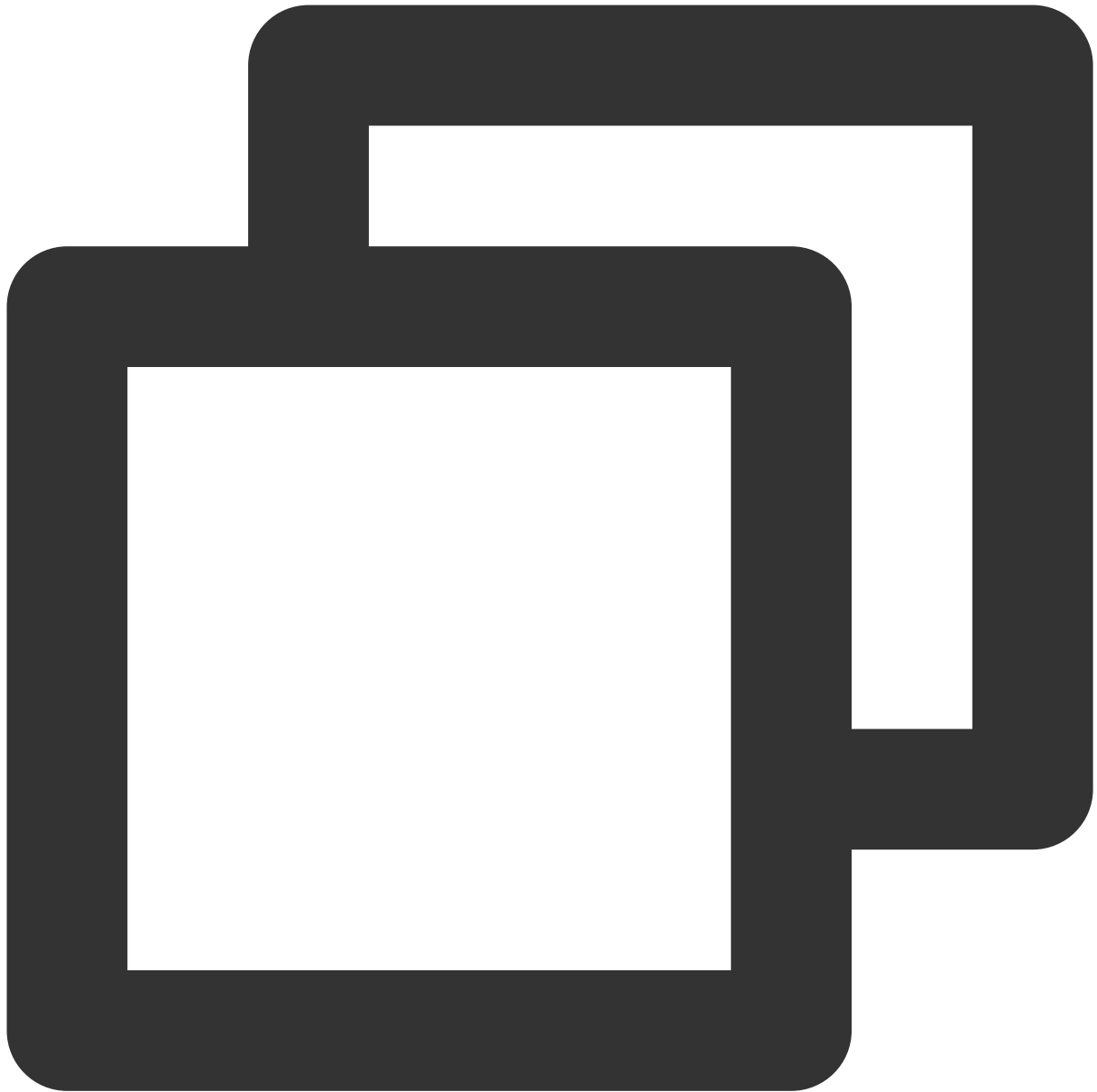
边缘函数基于 **V8 JavaScript 引擎** 设计实现的 Serverless 代码执行环境，提供了以下标准化的 Web APIs。

## JavaScript 标准内置对象

边缘函数支持所有 JavaScript 标准内置对象，详情参见 [MDN 官方文档：JavaScript Standard built-in objects](#)。

## URL

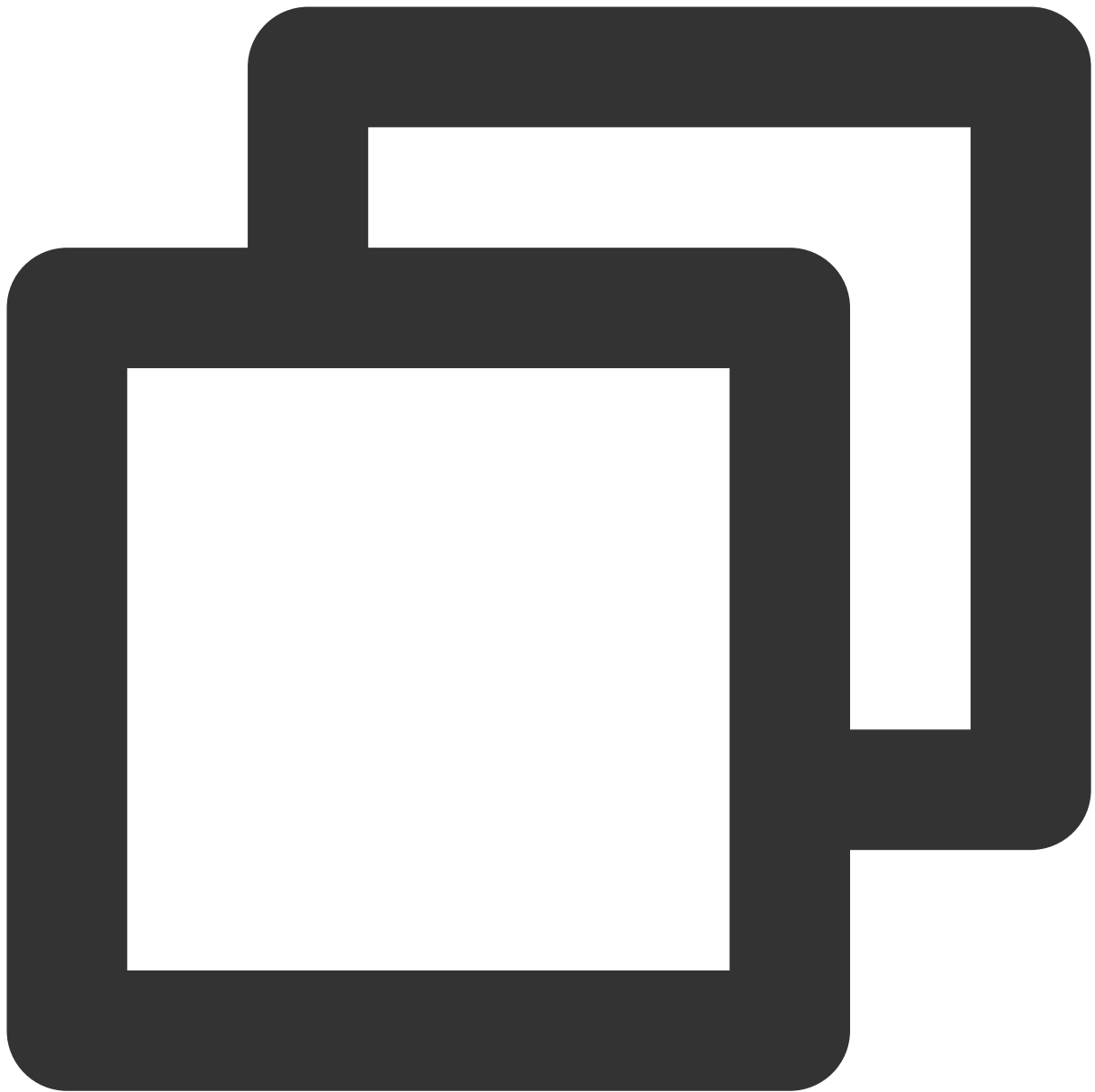




```
const urlInfo = new URL('https://www.tencentcloud.com/');
```

URL API 用于解析，构造，规范化和编码 URL，详情参见 [MDN 官方文档：URL](#)。

## Blob

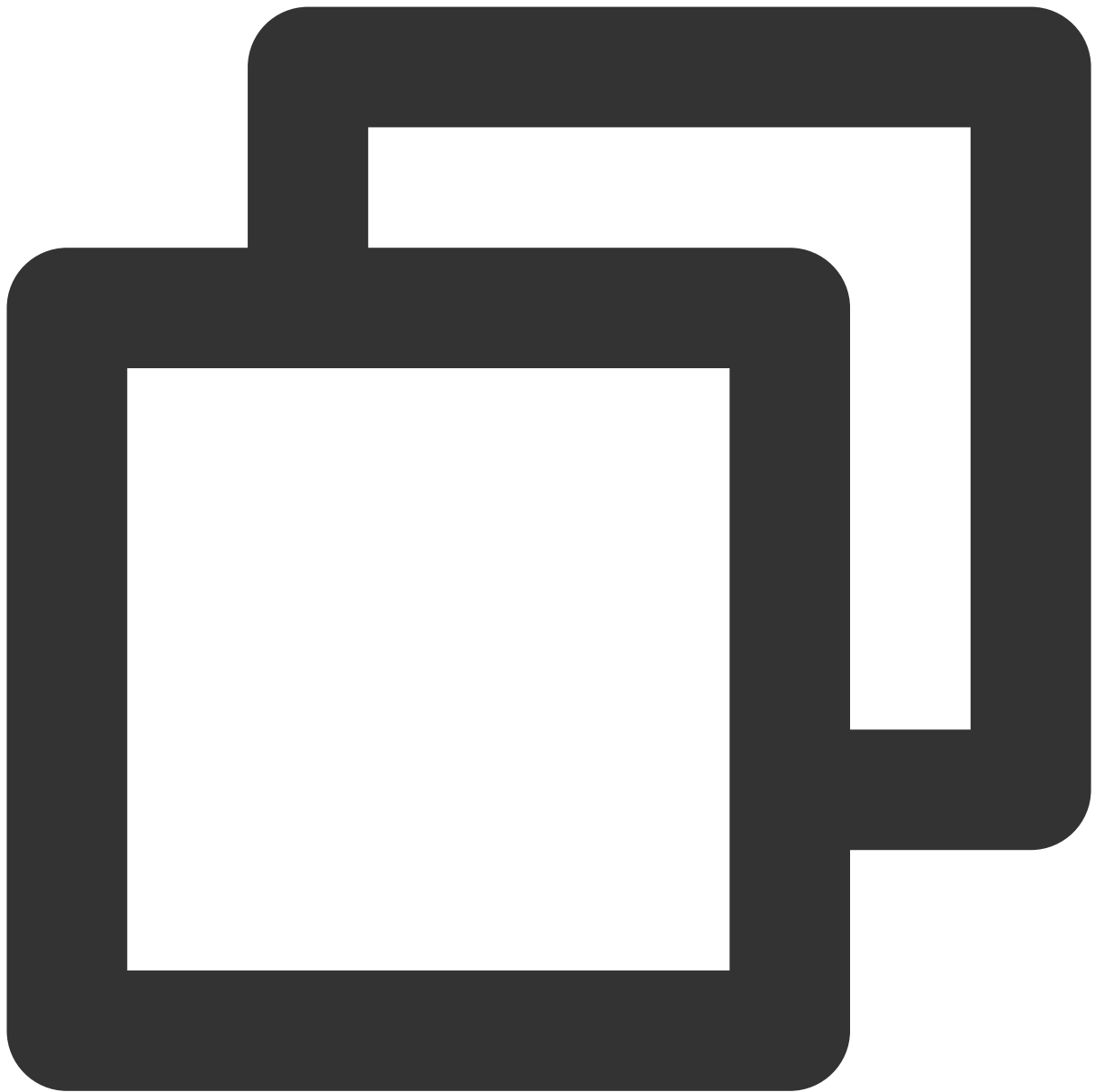


```
const blob = new Blob(['hello', 'world'], { type: 'text/plain' });
```

Blob API 表示不可变、原始数据的类文件对象，详情参见 [MDN 官方文档：Blob](#)。

## Base64

### btoa



```
function btoa(data: string | ArrayBuffer | ArrayBufferView): string;
```

执行 base64 编码，不支持 Unicode 字符串，详情参见 [MDN 官方文档 : btoa](#)。

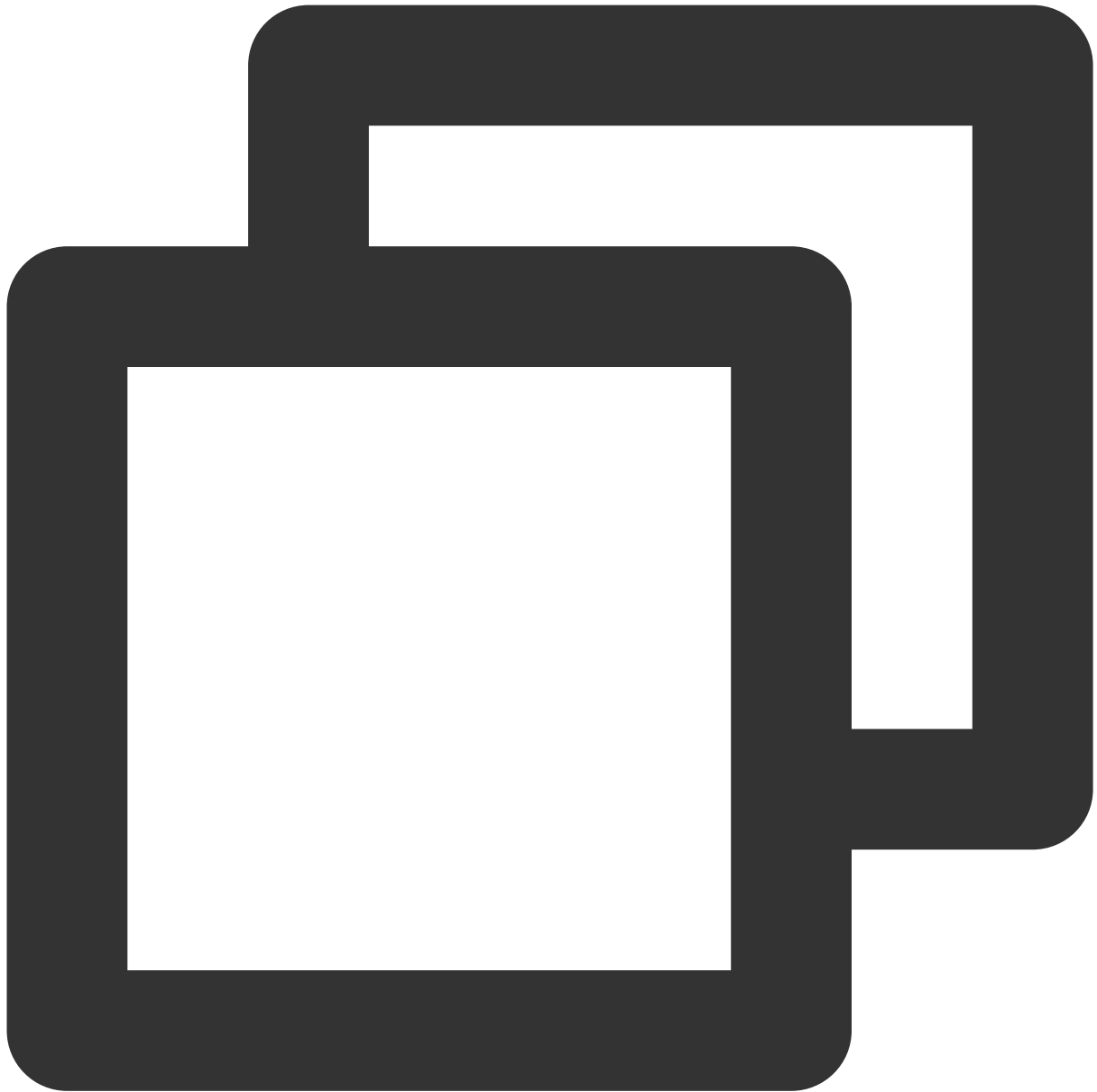
**atob**



```
function atob(data: string): string;
```

执行 base64 解码，不支持 Unicode 字符串，详情参见 [MDN 官方文档 : atob](#)。

## btoaUTF8



```
function btoaUTF8(data: string): string;
```

执行 base64 编码，支持 Unicode 字符串。

### **atobUTF8**



```
function atobUTF8(data: string): string;
```

执行 base64 解码，不支持 Unicode 字符串。

## 定时器

### **setTimeout**



```
setTimeout(func: function): number;  
setTimeout(func: function, delay: number): number;  
setTimeout(func: function, delay: number, ...args: any[]): number;
```

普通定时器，在定时器到期执行指定函数，详情参见 [MDN 官方文档：setTimeout](#)。

## clearTimeout

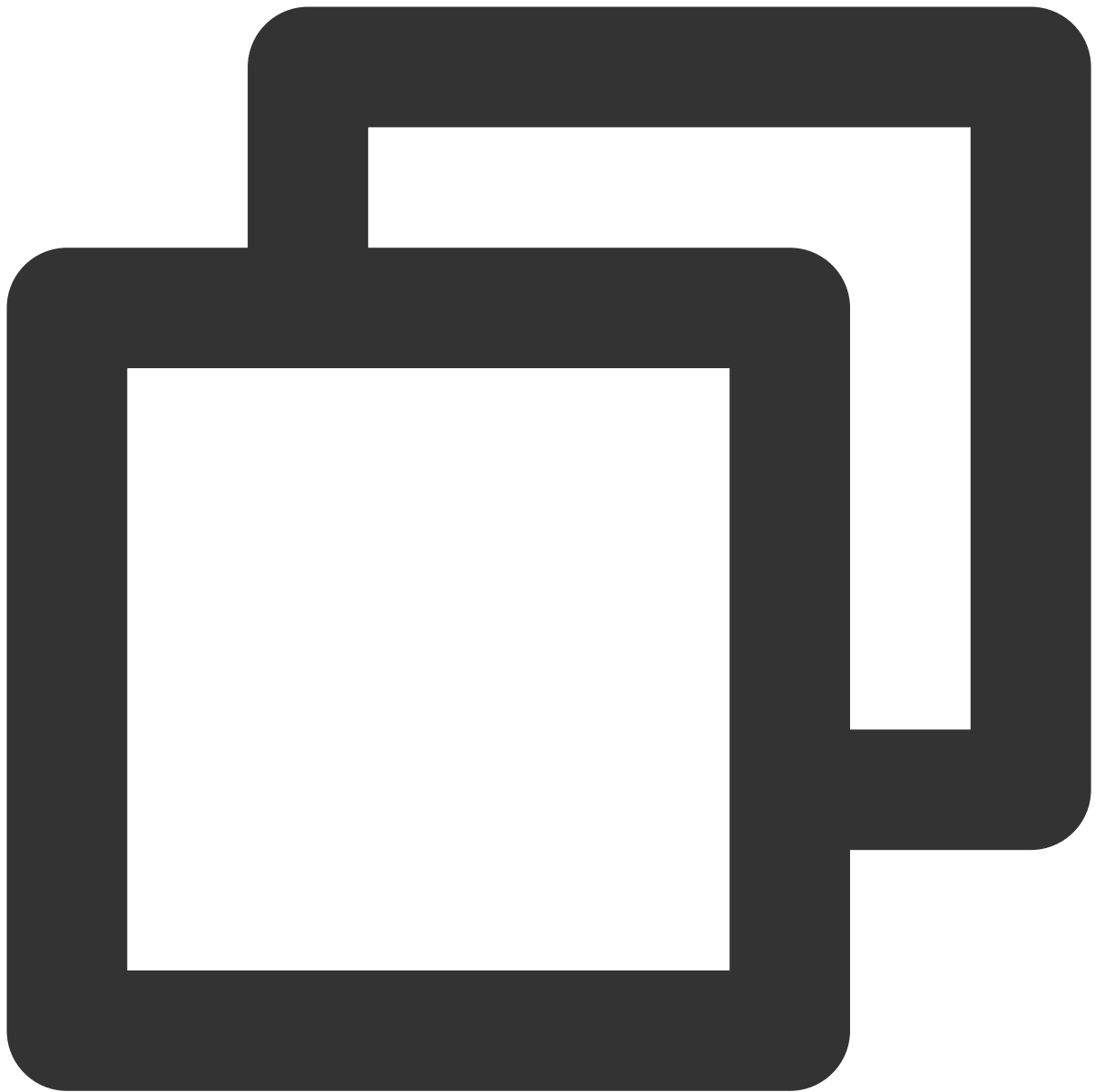


```
clearTimeout(timeoutID: number): void;
```

清除指定 `timeoutID` 的普通定时器，详情参见 [MDN 官方文档](#)：[clearTimeout](#)。

## setInterval

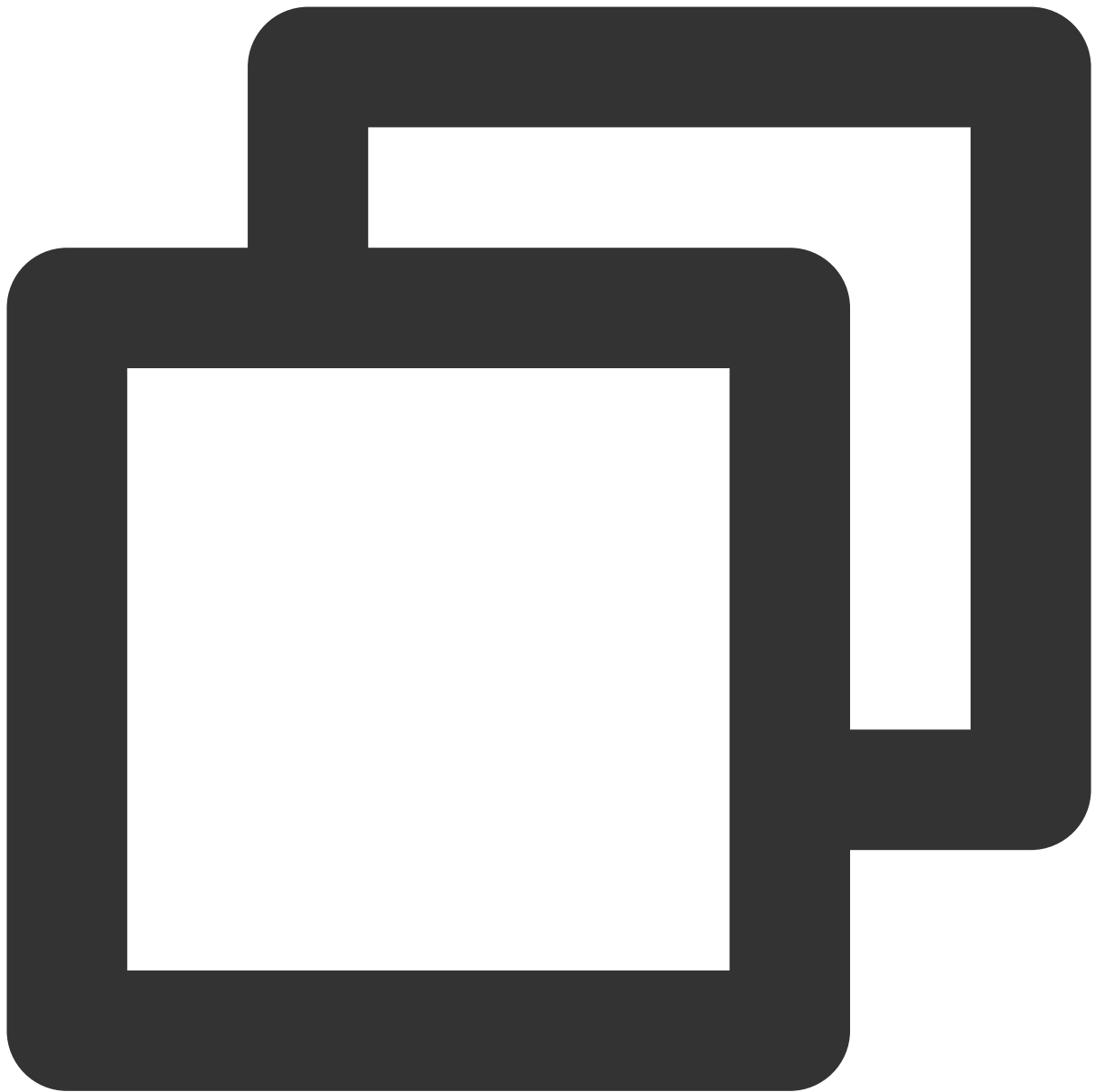




```
setInterval(func: function): number;  
setInterval(func: function, delay: number): number;  
setInterval(func: function, delay: number, ...args: any[]): number;
```

循环定时器, 每次定时器到期后执行指定函数, 详情参见 [MDN 官方文档: setInterval](#)。

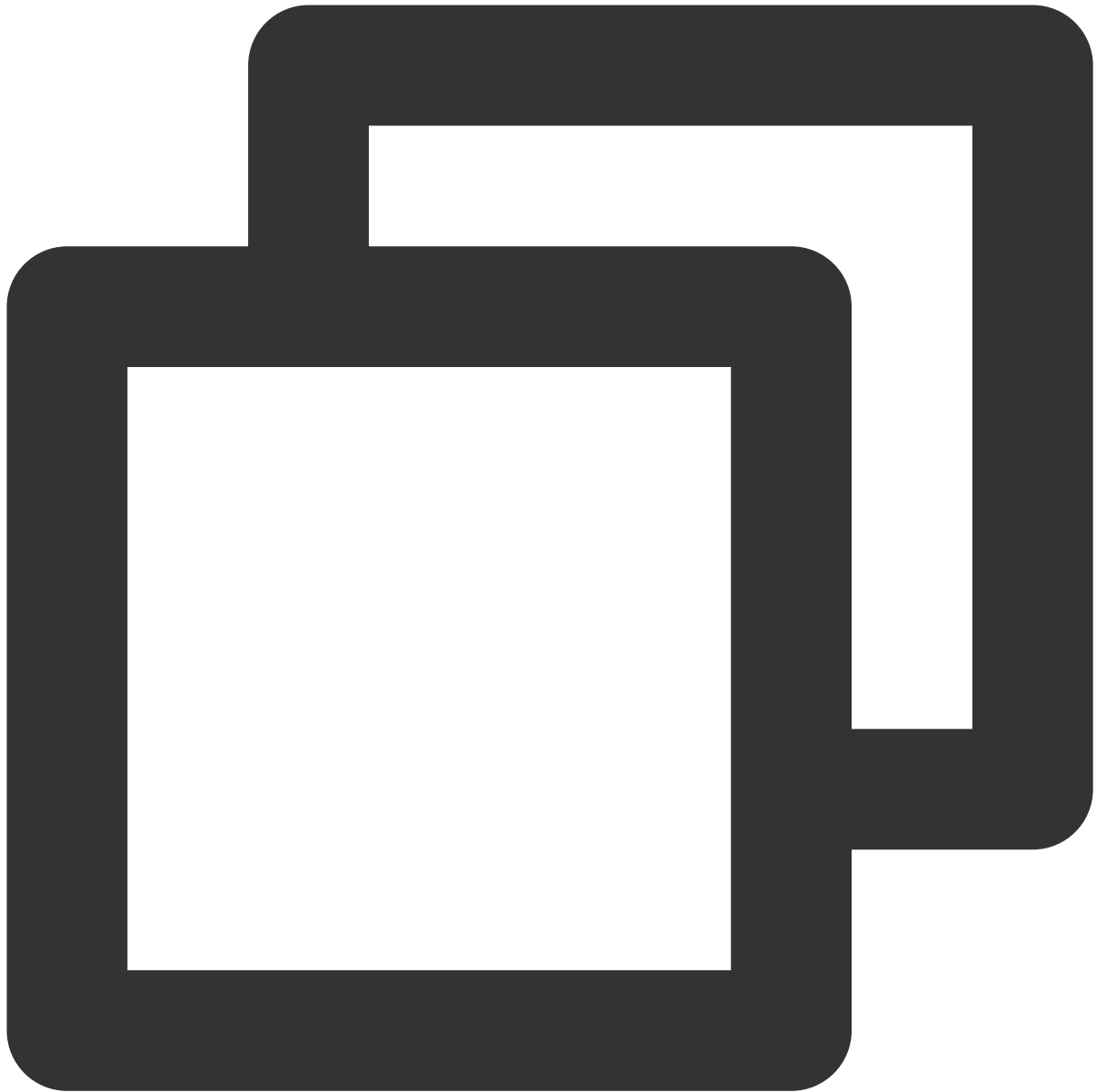
## clearInterval



```
clearInterval(intervalID: number): void;
```

清除一个循环定时器，详情参见 [MDN 官方文档：clearInterval](#)。

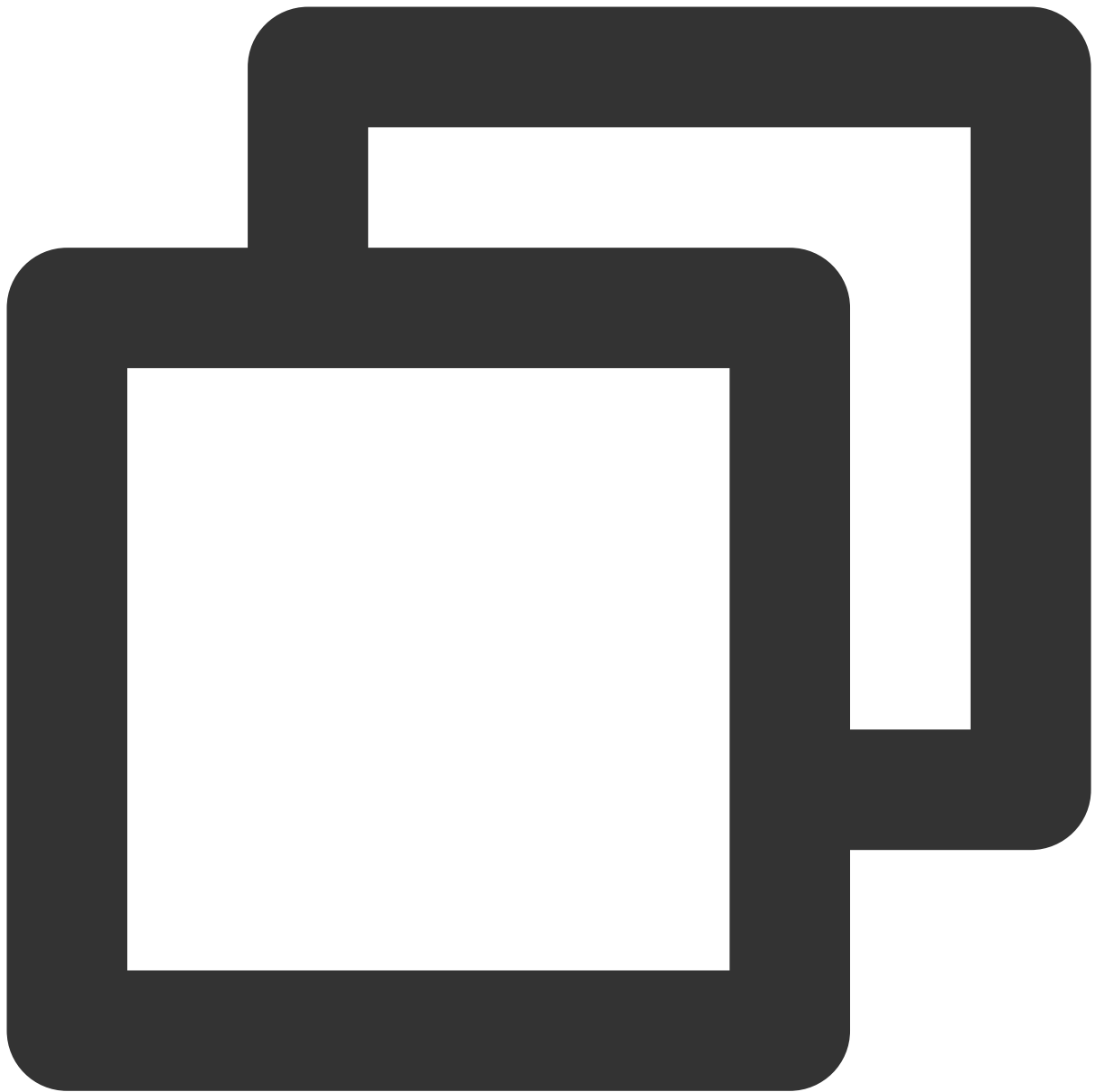
## setImmediate



```
setImmediate(func: function): number;  
setImmediate(func: function, ...args: any[]): number;
```

即时定时器, 在边缘函数栈清空之后执行指定函数, 详情参见 [MDN 官方文档: setImmediate](#)。

## clearImmediate

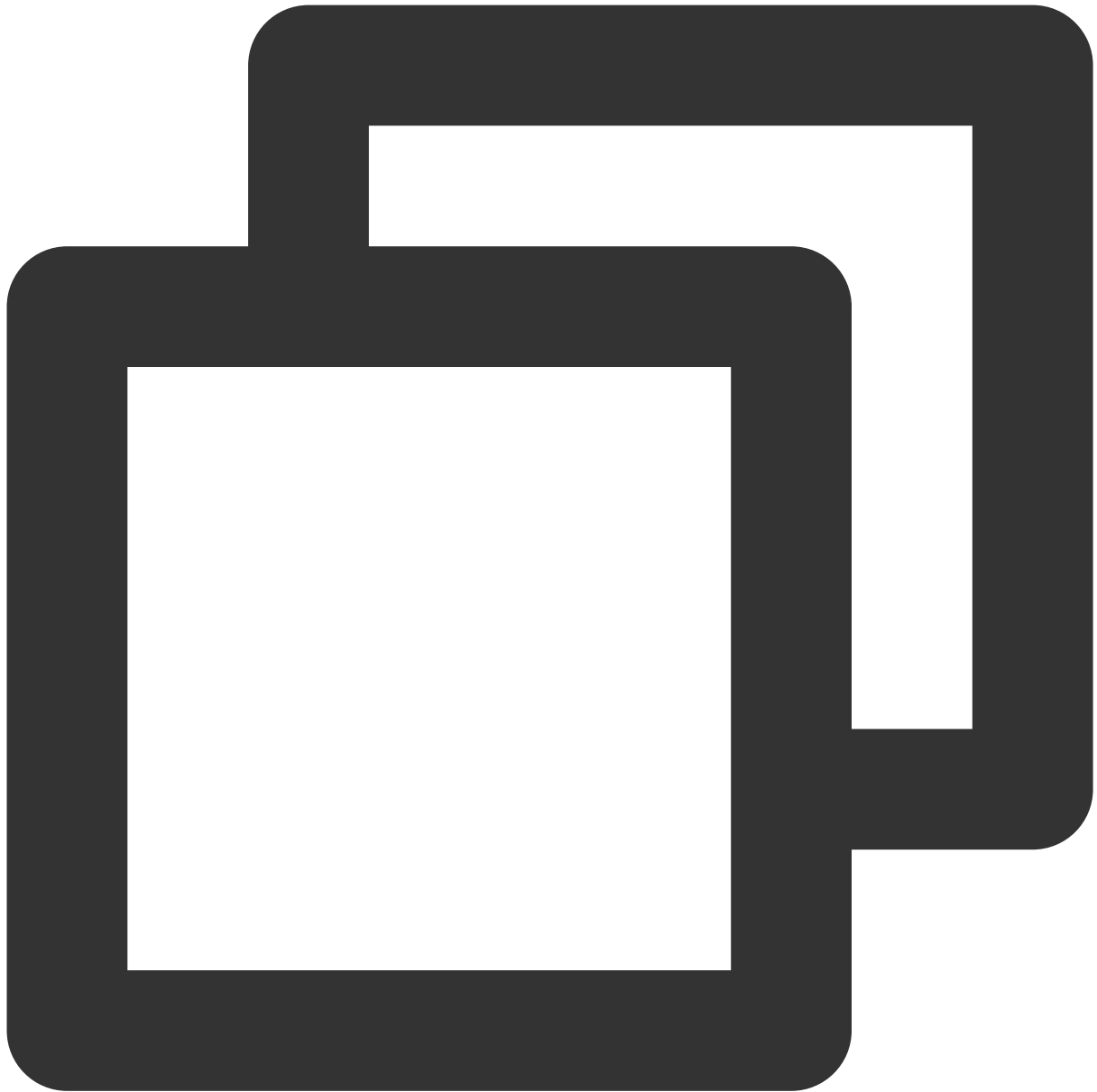


```
clearImmediate(immediateID: number): void;
```

清除一个即时定时器，详情参见 [MDK 官方文档：clearImmediate](#)。

## 事件发布与订阅

### EventTarget



```
const eventTarget = new EventTarget();
```

事件发布与订阅，详情参见 [MDN 官方文档：EventTarget](#)。

## Event

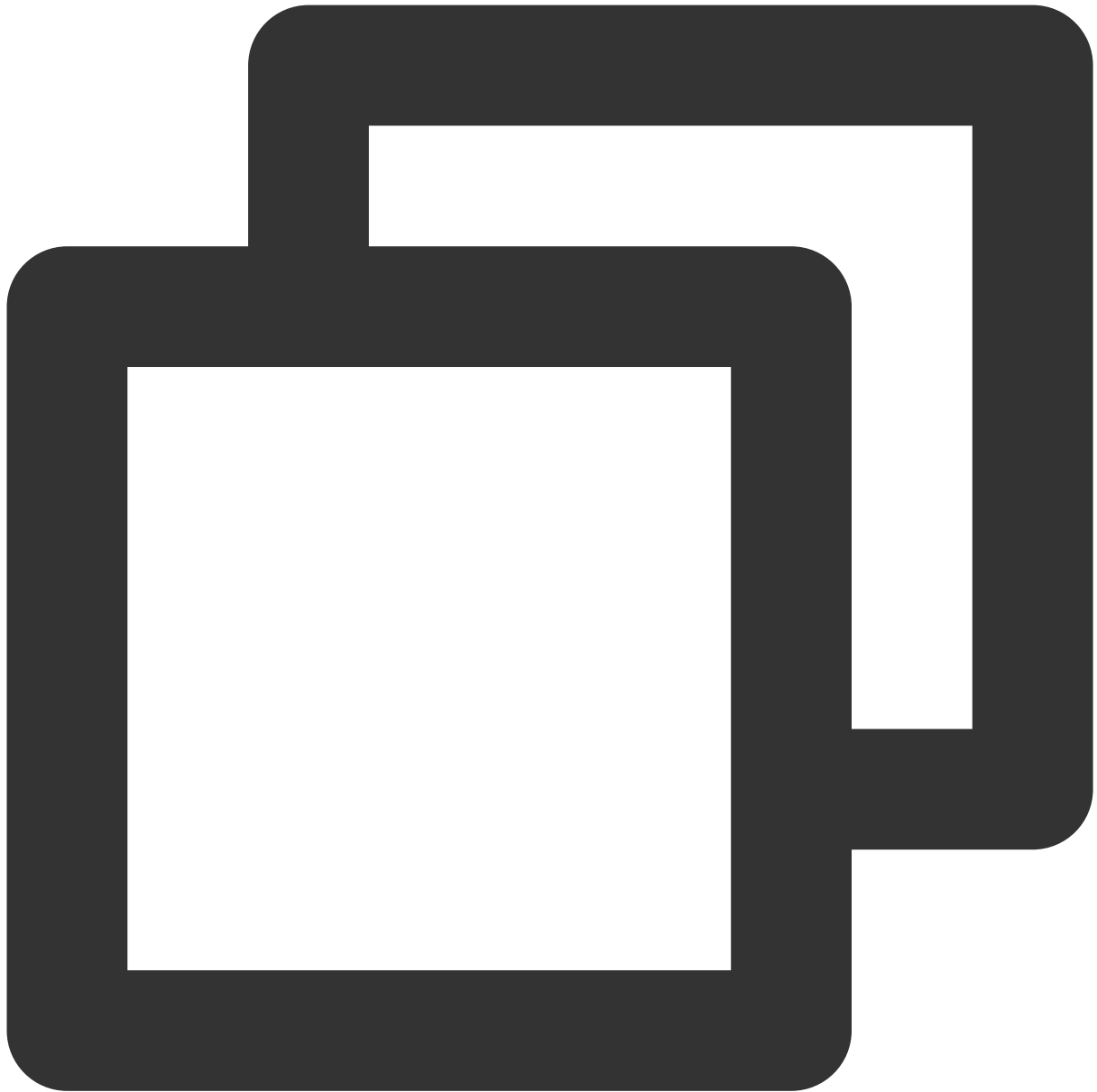


```
const event = new Event('type name');
```

基础事件，详情参见 [MDN 官方文档：Event](#)。

## 中止信号与控制器

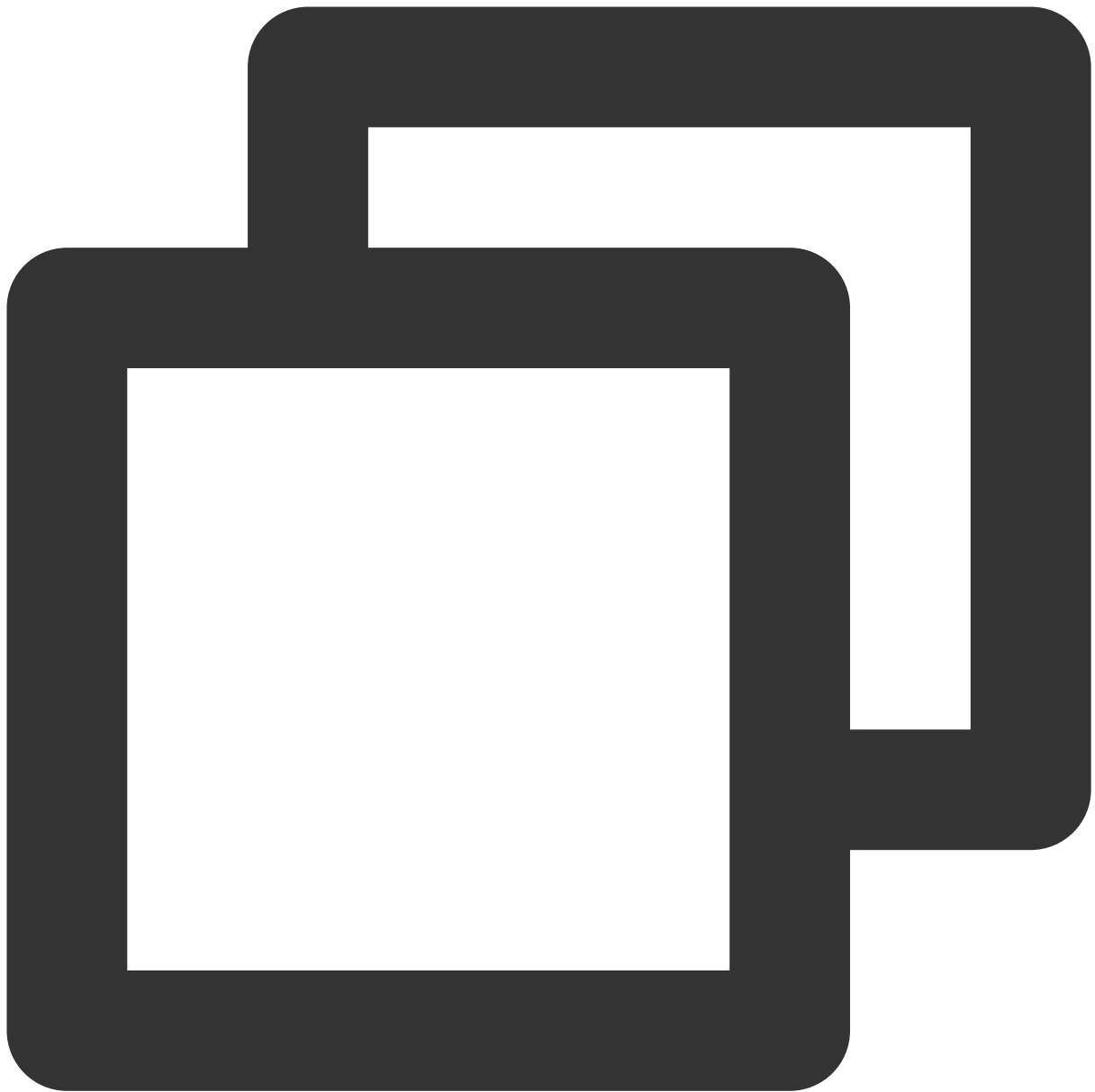
### AbortSignal



```
const signal = AbortSignal.abort();
```

中止信号，详情参见 [MDN 官方文档：AbortSignal](#)。

## AbortController



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

中止控制器，详情参见 [MDN 官方文档：AbortController](#)。

## 解压缩流

### CompressionStream





```
const { readable, writable } = new CompressionStream('gzip');
```

压缩数据流，支持 `gzip`，`deflate`，`br` 压缩方法，详情参见 [MDN 官方文档：CompressionStream](#)。

## DecompressionStream



```
const { readable, writable } = new DecompressionStream('gzip');
```

解压数据流，支持 `gzip`，`deflate`，`br` 解压方法，详情参见 [MDN 官方文档：DecompressionStream](#)。

---

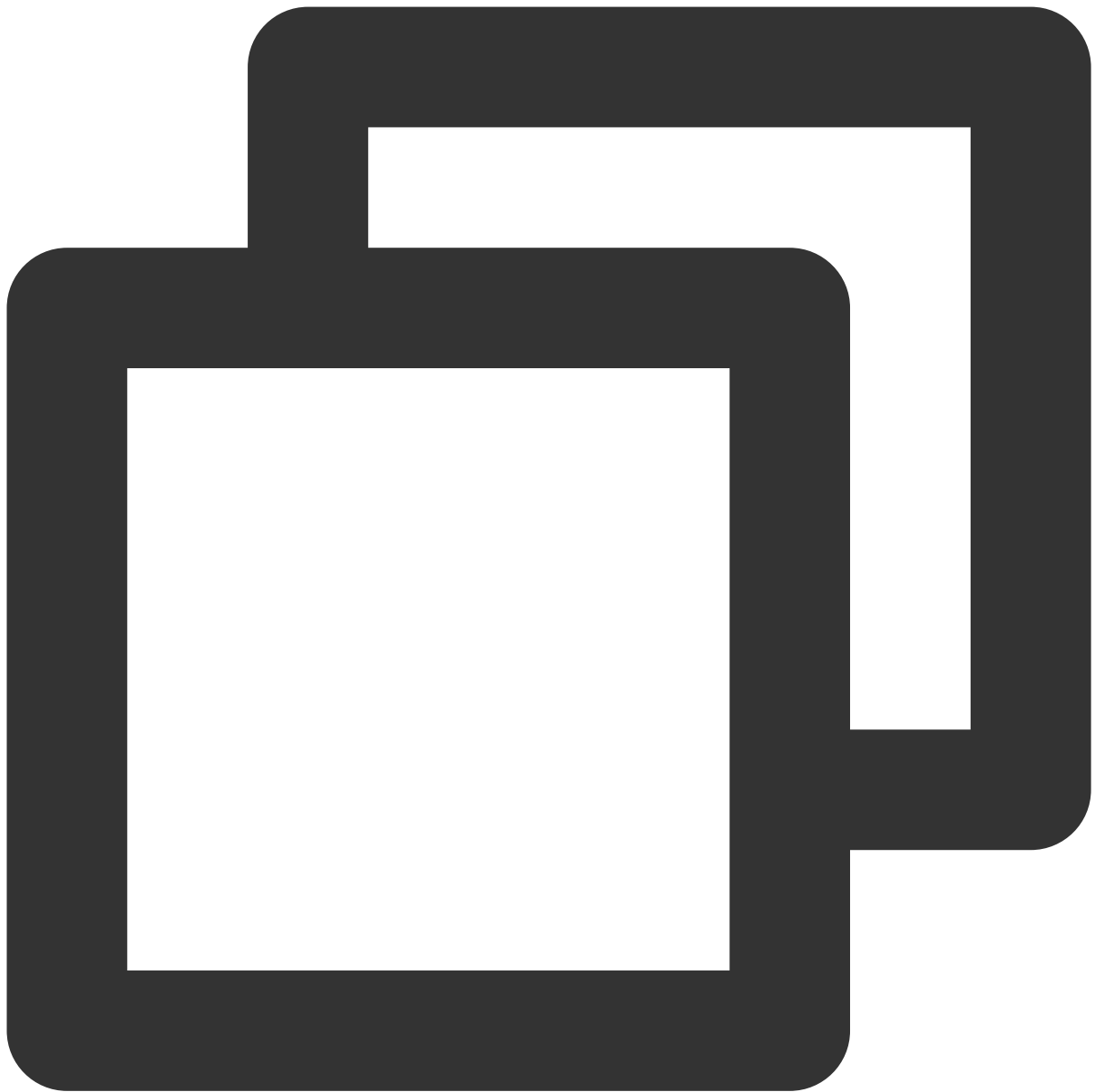
# 示例函数

## 返回 HTML 页面

最近更新时间：2024-01-25 14:28:10

使用边缘函数生成 HTML 页面，并在浏览器端访问预览该 HTML 页面。

## 示例代码



```
const html = `  
  <!DOCTYPE html>  
  <body>  
    <h1>Hello World</h1>  
    <p>This markup was generated by TencentCloud Edge Functions.</p>  
  </body>  
`;  
  
async function handleRequest(request) {  
  return new Response(html, {  
    headers: {
```

```
'content-type': 'text/html; charset=UTF-8',  
  'x-edgefunctions-test': 'Welcome to use Edge Functions.',  
},  
});  
  
addEventListener('fetch', event => {  
  event.respondWith(handleRequest(event.request));  
});
```

## 示例预览

在浏览器地址栏中输入匹配到边缘函数触发规则的 URL，即可预览到示例效果。



## 相关参考

[Runtime APIs: addEventListener](#)

[Runtime APIs: Response](#)

[Runtime APIs: FetchEvent](#)

# 返回 JSON

最近更新时间：2024-01-25 14:27:27

使用边缘函数生成 JSON，并在浏览器端访问预览该 JSON。

## 示例代码

```
const data = {
  content: 'hello world',
};

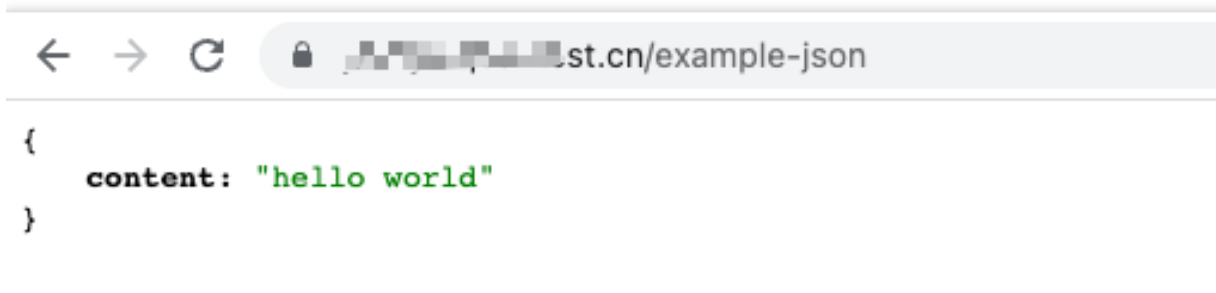
async function handleRequest(request) {
  // JSON 转为字符串
  const result = JSON.stringify(data, null, 2);

  return new Response(result, {
    headers: {
      'content-type': 'application/json; charset=UTF-8',
    },
  });
}

addEventListener('fetch', event => {
  return event.respondWith(handleRequest(event.request));
});
```

## 示例预览

在浏览器地址栏中输入匹配到边缘函数触发规则的 URL，即可预览到示例效果。



---

## 相关参考

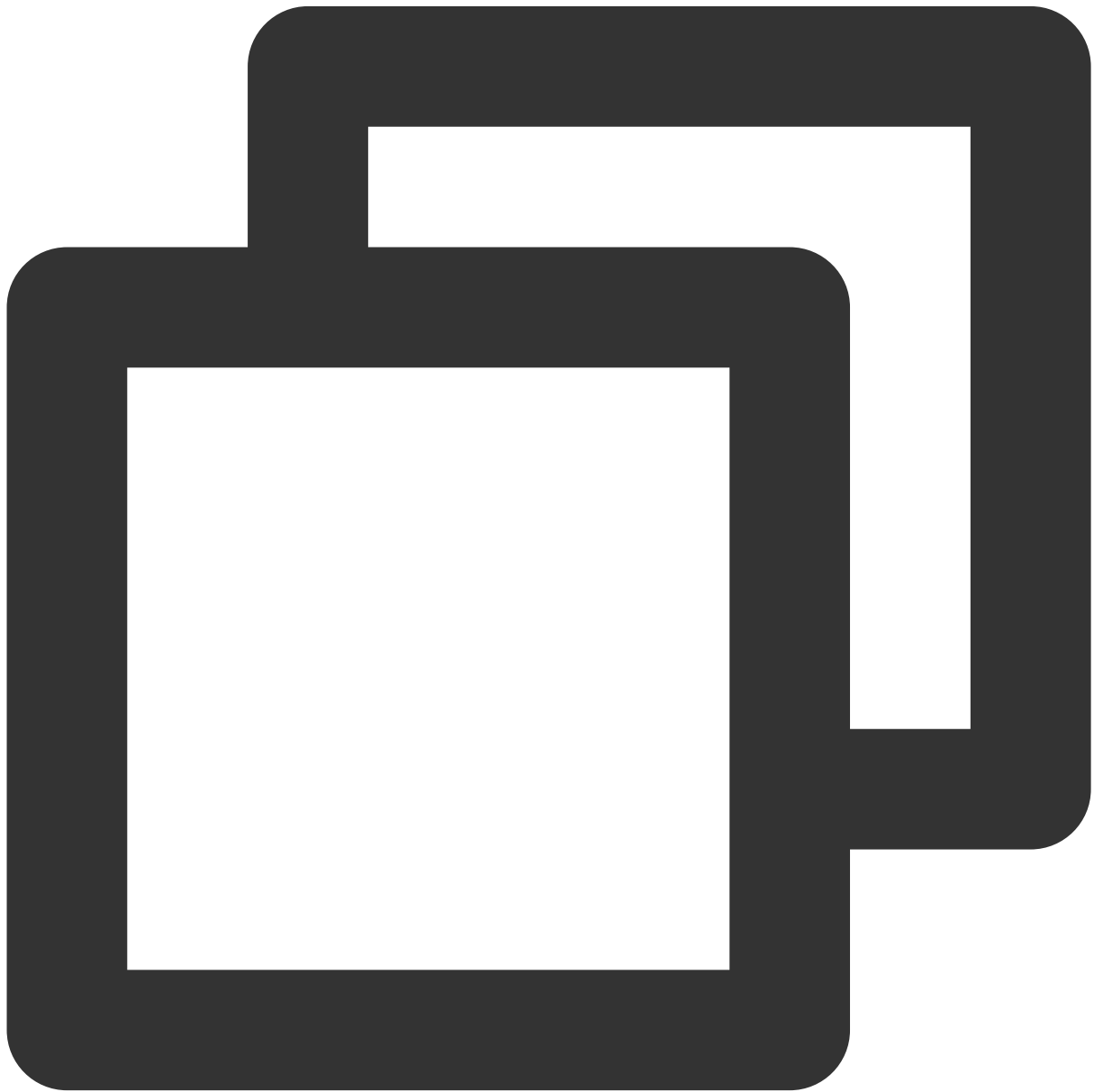
- [Runtime APIs: addEventListener](#)
- [Runtime APIs: Response](#)

# Fetch 远程资源

最近更新时间：2024-01-25 14:21:07

该示例使用 [Fetch API](#) 获取远程资源 jQuery.js 并响应给客户端。

## 示例代码





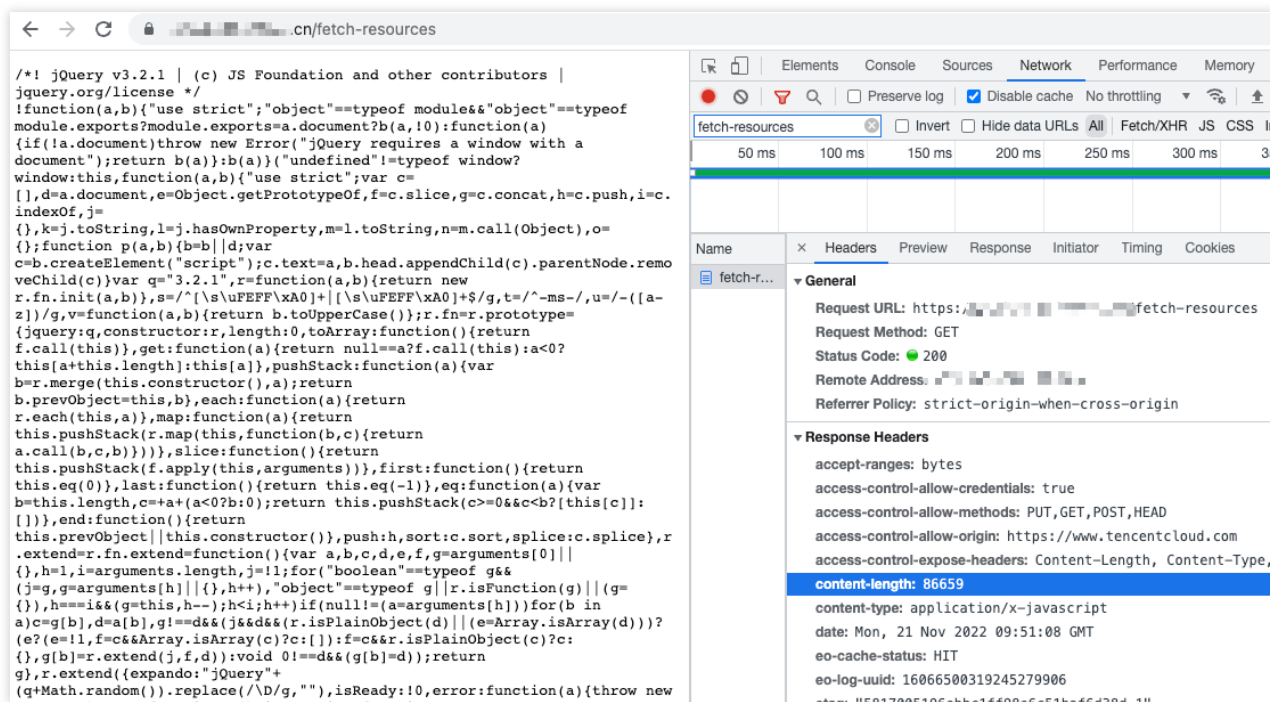
```

async function handleRequest(request) {
    // 获取远程资源
    const response = await fetch('https://static.cloudcachetci.com/qcloud/main/script
    return response;
}

addEventListener('fetch', event => {
    return event.respondWith(handleRequest(event.request));
});
    
```

## 示例预览

在浏览器地址栏中输入匹配到边缘函数触发规则的 URL，即可预览到示例效果。



The screenshot shows a browser window with the URL `...cn/fetch-resources`. The developer tools are open to the Network tab, showing a request to `fetch-resources` with a status of 200. The response headers include `content-length: 86659` and `content-type: application/x-javascript`. The console shows the jQuery library code being loaded.

## 相关参考

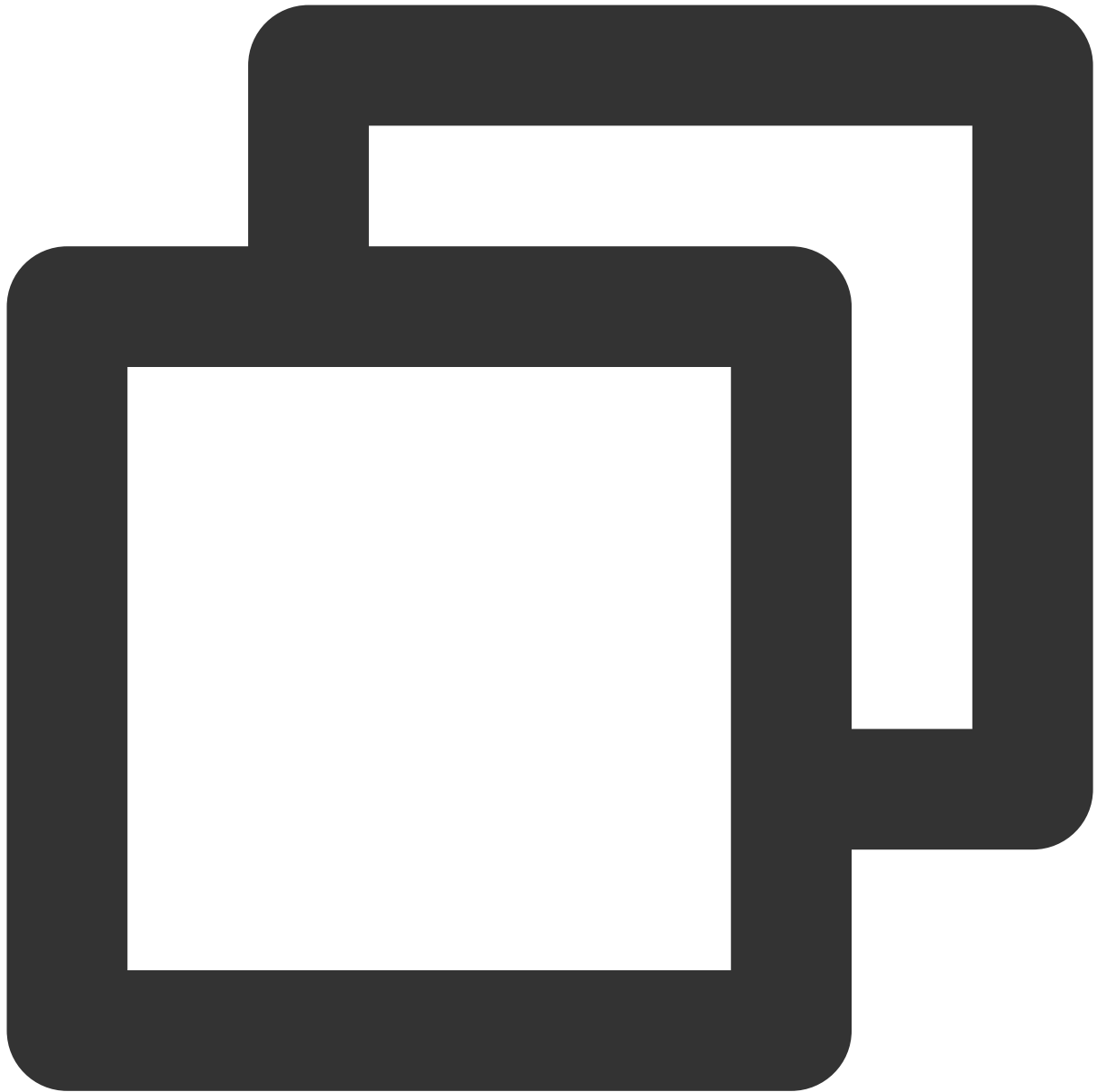
[Runtime APIs: Fetch](#)

# 请求头鉴权

最近更新时间：2024-01-25 11:44:23

该示例通过校验请求头 `x-custom-token` 的值，若其值等于 `token-123456` 则允许访问，否则拒绝访问。使用边缘函数实现了简单的权限控制。

## 示例代码



```
async function handleRequest(request) {
  const token = request.headers.get('x-custom-token');

  if (token === 'token-123456') {
    return new Response('hello world');
  }

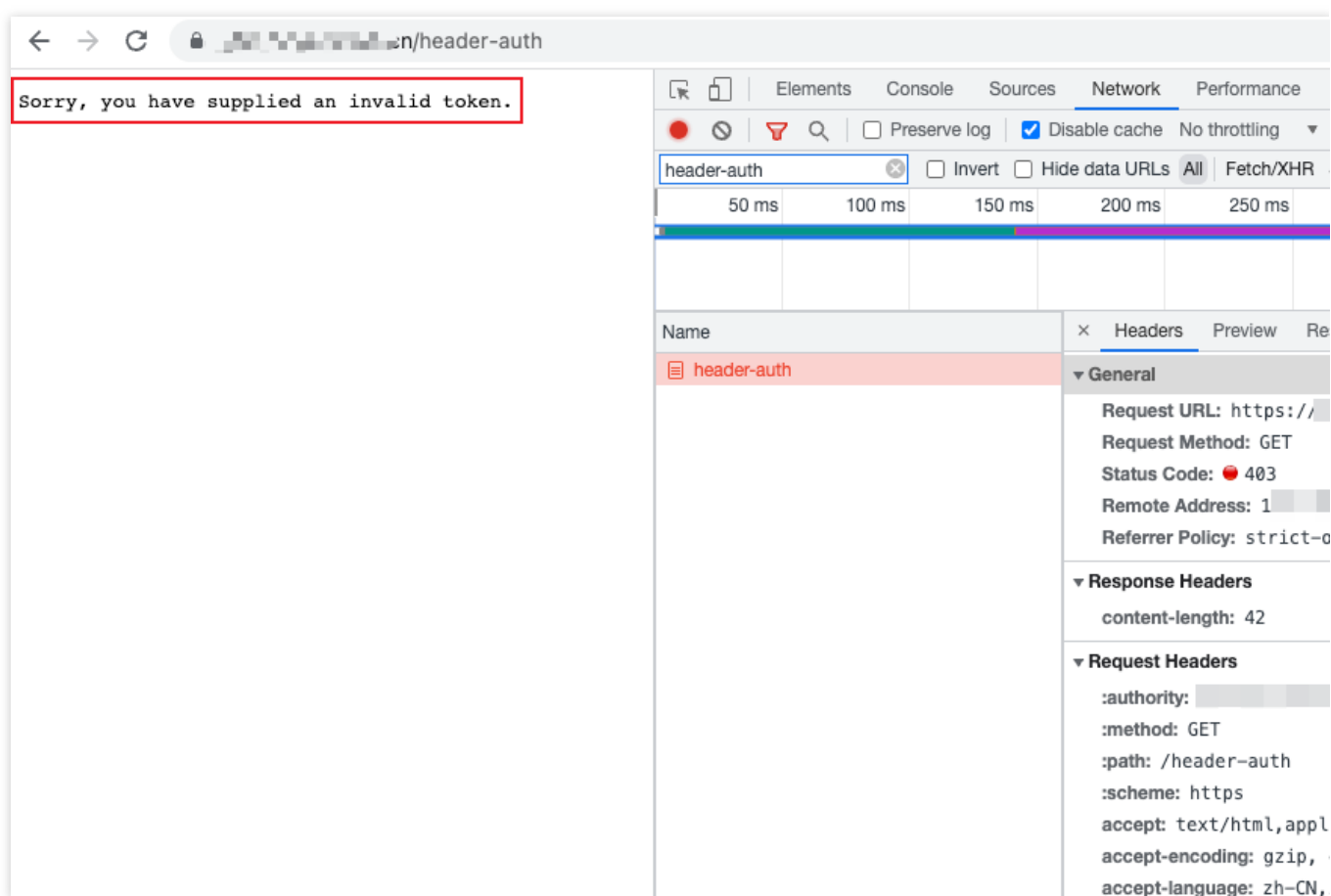
  // Incorrect key supplied. Reject the request.
  return new Response('Sorry, you have supplied an invalid token.', {
    status: 403,
  });
};
```

```
}  
  
addEventListener('fetch', event => {  
  event.respondWith(handleRequest(event.request));  
});
```

## 示例预览

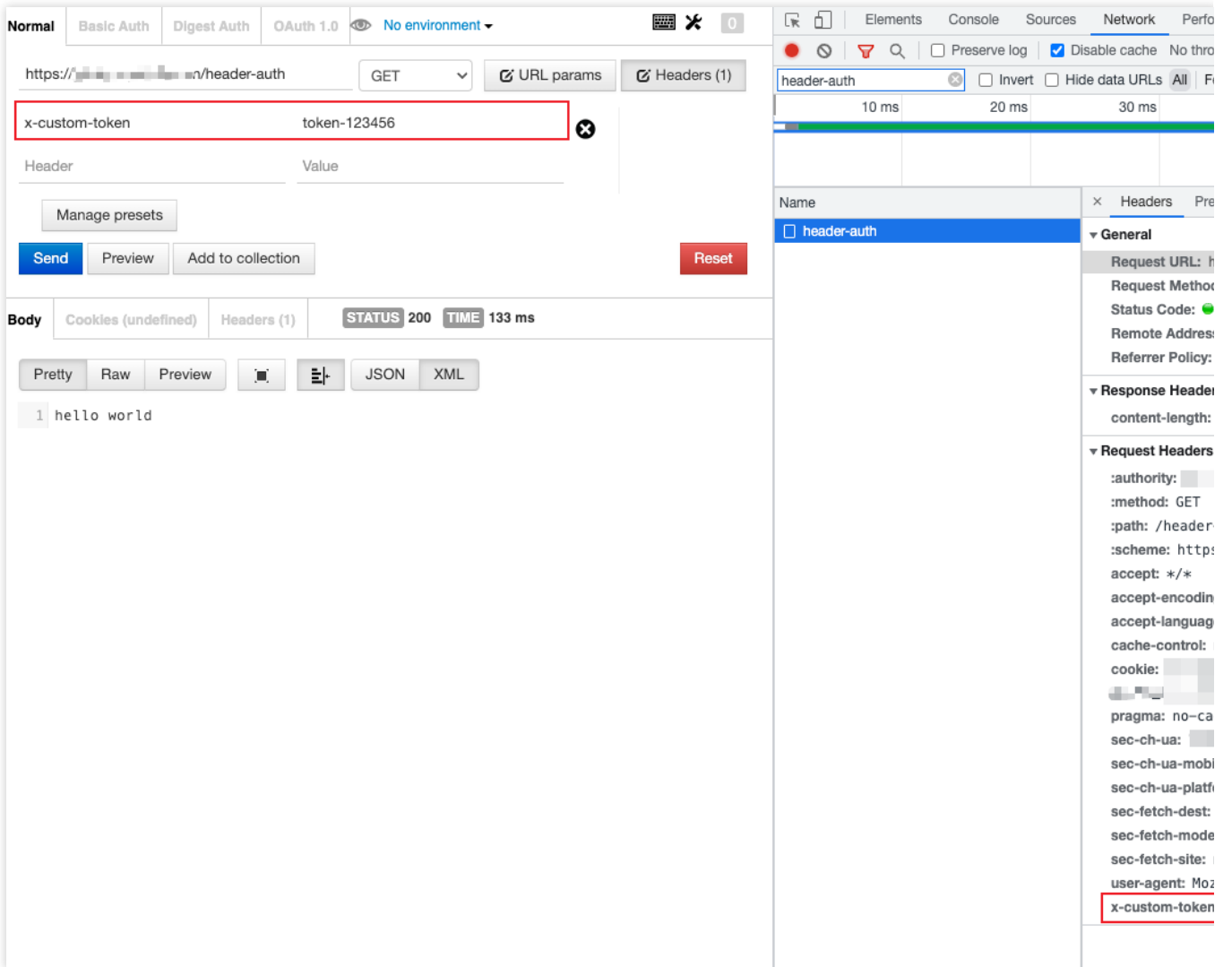
在浏览器地址栏中输入匹配到边缘函数触发规则的 URL，即可预览到示例效果。

鉴权不通过，拒绝访问。



The screenshot shows a browser window with the address bar containing a URL ending in /header-auth. The page content displays the message "Sorry, you have supplied an invalid token." in a red-bordered box. The browser's developer tools are open to the Network tab, showing a request named "header-auth" with a status code of 403. The request headers include :authority, :method: GET, :path: /header-auth, :scheme: https, accept: text/html,appl, accept-encoding: gzip, and accept-language: zh-CN. The response headers include content-length: 42.

鉴权通过，允许访问。



## 相关参考

[Runtime APIs: Headers](#)

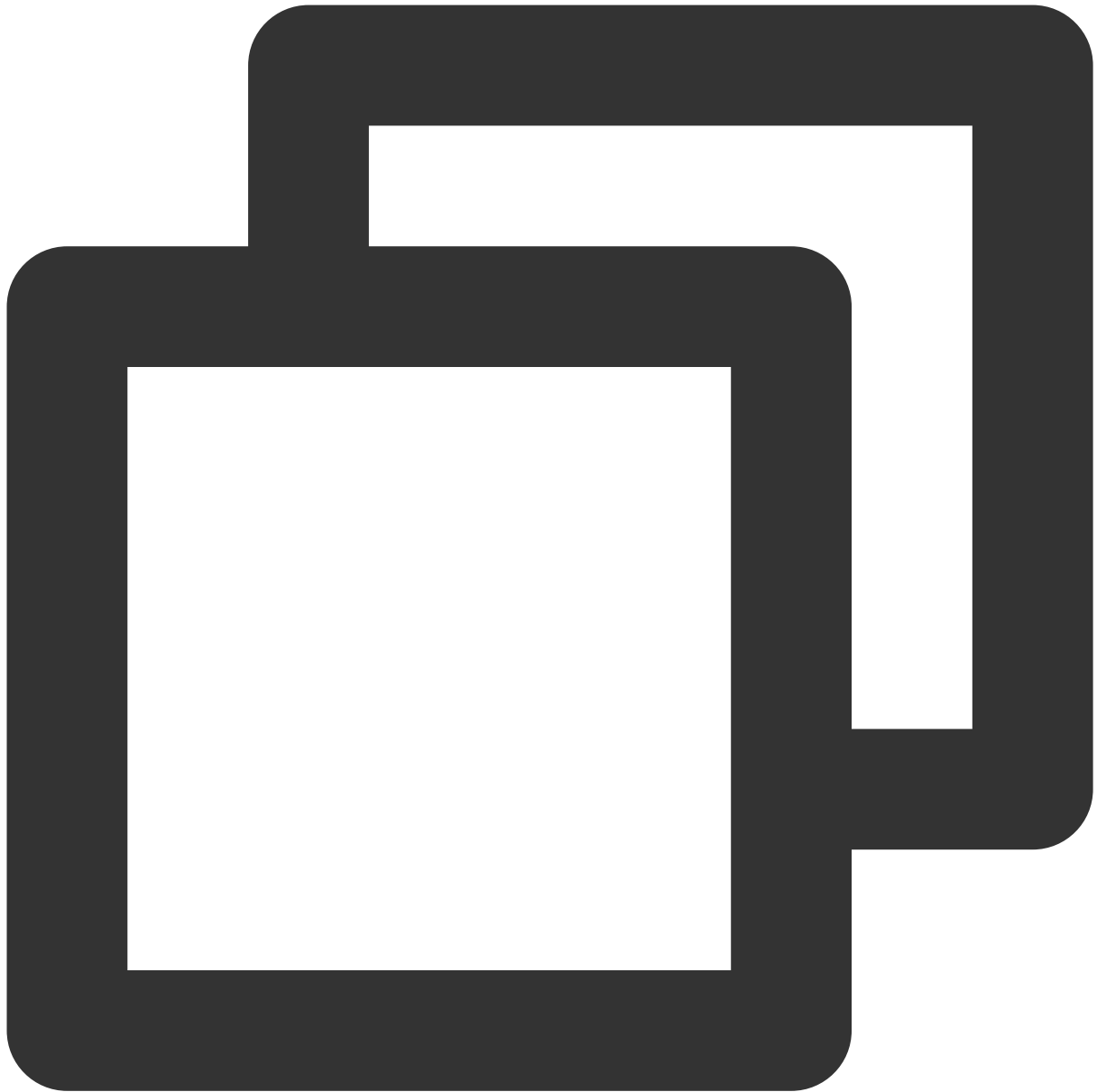
[Runtime APIs: Response](#)

# 修改响应头

最近更新时间：2023-11-24 15:09:45

该示例使用 [Fetch API](#) 实现对站点域名 `www.example.com` 的反向代理，通过边缘函数设置 HTTP 响应头，实现跨域资源共享 [CORS](#) (Cross-Origin Resource Sharing)。

## 示例代码



```
async function handleRequest(event) {
  const { request } = event;
  const urlInfo = new URL(request.url);

  const proxyRequest = new Request(`https://www.example.com${urlInfo.pathname}${url
    method: request.method,
    body: request.body,
    headers: request.headers,
    copyHeaders: true,
  });
  proxyRequest.headers.set('Host', 'www.example.com');
```

```
// fetch 反向代理
const response = await fetch(proxyRequest);

/** 添加自定义响应头 */
// 指定哪些源 (origin) 允许访问资源
response.headers.append('Access-Control-Allow-Origin', '*');
// 指定哪些 HTTP 方法 (如 GET, POST 等) 允许访问资源
response.headers.append('Access-Control-Allow-Methods', 'GET,POST');
// 指定了哪些 HTTP 头可以在正式请求头中出现
response.headers.append('Access-Control-Allow-Headers', 'Authorization');
// 预检请求的结果可以被缓存多久
response.headers.append('Access-Control-Max-Age', '86400');

/** 删除响应头 */
response.headers.delete('X-Cache');

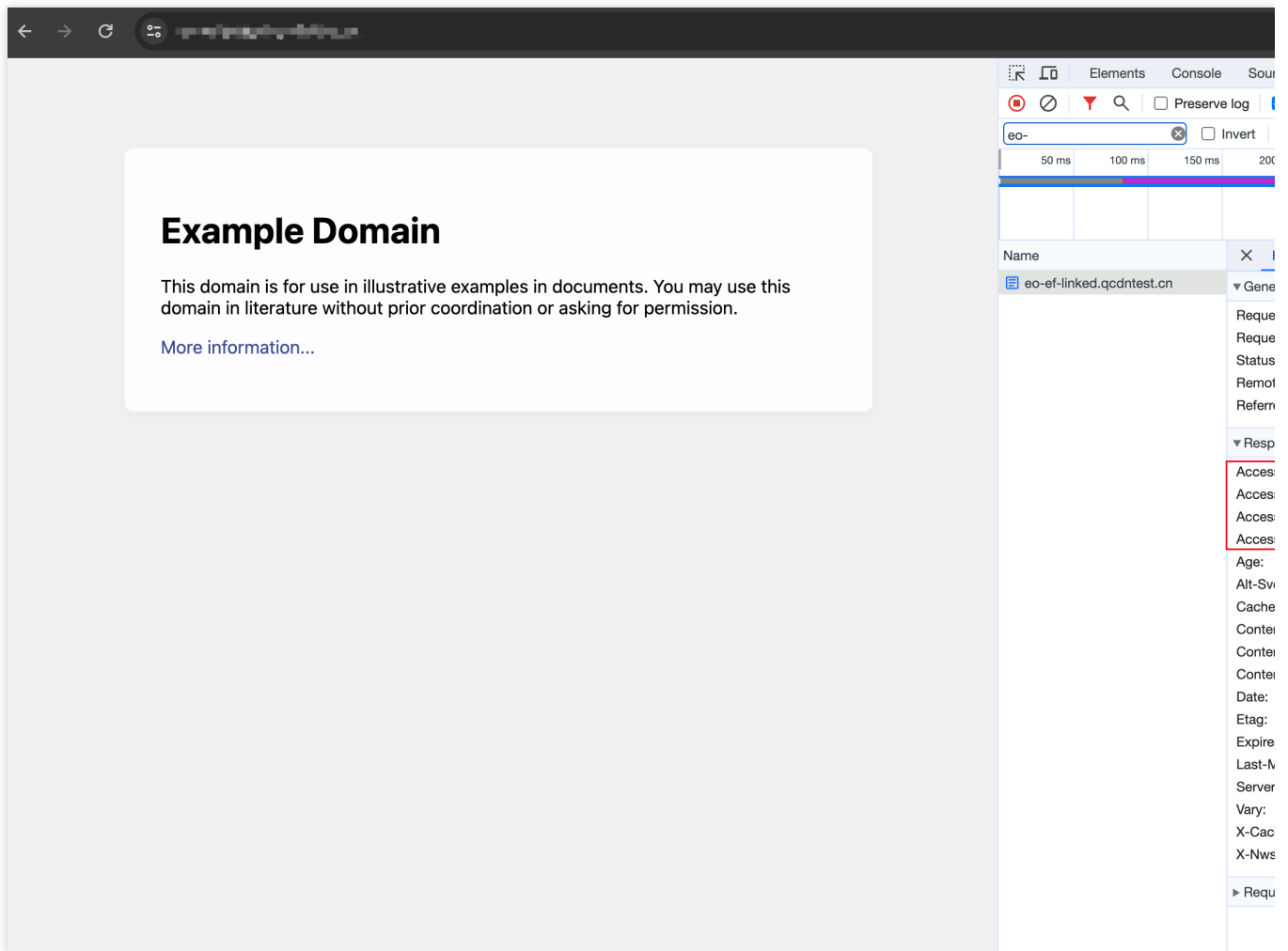
return response;
}

addEventListener('fetch', event => {
  event.respondWith(handleRequest(event));
});
```

## 示例预览

在浏览器地址栏中输入匹配到边缘函数触发规则的 URL，即可预览到示例效果。





## 相关参考

[Runtime APIs: Headers](#)

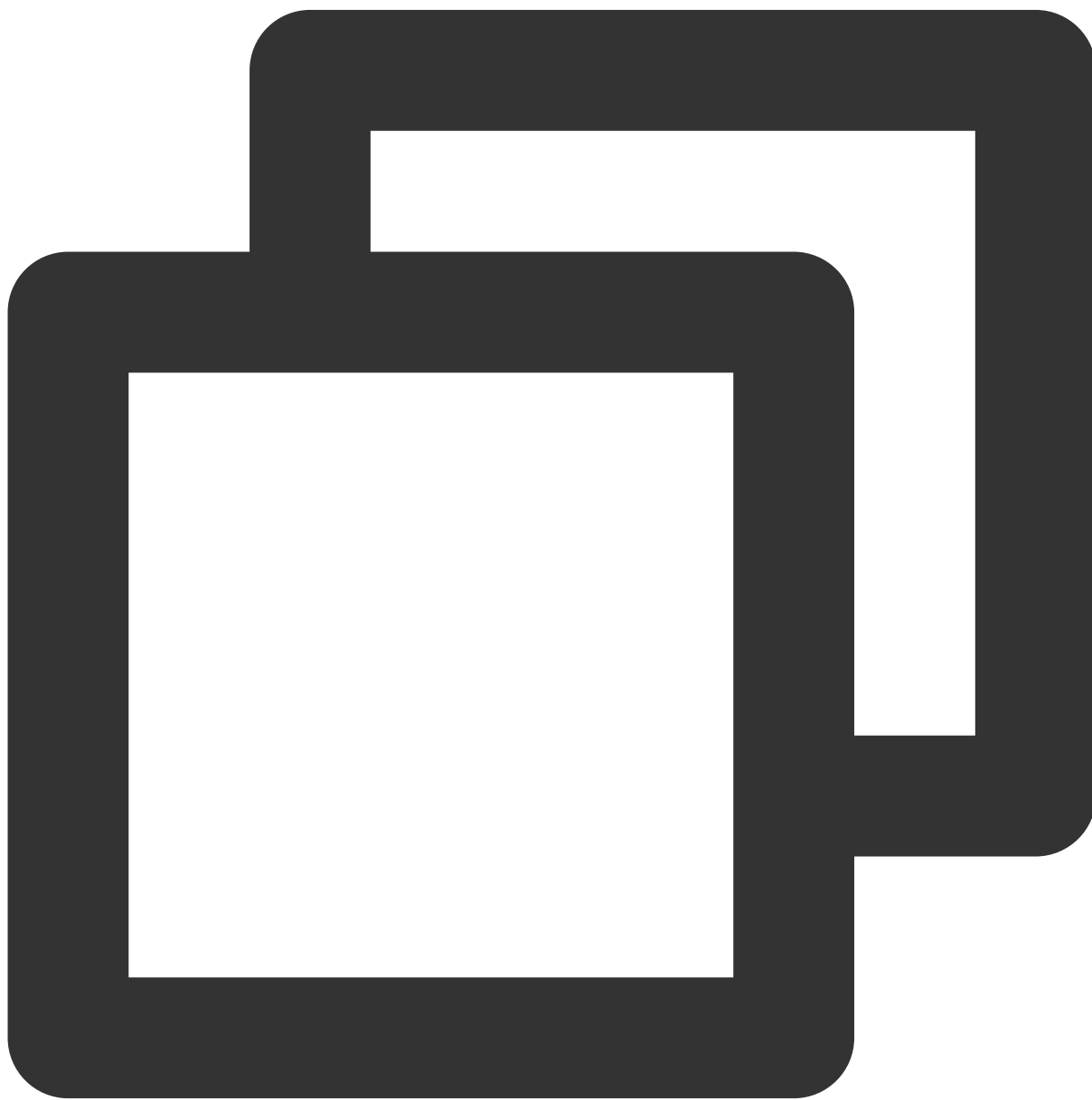
[Runtime APIs: Response](#)

# AB 测试

最近更新时间：2023-09-11 17:51:08

该示例通过 cookies 保存会话信息，对请求进行 A/B 测试控制。使用边缘函数实现了 A/B 测试的场景。

## 示例代码



```
// cookie 名称
const COOKIE_NAME = 'ABTest';

// cookie 值
const VALUE_A = 'index-a.html';
const VALUE_B = 'index-b.html';

// 根路径, 要求源站存在该路径, 并且该路径下有文件 index-a.html、index-b.html
const BASE_PATH = '/abtest';

async function handleRequest(request) {
  const urlInfo = new URL(request.url);

  // 判断 url 路径, 若访问非 abtest 的资源, 则直接响应。
  if (!urlInfo.pathname.startsWith(BASE_PATH)) {
    return fetch(request);
  }

  // 获取当前请求的 cookie
  const cookies = new Cookies(request.headers.get('cookie'));
  const abTestCookie = cookies.get(COOKIE_NAME);
  const cookieValue = abTestCookie?.value;

  // 如果 cookie 值为 A 测试, 返回 index-a.html
  if (cookieValue === VALUE_A) {
    urlInfo.pathname = `/${BASE_PATH}/${cookieValue}`;
    return fetch(urlInfo.toString());
  }

  // 如果 cookie 值为 B 测试, 返回 index-b.html
  if (cookieValue === VALUE_B) {
    urlInfo.pathname = `/${BASE_PATH}/${cookieValue}`;
    return fetch(urlInfo.toString());
  }

  // 不存在 cookie 信息, 则随机分配当前请求走 A 或 B 测试
  const testValue = Math.random() < 0.5 ? VALUE_A : VALUE_B;

  urlInfo.pathname = `/${BASE_PATH}/${testValue}`;

  const response = await fetch(urlInfo.toString());

  cookies.set(COOKIE_NAME, testValue, { path: '/', max_age: 60 });
  response.headers.set('Set-Cookie', getSetCookie(cookies.get(COOKIE_NAME)));
  return response;
}
```

```
// 拼接 Set-Cookie
function getSetCookie(cookie) {
  const cookieArr = [
    `${encodeURIComponent(cookie.name)}=${encodeURIComponent(cookie.value)}`,
  ];

  const key2name = {
    expires: 'Expires',
    max_age: 'Max-Age',
    domain: 'Domain',
    path: 'Path',
    secure: 'Secure',
    httponly: 'HttpOnly',
    samesite: 'SameSite',
  };

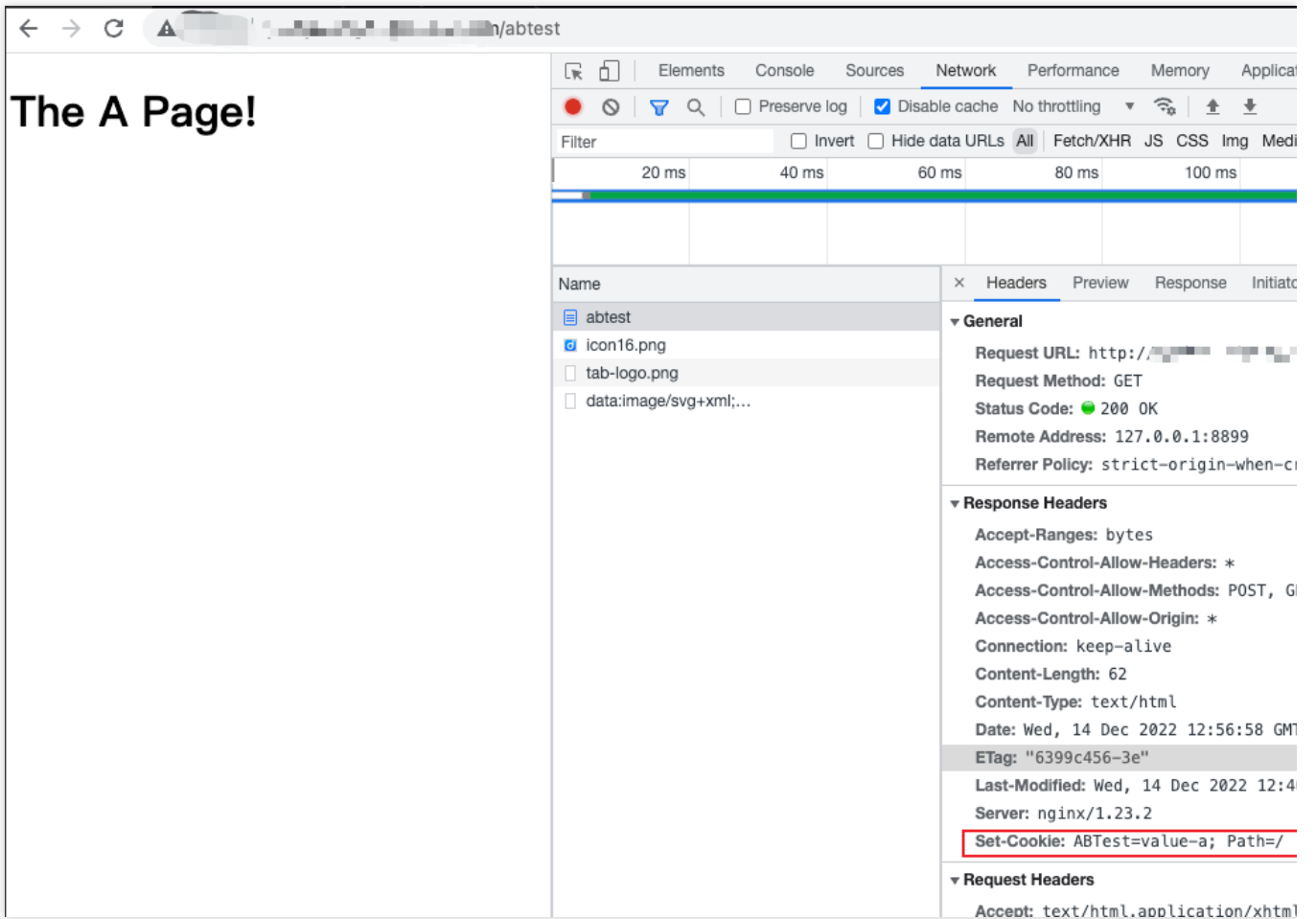
  Object.keys(key2name).forEach(key => {
    if (cookie[key]) {
      cookieArr.push(`${key2name[key]}=${cookie[key]}`);
    }
  });

  return cookieArr.join('; ');
}

addEventListener('fetch', event => {
  event.respondWith(handleRequest(event.request));
});
```

## 示例预览

在浏览器地址栏中输入匹配到边缘函数触发规则的 URL，即可预览到示例效果。



## 相关参考

[Runtime APIs: Cookies](#)

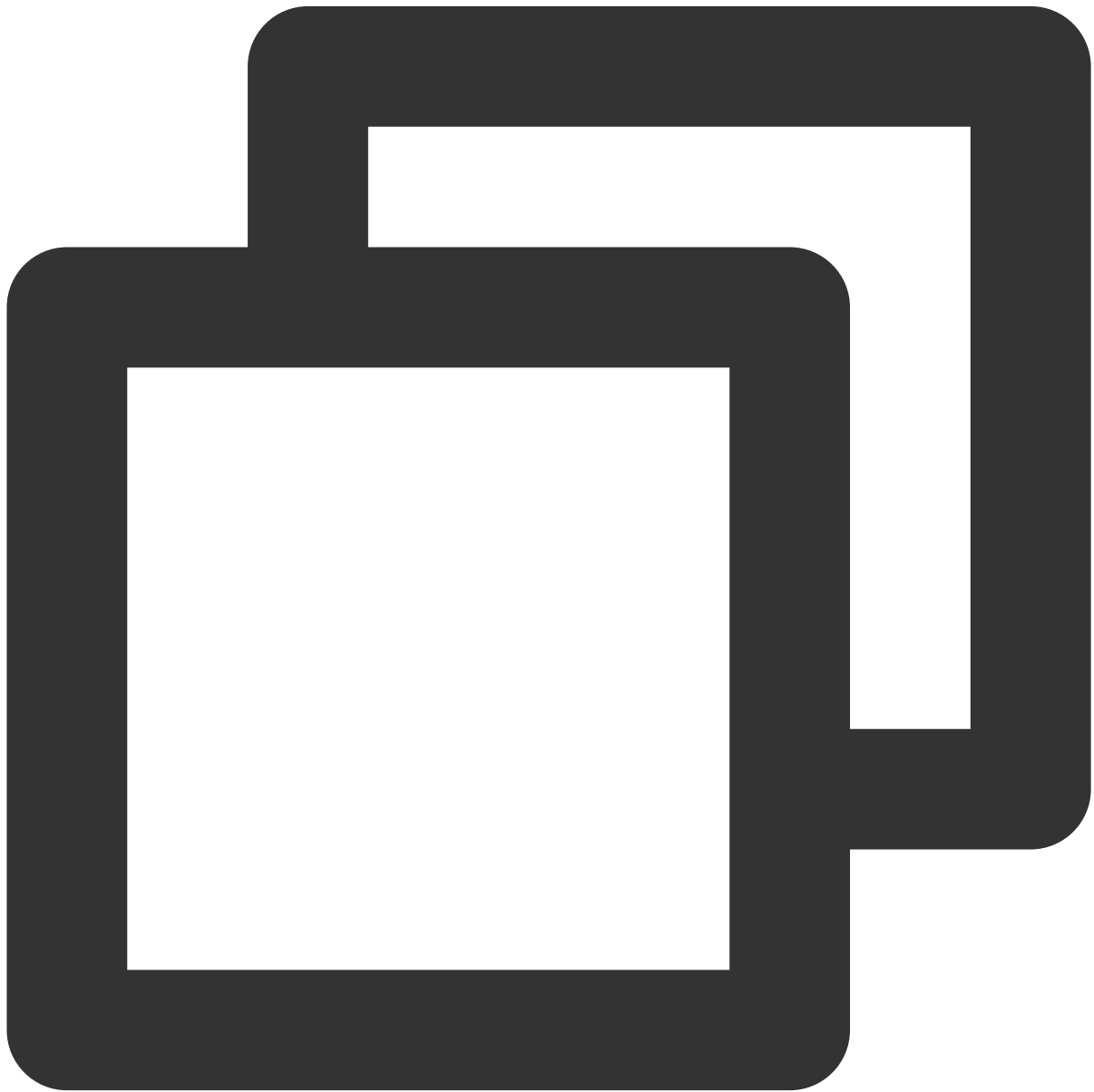
[Runtime APIs: Response](#)

# 设置 Cookie

最近更新时间：2023-09-11 17:49:08

该示例使用 Cookies 做访问计数，当浏览器访问边缘函数服务时，请求计数加 1。

## 示例代码



```
// cookie 名称
const COOKIE_NAME = 'count';

async function handleRequest(request) {
  // 获取当前请求的 Cookies, 并解析为对象
  const cookies = new Cookies(request.headers.get('cookie'));
  const cookieCount = cookies.get(COOKIE_NAME);
  // 计数累加
  const count = Number(cookieCount && cookieCount.value || 0) + 1;
  // 更新 cookie 的计数
  cookies.set(COOKIE_NAME, String(count));

  const response = new Response(`The count is: ${count}`);
  // 设置响应 cookies
  response.headers.set('Set-Cookie', getSetCookie(cookies.get(COOKIE_NAME)));
  return response;
}

// 拼接 Set-Cookie
function getSetCookie(cookie) {
  const cookieArr = [
    `${encodeURIComponent(cookie.name)}=${encodeURIComponent(cookie.value)}`,
  ];

  const key2name = {
    expires: 'Expires',
    max_age: 'Max-Age',
    domain: 'Domain',
    path: 'Path',
    secure: 'Secure',
    httponly: 'HttpOnly',
    samesite: 'SameSite',
  };

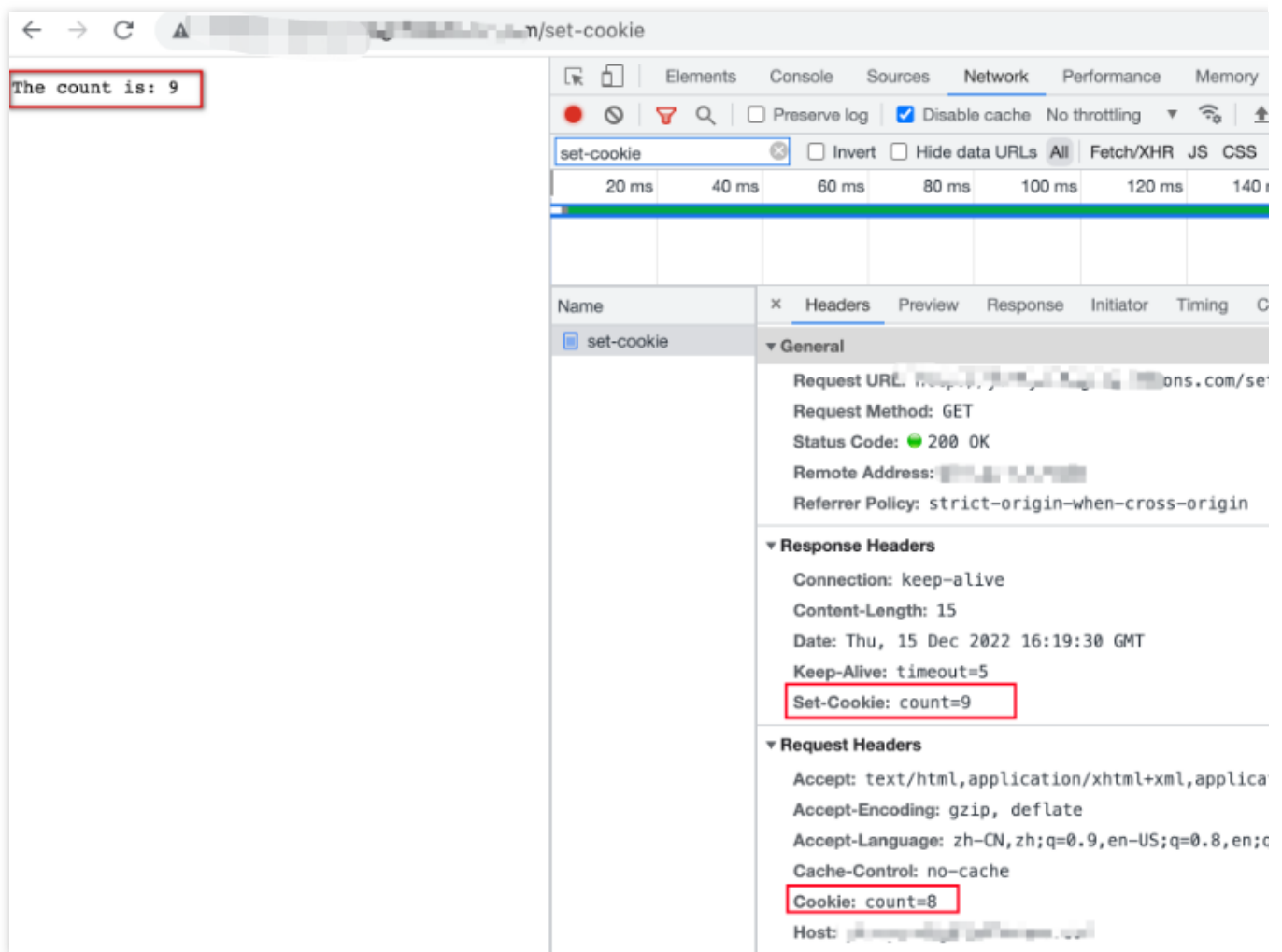
  Object.keys(key2name).forEach(key => {
    if (cookie[key]) {
      cookieArr.push(`${key2name[key]}=${cookie[key]}`);
    }
  });

  return cookieArr.join('; ');
}

addEventListener('fetch', (event) => {
  event.respondWith(handleRequest(event.request));
});
```

## 示例预览

在浏览器地址栏中输入匹配到边缘函数触发规则的 URL，即可预览到示例效果。



## 相关参考

[Runtime APIs: Cookies](#)

[Runtime APIs: Response](#)

[Runtime APIs: Request](#)

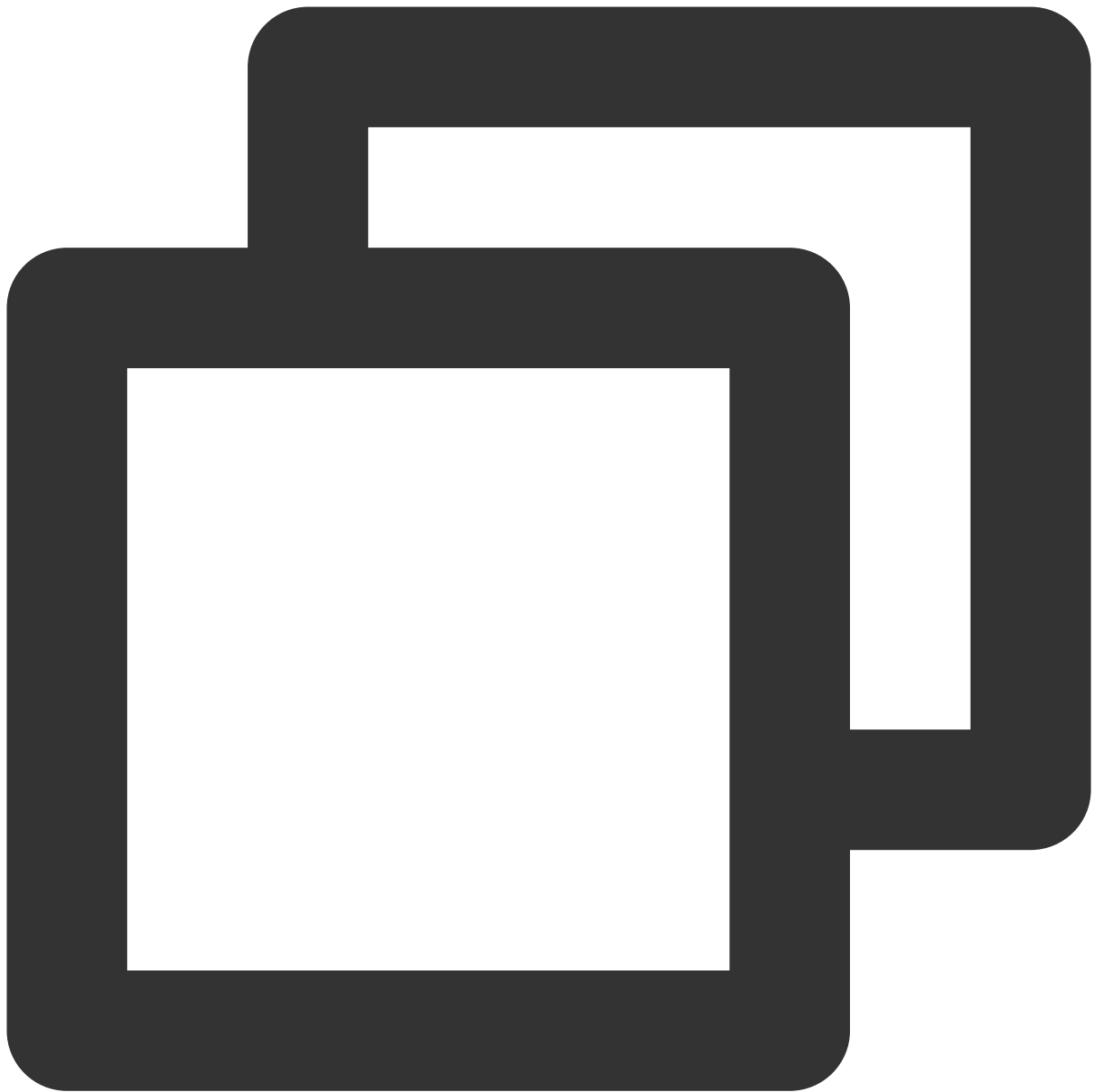


# 基于请求区域重定向

最近更新时间：2023-11-24 15:08:02

该示例通过判断客户端所属区域，自动重定向到所属区域的目标网址。实现了通过边缘函数根据客户端所属区域分发请求。

## 示例代码



```
// 所有区域网址集
const urls = {
  CN: 'https://www.example.com/zh-CN',
  US: 'https://www.example.com/en-US',
};

// 默认重定向网址
const defaultUrl = 'https://www.example.com/en-US';

/**
 * 根据当前请求所在的区域，重定向到目标网址
```

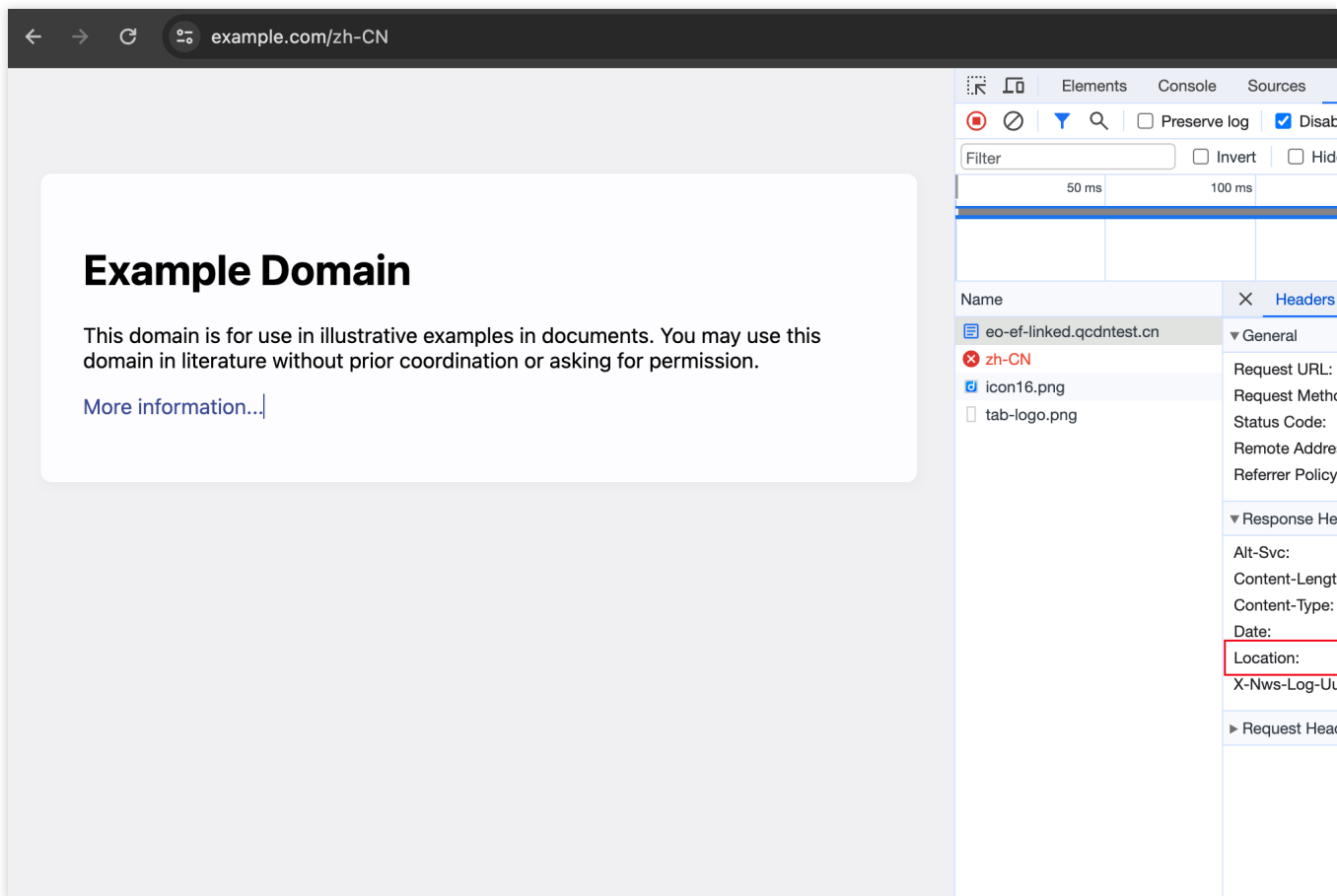
```
* @param { Request } request
*/
function handleRequest(request) {
  // 获取当前请求所在区域
  const alpha2code = request.eo.geo.countryCodeAlpha2;
  // 重定向目标网址
  const url = urls[alpha2code] || defaultUrl;

  return Response.redirect(url, 302);
}

addEventListener('fetch', event => {
  event.respondWith(handleRequest(event.request));
});
```

## 示例预览

在浏览器地址栏中输入匹配到边缘函数触发规则的 URL，即可预览到示例效果。



## 相关参考

---

[Runtime APIs: Request](#)

[Runtime APIs: Response](#)

# Cache API 使用

最近更新时间：2024-01-25 11:41:55

在边缘函数中，使用 [Fetch API](#) 获取远程资源 jQuery.js，借助 [Cache API](#) 把资源缓存到 EdgeOne 边缘节点，缓存时长为 10s。

## 示例代码



```
async function fetchJquery(event, request) {
  const cache = caches.default;
  // 缓存没有命中, 回源并缓存
  let response = await fetch(request);

  // 在响应头添加 Cache-Control, 设置缓存时长 10s
  response.headers.append('Cache-Control', 's-maxage=10');
  event.waitUntil(cache.put(request, response.clone()));

  // 未命中缓存, 设置响应头标识
  response.headers.append('x-edgefunctions-cache', 'miss');
```

```
return response;
}

async function handleEvent(event) {
  // 资源地址，也作为缓存键
  const request = new Request('https://static.cloudcachetci.com/qcloud/main/scripts
  // 缓存默认实例
  const cache = caches.default;

  try {
    // 获取关联的缓存内容，缓存过，接口底层不主动回源，抛出 504 错误
    let response = await cache.match(request);

    // 缓存不存在，重新获取远程资源
    if (!response) {
      return fetchJquery(event, request);
    }

    // 命中缓存，设置响应头标识
    response.headers.append('x-edgefunctions-cache', 'hit');

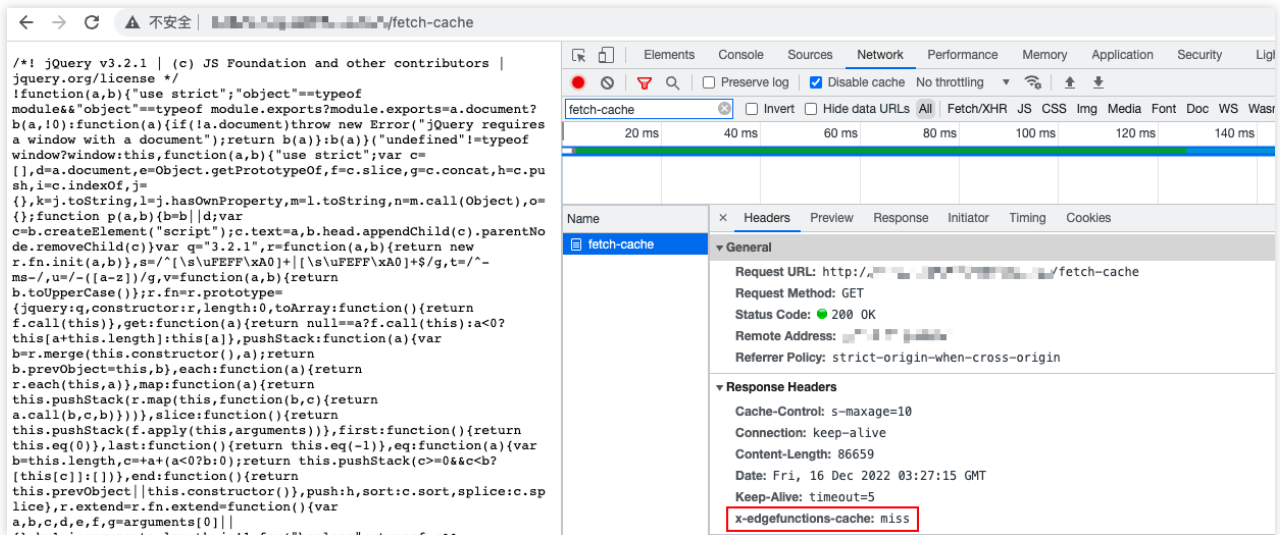
    return response;
  } catch (e) {
    await cache.delete(request);
    // 缓存过期或其他异常，重新获取远程资源
    return fetchJquery(event, request);
  }
}

addEventListener('fetch', (event) => {
  event.respondWith(handleEvent(event));
});
```

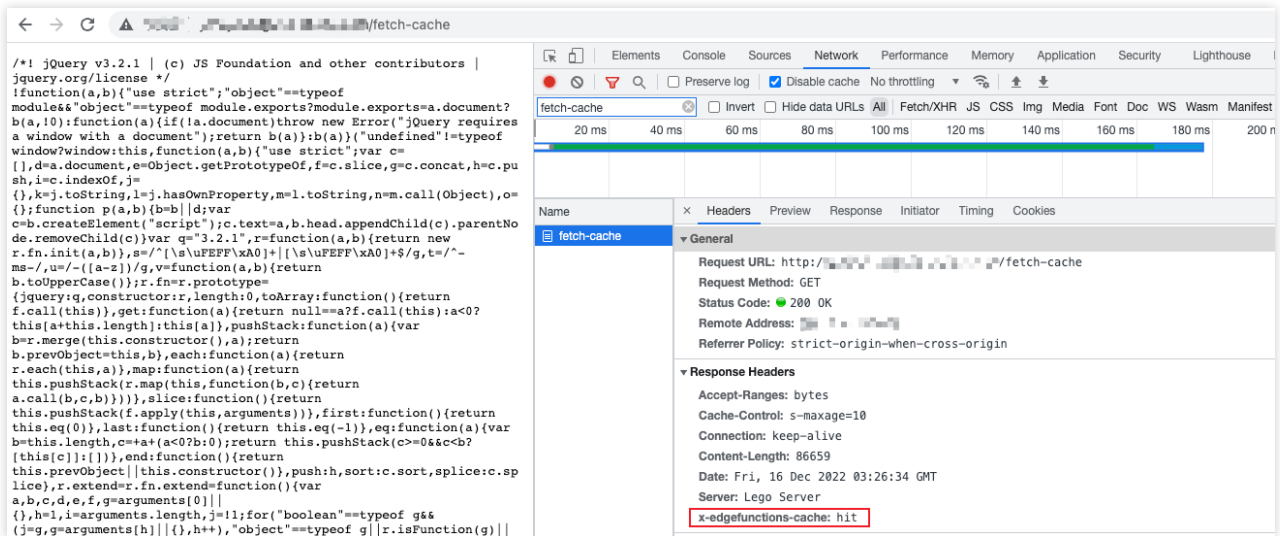
## 示例预览

在浏览器地址栏中输入匹配到边缘函数触发规则的 URL，即可预览到示例效果。

未命中缓存。



命中缓存。



## 相关参考

[Runtime APIs: Cache](#)

[Runtime APIs: Fetch](#)

[Runtime APIs: FetchEvent](#)

[Runtime APIs: Response](#)

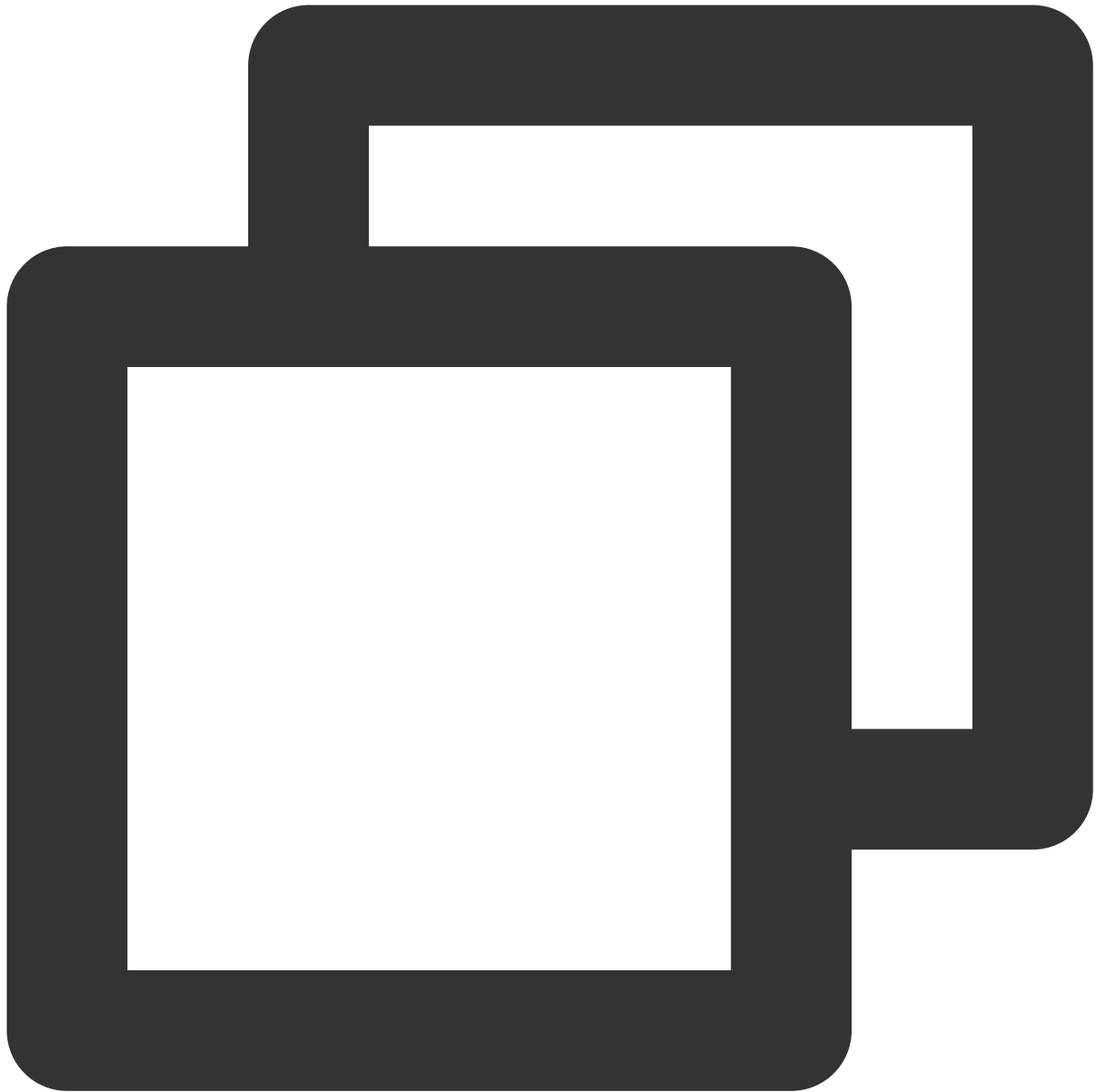


# 缓存 POST 请求

最近更新时间：2024-01-25 11:38:55

该示例对 POST 请求 body 计算 SHA-256 签名，作为缓存 key 的一部分，并使用 [Cache API](#) 将响应内容进行缓存，若存在缓存，则使用缓存内容响应客户端，否则使用 [Fetch API](#) 发起子请求获取远程资源。实现了，使用边缘函数缓存 POST 请求。

## 示例代码



```
function uint8ArrayToHex(arr) {
  return Array.prototype.map.call(arr, (x) => (('0' + x.toString(16)).slice(-2))).join('');
}

// sha256 签名摘要
async function sha256(message) {
  const msgBuffer = new TextEncoder().encode(message);
  const hashBuffer = await crypto.subtle.digest('SHA-256', msgBuffer);

  return uint8ArrayToHex(new Uint8Array(hashBuffer));
}
```

```
async function fetchContent(event, cacheKey) {
  const cache = caches.default;

  // 缓存没有命中, 回源并使用缓存
  const response = await fetch(event.request);

  // 在响应头添加 Cache-Control, 设置缓存时长
  response.headers.set('Cache-Control', 's-maxage=10');
  event.waitUntil(cache.put(cacheKey, response.clone()));

  // 未命中缓存, 设置响应头标识
  response.headers.append('x-edgefunctions-cache', 'miss');

  return response;
}

async function handleRequest(event) {
  const request = event.request;
  const body = await request.clone().text();

  // // 根据 request body 计算 hash
  const hash = await sha256(body);

  // request body 计算的 hash 值作为 cacheKey 的一部分
  const cacheKey = `${request.url}${hash}`;

  const cache = caches.default;

  try {
    // 获取关联的缓存 Response
    let response = await cache.match(cacheKey);

    if (!response) {
      return fetchContent(event, cacheKey);
    }

    // 命中缓存, 设置响应头标识
    response.headers.append('x-edgefunctions-cache', 'hit');

    return response;
  } catch (error) {
    await cache.delete(cacheKey);
    // 缓存过期或不存, 重新获取远程资源
    return fetchContent(event, cacheKey);
  }
}
```

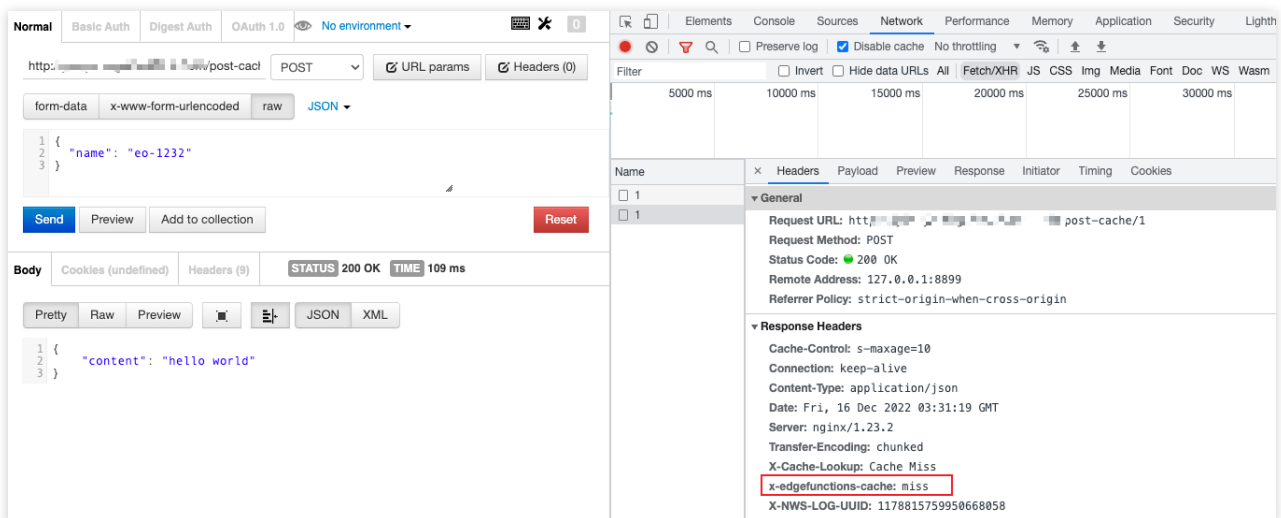
```
return response;
}

addEventListener('fetch', (event) => {
  try {
    const request = event.request;
    // 处理 POST 请求
    if (request.method.toUpperCase() === 'POST') {
      return event.respondWith(handleRequest(event));
    }
    // 非post 请求
    return event.respondWith(fetch(request));
  } catch (e) {
    return event.respondWith(new Response('Error thrown ' + e.message));
  }
});
```

## 示例预览

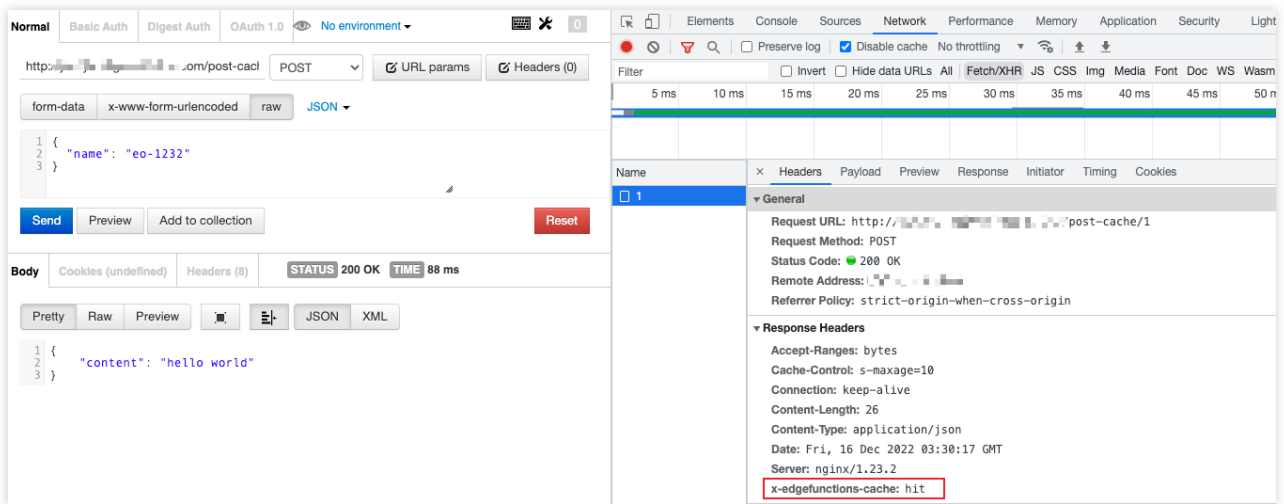
在浏览器地址栏中输入匹配到边缘函数触发规则的 URL，即可预览到示例效果。

未命中缓存。



The screenshot shows the Chrome DevTools Network tab. A POST request to 'http://.../post-cache' is selected. The request body is a JSON object: `{ "name": "eo-1232" }`. The response status is 200 OK with a time of 109 ms. The response body is a JSON object: `{ "content": "hello world" }`. The response headers are visible, including 'Cache-Control: s-maxage=10', 'Connection: keep-alive', 'Content-Type: application/json', 'Date: Fri, 16 Dec 2022 03:31:19 GMT', 'Server: nginx/1.23.2', 'Transfer-Encoding: chunked', 'X-Cache-Lookup: Cache Miss', and 'x-edgefunctions-cache: miss' (highlighted with a red box). The 'X-NWS-LOG-UUID' is also visible.

命中缓存。



## 相关参考

[Runtime APIs: Fetch](#)

[Runtime APIs: Cache](#)

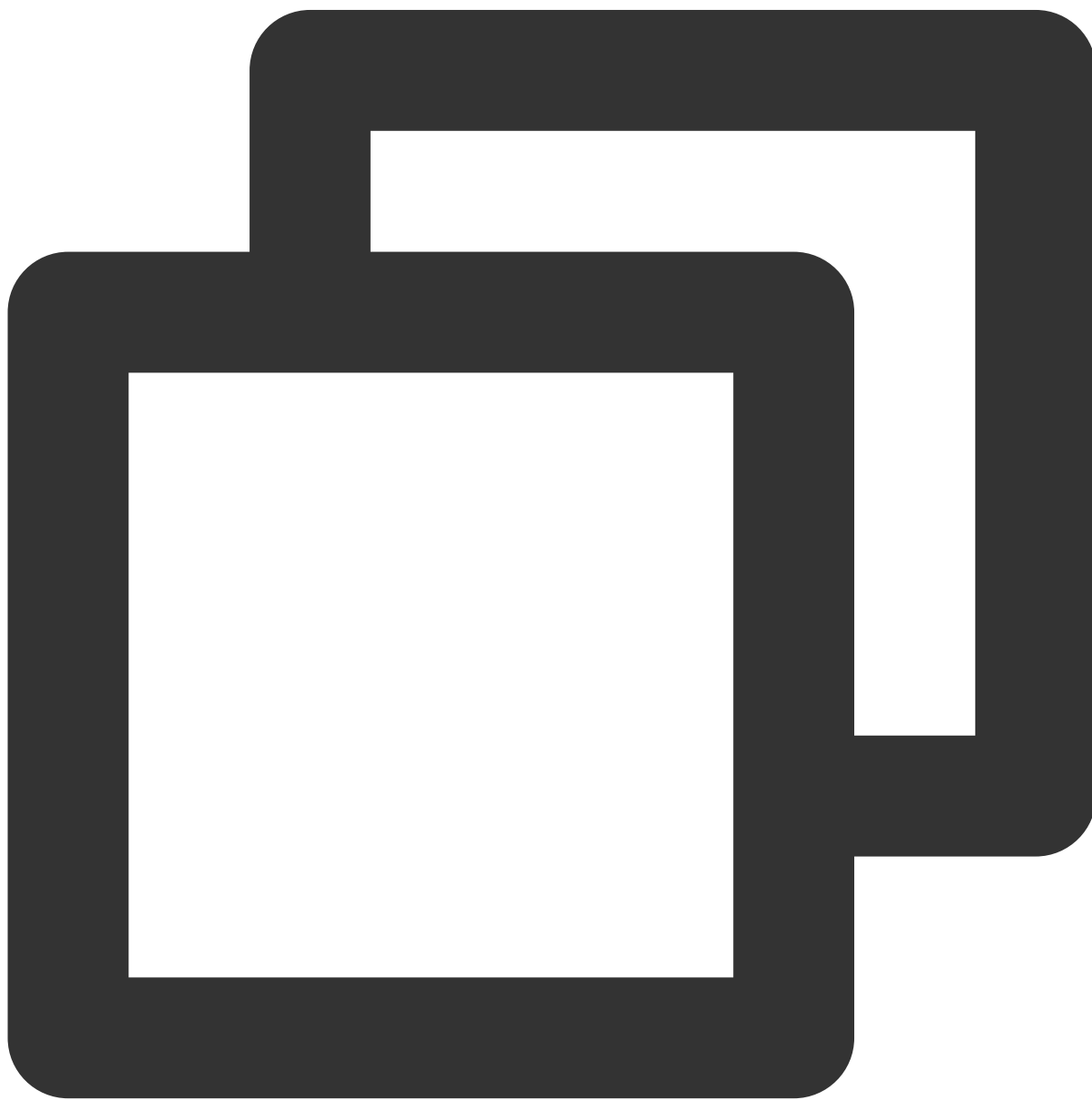
[Runtime APIs: Web Crypto](#)

# 流式响应

最近更新时间：2024-01-25 11:36:17

该示例使用 [Fetch API](#) 获取远程资源 jQuery.js 并流式响应给客户端。

## 示例代码



```
async function handleRequest(request) {
  const response = await fetch('https://static.cloudcachetci.com/qcloud/main/script

  if (response.status !== 200) {
    return response;
  }

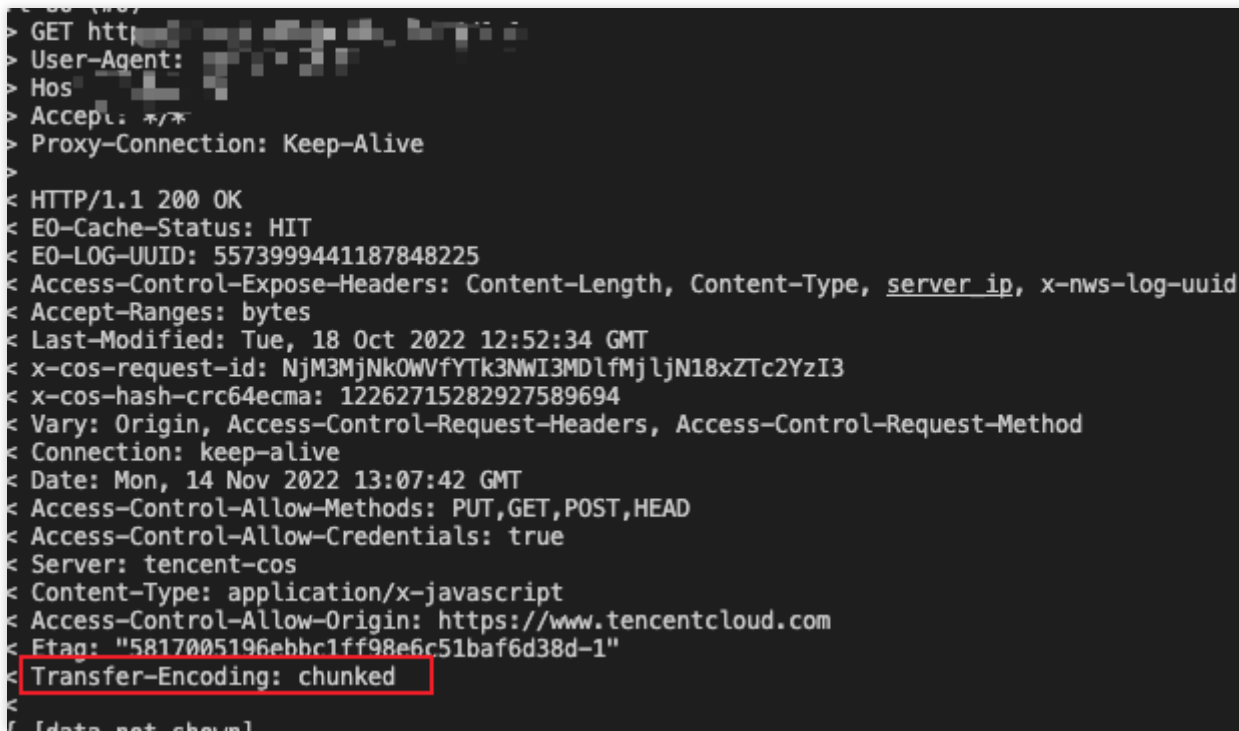
  // 生成可读端与可写端
  const { readable, writable } = new TransformStream();
  // 流式响应客户端
  response.body.pipeTo(writable);

  return new Response(readable, response);
}

addEventListener('fetch', event => {
  event.respondWith(handleRequest(event.request));
});
```

## 示例预览

在浏览器地址栏中输入匹配到边缘函数触发规则的 URL，即可预览到示例效果。



```
> GET http://...
> User-Agent: ...
> Hos
> Accept: */*
> Proxy-Connection: Keep-Alive
>
< HTTP/1.1 200 OK
< EO-Cache-Status: HIT
< EO-LOG-UUID: 5573999441187848225
< Access-Control-Expose-Headers: Content-Length, Content-Type, server_ip, x-nws-log-uuid
< Accept-Ranges: bytes
< Last-Modified: Tue, 18 Oct 2022 12:52:34 GMT
< x-cos-request-id: NjM3MjNkOWVfYTtk3NWI3MDlfMjlnN18xZTc2YzI3
< x-cos-hash-crc64ecma: 12262715282927589694
< Vary: Origin, Access-Control-Request-Headers, Access-Control-Request-Method
< Connection: keep-alive
< Date: Mon, 14 Nov 2022 13:07:42 GMT
< Access-Control-Allow-Methods: PUT,GET,POST,HEAD
< Access-Control-Allow-Credentials: true
< Server: tencent-cos
< Content-Type: application/x-javascript
< Access-Control-Allow-Origin: https://www.tencentcloud.com
< Etag: "5817005196ebhc1ff98e6c51baf6d38d-1"
< Transfer-Encoding: chunked
<
[data not shown]
```

---

## 相关参考

[Runtime APIs: TransformStream](#)

[Runtime APIs: Response](#)

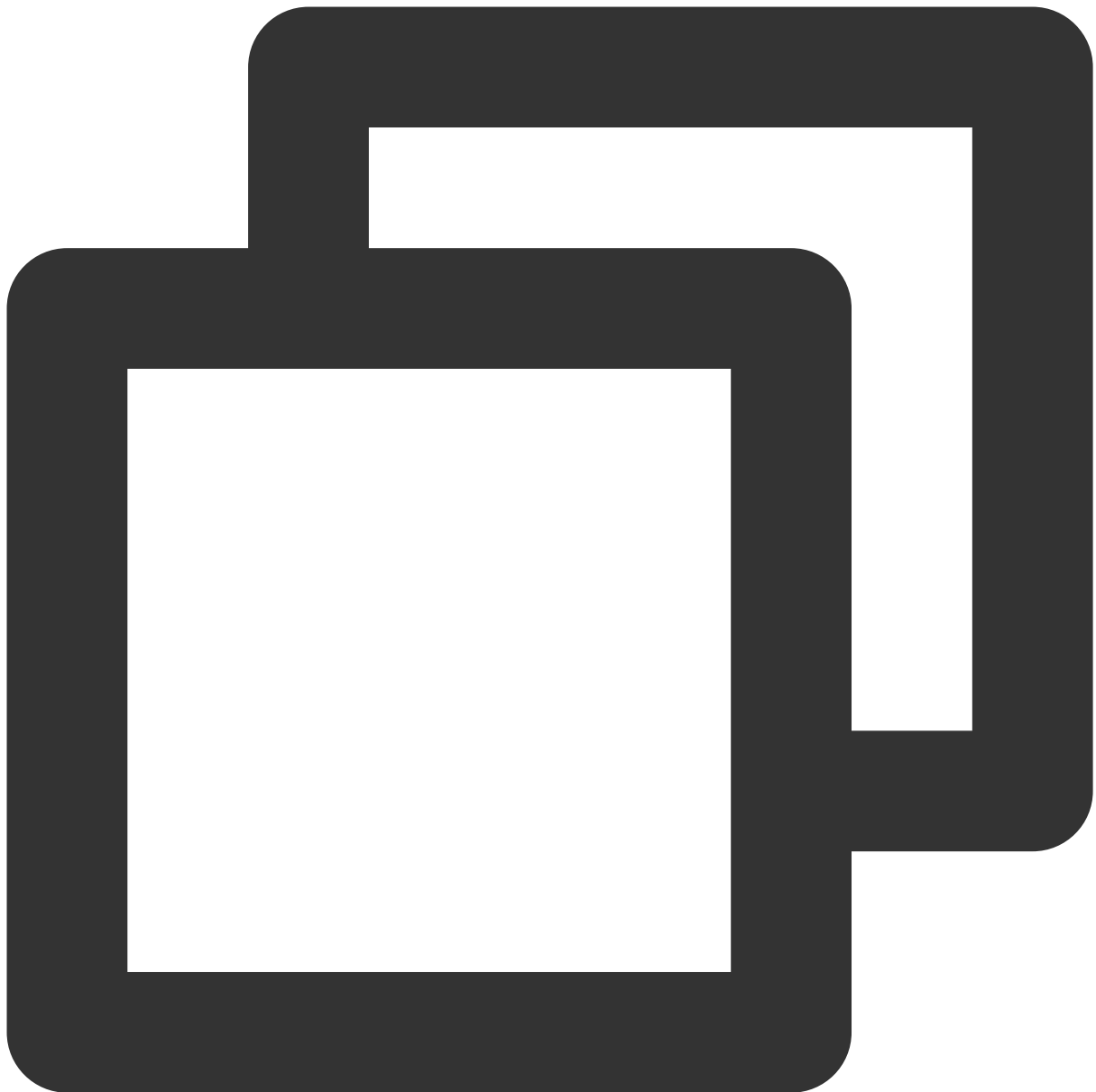


# 合并资源流式响应

最近更新时间：2024-01-25 11:34:51

该示例把三个视频合并为一个视频，在客户端按视频拼接顺序播放。实现了，使用边缘函数获取多个远程资源，并流式读取与拼接，最终流式响应客户端。

## 示例代码



```
async function combine(readableVideos, destination) {
  for (const readable of readableVideos) {
    // 流式响应
    await readable.pipeTo(destination, {
      preventClose: true
    });
  }

  const writer = destination.getWriter();
  writer.close();
  writer.releaseLock();
}
```

```
}

async function handleRequest(request) {
  // 视频片段地址
  const urls = [
    'https://laputa-1257579200.cos.ap-guangzhou.myqcloud.com/stream-01.mov',
    'https://laputa-1257579200.cos.ap-guangzhou.myqcloud.com/stream-02.mov',
    'https://laputa-1257579200.cos.ap-guangzhou.myqcloud.com/stream-03.mov',
  ];

  const requests = urls.map(url => fetch(url));
  const responses = await Promise.all(requests);
  // 获取视频片段的可读流
  const readableVideos = responses.map(res => res.body);

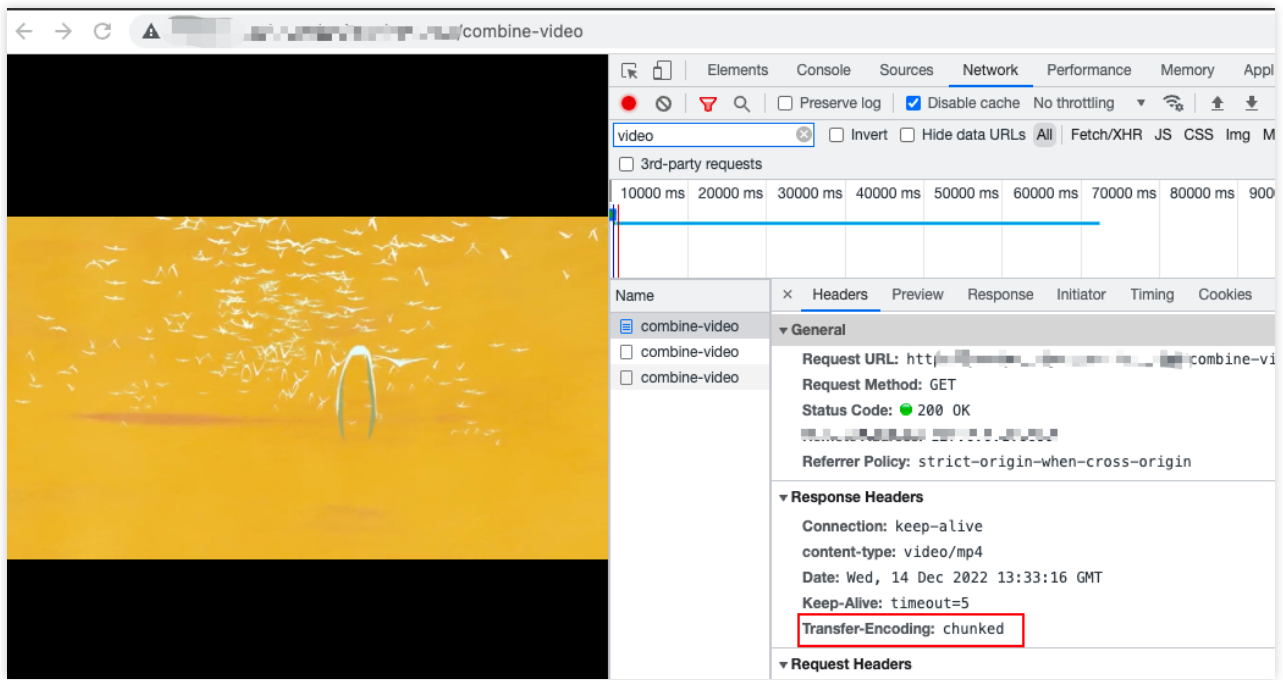
  const { readable, writable } = new TransformStream();
  // 合并视频
  combine(readableVideos, writable);

  return new Response(readable, {
    headers: {
      'content-type': 'video/mp4',
    }
  });
}

addEventListener('fetch', event => {
  event.respondWith(handleRequest(event.request));
});
```

## 示例预览

在浏览器地址栏中输入匹配到边缘函数触发规则的 URL，可预览到合并后的视频。查看响应头，视频以 **chunked** 方式传输。



## 相关参考

[Runtime APIs: TransformStream](#)

[Runtime APIs: Response](#)

# 防篡改校验

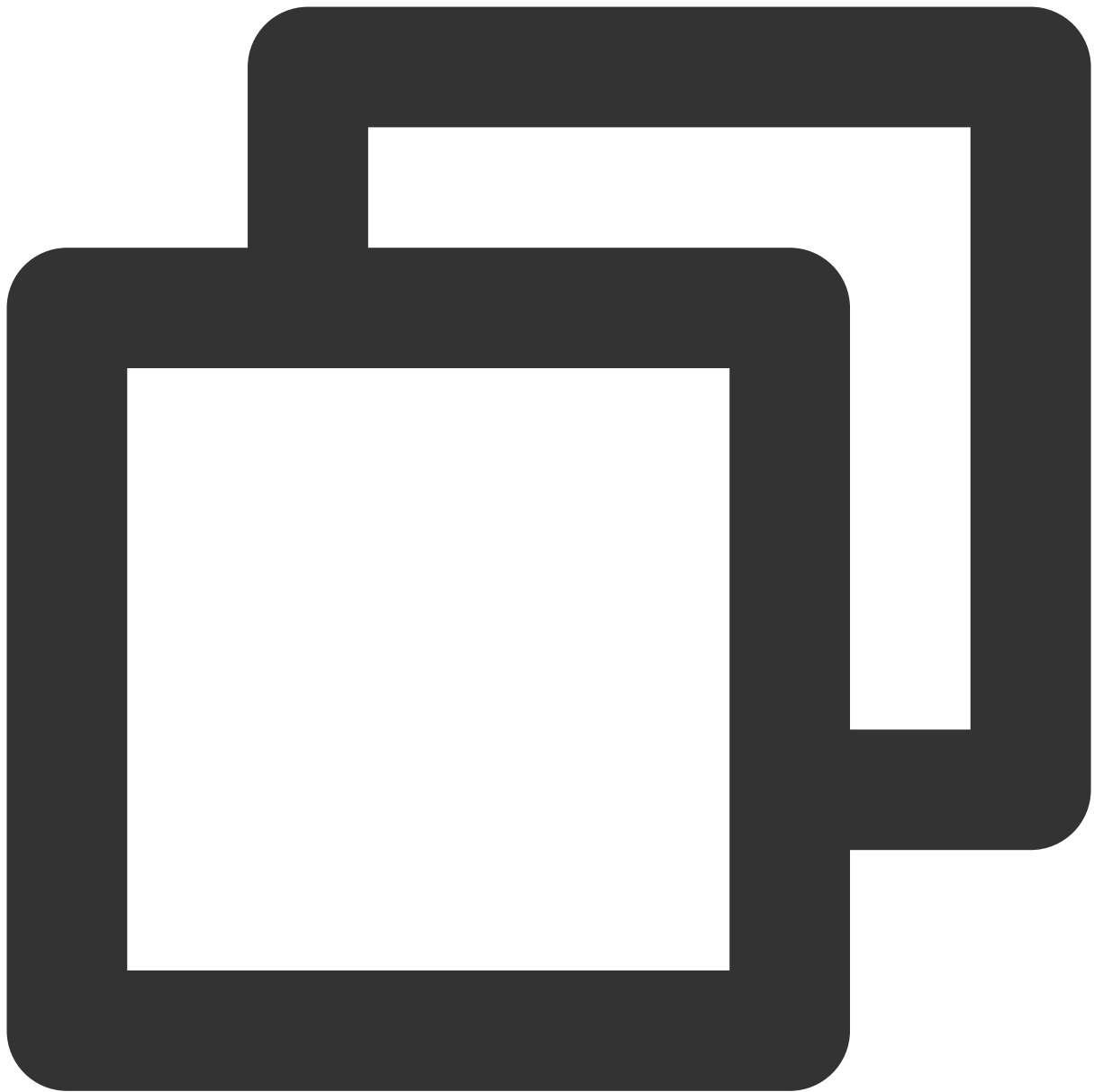
最近更新时间：2024-01-25 11:34:19

该示例通过计算 body 的 Sha-256 签名与源站生成的签名对比，若一致，则内容未被篡改，否则响应 416 状态码，表示内容被篡改。实现了，使用边缘函数校验源站响应内容是否被篡改。

## 注意：

该示例要求与源站配合使用，即源站需要具备一致的签名算法与防篡改规则。  
现网使用该示例提供的防篡改规则，需要在计算签名时加 Salt，避免攻击者破解。

## 示例代码



```
// 支持的文本文件类型
const textFileTypes = [
  'application/javascript',
  'text/html; charset=utf-8',
  'text/css; charset=utf-8',
  'text/xml; charset=utf-8'
];

// 支持的图片文件类型
const imageFileTypes = [
  'image/jpeg'
```

```
];

function uint8ArrayToHex(arr) {
  return Array.prototype.map
    .call(arr, (x) => (('0' + x.toString(16)).slice(-2)))
    .join('');
}

/**
 * 算法这里支持 MD5、SHA-1、SHA-256、SHA-384、SHA-512 之一，大小写不敏感。
 * 需要注意：这里源站在计算签名时，不要直接对源文件的数据直接进行签名，而是应该加入混淆数据，避免外
 * 然后在此处，也使用同样的方式进行计算对比，从而达到防篡改的目的
 **/
async function checkAndResponse(response, hash, algorithm) {
  const headers = response.headers;

  let checkHash = 'sorry! not match';
  let data = null;
  const contentType = headers.get('Content-Type');
  if (textFileTypes.includes(contentType) || imageFileTypes.includes(contentType))
    data = await response.arrayBuffer();
  }
  let ret = await crypto.subtle.digest({name: algorithm}, data);
  checkHash = uint8ArrayToHex(new Uint8Array(ret));

  headers.append(`X-Content-${algorithm}-Check`, checkHash);
  // 实时计算签名与源站签名对比，校验不通过返回 416，语义为无法满足用户的请求
  if (checkHash !== hash) {
    return new Response(null, {
      headers,
      status: 416
    });
  }

  return new Response(data, {
    headers,
    status: 200
  });
}

async function handleEvent(event) {
  // 获取源站返回内容，若 EdgeOne 节点存在缓存，则不会回源
  const response = await fetch(event.request);
  if (response.status === 200) {
    const headers = response.headers;
    // 源站内容签名头
    const hash = headers.get('X-Content-Sha256');
```

```

if (hash) {
    // 计算签名是否匹配，算法这里支持 MD5、SHA-1、SHA-256、SHA-384、SHA-512，大小写不敏感。
    return checkAndResponse(response, hash, 'Sha-256');
}

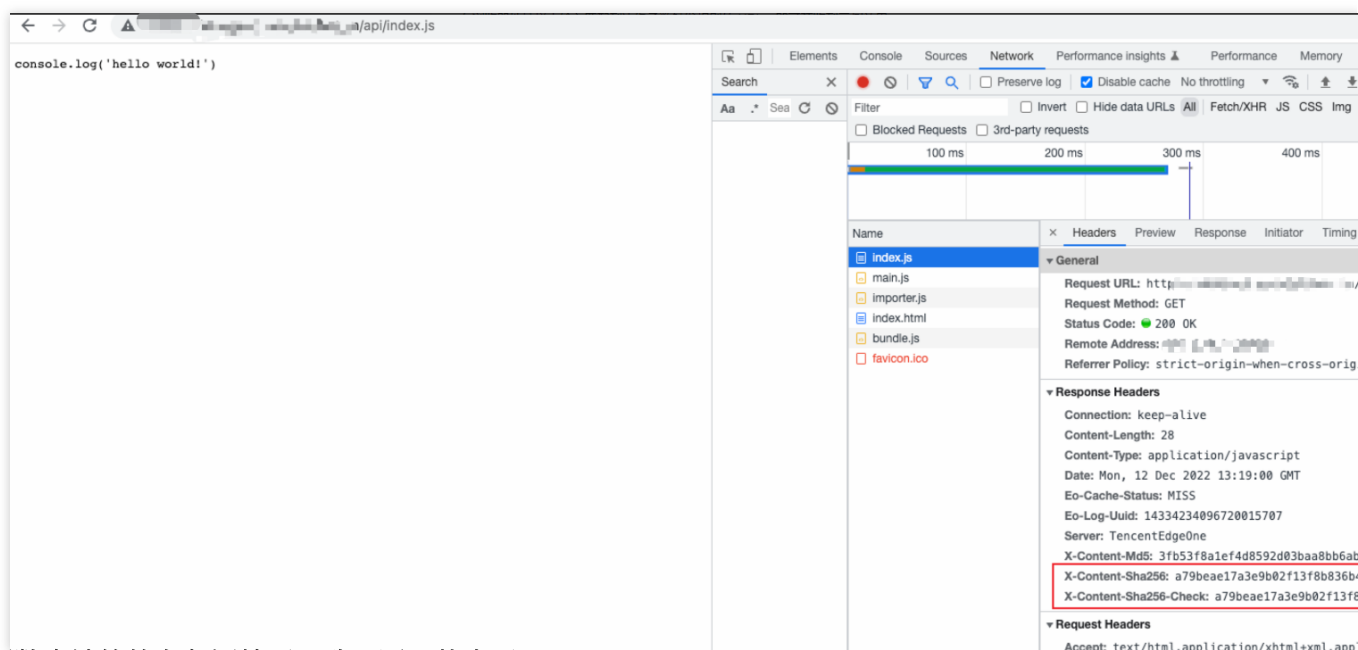
return response;
}

addEventListener('fetch', event => {
    event.respondWith(handleEvent(event));
});
    
```

## 示例预览

在浏览器地址栏中输入匹配到边缘函数触发规则的 URL，即可预览到示例效果。

边缘函数中计算签名与源站一致。

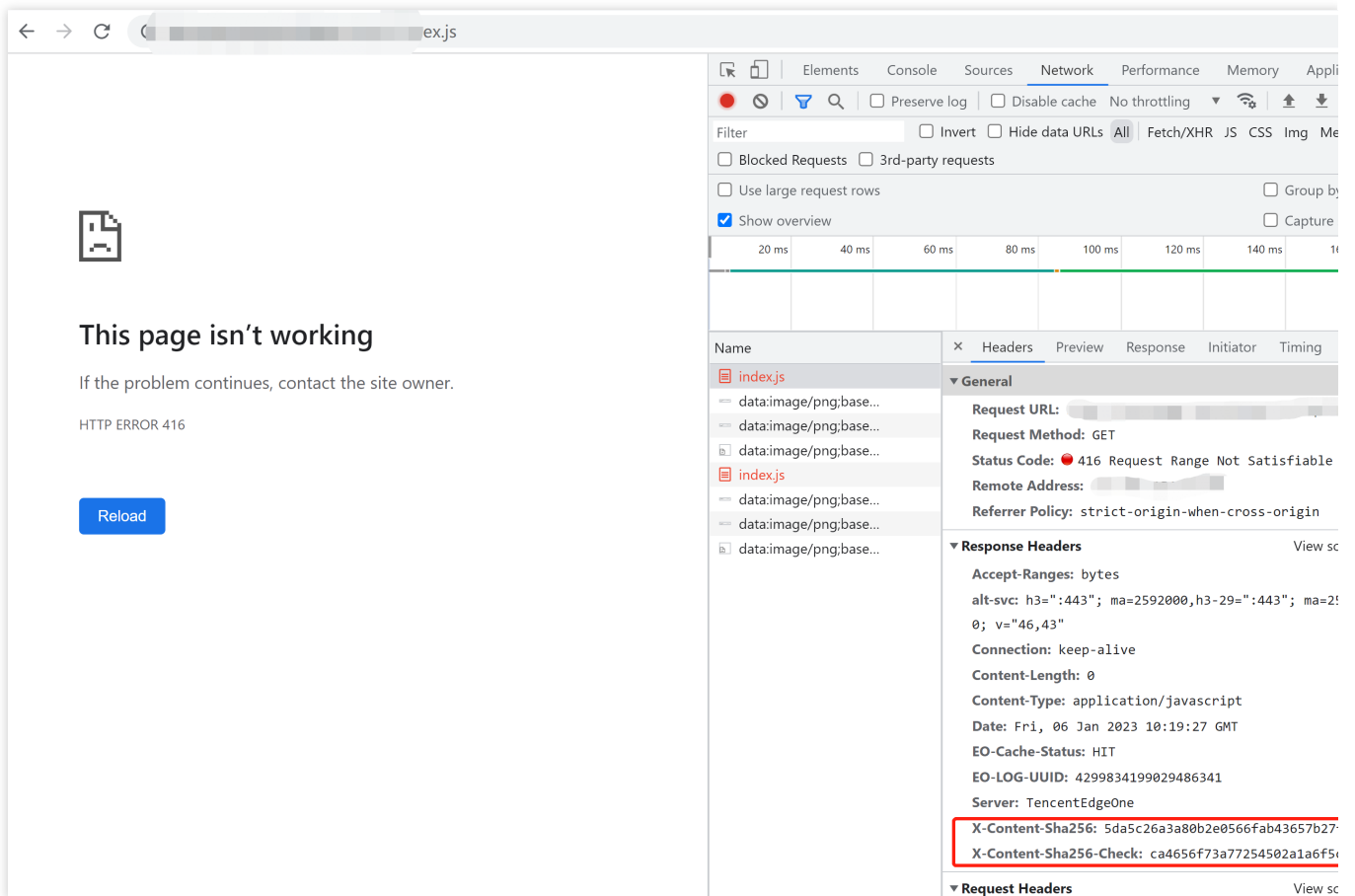


The screenshot shows a browser window with the address bar containing a URL ending in /api/index.js. The console on the left displays the message "hello world!". The Network tab on the right shows a request to index.js with a status of 200 OK. The response headers are visible, including:

- Connection: keep-alive
- Content-Length: 28
- Content-Type: application/javascript
- Date: Mon, 12 Dec 2022 13:19:00 GMT
- Eo-Cache-Status: MISS
- Eo-Log-Uuid: 14334234096720015707
- Server: TencentEdge0ne
- X-Content-Md5: 3fb53f8a1ef4d8592d03baa8bb6ab
- X-Content-Sha256: a79beae17a3e9b02f13f8b836b
- X-Content-Sha256-Check: a79beae17a3e9b02f13f8

边缘函数中计算签名与源站不一致，返回状态码 416。





## 相关参考

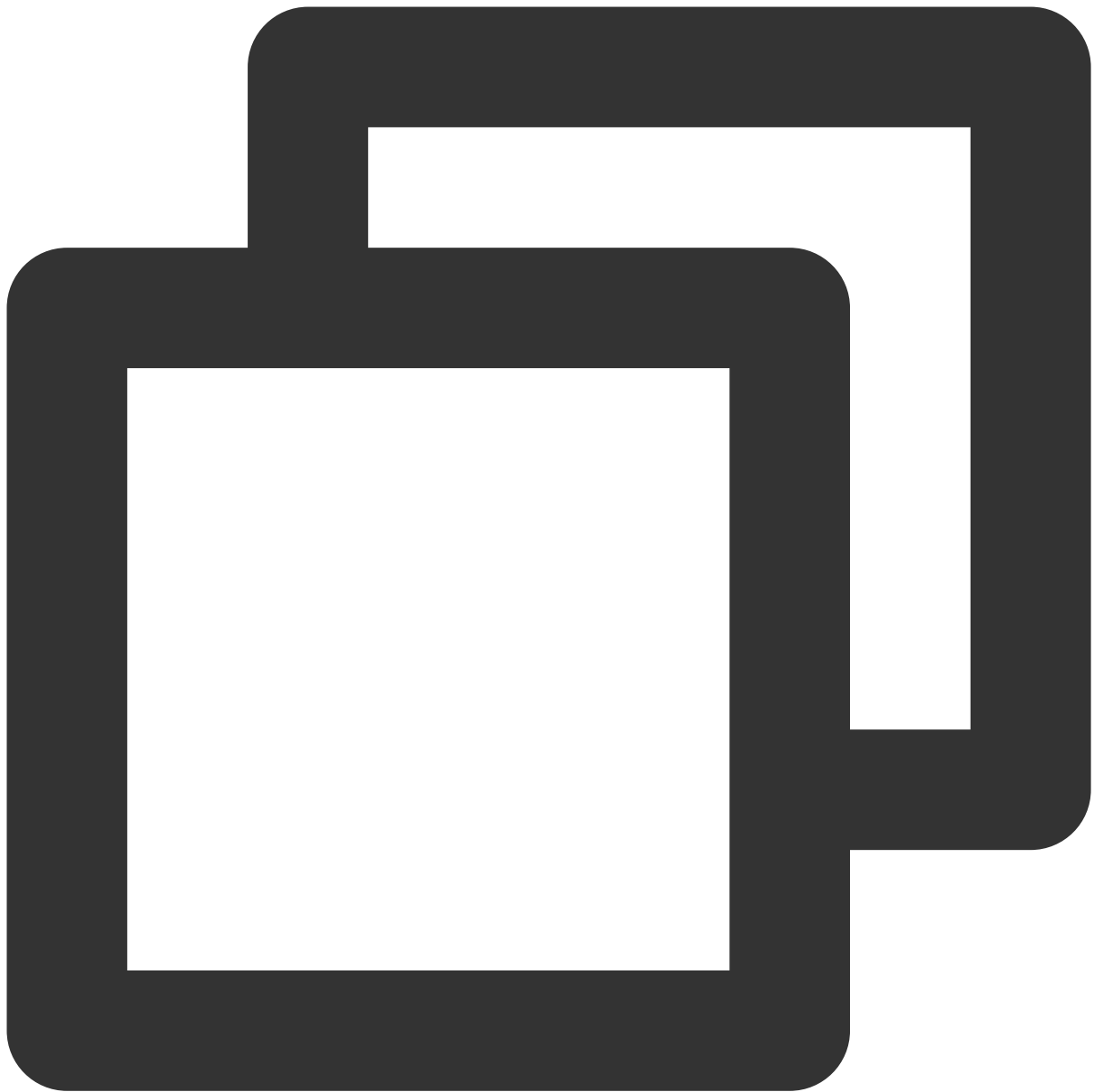
- [Runtime APIs: Fetch](#)
- [Runtime APIs: Web Crypto](#)
- [Runtime APIs: Headers](#)
- [Runtime APIs: Response](#)

# m3u8 改写与鉴权

最近更新时间：2023-04-13 11:14:30

该示例对 m3u8 改写，添加 Type A 鉴权，实现访问 .m3u8 与 .ts 片段资源的权限控制。开发者可根据需要修改代码，支持其他鉴权方式。

## 示例代码



```
// Type A鉴权私钥, 请自行设定并防止泄漏
const PK = '0123456789';
// 加密校验 key 的有效时间 (秒)
const TTL = 60;
const KEY_NAME = 'key';
const UID = 0;
const SUFFIX_LIST = ['.m3u8', '.ts'];

addEventListener('fetch', (event) => {
  event.respondWith(handleEvent(event));
});
```

```
async function handleEvent(event) {
  try {
    const { request } = event;
    const urlInfo = new URL(request.url);
    const suffix = getSuffix(urlInfo.pathname);

    // 检查文件后缀为：.m3u8, .ts
    if (!SUFFIX_LIST.includes(suffix)) {
      return fetch(request);
    }

    // Type A 鉴权
    const checkResult = await checkTypeA(urlInfo);
    if (!checkResult.flag) {
      return new Response(checkResult.message, {
        status: 403,
        headers: {
          'X-Auth-Err': checkResult.message
        },
      });
    }

    // 改写 .m3u8 并响应
    if (suffix === '.m3u8') {
      return fetchM3u8({
        request,
        querySign: {
          basePath: urlInfo.pathname.substring(0, urlInfo.pathname.lastIndexOf('/'))
            ...checkResult.querySign,
        }
      });
    }

    // 响应 .ts 资源
    if (suffix === '.ts') {
      return fetchTs(request);
    }
  } catch (error) {
    return new Response(error.stack, { status: 544 });
  }

  return fetch(request);
}

async function checkTypeA(urlInfo) {
  const sign = urlInfo.searchParams.get(KEY_NAME) || '';

```

```
const elements = sign.split('-');

if (elements.length !== 4) {
  return {
    flag: false,
    message: 'Invalid Sign Format',
  };
}

const [ts, rand, uid, md5hash] = elements;
if (ts === undefined || rand === undefined || uid === undefined || md5hash === un
  return {
    flag: false,
    message: 'Invalid Sign Format',
  };
}

if (!isNumber(ts)) {
  return {
    flag: false,
    message: 'Sign Expired',
  };
}

if (Date.now() > (Number(ts) + TTL) * 1000) {
  return {
    flag: false,
    message: 'Sign Expired',
  };
}

const hash = await md5([urlInfo.pathname, ts, rand, uid, PK].join('-'));
if (hash !== md5hash) {
  return {
    flag: false,
    message: 'Verify Sign Failed',
  };
}
return {
  flag: true,
  message: 'success',
  querySign: {
    rand,
    uid,
    md5hash,
    ts,
  },
},
```

```
};
}

async function fetchM3u8({ request, querySign }) {
  request.headers.delete('Accept-Encoding');
  let response = null;
  try {
    response = await fetch(request);
    if (response.status !== 200) {
      return response;
    }
  } catch (error) {
    return new Response('', {
      status: 504,
      headers: { 'X-Fetch-Err': 'Invalid Origin' }
    });
  }

  const content = await response.text();
  const lines = content.split('\n');

  const contentArr = await Promise.all(
    lines.map(line => rewriteLine({ line, querySign }))
  );

  return new Response(contentArr.join('\n'), response);
}

async function fetchTs(request) {
  let response = null;
  try {
    response = await fetch(request);
    if (response.status !== 200) {
      return response;
    }
  } catch (error) {
    return new Response('', {
      status: 504,
      headers: { 'X-Fetch-Err': 'Invalid Origin' }
    });
  }
  return response;
}

async function rewriteLine({ line, querySign }) {
  // 跳过空行
  if (/^\s*$/.test(line)) {
```

```

    return line;
}

if (line.charAt(0) === '#') {
    // 处理 #EXT-X-MAP
    if (line.startsWith('#EXT-X-MAP')) {
        const key = await createSign(querySign, line);
        line = line.replace(/URI="([\^"]+)"/, (matched, p1) => {
            return p1 ? matched.replace(p1, `${p1}?key=${key}`) : matched;
        });
    }
    return line;
}

const key = await createSign(querySign, line);

return `${line}?${KEY_NAME}=${key}`;
}

async function createSign(querySign, line) {
    const { ts, rand, uid = 0 } = querySign;
    const pathname = `${querySign.basePath}/${line}`;

    const md5hash = await md5([pathname, ts, rand, uid, PK].join('-'));
    const key = [ts, rand, uid, md5hash].join('-');

    return key;
}

function getSuffix(pathname) {
    const suffix = pathname.match(/\\.m3u8|\\.ts$/);
    return suffix ? suffix[0] : null;
}

function isNumber(num) {
    return Number.isInteger(Number(num));
}

function bufferToHex(arr) {
    return Array.prototype.map
        .call(arr, (x) => (x >= 16 ? x.toString(16) : '0' + x.toString(16)))
        .join('');
}

async function md5(text) {
    const buffer = await crypto.subtle.digest('MD5', TextEncoder().encode(text));
    return bufferToHex(new Uint8Array(buffer));
}


```

```
}

```

## 示例预览

在浏览器地址栏中输入匹配到边缘函数触发规则的 URL（如：<http://www.example.com/index.m3u8?key=1678873033-123456-0-32f4xxxxcabcxxx1602xxxx6756d8f4>），即可预览到示例效果。

<pre> 1 #EXTM3U 2 #EXT-X-VERSION:3 3 #EXT-X-TARGETDURATION:6 4 #EXT-X-MEDIA-SEQUENCE:0 5 #EXTINF:1.416667, 6 0-282375.ts 7 #EXTINF:2.875000, 8 282376-675671.ts 9 #EXTINF:1.708333, 10 675672-905595.ts 11 #EXTINF:3.208333, 12 3.ts 13 #EXTINF:2.000000, 14 4.ts 15 #EXTINF:2.000000, 16 5.ts 17 #EXTINF:2.000000, 18 6.ts 19 #EXTINF:2.000000, 20 7.ts 21 #EXTINF:3.708333, 22 8.ts 23 #EXTINF:5.791667, 24 9.ts 25 #EXTINF:3.375000, 26 10.ts 27 #EXT-X-ENDLIST </pre>		<pre> 1 #EXTM3U 2 #EXT-X-VERSION:3 3 #EXT-X-TARGETDURATION:6 4 #EXT-X-MEDIA-SEQUENCE:0 5 #EXTINF:1.416667, 6 0-282375.ts?key=1671181186-6767974945631037-0-58d3457cef4a1e3ce68a471b6966ef74 7 #EXTINF:2.875000, 8 282376-675671.ts?key=1671181186-6767974945631037-0-6c0a83fb9313c836734df13cad56d18a 9 #EXTINF:1.708333, 10 675672-905595.ts?key=1671181186-6767974945631037-0-948a4eaf14828bf8ff57120983b3b474 11 #EXTINF:3.208333, 12 3.ts?key=1671181186-6767974945631037-0-e10778ac5e94134a6e762de322cdc25e 13 #EXTINF:2.000000, 14 4.ts?key=1671181186-6767974945631037-0-3fb747e907e623822fbfbc9cb066768 15 #EXTINF:2.000000, 16 5.ts?key=1671181186-6767974945631037-0-193aa1d267b6892a8f89754cf9cf68a 17 #EXTINF:2.000000, 18 6.ts?key=1671181186-6767974945631037-0-e5e95cf873c8911ab7d0668716ea518a 19 #EXTINF:2.000000, 20 7.ts?key=1671181186-6767974945631037-0-bd882ed14c331d31c368682b436cc5f1 21 #EXTINF:3.708333, 22 8.ts?key=1671181186-6767974945631037-0-702fdc4c39f7499971ce3e0174cdd73a 23 #EXTINF:5.791667, 24 9.ts?key=1671181186-6767974945631037-0-eb91c87cadbf372adf95900f442aeace 25 #EXTINF:3.375000, 26 10.ts?key=1671181186-6767974945631037-0-2cfe4f6b1ea682820e020918e121ac71 27 #EXT-X-ENDLIST </pre>
---	---	--

## 相关参考

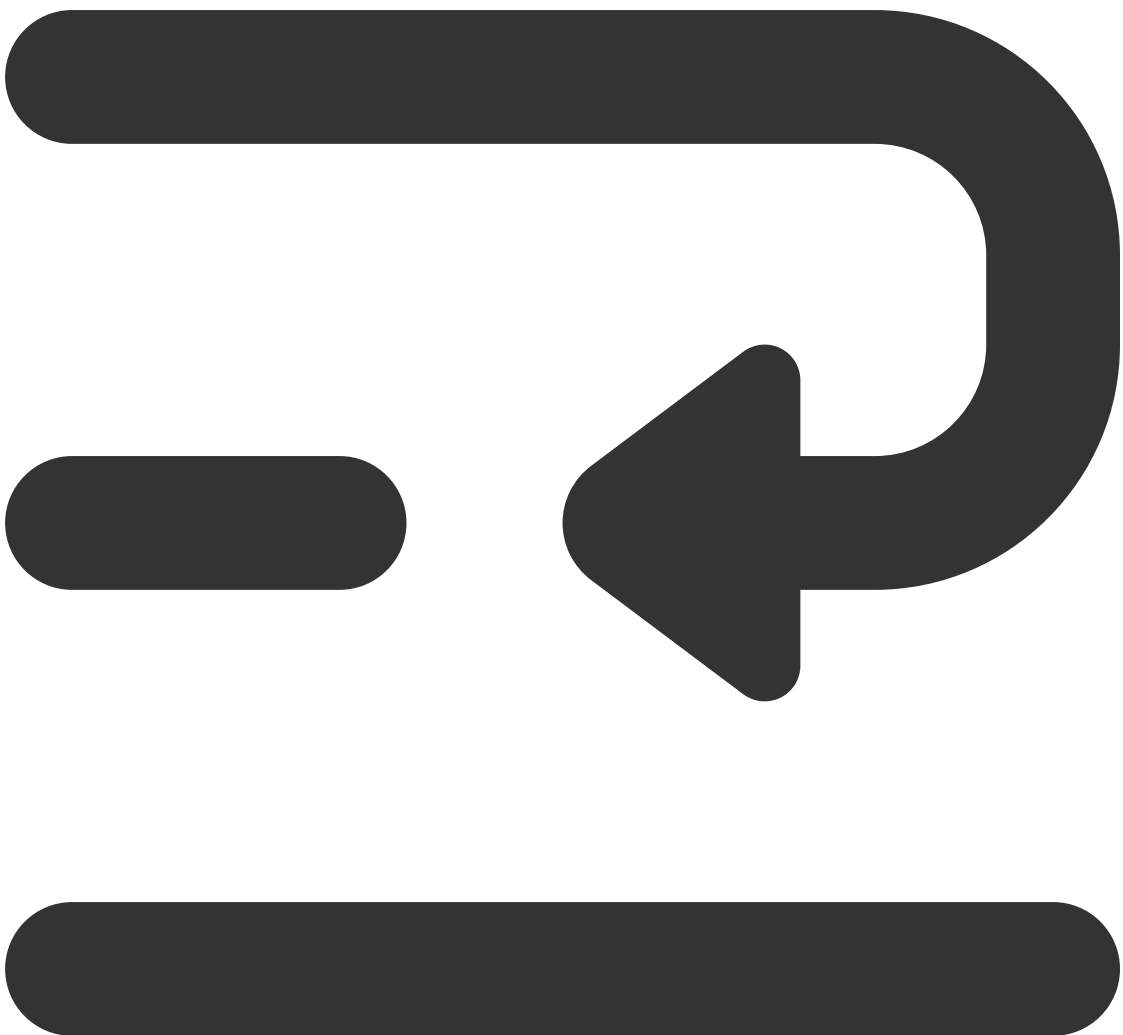
- [Runtime APIs: Fetch](#)
- [Runtime APIs: Web Crypto](#)
- [Runtime APIs: Response](#)
- [Type A 鉴权原理](#)

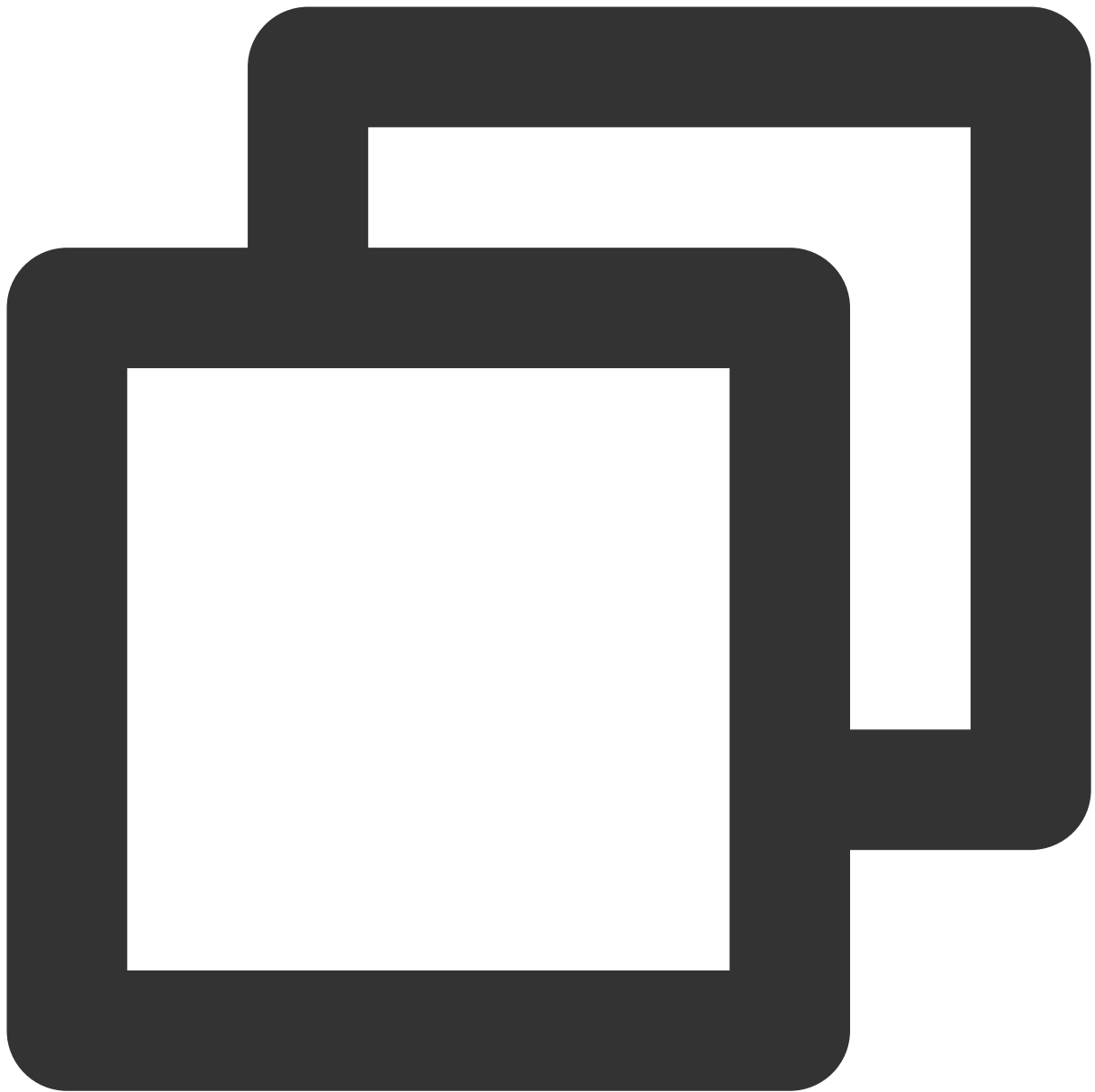


# 图片自适应缩放

最近更新时间：2023-08-02 16:46:01

该示例通过获取请求头中的 `User-Agent` 信息，来识别客户端类型，并使用 [fetch API](#) 获取源站图片，根据客户端类型对图片进行缩放，以实现图片自适应缩放的效果。这种实现方式可以提高网站的用户体验，使得图片在不同的设备上都能够以最佳的尺寸呈现。





```
addEventListener('fetch', event => {  
  // 当函数代码抛出未处理的异常时，边缘函数会将此请求转发回源站  
  event.passThroughOnException();  
  event.respondWith(handleEvent(event));  
});  
  
async function handleEvent(event) {  
  const { request } = event;  
  const urlInfo = new URL(request.url);  
  const userAgent = request.headers.get('user-agent');
```

```
// 请求非图片资源
if (!/\.\.(jpe?g|png)$/.test(urlInfo.pathname)) {
  return fetch(request);
}

// 移动端图片宽度
let width = 480;
const isPcClient = isPc(userAgent);

// PC 端图片宽度
if (isPcClient) {
  width = 1280;
}

// 图片缩放
const response = await fetch(request, {
  eo: {
    image: {
      width,
    }
  }
});

// 设置响应头
response.headers.set('x-ef-client', isPcClient ? 'pc' : 'mobile');
return response;
}

// 请求客户端类型判断
function isPc(userAgent) {
  const regex = /(phone|pad|pod|iPhone|iPod|ios|iPad|Android|Mobile|BlackBerry|IEMo

  if(regex.test(userAgent)) {
    return false;
  }

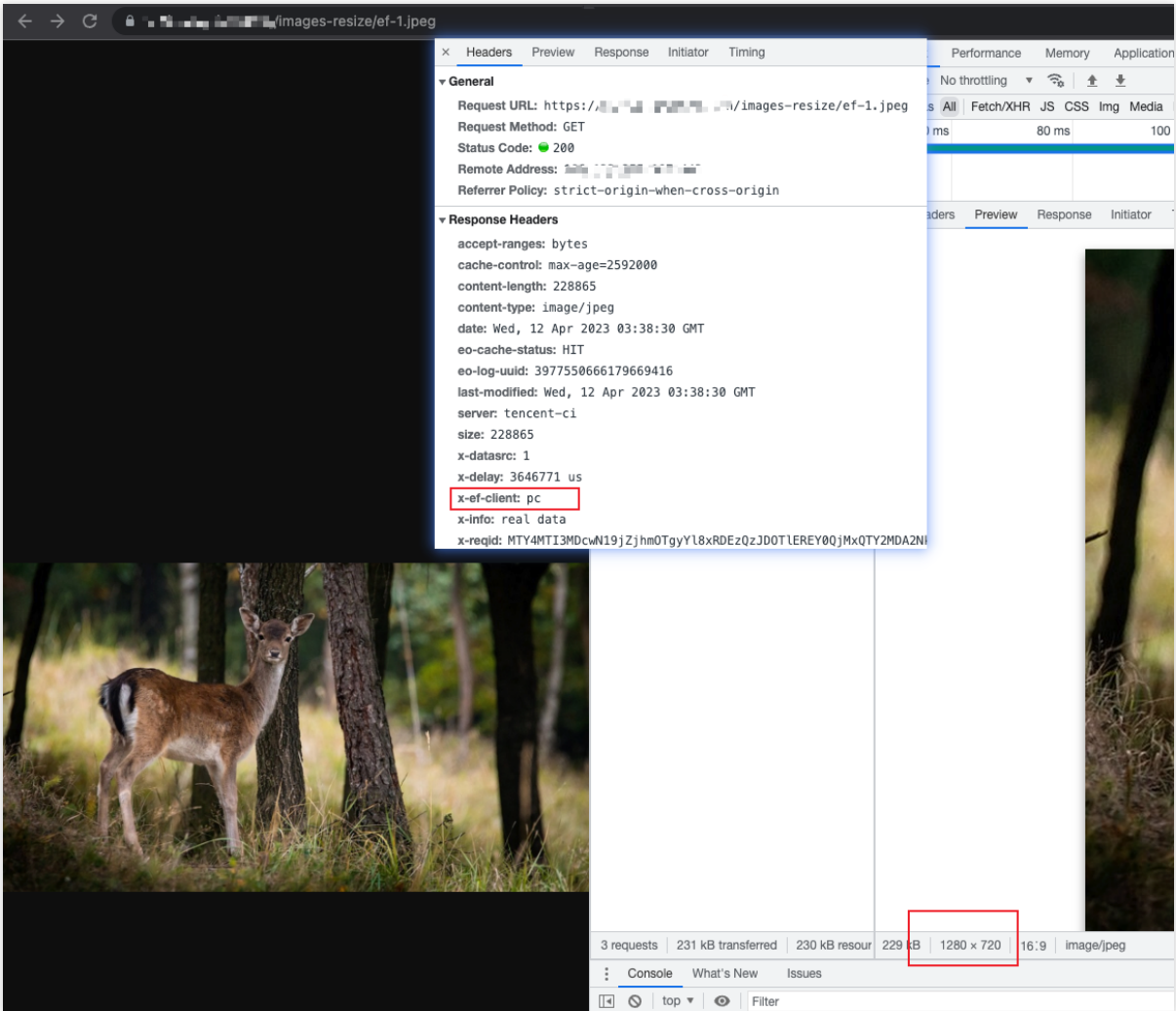
  return true;
}
```

## 示例预览

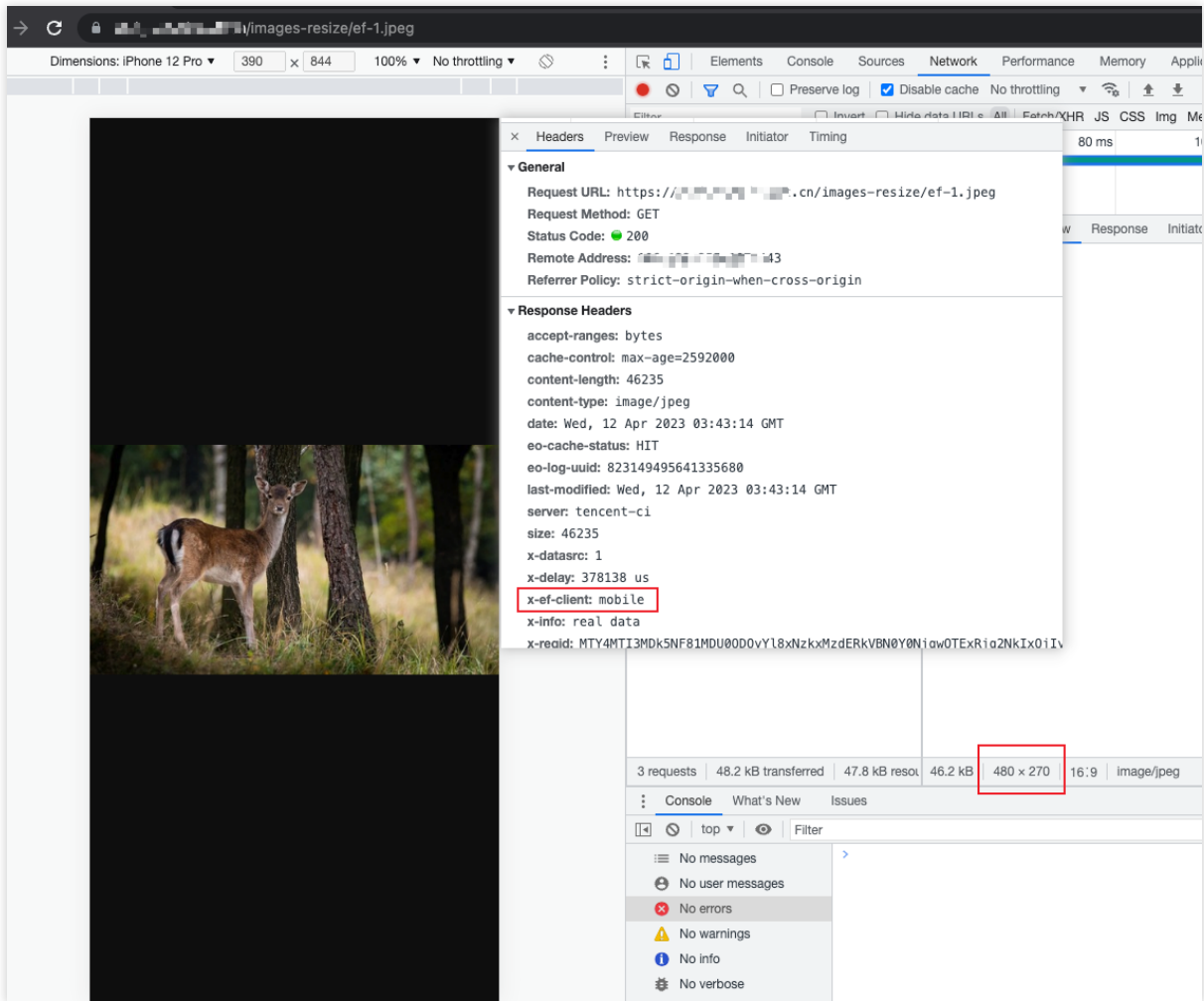
在 PC 端与移动端的浏览器地址栏中输入匹配到边缘函数触发规则的 URL

(如：`https://example.com/images-resize/ef-1.jpeg` )，即可预览到示例效果。

使用 PC 端访问图片，图片缩放为 1280 x 720。



使用移动端访问图片，图片缩放为 480 x 270。



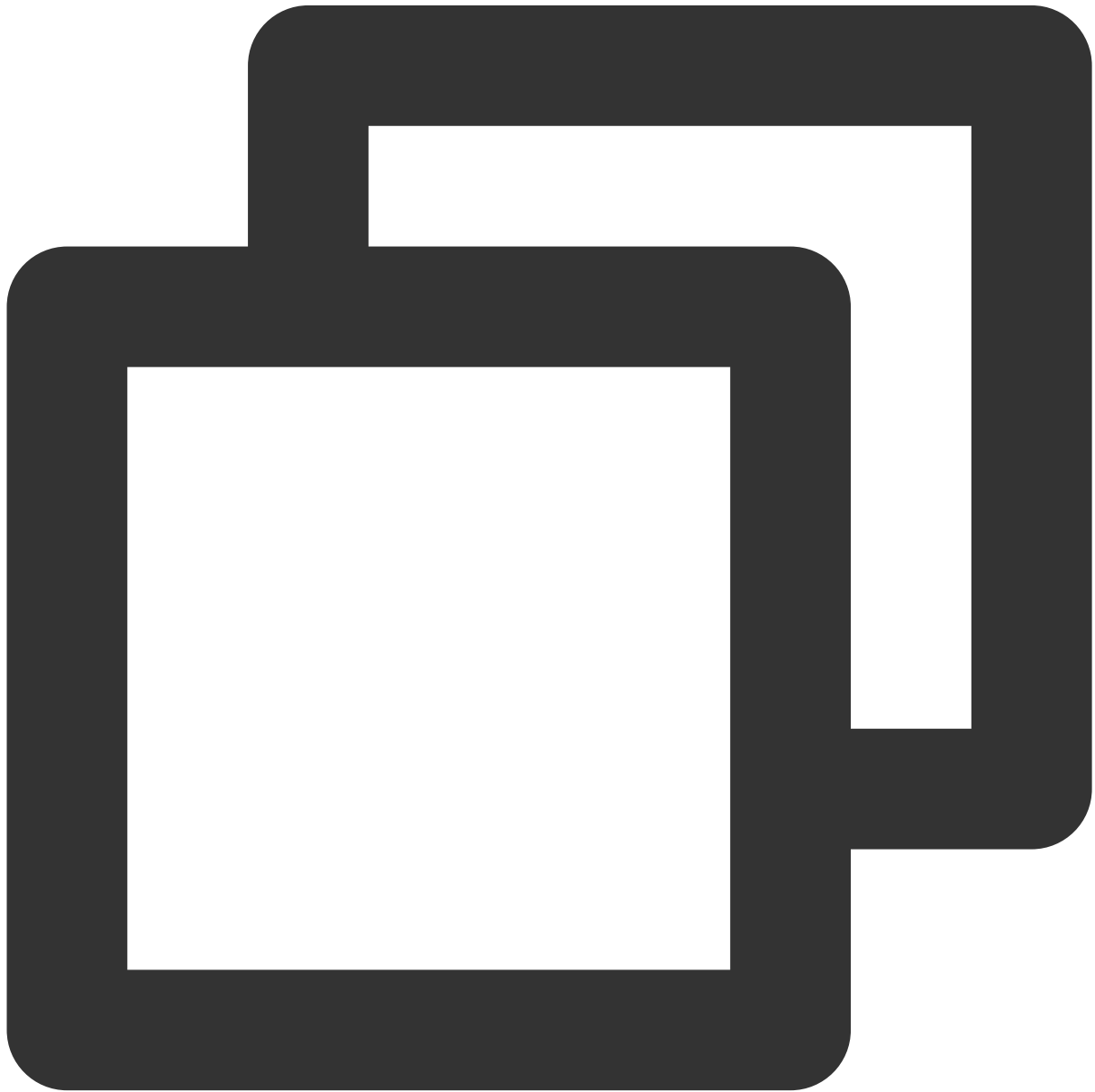
## 相关参考

- [Runtime APIs: Fetch](#)
- [Runtime APIs: Headers](#)
- [Runtime APIs: Response](#)
- [Runtime APIs: Request](#)

# 图片自适应 WebP

最近更新时间：2023-10-30 17:41:20

该示例通过对请求头 `Accept` 判断，如果包含 `image/webp`，边缘函数会将图片格式自动转换为 `WebP` 格式，并缓存在 EdgeOne 边缘节点。若您的 Web 应用展示了大量的 `PNG`，`JPEG` 格式图片，期望在边缘自动优化图片，减少流量带宽成本，可使用边缘函数实现平滑升级，把 `PNG`，`JPEG` 格式图片自动转换为 `WebP` 格式，并且业务代码 0 改动。更多图片转换格式，详情请参考 [ImageProperties](#)。



```
async function handleEvent(event) {
```

```
const { request } = event;

// 获取客户端支持的图片类型
const accept = request.headers.get('Accept');
const option = { eo: { image: {} } };

// 检查客户端是否支持 WebP 格式的图片, 若不支持响应原图
if (accept && accept.includes('image/webp')) {
  option.eo.image.format = 'webp';
}

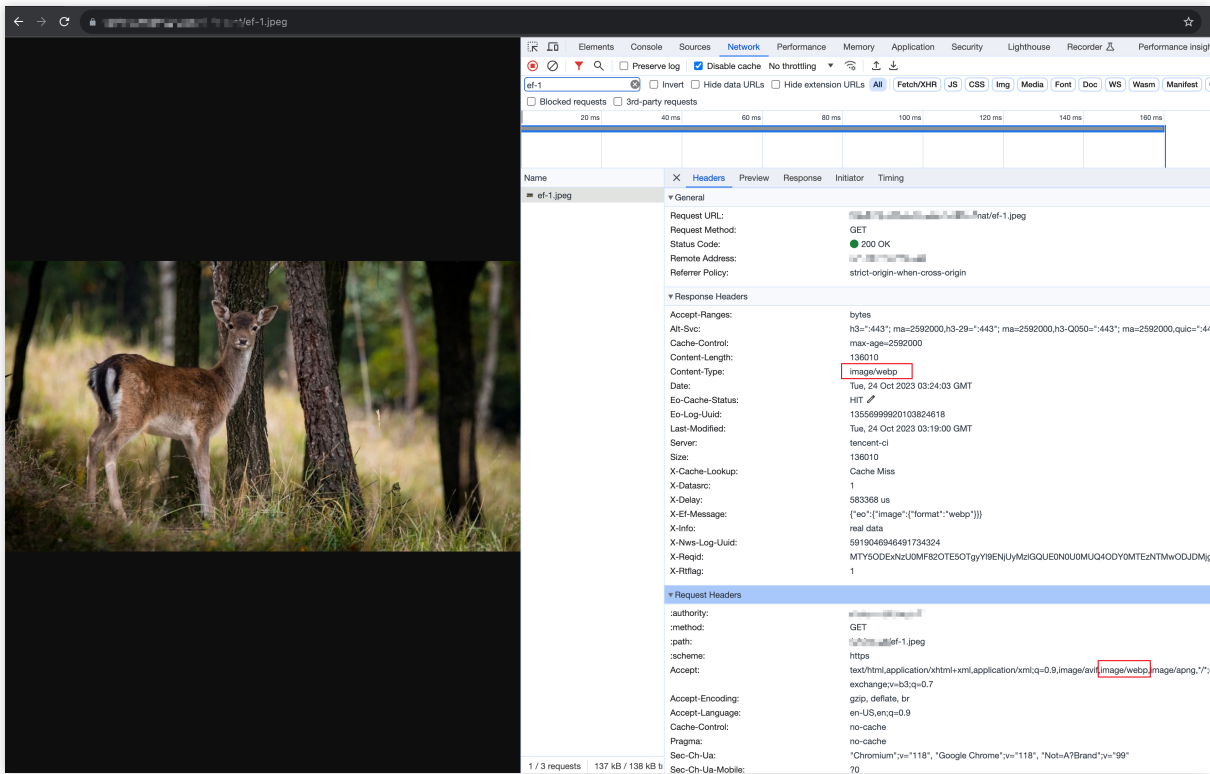
const response = await fetch(request, option);
return response;
}

addEventListener('fetch', event => {
  // 当函数代码抛出未处理的异常时, 边缘函数会将此请求转发回源站
  event.passThroughOnException();
  event.respondWith(handleEvent(event));
});
```

## 示例预览

在 PC 端与移动端的浏览器地址栏中输入匹配到边缘函数触发规则的 URL

(如: `https://example.com/images-format/ef-1.jpeg`), 即可自动转换为 Webp 格式图片。



## 相关参考

- [Runtime APIs: Fetch](#)
- [Runtime APIs: Headers](#)
- [Runtime APIs: Request](#)
- [Runtime APIs: FetchEvent](#)

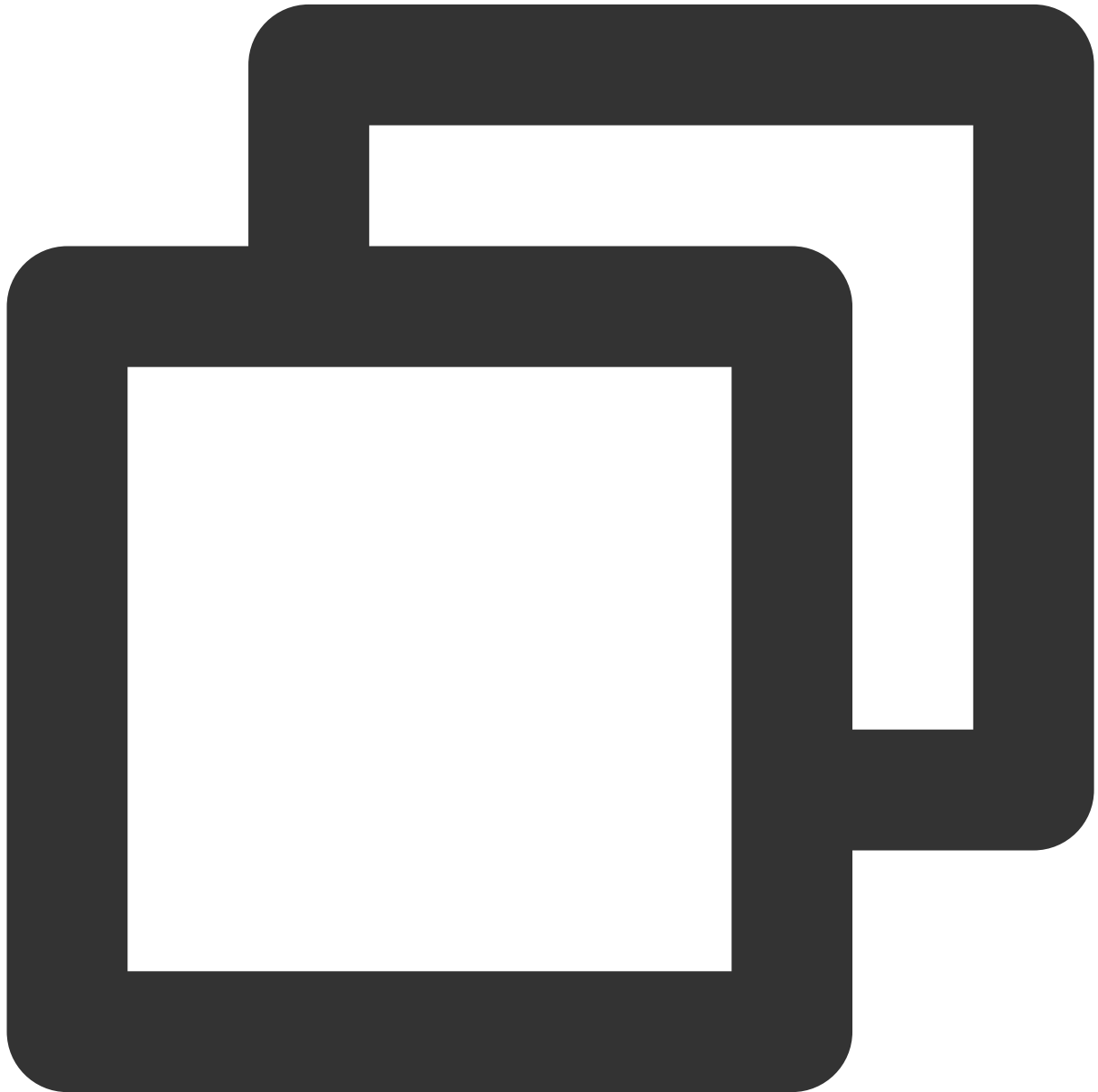


# 自定义 Referer 防盗链

最近更新时间：2023-10-30 17:44:42

Referer 防盗链技术是网站为了保护自己的资源，阻止其他网站非法引用其内容而采用的一种策略。该示例通过检查

HTTP 请求头中的 Referer 字段来判断请求来源，您可以灵活自定义改 Referer 的匹配规则，如果 Referer 不存在或者与允许的域名列表不匹配，边缘函数将拒绝请求并返回 403 状态码。



```
async function handleRequest(request) {  
  // 获取 Referer
```

```
const referer = request.headers.get('Referer');

// Referer 为空, 禁止访问
if (!referer) {
  return new Response(null, { status: 403 });
}

const urlInfo = new URL(request.url);
const refererRegExp = new RegExp(`^https?:\\/\\/\\/${urlInfo.hostname}\\/t-[0-9a-z]{

// Referer 不在白名单, 禁止访问
if (!refererRegExp.test(referer)) {
  return new Response(null, { status: 403 });
}

// 正常请求, 访问 EdgeOne 节点缓存或回源
return fetch(request);
}

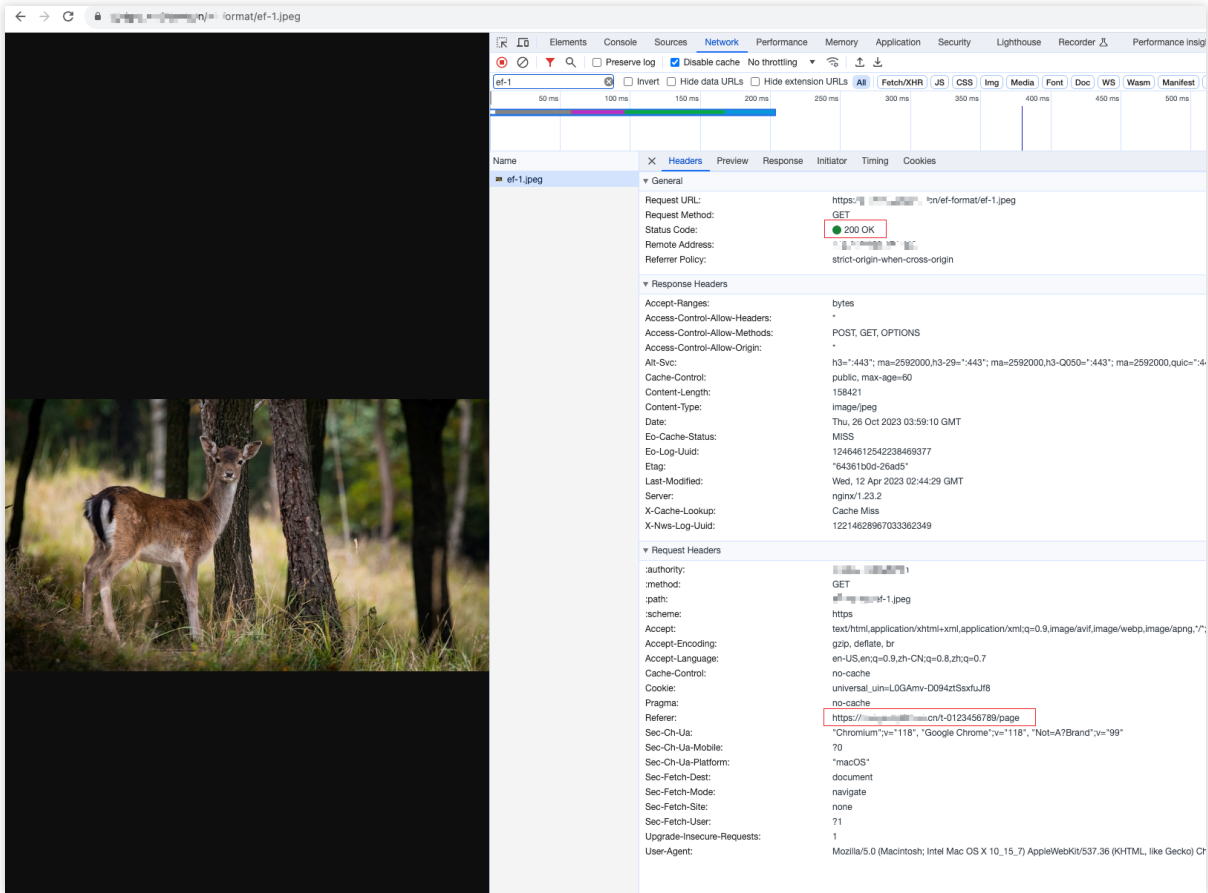
addEventListener('fetch', event => {
  // 当函数代码抛出未处理的异常时, 边缘函数会将此请求转发回源站
  event.passThroughOnException();
  event.respondWith(handleRequest(event.request));
});
```

## 示例预览

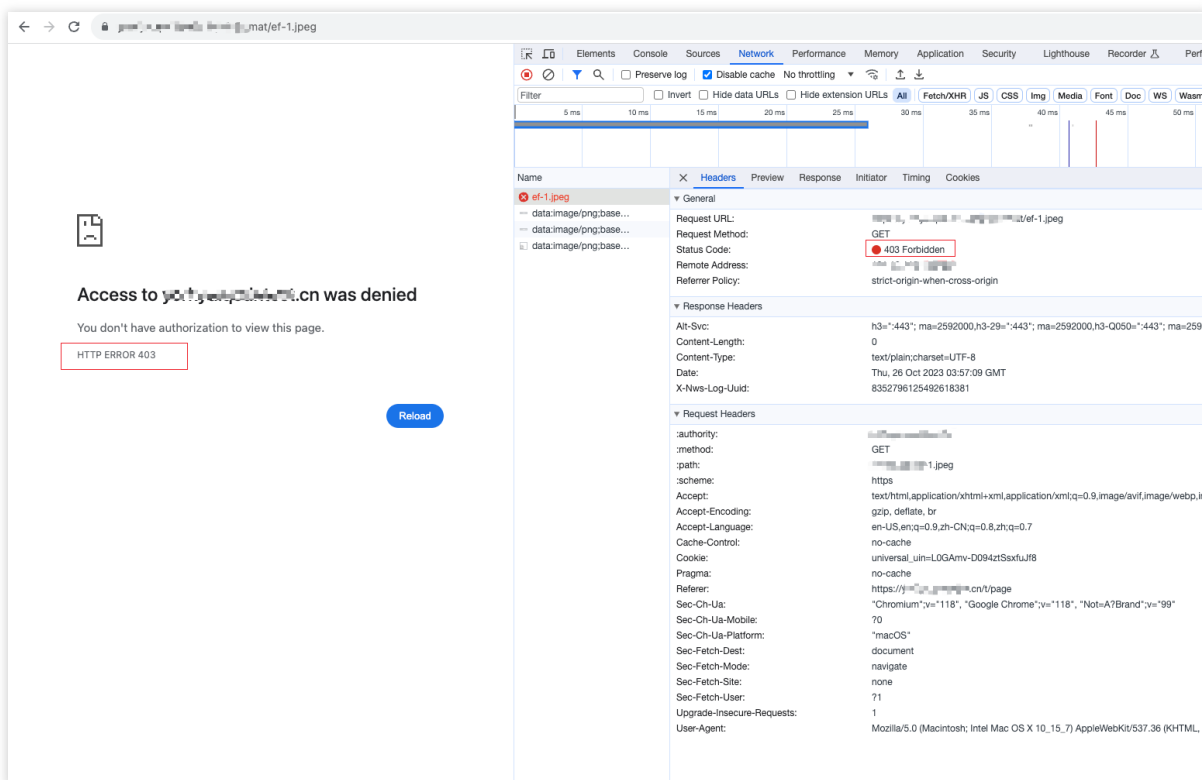
在 PC 端与移动端的浏览器地址栏中输入匹配到边缘函数触发规则的 URL

(如: `https://example.com/images/ef-1.jpeg`), 即可预览到示例效果。

HTTP 请求头 `Referer` 为 `https://example.com/t-0123456789/page`, 边缘函数正常响应图片。



HTTP 请求头 `Referer` 不在白名单，边缘函数识别为盗链并响应 403 状态码。



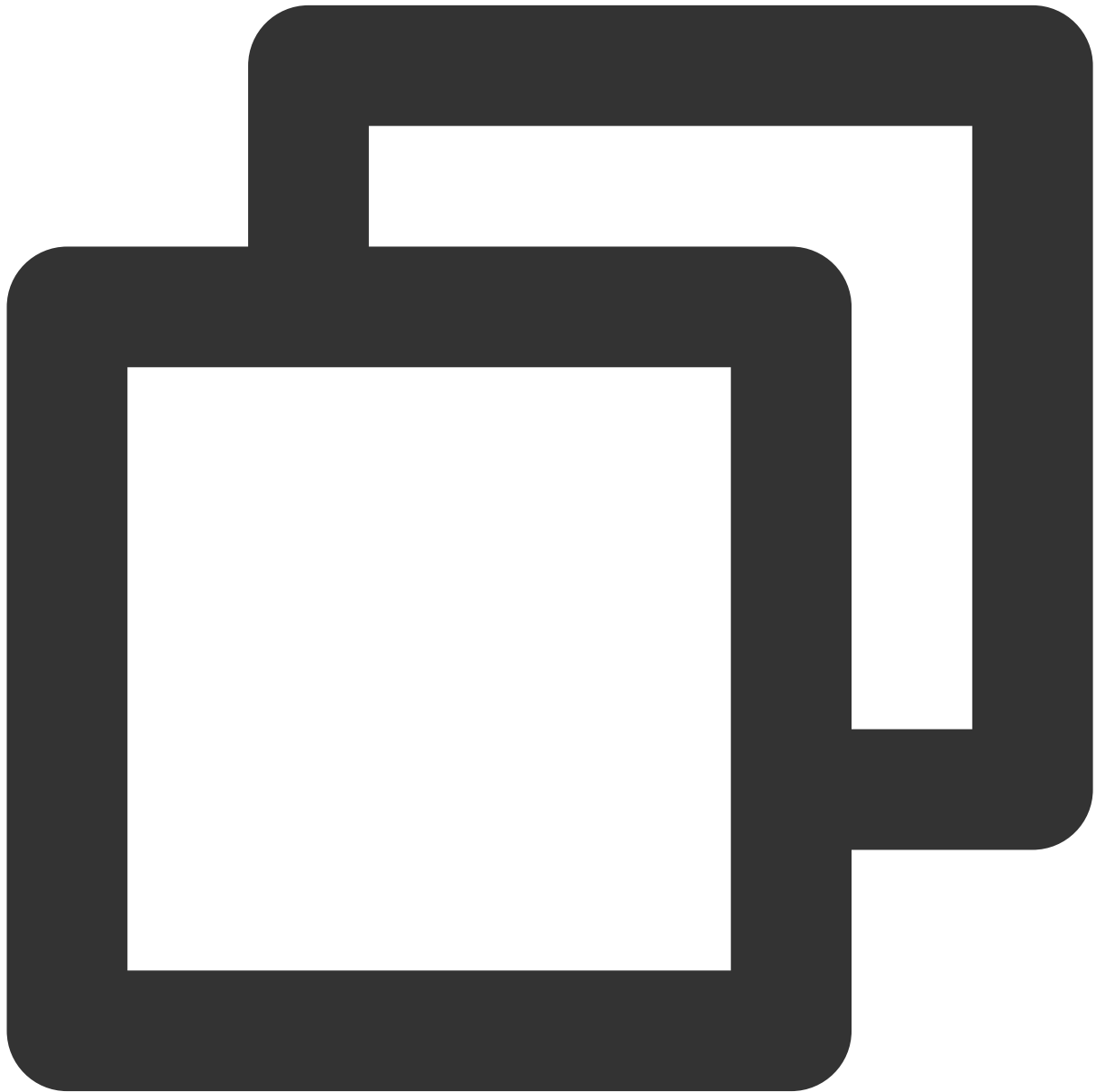
## 相关参考

- [Runtime APIs: Fetch](#)
- [Runtime APIs: Headers](#)
- [Runtime APIs: Response](#)

# 远程鉴权

最近更新时间：2023-10-30 17:45:57

为了避免客户的资源被非法用户访问，该示例将请求转发至客户指定的远程鉴权服务器，由该鉴权服务器对用户请求进行校验，边缘函数根据远程鉴权服务器返回的校验结果来决定是否允许访问目标资源，若鉴权不通过，则响应客户端 403 状态码。



```
async function handleRequest(request) {  
  // 远程鉴权 API 地址
```

```
const checkAuthUrl = 'https://www.example.com/';
// 发起远程鉴权
const checkAuthRes = await fetch(checkAuthUrl);

// 鉴权通过, 正常访问资源
if (checkAuthRes.status === 200) {
  return fetch(request, {
    headers: request.headers,
  });
}

// 鉴权不通过, 禁止访问资源
return new Response(null, {
  status: 403
});
}

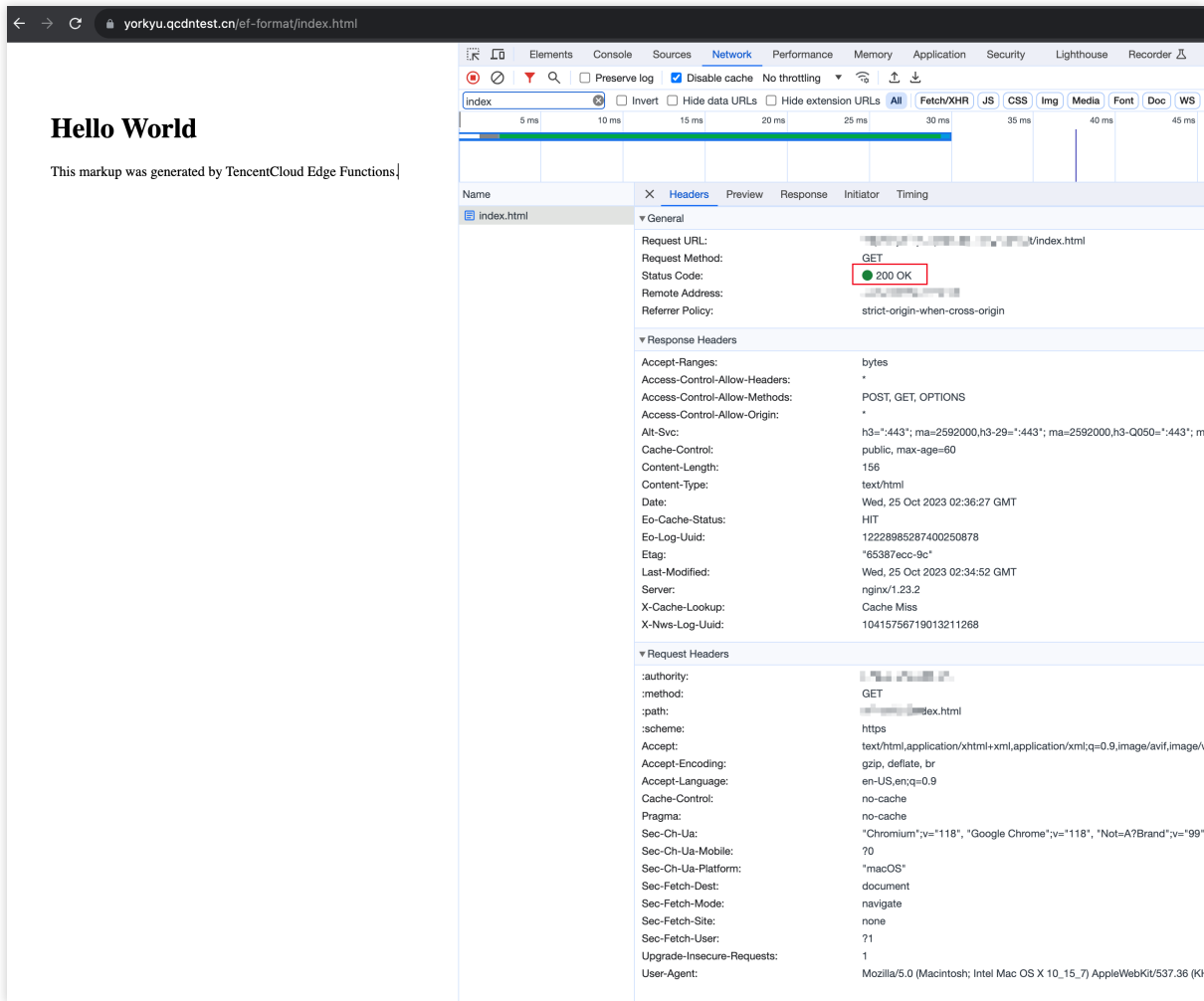
addEventListener('fetch', e => {
  e.respondWith(handleRequest(e.request));
});
```

## 示例预览

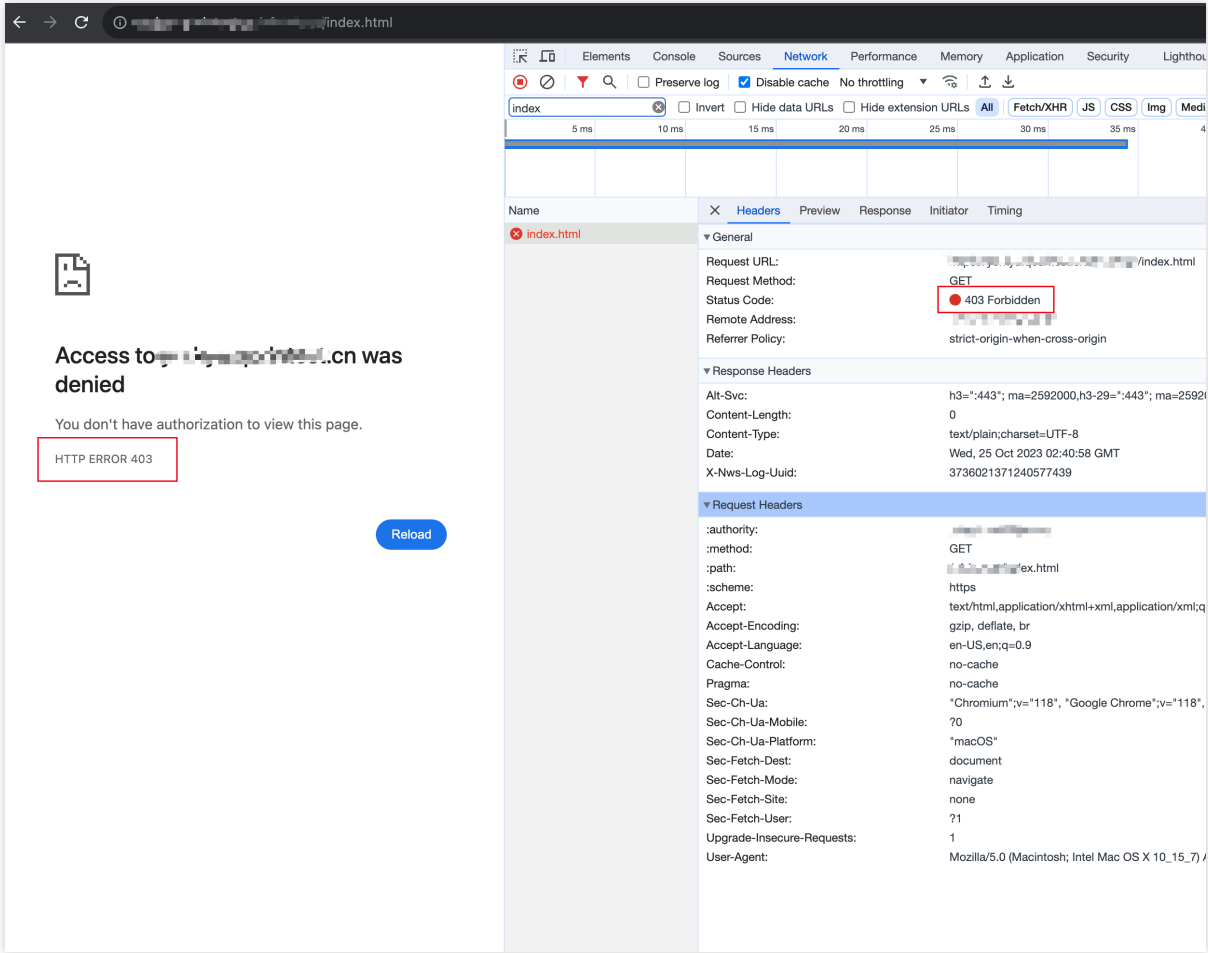
在 PC 端与移动端的浏览器地址栏中输入匹配到边缘函数触发规则的 URL

(如: `https://example.com/app/index.html` ), 即可预览到示例效果。

鉴权通过, 正常访问资源。



鉴权不通过，禁止访问资源。



## 相关参考

[Runtime APIs: Fetch](#)

[Runtime APIs: Response](#)



# HMAC 数字签名

最近更新时间：2023-12-13 10:53:12

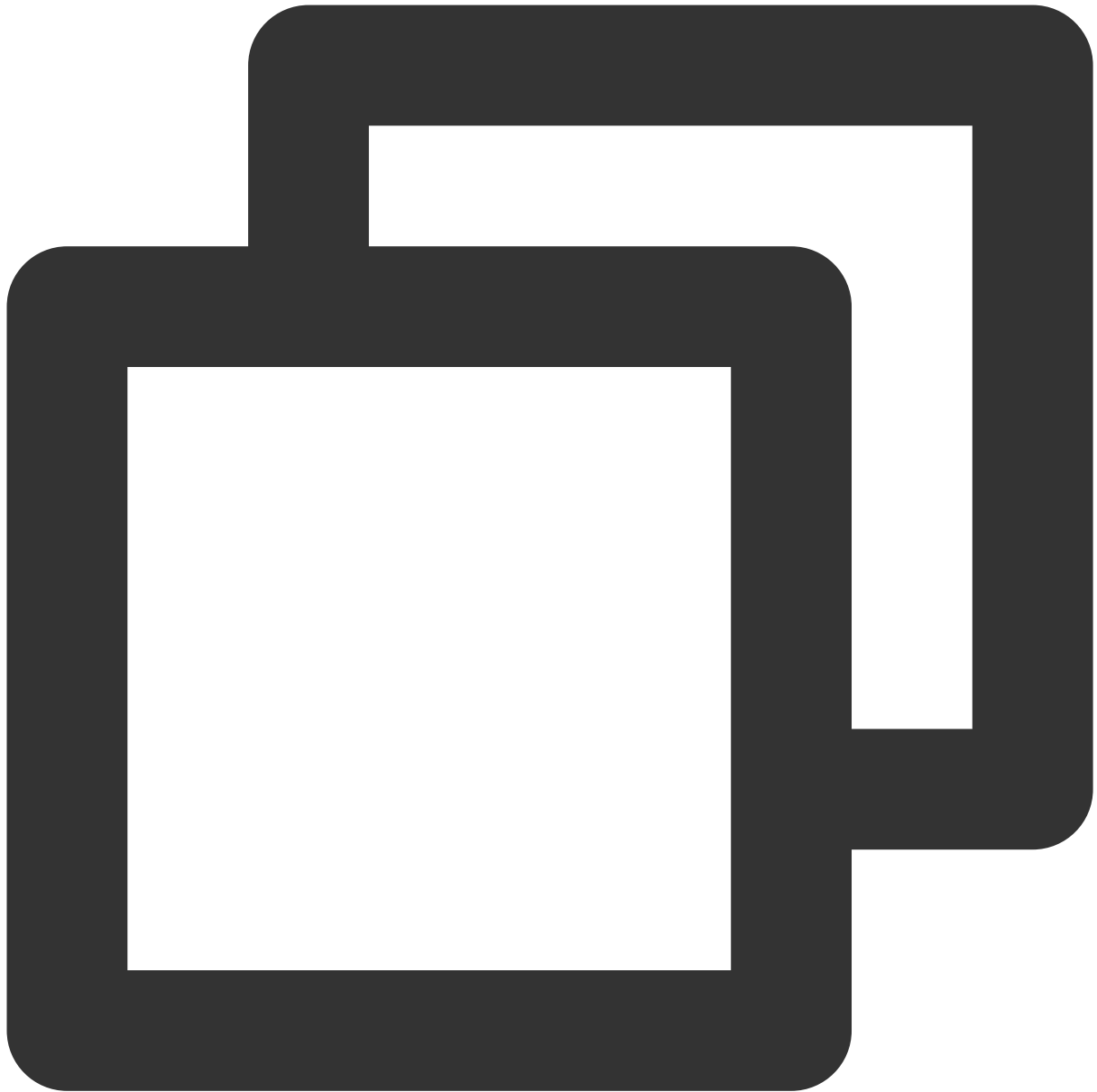
HMAC (Hash-based Message Authentication Code) 是一种基于哈希函数的消息认证码，主要用于验证数据的完整性和身份认证。该示例使用 `Web Crypto API` 实现 HMAC-SHA256 签名，并将签名信息存入请求头，配合源站实现数据完整性校验或身份认证，开发者可根据需要修改代码。

## 注意

该示例要求与源站配合使用，即源站需要具备对应的签名校验算法。

现网使用该示例提供的代码，需要按照注释修改代码。

## 示例代码



```
function uint8ArrayToHex(uint8Array) {
  return Array.prototype.map.call(uint8Array, x => ('0' + x.toString(16)).slice(-2))
}

async function generateHmac({ secretKey, message, hash }) {
  const encoder = new TextEncoder();
  const secretKeyBytes = encoder.encode(secretKey);
  const messageBytes = encoder.encode(message);

  const key = await crypto.subtle.importKey('raw', secretKeyBytes, { name: 'HMAC',
```

```
const signature = await crypto.subtle.sign('HMAC', key, messageBytes);
const signatureArray = new Uint8Array(signature);
return uint8ArrayToHex(signatureArray);
}

async function handleRequest(request) {
  const secretKey = 'YOUR_SECRET_KEY';
  // 使用时请修改为需要被签名的信息，一般为多个数据的组合，例如：时间戳+请求相关信息等
  const message = 'YOUR_MESSAGE';

  // hash 取 SHA-1、SHA-256、SHA-384、SHA-512 之一
  const hmac = await generateHmac({ secretKey, message, hash: 'SHA-256' });
  request.headers.set('Authorization', `HMAC-SHA256 ${hmac}`);

  return fetch(request);
}

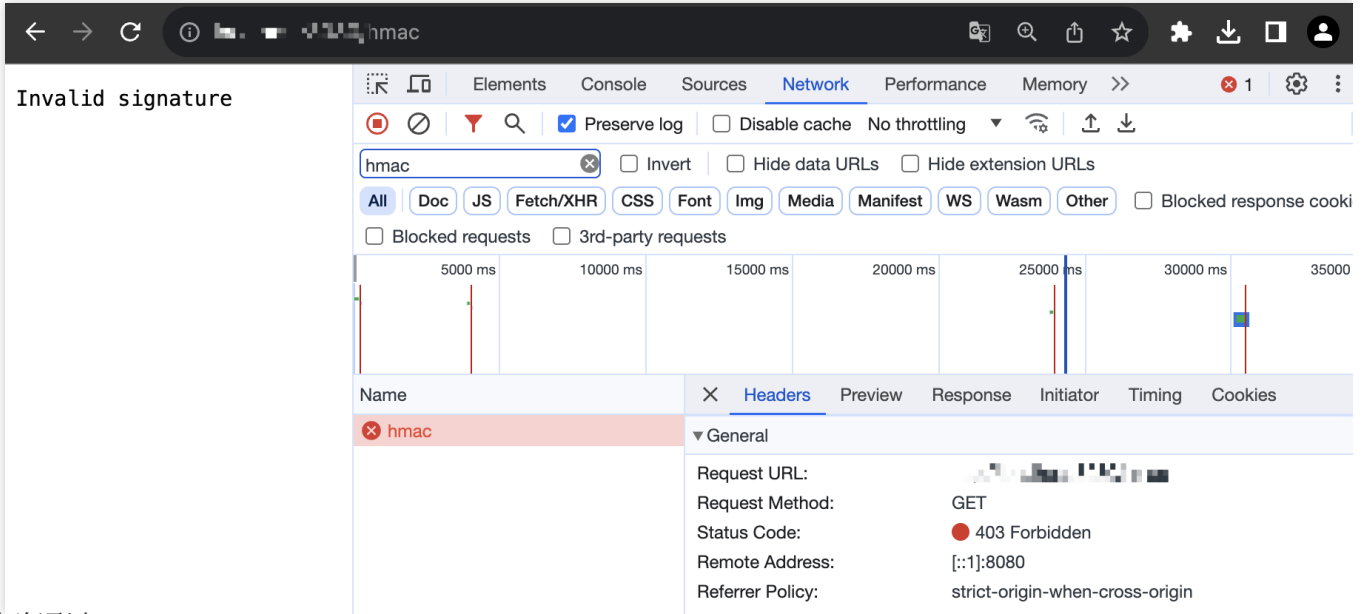
addEventListener('fetch', event => {
  event.passThroughOnException();
  event.respondWith(handleRequest(event.request));
});
```

## 示例预览

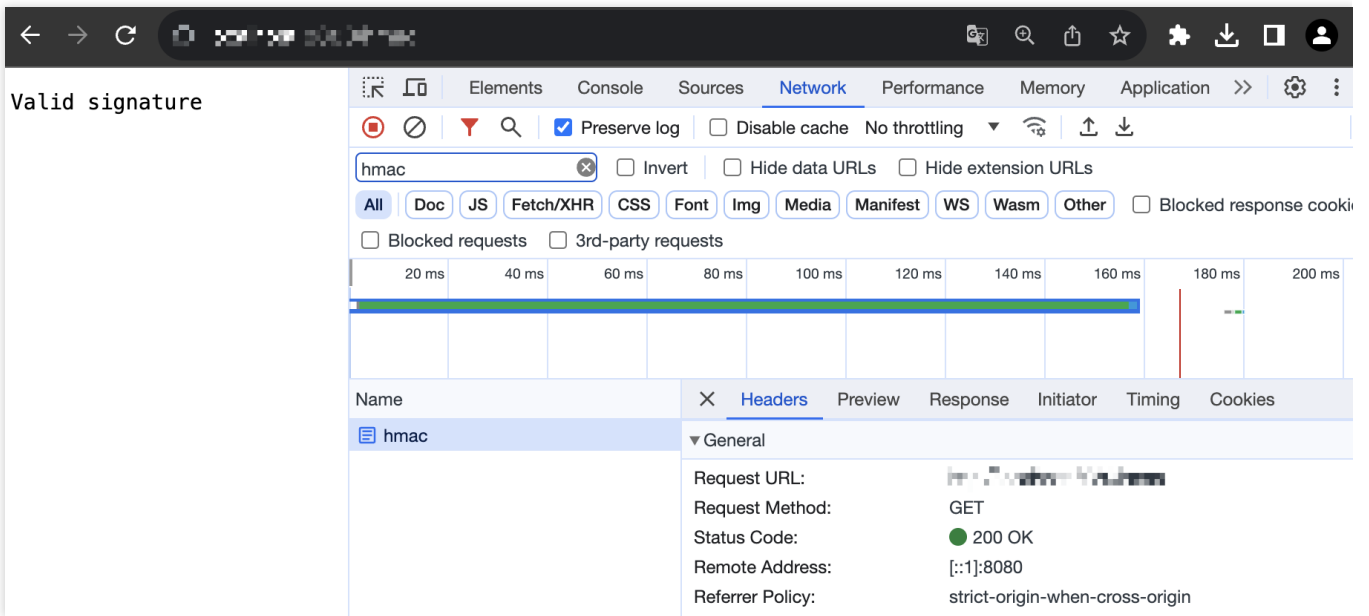
在 PC 端与移动端的浏览器地址栏中输入匹配到边缘函数触发规则的 URL

(如：`https://example.com/hmac` )，即可预览到示例效果。

身份校验未通过。



身份校验通过。



## 相关参考

[Runtime APIs: Fetch](#)

[Runtime APIs: Web Crypto](#)

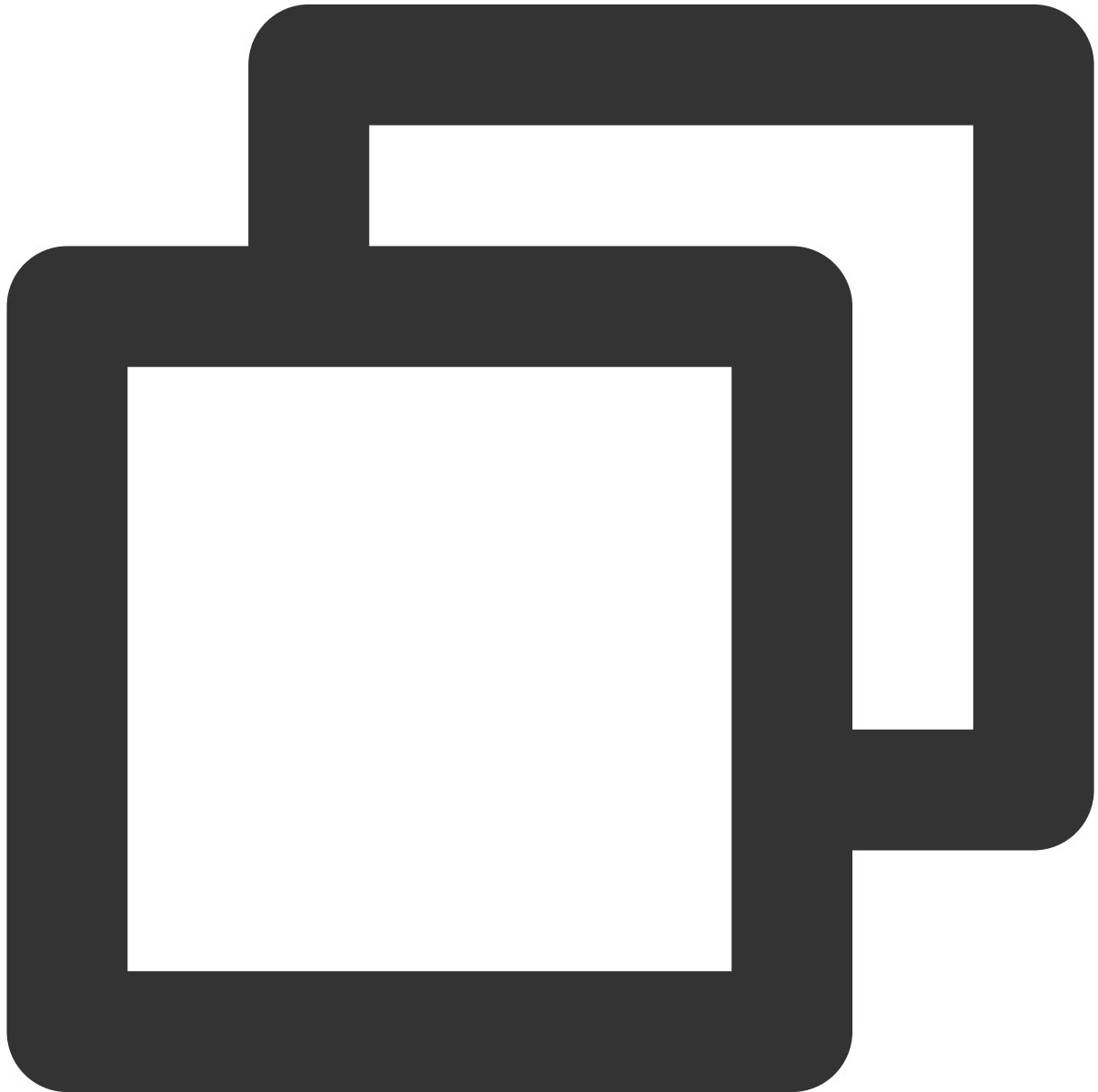
[Runtime APIs: Headers](#)

[Runtime APIs: Response](#)

# 自定义下载文件名

最近更新时间：2023-12-13 10:55:19

该示例通过修改响应头中的 `Content-Disposition` 信息，实现根据请求 URL 中的 `fileName` 参数修改下载文件名。



```
addEventListener('fetch', event => {
  event.passThroughOnException();
  event.respondWith(handleRequest(event.request));
});
```

```
});

async function handleRequest(request) {
  const url = new URL(request.url);
  const fileName = url.searchParams.get('fileName');

  const response = await fetch(request);

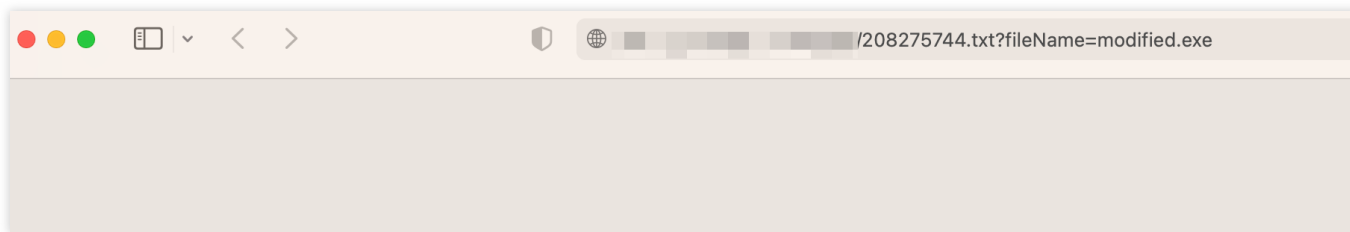
  // 判断响应状态码和 search 参数
  if (response.status !== 200 || !fileName) {
    return response;
  }

  // 修改 Content-Disposition 响应头
  response.headers.append('Content-Disposition', `attachment; filename="${fileName}"`);
  return response;
}
```

## 示例预览

在 PC 端与移动端的浏览器地址栏中输入匹配到边缘函数触发规则的 URL

(如：`https://example.com/origin.exe?fileName=modified.exe`)，即可预览到示例效果。



## 相关参考

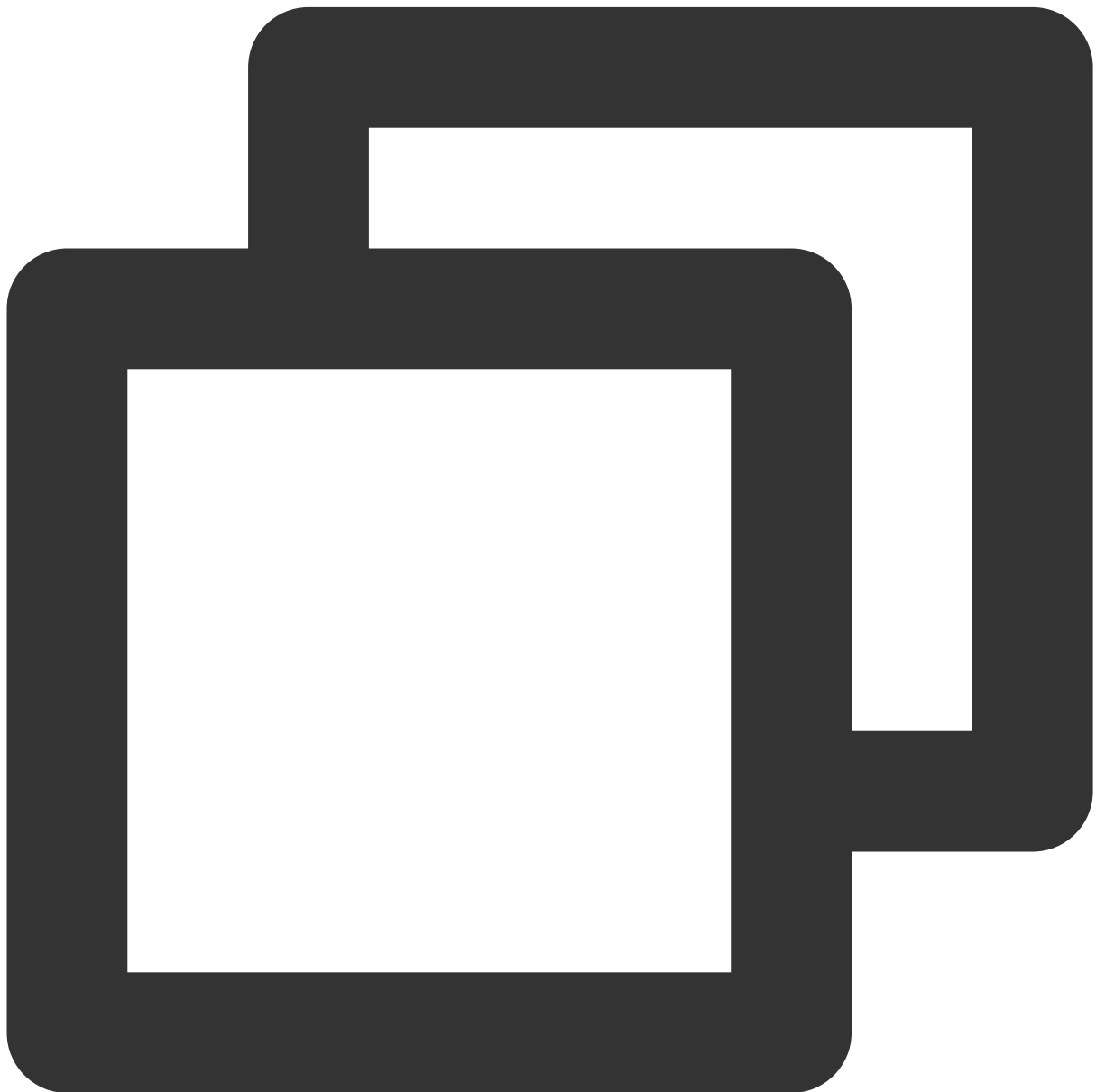
[Runtime APIs: Fetch](#)

[Runtime APIs: Response](#)

# 获取客户端 IP

最近更新时间：2023-12-13 10:56:46

由于前端无法直接获取客户端 IP 地址，在很多业务场景下，通常需要通过服务器端或第三方服务来获取客户端 IP。该示例根据 [规则引擎](#) 中开启的客户端 IP 头部 `EO-Client-IP`，来获取客户端 IP，并组装为 JSON 格式的数据响应客户端，实现了使用边缘函数获取客户端 IP。

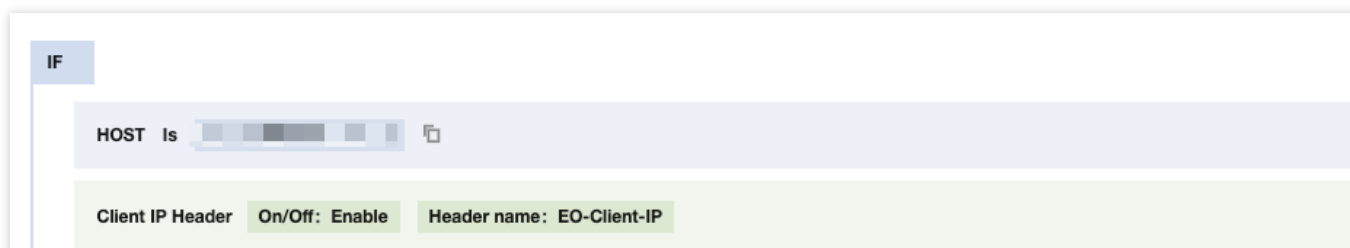


```
addEventListener('fetch', event => {
  event.respondWith(handleRequest(event.request));
});
```

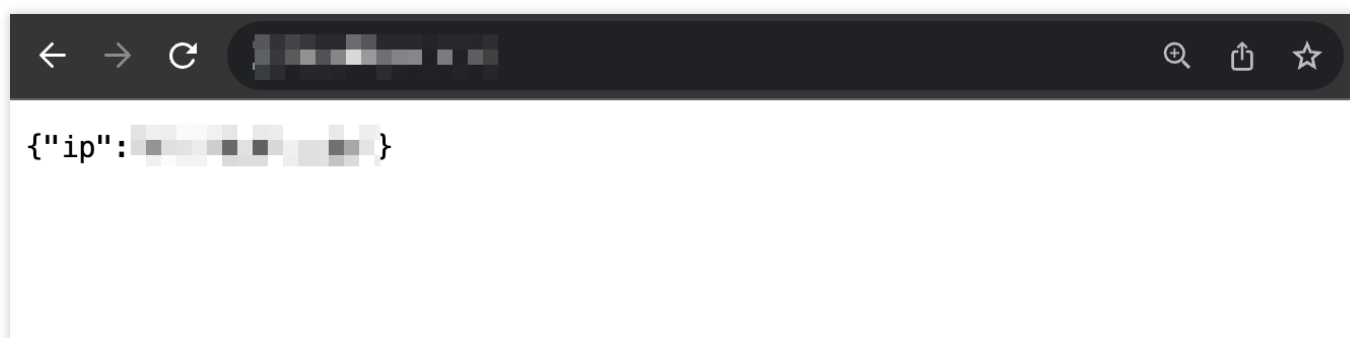
```
});  
  
function handleRequest(request) {  
  // 通过 EO-Client-IP 头部获取客户端 IP  
  const ip = request.headers.get('EO-Client-IP') || '';  
  // 响应 JSON 数据  
  return new Response(JSON.stringify({ ip }), {  
    headers: { 'content-type': 'application/json' },  
  });  
}
```

## 示例预览

首先，需要在 [规则引擎](#) 配置中，对需要触发边缘函数的域名打开客户端 IP 开关，并配置头部名称为 `EO-Client-IP`：



配置生效后，在 PC 端与移动端的浏览器地址栏中输入匹配到边缘函数触发规则的 URL（如：`https://example.com/ip`），即可获得到客户端 IP：



## 相关参考

[Runtime APIs: Fetch](#)

[Runtime APIs: Response](#)



# 最佳实践

## 通过边缘函数实现自适应图片格式转换

最近更新时间：2023-10-11 10:30:02

本文介绍了如何在不修改原始客户端请求 URL 的情况下，通过边缘函数根据客户端请求中携带的 `User-Agent` 头部自动判断需返回的图片文件格式，自动触发图片格式转换。

### 背景介绍

针对以大量图片内容为主的站点，例如：新闻传媒、电商平台、论坛等，图片文件的格式需根据浏览器的类型来进行适配，返回浏览器可兼容的图片格式，同时最大程度上压缩图片的大小，来节省流量。例如：

当用户使用 Chrome、Opera、Firefox、Edge 浏览器访问图片时，响应 `webp` 格式图片。

用户使用 Safari 浏览器访问图片时，响应 `jp2` 格式图片。

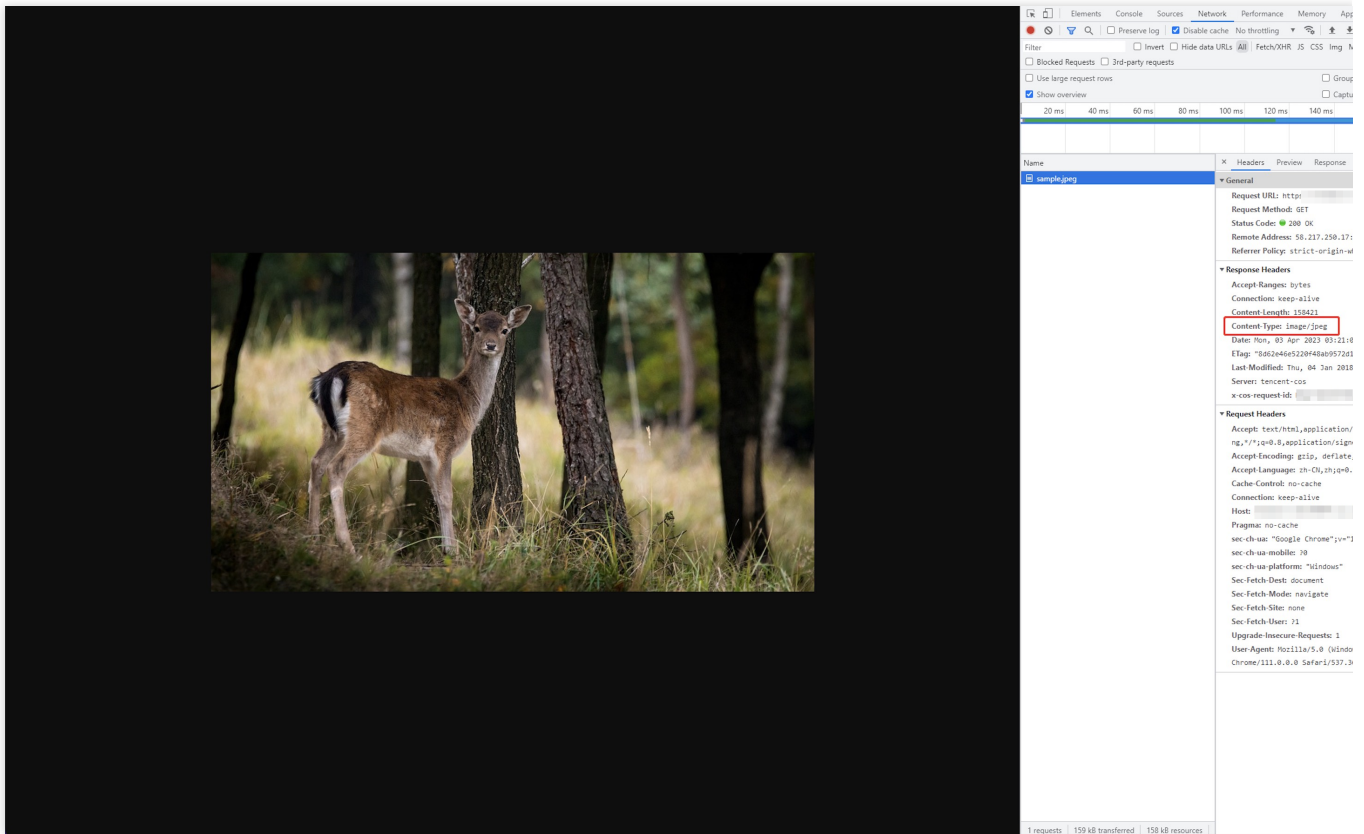
用户使用 IE 浏览器访问图片时，响应 `jxr` 格式图片。

通过其他浏览器访问图片时，统一响应 `webp` 格式图片。

边缘函数提供了灵活的图片处理能力，帮助您在不修改原始客户端请求 URL 的情况下，由 EdgeOne 的边缘函数来自动触发图片格式转换，自适应根据客户端的 `User-Agent` 信息来响应指定的图片格式。从而帮助您在不需要更改业务逻辑的情况下，自适应地为用户提供最佳格式的图片，减少流量消耗。如果您希望在请求 URL 中主动控制触发图片格式转换，也可以参考使用 [图片处理](#) 能力。

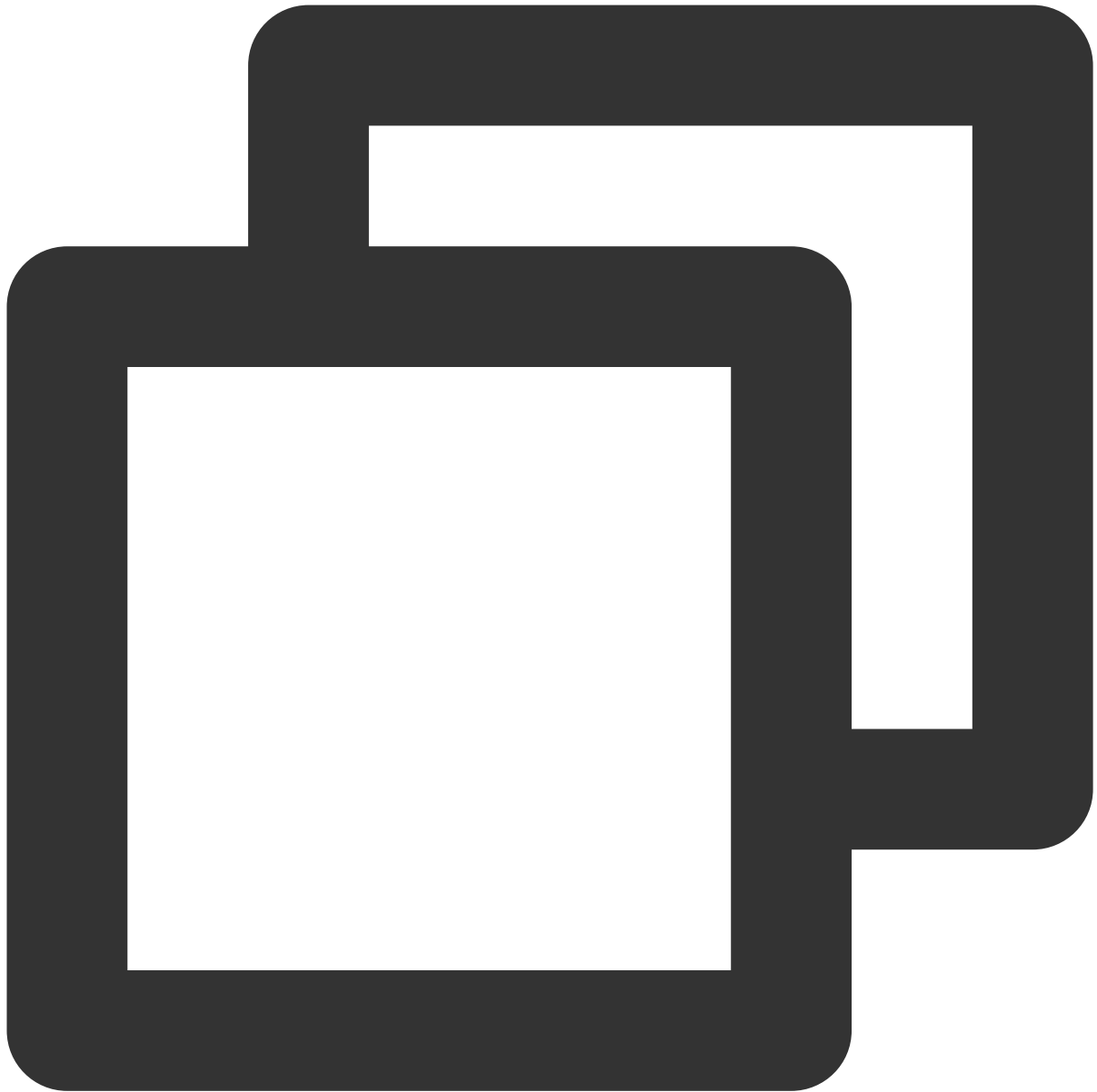
### 预设场景

当前已接入站点为：`example.com`，图片内容均存储于 `http://image.example.com/image/` 路径下，需将该路径下所有图片自适应根据客户端的浏览器类型响应最佳的图片格式。其中测试用原始图片请求 URL 为：`https://image.example.com/image/test.jpg`，访问后查看图片格式如下：



## 操作步骤

1. 登录 [边缘安全加速平台 EO 控制台](#)，通过站点列表，选择需配置的站点，进入站点管理二级菜单。
2. 在左侧导航栏中，单击**边缘函数 > 函数管理**。
3. 在函数管理页面，单击**新建函数**。
4. 在新建函数页面，输入函数名称、函数描述和函数代码。以下为该场景示例代码：



```
// 浏览器使用图片格式
const broswerFormat = {
  Chrome: 'webp',
  Opera: 'webp',
  Firefox: 'webp',
  Safari: 'jp2',
  Edge: 'webp',
  IE: 'jxr'
};

addEventListener('fetch', event => {
```

```
// 当函数代码抛出未处理的异常时，边缘函数会将此请求转发回源站
event.passThroughOnException();
event.respondWith(handleEvent(event));
});

async function handleEvent(event) {
  const { request } = event;
  const userAgent = request.headers.get('user-agent');
  const bs = getBrowser(userAgent);
  const format = browserFormat[bs];

  // 无需转换图片格式
  if (!format) {
    return fetch(request);
  }

  // 图片格式转换
  const response = await fetch(request, {
    eo: {
      image: {
        format
      }
    }
  });

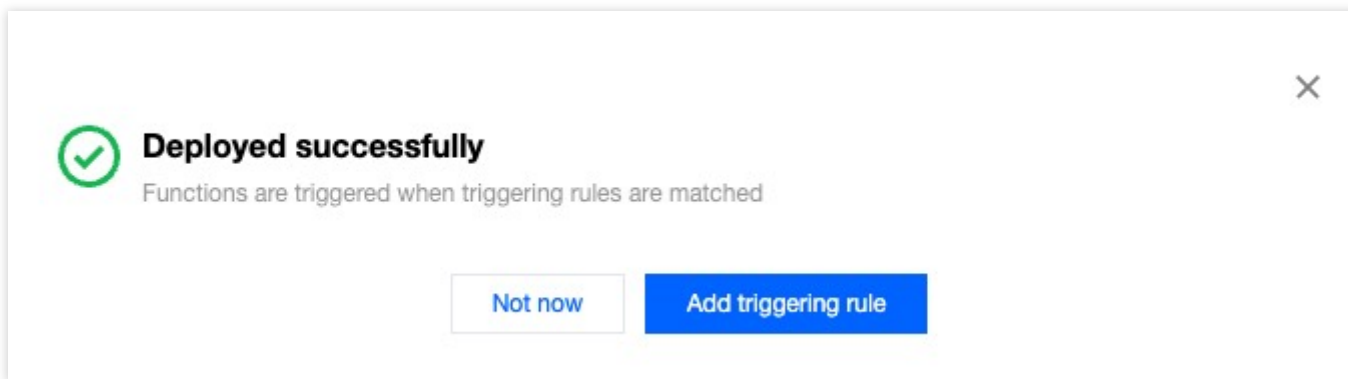
  // 设置响应头
  response.headers.set('x-ef-format', format);

  return response;
}

function getBrowser(userAgent) {
  if (/Edg/i.test(userAgent)) {
    return 'Edge'
  }
  if (/Trident/i.test(userAgent)) {
    return 'IE'
  }
  if (/Firefox/i.test(userAgent)) {
    return 'Firefox';
  }
  if (/Chrome/i.test(userAgent)) {
    return 'Chrome';
  }
  if (/Opera|OPR/i.test(userAgent)) {
    return 'Opera';
  }
}
```

```
if (/Safari/i.test(userAgent)) {  
  return 'Safari'  
}  
}
```

5. 编辑完成函数后，单击**创建函数并部署**，函数部署后，可直接单击**新增触发规则**，前往配置该函数的触发规则。

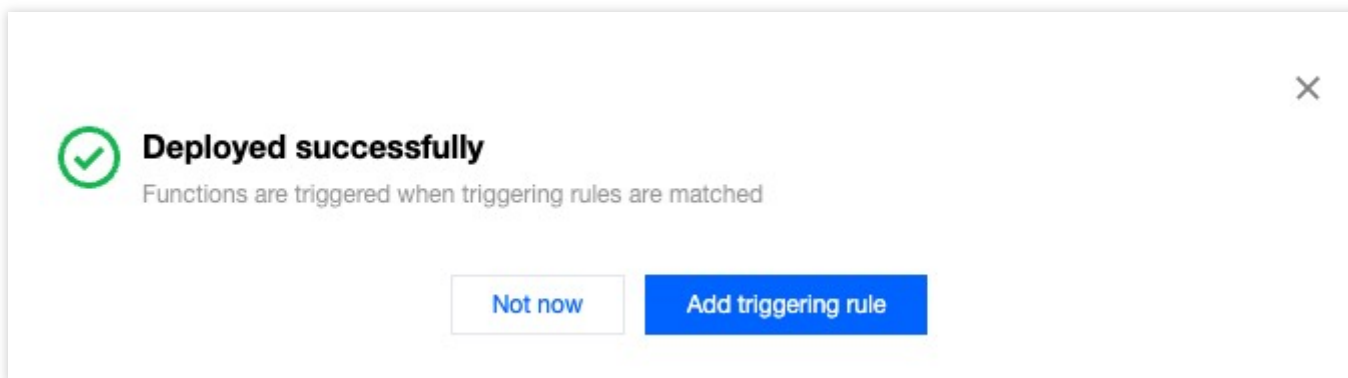


6. 在函数触发规则中，配置该函数的触发条件，根据当前的场景需求，您可以配置两条触发条件，以 And 逻辑触发。

该请求 HOST 等于 `Image.example.com`。

该请求 URL Path 等于 `/image/*`。

当请求 URL 同时符合以上条件时，将触发以上的边缘函数，对图片进行自动处理。



7. 单击**保存**触发规则即可生效。

8. 验证边缘函数的生效情况，您可以通过以下两种方式进行验证：

curl 请求测试

浏览器访问测试

您可以通过 curl 请求中携带指定的 User-Agent 进行测试。

测试 Chrome 等浏览器

测试 Safari 浏览器

## 测试 IE 浏览器

在 Mac/linux 环境下，以测试 chrome 浏览器为例，可以在终端内运行命令：`curl --user-agent "chrome"`

`https://image.example.com/image/test.jpg -i`

查看响应的 `Content-Type` 信息，是否为 `image/webp`。

```
~ % curl --user-agent "chrome" https://image.example.com/image/test.jpg -i
HTTP/1.1 200 OK
x-ef-format: webp
EO-LOG-UUID: 3816446099087859674
Cache-Control: max-age=2592000
Last-Modified: Fri, 14 Apr 2023 08:14:28 GMT
Accept-Ranges: bytes
Connection: keep-alive
Date: Fri, 14 Apr 2023 08:14:28 GMT
X-RtFlag: 1
X-ReqId: MTY4MTQ2MDA2Nl83NzE5OTgyY185QkZERTQ5OEFDQjE0N0Q4ODBBM0ZDMTQ5QjU4NzI4MA==
Size: 123220
X-DataSrc: 1
X-Info: real data
EO-Cache-Status: HIT
X-Delay: 2316668 us
Content-Type: image/webp
Server: tencent-cl
Content-Length: 123220
```

在 Mac/linux 环境下，在终端内运行命令：`curl --user-agent "safari"`

`https://image.example.com/image/test.jpg -i`

查看响应的 `Content-Type` 信息，是否为 `image/jp2`。

```
~ % curl --user-agent "safari"
HTTP/1.1 200 OK
x-ef-format: jp2
EO-LOG-UUID: 299090522723185511
Cache-Control: max-age=2592000
Last-Modified: Fri, 14 Apr 2023 08:42:27 GMT
Accept-Ranges: bytes
Connection: keep-alive
Date: Fri, 14 Apr 2023 08:42:27 GMT
X-RtFlag: 1
X-ReqId: MTY4MTQ2MTc0NV83NzE5OTgyYl8zRjRCODlE0DM2NDg0RTEyQTJGRTYyNTZDODI1MkEyMg==
Size: 121014
X-DataSrc: 1
X-Info: real data
EO-Cache-Status: MISS
X-Delay: 2747522 us
Content-Type: image/jp2
Server: tencent-ci
Content-Length: 121014
```

在 Mac/linux 环境下，在终端内运行命令：`curl --user-agent "Trident"`

`https://image.example.com/image/test.jpg -i`

查看响应的 `Content-Type` 信息，是否为 `image/vnd.ms-photo`。

```
~ % curl --user-agent "Trident"
HTTP/1.1 200 OK
x-ef-format: jxr
EO-LOG-UUID: 16823953457232177833
Cache-Control: max-age=2592000
Last-Modified: Fri, 14 Apr 2023 08:49:50 GMT
Accept-Ranges: bytes
Connection: keep-alive
Date: Fri, 14 Apr 2023 08:49:50 GMT
X-RtFlag: 1
X-ReqId: MTY4MTQ2MjE5MF83NzE5OTgyYl80MDU1MjdGMjJDMTQ0RjEwODlBNjRFREYBMUzMTkwNQ==
Size: 138140
X-DataSrc: 1
X-Info: real data
EO-Cache-Status: MISS
X-Delay: 97009 us
Content-Type: image/vnd.ms-photo
Server: tencent-ci
Content-Length: 138140
```

在不同的浏览器地址栏中打开控制台后，输入测试图片的地址

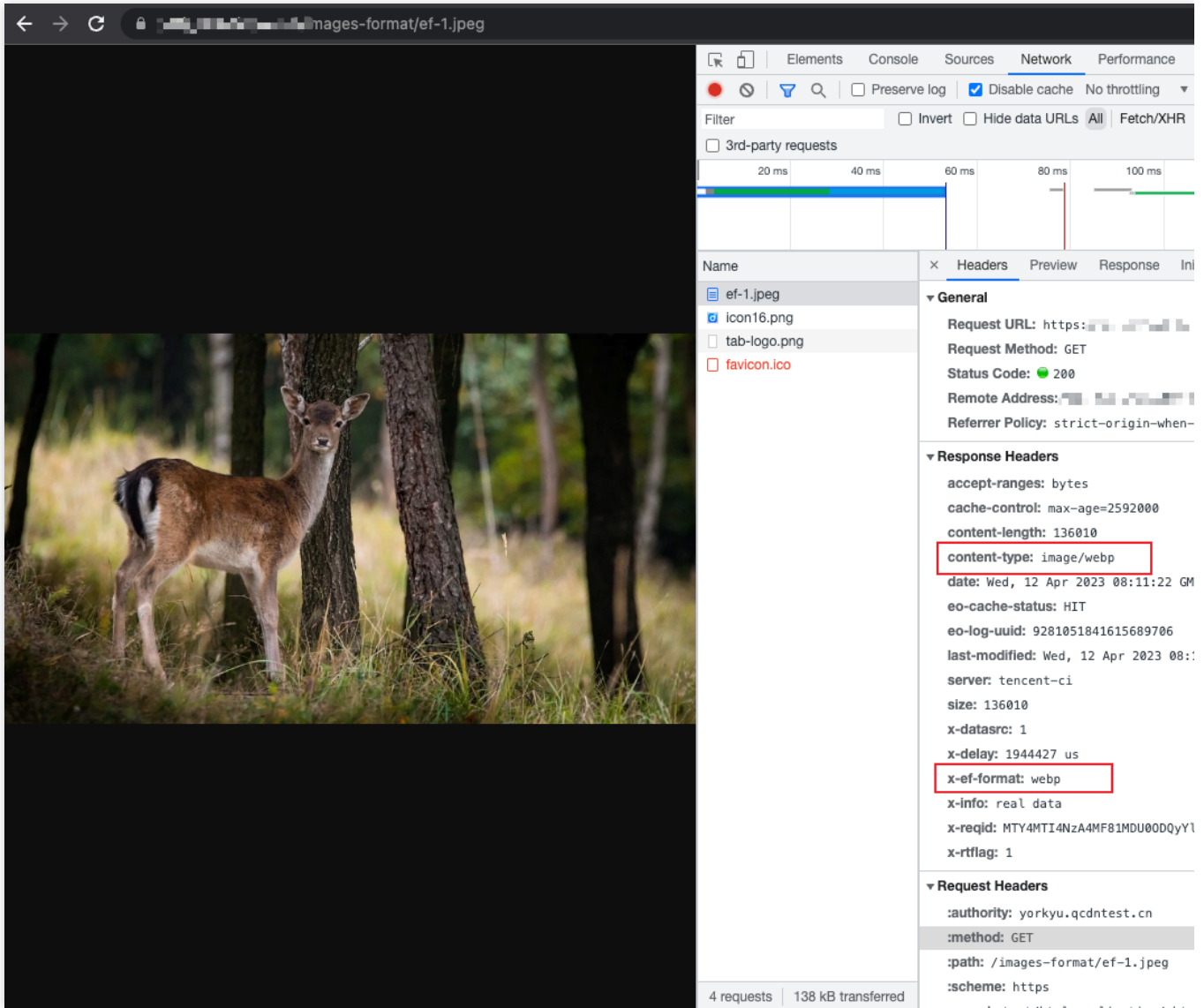
`https://image.example.com/image/test.jpg`，可通过响应图片的格式查看当前边缘函数是否已生效。

测试 Chrome 等浏览器

测试 Safari 浏览器

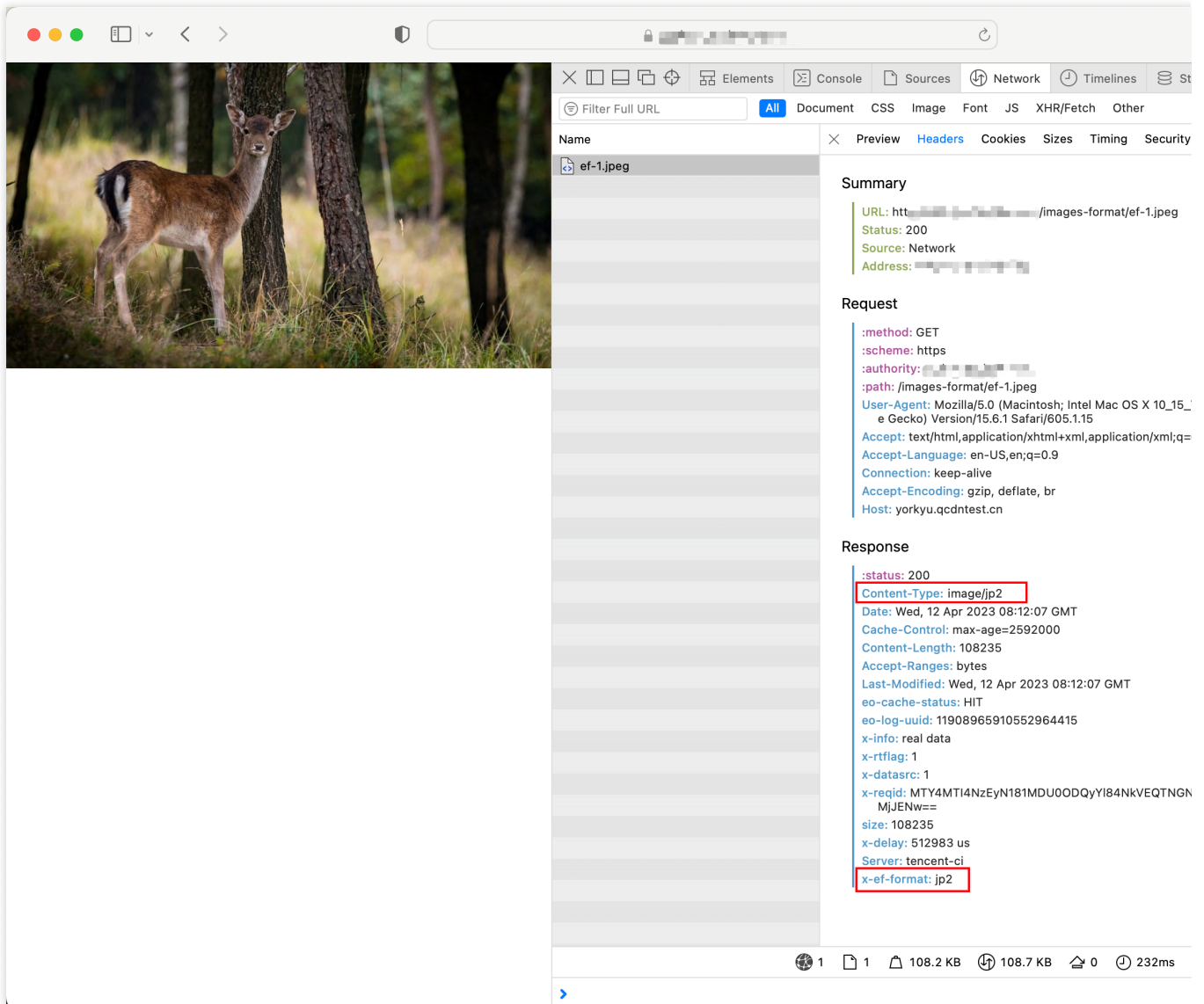
测试 IE 浏览器

在 chrome 浏览器中访问测试图片地址：`https://image.example.com/image/test.jpg`，该图片响应为 webp 格式。

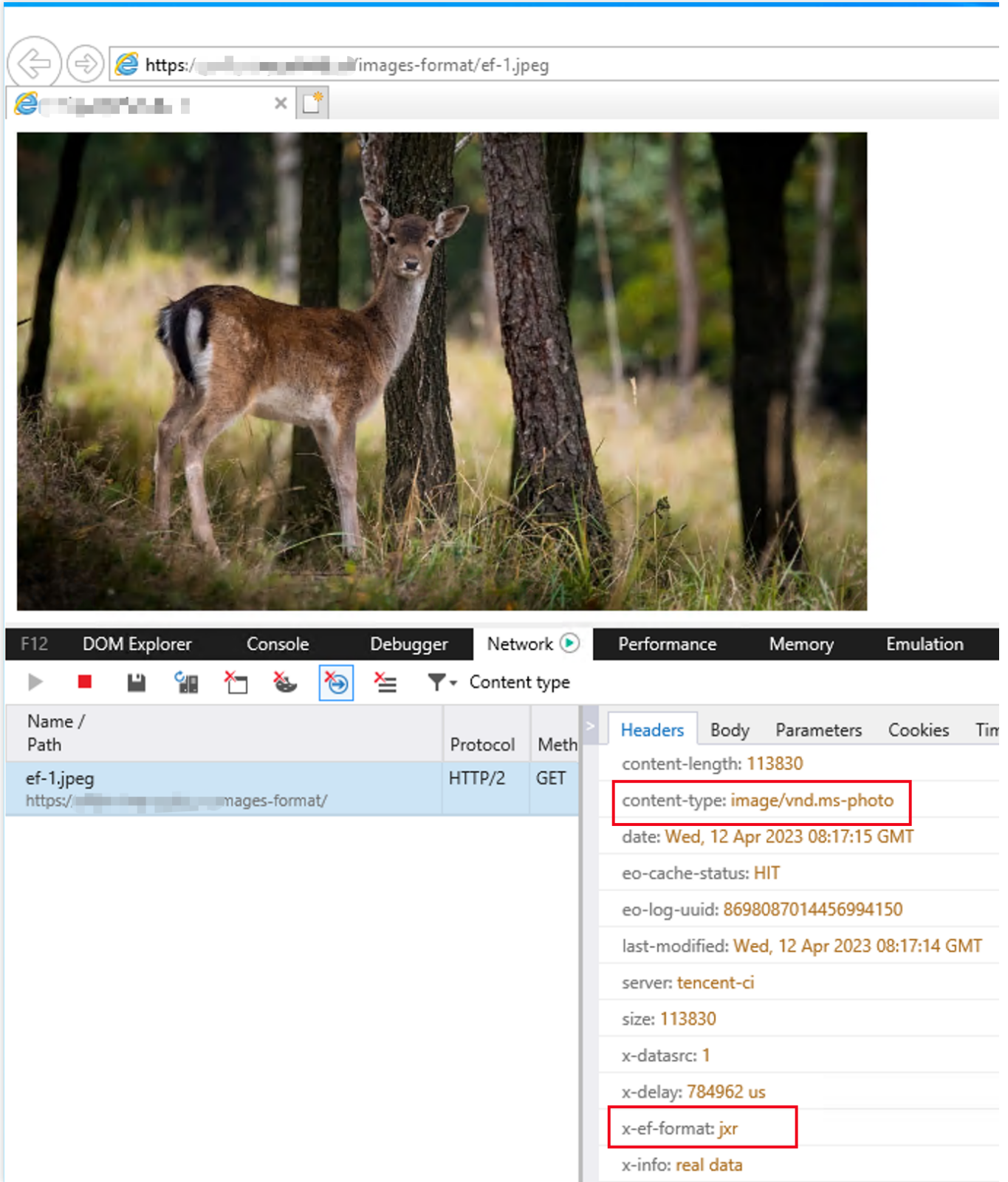


在 safari 浏览器中访问测试图片地址：`https://image.example.com/image/test.jpg`，该图片响应为 jp2 格式。





在 safari 浏览器中访问测试图片地址：`https://image.example.com/image/test.jpg`，该图片响应为 jxr 格式。



The screenshot shows a browser window with a URL bar containing 'https://...images-format/ef-1.jpeg'. Below the browser, the developer tools network tab is open, displaying a list of network requests. The selected request is 'ef-1.jpeg' with a protocol of 'HTTP/2' and a method of 'GET'. The 'Headers' pane on the right shows the following details:

Name	Value
content-length	113830
content-type	image/vnd.ms-photo
date	Wed, 12 Apr 2023 08:17:15 GMT
eo-cache-status	HIT
eo-log-uuid	8698087014456994150
last-modified	Wed, 12 Apr 2023 08:17:14 GMT
server	tencent-ci
size	113830
x-datasrc	1
x-delay	784962 us
x-ef-format	jxr
x-info	real data

了解更多

---

示例函数：图片自适应格式转换  
通过站点加速使用图片缩放