Red Hat AMQ 2021.Q3

使用 AMQ Python 客户端

用于 AMQ 客户端 2.10

Last Updated: 2023-02-20

Red Hat AMQ 2021.Q3 使用 AMQ Python 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 Python 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 Python 应用。

AMQ Python 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 Python 基于 Apache Qpid 中的 Proton API。如需详细的 API 文档，请参阅 AMQ Python 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
- 基于 OpenTracing 标准（RHEL 7 和 8）的分布式追踪

重要

AMQ 客户端中的分布式追踪只是一个技术预览功能。技术预览功能不被红帽产品服务等级协议 (SLA) 支持，且可能在功能方面有缺陷。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。有关红帽技术预览功能支持范围的详情，请参阅 https://access.redhat.com/support/offerings/techpreview/。

1.2. 支持的标准和协议

AMQ Python 支持行业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
- Cyrus SASL 支持 的简单身份验证和安全层(SASL)机制，包括 ANONYMOUS、PLAIN、SCRAM、EXTERNAL 和 GSSAPI(Kerberos)
- 使用 IPv6的现代 TCP

1.3. 支持的配置

有关 AMQ Python 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。
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 Python 发送并接收 *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 Python 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。
- 要在 Red Hat Enterprise Linux 上安装软件包，您必须 注册您的系统。
- 要使用 AMQ Python，您必须在您的环境中安装 Python。

2.2. 在 RED HAT ENTERPRISE LINUX 上安装

流程

1. 使用 **subscription-manager** 命令订阅所需的软件包软件仓库。将 ***<version>*** 替换为主发行流的 **2** 或 **2.9** 用于长期支持发行流。如果需要，使用 Red Hat Enterprise Linux 变体的值替换 ***<variant>*** （例如： **server** 或 **workstation**）。

   Red Hat Enterprise Linux 7

   ```
   $ sudo subscription-manager repos --enable=amq-clients-<version>-for-rhel-7-<variant>-rpms
   ```

   Red Hat Enterprise Linux 8

   ```
   $ sudo subscription-manager repos --enable=amq-clients-<version>-for-rhel-8-x86_64-rpms
   ```

2. 使用 **yum** 命令安装软件包。

   主发行流

   ```
   $ sudo yum install python3-qpid-proton python-qpid-proton-docs
   ```

   AMQ 客户端 2.9 长期支持流

   ```
   $ sudo yum install python-qpid-proton python-qpid-proton-docs
   ```

有关使用软件包的详情请参考 附录 B, *使用 Red Hat Enterprise Linux 软件包* 。

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. 为您的 Python 版本下载 **AMQ Clients 2.10.0 Python**.whl 文件。

Python 3.6	python_qpid_proton-0.35.0-cp36-cp36m-win_amd64.whl
Python 3.8	python_qpid_proton-0.35.0-cp38-cp38-win_amd64.whl

5. 打开命令提示窗口并使用 **pip install** 命令安装 .whl 文件。

Python 3.6

```
> pip install python_qpid_proton-0.35.0-cp36-cp36m-win_amd64.whl
```

Python 3.8

```
> pip install python_qpid_proton-0.35.0-cp38-cp38-win_amd64.whl
```

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.py** 示例。

```
$ cd /usr/share/proton/examples/python/
$ python helloworld.py
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。

下载并运行 Hello World 示例。

```
> curl -o helloworld.py https://raw.githubusercontent.com/apache/qpid-
proton/master/python/examples/helloworld.py
> python helloworld.py
Hello World!
```

第 4 章 示例

本章介绍如何通过示例程序使用 AMQ Python。

如需了解更多示例，请参阅 AMQ Python 示例套件 和 Qpid Proton Python 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

```
from __future__ import print_function

import sys

from proton import Message
from proton.handlers import MessagingHandler
from proton.reactor import Container

class SendHandler(MessagingHandler):
    def __init__(self, conn_url, address, message_body):
        super(SendHandler, self).__init__()

        self.conn_url = conn_url
        self.address = address
        self.message_body = message_body

    def on_start(self, event):
        conn = event.container.connect(self.conn_url)

        # To connect with a user and password:
        # conn = event.container.connect(self.conn_url, user="<user>", password="<password>")

        event.container.create_sender(conn, self.address)

    def on_link_opened(self, event):
        print("SEND: Opened sender for target address '{0}'".format
              (event.sender.target.address))

    def on_sendable(self, event):
        message = Message(self.message_body)
        event.sender.send(message)

        print("SEND: Sent message '{0}'".format(message.body))

        event.sender.close()
        event.connection.close()

def main():
    try:
        conn_url, address, message_body = sys.argv[1:4]
    except ValueError:
        sys.exit("Usage: send.py <connection-url> <address> <message-body>")
```

```
    handler = SendHandler(conn_url, address, message_body)
    container = Container(handler)
    container.run()

if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        pass
```

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。

```
$ python send.py 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

```
from __future__ import print_function

import sys

from proton.handlers import MessagingHandler
from proton.reactor import Container

class ReceiveHandler(MessagingHandler):
    def __init__(self, conn_url, address, desired):
        super(ReceiveHandler, self).__init__()

        self.conn_url = conn_url
        self.address = address
        self.desired = desired
        self.received = 0

    def on_start(self, event):
        conn = event.container.connect(self.conn_url)

        # To connect with a user and password:
        # conn = event.container.connect(self.conn_url, user="<user>", password="<password>")

        event.container.create_receiver(conn, self.address)

    def on_link_opened(self, event):
        print("RECEIVE: Created receiver for source address '{0}'".format
              (self.address))

    def on_message(self, event):
        message = event.message
```

```
        print("RECEIVE: Received message '{0}'".format(message.body))

        self.received += 1

        if self.received == self.desired:
            event.receiver.close()
            event.connection.close()

def main():
    try:
        conn_url, address = sys.argv[1:3]
    except ValueError:
        sys.exit("Usage: receive.py <connection-url> <address> [<message-count>]")

    try:
        desired = int(sys.argv[3])
    except (IndexError, ValueError):
        desired = 0

    handler = ReceiveHandler(conn_url, address, desired)
    container = Container(handler)
    container.run()

if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        pass
```

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。

```
$ python receive.py amqp://localhost queue1
```

第 5 章 使用 API

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

5.1. 处理消息传递事件

AMQ Python 是异步事件驱动 API。要定义应用程序如何处理事件，用户在 **MessagingHandler** 类上实施回调方法。然后，这些方法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

```
class ExampleHandler(MessagingHandler):
    def on_start(self, event):
        print("The container event loop has started")

    def on_sendable(self, event):
        print("A message can be sent")

    def on_message(self, event):
        print("A message is received")
```

这些只是几个常见案例事件。完整的集合记录在 API 引用 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，**on_connection_opened** 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

```
handler = ExampleHandler()
container = Container(handler)
container.run()
```

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 Python 进行连接时，它会将容器 ID 发送到远程对等点。要设置容器 ID，将其传递给 **Container** 构造器。

示例：设置容器身份

```
container = Container(handler)
container.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. 连接 URL

连接 URL 对用于建立新连接的信息进行编码。

连接 URL 语法

```
scheme://host[:port]
```

- *scheme* - 连接传输，可以为未加密 TCP 或 **amqp** 用于使用 SSL/TLS 加密的 TCP 进行连接传输。**amqps**
- *主机* - 远程网络主机。该值可以是主机名或数字 IP 地址。IPv6 地址必须括在方括号中。
- *port* - 远程网络端口。这个值是可选的。**amqp** 方案默认值为 5672，**amqps** 方案为 5671。

连接 URL 示例

```
amqps://example.com
amqps://example.net:56720
amqp://127.0.0.1
amqp://[::1]:2000
```

6.2. 创建传出连接

要连接到远程服务器，请使用 连接 URL 调用 **Container.connect()** 方法。这通常在 **MessagingHandler.on_start()** 方法内完成。

示例：创建出站连接

```
class ExampleHandler(MessagingHandler):
    def on_start(self, event):
        event.container.connect("amqp://example.com")

    def on_connection_opened(self, event):
        print("Connection", event.connection, "is open")
```

有关创建安全连接的详情请参考 第 7 章 *安全性*。

6.3. 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 Python 启用重新连接。如果连接丢失或连接尝试失败，客户端会在短暂延迟后重试。每次新尝试时，延迟呈指数级增长，默认为 10 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

```
container.connect("amqp://example.com", reconnect=False)
```

要控制连接尝试之间的延迟，请定义实现 **reset()** 和 **next()** 方法的类，并将 **reconnect** 连接选项设置为该类的实例。

示例：配置重新连接

```
class ExampleReconnect(object):
    def __init__(self):
        self.delay = 0

    def reset(self):
        self.delay = 0

    def next(self):
        if self.delay == 0:
            self.delay = 0.1
        else:
            self.delay = min(10, 2 * self.delay)

        return self.delay

container.connect("amqp://example.com", reconnect=ExampleReconnect())
```

next 方法以秒为单位返回下一个延迟。在重新连接进程开始前调用 **reset** 方法一次。

6.4. 配置故障切换

AMQ Python 允许您配置多个连接端点。如果连接失败，客户端将尝试连接到列表中的下一个。如果列表已用尽，则进程将重新开始。

要指定多个连接端点，将 **urls** 连接选项设置为连接 URL 列表。

示例：配置故障切换

```
urls = ["amqp://alpha.example.com", "amqp://beta.example.com"]
container.connect(urls=urls)
```

同时使用 **url** 和 **urls** 选项时出错。

6.5. 接受传入的连接

AMQ Python 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，请使用包含本地主机地址和端口的 URL **Container.listen()** 方法。

示例：接受进入的连接

```
class ExampleHandler(MessagingHandler):
    def on_start(self, event):
        event.container.listen("0.0.0.0")
```

```
    def on_connection_opened(self, event):
        print("New incoming connection", event.connection)
```

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.py 示例。

第 7 章 安全性

7.1. 使用 SSL/TLS 保护连接

AMQ Python 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，请使用连接 URL 和 **amqps** 方案。

示例：启用 SSL/TLS

```
container.connect("amqps://example.com")
```

7.2. 使用用户和密码连接

AMQ Python 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，请在 **connect()** 方法中设置 **user** 和 **password** 选项。

示例：使用用户和密码连接

```
container.connect("amqps://example.com", user="alice", password="secret")
```

7.3. 配置 SASL 身份验证

AMQ Python 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。

注意

客户端使用 Cyrus SASL 来执行身份验证。Cyrus SASL 使用插件来支持特定的 SASL 机制。您必须先安装相关的插件，然后才能使用特定的 SASL 机制。例如，您需要 **cyrus-sasl-plain** 插件才能使用 SASL PLAIN 身份验证。

要查看 Red Hat Enterprise Linux 中的 Cyrus SASL 插件列表，请使用 **yum search cyrus-sasl** 命令。要安装 Cyrus SASL 插件，请使用 **yum install PLUG-IN** 命令。

默认情况下，AMQ Python 允许本地 SASL 库配置支持的所有机制。要限制允许的机制并控制可以协商哪些机制，请使用 **allowed_mechs** 连接选项。它取包含以空格分隔的机制名称列表的字符串。

示例：配置 SASL 身份验证

```
container.connect("amqps://example.com", allowed_mechs="ANONYMOUS")
```

这个示例强制连接使用 **ANONYMOUS** 机制进行身份验证，即使我们连接的服务器还提供其他选项。有效机制包括 **ANONYMOUS**、**PLAIN**、**SCRAM-SHA-256**、**SCRAM-SHA-1**、**GSSAPI** 和 **EXTERNAL**。

AMQ Python 默认启用 SASL。要禁用它，将 **sasl_enabled** 连接选项设置为 false。

示例：禁用 SASL

```
event.container.connect("amqps://example.com", sasl_enabled=False)
```

7.4. 使用 KERBEROS 进行身份验证

Kerberos 是一种网络协议，用于根据加密票据的交换进行集中管理的身份验证。如需更多信息，请参阅使用 Kerberos。

1. 在您的操作系统中配置 Kerberos。请参阅在 Red Hat Enterprise Linux 中配置 Kerberos 以设置 Kerberos。
2. 在客户端应用中启用 **GSSAPI** SASL 机制。

```
container.connect("amqps://example.com", allowed_mechs="GSSAPI")
```

3. 使用 **kinit** 命令验证您的用户凭证并存储生成的 Kerberos 票据。

```
$ kinit <user>@<realm>
```

4. 运行 客户端程序。

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

```
class CapabilityOptions(SenderOption):
    def apply(self, sender):
        sender.target.capabilities.put_object(symbol("queue"))

class ExampleHandler(MessagingHandler):
    def on_start(self, event):
        conn = event.container.connect("amqp://example.com")
        event.container.create_sender(conn, "jobs", options=CapabilityOptions())
```

示例：从按需创建的主题接收

```
class CapabilityOptions(ReceiverOption):
    def apply(self, receiver):
        receiver.source.capabilities.put_object(symbol("topic"))

class ExampleHandler(MessagingHandler):
    def on_start(self, event):
        conn = event.container.connect("amqp://example.com")
        event.container.create_receiver(conn, "notifications", options=CapabilityOptions())
```

如需更多信息，请参阅以下示例：

- queue-send.py
- queue-receive.py
- topic-send.py
- topic-receive.py

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。

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**:

```
container = Container(handler)
container.container_id = "client-1"
```

2. 设置 **durability** 和 **expiry_policy** 属性，将接收器源配置为持久性：

```
class SubscriptionOptions(ReceiverOption):
    def apply(self, receiver):
        receiver.source.durability = Terminus.DELIVERIES
        receiver.source.expiry_policy = Terminus.EXPIRE_NEVER
```

3. 使用稳定名称（如 **sub-1** ）创建接收器，并应用源属性：

```
event.container.create_receiver(conn, "notifications",
                                name="sub-1",
                                options=SubscriptionOptions())
```

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.py 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**:

```
container = Container(handler)
container.container_id = "client-1"
```

2. 通过设置 **shared** 功能，将接收器源配置为共享：

```
class SubscriptionOptions(ReceiverOption):
    def apply(self, receiver):
        receiver.source.capabilities.put_object(symbol("shared"))
```

3. 使用稳定名称（如 **sub-1** ）创建接收器，并应用源属性：

```
event.container.create_receiver(conn, "notifications",
                                name="sub-1",
                                options=SubscriptionOptions())
```

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.py 示例。

第 9 章 消息发送

9.1. 发送消息

要发送消息，请覆盖 **on_sendable** 事件处理程序并调用 **Sender.send()** 方法。当 **Sender** 有足够的分数来发送至少一个信息时，**sendable** 事件会触发。

示例：发送消息

```
class ExampleHandler(MessagingHandler):
    def on_start(self, event):
        conn = event.container.connect("amqp://example.com")
        sender = event.container.create_sender(conn, "jobs")

    def on_sendable(self, event):
        message = Message("job-content")
        event.sender.send(message)
```

如需更多信息，请参阅 send.py 示例。

9.2. 跟踪发送的消息

发送消息时，发件人可以保留对代表传输的 **delivery** 对象的引用。发送消息后，接收方接受或拒绝消息。向发送方通知每次交付的结果。

要监控发送消息的结果，请覆盖 **on_accepted** 和 **on_rejected** 事件处理程序，并将交付状态更新映射到从 **send()** 返回的发送。

示例：跟踪发送的消息

```
def on_sendable(self, event):
    message = Message(self.message_body)
    delivery = event.sender.send(message)

def on_accepted(self, event):
    print("Delivery", event.delivery, "is accepted")

def on_rejected(self, event):
    print("Delivery", event.delivery, "is rejected")
```

9.3. 接收消息

要接收信息，创建一个接收器并覆盖 **on_message** 事件处理程序。

示例：接收消息

```
class ExampleHandler(MessagingHandler):
    def on_start(self, event):
        conn = event.container.connect("amqp://example.com")
        receiver = event.container.create_receiver(conn, "jobs")
```

```
    def on_message(self, event):
        print("Received message", event.message, "from", event.receiver)
```

如需更多信息，请参阅 receive .py 示例。

9.4. 确认收到的信息

要明确接受或拒绝交付，请使用 **on_message** 事件处理程序中的 **ACCEPTED** 或 **REJECTED** 状态的 **Delivery.update()** 方法。

示例：确认收到的信息

```
def on_message(self, event):
    try:
        process_message(event.message)
        event.delivery.update(ACCEPTED)
    except:
        event.delivery.update(REJECTED)
```

默认情况下，如果您没有明确确认交付，库会在 **on_message** 返回后接受它。要禁用此行为，将 **auto_accept** 接收器选项设置为 false。

第 10 章 错误处理

AMQ Python 中的错误可以通过两种不同的方式进行处理：

- 捕获异常
- 覆盖事件处理功能以截获 AMQP 协议或连接错误

10.1. 捕获异常

AMQ Python 从 **ProtonException** 类抛出的所有例外，后者从 Python **Exception** 类继承。

以下示例演示了如何捕获来自 AMQ Python 的任何异常：

示例：特定于 **API** 的异常处理

```
try:
    # Something that might throw an exception
except ProtonException as e:
    # Handle Proton-specific problems here
except Exception as e:
    # Handle more general problems here
}
```

如果您不需要特定于 API 的异常处理，则只需要捕获 **Exception**，因为 **ProtonException** 继承了它。

10.2. 处理连接和协议错误

您可以通过覆盖以下 **messaging_handler** 方法来处理协议级别的错误：

- **on_transport_error(event)**
- **on_connection_error(event)**
- **on_session_error(event)**
- **on_link_error(event)**

只要存在事件处理的具体对象的错误条件，就会调用这些事件处理函数。在调用错误处理程序后，也会调用适当的关闭处理程序。

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，则通常使用常规 **on_error** 处理器，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 **on_connection_error** 处理程序。客户端改为开始重新连接过程。

第 11 章 日志记录

11.1. 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 **PN_TRACE_FRM** 环境变量设置为 **1** ：

示例：启用协议日志记录

```
$ export PN_TRACE_FRM=1
$ <your-client-program>
```

要禁用协议日志，请取消设置 **PN_TRACE_FRM** 环境变量。

第 12 章 分布式追踪

12.1. 启用分布式追踪

客户端根据 OpenTracing 标准的 Jaeger 提供分布式追踪。使用以下步骤在应用程序中启用追踪：

1. 安装追踪依赖项。

   Red Hat Enterprise Linux 7

   ```
   $ sudo yum install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
   $ sudo yum install python2-pip
   $ pip install --user --upgrade setuptools
   $ pip install --user opentracing jaeger-client
   ```

   Red Hat Enterprise Linux 8

   ```
   $ sudo dnf install python3-pip
   $ pip3 install --user opentracing jaeger-client
   ```

2. 在您的计划中注册全球跟踪器。

   示例：全局追踪器配置

   ```
   from proton.tracing import init_tracer

   tracer = init_tracer("<service-name>")
   ```

如需有关 Jaeger 配置的更多信息，请参阅 Jaeger Sampling。

在测试或调试时，您可能需要强制 Jaeger 跟踪特定的操作。如需更多信息，请参阅 Jaeger Python 客户端文档。

要查看应用程序捕获的 trace，请使用 Jaeger Getting Started 运行 Jaeger 基础架构和控制台。

第 13 章 基于文件的配置

AMQ Python 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。

13.1. 文件位置

如果设置，AMQ Python 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 Python 会在以下位置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。

13.2. 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"
    }
}
```

13.3. 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 13.1. 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 14 章 互操作性

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

14.1. 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 Python 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 14.1. 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

表 14.2. 编码之前和解码之后的 AMQ Python 类型

AMQP 类型	编码前 AMQ Python 类型	解码后 AMQ Python 类型
null	None	None
boolean	bool	bool
char	proton.char	unicode
string	unicode	unicode
binary	bytes	bytes
byte	proton.byte	int
short	proton.short	int
int	proton.int32	long
long	long	long
ubyte	proton.ubyte	long
ushort	proton.ushort	long
uint	proton.uint	long
ulong	proton.ulong	long

AMQP 类型	编码前 AMQ Python 类型	解码后 AMQ Python 类型
float	proton.float32	float
double	float	float
array	proton.Array	proton.Array
list	list	list
map	dict	dict
symbol	proton.symbol	str
timestamp	proton.timestamp	long

表 14.3. AMQ Python 和其他 AMQ 客户端类型（2 中的 1 个）

编码前 AMQ Python 类型	AMQ C++ 类型	AMQ JavaScript 类型
None	nullptr	null
bool	bool	boolean
proton.char	wchar_t	number
unicode	std::string	string
bytes	proton::binary	string
proton.byte	int8_t	number
proton.short	int16_t	number
proton.int32	int32_t	number
long	int64_t	number
proton.ubyte	uint8_t	number
proton.ushort	uint16_t	number
proton.uint	uint32_t	number
proton.ulong	uint64_t	number

编码前 AMQ Python 类型	AMQ C++ 类型	AMQ JavaScript 类型
proton.float32	float	number
float	double	number
proton.Array	-	Array
list	std::vector	Array
dict	std::map	object
uuid.UUID	proton::uuid	number
proton.symbol	proton::symbol	string
proton.timestamp	proton::timestamp	number

表 14.4. AMQ Python 和其他 AMQ 客户端类型（2 个）

编码前 AMQ Python 类型	AMQ .NET 类型	AMQ Ruby 类型
None	null	nil
bool	System.Boolean	true, false
proton.char	System.Char	String
unicode	System.String	String
bytes	System.Byte[]	String
proton.byte	System.SByte	Integer
proton.short	System.Int16	Integer
proton.int32	System.Int32	Integer
long	System.Int64	Integer
proton.ubyte	System.Byte	Integer
proton.ushort	System.UInt16	Integer
proton.uint	System.UInt32	Integer

编码前 AMQ Python 类型	AMQ .NET 类型	AMQ Ruby 类型
proton.ulong	**System.UInt64**	**Integer**
proton.float32	**System.Single**	**Float**
float	**System.Double**	**Float**
proton.Array	-	**Array**
list	**Amqp.List**	**Array**
dict	**Amqp.Map**	**Hash**
uuid.UUID	**System.Guid**	-
proton.symbol	**Amqp.Symbol**	**Symbol**
proton.timestamp	**System.DateTime**	**Time**

14.2. 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 Python 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 14.5. AMQ Python 和 JMS 消息类型

AMQ Python 正文类型	JMS 消息类型
unicode	**TextMessage**
None	**TextMessage**
bytes	**BytesMessage**
任何其他类型	**ObjectMessage**

14.3. 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。

14.4. 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. 使用 RED HAT ENTERPRISE LINUX 软件包

这部分论述了如何将软件包作为 Red Hat Enterprise Linux 的 RPM 软件包使用。

为确保此产品的 RPM 软件包可用，您必须先 注册您的系统。

B.1. 概述

库或服务器等组件通常具有多个与之相关联的软件包。您不必全部安装它们。您只能安装您需要的产品。

主软件包通常具有最简单的名称，没有额外的限定符。此软件包提供在程序运行时使用组件所需的所有接口。

名称以 **-devel** 结尾的软件包包含 C 和 C++ 库的标头。编译时需要这些，以构建依赖于此软件包的程序。

名称以 **-docs** 结尾的软件包包含组件的文档和示例程序。

有关使用 RPM 软件包的更多信息，请参阅以下资源之一：

- Red Hat Enterprise Linux 7 - 安装和管理软件
- Red Hat Enterprise Linux 8 - 管理软件包

B.2. 搜索软件包

要搜索软件包，请使用 **yum search** 命令。搜索结果包括软件包名称，您可以在本节中列出的其他命令中用作 ***<package>*** 的值。

```
$ yum search <keyword>...
```

B.3. 安装软件包

要安装软件包，使用 **yum install** 命令。

```
$ sudo yum install <package>...
```

B.4. 查询软件包信息

要列出系统中安装的软件包，使用 **rpm -qa** 命令。

```
$ rpm -qa
```

要获取有关特定软件包的信息，请使用 **rpm -qi** 命令。

```
$ rpm -qi <package>
```

要列出与软件包关联的所有文件，请使用 **rpm -ql** 命令。

```
$ rpm -ql <package>
```

附录 C. 将 AMQ BROKER 与示例搭配使用

AMQ Python 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。

C.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>***。

C.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
...
```

C.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。

C.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 15:47:09 +1000 修订