Red Hat OpenShift Serverless 1.30

Functions

设置和使用 OpenShift Serverless 功能

Last Updated: 2023-09-20

Red Hat OpenShift Serverless 1.30 Functions

设置和使用 OpenShift Serverless 功能

法律通告

摘要

本文档提供有关开始使用 OpenShift Serverless 功能的信息，以及使用 Quarkus、Node.js、typetype 和 Python 开发和部署功能。

目录

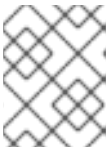
第 1 章 功能入门

功能生命周期管理包括创建和部署功能，之后可以调用它。您可以使用 **kn func** 工具在 OpenShift Serverless 上完成所有这些操作。

1.1. 先决条件

要在集群中启用 OpenShift Serverless 功能，您必须完成以下步骤：

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。



注意

功能部署为 Knative 服务。如果要将事件驱动的架构与您的功能搭配使用，还必须安装 Knative Eventing。

- 已安装 **oc** CLI。
- 已安装 Knative (**kn**) CLI。安装 Knative CLI 可让您使用 **kn func** 命令来创建和管理功能。
- 已安装 Docker Container Engine 或 Podman 版本 3.4.7 或更高版本。
- 您可以访问可用的镜像 registry，如 OpenShift Container Registry。
- 如果您使用 Quay.io 作为镜像 registry，您必须确保存储库不是私有的，或者按照 OpenShift Container Platform 文档中有关允许 Pod 引用其他安全 registry 中的镜像的内容进行操作。
- 如果使用 OpenShift Container Registry，集群管理员必须公开 registry。

1.2. 创建、部署和调用函数

在 OpenShift Serverless 中，您可以使用 **kn func** 创建、部署和调用功能。

流程

1. 创建功能项目：

```
$ kn func create -l <runtime> -t <template> <path>
```

示例命令

```
$ kn func create -l typescript -t cloudevents examplefunc
```

输出示例

```
Created typescript function in /home/user/demo/examplefunc
```

2. 进入功能项目目录：

示例命令

```
$ cd examplefunc
```

3. 本地构建并运行功能：

   示例命令

   ```
   $ kn func run
   ```

4. 将功能部署到集群中：

   ```
   $ kn func deploy
   ```

   输出示例

   ```
   Function deployed at: http://func.example.com
   ```

5. 调用函数：

   ```
   $ kn func invoke
   ```

   这将在本地或远程运行函数调用。如果两者都在运行，则调用本地。

1.3. OPENSHIFT CONTAINER PLATFORM 的其他资源

- 手动公开默认 registry
- Intellij Knative 插件的 marketplace 页面
- Visual Studio Code Knative 插件的 marketplace 页面
- 使用 Developer 视角创建应用程序

1.4. 后续步骤

- 请参阅在 Knative Eventing 中使用功能

第 2 章 创建功能

在构建和部署功能前，必须先创建它。您可以使用 Knative (**kn**) CLI 创建功能。

2.1. 使用 KNATIVE CLI 创建功能

您可以为命令行上的标志指定功能的路径、运行时、模板和镜像 registry，或者使用 **-c** 标志在终端中启动交互式体验。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。

流程

- 创建功能项目：

```
$ kn func create -r <repository> -l <runtime> -t <template> <path>
```

  - 可接受的运行时值包括 **quarkus**、**node**、**typescript**、**go**、**python**、**springboot** 和 **rust**。
  - 可接受的模板值包括 **http** 和 **cloudevents**。

    示例命令

```
$ kn func create -l typescript -t cloudevents examplefunc
```

    输出示例

```
Created typescript function in /home/user/demo/examplefunc
```

  - 或者，您可以指定包含自定义模板的存储库。

    示例命令

```
$ kn func create -r https://github.com/boson-project/templates/ -l node -t hello-world
examplefunc
```

    输出示例

```
Created node function in /home/user/demo/examplefunc
```

2.2. 在 WEB 控制台中创建功能

您可以使用 Red Hat OpenShift Serverless Web 控制台的 **Developer** 视角从 Git 仓库创建功能。

先决条件

- 在使用 Web 控制台创建功能前，集群管理员必须完成以下步骤：

- 在集群上安装 OpenShift Serverless Operator 和 Knative Serving。
- 在集群上安装 OpenShift Pipelines Operator。
- 创建以下管道任务，以便它们可用于集群中的所有命名空间：

  func-s2i 任务

  ```
  $ oc apply -f https://raw.githubusercontent.com/openshift-knative/kn-plugin-func/serverless-1.30/pkg/pipelines/resources/tekton/task/func-s2i/0.1/func-s2i.yaml
  ```

  func-deploy 任务

  ```
  $ oc apply -f https://raw.githubusercontent.com/openshift-knative/kn-plugin-func/serverless-1.30/pkg/pipelines/resources/tekton/task/func-s2i/0.1/func-s2i.yaml
  ```

  Node.js 功能

  ```
  $ oc apply -f https://raw.githubusercontent.com/openshift-knative/kn-plugin-func/serverless-1.30/pkg/pipelines/resources/tekton/task/func-s2i/0.1/func-s2i.yaml
  ```

- 您必须登录 Red Hat OpenShift Serverless web 控制台。
- 您必须创建项目，或者具有适当的角色和权限访问项目，以便在 Red Hat OpenShift Serverless 中创建应用程序和其他工作负载。
- 您必须创建或有权访问包含功能代码的 Git 存储库。存储库必须包含 **func.yaml** 文件，并使用 **s2i** 构建策略。

流程

1. 在 **Developer** 视角中，导航到 **+Add → Create Serverless 功能**。此时会显示 **Create Serverless 功能** 页面。
2. 输入指向包含功能代码的 Git 存储库的 **Git Repo URL**。
3. 在 **Pipelines** 部分：
   a. 选择 **Build, deploy and configure a Pipeline Repository**单选按钮，为您的功能创建新管道。
   b. 选择 **Use Pipeline from this cluster**单选按钮将功能连接到集群中的现有管道。
4. 点 **Create**。

验证

- 创建功能后，您可以在 **Developer** 视角的 **Topology** 视图中查看它。

第 3 章 本地运行功能

您可以使用 **kn func** 工具在本地运行功能。例如，这在将其部署到集群之前测试该功能非常有用。

3.1. 在本地运行一个函数

您可以使用 **kn func run** 命令在当前目录中本地运行函数，或者在 **--path** 标志指定的目录中运行。如果您运行的函数之前没有被构建，或者项目文件自上次构建以来已修改过，**kn func run** 命令将在运行它前构建该函数。

在当前目录中运行函数的命令示例

```
$ kn func run
```

在指定为路径的目录中运行函数的示例

```
$ kn func run --path=<directory_path>
```

您也可以在运行该函数前强制重建现有镜像，即使项目文件没有更改项目文件，则使用 **--build** 标志：

使用 build 标记的 run 命令示例

```
$ kn func run --build
```

如果将 **build** 标志设置为 false，这将禁用构建镜像，并使用之前构建的镜像运行该功能：

使用 build 标记的 run 命令示例

```
$ kn func run --build=false
```

您可以使用 help 命令了解更多有关 **kn func run** 命令选项的信息：

构建 help 命令

```
$ kn func help run
```

第 4 章 部署功能

您可以使用 **kn func** 工具将功能部署到集群中。

4.1. 部署功能

您可以使用 **kn func deploy** 命令将功能部署到集群中，作为 Knative 服务。如果已经部署了目标功能，则会使用推送到容器镜像 registry 的新容器镜像进行更新，并更新 Knative 服务。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您必须已创建并初始化要部署的功能。

流程

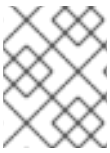
- 部署功能：

```
$ kn func deploy [-n <namespace> -p <path> -i <image>]
```

输出示例

```
Function deployed at: http://func.example.com
```

  - 如果没有指定 **namespace**，则该函数部署到当前命名空间中。
  - 此函数从当前目录中部署，除非指定了 **path**。
  - Knative 服务名称派生自项目名称，无法使用此命令进行更改。



注意

您可以使用 **Developer** 视角的 **+Add** 视图中的 **Import from Git** 或 **Create Serverless Function** 使用 Git 存储库 URL 创建无服务器功能。

第 5 章 构建函数

要运行功能，您必须首先构建 function 项目。这在使用 **kn func run** 命令时自动发生，但您也可以在不运行的情况下构建功能。

5.1. 构建函数

在运行功能前，您必须构建 function 项目。如果使用 **kn func run** 命令，则该函数会自动构建。但是，您可以使用 **kn func build** 命令在不运行的情况下构建函数，这对于高级用户或调试场景非常有用。

kn func build 命令创建可在您的计算机或 OpenShift Container Platform 集群中运行的 OCI 容器镜像。此命令使用功能项目名称和镜像 registry 名称为您的功能构建完全限定镜像名称。

5.1.1. 镜像容器类型

默认情况下，**kn func build** 使用 Red Hat Source-to-Image (S2I) 技术创建一个容器镜像。

使用 Red Hat Source-to-Image (S2I) 的 build 命令示例.

```
$ kn func build
```

5.1.2. 镜像 registry 类型

OpenShift Container Registry 默认用作存储功能镜像的镜像 registry。

使用 OpenShift Container Registry 的 build 命令示例

```
$ kn func build
```

输出示例

```
Building function image
Function image has been built, image: registry.redhat.io/example/example-function:latest
```

您可以使用 **--registry** 标志覆盖使用 OpenShift Container Registry 作为默认镜像 registry：

build 命令覆盖 OpenShift Container Registry 以使用 quay.io

```
$ kn func build --registry quay.io/username
```

输出示例

```
Building function image
Function image has been built, image: quay.io/username/example-function:latest
```

5.1.3. push 标记

您可以将 **--push** 标志添加到 **kn func build** 命令中，以便在成功构建后自动推送功能镜像：

使用 OpenShift Container Registry 的 build 命令示例

```
$ kn func build --push
```

5.1.4. help 命令

您可以使用 help 命令了解更多有关 **kn func build** 命令选项的信息：

构建 help 命令

```
$ kn func help build
```

第 6 章 列出现有功能

您可以列出现有功能。您可以使用 **kn func** 工具进行此操作。

6.1. 列出现有功能

您可以使用 **kn func list** 列出现有功能。如果要列出部署为 Knative 服务的功能，也可以使用 **kn service list**。

流程

- 列出现有功能：

```
$ kn func list [-n <namespace> -p <path>]
```

输出示例

```
NAME            NAMESPACE  RUNTIME  URL
READY
example-function  default    node     http://example-function.default.apps.ci-ln-g9f36hb-
d5d6b.origin-ci-int-aws.dev.rhcloud.com  True
```

- 列出部署为 Knative 服务的功能：

```
$ kn service list -n <namespace>
```

输出示例

```
NAME            URL                                                                LATEST
AGE   CONDITIONS   READY   REASON
example-function   http://example-function.default.apps.ci-ln-g9f36hb-d5d6b.origin-ci-int-
aws.dev.rhcloud.com   example-function-gzl4c   16m   3 OK / 3     True
```

第 7 章 调用函数

您可以通过调用它来测试部署的功能。您可以使用 **kn func** 工具进行此操作。

7.1. 使用测试事件调用部署的功能

您可以使用 **kn func invoke** CLI 命令发送测试请求，在本地或 OpenShift Container Platform 集群中调用功能。您可以使用此命令测试功能是否正常工作并且能够正确接收事件。本地调用函数可用于在功能开发期间进行快速测试。在测试与生产环境更接近的测试时，在集群中调用函数非常有用。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您必须已部署了要调用的功能。

流程

- 调用函数：

```
$ kn func invoke
```

  - **kn func invoke** 命令仅在当前运行本地容器镜像时或在集群中部署功能时才有效。
  - **kn func invoke** 命令默认在本地目录上执行，并假定此目录是一个功能项目。

第 8 章 删除功能

您可以删除功能。您可以使用 **kn func** 工具进行此操作。

8.1. 删除函数

您可以使用 **kn func delete** 命令删除功能。当不再需要某个函数时，这很有用，并有助于在集群中保存资源。

流程

- 删除函数：

```
$ kn func delete [<function_name> -n <namespace> -p <path>]
```

  - 如果没有指定要删除的功能的名称或路径，则会搜索当前目录以查找用于决定要删除的功能的 **func.yaml** 文件。
  - 如果没有指定命名空间，则默认为 **func.yaml** 文件中的 **namespace** 值。

第 9 章 在集群中构建和部署功能

您可以直接在集群中构建功能，而不是在本地构建功能。在本地开发机器上使用此工作流时，您只需要使用功能源代码。例如，当您无法在集群功能构建工具（如 docker 或 podman）安装时，这非常有用。

9.1. 在集群中构建和部署功能

您可以使用 Knative (**kn**) CLI 启动功能项目构建，然后直接将功能部署到集群中。若要以这种方式构建功能项目，您的功能项目的源代码必须存在于可供集群访问的 Git 存储库分支中。

先决条件

- 在集群中必须安装 Red Hat OpenShift Pipelines。
- 已安装 OpenShift CLI(**oc**)。
- 已安装 Knative (**kn**) CLI。

流程

1. 创建以下资源：
   a. 创建 **s2i** Tekton 任务，以便在管道中使用 Source-to-Image：

```
$ oc apply -f https://raw.githubusercontent.com/openshift-knative/kn-plugin-func/serverless-1.29.0/pkg/pipelines/resources/tekton/task/func-s2i/0.1/func-s2i.yaml
```

   b. 创建 **kn func** 部署 Tekton 任务，以便在管道中部署该功能：

```
$ oc apply -f https://raw.githubusercontent.com/openshift-knative/kn-plugin-func/serverless-1.29.0/pkg/pipelines/resources/tekton/task/func-deploy/0.1/func-deploy.yaml
```

2. 创建功能：

```
$ kn func create <function_name> -l <runtime>
```

3. 实施您功能的业务逻辑。然后，使用 Git 提交并推送更改。
4. 部署功能：

```
$ kn func deploy --remote
```

   如果您没有登录到功能配置中引用的容器 registry，系统会提示您为托管功能镜像的远程容器 registry 提供凭证：

   输出和提示示例

```
 Creating Pipeline resources
Please provide credentials for image registry used by Pipeline.
? Server: https://index.docker.io/v1/
```

```
? Username: my-repo
? Password: ********
  Function deployed at URL: http://test-function.default.svc.cluster.local
```

5. 要更新您的功能，使用 Git 提交并推送新的更改，然后再次运行 **kn func deploy --remote** 命令。

6. 可选。您可以使用 pipelines-as-code 将功能配置为在每个 Git 推送后在集群中构建：

   a. 为您的功能生成 Tekton **Pipelines** 和 **PipelineRuns** 配置：

   ```
   $ kn func config git set
   ```

   除了生成配置文件外，这个命令会连接到集群并验证是否已安装管道。通过使用令牌，它还会创建代表功能存储库的 Webhook。每次更改推送到存储库时，webhook 都会在集群中触发管道。

   您需要具有具有存储库访问权限的有效 GitHub 个人访问令牌，才能使用此命令。

   b. 提交并推送生成的 **.tekton/pipeline.yaml** 和 **.tekton/pipeline-run.yaml** 文件：

   ```
   $ git add .tekton/pipeline.yaml .tekton/pipeline-run.yaml
   $ git commit -m 'Add the Pipelines and PipelineRuns configuration'
   $ git push
   ```

   c. 在对功能进行更改后，提交并推送它。该函数通过使用创建的管道自动重新构建。

9.2. 指定功能修订

在集群中构建和部署功能时，您必须通过指定存储库中的 Git 存储库、分支和子目录来指定功能代码的位置。如果使用 **main** 分支，则不需要指定分支。同样，如果功能位于存储库的根目录，则不需要指定子目录。您可以在 **func.yaml** 配置文件中指定这些参数，或使用带有 **kn func deploy** 命令的标志。

先决条件

- 在集群中必须安装 Red Hat OpenShift Pipelines。
- 已安装 OpenShift (**oc**) CLI。
- 已安装 Knative (**kn**) CLI。

流程

- 部署功能：

```
$ kn func deploy --remote \ 1
                 --git-url <repo-url> \ 2
                 [--git-branch <branch>] \ 3
                 [--git-dir <function-dir>] 4
```

1. 使用 **--remote** 标志时，构建远程运行。
2. 将 **<repo-url>** 替换为 Git 存储库的 URL。

3. 将 **<branch>** 替换为 Git 分支、标签或提交。如果使用 **main** 分支的最新提交，您可以跳过此标志。

4. 如果 **<function-dir>** 与存储库根目录不同，请将 <function-dir> 替换为包含该函数的目录。

例如：

```
$ kn func deploy --remote \
                 --git-url https://example.com/alice/myfunc.git \
                 --git-branch my-feature \
                 --git-dir functions/example-func/
```

第 10 章 将事件源连接到功能

函数作为 Knative 服务部署到 OpenShift Container Platform 集群中。您可以将函数连接到 Knative Eventing 组件，以便它们可以接收传入的事件。

10.1. 使用 DEVELOPER 视角将事件源连接到函数

函数作为 Knative 服务部署到 OpenShift Container Platform 集群中。当使用 OpenShift Container Platform Web 控制台创建事件源时，您可以指定事件从该源发送到的部署函数。

先决条件

- OpenShift Serverless Operator、Knative Serving 和 Knative Eventing 已在 OpenShift Container Platform 集群中安装。
- 已登陆到 web 控制台，且处于 **Developer** 视角。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您已创建并部署了函数。

流程

1. 进入 **+Add → Event Source** 并选择您要创建的事件源类型，创建任何类型的事件源。
2. 在 **Create Event Source** 表单视图的 **Target** 部分中，在 **Resource** 列表中选择您的功能。
3. 点 **Create**。

验证

您可以通过查看 **Topology** 页面来验证事件源是否已创建并连接到该函数。

1. 在 **Developer** 视角中，导航到 **Topology**。
2. 查看事件源并点连接的函数来查看右侧面板中的函数详情。

第 11 章 功能开发参考指南

11.1. 开发 QUARKUS 功能

创建 Quarkus 功能项目后，您可以修改提供的模板文件，以将业务逻辑添加到您的功能中。这包括配置功能调用和返回的标头和状态代码。

11.1.1. 先决条件

- 在开发功能前，您必须完成 配置 OpenShift Serverless 功能 中的设置步骤。

11.1.2. Quarkus 功能模板结构

使用 Knative (**kn**) CLI 创建 Quarkus 功能时，项目目录类似于典型的 Maven 项目。另外，项目还包含用于配置功能的 **func.yaml** 文件。

http 和 **event** 触发器功能具有相同的模板结构：

模板结构

```
.
├── func.yaml 1
├── mvnw
├── mvnw.cmd
├── pom.xml 2
├── README.md
└── src
    ├── main
    │   ├── java
    │   │   └── functions
    │   │       ├── Function.java 3
    │   │       ├── Input.java
    │   │       └── Output.java
    │   └── resources
    │       └── application.properties
    └── test
        └── java
            └── functions 4
                ├── FunctionTest.java
                └── NativeFunctionIT.java
```

1. 用于确定镜像名称和 registry。
2. 项目对象模型（POM）文件包含项目配置，如依赖项的相关信息。您可以通过修改此文件来添加额外的依赖项。

   其他依赖项示例

   ```
   ...
    <dependencies>
     <dependency>
       <groupId>junit</groupId>
       <artifactId>junit</artifactId>
   ```

```
      <version>4.13</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.assertj</groupId>
      <artifactId>assertj-core</artifactId>
      <version>3.8.0</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
...
```

依赖项在第一次编译时下载。

3 功能项目必须包含标有 **@Funq** 的 Java 方法。您可以将此方法放置在 **Function.java** 类中。

4 包含可用于在本地测试功能的简单测试案例。

11.1.3. 关于调用 Quarkus 功能

您可以创建一个 Quarkus 项目来响应云事件，或创建响应简单 HTTP 请求的 Quarkus 项目。Knative 中的云事件作为 POST 请求通过 HTTP 传输，因此任一功能类型都可以侦听和响应传入的 HTTP 请求。

收到传入请求时，通过允许类型的实例调用 Quarkus 函数。

表 11.1. 功能调用选项

调用方法	实例中包含的数据类型	数据示例
HTTP POST 请求	请求正文中的 JSON 对象	**{ "customerId": "0123456", "productId": "6543210" }**
HTTP GET 请求	查询字符串中的数据	**?customerId=0123456&productId=6543210**
CloudEvent	**data** 属性中的 JSON 对象	**{ "customerId": "0123456", "productId": "6543210" }**

以下示例显示了接收并处理上表中列出的 **customerId** 和 **productId** 购买数据的函数：

Quarkus 功能示例

```
public class Functions {
    @Funq
    public void processPurchase(Purchase purchase) {
        // process the purchase
    }
}
```

包含购买数据的对应 **Purchase** JavaBean 类如下：

类示例

```
public class Purchase {
    private long customerId;
    private long productId;
    // getters and setters
}
```

11.1.3.1. 调用示例

以下示例代码定义了名为 **withBeans**、**withCloudEvent** 和 **withBinary** 的三个功能；

示例

```
import io.quarkus.funqy.Funq;
import io.quarkus.funqy.knative.events.CloudEvent;

public class Input {
    private String message;

    // getters and setters
}

public class Output {
    private String message;

    // getters and setters
}

public class Functions {
    @Funq
    public Output withBeans(Input in) {
        // function body
    }

    @Funq
    public CloudEvent<Output> withCloudEvent(CloudEvent<Input> in) {
        // function body
    }

    @Funq
    public void withBinary(byte[] in) {
        // function body
    }
}
```

Functions 类的 **withBeans** 功能可以通过以下方法调用：

- 带有 JSON 正文的 HTTP POST 请求：

```
$ curl "http://localhost:8080/withBeans" -X POST \
    -H "Content-Type: application/json" \
    -d '{"message": "Hello there."}'
```

- 带有查询参数的 HTTP GET 请求：

```
$ curl "http://localhost:8080/withBeans?message=Hello%20there." -X GET
```

- 二进制编码中的 **CloudEvent** 对象：

```
$ curl "http://localhost:8080/" -X POST \
  -H "Content-Type: application/json" \
  -H "Ce-SpecVersion: 1.0" \
  -H "Ce-Type: withBeans" \
  -H "Ce-Source: cURL" \
  -H "Ce-Id: 42" \
  -d '{"message": "Hello there."}'
```

- 结构化编码中的 **CloudEvent** 对象：

```
$ curl http://localhost:8080/ \
    -H "Content-Type: application/cloudevents+json" \
    -d '{ "data": {"message":"Hello there."},
          "datacontenttype": "application/json",
          "id": "42",
          "source": "curl",
          "type": "withBeans",
          "specversion": "1.0"}'
```

与 **withBeans** 函数类似，可以利用 **CloudEvent** 对象来调用 **Functions** 类的 **withCloudEvent** 功能。但是，与 **Beans** 不同，**CloudEvent** 无法通过普通 HTTP 请求来调用。

Functions 类的 **withBinary** 功能可通过以下方式调用：

- 二进制编码中的 **CloudEvent** 对象：

```
$ curl "http://localhost:8080/" -X POST \
  -H "Content-Type: application/octet-stream" \
  -H "Ce-SpecVersion: 1.0"\
  -H "Ce-Type: withBinary" \
  -H "Ce-Source: cURL" \
  -H "Ce-Id: 42" \
  --data-binary '@img.jpg'
```

- 结构化编码中的 **CloudEvent** 对象：

```
$ curl http://localhost:8080/ \
  -H "Content-Type: application/cloudevents+json" \
  -d "{ \"data_base64\": \"$(base64 --wrap=0 img.jpg)\",
        \"datacontenttype\": \"application/octet-stream\",
        \"id\": \"42\",
        \"source\": \"curl\",
        \"type\": \"withBinary\",
        \"specversion\": \"1.0\"}"
```

11.1.4. CloudEvent 属性

如果您需要读取或写入 CloudEvent 的属性，如 **type** 或 **subject**，您可以使用 **CloudEvent<T>** 通用接口和 **CloudEventBuilder** 构建器。**<T>** 类型参数必须是允许的类型之一。

在以下示例中，**CloudEventBuilder** 用于返回处理订购的成功或失败：

```
public class Functions {

    private boolean _processPurchase(Purchase purchase) {
        // do stuff
    }

    public CloudEvent<Void> processPurchase(CloudEvent<Purchase> purchaseEvent) {
        System.out.println("subject is: " + purchaseEvent.subject());

        if (!_processPurchase(purchaseEvent.data())) {
            return CloudEventBuilder.create()
                    .type("purchase.error")
                    .build();
        }
        return CloudEventBuilder.create()
                .type("purchase.success")
                .build();
    }
}
```

11.1.5. Quarkus 功能返回值

功能可以从允许类型列表中返回任何类型的实例。另外，他们可以返回 **Uni<T>** 类型，其中 **<T>** 类型参数可以是允许的类型的任何类型。

如果函数调用异步 API，因为返回的对象以与接收对象相同的格式序列化，**Uni<T>** 类型很有用。例如：

- 如果函数收到 HTTP 请求，则返回的对象将在 HTTP 响应的正文中发送。
- 如果函数通过二进制编码收到 **CloudEvent** 对象，则返回的对象将在二进制编码的 **CloudEvent** 对象的 data 属性中发送。

以下示例显示了获取购买列表的功能：

示例命令

```
public class Functions {
    @Funq
    public List<Purchase> getPurchasesByName(String name) {
      // logic to retrieve purchases
    }
}
```

- 通过 HTTP 请求调用此功能将生成 HTTP 响应，其中包含响应正文中的订购列表。
- 通过传入的 **CloudEvent** 对象调用此功能可生成 **CloudEvent** 响应，并在 **data** 属性中包括一个订购列表。

11.1.5.1. 允许的类型

功能的输入和输出可以是 **void**、**String**、或 **byte[]** 类型。此外，它们也可以是原语类型及其打包程序，例如 **int** 和 **Integer**。它们也可以是以下复杂的对象：Javabeans、映射、列表、数组和特殊的 **CloudEvents<T>** 类型。

映射、列出、数组、**CloudEvents<T>** 类型的 **<T>** 类型参数以及 Javabeans 的属性只能是此处列出的类型。

示例

```
public class Functions {
    public List<Integer> getIds();
    public Purchase[] getPurchasesByName(String name);
    public String getNameById(int id);
    public Map<String,Integer> getNameIdMapping();
    public void processImage(byte[] img);
}
```

11.1.6. 测试 Quarkus 功能

Quarkus 功能可以在您的计算机上进行本地测试。在使用 **kn func create** 创建功能时创建的默认项目中，有 **src/test/** 目录，其中包含基本的 Maven 测试。这些测试可以根据需要扩展。

先决条件

- 您已创建了 Quarkus 功能。
- 已安装 Knative (**kn**) CLI。

流程

1. 导航到您的功能的项目文件夹。
2. 运行 Maven 测试：

```
$ ./mvnw test
```

11.1.7. 覆盖存活度和就绪度探测值

您可以覆盖 Quarkus 功能的 **存活度和就绪度探测** 值。这可让您配置在功能上执行的健康检查。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。
- 已使用 **kn func create** 创建功能。

流程

1. 使用您自己的值覆盖 **/health/liveness** 和 **/health/readiness** 路径。您可以通过在 **func.yaml** 文件中设置 **QUARKUS_SMALLRYE_HEALTH_LIVENESS_PATH** 和 **QUARKUS_SMALLRYE_HEALTH_READINESS_PATH** 环境变量来完成此操作。

a. 要使用功能源覆盖路径，更新 **src/main/resources/application.properties** 文件中的路径属性：

```
quarkus.smallrye-health.root-path=/health 1
quarkus.smallrye-health.liveness-path=alive 2
quarkus.smallrye-health.readiness-path=ready 3
```

1. 根路径，其自动添加到 **存活度** 和 **就绪度** 路径的前面。
2. 存活度路径，设置为此处的 **/health/alive**。
3. 就绪度路径，设置为 **/health/ready**。

b. 要使用环境变量覆盖路径，请在 **func.yaml** 文件的构建块中定义路径变量：

```
build:
  builder: s2i
  buildEnvs:
  - name: QUARKUS_SMALLRYE_HEALTH_LIVENESS_PATH
    value: alive 1
  - name: QUARKUS_SMALLRYE_HEALTH_READINESS_PATH
    value: ready 2
```

1. **存活度路径，设置为此处的 /health/alive。**
2. **就绪度路径，设置为 /health/ready。**

2. **将新端点添加到 func.yaml 文件中，以便它们正确绑定到 Knative 服务的容器：**

```
deploy:
  healthEndpoints:
    liveness: /health/alive
    readiness: /health/ready
```

11.1.8. 后续步骤

- 构建和部署功能.

11.2. 开发 NODE.JS 功能

创建 Node.js 功能项目 后，您可以修改提供的模板文件，以将业务逻辑添加到您的功能中。这包括配置功能调用和返回的标头和状态代码。

11.2.1. 先决条件

- 在开发功能前，您必须完成 配置 OpenShift Serverless 功能 中的步骤。

11.2.2. Node.js 功能模板结构

使用 Knative (kn) CLI 创建 Node.js 功能时，项目目录类似于典型的 Node.js 项目。唯一的例外是额外的 func.yaml 文件，用于配置函数。

http 和 event 触发器功能具有相同的模板结构：

模板结构

```
.
├── func.yaml 1
├── index.js 2
├── package.json 3
├── README.md
└── test 4
    ├── integration.js
    └── unit.js
```

1 func.yaml 配置文件用于决定镜像名称和 registry。

2 您的项目必须包含可导出单一功能的 index.js 文件。

3 您不限于模板 package.json 文件中提供的依赖项。您可以添加其他依赖项，如在任何其他 Node.js 项目中一样。

添加 npm 依赖项示例

```
npm install --save opossum
```

为部署构建项目时，这些依赖项将包含在创建的运行时容器镜像中。

4

集成和单元测试脚本作为功能模板的一部分提供。

11.2.3. 关于调用 Node.js 功能

当使用 Knative (kn) CLI 创建功能项目时，您可以生成一个响应 CloudEvents 的项目，或者响应简单 HTTP 请求的项目。Knative 中的 CloudEvents 作为 POST 请求通过 HTTP 传输，因此两种功能类型都侦听并响应传入的 HTTP 事件。

Node.js 功能可以通过简单的 HTTP 请求调用。收到传入请求后，将通过 上下文 对象作为第一个参数来调用函数。

11.2.3.1. Node.js 上下文对象

通过提供 上下文 对象作为第一个参数来调用函数。此对象提供对传入 HTTP 请求信息的访问。

上下文对象示例

```
function handle(context, data)
```

此信息包括 HTTP 请求方法、通过请求发送的任何查询字符串或标头、HTTP 版本和请求正文。传入包含 CloudEvent 的请求将进入 CloudEvent 实例附加到上下文对象，以便使用 context.cloudevent 访

问它。

11.2.3.1.1. 上下文对象方法

上下文（context）对象具有单一方法 cloudEventResponse () ，它接受数据值并返回 CloudEvent。

在 Knative 系统中，如果发送 CloudEvent 的事件代理调用将部署为服务的功能，代理会检查响应。如果响应是 CloudEvent，则此事件由代理处理。

上下文对象方法示例

```
// Expects to receive a CloudEvent with customer data
function handle(context, customer) {
  // process the customer
  const processed = handle(customer);
  return context.cloudEventResponse(customer)
    .source('/handle')
    .type('fn.process.customer')
    .response();
}
```

11.2.3.1.2. CloudEvent 数据

如果传入的请求为 CloudEvent，则从事件中提取与 CloudEvent 相关的任何数据，并作为第二个参数提供。例如，如果收到在它的数据属性中包含类似如下的 JSON 字符串的 CloudEvent：

```
{
  "customerId": "0123456",
  "productId": "6543210"
}
```

在调用时。函数的第二个参数（在上下文 对象后），将是带有 customerId 和 productId 属性的 JavaScript 对象。

签名示例

-

```
function handle(context, data)
```

本例中 data 参数是一个 JavaScript 对象，其中包含 customerId 和 productId 属性。

11.2.4. Node.js 功能返回值

功能可以返回任何有效的 JavaScript 类型，或者没有返回值。当函数没有指定返回值且未指示失败时，调用者会收到 204 No Content 响应。

功能也可以返回 CloudEvent 或一个 Message 对象，以便将事件推送到 Knative Eventing 系统。在这种情况下，开发人员不需要了解或实施 CloudEvent 消息传递规范。使用响应提取并发送返回值中的标头和其他相关信息。

示例

```
function handle(context, customer) {
  // process customer and return a new CloudEvent
  return new CloudEvent({
    source: 'customer.processor',
    type: 'customer.processed'
  })
}
```

11.2.4.1. 返回的标头

您可以通过在 return 对象中添加 headers 属性来设置响应标头。这些标头会提取并发送至调用者。

响应标头示例

```
function handle(context, customer) {
  // process customer and return custom headers
  // the response will be '204 No content'
```

```
  return { headers: { customerid: customer.id } };
}
```

11.2.4.2. 返回状态代码

您可以通过在返回对象中添加 statusCode 属性来设置 return 到调用者的状态代码：

状态代码示例

```
function handle(context, customer) {
  // process customer
  if (customer.restricted) {
    return { statusCode: 451 }
  }
}
```

也可以为函数创建和丢弃的错误设置状态代码：

错误状态代码示例

```
function handle(context, customer) {
  // process customer
  if (customer.restricted) {
    const err = new Error('Unavailable for legal reasons');
    err.statusCode = 451;
    throw err;
  }
}
```

11.2.5. 测试 Node.js 功能

Node.js 功能可以在您的计算机上本地测试。在使用 kn func create 创建功能时创建的默认项目中，

有一个 test 文件夹，其中包含一些简单的单元和集成测试。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (kn) CLI。
- 已使用 kn func create 创建功能。

流程

1. 导航到您的功能的 test 文件夹。
2. 运行测试：

```
$ npm test
```

11.2.6. 覆盖存活度和就绪度探测值

您可以覆盖 Node.js 功能的 存活度和就绪度探测 值。这可让您配置在功能上执行的健康检查。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (kn) CLI。
- 已使用 kn func create 创建功能。

流程

1. 在功能代码中，创建 Function 对象，它实现以下接口：
   -

```
export interface Function {
 init?: () => any; 1

 shutdown?: () => any; 2

 liveness?: HealthCheck; 3

 readiness?: HealthCheck; 4

 logLevel?: LogLevel;

 handle: CloudEventFunction | HTTPFunction; 5
}
```

1 初始化功能，在服务器启动前调用。此函数是可选的，应该同步。

2 关闭功能，在服务器停止后调用。此函数是可选的，应该同步。

3 存活度函数调用，以检查服务器是否处于活动状态。此函数是可选的，如果服务器处于活动状态，应返回 200/OK。

4 就绪度函数调用，以检查服务器是否准备好接受请求。此函数是可选的，如果服务器就绪，应返回 200/OK。

5 处理 HTTP 请求的功能。

例如，将以下代码添加到 index.js 文件中：

```
const Function = {

 handle: (context, body) => {
  // The function logic goes here
  return 'function called'
 },

 liveness: () => {
  process.stdout.write('In liveness\n');
  return 'ok, alive';
```

```
  }, (1)

  readiness: () => {
    process.stdout.write('In readiness\n');
    return 'ok, ready';
  } (2)
};

Function.liveness.path = '/alive'; (3)
Function.readiness.path = '/ready'; (4)

module.exports = Function;
```

(1) 自定义 存活度 功能。

(2) 自定义 就绪度 功能。

(3) 自定义 存活度 端点。

(4) 自定义 就绪度 端点。

作为 Function.liveness.path 和 Function.readiness.path 的替代选择，您可以使用 LIVENESS_URL 和 READINESS_URL 环境变量指定自定义端点：

```
run:
  envs:
  - name: LIVENESS_URL
    value: /alive (1)
  - name: READINESS_URL
    value: /ready (2)
```

(1) 存活度路径，设置为 /alive。

(2) 就绪度路径，设置为 /ready。

2. 将新端点添加到 func.yaml 文件中，以便它们正确绑定到 Knative 服务的容器：

```
deploy:
  healthEndpoints:
    liveness: /alive
    readiness: /ready
```

11.2.7. Node.js 上下文对象引用

该 上下文 对象具有多个属性，可供函数开发人员访问。访问这些属性可提供有关 HTTP 请求的信息，并将输出写入集群日志。

11.2.7.1. log

提供一个日志记录对象，可用于将输出写入集群日志。日志遵循 Pino 日志记录 API。

日志示例

```
function handle(context) {
  context.log.info("Processing customer");
}
```

您可以使用 kn func invoke 命令访问功能：

示例命令

```
$ kn func invoke --target 'http://example.function.com'
```

输出示例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":
1,"msg":"Processing customer"}
```

您可以将日志级别更改为 fatal、error、warn、info、debug、trace 或 silent 之一。为此，请使用 config 命令将其中的一个值分配给环境变量 FUNC_LOG_LEVEL，以更改 logLevel 的值。

11.2.7.2. query

返回请求的查询字符串（如果有），作为键值对。这些属性也可在上下文对象本身中找到。

查询示例

```
function handle(context) {
  // Log the 'name' query parameter
  context.log.info(context.query.name);
  // Query parameters are also attached to the context
  context.log.info(context.name);
}
```

您可以使用 kn func invoke 命令访问功能：

示例命令

```
$ kn func invoke --target 'http://example.com?name=tiger'
```

输出示例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":
1,"msg":"tiger"}
```

11.2.7.3. 正文（body）

如果有，返回请求正文。如果请求正文包含 JSON 代码，这将会进行解析，以便属性可以直接可用。

body 示例

```
function handle(context) {
  // log the incoming request body's 'hello' parameter
  context.log.info(context.body.hello);
}
```

您可以使用 curl 命令调用该函数：

示例命令

```
$ kn func invoke -d '{"Hello": "world"}'
```

输出示例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":
1,"msg":"world"}
```

11.2.7.4. 标头

将 HTTP 请求标头返回为对象。

标头示例

```
function handle(context) {
  context.log.info(context.headers["custom-header"]);
}
```

您可以使用 kn func invoke 命令访问功能：

示例命令

```
$ kn func invoke --target 'http://example.function.com'
```

输出示例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":
1,"msg":"some-value"}
```

11.2.7.5. HTTP 请求

方法

以字符串形式返回 HTTP 请求方法。

httpVersion

以字符串形式返回 HTTP 版本。

httpVersionMajor

将 HTTP 主版本号返回为字符串。

httpVersionMinor

以字符串形式返回 HTTP 次要版本号。

11.2.8. 后续步骤

- 构建和部署功能.

11.3. 开发类型脚本功能

创建 TypeScript 功能项目 后，您可以修改提供的模板文件，以将业务逻辑添加到您的功能中。这包括配置功能调用和返回的标头和状态代码。

11.3.1. 先决条件

- 在开发功能前，您必须完成 配置 OpenShift Serverless 功能 中的步骤。

11.3.2. TypeScript 功能模板结构

使用 Knative (kn) CLI 创建 TypeScript 功能时，项目目录类似于典型的 TypeScript 项目。唯一的例外是额外的 func.yaml 文件，用于配置函数。

http 和 event 触发器功能具有相同的模板结构：

模板结构

```
.
├── func.yaml 1
├── package.json 2
├── package-lock.json
├── README.md
├── src
│   └── index.ts 3
└── test 4
```

```
│   ├── integration.ts
│   └── unit.ts
└── tsconfig.json
```

1

func.yaml 配置文件用于决定镜像名称和 registry。

2

您不限于模板 package.json 文件中提供的依赖项。您可以添加额外的依赖项，如任何其他 TypeScript 项目中一样。

添加 npm 依赖项示例

```
npm install --save opossum
```

为部署构建项目时，这些依赖项将包含在创建的运行时容器镜像中。

3

您的项目必须包含 src/index.js 文件，该文件导出名为 handle 的函数。

4

集成和单元测试脚本作为功能模板的一部分提供。

11.3.3. 关于调用 TypeScript 函数

当使用 Knative (kn) CLI 创建功能项目时，您可以生成一个响应 CloudEvents 的项目，或者响应简单 HTTP 请求的项目。Knative 中的 CloudEvents 作为 POST 请求通过 HTTP 传输，因此两种功能类型都侦听并响应传入的 HTTP 事件。

TypeScript 函数可通过简单的 HTTP 请求调用。收到传入请求后，将通过 上下文 对象作为第一个参数来调用函数。

11.3.3.1. TypeScript 上下文对象

若要调用函数，您可以提供一个 context 对象作为第一个参数。访问 context 对象的属性可以提供有关传入 HTTP 请求的信息。

上下文对象示例

```
function handle(context:Context): string
```

此信息包括 HTTP 请求方法、通过请求发送的任何查询字符串或标头、HTTP 版本和请求正文。传入包含 CloudEvent 的请求将进入 CloudEvent 实例附加到上下文对象，以便使用 context.cloudevent 访问它。

11.3.3.1.1. 上下文对象方法

上下文（context）对象具有单一方法 cloudEventResponse () ，它接受数据值并返回 CloudEvent。

在 Knative 系统中，如果发送 CloudEvent 的事件代理调用将部署为服务的功能，代理会检查响应。如果响应是 CloudEvent，则此事件由代理处理。

上下文对象方法示例

```
// Expects to receive a CloudEvent with customer data
export function handle(context: Context, cloudevent?: CloudEvent): CloudEvent {
  // process the customer
  const customer = cloudevent.data;
  const processed = processCustomer(customer);
  return context.cloudEventResponse(customer)
    .source('/customer/process')
    .type('customer.processed')
    .response();
}
```

11.3.3.1.2. 上下文类型

TypeScript 类型定义文件导出以下类型以便在您的功能中使用。

导出类型定义

```
// Invokable is the expeted Function signature for user functions
export interface Invokable {
  (context: Context, cloudevent?: CloudEvent): any
}

// Logger can be used for structural logging to the console
export interface Logger {
 debug: (msg: any) => void,
 info:  (msg: any) => void,
 warn:  (msg: any) => void,
 error: (msg: any) => void,
 fatal: (msg: any) => void,
 trace: (msg: any) => void,
}

// Context represents the function invocation context, and provides
// access to the event itself as well as raw HTTP objects.
export interface Context {
  log: Logger;
  req: IncomingMessage;
  query?: Record<string, any>;
  body?: Record<string, any>|string;
  method: string;
  headers: IncomingHttpHeaders;
  httpVersion: string;
  httpVersionMajor: number;
  httpVersionMinor: number;
  cloudevent: CloudEvent;
  cloudEventResponse(data: string|object): CloudEventResponse;
}

// CloudEventResponse is a convenience class used to create
// CloudEvents on function returns
export interface CloudEventResponse {
  id(id: string): CloudEventResponse;
  source(source: string): CloudEventResponse;
  type(type: string): CloudEventResponse;
  version(version: string): CloudEventResponse;
  response(): CloudEvent;
}
```

11.3.3.1.3. CloudEvent 数据

如果传入的请求为 CloudEvent，则从事件中提取与 CloudEvent 相关的任何数据，并作为第二个参数提供。例如，如果收到在它的数据属性中包含类似如下的 JSON 字符串的 CloudEvent：

```
{
  "customerId": "0123456",
  "productId": "6543210"
}
```

在调用时。函数的第二个参数（在上下文 对象后），将是带有 customerId 和 productId 属性的 JavaScript 对象。

签名示例

```
function handle(context: Context, cloudevent?: CloudEvent): CloudEvent
```

本例中的 cloudevent 参数是一个 JavaScript 对象，包含 customerId 和 productId 属性。

11.3.4. TypeScript 功能返回值

功能可以返回任何有效的 JavaScript 类型，或者没有返回值。当函数没有指定返回值且未指示失败时，调用者会收到 204 No Content 响应。

功能也可以返回 CloudEvent 或一个 Message 对象，以便将事件推送到 Knative Eventing 系统。在这种情况下，开发人员不需要了解或实施 CloudEvent 消息传递规范。使用响应提取并发送返回值中的标头和其他相关信息。

示例

```
export const handle: Invokable = function (
```

```
  context: Context,
  cloudevent?: CloudEvent
): Message {
  // process customer and return a new CloudEvent
  const customer = cloudevent.data;
  return HTTP.binary(
    new CloudEvent({
      source: 'customer.processor',
      type: 'customer.processed'
    })
  );
};
```

11.3.4.1. 返回的标头

您可以通过在 return 对象中添加 headers 属性来设置响应标头。这些标头会提取并发送至调用者。

响应标头示例

```
export function handle(context: Context, cloudevent?: CloudEvent): Record<string, any> {
  // process customer and return custom headers
  const customer = cloudevent.data as Record<string, any>;
  return { headers: { 'customer-id': customer.id } };
}
```

11.3.4.2. 返回状态代码

您可以通过在返回对象中添加 statusCode 属性来设置 return 到调用者的状态代码：

状态代码示例

```
export function handle(context: Context, cloudevent?: CloudEvent): Record<string, any> {
  // process customer
  const customer = cloudevent.data as Record<string, any>;
  if (customer.restricted) {
    return {
      statusCode: 451
```

```
    }
  }
  // business logic, then
  return {
    statusCode: 240
  }
}
```

也可以为函数创建和丢弃的错误设置状态代码：

错误状态代码示例

```
export function handle(context: Context, cloudevent?: CloudEvent): Record<string, string> {
  // process customer
  const customer = cloudevent.data as Record<string, any>;
  if (customer.restricted) {
    const err = new Error('Unavailable for legal reasons');
    err.statusCode = 451;
    throw err;
  }
}
```

11.3.5. 测试类型脚本功能

TypeScript 功能可在您的计算机上本地测试。在使用 kn func create 创建功能时创建的默认项目中，有一个 test 目录，其中包含一些简单的单元和集成测试。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (kn) CLI。
- 已使用 kn func create 创建功能。

流程

1. 如果您之前还没有运行测试，请首先安装依赖项：

```
$ npm install
```

2. 导航到您的功能的 test 文件夹。

3. 运行测试：

```
$ npm test
```

11.3.6. 覆盖存活度和就绪度探测值

您可以覆盖 TypeScript 功能的 存活度和就绪度探测 值。这可让您配置在功能上执行的健康检查。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (kn) CLI。
- 已使用 kn func create 创建功能。

流程

1. 在功能代码中，创建 Function 对象，它实现以下接口：

```
export interface Function {
  init?: () => any; 1

  shutdown?: () => any; 2

  liveness?: HealthCheck; 3

  readiness?: HealthCheck; 4

  logLevel?: LogLevel;
```

```
  handle: CloudEventFunction | HTTPFunction; 5
}
```

1

初始化功能，在服务器启动前调用。此函数是可选的，应该同步。

2

关闭功能，在服务器停止后调用。此函数是可选的，应该同步。

3

存活度函数调用，以检查服务器是否处于活动状态。此函数是可选的，如果服务器处于活动状态，应返回 200/OK。

4

就绪度函数调用，以检查服务器是否准备好接受请求。此函数是可选的，如果服务器就绪，应返回 200/OK。

5

处理 HTTP 请求的功能。

例如，将以下代码添加到 index.js 文件中：

```
const Function = {

  handle: (context, body) => {
    // The function logic goes here
    return 'function called'
  },

  liveness: () => {
    process.stdout.write('In liveness\n');
    return 'ok, alive';
  }, 1

  readiness: () => {
    process.stdout.write('In readiness\n');
    return 'ok, ready';
  } 2
};

Function.liveness.path = '/alive'; 3
```

```
Function.readiness.path = '/ready'; 4

module.exports = Function;
```

1 自定义 存活度 功能。

2 自定义 就绪度 功能。

3 自定义 存活度 端点。

4 自定义 就绪度 端点。

作为 Function.liveness.path 和 Function.readiness.path 的替代选择，您可以使用 LIVENESS_URL 和 READINESS_URL 环境变量指定自定义端点：

```
run:
 envs:
 - name: LIVENESS_URL
   value: /alive 1
 - name: READINESS_URL
   value: /ready 2
```

1 存活度路径，设置为 /alive。

2 就绪度路径，设置为 /ready。

2. 将新端点添加到 func.yaml 文件中，以便它们正确绑定到 Knative 服务的容器：

```
deploy:
 healthEndpoints:
   liveness: /alive
   readiness: /ready
```

-

11.3.7. TypeScript 上下文对象引用

该 上下文 对象具有多个属性，可供函数开发人员访问。访问这些属性可提供有关传入 HTTP 请求的信息，并将输出写入集群日志。

11.3.7.1. log

提供一个日志记录对象，可用于将输出写入集群日志。日志遵循 Pino 日志记录 API。

日志示例

```
export function handle(context: Context): string {
  // log the incoming request body's 'hello' parameter
  if (context.body) {
    context.log.info((context.body as Record<string, string>).hello);
  } else {
    context.log.info('No data received');
  }
  return 'OK';
}
```

您可以使用 kn func invoke 命令访问功能：

示例命令

```
$ kn func invoke --target 'http://example.function.com'
```

输出示例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":
1,"msg":"Processing customer"}
```

您可以将日志级别更改为 fatal、error、warn、info、debug、trace 或 silent 之一。为此，请使用 config 命令将其中的一个值分配给环境变量 FUNC_LOG_LEVEL，以更改 logLevel 的值。

11.3.7.2. query

返回请求的查询字符串（如果有），作为键值对。这些属性也可在上下文对象本身中找到。

查询示例

```
export function handle(context: Context): string {
    // log the 'name' query parameter
  if (context.query) {
    context.log.info((context.query as Record<string, string>).name);
  } else {
    context.log.info('No data received');
  }
  return 'OK';
}
```

您可以使用 kn func invoke 命令访问功能：

示例命令

```
$ kn func invoke --target 'http://example.function.com' --data '{"name": "tiger"}'
```

输出示例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":
1,"msg":"tiger"}
```

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":
1,"msg":"tiger"}
```

11.3.7.3. 正文（body）

返回请求正文（如果有）。如果请求正文包含 JSON 代码，这将会进行解析，以便属性可以直接可用。

body 示例

```
export function handle(context: Context): string {
  // log the incoming request body's 'hello' parameter
  if (context.body) {
    context.log.info((context.body as Record<string, string>).hello);
  } else {
    context.log.info('No data received');
  }
  return 'OK';
}
```

您可以使用 kn func invoke 命令访问功能：

示例命令

```
$ kn func invoke --target 'http://example.function.com' --data '{"hello": "world"}'
```

输出示例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":
1,"msg":"world"}
```

11.3.7.4. 标头

将 HTTP 请求标头返回为对象。

标头示例

```
export function handle(context: Context): string {
  // log the incoming request body's 'hello' parameter
  if (context.body) {
    context.log.info((context.headers as Record<string, string>)['custom-header']);
  } else {
    context.log.info('No data received');
  }
  return 'OK';
}
```

您可以使用 curl 命令调用该函数：

示例命令

```
$ curl -H'x-custom-header: some-value'' http://example.function.com
```

输出示例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":
1,"msg":"some-value"}
```

11.3.7.5. HTTP 请求

方法

以字符串形式返回 HTTP 请求方法。

httpVersion

以字符串形式返回 HTTP 版本。

httpVersionMajor

将 HTTP 主版本号返回为字符串。

httpVersionMinor

以字符串形式返回 HTTP 次要版本号。

11.3.8. 后续步骤

- 构建和部署功能.
- 有关使用功能进行日志记录的更多信息，请参阅 Pino API 文档。

11.4. 开发 PYTHON 功能

重要

使用 Python 的 OpenShift Serverless 功能只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议（SLA）支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅技术预览功能支持范围。

创建 PythonG 功能项目后，您可以修改提供的模板文件，以将业务逻辑添加到您的功能中。这包括配置功能调用和返回的标头和状态代码。

11.4.1. 先决条件

- 在开发功能前，您必须完成 配置 OpenShift Serverless 功能 中的步骤。

11.4.2. Python 功能模板结构

使用 Knative (kn) CLI 创建 Python 功能时，项目目录类似于典型的 Python 项目。Python 功能的限制非常少。唯一的要求是项目包含一个 func.py 文件，其中包含一个 main () 函数，以及一个 func.yaml 配置文件。

开发人员不限于模板 requirements.txt 文件中提供的依赖项。可以像在任何其他 Python 项目中一样添加其他依赖项。为部署构建项目时，这些依赖项将包含在创建的运行时容器镜像中。

http 和 event 触发器功能具有相同的模板结构：

模板结构

```
fn
├── func.py 1
├── func.yaml 2
├── requirements.txt 3
└── test_func.py 4
```

1 包含 main () 函数。

2 用于确定镜像名称和 registry。

3 与在任何其他 Python 项目中一样，可以向 requirements.txt 文件中添加其他依赖项。

4 包含一个简单的单元测试，可用于在本地测试您的功能。

11.4.3. 关于调用 Python 功能

Python 功能可以通过简单的 HTTP 请求调用。收到传入请求后，将通过 上下文 对象作为第一个参数来调用函数。

上下文 对象是一个 Python 类，具有两个属性：

- request 属性始终存在，包含 Flask 请求（request） 对象。
- 如果传入请求是 CloudEvent 对象，则第二个属性 cloud_event 会被填充。

开发人员可以从上下文对象访问任何 CloudEvent 数据。

上下文对象示例

```
def main(context: Context):
    """
    The context parameter contains the Flask request object and any
    CloudEvent received with the request.
    """
    print(f"Method: {context.request.method}")
    print(f"Event data {context.cloud_event.data}")
    # ... business logic here
```

11.4.4. Python 功能返回值

功能可以返回 Flask 支持的任何值。这是因为调用框架将这些值直接代理到 Flask 服务器。

示例

```
def main(context: Context):
    body = { "message": "Howdy!" }
    headers = { "content-type": "application/json" }
    return body, 200, headers
```

功能可以将标头和响应代码设置为从函数调用的次要和第三响应值。

11.4.4.1. 返回 CloudEvents

开发人员可以使用 @event decorator 告知调用器，在发送响应前，函数返回值必须转换为 CloudEvent。

示例

```
@event("event_source"="/my/function", "event_type"="my.type")
def main(context):
    # business logic here
    data = do_something()
    # more data processing
    return data
```

这个示例发送 CloudEvent 作为响应值，类型为 "my.type"，源是 "/my/function"。CloudEvent data属性设置为返回的 data 变量。event_source 和 event_type decorator 属性都是可选的。

11.4.5. 测试 Python 功能

您可以在计算机上本地测试 Python 功能。default 项目包含一个 test_func.py 文件，它为函数提供了一个简单的单元测试。

注意

Python 功能的默认测试框架是 unittest。如果您愿意，可以使用不同的测试框架。

先决条件

- 要在本地运行 Python 功能测试，您必须安装所需的依赖项：

```
$ pip install -r requirements.txt
```

流程

1. 导航到包含 test_func.py 文件的函数的文件夹。

2. 运行测试：

```
$ python3 test_func.py
```

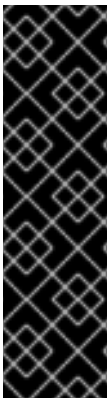
11.4.6. 后续步骤

- 构建和部署功能.

第 12 章 配置功能

12.1. 使用 CLI 从功能访问 SECRET 和配置映射

将功能部署到集群后，可以访问存储在 secret 和配置映射中的数据。此数据可以挂载为卷，或分配到环境变量。您可以使用 Knative CLI 以互动方式配置此访问，或者通过编辑功能配置 YAML 文件来手动配置。



重要

要访问 secret 和配置映射，必须在集群中部署该功能。此功能不适用于本地运行的函数。

如果无法访问 secret 或配置映射值，则部署会失败，并显示一条错误消息，指定不可访问的值。

12.1.1. 以互动方式修改对 secret 和配置映射的功能访问

您可以使用 kn func config 互动程序来管理您的功能访问的 secret 和配置映射。可用的操作包括列表、添加和删除配置映射和 secret 中存储的值，以及列出、添加和删除卷。通过此功能，您可以管理集群中存储的数据，可以被您的功能访问。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (kn) CLI。
- 您已创建了一个功能。

流程

1. 在功能项目目录中运行以下命令：

```
$ kn func config
```

或者，您可以使用 **--path** 或 **-p** 选项指定功能项目目录。

2. 使用交互式界面执行必要的操作。例如，使用工具列出配置的卷会生成类似如下的输出：

```
$ kn func config
? What do you want to configure? Volumes
? What operation do you want to perform? List
Configured Volumes mounts:
- Secret "mysecret" mounted at path: "/workspace/secret"
- Secret "mysecret2" mounted at path: "/workspace/secret2"
```

这个方案显示互动工具中所有可用的操作以及如何导航到它们：

```
kn func config
   ├─> Environment variables
   │              ├─> Add
   │              │    ├─> ConfigMap: Add all key-value pairs from a config map
   │              │    ├─> ConfigMap: Add value from a key in a config map
   │              │    ├─> Secret: Add all key-value pairs from a secret
   │              │    └─> Secret: Add value from a key in a secret
   │              ├─> List: List all configured environment variables
   │              └─> Remove: Remove a configured environment variable
   └─> Volumes
         ├─> Add
         │    ├─> ConfigMap: Mount a config map as a volume
         │    └─> Secret: Mount a secret as a volume
         ├─> List: List all configured volumes
         └─> Remove: Remove a configured volume
```

3. 可选。部署该功能以使更改生效：

```
$ kn func deploy -p test
```

12.1.2. 使用专用命令以互动方式修改对 secret 和配置映射的功能访问

每次运行 **kn func config** 时，您需要浏览整个对话框来选择您需要的操作，如上一节中所示。要保存步骤，您可以通过运行更具体的 **kn func config** 命令来直接执行特定的操作：

- 列出配置的环境变量：

```
$ kn func config envs [-p <function-project-path>]
```

- 在功能配置中添加环境变量：

```
$ kn func config envs add [-p <function-project-path>]
```

- 从功能配置中删除环境变量：

```
$ kn func config envs remove [-p <function-project-path>]
```

- 列出配置的卷：

```
$ kn func config volumes [-p <function-project-path>]
```

- 在功能配置中添加卷：

```
$ kn func config volumes add [-p <function-project-path>]
```

- 从功能配置中删除卷：

```
$ kn func config volumes remove [-p <function-project-path>]
```

12.2. 使用 FUNC.YAML 文件配置功能项目

func.yaml 文件包含功能项目的配置。执行 kn func 命令时使用 func.yaml 中指定的值。例如，当运行 kn func build 命令时，会使用 build 字段中的值。在某些情况下，您可以使用命令行标志或环境变量覆盖这些值。

12.2.1. 从 func.yaml 字段引用本地环境变量

如果要避免在功能配置中存储敏感信息，如 API 密钥，您可以添加对本地环境中可用的环境变量的引用。您可以通过修改 func.yaml 文件中的 envs 字段来完成此操作。

先决条件

- 您需要创建 function 项目。

- 本地环境需要包含您要引用的变量。

流程

- 要引用本地环境变量，请使用以下语法：

```
{{ env:ENV_VAR }}
```

将 ENV_VAR 替换为您要用于本地环境中的变量名称。

例如，您可能在本地环境中提供 API_KEY 变量。您可以将其值分配给 MY_API_KEY 变量，然后您可以在功能内直接使用该变量：

功能示例

```
name: test
namespace: ""
runtime: go
...
envs:
- name: MY_API_KEY
  value: '{{ env:API_KEY }}'
...
```

12.2.2. 在功能中添加注解

您可以将 Kubernetes 注解添加到部署的 Serverless 功能中。注解可让您将任意元数据附加到函数，例如，关于功能目的的备注。注解添加到 func.yaml 配置文件的 annotations 部分。

功能注解功能有两个限制：

- 当功能注解传播到集群中的对应 Knative 服务后，无法通过从 func.yaml 文件中删除该服务来将其从服务中删除。您必须通过直接修改服务的 YAML 文件或使用 OpenShift Container Platform Web 控制台从 Knative 服务中删除注解。

- 您无法设置 Knative 设置的注解，例如 autoscaling 注解。

12.2.3. 在功能中添加注解

您可以在功能中添加注解。与标签类似，注解被定义为键值映射。例如，注解可用于提供与功能相关的元数据，如函数的作者。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (kn) CLI。
- 您已创建了一个功能。

流程

1. 为您的功能打开 func.yaml 文件。
2. 对于您要添加的每个注解，将以下 YAML 添加到 annotations 部分：

```
name: test
namespace: ""
runtime: go
...
annotations:
  <annotation_name>: "<annotation_value>" (1)
```

(1) 将 <annotation_name>: "<annotation_value>" 替换为您的注解。

例如，要指明某个函数由 Alice 编写，您可以包含以下注解：

```
name: test
namespace: ""
runtime: go
```

```
...
annotations:
  author: "alice@example.com"
```

3. 保存配置。

下次将功能部署到集群中时，注解会添加到对应的 Knative 服务中。

12.2.4. 其他资源

- 功能入门
- 自动扩展的 Knative 文档
- 管理容器的资源的 Kubernetes 文档
- 有关配置并发的 Knative 文档

12.2.5. 手动添加对 secret 和配置映射的功能访问

您可以将用于访问 secret 和配置映射的配置手动添加到您的功能中。这可能最好使用 kn func config 交互式实用程序和命令，例如您已有配置片段时。

12.2.5.1. 将 secret 挂载为卷

您可以将 secret 挂载为卷。挂载 secret 后，您可以作为常规文件从函数访问它。这可让您存储在功能所需的集群数据中，例如，函数需要访问的 URI 列表。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (kn) CLI。

- 您已创建了一个功能。

流程

1. 为您的功能打开 func.yaml 文件。
2. 对于您要挂载为卷的每个 secret，将以下 YAML 添加到 volumes 部分：

```
name: test
namespace: ""
runtime: go
...
volumes:
- secret: mysecret
  path: /workspace/secret
```

- 将 mysecret 替换为目标 secret 的名称。
- 将 /workspace/secret 替换为您要挂载 secret 的路径。

例如，要挂载 addresses secret，请使用以下 YAML：

```
name: test
namespace: ""
runtime: go
...
volumes:
- configMap: addresses
  path: /workspace/secret-addresses
```

3. 保存配置。

12.2.5.2. 将配置映射挂载为卷

您可以将配置映射挂载为卷。挂载配置映射后，您可以作为常规文件从函数访问它。这可让您存储在功能所需的集群数据中，例如，函数需要访问的 URI 列表。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (kn) CLI。
- 您已创建了一个功能。

流程

1. 为您的功能打开 func.yaml 文件。
2. 对于您要挂载为卷的每个配置映射，请将以下 YAML 添加到 volumes 部分：

```
name: test
namespace: ""
runtime: go
...
volumes:
- configMap: myconfigmap
  path: /workspace/configmap
```

   - 将 myconfigmap 替换为目标配置映射的名称。
   - 使用您要挂载配置映射的路径替换 /workspace/configmap。

     例如，要挂载 addresses 配置映射，请使用以下 YAML：

```
name: test
namespace: ""
runtime: go
...
volumes:
- configMap: addresses
  path: /workspace/configmap-addresses
```

3. 保存配置。

12.2.5.3. 从 secret 中定义的键值设置环境变量

您可以从定义为 secret 的键值设置环境变量。然后，之前存储在 secret 中的值可以被函数在运行时作为环境变量访问。这有助于获取存储在 secret 中的值，如用户的 ID。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (kn) CLI。
- 您已创建了一个功能。

流程

1. 为您的功能打开 func.yaml 文件。
2. 对于您要分配给环境变量的 secret 键值对的每个值，请将以下 YAML 添加到 envs 部分：

```
name: test
namespace: ""
runtime: go
...
envs:
- name: EXAMPLE
  value: '{{ secret:mysecret:key }}'
```

- 将 EXAMPLE 替换为环境变量的名称。
- 将 mysecret 替换为目标 secret 的名称。
- 使用映射到目标值的键替换 key。

  例如，要访问存储在 userdetailssecret 中的用户 ID，请使用以下 YAML：

```
name: test
namespace: ""
runtime: go
```

```
...
envs:
- value: '{{ configMap:userdetailssecret:userid }}'
```

3. 保存配置。

12.2.5.4. 从配置映射中定义的键值设置环境变量

您可以从定义为配置映射的键值设置环境变量。然后，之前存储在配置映射中的值可以被函数在运行时作为环境变量访问。这对于获取配置映射中存储的值（如用户的 ID）非常有用。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (kn) CLI。
- 您已创建了一个功能。

流程

1. 为您的功能打开 func.yaml 文件。
2. 对于您要分配给环境变量的配置映射键值对中的每个值，请将以下 YAML 添加到 envs 部分：

```
name: test
namespace: ""
runtime: go
...
envs:
- name: EXAMPLE
  value: '{{ configMap:myconfigmap:key }}'
```

   - 将 EXAMPLE 替换为环境变量的名称。
   - 将 myconfigmap 替换为目标配置映射的名称。

- 使用映射到目标值的键替换 key。

  例如，要访问存储在 userdetailsmap 中的用户 ID，请使用以下 YAML：

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ configMap:userdetailsmap:userid }}'
```

3. 保存配置。

12.2.5.5. 从 secret 中定义的所有值设置环境变量

您可以从 secret 中定义的所有值设置环境变量。然后，之前存储在 secret 中的值可以被函数在运行时作为环境变量访问。这可用于同时访问存储在 secret 中的一组值，例如，一组与用户相关的数据。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (kn) CLI。
- 您已创建了一个功能。

流程

1. 为您的功能打开 func.yaml 文件。
2. 对于您要导入所有键值对作为环境变量的每个 secret，请将以下 YAML 添加到 envs 部分：

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ secret:mysecret }}' (1)
```

-

1

将 mysecret 替换为目标 secret 的名称。

例如，要访问存储在 userdetailssecret 中的所有用户数据，请使用以下 YAML：

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ configMap:userdetailssecret }}'
```

3. 保存配置。

12.2.5.6. 从配置映射中定义的所有值设置环境变量

您可以从配置映射中定义的所有值设置环境变量。然后，之前存储在配置映射中的值可以被函数在运行时作为环境变量访问。这可用于同时访问配置映射中存储的值集合，例如，一组与用户相关的数据。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (kn) CLI。
- 您已创建了一个功能。

流程

1. 为您的功能打开 func.yaml 文件。
2. 对于您要导入所有键值对作为环境变量的每个配置映射，请将以下 YAML 添加到 envs 部分：

```
name: test
```

```
namespace: ""
runtime: go
...
envs:
- value: '{{ configMap:myconfigmap }}' (1)
```

(1) 将 myconfigmap 替换为目标配置映射的名称。

例如，要访问存储在 userdetailsmap 中的所有用户数据，请使用以下 YAML：

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ configMap:userdetailsmap }}'
```

3. 保存该文件。

12.3. FUNC.YAML 中的可配置字段

您可以配置一些 func.yaml 字段。

12.3.1. func.yaml 中的可配置字段

在创建、构建和部署您的功能时，func.yaml 中的许多字段会自动生成。但是，您也可以手动修改以更改操作，如函数名称或镜像名称。

12.3.1.1. buildEnvs

buildEnvs 字段允许您设置环境变量，供构建您的功能的环境使用。与使用 envs 设置的变量不同，使用 buildEnv 的变量集合在函数运行时不可用。

您可以直接从值设置 buildEnv 变量。在以下示例中，名为 EXAMPLE1 的 buildEnv 变量被直接分配为 one 值：

```
buildEnvs:
- name: EXAMPLE1
  value: one
```

您还可以从本地环境变量设置 buildEnv 变量。在以下示例中，名为 EXAMPLE2 的 buildEnv 变量被分配了 LOCAL_ENV_VAR 本地环境变量的值：

```
buildEnvs:
- name: EXAMPLE1
  value: '{{ env:LOCAL_ENV_VAR }}'
```

12.3.1.2. envs

envs 字段允许您在运行时设置环境变量供您的功能使用。您可以通过几种不同方式设置环境变量：

1. 直接来自一个值。
2. 来自分配给本地环境变量的值）。如需更多信息，请参阅"引用来自 func.yaml 字段中的本地环境变量"。
3. 从存储在 secret 或配置映射中的键值对。
4. 您还可以导入存储在 secret 或配置映射中的所有键值对，其键用作所创建的环境变量的名称。

这个示例演示了设置环境变量的不同方法：

```
name: test
namespace: ""
runtime: go
...
envs:
- name: EXAMPLE1 1
  value: value
- name: EXAMPLE2 2
  value: '{{ env:LOCAL_ENV_VALUE }}'
- name: EXAMPLE3 3
  value: '{{ secret:mysecret:key }}'
- name: EXAMPLE4 4
```

```
  value: '{{ configMap:myconfigmap:key }}'
- value: '{{ secret:mysecret2 }}' 5
- value: '{{ configMap:myconfigmap2 }}' 6
```

1

直接从值设置的环境变量。

2

从分配给本地环境变量的值设置的环境变量。

3

从存储在 secret 中的键值对分配的环境变量。

4

从配置映射中存储的键值对分配的环境变量。

5

从 secret 的键值对导入的一组环境变量。

6

从配置映射的键值对导入的一组环境变量。

12.3.1.3. builder

builder 字段指定函数用于构建镜像的策略。它接受 pack 或 s2i 的值。

12.3.1.4. build

build 字段指示如何构建函数。local 值表示该函数在您的机器上本地构建。git 值表示函数使用 git 字段中指定的值来在集群中构建。

12.3.1.5. 卷

volumes 字段允许您将 secret 和配置映射作为可在指定路径的函数访问的卷挂载，如下例所示：

```
name: test
```

```
namespace: ""
runtime: go
...
volumes:
- secret: mysecret (1)
  path: /workspace/secret
- configMap: myconfigmap (2)
  path: /workspace/configmap
```

(1) mysecret secret 作为驻留于 /workspace/secret 的卷挂载。

(2) myconfigmap 配置映射作为驻留于 /workspace/configmap 的卷挂载。

12.3.1.6. 选项

options 字段允许您修改部署的功能的 Knative Service 属性，如自动扩展。如果未设置这些选项，则使用默认的选项。

这些选项可用：

- scale
  - min：最小副本数。必须是一个非负的整数。默认值为 0。
  - max ：最大副本数。必须是一个非负的整数。默认值为 0，这代表没有限制。
  - metric ：定义 Autoscaler 监视哪一指标类型。它可以被设置为 concurrency（默认），或 rps。
  - target：建议根据同时传入的请求数量，何时向上扩展。target 选项可以是大于 0.01 的浮点值。除非设置了 options.resources.limits.concurrency，否则默认为 100，在这种情况下，目标默认为其值。
  - utilization：向上扩展前允许的并发请求利用率百分比.它可以是 1 到 100 之间的一个

浮点值。默认值为 70。

- 资源
  - requests
    - cpu：具有部署功能的容器的 CPU 资源请求。
    - memory：具有部署功能的容器的内存资源请求。
  - limits
    - cpu：具有部署功能的容器的 CPU 资源限值。
    - memory：具有部署功能的容器的内存资源限制。
    - concurrency：单个副本处理的并发请求的硬限制。它可以是大于或等于 0 的整数值，默认为 0 - 表示无限制。

这是 scale 选项配置示例：

```
name: test
namespace: ""
runtime: go
...
options:
  scale:
    min: 0
    max: 10
    metric: concurrency
    target: 75
    utilization: 75
  resources:
    requests:
      cpu: 100m
      memory: 128Mi
    limits:
```

```
      cpu: 1000m
      memory: 256Mi
      concurrency: 100
```

12.3.1.7. image

image 字段在构建后为您的功能设置镜像名称。您可以修改此字段。如果您这样做，在下次运行 kn func build 或 kn func deploy 时，功能镜像将使用新名称创建。

12.3.1.8. imageDigest

在部署函数时，imageDigest 字段包含镜像清单的 SHA256 哈希。不要修改这个值。

12.3.1.9. labels

labels 字段允许您在部署的功能中设置标签。

您可以直接从值设置标签。在以下示例中，带有 role 键的标签直接被分配了 backend 的值：

```
labels:
- key: role
  value: backend
```

您还可以从本地环境变量设置标签。在以下示例中，为带有 author 键的标签分配 USER 本地环境变量的值：

```
labels:
- key: author
  value: '{{ env:USER }}'
```

12.3.1.10. name

name 字段定义您的函数的名称。该值在部署时用作 Knative 服务的名称。您可以更改此字段来重命名后续部署中的函数。

12.3.1.11. namespace

namespace 字段指定部署您的功能的命名空间。

12.3.1.12. runtime

runtime 字段指定您的功能的语言运行时，如 python。