# Red Hat AMQ 2021.Q1

## AMQ Python クライアントの使用

AMQ Clients 2.9 向け

Last Updated: 2023-02-27

# Red Hat AMQ 2021.Q1 AMQ Python クライアントの使用

AMQ Clients 2.9 向け

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.

## 法律上の通知



## 概要

本ガイドでは、クライアントをインストールして設定する方法、実例を実行し、他の AMQ コンポーネントでクライアントを使用する方法を説明します。

# 目次

# 多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、弊社 の CTO、Chris Wright のメッセージを参照してください。

# 第1章 概要

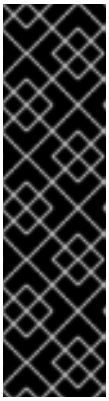
AMQ Python は、メッセージングアプリケーションを開発するためのライブラリーです。また、AMQP メッセージを送受信する Python アプリケーションを作成できます。

AMQ Python は AMQ Clients (複数の言語やプラットフォームをサポートするメッセージングライブラリースイート) に含まれています。クライアントの概要は、AMQ Clients の概要 を参照してください。本リリースに関する詳細は、『AMQ Clients 2.9 リリースノート』 を参照してください。

AMQ Python は、Apache Qpid の Proton API をベースとしています。詳細な API ドキュメントは、AMQ Python API リファレンス を参照してください。

## 1.1. 主な特長

- 既存のアプリケーションとの統合を簡素化するイベント駆動型の API
- セキュアな通信用の SSL/TLS
- 柔軟な SASL 認証
- 自動再接続およびフェイルオーバー
- AMQP と言語ネイティブのデータ型間のシームレスな変換
- AMQP 1.0 の全機能へのアクセス
- OpenTracing 標準（RHEL 7 および 8）に基づく分散トレーシング



**重要**

AMQ Clients での分散トレーシングはテクノロジープレビュー機能です。テクノロジープレビュー機能は、Red Hat 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。Red Hat は実稼働環境でこれらを使用することを推奨していません。これらの機能は、近々発表予定の製品機能をリリースに先駆けてご提供することにより、開発プロセスの中でお客様に機能性のテストとフィードバックをしていただくことを目的としています。Red Hat のテクノロジープレビュー機能のサポート範囲に関する詳細は、https://access.redhat.com/ja/support/offerings/techpreview/ を参照してください。

## 1.2. サポートされる標準およびプロトコル

AMQ Python は、以下の業界標準およびネットワークプロトコルをサポートします。

- Advanced Message Queueing Protocol (AMQP) のバージョン 1.0
- SSL の後継である TLS (Transport Layer Security) プロトコルのバージョン 1.0、1.1、1.2、および 1.3
- ANONYMOUS、PLAIN、SCRAM、EXTERNAL、および GSSAPI (Kerberos) を含む、Cyrus SASL でサポートされる単純な認証およびセキュリティーレイヤー (SASL) メカニズム
- IPv6 での最新の TCP

## 1.3. サポートされる構成

AMQ Python は、以下に示す OS および言語のバージョンをサポートします。詳細は、「Red Hat AMQ 7 Supported Configurations」を参照してください。

- Red Hat Enterprise Linux 7 と Python 2.7
- Python 3.6 を使用する Red Hat Enterprise Linux 8
- Python 3.6 および Python 3.8 を使用する Microsoft Windows 10 Pro
- Microsoft Windows Server 2012 R2 および 2016（Python 3.6 および Python 3.8）

AMQ Python は、以下の AMQ コンポーネントおよびバージョンと組み合わせてサポートされます。

- すべてのバージョンの AMQ Broker
- すべてのバージョンの AMQ Interconnect
- A-MQ 6 バージョン 6.2.1 以降

## 1.4. 用語および概念

本セクションでは、コア API エンティティーを紹介し、コア API が連携する方法を説明します。

**表1.1 API の用語**

| エンティティー | 説明 |
|---|---|
| **Container** | 接続の最上位のコンテナー。 |
| **接続** | ネットワーク上の 2 つのピア間の通信チャネル。これにはセッションが含まれます。 |
| **Session** | メッセージの送受信を行うためのコンテキスト。送信者および受信者が含まれます。 |
| **sender** | メッセージをターゲットに送信するためのチャネル。これにはターゲットがあります。 |
| **receiver** | ソースからメッセージを受信するためのチャネル。これにはソースがあります。 |
| **Source** | メッセージの名前付きの発信元。 |
| **Target** | メッセージの名前付き受信先。 |
| **メッセージ** | 情報のアプリケーション固有の部分。 |
| **Delivery** | メッセージの転送。 |

AMQ Python は **メッセージ** を送受信します。メッセージは、**senders** と **receivers** を介して、接続されたピアの間で転送されます。送信側および受信側は **セッション** 上で確立されます。セッションは**接**

**続**上で確立されます。接続は、一意に識別された 2 つの **コンテナー** 間で 確立されます。コネクションには複数のセッションを含めることができますが、多くの場合、必要ありません。API を使用すると、セッションが必要でない限り、セッションを無視できます。

送信ピアは、メッセージ送信用の送信者を作成します。送信側には、リモートピアでキューまたはトピックを識別する **ターゲット** があります。受信ピアは、メッセージ受信用の受信者を作成します。受信側には、リモートピアでキューまたはトピックを識別する **ソース** があります。

メッセージの送信は **配信** と呼ばれます。メッセージとは、送信される内容のことで、ヘッダーやアノテーションなどのすべてのメタデータが含まれます。配信は、そのコンテンツの移動に関連するプロトコルエクスチェンジです。

配信が完了したことを示すには、送信側または受信側セットのいずれかが解決します。送信側または受信側が解決されたことを知らせると、その配信の通信ができなくなります。受信側は、メッセージを受諾するか、拒否するかどうかを指定することもできます。

## 1.5. 本書の表記慣例

**sudo コマンド**
本書では、root 権限を必要とするすべてのコマンドに対して **sudo** が使用されています。すべての変更がシステム全体に影響する可能性があるため、**sudo** を使用する場合は注意が必要です。**sudo** の詳細は、sudo コマンドの使用を参照してください。

**ファイルパス**
本書では、すべてのファイルパスが Linux、UNIX、および同様のオペレーティングシステムで有効です（例： **/home/andrea**）。Microsoft Windows では、同等の Windows パスを使用する必要があります（例： **C:\Users\andrea**）。

**変数テキスト**
本書では、変数を含むコードブロックが紹介されていますが、これは、お客様の環境に固有の値に置き換える必要があります。可変テキストは矢印の中括弧で囲まれ、斜体の等幅フォントとしてスタイル設定されます。たとえば、以下のコマンドでは **<project-dir>** は実際の環境の値に置き換えます。

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

# 第2章 インストールシステム

本章では、環境に AMQ Python をインストールする手順を説明します。

## 2.1. 前提条件

- AMQ リリースファイルおよびリポジトリーにアクセスするには、サブスクリプション が必要です。
- パッケージを Red Hat Enterprise Linux にインストールするには、システムが登録されている必要があります。
- AMQ Python を使用するには、お使いの環境に Python をインストールする必要があります。

## 2.2. 「RED HAT ENTERPRISE LINUX へのインストール」 を参照してください。

**手順**

1. **subscription-manager** コマンドを使用して、必要なパッケージリポジトリーをサブスクライブします。必要に応じて、**<variant>** を Red Hat Enterprise Linux のバリアントの値（例えば、**server** または **workstation**）に置き換えます。

   Red Hat Enterprise Linux 7

   ```
   $ sudo subscription-manager repos --enable=amq-clients-2-for-rhel-7-<variant>-rpms
   ```

   Red Hat Enterprise Linux 8

   ```
   $ sudo subscription-manager repos --enable=amq-clients-2-for-rhel-8-x86_64-rpms
   ```

2. **yum** コマンドを使用して、**python-qpid-proton** パッケージおよび **python-qpid-proton-docs** パッケージをインストールします。

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

パッケージの使用方法は、付録B *Red Hat Enterprise Linux パッケージの使用* を参照してください。

## 2.3. 「MICROSOFT WINDOWS へのインストール」 を参照してください。

**手順**

1. ブラウザーを開き、access.redhat.com/downloads で Red Hat カスタマーポータルの **製品のダウンロード** ページにログインします。
2. **INTEGRATION AND AUTOMATION** カテゴリーで **Red Hat AMQ Clients** エントリーを見つけます。
3. **Red Hat AMQ Clients** をクリックします。**Software Downloads** ページが開きます。
4. お使いの **Python バージョン用の AMQ Clients 2.9.0 Python** .whl ファイルをダウンロードします。

| | |
|---|---|
| **Python 3.6** | python_qpid_proton-0.33.0-cp36-cp36m-win_amd64.whl |
| **Python 3.8** | python_qpid_proton-0.33.0-cp38-cp38-win_amd64.whl |

5. コマンドプロンプトウィンドウを開き、**pip install** コマンドを使用して .whl ファイルをインストールします。

### Python 3.6

```
> pip install python_qpid_proton-0.33.0-cp36-cp36m-win_amd64.whl
```

### Python 3.8

```
> pip install python_qpid_proton-0.33.0-cp38-cp38-win_amd64.whl
```

# 第3章 はじめに

本章では、環境を設定して簡単なメッセージングプログラムを実行する手順を説明します。

## 3.1. 前提条件

- ご使用の環境のインストール手順を完了する必要があります。
- インターフェース **localhost** およびポート **5672** で接続をリッスンする AMQP 1.0 メッセージブローカーが必要です。匿名アクセスを有効にする必要があります。詳細は、ブローカーの開始を参照してください。
- **examples** という名前のキューが必要です。詳細は、キューの作成 を参照してください。

## 3.2. RED HAT ENTERPRISE LINUX での HELLO WORLD の実行

Hello World の例では、ブローカーへの接続を作成し、グリーティングを含むメッセージを**examples**キューに送信して、受信しなおします。成功すると、受信したメッセージをコンソールに出力します。

examples ディレクトリーに移動し、**helloworld.py** の例を実行します。

```
$ cd /usr/share/proton/examples/python/
$ python helloworld.py
Hello World!
```

## 3.3. MICROSOFT WINDOWS での HELLO WORLD の実行

Hello World の例では、ブローカーへの接続を作成し、グリーティングを含むメッセージを**examples**キューに送信して、受信しなおします。成功すると、受信したメッセージをコンソールに出力します。

Hello World の例をダウンロードして実行します。

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

# 第4章 例

本章では、サンプルプログラムで AMQ Python を使用する方法について説明します。

その他の例は、AMQ Python サンプルのスイート と Qpid Proton Python の例 を参照してください。

## 4.1. メッセージの送信

このクライアントプログラムは **<connection-url>** を使用してサーバーに接続し、ターゲット **<address>** の送信者を作成し、**<message-body>** を含むメッセージを送信して接続を切断して終了します。

**例: メッセージの送信**

```
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
```

**サンプルの実行**

サンプルプログラムを実行するには、サンプルプログラムをローカルファイルにコピーし、**python** コマンドを使用して呼び出します。詳細は、「3章*はじめに*」を参照してください。

```
$ python send.py amqp://localhost queue1 hello
```

## 4.2. メッセージの受信

このクライアントプログラムは **<connection-url>** を使用してサーバーに接続し、ソース **<address>** の受信側を作成し、終了するか、**<count>** メッセージに到達するまでメッセージを受信します。

**例: メッセージの受信**

```
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
```

**サンプルの実行**

サンプルプログラムを実行するには、サンプルプログラムをローカルファイルにコピーし、**python** コマンドを使用して呼び出します。詳細は、「3章*はじめに*」を参照してください。

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

# 第5章 API の使用

詳細は、「 AMQ Python API リファレンス」および「 AMQ Python サンプルスイート 」を参照してください。

## 5.1. メッセージングイベントの処理

AMQ Python は非同期イベント駆動型 API です。アプリケーションがイベントを処理する方法を定義するために、ユーザーは **MessagingHandler** クラスでコールバックメソッドを実装します。これらの方法は、ネットワークアクティビティーとして呼び出され、タイマーが新規イベントをトリガーします。

**例: メッセージングイベントの処理**

```
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")
```

これらはごく一部の一般的なケースイベントのみです。完全セットは API リファレンス に文書化されています。

## 5.2. イベント関連のオブジェクトへのアクセス

**イベント**引数には、イベントが関係するオブジェクトにアクセスするための属性が含まれます。たとえば、**on_connection_opened** イベントはイベント **connection** 属性を設定します。

イベントのプライマリーオブジェクトに加えて、イベントのコンテキストを形成する全オブジェクトも設定されます。特定のイベントに対する関連性のない属性は null です。

**例: イベント関連のオブジェクトへのアクセス**

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

## 5.3. コンテナーの作成

コンテナーは最上位の API オブジェクトです。これは、接続を作成するエントリーポイントであり、メインのイベントループを実行します。多くの場合、これはグローバルイベントハンドラーで構築されます。

**例: コンテナーの作成**

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

## 5.4. コンテナーアイデンティティーの設定

各コンテナーインスタンスには、コンテナー ID と呼ばれる一意のアイデンティティーがあります。AMQ Python がネットワーク接続を作成する場合、コンテナー ID をリモートピアに送信します。コンテナー ID を設定するには、これを **Container** コンストラクターに渡します。

**例: コンテナーアイデンティティーの設定**

```
container = Container(handler)
container.container_id = "job-processor-3"
```

ユーザーが ID を設定しない場合には、コンテナーが処理されると、ライブラリーは UUID を生成します。

# 第6章 ネットワーク接続

## 6.1. 接続 URL

接続 URL は、新規接続の確立に使用される情報をエンコードします。

**接続 URL 構文**

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

- **スキーム** - 暗号化されていない TCP の **amqp**、または SSL/TLS 暗号化のある TCP の **amqps** のいずれかの接続トランスポート。
- **ホスト** - リモートのネットワークホスト。値は、ホスト名または数値の IP アドレスの場合があります。IPv6 アドレスは角括弧で囲む必要があります。
- **ポート** - リモートネットワークポート。.この値はオプションです。デフォルト値は、**amqp** スキームの場合は 5672 で、**amqps** スキームの場合は 5671 です。

**接続 URL サンプル**

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

## 6.2. 外向き接続の作成

リモートサーバーに接続するには、接続 URL で **Container.connect（）** メソッドを呼び出します。通常、これは **MessagingHandler.on_start（）** メソッド内で行われます。

**例: 外向き接続の作成**

```
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")
```

セキュアな接続の作成に関する詳細は、*7章 セキュリティー* を参照してください。

## 6.3. 再接続の設定

再接続することで、クライアントは失われた接続から復旧できます。これは、一時的なネットワークまたはコンポーネントの障害後に、分散システムのコンポーネントが再確立されるように使用されます。

AMQ Python はデフォルトで再接続を有効にします。接続が失われた場合、または接続の試行が失敗した場合、クライアントは少し遅れて再試行します。遅延は、デフォルトの最大値 10 秒まで、新しい試行ごとに指数関数的に増加します。

再接続を無効にするには、**reconnect** 接続オプションを **False** に設定します。

**例: 再接続の無効化**

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

接続試行間の遅延を制御するには、**reset()** メソッドおよび **next（）** メソッドを実装するクラスを定義し、**再接続** 接続オプションをそのクラスのインスタンスに設定します。

**例: 再接続の設定例**

```
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** メソッドは、次の遅延を秒単位で返します。**reset** メソッドは、再接続プロセスが開始する前に一度呼び出されます。

## 6.4. フェイルオーバーの設定

AMQ Python では、複数の接続エンドポイントを設定できます。ある接続に失敗すると、クライアントはリスト内の次の接続を試みます。一覧が使い切られると、プロセスは最初から開始します。

複数の接続エンドポイントを指定するには、**urls** 接続オプションを接続 URL の一覧に設定します。

**例: フェイルオーバーの設定**

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

**url** および **urls** オプションを同時に使用するのはエラーです。

## 6.5. 内向き接続の許可

AMQ Python はインバウンドネットワーク接続を受け入れ、カスタムメッセージングサーバーを構築できます。

接続のリッスンを開始するには、**Container.listen()** メソッドを使用して、ローカルホストアドレスとリッスンするポートが含まれる URL を指定します。

**例: 内向き接続の許可**

```
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)
```

特別な IP アドレス **0.0.0.0** は、利用可能なすべての IPv4 インターフェースでリッスンします。すべての IPv6 インターフェースをリッスンするには **[::0]** を使用します。

詳細は、サーバー receive.py の例 を参照してください。

# 第7章 セキュリティー

## 7.1. SSL/TLS を使用した接続のセキュリティー保護

AMQ Python は SSL/TLS を使用して、クライアントとサーバー間の通信を暗号化します。

SSL/TLS を使用してリモートサーバーに接続するには、**amqps** スキームで接続 URL を使用します。

**例: SSL/TLS の有効化**

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

## 7.2. ユーザーとパスワードを使用した接続

AMQ Python は、ユーザーとパスワードによる接続を認証できます。

認証に使用する認証情報を指定するには、**connect()** メソッドに **user** および **password** オプションを設定します。

**例: ユーザーとパスワードを使用した接続**

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

## 7.3. SASL 認証の設定

AMQ Python は SASL プロトコルを使用して認証を実行します。SASL はさまざまな認証 **メカニズム** を使用できます。2 つのネットワークピアが接続すると、許可されたメカニズムが交換され、両方で許可されている最も強力なメカニズムが選択されます。

**注記**

クライアントは Cyrus SASL を使用して認証を実行します。Cyrus SASL は、プラグインを使用して特定の SASL メカニズムをサポートします。特定の SASL メカニズムを使用する前に、関連するプラグインをインストールする必要があります。たとえば、SASL PLAIN 認証を使用するには、**cyrus-sasl-plain** プラグインが必要です。

Red Hat Enterprise Linux の Cyrus SASL プラグインのリストを表示するには、**yum search cyrus-sasl** コマンドを使用します。Cyrus SASL プラグインをインストールするには、**yum install PLUG-IN** コマンドを使用します。

デフォルトでは、AMQ Python はローカルの SASL ライブラリー設定でサポートされるすべてのメカニズムを許可します。許可されるメカニズムを制限し、ネゴシエートできるメカニズムを制御するには、**allowed_mechs** 接続オプションを使用します。スペースで区切られたメカニズム名のリストが含まれる文字列を取ります。

**例: SASL 認証の設定**

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

この例では、サーバーが他のオプションを提供するように接続しても、**ANONYMOUS** メカニズムを使用した認証を強制します。有効なメカニズムには、**ANONYMOUS**、**PLAIN**、**SCRAM-SHA-256**、**SCRAM-SHA-1**、**GSSAPI**、**EXTERNAL** が含まれます。

AMQ Python はデフォルトで再接続を有効にします。これを無効にするには、**sasl_enabled** 接続オプションを false に設定します。

**例: SASL の無効化**

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

## 7.4. KERBEROS を使用した認証

Kerberos は、暗号化されたチケットの交換に基づいて一元管理された認証用のネットワークプロトコルです。詳細は、「 Kerberos の使用 」を参照してください。

1. オペレーティングシステムで Kerberos を設定します。Red Hat Enterprise Linux で Kerberos を設定するには「 Kerberos を設定する 」を参照してください。
2. クライアントアプリケーションで **GSSAPI** SASL メカニズムを有効にします。

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

3. **kinit** コマンドを使用して、ユーザーの認証情報を認証し、作成された Kerberos チケットを保存します。

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

4. クライアントプログラムを実行します。

# 第8章 送信者と受信者

クライアントは、送信者と受信者のリンクを使用して、メッセージ配信のチャネルを表現します。送信者と受信者は一方向であり、送信元はメッセージの発信元に、ターゲットはメッセージの宛先になります。

ソースとターゲットは、多くの場合、メッセージブローカーのキューまたはトピックを参照します。ソースは、サブスクリプションを表すためにも使用されます。

## 8.1. オンデマンドでのキューとトピックの作成

メッセージサーバーによっては、キューとトピックのオンデマンド作成をサポートします。送信側または受信側が割り当てられている場合、サーバーは送信側ターゲットアドレスまたは受信側ソースアドレスを使用して、アドレスに一致する名前でキューまたはトピックを作成します。

メッセージサーバーは通常、キュー（1対1のメッセージ配信用）またはトピック（1対多のメッセージ配信用）を作成します。クライアントは、ソースまたはターゲットに **queue** または **topic** 機能を設定してどちらを優先するかを示すことができます。

キューまたはトピックセマンティクスを選択するには、以下の手順に従います。

1. キューとトピックを自動的に作成するようにメッセージサーバーを設定します。多くの場合、これがデフォルト設定になります。

2. 以下の例のように、送信者ターゲットまたは受信者ソースに**キュー**または**トピック**機能を設定します。

**例: オンデマンドで作成されたキューへの送信**

```
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())
```

**例: オンデマンドで作成されたトピックからの受信**

```
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())
```

詳細は、以下の例を参照してください。

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

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

## 8.2. 永続サブスクリプションの作成

永続サブスクリプションは、メッセージの受信側を表すリモートサーバーの状態です。通常、メッセージ受信者は、クライアントが終了すると、破棄されます。ただし、永続サブスクリプションは永続的であるため、クライアントはそれらのサブスクリプションの割り当てを解除してから、後で再度アタッチできます。デタッチ時に受信したすべてのメッセージは、クライアントの再割り当て時に利用できます。

永続サブスクリプションは、クライアントコンテナー ID とレシーバー名を組み合わせてサブスクリプション ID を形成することで一意に識別されます。これらには、サブスクリプションを回復できるように、安定した値が必要です。

永続サブスクリプションを作成するには、以下の手順に従います。

1. 接続コンテナー ID を **client-1** などの安定した値に設定します。

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

2. **durability** および **expiry_policy** プロパティーを設定して、受信側ソースで持続性を設定します。

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

3. **sub-1** などの安定した名前で受信側を作成し、ソースプロパティーを適用します。

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

サブスクリプションからデタッチするには、**Receiver.detach()** メソッドを使用します。サブスクリプションを終了するには、**Receiver.close()** メソッドを使用します。

詳細は、durable-subscribe.py の例 を参照してください。

## 8.3. 共有サブスクリプションの作成

共有サブスクリプションとは、1 つ以上のメッセージレシーバーを表すリモートサーバーの状態のことです。このサブスクリプションは共有されているため、複数のクライアントが同じメッセージのストリームから消費できます。

クライアントは、受信者のソースに**shared** 機能を設定して、共有サブスクリプションを設定します。

共有サブスクリプションは、クライアントコンテナー ID とレシーバー名を組み合わせてサブスクリプション ID を形成することで一意に識別されます。複数のクライアントプロセスで同じサブスクリプションを特定できるように、これらに安定した値を指定する必要があります。**shared** に加えて **global** 機能が設定されている場合、サブスクリプション識別に受信者名だけが使用されます。

永続サブスクリプションを作成するには、以下の手順に従います。

1. 接続コンテナー ID を **client-1** などの安定した値に設定します。

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

2. **shared** 機能を設定して、共有用の受信側ソースを設定します。

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

3. **sub-1** などの安定した名前で受信側を作成し、ソースプロパティーを適用します。

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

サブスクリプションからデタッチするには、**Receiver.detach()** メソッドを使用します。サブスクリプションを終了するには、**Receiver.close()** メソッドを使用します。

詳細は、shared-subscribe.py の例 を参照してください。

# 第9章 メッセージ配信

## 9.1. メッセージの送信

メッセージを送信するには、**on_sendable** イベントハンドラーを上書きし、**Sender.send()** メソッドを呼び出します。**sendable** なイベントは、プロースト **Sender** に少なくとも 1 つのメッセージを送信するのに十分なクレジットがある場合に実行されます。

**例: メッセージの送信**

```
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)
```

詳細は、send.py の例 を参照してください。

## 9.2. 送信されたメッセージの追跡

メッセージを送信する場合、送信側は転送を表す **delivery** オブジェクトへの参照を保持することができます。メッセージが配信されると、受信側はこれを受け入れるか、拒否します。各配信の結果が送信者に通知されます。

送信されたメッセージの結果を監視するには、**on_accepted** イベントハンドラーおよび **on_rejected** イベントハンドラーを上書きし、配信状態の更新を **send（）** から返された配信にマップします。

**例: 送信したメッセージの追跡**

```
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. メッセージの受信

メッセージの受信には、レシーバーを作成し、**on_message** イベントハンドラーを上書きします。

**例: メッセージの受信**

```
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)
```

詳細は、receive.py の例 を参照してください。

## 9.4. 受信したメッセージの承認

配信を明示的に許可または拒否するには、**on_message** イベントハンドラーで **ACCEPTED** または **REJECTED** 状態の **Delivery.update（）** メソッドを使用します。

**例: 受信したメッセージの承認**

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

デフォルトでは、配信を明示的に確認しないと、ライブラリーは **on_message** が返された後に受け入れます。この動作を無効にするには、**auto_accept** receiver オプションを false に設定します。

# 第10章 エラー処理

AMQ Python でのエラーは、以下の 2 つの方法で処理できます。

- 例外のキャッチ
- AMQP プロトコルまたは接続エラーを傍受するためのイベント処理関数の上書き

## 10.1. 例外のキャッチ

AMQ Python がスローするすべての例外は、**ProtonException** クラスから継承され、Python **Exception** クラスから継承されます。

以下の例は、AMQ Python から発生した例外をキャッチする方法を示しています。

**例: API 固有の例外処理**

```
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
}
```

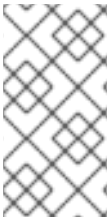
API 固有の例外処理が必要ない場合は、**ProtonException** が継承されるため 、 **ProtonException** のみをキャッチする必要があります。

## 10.2. 接続およびプロトコルエラーの処理

以下の **messaging_handler** メソッドを上書きすると、プロトコルレベルのエラーを処理できます。

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

これらのイベント処理関数は、イベント内の特定のオブジェクトにエラー状態が発生するたびに呼び出されます。エラーハンドラーを呼び出すと、適切なクローズハンドラーも呼び出されます。



**注記**

クローズハンドラーはエラー発生時に呼び出されるため、エラーハンドラー内でのみ処理する必要があります。リソースのクリーンアップは、近辺にあるハンドラーで管理できます。特定のオブジェクトに固有のエラー処理がない場合は、通常は、一般的な **on_error** ハンドラーを使用してより具体的なハンドラーを用意しません。

### 注記

再接続が有効になっており、リモートサーバーが **amqp:connection:forced** の条件で接続が切断されると、クライアントはこれをエラーとして処理しないため、**on_connection_error** ハンドラーは実行されません。代わりに、クライアントが再接続プロセスを開始します。

# 第11章 ロギング

## 11.1. プロトコルロギングの有効化

クライアントは AMQP プロトコルフレームをコンソールに記録できます。多くの場合、このデータは問題の診断時に重要になります。

プロトコルロギングを有効にするには、**PN_TRACE_FRM** 環境変数を **1** に設定します。

**例: プロトコルロギングの有効化**

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

プロトコルロギングを無効にするには、**PN_TRACE_FRM** 環境変数の設定を解除します。

# 第12章 分散トレース

## 12.1. 分散トレースの有効化

クライアントは、OpenTracing 標準の Jaeger 実装に基づいて分散トレーシングを提供します。アプリケーションでトレースを有効にするには、以下の手順に従います。

1. トレーシング依存関係をインストールします。

   **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. プログラムにグローバルトレーサーを登録します。

   **例: グローバルトレーサー設定**

   ```
   from proton.tracing import init_tracer

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

Jaeger の設定に関する詳細は、「Jaeger Sampling」を参照してください。

テストまたはデバッグの際に、Jaeger が特定の操作を追跡できるように強制できます。詳細は、Jaeger Python クライアントのドキュメント を参照してください。

アプリケーションをキャプチャーするトレースを表示するには、Jaeger Getting Started を使用して Jaeger インフラストラクチャーおよびコンソールを実行します。

# 第13章 ファイルベースの設定

AMQ Python は、**connect.json** という名前のローカルファイルからの接続確立に使用される設定オプションを読み取ることができます。これにより、デプロイメント時にアプリケーションで接続を設定できます。

ライブラリーは、接続オプションを指定せずにアプリケーションがコンテナーの **connect** メソッドを呼び出すと、ファイルの読み取りを試みます。

## 13.1. ファイルの場所

設定されている場合には、AMQ Python は **MESSAGING_CONNECT_FILE** 環境変数の値を使用して設定ファイルを検索します。

**MESSAGING_CONNECT_FILE** が設定されていない場合には、AMQ Python は以下の場所で **connect.json** という名前のファイルを検索します。最初の一致で停止します。

**Linux の場合:**

1. **$PWD/connect.json**: **$PWD** はクライアントプロセスの現在の作業ディレクトリーです。
2. **$HOME/.config/messaging/connect.json**: **$HOME** は現在のユーザーのホームディレクトリーに置き換えます。
3. **/etc/messaging/connect.json**

**Windows の場合:**

1. **%cd%/connect.json**: **%cd%** はクライアントプロセスの現在の作業ディレクトリーです。

**connect.json** ファイルが見つからない場合、ライブラリーはすべてのオプションにデフォルト値を使用します。

## 13.2. ファイル形式

**connect.json** ファイルには JSON データが含まれ、JavaScript コメントの追加サポートが提供されます。

設定属性はすべてオプションであるか、デフォルト値があるため、簡単な例では詳細をいくつか指定するだけで済みます。

**例: 簡単な connect.json ファイル**

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

SASL および SSL/TLS オプションは、**"sasl"** および **"tls"** namespace で入れ子になっています。

**例: SASL および SSL/TLS オプションを含む connect.json ファイル**

```
{
```

```
    "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. 設定オプション

ドット(.)を含むオプションキーは、namespace にネストされた属性を表します。

**表13.1 connect.jsonの設定オプション**

| キー | 値のタイプ | デフォルト値 | 説明 |
|---|---|---|---|
| **scheme** | string | **"amqps"** | SSL/TLS のクリアテキストまたは **"amqps"** の場合は **"amqp"** |
| **host** | string | **"localhost"** | リモートホストのホスト名または IP アドレス |
| **ポート** | 文字列または番号 | **"amqps"** | ポート番号またはポートリテラル |
| **user** | string | None | 認証のユーザー名 |
| **password** | string | None | 認証のパスワード |
| **sasl.mechanisms** | リストまたは文字列 | **none（システムのデフォルト）** | 有効な SASL メカニズムの JSON リスト。ベア文字列は 1 つのメカニズムを表します。指定がない場合、クライアントはシステムによって提供されるデフォルトのメカニズムを使用します。 |
| **sasl.allow_insecure** | boolean | **false** | クリアテキストパスワードを送信するメカニズムの有効化 |
| **tls.cert** | string | None | クライアント証明書のファイル名またはデータベース ID |
| **tls.key** | string | None | クライアント証明書の秘密鍵のファイル名またはデータベース ID |
| **tls.ca** | string | None | CA 証明書のファイル名、ディレクトリー、またはデータベース ID |
| **tls.verify** | boolean | **true** | ホスト名が一致する、有効なサーバー証明書が必要 |

# 第14章 相互運用性

本章では、AMQ Python を他の AMQ コンポーネントと組み合わせて使用する方法を説明します。AMQ コンポーネントの互換性の概要は、「 製品の概要 」を参照してください。

## 14.1. 他の AMQP クライアントとの相互運用

AMQP メッセージは AMQP タイプシステムを使用して構成されます。このような一般的な形式は、異なる言語の AMQP クライアントが相互に対話できる理由の 1 つです。

メッセージを送信する場合、AMQ Python は自動的に言語ネイティブの型を AMQP でエンコードされたデータに変換します。メッセージの受信時に、リバース変換が行われます。

**注記**

AMQP タイプの詳細は、Apache Qpid プロジェクトによって維持されるインタラクティブタイプリファレンスを参照してください。

**表14.1 AMQP 型**

| AMQP 型 | 説明 |
|---|---|
| **null** | 空の値 |
| **boolean** | true または false の値 |
| **char** | 単一の Unicode 文字 |
| **string** | Unicode 文字のシーケンス |
| **binary** | バイトのシーケンス |
| **byte** | 署名済み 8 ビット整数 |
| **short** | 署名済み 16 ビット整数 |
| **int** | 署名済み 32 ビット整数 |
| **long** | 署名済み 64 ビット整数 |
| **ubyte** | 署名なしの 8 ビット整数 |
| **ushort** | 署名なしの 16 ビット整数 |
| **uint** | 署名なしの 32 ビット整数 |
| **ulong** | 署名なしの 64 ビット整数 |
| **float** | 32 ビット浮動小数点数 |

| AMQP 型 | 説明 |
|---|---|
| **double** | 64 ビット浮動小数点数 |
| **array** | 単一型の値シーケンス |
| **list** | 変数型の値シーケンス |
| **map** | 異なるキーから値へのマッピング |
| **uuid** | ユニバーサルー意識別子 |
| **symbol** | 制限されたドメインからの 7 ビットの ASCII 文字列 |
| **timestamp** | 絶対的な時点 |

表14.2 エンコード前およびデコード後における AMQ Python タイプ

| AMQP 型 | エンコード前の AMQ Python タイプ | デコード後の AMQ Python タイプ |
|---|---|---|
| **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 型 | エンコード前の AMQ Python タイプ | デコード後の AMQ Python タイプ |
|---|---|---|
| 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 およびその他の AMQ クライアントタイプ (1/2)

| エンコード前の AMQ Python タイプ | AMQ C++ タイプ | AMQ JavaScript タイプ |
|---|---|---|
| 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 |

| エンコード前の AMQ Python タイプ | AMQ C++ タイプ | AMQ JavaScript タイプ |
|---|---|---|
| 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 およびその他の AMQ クライアントタイプ (2/2)

| エンコード前の AMQ Python タイプ | AMQ .NET タイプ | AMQ Ruby タイプ |
|---|---|---|
| None | null | nil |
| bool | System.Boolean | true, false |
| proton.char | System.Char | 文字列 |
| unicode | system.String | 文字列 |
| bytes | System.Byte[] | 文字列 |
| 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 |

| エンコード前の AMQ Python タイプ | AMQ .NET タイプ | AMQ Ruby タイプ |
|---|---|---|
| **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. AMQ JMS での相互運用

AMQP は JMS メッセージングモデルへの標準マッピングを定義します。本セクションでは、そのマッピングのさまざまな側面について説明します。詳細は、AMQ JMS Interoperability の章を参照してください。

### JMS メッセージタイプ

AMQ Python は、本文タイプが異なる、単一のメッセージを提供します。一方、JMS API は異なるメッセージタイプを使用してさまざまな種類のデータを表します。次の表は、特定の本文タイプが JMS メッセージタイプにどのようにマップされるかを示しています。

作成される JMS メッセージタイプをさらに明示的に制御するには、**x-opt-jms-msg-type** メッセージアノテーションを設定できます。詳細は、AMQ JMS Interoperability の章を参照してください。

**表14.5 AMQ Python および JMS メッセージタイプ**

| AMQ Python ボディーのタイプ | JMS メッセージタイプ |
|---|---|
| **unicode** | **TextMessage** |
| **None** | **TextMessage** |
| **bytes** | **BytesMessage** |
| それ以外のタイプ | **ObjectMessage** |

## 14.3. AMQ BROKER への接続

AMQ Broker は AMQP 1.0 クライアントと相互運用するために設計されています。以下を確認して、ブローカーが AMQP メッセージング用に設定されていることを確認します。

- ネットワークファイアウォールのポート 5672 が開いている。
- AMQ Broker AMQP アクセプターが有効になっている。デフォルトのアクセプター設定 を参照してください。
- 必要なアドレスがブローカーに設定されている。アドレス、キュー、およびトピック を参照してください。
- ブローカーはクライアントからのアクセスを許可するように、クライアントは必要なクレデンシャルを送信するように設定されます。Broker Security を参照してください。

## 14.4. AMQ INTERCONNECT への接続

AMQ Interconnect は AMQP 1.0 クライアントであれば機能します。以下をチェックして、コンポーネントが正しく設定されていることを確認します。

- ネットワークファイアウォールのポート 5672 が開いている。
- ルーターはクライアントからのアクセスを許可するように、クライアントは必要なクレデンシャルを送信するように設定されます。ネットワーク接続のセキュリティー保護 を参照してください。

# 付録A サブスクリプションの使用

AMQ は、ソフトウェアサブスクリプションから提供されます。サブスクリプションを管理するには、Red Hat カスタマーポータルでアカウントにアクセスします。

## A.1. アカウントへのアクセス

**手順**

1. access.redhat.com に移動します。
2. アカウントがない場合は、作成します。
3. アカウントにログインします。

## A.2. サブスクリプションのアクティベート

**手順**

1. access.redhat.com に移動します。
2. **サブスクリプション** に移動します。
3. **Activate a subscription** に移動し、16 桁のアクティベーション番号を入力します。

## A.3. リリースファイルのダウンロード

.zip、.tar.gz およびその他のリリースファイルにアクセスするには、カスタマーポータルを使用してダウンロードする関連ファイルを検索します。RPM パッケージまたは Red Hat Maven リポジトリーを使用している場合は、この手順は必要ありません。

**手順**

1. ブラウザーを開き、access.redhat.com/downloads で Red Hat カスタマーポータルの **Product Downloads** ページにログインします。
2. **JBOSS INTEGRATION AND AUTOMATION** カテゴリーの **Red Hat AMQ** エントリーを見つけます。
3. 必要な AMQ 製品を選択します。**Software Downloads** ページが開きます。
4. コンポーネントの **Download** リンクをクリックします。

## A.4. パッケージ用システムの登録

この製品の RPM パッケージを Red Hat Enterprise Linux にインストールするには、システムが登録されている必要があります。ダウンロードしたリリースファイルを使用している場合は、この手順は必要ありません。

**手順**

1. access.redhat.com に移動します。

2. **Registration Assistant** に移動します。
3. ご使用の OS バージョンを選択し、次のページに進みます。
4. システムの端末に一覧表示されたコマンドを使用して、登録を完了します。

システムを登録する方法は、以下のリソースを参照してください。

- Red Hat Enterprise Linux 7 - システム登録およびサブスクリプション管理
- Red Hat Enterprise Linux 8 - システム登録およびサブスクリプション管理

# 付録B RED HAT ENTERPRISE LINUX パッケージの使用

本セクションでは、Red Hat Enterprise Linux の RPM パッケージとして配信されるソフトウェアを使用する方法を説明します。

この製品の RPM パッケージを利用できるようにするには、最初に システムを登録 する必要があります。

## B.1. 概要

ライブラリーやサーバーなどのコンポーネントには多くの場合、複数のパッケージが関連付けられています。それらをすべてインストールする必要はありません。必要なものだけをインストールできます。

プライマリーパッケージは、通常、追加の修飾子がない最もシンプルな名前です。このパッケージは、プログラムのランタイム時にコンポーネントを使用するために必要なすべてのインターフェースを提供します。

**-devel** で終わる名前を持つパッケージには、C ライブラリーおよび C++ ライブラリーのヘッダーが含まれます。このパッケージに依存するプログラムを構築する際の、コンパイル時に必要になります。

**-docs** で終わる名前を持つパッケージには、コンポーネントのドキュメントとサンプルプログラムが含まれます。

RPM パッケージ を使用する方法は、以下のリソースのいずれかを参照してください。

- Red Hat Enterprise Linux 7 - ソフトウェアのインストールおよび管理
- Red Hat Enterprise Linux 8 - ソフトウェアパッケージの管理

## B.2. パッケージの検索

パッケージを検索するには、**yum search** コマンドを使用します。検索結果にはパッケージ名が含まれます。パッケージ名は、このセクションに記載されている他のコマンドで **<package>** の値として使用できます。

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

## B.3. パッケージのインストール

パッケージをインストールするには、**yum install** コマンドを使用します。

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

## B.4. パッケージ情報のクエリー

システムにインストールされているパッケージを一覧表示するには、**rpm -qa** コマンドを使用します。

```
$ rpm -qa
```

特定のパッケージに関する情報を取得するには、**rpm -qi** コマンドを使用します。

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

パッケージに関連するファイルを一覧表示するには、**rpm -ql** コマンドを使用します。

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

# 付録C 例で AMQ ブローカーの使用

AMQ Python の例では、名前が **examples**というキューが含まれる実行中のメッセージブローカーが必要です。以下の手順に従って、ブローカーをインストールして起動し、キューを定義します。

## C.1. ブローカーのインストール

『 **Getting Started with AMQ Broker**』の手順に従って、ブローカーをインストールし、ブローカーインスタンスを作成 します。匿名アクセスを有効にします。

以下の手順では、ブローカーインスタンスの場所を **<broker-instance-dir>** と呼びます。

## C.2. ブローカーの起動

**手順**

1. **artemis run** コマンドを使用してブローカーを起動します。

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

2. 起動時にログに記録された重大なエラーがないか、コンソールの出力を確認してください。ブローカーでは、準備が整うと **Server is now live** とログが記録されます。

```
$ 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. キューの作成

新しいターミナルで、**artemis queue** コマンドを使用して **examples** という名前のキューを作成します。

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

プロンプトで質問に Yes または No で回答するように求められます。すべての質問に **N** (いいえ) と回答します。

キューが作成されると、ブローカーはサンプルプログラムで使用できるようになります。

## C.4. ブローカーの停止

サンプルの実行が終了したら、**artemis stop** コマンドを使用してブローカーを停止します。

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

**改訂日時：** 2022-09-08 16:30:08 +1000