Red Hat build of Rhea 3.0

使用 Rhea

使用 JavaScript 开发 AMQ 消息传递客户端

Last Updated: 2024-09-28

Red Hat build of Rhea 3.0 使用 Rhea

使用 JavaScript 开发 AMQ 消息传递客户端

法律通告

摘要

本指南论述了如何与其他 AMQ 组件安装和使用您的客户端。

目录

使开源包含更多

红帽致力于替换我们的代码、文档和 Web 属性中存在问题的语言。我们从这四个术语开始：master、slave、黑名单和白名单。由于此项工作十分艰巨，这些更改将在即将推出的几个发行版本中逐步实施。有关更多详情，请参阅我们的首席技术官 Chris Wright 提供的消息。

第 1 章 概述

RHEA 是用于开发消息传递应用的库。它允许您编写用于发送和接收 AMQP 消息的 JavaScript 应用程序。

红帽构建的 Rhea 是 AMQ 客户端的一部分，这是支持多种语言和平台的一系列消息传递库。有关可用客户端的详情，请参考 AMQ 客户端。

红帽构建的 Rhea 基于 Rhea 消息传递库。有关详细的 API 文档，请参阅 Rhea API 参考。

1.1. 主要特性

- 简化了与现有应用程序集成的事件驱动的 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 和语言原生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

红帽构建的 Rhea 支持以下行业认可的标准和网络协议：

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

1.3. 支持的配置

有关红帽构建的 Rhea 支持的配置 的当前信息，请参阅红帽客户门户网站上的 Red Hat AMQ 支持的配置。

1.4. 术语和概念

本节介绍核心 API 实体，并描述它们如何一起工作。

表 1.1. API 术语

实体	描述
Container	连接的顶级容器。
连接	网络上两个对等点间的通信的频道。它包含会话。

实体	描述
会话	发送和接收消息的上下文。它包含发送者和接收器。
sender	将信息发送到目标的频道。它有一个目标。
receiver	从源接收消息的频道。它有一个源。
Source	用于消息的命名源点。
目标	消息的命名目的地。
消息	特定于应用程序的信息片段。
交付	消息传输。

红帽构建的 Rhea *发送并接收信息*。通过 *发送方* 和接收器在连接的对等点之间传输消息。*发件人和接收器通过会话 建立。会话通过 连接建立。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，但通常不需要这样做。API 允许您忽略会话，除非您需要它们。*

发送对等点会创建发送邮件的发送者。发件人具有在远程对等点上标识队列或主题 的目标。接收对等点会创建一个接收消息的接收器。接收器有一个 源，用于标识远程对等点上的队列或主题。

消息发送称为发送。消息是发送的内容，包括标头和注解等所有元数据。交付是与该内容传输相关联的协议交换。

要指示发送已经完成，发送是发送，可以是发送，也可以是接收方。当另一端了解了它时，它将不再传达该交付。接收器也可以指示它接受或拒绝该消息。

1.5. 文档惯例

sudo 命令
*在本文档中，**sudo** 用于任何需要 root 特权的命令。使用 **sudo** 时请谨慎操作，因为任何更改都可能会影响整个系统。有关 **sudo** 的更多信息，请参阅使用 sudo 命令。*

文件路径
*在本文档中，所有文件路径都对 Linux、UNIX 和类似操作系统（例如 **/home/andrea**）有效。在 Microsoft Windows 上，您必须使用对应的 Windows 路径（例如 **C:\Users\andrea**）。*

变量文本
*本文档包含代码块，其中的变量必须替换为特定于您的环境的值。变量文本以箭头括号括起，样式为方便的 monospace。例如，使用以下命令将 < **project-dir&gt**; 替换为环境的值：*

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

第 2 章 安装

本章介绍了在您的环境中安装红帽构建 Rhea 的步骤。

2.1. 先决条件

- 您必须具有访问 AMQ 发行文件和软件仓库的订阅。
- 您必须在您的环境中安装 **npm** 命令行工具。如需更多信息，请参阅 npm 网站。
- 要使用红帽构建的 Rhea，您必须在环境中安装 Node.js。如需更多信息，请参阅 Node.js 网站。
- 红帽构建的 Rhea 依赖于 Node.js **调试** 模块。有关安装说明，请参阅 debug npm 页面。

2.2. 使用红帽 NPM REGISTRY

配置 NPM 环境，以从 Red Hat NPM registry 下载客户端库。

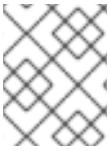
流程

1. 使用 **npm config set** 命令将 Red Hat NPM registry 添加到您的环境中：

```
$ sudo npm config set @redhat:registry https://npm.registry.redhat.com
```

2. 使用 **npm install** 命令安装客户端：

```
$ sudo npm install -g @redhat/rhea@3.0.2-redhat-00001
```



注意

上面的步骤适用于系统范围的安装。您可以在没有 **sudo** 的情况下运行命令，且没有 **-g** 选项来执行本地安装。

要将您的环境配置为使用已安装的库，将 **node_modules/@redhat** 目录添加到 **NODE_PATH** 环境变量中：

Red Hat Enterprise Linux

```
$ export NODE_PATH=/usr/local/lib/node_modules/@redhat:$NODE_PATH
```

Windows

```
$ set NODE_PATH=%AppData%\Roaming\npm\node_modules\@redhat;%NODE_PATH%
```

要测试您的安装，请使用以下命令：如果成功导入了已安装的库，它会将 **OK** 打印到控制台。

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

2.3. 在浏览器中部署客户端

红帽构建的 Rhea 可在网页浏览器中运行。NPM 软件包在以下位置包含名为 **rhea.js** 的文件，该文件可用于基于浏览器的应用：

```
/usr/local/lib/node_modules/@redhat/rhea/dist/rhea.js
```

将 **rhea.js** 文件复制到 Web 服务器公开的位置，并使用 HTML < **script&gt**; 元素引用它，如下例所示：

示例：在浏览器中运行客户端

```
<!DOCTYPE html>
<html>
<head>
  <title>Example</title>
  <script src="rhea.js"></script>
</head>
<body>
  <script>
    const rhea = require("rhea");
    const container = rhea.create_container();

    container.on("message", (event) => {
        console.log(event.message.body);
    });

    const ws = container.websocket_connect(WebSocket);
    const details = ws("ws://example.net:5673", ["binary", "AMQPWSB10", "amqp"])

    const conn = container.connect({"connection_details": details});
    conn.open_receiver("notifications");
  </script>
</body>
```

2.4. 安装示例

1. 使用 **git clone** 命令将源存储库克隆到名为 **rhea** 的本地目录：

```
$ git clone https://github.com/amqp/rhea.git
```

进入 **rhea** 目录，并使用 git checkout 命令签出与此发行版本关联的提交：

```
$ cd rhea
$ git checkout 3.0.2
```

在本指南中，生成的本地目录被称为 &lt **;source-dir** >。

第 3 章 开始使用

本章介绍了设置环境并运行简单的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权限。如需更多信息，请参阅 启动代理。
- 您必须有一个名为 example **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 示例创建了一个与代理的连接，发送一条消息，其中包含到 **示例** 队列的问候消息，然后接收它。成功时，它会将收到的消息输出到控制台。

更改到 examples 目录，再运行 **helloworld.js** 示例。

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

3.3. 在 MICROSOFT WINDOWS 上运行 HELLO WORLD

Hello World 示例创建了一个与代理的连接，发送一条消息，其中包含到 **示例** 队列的问候消息，然后接收它。成功时，它会将收到的消息输出到控制台。

更改到 examples 目录，再运行 **helloworld.js** 示例。

```
> cd <source-dir>/examples
> node helloworld.js
Hello World!
```

第 4 章 例子

本章演示了通过示例程序使用红帽构建的 Rhea。

有关更多示例，请参阅 Rhea 示例套件和 Rhea 示例。

4.1. 发送消息

此客户端程序使用 < **connection-url** > 连接到服务器，为目标 <address& **gt;** 创建发件人，发送一条消息，包含 **<message-body** >、关闭连接并退出。

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，请将其复制到本地文件，并使用 **node** 命令调用它。如需更多信息，请参阅 第 3 章 开始使用。

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

4.2. 接收信息

此客户端程序使用 < **connection-url** > 连接到服务器，为源 <address& **gt;** 创建一个接收器，并接收信息直到终止或到达 < **count>** 信息。

示例：接收信息

```
"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，请将其复制到本地文件，并使用 ***python*** *命令调用它。如需更多信息，请参阅 第 3 章 开始使用。*

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

第 5 章 使用 API

如需更多信息，请参阅 Rhea API 参考、Rhea 示例套件和 Rhea 示例。

5.1. 处理消息传递事件

红帽构建的 Rhea 是一个事件驱动的 API。要定义应用程序如何处理事件，用户会在容器对象上注册 event-handling 功能。然后，这些功能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");
});
```

这些只是几个常见情况事件。完整集记录在 Rhea API 参考 中。

5.2. 访问与事件相关的对象

event 参数具有用于访问事件所针对的对象的属性。例如， connection _open 事件设置事件连接属性。

除了事件的主对象外，也会设置组成该事件上下文的所有对象。没有与特定事件相关的属性为 null。

示例：访问与事件相关的对象

```
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。它通常由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

每个容器实例具有唯一的身份，称为容器 ID。当红帽构建的 Rhea 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，请将 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"});
```

如果用户未设置 ID，则库将在构建容器时生成 UUID。

第 6 章 网络连接

6.1. 创建出站连接

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

示例：创建传出连接

```
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，请参考 第 7 章 安全性。

6.2. 配置重新连接

重新连接可让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故障后重新建立通信。

红帽构建的 Rhea 默认启用重新连接。如果连接尝试失败，客户端将在短暂延迟后重试。每次新尝试时，延迟会增加指数，最多为 60 秒。

要禁用重新连接，请将 重新连接 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的数量，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. 配置故障转移

红帽构建的 Rhea 允许您以编程方式配置备用连接端点程序。

要指定多个连接端点，请定义一个函数来返回新的连接选项，并在 connection_details 选项中传递函数。每次连接尝试调用该函数一次。

示例：配置故障切换

```
var hosts = [{hostname: "alpha.example.com", port: 5672}, {hostname:"beta.example.com",
port: 5672}];
var index = -1;

function failover_fn() {
  index += 1;

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

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

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

container.connect(opts);
```

这个示例为主机列表实施重复循环故障转移。您可以使用此接口来实现自己的故障切换行为。

6.4. 接受进入的连接

红帽构建的 Rhea 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。

若要开始侦听连接，可使用 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：接受传入的连接

```
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);
```

特殊的 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 receive.js 示例。

第 7 章 安全性

7.1. 使用 SSL/TLS 保护连接

红帽构建的 Rhea 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，请将 传输连接 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。在测试环境中有时会出现这种情况。要绕过证书授权，请将 rejectUnauthorized connection 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

红帽构建的 Rhea 可以通过用户和密码验证连接。

要指定用于身份验证的凭证，请设置 用户名和密码 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

红帽构建的 Rhea 使用 SASL 协议来执行身份验证。SASL 可以使用多种不同的 验证机制。当两个网络对等连接时，它们交换其允许的机制，并选择了这两者允许的最强机制。

红帽构建的 Rhea 根据存在用户和密码信息启用 SASL 机制。如果同时指定了用户和密码，则使用 PLAIN。如果只指定用户，则使用 ANONYMOUS。如果没有指定，则禁用 SASL。

第 8 章 发件人和接收器

客户端使用发送方和接收器链接来代表传递消息的频道。发件人和接收器是单向的，消息来源的源结尾和消息目的地的目标结尾。

源和目标通常指向消息代理上的队列或主题。源也用于代表订阅。

8.1. 根据需要创建队列和主题

有些消息服务器支持按需创建队列和主题。附加发送方或接收器时，服务器使用发送者目标地址或接收器源地址来创建名称与地址匹配的队列或主题。

邮件服务器通常默认为创建队列（用于一对一消息发送）或主题（用于一对多消息发送）。客户端可以通过在源或目标上设置 队列或主题 功能来指示首选情况。

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上设置 队列或主题 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

持久化订阅是远程服务器上的一个状态，代表一个消息接收器。通常，当客户端关闭时，消息接收方会被丢弃。但是，由于持久订阅是持久的，客户端可以从它们分离，之后再重新连接。当客户端重新附加时，任何在分离时收到的消息都可用。

持久化订阅通过组合客户端容器 ID 和接收器名称来组成订阅 ID 来唯一标识。它们必须具有稳定的值，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 ）创建一个接收器，并通过设置 durable 和 expiry_policy 属性来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);
```

要从订阅分离，请使用 receiver.detach （） 方法。要终止订阅，请使用 receiver.close （） 方法。

如需更多信息，请参阅 durable-subscribe.js 示例。

8.3. 创建共享订阅

共享订阅是远程服务器中代表一个或多个消息接收器的状态。由于它是共享的，多个客户端可以从同一消息流使用。

客户端通过在接收器源上设置 共享功能来配置共享 订阅。

共享订阅通过组合客户端容器 ID 和接收器名称来组成订阅 ID 来唯一标识。它们必须具有稳定的值，以便多个客户端进程可以找到相同的订阅。如果除了 共享 外设置了 全局 功能，则仅接收方名称用于标识订阅。

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（如 sub-1 ）创建一个接收器，并通过设置共享功能来配置用于 共享的 接收器源：

```
var receiver_opts = {
    source: {
        address: "notifications",
        name: "sub-1",
        capabilities: ["shared"]
    }
}

conn.open_receiver(receiver_opts);
```

要从订阅分离，请使用 receiver.detach （） 方法。要终止订阅，请使用 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

红帽构建的 Rhea 中的错误可以通过截获与 AMQP 协议或连接错误对应的命名事件来处理。

9.1. 处理连接和协议错误

您可以通过截获以下事件来处理协议级别的错误：

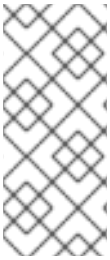
- connection_error
- session_error
- sender_error
- receiver_error
- protocol_error
- 错误

每当事件中存在带有特定对象的错误条件时，这些事件都会触发。调用错误处理程序后，也会调用对应的 & lt;object> _close 处理程序。

event 参数具有用于访问 error 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

由于在出现任何错误时调用关闭处理程序，因此仅需要在错误处理程序中处理错误本身。资源清理可以通过关闭的处理程序来管理。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，且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具有 `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。相反，客户端会开始重新连接过程。

第 10 章 日志记录

10.1. 配置日志记录

红帽构建的 Rhea 使用 JavaScript debug 模块来实施 日志记录。

例如，要启用详细的客户端日志记录，请将 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 协议帧记录到控制台。诊断问题时，这些数据通常至关重要。

要启用协议日志记录，请将 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

红帽构建的 Rhea 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。

当应用调用容器连接方法时，库会尝试读取文件，而无需提供任何连接选项。

11.1. 文件位置

如果设置，红帽构建的 Rhea 使用 MESSAGING_CONNECT_FILE 环境变量的值来定位配置文件。

如果没有设置 MESSAGING_CONNECT_FILE，红帽构建的 Rhea 会在以下位置搜索名为 connect.json 的文件，并按照所示的顺序搜索名为 connect.json 的文件。它在第一次遇到的匹配项时停止。

对于 Linux：

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 上：

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，则库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，具有对 JavaScript 注释的额外支持。

所有配置属性都是可选的，或者具有默认值，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

选项键包含一个点(.)，代表嵌套在命名空间内的属性。

表 11.1. connect.json中的配置选项

键	值类型	默认值	描述
scheme	string	**"amqps"**	**"AMQP"** 用于明文或 SSL/TLS **"amqps"**
主机	string	**"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	string	*None*	进行身份验证的用户名
password	string	*None*	进行身份验证的密码
sasl.mechanisms	list 或 string	*none （系统默认设置）*	启用 SASL 机制的 JSON 列表。裸机字符串代表一种机制。如果未指定，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	string	*None*	客户端证书的文件名或数据库 ID
tls.key	string	*None*	客户端证书的私钥的文件名或数据库 ID
tls.ca	string	*None*	CA 证书的文件名、目录或数据库 ID
tls.verify	布尔值	**true**	需要具有匹配主机名的有效服务器证书

第 12 章 互操作性

本章讨论了如何将红帽构建 Rhea 与其他 AMQ 组件结合使用。有关 AMQ 组件的兼容性概述，请参阅产品简介。

12.1. 与其他 AMQP 客户端交互

AMQP 消息使用 AMQP 类型系统 组成。这种通用格式是以不同语言的 AMQP 客户端能够相互互操作的原因之一。

发送消息时，红帽构建的 Rhea 会自动将语言原生类型转换为 AMQP 编码的数据。收到信息时，反向转换就发生。

注意

有关 AMQP 类型的更多信息，请访问由 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	一系列 Unicode 字符
binary	一个字节序列
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

AMQP 类型	描述
ushort	未签名的 16 位整数
uint	未签名的 32 位整数
ulong	未签名的 64 位整数
浮点值	32 位浮点号
double	64 位浮点号
数组	单个类型的值的序列
list	变量类型的序列值
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	一个绝对时间点

JavaScript 具有比 AMQP 可以编码的原生类型。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. RHEA 类型编码前和解码后

AMQP 类型	RHEA 在编码前类型	解码后 RHEA 类型
null	null	null
布尔值	布尔值	布尔值
char	wrap_char (number)	number
string	string	string
二进制	wrap_binary (string)	string
byte	wrap_byte(number)	number
short	wrap_short(number)	number

AMQP 类型	RHEA 在编码前类型	解码后 RHEA 类型
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
ulong	wrap_ulong(number)	number
浮点值	wrap_float (number)	number
double	wrap_double(number)	number
数组	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. RHEA 和其他 AMQ 客户端类型(2 为 1)

RHEA 在编码前类型	AMQ C++ 类型	红帽构建的 Apache Qpid Proton DotNet 类型
null	nullptr	null
布尔值	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

RHEA 在编码前类型	AMQ C++ 类型	红帽构建的 Apache Qpid Proton DotNet 类型
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**
wrap_uint(number)	**uint32_t**	**System.UInt32**
wrap_ulong(number)	**uint64_t**	**System.UInt64**
wrap_float (number)	**浮点值**	**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. RHEA 和其他 AMQ 客户端类型(2 个)

RHEA 在编码前类型	红帽构建的 Apache Qpid Proton Python 类型	
null	**None**	
布尔值	**bool**	
string	**Unicode**	
wrap_char (number)	**Unicode**	

RHEA 在编码前类型	红帽构建的 Apache Qpid Proton Python 类型	
wrap_binary (string)	bytes	
wrap_byte(number)	int	
wrap_short(number)	int	
wrap_int (number)	long	
wrap_long (number)	long	
wrap_ubyte(number)	long	
wrap_ushort(number)	long	
wrap_uint(number)	long	
wrap_ulong(number)	long	
wrap_float (number)	浮点值	
wrap_double(number)	浮点值	
wrap_array (Array, code)	proton.Array	
wrap_list (Array)	list	
wrap_map(object)	dict	
wrap_uuid(number)	-	
wrap_symbol(string)	str	
wrap_timestamp(number)	long	

12.2. 使用红帽构建的 APACHE QPID JMS 进行交互

AMQP 定义标准映射到 JMS 消息传递模型。本节讨论该映射的各个方面。如需更多信息，请参阅红帽构建的 Apache Qpid JMS Interoperability 章节。

JMS 消息类型

红帽构建的 Rhea 提供了一个单一消息类型，其正文类型可能会有所不同。相反，JMS API 使用不同的消息类型来代表不同类型的数据。下表标明特定的正文类型如何映射到 JMS 消息类型。

若要更明确地控制生成的 JMS 消息类型，您可以设置 x-opt-jms-msg-type 消息注释。如需更多信息，请参阅红帽构建的 Apache Qpid JMS Interoperability 章节。

表 12.5. RHEA 和 JMS 消息类型

RHEA 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 将打开。
- 启用 AMQ Broker AMQP acceptor。请参阅 默认接受者设置。
- 在代理上配置必要的地址。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。

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 目录中的红帽 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 8 - 注册系统并管理订阅
- Red Hat Enterprise Linux 9 - 注册系统并管理订阅

附录 B. 使用带有示例的 AMQ BROKER

红帽构建的 Rhea 示例需要一个正在运行的消息代理，其中包含名为 example 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 中的说明 来安装代理 并创建代理实例。启用匿名访问。

以下流程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，以查看启动期间记录的所有关键错误。代理日志服务器 现在在服务器就绪时处于实时 状态。

```
$ 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 example 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 yes 或没有问题。全部答案为 N。

创建队列后，代理就可以与示例程序一起使用。

B.4. 停止代理

运行完示例后，请使用 artemis stop 命令来停止代理。

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

更新于 2024-02-10