

|  |  |
| --- | --- |
| Kafkaコネクタ構築手順  Snmptrap受信ソースコネクタ開発事例 | 要約  この開発ガイドは、Kafka Connectにおけるソースコネクタの開発について、snmptrapを受信するソースコネクタの開発を事例に解説しています。具体的には、snmptrapを受信するソースコネクタの開発手順や、設定ファイルの記述方法、動作確認などを詳しく説明しています。このガイドを通じて、Kafka Connectにおけるソースコネクタの開発の基本的な考え方や手順を学ぶことができます。また、snmptrapを受信するソースコネクタの開発に関心がある場合は、具体的な実装例を通じて理解を深めることができます。  Keiji Jin@redhat  AMQ Streams Connector開発ガイド |

[1. はじめに 2](#_Toc131792970)

[2. Kafka Connect開発の利点 2](#_Toc131792971)

[3. Kafka Connector開発手順 3](#_Toc131792972)

[4. 設定ファイルの記述 4](#_Toc131792973)

[5. Configクラスの実装 5](#_Toc131792974)

[6. Connectorクラスの実装 9](#_Toc131792975)

[7. SourceTaskクラスの実装 11](#_Toc131792976)

[8. Kafka Connectorのビルド・デプロイと実行 15](#_Toc131792977)

[必要なもの 15](#_Toc131792978)

[snmptrap connectorのビルド 15](#_Toc131792979)

[snmptrap connectorのデプロイ 16](#_Toc131792980)

[Kafka Connectの実行 16](#_Toc131792981)

[snmptrap connectorのテスト 17](#_Toc131792982)

[9. Kafka Connectの最適化とチューニング 17](#_Toc131792983)

[10. おわりに 18](#_Toc131792984)

[付録：pom.xmlファイルについて 20](#_Toc131792985)

# はじめに

Kafkaは、分散ストリーム処理を行うためのオープンソースプラットフォームです。Kafka Connectは、Kafkaクラスターと外部システムを接続するためのフレームワークです。Kafka Connectを使用することで、さまざまなシステムからデータを収集し、Kafkaクラスターに格納することができます。また、Kafkaクラスターからデータを取得し、外部システムに転送することもできます。

Kafka Connectは、Kafka Connectorを使用して、外部システムとの接続を容易にします。Kafka Connectorは、Javaで書かれたプラグインであり、データソースまたはターゲットシステムのAPIを使用して、Kafka Connectフレームワークと通信します。Connectorは、データの移動に必要なすべての設定を管理し、データをシンクロナイズするためのタスクを定義します。

このガイドは、Kafka Connectorの開発についての詳細な情報を提供します。ガイドは、Kafka ConnectフレームワークとKafka Connectorの概要から始まり、Connectorの開発手順、実装プログラムの詳細、デプロイと実行方法、そして最適化とチューニングについて説明します。このガイドを読むことで、Kafka Connectorの開発に必要な知識とスキルを習得し、Kafka Connectを使用して、外部システムとKafkaクラスターをシームレスに統合することができます。

# Kafka Connect開発の利点

Kafka Connectorを開発する利点は以下の通りです。

1. データの統合：Kafka Connectorを使用することで、さまざまなデータソースを統合することができます。これにより、データを異なるシステム間で容易に移動でき、よりシームレスなデータパイプラインを構築できます。
2. 柔軟性：Kafka Connectorを使用することで、Kafkaプラットフォームと他のシステムとの接続が柔軟になります。ConnectorはJavaで書かれているため、様々なシステムやデータソースに対応することができます。
3. 拡張性：Kafka Connectorは、Kafka Connect APIを介して拡張できます。新しいConnectorを簡単に開発できるため、既存のデータソースやターゲットに加えて、新しいシステムやプラグインを追加することができます。
4. パフォーマンス：Kafka Connectorは、高速でスケーラブルなデータ処理を提供します。Connectorは、Kafkaクラスターに直接接続されるため、高速で効率的に動作します。
5. シンプルさ：Kafka Connectorは、Kafka Connectフレームワークに基づいているため、シンプルで扱いやすくなっています。これにより、開発者はConnectorを簡単に設定、実行、管理することができます。

# Kafka Connector開発手順

Kafka Connectorを開発する手順は、以下の通りです。

1. Connectorの種類を決定する

* ソースコネクタ：外部システムからデータを取得してKafkaに投入する
* シンクコネクタ：Kafkaからデータを取得して外部システムに送信する
* トランスフォームコネクタ：ソースコネクタから取得したデータを変換してKafkaに投入する、またはシンクコネクタから受信したデータを変換して外部システムに送信する

1. Connectorの設定ファイルを作成する

* Connectorの名前、クラス名、トピック、接続先、変換方法などの設定を記述する

1. Connectorの実装を作成する

* Connectorの種類に応じたクラスを実装する
* 必要に応じてタスクを実装する

1. Connectorをビルドする

* Mavenなどのビルドツールを使用して、Connectorをビルドする

1. Connectorを配置する

* Kafka Connectが実行されているホスト上に、Connectorのビルドしたファイルを配置する

1. Connectorを起動する

* Kafka Connect REST APIを使用して、Connectorを起動する

1. Connectorの動作確認を行う

* Connectorが正常に動作しているかを確認する

以上が、Kafka Connectorを開発するための基本的な手順となります。実際の開発では、設定や実装の詳細が異なる場合がありますので、公式ドキュメントやサンプルコードなどを参考にしながら開発を進めていくことが重要です。

1. 今回開発するのは、snmptrapを受信して、Kafkaに投入するコネクタなので、ソースコネクタです。
2. 設定ファイル（SnmptrapSource.properties）に必要情報を書き込みます。
3. 今回はソースコネクタなので、Configクラス、Connectorクラス、SourceTaskクラスを実装します。

※なお、ソースコードは以下のURLにあります。

https://github.com/keijijin/snmptrap-connector.git

# 設定ファイルの記述

設定ファイル（SnmptrapSourceConnector.properties）：

name=SnmptrapSourceConnector

connector.class=com.example.kafka.SnmptrapSourceConnector

bootstrap.servers=localhost:9092

topic=demo-1

key.serializer.class=org.apache.kafka.common.serialization.StringSerializer

value.serializer.class=org.apache.kafka.common.serialization.StringSerializer

snmp.host=0.0.0.0

snmp.port=162

snmp.community=public

設定ファイル（SnmptrapSourceConnector.properties）の説明：

この設定ファイルは、SnmptrapSourceConnectorという名前のConnectorを定義しています。

* **connector.class** : Connectorのクラス名を指定しています。この場合は、**com.example.kafka.SnmptrapSourceConnector**というクラスが指定されています。
* **bootstrap.servers** : Kafkaのブローカーのアドレスとポート番号を指定しています。この場合は、**localhost:9092**というアドレスが指定されています。
* **topic** : データを投入するKafkaのトピック名を指定しています。この場合は、**demo-1**というトピック名が指定されています。
* **key.serializer.class** : データのキーに使用するシリアライザーのクラス名を指定しています。この場合は、文字列を使用する**StringSerializer**が指定されています。
* **value.serializer.class** : データの値に使用するシリアライザーのクラス名を指定しています。この場合は、文字列を使用する**StringSerializer**が指定されています。
* **snmp.host** : SNMPトラップを送信するホストのIPアドレスを指定しています。この場合は、**0.0.0.0**というアドレスが指定されています。
* **snmp.port** : SNMPトラップを受信するポート番号を指定しています。この場合は、**162**というポート番号が指定されています。
* **snmp.community** : SNMPトラップのコミュニティ名を指定しています。この場合は、**public**というコミュニティ名が指定されています。

この設定ファイルは、SnmptrapSourceConnectorを起動する際に使用されます。

# Configクラスの実装

SnmptrapSourceConnectorConfig.java

package com.example.kafka;

import org.apache.kafka.common.config.AbstractConfig;

import org.apache.kafka.common.config.ConfigDef;

import java.util.Map;

*public* class SnmptrapSourceConnectorConfig extends AbstractConfig {

*public* *static* *final* String BOOTSTRAP\_SERVERS\_CONFIG = "bootstrap.servers";

*public* *static* *final* String BOOTSTRAP\_SERVERS\_DOC = "Target bootstrap servers";

*public* *static* *final* String TOPIC\_CONFIG = "topic";

*public* *static* *final* String TOPIC\_DOC = "Topic to write to";

*public* *static* *final* String SNMP\_HOST\_CONFIG = "snmp.host";

*public* *static* *final* String SNMP\_HOST\_DOC = "Snmp host which receive snmptrap";

*public* *static* *final* String SNMP\_PORT\_CONFIG = "snmp.port";

*public* *static* *final* String SNMP\_PORT\_DOC = "Snmp port which receive snmptrap";

*public* *static* *final* String SNMP\_COMMUNITY\_CONFIG = "snmp.community";

*public* *static* *final* String SNMP\_COMMUNITY\_DOC = "Snmp community string (e.g. public)";

*public* *static* *final* String KEY\_SERIALIZER\_CLASS\_CONFIG = "key.serializer.class";

*public* *static* *final* String KEY\_SERIALIZER\_CLASS\_DOC = "key serializer class";

*public* *static* *final* String VALUE\_SERIALIZER\_CLASS\_CONFIG = "value.serializer.class";

*public* *static* *final* String VALUE\_SERIALIZER\_CLASS\_DOC = "value serializer class";

*public* SnmptrapSourceConnectorConfig(ConfigDef *config*, Map<String, String> *parsedConfig*) {

super(*config*, *parsedConfig*);

}

*public* SnmptrapSourceConnectorConfig(Map<String, String> *parsedConfig*) {

super(conf(), *parsedConfig*);

}

*public* *static* ConfigDef conf() {

*return* *new* ConfigDef()

.define(BOOTSTRAP\_SERVERS\_CONFIG, ConfigDef.Type.STRING, ConfigDef.Importance.HIGH, BOOTSTRAP\_SERVERS\_DOC)

.define(TOPIC\_CONFIG, ConfigDef.Type.STRING, ConfigDef.Importance.HIGH, TOPIC\_DOC)

.define(SNMP\_HOST\_CONFIG, ConfigDef.Type.STRING, ConfigDef.Importance.HIGH, SNMP\_HOST\_DOC)

.define(SNMP\_PORT\_CONFIG, ConfigDef.Type.INT, 162, ConfigDef.Importance.HIGH, SNMP\_PORT\_DOC)

.define(SNMP\_COMMUNITY\_CONFIG, ConfigDef.Type.STRING, ConfigDef.Importance.HIGH, SNMP\_COMMUNITY\_DOC)

.define(KEY\_SERIALIZER\_CLASS\_CONFIG, ConfigDef.Type.STRING, ConfigDef.Importance.HIGH, KEY\_SERIALIZER\_CLASS\_DOC)

.define(VALUE\_SERIALIZER\_CLASS\_CONFIG, ConfigDef.Type.STRING, ConfigDef.Importance.HIGH, VALUE\_SERIALIZER\_CLASS\_DOC)

;

}

*public* String getBootstrapServersConfig() {

*return* this.getString(BOOTSTRAP\_SERVERS\_CONFIG);

}

*public* String getTopicConfig() {

*return* this.getString(TOPIC\_CONFIG);

}

*public* String getSnmpHostConfig() {

*return* this.getString(SNMP\_HOST\_CONFIG);

}

*public* Integer getSnmpPortConfig() {

*return* this.getInt(SNMP\_PORT\_CONFIG);

}

*public* String getSnmpCommunityConfig() {

*return* this.getString(SNMP\_COMMUNITY\_CONFIG);

}

*public* String getKeySerializerClassConfig() { *return* this.getString(KEY\_SERIALIZER\_CLASS\_CONFIG); }

*public* String getValueSerializerClassConfig() { *return* this.getString(VALUE\_SERIALIZER\_CLASS\_CONFIG); }

}

これは、SnmptrapSourceConnectorの設定ファイルを定義するためのクラスです。以下の機能を持っています。

* ConfigDefオブジェクトを使用して、各設定項目のデフォルト値、型、説明、優先度を定義します。
* AbstractConfigクラスを拡張して、設定ファイルのパースと検証を行います。
* 各設定項目の値を取得するためのgetterメソッドを提供します。

具体的には、以下の設定項目が定義されています。

* bootstrap.servers：Kafkaのブートストラップサーバーのアドレス
* topic：書き込み先のトピック名
* snmp.host：SNMPトラップを受信するホストのアドレス
* snmp.port：SNMPトラップを受信するポート番号（デフォルトは162）
* snmp.community：SNMPコミュニティ文字列
* key.serializer.class：キーのシリアライザのクラス名
* value.serializer.class：値のシリアライザのクラス名

これらの設定項目には、それぞれ適切な型や優先度が定義されています。また、設定ファイルをパースして値を取得するためのgetterメソッドも提供されています。これらの設定項目を使用して、SnmptrapSourceConnectorを構成することができます。

フォームの始まり

フォームの終わり

# Connectorクラスの実装

SnmptrapSourceConnector.java

package com.example.kafka;

import org.apache.kafka.common.config.ConfigDef;

import org.apache.kafka.connect.connector.Task;

import org.apache.kafka.connect.source.SourceConnector;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import java.util.ArrayList;

import java.util.List;

import java.util.Map;

*public* class SnmptrapSourceConnector extends SourceConnector {

*private* *static* Logger log = LoggerFactory.getLogger(SnmptrapSourceConnector.*class*);

*private* SnmptrapSourceConnectorConfig config;

@Override

*public* void start(Map<String, String> *map*) {

config = *new* SnmptrapSourceConnectorConfig(*map*);

}

@Override

*public* Class<? extends Task> taskClass() {

*return* SnmptrapSourceTask.*class*;

}

@Override

*public* List<Map<String, String>> taskConfigs(int *i*) {

ArrayList<Map<String, String>> configs = *new* ArrayList<>(1);

configs.add(config.originalsStrings());

*return* configs;

}

@Override

*public* void stop() {

}

@Override

*public* ConfigDef config() {

*return* SnmptrapSourceConnectorConfig.conf();

}

@Override

*public* String version() {

*return* "1.0.0";

}

}

このプログラムは、Kafka Connect用のSnmptrapSourceConnectorを実装したものです。SnmptrapSourceConnectorは、SNMPトラップを受信してKafkaトピックにプッシュするために使用されます。

このプログラムは、SourceConnectorインタフェースを実装しており、start()、taskClass()、taskConfigs()、stop()、config()、version()の6つのメソッドを提供します。

* start()メソッドは、Connectorが開始されるときに呼び出され、設定を解析し、ConnectorConfigを作成します。
* taskClass()メソッドは、Connectorが生成するTaskのクラスを返します。
* taskConfigs()メソッドは、Connectorのタスクの構成を生成します。
* stop()メソッドは、Connectorが停止されるときに呼び出されます。
* config()メソッドは、ConnectorConfigの設定を返します。
* version()メソッドは、Connectorのバージョンを返します。

このプログラムは、SnmptrapSourceTaskを実行するための構成情報を提供し、SnmptrapSourceTaskによって受信したSNMPトラップをKafkaトピックにプッシュするための設定を行います。

# SourceTaskクラスの実装

SnmptrapSourceTask.java

package com.example.kafka;

import org.apache.kafka.connect.source.SourceRecord;

import org.apache.kafka.connect.source.SourceTask;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.snmp4j.\*;

import org.snmp4j.mp.MPv3;

import org.snmp4j.security.\*;

import org.snmp4j.smi.\*;

import org.snmp4j.transport.DefaultUdpTransportMapping;

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;

import java.util.Map;

*public* class SnmptrapSourceTask extends SourceTask {

*private* Logger log = LoggerFactory.getLogger(SnmptrapSourceTask.*class*);

*private* String topic;

*private* List<SourceRecord> records = *new* ArrayList<SourceRecord>();

*private* Snmp snmp;

*public* String version() {

*return* "1.0.0";

}

*public* void start(Map<String, String> *props*) {

log.info("Configs: {}", *props*.toString());

topic = *props*.get("topic");

String snmpHost = *props*.get("snmp.host");

String snmpPort = *props*.get("snmp.port");

String snmpCommunity = *props*.get("snmp.community");

*try* {

UdpAddress address = *new* UdpAddress(snmpHost + "/" + snmpPort);

TransportMapping<UdpAddress> transport = *new* DefaultUdpTransportMapping(address);

snmp = *new* Snmp(transport);

USM usm = *new* USM(SecurityProtocols.getInstance(), *new* OctetString(MPv3.createLocalEngineID()), 0);

SecurityModels.getInstance().addSecurityModel(usm);

snmp.listen();

snmp.addCommandResponder(*new* CommandResponder() {

*public* void processPdu(CommandResponderEvent *event*) {

PDU pdu = *event*.getPDU();

*if* (pdu != null) {

String fromAddress = *event*.getPeerAddress().toString();

String[] parts = fromAddress.split("/");

String fromHost = parts[0];

String messageString = "fromHost[" + fromHost + "], " + pdu.toString();

log.debug("Received PDU: {}", messageString);

synchronized (records) {

SourceRecord record = *new* SourceRecord(null, null, topic, null, null, null, messageString);

records.add(record);

records.notifyAll();

}

}

}

});

} *catch* (IOException *e*) {

*throw* *new* RuntimeException(e);

}

}

*public* List<SourceRecord> poll() throws InterruptedException {

synchronized (records) {

*while* (records.isEmpty()) {

records.wait(1000);

}

List<SourceRecord> polledRecords = *new* ArrayList<>(records);

records.clear();

*return* polledRecords;

}

}

*public* void stop() {

*try* {

snmp.close();

synchronized (records) {

records.notifyAll();

}

} *catch* (IOException *e*) {

*throw* *new* RuntimeException(e);

}

}

}

このプログラムは、SNMPトラップをKafkaトピックに書き込むKafka Connectソースコネクタの実装です。以下は、このプログラムの詳細です。

* クラスSnmptrapSourceTaskは、org.apache.kafka.connect.source.SourceTaskを実装しています。
* startメソッドは、Kafka Connectがこのタスクを開始するときに呼び出されます。startメソッドでは、Kafka Connectが構成から取得した情報を使用して、SNMPアドレス、SNMPポート、SNMPコミュニティなどの情報を解析し、トラップ受信を待ちます。SNMPトラップが受信された場合、受信されたトラップをKafkaトピックに書き込むSourceRecordを作成します。
* pollメソッドは、Kafka Connectがこのタスクからレコードをポーリングするために使用します。この実装では、SNMPトラップが受信されるたびに、受信されたトラップを記録しておくリスト(records)に追加します。Kafka Connectがポーリングするときは、リスト内のレコードをすべて取得して、リストを空にします。
* stopメソッドは、Kafka Connectがこのタスクを停止するときに呼び出されます。この実装では、SNMP通信を停止し、ポーリングを待機しているスレッドをすべて解放します。

このソースコネクタは、SnmptrapSourceConnectorクラスとSnmptrapSourceTaskクラスの両方を使用します。SnmptrapSourceConnectorクラスは、Kafka Connectがタスクを実行するために必要な情報を解析し、SnmptrapSourceTaskクラスを指定します。SnmptrapSourceTaskクラスは、SNMPトラップを受信してKafkaトピックに書き込むための具体的な実装を提供します。

# Kafka Connectorのビルド・デプロイと実行

## 必要なもの

* JDK 11
* Maven 3.8.6
* Kafka 2.0

## snmptrap connectorのビルド

1. Snmptrap Connectorのソースコードをクローンします。

* $ git clone https://github.com/keijijin/snmptrap-connector.git

1. クローンしたディレクトリに移動し、mvnコマンドを使用してプロジェクトをビルドします。

* $ cd snmptrap-connector  
  $ mvn clean package

1. 成功すれば、targetディレクトリにsnmptrap-connector-1.0.0-SNAPSHOT-packageディレクトリが生成されます。

## snmptrap connectorのデプロイ

1. デプロイ先のディレクトリを作成します。 $ mkdir /path/to/kafka/connectors
2. targetディレクトリからsnmptrap-connector-1.0.0-SNAPSHOT-packageをコピーして、デプロイ先のディレクトリに配置します。 $ cp -r target/snmptrap-connector-1.0.0-SNAPSHOT-package /path/to/kafka/connectors/.
3. configディレクトリのSnmptrapSourceConnector.propertiesを編集し、kafkaのconfigディレクトリにコピーします。
   * 編集(必要にお応じて修正して下さい)：

* bootstrap.servers=localhost:9092  
  topic=demo-1
  + コピー：
* $ cp config/SnmptrapSourceConnector.properties /path/to/kafka/config/.

1. configディレクトリのconnect-standalone.propertiesのplugin.path=を編集します plugin.path=/path/to/connectors

## Kafka Connectの実行

1. 以下のコマンドを実行して、Kafka Connectを起動します。 $ cd /path/to/kafka $ ./bin/connect-standalone.sh config/connect-standalone.properties config/SnmptrapSourceConnector.properties
2. 成功すれば、Kafka Connectが起動し、指定されたプロパティに従ってコネクタが実行されます。 [2023-04-06 13:02:26,917] INFO Created connector SnmptrapSourceConnector (org.apache.kafka.connect.cli.ConnectStandalone:109) [2023-04-06 13:02:26,961] INFO [SnmptrapSourceConnector|task-0] WorkerSourceTask{id=SnmptrapSourceConnector-0} Source task finished initialization and start (org.apache.kafka.connect.runtime.AbstractWorkerSourceTask:271)

## snmptrap connectorのテスト

* 以下のスクリプトを実行し、snmptrapをsnmp.host:snmp.port宛に送信しましょう。 $ sudo ./scripts/send-snmp.sh 127.0.0.1 333 999 "Hello World"
* bootstrap.serverのtopicにメッセージが送られていれば成功です。 Received message from Kafka: {"schema":null,"payload":"fromHost[127.0.0.1], V1TRAP[reqestID=0,timestamp=1:54:38.52,enterprise=1.3.6.1.4.1.8072.9999.0.2.35,genericTrap=0,specificTrap=333, VBS[1.3.6.1.4.1.8072.9999.0.2.35.0.333.1 = 999; 1.3.6.1.4.1.8072.9999.0.2.35.0.333.2 = Hello World]]"}

# Kafka Connectの最適化とチューニング

Kafka Connectorの最適化とチューニングには、以下のような方法があります。

1. バッチサイズの調整: デフォルトでは、Kafka Connectは1つのメッセージを処理するために1回のリクエストを送信します。しかし、大量の小さなメッセージがある場合は、バッチサイズを増やすことで、1回のリクエストで複数のメッセージを処理することができます。これにより、リクエストのオーバーヘッドを削減し、処理速度を向上させることができます。
2. タスク数の調整: Kafka Connectは、Connectorの実行に必要なTaskを自動的にスケーリングします。しかし、Task数を調整することで、より効率的に処理を行うことができます。Task数を減らすことで、システム全体の負荷を減らし、Task数を増やすことで、処理速度を向上させることができます。
3. スロットリングの設定: Kafka Connectは、ProducerやConsumerによるリクエストをスロットリングすることができます。スロットリングを使用することで、システム全体の負荷を調整することができます。また、スロットリングを使用することで、Kafka Connect自体の負荷を調整することもできます。
4. コネクタのバッファリング: コネクタによっては、多数の小さなリクエストを送信する場合があります。このような場合は、リクエストのバッファリングを行うことで、効率的に処理を行うことができます。コネクタごとに異なるバッファリング設定を行うことができます。
5. パフォーマンスモニタリング: Kafka Connectのパフォーマンスを監視し、ボトルネックやパフォーマンスの低下を検出することが重要です。Kafka Connectのログを分析し、スループットやレイテンシを計測することで、ボトルネックを特定することができます。また、プロファイリングツールを使用することで、処理の遅延やCPU使用率の増加などを特定することができます。

以上のような方法を使用することで、Kafka Connectorのパフォーマンスを最適化し、システムの効率性を向上させることができます。

フォームの始まり

フォームの終わり

# おわりに

本ドキュメントでは、Kafka Connectorについての基本的な概念や開発方法、最適化とチューニング方法について説明してきました。Kafka Connectorは、様々なデータソースをKafkaに接続し、データの連携や分析を行う上で非常に重要な役割を果たします。

Kafka Connectorの開発にあたっては、Connectorの種類や設定、タスクのスケーリング方法、データの変換方法など、多くの要素を考慮する必要があります。また、最適化やチューニングによって、より高速で効率的な処理を実現することができます。

Kafka Connectorを利用することで、複雑なシステム間のデータ連携を簡単かつ効率的に行うことができます。本ドキュメントが、Kafka Connectorを利用する上での理解と開発の助けとなることを願っています。

# 付録：pom.xmlファイルについて

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.sample</groupId>

<artifactId>snmptrap-connector</artifactId>

<packaging>jar</packaging>

<version>1.0.0-SNAPSHOT</version>

<name>snmptrap-connector</name>

<url>http://maven.apache.org</url>

<properties>

*<!-- <kafka.version>3.4.0</kafka.version>-->*

<kafka.version>3.3.1.redhat-00008</kafka.version>

<snmp4j.version>3.7.7</snmp4j.version>

</properties>

<dependencies>

<dependency>

<groupId>org.apache.kafka</groupId>

<artifactId>kafka-clients</artifactId>

<version>${kafka.version}</version>

</dependency>

*<!-- https://mvnrepository.com/artifact/org.apache.kafka/connect-api -->*

<dependency>

<groupId>org.apache.kafka</groupId>

<artifactId>connect-api</artifactId>

<version>${kafka.version}</version>

</dependency>

*<!-- https://mvnrepository.com/artifact/org.snmp4j/snmp4j -->*

<dependency>

<groupId>org.snmp4j</groupId>

<artifactId>snmp4j</artifactId>

<version>${snmp4j.version}</version>

</dependency>

*<!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-log4j12 -->*

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.7.25</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-jar-plugin</artifactId>

<configuration>

<archive>

<manifest>

<addDefaultImplementationEntries>true</addDefaultImplementationEntries>

<addDefaultSpecificationEntries>true</addDefaultSpecificationEntries>

</manifest>

</archive>

</configuration>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>2.5.1</version>

<inherited>true</inherited>

<configuration>

<source>11</source>

<target>11</target>

</configuration>

</plugin>

<plugin>

<artifactId>maven-assembly-plugin</artifactId>

*<!-- <version>3.5.0</version>-->*

<configuration>

<descriptors>

<descriptor>src/main/assembly/package.xml</descriptor>

</descriptors>

</configuration>

<executions>

<execution>

<id>make-assembly</id>

<phase>package</phase>

<goals>

<goal>single</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

<resources>

<resource>

<directory>src/main/resources</directory>

<filtering>true</filtering>

</resource>

</resources>

</build>

</project>

このファイルはApache Mavenプロジェクトの設定ファイルです。MavenはJavaプロジェクトのビルド、テスト、デプロイを自動化するためのツールであり、プロジェクトの依存関係を管理するために使用されます。

このPOMファイルは、プロジェクトのグループID、アーティファクトID、バージョンなどのプロジェクトメタデータを定義しています。また、プロジェクトが依存しているライブラリのバージョンも定義しています。

dependenciesセクションでは、プロジェクトが依存するライブラリを定義しています。この場合、Kafkaクライアント、Kafka Connect API、snmp4j、slf4j-log4j12、およびjunitが含まれます。

buildセクションでは、Mavenがプロジェクトをビルドする方法を指定しています。例えば、maven-jar-pluginは、プロジェクトをjarファイルにパッケージするためのプラグインであり、maven-compiler-pluginは、プロジェクトをコンパイルするためのプラグインです。また、maven-assembly-pluginは、プロジェクトを単一のアーカイブファイルにパッケージするためのプラグインです。resourcesセクションでは、プロジェクトのリソースを定義しています。