Red Hat AMQ 2021.Q3

使用 AMQ JavaScript 客户端

用于 AMQ 客户端 2.10

Last Updated: 2023-02-20

Red Hat AMQ 2021.Q3 使用 AMQ JavaScript 客户端

用于 AMQ 客户端 2.10

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

法律通告

摘要

本指南描述了如何安装和配置客户端，运行实践示例，并将您的客户端与其他 AMQ 组件搭配使用。

目录

使开源包含更多

红帽承诺替换我们的代码、文档和网页属性中存在问题的语言。我们从这四个术语开始： master、slave、blacklist 和 whitelist。这些更改将在即将发行的几个发行本中逐渐实施。详情请查看 CTO Chris Wright 信息。

第 1 章 概述

AMQ JavaScript 是用于开发消息传递应用的库。它允许您编写发送和接收 AMQP 消息的 JavaScript 应用。

AMQ JavaScript 是 AMQ 客户端的一部分，这是一套支持多种语言和平台的消息传递库。有关客户端的概述，请参阅 AMQ 客户端概述。有关此发行版本的详情，请参考 AMQ Clients 2.10 发行注记。

AMQ JavaScript 基于 Rhea 消息传递库。有关详细的 API 文档，请参阅 AMQ JavaScript API 参考。

1.1. 主要特性

- 简化与现有应用程序的集成的事件驱动的 API
- 用于安全通信的 SSL/TLS
- 灵活的 SASL 身份验证
- 自动重新连接和故障转移
- AMQP 和原生语言数据类型之间的无缝转换
- 访问 AMQP 1.0 的所有特性和功能

1.2. 支持的标准和协议

AMQ JavaScript 支持以下业界认可的标准和网络协议：

- 高级消息队列协议(AMQP)版本 1.0
- 传输层安全(TLS)协议的版本 1.0、1.1、1.2 和 1.3，后跟 SSL
- 简单身份验证和安全层(SASL)机制 ANONYMOUS、PLAIN 和 EXTERNAL
- 使用 IPv6的现代 TCP

1.3. 支持的配置

有关 AMQ JavaScript 支持的配置，请参阅红帽客户门户网站中的 Red Hat AMQ 7 支持的配置。

1.4. 术语和概念

本节介绍核心 API 实体，并描述它们如何协同运作。

表 1.1. API 术语

实体	描述
Container	连接的顶级容器。
连接	个网络上两个同级之间的通信通道。它包含会话。

实体	描述
会话	用于发送和接收消息的上下文。它包含发送方和接收方。
sender	用于将消息发送到目标的频道。它有一个目标。
receiver	从源接收信息的频道。它有一个源。
Source	消息的指定来源点。
目标	消息的指定目的地。
消息	特定于应用的信息。
交付	消息传输。

AMQ JavaScript 发送和接收 *消息*。消息通过 *发送方和接收方在连接的对等点 之*间传输 。通过 *会话* 创建发件人和接收方。通过 *连接* 建立会话。连接在两个唯一标识 *的容器 之*间建立。虽然连接可以有多个会话，但通常不需要。API 允许您忽略会话，除非您需要它们。

发送对等点会创建一个发送者来发送消息。发送方具有在远程同级上标识队列或主题 *的目标*。接收方创建接收方来接收消息。接收方具有一个 *源*，用于标识远程对等点上的队列或主题。

消息的发送称为 *发送*。消息是发送的内容，包括标头和注释等所有元数据。交付是指与该内容的传输相关的协议交换。

为了表示某一交付已完成，发件人或接收方都可处理该交付。当另一边了解到它已被实施时，将不会再交流该交付。接收方也可以指示它接受还是拒绝消息。

1.5. 文档惯例

sudo 命令

在本文档中，**sudo** 用于任何需要 root 权限的命令。使用 **sudo** 时要小心，因为任何更改都可能会影响整个系统。有关 **sudo** 的详情请参考 使用 sudo 命令。

文件路径

在这个文档中，所有文件路径都对 Linux、UNIX 和类似操作系统有效（例如 **/home/andrea**）。在 Microsoft Windows 中，您必须使用等效的 Windows 路径（例如 **C:\Users\andrea**）。

变量文本

本文档包含代码块，它们需要使用特定于环境的值替换。变量文本括在箭头大括号内，样式为圆形单空间。例如，在以下命令中，将 ***<project-dir>*** 替换为您的环境的值：

```
$ cd <project-dir>
```

第 2 章 安装

本章指导您完成在您的环境中安装 AMQ JavaScript 的步骤。

2.1. 先决条件

- 您必须有 订阅 才能访问 AMQ 发行文件和存储库。
- 要使用 AMQ JavaScript，您必须在您的环境中安装 Node.js。如需更多信息，请参阅 Node.js 网站。
- AMQ JavaScript 依赖于 Node.js **debug** 模块。有关安装说明请查看 npm 页面。

2.2. 在 RED HAT ENTERPRISE LINUX 上安装

流程

1. 打开浏览器并登录红帽客户门户网站 **产品下载页面**，网址为 access.redhat.com/downloads。
2. 在 **INTEGRATION AND AUTOMATION** 类别中找到 **红帽 AMQ 客户端** 条目。
3. 单击 **Red Hat AMQ Clients**。此时会打开 **Software Downloads** 页面。
4. 下载 **AMQ 客户端 2.10.0 JavaScript**.zip 文件。
5. 使用 **unzip** 命令将文件内容提取到您选择的目录中。

```
$ unzip amq-clients-2.10.0-javascript.zip
```

当您提取 .zip 文件的内容时，将创建一个名为 **amq-clients-2.10.0-javascript** 的目录。这是安装的顶级目录，在整个文档中被称为 ***<install-dir>***。

要将环境配置为使用安装的库，请将 **node_modules** 目录添加到 **NODE_PATH** 环境变量中。

```
$ cd amq-clients-2.10.0-javascript
$ export NODE_PATH=$PWD/node_modules:$NODE_PATH
```

要使此配置对所有新控制台会话生效，请在 **$HOME/.bashrc** 文件中设置 **NODE_PATH**。

要测试您的安装，请使用以下命令：如果成功导入安装的库，它会在控制台中打印 **OK**。

```
$ node -e 'require("rhea")' && echo OK
OK
```

2.3. 在 MICROSOFT WINDOWS 上安装

流程

1. 打开浏览器并登录红帽客户门户网站 **产品下载页面**，网址为 access.redhat.com/downloads。
2. 在 **INTEGRATION AND AUTOMATION** 类别中找到 **红帽 AMQ 客户端** 条目。

3. 单击 **Red Hat AMQ Clients**。此时会打开 **Software Downloads** 页面。
4. 下载 **AMQ 客户端 2.10.0 JavaScript**.zip 文件。
5. 通过右键单击 zip 文件并选择"**提取 所有"，将文件内容提取到** 您选择的目录。

当您提取 .zip 文件的内容时，将创建一个名为 **amq-clients-2.10.0-javascript** 的目录。这是安装的顶级目录，在整个文档中被称为 ***<install-dir>***。

要将环境配置为使用安装的库，请将 **node_modules** 目录添加到 **NODE_PATH** 环境变量中。

```
$ cd amq-clients-2.10.0-javascript
$ set NODE_PATH=%cd%\node_modules;%NODE_PATH%
```

2.4. 准备在浏览器中使用的库

AMQ JavaScript 可以在 Web 浏览器中运行。要创建与浏览器兼容的库版本，请使用 **npm run browserify** 命令。

```
$ cd amq-clients-2.10.0-javascript/node_modules/rhea
$ npm install
$ npm run browserify
```

这会生成一个名为 **rhea.js** 的文件，该文件可在基于浏览器的应用程序中使用。

第 3 章 入门

本章将引导您完成设置环境并运行简单消息传递程序的步骤。

3.1. 先决条件

- 您必须为您的环境完成 安装过程。
- 您必须有一个 AMQP 1.0 消息代理，侦听接口 **localhost** 和端口 **5672** 上的连接。它必须启用匿名访问。如需更多信息，请参阅 启动代理。
- 您必须有一个名为 **examples** 的队列。如需更多信息，请参阅 创建队列。

3.2. 在 RED HAT ENTERPRISE LINUX 上运行 HELLO WORLD

Hello World 示例创建与代理的连接，发送一条包含问候语的消息到 **examples** 队列，然后重新接收它。成功后，它会将收到的消息打印到控制台。

更改到示例目录并运行 **helloworld.js** 示例。

```
$ cd <install-dir>/node_modules/rhea/examples
$ node helloworld.js
Hello World!
```

3.3. 在 MICROSOFT WINDOWS 上运行 HELLO WORLD

Hello World 示例创建与代理的连接，发送一条包含问候语的消息到 **examples** 队列，然后重新接收它。成功后，它会将收到的消息打印到控制台。

更改到示例目录并运行 **helloworld.js** 示例。

```
> cd <install-dir>/node_modules/rhea/examples
> node helloworld.js
Hello World!
```

第 4 章 示例

本章介绍如何通过示例程序使用 AMQ JavaScript。

有关更多示例，请参阅 AMQ JavaScript 示例套件 和 Rhea 示例。

4.1. 发送消息

这个客户端程序使用 **<connection-url>** 连接到服务器，为目标 **<address>** 创建一个发送程序，发送一条包含 **<message-body>** 的消息，关闭连接，然后退出。

示例：发送消息

```
"use strict";

var rhea = require("rhea");
var url = require("url");

if (process.argv.length !== 5) {
    console.error("Usage: send.js <connection-url> <address> <message-body>");
    process.exit(1);
}

var conn_url = url.parse(process.argv[2]);
var address = process.argv[3];
var message_body = process.argv[4];

var container = rhea.create_container();

container.on("sender_open", function (event) {
    console.log("SEND: Opened sender for target address '" +
                event.sender.target.address + "'");
});

container.on("sendable", function (event) {
    var message = {
        body: message_body
    };

    event.sender.send(message);

    console.log("SEND: Sent message '" + message.body + "'");

    event.sender.close();
    event.connection.close();
});

var opts = {
    host: conn_url.hostname,
    port: conn_url.port || 5672,
    // To connect with a user and password:
    // username: "<username>",
    // password: "<password>",
};
```

```
var conn = container.connect(opts);
conn.open_sender(address);
```

运行示例
要运行示例程序，将其复制到本地文件中并使用 **node** 命令调用它。如需更多信息，请参阅 第 3 章 入门。

```
$ node send.js amqp://localhost queue1 hello
```

4.2. 接收消息

这个客户端程序使用 **<connection-url>** 连接到服务器，为源 **<address>** 创建一个接收器，并在其终止或到达 **<count>** 信息前接收信息。

示例：接收消息

```
"use strict";

var rhea = require("rhea");
var url = require("url");

if (process.argv.length !== 4 && process.argv.length !== 5) {
    console.error("Usage: receive.js <connection-url> <address> [<message-count>]");
    process.exit(1);
}

var conn_url = url.parse(process.argv[2]);
var address = process.argv[3];
var desired = 0;
var received = 0;

if (process.argv.length === 5) {
    desired = parseInt(process.argv[4]);
}

var container = rhea.create_container();

container.on("receiver_open", function (event) {
    console.log("RECEIVE: Opened receiver for source address '" +
                event.receiver.source.address + "'");
});

container.on("message", function (event) {
    var message = event.message;

    console.log("RECEIVE: Received message '" + message.body + "'");

    received++;

    if (received == desired) {
        event.receiver.close();
        event.connection.close();
    }
```

```
});

var opts = {
    host: conn_url.hostname,
    port: conn_url.port || 5672,
    // To connect with a user and password:
    // username: "<username>",
    // password: "<password>",
};

var conn = container.connect(opts);
conn.open_receiver(address);
```

运行示例

要运行示例程序，将其复制到本地文件中并使用 **python** 命令调用它。如需更多信息，请参阅 第 3 章 入门。

```
$ node receive.js amqp://localhost queue1
```

第 5 章 使用 API

如需更多信息，请参阅 AMQ JavaScript API 参考 和 AMQ JavaScript 示例套件。

5.1. 处理消息传递事件

AMQ JavaScript 是异步事件驱动 API。要定义应用程序如何处理事件，用户在 **container** 对象中注册事件处理功能。这些功能随后称为网络活动或计时器触发新事件。

示例：处理消息传递事件

```
var rhea = require("rhea");
var container = rhea.create_container();

container.on("sendable", function (event) {
    console.log("A message can be sent");
});

container.on("message", function (event) {
    console.log("A message is received");
});
```

这些只是几个常见案例事件。完整集合记录在 AMQ JavaScript API 参考 中。

5.2. 访问事件相关的对象

event 参数具有用于访问事件相关对象的属性。例如，**connection_open** 事件设置事件 **connection** 属性。

除了事件的主要对象外，也设置组成事件上下文的所有对象。与特定事件无关的属性为 null。

示例：访问事件相关的对象

```
event.container
event.connection
event.session
event.sender
event.receiver
event.delivery
event.message
```

5.3. 创建容器

容器是顶级 API 对象。它是创建连接的入口点，它负责运行主事件循环。它通常通过全局事件处理程序构建。

示例：创建容器

```
var rhea = require("rhea");
var container = rhea.create_container();
```

5.4. 设置容器身份

每个容器实例都有一个名为容器 ID 的唯一身份。当 AMQ JavaScript 进行网络连接时，它会将容器 ID 发送到远程对等点。要设置容器 ID，将 **id** 选项传递给 **create_container** 方法。

示例：设置容器身份

```
var container = rhea.create_container({id: "job-processor-3"});
```

如果用户没有设置 ID，则库将在容器精简时生成 UUID。

第 6 章 网络连接

6.1. 创建传出连接

要连接到远程服务器，将包含主机和端口的连接选项传递给 **container.connect()** 方法。

示例：创建出站连接

```
container.on("connection_open", function (event) {
    console.log("Connection " + event.connection + " is open");
});

var opts = {
    host: "example.com",
    port: 5672
};

container.connect(opts);
```

默认主机是 **localhost**。默认端口为 5672。

有关创建安全连接的详情，第 7 章 *安全性*。

6.2. 配置重新连接

重新连接允许客户端从丢失的连接中恢复。它用于确保分布式系统中的组件在临时网络或组件失败后重新建立通信。

默认情况下，AMQ JavaScript 启用重新连接。如果连接尝试失败，客户端会在短暂的延迟后再次尝试。每次新尝试的延迟都呈指数级增长，默认为 60 秒。

要禁用重新连接，将 **reconnect** 连接选项设置为 **false**。

示例：禁用重新连接

```
var opts = {
    host: "example.com",
    reconnect: false
};

container.connect(opts);
```

要控制连接尝试之间的延迟，请设置 **initial_reconnect_delay** 和 **max_reconnect_delay** 连接选项。延迟选项以毫秒为单位指定。

要限制重新连接尝试数量，请设置 **reconnect_limit** 选项。

示例：配置重新连接

```
var opts = {
    host: "example.com",
    initial_reconnect_delay: 100,
    max_reconnect_delay: 60 * 1000,
```

```
    reconnect_limit: 10
};

container.connect(opts);
```

6.3. 配置故障切换

AMQ JavaScript 允许您以编程方式配置备用连接端点。

要指定多个连接端点，定义返回新连接选项并在 **connection_details** 选项中传递函数的功能。每次尝试连接时会调用一次函数。

示例：配置故障切换

```
var hosts = ["alpha.example.com", "beta.example.com"];
var index = -1;

function failover_fn() {
    index += 1;

    if (index == hosts.length) index = 0;

    return {host: hosts[index].hostname};
};

var opts = {
    host: "example.com",
    connection_details: failover_fn
}

container.connect(opts);
```

此示例为主机列表实施重复循环故障转移。您可以使用这个接口来实施您自己的故障切换行为。

6.4. 接受传入的连接

AMQ JavaScript 可以接受入站网络连接，使您能够构建自定义消息传递服务器。

要开始侦听连接，使用 **container.listen()** 方法以及包含要侦听的本地主机地址和端口的选项。

示例：接受进入的连接

```
container.on("connection_open", function (event) {
    console.log("New incoming connection " + event.connection);
});

var opts = {
    host: "0.0.0.0",
    port: 5672
};

container.listen(opts);
```

特殊 IP 地址 **0.0.0.0** 侦听所有可用的 IPv4 接口。要侦听所有 IPv6 接口，请使用 **[::0]**。

如需更多信息，请参阅 服务器接收.js 示例。

第 7 章 安全性

7.1. 使用 SSL/TLS 保护连接

AMQ JavaScript 使用 SSL/TLS 来加密客户端和服务器之间的通信。

要使用 SSL/TLS 连接到远程服务器，将 **transport** 连接选项设置为 **tls**。

示例：启用 SSL/TLS

```
var opts = {
  host: "example.com",
  port: 5671,
  transport: "tls"
};

container.connect(opts);
```

注意

默认情况下，客户端将拒绝与带有不可信证书的服务器的连接。测试环境中有时会出现这种情况。要绕过证书授权，将 **rejectUnauthorized** 连接选项设置为 **false**。请注意，这会破坏您的连接的安全性。

7.2. 使用用户和密码连接

AMQ JavaScript 可以验证与用户和密码的连接。

要指定用于身份验证的凭证，请设置 **username** 和 **password** 连接选项。

示例：使用用户和密码连接

```
var opts = {
  host: "example.com",
  username: "alice",
  password: "secret"
};

container.connect(opts);
```

7.3. 配置 SASL 身份验证

AMQ JavaScript 使用 SASL 协议来执行身份验证。SASL 可以使用多种不同的身份验证 *机制*。两个网络对等连接时，它们将交换允许的机制，同时选择两者允许的最强大机制。

AMQ JavaScript 根据用户和密码信息的存在性启用 SASL 机制。如果同时指定了用户名和密码，则会使用 **PLAIN**。如果只指定用户，则使用 **ANONYMOUS**。如果未指定，则禁用 SASL。

第 8 章 发送者和接收方

客户端使用发送方和接收器链接来表示用于发送消息的通道。发送者和接收方是单向的，消息来源的结尾，消息目的地的目标结束。

源和目标通常指向消息代理上的队列或主题。源也用于表示订阅。

8.1. 按需创建队列和主题

某些消息服务器支持按需创建队列和主题。连接了发送方或接收方时，服务器使用发送方目标地址或接收器源地址来创建名称与该地址匹配的队列或主题。

消息服务器通常默认为创建队列（用于一对一消息发送）或主题（一对多消息发送）。客户端可以通过在源或目标中设置 **queue** 或 **topic** 功能来指示它首选的内容。

要选择队列或主题语义，请按照以下步骤执行：

1. 配置您的消息服务器，以自动创建队列和主题。这通常是默认配置。
2. 在发送者目标或接收器源中设置 **queue** 或 **topic** 功能，如下例所示。

示例：发送到按需创建的队列

```
var conn = container.connect({host: "example.com"});

var sender_opts = {
    target: {
        address: "jobs",
        capabilities: ["queue"]
    }
}

conn.open_sender(sender_opts);
```

示例：从按需创建的主题接收

```
var conn = container.connect({host: "example.com"});

var receiver_opts = {
    source: {
        address: "notifications",
        capabilities: ["topic"]
    }
}

conn.open_receiver(receiver_opts);
```

如需了解更多详细信息，请参阅以下示例：

- queue-send.js
- queue-receive.js
- topic-send.js

- topic-receive.js

8.2. 创建持久订阅

持久订阅是远程服务器上代表消息接收器的一种状态。通常，当客户端关闭时，消息接收器会被丢弃。但是，由于持久订阅是永久的，客户端可以从它们分离，然后在以后重新连接。当客户端重新附加时，分离时收到的所有消息都可用。

持久订阅通过组合客户端容器 ID 和接收器名称组成订阅 ID 来唯一标识。这些必须具有稳定值，以便可以恢复订阅。

1. 将连接容器 ID 设置为 stable 值，如 **client-1**:

```
var container = rhea.create_container({id: "client-1"});
```

2. 使用稳定名称（如 **sub-1** ）创建接收器，并通过设置 **durable** 和 **expiry_policy** 属性配置接收器源以实现持久性：

```
var receiver_opts = {
   source: {
      address: "notifications",
      name: "sub-1",
      durable: 2,
      expiry_policy: "never"
   }
}

conn.open_receiver(receiver_opts);
```

要从订阅中分离，使用 **receiver.detach()** 方法。要终止订阅，使用 **receiver.close()** 方法。

如需更多信息，请参阅 persistent -subscribe.js 示例。

8.3. 创建共享订阅

共享订阅是远程服务器上代表一个或多个消息接收器的一种状态。由于它是共享的，所以多个客户端可以从同一消息流消耗。

客户端通过在接收器源上设置 **shared** 功能来配置共享订阅。

共享订阅通过组合客户端容器 ID 和接收器名称组成订阅 ID 来唯一标识。这些必须具有稳定值，以便多个客户端进程可以找到相同的订阅。如果除了 **shared** 外还设置了 **global** 能力，则只使用接收器名称来标识订阅。

要创建持久订阅，请按照以下步骤执行：

1. 将连接容器 ID 设置为 stable 值，如 **client-1**:

```
var container = rhea.create_container({id: "client-1"});
```

2. 使用稳定名称创建接收器，如 **sub-1**，并通过设置 **shared** 功能配置接收器源进行共享：

```
var receiver_opts = {
   source: {
```

```
        address: "notifications",
        name: "sub-1",
        capabilities: ["shared"]
    }
}

conn.open_receiver(receiver_opts);
```

要从订阅中分离，使用 **receiver.detach()** 方法。要终止订阅，使用 **receiver.close()** 方法。

如需更多信息，请参阅 shared-subscribe.js 示例。

第 9 章 错误处理

AMQ JavaScript 中的错误可以通过截获与 AMQP 协议或连接错误对应的指定事件来处理。

9.1. 处理连接和协议错误

您可以通过拦截以下事件来处理协议级别错误：

- **connection_error**
- **session_error**
- **sender_error**
- **receiver_error**
- **protocol_error**
- **error**

每当发生事件的特定对象存在错误条件时，都会触发这些事件。调用错误处理程序后，也会调用对应的 ***<object>*_close** 处理程序。

event 参数具有用于访问错误对象的 **error** 属性。

示例：处理错误

```
container.on("error", function (event) {
    console.log("An error!", event.error);
});
```

注意

由于在发生任何错误时会调用关闭的处理程序，因此只需要在错误处理程序内处理错误本身。资源清理可以通过关闭的处理程序进行管理。如果没有特定于特定对象的错误处理，则通常要处理常规 **error** 事件，但没有更具体的处理程序。

注意

当启用重新连接且远程服务器关闭与 **amqp:connection:forced** 条件的连接时，客户端不会将其视为错误，因此不会触发 **connection_error** 事件。客户端改为开始重新连接过程。

第 10 章 日志记录

10.1. 配置日志记录

AMQ JavaScript 使用 JavaScript 调试模块 来实施日志记录。

例如，要启用详细的客户端日志记录，将 **DEBUG** 环境变量设置为 **rhea*** ：

示例：启用详细的日志记录

```
$ export DEBUG=rhea*
$ <your-client-program>
```

10.2. 启用协议日志记录

客户端可以将 AMQP 协议框架记录到控制台。诊断问题时，这些数据通常至关重要。

要启用协议日志记录，将 **DEBUG** 环境变量设置为 **rhea:frames** ：

示例：启用协议日志记录

```
$ export DEBUG=rhea:frames
$ <your-client-program>
```

第 11 章 基于文件的配置

AMQ JavaScript 可以读取用于从名为 **connect.json** 的本地文件建立连接的配置选项。这可让您在部署时在应用程序中配置连接。

当应用调用容器 **connect** 方法时，库会尝试读取该文件，而不提供任何连接选项。

11.1. 文件位置

如果设置，AMQ JavaScript 使用 **MESSAGING_CONNECT_FILE** 环境变量的值来查找配置文件。

如果没有设置 **MESSAGING_CONNECT_FILE**，AMQ JavaScript 会按照所示的顺序搜索名为 **connect.json** 的文件。它会在遇到的第一个匹配项时停止。

在 Linux 中：

1. **$PWD/connect.json**其中 **$PWD** 是客户端进程的当前工作目录
2. **$HOME/.config/messaging/connect.json**，其中 **$HOME** 是当前用户主目录
3. **/etc/messaging/connect.json**

在 Windows 中：

1. **%cd%/connect.json**其中 **%cd%** 是客户端进程的当前工作目录

如果没有找到 **connect.json** 文件，程序库会为所有选项使用默认值。

11.2. 文件格式

connect.json 文件包含 JSON 数据，额外支持 JavaScript 注释。

所有配置属性都是可选的或具有默认值，因此一个简单的示例只需要提供几个详情：

示例：一个简单的 **connect.json** 文件

```
{
   "host": "example.com",
   "user": "alice",
   "password": "secret"
}
```

SASL 和 SSL/TLS 选项嵌套在 **"sasl"** 和 **"tls"** 命名空间下：

示例：带有 **SASL** 和 **SSL/TLS** 选项的 **connect.json** 文件

```
{
   "host": "example.com",
   "user": "ortega",
   "password": "secret",
   "sasl": {
      "mechanisms": ["SCRAM-SHA-1", "SCRAM-SHA-256"]
   },
   "tls": {
```

```
        "cert": "/home/ortega/cert.pem",
        "key": "/home/ortega/key.pem"
    }
}
```

11.3. 配置选项

选项键包含句点(.)代表嵌套在命名空间内的属性。

表 11.1. 中的配置选项 connect.json

键	值类型	默认值	描述
scheme	字符串	**"amqps"**	**"amqp"** 用于明文或 **"amqps"** 用于 SSL/TLS
host	字符串	**"localhost"**	远程主机的主机名或 IP 地址
port	字符串或数字	**"amqps"**	端口号或端口字面
user	字符串	*无*	用于身份验证的用户名
password	字符串	*无*	身份验证密码
sasl.mechanisms	列出或字符串	*none（系统默认）*	已启用 SASL 机制的 JSON 列表。裸机字符串表示一种机制。如果没有指定，客户端将使用系统提供的默认机制。
sasl.allow_insecure	布尔值	**false**	启用发送明文密码的机制
tls.cert	字符串	*无*	客户端证书的文件名或数据库 ID
tls.key	字符串	*无*	客户端证书私钥的文件名或数据库 ID
tls.ca	字符串	*无*	CA 证书的文件名、目录或数据库 ID
tls.verify	布尔值	**true**	需要一个带有匹配主机名的有效服务器证书

第 12 章 互操作性

本章讨论如何与其他 AMQ 组件结合使用 AMQ JavaScript。有关 AMQ 组件兼容性的概述，请参阅 产品简介。

12.1. 与其他 AMQP 客户端互操作

AMQP 消息通过 AMQP 类型系统来 构成。这种常见格式是不同语言的 AMQP 客户端能够相互互操作的原因之一。

发送消息时，AMQ JavaScript 会自动将语言原生类型转换为 AMQP 编码的数据。接收消息时，会进行反向转换。

注意

有关 AMQP 类型的更多信息，请参阅 Apache Qpid 项目维护 的交互式类型参考。

表 12.1. AMQP 类型

AMQP 类型	描述
null	一个空值
boolean	true 或 false 值
char	单个 Unicode 字符
string	Unicode 字符序列
binary	字节序列
byte	签名的 8 位整数
short	签名的 16 位整数
int	签名的 32 位整数
long	签名的 64 位整数
ubyte	未签名的 8 位整数
ushort	未签名的 16 位整数
uint	未签名 32 位整数
ulong	未签名 64 位整数
float	32 位浮动点数

AMQP 类型	描述
double	64 位浮动点数
array	单个类型值序列
list	变量类型的一系列值
map	从不同键到值的映射
uuid	通用唯一标识符
symbol	来自受限域的 7 位 ASCII 字符串
timestamp	绝对时间点

JavaScript 具有比 AMQP 可以编码的原生类型更少。要发送包含特定 AMQP 类型的消息，请使用 **rhea/types.js** 模块中的 **wrap_** 功能。

表 12.2. 编码之前和解码之后的 AMQ JavaScript 类型

AMQP 类型	编码前 AMQ JavaScript 类型	解码后 AMQ JavaScript 类型
null	**null**	**null**
boolean	**boolean**	**boolean**
char	**wrap_char(number)**	**number**
string	**string**	**string**
binary	**wrap_binary(string)**	**string**
byte	**wrap_byte(number)**	**number**
short	**wrap_short(number)**	**number**
int	**wrap_int(number)**	**number**
long	**wrap_long(number)**	**number**
ubyte	**wrap_ubyte(number)**	**number**
ushort	**wrap_ushort(number)**	**number**
uint	**wrap_uint(number)**	**number**

AMQP 类型	编码前 AMQ JavaScript 类型	解码后 AMQ JavaScript 类型
ulong	wrap_ulong(number)	number
float	wrap_float(number)	number
double	wrap_double(number)	number
array	wrap_array(Array, code)	Array
list	wrap_list(Array)	Array
map	wrap_map(object)	object
uuid	wrap_uuid(number)	number
symbol	wrap_symbol(string)	string
timestamp	wrap_timestamp(number)	number

表 12.3. AMQ JavaScript 和其他 AMQ 客户端类型（2 中的 1 个）

编码前 AMQ JavaScript 类型	AMQ C++ 类型	AMQ .NET 类型
null	nullptr	null
boolean	bool	System.Boolean
wrap_char(number)	wchar_t	System.Char
string	std::string	System.String
wrap_binary(string)	proton::binary	System.Byte[]
wrap_byte(number)	int8_t	System.SByte
wrap_short(number)	int16_t	System.Int16
wrap_int(number)	int32_t	System.Int32
wrap_long(number)	int64_t	System.Int64
wrap_ubyte(number)	uint8_t	System.Byte
wrap_ushort(number)	uint16_t	System.UInt16

编码前 AMQ JavaScript 类型	AMQ C++ 类型	AMQ .NET 类型
wrap_uint(number)	uint32_t	System.UInt32
wrap_ulong(number)	uint64_t	System.UInt64
wrap_float(number)	float	System.Single
wrap_double(number)	double	System.Double
wrap_array(Array, code)	-	-
wrap_list(Array)	std::vector	Amqp.List
wrap_map(object)	std::map	Amqp.Map
wrap_uuid(number)	proton::uuid	System.Guid
wrap_symbol(string)	proton::symbol	Amqp.Symbol
wrap_timestamp(number)	proton::timestamp	System.DateTime

表 12.4. AMQ JavaScript 和其他 AMQ 客户端类型（共 2 个）

编码前 AMQ JavaScript 类型	AMQ Python 类型	AMQ Ruby 类型
null	None	nil
boolean	bool	true, false
wrap_char(number)	unicode	String
string	unicode	String
wrap_binary(string)	bytes	String
wrap_byte(number)	int	Integer
wrap_short(number)	int	Integer
wrap_int(number)	long	Integer
wrap_long(number)	long	Integer
wrap_ubyte(number)	long	Integer

编码前 AMQ JavaScript 类型	AMQ Python 类型	AMQ Ruby 类型
wrap_ushort(number)	**long**	**Integer**
wrap_uint(number)	**long**	**Integer**
wrap_ulong(number)	**long**	**Integer**
wrap_float(number)	**float**	**Float**
wrap_double(number)	**float**	**Float**
wrap_array(Array, code)	**proton.Array**	**Array**
wrap_list(Array)	**list**	**Array**
wrap_map(object)	**dict**	**Hash**
wrap_uuid(number)	-	-
wrap_symbol(string)	**str**	**Symbol**
wrap_timestamp(number)	**long**	**Time**

12.2. 使用 AMQ JMS 进行互操作

AMQP 定义与 JMS 消息传递模型的标准映射。本节讨论该映射的方方面面。如需更多信息，请参阅 AMQ JMS Interoperability 一章。

JMS 消息类型

AMQ JavaScript 提供单一消息类型，其正文类型可能有所不同。相比之下，JMS API 使用不同的消息类型来表示不同类型的数据。下表指明了特定正文类型如何映射到 JMS 消息类型。

为了更明确地控制生成的 JMS 消息类型，您可以设置 **x-opt-jms-msg-type** 消息注解。如需更多信息，请参阅 AMQ JMS Interoperability 一章。

表 12.5. AMQ JavaScript 和 JMS 消息类型

AMQ JavaScript 正文类型	JMS 消息类型
string	**TextMessage**
null	**TextMessage**
wrap_binary(string)	**BytesMessage**
任何其他类型	**ObjectMessage**

12.3. 连接到 AMQ BROKER

AMQ Broker 旨在与 AMQP 1.0 客户端互操作。检查以下内容以确保为 AMQP 消息传递配置了代理：

- 网络防火墙中的端口 5672 已打开。
- 启用了 AMQ Broker AMQP 接收器。请参阅 默认接收器设置。
- 代理上配置了必要的地址。请参阅地址、队列和主题。
- 代理配置为允许来自您的客户端的访问，客户端被配置为发送所需的凭证。请参阅 Broker 安全。

12.4. 连接到 AMQ INTERCONNECT

AMQ 互连可与任何 AMQP 1.0 客户端配合工作。检查以下内容以确保正确配置了组件：

- 网络防火墙中的端口 5672 已打开。
- 路由器配置为允许从您的客户端进行访问，并且客户端配置为发送所需的凭据。请参阅保护网络连接。

附录 A. 使用您的订阅

AMQ 通过软件订阅提供。要管理您的订阅，请访问红帽客户门户中的帐户。

A.1. 访问您的帐户

流程

1. 转至 access.redhat.com。
2. 如果您还没有帐户，请创建一个帐户。
3. 登录到您的帐户。

A.2. 激活订阅

流程

1. 转至 access.redhat.com。
2. 导航到 **My Subscriptions**。
3. 导航到 **激活订阅** 并输入您的 16 位激活号。

A.3. 下载发行文件

要访问 .zip、.tar.gz 和其他发布文件，请使用客户门户查找要下载的相关文件。如果您使用 RPM 软件包或 Red Hat Maven 存储库，则不需要这一步。

流程

1. 打开浏览器并登录红帽客户门户网站 **产品下载页面**，网址为 access.redhat.com/downloads。
2. 查找 **INTEGRATION 类别中 的红帽 AMQ** 条目。
3. 选择所需的 AMQ 产品。此时会打开 **Software Downloads** 页面。
4. 单击组件的 **Download** 链接。

A.4. 为系统注册软件包

要在 Red Hat Enterprise Linux 上安装此产品的 RPM 软件包，必须注册您的系统。如果您使用下载的发行文件，则不需要这一步。

流程

1. 转至 access.redhat.com。
2. 进入 **Registration Assistant**。
3. 选择您的操作系统版本，再继续到下一页。
4. 使用您的系统终端中列出的命令完成注册。

有关注册您的系统的更多信息，请参阅以下资源之一：

- Red Hat Enterprise Linux 7 - 注册系统并管理订阅
- Red Hat Enterprise Linux 8 - 注册系统并管理订阅

附录 B. 将 AMQ BROKER 与示例搭配使用

AMQ JavaScript 示例需要一个正在运行的消息代理，它带有名为 **examples** 的队列。使用以下步骤安装和启动代理并定义队列。

B.1. 安装代理

按照 AMQ Broker 入门 以 安装代理 并创建代理 实例 中的内容进行操作。启用匿名访问。

以下步骤将代理实例的位置称为 ***<broker-instance-dir>***。

B.2. 启动代理

流程

1. 使用 **artemis run** 命令启动代理。

```
$ <broker-instance-dir>/bin/artemis run
```

2. 检查控制台输出中是否有启动过程中记录的严重错误。代理日志记录 **Server is now live** （当它就绪时）。

```
$ example-broker/bin/artemis run

        __  __  ____    ____           _
       /  \  |  \/  |/ __ \  |  _ \         | |
      / /\ \ | \  / | |  | | | |_) |_ __ ___ | | _____ _ __
     / /  \ \| |\/| | |  | | |  _ <| '__/ _ \| |/ / _ \ '__|
    / _____ \ |  | | |__| | | |_) | | | (_) |   <  __/ |
   /_/     \_\_|  |_|\____\_\ |____/|_|  \___/|_|\_\___|_|

    Red Hat AMQ <version>

2020-06-03 12:12:11,807 INFO  [org.apache.activemq.artemis.integration.bootstrap]
AMQ101000: Starting ActiveMQ Artemis Server
...
2020-06-03 12:12:12,336 INFO  [org.apache.activemq.artemis.core.server] AMQ221007:
Server is now live
...
```

B.3. 创建队列

在新终端中，使用 **artemis queue** 命令创建名为 **examples** 的队列。

```
$ <broker-instance-dir>/bin/artemis queue create --name examples --address examples --auto-
create-address --anycast
```

系统将提示您回答一系列"是"或无问题。对所有设备回答 **N**。

队列创建后，代理就可与示例程序配合使用。

B.4. 停止代理

运行完示例后，使用 **artemis stop** 命令停止代理。

```
$ <broker-instance-dir>/bin/artemis stop
```

2021-08-31 15:45:45 +1000 修订