Red Hat build of Node.js 20

开始用于红帽构建的 Node.js 的断路器添加的断路器

在 Node.js 应用中使用断路器附加组件

Last Updated: 2024-05-03

Red Hat build of Node.js 20 开始用于红帽构建的 Node.js 的断路器添加的断路器

在 Node.js 应用中使用断路器附加组件

法律通告

摘要

红帽构建的 Node.js 的断路器附加组件是提供断路器功能的开源 Opossum 模块完全支持的实施。您可以使用 Node.js 应用中的断路器附加组件来确保应用程序故障被监控并适当处理。在使用断路器附加组件时，您还可以定义回退功能，以便在应用失败时执行指定操作。您还可以定义事件处理程序，以侦听不同类型的断路器事件。

目录

第 1 章 红帽构建的 NODE.JS 的断路器附加组件简介

红帽为红帽构建的 Node.js 提供断路器附加组件，这是开源 Opossum 模块完全支持的实施。Opossum 是 NodeShift 项目团队开发的 Node.js 模块，提供断路器功能。您可以使用 Node.js 应用中的断路器附加组件来确保应用程序故障被监控并适当处理。

注意

在本文档的后续部分中，术语"断路器"是指红帽构建的 Node.js 的断路器附加组件。

1.1. 在微服务架构中处理应用程序故障的重要性

断路器附加组件有助于减少对服务架构故障的影响，服务异步调用其他服务。

在以下情况下可能会出现故障：

- 如果因为延迟问题和通常高的流量卷，服务无法及时响应请求
- 如果因为网络连接问题临时无法使用服务

在这些情况下，其他服务可以通过尝试联系太慢或无法响应的服务来耗尽关键资源。这些其他服务则变得不可用，这会导致影响整个微服务架构的级联故障。通过在 Node.js 应用程序代码中使用断路器附加组件，您可以使应用更具备容错能力和弹性。

1.2. 断路器设计模式

断路器模式旨在处理服务架构中故障，并防止跨多个系统的级联故障。断路器可以充当客户端功能的代理，该函数将请求发送到可能失败的远程服务端点。您可以在断路器对象中嵌套发送到远程服务端点的任何请求。然后，断路器调用远程服务并监控其状态。如果失败次数达到指定百分比阈值，则会触发断路器。然后，断路器确保对远程服务的任何进一步请求都自动阻止，并在指定时间段内出现错误消息或回退响应。

根据断路器设计模式，断路器在任何特定时间都具有以下状态之一：

关闭

失败的数量低于指定的百分比阈值，可触发断路器。您可以将百分比阈值定义为传递给 Node.js 应用中断路器的可选参数。默认百分比阈值为 50%。当失败数量较低且断路器处于关闭状态时，客户端功能可以继续向远程服务端点发送请求。

Open

失败的数量已达到指定百分比阈值，该阈值会触发断路器。例如，根据默认选项，如果 50% 的请求失败，则断路器进入 开放 状态。然后，断路器会在指定的超时时间内阻止对远程服务端点的所有调用，并显示错误消息或回退响应。您可以将超时周期定义为传递给 Node.js 应用中断路器的可选参数。默认超时时间为 30 秒。

半打开

指定的超时时间已结束。例如，基于默认选项，超时期限结束，断路器在 30 秒后进入 半开状态。然后，断路器尝试向远程服务端点发送有限数量的测试调用。如果测试调用成功，断路器进入 关闭状态，客户端功能可以继续向远程服务发送请求。但是，如果测试调用失败，断路器恢复到 开放 状态，并且继续阻止指定超时期间对远程服务的所有调用。

第 2 章 从红帽客户 REGISTRY 安装断路器附加组件

红帽构建的 Node.js 的断路器附加组件是开源 Opossum 模块完全支持的实施。红帽将断路器附加组件作为 @redhat/opossum 模块提供。您可以从红帽客户 registry 下载并安装 @redhat/opossum 模块。

流程

1. 在命令行中，访问 Node.js 应用的根目录。

2. 要指定安装 Node.js 模块的下载路径，请完成以下步骤：

   a. 在应用程序的根目录中，创建名为 .npmrc 的文件。

   b. 要指定红帽客户 registry 的路径，请在 .npmrc 文件中输入以下行：

   ```
   @redhat:registry=https://npm.registry.redhat.com
   ```

   c. 要指定 npm registry 的路径，请在 .npmrc 文件中输入以下行：

   ```
   registry=https://registry.npmjs.org
   ```

   d. 保存 .npmrc 文件。

3. 要下载并安装 @redhat/opossum 模块，请输入以下命令：

   ```
   $ npm install @redhat/opossum
   ```

验证

1. 如果您的项目中已存在 node_modules 目录，则此目录中会自动安装 @redhat/opossum 模块。

2. 如果项目中不存在 node_modules 目录，则 Node.js 应用程序的根目录中会自动创建 node_modules 子目录，并且此子目录中会自动安装 @redhat/opossum 模块。

其他资源

- 红帽客户 **registry** 位于 https://npm.registry.redhat.com。

第 3 章 使用断路器附加组件的示例应用

您可以使用断路器附加组件在 Node.js 应用中实施断路器模式。本例演示了如何使用断路器附加组件报告远程服务的故障，并限制对失败服务的访问，直到它变为可用来处理请求。

3.1. 示例应用程序概述

这个示例应用由两个微服务组成：

greeting-service

这是指向应用程序的入口点。Web 客户端调用 greeting-service 来请求问候语。然后，问候服务向远程 name-service 发送一个嵌套在断路器对象中的请求。

name-service

name-service 从 greeting-service 接收请求。Web 客户端界面包含一个切换按钮，您可以单击该按钮来模拟远程名称服务的可用性或故障。如果切换按钮目前在 上 设置为，name-service 会发送响应以完成问候语。但是，如果切换按钮目前设置为 off，name-service 会发送错误以指示该服务当前不可用。

3.2. GREETING-SERVICE 示例

greeting-service 导入断路器附加组件。greeting -service 通过将这些调用嵌套在断路器对象中来保护对远程名称服务 的调用。

greeting-service 会公开以下端点：

- /api/greeting 端点从请求问候的 Web 客户端接收调用。作为处理客户端请求的一部分，/api/greeting 端点会向远程 name-service 中的 /api/name 端点发送调用。对 /api/name 端点的调用被嵌套在断路器对象中。如果远程 name-service 当前可用，/api/greeting 端点会向客户端发送问候端点。但是，如果远程 name-service 当前不可用，/api/greeting 端点会向客户端发送错误响应。

- /api/cb-state 端点返回断路器的当前状态。如果设置为 open，则断路器目前阻止请求到达失败的服务。如果此值设为 closed，则断路器目前允许请求访问服务。

以下代码示例演示了如何开发 greeting-service ：

```
'use strict';
const path = require('path');
const http = require('http');
const express = require('express');
const bodyParser = require('body-parser');

// Import the circuit breaker add-on
const Opossum = require('@redhat/opossum');

const probe = require('kube-probe');
const nameService = require('./lib/name-service-client');

const app = express();
const server = http.createServer(app);

// Add basic health check endpoints
probe(app);

const nameServiceHost = process.env.NAME_SERVICE_HOST || 'http://nodejs-circuit-breaker-
redhat-name:8080';

// Set some circuit breaker options
const circuitOptions = {
  timeout: 3000, // If name service takes longer than 0.3 seconds,
                 // trigger a failure
  errorThresholdPercentage: 50, // When 50% of requests fail,
                                // trip the circuit
  resetTimeout: 10000 // After 10 seconds, try again.
};

// Create a new circuit breaker instance and pass the remote nameService
// as its first parameter
const circuit = new Opossum(nameService, circuitOptions);

// Create the app with an initial websocket endpoint
require('./lib/web-socket')(server, circuit);

// Serve index.html from the file system
app.use(express.static(path.join(__dirname, 'public')));
// Expose the license.html at http[s]://[host]:[port]/licences/licenses.html
app.use('/licenses', express.static(path.join(__dirname, 'licenses')));

// Send and receive json
app.use(bodyParser.json());

// Greeting API
app.get('/api/greeting', (request, response) => {
  // Use the circuit breaker's fire method to execute the call
  // to the name service
  circuit.fire(`${nameServiceHost}/api/name`).then(name => {
    response.send({ content: `Hello, ${name}`, time: new Date() });
  }).catch(console.error);
});

// Circuit breaker state API
app.get('/api/cb-state', (request, response) => {
```

```
  response.send({ state: circuit.opened ? 'open' : 'closed' });
});

app.get('/api/name-service-host', (request, response) => {
  response.send({ host: nameServiceHost });
});

module.exports = server;
```

3.3. NAME-SERVICE 示例

name-service 公开以下端点：

- /api/name 端点从 greeting-service 接收调用。如果 name-service 当前可用，/api/name 端点会发送 World! 响应以完成问候语。但是，如果 name-service 当前不可用，/api/name 端点会发送 Name service down 错误，其 HTTP 状态代码为 500。

- /api/state 端点控制 /api/name 端点的当前行为。它决定服务发送响应还是失败并显示错误消息。

以下代码示例演示了如何开发 name-service ：

```
'use strict';

const path = require('path');
const http = require('http');
const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');
const probe = require('kube-probe');

const app = express();
const server = http.createServer(app);

// Adds basic health-check endpoints
probe(app);

let isOn = true;
const { update, sendMessage } = require('./lib/web-socket')(server, _ => isOn);

// Send and receive json
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));

// CORS support
app.use(cors());
```

```
// Name service API
app.get('/api/name', (request, response) => {
  if (isOn) {
    response.send('World!');
  } else {
    response.status(500).send('Name service down');
  }
  sendMessage(`${new Date()} ${isOn ? 'OK' : 'FAIL'}`);
});

// Current state of service
app.put('/api/state', (request, response) => {
  isOn = request.body.state === 'ok';
  response.send({ state: isOn });
  update();
});

app.get('/api/info',
  (request, response) => response.send({ state: isOn ? 'ok' : 'fail' }));

// Expose the license.html at http[s]://[host]:[port]/licenses/licenses.html
app.use('/licenses', express.static(path.join(__dirname, 'licenses')));

module.exports = server;
```

其他资源

- 有关部署和运行示例断路器应用的更多信息，请参阅 Node.js 运行时指南。
- 有关可用于断路器附加组件的成员类型的更多信息，请参阅 Circuit Breaker Add-on 5.0.0 API 文档。

第 4 章 使用带有断路器附加组件的回退函数的示例应用

您可以定义可与 Node.js 应用中的断路器附加组件搭配使用的回退功能。fallback 功能指定在调用远程服务失败时执行的操作。例如，您可以使用回退功能向客户端发送关于服务故障的自定义消息。当断路器具有 开放状态时，每次尝试联系远程服务时，将持续执行回退功能。

4.1. 由断路器附加组件提供的回退方法

您可以使用断路器对象的 fallback 方法，将定义的 fallback 功能添加到断路器中。如果在执行断路器触发 方法后对远程服务的调用失败，则回退功能会自动调用。

有关可以与断路器附加组件一起使用的 fallback 方法和其他成员类型的更多信息，请参阅 Circuit Breaker Add-on 5.0.0 API 文档。

4.2. 定义回退功能的应用程序代码示例

本例基于 greeting-service，它向可能失败的远程 name-service 发送调用。本例演示了如何定义将以下消息输出到 Web 控制台的回退功能： This is fallback function。

```
...

// Import the circuit breaker add-on
const Opossum = require('@redhat/opossum');

...

// Create a new circuit breaker instance and pass the
// remote nameService as its first parameter
const circuit = new Opossum(nameService, circuitOptions);

// Define a fallback function that will be called when
// the remote nameService fails
function fallback(result) {
  console.log('This is the fallback function', result);
}

// Use the circuit breaker's fallback method to add the
// fallback function to the circuit breaker instance
circuit.fallback(fallback);

...

// Greeting API
app.get('/api/greeting', (request, response) => {
  // Use the circuit breaker's fire method to execute the call
  // to the name service
```

```
  circuit.fire(`${nameServiceHost}/api/name`).then(name => {
    response.send({ content: `Hello, ${name}`, time: new Date() });
  }).catch(console.error);
});

...
```

其他资源

- 有关部署和运行示例 cicuit breaker 应用的更多信息，请参阅 Node.js 运行时指南。
- 有关可用于断路器附加组件的成员类型的更多信息，请参阅 Circuit Breaker Add-on 5.0.0 API 文档。

第 5 章 断路器附加组件发出的事件处理

断路器附加组件根据断路器模式的可更改状态发出不同类型的操作的事件。您可以在 Node.js 应用中实施事件处理，以处理这些不同类型的事件，并在发生时执行一些操作。通过使用事件处理程序，您可以控制应用如何响应断路器的当前行为。

5.1. 断路器附加组件发出的事件类型

断路器附加组件发出以下类型的事件：

fire

当执行断路器 触发 方法来调用远程服务时，将发出此事件。

拒绝

当断路器为 open 或 half-open 状态时会发出此事件。

timeout

当断路器操作的超时时间到期时，会发出此事件。

success

当断路器操作成功完成时，会发出此事件。

失败

当断路器操作失败并且断路器返回错误响应时，将发出此事件。

open

当断路器进入 开放 状态时，将发出此事件。

关闭

当断路器进入关闭状态时，会发出此事件。

halfOpen

当断路器进入 半开状态 时，将发出此事件。

fallback

当断路器具有在调用远程服务失败后执行的回退函数时，会发出此事件。

semaphoreLocked

当断路器处于全部容量且无法执行请求时，会发出此事件。

healthCheckFailed

当用户提供的健康检查功能返回被拒绝的承诺时，会发出此事件。

5.2. 使用 CIRCUIT BREAKER 事件的事件处理程序的示例应用

本例演示了如何侦听断路器发出的不同事件类型。在本例中，使用 fetch 函数来请求微服务 URL。根据事件类型，相关消息会输出到 web 控制台。

```
...

// Import the circuit breaker add-on
const Opossum = require('@redhat/opossum');

...

// Create a new circuit breaker instance and pass the
// protected function call as its first parameter
const circuit = new Opossum(() => $.get(route), circuitOptions);

...

// Listen for success events emitted when the
// circuit breaker action completes successfully
circuit.on('success',
  (result) => {
    console.log(`SUCCESS: ${JSON.stringify(result)}`);
  }

// Listen for timeout events emitted when the timeout
// period for the circuit breaker action expires
circuit.on('timeout',
  (result) => {
    console.log(`TIMEOUT: ${url} is taking too long to respond.`);
  }

// Listen for reject events emitted when the
// circuit breaker has an open or half-open state
circuit.on('reject',
  (result) => {
    console.log(`REJECTED: The breaker for ${url} is open. Failing fast.`);
  }

// Listen for open events emitted when the
```

```
// circuit breaker moves to an open state
circuit.on('open',
  (result) => {
    console.log(`OPEN: The breaker for ${url} just opened.`);
  }

// Listen for halfOpen events emitted when the
// circuit breaker moves to a half-open state
circuit.on('halfOpen',
  (result) => {
    console.log(`HALF_OPEN: The breaker for ${url} is half open.`);
  }

// Listen for close events emitted when the
// circuit breaker moves to a closed state
circuit.on('close',
  (result) => {
    console.log(`CLOSE: The breaker for ${url} has closed. Service OK.`);
  }

// Listen for fallback events emitted when
// a fallback function is executed
circuit.on('fallback',
  (result) => {
    console.log(`FALLBACK: ${JSON.stringify(data)}`);
  }
```

其他资源

- 有关可用于断路器附加组件的成员类型的更多信息，请参阅 Circuit Breaker Add-on 5.0.0 API 文档。