



# HDZ

## Huawei Developer Zone

---

**开放、创新、多元**的开发者社区组织



# Knative应用开发指南

当当网云原生实验室 李志伟

## 个人简介

---

李志伟

当当网云原生实验室负责人，负责云原生产品线的研发以及落地工作。

[lizhiwei@dangdang.com](mailto:lizhiwei@dangdang.com)

微信：



## 图书推荐

### Knative实战--基于Kubernetes的无服务器架构实践

作者：李志伟、游杨

出版社：机械工业出版社

读者对象:

- ✓Serverless应用开发者
- ✓Knative运维开发人员
- ✓对Serverless技术感兴趣的读者
- ✓云原生架构师

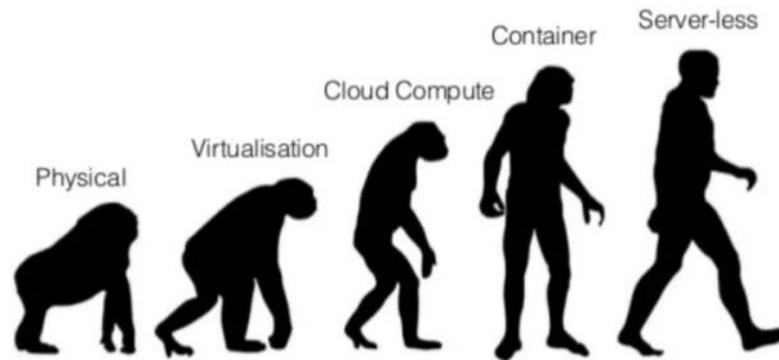


# 什么是Serverless

## Serverless = FaaS + BaaS

无服务器架构：开发者实现的业务逻辑运行在无状态的计算容器中，它由事件触发，完全被第三方管理，其业务层面的状态存储在数据库和其他存储资源中。

- ✓ 无需管理基础设施
- ✓ 按用付费
- ✓ 事件驱动
- ✓ 自动化构建部署



## 为什么 Serverless

---

“Serverless 计算将会成为云时代默认的计算范式，并取代 Serverful (传统云) 计算模式。”

“Serverless 简化了云计算的编程，代表了程序员生产力的又一次的变革，就像编程语言从汇编时代演变为高级语言时代。”

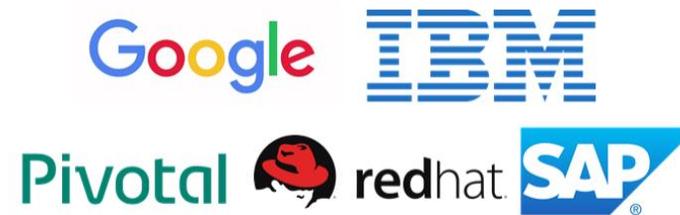
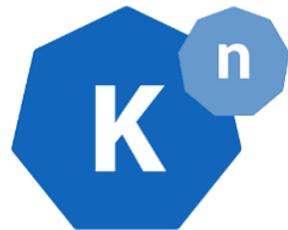
-- UC Berkeley 2019

## 关于Knative

---

Knative 是谷歌发起的基于K8S平台的Serverless 开源项目, 致力将Serverless标准化。它基于K8S平台, 用于部署和管理现代无服务器工作负载。

项目地址 : <https://github.com/knative>



# 关于Knative

## Knative在K8S生态中的定位



# 关于Knative

---

Knative是为开发者准备的Serverless平台

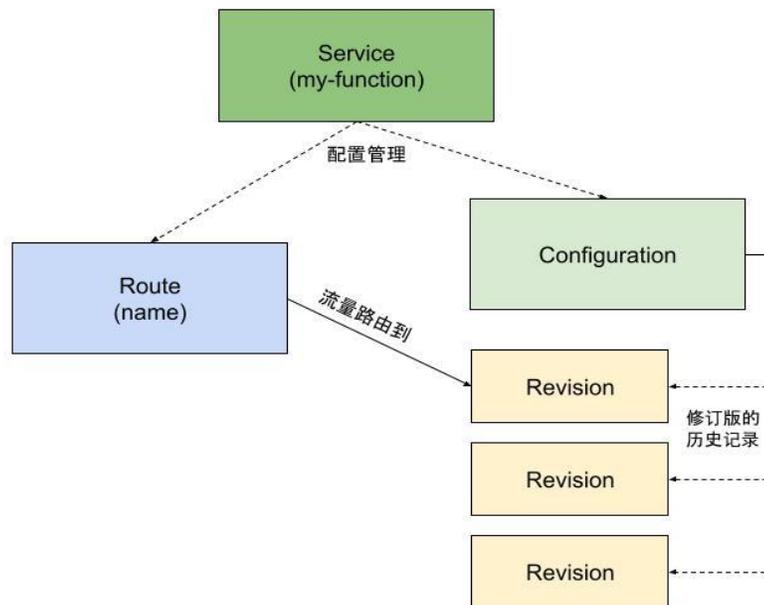
- 抽象复杂的细节让开发者关注在业务本身 ( Service-based , Event-driven )
- 解决无聊却困难的构建 , 部署 , 服务治理步骤
- 开放和可移植 ( Open & portable )

# Knative的总体设计

Serving	Eventing
服务管理	事件驱动
<ul style="list-style-type: none"><li>✓ 请求驱动计算，支持伸缩到零</li><li>✓ 按需自动伸缩工作负载的大小</li><li>✓ 支持蓝绿部署及流量管理</li></ul>	<ul style="list-style-type: none"><li>✓ 管理和交付事件</li><li>✓ 将服务绑定到事件</li><li>✓ 抽象发布/订阅细节</li></ul>
 Knative	

# Knative的服务管理组件解决了哪些问题

- ✓ 实现了用户容器的快速部署
- ✓ 自动伸缩，支持缩容到零
- ✓ 更高级的服务路由和流量控制
- ✓ 容器和配置的版本管理



# Knative开发实战

## 服务的构建与编排

### 1. 创建一个简单的web服务程序helloworld.go

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "os"
)

func handler(w http.ResponseWriter, r *http.Request) {
    log.Print("helloworld: received a request")
    target := os.Getenv("TARGET")
    if target == "" {
        target = "World"
    }
    fmt.Fprintf(w, "Hello %s!\n", target)
}

func main() {
    log.Print("helloworld: starting server...")

    http.HandleFunc("/", handler)

    port := os.Getenv("PORT")
    if port == "" {
        port = "8080"
    }

    log.Printf("helloworld: listening on port %s", port)
    log.Fatal(http.ListenAndServe(fmt.Sprintf(":%s", port), nil))
}
```

# Knative开发实战

## 服务的构建与编排

1. 创建一个简单的web服务程序helloworld.go
2. 使用Dockerfile构建源码并生成容器镜像

```
FROM golang:1.13 as builder
WORKDIR /app
COPY go.* ./
RUN go mod download

# Copy local code to the container image.
COPY . ./
RUN CGO_ENABLED=0 GOOS=linux go build -mod=readonly -v -o server

FROM alpine:3
RUN apk add --no-cache ca-certificates

# Copy the binary to the production image from the builder stage.
COPY --from=builder /app/server /server

# Run the web service on container startup.
CMD ["/server"]
```

```
# 在本地主机构建容器。
docker build -t {username}/helloworld-go .
```

```
# 将容器Push到Docker容器镜像仓库
docker push {username}/helloworld-go
```

# Knative开发实战

## 服务的构建与编排

1. 创建一个简单的web服务程序helloworld.go
2. 使用Dockerfile构建源码并生成容器镜像
3. 部署helloworld-go服务

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: helloworld-go # Service名称
  namespace: default
spec:
  template:
    metadata:
      name: helloworld-go-v1 # Knative Revision名称, 如果未设置系统将会自动生成。
    annotations:
      autoscaling.knative.dev/class: "kpa.autoscaling.knative.dev"
      # Autoscaler的实现方式, 可选值有"kpa.autoscaling.knative.dev" 或 "hpa.autoscaling.knative.dev"
      autoscaling.knative.dev/metric: "concurrency" # 度量指标为concurrency (默认值), 还可以选择
      # rps或cpu。
      autoscaling.knative.dev/target: "10" # 设置单个Pod最大并发数为10, 默认值为100。
      autoscaling.knative.dev/minScale: "1" # minScale表示最小保留实例数为1
      autoscaling.knative.dev/maxScale: "100" # maxScale表示最大扩容实例数为100
    spec:
      containers:
      - image: {username}/helloworld-go
        env:
        - name: TARGET
          value: "Go Sample v1"
        livenessProbe:
          httpGet:
            path: /
        readinessProbe:
          httpGet:
            path: /
```

#运行下面的命令部署服务  
kubectl apply -f service.yaml

# Knative开发实战

## 服务的构建与编排

1. 创建一个简单的web服务程序helloworld.go
2. 使用Dockerfile构建源码并生成容器镜像
3. 部署helloworld-go服务
4. **版本更新到v2**

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: helloworld-go # Service名称
  namespace: default
spec:
  template:
    metadata:
      name: helloworld-go-v2 # Knative Revision名称
      annotations:
        autoscaling.knative.dev/class: "kpa.autoscaling.knative.dev"
        # Autoscaler的实现方式,可选值有"kpa.autoscaling.knative.dev"或"hpa.autoscaling.knative.dev"
        autoscaling.knative.dev/metric: "concurrency" # 度量指标为concurrency(默认值),还可以选
        # 择rps或cpu。
        autoscaling.knative.dev/target: "10" # 设置单个Pod最大并发数为10,默认值为100。
        autoscaling.knative.dev/minScale: "1" # minScale表示最小保留实例数为1
        autoscaling.knative.dev/maxScale: "100" # maxScale表示最大扩容实例数为100
    spec:
      containers:
        - image: {username}/helloworld-go
          env:
            - name: TARGET
              value: "Go Sample v2"
          livenessProbe:
            httpGet:
              path: /
          readinessProbe:
            httpGet:
              path: /
```

#运行下面的命令部署服务  
kubectl apply -f service.yaml

# Knative开发实战

## 服务的构建与编排

1. 创建一个简单的web服务helloworld.go
2. 使用Dockerfile构建源码并生成容器镜像
3. 部署helloworld-go服务
4. 版本更新到v2
5. 灰度发布

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: helloworld-go # Service名称
  namespace: default
spec:
  template:
    metadata:
      name: helloworld-go-v2 # Knative Revision名称
    spec:
      containers:
        - image: cnlab/helloworld-go
          env:
            - name: TARGET
              value: "Go Sample v2"
          livenessProbe:
            httpGet:
              path: /
          readinessProbe:
            httpGet:
              path: /
      traffic:
        - tag: v1
          revisionName: helloworld-go-v1 # Revision的名称
          percent: 80 #流量切分的百分比的数字值。
        - tag: v2
          revisionName: helloworld-go-v2 # Revision的名称
          percent: 20 #流量切分的百分比的数字值。
        - tag: latest #默认最新的revision。
          latestRevision: true
          percent: 0 #关闭默认流量分配。
```

#运行下面的命令部署服务  
kubectl apply -f service.yaml

# Knative开发实战

---

## 服务的构建与编排

1. 创建一个简单的web服务helloworld.go
2. 使用Dockerfile构建源码并生成容器镜像
3. 部署helloworld-go服务
4. 版本更新到v2
5. 灰度发布
6. 访问测试

```
# curl -H "Host:helloworld-go.default.example.com" http://$IP_ADDRESS  
Hello Go Sample v1!  
# curl -H "Host:helloworld-go.default.example.com" http://$IP_ADDRESS  
Hello Go Sample v2!。
```

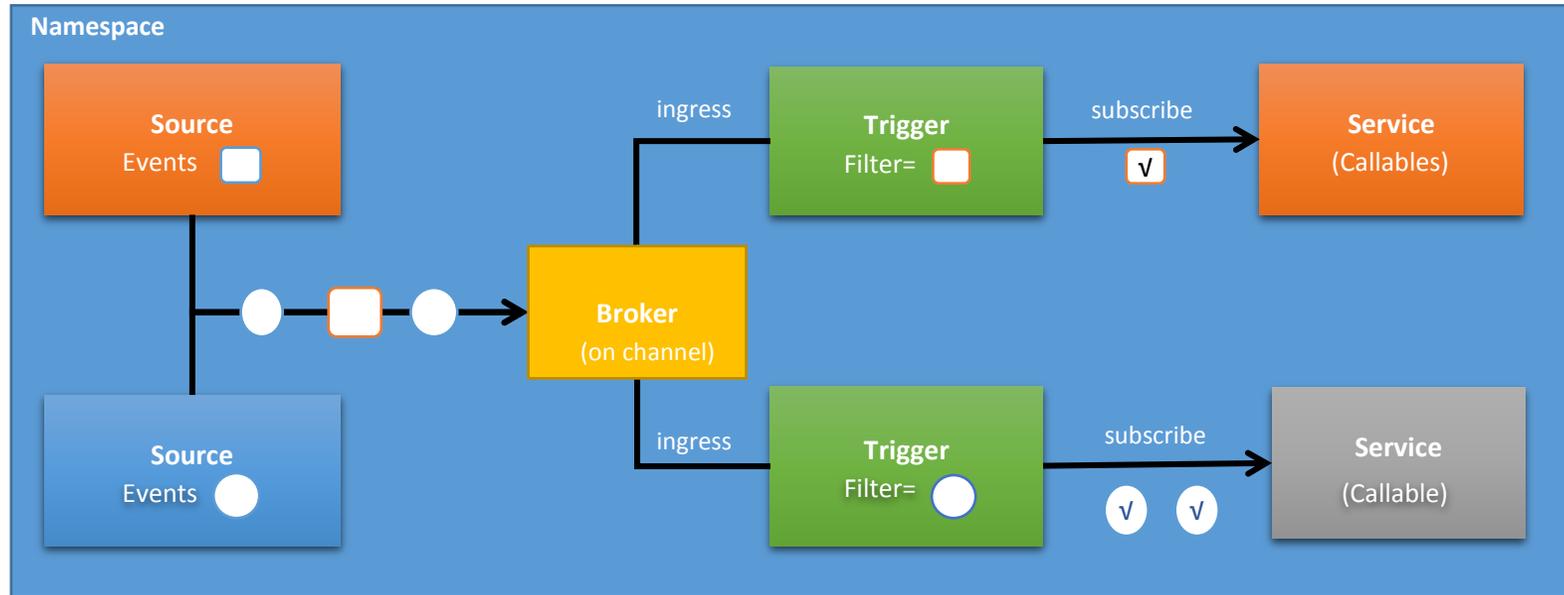
# Knative Eventing

---

**核心功能：对发布/订阅细节进行抽象处理，帮助开发人员摆脱相关负担**

- ✓ 声明式地绑定event sources,triggers和services
- ✓ 从少量事件到实时stream pipelines动态扩展
- ✓ 采用CloudeEvents标准
- ✓ 抽象的事件来源，解耦具体事件源类型(eg. Kafka,CronJob,Github,kubernetes ... )
- ✓ Channel的实现可插拔(eg. InMemory, Kafka, Nats, PubSub )

# Knative Eventing



**Sources:** 抽象的事件来源

**Channel:** 处理事件的转发和持久化

**Broker:** 把事件发送给订阅者

**Trigger:** 订阅来自特定broker的事件

# Knative开发实战

## 事件驱动应用范例

### 1. 创建一个简单的事件处理程序helloworld.go

```
import (  
    "context"  
    "log"  
  
    cloudevents "github.com/cloudevents/sdk-go/v2"  
    "github.com/google/uuid"  
)  
  
func receive(ctx context.Context, event cloudevents.Event) (*cloudevents.Event,  
cloudevents.Result) {  
    log.Printf("Event received. \n%s\n", event)  
    data := &HelloWorld{}  
    if err := event.DataAs(data); err != nil {  
        log.Printf("Error while extracting cloudevent Data: %s\n", err.Error())  
        return nil, cloudevents.NewHTTPResult(400, "failed to convert data: %s", err)  
    }  
    log.Printf("Hello World Message from received event %q", data.Msg)  
  
    // Respond with another event (optional)  
    newEvent := cloudevents.NewEvent()  
    newEvent.SetID(uuid.New().String())  
    newEvent.SetSource("knative/eventing/samples/hello-world")  
    newEvent.SetType("dev.knative.samples.hifromknative")  
    if err := newEvent.SetData(cloudevents.ApplicationJSON, HiFromKnative{Msg: "Hi from  
helloworld-go app!"}); err != nil {  
        return nil, cloudevents.NewHTTPResult(500, "failed to set response data: %s", err)  
    }  
    log.Printf("Responding with event\n%s\n", newEvent)  
    return &newEvent, nil  
}  
  
func main() {  
    log.Print("Hello world sample started.")  
    c, err := cloudevents.NewDefaultClient()  
    if err != nil {  
        log.Fatalf("failed to create client, %v", err)  
    }  
    log.Fatal(c.StartReceiver(context.Background(), receive))  
}
```

# Knative开发实战

## 事件驱动应用范例

1. 创建一个简单的事件处理程序helloworld.go
2. 定义Dockerfile构建源码并生成容器镜像

```
# Use the official Golang image to create a build artifact.
# This is based on Debian and sets the GOPATH to /go.
# https://hub.docker.com/_/golang
FROM golang:1.14 as builder

# Copy local code to the container image.
WORKDIR /app

# Retrieve application dependencies using go modules.
# Allows container builds to reuse downloaded dependencies.
COPY go.* ./
RUN go mod download

# Copy local code to the container image.
COPY . ./

# Build the binary.
# -mod=readonly ensures immutable go.mod and go.sum in container builds.
RUN CGO_ENABLED=0 GOOS=linux go build -mod=readonly -v -o helloworld

# Use a Docker multi-stage build to create a lean production image.
# https://docs.docker.com/develop/develop-images/multistage-build/#use-multi-stage-builds
FROM alpine:3
RUN apk add --no-cache ca-certificates

# Copy the binary to the production image from the builder stage.
COPY --from=builder /app/helloworld /helloworld

# Run the web service on container startup.
CMD ["/helloworld"]
```

# 在本地主机构建容器。

```
docker build -t {username}/helloworld-go .
```

# 将容器Push到Docker容器镜像仓库

```
docker push {username}/helloworld-go
```

# Knative开发实战

## 事件驱动应用范例

1. 创建一个简单的事件处理程序helloworld.go
2. 定义Dockerfile构建源码并生成容器镜像
3. 部署事件处理服务

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: helloworld-go
  namespace: knative-samples
spec:
  replicas: 1
  selector:
    matchLabels: &labels
    app: helloworld-go
  template:
    metadata:
      labels: *labels
    spec:
      containers:
        - name: helloworld-go
          image: docker.io/{username}/helloworld-go
---
# Service that exposes helloworld-go app.
kind: Service
apiVersion: v1
metadata:
  name: helloworld-go
  namespace: knative-samples
spec:
  selector:
    app: helloworld-go
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

#运行下面的命令部署服务  
kubectl apply -f service.yaml

# Knative开发实战

## 事件驱动应用范例

1. 创建一个简单的事件处理程序helloworld.go
2. 定义Dockerfile构建源码并生成容器镜像
3. 部署事件处理服务
4. 定义Broker和Trigger

```
# A default broker
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  name: default
  namespace: knative-samples
  annotations:
    eventing.knative.dev/broker.class: MTChannelBasedBroker
spec: {}

---
# Knative Eventing Trigger to trigger the helloworld-go service
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: helloworld-go
  namespace: knative-samples
spec:
  broker: default
  filter:
    attributes:
      type: dev.knative.samples.helloworld
      source: dev.knative.samples/helloworldsource
  subscriber:
    ref:
      apiVersion: v1
      kind: Service
      name: helloworld-go
```

#运行下面的命令完成服务的事件订阅  
kubectl apply -f subscription.yaml

# Knative开发实战

## 事件驱动应用范例

1. 创建一个简单的事件处理程序helloworld.go
2. 定义Dockerfile构建源码并生成容器镜像
3. 部署事件处理服务
4. 定义Broker和Trigger
5. 测试验证

```
# 创建一个带有curl命令的Pod
$ kubectl -n knative-samples run curl --image=radial/busyboxplus:curl -it

# 通过curl命令向Broker发起请求
[ root@curl:/ ]$ curl -v "broker-ingress.knative-
eventing.svc.cluster.local/knative-samples/default" \
-X POST \
-H "Ce-Id: 536808d3-88be-4077-9d7a-a3f162705f79" \
-H "Ce-specversion: 1.0" \
-H "Ce-Type: dev.knative.samples.helloworld" \
-H "Ce-Source: dev.knative.samples/helloworldsource" \
-H "Content-Type: application/json" \
-d '{"msg":"Hello World from the curl pod."}'

# kubectl --namespace knative-samples logs -l app=helloworld-go --tail=50
Event received.
Validation: valid
Context Attributes,
specversion: 1.0
type: dev.knative.samples.helloworld
source: dev.knative.samples/helloworldsource
id: 536808d3-88be-4077-9d7a-a3f162705f79
time: 2019-10-04T22:35:26.05871736Z
datacontenttype: application/json
Extensions,
knativearrivaltime: 2019-10-04T22:35:26Z
knativehistory: default-kn2-trigger-kn-channel.knative-samples.svc.cluster.local
traceparent: 00-971d4644229653483d38c46e92a959c7-92c66312e4bb39be-00
Data,
{"msg":"Hello World from the curl pod."}

Hello World Message "Hello World from the curl pod."
Responded with event
Validation: valid
Context Attributes,
specversion: 1.0
type: dev.knative.samples.hifromknative
source: knative/eventing/samples/hello-world
id: 37458d77-01f5-411e-a243-a459bbf79682
datacontenttype: application/json
Data,
{"msg":"Hi from Knative!"}
```

## 总结

---

- Knative秉承请求驱动计算，计算资源按需分配原则，可以有效提高计算资源的利用率
- 无服务器计算可以显著地减少运维工作量
- 业务代码与平台代码的隔离带来开发的敏捷性，使得开发者工作效能更高
- 计算资源脱离服务器的抽象的无服务器架构是云计算未来的发展趋势



# 谢谢

