微服务框架：如果不用Spring Boot，还可以选择谁 (https://www.kubernetes.org.cn/9526.html)

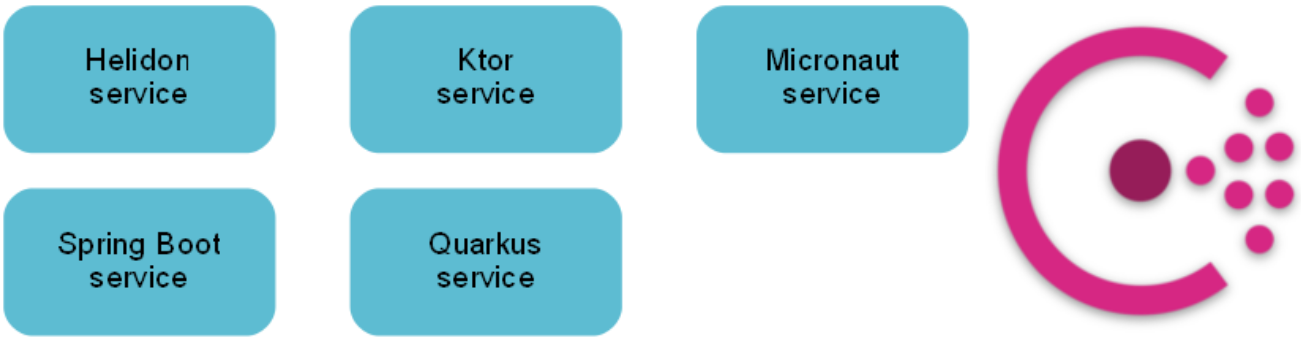
2021-08-01 08:43　王延飞 (https://www.kubernetes.org.cn/author/fly)　分类：微服务 (https://www.kubernetes.org.cn/microservices)　阅读(2335)　　评论(0)

前言

在 Java 和 Kotlin 中, 除了使用Spring Boot创建微服务外，还有很多其他的替代方案。

名称	版本	发布时间	开发商	GitHub
Helidon SE (https://helidon.io/)	1.4.1	2019年	甲骨文	链接 (https://github.com/oracle/helidon)
Ktor (https://ktor.io/)	1.3.0	2018年	JetBrains	链接 (https://github.com/ktorio/ktor)
Micronaut (https://micronaut.io/)	1.2.9	2018年	Object Computing	链接 (https://github.com/micronaut-projects/micronaut-core)
Quarkus (https://quarkus.io/)	1.2.0	2019年	Red Hat	链接 (https://github.com/quarkusio/quarkus)
Spring Boot (https://spring.io/projects/spring-boot)	2.2.4	2014年	Pivotal	链接 (https://github.com/spring-projects/spring-boot)

本文，基于这些微服务框架，创建了五个服务，并使用Consul (https://www.consul.io/)的服务发现模式实现服务间的相互通信。因此，它们形成了异构微服务架构（Heterogeneous Microservice Architecture, 以下简称 MSA）：



本文简要考虑了微服务在各个框架上的实现（更多细节请查看源代码：https (https://github.com/rkudryashov/heterogeneous-microservices) : //github.com/rkudryashov/heterogeneous-microservices (https://github.com/rkudryashov/heterogeneous-microservices)）

- 技术栈：

> - JDK 13
> - Kotlin
> - Gradle (Kotlin DSL)
> - JUnit 5

- 功能接口（HTTP API）：

  - GET /application-info{?request-to=some-service-name}
    - — 返回微服务的一些基本信息（名称、框架、发布年份）
  - GET /application-info/logo
    - — 返回logo信息

- 实现方式：

  - 使用文本文件的配置方式
  - 使用依赖注入
  - HTTP API

- MSA：

  - 使用服务发现模式（在Consul中注册，通过客户端负载均衡的名称请求另一个微服务的HTTP API）
  - 构建一个 uber-JAR

先决条件

- JDK 13
- Consul (https://www.consul.io/)

从头开始创建应用程序

要基于其中一个框架上生成新项目，你可以使用web starter 或其他选项（例如，构建工具或 IDE）：

名称	Web starter	指南	支持的开发语言
Helidon	链接 (https://start.microprofile.io/)(MP)	链接 (https://helidon.io/docs/latest/#/guides/02_quickstart-se)(SE) 链接 (https://helidon.io/docs/latest/#/guides/03_quickstart-mp)(MP)	Java，Kotlin
Ktor	链接 (https://start.ktor.io/)	链接 (https://ktor.io/quickstart/index.html)	Kotlin
Micronaut	链接 (https://docs.micronaut.io/latest/guide/index.html#cli)	链接 (https://guides.micronaut.io/creating-your-first-micronaut-app/guide)	Groovy、Java、Kotlin
Quarkus	链接 (https://code.quarkus.io/)	链接 (https://quarkus.io/guides/getting-started)	Java、Kotlin、Scala

名称	Web starter	指南	支持的开发语言
Spring Boot	链接 (https://start.spring.io/)	链接 (https://spring.io/guides/gs/spring-boot)	Groovy、Java、Kotlin

Helidon服务

该框架是在 Oracle 中创建以供内部使用，随后成为开源。Helidon 非常简单和快捷，它提供了两个版本：标准版（SE）和MicroProfile（MP）。在这两种情况下，服务都是一个常规的 Java SE 程序。（在Helidon (https://github.com/oracle/helidon/wiki/FAQ)上了解更多信息）

Helidon MP 是 Eclipse MicroProfile (https://microprofile.io/)的实现之一，这使得使用许多 API 成为可能，包括 Java EE 开发人员已知的（例如 JAX-RS、CDI等）和新的 API（健康检查、指标、容错等）。在 Helidon SE 模型中，开发人员遵循“没有魔法”的原则，例如，创建应用程序所需的注解数量较少或完全没有。

Helidon SE 被选中用于微服务的开发。因为Helidon SE 缺乏依赖注入的手段，因此为此使用了Koin (https://insert-koin.io/)。

以下代码示例，是包含 main 方法的类。为了实现依赖注入，该类继承自KoinComponent。

> 首先，Koin 启动，然后初始化所需的依赖并调用startServer()方法—-其中创建了一个WebServer类型的对象，应用程序配置和路由设置传递到该对象；
>
> 启动应用程序后在Consul注册：

```
object HelidonServiceApplication : KoinComponent {

    @JvmStatic
    fun main(args: Array<String>) {
        val startTime = System.currentTimeMillis()
        startKoin {
            modules(koinModule)
        }

        val applicationInfoService: ApplicationInfoService by inject()
        val consulClient: Consul by inject()
        val applicationInfoProperties: ApplicationInfoProperties by inject()
        val serviceName = applicationInfoProperties.name

        startServer(applicationInfoService, consulClient, serviceName, startTime)
    }
}

fun startServer(
    applicationInfoService: ApplicationInfoService,
    consulClient: Consul,
    serviceName: String,
    startTime: Long
): WebServer {
    val serverConfig = ServerConfiguration.create(Config.create().get("webserver"))

    val server: WebServer = WebServer
        .builder(createRouting(applicationInfoService))
        .config(serverConfig)
        .build()

    server.start().thenAccept { ws ->
        val durationInMillis = System.currentTimeMillis() - startTime
        log.info("Startup completed in $durationInMillis ms. Service running at: http://localhost:" + ws.por
        // register in Consul
        consulClient.agentClient().register(createConsulRegistration(serviceName, ws.port()))
    }

    return server
}
```

路由配置如下：

```
private fun createRouting(applicationInfoService: ApplicationInfoService) = Routing.builder()
    .register(JacksonSupport.create())
    .get("/application-info", Handler { req, res ->
        val requestTo: String? = req.queryParams()
            .first("request-to")
            .orElse(null)

        res
            .status(Http.ResponseStatus.create(200))
            .send(applicationInfoService.get(requestTo))
    })
    .get("/application-info/logo", Handler { req, res ->
        res.headers().contentType(MediaType.create("image", "png"))
        res
            .status(Http.ResponseStatus.create(200))
            .send(applicationInfoService.getLogo())
    })
    .error(Exception::class.java) { req, res, ex ->
        log.error("Exception:", ex)
        res.status(Http.Status.INTERNAL_SERVER_ERROR_500).send()
    }
    .build()
```

该应用程序使用HOCON (https://en.wikipedia.org/wiki/HOCON)格式的配置文件：

```
webserver {
  port: 8081
}

application-info {
  name: "helidon-service"
  framework {
    name: "Helidon SE"
    release-year: 2019
  }
}
```

还可以使用 JSON、YAML 和properties 格式的文件进行配置（在Helidon 配置文档 (https://helidon.io/docs/latest/#/config/01_introduction)中了解更多信息）。

Ktor服务

该框架是为 Kotlin 编写和设计的。和 Helidon SE 一样，Ktor 没有开箱即用的 DI，所以在启动服务器依赖项之前应该使用 Koin 注入：

```
val koinModule = module {
    single { ApplicationInfoService(get(), get()) }
    single { ApplicationInfoProperties() }
    single { ServiceClient(get()) }
    single { Consul.builder().withUrl("https://localhost:8500").build() }
}

fun main(args: Array<String>) {
    startKoin {
        modules(koinModule)
    }
    val server = embeddedServer(Netty, commandLineEnvironment(args))
    server.start(wait = true)
}
```

应用程序需要的模块在配置文件中指定（HOCON格式；更多配置信息参考Ktor配置文档(https://ktor.io/docs/configurations.html)），其内容如下：

```
ktor {
  deployment {
    host = localhost
    port = 8082
    environment = prod
    // for dev purpose
    autoreload = true
    watch = [io.heterogeneousmicroservices.ktorservice]
  }
  application {
    modules = [io.heterogeneousmicroservices.ktorservice.module.KtorServiceApplicationModuleKt.module]
  }
}

application-info {
  name: "ktor-service"
  framework {
    name: "Ktor"
    release-year: 2018
  }
}
```

在 Ktor 和 Koin 中，术语“模块”具有不同的含义。

在 Koin 中，模块类似于 Spring 框架中的应用程序上下文。Ktor的模块是一个用户定义的函数，它接受一个Application类型的对象，可以配置流水线、注册路由、处理请求等：

```
fun Application.module() {
    val applicationInfoService: ApplicationInfoService by inject()

    if (!isTest()) {
        val consulClient: Consul by inject()
        registerInConsul(applicationInfoService.get(null).name, consulClient)
    }

    install(DefaultHeaders)
    install(Compression)
    install(CallLogging)
    install(ContentNegotiation) {
        jackson {}
    }

    routing {
        route("application-info") {
            get {
                val requestTo: String? = call.parameters["request-to"]
                call.respond(applicationInfoService.get(requestTo))
            }
            static {
                resource("/logo", "logo.png")
            }
        }
    }
}
```

> 此代码是配置请求的路由，特别是静态资源logo.png。

下面是基于Round-robin算法结合客户端负载均衡实现服务发现模式的代码：

```
class ConsulFeature(private val consulClient: Consul) {

    class Config {
        lateinit var consulClient: Consul
    }

    companion object Feature : HttpClientFeature<Config, ConsulFeature> {
        var serviceInstanceIndex: Int = 0

        override val key = AttributeKey<ConsulFeature>("ConsulFeature")

        override fun prepare(block: Config.() -> Unit) = ConsulFeature(Config().apply(block).consulClient)

        override fun install(feature: ConsulFeature, scope: HttpClient) {
            scope.requestPipeline.intercept(HttpRequestPipeline.Render) {
                val serviceName = context.url.host
                val serviceInstances =
                    feature.consulClient.healthClient().getHealthyServiceInstances(serviceName).response
                val selectedInstance = serviceInstances[serviceInstanceIndex]
                context.url.apply {
                    host = selectedInstance.service.address
                    port = selectedInstance.service.port
                }
                serviceInstanceIndex = (serviceInstanceIndex + 1) % serviceInstances.size
            }
        }
    }
}
```

主要逻辑在install方法中：在Render请求阶段（在Send阶段之前执行）首先确定被调用服务的名称，然后consulClient请求服务的实例列表，然后通过循环算法定义一个实例正在调用。因此，以下调用成为可能：

```
fun getApplicationInfo(serviceName: String): ApplicationInfo = runBlocking {
    httpClient.get<ApplicationInfo>("http://$serviceName/application-info")
}
```

Micronaut 服务

Micronaut 由Grails (https://grails.org/)框架的创建者开发，灵感来自使用 Spring、Spring Boot 和 Grails 构建服务的经验。该框架目前支持 Java、Kotlin 和 Groovy 语言。依赖是在编译时注入的，与 Spring Boot 相比，这会导致更少的内存消耗和更快的应用程序启动。

主类如下所示：

```
object MicronautServiceApplication {

    @JvmStatic
    fun main(args: Array<String>) {
        Micronaut.build()
            .packages("io.heterogeneousmicroservices.micronautservice")
            .mainClass(MicronautServiceApplication.javaClass)
            .start()
    }
}
```

基于 Micronaut 的应用程序的某些组件与它们在 Spring Boot 应用程序中的对应组件类似，例如，以下是控制器代码：

```
@Controller(
    value = "/application-info",
    consumes = [MediaType.APPLICATION_JSON],
    produces = [MediaType.APPLICATION_JSON]
)
class ApplicationInfoController(
    private val applicationInfoService: ApplicationInfoService
) {

    @Get
    fun get(requestTo: String?): ApplicationInfo = applicationInfoService.get(requestTo)

    @Get("/logo", produces = [MediaType.IMAGE_PNG])
    fun getLogo(): ByteArray = applicationInfoService.getLogo()
}
```

Micronaut 中对 Kotlin 的支持建立在kapt (https://kotlinlang.org/docs/reference/kapt.html)编译器插件的基础上（参考Micronaut Kotlin 指南 (https://docs.micronaut.io/latest/guide/index.html#kotlin)了解更多详细信息）。

构建脚本配置如下：

```
plugins {
    ...
    kotlin("kapt")
    ...
}

dependencies {
    kapt("io.micronaut:micronaut-inject-java:$micronautVersion")
    ...
    kaptTest("io.micronaut:micronaut-inject-java:$micronautVersion")
    ...
}
```

以下是配置文件的内容：

```
micronaut:
  application:
    name: micronaut-service
  server:
    port: 8083

consul:
  client:
    registration:
      enabled: true

application-info:
  name: ${micronaut.application.name}
  framework:
    name: Micronaut
    release-year: 2018
```

JSON、properties和 Groovy 文件格式也可用于配置（参考Micronaut 配置指南(https://docs.micronaut.io/latest/guide/index.html#config)查看更多详细信息）。

Quarkus服务

Quarkus是作为一种应对新部署环境和应用程序架构等挑战的工具而引入的，在框架上编写的应用程序将具有低内存消耗和更快的启动时间。此外，对开发人员也很友好，例如，开箱即用的实时重新加载。

Quarkus 应用程序目前没有 main 方法，但也许未来会出现（GitHub 上的问题(https://github.com/quarkusio/quarkus/issues/284)）。

对于熟悉 Spring 或 Java EE 的人来说，Controller 看起来非常熟悉：

```
@Path("/application-info")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
class ApplicationInfoResource(
    @Inject private val applicationInfoService: ApplicationInfoService
) {

    @GET
    fun get(@QueryParam("request-to") requestTo: String?): Response =
        Response.ok(applicationInfoService.get(requestTo)).build()

    @GET
    @Path("/logo")
    @Produces("image/png")
    fun logo(): Response = Response.ok(applicationInfoService.getLogo()).build()
}
```

如你所见，bean 是通过@Inject注解注入的，对于注入的 bean，你可以指定一个范围，例如：

```
@ApplicationScoped
class ApplicationInfoService(
    ...
) {
...
}
```

为其他服务创建 REST 接口，就像使用 JAX-RS 和 MicroProfile 创建接口一样简单：

```
@ApplicationScoped
@Path("/")
interface ExternalServiceClient {
    @GET
    @Path("/application-info")
    @Produces("application/json")
    fun getApplicationInfo(): ApplicationInfo
}

@RegisterRestClient(baseUri = "http://helidon-service")
interface HelidonServiceClient : ExternalServiceClient

@RegisterRestClient(baseUri = "http://ktor-service")
interface KtorServiceClient : ExternalServiceClient

@RegisterRestClient(baseUri = "http://micronaut-service")
interface MicronautServiceClient : ExternalServiceClient

@RegisterRestClient(baseUri = "http://quarkus-service")
interface QuarkusServiceClient : ExternalServiceClient

@RegisterRestClient(baseUri = "http://spring-boot-service")
interface SpringBootServiceClient : ExternalServiceClient
```

但是它现在缺乏对服务发现 (Eureka (https://github.com/quarkusio/quarkus/issues/2052)和Consul (https://github.com/quarkusio/quarkus/issues/5812)) 的内置支持，因为该框架主要针对云环境。因此，在 Helidon 和 Ktor 服务中， 我使用了Java类库方式的Consul 客户端 (https://github.com/rickfast/consul-client)。

首先，需要注册应用程序：

```
@ApplicationScoped
class ConsulRegistrationBean(
    @Inject private val consulClient: ConsulClient
) {

    fun onStart(@Observes event: StartupEvent) {
        consulClient.register()
    }
}
```

然后需要将服务的名称解析到其特定位置；

解析是通过从 Consul 客户端获得的服务的位置替换 requestContext的URI 来实现的：

```
@Provider
@ApplicationScoped
class ConsulFilter(
    @Inject private val consulClient: ConsulClient
) : ClientRequestFilter {

    override fun filter(requestContext: ClientRequestContext) {
        val serviceName = requestContext.uri.host
        val serviceInstance = consulClient.getServiceInstance(serviceName)
        val newUri: URI = URIBuilder(URI.create(requestContext.uri.toString()))
            .setHost(serviceInstance.address)
            .setPort(serviceInstance.port)
            .build()

        requestContext.uri = newUri
    }
}
```

Quarkus也支持通过properties 或 YAML 文件进行配置（参考Quarkus 配置指南 (https://quarkus.io/guides/config) 了解更多详细信息）。

Spring Boot服务

创建该框架是为了使用 Spring Framework 生态系统，同时有利于简化应用程序的开发。这是通过auto-configuration实现的。

以下是控制器代码：

```
@RestController
@RequestMapping(path = ["application-info"], produces = [MediaType.APPLICATION_JSON_VALUE])
class ApplicationInfoController(
    private val applicationInfoService: ApplicationInfoService
) {

    @GetMapping
    fun get(@RequestParam("request-to") requestTo: String?): ApplicationInfo = applicationInfoService.get(re

    @GetMapping(path = ["/logo"], produces = [MediaType.IMAGE_PNG_VALUE])
    fun getLogo(): ByteArray = applicationInfoService.getLogo()
}
```

微服务由 YAML 文件配置：

```
spring:
  application:
    name: spring-boot-service

server:
  port: 8085

application-info:
  name: ${spring.application.name}
  framework:
    name: Spring Boot
    release-year: 2014
```

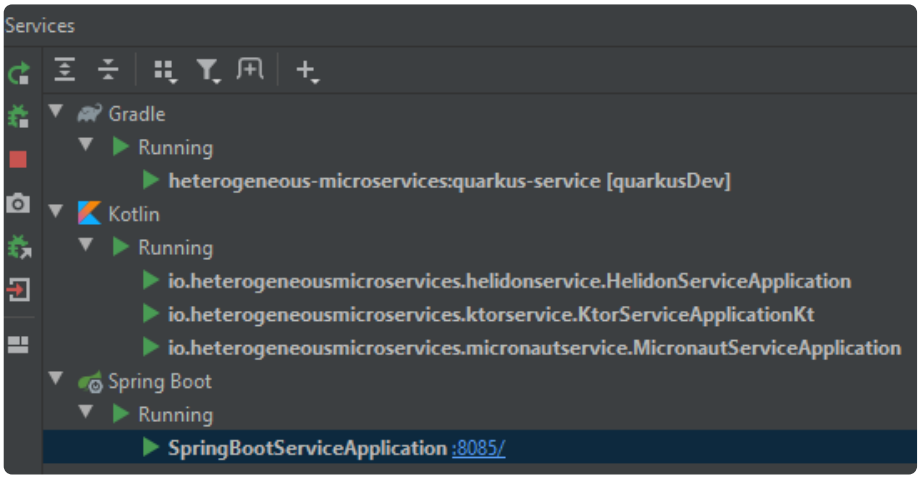
也可以使用properties文件进行配置（更多信息参考Spring Boot 配置文档 (https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html)）。

启动微服务

在启动微服务之前，你需要安装Consul (https://www.consul.io/docs/install/index.html)和 启动代理 (https://learn.hashicorp.com/consul/getting-started/agent)-例如，像这样：consul agent -dev。

你可以从以下位置启动微服务：

- IDE中启动微服务IntelliJ IDEA 的用户可能会看到如下内容：

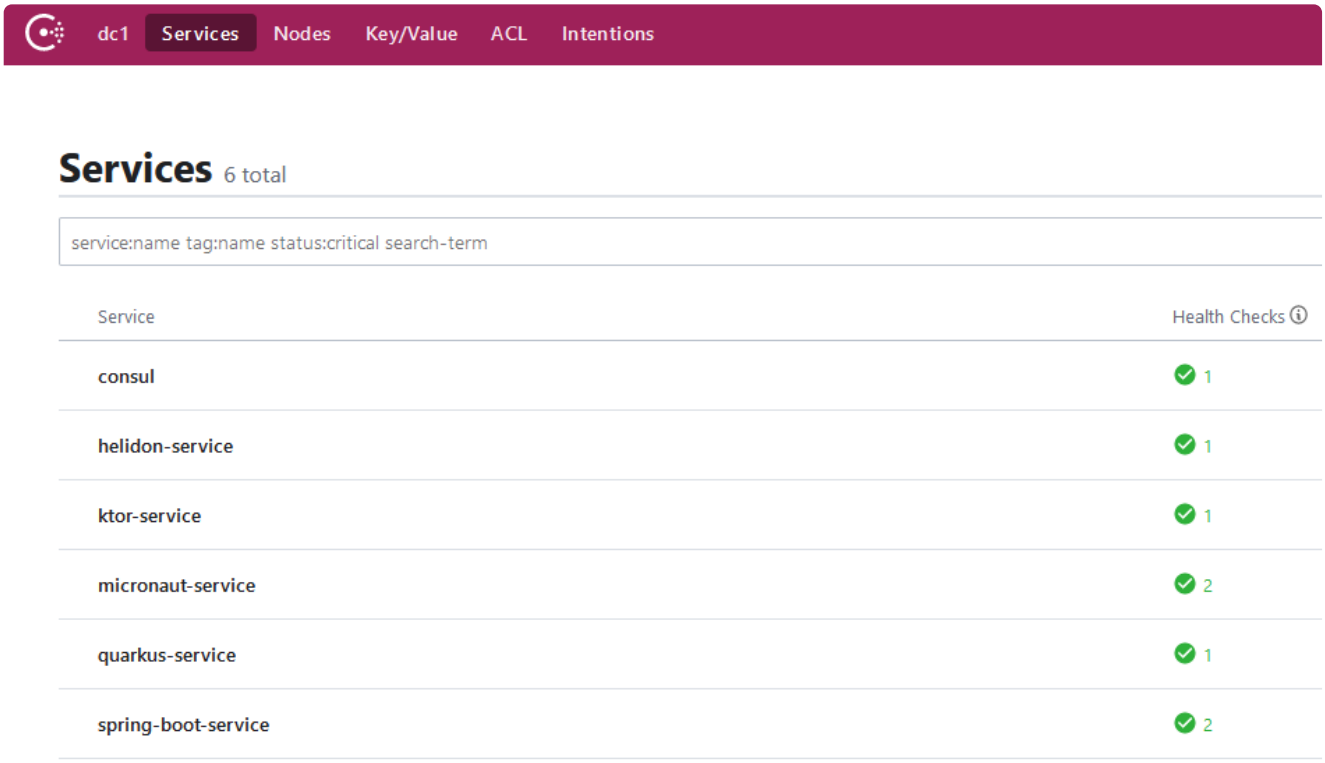


  要启动 Quarkus 服务，你需要启动quarkusDev的Gradle 任务。

- console中启动微服务在项目的根文件夹中执行：

```
java -jar helidon-service/build/libs/helidon-service-all.jar

java -jar ktor-service/build/libs/ktor-service-all.jar

java -jar micronaut-service/build/libs/micronaut-service-all.jar

java -jar quarkus-service/build/quarkus-service-1.0.0-runner.jar

java -jar spring-boot-service/build/libs/spring-boot-service.jar
```

启动所有微服务后，访问http://localhost:8500/ui/dc1/services，你将看到：



API测试

以Helidon服务的API测试结果为例：

- GET http://localhost:8081/application-info

```
{
  "name": "helidon-service",
  "framework": {
    "name": "Helidon SE",
    "releaseYear": 2019
  },
  "requestedService": null
}
```

- GET http://localhost:8081/application-info?request-to=ktor-service

```
{
  "name": "helidon-service",
  "framework": {
    "name": "Helidon SE",
    "releaseYear": 2019
  },
  "requestedService": {
    "name": "ktor-service",
    "framework": {
          "name": "Ktor",
          "releaseYear": 2018
    },
    "requestedService": null
  }
}
```

- GET http://localhost:8081/application-info/logo返回logo信息

你可以使用Postman (https://www.getpostman.com/) 、IntelliJ IDEA HTTP 客户端 (https://www.jetbrains.com/help/idea/http-client-in-product-code-editor.html) 、浏览器或其他工具测试微服务的 API接口 。

不同微服务框架对比

不同微服务框架的新版本发布后，下面的结果可能会有变化；你可以使用此GitHub项目 (https://github.com/rkudryashov/heterogeneous-microservices)自行检查最新的对比结果 。

程序大小

为了保证设置应用程序的简单性，构建脚本中没有排除传递依赖项，因此 Spring Boot 服务 uber-JAR 的大小大大超过了其他框架上的类似物的大小（因为使用 starters 不仅导入了必要的依赖项；如果需要，可以通过排除指定依赖来减小大小）：

> 备注：什么是 maven的uber-jar
>
> 在maven的一些文档中我们会发现 “uber-jar”这个术语，许多人看到后感到困惑。其实在很多编程语言中会把super叫做uber（因为super可能是关键字）， 这是上世纪80年代开始流行的，比如管superman叫uberman。所以uber-jar从字面上理解就是super-jar，这样的jar不但包含自己代码中的class ，也会包含一些第三方依赖的jar，也就是把自身的代码和其依赖的jar全打包在一个jar里面了，所以就很形象的称其为super-jar ， uber-jar来历就是这样的。

微服务	程序大小(MB)
Helidon服务	17,3
Ktor服务	22,4
Micronaut 服务	17,1
Quarkus服务	24,4
Spring Boot服务	45,2

启动时长

每个应用程序的启动时长都是不固定的：

微服务	开始时间(秒)
Helidon服务	2,0
Ktor服务	1,5
Micronaut 服务	2,8
Quarkus服务	1,9
Spring Boot服务	10,7

值得注意的是，如果你将 Spring Boot 中不必要的依赖排除，并注意设置应用的启动参数（例如，只扫描必要的包并使用 bean 的延迟初始化），那么你可以显著地减少启动时间。

内存使用情况

对于每个微服务，确定了以下内容：

- 通过-Xmx参数，指定微服务所需的堆内存大小
- 通过负载测试服务健康的请求(能够响应不同的请求)
- 通过负载测试50 个用户 * 1000 个的请求
- 通过负载测试500 个用户 * 1000 个的请求

堆内存只是为应用程序分配的总内存的一部分。例如，如果要测量总体内存使用情况，可以参考本指南(https://quarkus.io/guides/performance-measure)。

对于负载测试，使用了Gatling (https://gatling.io/)和Scala脚本 (https://github.com/rkudryashov/heterogeneous-microservices/blob/master/_misc/load_testing/load-test.scala) 。

> - 负载生成器和被测试的服务在同一台机器上运行（Windows 10、3.2 GHz 四核处理器、24 GB RAM、SSD）。
> - 服务的端口在 Scala 脚本中指定。
> - 通过负载测试意味着微服务已经响应了所有时间的所有请求。

微服务	堆内存大小（MB）	堆内存大小（MB）	堆内存大小（MB）
	对于健康服务	对于 50 * 1000 的负载	对于 500 * 1000 的负载
Helidon服务	11	9	11
Ktor服务	13	11	15
Micronaut 服务	17	15	19
Quarkus服务	13	17	21
Spring Boot服务	18	19	23

需要注意的是，所有微服务都使用 Netty HTTP 服务器。

结论

通过上文，我们所需的功能——一个带有 HTTP API 的简单服务和在 MSA 中运行的能力——在所有考虑的框架中都取得了成功。

是时候开始盘点并考虑他们的利弊了。

Helidon标准版

优点

> 创建的应用程序，只需要一个注释（@JvmStatic）

缺点

> 开发所需的一些组件缺少开箱即用（例如，依赖注入和与服务发现服务器的交互）

Helidon MicroProfile

微服务还没有在这个框架上实现，所以这里简单说明一下。

优点

> Eclipse MicroProfile 实现
>
> 本质上，MicroProfile 是针对 MSA 优化的 Java EE。因此，首先你可以访问各种 Java EE API，包括专门为 MSA 开发的 API，其次，你可以将 MicroProfile 的实现更改为任何其他实现（例如：Open Liberty、WildFly Swarm 等）

Ktor

优点

> - 轻量级的允许你仅添加执行任务直接需要的那些功能
> - 应用参数所有参数的良好结果

缺点

> - 依赖于Kotlin，即用其他语言开发可能是不可能的或不值得的
> - 微框架：参考Helidon SE (https://romankudryashov.com/blog/2020/01/heterogeneous-microservices/#_standard_edition)
> - 目前最流行的两种 Java 开发模型（Spring Boot/Micronaut）和 Java EE/MicroProfile）中没有包含该框架，这会导致：
>   - 难以寻找专家
>   - 由于需要显式配置所需的功能，因此与 Spring Boot 相比，执行任务的时间有所增加

Micronaut

优点

> - AOT如前所述，与 Spring Boot 上的模拟相比，AOT 可以减少应用程序的启动时间和内存消耗
> - 类Spring开发模式有 Spring 框架经验的程序员不会花太多时间来掌握这个框架
> - Micronaut for Spring (https://micronaut-projects.github.io/micronaut-spring/latest/guide/index.html)可以改变现有的Spring Boot应用程序的执行环境到Micronaut中（有限制）

Quarkus

优点

> - Eclipse MicroProfile 的实现
> - 该框架为多种 Spring 技术提供了兼容层：DI (https://quarkus.io/guides/spring-di)、 Web (https://quarkus.io/guides/spring-web)、Security (https://quarkus.io/guides/spring-security)、Data JPA (https://quarkus.io/guides/spring-data-jpa)

Spring Boot

优点

> - 平台成熟度和生态系统对于大多数日常任务，Spring的编程范式已经有了解决方案，也是很多程序员习惯的方式。此外，starter和auto-configuration的概念简化了开发
> - 专家多，文档详细

我想很多人都会同意 Spring 在不久的将来仍将是 Java/Kotlin开发领域领先的框架。

缺点

- 应用参数多且复杂但是，有些参数，如前所述，你可以自己优化。还有一个Spring Fu (https://github.com/spring-projects/spring-fu)项目的存在，该项目正在积极开发中，使用它可以减少参数。

> Helidon SE 和 Ktor 是 微框架 (https://en.wikipedia.org/wiki/Microframework)，Spring Boot 和 Micronaut 是全栈框架，Quarkus 和 Helidon MP 是 MicroProfile 框架。微框架的功能有限，这会减慢开发速度。

我不敢判断这个或那个框架会不会在近期“大更新”，所以在我看来，目前最好继续观察，使用熟悉的框架解决工作问题。

同时，如本文所示，新框架在应用程序参数设置方面赢得了 Spring Boot。如果这些参数中的任何一个对你的某个微服务至关重要，那么也许值得关注。但是，我们不要忘记，Spring Boot 一是在不断改进，二是它拥有庞大的生态系统，并且有相当多的 Java 程序员熟悉它。此外，还有未涉及的其他框架：Vert.x、Javalin 等，也值得关注。

参考链接: https://dzone.com/articles/not-only-spring-boot-a-review-of-alternatives (https://dzone.com/articles/not-only-spring-boot-a-review-of-alternatives)

关注微信公众号，加入社区

上一篇: 云原生时代，如何确保容器的全生命周期安全？ (https://www.kubernetes.org.cn/9492.html)

下一篇: 微服务的设计模式，你用了几个 (https://www.kubernetes.org.cn/9532.html)