Red Hat build of Apache Camel 4.4

红帽构建的 Apache Camel for Quarkus 入门

红帽构建的 Apache Camel for Quarkus 入门

Last Updated: 2024-06-29

Red Hat build of Apache Camel 4.4 红帽构建的 Apache Camel for Quarkus 入门

红帽构建的 Apache Camel for Quarkus 入门

法律通告

摘要

本指南介绍了红帽构建的 Apache Camel for Quarkus，并解释了使用红帽构建的 Apache Camel for Quarkus 创建和部署应用程序的各种方法。

目录

前言

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。详情请查看 CTO Chris Wright 的信息。

第 1 章 红帽构建的 APACHE CAMEL FOR QUARKUS 入门

本指南介绍了红帽构建的 Apache Camel for Quarkus，这是创建项目的方法以及如何开始使用红帽构建的 Apache Camel for Quarkus 构建应用程序：

1.1. RED HAT BUILD OF APACHE CAMEL FOR QUARKUS 概述

红帽构建的 Apache Camel for Quarkus 将 Apache Camel 及其大量组件库的集成功能引入 Quarkus 运行时。

使用红帽构建的 Apache Camel for Quarkus 的好处包括：

- 允许用户利用性能优势、developer joy 和 Quarkus 提供的容器第一个 ethos。
- 为许多 Apache Camel 组件提供 Quarkus 扩展。
- 利用 Camel 中带来的许多性能改进，从而降低内存占用率，减少对反映和更快的启动时间的影响。
- 您可以使用 Java DSL 定义 Camel 路由。

1.2. 工具

1.2.1. IDE 插件

Quarkus 具有大多数流行开发 IDE 的插件，它们提供 Quarkus 语言支持、代码/配置完成、项目创建向导等。该插件在每个相应的 IDE 市场中提供。

- VS Code 扩展
- Eclipse 插件 （当前不支持）
- IntelliJ 插件 （目前不支持）

检查插件文档，了解如何为您的首选 IDE 创建项目。

1.2.2. Camel 内容协助

以下插件支持编辑 Camel 路由和 **application.properties** 时的内容协助：

- VS Code Language support for Camel - Camel 扩展软件包的一部分
- Apache Camel 的调试 适配器，调试使用 Java、YAML 或 XML 本地编写的 Camel 集成。
  - 有关 开发支持范围的更多信息，请参阅开发支持覆盖范围
- Camel 的 Eclipse 桌面语言支持 - Jboss 工具的一部分
- Apache Camel IDEA 插件 （并不总是为最新版本）
- 其他 支持语言服务器 协议的用户可以选择手动安装和配置 Camel 语言服务器

1.3. 使用红帽构建的 APACHE CAMEL FOR QUARKUS 构建第一个项目

1.3.1. 概述

您可以使用 code.quarkus.redhat.com 来生成 Quarkus Maven 项目，该项目会自动添加并配置要在应用程序中使用的扩展。

本节介绍了使用 Red Hat build of Apache Camel for Quarkus 创建 Quarkus Maven 项目的过程，包括：

- 使用 code.quarkus.redhat.com创建框架应用程序
- 添加简单的 Camel 路由
- 探索应用程序代码
- 以开发模式编译应用程序
- 测试应用程序

1.3.2. 使用 code.quarkus.redhat.com 生成框架应用程序

您可以在 code.quarkus.redhat.com 上引导并生成项目。

红帽构建的 Apache Camel for Quarkus 扩展位于 'Integration' 标题下。

如果您需要额外的扩展，请使用 'search' 字段来查找它们。

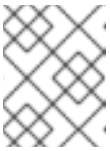
选择您要处理的组件扩展，并点击 'Generate your application' 下载基本框架项目。

您还可以直接将项目推送到 GitHub。

有关使用 **code.quarkus.redhat.com** 生成 Quarkus Maven 项目的更多信息，请参阅 *Getting started with Red Hat build of Quarkus 指南中的使用 code.quarkus.redhat.com 创建 Quarkus Maven 项目*。

流程

1. 在 code.quarkus.redhat.com 网站中，选择以下扩展：
   - **camel-quarkus-rest**
   - **camel-quarkus-jackson**
   - **camel-quarkus-direct**

   注意

   不要在 code.quarkus.redhat.com 上编译应用程序（在流程的最后一步中）。反之，使用下面 第 1.3.5 节 “开发模式” 部分描述的编译命令。

2. 进入从上一步中提取生成的项目文件的目录：

```
$ cd <directory_name>
```

1.3.3. 探索应用程序代码

应用程序有两个编译依赖项，它们在 **com.redhat.quarkus.platform:quarkus-camel-bom** 中管理，它们导入在 < **dependencyManagement>** 中。

pom.xml

```
<quarkus.platform.artifact-id>quarkus-bom</quarkus.platform.artifact-id>
<quarkus.platform.group-id>com.redhat.quarkus.platform</quarkus.platform.group-id>
<quarkus.platform.version>
    <!-- The latest 3.8.x version from
https://maven.repository.redhat.com/ga/com/redhat/quarkus/platform/quarkus-bom -->
</quarkus.platform.version>

...

<dependency>
    <groupId>${quarkus.platform.group-id}</groupId>
    <artifactId>${quarkus.platform.artifact-id}</artifactId>
    <version>${quarkus.platform.version}</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
<dependency>
    <groupId>${quarkus.platform.group-id}</groupId>
    <artifactId>quarkus-camel-bom</artifactId>
    <version>${quarkus.platform.version}</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
```

注意

有关 BOM 依赖项管理的更多信息，请参阅使用红帽构建的 Apache Camel for Quarkus 开发应用程序

该应用由 **src/main/resources/application.properties** 中定义的属性进行配置，例如，可以在其中设置 **camel.context.name**。

1.3.4. 添加简单的 Camel 路由

流程

1. 在 **src/main/java/org/acme/** 子文件夹中，创建名为 **Routes.java** 的文件。
2. 按照以下代码片段所示，添加 Camel Rest 路由：

Routes.java

```
package org.acme;

import java.util.Arrays;
import java.util.List;
import java.util.Objects;
import java.util.concurrent.CopyOnWriteArrayList;

import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.model.rest.RestBindingMode;
```

```
import io.quarkus.runtime.annotations.RegisterForReflection;

public class Routes extends RouteBuilder {
    private final List<Fruit> fruits = new CopyOnWriteArrayList<>(Arrays.asList(new
Fruit("Apple")));

    @Override
    public void configure() throws Exception {
        restConfiguration().bindingMode(RestBindingMode.json);

        rest("/customers/")
            .get("/{id}").to("direct:customerDetail")
            .get("/{id}/orders").to("direct:customerOrders")
            .post("/neworder").to("direct:customerNewOrder");

    }

    @RegisterForReflection // Let Quarkus register this class for reflection during the native
build
    public static class Fruit {
        private String name;

        public Fruit() {
        }

        public Fruit(String name) {
            this.name = name;
        }

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }

        @Override
        public int hashCode() {
            return Objects.hash(name);
        }

        @Override
        public boolean equals(Object obj) {
            if (this == obj)
                return true;
            if (obj == null)
                return false;
            if (getClass() != obj.getClass())
                return false;
            Fruit other = (Fruit) obj;
            return Objects.equals(name, other.name);
        }
```

```
    }

}
```

1.3.5. 开发模式

```
$ mvn clean compile quarkus:dev
```

此命令编译项目，启动应用程序，并允许 Quarkus 工具监视工作区中的更改。您对项目所做的任何修改将自动在正在运行的应用程序中生效。

您可以在浏览器中检查应用程序。（例如，对于 **rest-json** 示例应用程序，请访问 **http://localhost:8080/fruits**。）

例如，如果您更改了应用程序代码，请将 'Apple' 更改为 'Orange'，应用程序会自动更新。要查看应用的更改，请刷新浏览器。

有关 开发模式的详情， 请参阅 Quarkus 文档开发模式 部分。

1.3.6. 测试

1.3.6.1. JVM 模式

要测试我们在 JVM 模式中创建的 Camel Rest 路由，请添加测试类，如下所示：

流程

1. 在 **src/test/java/org/acme/** 子文件夹中，创建名为 **RoutesTest.java** 的文件。
2. 添加 **RoutesTest** 类，如以下代码片段所示：

RoutesTest.java

```
package org.acme;

import io.quarkus.test.junit.QuarkusTest;
import org.junit.jupiter.api.Test;

import static io.restassured.RestAssured.given;
import org.hamcrest.Matchers;

@QuarkusTest
public class RoutesTest {

    @Test
    public void testFruitsEndpoint() {

        /* Assert the initial fruit is there */
        given()
                .when().get("/fruits")
                .then()
                .statusCode(200)
                .body(
                        "$.size()", Matchers.is(1),
```

```
                "name", Matchers.contains("Orange"));

        /* Add a new fruit */
        given()
                .body("{\"name\": \"Pear\"}")
                .header("Content-Type", "application/json")
                .when()
                .post("/fruits")
                .then()
                .statusCode(200);

        /* Assert that pear was added */
        given()
                .when().get("/fruits")
                .then()
                .statusCode(200)
                .body(
                        "$.size()", Matchers.is(2),
                        "name", Matchers.contains("Orange", "Pear"));
    }

}
```

JVM 模式测试由 **test** Maven 阶段的 **maven-surefire-plugin** 运行：

```
$ mvn clean test
```

1.3.6.2. 原生模式

要测试我们在原生模式中创建的 Camel Rest 路由，请添加测试类，如下所示：

流程

1. 在 **src/test/java/org/acme/** 子文件夹中创建名为 **NativeRoutesIT.java** 的文件。
2. 添加 **NativeRoutesIT** 类，如以下代码片段所示：

   NativeRoutesIT.java

   ```
   package org.acme;

   import io.quarkus.test.junit.NativeImageTest;

   @NativeImageTest
   public class NativeRoutesIT extends RoutesTest {

       // Execute the same tests but in native mode.
   }
   ```

   原生模式测试由 **verify** 阶段 **Maven-failsafe-plugin** 验证。

3. 传递 **native** 属性以激活运行它们的配置集：

   ```
   $ mvn clean verify -Pnative
   ```

提示

如需了解更多详细信息，以及如何使用 **CamelTestSupport** 测试风格，请参阅 测试 Camel Quarkus 扩展。

1.3.7. 打包并运行应用程序

1.3.7.1. JVM 模式

流程

1. 运行 **mvn package** 以准备一个精简 **jar** 在库存 JVM 上运行：

```
$ mvn clean package
$ ls -lh target/quarkus-app
...
-rw-r--r--. 1 user user 238K Oct 11 18:55  quarkus-run.jar
...
```

注意

瘦 **jar** 仅包含应用程序代码。您还需要 **target/quarkus-app/lib** 中的依赖项来运行它。

2. 运行 jar，如下所示：

```
$ java -jar target/quarkus-app/quarkus-run.jar
...
[io.quarkus] (main) Quarkus started in 1.163s. Listening on: http://[::]:8080
```

注意

引导时间应该大约一秒钟。

1.3.7.2. 原生模式

流程

要准备原生可执行文件，请执行以下操作：

1. 运行命令 **mvn clean package -Pnative**:

```
$ mvn clean package -Pnative
$ ls -lh target
...
-rwxr-xr-x. 1 user user  46M Oct 11 18:57  code-with-quarkus-1.0.0-SNAPSHOT-runner
...
```

注意

运行程序 没有 **.jar** 扩展，并且设置了 **x** （可执行）权限。您可以直接运行它：

```
$ ./target/*-runner
...
[io.quarkus] (main) Quarkus started in 0.013s. Listening on: http://[::]:8080
...
```

应用程序以 13 毫秒启动。

2. 使用 **ps -o rss,command -p $(pgrep code-with)** 命令查看内存用量：

```
$ ps -o rss,command -p $(pgrep code-with)
  RSS COMMAND
65852 ./target/code-with-quarkus-1.0.0-SNAPSHOT-runner
```

该应用使用 65 MB 内存。

提示

有关准备原生可执行文件的更多信息，请参阅 Compiling *your Quarkus 应用程序到原生可执行文件指南中的 Produc ing a native executable* 。

提示

Quarkus 原生可执行文件指南 包含更多详细信息，包括 创建容器镜像的步骤。

1.4. 测试 CAMEL QUARKUS 扩展

测试提供了一种确保 Camel 路由与预期随着时间的推移的好方法。如果还没有，请阅读 Camel Quarkus 用户指南第一步，以及 Quarkus 文档链接：测试应用程序 部分。https://camel.apache.org/camel-quarkus/3.8.x/user-guide/first-steps.html

在 Quarkus 中测试路由的最简单方法是编写本地集成测试。这具有涵盖 JVM 和原生模式的优点。

在 JVM 模式中，您可以使用 **CamelTestSupport** 风格测试。

1.4.1. 在 JVM 模式下运行

在 JVM 模式中，使用 **@QuarkusTest** 注释来 bootstrap Quarkus，并在 **@Test** 逻辑执行前启动 Camel 路由。

例如：

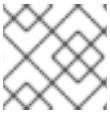
```
import io.quarkus.test.junit.QuarkusTest;
import org.junit.jupiter.api.Test;

@QuarkusTest
class MyTest {
    @Test
    public void test() {
        // Use any suitable code that sends test data to the route and then assert outcomes
        ...
    }
}
```

提示

您可以在 Camel Quarkus 源中找到示例实现：

- MessageTest.java

1.4.2. 在原生模式下运行



注意

始终测试您的应用程序是否在原生模式下对所有支持的扩展工作。

您可以通过继承相应 JVM 模式类中的逻辑来重复利用为 JVM 模式定义的测试逻辑。

添加 **@QuarkusIntegrationTest** 注释，以告知 Quarkus JUnit 扩展在测试到原生镜像下编译应用程序，并在运行测试前启动它。

```
import io.quarkus.test.junit.QuarkusIntegrationTest;

@QuarkusIntegrationTest
class MyIT extends MyTest {
    ...
}
```

提示

您可以在 Camel Quarkus 源中找到示例实现：

- MessageRecordIT.java

1.4.3. @QuarkusTest 和 @QuarkusIntegrationTest之间的区别

原生可执行文件不需要运行 JVM，且无法在 JVM 中运行，因为它是原生代码，而不是字节码。

将测试编译到原生代码时没有点，因此它们使用传统的 JVM 运行。

这意味着，测试和应用程序之间的通信必须经过网络(HTTP/REST 或应用程序发送的任何其他协议)，通过监视文件系统（例如，日志文件）或其他进程间通信。

1.4.3.1. JVM 模式中的 @QuarkusTest

在 JVM 模式中，带有 **@QuarkusTest** 标注的测试在与测试下的应用相同的 JVM 中执行。

这意味着，您可以使用 **@Inject** 将应用中的 Bean 添加到测试代码中。

您还可以使用 **@javax.enterprise.inject.Alternative** 和 **@javax.annotation.Priority** 定义新的 Bean，甚至覆盖来自应用的 Bean。

1.4.3.2. 原生模式的 @QuarkusIntegrationTest

在原生模式中，在独立于正在运行的原生应用的进程中托管的 JVM 中带有 **@QuarkusIntegrationTest** 执行的测试。

QuarkusIntegrationTest 提供了通过 **@QuarkusTest** 不提供的额外功能：

- 在 JVM 模式中，您可以启动并测试 Quarkus 构建生成的可运行的应用程序 JAR。
- 在原生模式中，您可以启动和测试 Quarkus 构建生成的原生应用程序。
- 如果您将容器镜像添加到构建中，容器会启动并根据它执行的测试。

有关 **QuarkusIntegrationTest** 的更多信息，请参阅 Quarkus 测试指南。

1.4.4. 使用外部服务测试

1.4.4.1. Testcontainers

有时，应用程序需要访问一些外部资源，如消息传递代理、数据库或其他服务。

如果容器镜像可用于感兴趣的服务，您可以使用 Testcontainers 在测试过程中启动和配置服务。

1.4.4.1.1. 使用 **QuarkusTestResourceLifecycleManager** 传递配置数据

要使应用正常工作，在启动应用程序之前，通常需要将连接配置数据（主机、端口、用户、远程服务的密码）传递给应用程序。

在 Quarkus 生态系统中，**QuarkusTestResourceLifecycleManager** 满足此目的。

您可以在 **start** （） 方法中启动一个或多个 Testcontainers，并以 **Map** 的形式从方法返回连接配置。

然后，这个映射的条目会根据模式以不同的方式传递给应用程序：

- 原生模式：命令行(**-Dkey=value**)
- JVM 模式：特殊的 MicroProfile 配置提供程序

注意

这些设置的优先级高于 **application.properties** 文件中的设置。

```
import java.util.Map;
import java.util.HashMap;

import io.quarkus.test.common.QuarkusTestResourceLifecycleManager;
import org.testcontainers.containers.GenericContainer;
import org.testcontainers.containers.wait.strategy.Wait;

public class MyTestResource implements QuarkusTestResourceLifecycleManager {

    private GenericContainer myContainer;

    @Override
    public Map<String, String> start() {
        // Start the needed container(s)
        myContainer = new GenericContainer(...)
                .withExposedPorts(1234)
                .waitingFor(Wait.forListeningPort());
```

```
        myContainer.start();

        // Pass the configuration to the application under test
        return new HashMap<>() {{
                put("my-container.host", container.getContainerIpAddress());
                put("my-container.port", "" + container.getMappedPort(1234));
        }};
    }

    @Override
    public void stop() {
        // Stop the needed container(s)
        myContainer.stop();
        ...
    }
```

*使用 **@QuarkusTestResource** 从测试类引用定义的测试资源：*

```
import io.quarkus.test.common.QuarkusTestResource;
import io.quarkus.test.junit.QuarkusTest;

@QuarkusTest
@QuarkusTestResource(MyTestResource.class)
class MyTest {
    ...
}
```

提示

您可以在 Camel Quarkus 源中找到示例实现：

- *NatsTestResource.java*

1.4.4.2. WireMock

例如，如果测试不可用、不可靠或昂贵，您可以与第三方服务和 API 存根 HTTP 交互，而不是让测试连接到实时端点。

您可以使用 WireMock 来模拟和记录 HTTP 交互。它在整个 Camel Quarkus 测试套件中广泛使用，用于各种组件扩展。

1.4.4.2.1. 设置 WireMock

流程

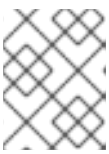
1. *设置 WireMock 服务器。*

注意

在测试中配置 Camel 组件，以通过 WireMock 代理传递 HTTP 交互。您可以通过配置决定 API 端点 URL 的组件属性来达到此目的。

```
import static com.github.tomakehurst.wiremock.client.WireMock.aResponse;
```

```
import static com.github.tomakehurst.wiremock.client.WireMock.get;
import static com.github.tomakehurst.wiremock.client.WireMock.urlEqualTo;
import static
com.github.tomakehurst.wiremock.core.WireMockConfiguration.wireMockConfig;

import java.util.HashMap;
import java.util.Map;

import com.github.tomakehurst.wiremock.WireMockServer;

import io.quarkus.test.common.QuarkusTestResourceLifecycleManager;

public class WireMockTestResource implements QuarkusTestResourceLifecycleManager {

    private WireMockServer server;

    @Override
    public Map<String, String> start() {
        // Setup & start the server
        server = new WireMockServer(
            wireMockConfig().dynamicPort()
        );
        server.start();

        // Stub a HTTP endpoint. Note that WireMock also supports a record and playback mode
        // http://wiremock.org/docs/record-playback/
        server.stubFor(
            get(urlEqualTo("/api/greeting"))
                .willReturn(aResponse()
                    .withHeader("Content-Type", "application/json")
                    .withBody("{\"message\": \"Hello World\"}")));

        // Ensure the camel component API client passes requests through the WireMock proxy
        Map<String, String> conf = new HashMap<>();
        conf.put("camel.component.foo.server-url", server.baseUrl());
        return conf;
    }

    @Override
    public void stop() {
        if (server != null) {
            server.stop();
        }
    }
}
```



注意

有时，有些操作比较简单，需要一些额外的工作来配置 API 客户端库。例如，对于 Twilio。

2. *确保您的 test 类具有* ***@QuarkusTestResource*** *注释，并将适当的 test 资源类指定为值。*

```
import io.quarkus.test.common.QuarkusTestResource;
import io.quarkus.test.junit.QuarkusTest;
```

```
@QuarkusTest
@QuarkusTestResource(WireMockTestResource.class)
class MyTest {
    ...
}
```

WireMock 服务器在所有测试执行前启动，并在所有测试完成后关闭。

提示

您可以在 Camel Quarkus 集成测试源树中找到示例实现：

- Geocoder.

1.4.5. 使用 CamelQuarkusTestSupport

从 Camel Quarkus 2.13.0 开始，您可以使用 **CamelQuarkusTestSupport** 测试。它是 **CamelTestSupport** 的替代品。

注意

这将仅在 JVM 模式中工作。

1.4.5.1. 在 JVM 模式中使用 CamelQuarkusTestSupport 测试

将以下依赖项添加到模块中（最好在 测试 范围内）：

```
<dependency>
    <groupId>org.apache.camel.quarkus</groupId>
    <artifactId>camel-quarkus-junit5</artifactId>
    <scope>test</scope>
</dependency>
```

您可以在测试中使用 **CamelQuarkusTestSupport**，如下所示：

```
@QuarkusTest
@TestProfile(SimpleTest.class) //necessary only if "newly created" context is required for the test (worse performance)
public class SimpleTest extends CamelQuarkusTestSupport {
    ...
}
```

1.4.5.2. 使用 CamelQuarkusTestSupport时的限制

使用 'CamelQuarkusTestSupport 时，有几个限制：

1.4.5.2.1. Methods

有些方法没有执行。使用以 开头 的新方法 ：

未执行	改为使用
afterAll	doAfterAll
afterEach	doAfterEach
afterTestExecution	doAfterTestExecution
beforeAll	doBeforeAll
beforeEach	doBeforeEach

注意

*如果您使用 **@TestInstance (TestInstance.Lifecycle.PER_METHOD)**，**doAfterConstruct** 代表在每次测试前的回调。这与 **beforeAll** 的不同。*

1.4.5.2.2. 注解

*您必须使用 **@io.quarkus.test.junit.QuarkusTest** 标注测试类，并扩展 **org.apache.camel.quarkus.test.CamelQuarkusTestSupport**。*

1.4.5.2.3. 启动和停止

- *您不能在单个应用程序的生命周期中停止并重启相同的 **CamelContext** 实例。您可以调用 **CamelContext.stop** （），但 **CamelContext.start** （） 无法正常工作。*
- ***CamelContext** 通常绑定到启动和停止应用程序（在测试时）。*
- *test 下的应用程序会针对给定 Maven/Gradle 模块的所有测试类启动一次。Quarkus JUnit Extension 控制应用程序的启动和停止。您必须明确告知应用程序停止。*

1.4.5.2.4. 重启应用程序

*要强制 Quarkus JUnit Extension 为给定的测试类重启应用程序和 **CamelContext**，您需要将唯一的 **@io.quarkus.test.junit.TestProfile** 分配给该类。*

具体步骤请查看 Quarkus 文档中的 测试不同配置集。

*对于类似效果，您还可以使用 **@io.quarkus.test.common.QuarkusTestResource**。*

1.4.5.2.5. Bean 生产

*Camel Quarkus 在构建阶段执行 Bean 的生产。由于测试是一起构建的，所以排除的行为是在 **CamelQuarkusTestSupport** 中实施。如果某一次测试中使用了特定类型的生产者和名称，则实例将与其余的测试相同。*

1.4.5.2.6. JUnit Jupiter 回调可能无法正常工作

这些 JUnit Jupiter 回调和注解可能无法正常工作：

回调	注解
BeforeEachCallback	@BeforeEach
AfterEachCallback	@AfterEach
AfterAllCallback	@AfterAll
BeforeAllCallback	@BeforeAll
BeforeTestExecutionCallback	
AfterTestExecutionCallback	

*如需更多信息，请参阅通过 QuarkusTest*Callback 文档 Enrichment。*

1.4.5.2.7. 使用 recommendationsWith

*当 **recommendations** With 设为 true 时，所有未建议的路由都不会启动。您必须为这些路由执行方法 **CamelQuarkusTestSupport.startRouteDefinitions** （） 以启动它们。*

1.4.5.2.8. 使用 @Produces

*使用带有覆盖方法 **createRouteBuilder** （） 的 **@Produces**。**@Produces** 和 **RouteBuilder** （） 的组合可能无法正常工作。*

1.4.5.2.9. 配置路由

*要配置应用程序中的哪些路由(**src/main/java**)来包含或排除，您可以使用：*

- ***quarkus.camel.routes-discovery.exclude-patterns***
- ***quarkus.camel.routes-discovery.include-patterns***

如需了解更多详细信息，请参阅核心 文档。

第 2 章 部署 QUARKUS 应用程序

您可以使用以下构建策略在 OpenShift 上部署 Quarkus 应用程序：

- Docker 构建
- S2I Binary
- 源 S2I

有关这些构建策略的详情，请参阅 Chapter 1。OpenShift 构建策略和 Quarkus，将 Quarkus 应用程序部署到 OpenShift Container Platform 指南。

注意

OpenShift Docker 构建策略是首选的构建策略，支持针对 JVM 的 Quarkus 应用，以及编译成原生可执行文件的 Quarkus 应用。您可以使用 **quarkus.openshift.build-strategy** 属性配置部署策略。

第 3 章 在本地设置 MAVEN

典型的红帽构建的 Apache Camel 应用程序开发使用 Maven 来构建和管理项目。

以下主题描述了如何在本地设置 Maven：

- 第 3.1 节 “准备设置 Maven”
- 第 3.2 节 “将红帽软件仓库添加到 Maven”
- 第 3.3 节 “使用本地 Maven 存储库”
- 第 3.4 节 “使用环境变量或系统属性设置 Maven 镜像”
- 第 3.5 节 “关于 Maven 工件和协调”

3.1. 准备设置 MAVEN

Maven 是一个来自 Apache 的免费开源构建工具。通常，您可以使用 Maven 构建 Fuse 应用程序。

流程

1. 从 Maven 下载页面 下载 Maven 3.8.6 或更高版本。

   提示

   要验证您已安装了正确的 Maven 和 JDK 版本，请打开终端并输入以下命令：

   ```
   mvn --version
   ```

   检查输出，以验证 Maven 版本 3.8.6 或更新版本，并使用 OpenJDK 17。

2. 确定您的系统已连接到互联网。
   在构建项目时，默认行为是 Maven 搜索外部存储库并下载所需的工件。Maven 查找可通过互联网访问的存储库。

   您可以更改此行为，以便 Maven 仅搜索位于本地网络上的存储库。也就是说，Maven 可以在离线模式下运行。在离线模式中，Maven 会在其本地存储库中查找工件。请参阅 第 3.3 节 “使用本地 Maven 存储库”。

3.2. 将红帽软件仓库添加到 MAVEN

要访问位于 Red Hat Maven 存储库中的工件，您需要将这些存储库添加到 Maven 的 **settings.xml** 文件中。

Maven 在用户主目录的 **.m2** 目录中查找 **settings.xml** 文件。如果没有用户指定的 **settings.xml** 文件，Maven 将使用 **M2_HOME/conf/ settings.xml** 中的系统级 settings.xml 文件。

前提条件

您知道要在其中添加红帽软件仓库的 **settings.xml** 文件的位置。

流程

在 **settings.xml** 文件中，为红帽 软件仓库 添加软件仓库元素，如下例所示：

注意

*如果您使用 **camel-jira** 组件，还要添加 atlassian 存储库。*

```
<?xml version="1.0"?>
<settings>

 <profiles>
  <profile>
   <id>extra-repos</id>
   <activation>
    <activeByDefault>true</activeByDefault>
   </activation>
   <repositories>
    <repository>
        <id>redhat-ga-repository</id>
        <url>https://maven.repository.redhat.com/ga</url>
        <releases>
            <enabled>true</enabled>
        </releases>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
     </repository>
     <repository>
        <id>atlassian</id>
        <url>https://packages.atlassian.com/maven-external/</url>
        <name>atlassian external repo</name>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
        <releases>
            <enabled>true</enabled>
        </releases>
     </repository>
   </repositories>
   <pluginRepositories>
    <pluginRepository>
        <id>redhat-ga-repository</id>
        <url>https://maven.repository.redhat.com/ga</url>
        <releases>
            <enabled>true</enabled>
        </releases>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
     </pluginRepository>
   </pluginRepositories>
  </profile>
 </profiles>

 <activeProfiles>
  <activeProfile>extra-repos</activeProfile>
```

```
  </activeProfiles>

</settings>
```

3.3. 使用本地 MAVEN 存储库

如果您在没有互联网连接的情况下运行容器，并且需要部署一个具有离线依赖项的应用程序，您可以使用 Maven 依赖项插件将应用的依赖项下载到 Maven 离线存储库中。然后，您可以将此自定义 Maven 离线存储库分发到没有互联网连接的机器。

流程

1. 在包含 **pom.xml** 文件的项目目录中，运行以下命令来下载 Maven 项目的存储库，如下所示：

```
mvn org.apache.maven.plugins:maven-dependency-plugin:3.1.0:go-offline -
Dmaven.repo.local=/tmp/my-project
```

   在本例中，构建项目所需的 Maven 依赖项和插件将下载到 **/tmp/my-project** 目录中。

2. 将此自定义 Maven 离线存储库在内部分发到没有互联网连接的任何机器。

3.4. 使用环境变量或系统属性设置 MAVEN 镜像

在运行应用程序时，您需要访问 Red Hat Maven 存储库中的工件。这些存储库添加到 Maven 的 **settings.xml** 文件中。Maven 检查 **settings.xml** 文件的以下位置：

- 查找指定的 url
- if not found looks for **${user.home}/.m2/settings.xml**
- 如果没有找到 **${maven.home}/conf/settings.xml**的查找
- 如果未找到，则查找 **${M2_HOME}/conf/settings.xml**
- 如果没有找到位置，则会创建空的 **org.apache.maven.settings.Settings** 实例。

3.4.1. 关于 Maven 镜像

Maven 使用一组远程存储库来访问工件，这些工件目前在本地存储库中不可用。存储库列表几乎始终包含 Maven Central 存储库，但对于 Red Hat Fuse，它还包含 Maven 红帽存储库。在某些情况下，无法访问不同的远程存储库，您可以使用 Maven 镜像的机制。镜像将特定的存储库 URL 替换为不同的存储库 URL，因此搜索远程工件时的所有 HTTP 流量都可以定向到单个 URL。

3.4.2. 在 settings.xml中添加 Maven 镜像

要设置 Maven 镜像，请在 Maven 的 **settings.xml** 中添加以下部分：

```
<mirror>
    <id>all</id>
    <mirrorOf>*</mirrorOf>
    <url>http://host:port/path</url>
</mirror>
```

如果在 **settings.xml** 文件中找不到以上部分，则不会使用 mirror。要在不提供 XML 配置的情况下指定全局镜像，您可以使用系统属性或环境变量。

3.4.3. 使用环境变量或系统属性设置 Maven 镜像

要使用环境变量或系统属性设置 Maven 镜像，您可以添加：

- 名为 **MAVEN_MIRROR_URL** 的环境变量到 **bin/setenv** 文件
- 名为 **mavenMirrorUrl** to **etc/system.properties** 文件的系统属性

3.4.4. 使用 Maven 选项指定 Maven 镜像 url

要使用替代的 Maven 镜像 url，在环境变量或系统属性之外，在运行应用程序时使用以下 maven 选项：

- **-DmavenMirrorUrl=mirrorId::mirrorUrl**
  for example, **-DmavenMirrorUrl=my-mirror::http://mirror.net/repository**
- **-DmavenMirrorUrl=mirrorUrl**
  例如，**-DmavenMirrorUrl=http://mirror.net/repository**。在本例中，<mirror> 的<id> 只是一个镜像(mirror)。

3.5. 关于 MAVEN 工件和协调

在 Maven 构建系统中，基本构建块是一个 工件。构建后，工件的输出通常是一个存档，如 JAR 或 WAR 文件。

Maven 的一个关键方面是能够找到工件和管理它们之间的依赖关系。Maven 协调 是一组用于标识特定工件位置的值。基本协调有三个值，格式为：

groupId:artifactId:version

有时，Maven 通过 打包值或使用打包 值和 分类器 值增加一个基本的协调。Maven 协调可以具有以下形式之一：

```
groupId:artifactId:version
groupId:artifactId:packaging:version
groupId:artifactId:packaging:classifier:version
```

以下是值的描述：

groupdId

定义工件名称的范围。您通常使用所有或部分软件包名称作为组 ID。例如，**org.fusesource.example**。

artifactId

定义相对于组 ID 的工件名称。

version

指定工件的版本。版本号最多可有四个部分：**n.n.n.n**，其中版本号的最后一部分可以包含非数字字符。例如，**1.0-SNAPSHOT** 的最后一部分是字母数字字符子字符串 **0-SNAPSHOT**。

打包

定义构建项目时生成的打包实体。对于 OSGi 项目，打包是 捆绑包。默认值为 **jar**。

分类器

可让您区分从同一 POM 构建但具有不同内容的工件。

工件的 POM 文件中的元素定义工件的组 ID、工件 ID、打包和版本，如下所示：

```
<project ... >
  ...
  <groupId>org.fusesource.example</groupId>
  <artifactId>bundle-demo</artifactId>
  <packaging>bundle</packaging>
  <version>1.0-SNAPSHOT</version>
  ...
</project>
```

要定义对上述工件的依赖项，您可以在 POM 文件中添加以下 dependencies 元素：

```
<project ... >
  ...
  <dependencies>
    <dependency>
      <groupId>org.fusesource.example</groupId>
      <artifactId>bundle-demo</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
  </dependencies>
  ...
</project>
```

注意

不需要在前面的依赖项中指定 **bundle** 软件包类型，因为捆绑包只是特定类型的 **JAR** 文件，**jar** 是默认的 **Maven** 软件包类型。但是，如果您需要在依赖项中明确指定打包类型，您可以使用 **type** 元素。

第 4 章 例子

下表中列出的快速入门示例可以从 **Camel Quarkus Examples** **Git** 存储库克隆或下载。

示例数：**1**

示例	描述
带有 Bindy 和 FTP 的文件消费者	演示了如何使用 CSV 文件，marshal 和 unmarshal，并通过 FTP 将其发送。

4.1. 文件消费者快速入门示例

您可以从 **Camel Quarkus Examples** **Git** 存储库下载或克隆快速入门。示例位于 **file-bindy-ftp** 目录中。

提取 **zip** 文件的内容或将存储库克隆到本地文件夹，例如一个名为 **quickstarts** 的新文件夹。

您可以在命令行中以开发模式运行此示例。通过使用开发模式，您可以快速迭代开发中的集成，并快速获得对代码的反馈。如需了解更多详细信息，请参阅 **Camel Quarkus** 用户指南中的开发模式 部分。

注意

如果您需要配置容器资源限制或启用 **Quarkus Kubernetes** 客户端来信任自签名证书，您可以在 **src/main/resources/application.properties** 文件中找到这些配置选项。

先决条件

- 有 集群管理员 对 **OpenShift** 集群的访问权限。
- 您可以访问 **SFTP** 服务器，并且您已在应用属性配置文件中设置服务器属性（由 **ftp**前缀）：**src/main/resources/application.properties**。

流程

1. 使用 **Maven** 以开发模式构建示例应用程序：

```
$ cd quickstarts/file-bindy-ftp
$ mvn clean compile quarkus:dev
```

应用程序每 **10** 秒触发定时器组件，生成一些随机的**"books"**数据，并在带有 **100** 条目的临时目录中创建 **CSV** 文件。在控制台中显示以下信息：

```
[route1] (Camel (camel-1) thread #3 - timer://generateBooks) Generating randomized books CSV data
```

接下来，**CSV** 文件由文件消费者读取，**Bindy** 用于将单个数据行放入 **Book** 对象：

```
[route2] (Camel (camel-1) thread #1 - file:///tmp/books) Reading books CSV data from 89A0EE24CB03A69-0000000000000000
```

接下来，**Book** 对象集合被分成单独的项目，并根据 **genre** 属性进行聚合：

```
[route3] (Camel (camel-1) thread #0 - AggregateTimeoutChecker) Processed 34 books for genre 'Action'
[route3] (Camel (camel-1) thread #0 - AggregateTimeoutChecker) Processed 31 books for genre 'Crime'
[route3] (Camel (camel-1) thread #0 - AggregateTimeoutChecker) Processed 35 books for genre 'Horror'
```

最后，聚合的书集合可以返回至 **CSV** 格式，并上传到测试 **FTP** 服务器。

```
[route4] (Camel (camel-1) thread #2 - seda://processed) Uploaded books-Action-89A0EE24CB03A69-0000000000000069.csv
[route4] (Camel (camel-1) thread #2 - seda://processed) Uploaded books-Crime-89A0EE24CB03A69-0000000000000069.csv
[route4] (Camel (camel-1) thread #2 - seda://processed) Uploaded books-Horror-89A0EE24CB03A69-0000000000000069.csv
```

2. 要在 **JVM** 模式下运行应用程序，请输入以下命令：

```
$ mvn clean package -DskipTests
$ java -jar target/*-runner.jar
```

3. 您可以输入以下命令来将示例应用程序构建和部署到 **OpenShift**：

```
$ mvn clean package -DskipTests -Dquarkus.kubernetes.deploy=true
```

4. 检查 **pod** 是否正在运行：

```
$oc get pods

NAME                                    READY   STATUS    RESTARTS   AGE
camel-quarkus-examples-file-bindy-ftp-1-d72mb   1/1     Running   0          5m15s
ssh-server-deployment-5f6f685658-jtr9n          1/1     Running   0          5m28s
```

5. 可选：输入以下命令监控应用程序日志：

```
oc logs -f camel-quarkus-examples-file-bindy-ftp-5d48f4d85c-sjl8k
```

其他资源

- 使用红帽构建的 **Apache Camel for Quarkus** 开发应用程序
- **Camel Quarkus** 用户指南