目录

代码开发概述

云函数（Kingsoft Cloud Function，KCF）提供代码部署环境，本文主要介绍代码部署的相关概念。

运行环境

运行环境/运行时提供针对不同语言的、在执行环境中运行的环境。

函数入口

KCF平台在调用云函数时，首先会寻找指定路径作为入口，执行用户的代码。此时，用户需在代码中以path的形式进行设置（示例代码中提供了设置方法）。

监控端口

实际处理请求时，您的 Http Server 通过监听指定的端口（默认8080端口）接收 HTTP 请求，并转发给后端服务完成逻辑处理并返回给用户。

CloudEvents信息格式规范说明

事件源（这里指的是KS3）发布事件到KCF需要按照CloudEvents规范。关于CloudEvents规范的更多信息，请参见CloudEvents 1.0。

以下是事件源发布到云函数KCF的示例事件。


```
{
  "id": "45ef4dewdwe1-7c35-447a-bd93-fab****",
    "source": "kcs:ks3",
    "type": "ks3:ObjectCreated:PutObject",
    "specversion": "1.0",
    "datacontenttype": "application/json",
    "subject": "/cc",
    "extensions": {
        "region": "BEIJING",
        "eventversion": "1.0",
        "userid": "73400852",
        "accountid": "73404680"
    },
    "data": {
        "request": {
            "sourceIPAddress": "127.0.0.1"
        },
        "response": {
            "requestId": "daab11b695ea4c4ea7a1a71ce36d1100"
        },
        "ks3": {
            "bucket": {
                "name": "kcf-pj",
                "ownerid": "73404680"
            },
            "object": {
                "internalurl": "evt-jinyw.ks3-cn-beijing-internal.ksyuncs.com/ssssss.jpg",
                "etag": "etag-xxxxxxx",
                "objectsize": "1024",
                "url": "evt-jinyw.ks3-cn-beijing.ksyuncs.com/ssssss.jpg",
                "key": "ssssss.jpg"
            }
        }
    }
}
```


CloudEvents参数说明

事件函数中涉及的CloudEvents参数说明如下所示：	参数	类型	是否必选	示例值	说明
id	String	是	45ef4dewdwe1-7c35-447a-bd93-fab****	事件ID。标识事件的唯一值。事件通过规则路由到目标，可根据id跟踪事件。	
source	String	是	kcs:ks3	事件源。提供事件的服务	

type	String	是	ks3:ObjectCreated:PutObject	事件类型。描述事件源相关的事件类型。
datacontenttype	String	是	application/json	参数data的内容形式
specversion	String	是	1.0	CloudEvents协议版本。
subject	String	是	/cc	事件主题
extensions	Struct	否	见表格下方extensions结构体	扩展属性，用于存放账号、地域等公共属性。
data	Struct	是	见表格下方data结构体	事件内容。JSON对象，内容由发起事件的服务决定。CloudEvents可能包含事件发生时由事件生产者给定的上下文，data中封装了这些信息。

extensions结构体

```
 {
        "region": "BEIJING",
        "eventversion": "1.0",
        "accountid": "7320"
 }
```

data结构体

```
{
        "request": {
            "sourceIPAddress": "127.0.0.1"
        },
        "response": {
            "requestId": "daab11b695ea4c4ea7a1a71ce36d1100"
        },
        "ks3": {
            "bucket": {
                "name": "kcf",
                "ownerid": "7340"
            },
            "object": {
                "internalurl": "evt.ks3-cn-beijing-internal.ksyuncs.com/ssssss.jpg",
                "etag": "etag-xxxxxxx",
                "objectsize": "1024",
                "url": "evt.ks3-cn-beijing.ksyuncs.com/ssssss.jpg",
                "key": "ssssss.jpg"
            }
        }
}
```

Cloudevents对象模型示例

Java

```
import lombok.Data;

@Data
public class Ks3CloudEventData {
    RequestData request;
    ResponseData response;
    Ks3Data ks3;

}

@Data
public class Ks3Data {
    Bucket bucket;
    DataObject object;
}

@Data
public class Bucket {
    String name;
    String ownerid;
}

@Data
public class DataObject {
    String internalurl;
    String etag;
    String objectsize;
    String url;
```

```
    String key;
}

@Data
public class RequestData {
    String sourceIPAddress;
}

@Data
public class ResponseData {
    String requestId;
}
```

Java运行环境

本文主要介绍Java运行环境特点及日志打印配置。

Java运行环境

Java版本选择

云函数 KCF 目前支持的 Java 开发语言包括如下版本：

- Java 8（Open JDK）

  注意事项
  Java 语言由于需要编译后才可以在 JVM 虚拟机中运行。因此在 KCF 中的使用方式，和 Python、Node.js 等脚本型语言不同，有如下限制：

  - 不支持上传代码：使用 Java 语言仅支持上传已经开发完成编译打包后的JAR包。 KCF 环境不提供 Java 的编译能力。
  - 不支持在线编辑：由于不支持上传代码，所以不支持在线编辑代码。Java 运行时的函数，在代码页面仅能看到通过页面上传或 KS3 提交代码的方法。

  ### 函数日志

  云函数与Klog日志服务集成（默认关闭，可通过控制台新建函数页面开启），云函数会将函数调用的记录以及函数代码中打印的日志全部存储到日志库中，您可以通过控制台云函数提供的调用信息模块查询函数日志，方便调试及定位问题。**日志采集服务只能采集控制台（console）输出的日志，自定义开发时避免日志从文件输出（可通过日志资源文件配置，介绍中给出示例配置文件）。**

开发语言	编程语言内嵌的打印日志语句	日志框架打印语句	日志框架实现
JAVA	System.out.println()	private static final org.slf4j.Logger log = org.slf4j.LoggerFactory.getLogger(Object.class) log.info(“Hello”);	slf4j + log4j2、slf4j + logback

slf4j+logback

非Spring项目 pom.xml依赖

```
<dependencies>
    // 其他依赖省略
    // 引入logback
    <dependency>
       <groupId>ch.qos.logback</groupId>
       <artifactId>logback-classic</artifactId>
       <version>1.2.11</version>
       </dependency>
</dependencies>
```

build.gradle依赖

```
dependencies {
   // 其他依赖省略
  implementation 'ch.qos.logback:logback-classic:1.2.11'
}
```

Spring项目 Logback作为sping boot内置日志框架，实际开发中不需要直接引入该依赖。 配置文件（src/main/resource目录下，日志输出设置为console）
logbcak.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="false">

    <!--控制台输出appender-->
    <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
        <!--设置输出格式-->
        <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
 <!--格式化输出：%d表示日期，%thread表示线程名，%-5level：级别从左显示5个字符宽度%msg：日志消息，%n是换行符-->
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} - %msg%n</pattern>
            <!--设置编码-->
            <charset>UTF-8</charset>
        </encoder>
    </appender>

    <!--指定基础的日志输出级别-->
    <root level="INFO">
        <!--appender将会添加到这个logger-->
        <appender-ref ref="console"/>
    </root>
</configuration>
```

slf4j + log4j2

非Spring项目 pom.xml依赖

```
<dependencies>
          // 其他依赖省略
          // 引入 Log4j2
          <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-slf4j-impl</artifactId>
            <version>2.17.2</version>
        </dependency>
</dependencies>
```

build.gradle依赖

```
dependencies {
  // 其他依赖省略
  implementation 'org.apache.logging.log4j:log4j-slf4j-impl:2.17.2'
}
```

Spring项目 pom.xml

```
<dependencies>
    <dependency>
      <!-- 排除 spring-boot-starter-logging 默认使用logbcak日志框架-->
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-logging</artifactId>
      <exclusions>
<exclusion>
          <groupId>*</groupId>
          <artifactId>*</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
    <!--引入 Log4j2-->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
</dependencies>
```

build.gradle Spring-boot、spring-function、spring-reactive在spring多个框架中引入了spring-boot-starter-logging依赖，可以在项目级别排除。

```
// 排除 spring-boot-starter-logging
configurations {
    compile.exclude group: 'org.springframework.boot', module: 'spring-boot-starter-logging'
    implementation.exclude group: 'org.springframework.boot', module: 'spring-boot-starter-logging'
    testImplementation.exclude group: 'org.springframework.boot', module: 'spring-boot-starter-logging'
}
dependencies {
    // 其他依赖省略
    // 引入 Log4j2
implementation 'org.springframework.boot:spring-boot-starter-log4j2'
}
```

配置文件（src/main/resource目录下，日志输出设置为console）

log4j2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN"  monitorInterval="30">
    <!-- 变量配置 -->
    <Properties>
        <!-- 日志输出格式 -->
        <property name="LOG_PATTERN"
                  value="%d{yyyy-MM-dd HH:mm:ss.SSS} %highlight{%-5level} [%t] %highlight{%c{1.}.%M(%L)}: %msg%n"/>
    </Properties>

    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="${LOG_PATTERN}"/>
            <!-- onMatch="ACCEPT" 只输出 level 级别及级别优先级更高的 Log , onMismatch="DENY" 其他拒绝输出  -->
            <ThresholdFilter level="debug" onMatch="ACCEPT" onMismatch="DENY"/>
        </Console>
    </Appenders>
    <Loggers>
        <Root level="INFO">
            <AppenderRef ref="Console"/>
        </Root>
    </Loggers>
</Configuration>
```

Http Server自定义模块代码模板

云函数通过事件触发的方式运行，触发器在触发函数时，统一采用Cloudevents数据结构作为消息传递的格式。不同的HttpServer在处理Cloudevents消息时有所不同。下面列举出常见的Http Server示例：

语言	HttpServer类型	示例代码下载	日志框架	备注
Java	Basic Http Server	示例代码下载	slf4j + logback	轻量级Jetty服务器
Java	Spring Boot	示例代码下载	slf4j + logback	快速开发Spring应用
Java	Quarkus	示例代码下载	Internally JBoss Logging	Quarkus
Java	Spring Function	示例代码下载	slf4j + logback	Spring Function
Java	Spring Reactive	示例代码下载	slf4j + logback	Spring Webflux
Java	Vert.x	示例代码下载	slf4j + logback	异步编程、非阻塞式

- 代码下载说明

  ```
  git clone https://github.com/cloudevents/sdk-java.git
  ```

  切换到tag 2.3.0：

  ```
  git checkout 2.3.0
  ```

  进入examples目录下，选择对应框架进行编译。

如果您需要使用自定义的模块，则需要在打Jar包时，将依赖与代码一起打包。下文以OpenJDK 8为例演示如何通过Maven（Gradle）编译的Java自定义模块打包到Java项目中。 安装Java和Maven（Gradle）。 关于Java的详细信息，请参见官网。 关于Maven的详细信息，请参见Installing Apache Maven。 关于Gradle的详细信息，请参见Installing Gradle。

Basic Http Server应用示例

创建一个Basic Http Java项目，示例如下：

1. 添加pom依赖

在pom.xml文件需要添加下列依赖内容：

```
<properties>
    <project.version>2.3.0</project.version>
  </properties>
```

```
 <dependencies>
        <dependency>
            <groupId>io.cloudevents</groupId>
            <artifactId>cloudevents-http-basic</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>io.cloudevents</groupId>
            <artifactId>cloudevents-json-jackson</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>org.eclipse.jetty</groupId>
            <artifactId>jetty-server</artifactId>
            <version>9.4.41.v20210516</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>2.0.0-alpha6</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
            <version>2.0.0-alpha6</version>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.22</version>
        </dependency>
    </dependencies>
```

2. 新建一个main函数

创建Server,加载handlers，启动服务start，最后加入服务器join。

```
package io.cloudevents.examples.http.basic.server;

import io.cloudevents.examples.http.basic.handler.CustomHandler;
import io.cloudevents.examples.http.basic.handler.HealthCheckHandler;
import lombok.extern.slf4j.Slf4j;
import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.server.handler.HandlerList;

import java.io.InputStream;
import java.net.InetSocketAddress;
import java.util.Properties;

@Slf4j
public class JettyServer {
    public static void main(String[] args) throws Exception {
        String serverPort = loadHttpServerPort();
        if (serverPort == null || serverPort.equals("")) {
            log.error("Usage: HTTPServer <port>");
            return;
        }
        Server server = new Server(new InetSocketAddress("localhost", Integer.parseInt(serverPort)));
        HandlerList handlerList = new HandlerList();
        handlerList.addHandler(new HealthCheckHandler());
        handlerList.addHandler(new CustomHandler());
        server.setHandler(handlerList);
        server.start();
        server.join();
    }

    public static String loadHttpServerPort() {
        Properties properties = new Properties();
        try {
            InputStream inputStream = ClassLoader.getSystemResourceAsStream("application.properties");
            properties.load(inputStream);
            String port = properties.getProperty("server.port");
            log.info("Http Server Port: {}", port);
            return port;
        } catch (Exception e) {
            log.error("Load Properties From Config error", e);
            return null;
        }
    }
}
```

3. Http Server监听端口资源配置

Http Server启动端口可以通过resources/application.properties文件进行配置。

```
server.port=8080
```

4. 添加Handler实现

用于处理接收请求的业务逻辑，您可以自定义handler进行业务处理，下面列举出2个handler示例 HealthCheckHandler用于检测函数是否正常启动

```
package io.cloudevents.examples.http.basic.handler;

import lombok.extern.slf4j.Slf4j;
import org.eclipse.jetty.http.HttpStatus;
import org.eclipse.jetty.server.Request;
import org.eclipse.jetty.server.handler.AbstractHandler;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.OutputStream;
import java.nio.charset.StandardCharsets;

@Slf4j
public class HealthCheckHandler extends AbstractHandler {

    @Override
    public void handle(String uri,
                       Request request,
                       HttpServletRequest httpServletRequest,
                       HttpServletResponse httpServletResponse) throws IOException, ServletException {
        if ("/health".equalsIgnoreCase(uri)) {
            log.info("health check");
            httpServletResponse.setContentType("application/json;charset=UTF-8");
            OutputStream outputStream = httpServletResponse.getOutputStream();
            String data = "Hands Up";
            byte[] dataByteArr = data.getBytes(StandardCharsets.UTF_8);
            outputStream.write(dataByteArr);
            httpServletResponse.setContentLength(data.length());
            httpServletResponse.setStatus(HttpStatus.OK_200);
            ((Request) request).setHandled(true);
        }
    }
}
```

CustomHandler用于处理事件函数请求

```
package io.cloudevents.examples.http.basic.handler;

import com.fasterxml.jackson.databind.ObjectMapper;
import io.cloudevents.CloudEvent;
import io.cloudevents.core.message.MessageReader;
import io.cloudevents.core.message.MessageWriter;
import io.cloudevents.examples.http.basic.Foo;
import io.cloudevents.examples.http.basic.IOUtils;
import io.cloudevents.http.HttpMessageFactory;
import lombok.extern.slf4j.Slf4j;
import org.eclipse.jetty.http.HttpStatus;
import org.eclipse.jetty.server.Request;
import org.eclipse.jetty.server.handler.AbstractHandler;

import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.UncheckedIOException;
import java.util.Enumeration;
import java.util.function.BiConsumer;
import java.util.function.Consumer;

@Slf4j

public class CustomHandler extends AbstractHandler {
    private static final ObjectMapper objectMapper = new ObjectMapper();

    @Override
    public void handle(String uri,
                       Request request,
```

```
                        HttpServletRequest httpServletRequest,
                        HttpServletResponse httpServletResponse) throws IOException, ServletException {
        if (!"/event-invoke".equalsIgnoreCase(uri)) {
            httpServletResponse.setStatus(HttpStatus.NOT_FOUND_404);
            return;
        }
        if (!"POST".equalsIgnoreCase(request.getMethod())) {
            httpServletResponse.setStatus(HttpStatus.METHOD_NOT_ALLOWED_405);
            return;
        }

        CloudEvent receivedEvent = createMessageReader(httpServletRequest).toEvent();
        log.info("begin receive event: {}", receivedEvent);
        log.info("receive event string data: {}", new String(receivedEvent.getData().toBytes()));
        Ks3CloudEventData ks3Data = objectMapper.readValue(receivedEvent.getData().toBytes(), Ks3CloudEventData.class);
        log.info("ks3Data : {}", ks3Data );

        createMessageWriter(httpServletResponse).writeBinary(receivedEvent);
        ((Request) request).setHandled(true);
    }

    private static MessageReader createMessageReader(HttpServletRequest httpServletRequest) throws IOException {
        Consumer<BiConsumer<String, String>> forEachHeader = processHeader -> {
            Enumeration<String> headerNames = httpServletRequest.getHeaderNames();
            while (headerNames.hasMoreElements()) {
                String name = headerNames.nextElement();
                processHeader.accept(name, httpServletRequest.getHeader(name));

            }
        };
        byte[] body = IOUtils.toByteArray(httpServletRequest.getInputStream());
        return HttpMessageFactory.createReader(forEachHeader, body);
    }
```

其中createMessageReader方法将httpServletRequest请求数据转换成cloudevents对象格式，请求对象和和请求头均可以通过cloudevents对象进行获取。 createMessageWriter将cloudevents对象转换成HttpServletResponse协议对象来返回响应头和响应体。

5. 编译打包

在pom.xml文件中添加maven-assembly-plugin插件（打包插件您可以自行选择，此处将maven-assembly-plugin插件作为示例）。

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.1</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
                <encoding>UTF-8</encoding>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                    <configuration>
                        <archive>
                            <manifest>
                                <mainClass>
io.cloudevents.examples.http.basic.server.JettyServer
                                </mainClass>
                            </manifest>
                        </archive>
                        <descriptorRefs>
<descriptorRef>jar-with-dependencies</descriptorRef>
                        </descriptorRefs>
                        <appendAssemblyId>false</appendAssemblyId>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

您可以使用命令将代码和及其依赖打包成可执行的jar包。

```
maven package
```

编译后的jar包位于项目文件内的target目录内，并根据pom.xml内的artifactId、version字段命名为cloudevents-basic-http-example-2.3.0.jar

6. Jar包上传云函数

- 登陆云函数控制台
- 在顶部菜单栏，选择地域和命名空间
- 在函数管理页面，选择新建函数
- 在环境配置页面，在代码上传的下拉列表下，选择上传代码包，或通过对象存储（ks3）上传方式上传打包好的Jar包。
- 监听端口和资源配置文件保持一致（默认8080）
- 点击最下方创建按钮

7. 示例支持Gradle打包编译方式

gradle使用build.gradle配置文件进行编译。build.gradle配置文件如下：

```
plugins {
    id 'java'
    id 'com.github.johnrengelman.shadow' version '7.1.2'
}

group 'org.example' version '1.0-SNAPSHOT'

sourceCompatibility = 1.8

tasks.withType(JavaCompile) {     options.encoding = 'UTF-8' }

repositories {    mavenCentral() }

dependencies {     implementation 'io.cloudevents:cloudevents-http-basic:2.3.0'
    implementation 'io.cloudevents:cloudevents-json-jackson:2.3.0'
    implementation 'org.eclipse.jetty:jetty-server:9.4.41.v20210516'     implementation 'org.slf4j:slf4j-api:2.0.0-alpha6'     implementation 'org.slf4j:slf4j-simple:2.0.0-alpha6'
    implementation "org.projectlombok:lombok:1.18.22"
    annotationProcessor "org.projectlombok:lombok:1.18.22" }

description = 'cloudevents-basic-http-example'

shadowJar {     manifest {         attributes 'Main-Class': 'io.cloudevents.examples.http.basic.JettyServer'
    } }

test {     useJUnitPlatform() }
```

其中com.github.johnrengelman.shadow插件将程序打包fat jar，包含程序运行所有依赖的Jar包，可以直接使用Java -jar 【name】运行。其中attributes 下Main-Class属性需要和代码中Main函数名称保持一致。
在项目的根目录下执行gradle shadow命令打包，编译输出如下：

```
gradle shadow
```

编译输出示例

```
BUILD SUCCESSFUL in 1s 1 actionable task: 1 executed
```

编译后的jar包位于项目文件内的build/libs目录下。
如果显示编译失败，请根据输出的编译错误信息调整代码。

Spring Boot应用示例

1. 添加pom依赖

```
<properties>
    <spring-boot.version>2.3.2.RELEASE</spring-boot.version>
    <cloudevents.version>2.3.0</cloudevents.version>
  </properties>

 <dependencyManagement>
    <dependencies>
```

```
      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-dependencies</artifactId>
        <version>${spring-boot.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <version>${spring-boot.version}</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-jersey</artifactId>
      <version>${spring-boot.version}</version>
    </dependency>
    <dependency>
      <groupId>io.cloudevents</groupId>
      <artifactId>cloudevents-core</artifactId>
      <version>${cloudevents.version}</version>
    </dependency>
    <!-- To use the json format and the cloudevent data mapper -->
    <dependency>
      <groupId>io.cloudevents</groupId>
      <artifactId>cloudevents-json-jackson</artifactId>
      <version>${cloudevents.version}</version>
    </dependency>
    <dependency>
      <groupId>io.cloudevents</groupId>
      <artifactId>cloudevents-http-restful-ws</artifactId>
      <version>${cloudevents.version}</version>
    </dependency>
    <!-- lombok log annotations -->
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <version>1.18.22</version>
    </dependency>
  </dependencies>
```

2. 添加启动类

```
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

3. 修改监听端口资源配置

在resource目录下的application.propertiesl文件中配置监听端口号等信息。

```
server.port=8080
```

4. 提供接口服务

创建一个MainResource类，提供一个简单的接口服务，用于接收cloudevents事件

```
package com.example.demo.controller;

import com.example.demo.model.Ks3CloudEventData;
import com.fasterxml.jackson.databind.ObjectMapper;
import io.cloudevents.CloudEvent;
import io.cloudevents.core.builder.CloudEventBuilder;
import io.cloudevents.core.data.PojoCloudEventData;
import io.cloudevents.jackson.PojoCloudEventDataMapper;
import lombok.extern.slf4j.Slf4j;
```

```
import org.springframework.beans.factory.annotation.Autowired;

import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import java.io.IOException;

import static io.cloudevents.core.CloudEventUtils.mapData;

@Path("/")
@Slf4j
public class MainResource {

    @Autowired
    ObjectMapper objectMapper;

    @POST
    @Path("event-invoke")
    public Response fcEventInvoke(CloudEvent inputEvent) throws Exception {
        log.info("receive event message!");
        log.info("event type: {}", inputEvent.getType());
        String ak = (String) inputEvent.getExtension("ak");
        String sk = (String) inputEvent.getExtension("sk");
        //获取extension属性
        log.info("token: " + ak + sk);
        //将data字符数据序列化对象
        PojoCloudEventData<Ks3CloudEventData> cloudEventData = mapData(inputEvent, PojoCloudEventDataMapper.from(objectMapper, Ks
3CloudEventData.class));
        if (cloudEventData == null) {
            return Response.status(Response.Status.BAD_REQUEST)
                    .type(MediaType.APPLICATION_JSON)
                    .entity("Event should contain ks3CloudEventData")
                    .build();
        }
       log.info("cloudeventdata: {}", new String(cloudEventData.toBytes()));
       Ks3CloudEventData ks3Data = cloudEventData.getValue();
       log.info("ks3data: {}", ks3Data );
       CloudEvent outputEvent = CloudEventBuilder.from(inputEvent)
           .withData("event invoke".getBytes())
           .withExtension("path", "/event-invoke")
           .build();
        log.info("cloudevent output: {}", outputEvent);
        return Response.ok(outputEvent).build();
    }

    @GET
    @Path("health")
    public Response Health() throws Exception {
        log.info("Health Up");
        return Response.ok("Up").build();
    }
}
```

其中health路径用于用户程序的健康检查，event-invoke用于事件函数请求响应逻辑，您可以自定义其他的path。

5. 添加JerseyConfiguration类

用于将cloudevents消息进行serializes/deserializes化。

```
package com.example.demo.controller;

import io.cloudevents.http.restful.ws.CloudEventsProvider;
import org.glassfish.jersey.server.ResourceConfig;
import org.springframework.context.annotation.Configuration;

@Configuration
public class JerseyConfiguration extends ResourceConfig {

    public JerseyConfiguration() {
        // Configure Jersey to load the CloudEventsProvider (which serializes/deserializes CloudEvents)
        // and our resource
        registerClasses(CloudEventsProvider.class, MainResource.class);
    }

}
```

6. Spring Boot 编译打包

在pom.xml文件中添加spring-boot-maven-plugin 插件（打包插件您可以自行选择，此处将spring-boot-maven-plugin插件

作为示例）。

```
<build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <version>${spring-boot.version}</version>
                <configuration>
                    <mainClass>com.example.demo.DemoApplication</mainClass>
                </configuration>
                <executions>
                    <execution>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
```

通过 spring-boot-maven-plugin 构建一个包含所有依赖的 jar 包（FatJar），执行命令：

```
mvn package
```

编译后的jar包位于项目文件内的target目录内，可以通过java -jar 【name】运行

7. Gradle编译打包

build.gradle配置文件如下：

```
plugins {
 id 'org.springframework.boot' version '2.3.2.RELEASE'
 id 'io.spring.dependency-management' version '1.0.11.RELEASE'
 id 'java'
}

group = 'com.example'
version = '1.0-SNAPSHOT'
sourceCompatibility = '1.8'

repositories {
 mavenCentral()
}

dependencies {
 implementation 'org.springframework.boot:spring-boot-starter-jersey'
 compileOnly 'org.projectlombok:lombok'
 annotationProcessor 'org.projectlombok:lombok'
 testImplementation 'org.springframework.boot:spring-boot-starter-test'
 implementation 'io.cloudevents:cloudevents-core:2.3.0'
 implementation 'io.cloudevents:cloudevents-json-jackson:2.3.0'
 implementation 'io.cloudevents:cloudevents-http-restful-ws:2.3.0'
}

tasks.named('test') {
 useJUnitPlatform()
}
```

在项目的根目录下执行下面命令打包，可将spring boot项目打包成一个包含所有依赖的应用程序

```
gradle bootJar
```

编译后的jar包位于项目文件内的build/libs目录下。 如果显示编译失败，请根据输出的编译错误信息调整代码。

Quarkus应用示例

1. 添加pom依赖

```
<properties>
        <quarkus-plugin.version>1.10.3.Final</quarkus-plugin.version>
        <quarkus.platform.artifact-id>quarkus-universe-bom</quarkus.platform.artifact-id>
        <quarkus.platform.group-id>io.quarkus</quarkus.platform.group-id>
        <quarkus.platform.version>1.10.3.Final</quarkus.platform.version>
        <project.version>2.3.0</project.version>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
```

```
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven-surefire-plugin.version>2.22.1</maven-surefire-plugin.version>
    </properties>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>${quarkus.platform.group-id}</groupId>
                <artifactId>${quarkus.platform.artifact-id}</artifactId>
                <version>${quarkus.platform.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.quarkus</groupId>
            <artifactId>quarkus-resteasy-jackson</artifactId>
        </dependency>
        <dependency>
            <groupId>io.quarkus</groupId>
            <artifactId>quarkus-rest-client</artifactId>
        </dependency>
        <dependency>
            <groupId>io.cloudevents</groupId>
            <artifactId>cloudevents-api</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>io.cloudevents</groupId>
            <artifactId>cloudevents-http-restful-ws</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>io.cloudevents</groupId>
            <artifactId>cloudevents-json-jackson</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>io.quarkus</groupId>
            <artifactId>quarkus-junit5</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>io.rest-assured</groupId>
            <artifactId>rest-assured</artifactId>
            <scope>test</scope>
        </dependency>
        <!-- lombok log annotations -->
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.22</version>
            <scope>compile</scope>
        </dependency>
    </dependencies>
```

2. 资源配置

在resource目录下新建application.properties文件，用于http server端口、打包方式、log日志配置

```
# Configuration file
# server port
quarkus.http.port=8080
#Uber-Jar Creation
quarkus.package.type=uber-jar
# log config
quarkus.log.console.format=%d{HH:mm:ss,SSS} %-5p [%c{2.}] (%t) %s%e%n
quarkus.log.console.level=INFO
quarkus.log.console.color=false
```

3. 提供接口服务

创建一个EventResource 类，提供一个简单的接口服务，用于接收cloudevents事件。

```
package io.cloudevents.examples.quarkus.resources;

import com.fasterxml.jackson.databind.ObjectMapper;
import io.cloudevents.CloudEvent;
import io.cloudevents.examples.quarkus.model.Ks3CloudEventData;
import io.cloudevents.jackson.PojoCloudEventDataMapper;
```

```
import lombok.extern.slf4j.Slf4j;

import javax.inject.Inject;
import javax.ws.rs.*;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;

@Path("/")
@Slf4j
public class EventResource {

    @Inject
    ObjectMapper mapper;

    @Context
    UriInfo uriInfo;

    @POST
    @Path("event-invoke")
    public Response create(CloudEvent event) {
        if (event == null || event.getData() == null) {
            throw new BadRequestException("Invalid data received. Null or empty event");
        }
        log.info("ak: {}", event.getExtension("ak"));
        Ks3CloudEventData eventData = PojoCloudEventDataMapper
                .from(mapper, Ks3CloudEventData.class)
                .map(event.getData())
                .getValue();
        log.info("Received eventData: {}", eventData);
        return Response
                .ok(uriInfo.getAbsolutePathBuilder().build(event.getId()))
                .build();
    }
}
```

创建一个HealthResource类，用户程序的健康检查。

```
package io.cloudevents.examples.quarkus.resources;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;

@Path("/health")
public class HealthResource {

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public CompletionStage<String> hello() {
        //使用异步响应
        return CompletableFuture.supplyAsync(() -> "Hands up!");
    }
}
```

您可以自定义其他的path处理业务逻辑。

4. Quarkus项目编译打包

在pom.xml文件中添加quarkus-maven-plugin插件。

```
    <build>
        <plugins>
            <plugin>
                <groupId>io.quarkus</groupId>
                <artifactId>quarkus-maven-plugin</artifactId>
                <version>1.10.3.Final</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>generate-code</goal>
                            <goal>generate-code-tests</goal>
                            <goal>build</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
```

```
    </build>
```

您可以使用命令将代码和及其依赖打包成可执行的jar包。

```
mvn clean package
```

编译后的jar包位于项目文件内的target目录内（文件名带runner字段），可以通过java -jar 【name】运行。

```
cloudevents-restful-ws-quarkus-example-2.3.0-runner.jar
```

5. Gradle编译打包

build.gradle配置文件如下： build.gradle：

```
plugins {
    id 'java'
    id 'io.quarkus' version '1.10.3.Final'
}

repositories {
    mavenLocal()
    mavenCentral()
}

dependencies {
    implementation enforcedPlatform("io.quarkus:quarkus-universe-bom:1.10.3.Final")
    implementation 'io.quarkus:quarkus-resteasy-jackson:1.10.3.Final'
    implementation 'io.quarkus:quarkus-rest-client:1.10.3.Final'
    implementation 'io.cloudevents:cloudevents-api:2.3.0'
    implementation 'io.cloudevents:cloudevents-http-restful-ws:2.3.0'
    implementation 'io.cloudevents:cloudevents-json-jackson:2.3.0'
    implementation 'org.projectlombok:lombok:1.18.22'
    testImplementation 'io.quarkus:quarkus-junit5:1.10.3.Final'
    testImplementation 'io.rest-assured:rest-assured:4.3.2'
    implementation "org.projectlombok:lombok:1.18.22"
    annotationProcessor "org.projectlombok:lombok:1.18.22"
}

group = 'io.cloudevents'
version = '2.3.0'
description = 'cloudevents-restful-ws-quarkus-example'

java {
    sourceCompatibility = JavaVersion.VERSION_1_8
    targetCompatibility = JavaVersion.VERSION_1_8
}

compileJava {
    options.encoding = 'UTF-8'
    options.compilerArgs << '-parameters'
}

compileTestJava {
    options.encoding = 'UTF-8'
}
```

在项目的根目录执行下面命令，可在本地执行应用程序

```
gradle quarkusDev
```

在项目的根目录下执行下面命令打可将项目打包成一个包含所有依赖的应用程序

```
gradle build
```

编译后的jar包位于项目文件内的build目录下（文件名带runner字段）

```
build/cloudevents-restful-ws-quarkus-example-2.3.0-runner.jar
```

如果显示编译失败，请根据输出的编译错误信息调整代码。

Spring Function应用示例

1. pom依赖

```
<properties>
   <spring-boot.version>2.4.3</spring-boot.version>
   <project.version>2.3.0</project.version>
</properties>
```

```
<dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-dependencies</artifactId>
                <version>${spring-boot.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
</dependencyManagement>

<dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-function-web</artifactId>
            <version>3.1.1</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-webflux</artifactId>
        </dependency>
        <dependency>
            <groupId>io.cloudevents</groupId>
            <artifactId>cloudevents-spring</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>io.cloudevents</groupId>
            <artifactId>cloudevents-http-basic</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>io.cloudevents</groupId>
            <artifactId>cloudevents-json-jackson</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.22</version>
        </dependency>
    </dependencies>
```

2. 添加启动类，注册序列化、反序列化配置类

```
package io.cloudevents.examples.spring;

import io.cloudevents.CloudEvent;
import io.cloudevents.core.builder.CloudEventBuilder;
import io.cloudevents.spring.messaging.CloudEventMessageConverter;
import io.cloudevents.spring.webflux.CloudEventHttpMessageReader;
import io.cloudevents.spring.webflux.CloudEventHttpMessageWriter;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.codec.CodecCustomizer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.codec.CodecConfigurer;

import java.net.URI;
import java.util.UUID;
import java.util.function.Function;

@SpringBootApplication
public class DemoApplication {

 public static void main(String[] args) throws Exception {
  SpringApplication.run(DemoApplication.class, args);
 }

 /**
  * Configure a MessageConverter for Spring Cloud Function to pick up and use to
  * convert to and from CloudEvent and Message.
  */
 @Configuration
 public static class CloudEventMessageConverterConfiguration {
```

```
  @Bean
  public CloudEventMessageConverter cloudEventMessageConverter() {
   return new CloudEventMessageConverter();
  }
 }

 /**
  * Configure an HTTP reader and writer so that we can process CloudEvents over
     * HTTP via Spring Webflux.
  */
 @Configuration
 public static class CloudEventHandlerConfiguration implements CodecCustomizer {

  @Override
  public void customize(CodecConfigurer configurer) {
   configurer.customCodecs().register(new CloudEventHttpMessageReader());
   configurer.customCodecs().register(new CloudEventHttpMessageWriter());
  }

 }

}
```

3. 监听端口资源配置

在resource目录下的application.properties文件中配置监听端口号等信息。

```
server.port=8080
```

4. 提供接口服务

创建一个MainResource类，提供一个简单的接口服务

```
package io.cloudevents.examples.spring;

import io.cloudevents.CloudEvent;
import io.cloudevents.core.builder.CloudEventBuilder;
import lombok.extern.slf4j.Slf4j;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.net.URI;
import java.util.UUID;
import java.util.function.Function;
import java.util.function.Supplier;

@Slf4j
@Configuration
public class MainResource {

    @Bean
    public Function<CloudEvent, CloudEvent> events() {
        log.info("receive event");
        return event -> CloudEventBuilder.from(event)
                .withId(UUID.randomUUID().toString())
                .withSource(URI.create("https://spring.io/foos"))
                .withType("io.spring.event.Foo")
                .withData(event.getData().toBytes())
                .build();
    }

    @Bean
    public Supplier<String> health() {
        log.info("receive health check");
        return () -> "Hands up";
    }

}
```

其中events处理事件函数请求逻辑；health用于程序本身健康检测。您可以自定义其他function进行业务处理。

5. Maven编译打包

在pom.xml文件中添加spring-boot-maven-plugin 插件（打包插件您可以自行选择，此处将spring-boot-maven-plugin插件作为示例）

```
<build>
       <plugins>
           <plugin>
               <groupId>org.springframework.boot</groupId>
```

```
                <artifactId>spring-boot-maven-plugin</artifactId>
                <version>${spring-boot.version}</version>
                <configuration>
                  <mainClass>io.cloudevents.examples.spring.DemoApplication</mainClass>
                </configuration>
                <executions>
                    <execution>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
```

通过 spring-boot-maven-plugin 构建一个包含所有依赖的 jar 包（FatJar），执行命令打包

```
maven package
```

编译后的jar包位于项目文件内的target目录内，可以通过java -jar 【name】运行。

6. Gradle编译打包

build.gradle配置文件如下：

```
plugins {
    id 'org.springframework.boot' version '2.4.3'
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    id 'java'
}
group = 'com.example'
version = '1.0-SNAPSHOT'
sourceCompatibility = '1.8'

repositories {
 mavenCentral()
}

dependencies {
    implementation 'org.springframework.cloud:spring-cloud-function-web:3.1.1'
    implementation 'org.springframework.boot:spring-boot-starter-webflux:2.4.3'
    implementation 'io.cloudevents:cloudevents-spring:2.3.0'
    implementation 'io.cloudevents:cloudevents-http-basic:2.3.0'
    implementation 'io.cloudevents:cloudevents-json-jackson:2.3.0'
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test:2.4.3'
}

description = 'cloudevents-spring-function-example'
```

在项目的根目录下执行下面命令打包

```
gradle bootJar
```

编译后的jar包位于项目文件内的build/libs目录下。 如果显示编译失败，请根据输出的编译错误信息调整代码。

Spring Reactive应用示例

1. pom依赖

```
<properties>
   <spring-boot.version>2.4.3</spring-boot.version>
   <project.version>2.3.0</project.version>
</properties>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-dependencies</artifactId>
                <version>${spring-boot.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

```
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-webflux</artifactId>
        </dependency>
        <dependency>
            <groupId>io.cloudevents</groupId>
            <artifactId>cloudevents-spring</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>io.cloudevents</groupId>
            <artifactId>cloudevents-http-basic</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>io.cloudevents</groupId>
            <artifactId>cloudevents-json-jackson</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.22</version>
        </dependency>
    </dependencies>
```

2. 添加启动类，注册序列化、反序列化配置类

```
package io.cloudevents.examples.spring;

import io.cloudevents.spring.webflux.CloudEventHttpMessageReader;
import io.cloudevents.spring.webflux.CloudEventHttpMessageWriter;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.codec.CodecCustomizer;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.codec.CodecConfigurer;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) throws Exception {
        SpringApplication.run(DemoApplication.class, args);
    }

    @Configuration
    public static class CloudEventHandlerConfiguration implements CodecCustomizer {

        @Override
        public void customize(CodecConfigurer configurer) {
            configurer.customCodecs().register(new CloudEventHttpMessageReader());
            configurer.customCodecs().register(new CloudEventHttpMessageWriter());
        }

    }

}
```

3. 监听端口资源配置

在resource目录下的application.properties文件中配置监听端口号等信息。

```
server.port=8080
```

4. 提供接口服务

创建一个MainResource类，提供一个简单的接口服务

```
package io.cloudevents.examples.spring;

import io.cloudevents.CloudEvent;
import io.cloudevents.core.builder.CloudEventBuilder;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.ResponseEntity;
```

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Mono;

import java.net.URI;
import java.util.UUID;

@Slf4j
@RestController
public class MainResource {

    @PostMapping("/event-invoke")
    // Use CloudEvent API and manual type conversion of request and response body
    public Mono<CloudEvent> event(@RequestBody Mono<CloudEvent> body) {
        log.info("receive event");
        return body.map(event -> CloudEventBuilder.from(event) //
                .withId(UUID.randomUUID().toString()) //
                .withSource(URI.create("https://spring.io/foos")) //
                .withType("io.spring.event.Foo") //
                .withData(event.getData().toBytes()) //
                .build());
    }

    @GetMapping("/health")
    //It doesn't use the `CloudEvent` data type directly, but instead models the request and response body
    public ResponseEntity<String> health() {
        log.info("receive health check");
        return ResponseEntity.ok().body("Hands up");
    }
}
```

其中event-invoke处理事件函数请求逻辑；health用于程序本身健康检测。您可以自定义其他path进行业务处理。

5. Maven编译打包

在pom.xml文件中添加spring-boot-maven-plugin 插件

```
 <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <version>${spring-boot.version}</version>
                <configuration>
                  <mainClass>io.cloudevents.examples.spring.DemoApplication</mainClass>
                </configuration>
                <executions>
                    <execution>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
```

通过 spring-boot-maven-plugin 构建一个包含所有依赖的 jar 包（FatJar），执行命令打包

```
maven package
```

编译后的jar包位于项目文件内的target目录内，可以通过java -jar 【name】运行。

6. Gradle编译打包

build.gradle配置文件如下：

```
plugins {
    id 'org.springframework.boot' version '2.4.3'
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    id 'java'
}

group = 'com.example'
version = '1.0-SNAPSHOT'
sourceCompatibility = '1.8'

repositories {
```

```
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-webflux:2.4.3'
    implementation 'io.cloudevents:cloudevents-spring:2.3.0'
    implementation 'io.cloudevents:cloudevents-http-basic:2.3.0'
    implementation 'io.cloudevents:cloudevents-json-jackson:2.3.0'
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test:2.4.3'
}

description = 'cloudevents-spring-reactive-example'
```

在项目的根目录下执行下面命令打包

```
gradle bootJar
```

编译后的jar包位于项目文件内的build/libs目录下。 如果显示编译失败，请根据输出的编译错误信息调整代码。

Vert.x应用示例

1. pom依赖

```
<properties>
        <vertx.version>4.0.0</vertx.version>
   <project.version>2.3.0</project.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>io.cloudevents</groupId>
            <artifactId>cloudevents-http-vertx</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>io.cloudevents</groupId>
            <artifactId>cloudevents-json-jackson</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>2.0.0-alpha6</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
            <version>2.0.0-alpha6</version>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.22</version>
        </dependency>
        <dependency>
            <groupId>io.vertx</groupId>
            <artifactId>vertx-web</artifactId>
            <version>${vertx.version}</version>
        </dependency>
        <dependency>
            <groupId>io.vertx</groupId>
            <artifactId>vertx-core</artifactId>
            <version>${vertx.version}</version>
        </dependency>
    </dependencies>
```

2. 监听端口资源配置

Http Server启动端口可以通过resources/application.properties文件配置

```
server.port=8080
```

3. 创建Vert.x Http Server服务，注册请求路由

```
package io.cloudevents.examples.vertx;
```

```
import io.cloudevents.examples.vertx.handle.EventInvokeHandle;
import io.cloudevents.examples.vertx.handle.HealthCheckHandle;
import io.vertx.core.AbstractVerticle;
import io.vertx.core.http.HttpServer;
import io.vertx.ext.web.Router;
import lombok.extern.slf4j.Slf4j;

import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;

@Slf4j
public class CloudEventServerVerticle extends AbstractVerticle {

    @Override
    public void start() {
        String serverPort = loadHttpServerPort();
        if (serverPort == null || serverPort.equals("")) {
            log.error("Usage: HTTPServer <port>");
            return;
        }
        final int port = Integer.parseInt(serverPort);

        // Create HTTP server.
        HttpServer server = vertx.createHttpServer();

        //创建router对象
        Router router = Router.router(vertx);

        //注册health check地址
        router.get("/health").handler(new HealthCheckHandle());
        //注册event-invoke地址
        router.post("/event-invoke").handler(new EventInvokeHandle());
        // 将请求交给路由处理
        server.requestHandler(router).exceptionHandler(System.out::println).listen(port, res -> {
            if (res.succeeded()) {
                System.out.println(
                        "Server listening on port: " + res.result().actualPort()
                );
            } else {
                System.err.println(res.cause().getMessage());
            }
        });
    }

    public static String loadHttpServerPort() {
        Properties properties = new Properties();
        InputStream inputStream = null;
        try {
            inputStream = ClassLoader.getSystemResourceAsStream("application.properties");
            properties.load(inputStream);
            String port = properties.getProperty("server.port");
            log.info("Http Server Port: {}", port);
            return port;
        } catch (Exception e) {
            log.error("Load Properties From Config error", e);
            return null;
        } finally {
            if (inputStream != null) {
                try {
                    inputStream.close();
                } catch (IOException e) {
                    log.error("Close InputStream error", e);
                }
            }
        }
    }

}
```

4. 添加Main函数，部署Vert.x服务

```
package io.cloudevents.examples.vertx;

import io.vertx.core.Vertx;

public class VertxHTTPServer {

    public static void main(String[] args) {
        Vertx.vertx().deployVerticle(new CloudEventServerVerticle());
    }
}
```

5. 添加Handler实现类，用于处理不同的路由请求

健康检查Handler

```
package io.cloudevents.examples.vertx.handle;

import io.vertx.core.Handler;
import io.vertx.ext.web.RoutingContext;
import lombok.extern.slf4j.Slf4j;

@Slf4j
public class HealthCheckHandle implements Handler<RoutingContext> {
    @Override
    public void handle(RoutingContext context) {
        log.info("receive health check");
        context.response().end("Hands Up");
    }
}
```

事件函数请求Handler

```
package io.cloudevents.examples.vertx.handle;

import io.cloudevents.CloudEventData;
import io.cloudevents.core.message.MessageReader;
import io.cloudevents.http.vertx.VertxMessageFactory;
import io.vertx.core.Handler;
import io.vertx.ext.web.RoutingContext;
import lombok.extern.slf4j.Slf4j;

import java.util.Optional;

@Slf4j
public class EventInvokeHandle implements Handler<RoutingContext> {
    @Override
    public void handle(RoutingContext context) {
        VertxMessageFactory.createReader(context.request()).map(MessageReader::toEvent)
                .onSuccess(event -> {
                    // Print out the event.
                    log.info("receive event : {}", event);
                    log.info("specVersion: {}", event.getSpecVersion());
                    Optional.ofNullable(event.getData()).map(CloudEventData::toBytes).map(String::new).ifPresent(data ->
                            log.info("receive event data: {}", data));
                    Optional.ofNullable(event.getExtension("ak")).map(Object::toString).ifPresent(System.out::println);
                    // Write the same event as response in binary mode.
                    VertxMessageFactory.createWriter(context.response()).writeBinary(event);
                })
                .onFailure(System.err::println);
    }
}
```

6. Maven编译打包

在pom.xml文件中添加maven-assembly-plugin插件

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.1</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
                <encoding>UTF-8</encoding>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                    <configuration>
                        <archive>
                            <manifest>
                                <mainClass>
                                    io.cloudevents.examples.vertx.VertxHTTPServer
```

```
                                </mainClass>
                            </manifest>
                        </archive>
                        <descriptorRefs>
                            <descriptorRef>jar-with-dependencies</descriptorRef>
                        </descriptorRefs>
                        <appendAssemblyId>false</appendAssemblyId>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>

</build>
```

使用下面命令将代码和及其依赖打包成可执行的jar包。

```
maven package
```

编译后的jar包位于项目文件内的target目录内。

7. Gradle编译打包

build.gradle配置文件如下：

```
plugins {
    id 'java'
    id 'com.github.johnrengelman.shadow' version '7.1.2'
}

group 'org.example'
version '1.0-SNAPSHOT'

sourceCompatibility = 1.8

tasks.withType(JavaCompile) {
    options.encoding = 'UTF-8'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'io.cloudevents:cloudevents-http-vertx:2.3.0'
    implementation 'io.cloudevents:cloudevents-json-jackson:2.3.0'
    implementation 'org.slf4j:slf4j-api:2.0.0-alpha6'
    implementation 'org.slf4j:slf4j-simple:2.0.0-alpha6'
    implementation 'io.vertx:vertx-web:4.0.0'
    implementation 'io.vertx:vertx-core:4.0.0'
    implementation "org.projectlombok:lombok:1.18.22"
    annotationProcessor "org.projectlombok:lombok:1.18.22"
}

shadowJar {
    manifest {
        attributes 'Main-Class': 'io.cloudevents.examples.vertx.VertxHTTPServer'
    }
}

test {
    useJUnitPlatform()
}
```

在项目的根目录下执行下面命令打包

```
gradle shadow
```

编译后的jar包位于项目文件内的build/libs目录下。 如果显示编译失败，请根据输出的编译错误信息调整代码。