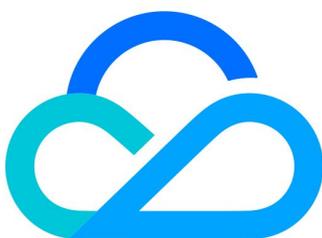


边缘安全加速平台 EO

边缘函数



腾讯云

【 版权声明 】

©2013–2025 腾讯云版权所有

本文档（含所有文字、数据、图片等内容）完整的著作权归腾讯云计算（北京）有限责任公司单独所有，未经腾讯云事先明确书面许可，任何主体不得以任何形式复制、修改、使用、抄袭、传播本文档全部或部分内容。前述行为构成对腾讯云著作权的侵犯，腾讯云将依法采取措施追究法律责任。

【 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。未经腾讯云及有关权利人书面许可，任何主体不得以任何方式对前述商标进行使用、复制、修改、传播、抄录等行为，否则将构成对腾讯云及有关权利人商标权的侵犯，腾讯云将依法采取措施追究法律责任。

【 服务声明 】

本文档意在向您介绍腾讯云全部或部分产品、服务的当时的相关概况，部分产品、服务的内容可能不时有所调整。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默示的承诺或保证。

【 联系我们 】

我们致力于为您提供个性化的售前购买咨询服务，及相应的技术售后服务，任何问题请联系 4009100100或 95716。

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

Web 调试

环境变量

AI 代码助手

Runtime APIs

addEventListener

Cache

Cookies

Encoding

Fetch

FetchEvent

Headers

Request

Response

Streams

ReadableStream

ReadableStreamBYOBReader

ReadableStreamDefaultReader

TransformStream

WritableStream

WritableStreamDefaultWriter

Web Crypto

Web Standards

Images

ImageProperties

示例函数

示例概览

301 重定向

获取客户端 URL 信息

基于客户端地理特征定制化

获取客户端地理特征信息

- 批量重定向
- 基于正则的 URL 改写
- 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 限制规则
- 远程鉴权
- HMAC 数字签名
- 自定义下载文件名
- 获取客户端 IP
- 复杂回源 URL 重写

实践教程

概览

根据用户 IP/地理位置回源

EdgeOne 实现基于客户端 IP 地址的会话保持

EdgeOne 实现基于客户端地理特征的指定回源

APK 动态打包

EdgeOne 实现 Android APK 的动态打包

功能简介

步骤一：预处理 Android APK 母包

步骤二：EdgeOne 边缘函数写入渠道信息到 APK 包

步骤三：测试并验证生效结果

通过边缘函数实现自适应图片格式转换

边缘函数

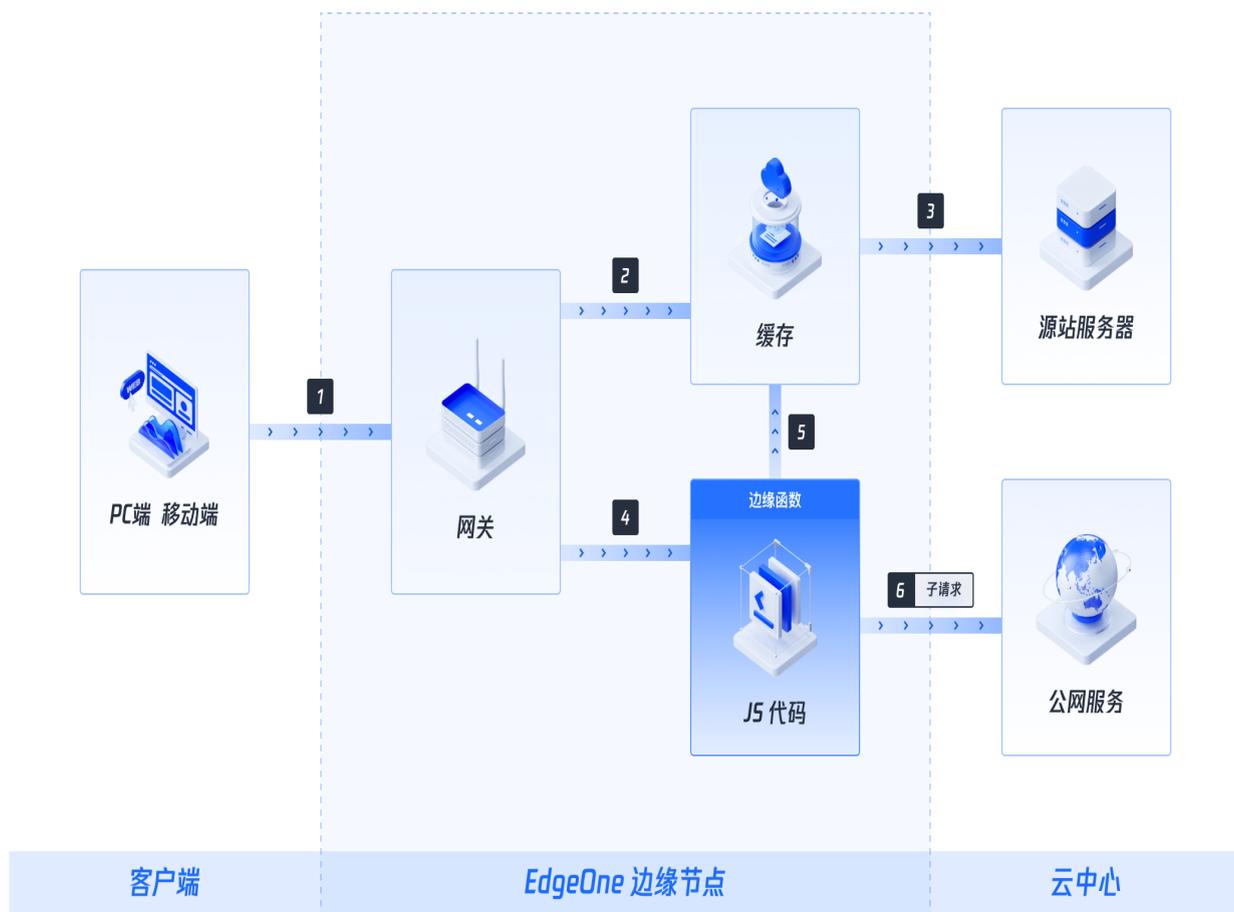
概述

最近更新时间：2024-06-04 16:07:41

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 3200+ 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



使用限制

| 内容 | 限制 | 说明 |
|---------|-------|----------------------|
| 单站点函数数量 | 100 个 | 单个站点最多支持创建边缘函数个数为100 |

| | | |
|--------------|------------|--|
| 单站点触发规则数量 | 200 条 | 单个站点最多支持创建函数的触发规则数量为 200 |
| 函数名称长度 | 30 字符 | 2~30个字符，最多支持30个字符 |
| 代码包大小 | 5 MB | 单个函数代码包大小最多支持 5 MB |
| 请求 body 大小 | 1 MB | 客户端请求携带 body 最多支持 1 MB |
| CPU 时间 | 200 ms | 函数单次执行分配的 CPU 时间片，不包含 I/O 等待时间 |
| 开发语言 | JavaScript | 目前仅支持 JavaScript |
| console 调用次数 | 20 次 | 每个函数内最多允许调用 20 次 console 方法。超过 20 次后，将不再执行打印操作 |
| 循环执行次数 | 100000 次 | 函数中 <code>for</code> 、 <code>for in</code> 、 <code>for of</code> 、 <code>while</code> 、 <code>do while</code> 循环限制执行不超过 100000 次 |

快速指引

最近更新时间：2024-04-19 16:10:51

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-edgesfunctions。
 - 描述：非必填，最多支持60个字符。如：自定义 HTML 页面和响应头。
 - 代码预览：模板对应的代码。
5. 单击**创建并部署**，当弹窗如下对话框，即表示部署成功。



部署成功

函数触发需要匹配触发规则

暂不需要

新增触发规则

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。



e-...c/新增触发规则

请求URL匹配到触发条件后则执行函数。

站点: [Redacted]

描述: 匹配host测试
您还可以输入52个字符

触发条件

if

| 匹配类型 | 运算符 | 值 |
|------|-----|----|
| HOST | 等于 | ub |

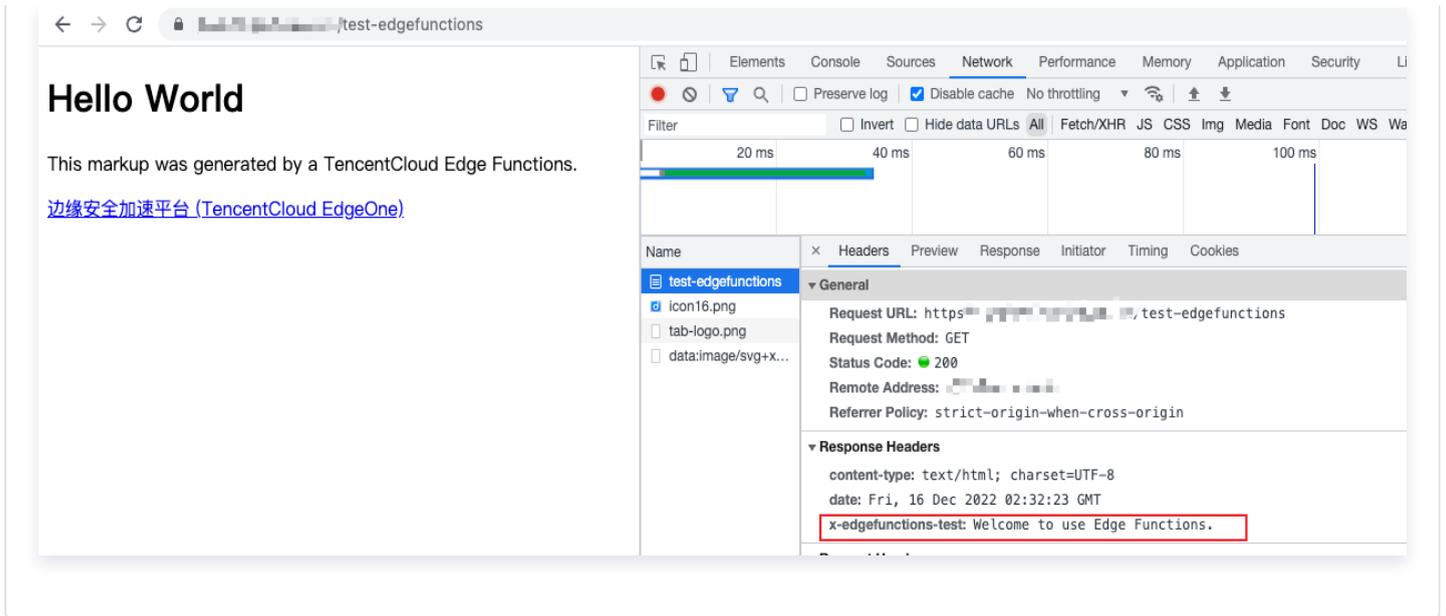
+ And + Or

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

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：



curl 验证

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-edgelfunctions`，可查看响应如下：

```
→ ~ curl -i https://[redacted] ions
HTTP/2 200
x-edgelfunctions-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：2024-04-19 16:10:51

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。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。
 - **代码预览**：模板对应的代码。

函数名称 -zone-2 390

只能包含小写字母、数字、连字符，以字母开头，以数字或字母结尾，2~30个字符，不允许连续连字符；创建后无法修改。

描述 非必填

您还可以输入60个字符

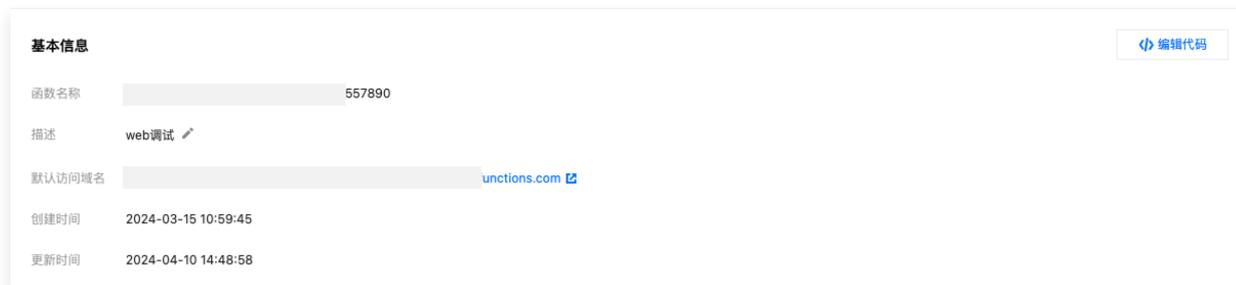
代码预览

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

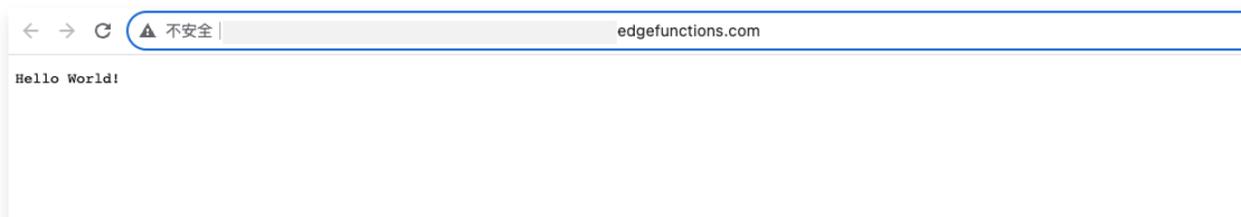
5. 单击**创建并部署**，当弹窗如下对话框，即表示部署成功。



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。



如部署默认函数代码后预期生效则如下所示：



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. 单击**确定**，即可创建触发规则。

触发规则

[新增触发规则](#)

| 规则 ID | 描述 | 触发条件 | 创建时间 | 更新时间 | 操作 |
|------------|----|---------------------------|---------------------|---------------------|---------------------------------------|
| rule- x | - | if (\$host equal "16... | 2024-03-15 15:34:02 | 2024-03-15 15:34:02 | 编辑 删除 |
| rule- > | - | if (\$host equal "1 cd... | 2024-03-15 15:31:18 | 2024-03-15 15:31:54 | 编辑 删除 |

共 2 条 10 条 / 页 1 / 1 页

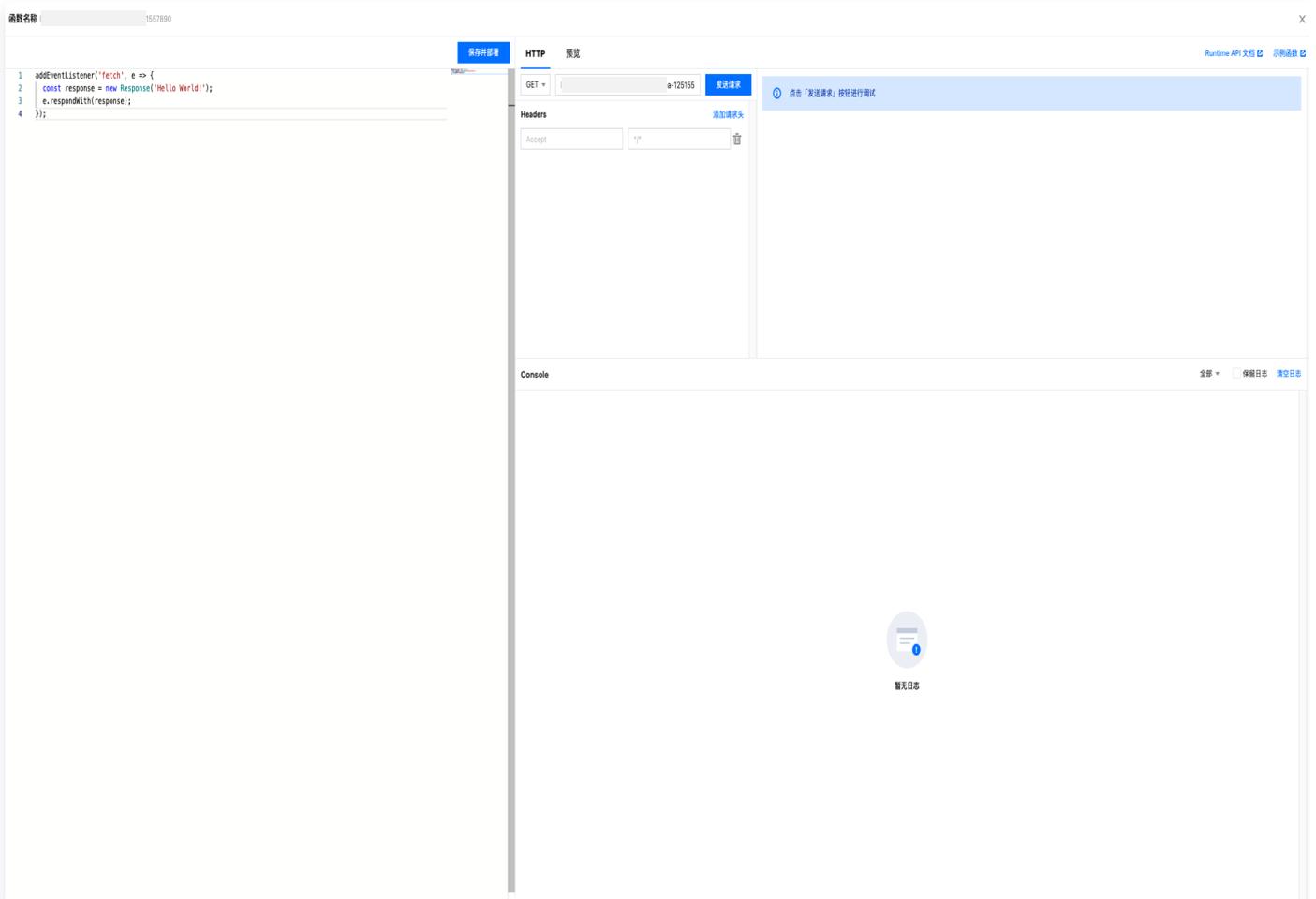
编辑边缘函数

1. 在函数管理页面，选择需要修改的函数，单击该**函数名称**，在函数详情基本信息页单击**编辑代码**。

基本信息 [编辑代码](#)

| | |
|--------|------------------------------------|
| 函数名称 | 557890 |
| 描述 | web调试 编辑 |
| 默认访问域名 | functions.com 外部链接 |
| 创建时间 | 2024-03-15 10:59:45 |
| 更新时间 | 2024-04-10 14:48:58 |

2. 修改函数代码后单击**保存并部署**或**Ctrl + S**。

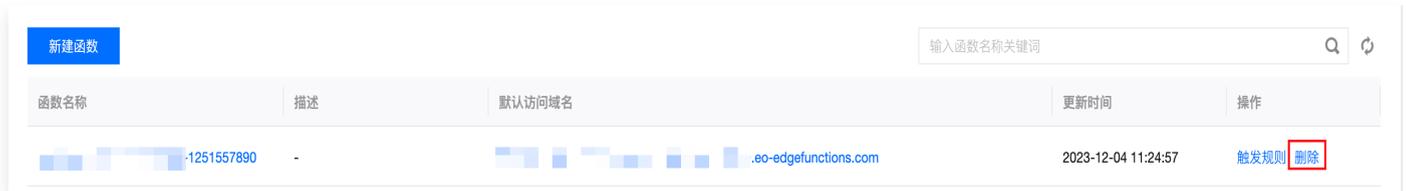


3. 将会提示当前函数已存在触发规则，单击**确认部署**，即可完成更新部署操作。



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。



The screenshot shows the 'Function Management' page in the Tencent Cloud console. At the top left is a blue button labeled '新建函数' (New Function). To the right is a search bar with the placeholder text '输入函数名称关键词' (Enter function name keyword) and search icons. Below is a table with the following columns: '函数名称' (Function Name), '描述' (Description), '默认访问域名' (Default Access Domain), '更新时间' (Update Time), and '操作' (Action). A single function is listed with a name ending in '-1251557890', a description of '-', and a domain ending in '.eo-edgefunctions.com'. The update time is '2023-12-04 11:24:57'. In the '操作' column, there are two links: '触发规则' (Trigger Rule) and '删除' (Delete), with the '删除' link highlighted by a red rectangular box.

| 函数名称 | 描述 | 默认访问域名 | 更新时间 | 操作 |
|------|----|-------------------------|---------------------|---|
| ... | - | ...eo-edgefunctions.com | 2023-12-04 11:24:57 | 触发规则 删除 |

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：2024-08-23 15:07:21

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。

新增触发规则



请求URL匹配到触发条件后则执行函数。

站点

描述

您还可以输入60个字符

执行函数

[前往创建](#)

触发条件

If

匹配类型



[+ And](#) [+ Or](#)

确定

取消

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。

编辑触发规则



请求URL匹配到触发条件后则执行函数。

站点

描述

您还可以输入60个字符

执行函数



前往创建

触发条件

If

匹配类型 ⓘ

运算符

值

HOST

等于



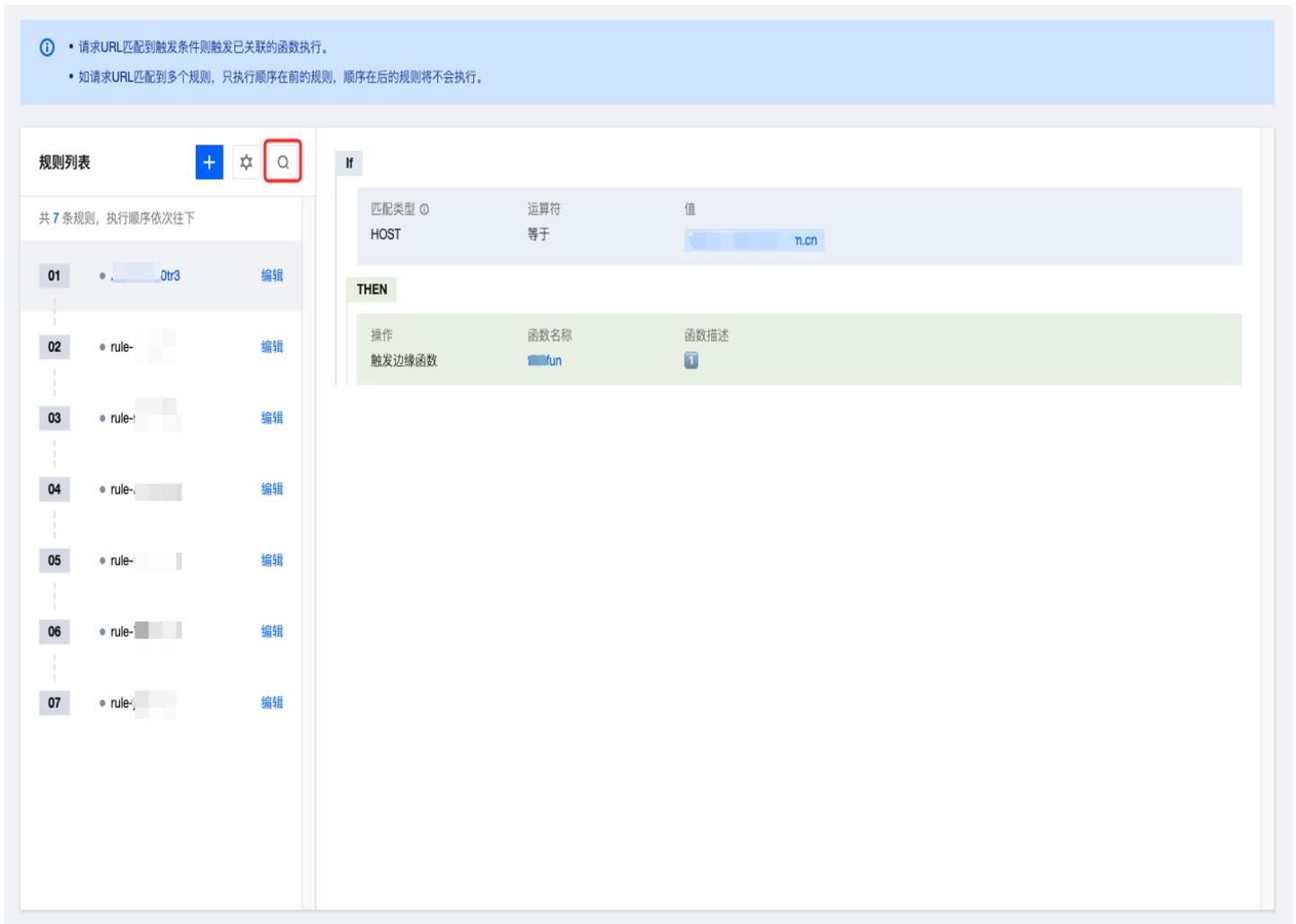
+ And + Or

确定

取消

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 ⚙️ 图标。
2. 选择需要删除的规则，单击 🗑️ 图标。



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）。

① 请求URL匹配到触发条件则触发已关联的函数执行。
② 如请求URL匹配到多个规则，只执行顺序在前的规则，顺序在后的规则将不会执行。

The screenshot displays the configuration interface for trigger rules. On the left, a list of four rules (01, 02, 03, 04) is shown with their execution order. Rule 01 is highlighted with a red box around its 'down' arrow icon, indicating it is being moved down. Rule 02 is also highlighted with a red box around its 'down' arrow icon. On the right, the configuration details for a selected rule are shown. The 'If' section contains two conditions: 'HOST' equals 'com.cn' and 'URL Path' equals '/123'. The 'Then' section shows the action '触发边缘函数' (Trigger Edge Function) with the function name '22'.

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。

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">边缘安全加速平台
    (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">边缘安全加速平台
  (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));
});
```

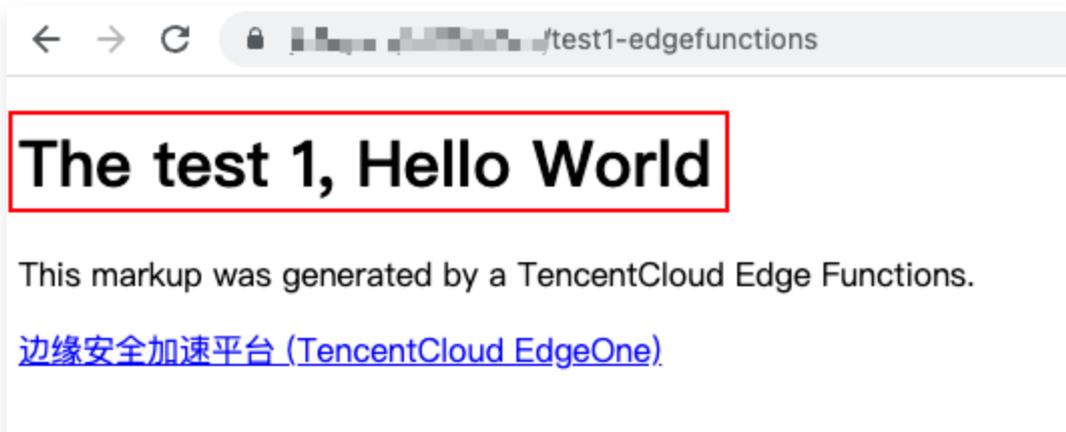
2. 在触发配置页面, 可查看触发规则, 如下图所示:



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：

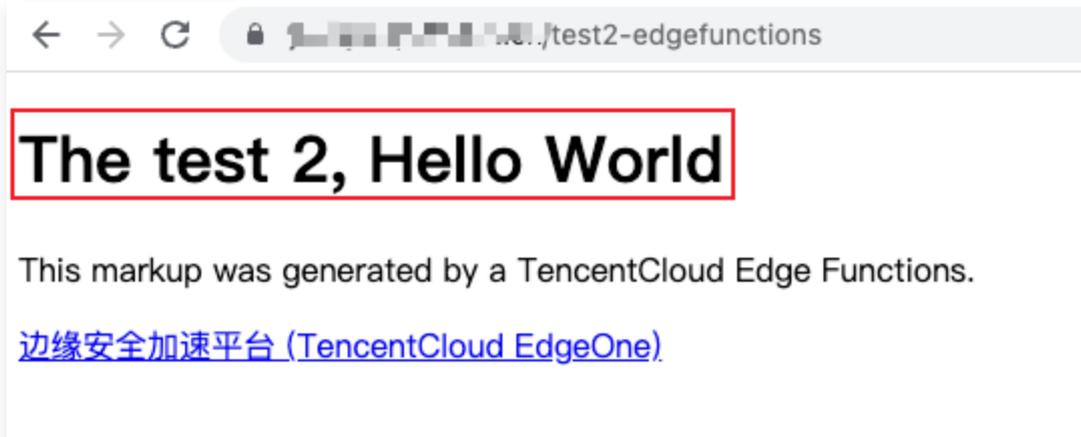


5. 函数 test2 单击下后，单击保存。



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的顺序调整。

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



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。

Web 调试

最近更新时间：2024-05-30 16:31:51

Web 调试页面主要用于调试边缘函数的运行结果，您可以通过在边缘函数内，单击[编辑代码](#)，进入该调试页面内。

操作步骤

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. 在函数管理页面，单击[具体函数名称 > 编辑代码](#)进入到 Web 调试页面。如未创建函数，请通过 [创建并部署函数](#) 指引先完成函数的创建。
4. 调整代码后，单击[保存并部署](#)即可快速将调整后的代码部署到现网，体验现网的 Console 调试、HTTP 请求与响应调试以及请求预览等功能。详细操作请参见 [支持的 Web 调试功能](#)。

❗ 说明：

为了有效地进行 Console 调试、HTTP 请求与响应调试以及请求预览操作，必须确保函数的默认域名已生效，或者已按照 [触发规则](#) 配置了函数。请在确认函数的默认域名已生效或已配置函数触发规则后，再进行上述操作。

支持的 Web 调试能力

Web 调试支持现网如下场景的调试操作：

- **Console 调试：**支持打印 Console 信息，包含 LOG、INFO、NOTICE、DEBUG、WARN 和 ERROR 信息。
 - LOG：输出通用信息，包括程序的状态、变量值等，帮助开发者获取程序运行的即时信息。
 - INFO：输出提示信息，如操作成功、数据加载完成等，向开发者展示程序的正向反馈。
 - DEBUG：输出调试信息，有助于理解程序运行的细节。
 - WARN：输出警告信息，说明程序遇到没立即出错，但值得关注的问题。
 - ERROR：输出错误信息，用于记录程序运行时的错误，需要开发者解决。
 - NOTICE：输出注意信息，介于 INFO 和 WARN 之间，用于提醒开发者重要非紧急的情况。
- **HTTP 请求与响应调试：**支持添加 HTTP 请求头并发送 HTTP 请求，可快速查看响应内容，包含响应状态、响应头和响应体。
 - 自定义 HTTP 请求头：允许用户添加或修改 HTTP 请求头，以模拟不同的请求场景。
 - 即时响应预览：发送请求后，能够立即查看 HTTP 响应，包括响应状态码、响应头和响应体。
- **请求预览：**查看请求结果的响应体内容，支持用户在控制台直观地预览和分析相应数据。

Console 调试

在 HTTP 页面，支持选择请求的方式为 GET、POST、PUT、PATCH、DELETE、HEAD 和 OPTIONS，单击**发送请求**，即可打印代码中的 Console 信息，其类型包含：LOG、INFO、NOTICE、DEBUG、WARN 和 ERROR。

The screenshot shows the HTTP interface with the following components:

- Code Editor:** Contains JavaScript code for a `handleRequest` function that logs HTML content, sends an info message, and sends a warn message.
- HTTP Method:** Set to GET, with the URL `http://webtiaol`.
- Headers:** A table with one header: `Accept` with value `/*/*`.
- Status:** 200.
- Response Headers:**
 - `content-length`: 200
 - `content-type`: text/html; charset=UTF-8
 - `date`: Mon, 25 Mar 2024 12:01:40 GMT
 - `x-edgefunctions-test`: Welcome to use Edge Functions.
 - `x-nws-log-uuid`: 6077307691878375106
- Response Body:** Contains the HTML content generated by the code.
- Console:** Shows three log entries:
 - 2024/3/25 20:01:42 log: Full HTML response.
 - 2024/3/25 20:01:42 info: "info-test"
 - 2024/3/25 20:01:42 warn: "warn-test"

筛选 Console 日志类型

在 HTTP 页面，选择**全部**下拉选项，支持筛选过滤具体的某一类 Console 信息，如下图所示：

The screenshot shows the Console interface with a dropdown menu open for filtering log types. The dropdown menu includes the following options:

- 全部 (All)
- LOG
- INFO
- NOTICE
- DEBUG
- WARN
- ERROR

保留所有调试日志

在 HTTP 页面，勾选**保留日志**，可保留每次点击发送请求后的 Console 信息，如下图所示：

| Console | | | 全部 ▾ | <input checked="" type="checkbox"/> 保留日志 | 清空日志 |
|--------------------|------|--|------|--|------|
| 2024/3/25 20:10:55 | log | "\n <!DOCTYPE html>\n <body>\n <h1>Hello World</h1>\n <p>This markup was generated by TencentCloud Edge Functions.</p>\n TencentCloud EdgeOne\n </body>\n" | | handleRequest 11:12 | |
| 2024/3/25 20:11:03 | info | "info-test" | | handleRequest 12:12 | |
| 2024/3/25 20:11:03 | warn | "warn-test" | | handleRequest 13:12 | |
| 2024/3/25 20:11:05 | log | "\n <!DOCTYPE html>\n <body>\n <h1>Hello World</h1>\n <p>This markup was generated by TencentCloud Edge Functions.</p>\n TencentCloud EdgeOne\n </body>\n" | | handleRequest 11:12 | |
| 2024/3/25 20:11:05 | info | "info-test" | | handleRequest 12:12 | |
| 2024/3/25 20:11:05 | warn | "warn-test" | | handleRequest 13:12 | |

清空 Console 日志

在 HTTP 页面，单击**清空日志**，则可清除当前已打印的 Console 信息。

Console 信息构成：

- 时间：Console 信息打印的时间点，如：2024/3/25 20:01:42。
- 类型：Console 信息的类型，包含：LOG、INFO、NOTICE、DEBUG、WARN 和 ERROR。
- 内容：Console 信息具体类型打印的内容，例如 info-test。
- 函数名称及位置：Console 归属的具体函数及所处代码的行和列，例如 `handleRequest 11:12` 表示为 Console 归属在 `handleRequest` 的函数中，且处于代码的第11行和第12列。

HTTP 请求与响应调试

在 HTTP 页面，按场景添加需要测试的请求头，单击**发送请求**，获取响应状态码、响应头和响应 body。

```
1 const html = `
2 <!DOCTYPE html>
3 <body>
4 <h1>Hello World</h1>
5 <p>This markup was generated by TencentCloud Edge Functions.</p>
6 <a href="https://cloud.tencent.com/product/teo">TencentCloud Edge0ne</a>
7 </body>
8 `;
9
10 async function handleRequest(request) {
11   const headers = {
12     'content-type': 'text/html; charset=UTF-8',
13     'x-edgefunctions-test': 'Welcome to use Edge Functions.',
14   };
15
16   // Check if the 'access-token' in the request headers is '123'
17   if (request.headers.get('access-token') === '123') {
18     // If so, add 'authorization' to the response headers
19     headers['authorization'] = 'pass';
20   }
21
22   return new Response(html, {
23     headers: headers,
24   });
25 }
26
27 addEventListener('fetch', event => {
28   event.respondWith(handleRequest(event.request));
29 });
```

HTTP 预览

GET http://webtiac 55 发送请求

Headers

access-token 123 添加请求头

Status 200

Headers

- authorization: pass
- content-length: 200
- content-type: text/html; charset=UTF-8
- date: Mon, 25 Mar 2024 07:45:19 GMT
- x-edgefunctions-test: Welcome to use Edge Functions.
- x-nws-log-uuid: 13758604057989290199

Body

```
"\n <!DOCTYPE html>\n <body>\n <h1>Hello World</h1>\n <p>This markup was generated by TencentCloud Edge Functions.</p>\n <a href="https://cloud.tencent.com/product/teo">TencentCloud Edge0ne</a>\n </body>\n"
```

Console 全部 保留日志 清空日志

暂无日志

上图为单击**添加请求头** access-token，其值设置为123，代码语义逻辑为判断请求头包含 access-token 且其值为 123，则返回响应头 authorization 且其值设置为 pass。

请求预览

在预览页面，单击**发送请求**则可直接渲染并预览响应体 body 的内容。

webtiaolay-zone- 390

X

保存并部署

HTTP

预览

Runtime API 文档 示例函数

```
1 const html = `
2 <!DOCTYPE html>
3 <body>
4 <h1>Hello World</h1>
5 <p>This markup was generated by TencentCloud Edge Functions.</p>
6 <a href="https://cloud.tencent.com/product/teo">TencentCloud EdgeOne</a>
7 </body>
8 `;
9
10 async function handleRequest(request) {
11   return new Response(html, {
12     headers: {
13       'content-type': 'text/html; charset=UTF-8',
14       'x-edgefunctions-test': 'Welcome to use Edge Functions.',
15     },
16   });
17 }
18
19 addEventListener('fetch', event => {
20   event.respondWith(handleRequest(event.request));
21 });
```

http://webtiaolay-zone- 00.eo-edgefunctions.com

发送请求

Hello World

This markup was generated by TencentCloud Edge Functions.

[TencentCloud EdgeOne](#)

Console

全部 保留日志 清空日志



暂无日志

环境变量

最近更新时间：2024-07-25 16:21:41

操作场景

环境变量是在边缘函数的运行环境中设置的键值对，该键值对可以在边缘函数脚本中作为全局变量访问，而不需要显式地导入或初始化，通过使用环境变量，可实现函数代码与配置的解耦，其常见作用如下：

- **配置解耦**：环境变量允许开发者在不修改函数代码的情况下，对边缘函数进行配置，这意味着开发者可以为不同的环境（如开发、测试和生产环境）设置不同的环境变量值，从而控制函数的行为。
- **安全性**：通过将敏感信息（如 API 密钥）存储在环境变量中，而不是硬编码在代码中，可提高应用程序的安全性。这样做可以减少代码库中的敏感信息泄露风险，并简化密钥管理。
- **灵活性**：环境变量为边缘函数提供了很好的灵活性，使开发者能够根据需要调整边缘函数的行为。例如，开发者可以使用环境变量实现灰度发布，通过控制不同用户群体访问不同版本的代码或配置。

操作介绍

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. 在函数管理页面，单击**具体函数名称**，在页面底部找到**环境变量**模块，如未创建函数，请通过 [创建并部署函数](#) 指引先完成函数的创建。

创建并部署环境变量

1. 在环境变量模块，单击**快速添加**，配置相关参数。

新建环境变量 ×

变量名

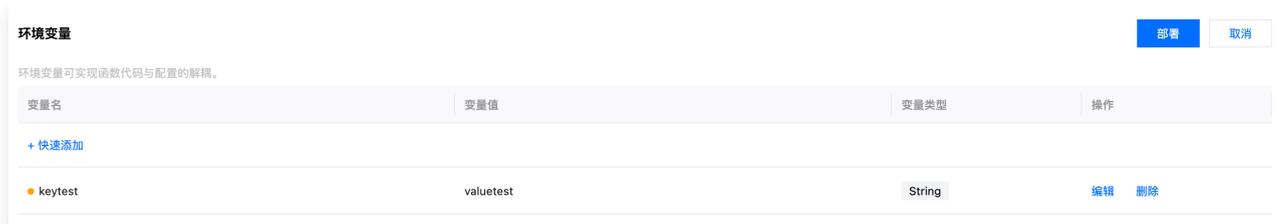
只能包含大小写字母、数字，特殊字符仅支持 @.-_，并且长度限制为 1 到 64 个字符。

变量类型 String JSON

变量值

| 参数名称 | 说明 |
|------|--|
| 变量名 | 必填项，只能包含大小写字母、数字，特殊字符仅支持 @、.、-、_、并且长度限制为 1 到 64 个字符；变量名不允许重复，且创建后无法修改。如：keytest。 |
| 变量类型 | 必填项，支持 String 和 JSON 两种类型的选择。 <ul style="list-style-type: none">String：选择该项后，输入变量值的内容会以字符串的方式保存，具体使用请参考 边缘函数引用环境变量 的变量类型为 String 章节。JSON：选择该项后，输入变量值的内容会以 JSON 数组方式保存，边缘函数会自动把该变量值解析为 JavaScript 对象，无需手动调用 JSON.parse() 处理，具体使用请参考 边缘函数引用环境变量 的变量类型为 JSON 章节。 |
| 变量值 | 必填项，最大支持 5 kb，如：类型选择为 String 的情况下变量值输入 valuetest，如变量类型为 JSON，变量值会校验输入的内容是否 JSON 数据结构，如非 JSON 数据结构，则会有异常提示。 |

2. 单击**确定**，即可完成环境变量的创建，单个边缘函数最多支持创建 64 个环境变量。



3. 单击部署操作后，则可生效。

边缘函数引入环境变量

- 如函数引用的环境变量未包含特殊字符：`@`、`.`、`-`、`,`、`'`、`\"` 可通过 `env.envname` 形式引用，如：环境变量 `envname` 为：`keytest`，则边缘函数代码中引用的方式为 `env.keytest`，具体使用可参考变量类型为 `String` 章节。
- 如函数引用的环境变量名称包含特殊字符：`@`、`.`、`-`、`,`、`'`、`\"` 可通过 `env['envname']` 形式引用，如：环境变量 `envname` 为：`test-@.-a`，则边缘函数代码中引用的方式为 `env['test-@.-a']`。

如下为边缘函数引用环境变量类型分别为 `String` 和 `JSON` 的示例代码，开发者可按照实际情况进行调整。

变量类型为 `String`

通过上述创建并部署环境变量步骤，创建环境变量名为 `keytest`，变量值为 `valuetest`，边缘函数引用参考如下：

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

// 处理请求的函数
async function handleRequest(request) {
  // 从环境变量获取值，此环境变量需在边缘函数环境变量已创建并部署
  const valueFromEnv = env.keytest;
  // 创建响应
  const response = new Response(valueFromEnv, {
    headers: {
      'Content-Type': 'text/plain' // 设置响应的 Content-Type
    }
  });
};

// 返回响应
return response;
}
```

上述代码的语义简述为获取函数已创建并部署的环境变量，并将其以文本的方式响应给客户端。

部署代码，通过访问触发规则或者函数默认访问即可查看结果。



变量类型为 JSON

重复创建并部署环境变量步骤，创建并部署类型为 JSON 变量名为 keytestjson 的环境变量，如下所示：

环境变量

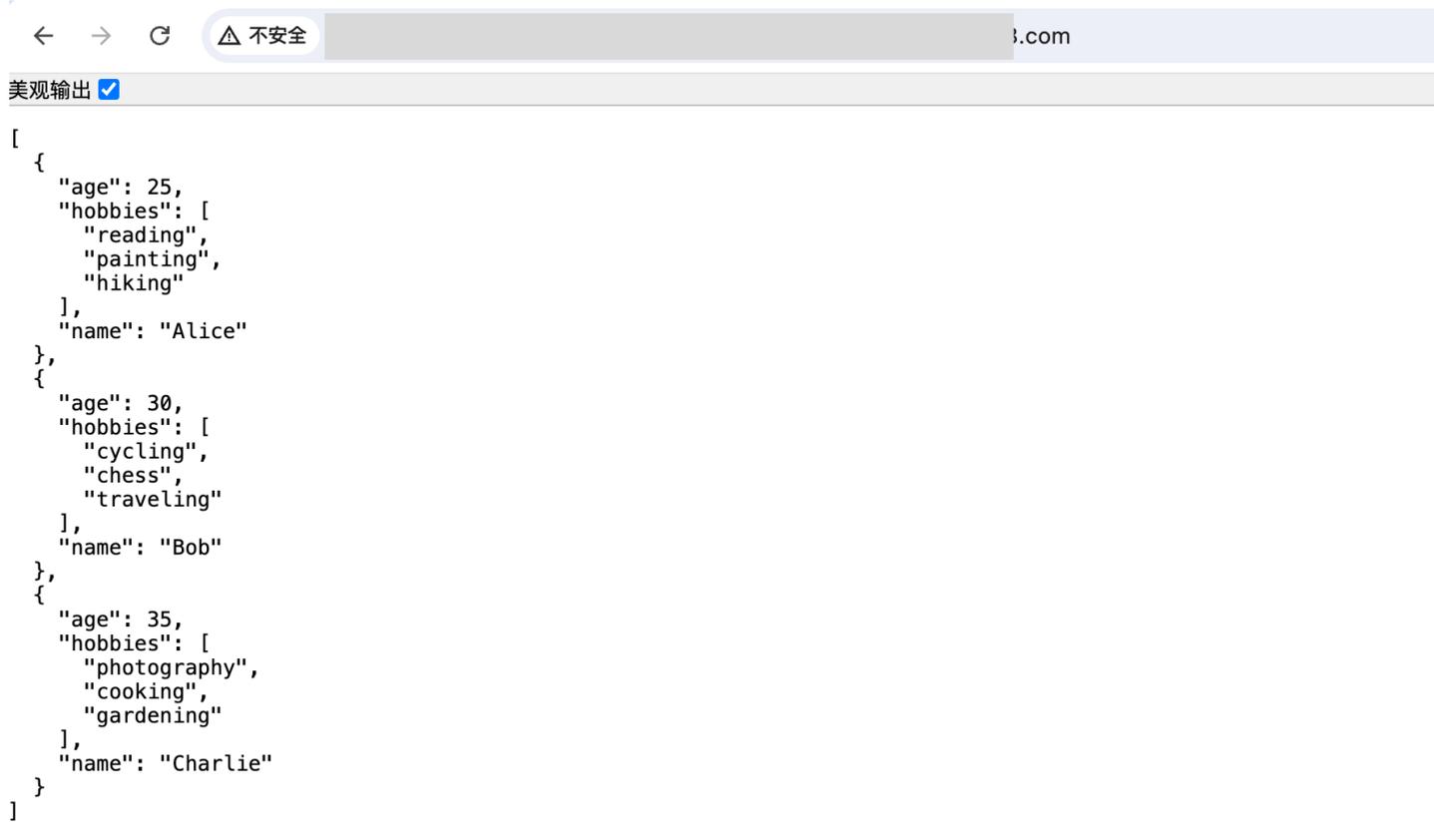
环境变量可实现函数代码与配置的解耦。

| 变量名 | 变量值 | 变量类型 | 操作 |
|-------------|---|--------|-------|
| + 快速添加 | | | |
| keytestjson | [{"name": "Alice", "age": 25, "hobbies": ["reading", "painting", "hiking"]}, ...] | JSON | 编辑 删除 |
| keytest | valuetest | String | 编辑 删除 |

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

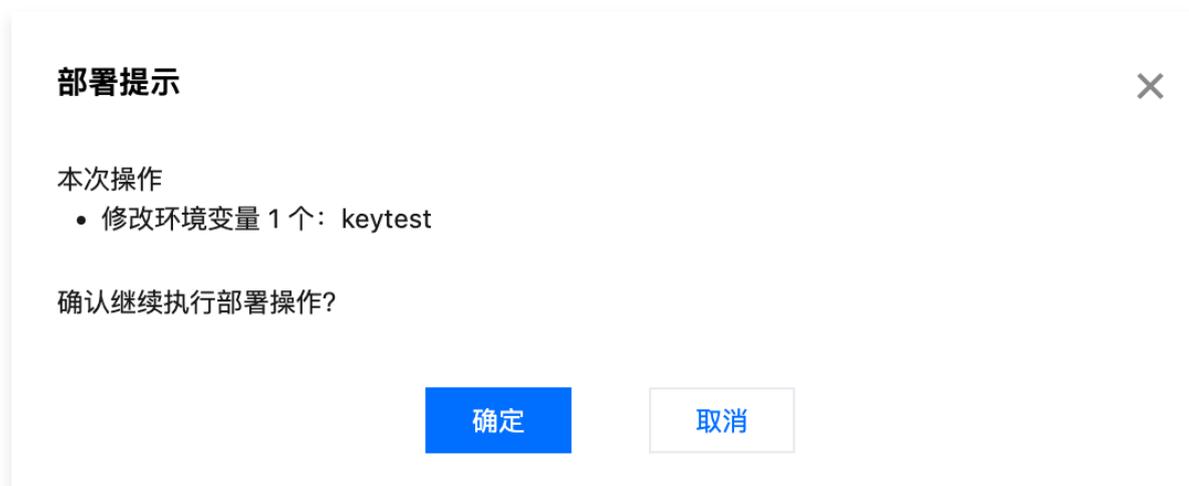
async function handleRequest(request) {
  // 从环境变量获取值，此环境变量需在边缘函数环境变量已创建并部署
  const myJsonData = env.keytestjson;
  // 创建响应体
  const response = new Response(JSON.stringify(myJsonData), {
    headers: {
      'Content-Type': 'application/json'
    }
  });
  // 返回响应
  return response;
}
```

上述代码的语义简述为获取函数已创建并部署的环境变量，并将其以 JSON 数据的方式响应给客户端。部署代码，通过访问触发规则或者函数默认访问即可查看结果。



编辑环境变量

1. 在环境变量模块，单击某环境变量的编辑，在弹窗中支持变更变量类型和变量值，单击**确定**，即可保存编辑内容。
2. 在环境变量模块，单击**部署**，弹窗部署提示本次操作的具体更新内容，单击**确定**即可完成部署。



删除环境变量

在环境变量板块，单击某环境变量的**删除**后，单击**部署**，弹窗部署提示本次操作具体删除的环境变量，单击**确定**即可完成部署。

部署提示



本次操作

- 删除环境变量 1 个: keytest
- ⚠ 如删除函数已引用的环境变量，请调整函数代码中相关引用的逻辑。

确认继续执行部署操作？

确定

取消

⚠ 注意：

如函数已引用的环境变量经上述操作删除后，请调整函数代码中相关引用的逻辑。

AI 代码助手

最近更新时间：2024-12-23 15:32:31

概述

为了降低边缘函数开发者的体验门槛和开发难度，EdgeOne 推出了专为边缘函数服务的 AI 代码助手，该工具旨在通过自然语言交互，帮助开发者更高效地编码、修复和解释代码，从而提升开发效率和代码质量。边缘函数 AI 代码助手提供以下主要功能：

- **代码生成**：可以通过自然语言描述需求，代码助手将自动生成相应的边缘函数的业务代码。例如：输入“创建一个 A/B Test 的边缘函数”，助手将生成相应的代码片段。
- **代码修复**：代码助手能够自动检测代码中的错误，并提供修复建议。您只需上传或选中有问题代码，助手将分析并提供解决方案。
- **代码解释**：代码助手能够对现有的边缘函数代码进行详细解释，帮助您理解代码的功能和逻辑，您可上传代码片段，助手将逐行解释其作用。
- **功能答疑**：您可就边缘函数的功能和特性提出问题，助手将提供详细的解答和指导。例如：请解释边缘函数的 addEventListener 使用方法。

操作介绍

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. 在函数管理页面，单击**具体函数名称**，在基本信息页面单击**编辑代码**。
4. 在编辑代码详情页面，单击右上角的  AI 助手 即可唤起 AI 助手。



AI 助手当前支持的主要能力包括：代码生成、代码修复、代码解释和功能答疑，接下来依次介绍具体的用法。

代码生成

可通过自然语言描述业务需求，代码助手将自动生成相应的边缘函数的业务代码，如：

请用边缘函数实现，如果请求来自中国，则用中文 HTML 响应：中国用户，您好；如果请求来自美国，则用英文 HTML 响应：美国用户，您好；其他国家的客户端请求则用英文 HTML 响应：暂不支持服务。

AI 助手则会根据您的业务提示词生成对应的边缘函数代码

边缘函数 AI 助手

如果请求来自美国，则响应 HTML 的内容为英文的欢迎你来到美国，如果来自中国，中文 HTML 响应欢迎你来到中国，其他国家的客户端请求则用英文响应咱不支持的区域

 边缘函数 AI 助手

为了实现根据请求来源区域返回不同语言的HTML响应，我们可以使用边缘函数来检测请求的地理位置，并根据该信息返回相应的HTML内容。以下是实现该功能的示例代码：

```
javascript

const htmlEnUs = `
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Welcome</title>
</head>
<body>
  <h1>Welcome to the United States</h1>
</body>
</html>
`;

const htmlZhCn = `
```

您可以通过单击复制 icon  或者插入到光标处  将生成的代码快速填充到边缘函数的代码框。

代码修复

您可选中代码后，在 AI 助手的输入框输入 `/`，选择代码修复。

/代码解释 解释一段代码运行原理及逻辑

/代码修复 修复一段代码中的 Bug

 line: 1-64
const htmlEnUsa = ` <!DOCTYPE html> <html lang="en"> <head> <meta charset="UTF-8"> <title>Welc...

输入参考提示词：**请检测问题并修复代码**，按下回车键，AI 助手则会指出代码问题，并提供修复后的代码。

边缘函数 AI 助手

**代码修复** 检测问题并修复代码

边缘函数 AI 助手

在用户提供的边缘函数中，存在以下问题：

1. 在 switch 语句中，responseHtml = htmlEnUs; 应该是 responseHtml = htmlEnUsa;，因为定义的变量名是 htmlEnUsa 而不是 htmlEnUs。
2. async funtion handleRequest(request) 中的 funtion 应该是 function，这是一个拼写错误。

修复后的代码如下：

javascript



```
const htmlEnUsa = `  
<!DOCTYPE html>  
<html lang="en">  
<head>  
  <meta charset="UTF-8">  
  <title>Welcome</title>  
</head>  
<body>
```



line: 1-64

const htmlEnUsa = `<!DOCTYPE html> <html lang="en"> <head> <meta charset="UTF-8"> <title>Welc...

问我任何问题 或 键入 / 获取指令

代码解释

您可选中代码后，在 AI 助手的输入框输入 `/`，选择代码解释。

/代码解释

解释一段代码运行原理及逻辑

/代码修复

修复一段代码中的 Bug

</head>



line: 1-64

```
const htmlEnUsa = ` <!DOCTYPE html> <html lang="en"> <head> <me...`
```

/

输入参考提示词：**请解释代码**，按下回车键，则 AI 助手会详细解释代码的作用和用法。

边缘函数 AI 助手



line: 1-64

```
const htmlEnUsa = ` <!DOCTYPE html> <html lang="en"> <head> <meta charset="UTF-8"> <titl...`
```

代码解释 请解释代码

边缘函数 AI 助手

边缘函数作用概括

用户提供的边缘函数的主要作用是根据客户端的地理位置（国家代码）返回定制化的HTML页面。如果客户端来自美国（US），则返回英文欢迎页面；如果来自中国（CN），则返回中文欢迎页面；对于其他地区的客户端，返回一个默认的不支持该区域的页面。

边缘函数解析

1. HTML模板定义

用户定义了三个HTML模板字符串，分别用于美国、中国和其他地区的响应内容。

javascript



```
const htmlEnUsa = ` <!DOCTYPE html> <html lang="en"> <head>
```

功能答疑

您可就边缘函数的功能和特性提出问题，AI 助手将提供详细的解答和指导，如：

请详细解释 `request.eo.geo` 的用法。

边缘函数 AI 助手

请详细解释request.eo.geo的用法

 边缘函数 AI 助手

`request.eo.geo` 是腾讯云边缘函数中的一个对象，用于获取客户端请求的地理位置信息。这个对象包含了客户端的经纬度、国家、省份、城市等信息。以下是 `request.eo.geo` 的详细用法：

获取地理位置信息

您可以通过 `request.eo.geo` 对象获取客户端的地理位置信息。例如：

```
javascript

exports.main_handler = async (event, context) => {
  const { latitude, longitude, country, province, city } = event.request.eo.geo
  return {
    statusCode: 200,
    body: JSON.stringify({
      message: '获取地理位置信息成功',
      latitude,
      longitude,
      country,
      province,
      city
    })
  };
};
```

在 AI 助手响应的回答中，您可获取到详细文档的具体出处，可进一步单击前往了解详情，如：

边缘函数 AI 助手



```
const geo = event.request.eo.geo;
if (!geo) {
  return {
    statusCode: 400,
    body: JSON.stringify({
      message: '无法获取地理位置信息'
    })
  };
}

const { latitude, longitude, country, province, city } = geo;
return {
  statusCode: 200,
  body: JSON.stringify({
    message: '获取地理位置信息成功',
    latitude,
    longitude,
    country,
    province,
    city
  })
};
};
```

参考文档

[边缘函数请求对象](#)

通过上述代码示例和解释，你应该能够理解 `request.eo.geo` 的用法，并能够在你的边缘函数中正确地使用它来获取和处理客户端的地理位置信息。



内容由 AI 生成，仅供参考

Runtime APIs

addEventListener

最近更新时间：2024-04-18 14:38:02

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): void;
```

参数

| 参数名称 | 类型 | 必填 | 说明 |
|----------|---|----|--|
| type | string | 是 | 事件类型。 <ul style="list-style-type: none">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 | <code>(event: FetchEvent) => void</code> | 是 | 事件监听器。用于处理事件回调。 <ul style="list-style-type: none">注册 <code>fetch</code> 请求事件生成 <code>FetchEvent</code> 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：2024-08-23 15:07:21

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。 <ul style="list-style-type: none">如果该值为 "default" 则表示默认实例，也可直接使用 <code>caches.default</code> 获取默认实例。 |

实例方法

match

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

Promise<Response | undefined>

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 Request | 是 | 请求对象，headers 说明如下。 <ul style="list-style-type: none"> GET request 只支持 GET 方法，当类型为 string 时，将被作为 URL 构造 Request 对象。 Range 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。 If-Modified-Since request 包含 If-Modified-Since 头部时，如果缓存的 Response 存在 Last-Modified 头部，且 Last-Modified 与 If-Modified-Since 相等，返回 304 响应。 If-None-Match request 包含 If-None-Match 头部时，如果缓存的 Response 存在 ETag 头部，且 ETag 与 If-None-Match 相等，返回 304 响应。 |
| options | MatchOptions | 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 Request | 是 | 缓存 key，说明如下。 <ul style="list-style-type: none"> GET 参数 request 仅支持 GET 方法，其他方法，将抛出参数错误。 string 当参数 request 类型为 string 时，将被作为 URL 构造 Request 对象。 |
| response | Response | 是 | 缓存内容，说明如下。 <ul style="list-style-type: none"> Cache-Control 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。 Pragma 当 Cache-Control 未设置，并且 Pragma 为 no-cache。此时表示不缓存。 ETag 当 cache.match 参数 request 包含 If-None-Match 头部时，可关联 ETag 使用。 Last-Modified 当 cache.match 参数 request 包含 If-Modified-Since 头部时，可关联 Last-Modified 使用。 416 Range Not Satisfiable 当参数 response 对象为 416 Range Not Satisfiable 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 |
|----------------------|--|----|---|
| <code>request</code> | <code>string</code> <code>Request</code> | 是 | 缓存 key，说明如下。 <ul style="list-style-type: none">• GET 参数 <code>request</code> 仅支持 GET 方法• <code>string</code> 当参数 <code>request</code> 类型为 <code>string</code> 时，将被作为 URL 构造 <code>Request</code> 对象。 |
| <code>options</code> | <code>DeleteOptions</code> | 否 | 配置选项。 |

DeleteOptions

| 属性名 | 类型 | 示例值 | 说明 |
|---------------------------|----------------------|-------------------|--|
| <code>ignoreMethod</code> | <code>boolean</code> | <code>true</code> | 是否忽略 <code>request</code> 的方法名。为 <code>true</code> 时，会忽略 <code>Request</code> 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：2024-04-18 14:38:02

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 Set-Cookie 字符串。 |
| isSetCookie | boolean | 否 | 参数 cookieStr 是否是 Set-Cookie 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 | <code>Cookie</code> 名称，取值说明如下。 <ul style="list-style-type: none">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 Cookie 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 HTTP Date 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, 详情参见 [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 | Cookie 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，详情请参见 [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 domain 和 path 可支持 *，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 GMT; path=/; domain=tencentcloud.com;
samesite=tencentcloud.com', true);

  // 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：2024-04-18 14:38:02

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 | <code>Uint8Array</code> | 是 | 编码器输出值。 |

- 返回值 `EncodeIntoResult`

| 属性名 | 类型 | 说明 |
|---------|--------|-------------------------------------|
| read | number | 已转换为 UTF-8 的 UTF-16 单元数。 |
| written | number | 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?:  
DecoderOptions | undefined): TextEncoder;
```

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 <code>UTF-8</code> 。可选的 <code>label</code> 值参考 MDN 官方文档 。 |
| options | DecoderOptions 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 byte-order marker 。 |

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, options?: DecodeOptions | undefined): string;
```

⚠ 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 | ArrayBuffer ArrayBufferView undefined | 否 | 待解码的字节流。 <ul style="list-style-type: none">buffer 最大长度为 100M，超出长度会抛出异常。 |
| options | DecodeOptions | 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 false，取值说明如下。 <ul style="list-style-type: none">true 表示以 <code>chunk</code> 的方式处理数据，即流式解码。false 表示 <code>chunk</code> 已结束或未使用 <code>chunk</code> 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：2025-04-24 17:00:42

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<Response>
```

参数

| 参数名称 | 类型 | 必填 | 说明 |
|-------------|----------------------------------|----|---|
| request | string Request | 是 | 指定将要获取的请求资源。 |
| requestInit | RequestInit | 否 | 请求对象的初始化配置项。详情请参见 RequestInit 。 |

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

- 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。
- `fetch(request)` 指定的 `request.url` 中的 HOST 和客户端请求 URL 中的 HOST 相同。
- `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。

超时时间设置

`fetch` 支持通过 `requestInit.eo.timeoutSetting` 配置请求的超时时间，默认超时时间为15秒，最大可设置为300秒。

```
fetch(url, {
  eo: {
    timeoutSetting: {
      connectTimeout: 60000, // 建立连接的超时时间，单位：毫秒
      readTimeout: 60000, // 读取响应的超时时间，单位：毫秒
      writeTimeout: 60000, // 发送请求的超时时间，单位：毫秒
    }
  }
});
```

```
    }  
  }  
})
```

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`。
- 时间限制：边缘函数中使用 `fetch` 发起请求的默认超时时间为 15 秒，最大可设置为 300 秒。

⚠ 注意

每一次重定向都会会计入请求次数，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](#)
- [示例函数: 图片自适应格式](#)
- [示例函数: 图片响应式缩放](#)

FetchEvent

最近更新时间：2024-04-18 14:38:02

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 | <code>Response</code> <code>Promise<Response></code> | 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 | <code>Promise<Response></code> | 是 | 等待完成的 <code>Promise</code> 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：2025-01-17 11:49:11

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 | 初始化 Headers 对象，参数类型说明如下： <ul style="list-style-type: none">object 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。Array<[string, string]> 数组的每一个元素为 <code>key/value</code> 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。Headers 拷贝 Headers 对象，并把所有字段初始化到新的 Headers 对象中。 |

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。

⚠ 注意：

`headers` 对象写入时存在以下限制：

- `name` 长度不能超过 255 字节，且只能由数字、大小写字母、-、_ 组成，否则将抛出异常。

- value 长度不能超过 4095 字节，字符串首尾的空白字符 '\n'、'\r'、'\t'、以及空格会被自动去除，且字符串中间不能包含 '\0'、'\n'、'\r'，否则将抛出异常。

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: (value: string, name: 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。

⚠ 注意:

`headers` 对象写入时存在以下限制:

- `name` 长度不能超过 255 字节，且只能由数字、大小写字母、`-`、`_` 组成，否则将抛出异常。
- `value` 长度不能超过 4095 字节，字符串首尾的空白字符 `\n`、`\r`、`\t`、以及空格会被自动去除，且字符串中间不能包含 `\0`、`\n`、`\r`，否则将抛出异常。

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：2025-06-09 15:29:31

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 Request | 是 | URL 字符串或 <code>Request</code> 对象。 |
| options | RequestInit | 否 | <code>Request</code> 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 (<code>GET</code> 、 <code>POST</code> 等)。 |
| headers | Headers | 否 | - | 请求头部信息。 |
| body | string Blob ArrayBuffer ArrayBufferView ReadableStream | 否 | - | 请求体。 |

| | | | | |
|-------------|---|---|----------|--|
| redirect | string | 否 | follow | 重定向策略，支持 manual、error 和 follow。 |
| maxFollow | number | 否 | 12 | 最大可重定向次数。 |
| version | string | 否 | HTTP/1.1 | HTTP 版本，支持 HTTP/1.0、HTTP/1.1 和 HTTP/2.0。 |
| copyHeaders | boolean | 否 | - | 非 Web APIs 标准选项，表示是否拷贝传入的 Request 对象的 headers。 |
| eo | RequestInitEoProperties | 否 | - | 非 Web APIs 标准选项，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 fetch 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： <ul style="list-style-type: none"> IP 不允许带 scheme 以及端口号。 IPv6 无需使用方括号包裹。 |
| image | ImageProperties | 否 | 图片处理参数配置项。 |

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 `follow`。

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 | GeoProperties | 描述客户请求的位置信息。 | - |
| clientIp | string | 客户端IP地址 | 192.168.1.1 |
| uuid | string | 请求的唯一标识，作用同 EO-LOG-UUID | - |

GeoProperties

描述客户请求的位置信息。

| 属性名 | 类型 | 说明 | 示例值 |
|--------------------|--------|---|-----------|
| asn | number | ASN | 132203 |
| countryName | string | 国家名 | Singapore |
| countryCodeAlpha2 | string | 国家的 ISO-3611 alpha2 代码 | SG |
| countryCodeAlpha3 | string | 国家的 ISO-3611 alpha3 代码 | SGP |
| countryCodeNumeric | string | 国家的 ISO-3611 numeric 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。 <ul style="list-style-type: none"><code>true</code> 复制原对象的请求头。<code>false</code> 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`。

参数

| 参数名称 | 类型 | 必填 | 说明 |
|---------|----------------------|----|-----------------------------|
| cookies | <code>Cookies</code> | 否 | 新的 <code>Cookies</code> 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：2024-04-18 09:47:31

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 | null | undefined, init?: ResponseInit);
```

参数

| 参数名称 | 类型 | 必填 | 说明 |
|------|---|----|------------------------------------|
| body | string ArrayBuffer Blob ReadableStream null undefined | 是 | <code>Response</code> 对象的 body 内容。 |
| init | ResponseInit | 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 | Headers | 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

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

响应状态代码。

statusText

```
// response.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): Response;
```

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。 <ul style="list-style-type: none"><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：2025-06-09 15:20:31

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 | ReaderOptions | 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 | <code>Reader</code> 类型，默认值为 <code>undefined</code> ，取值说明如下。 <ul style="list-style-type: none"><code>undefined</code> 创建 ReadableStreamDefaultReader 类型的 <code>Reader</code>。<code>byob</code> 创建 ReadableStreamBYOBReader 类型的 <code>Reader</code>。 |

pipeThrough

```
readable.pipeThrough(transfromStream: TransfromStream, options?:  
PipeToOptions): ReadableStream;
```

流的管道处理。将当前可读流数据传输到参数 `transfromStream` 的 `writable` 端，并返回 `transfromStream` 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 | TransfromStream | 是 | 当前流传输到的目标对象。 |
| options | PipeToOptions | 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 | AbortSignal | 否 | 当 <code>signal</code> 被 abort 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 | WritableStream | 是 | 可写流。 |
| options | PipeToOptions | 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：2024-09-06 17:01:51

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: boolean}>;
```

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：2024-09-06 17:01:51

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：2025-01-08 11:38:22

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,
writableStrategy?: WritableStrategy);
```

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 | WritableStrategy | 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：2024-09-06 17:01:51

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：2024-04-18 14:38:02

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 | <code>Chunk</code> | 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：2024-06-04 16:07:41

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 | Int8Array Uint8Array Uint8ClampedArray Int16Array Uint16Array Int32Array Uint32Array BigInt64Array BigUint64Array | 是 | 随机数缓冲区，不超过 65536 字节。详情参见 TypedArray 。 |
|--------|---|---|---|

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](#)。

对于 `RSA-OAEP` 算法，要求 data 长度不能超过 $\text{modulusLength}/8 - 2 * \text{hLen} - 2$ ，其中 hLen 的取值逻辑为：

- SHA-1: hLen = 20 byte
- SHA-256: hLen = 32 byte
- SHA-384: hLen = 48 byte
- SHA-512: hLen = 64 byte

对于 `AES-CTR`，`AES-CBC`，`AES-GCM`，限制 data 长度 1MB。

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](#)。

对于 `RSA-OAEP` 算法，data 长度为 $\text{modulusLength}/8$ 。

对于 `AES-CTR`，`AES-CBC`，`AES-GCM`，限制 data 长度 1MB。

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，详情参见 [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 |
|-------------|---------------|----|------------------|
| type | string | 是 | 密钥类型。 |
| extractable | boolean | 是 | 密钥是否可导出。 |
| algorithm | object | 是 | 算法相关, 包含算法需要的字段。 |
| usages | Array<string> | 是 | 密钥的用途。 |

CryptoKeyPair

`CryptoKeyPair` 表示用加密算法生成的密钥对，详情参见 [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 |
|------------|-----------|----|---------------------------------|
| privateKey | CryptoKey | 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。 |
| publicKey | CryptoKey | 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() | digest() |
|-------------------|------------------------|--------------------|--------------------------|-----------------------------|---------------|-------------|-------------|----------|
| RSASSA-PKCS1-v1_5 | - | ✓ | - | - | ✓ | ✓ | ✓ | - |
| RSA-PSS | - | ✓ | - | - | ✓ | ✓ | ✓ | - |
| RSA-OAEP | ✓ | - | ✓ | - | ✓ | ✓ | ✓ | - |
| 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](#)

Web Standards

最近更新时间：2023-11-21 09:49:42

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](#)。

Images

ImageProperties

最近更新时间：2024-09-19 14:44:42

边缘函数支持在发起 [Fetch](#) 请求或初始化 [Request](#) 对象时，通过 [RequestInitEoProperties](#) 中的 [ImageProperties](#) 参数自定义图片处理行为。[ImageProperties](#) 为非 Web APIs 标准选项，使用方式如下。发起 [Fetch](#) 请求时，设置 [ImageProperties](#) 参数：

```
addEventListener('fetch', (event) => {
  const response = fetch(event.request, { eo: { image: { format: "avif" } } });
});
event.respondWith(response);
});
```

初始化 [Request](#) 对象时，设置 [ImageProperties](#) 参数：

```
async function handleRequest() {
  const request = new Request("https://www.example.com/test.jpg", { eo: { image: { format: "avif" } } });
  const response = await fetch(request);
  return response;
}

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

边缘函数支持通过设置不同的 [ImageProperties](#) 参数，实现多种图片处理能力，包括 [图片格式转换](#)、[图片质量变换](#)、[图片缩放](#)、[图片裁剪](#)，具体配置如下。

格式转换

通过指定 `format` 参数，将原图转换为指定格式。

| 参数名称 | 类型 | 必填 | 说明 |
|---------------------|---------------------|----|---|
| <code>format</code> | <code>string</code> | 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> 。 |

| | | | |
|-------------------|---------|---|--------------------------------|
| avoidSizeIncrease | boolean | 否 | 配置此参数时，如果处理后图片体积大于原图，会返回原图不处理。 |
|-------------------|---------|---|--------------------------------|

示例：

```
// 将原图转换为 avif 格式
eo: { image: { format: "avif" } }
```

质量变换

通过指定 `quality` 等参数，对图片的质量进行调节。

| 参数名称 | 类型 | 必填 | 说明 |
|-------------------|-----------------|----|--|
| quality | string number | 否 | 图片的绝对质量，取值范围 0 - 100；取原图质量和指定质量的最小值；quality 后面加 "!" 表示强制使用指定值，例如："90!"。 |
| rquality | string number | 否 | 图片的相对质量，取值范围 0 - 100，数值以原图质量为标准。例如原图质量为 80，rquality 设置为 80 后，得到处理结果图的照片质量为 64 (80x80%)。 |
| lquality | string number | 否 | 图片的最低质量，取值范围 0 - 100，设置结果图的质量参数最小值； <ul style="list-style-type: none"> 例如原图质量为 85，将 lquality 设置为 80 后，处理结果图的照片质量为 85； 例如原图质量为 60，将 lquality 设置为 80 后，处理结果图的照片质量会被提升至 80； |
| avoidSizeIncrease | boolean | 否 | 配置此参数时，如果处理后图片体积大于原图，会返回原图不处理。 |

ⓘ 说明：

质量变换参数有生效优先级，默认 `lquality > quality > rquality`，如果同时配置，仅最高优先级生效。

示例：

```
// 设置图片的绝对质量为 80
eo: { image: { quality: 80 } }

// 设置图片的相对质量为 80
```

```
eo: { image: { rquality: 80 } }
```

```
// 设置图片的最低质量 80
```

```
eo: { image: { lquality: 80 } }
```

图片缩放

图片缩放操作受 `fit` 参数控制，通过设定不同的 `fit` 参数，可以实现不同类型的缩放操作。

| 参数名称 | 类型 | 必填 | 说明 |
|------|--------|----|--|
| fit | string | 否 | 设置不同的图片缩放与裁剪模式，支持的 fit 参数取值为： <ul style="list-style-type: none">• contain: (默认) 对图片进行保留宽高比的缩放操作。• cover: 按像素值对图片进行不保留宽高比的缩放操作。• percent: 按百分比对图片进行不保留宽高比的缩放操作。 |

说明:

不同 `fit` 参数对应的行为，将在下文进行详细说明。

等比缩放

未设置 `fit` 参数或 `fit` 参数设置为 `contain` 时，对图片进行保留宽高比的缩放操作。

| 参数名称 | 类型 | 必填 | 说明 |
|--------|-----------------|----|--|
| width | string number | 否 | 指定目标图片宽度缩放为 width 像素；仅设置 width 参数时，高度等比缩放。 |
| height | string number | 否 | 指定目标图片高度缩放为 height 像素；仅设置 height 参数时，宽度等比缩放。 |
| long | string number | 否 | 指定目标图片长边缩放为 long 像素；短边等比缩放。 |
| short | string number | 否 | 指定目标图片短边缩放为 short 像素；长边等比缩放。 |

说明:

- 上述参数有生效优先级，`long` > `short` > (`width` | `height`)，如果同时配置，仅最高优先级生效。

- 若同时配置 `width` 和 `height` 参数，则限定缩略图的宽度和高度的最大值分别为 `width` 和 `height`，进行等比缩放。

除上述功能外，也支持按像素数量进行等比缩放。

| 参数名称 | 类型 | 必填 | 说明 |
|-------------------|------------------------------|----|--|
| <code>area</code> | <code>string number</code> | 否 | 等比缩放图片，缩放后的图像，总像素数量不超过 <code>area</code> 。 |

示例：

```
// 设置按长边进行等比缩放，缩放后长边长度为 100px
eo: { image: { long: 100 } }

// 设置按短边进行等比缩放，缩放后短边长度为 100px
eo: { image: { short: 100 } }

// 设置按宽度进行等比缩放，缩放后宽度为 100px
eo: { image: { width: 100 } }

// 设置按高度进行等比缩放，缩放后高度为 100px
eo: { image: { height: 100 } }

// 设置按高度和宽度进行等比缩放，缩放后宽度最大为 100px，高度最大为 100px，等比缩放
eo: { image: { width: 100, height: 100 } }

// 设置按总像素数量进行等比缩放，缩放后总像素值不超过 10000px
eo: { image: { area: 10000 } }
```

非等比缩放

`fit` 参数设置为 `cover` 时，按像素值对图片进行不保留宽高比的缩放操作。

| 参数名称 | 类型 | 必填 | 说明 |
|---------------------|------------------------------|----|-------------------------------------|
| <code>width</code> | <code>string number</code> | 否 | 指定目标图片宽度缩放为 <code>width</code> 像素。 |
| <code>height</code> | <code>string number</code> | 否 | 指定目标图片高度缩放为 <code>height</code> 像素。 |

说明:

此模式下, `width` 参数和 `height` 参数需要同时设置, 如果仅配置其中之一, 图片将被处理为 `width x width` 或 `height x height`。

示例:

```
// 设置按像素值进行非等比缩放, 缩放后宽度为 100px, 高度为 100px, 不保留宽高比
eo: { image: { fit: 'cover', width: 100, height: 100 } }
```

`fit` 参数设置为 `percent` 时, 按百分比对图片进行不保留宽高比的缩放操作。

| 参数名称 | 类型 | 必填 | 说明 |
|---------------------|-----------------|----|---|
| <code>width</code> | string number | 否 | 指定目标图片宽度缩放为 <code>width%</code> , 取值范围 0 - 100; 仅设置 <code>width</code> 参数时, 高度保持不变。 |
| <code>height</code> | string number | 否 | 指定目标图片高度缩放为 <code>height%</code> , 取值范围 0 - 100; 仅设置 <code>height</code> 参数时, 宽度保持不变。 |

说明:

此模式下, `width` 参数和 `height` 参数各自独立, 如果同时配置 `width` 和 `height`, 图片将被处理为 `width% x height%`。

示例:

```
// 设置按宽度百分比进行非等比缩放, 缩放后宽度为原图的 50%
eo: { image: { fit: 'percent', width: 50 } }

// 设置按高度百分比进行非等比缩放, 缩放后高度为原图的 50%
eo: { image: { fit: 'percent', height: 50 } }

// 设置按宽度和高度百分比进行非等比缩放, 缩放后宽度为原图的 50%, 高度为原图的 50%
eo: { image: { fit: 'percent', width: 50, height: 50 } }
```

图片裁剪

图片裁剪操作受 `fit` 参数控制, 通过设定不同的 `fit` 参数, 可以实现不同类型的裁剪操作。

| 参数名称 | 类型 | 必 | 说明 |
|------|----|---|----|
|------|----|---|----|

| | | 填 | |
|-----|--------|---|---|
| fit | string | 否 | 设置不同的图片缩放与裁剪模式，支持的 fit 参数取值为： <ul style="list-style-type: none"> • cut: 对图片进行原始裁剪。 • crop: 对图片进行缩放裁剪。 |

说明:

不同 fit 参数对应的行为，将在下文进行详细说明。

原图裁剪

fit 参数设置为 cut 时，对图片进行原图裁剪。

| 参数名称 | 类型 | 必填 | 说明 |
|---------|---|----|--|
| width | string number | 否 | 指定目标图片宽度裁剪为 width 像素，仅设置 width 参数时，高度保持不变。 |
| height | string number | 否 | 指定目标图片高度裁剪为 height 像素；仅设置 height 参数时，宽度保持不变。 |
| gravity | string { [key: string]: number string } | 否 | 图片裁剪的锚点位置，可设置为九宫格方位值或坐标对象： <ul style="list-style-type: none"> • 九宫格方位值参考 九宫格方位图 进行取值。 • 坐标对象为 { x: 100, y: 100 } 的形式，其中 x 含义为相对于图片左上顶点水平向右偏移 x 像素；y 含义为相对于图片左上顶点水平向下偏移 y 像素。 |

说明:

若不设置 gravity 参数，则默认在左上顶点 northwest 进行裁剪。

示例:

```
// 设置按宽度进行原始裁剪，裁剪后宽度为 100px
eo: { image: { fit: 'cut', width: 100 } }

// 设置按高度进行原始裁剪，裁剪后高度为 100px
eo: { image: { fit: 'cut', height: 100 } }

// 设置按宽度和高度进行原始裁剪，裁剪后宽度为 100px，高度为 100px
eo: { image: { fit: 'cut', width: 100, height: 100 } }
```

```
// 设置按宽度和高度进行原始裁剪，指定锚点位置（九宫格），裁剪后宽度为 100px，高度为 100px
eo: { image: { fit: 'cut', width: 100, height: 100, gravity: 'center' } }

// 设置按宽度和高度进行原始裁剪，指定锚点位置（坐标），裁剪后宽度为 100px，高度为 100px
eo: { image: { fit: 'cut', width: 100, height: 100, gravity: { x: 100, y: 100 } } }
```

缩放裁剪

`fit` 参数设置为 `crop` 时，对图片进行缩放裁剪。

| 参数名称 | 类型 | 必填 | 说明 |
|---------|-----------------|----|--|
| width | string number | 否 | 指定目标图片宽度裁剪为 width 像素，仅设置 width 参数时，高度保持不变。 |
| height | string number | 否 | 指定目标图片高度裁剪为 height 像素；仅设置 height 参数时，宽度保持不变。 |
| gravity | string | 否 | 图片裁剪的锚点位置，可设置为九宫格方位值；九宫格方位值参考 九宫格方位图 进行取值。 |

说明：
此模式下，`gravity` 参数仅支持设置为九宫格方位值，若不设置 `gravity` 参数，则默认在中心点 `center` 进行裁剪。

示例：

```
// 设置按宽度进行缩放裁剪，裁剪后宽度为 100px
eo: { image: { fit: 'crop', width: 100 } }

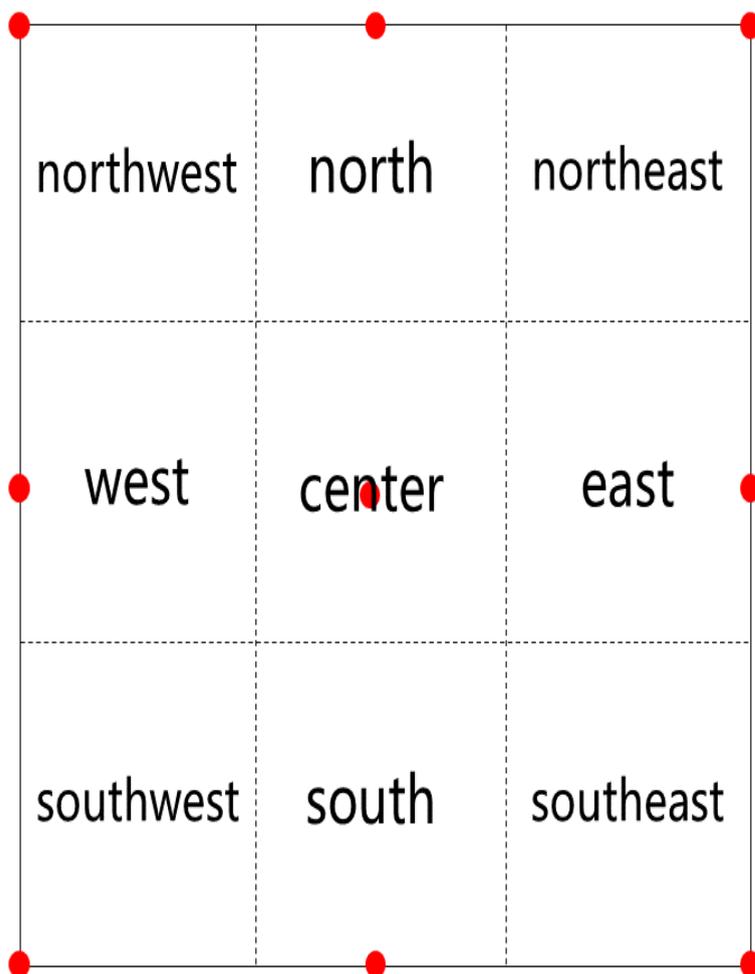
// 设置按高度进行缩放裁剪，裁剪后高度为 100px
eo: { image: { fit: 'crop', height: 100 } }

// 设置按宽度和高度进行缩放裁剪，裁剪后宽度为 100px，高度为 100px
eo: { image: { fit: 'crop', width: 100, height: 100 } }
```

```
// 设置按宽度和高度进行缩放裁剪，指定锚点位置（九宫格），裁剪后宽度为 100px，高度为 100px
eo: { image: { fit: 'crop', width: 100, height: 100,
gravity: 'northwest' } }
```

九宫格方位图

九宫格方位图可为图片的多种操作提供位置参考；红点为各区域位置的原点（通过 gravity 参数选定各区域后位移操作会以相应远点为参照）。



示例函数

示例概览

最近更新时间：2025-03-10 10:26:42

内容生成

| 标题 | 示例简介 |
|----------------------------|--------------------------|
| 返回 HTML 页面 | 通过校验请求头和值，如校验通过则响应，否则拒绝。 |
| 返回 JSON | 生成 JSON 并响应给客户端。 |

访问控制

| 标题 | 示例简介 |
|---------------------------------|--|
| 请求头鉴权 | 通过校验请求头和值，如校验通过则响应，否则拒绝。 |
| 防篡改校验 | 计算请求 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。 |
| m3u8 改写与鉴权 | .m3u8 与 .ts 片段资源的权限控制。 |
| 自定义 Referer 防盗链 | 根据请求头 Referer 实现自定义访问控制。 |
| 远程鉴权 | 将请求转发至指定的远程鉴权服务器，校验通过放行，否则拒绝。 |
| HMAC 数字签名 | 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。 |

重定向

| 标题 | 示例简介 |
|---------------------------|---------------------------------|
| 基于请求区域重定向 | 根据客户端所属区域，自动重定向到所属区域的目标网址。 |
| 301 重定向 | 客户端请求自动并永久地 301 重定向至预设的网站地址。 |
| 批量重定向 | 通过预定义的重定向映射表，实现指定路径自动跳转至对应 URL。 |

获取客户端信息

| 标题 | 示例简介 |
|--------------|---|
| 获取客户端 IP | 联合规则引擎，获取客户端 IP，并响应给客户端。 |
| 获取客户端 URL 信息 | 根据传入的 HTTP 请求，将请求 URL 的详细信息以 HTML 页面返回。 |

Geolocation

| 标题 | 示例简介 |
|--------------|---|
| 基于客户端地理特征定制化 | 基于客户端所处的国家提供定制化内容。 |
| 获取客户端地理特征信息 | 响应客户端的地理位置信息，包含 ASN、国家名称、国家代码、地区名称、地区代码、城市名称经度和纬度等。 |

图片处理

| 标题 | 示例简介 |
|------------|------------------------------|
| 图片自适应缩放 | 根据请求头 User-Agent 实现自适应缩放。 |
| 图片自适应 WebP | 根据请求头 Accept 实现格式自动转换成 webp。 |

头部处理

| 标题 | 示例简介 |
|----------|--|
| 修改响应头 | 自定义响应头，如添加、修改和删除。 |
| 自定义下载文件名 | 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。 |

Cache 缓存

| 标题 | 示例简介 |
|---------------|---|
| 通过 Cache 缓存资源 | Fetch 获取远程资源，通过 Cache API 实现资源缓存到 EdgeOne 边缘节点。 |
| 缓存 POST 请求 | 针对 POST 请求 body 计算 SHA-256 签名作为缓存 key，通过 Cache API 缓存到边缘节点。 |

流式响应

| 标题 | 示例简介 |
|--------------------------|----------------------------|
| 流式响应 | 获取远程资源并实现流式响应给客户端。 |
| 合并资源流式响应 | 远程获取多个视频资源，流式读取与拼接，再响应客户端。 |

Fetch

| 标题 | 示例简介 |
|------------------------|-------------------------|
| 获取远程资源 | 通过 Fetch 获取远程资源并响应给客户端。 |

Cookie

| 标题 | 示例简介 |
|---------------------------|-------------------------|
| 设置 Cookie | 通过 Fetch 获取远程资源并响应给客户端。 |

Testing

| 标题 | 示例简介 |
|-----------------------|---------------------------------|
| AB 测试 | 通过 Cookies 保存会话信息，对请求进行 A/B 测试。 |

301 重定向

最近更新时间：2024-06-14 17:33:21

该示例展示了如何使用 HTTP 301 状态码，将客户端请求自动并永久地重定向至预设的网站地址，常用于网站的永久迁移或合并。

示例代码

```
// 目标重定向地址
const destinationLocation = 'https://www.example.com';

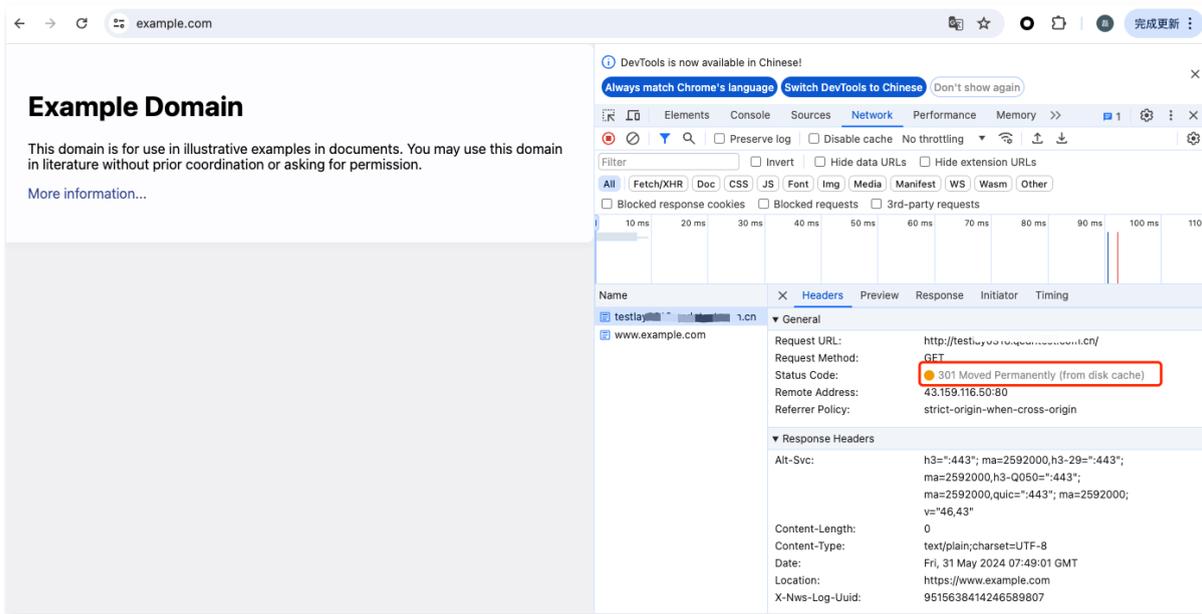
// HTTP 状态码，用于永久重定向
const statusCode = 301;

// 处理请求的异步函数
async function handleRequest(request) {
  // 返回一个重定向响应，使用指定的目标位置和状态码
  return Response.redirect(destinationLocation, statusCode);
}

// 添加 fetch 事件监听器，拦截请求并使用 handleRequest 函数进行响应
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](#)

获取客户端 URL 信息

最近更新时间：2024-06-14 17:33:21

该示例捕获传入的 HTTP 请求，并返回 HTML 页面，该页面显示了请求 URL 的详细信息，可用于调试和展示请求参数、路径及来源等。

示例代码

```
// 添加 fetch 事件监听器，当有请求进入时触发，使用 handleRequest 函数处理请求，并返回响应
addEventListener('fetch', event => {
  event.respondWith(handleRequest(event.request))
})

function handleRequest(request) {
  // 解析请求 URL
  const url = new URL(request.url)

  // 提取 URL 各个组成部分
  const {
    href, // 完整 URL
    protocol, // 协议 (如 http:)
    hostname, // 主机名 (如 example.com)
    port, // 端口 (如果有指定)
    pathname, // 路径 (如 /path)
    search, // 查询字符串 (如 ?query=123)
    hash // 井号后面的片段 (如 #section)
  } = url

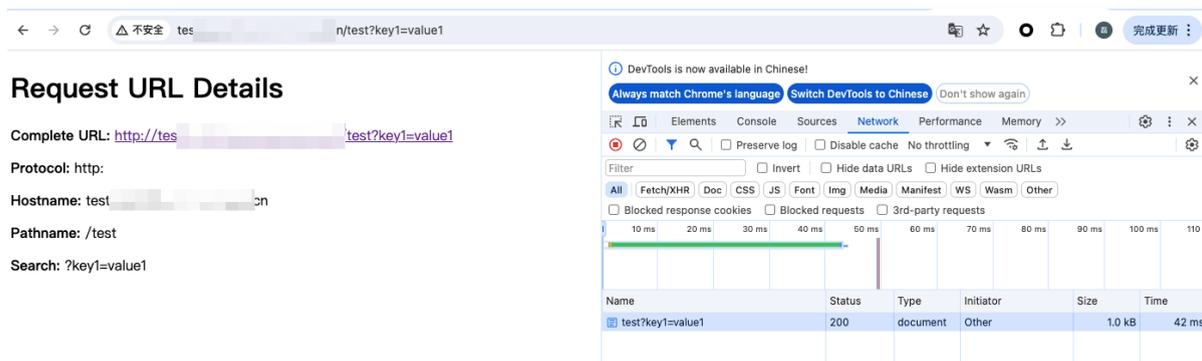
  // 构造 HTML 响应内容
  const htmlContent = `
    <!DOCTYPE html>
    <html lang="en">
    <head>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
      <title>Request URL Details</title>
    </head>
    <body>
      <h1>Request URL Details</h1>
  `
```

```
<p><strong>Complete URL:</strong> <a href="\${href}">\${href}</a></p>
<p><strong>Protocol:</strong> \${protocol}</p>
<p><strong>Hostname:</strong> \${hostname}</p>
\${port ? `<p><strong>Port:</strong> \${port}</p>` : ''}
<p><strong>Pathname:</strong> \${pathname}</p>
<p><strong>Search:</strong> \${search}</p>
\${hash ? `<p><strong>Hash:</strong> \${hash}</p>` : ''}
</body>
</html>
`;

// 返回 HTML 格式的响应
return new Response(htmlContent, {
  headers: {
    'Content-Type': 'text/html;charset=UTF-8'
  }
});
}
```

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](#)

基于客户端地理特征定制化

最近更新时间：2024-06-14 17:33:21

该示例捕获传入的 HTTP 请求，并基于客户端所处的国家提供定制化的欢迎信息及地理位置（纬度和经度），可用于全球化的个性化用户定制体验。

示例代码

```
// 添加fetch事件监听器，当有请求进入时触发，使用handleRequest函数处理请求，并返回响应
addEventListener('fetch', event => {
  event.respondWith(handleRequest(event.request))
})

function handleRequest(request) {
  // 获取 EdgeOne 提供的国家代码
  const countryCode = request.eo.geo.countryCodeAlpha2;
  // 根据国家代码选择对应的语言和欢迎信息
  let responseText;
  switch (countryCode) {
    case 'CN': // 中国
      responseText = `中国的用户您好！您的纬度和经度分别为
${request.eo.geo.latitude},${request.eo.geo.longitude}`;
      break;
    case 'KR': // 韩国
      responseText = `한국어 사용자 환영! ${request.eo.geo.latitude} ${request.eo.geo.longitude}`;
      break;
    case 'DE': // 德国
      responseText = `Willkommen in Deutschland! Ihre Breiten- und
Längengrad sind ${request.eo.geo.latitude} und
${request.eo.geo.longitude}`;
      break;
    case 'US': // 美国
      responseText = `Hello from the USA! Your latitude and
longitude are ${request.eo.geo.latitude} and
${request.eo.geo.longitude}`;
      break;
    default: // 其他情况默认响应英文
      responseText = `Welcome to our service! Your latitude and
longitude are ${request.eo.geo.latitude} and
${request.eo.geo.longitude}`;
      break;
  }
}
```

```
}  
// 返回响应  
return new Response(responseText, {  
  headers: {  
    'Content-Type': 'text/plain;charset=UTF-8'  
  }  
})  
}
```

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，如当前请求 URL 的客户端位于中国，则浏览器会响应为中国的欢迎页面和客户端所处地理位置的纬度和经度。

中国的用户您好! 您的纬度和经度分别为114.05786895751953, 22.54309844970703

| Name | Status | Type | Initiator | Size | Time |
|-------------|--------|------------|-----------|-------|--------|
| testlay.com | 200 | document | Other | 363 B | 468 ms |
| favicon.ico | 200 | text/plain | Other | 362 B | 173 ms |

相关参考

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

获取客户端地理特征信息

最近更新时间：2024-07-15 16:08:11

该示例捕获 HTTP 请求并响应用户地理位置信息，生成包含 ASN、国家、地区、城市及经纬度的 HTML 页面，可用于调试或信息地理位置信息展示。

示例代码

```
// 添加fetch事件监听器，当有请求进入时触发，使用handleRequest函数处理请求，并返回响应
addEventListener('fetch', event => {
  event.respondWith(handleRequest(event.request));
});

// 定义一个异步函数handleRequest，用于处理传入的请求
async function handleRequest(request) {
  // 初始化一个空字符串用于存放HTML内容
  let html_content = "";
  // 定义HTML的样式
  let html_style = "body{padding:6em; font-family: sans-serif;}
h1{color:#0000ff;}";

  // 获取用户的地理位置信息
  html_content += "<p> asn: " + request.eo.geo.asn + "</p>"; // 自治域系统编号
  html_content += "<p> countryName: " + request.eo.geo.countryName + "</p>"; // 国家名称
  html_content += "<p> countryCodeAlpha2: " + request.eo.geo.countryCodeAlpha2 + "</p>"; // 国家两字母代码
  html_content += "<p> countryCodeAlpha3: " + request.eo.geo.countryCodeAlpha3 + "</p>"; // 国家三字母代码
  html_content += "<p> countryCodeNumeric: " + request.eo.geo.countryCodeNumeric + "</p>"; // 国家数字代码
  html_content += "<p> regionName: " + request.eo.geo.regionName + "</p>"; // 地区名称
  html_content += "<p> regionCode: " + request.eo.geo.regionCode + "</p>"; // 地区代码
  html_content += "<p> cityName: " + request.eo.geo.cityName + "</p>"; // 城市名称
  html_content += "<p> Latitude: " + request.eo.geo.latitude + "</p>"; // 纬度
}
```

```
html_content += "<p> Longitude: " + request.eo.geo.longitude + "  
</p>"; // 经度  
  
// 构建HTML响应内容  
let html = `<!DOCTYPE html>  
  <head>  
    <title> Geolocation: Hello World By Edge Functions.</title>  
    <style> ${html_style} </style>  
  </head>  
  <body>  
    <h1>Geolocation: Hello World By Edge Functions.</h1>  
    <p> Welcome to try out the geolocation feature of Edge  
Functions.</p>  
    ${html_content}  
  </body>`;   
  
// 返回一个新的Response对象, 包含HTML内容和相应的headers  
return new Response(html, {  
  headers: {  
    "content-type": "text/html;charset=UTF-8", // 设置响应的Content-  
Type头部, 指定返回内容为HTML  
  },  
});  
}
```

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。

Geolocation: Hello World By Edge Functions.

Welcome to try out the geolocation feature of Edge Functions.

asn: 17623

countryName: China

countryCodeAlpha2: CN

countryCodeAlpha3: CHN

countryCodeNumeric: 156

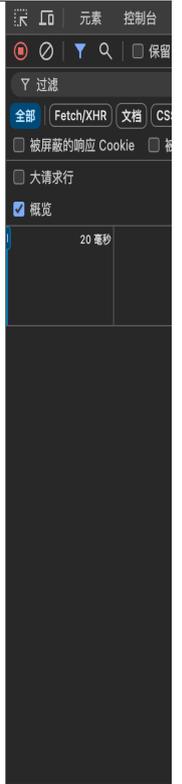
regionName: Guangdong Sheng

regionCode: CN-GD

cityName: Unknown

Latitude: 114.05786895751953

Longitude: 22.54309844970703



相关参考

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

批量重定向

最近更新时间：2024-06-14 17:33:21

该示例捕获传入的 HTTP 请求，并通过预定义的重定向映射表，实现了指定路径自动跳转至对应 URL，可用于网站迁移或错误页面的自定义处理。

示例代码

```
// 添加fetch事件监听器，当有请求进入时触发，使用handleRequest函数处理请求，并返回响应
async function handleRequest(request) {
  // 定义目标外部主机名
  const yourExternalHostname = "www.example.com";
  // 创建路径到重定向URL的映射
  const redirectMap = new Map([
    ["/foo", "https://" + yourExternalHostname + "/redirect1"],
    ["/bar", "https://" + yourExternalHostname + "/redirect2"],
    ["/baz", "https://" + yourExternalHostname + "/redirect3"],
  ]);
  // 解析请求的URL
  const url = new URL(request.url);
  // 获取URL的路径部分
  const path = url.pathname;
  // 检查路径是否在重定向映射中，如果是则进行重定向
  if (redirectMap.has(path)) {
    return Response.redirect(redirectMap.get(path), 301);
  } else {
    // 如果路径不在映射中，返回404状态
    return new Response('Not Found', { status: 404 });
  }
}

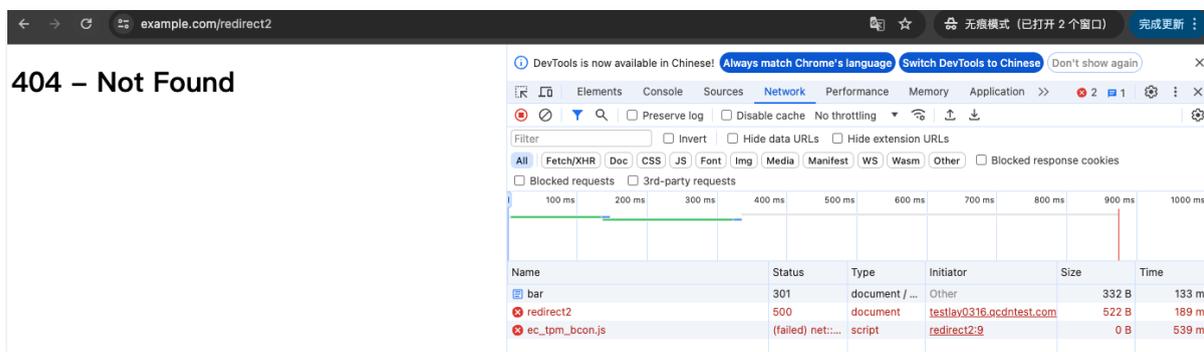
// 当有请求事件发生时，使用handleRequest函数处理
addEventListener('fetch', event => {
  event.respondWith(handleRequest(event.request));
});
```

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，在路径中携带/bar将实现到自动 301 重定向至

`https://www.example.com/redirect2`

预览如下，当前显示 404 因目标主机名 `www.example.com` 路径 `/redirect2` 下无资源，您需按实际替换目标主机名和路径。



相关参考

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

基于正则的 URL 改写

最近更新时间：2024-12-23 15:32:31

该示例捕获传入的 HTTP 请求，根据 URL 路径使用正则表达式进行匹配，并在匹配到特定内容时实现 URL 改写访问 index.html 页面。适用于将对指定目录下的任意请求统一重写为访问主页（通常是index.html），以便通过前端路由来处理不同的页面请求。

示例代码

```
async function handleEvent(event) {
  const { request } = event;
  const urlInfo = new URL(request.url);
  // 正则表达式匹配/test/后面跟着1或2个非斜杠的路径段
  const regexp = /^\/test\/((^[^\/]+)?(?:\/([^[^\/]+)))?\/?$/;
  // 检查路径是否匹配正则表达式
  if (regexp.test(urlInfo.pathname)) {
    const matches = urlInfo.pathname.match(regexp);
    let newPathname = '/test/';
    // 构造新的路径名，根据匹配的路径段数量，可能是一个或两个
    newPathname += matches[1]; // 第一个路径段
    if (matches[2]) {
      newPathname += '/' + matches[2]; // 第二个路径段，如果有的话
    }
    // 确保以index.html结尾
    newPathname += '/index.html';
    // 更新URLInfo的pathname
    urlInfo.pathname = newPathname;
  }

  // 使用更新后的URLInfo发起请求
  const response = await fetch(urlInfo.toString(), {
    method: request.method,
    headers: request.headers,
    redirect: 'manual',
    body: request.body,
  });
  // 将响应返回给事件
  return event.respondWith(response);
}

// 为每个fetch事件调用handleEvent函数
addEventListener('fetch', (event) => {
```

```
handleEvent(event);  
});
```

示例预览

1. 在浏览器地址栏中输入匹配到边缘函数触发规则的 URL，在路径中携带 /test/segment1 实现动态改写访问

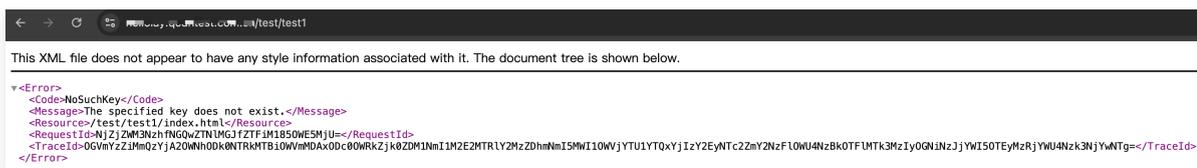
`https://www.example.com/test/segment1/index.html`



2. 在浏览器地址栏中输入匹配到边缘函数触发规则的 URL，在路径中携带 /test/segment1/segment2 实现动态改写访问 `https://www.example.com/test/segment1/segment2/index.html`



3. 在浏览器地址栏中输入匹配到边缘函数触发规则的 URL，在路径中携带非正则匹配内容，如: /test/test1



相关参考

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

返回 HTML 页面

最近更新时间：2023-01-09 10:19:14

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：2023-01-09 10:21:12

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',
};

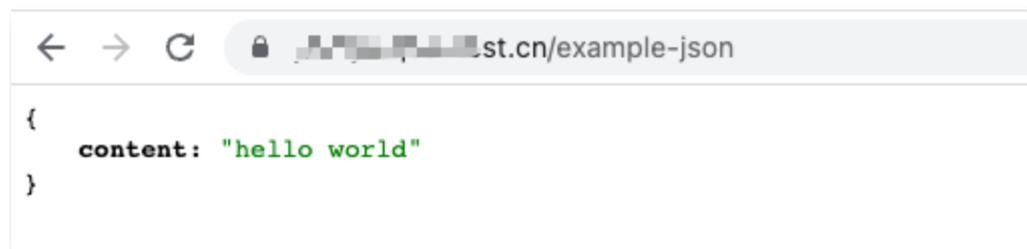
async function handleRequest(request) {
  // JSON 转为字符串
  const result = JSON.stringify(data, null, 2);

  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

最近更新時間：2023-01-09 10:37:47

該示例使用 [Fetch API](#) 獲取遠程資源 [jQuery.js](#) 並響應給客戶端。

示例代碼

```
async function handleRequest(request) {
  // 獲取遠程資源
  const response = await
  fetch('https://static.cloudcachetci.com/qcloud/main/scripts/release/comm
on/vendors/jquery-3.2.1.min.js');
  return response;
}

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

示例預覽

在瀏覽器地址欄中輸入匹配到邊緣函數觸發規則的 URL，即可預覽到示例效果。

The screenshot shows a browser window with the address bar containing a URL ending in /fetch-resources. The console displays the following information:

- Request URL: https://static.cloudcachetci.com/qcloud/main/scripts/release/common/vendors/jquery-3.2.1.min.js
- Request Method: GET
- Status Code: 200
- Remote Address: static.cloudcachetci.com
- Referrer Policy: strict-origin-when-cross-origin

The response headers are also visible:

- accept-ranges: bytes
- access-control-allow-credentials: true
- access-control-allow-methods: PUT, GET, POST, HEAD
- access-control-allow-origin: https://www.tencentcloud.com
- access-control-expose-headers: Content-Length, Content-Type,
- content-length: 86659
- content-type: application/x-javascript
- date: Mon, 21 Nov 2022 09:51:08 GMT
- eo-cache-status: HIT
- eo-log-uuid: 16066500319245279906

相關參考

- Runtime APIs: Fetch

请求头鉴权

最近更新时间：2023-01-06 16:37:18

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');
  }

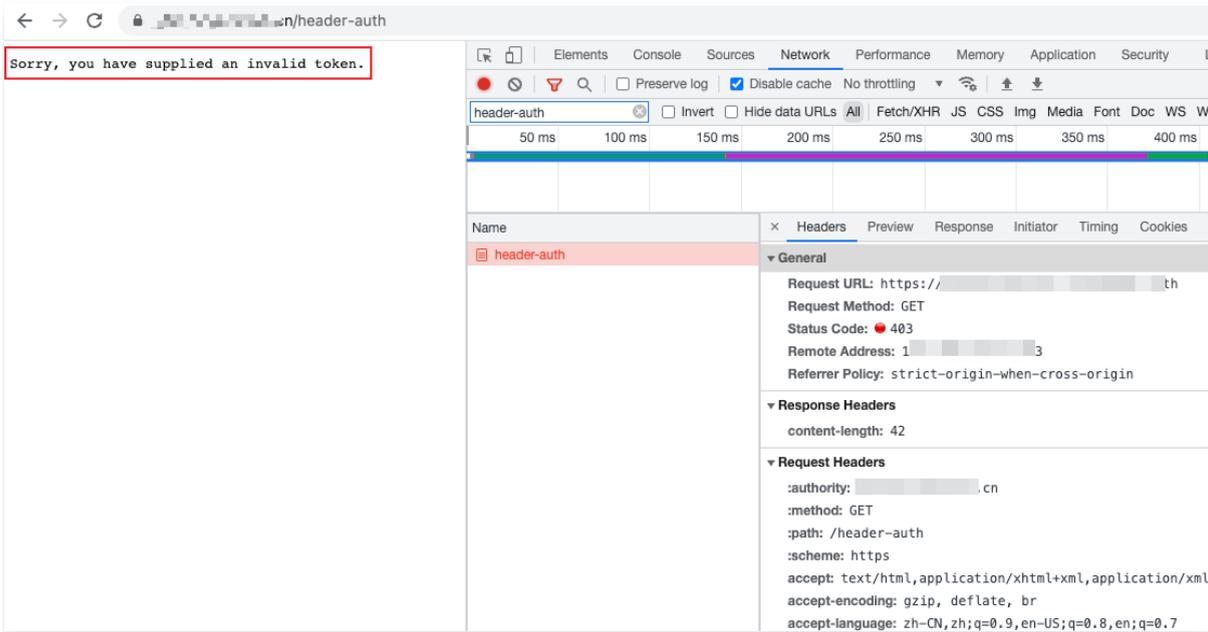
  // Incorrect key supplied. Reject the request.
  return new Response('Sorry, you have supplied an invalid token.', {
    status: 403,
  });
}

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

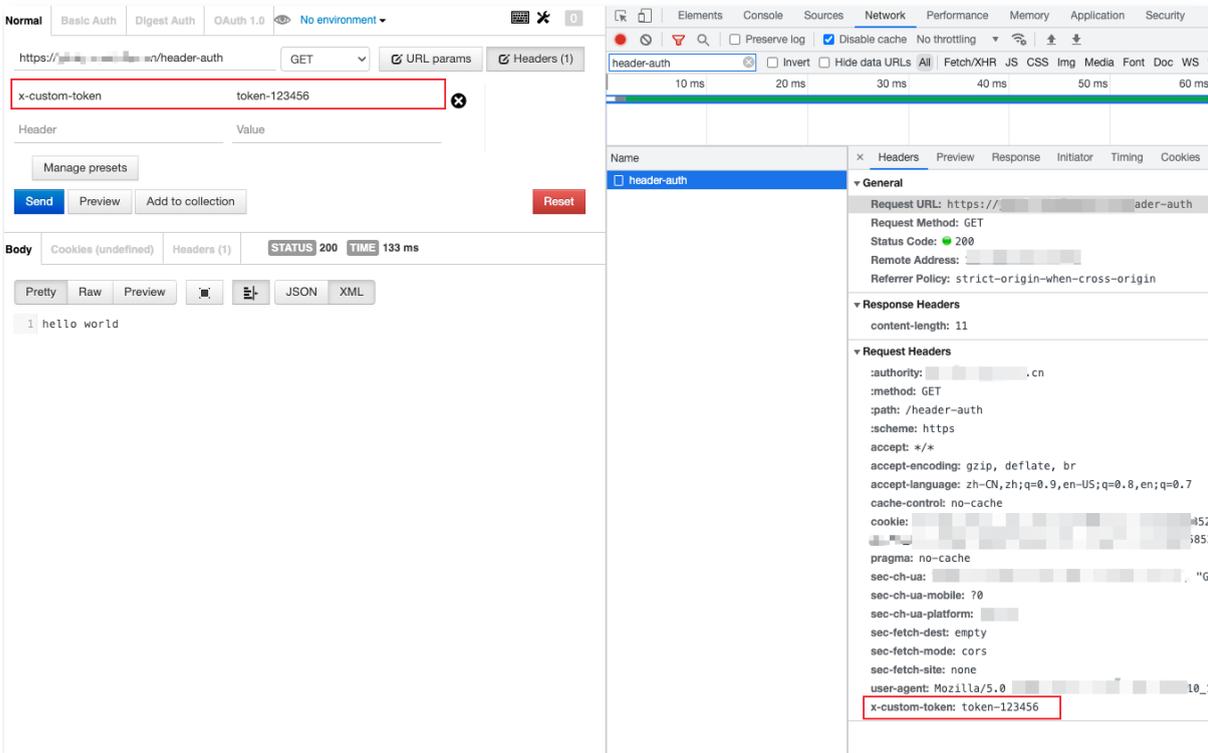
示例预览

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。



● 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：2024-08-23 15:07:21

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${urlInfo.pathname}${urlInfo.search}`, {
  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');

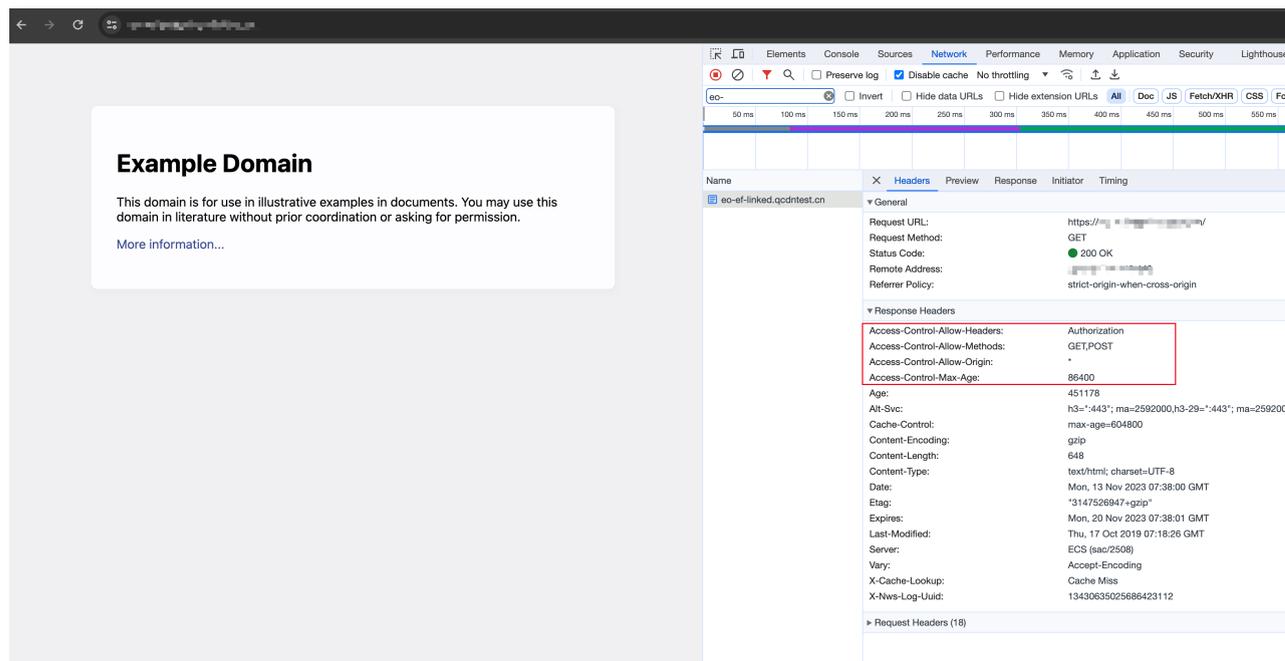
return response;
}

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：2024-04-18 09:47:31

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',
};

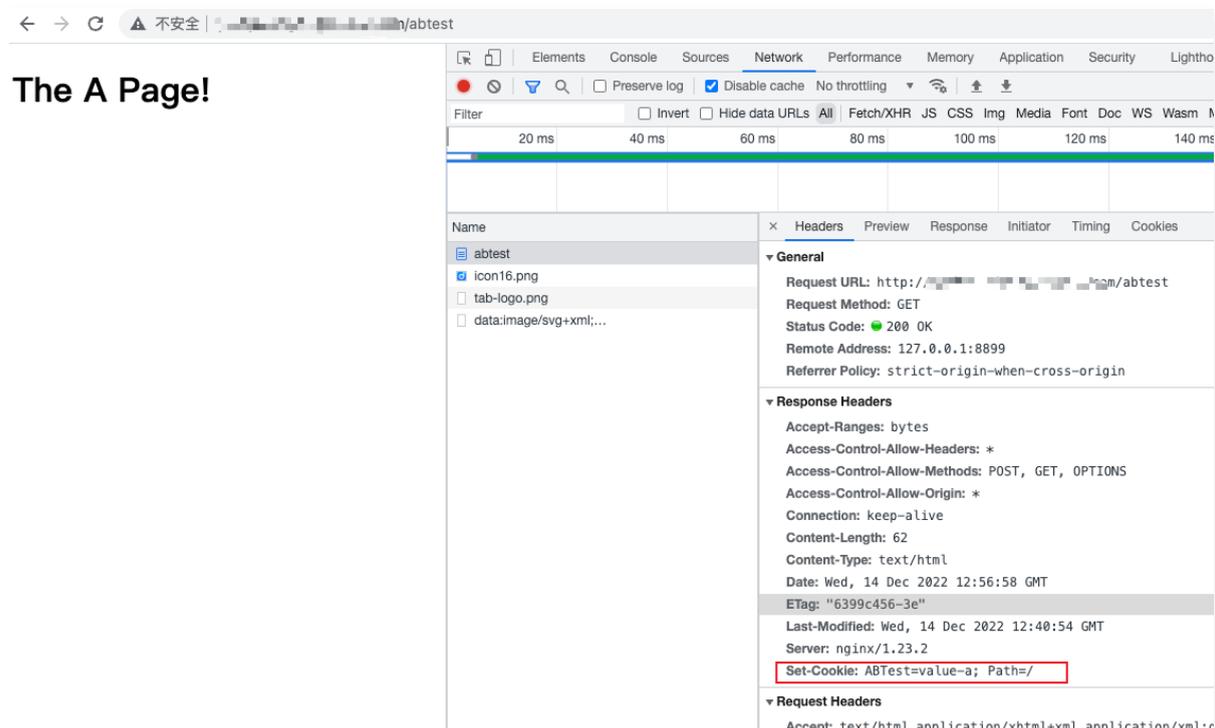
Object.keys(key2name).forEach(key => {
  if (cookie[key]) {
    cookieArr.push(`${key2name[key]}=${cookie[key]}`);
  }
});

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。



The screenshot shows a browser window with the address bar containing a URL ending in /abtest. The developer tools network tab is open, displaying a list of resources. The selected resource is 'abtest', which is a GET request to 'http://[redacted]/abtest'. The response status is 200 OK. The response headers include 'Set-Cookie: ABTest=value-a; Path=/' which is highlighted with a red box. Other headers include 'Accept-Ranges: bytes', 'Access-Control-Allow-Headers: *', 'Access-Control-Allow-Methods: POST, GET, OPTIONS', 'Access-Control-Allow-Origin: *', 'Connection: keep-alive', 'Content-Length: 62', 'Content-Type: text/html', 'Date: Wed, 14 Dec 2022 12:56:58 GMT', 'ETag: "6399c456-3e"', 'Last-Modified: Wed, 14 Dec 2022 12:40:54 GMT', and 'Server: nginx/1.23.2'. The request headers include 'Accept: text/html,application/xhtml+xml,application/xml;...'.

相关参考

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

设置 Cookie

最近更新时间：2024-04-18 09:47:31

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。

The screenshot shows a browser window with the address bar containing a URL ending in /set-cookie. The page content displays "The count is: 9". The developer tools are open to the Network tab, showing a "set-cookie" request. The request details are as follows:

| Name | Headers | Preview | Response | Initiator | Timing | Cookies |
|------------|--|---------|----------|-----------|--------|---------|
| set-cookie | <p>General</p> <p>Request URL: ...ons.com/set-cookie</p> <p>Request Method: GET</p> <p>Status Code: 200 OK</p> <p>Remote Address: ...</p> <p>Referrer Policy: strict-origin-when-cross-origin</p> <p>Response Headers</p> <p>Connection: keep-alive</p> <p>Content-Length: 15</p> <p>Date: Thu, 15 Dec 2022 16:19:30 GMT</p> <p>Keep-Alive: timeout=5</p> <p>Set-Cookie: count=9</p> <p>Request Headers</p> <p>Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/...</p> <p>Accept-Encoding: gzip, deflate</p> <p>Accept-Language: zh-CN,zh;q=0.9,en-US;q=0.8,en;q=0.7</p> <p>Cache-Control: no-cache</p> <p>Cookie: count=8</p> <p>Host: ...</p> | | | | | |

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：2024-08-23 15:07:21

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';

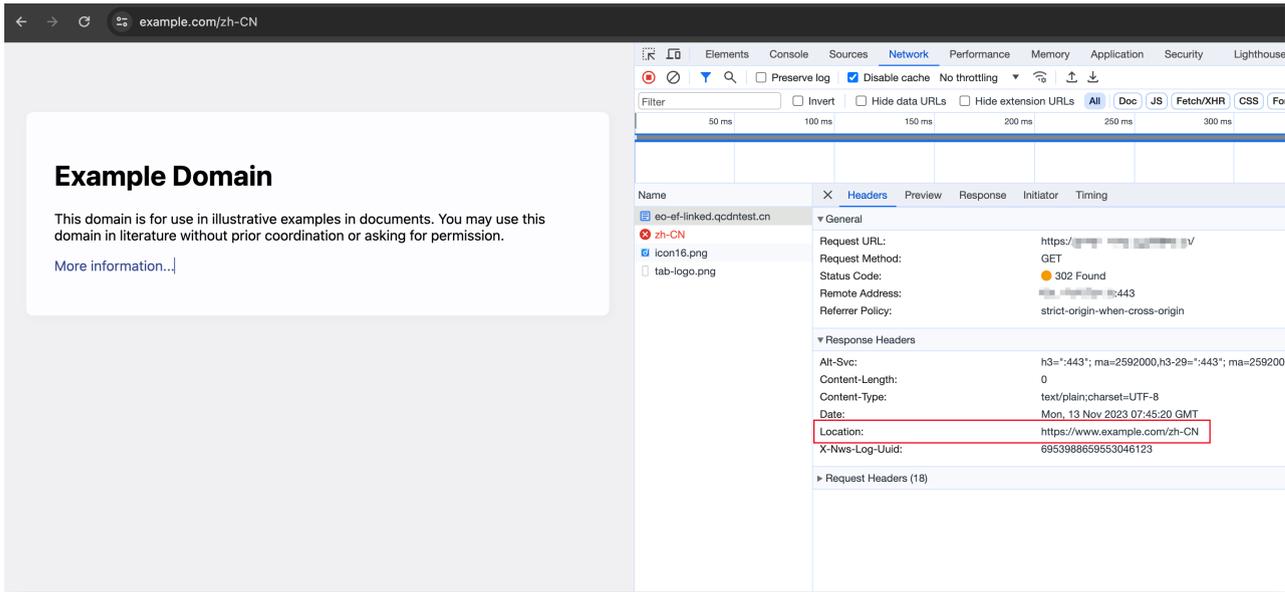
/**
 * 根据当前请求所在的区域，重定向到目标网址
 * @param { Request } request
 */
function handleRequest(request) {
  // 获取当前请求所在区域
  const alpha2code = request.eo.geo.countryCodeAlpha2;
  // 重定向目标网址
  const url = urls[alpha2code] || defaultUrl;

  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（例如：<https://www.example.com/en-US>），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：2024-08-23 15:07:21

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/release/co
  mmon/vendors/jquery-3.2.1.min.js');
  // 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);
}
}

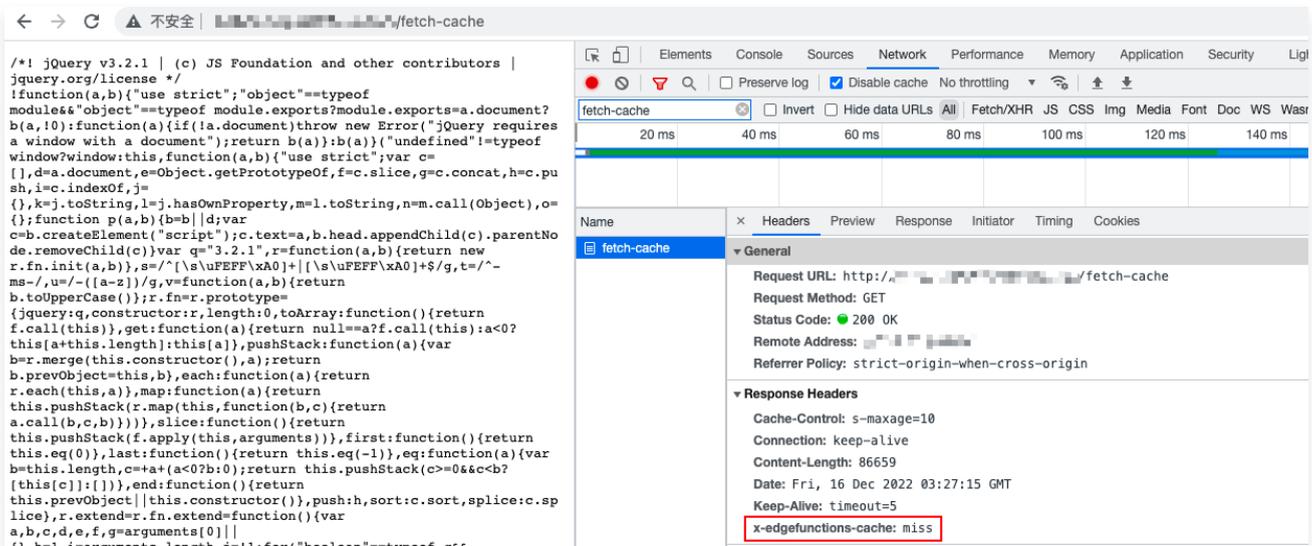
addEventListener('fetch', (event) => {
event.respondWith(handleEvent(event));
});

```

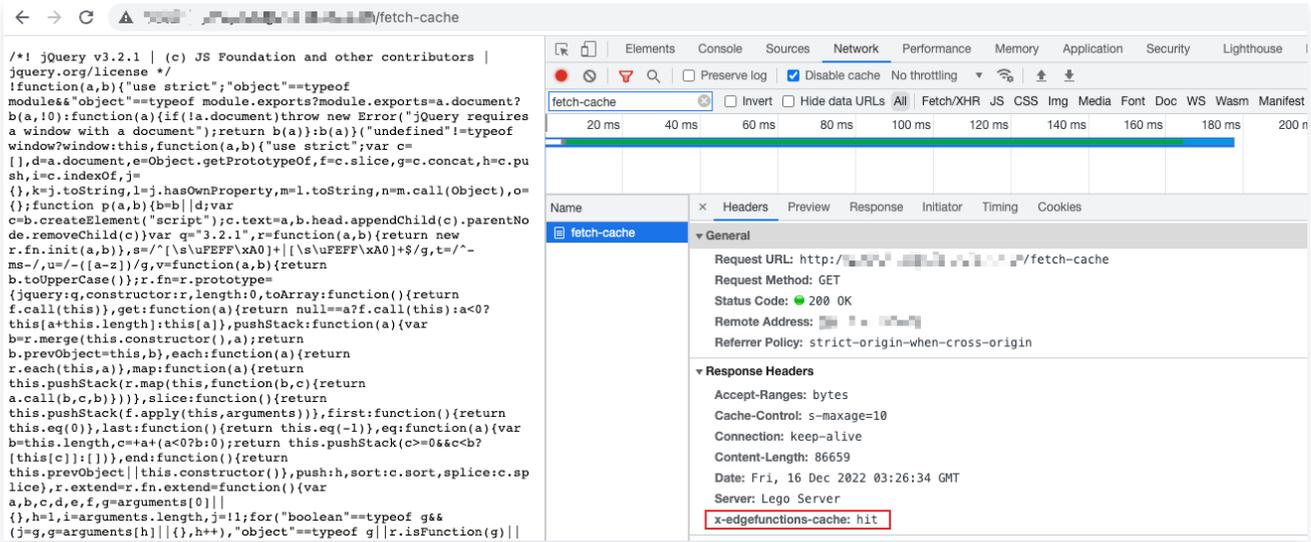
示例预览

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：2024-08-23 15:07:21

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);
  // 缓存过期或不存在, 重新获取远程资源
  return fetchContent(event, cacheKey);
}

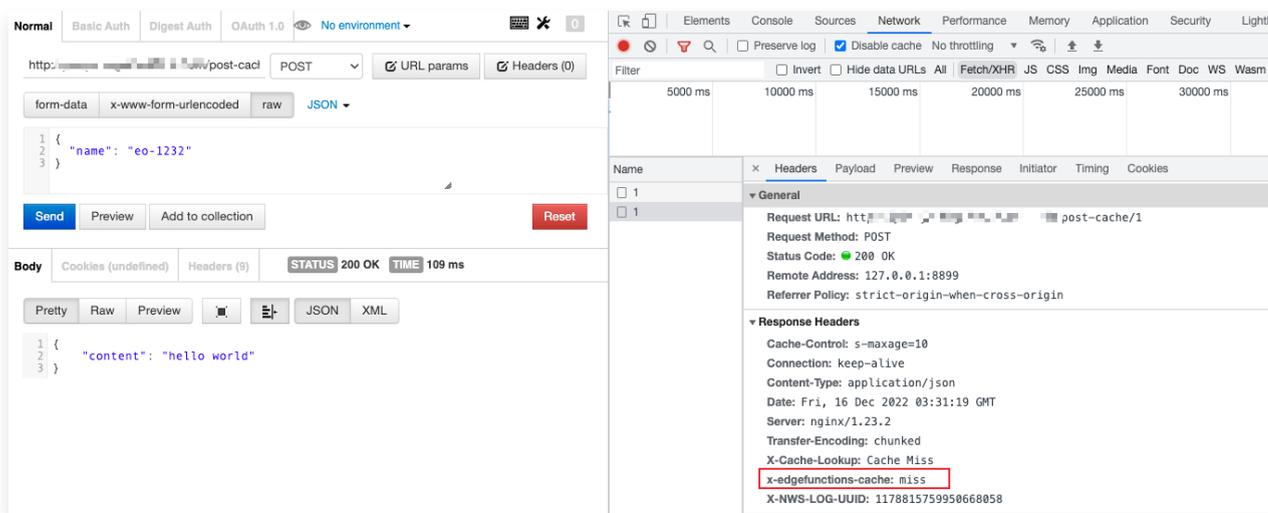
return response;
}

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));
  }
});
```

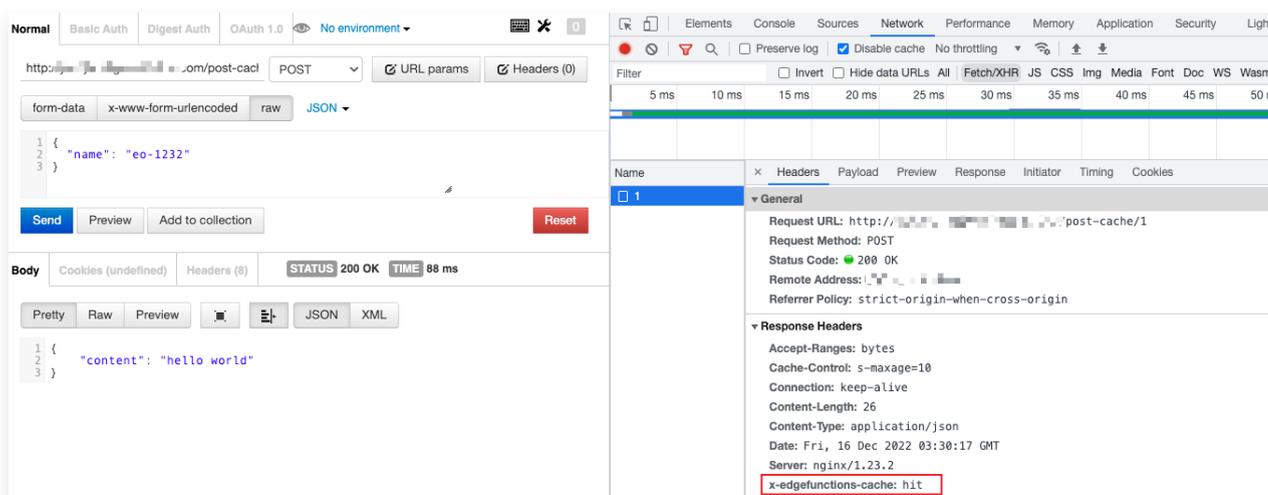
示例预览

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。



- 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：2024-08-23 15:07:21

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.cloudcacheci.com/qcloud/main/scripts/release/comm
  on/vendors/jquery-3.2.1.min.js');

  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: ...
> Host: ...
> Accept: */*
> Proxy-Connection: Keep-Alive
>
< HTTP/1.1 200 OK
< E0-Cache-Status: HIT
< E0-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: NjM3MjNkOWVfYTk3NWl3MDlfMjlnN18xZTc2YzI3
< 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：2025-06-10 11:33:52

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 sequentialCombine(urls, destination) {
  try {
    // 按顺序处理每个URL
    for (let i = 0; i < urls.length; i++) {
      const url = urls[i];

      // 获取当前片段
      const response = await fetch(url);

      if (!response.ok) {
        console.error(`获取视频片段失败: ${url}, 状态码: ${response.status}`);
        continue;
      }

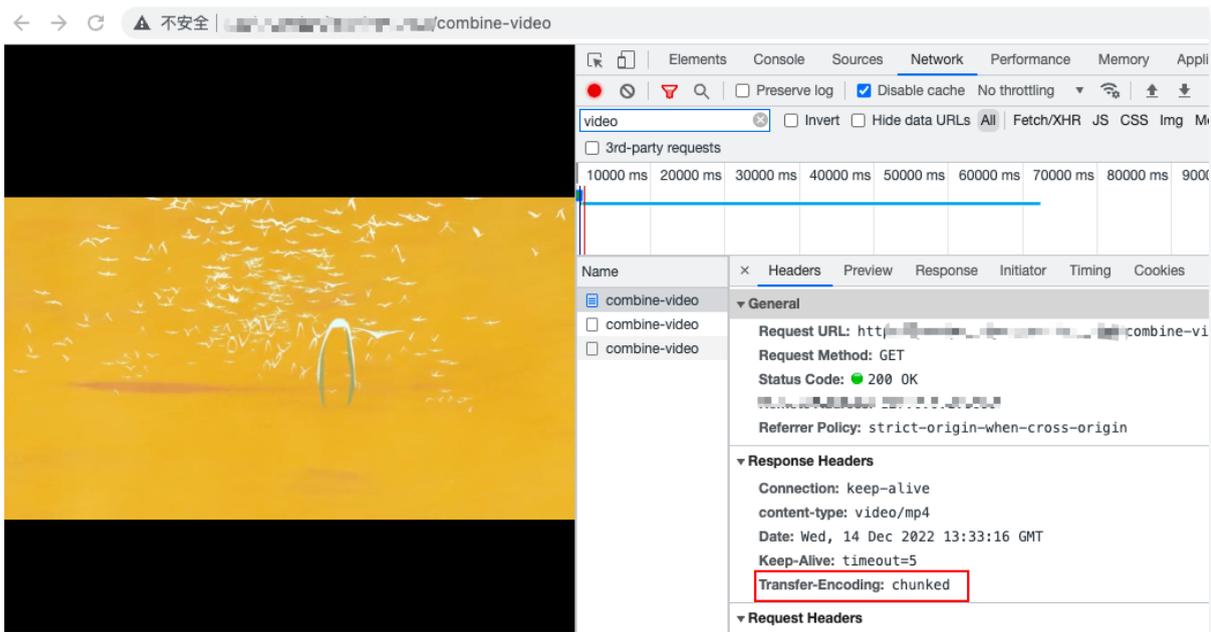
      // 获取可读流
      const readable = response.body;

      // 立即执行pipeTo, 将当前片段写入目标流
      try {
        await readable.pipeTo(destination, {
          preventClose: true // 保持流开放以便后续写入
        });
      } catch (e) {
        console.error(`流处理错误 (${url}): ${e.message}`);
      }
    }
  } catch (err) {
    console.error(`合并视频流错误: ${err.message}`);
  } finally {
    // 完成所有片段处理后关闭流
    const writer = destination.getWriter();
    writer.close();
    writer.releaseLock();
  }
}
```

```
}  
}  
  
async function handleRequest(request) {  
  // 视频片段地址  
  const urls = [  
    'https://vod.example.com/stream-01.mov',  
    'https://vod.example.com/stream-02.mov',  
    'https://vod.example.com/stream-03.mov',  
  ];  
  
  // 创建转换流  
  const { readable, writable } = new TransformStream();  
  
  // 按顺序获取并合并视频片段  
  sequentialCombine(urls, writable);  
  
  // 返回合并后的视频流响应  
  return new Response(readable, {  
    headers: {  
      'content-type': 'video/mp4',  
    },  
  });  
}  
  
// 监听fetch事件  
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：2024-05-06 11:16:21

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，避免外部攻击者破解。
 * 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;
}

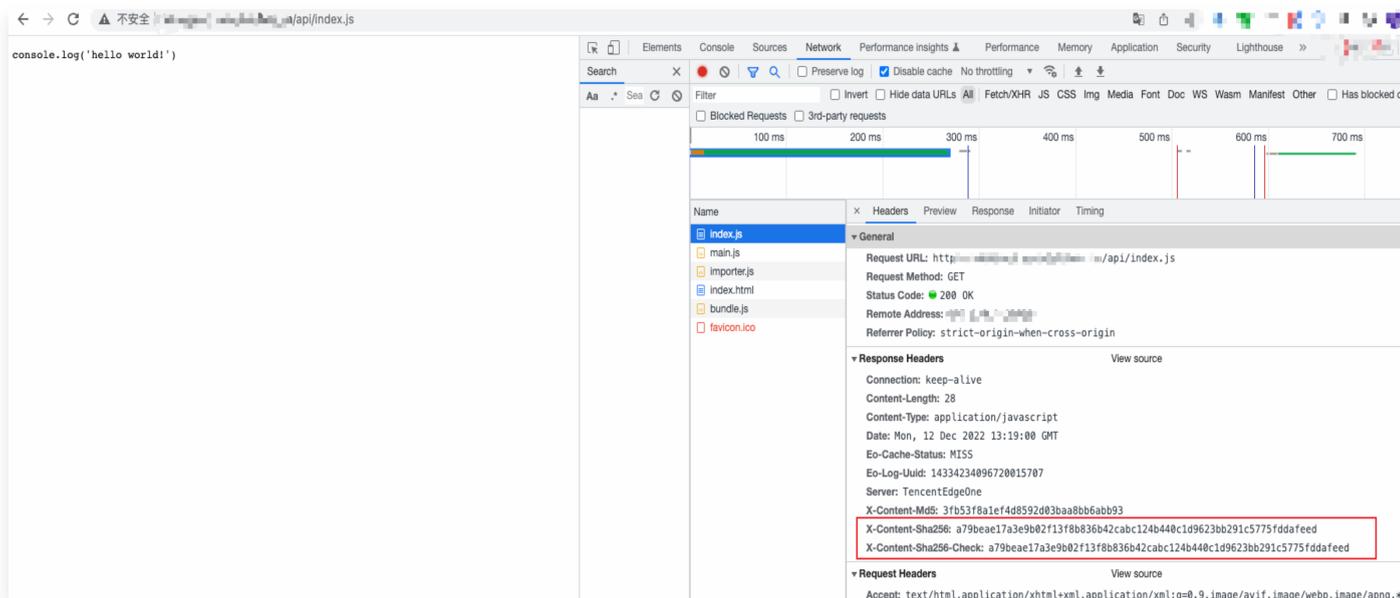
addEventListener('fetch', event => {
```

```
event.respondWith(handleEvent(event));  
});
```

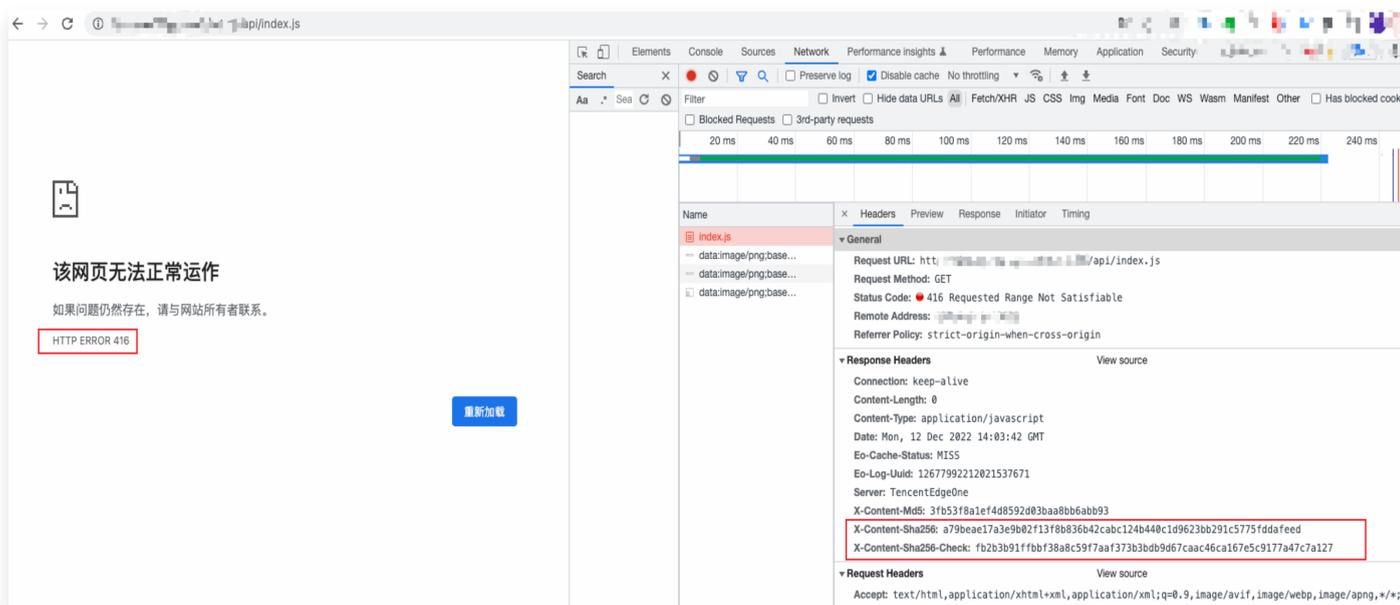
示例预览

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。



- 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：2024-05-27 17:41:21

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 = 14400;
const KEY_NAME = 'key';
const UID = 0;
const SUFFIX_LIST = ['.m3u8', '.ts'];

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

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

    if (!SUFFIX_LIST.includes(suffix)) {
      return;
    }

    const checkResult = await checkTypeA(urlInfo);
    if (!checkResult.flag) {
      const response = new Response(null, {
        status: 403,
        headers: {
          'X-Auth-Err': checkResult.message,
        },
      });
      return event.respondWith(response);
    }

    if (suffix === '.m3u8') {
```

```
const response = await fetchM3u8({
  request,
  querySign: {
    basePath: urlInfo.pathname.substring(0,
urlInfo.pathname.lastIndexOf('/')),
    ...checkResult.querySign,
  },
});
return event.respondWith(response);
}

if (suffix === '.ts') {
  return;
}
} catch (error) {
  return new Response(null, { status: 590 });
}

return;
}

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 === undefined) {
    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 }) {
  let response = null;
  request.headers.delete('Range');
  try {
    response = await fetch(request);
    if (response.status !== 200) {
      return response;
    }
  } catch (error) {
    console.log(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 rewriteLine({ line, querySign }) {
  if (/^\s*$/.test(line)) {
    return line;
  }

  if (line.charAt(0) === '#') {
    // Process #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(/\.([\^\.]+)$/);
  return suffix ? `.${suffix[1]}` : null;
}

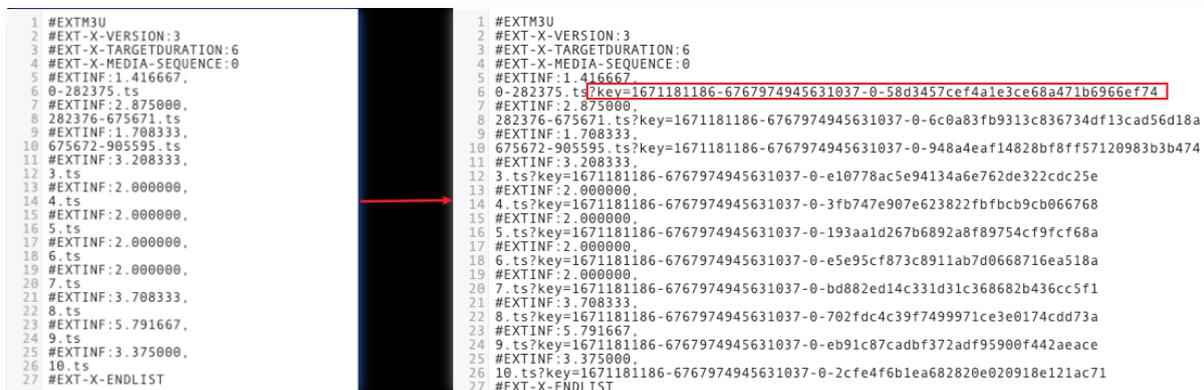
function isNumber(num) {
  return Number.isInteger(Number(num));
}

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。



```
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

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=1671181186-6767974945631037-0-193aa1d267b6892a8f89754cf9fcf68a
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-eb91c87c8dbf372adf95900f442aeace
25 #EXTINF:3.375000,
26 10.ts?key=1671181186-6767974945631037-0-2cfe4f6b1ea682820e020918e121ac71
27 #EXT-X-ENDLIST
```

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：2024-08-23 15:07:21

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合适的尺寸呈现。

```
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|IEMobile|
    MQQBrowser|JUC|Fennec|wOSBrowser|BrowserNG|WebOS|Symbian|Windows
    Phone)/i;

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

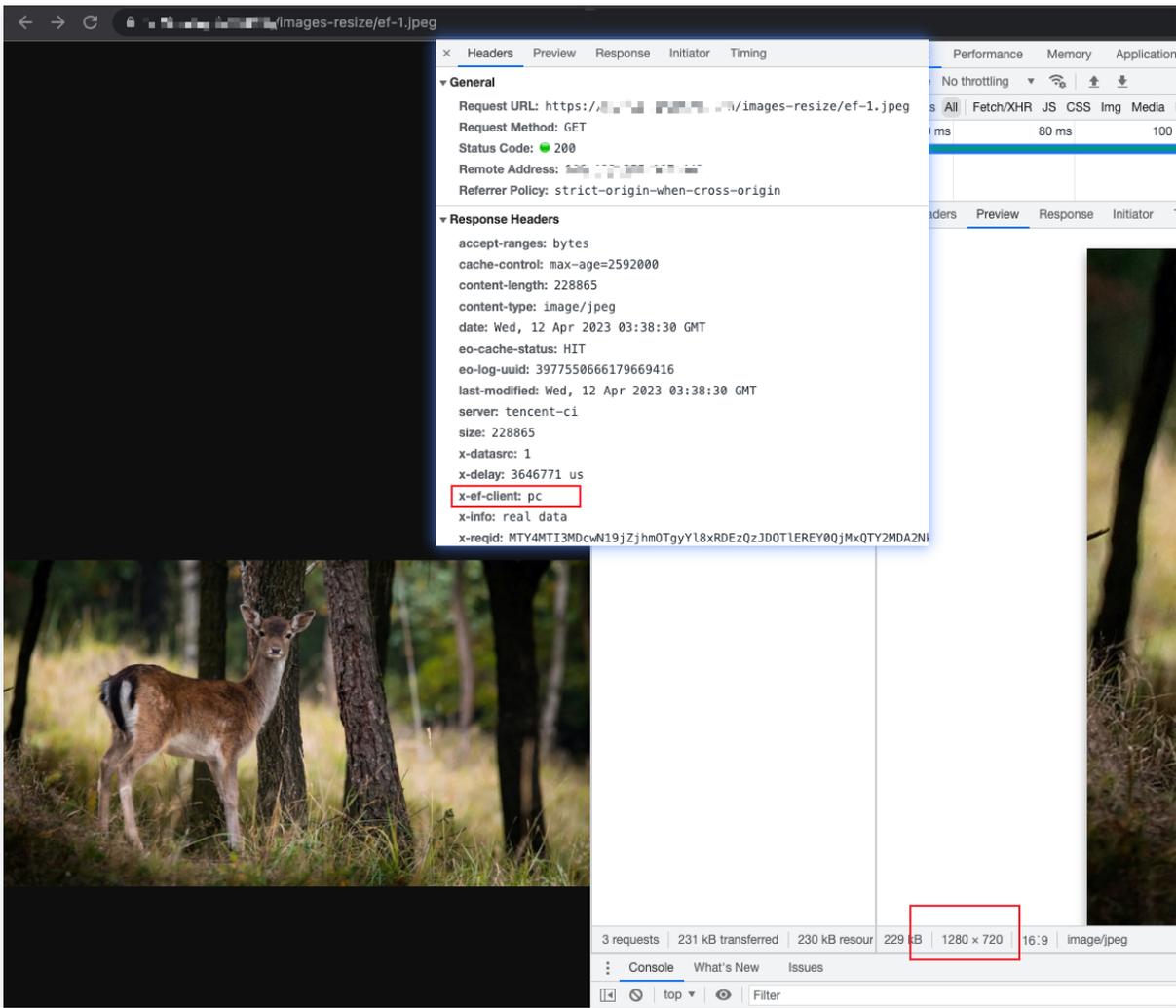
    return true;
}
```

示例预览

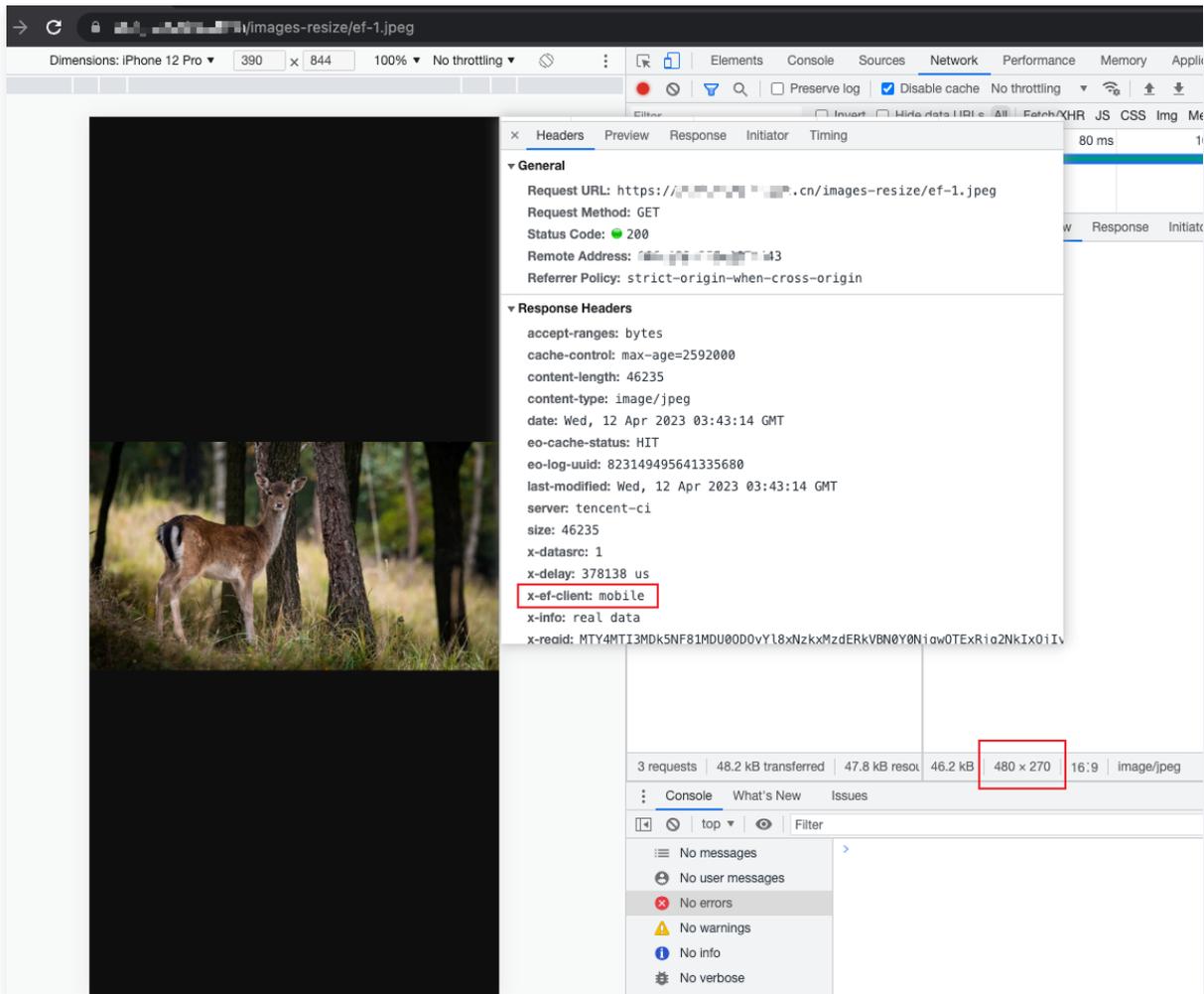
在 PC 端与移动端的浏览器地址栏中输入匹配到边缘函数触发规则的 URL（如：

`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：2024-07-05 15:41:41

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](#)。

⚠ 注意：

- 该示例仅在源文件的响应头 `Content-Type` 指定的 MIME 类型为图像（`image/*`）时，才能正常执行转换功能。
- 该示例暂不支持转换 `SVG` 格式图片。
- 建议边缘函数触发规则配置添加上文件后缀 `.png`、`.jpeg`、`.jpg` 等图片后缀。

```
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

自定义 Referer 限制规则

最近更新时间：2024-06-27 17:09:01

该示例通过检查 HTTP 请求头中的 Referer 字段，可以判断出请求的来源，您可根据自己的需求，灵活地设定 Referer 字段的匹配规则，如请求中的 Referer 字段缺失，或者其值与预设的域名列表不匹配，边缘函数将对此类请求进行拦截并返回403状态码，以表明访问被拒绝，常用于网站限制资源的访问。

```
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 });
  }

  // 设置 Referer 白名单
  const urlInfo = new URL(request.url);
  const refererRegExp = new RegExp(`^https?:\\/${urlInfo.hostname}\\/t-[0-9a-z]{10}\\/.*$`);

  // 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);
}

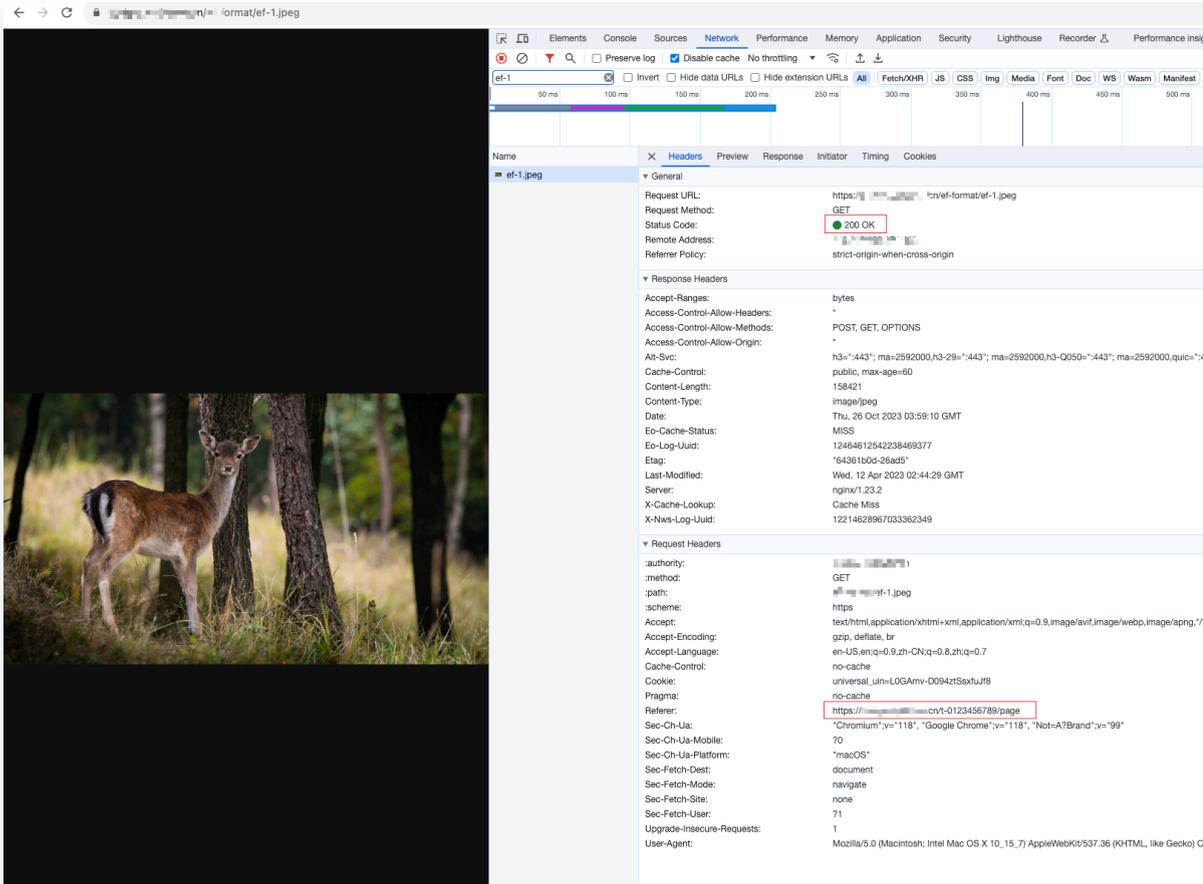
addEventListener('fetch', event => {
  // 当函数代码抛出未处理的异常时，边缘函数会将此请求转发回源站
  event.passThroughOnException();
  event.respondWith(handleRequest(event.request));
});
```

示例预览

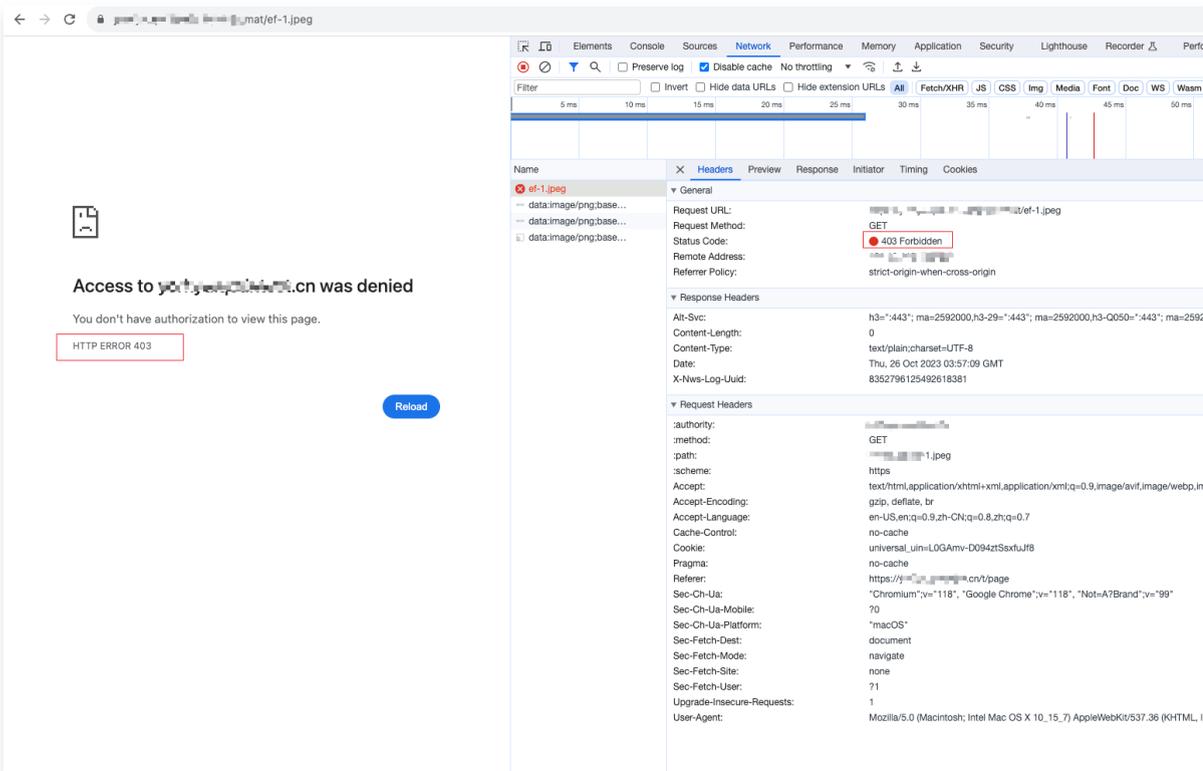
在 PC 端与移动端的浏览器地址栏中输入匹配到边缘函数触发规则的 URL（如：

`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：2024-08-23 14:25:12

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
  });
}

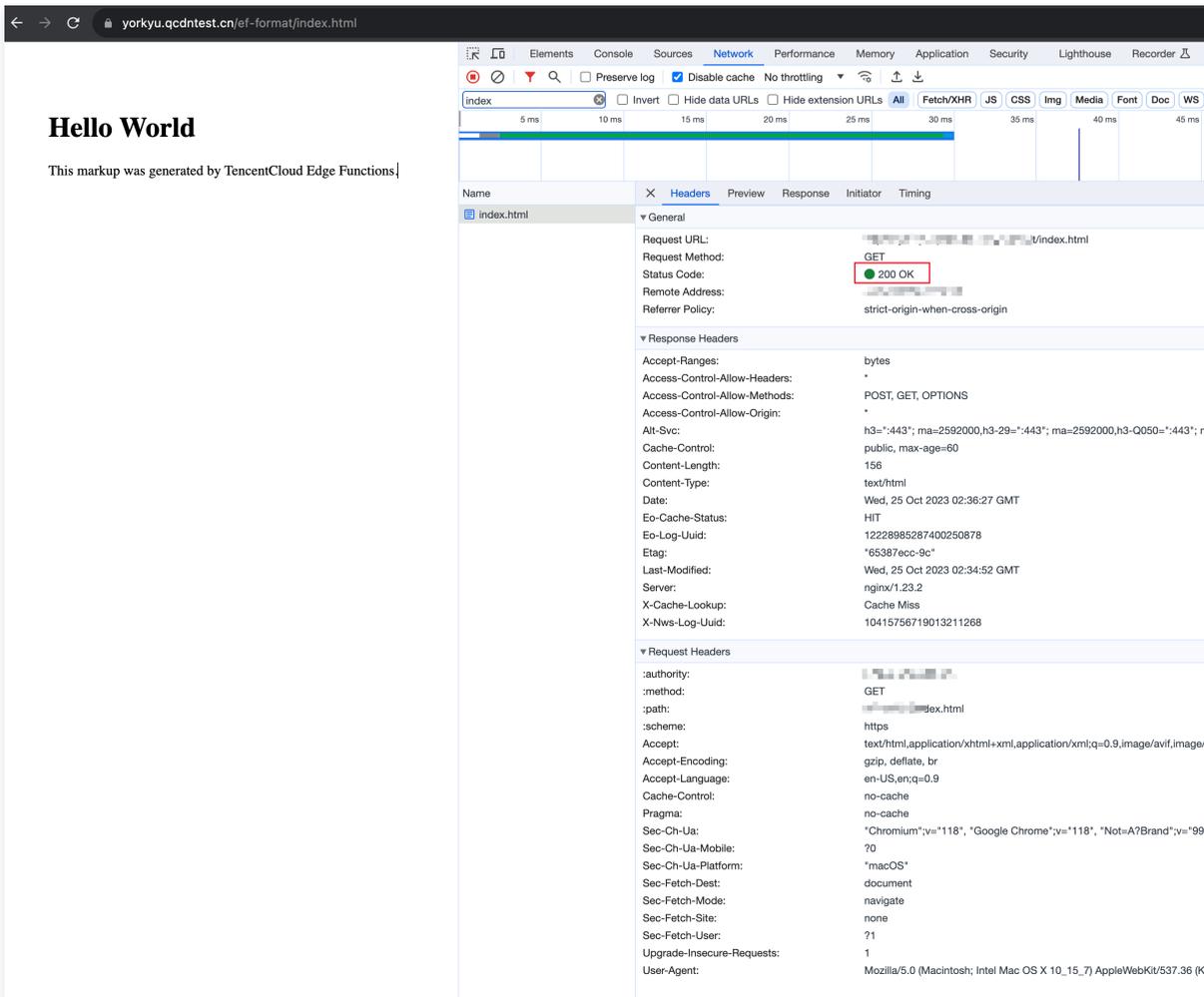
addEventListener('fetch', e => {
  e.respondWith(handleRequest(e.request));
});
```

示例预览

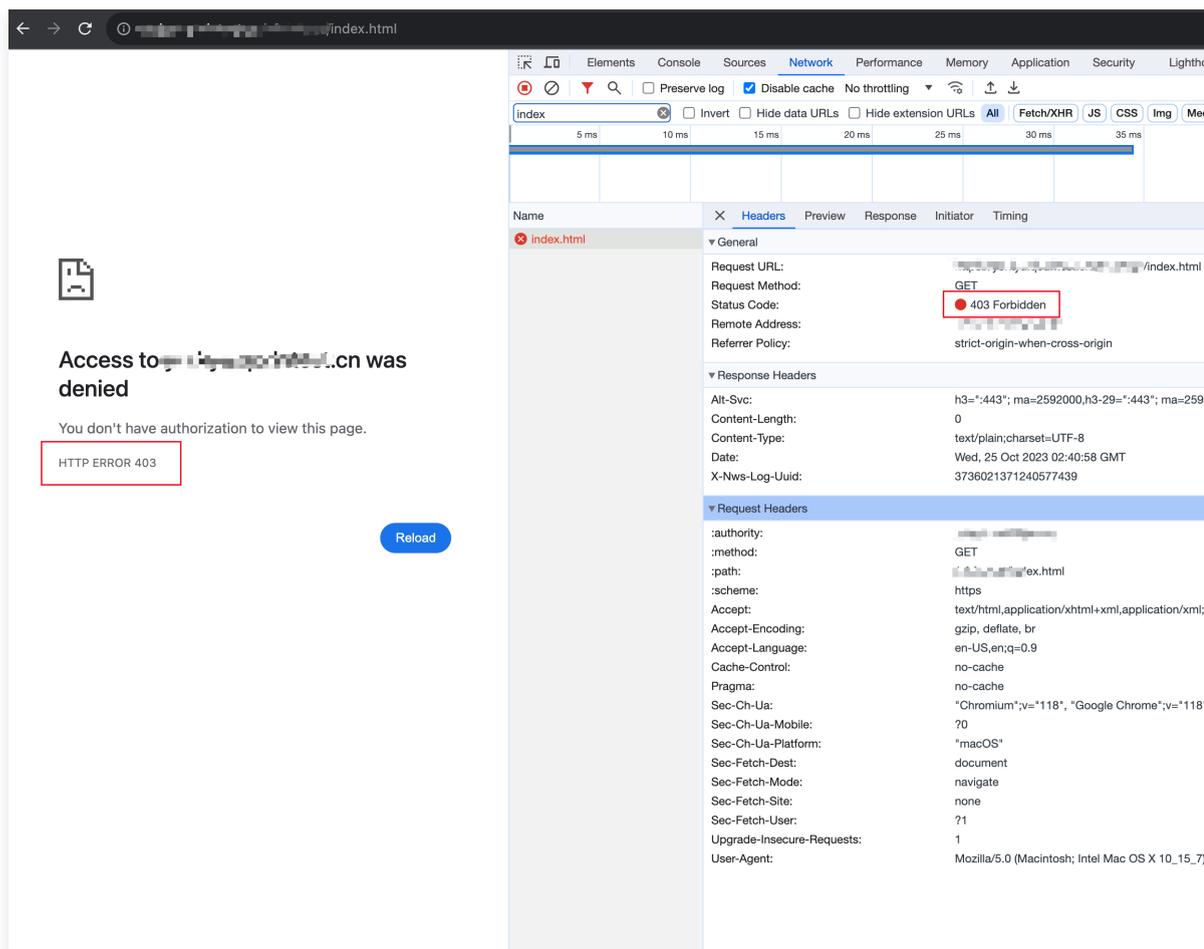
在 PC 端与移动端的浏览器地址栏中输入匹配到边缘函数触发规则的 URL（如：

`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：2024-08-23 14:25:12

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)).join('');
}

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', hash }, false, ['sign']);

  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}`);
}
```

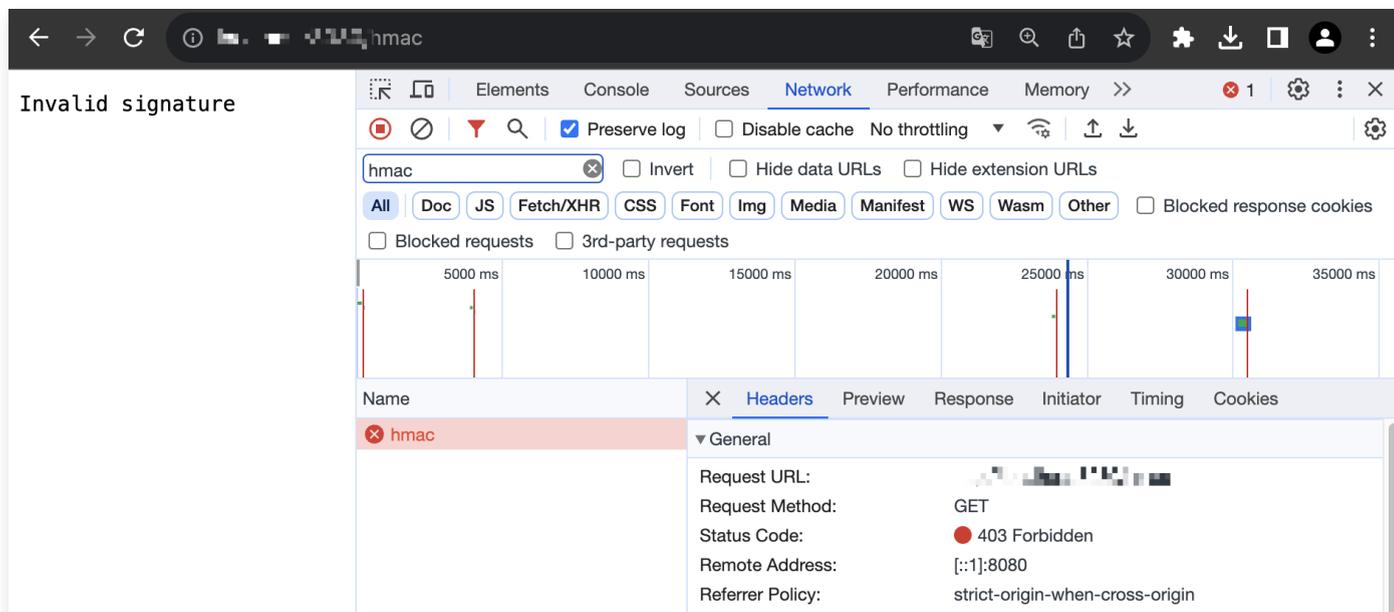
```
return fetch(request);
}

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

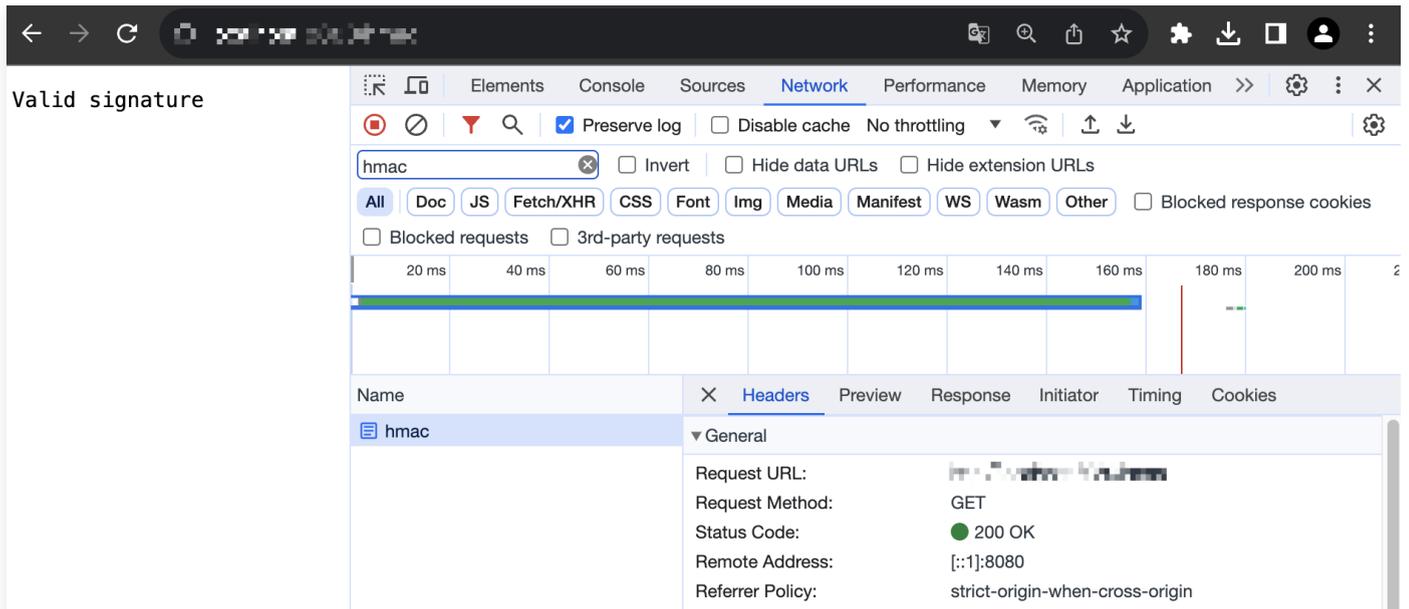
示例预览

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：2024-08-23 14:25:12

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);

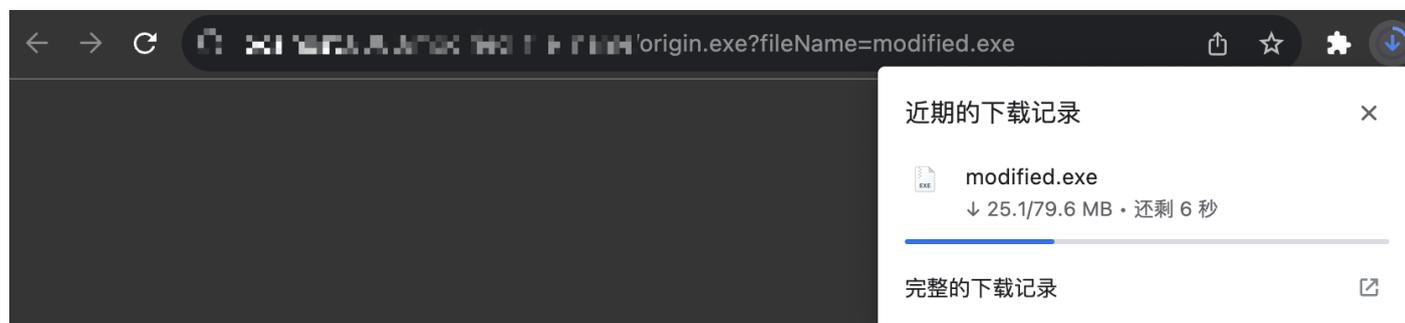
  // 判断响应状态码和 search 参数
  if (response.status !== 200 || !fileName) {
    return response;
  }

  // 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：2025-06-10 11:33:52

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。该示例通过 `Request` 对象的 `eo.clientIp` 属性获取客户端 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。

示例代码

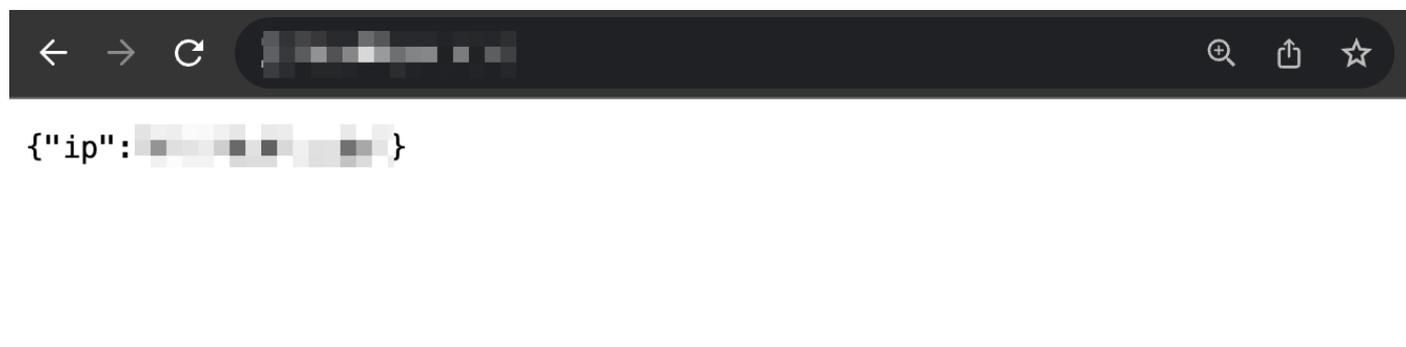
```
function handleRequest(request) {
  // 通过 request.eo.clientIp 获取客户端 IP
  const ip = request.eo.clientIp || '';

  // 响应 JSON 数据
  return new Response(JSON.stringify({ ip }), {
    headers: { 'content-type': 'application/json' },
  });
}

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

复杂回源 URL 重写

最近更新时间：2025-05-30 10:32:21

在某些业务场景下，需要对客户端请求的 URL 进行修改后再回源。本示例通过边缘函数实现了两种复杂 URL 重写场景：

- 路径正则替换：**使用正则表达式捕获组进行路径替换，将 `/a` 开头的路径替换为 `/path-a`，同时保留后续路径结构。通过路径正则替换可以将旧路径无缝映射到新路径，例如在内容管理系统改版时资源目录结构可能发生变化，使用正则捕获组可以保留原始路径中的关键部分并确保内容持续可访问，同时适应新的目录结构。
- 路径大小写转换：**将整个路径转换为小写，符合 Web 标准最佳实践。例如对于使用对象存储服务（如腾讯云 [对象存储](#)）作为源站的网站，由于对象存储通常区分大小写，统一转换可以简化资源管理并避免因大小写错误导致的资源访问失败。

示例代码

```
async function handleEvent(event) {
  try {
    const request = event.request;
    const url = new URL(request.url);
    let pathname = url.pathname;

    // 使用正则表达式进行路径替换，将 /a 或 /a/xxx 替换为 /path-a 或 /path-a/xxx
    if (pathname.startsWith('/a')) {
      const aPathRegex = /^\/a(\/.*)?$/;
      pathname = pathname.replace(aPathRegex, '/path-a$1');
    }

    // 路径大小写转换，直接将整个路径转换为小写
    if (pathname.startsWith('/b')) {
      pathname = pathname.toLowerCase();
    }

    url.pathname = pathname;

    // 创建新的请求
    const newRequest = new Request(url.toString(), {
      method: request.method,
      headers: request.headers,
      body: request.body,
      redirect: 'manual'
    });
```

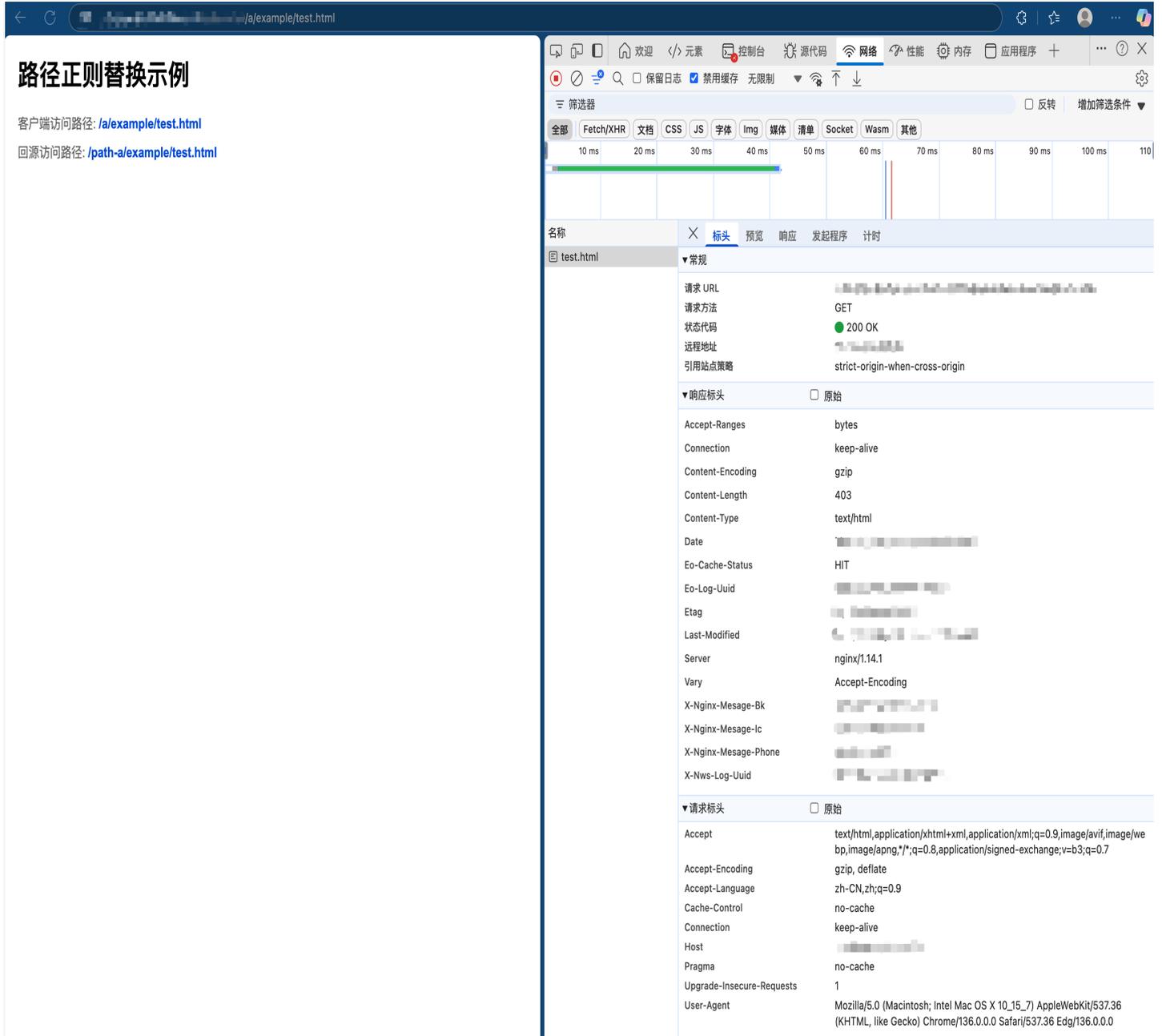
```
const response = await fetch(newRequest);
return event.respondWith(response);
} catch (err) {
  console.log(err);
}
}

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

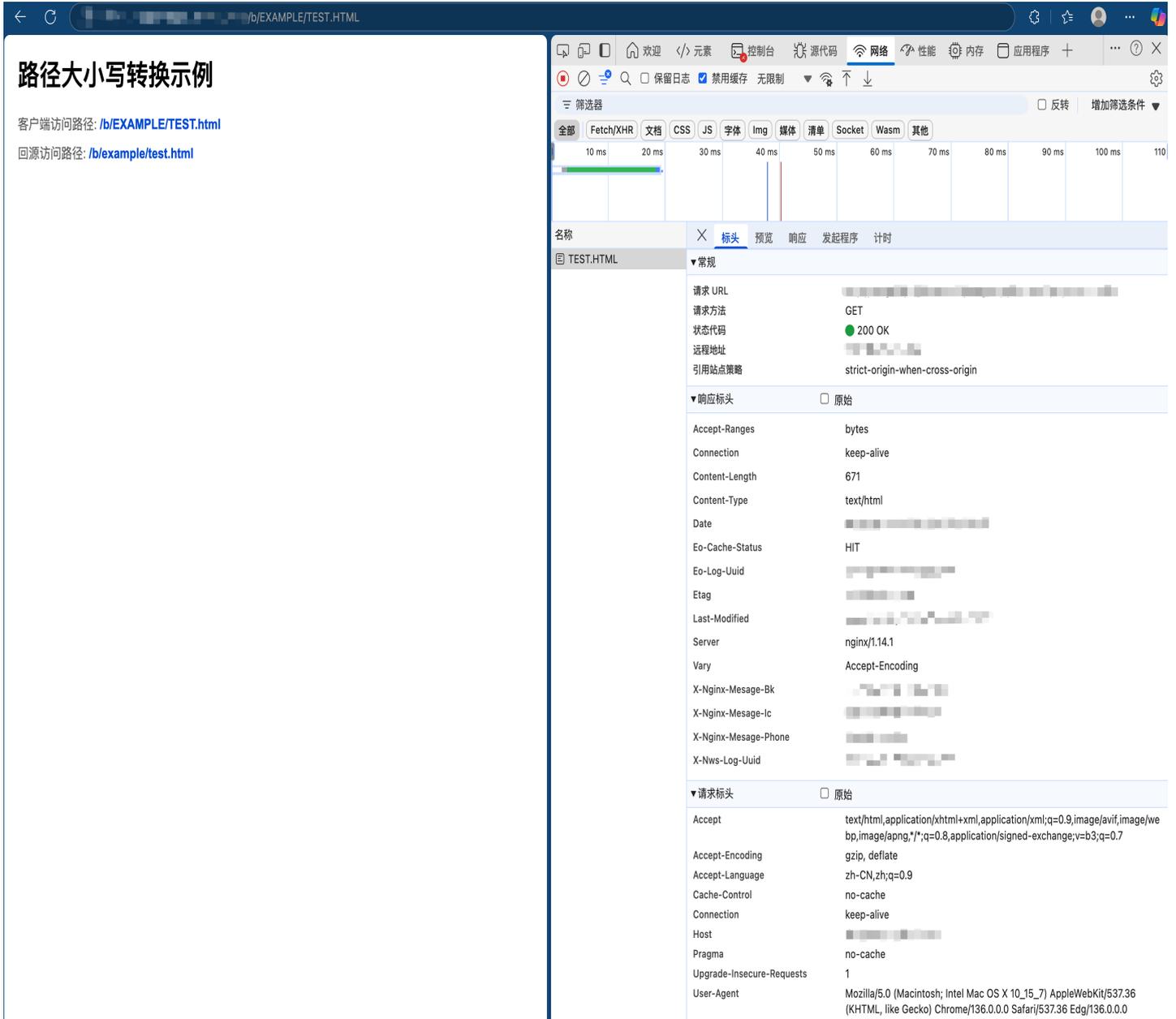
示例预览

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。

- 路径正则替换：将 `/a` 开头的路径替换为 `/path-a`，同时保留后续路径结构。



- 路径大小写转换：将整个路径转换为小写。



相关参考

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

实践教程

概览

最近更新时间：2024-07-25 16:21:42

| 标题 | 实践简介 |
|--|---|
| 通过边缘函数实现自适应图片格式转换 | 本文介绍了在不修改原始客户端请求 URL 的情况下，如何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。 |
| EdgeOne 实现基于客户端 IP 地址的会话保持 | 本文介绍了如何通过 EdgeOne 的边缘函数加规则引擎实现同一客户端 IP 地址的请求始终被转发到同一台后端源站，从而保持会话的连续性和数据的一致性。 |
| EdgeOne 实现基于客户端地理位置的指定回源 | 本文介绍了如何通过 EdgeOne 的边缘函数加规则引擎实现基于用户地理位置回源到指定的源站。 |

根据用户 IP/地理位置回源

EdgeOne 实现基于客户端 IP 地址的会话保持

最近更新时间：2024-12-31 15:50:32

本文档学习预计需要10分钟，通过学习该文档，您可以了解到：

1. 什么是基于客户端 IP 地址进行会话保持？为什么需要？
2. 基于客户端 IP 地址的会话保持的适用场景。
3. 基于客户端 IP 地址的技术架构及原理说明。
4. EdgeOne 边缘函数加规则引擎实现基于客户端 IP 地址的会话保持的具体步骤。

背景介绍

随着互联网的快速发展，企业业务不断扩展并深化用户体验，单一的源站服务器逐渐无法满足处理大量并发请求的需求，为了提升服务的可用性和可扩展性，企业开始采用负载均衡技术，将用户请求分发到多个后端源站上进行处理。然而，在业务发展的初期，由于用户量相对较少，会话管理相对简单，通常不会遇到基于客户端 IP 地址的会话保持问题，但随着业务的进一步发展，特别是在以下场景中，基于客户端 IP 地址的会话保持需求变得尤为迫切：

- 用户登录状态保持：在一些需要用户登录的应用中，如电子商务网站、在线银行等，用户登录后会在源站上生成一个会话（Session），用于记录用户的登录状态、购物车信息、订单详情等。如果用户在浏览过程中被分配到不同的后端源站，可能会因为会话信息的丢失而需要重新登录，严重影响用户体验。
- 数据一致性要求高的业务：在一些对数据一致性要求极高的业务中，如金融交易、在线支付等，如果会话被分配到不同的源站，可能会导致数据不一致或丢失，给用户和企业带来严重的损失。

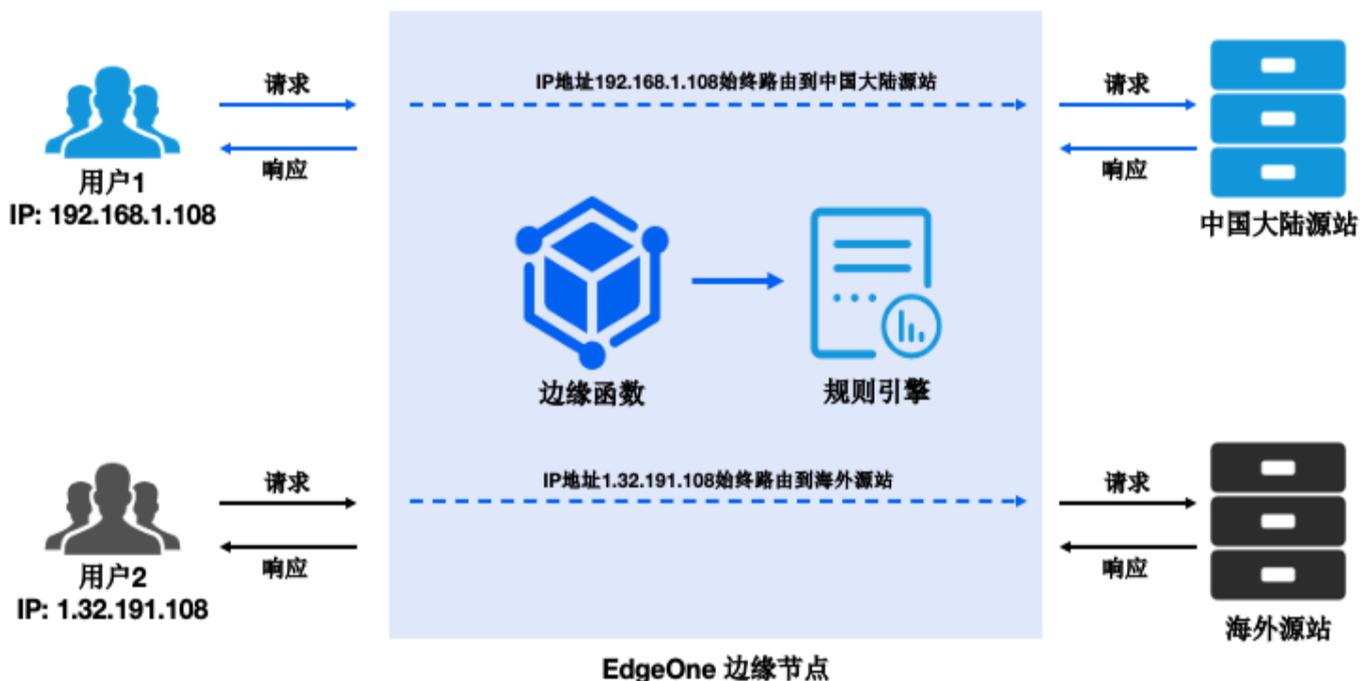
EdgeOne 在上述背景下推出基于客户端 IP 地址的会话保持，即通过 EdgeOne 的边缘函数加规则引擎实现来自同一客户端 IP 地址的请求始终被转发到同一台后端源站上，从而保持会话的连续性和数据的一致性。

适用场景

通过识别客户端的 IP 地址来确保同一个客户端的请求被定向到同一个源站，适用于以下业务场景：

- 金融服务：在线银行、股票交易等应用需要确保用户在整个交易过程中的所有请求都通过同一源站处理，以维护交易的安全性和一致性。
- 电子商务网站：确保用户登录后，其所有请求都被路由到同一源站，以保持购物车信息、用户偏好设置和登录状态。

技术架构



基于客户端 IP 地址的会话保持具体原理为边缘函数根据客户端的 IP 地址（如上图的 IP：192.168.1.108），通过哈希算法将客户端映射到不同的源站，规则引擎获取在边缘函数中自定义的回源请求头，根据请求头的值实现同一个客户端总是回源到同一个源站，以此来实现客户端到特定源站的一致性。此方案不仅可以提升用户体验，还可以确保业务数据的准确性。

预设场景

假设您是一家全球性应用服务的技术负责人，已将您的站点域名 `example.com` 接入到 EdgeOne。您期望根据用户的 IP 地址的哈希值始终将请求路由到对应的源站，确保用户无论身处何地，同一用户的请求始终被路由到同一源站，有助于优化缓存效率、简化会话管理、实现负载均衡和提供个性化服务，并确保数据处理的合法合规。

在这个场景中，您面对的是数百万不同的用户，这些用户的请求需要被均匀地分配到中国大陆的源站和新加坡的源站，同时您期望同一个 IP 地址的请求始终被路由到同一源站，以实现一致性的用户体验和高效的资源利用。在该示例中，将需要转发到中国大陆的源站的 IP 通过边缘函数增加一个回源请求头为：

`X-Forwarded-For-Origin: originGroup1`，将需要转发到新加坡的源站的 IP 通过边缘函数增加一个回源请求头为：`X-Forwarded-For-Origin: originGroup2`。

操作步骤

步骤1：接入 EdgeOne

参照 [从零快速开始接入 EdgeOne](#)，完成站点接入及域名接入。

步骤2：创建并配置边缘函数

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. 在选择模板创建页面，选择为**创建 Hello World**后，单击**下一步**。
5. 在新建函数页面，输入函数名称、函数描述和函数代码。以下为基于客户端 IP 地址的会话保持示例代码：

```
// 根据客户端ip地址，返回到不同的源站组，也就是在IP地址不变的情况下，同一个客户端回到同一个源站
const ORIGIN_GROUPS = ["originGroup1", "originGroup2"];

// 定义虚拟节点数，若源站组 (ORIGIN_GROUPS) 较多时，建议调低虚拟节点数
const VIRTUAL_NODES_PER_GROUP = 15;
const ORIGIN_HEADER_NAME = 'X-Forwarded-For-Origin';
let virtualNodesHashesCache = null;

// 定义全局变量来跟踪函数调用次数
addEventListener("fetch", (event) => {
  handleRequest(event.request);
});

async function handleRequest(request) {
  // 通过 EO-Client-IP 头部获取客户端 IP
  const ip = request.headers.get("EO-Client-IP") || "";

  // 如果缓存中没有虚拟节点哈希值，则生成虚拟节点哈希值
  if (!virtualNodesHashesCache) {
    virtualNodesHashesCache = await generateVirtualNodesHashes();
  }

  const group = await findSourceGroupForIp(
    ip,
    virtualNodesHashesCache.hashes,
    virtualNodesHashesCache.mapping
  );
  console.log(`Group: ${group}`);

  request.headers.set(ORIGIN_HEADER_NAME, group)

  return;
}

// 生成虚拟节点的哈希值
async function generateVirtualNodesHashes() {
```

```
const virtualNodesHashes = {};  
  
for (let group in ORIGIN_GROUPS) {  
  for (let i = 0; i < VIRTUAL_NODES_PER_GROUP; i++) {  
    const virtualNodeIdentifier = `${group}-VN${i}`;  
    const hash = await md5(virtualNodeIdentifier);  
    if (!virtualNodesHashes[hash]) {  
      virtualNodesHashes[hash] = group;  
    }  
  }  
}  
  
const hashes = Object.keys(virtualNodesHashes).sort();  
  
return { hashes, mapping: virtualNodesHashes }; // 返回排序后的哈希值数组和映射  
}  
  
// 映射客户端IP到虚拟节点，并找到对应的源站组  
async function findSourceGroupForIp(ip, hashes, mapping) {  
  
  // 使用md5函数计算IP的哈希值  
  const ipHash = await md5(ip);  
  let closestHash = hashes.find((hash) => hash > ipHash) || hashes[0];  
  
  // 根据找到的虚拟节点哈希值，从虚拟节点哈希表中获取对应的源站组名称  
  const selectedGroupName = mapping[closestHash];  
  const selectedGroupIPs = ORIGIN_GROUPS[selectedGroupName];  
  
  // 打印日志，显示客户端IP、哈希值、最接近的虚拟节点哈希值、选定的源站组名称  
  console.log(  
    `IP: ${ip}, Hash: ${ipHash}, Closest Hash: ${closestHash},  
    Selected Group: ${selectedGroupName}`  
  );  
  
  // 返回选定的源站组名称  
  return selectedGroupIPs;  
}  
  
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));  
}
```

步骤3：配置并部署边缘函数的触发规则

1. 编辑完成函数后，单击**创建并部署**，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。



2. 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，您可以配置多条触发条件，以 And 逻辑触发。

此处仅配置该请求 HOST 等于 `example.com`。

当请求 URL 同时符合以上条件时，将触发步骤1中的边缘函数，实现基于客户端 IP 地址的会话保持功能。



3. 单击**确定**触发规则即可生效。

步骤4：配置规则引擎

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**站点加速**，进入站点全局配置页面，单击**规则引擎** Tab 页。
3. 在规则引擎页面，单击**创建规则**，选择**新增空白规则**。
4. 在规则编辑页面，选择 **HOST** 匹配类型以匹配指定域名的请求。

此处仅配置该请求 HOST 等于 `example.com`。

5. 在规则编辑页面，打开客户端 IP 头部，参照 [获取客户端 IP](#)。

此处仅配置头部名称为 EO-Client-IP。

6. 在规则编辑页面，单击+IF，根据边缘函数的函数代码中的请求头取值，配置不同的源站组。

此处仅配置当满足条件为 HTTP 请求头 X-Forwarded-For-Origin 等于 originGroup1 时，请求将被转发至中国大陆的源站组进行处理；当满足条件为 HTTP 请求头 X-Forwarded-For-Origin 等于 originGroup2 时，请求将被转发至新加坡的源站组进行处理。



7. 单击保存并发布规则引擎即可生效。

步骤5：验证部署效果

经过测试，本示例展现出了良好的负载均衡能力，负载均衡占比在50%上下浮动，并且能够有效保持用户会话的一致性，证明了部署效果符合预期。

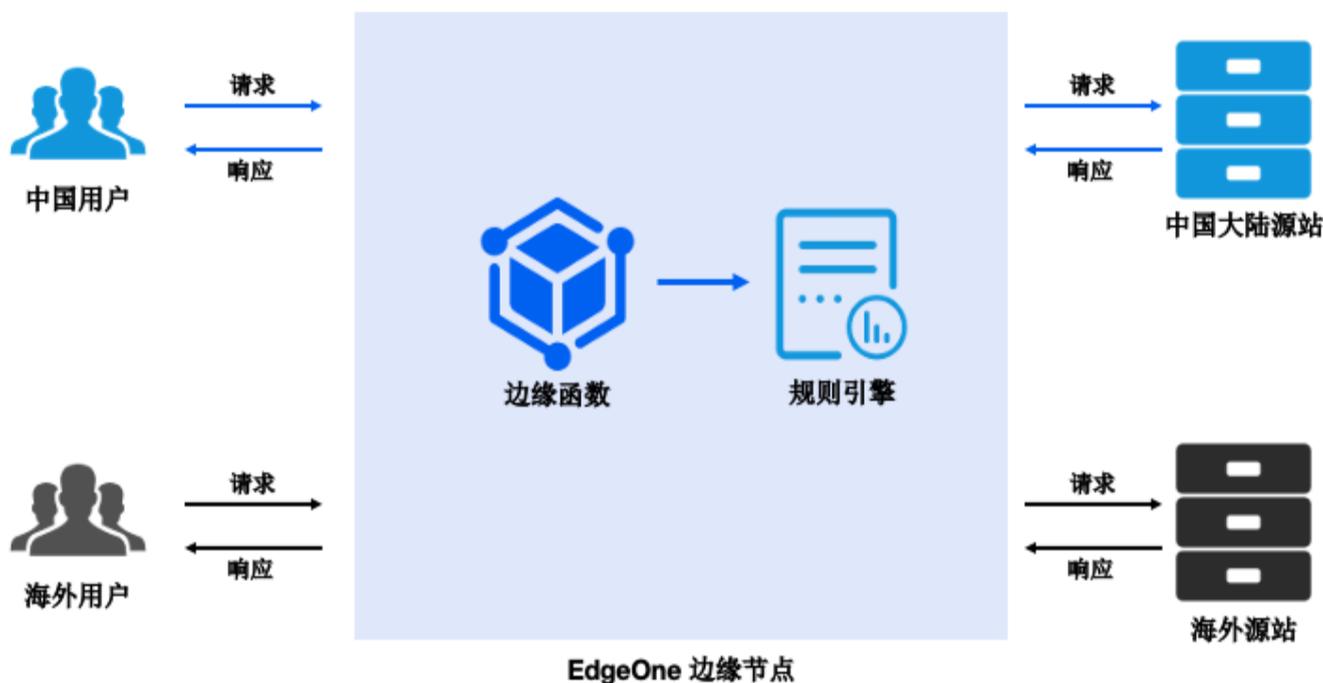
EdgeOne 实现基于客户端地理特征的指定回源

最近更新时间：2024-11-29 14:52:12

本文档学习预计需要10分钟，通过学习该文档，您可以了解到：

1. 为什么需要根据客户端地理特征信息指定回源？
2. 根据客户端地理特征信息指定回源的适用场景。
3. EdgeOne 边缘函数加规则引擎实现基于客户端地理特征指定回源的具体步骤。

背景介绍



在当今全球化的业务中，企业必须提供能够跨越地理界限的服务，确保无论用户身处何地都能获得一致的体验。全球化服务诉求要求企业能够快速响应不同地区用户的需求，提供个性化的内容和服务。此外，随着数据保护法规在全球范围内的日益严格，如欧盟的通用数据保护条例（GDPR），企业在处理用户数据时必须确保合规性，避免法律风险和声誉损失。

通过 EdgeOne 边缘函数加规则引擎实现了基于用户地理位置回源到就近指定源站的方案可应对上述挑战，它通过分析用户的地理位置和网络请求，将请求路由到指定最佳的源站。具体原理为边缘函数根据客户端所处地理位置，自定义回源请求头，规则引擎根据此请求头回源到指定的源站。此方案不仅提升了服务的响应速度和性能，还确保了数据处理的合规性，无论用户位于全球的哪个区域，都能享受到快速且符合当地法规的服务体验，有助于您的企业在全全球市场中保持竞争力。

适用场景

EdgeOne 根据全球用户客户端所处地域分发到指定地域的源站，以下是具体的应用场景：

- **企业出海：**帮助企业出海场景在全球范围内提供定制化的服务。例如在金融服务领域，确保交易请求被快速准确地路由到最近的服务器，减少延迟，同时提供地区特定的投资建议和 market 分析，以满足不同地区用户的需求。此外，跨境电商平台通过定位用户地理位置，提供本地化的产品和服务，优化库存和物流策略，提高用户体验和运营效率。
- **数据隐私合规：**随着数据保护法规的日益严格，企业需要确保其数据处理活动符合用户的地理位置所对应的法规要求。企业可以将数据请求路由到符合当地数据保护法规的服务器，确保数据的合规性。这不仅有助于企业避免法律风险，也是对用户隐私的尊重，有助于建立用户信任，为企业的可持续发展打下坚实基础。

预设场景

假设您是一家全球性电商平台的技术负责人，已将您的站点域名 `example.com` 接入到 EdgeOne。您的目标是优化网站的访问速度和用户体验，确保全球用户都能快速访问网站内容。为了实现这一目标，您计划根据用户的地理位置将请求动态路由到最近的源站，同时确保数据处理活动符合用户所在地区的数据保护法规要求。

在这个场景中，您设定了两个客户端地区和两个对应的源站组：

中国大陆客户端：对于来自中国大陆的用户，您期望将他们的请求路由至位于**中国大陆的源站组**，这样可以确保数据在本地处理，减少数据传输的延迟，提高访问速度。同时，这也有助于确保数据处理活动符合中国大陆的数据保护法规，保护用户隐私，避免法律风险。需要将请求路由至中国大陆的源站，将通过边缘函数为请求定义一个回源请求头为：`X-Forwarded-For-Origin: cn`

新加坡客户端：对于来自新加坡（代表中国大陆以外地区的用户）的用户，您期望将他们的请求路由至位于**新加坡的源站组**。通过利用地理位置的优势减少延迟，提升用户体验的同时，也确保数据处理遵守新加坡的数据保护法规，维护用户数据的隐私安全。需要将请求路由至新加坡的源站，将通过边缘函数为请求定义一个回源请求头为：

```
X-Forwarded-For-Origin: sg
```

操作步骤

步骤1：接入 EdgeOne

参照 [从零快速开始接入 EdgeOne](#)，完成站点接入及域名接入。

步骤2：创建并配置边缘函数

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. 在选择模板创建页面，单击**创建 Hello World**后，单击**下一步**。
5. 在新建函数页面，输入函数名称、函数描述和函数代码。以下为根据客户端地理特征信息指定回源示例代码：

```
// 域名请求地域与源站组映射表
const ROUTE_CLIENT_ORIGIN_MAP = {
```

```
'example.com': {
  CN: 'cn',
  _DEFAULT_: 'sg',
},
};

// 定义用于标识客户端原始地域的HTTP头名称
const ORIGIN_HEADER_NAME = 'X-Forwarded-For-Origin';
// 定义境外地区的标识符
const OVERSEAS_AREA = '!CN';
// 定义默认地区的标识符
const DEFAULT_AREA = '_DEFAULT_';

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

async function handleEvent(event) {
  const { request } = event;
  // 删除请求头中可能存在的客户端原始地域标识，避免外部干扰
  request.headers.delete(ORIGIN_HEADER_NAME);

  let host = request.headers.get('host');

  // 尝试获取请求地理位置信息中的国家代码，本示例中为'CN'
  let countryCodeAlpha2 = request.eo.geo?.countryCodeAlpha2;

  // 根据主机名获取对应的地域与源站组映射，本示例中为{CN:'cn',_DEFAULT_: 'sg'}
  const clientOriginMap = ROUTE_CLIENT_ORIGIN_MAP[host];

  if (clientOriginMap) {
    // 尝试根据国家代码获取源站组名称，本示例中为'cn'
    const originName = clientOriginMap[countryCodeAlpha2];
    if (originName) {
      // 设置请求头，标识客户端原始地域
      request.headers.set(ORIGIN_HEADER_NAME, originName);
    } else if (clientOriginMap[OVERSEAS_AREA]) {
      // 设置请求头为境外地区标识
      request.headers.set(ORIGIN_HEADER_NAME,
clientOriginMap[OVERSEAS_AREA]);
    } else if (clientOriginMap[DEFAULT_AREA]) {
      // 如果没有匹配到具体地区，使用默认地区，本示例中为'sg'
      request.headers.set(ORIGIN_HEADER_NAME,
```

```
clientOriginMap[DEFAULT_AREA]);
    }
}

return fetch(request);
}
```

步骤3：配置并部署边缘函数的触发规则

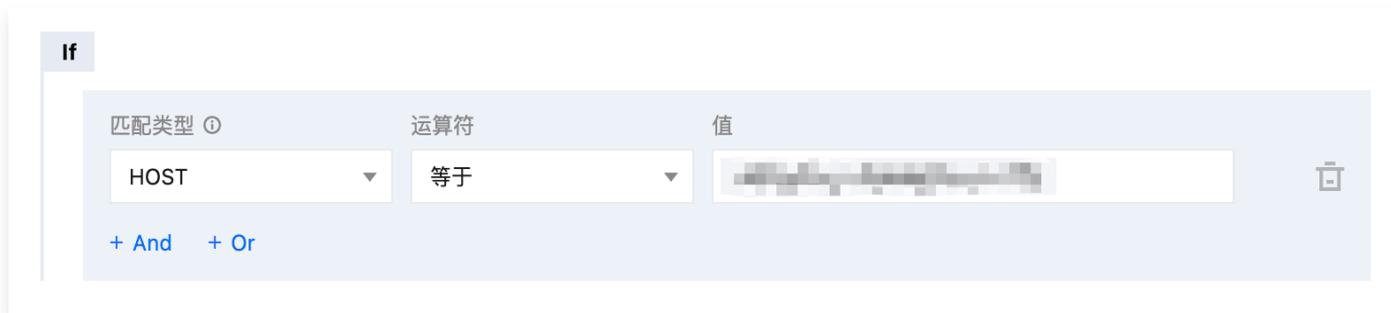
1. 编辑完成函数后，单击**创建并部署**，函数部署后，可直接单击**新增触发规则**，前往配置该函数的触发规则。



2. 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，您可以配置多条触发条件，以 And 逻辑触发。

此处仅配置该请求 HOST 等于 `example.com`。

当请求 URL 同时符合以上条件时，将触发步骤1中的边缘函数，实现根据客户端地理特征信息指定回源功能。



3. 单击**确定**触发规则即可生效。

步骤4：配置规则引擎

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**站点加速**，进入站点全局配置页面，单击**规则引擎** Tab 页。
3. 在规则引擎页面，单击**创建规则**，选择**新增空白规则**。
4. 在规则编辑页面，选择 **HOST** 匹配类型以匹配指定域名的请求。

此处仅配置该请求 HOST 等于 `example.com`。

5. 在规则编辑页面，单击**+IF**，根据边缘函数的函数代码中的请求头取值，配置不同的源站组。

此处仅配置当满足条件为HTTP请求头 `X-Forwarded-For-Origin` 等于 `cn` 时，请求将被转发至中国大陆的源站组进行处理；当满足条件为 HTTP 请求头 `X-Forwarded-For-Origin` 等于 `sg` 时，请求将被转发至新加坡的源站组进行处理。



6. 单击**保存并发布**规则引擎即可生效。

步骤5：验证部署效果

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

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://example.com
```

对于地理位置为 CN 的请求，边缘函数将请求回源到中国大陆的源站组：

```
→ ~ curl --user-agent "Chrome"
<!DOCTYPE html>
<html lang="zh-cn">
<head>
  <meta charset="UTF-8">
  <title>上海介绍</title>
  <style>
    h1 {
      font-weight: bold; /* 加粗文本 */
    }
  </style>
</head>
<body>
  <h1>这里是上海</h1>
</body>
</html>
```

对于地理位置为非 CN 的请求，边缘函数将请求回源到指新加坡的源站组：

```
→ ~ curl --user-agent "Chrome"
<!DOCTYPE html>
<html lang="zh-cn">
<head>
  <meta charset="UTF-8">
  <title>新加坡介绍</title>
  <style>
    h1 {
      font-weight: bold; /* 这会加粗文本 */
    }
  </style>
</head>
<body>
  <h1>这里是新加坡</h1>
</body>
</html>
```

浏览器访问测试

在 Chrome 浏览器中访问测试地址：`https://example.com`

对于地理位置为 CN 的请求，边缘函数将请求回源到中国大陆的源站组：

这里是上海

```
元素 控制台 源代码/来源 网络 性能 内存 应用
<!DOCTYPE html>
<html lang="zh-cn">
  <head> </head>
  <body>
    <h1>这里是上海</h1>
  </body>
</html>
```

对于地理位置为非 CN 的请求，边缘函数将请求回源到新加坡的源站组：

这里是新加坡

```
元素 控制台 源代码/来源 网络 性能 内存 应用
<!DOCTYPE html>
<html lang="zh-cn">
  <head> </head>
  <body>
    <h1>这里是新加坡</h1>
  </body>
</html>
```

了解更多

- [边缘安全加速平台 EO 获取客户端地理特征信息-边缘函数](#)
- [边缘安全加速平台 EO 基于客户端地理特征定制化-边缘函数](#)
- [边缘安全加速平台 EO 基于请求区域重定向-边缘函数](#)

APK 动态打包

EdgeOne 实现 Android APK 的动态打包

功能简介

最近更新时间：2024-09-06 17:01:52

本文主要介绍了如何通过腾讯云 EdgeOne + 腾讯云 COS + 腾讯云 SCF 产品组合在边缘实现 Android APK 多渠道的动态打包方案，相比传统的打包方案，具备一站式动态打包和加速的能力，可减少 APK 多渠道包的维护难度，降低接入成本。

背景介绍

APK (Android 应用程序包) 是 Android 应用程序的安装包，当 App 发布新版本，通常需要为每个渠道制作不同的渠道安装包，并将其上传到对应的应用市场。当用户从特定渠道下载并安装 App 后再上报数据，管理人员通过渠道标识来统计各个渠道的数据，如渠道下载量、转化率等关键数据，但是会碰到如下问题：

1. 渠道包维护成本高：Android App 开发完成后通常会在线上线下多渠道推广，如线上的应用市场、网盟和搜索引擎，线下的地推等，线上线下渠道商加起来有的多达上千个，如每个渠道需要维护一套渠道包，则投入成本高昂，效率低。
2. 渠道统计困难：渠道多的情况下，不同渠道的安装付费转化率需要统计，但传统的渠道统计依赖邀请码或手工等方式，自动化统计效果不佳。
3. 加速效果不理想：使用 CDN 对 APK 进行下载加速时，每一个 APK 渠道包都需要各自缓存，加速效果不均匀。

因此 EdgeOne 在上述背景下推出了边缘 Android APK 多渠道的动态打包方案。

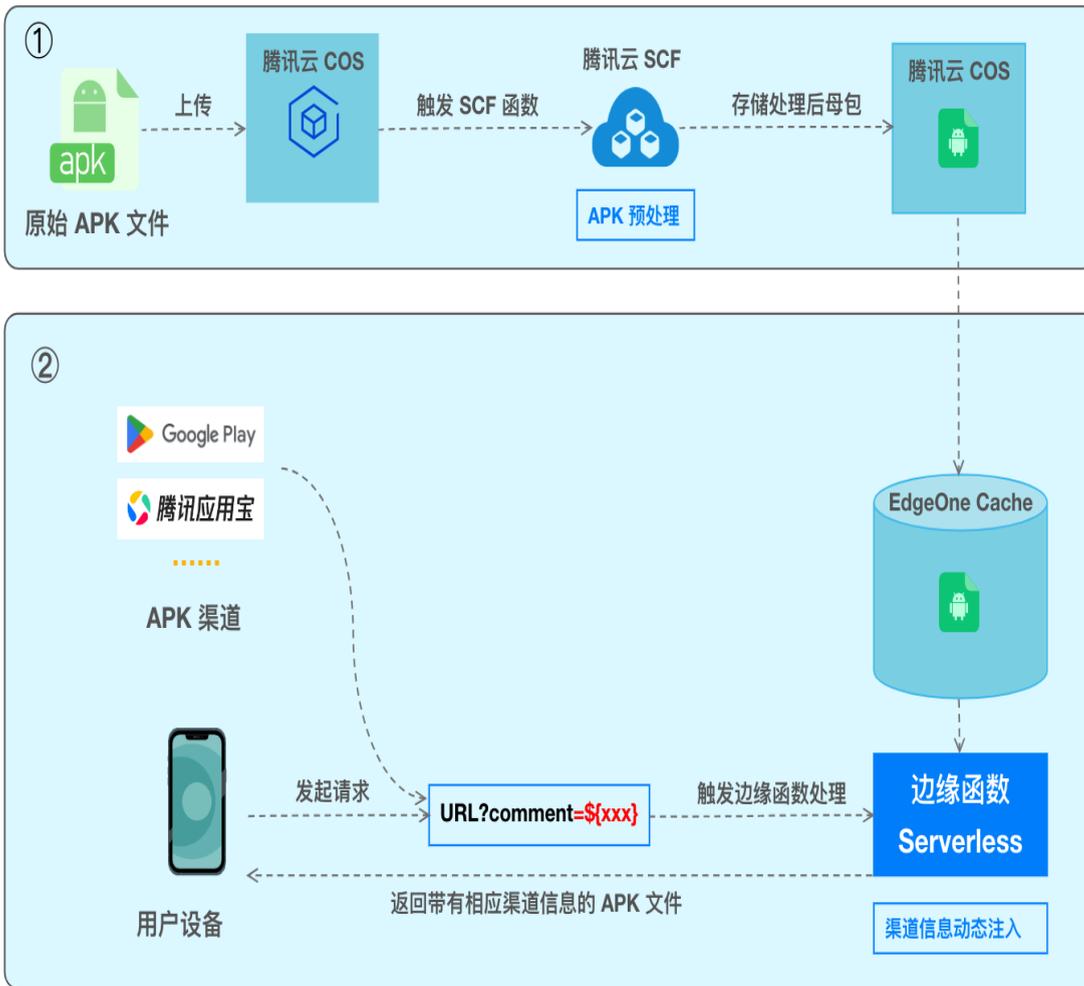
原理介绍

Android APK 多渠道的动态打包的实现有如下关键条件：

1. APK 包预处理：对 APK 母包注入空白数据，并将 APK 包处理为合法文件。
2. 下载 APK 包时注入渠道信息：当用户执行下载操作，将渠道信息动态地注入到 APK 包的适当位置并返回给用户下载。

通过以上方式将预处理 APK 包和渠道信息注入操作完成解耦，整个方案流程如图所示：

边缘 APK 多渠道动态打包



关键步骤 ①

将 APK 上传至腾讯云 COS 后，触发 SCF 函数为 APK 注入空白数据，并将 APK 包处理为合法文件。

关键步骤 ②

不同渠道的用户发起访问请求APK 后，EdgeOne 将通过边缘函数读取 comment 信息，并注入到 APK 包的适当位置。

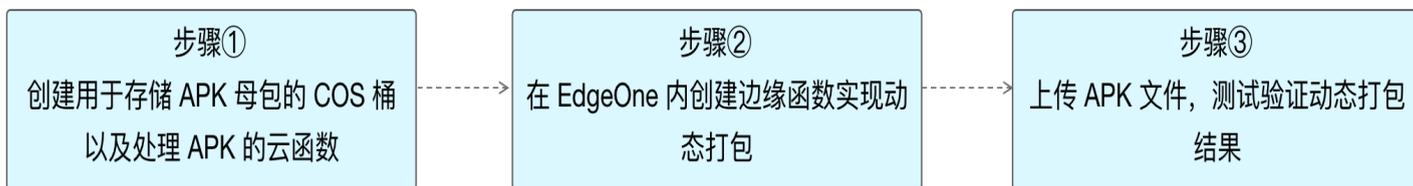
方案优势

1. 降低渠道包维护成本：开发者只需维护一份原始的 Android APK 母包，不必为每个渠道商单独维护，EdgeOne 默认提供打包工具，用户只需简单 UI 配置模板即可完成部署，有效提升多渠道打包的效率。
2. 精准高效的渠道统计：用户只需通过访问 URL + 带有渠道参数触发边缘函数，即可动态将渠道标识插入到 apk 包并返回给用户下载，同时可完成数据的上报统计。

| | EdgeOne 边缘 APK 动态打包 | VasDolly | Walle | ApkTool | Android Gradle Plugin |
|----------|------------------------|----------|-------|-------------|--------------------------|
| 打包速度 | 快 | 快 | 较快 | 需要解压和签名, 较慢 | 需要重新打包构建, 慢 |
| 渠道信息注入形式 | 动态 | 静态 | 静态 | 静态 | 静态 |
| 渠道信息注入端 | 边缘 | 源站 | 源站 | 源站 | 源站 |
| APK 出包数量 | 1 个 | 多个 | 多个 | 多个 | 多个 |
| 打包与加速 | EdgeOne 一站式打包 & 加速 | 不支持加速 | 不支持加速 | 不支持加速 | 不支持加速 |

操作步骤

假设您是一家游戏厂商，有 Android App 新游戏（例如：`v2_src.apk`）要发布到多个不同的渠道增加曝光量吸引更多玩家，例如：各大应用市场、应用商店、社交媒体平台、游戏论坛、广告平台等，您期望高效地针对每个渠道注入渠道标识、统计渠道收益和加速 apk 的下载。发布统一使用 `apk.example.com` 的域名进行分发。



- 步骤一：预处理 Android APK 母包
- 步骤二：EdgeOne 边缘函数写入渠道信息到 APK 包
- 步骤三：测试并验证生效结果

步骤一：预处理 Android APK 母包

最近更新时间：2024-08-23 14:25:12

本文将介绍如何通过对象存储 COS 和云函数 SCF，实现对 Android APK 母包的预处理。

准备工作

1. 已开通 [对象存储 COS](#) 和 [云函数 SCF](#)，记录该存储桶名称及地域信息。
2. 根据 [站点接入](#) 指引添加站点，购买 EdgeOne 套餐。
3. 已在 EdgeOne 控制台 [添加加速域名](#) `www.example.com`，且源站配置为腾讯云 COS。

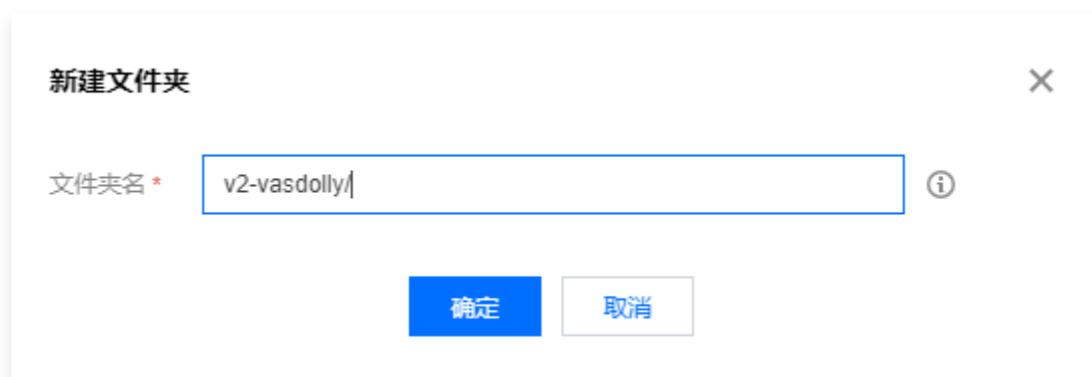
步骤1：上传 Android APK 母包

在对象存储 COS 中，上传 Android APK 母包。

1. 登录 [对象存储 COS 控制台](#)，在左侧菜单栏中，单击 [存储桶列表](#)。
2. 在存储桶列表页面，单击用于存储 APK 母包的存储桶名称。
3. 在文件列表，单击 [创建文件夹](#) 用于上传 APK 母包的目录，输入文件夹名称，例如：`v2-vasdolly/`，单击 [确定](#)。

⚠ 注意：

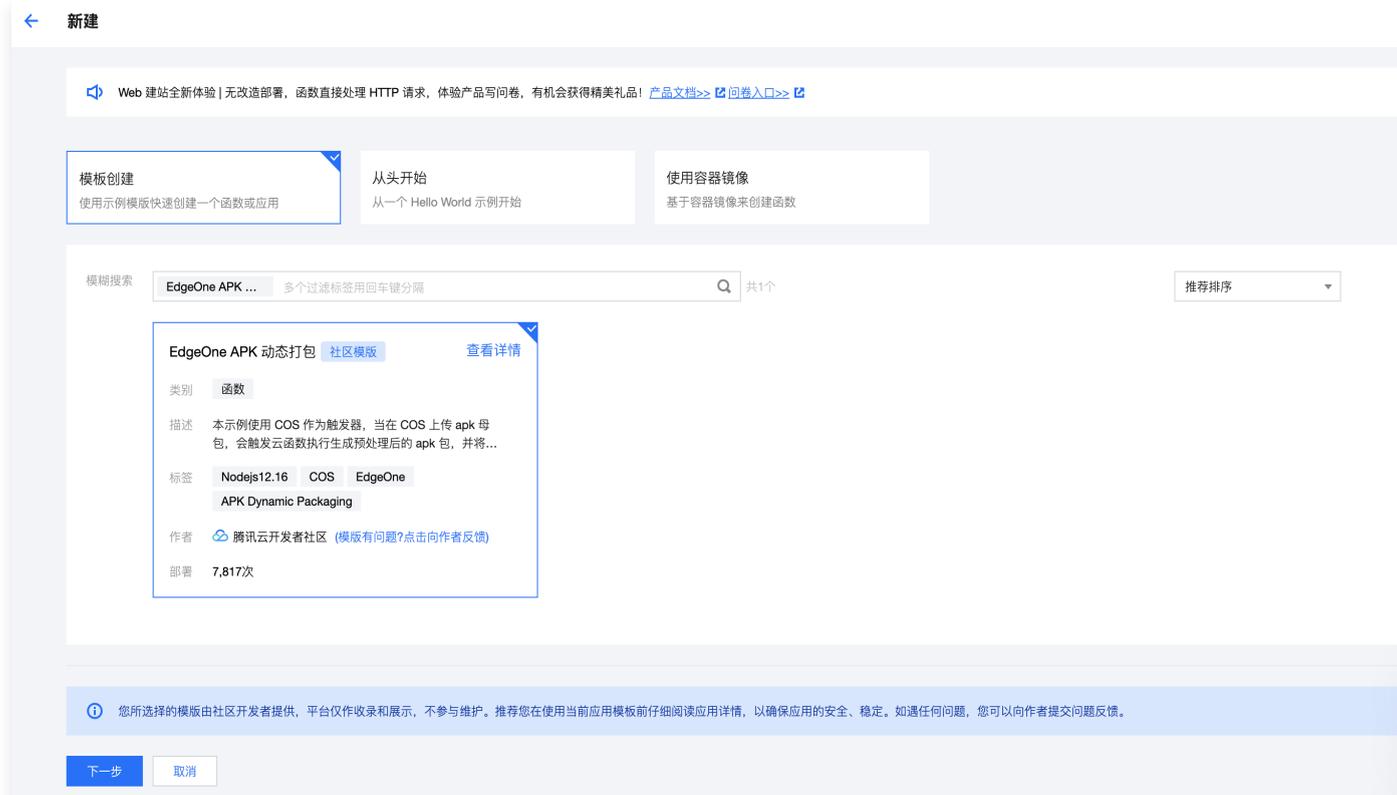
请勿直接将根目录作为 APK 母包的上传目录。



步骤2：新建模板函数

在云函数 SCF 通过 EdgeOne APK 动态打包的模板新建函数。

1. 登录 [云函数服务控制台](#)，在左侧菜单栏中，单击 [函数服务](#)。
2. 在函数服务页面，单击 [新建](#)，选择 [使用模板创建](#)，在模糊搜索中输入 [EdgeOne APK 动态打包](#) 并选中，单击 [下一步](#)。



3. 在“函数配置”页面，分别配置如下参数：

● 基础配置

- **函数名称：**创建函数时将自动生成一个函数名称，您可以选择自定义修改为一个易识别的函数名称。
- **地域：**请选择 **步骤1** 中所使用的对象存储桶 COS 的所在地域，例如：广州。
- **描述：**说明此函数的用途。
- **运行角色：**默认勾选启用，配置并使用 SCF 模板运行角色。如使用已有角色，请确保已有角色已包含 `QcloudCOSFullAccess` 的预设策略。

基础配置

函数名称 *

eo-edgepack-template-1695212f

只能包含字母、数字、下划线、连字符，以字母开头，以数字或字母结尾，2~60个字符

地域 *

广州

描述 *

本示例使用 COS 作为触发器，当在 COS 上传 apk 母包，会触发云函数执行生成预处理后的 apk 包，并将该包输出到 COS 指定的目录。

最大支持1000个英文字母、数字、空格、逗号、句号、中文

运行角色 *

启用 ⓘ

为保证该函数模版正常访问其他云服务，请选择配置并使用 SCF 模版运行角色或选择包含 QcloudCOSFullAccess,

配置并使用SCF模版运行角色 ⓘ

使用已有角色

- **函数代码：** 模板已内置默认函数代码实现 Android APK 处理母包的能力，无需改动。
- **环境配置：**

单击**高级配置**，选择**环境配置**，您需要在环境变量中添加以下 key 和对应的 value 值，其余配置保持默认即可：

- **outputPath (必填)：** 自定义云函数 SCF 处理 APK 母包后输出到 COS 存储桶的目录，如：
/v2-vasdolly_output。
- **packVersion (必填)：** 不同的 APK 版本所使用的签名版本信息，不同签名版本填写的值如下：

| APK 签名版本 | packVersion 值 |
|----------|--|
| v1 | v1 |
| v2 | 请填写 v2-VasDolly、v2-Walle 或 v2-Custom： <ul style="list-style-type: none"> ● v2-VasDolly：将渠道信息保存在 ID 为 0x881155ff（VasDolly 默认）的 ID-Value 对。 ● v2-Walle：将渠道信息保存在 ID 为 0x71777777（Walle 默认）的 ID-Value 对。 ● v2-Custom：将渠道信息保存在 ID 为 blockId（由 blockId 环境变量指定）的 ID-Value 对。 |

- **blockId (选填)：** 如果使用 v2-Custom 方式进行预处理，需要指定 blockId。

填写示例：

高级配置

命名空间

环境配置

内存 (i)

初始化超时时间 秒 (i)
时间范围：3-300秒

执行超时时间 秒 (i)
时间范围：1-900秒

环境变量

| key | value | |
|-------------|----------------------------|---|
| outputPath | /v2-vasdolly-layton output | X |
| packVersion | v2-VasDolly | X |

[导入](#)

- (可选) 文件系统：如果上传到 COS 的 APK 母包大于 200M，需前往 [文件存储控制台](#) 开通 CFS 服务，并启用文件系统，用于扩展云函数的本地存储空间：

网络配置

公网访问 启用 (i)

固定公网出口IP 启用 (i)

私有网络 启用 (i)
 [新建私有网络](#)

固定内网出口IP 启用 (i)
 固定内网出口IP需要在VPC私有网络下使用，请勾选私有网络。

文件系统

文件系统 启用 (i) 注意事项：文件系统不支持私有网络是9.x.x.x网段的场景。

文件系统ID [新建文件系统](#)

请先选择vpc

挂载点ID

用户ID

用户组ID

远程目录

本地目录

⚠ 注意:

由于 SCF 侧的限制，每个云函数在执行过程中，拥有一块 500MB 的临时磁盘空间，处理 APK 文件时，磁盘中同时存在原始 APK 文件和处理后的 APK 文件，因此处理过大的 APK 文件，需要挂在额外的文件存储系统，详情请参见 [挂载 CFS 文件系统](#)。

● 触发器管理

在触发器配置中，选择 COS Bucket 为与此云函数 SCF 同地域的存储桶，可输入存储桶名称进行模糊查询，例如：`apk-test-1251557890.cos.ap-guangzhou.myqcloud.com`。其余配置项保持为默认配置即可。

- 触发方式：选择 COS 触发。
- COS Bucket：请选择该可用区下母包所在的 COS 存储桶。
- 事件类型：选择全部创建。
- 前缀过滤：请输入母包上传的目录，如您的母包在 `v2-vasdolly` 目录下，即填写 `v2-vasdolly/`。
- 后缀过滤：请输入 `.apk`。

上述信息填写完成后，即只针对指定指定 COS 存储桶的 `v2-vasdolly/` 目录下且文件后缀为 `.apk` 的文件上传时才会触发云函数 SCF 执行。

触发器配置
创建触发器 腾讯云消息队列 CMQ 产品计划于 2022 年 6 月前完成全量下线，产品迁移过程中，不再支持新建 CMQ 触发器，已有触发器数据链路不受影响，详见 [CMQ 产品文档](#)

自定义创建

| | |
|---|--|
| 触发别名/版本 | 别名: 默认流量 |
| 触发方式 | COS 触发 |
| COS 可将事件发布给 SCF 函数并将事件数据作为参数来调用该函数，详情请 查阅文档 | |
| COS Bucket | <input type="text" value=""/> .cos.ap-guangzhou.myqcloud.com 新建 COS Bucket |
| 事件类型 | 全部创建 |
| 前缀过滤 | <input type="text"/> |
| 后缀过滤 | <input type="text"/> |
| 立即启用 | <input checked="" type="checkbox"/> 启用 |

暂不创建

4. 单击完成，即可完成 EdgeOne APK 动态打包函数的创建。

ⓘ 说明:

下一步请参考 [步骤二: EdgeOne 边缘函数写入渠道信息到 APK 包](#)。

步骤二：EdgeOne 边缘函数写入渠道信息到 APK 包

最近更新时间：2025-04-24 17:00:42

通过 EdgeOne 边缘函数，我们可以动态地将渠道信息写入到 APK 包内。用户只需访问与边缘函数绑定的域名并进行触发配置，就可以触发该边缘函数，从而实现 APK 的动态打包和加速分发。

步骤1：添加用于加速分发的加速域名

请根据 [添加加速域名](#) 指引添加加速域名，例如：`www.example.com` 且源站配置为 [Android APK 母包](#) 所在的对象存储 COS，如下所示：

说明：
该域名将用于访问下载 APK 安装包。



步骤2：创建用于触发渠道信息写入的边缘函数

1. 根据 [函数管理](#) 指引创建一个边缘函数，将如下代码复制到函数代码内。

```
const CUSTOM_BLOCK_VALUE_LENGTH = 10240;
const APK_SIGNING_BLOCK_MAGIC_LENGTH = 16;
const APK_SIGNING_BLOCK_OFFSET_LENGTH = 8;

const APK_COMMENT_LENGTH = 512;

class EdgePack {
  totalSize;
  signVersion;
  centralDirectoryOffset;
  customBlockValueStart;
  customBlockValueEnd;
  rangeRelativeOffset;
  customInfo;
```

```
constructor() {
  this.totalSize = null;
  this.signVersion = null;
  this.centralDirectoryOffset = null;
  this.customBlockValueStart = null;
  this.customBlockValueEnd = null;
  this.rangeRelativeOffset = null;
  this.customInfo = null;
}

async handle(event) {
  const { request } = event;

  const headers = new Headers(request.headers);

  const modifiedRequest = new Request(request, { headers });

  if (!this.checkRequest(modifiedRequest)) {
    return;
  }

  let response = null;
  try {
    const headRequest = new Request(modifiedRequest.url, {
      method: 'HEAD',
      headers: modifiedRequest.headers,
    });
    response = await fetch(headRequest);
  } catch (err) {
    const error = {
      code: 'FETCH_ORIGIN_ERROR',
      message: err?.message,
    };
    response = new Response(JSON.stringify(error), {
      status: 590,
    });
  }

  if (!this.checkResponse(response)) {
    return event.respondWith(response);
  }
}
```

```
response.headers.set('Cache-Control', 'max-age=0');

const streamResponse = new Response(
  await this.combineStreams(modifiedRequest),
  response
);

event.respondWith(streamResponse);
}

getRelativeOffset(response) {
  const start = this.customBlockValueStart;
  const end = this.customBlockValueEnd;

  const range = response.headers.get('Content-Range');

  if (!range) return start;

  const match = range.match(/bytes\s*(\d*)-(\d*)/i);
  if (!match || match?.length < 2) {
    return start;
  }

  if (+match[2] < start || +match[1] > end) {
    return null;
  }

  return start - +match[1];
}

checkRequest(request) {
  if (request.method !== 'GET') {
    return false;
  }

  if (request.headers.has('Range')) {
    return false;
  }

  const { pathname, searchParams } = new URL(request.url);

  const comment = searchParams?.get('comment');
```

```
if (!pathname.endsWith('.apk') || !comment) {
    return false;
}

this.customInfo = comment;
return true;
}

checkResponse(response) {
    if (response.status !== 200 && response.status !== 206) {
        return false;
    }

    const contentLength = response.headers.get('Content-Length');

    if (response.body === null || contentLength === null) {
        return false;
    }

    this.totalSize = Number(contentLength);

    const cosOffsetHeader = response.headers.get('x-cos-meta-edgepack-
offset');
    const cosTypeHeader = response.headers.get('x-cos-meta-edgepack-
type');

    if (!cosOffsetHeader || !cosTypeHeader) {
        return false;
    }

    this.signVersion = cosTypeHeader;
    this.centralDirectoryOffset = Number(cosOffsetHeader);

    if (this.signVersion === 'v1') {
        this.customBlockValueStart = this.totalSize -
APK_COMMENT_LENGTH;
        this.customBlockValueEnd = this.totalSize;
    } else {
        this.customBlockValueStart =
            this.centralDirectoryOffset -
CUSTOM_BLOCK_VALUE_LENGTH -
APK_SIGNING_BLOCK_MAGIC_LENGTH -
APK_SIGNING_BLOCK_OFFSET_LENGTH;
```

```
        this.customBlockValueEnd = this.centralDirectoryOffset;
    }

    this.rangeRelativeOffset = this.getRelativeOffset(response);

    if (this.rangeRelativeOffset === null) {
        return false;
    }

    return true;
}

async combineStreams(request) {
    const { readable, writable } = new TransformStream();
    this.handleStream(request, writable);
    return readable;
}

async handleStream(request, writable) {
    const comment = this.customInfo;
    const relativeOffset = this.rangeRelativeOffset;

    const encoder = new TextEncoder();
    const section = encoder.encode(comment);

    try {
        const apkHeader = await this.apkHeaderStream(request);

        try {
            await apkHeader.pipeTo(writable, {
                preventClose: true,
            });
        } catch (e) {
            console.error('HEADER_STREAM_ERROR: ', e);
        }
    }

    // 返回的是Blob数据
    const apkBody = await this.apkBodyStream(
        request,
        section,
        relativeOffset
    );
}
```

```
const apkBodyStream = apkBody.stream();

try {
  await apkBodyStream.pipeTo(writable, {
    preventClose: true,
  });
} catch (e) {
  console.error('BODY_STREAM_ERROR: ', e);
}

const apkTail = await this.apkTailStream(request);

try {
  await apkTail.pipeTo(writable, {
    preventClose: true,
  });
} catch (e) {
  console.error('TAIL_STREAM_ERROR: ', e);
}
} catch (err) {
  console.error('HANDLE_STREAM_ERROR: ', err);
} finally {
  let writer = writable.getWriter();
  writer.close();
  writer.releaseLock();
}
}

async apkHeaderStream(request) {
  const headers = new Headers(request.headers);
  headers.set('Range', `bytes=0-${this.customBlockValueStart - 1}`);

  // 获取签名块之前的部分
  const headResponse = await fetch(request, {
    headers: headers,
  });

  return headResponse.body;
}

async apkBodyStream(request, section = null, relativeOffset = 0) {
  const headers = new Headers(request.headers);
  headers.set(
```

```
    'Range',
    `bytes=${this.customBlockValueStart}-${this.customBlockValueEnd
- 1}`
  );

  const middleResponse = await fetch(request, {
    headers: headers,
  });

  const reader = middleResponse.body.getReader();

  let outputBuffers = [];
  try {
    let handledBytes = this.customBlockValueStart;
    while (true) {
      const result = await reader.read();

      if (result.done) {
        console.log('APK_BODY_STREAM_DONE');
        break;
      }

      const startByteOffset = handledBytes;
      const buffer = result.value;
      handledBytes += buffer.byteLength;

      const min = Math.max(startByteOffset, relativeOffset);
      const max = Math.min(relativeOffset + section.byteLength,
handledBytes);

      if (min < max) {
        const bufferStart = min - startByteOffset;
        const sectionStart = min - relativeOffset;
        const sectionEnd = max - relativeOffset;

        const replacement = section.subarray(sectionStart,
sectionEnd);

        new Uint8Array(buffer).set(replacement, bufferStart);
      }

      outputBuffers.push(buffer);
    }
  }
```

```
    } catch (err) {
      console.error('APK_BODY_STREAM_ERROR: ', err);
    }
    return new Blob(outputBuffers);
  }

  async apkTailStream(request) {
    const headers = new Headers(request.headers);
    headers.set(
      'Range',
      `bytes=${this.customBlockValueEnd}-${this.totalSize - 1}`
    );

    const tailResponse = await fetch(request, {
      headers: headers,
    });

    return tailResponse.body;
  }
}

async function handleEvent(event) {
  const edgepack = new EdgePack();
  await edgepack.handle(event);
}

addEventListener('fetch', handleEvent);
```

2. 完成部署函数后，根据指引 [函数管理](#) 配置触发规则，其 HOST 值为 [步骤1](#) 创建的加速域名，如下图所示：

v/新增触发规则 X

! 请求URL匹配到触发条件后则执行函数。

站点 .cn

描述

您还可以输入45个字符

触发条件

If

And

| 匹配类型 ① | 运算符 | 值 | |
|--------|-----|--------------------------------------|--|
| HOST | 等于 | <input type="text" value="..."/> .cn | |
| 文件后缀 | 等于 | <input type="text" value="apk"/> | |

[+ And](#) [+ Or](#)

3. 单击**确定**，即可完成触发规则的创建。用户访问域名 `www.example.com` 且文件后缀为 `apk` 时，即可触发边缘函数进行动态打包。

! **说明：**

下一步请参考 [步骤三：测试并验证生效结果](#)。

步骤三：测试并验证生效结果

最近更新时间：2024-08-23 14:25:12

步骤1：验证云函数 SCF 是否已成功完成对 Android APK 母包的预处理

1. 登录 [对象存储 COS 控制台](#)，在左侧菜单栏中，单击存储桶列表。
2. 在存储桶列表页面，单击用于存储 APK 母包的存储桶名称。
3. 在文件列表页面，单击进入 `v2-vasdolly/` 文件目录，单击上传文件并选择 `.apk` 结尾的文件，如：`v2_src.apk`，单击上传。



4. 如云函数 SCF 已成功完成对 Android APK 母包的处理，则会在对象存储 COS 上传目录的同级目录下自动生成在一个文件输出的目录，具体路径为 [新建模板函数](#) 中，环境配置 `outputPath` 所填写的目录，例如：`/v2-vasdolly_output`，单击该目录的名称进入到目录，即可看到云函数 SCF 已预处理好的新的 APK 母包。



步骤2: 验证 EdgeOne 边缘函数是否已将渠道信息写入到 Android APK 包

在浏览器中输入带有渠道信息的 URL，例如：`http://www.example.com/v2_src.apk?comment=test` 即可触发边缘函数，动态地将渠道信息注入到指定位置，其中 `comment` 为您在 [创建用于触发渠道信息写入的边缘函数](#) 中定义的渠道参数。以 `v2-VasDolly` 方式为例，可以使用 `VasDolly` 工具来读取动态注入的渠道信息：

```
→ jar git:(master) × java -jar VasDolly_3.0.4.jar get -c ./v2_test.apk
try to read channel info from apk : ./v2_test.apk
find V2 signature block Id : 1896449818
getBytesValueById , destApk [pos=0 lim=658 cap=658], -2012129793=java.nio.HeapByteBuffer[pos=0 lim=10240 cap=10240]
getBytesValueById , id = -2012129793 , value = java.nio.HeapByteBuffer[pos=0 lim=10240 cap=10240]

Channel test-apk-edge-pack
```


通过边缘函数实现自适应图片格式转换

最近更新时间：2023-09-27 09:43:56

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

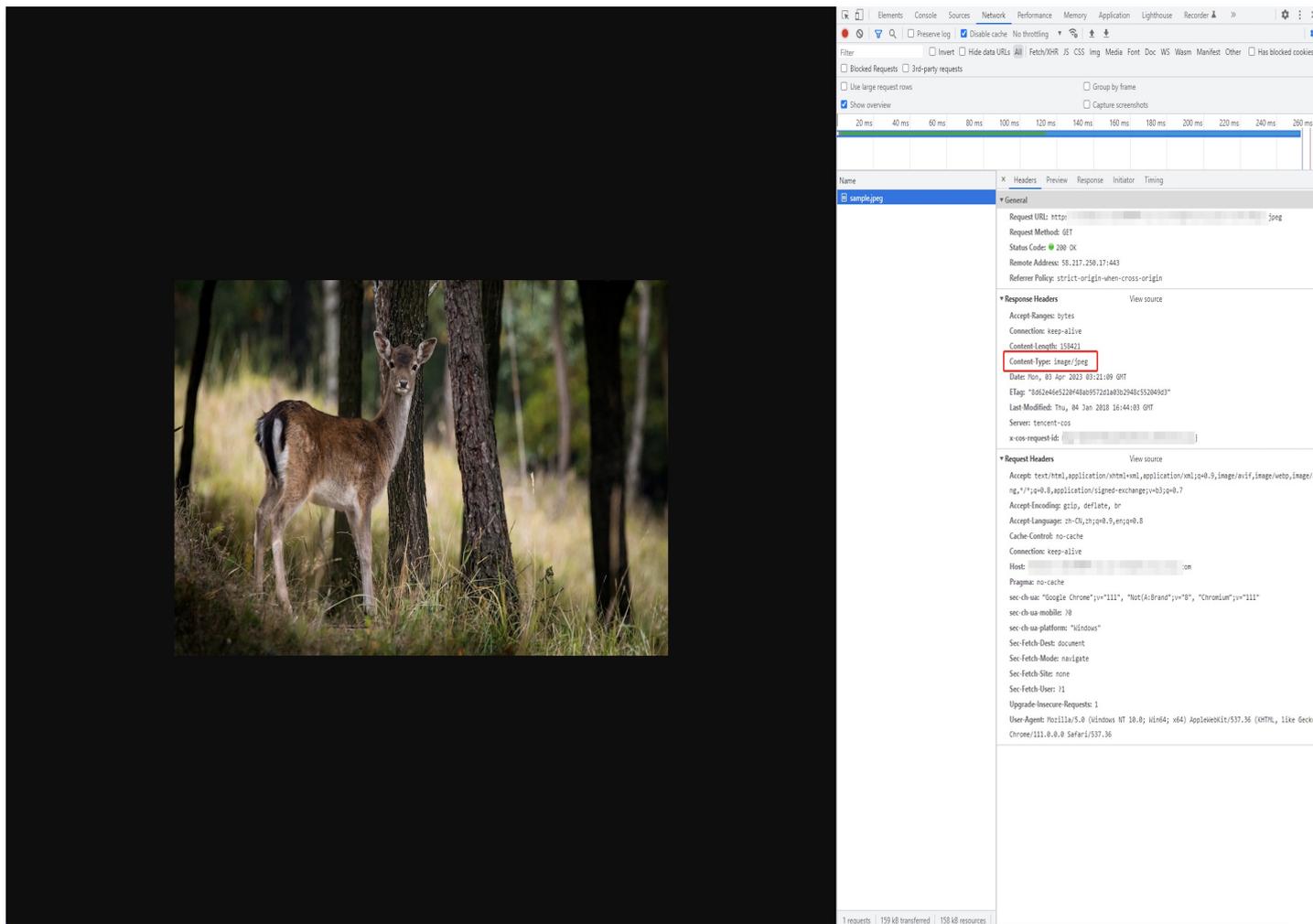
针对以大量图片内容为主的站点，例如：新闻传媒、电商平台、论坛等，图片文件的格式需根据浏览器的类型来进行适配，返回浏览器可兼容的图片格式，同时最大程度上压缩图片的大小，来节省流量。例如：

- 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 browserFormat = {
  Chrome: 'webp',
  Opera: 'webp',
  Firefox: 'webp',
  Safari: 'jpg',
  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。



- 单击**保存**触发规则即可生效。
- 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

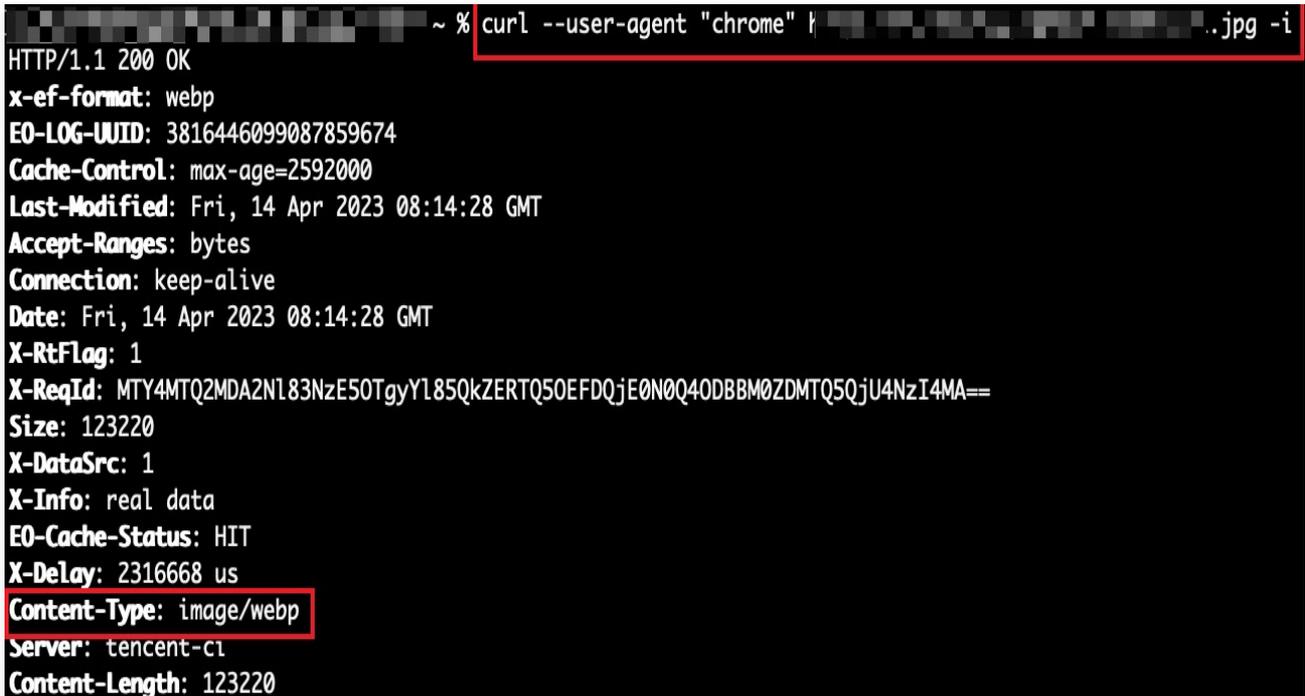
您可以通过 curl 请求中携带指定的 User-Agent 进行测试。

测试 Chrome 等浏览器

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
```

测试 Safari 浏览器

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" https://image.example.com/image/test.jpg -i
HTTP/1.1 200 OK
x-ef-format: jp2
EO-LOG-WUID: 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: MTY4MTQ2MTc0NV83NzE5OTgyYl8zRjRCODlEODM2NDg0RTEyQTJGRTYyNTZDODI1MkEyMg==
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
```

测试 IE 浏览器

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" https://image.example.com/image/test.jpg -i
HTTP/1.1 200 OK
x-ef-format: jxr
EO-LOG-WUID: 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: MTY4MTQ2MjE5MF83NzE5OTgyYl80MDU1MjdGMjJDMTQ0RjEwODlBNjRFREBFMjUzMkVwNQ==
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
```

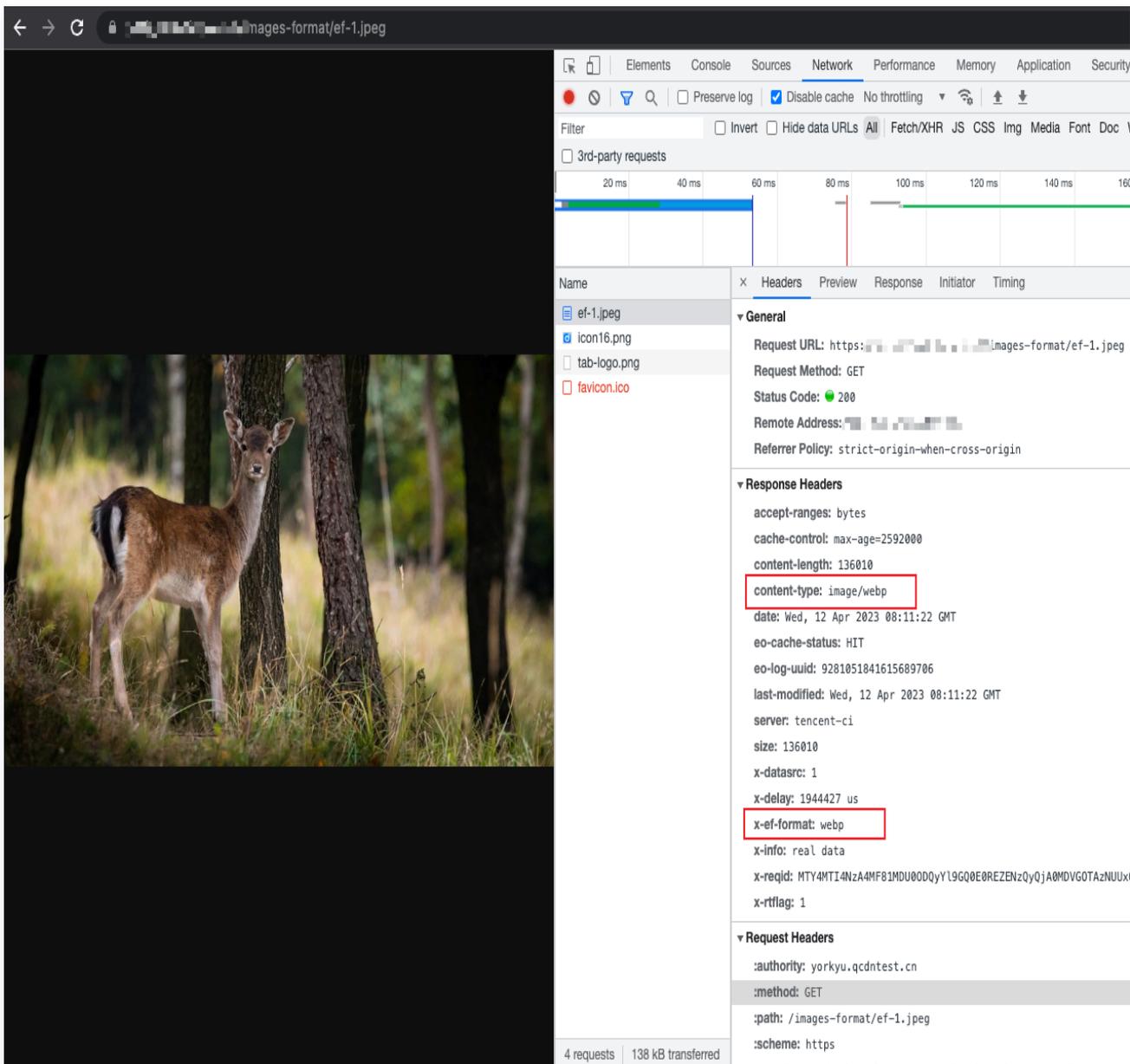
浏览器访问测试

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。

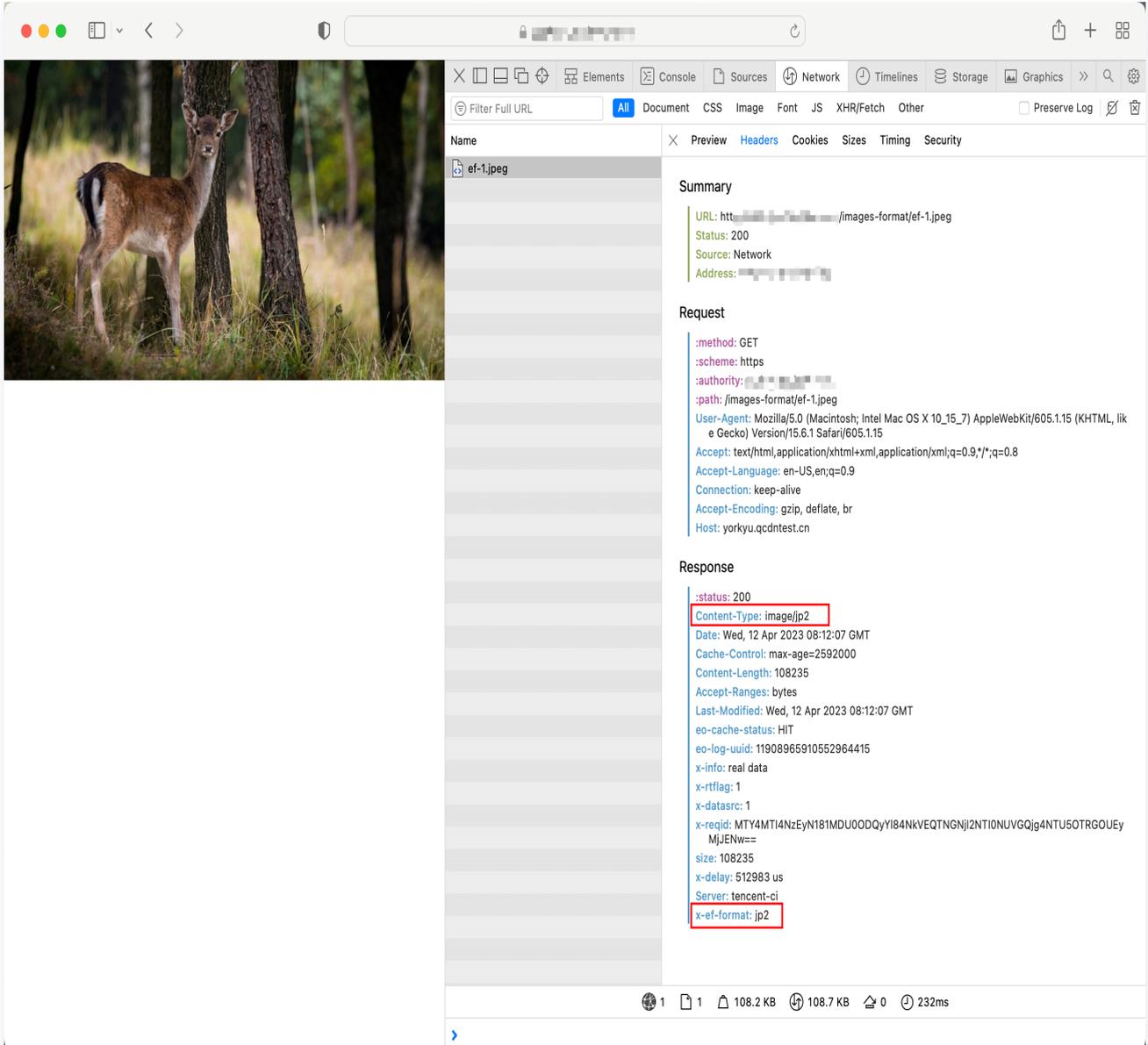
测试 Chrome 等浏览器

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。



测试 Safari 浏览器

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。



测试 IE 浏览器

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 web browser displaying a photograph of a deer in a forest. Below the image, the browser's developer tools are open to the Network tab, showing the details for a request to 'ef-1.jpeg'. The 'Headers' sub-tab is selected, and several headers are visible, with two highlighted by red boxes: 'content-type: image/vnd.ms-photo' and 'x-ef-format: jxr'.

| Name / Path | Protocol | Method | Headers |
|---|----------|--------|--|
| ef-1.jpeg https://[redacted]images-format/ | HTTP/2 | GET | <ul style="list-style-type: none"> 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](#)