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。

启动命令

配置程序的启动命令用于启动您的函数，执行路径为您上传的代码包的根目录。如java -jar demo.jar。

监听端口

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。

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）
- Java 11（Open JDK）
- Java 17（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。

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>
```

使用下面命令将代码和及其依赖打包成可执行的jar包，编译后的jar包位于项目文件内的target目录内。

```
maven package
```

编译好的jar包需用zip工具进行压缩，用于后续控制台的代码包上传。

6. 代码包上传云函数

- 登录云函数控制台
- 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**
- 在环境配置中，代码上传方式选择**通过zip包上传**，或**通过对象存储（ks3）上传**
- 完善其他函数配置，点击**创建**按钮完成函数创建。

7. Gradle编译打包

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-Java 8/11

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>
```

使用下面命令将代码和及其依赖打包成可执行的jar包，编译后的jar包位于项目文件内的target目录内。

```
maven package
```

编译好的jar包需用zip工具进行压缩，用于后续控制台的代码包上传。

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。

Spring Boot应用示例-Java 17

Spring Boot 2.5.5以前版本不支持Java17，需要使用高于此版本的Spring Boot进行开发。

1. 添加pom依赖

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.5.5</version>
    <relativePath/>
</parent>
<modelVersion>4.0.0</modelVersion>

<groupId>com.ksyun.kcf</groupId>
<artifactId>cloudevents-spring-boot-example</artifactId>
<version>1.0</version>

<properties>
    <spring-boot.version>2.5.5</spring-boot.version>
    <spring.version>5.2.8.RELEASE</spring.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>17</java.version>
    <cloudevents.version>2.3.0</cloudevents.version>
</properties>

<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>
    <dependency>
      <!-- 排除 spring-boot-starter-logging -->
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-logging</artifactId>
      <exclusions>
        <exclusion>
          <groupId>*</groupId>
          <artifactId>*</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
    <!-- 其他依赖省略 -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
```

```
      <artifactId>spring-boot-starter-log4j2</artifactId>
    </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 Application {

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

3.修改监听端口资源配置

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，提供几个简单的接口服务。

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

import com.example.demo.model.Bucket;
import com.example.demo.model.Ks3CloudEventData;
import com.example.demo.model.ResponseData;
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.*;
import javax.ws.rs.core.Response;
import java.util.UUID;

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

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

    public static final String EVENT_TYPE = "kcf";

    @Autowired
    ObjectMapper objectMapper;

    @POST
    @Path("event-invoke")
    public Response fcEventInvoke(CloudEvent inputEvent) throws Exception {
        log.info("receive event message, event type: {}", inputEvent.getType());
        //将data字符数据序列化对象
        PojoCloudEventData<Ks3CloudEventData> cloudEventData = mapData(inputEvent, PojoCloudEventDataMapper.from(objectMapper, Ks
3CloudEventData.class));
        log.info("cloudeventdata: {}", new String(cloudEventData.toBytes()));
        Ks3CloudEventData userData = cloudEventData.getValue();
        log.info("user data:  {}", userData);
        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();
    }
```

```
    @POST
    @Path("http-invoke")
    @Produces({"application/json"})
    public Response fcHttpPostInvoke(Bucket bucket) throws Exception {
        log.info("receive http post message: {}", bucket);
        ResponseData responseData = new ResponseData();
        responseData.setRequestId(UUID.randomUUID().toString());
        return Response.ok(responseData).build();
    }

    @GET
    @Path("http-invoke")
    public Response fcHttpGetInvoke() throws Exception {
        log.info("receive http get message");
        return Response.ok("Get message Ok").build();
    }

    @PUT
    @Path("http-invoke/{id}")
    public Response fcHttpPutInvoke(@PathParam("id") String id) throws Exception {
        log.info("receive http put message: {}", id);
        return Response.ok("Put message Ok").build();
    }

    @DELETE
    @Path("http-invoke/{id}")
    public Response fcHttpDeleteInvoke(@PathParam("id") String id) throws Exception {
        log.info("receive http delete message: {}", id);
        return Response.ok("Delete message Ok").build();
    }

    @PATCH
    @Path("http-invoke/{id}")
    public Response fcHttpPatchInvoke(@PathParam("id") String id) throws Exception {
        log.info("receive http patch message: {}", id);
        return Response.ok("Patch message Ok").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，http-invoke用于http请求响应逻辑，您可以自定义其他path。

5.添加JerseyConfiguration类

用于将cloudevents消息进行serializes/deserializes。

```
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.Application</mainClass>
        </configuration>
        <executions>
          <execution>
            <goals>
```

```
            <goal>repackage</goal>
          </goals>
        </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，编译后的jar包位于项目文件内的target目录内。

```
maven package
```

编译好的jar包需用zip工具进行压缩，用于后续控制台的代码包上传。

7.Gradle编译打包

build.gradle配置文件如下

```
plugins {
    id 'org.springframework.boot' version '2.5.5'
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    id 'java'
}

group = 'com.example'
version = '1.0-SNAPSHOT'
sourceCompatibility = '17'

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:2.5.5'
    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包位于项目文件夹内的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>
```

使用下面命令将代码和及其依赖打包成可执行的jar包，编译后的jar包位于项目文件内的target目录内。

```
maven package
```

编译好的jar包需用zip工具进行压缩，用于后续控制台的代码包上传。

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>
```

使用下面命令将代码和及其依赖打包成可执行的jar包，编译后的jar包位于项目文件内的target目录内。

```
maven package
```

编译好的jar包需用zip工具进行压缩，用于后续控制台的代码包上传。

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>
```

使用下面命令将代码和及其依赖打包成可执行的jar包，编译后的jar包位于项目文件内的target目录内。

```
maven package
```

编译好的jar包需用zip工具进行压缩，用于后续控制台的代码包上传。

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，编译后的jar包位于项目文件内的target目录内。

```
maven package
```

编译好的jar包需用zip工具进行压缩，用于后续控制台的代码包上传。

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。

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。

> 注意：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>
```

请求调用示例

事件请求示例

- 健康检查

  ```
  curl --location --request GET 'http://127.0.0.1:8080/health'
  ```

- 发送Cloudevent

  ```
  curl --location --request POST 'http://127.0.0.1:8080/event-invoke' \
  --header 'Content-type:  application/json' \
  --header 'Ce-id:  1' \
  --header 'Ce-source:  cloud-event-example' \
  --header 'Ce-type: kcf' \
  --header 'Ce-specversion:  1.0' \
  --data '{
    "username": "slinkydeveloper",
    "firstName": "Francesco",
    "lastName": "Guardiani",
    "age": 23
  }'
  ```

HTTP请求调用示例

- GET调用

  ```
  curl -X GET http://127.0.0.1:80880/http-invoke
  ```

- POST调用

  ```
  curl -X POST http://127.0.0.1:80880/http-invoke
  ```

- PUT调用

  ```
  curl -X PUT http://127.0.0.1:80880/http-invoke/1
  ```

- PATCH调用

  ```
  curl -X PATCH http://127.0.0.1:80880/http-invoke/1
  ```

- DELETE调用

  ```
  curl -X DELETE http://127.0.0.1:80880/http-invoke/1
  ```

Golang运行环境

环境说明

云函数已支持Go 1版本，推荐使用Go 1.8或以上版本。

请求处理程序开发方法

本文介绍在云函数KCF中使用Golang语言开发请求处理程序的相关概念和方法。

请求处理程序分为事件请求处理程序（Event Handler）和HTTP请求处理程序（HTTP Handler）；其中事件请求由各种事件源触发生成，HTTP请求则由HTTP触发器触发生成。

请求处理程序的具体配置示例如下：

事件请求处理程序（Event Handler）

介绍Go事件请求处理程序的结构和特点。

使用示例

在Go语言的代码中，使用go mod引入cloudevents官方的SDK库 ：

```
go get github.com/cloudevents/sdk-go/v2@v2.6.0
```

在开发代码中导入cloudevents包依赖

```
import cloudevents "github.com/cloudevents/sdk-go/v2"
```

cloudevents + gin框架 代码示例

```
package main

import (
    cloudevents "github.com/cloudevents/sdk-go/v2"
    "github.com/gin-gonic/gin"
    "io/ioutil"
    "log"
)

func main() {
    r := gin.Default()
    r.GET("/health", func(c *gin.Context) {
        c.JSON(200, gin.H{
            "message": "health up",
        })
    })
    r.POST("/event-invoke", eventHandlerDirect)
    r.NoRoute(func(c *gin.Context) {
        c.JSON(404, gin.H{
            "message": "not found",
        })
    })
    log.Printf("will listen on :8080\n")
    if err := r.Run(":8080"); err != nil {
        log.Fatalf("unable to start http server, %s", err)
    }
}

func eventHandlerDirect(c *gin.Context) {
    log.Printf("receive cloudevent")
```

```
    // Unmarshal JSON To cloudevents
    event := cloudevents.NewEvent()
    bytes, _ := ioutil.ReadAll(c.Request.Body)
    err := event.UnmarshalJSON(bytes)
    //err := c.Bind(&event)
    if err != nil {
        log.Printf("Unmarshal cloutevent error: %s", err.Error())
        c.JSON(400, gin.H{
            "message": err.Error(),
        })
        return
    }
    // decode body data
    data := &Sample{}
    if err := event.DataAs(data); err != nil {
        log.Printf("failed to get playload data: %s", err)
        c.JSON(400, gin.H{
            "message": "bad data",
        })
        return
    }
    log.Printf("get playload data: %s\n", data)
    log.Printf("----------------------------\n")
    c.JSON(200, gin.H{
        "message": "receive cloudevent success",
    })
}
```

示例解析如下：

- package main: 在Go语言中，Go应用程序都包含一个名为main的包。
- func main()：运行函数代码的入口点，Go程序必须包含main函数。
- import：需要引用的依赖包，主要包括以下包：
  - github.com/cloudevents/sdk-go/v2： cloudevents的核心库。
  - github.com/gin-gonic/gin Go Web服务器。
- 处理事件请求的方法（即Event Handler），参数含义如下：
  - context：云函数Go语言的Context对象。
- data结构体

```
type Sample struct {
Request  Request  `json:"request,omitempty"`
Response Response `json:"response,omitempty"`
Ks3      *Ks3     `json:"ks3,omitempty"`
}
type Request struct {
SourceIPAddress string `json:"sourceIPAddress,omitempty"`
}
type Response struct {
RequestID string `json:"requestId,omitempty"`
}
```

type Ks3 struct { Bucket Bucket `json:"bucket,omitempty"` Object Object `json:"object,omitempty"` }

type Bucket struct { Name string `json:"name,omitempty"` Ownerid string `json:"ownerid,omitempty"` } type Object struct { Internalurl string `json:"internalurl,omitempty"` Etag string `json:"etag,omitempty"` Objectsize string `json:"objectsize,omitempty"` URL string `json:"url,omitempty"` Key string `json:"key,omitempty"` }

cloudevents详细介绍：

- [doc](https://github.com/cloudevents/spec/blob/v1.0/spec.md)
- [samples](https://github.com/cloudevents/sdk-go/tree/main/samples/http)

HTTP请求处理程序（HTTP Handler）

介绍Go Http请求处理程序的结构和特点

使用示例

代码示例

package main

import ("github.com/gin-gonic/gin" "io/ioutil" "log")

// go build with vendor // go build -mod=vendor -o gin cmd/gin_http/main.go

func main() { r := gin.Default() r.GET("/health", func(c *gin.Context) { c.JSON(200, gin.H{ "message": "health up", }) }) r.POST("/http-invoke", postHandler) r.GET("/http-invoke", getHandler) r.DELETE("/http-invoke",

```
deleteHandler) r.PUT("/http-invoke", putHandler) r.PATCH("/http-invoke", patchHandler) r.NoRoute(func(c
gin.Context) { c.JSON(404, gin.H{ "message": "not found", }) }) log.Printf("will listen on :8080\n") if err
:= r.Run(":8080"); err != nil { log.Fatalf("unable to start http server, %s", err) } }

func postHandler(c *gin.Context) { log.Printf("receive http post message") // do something with api request
bytes, _ := ioutil.ReadAll(c.Request.Body) log.Printf("post body: %s", string(bytes)) c.JSON(200, gin.H{
"message": "receive http post message", }) }

func getHandler(c *gin.Context) { log.Printf("receive http get message") // do something with api request
c.JSON(200, gin.H{ "message": "receive http get message", }) }

func putHandler(c *gin.Context) { log.Printf("receive http put message") // do something with api request
c.JSON(200, gin.H{ "message": "receive http put message", }) }

func patchHandler(c *gin.Context) { log.Printf("receive http patch message") // do something with api request
c.JSON(200, gin.H{ "message": "receive http patch message", }) }

func deleteHandler(c *gin.Context) { log.Printf("receive http delete message") // do something with api
request c.JSON(200, gin.H{ "message": "receive http delete message", }) }
```

示例解析如下：

```
+ package main: 在Go语言中，Go应用程序都包含一个名为main的包。
+ func main()：运行函数代码的入口点，Go程序必须包含main函数。
+ import：需要引用函数计算依赖的包，主要包括以下包：
  * github.com/gin-gonic/gin Go Web服务器。
+ 处理HTTP请求的方法（即HTTP Handler），参数含义如下：
  * context：函数计算Go语言的Context对象。
```

编译与部署

Golang环境的云函数，仅支持zip包上传，您可以选择使用本地上传zip包或通过ks3对象存储引用zip包。zip包内包含的应该是编译后的可执行二进制文件。

编译

Golang 编译可以在任意平台上通过指定OS及ARCH完成跨平台的编译，因此在Linux、Windows或MacOS下都可以进行编译。 针对不同环境编译说明如下：

在Linux或macOS下编译

设置GOOS=linux，确保编译后的可执行文件与云函数平台的Go运行系统环境兼容，尤其是在非Linux环境中编译时。

- 针对Linux操作系统，建议使用纯静态编译，配置CGO_ENABLED=0，确保可执行文件不依赖任何外部依赖库（如libc库），避免出现编译环境和Go运行时环境依赖库的兼容问题。示例如下：

```
GOOS=linux CGO_ENABLED=0 go build -o main main.go
```

- 针对M1 macOS（或其他ARM架构的机器） 配置GOARCH=amd64，实现跨平台编译，示例如下：

```
GOOS=linux GOARCH=amd64 go build main.go
```

在 Windows下编译

按 Windows + R 打开运行窗口，输入 cmd 后按 Enter。

```
- set GOOS=linux
- set GOARCH=amd64
- go build -o main main.go
```

打包

使用zip工具打包上一步生成的二进制文件，注意二进制文件需要在 zip 包根目录。

```
zip main.zip main
```

函数日志

您可以在程序中使用log库或fmt库中的方法打印日志：

- fmt.Println
- log.Printf

例如，执行以下代码，可以在函数日志中查询输出内容

```
func eventReceiver(event cloudevents.Event) (*cloudevents.Event, error) {
    // do something with event.
    log.Printf("receive event message")
    fmt.Printf("%s", event)
    return nil, nil
}
```

请求调用示例

事件请求调用示例

- 健康检查

```
curl --location --request GET 'http://127.0.0.1:8080/health'
```

- 发送Cloudevent:

```
curl --location --request POST 'http://127.0.0.1:8080/event-invoke' \
--header 'Content-type: application/cloudevents+json' \
--data '{
    "id": "1",
    "source": "kcs:ks3",
    "type": "ks3:ObjectCreated:PutObject",
    "specversion": "1.0",
    "datacontenttype": "application/json",
    "subject": "/",
    "comValue1": "extension value1",
    "comValue2": "extension value2",
    "data": {
        "request": {
            "sourceIPAddress": "127.0.0.1"
        },
        "response": {
            "requestId": "daab11b695ea4c4ea7a1a71ce36d1100"
        },
        "ks3": {
            "bucket": {
                "name": "kcf-pkg",
                "ownerid": "7340"
            },
            "object": {
                "internalurl": "evt-jin.ks3-cn-beijing-internal.ksyuncs.com/a.jpg",
                "etag": "etag-xxxxxxx",
                "objectsize": "1024",
                "url": "evt-jin.ks3-cn-beijing.ksyuncs.com/a.jpg",
                "key": "a.jpg"
            }
        }
    }
}'
```

HTTP请求调用示例

- GET调用

  ```
  curl -X POST http://127.0.0.1:8080/http-invoke
  ```

- POST调用

  ```
  curl -X POST http://127.0.0.1:80880/http-invoke
  '{
  "id": "a123",
  }'                          //用户自定义body内容
  ```

- PUT调用

  ```
  curl -X PUT http://127.0.0.1:8080/http-invoke
  ```

- PATCH调用

  ```
  curl -X PATCH http://127.0.0.1:8080/http-invoke
  ```

- DELETE调用

```
curl -X DELETE http://127.0.0.1:8080/http-invoke
```

Python运行环境

Python版本

云函数目前支持的Python运行时环境如下。

版本	操作系统	架构
Python 3.10	Linux	x86_64

相关环境变量

目前运行环境中内置的 Python 相关环境变量见下表：

环境变量 Key	具体值或值来源
PYTHONPATH	/opt/python

请求处理程序开发方法

本文介绍在云函数KCF中使用Python语言开发请求处理程序的相关概念和方法。

请求处理程序分为事件请求处理程序（Event Handler）和HTTP请求处理程序（HTTP Handler）；其中事件请求由各种事件源触发生成，HTTP请求则由HTTP触发器触发生成。 请求处理程序的具体配置示例如下：

事件请求处理程序（Event Handler）

介绍Python事件请求处理程序的结构和特点。

使用示例

在Python语言的代码中，使用pip引入Flask Web框架时，同时引入cloudevents官方的SDK库 ：

```
pip3 install flask -t .
pip3 install cloudevents -t .
```

flask框架 + cloudevents 代码示例

```
import json
import os
# import sys
# import flask dependency
#         - name: PYTHONPATH
#           value: /opt/python
# sys.path.append('/opt/python')
from cloudevents.http import from_http
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route("/event-invoke", methods=["POST"])
def event_invoke():
    # create a CloudEvent
    event = from_http(request.headers, request.get_data())

    # you can access cloudevent fields as seen below
    print(
        f"Found {event['id']} from {event['source']} with type "
        f"{event['type']} and specversion {event['specversion']}"
    )

    return "receive event message", 200

@app.route("/health")
def health_check():
    return "<p>Hello, World Flask!</p>", 200

if __name__ == "__main__":
    # app.run(port=8080)
    app.run(host='0.0.0.0', port=8080, debug=True)
```

requirements.txt

```
cloudevents==1.6.1
Flask==2.1.2
```

Event Handler的示例解析如下：

- handler函数。与函数计算控制台配置的触发器请求路径相对应。例如，为KCF函数配置的handler为event-invoke，那么函数计算会去加载main.py中定义的handler函数，并从handler函数开始执行。
- from_http: 将事件请求内容转换成cloudevents结构体。

cloudevents详细介绍：

- doc
- samples

HTTP请求处理程序（HTTP Handler）

介绍Python HTTP请求处理程序的结构和特点。

在Python语言的代码中，使用pip引入Flask web框架，J将Flask库安装到当前目录下 ：

```
pip3 install flask -t .
```

flask框架代码示例

```
import json
import os
# import flask dependency with layer
# sys.path.append('/opt/python')
from cloudevents.http import from_http
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route("/http-invoke", methods=["GET"])
def get_handler():
    print("receive http get msg")
    print('request.args = ', request.args)
    return "receive get msg", 200

@app.route("/http-invoke", methods=["POST"])
def post_handler():
    print("receive http post msg")
    print('request.args = ', request.args)
    data = request.stream.read()
    if data is None or data == b"":
        # Empty string will cause data to be marshalled into None
        return "body data  is none", 400
    json_data = json.loads(data)
    print('request.body = ', json_data['id'])
    # print('id =', data['id'])
    # return "receive post msg", 200
    return jsonify({'id': "a123"}), 200

@app.route("/http-invoke", methods=["PUT"])
def put_handler():
    print("receive http put msg")
    print('request.args = ', request.args)
    return "receive put msg", 200

@app.route("/http-invoke", methods=["delete"])
def delete_handler():
    print("receive http delete msg")
    print('request.args = ', request.args)
    return "receive delete msg", 200

@app.route("/http-invoke", methods=["PATCH"])
def patch_handler():
    print("receive http patch msg")
    print('request.args = ', request.args)
    return "receive patch msg", 200

def print_env():
    env_dist = os.environ  # environ是在os.py中定义的一个dict environ = {}
    for key in env_dist:
        print(key + ' : ' + env_dist[key])
```

```
if __name__ == "__main__":
    # app.run(port=8080)
    app.run(host='0.0.0.0', port=8080, debug=True)
```

requirements.txt

```
Flask==2.1.2
```

HTTP Handler的示例解析如下：

- handler：方法名称。与函数计算控制台配置的请求路径（函数入口）相对应。例如，为FC函数配置的handler为/http-invoke，那么函数计算会去加载main.py中定义的handler函数，并从handler函数开始执行。
- return "receive patch msg", 200: 返回信息。此返回值将作为HTTP响应的Body返回给客户端。

安装依赖与打包

安装依赖

使用pip直接安装依赖

```
pip3 install flask -t .
```

使用requirements文件生成依赖

1. 在 `requirements.txt` 中配置依赖信息。例如示例中使用到的依赖如下： requirements.txt

```
Flask==2.1.2
```

生成requirements.txt方法：

- 方法1 生成当前pythons环境下所有类库依赖的 requirements.txt 文件

  ```
  pip freeze > requirements.txt   //包括当前项目项目中未使用的类库
  ```

- 方法2 生成当前项目中的使用到的安装包

  ```
  pip3 install pipreqs
  pipreqs . --encoding=utf8 --force
  ```

2. 通过在代码目录下执行 `pip3 install -r requirements.txt -t .` 命令安装依赖包。通过使用 `-t` 参数，可以指定依赖包的安装目录，可以使用 `-t .` 安装在当前目录下。

打包

在Linux或macOS下编译

进入代码目录，执行`zip code.zip -r ./*`。

在 Windows下打包

进入代码目录，选中所有文件，单击鼠标右键，选择打包为ZIP包。

> 注意：您在Windows系统或macOS系统上安装emoji依赖库时可能会带有二进制文件，会导致您的代码包上传到函数计算后运行失败。此时，建议您在linux环境下安装依赖并打包。

函数日志

打印日志

函数往标准输出stdout打印的日志内容会被收集到创建函数时指定的日志池中，您可以使用以下方式打印日志。

- print
- logging

使用print打印日志

例如，执行以下代码，可以在函数日志中查询输出内容

```
def my_handler():
    print("hello world")
```

使用logging模块打印日志

例如，执行以下代码，可以在函数日志中查询输出内容

```
import logging
logging.basicConfig(level=logging.INFO, stream=sys.stdout)
logger = logging.getLogger()

def my_handler():
    logger.info('hello world')
```

请求调用示例

事件请求示例

- 健康检查

```
curl --location --request GET 'http://127.0.0.1:8080/health'
```

- 发送Cloudevent:

```
curl --location --request POST 'http://127.0.0.1:8080/event-invoke' \
--header 'Content-type: application/cloudevents+json' \
--data '{
    "id": "1",
    "source": "kcs:ks3",
    "type": "ks3:ObjectCreated:PutObject",
    "specversion": "1.0",
    "datacontenttype": "application/json",
    "subject": "/",
    "comValue1": "extension value1",
    "comValue2": "extension value2",
    "data": {
        "request": {
            "sourceIPAddress": "127.0.0.1"
        },
        "response": {
            "requestId": "daab11b695ea4c4ea7a1a71ce36d1100"
        },
        "ks3": {
            "bucket": {
                "name": "kcf-pkg",
                "ownerid": "7340"
            },
            "object": {
                "internalurl": "evt-jin.ks3-cn-beijing-internal.ksyuncs.com/a.jpg",
                "etag": "etag-xxxxxxx",
                "objectsize": "1024",
                "url": "evt-jin.ks3-cn-beijing.ksyuncs.com/a.jpg",
                "key": "a.jpg"
            }
        }
    }
}'
```

HTTP请求调用示例

- GET调用

```
curl -X GET http://127.0.0.1:80880/http-invoke
```

- POST调用

```
curl -X POST http://127.0.0.1:80880/http-invoke
'{
"id": "a123",
}'                          //用户自定义body内容
```

- PUT调用

```
curl -X PUT http://127.0.0.1:80880/http-invoke/1
```

- PATCH调用

```
curl -X PATCH http://127.0.0.1:80880/http-invoke/1
```

- DELETE调用

```
curl -X DELETE http://127.0.0.1:80880/http-invoke/1
```

Node.js运行环境

Node.js 版本

云函数目前支持的Node.js运行时环境如下。

版本	操作系统	架构
Node.js 16.17	Linux	x86_64
Node.js 14.20	Linux	x86_64
Node.js 12.22	Linux	x86_64
Node.js 10.24	Linux	x86_64

相关环境变量

目前运行环境中内置的 Node.js 相关环境变量见下表：

环境变量 Key	具体值或值来源
NODE_PATH	/opt/nodejs/node_modules

请求处理程序开发方法

本文介绍在云函数KCF中使用Node.js语言开发请求处理程序的相关概念和方法。

请求处理程序分为事件请求处理程序（Event Handler）和HTTP请求处理程序（HTTP Handler）；其中事件请求由各种事件源触发生成，HTTP请求则由HTTP触发器触发生成。 请求处理程序的具体配置示例如下：

事件请求处理程序（Event Handler）

介绍Node.js事件请求处理程序的结构和特点。

在Node.js语言的代码中，使用npm引入Express Web框架时，同时引入cloudevents官方的SDK库 ：

```
npm install express -t .
npm install cloudevents -t .
```

express框架 + cloudevents 代码示例

```
const app = require("express")();
const { HTTP } = require("cloudevents");

app.get("/health", (req, res) => {
  console.log("health check message received");
  res.json("health");
});

app.post("/event-invoke", (req, res) => {
  const receivedEvent = HTTP.toEvent({ headers: req.headers, body: req.body });
  console.log("event received:", receivedEvent);
  res.json("event received");
});

app.use("/", (req, res, err, next) => {
  if (err) {
    console.log("err", err);
  }
});

const port = process.env.PORT || 8080;

const http = require("http");
const server = http.createServer(app);

server.on("error", (err) => {
  console.error("error", err);
```

```
});

server.listen(port, () => {
  console.log(`listening on ${port}`);
});
```

Event Handler的示例解析如下：

- req：HTTP请求结构体。
- res：HTTP返回结构体。
- HTTP.toEvent：将http请求内容转换成cloudevents结构体。

cloudevents详细介绍：

- doc
- samples

HTTP请求处理程序（HTTP Handler）

介绍Node.js HTTP请求处理程序的结构和特点。

在Node.js语言的代码中，使用npm引入Express Web框架时，将express库安装到当前目录下 ：

```
npm install express -t .
```

express框架代码示例

```
const app = require("express")();

app.all("/http-invoke", (req, res) => {
  const message = `${req.method} message received`;
  console.log(message);
  res.json(message);
});

app.use("/", (req, res, err, next) => {
  if (err) {
    console.log("err", err);
  }
});

const port = process.env.PORT || 8080;

const http = require("http");
const server = http.createServer(app);

server.on("error", (err) => {
  console.error("error", err);
});

server.listen(port, () => {
  console.log(`listening on ${port}`);
});
```

HTTP Handler的示例解析如下：

- req：HTTP请求结构体。
- res：HTTP返回结构体

安装依赖与打包

安装依赖

使用npm安装依赖

在mycode目录下执行`npm install express -t .`安装express依赖库到当前目录

打包

在Linux或macOS下编译

进入代码目录，执行`zip code.zip -r ./*`。

注意：请确保您创建的index.js文件位于包的根目录。

在 Windows下打包

进入代码目录，选中所有文件，单击鼠标右键，选择打包为ZIP包。

注意：由于函数计算的运行环境是Linux系统，您在Windows系统或macOS系统上安装依赖库时可能会带有二进制文件，会导致您的代码包上传到函数计算后运行失败。此时，建议您在linux环境下安装依赖并打包。

函数日志

打印日志

函数往标准输出stdout打印的日志内容会被收集到创建函数时指定的日志池中，您可以使用以下方式打印日志。

- console.log

使用console打印日志

例如，执行以下代码，可以在函数日志中查询输出内容

```
app.all("/http-invoke", (req, res) => {
  const message = `hello world`;
  console.log(message);
  res.json(message);
});
```

请求调用示例

事件请求示例

- 健康检查

```
  curl --location --request GET 'http://127.0.0.1:8080/health'
```

- 发送Cloudevent:

```
curl --location --request POST 'http://127.0.0.1:8080/event-invoke' \
--header 'Content-type: application/cloudevents+json' \
--data '{
    "id": "1",
    "source": "kcs:ks3",
    "type": "ks3:ObjectCreated:PutObject",
    "specversion": "1.0",
    "datacontenttype": "application/json",
    "subject": "/",
    "comValue1": "extension value1",
    "comValue2": "extension value2",
    "data": {
        "request": {
            "sourceIPAddress": "127.0.0.1"
        },
        "response": {
            "requestId": "daab11b695ea4c4ea7a1a71ce36d1100"
        },
        "ks3": {
            "bucket": {
                "name": "kcf-pkg",
                "ownerid": "7340"
            },
            "object": {
                "internalurl": "evt-jin.ks3-cn-beijing-internal.ksyuncs.com/a.jpg",
                "etag": "etag-xxxxxxx",
                "objectsize": "1024",
                "url": "evt-jin.ks3-cn-beijing.ksyuncs.com/a.jpg",
                "key": "a.jpg"
            }
        }
    }
}'
```

HTTP请求调用示例

- GET调用

```
curl -X GET http://127.0.0.1:80880/http-invoke
```

- POST调用

```
curl -X POST http://127.0.0.1:80880/http-invoke
'{
"id": "a123",
}'                              //用户自定义body内容
```

- PUT调用

```
curl -X PUT http://127.0.0.1:80880/http-invoke/1
```

- PATCH调用

```
curl -X PATCH http://127.0.0.1:80880/http-invoke/1
```

- DELETE调用

```
curl -X DELETE http://127.0.0.1:80880/http-invoke/1
```