微服务

- 微服务是系统架构上的一种设计风格， 它的主旨是将一个原本独立的系统拆分成多个小型服务，这些小型服务都在各自独立的进程中运行，服务之间通过基于HTTP的RESTful API进行通信协作

服务组件化

- 组件， 是一个可以独立更换和升级的单元。
- 单体系统部署在一个进程内， 往往修改了一个很小的功能， 为了部署上线会影响其他功能的运行
- 微服务将系统中的不同功能模块拆分成多个不同的服务， 这些服务都能够独立部署和扩展，可以有效避免一个服务的修改引起整个系统的重新部署

服务通信

- 微服务架构中， 由于服务不在一个进程中， 组件间的通信模式发生了改变， 若仅仅将原本在进程内的方法调用改成 RPC 方式的调用， 会导致微服务之间产生烦琐的通信， 使得系统表现更为糟糕， 所以， 需要更粗粒度的通信协议。
- 在微服务架构中， 通常会使用以下两种服务调用方式:
    - 使用HTTP的RESTfulAPI或轻量级的消息发送协议， 实现信息传递与服务调用的触发
    - 通过在轻量级消息总线上传递消息， 类似 RabbitMQ 等 一 些提供可靠异步交换的中间件。

去中心化管理数据

- 在实施微服务架构时， 都希望让每一个服务来管理其自有的数据库， 这就是数据管理的去中心化
    - 在去中心化过程中， 我们除了将原数据库中的存储内容拆分到新的同平台的其他数据 库实例中之外(如把原本存储 在 MySQL 中的表拆分后，存储到多个不同的 MySQL 实例中)， 也可以将一些具有特殊结构或业务特性的数据存储到一些其他技术的数据库实例中(如把日志信息存储到 MongoDB 中或把用户登录信息存储到 Redis 中)
    - 由于数据存储于不同的数据库实例中后， 数据一致性也成为微服务架构中亟待解决的问题之一
    - 分布式事务本身的实现难度就非常大， 所以在微服务架构中， 更强调在各服务之间进行 “ 无事务” 的调用， 而对于数据一致性， 只要求数据在最后的处理状态是一致的即可;若在过程中发现错误， 通过补偿机制来进行处理，使得错误数据能够达到最终的一致性。

基础设施自动化

- 在微服务架构中， 务必从一开始就构建起 “ 待续交付” 平台来支撑整个实施过程， 该平台需要两大内容， 缺一不可:
    - 自动化测试:每次部署前的强心剂， 尽可能地获得对正在运行的软件的信心
    - 自动化部署:解放烦琐枯燥的重复操作以及对多环境的配置管理

容错设计

- 在微服务架构中， 快速检测出故障源并尽可能地自动恢复服务是必须被设计和考虑的。 通常，都希望在每个服务中实现监控和日志记录的组件， 比如服务状态、 断路器状态、 吞吐量 、 网络延迟等关键数据的仪表盘等。

SpringCloud

- SpringCloud是一个基于SpringBoot实现的微服务架构开发 工具。 它为微服务架构中涉及的 配置管理、 服务治理、 断路器、 智能路由、 微代理、 控制总线、 全局锁、 决策竞选、分布式会话和集群状态管理等操作提供了 一 种简单的开发方式

子项目

- `SpringCloudConfig`： 配置管理工具， 支持使用Git存储 配置内容， 可以使用它实现应用配置的外部化存储， 并支持客户端配置信息刷新、 加密/解密配置内容 等
- `SpringCloudNetflix`： 核心 组件， 对多个Netflix OSS开源套件进行整合。
  - `Eureka`： 服务治理组件， 包含服务注册中心、 服务注册与发现机制的实现
  - `Hystrix`： 容错管理组件，实现断路器模式， 帮助服务依赖中出现的延迟和为故障提供强大的容错能力
  - `Ribbon`： 客户端负载均衡的服务调用组件
  - `Feign`： 基于Ribbon 和 Hystrix 的声明式服务调用组件
  - `Zuul`： 网关组件， 提供智能路由、 访问过滤等功能
  - Archaius: 外部化配置组件
- `Spring Cloud Bus`： 事件、 消息总线， 用于传播集群中的状态变化或事件， 以触发后续的处理， 比如用来动态刷新配置等
- `Spring Cloud Stream`： 通过 Redis、 Rabbit 或者 Kafka 实现的消费微服务， 可以通过简单的声明式模型来发送和接收消息
- `Spring Cloud Sleuth`： Spring Cloud 应用的分布式跟踪实现， 可以完美整合 Zipkin
- Spring Cloud Cluster: 针对 ZooKeeper、 Redis、 Hazelcast、 Consul 的选举算法和通用状态模式的实现
- Spring Cloud Consul: 服务发现与配置管理工具
- Spring Cloud Security: 安全工具包， 提供在 Zuul 代理中对 0Auth2 客户端请求的中继器
- SpringCloudZooKeeper：基于ZooKeeper的服务发现与配置管理组件。
- Spring Cloud Starters：Spring Cloud 的基础组件， 它是基于 Spring Boot 风格项目的 基础依赖模块

版本说明

- `1.5.8.RELEASE`、`java8`

```
<!--全栈Web开发模块， 包含嵌入式Tomcat、 Spring MVC-->
<dependency>
  <groupId>org.springframework.boot</groupId>
```

```
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<!--通用测试模块， 包含JUnit、 Hamcrest、 Mockito -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>

<!--版本要对应好  1.5.8 - Dalston.RELEASE -->
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Dalston.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

- 打包说明
  - 打包分为依赖jar包和可执行jar包，两者要分开，可执行包不可以被其他项目依赖

```
<!--配置 打可执行jar包和依赖jar包-->
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <classifier>exec</classifier>
      </configuration>
    </plugin>
    <plugin>
      <artifactId>maven-jar-plugin</artifactId>
      <executions>
        <execution>
          <id>exec</id>
          <phase>package</phase>
          <goals>
            <goal>jar</goal>
          </goals>
          <configuration>
            <classifier>exec</classifier>
          </configuration>
```

```
        </execution>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>jar</goal>
          </goals>
          <configuration>
            <!-- Need this to ensure application.yml is excluded -->
            <forceCreation>true</forceCreation>
            <excludes>
              <exclude>application.yml</exclude>
            </excludes>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>

<!--默认只打可执行jar包-->
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

SpringBoot

- 用于构建微服务的基础框架
  - 解决配置问题：
    - SpringBoot的宗旨并非要重写Spring或是替代Spring, 而是希望通过设计大量的自动化配置等方式来简化Spring原有样板化的配置，使得开发者可以快速构建应用。
  - 依赖管理工作变得简单：
    - 通过一系列Staiter POMs的定义， 让我们整合各项功能的时候，不需要在 Maven的pom.xml中维护那些错综复杂的依赖关系， 而是通过类似模块化的Starter模块定义来引用， 使得依赖管理工作变得更为简单
    - Starter POMs是一系列轻便的依赖包， 是一套一站式的Spring相关技术的解决方案。开发者在使用和整合模块时， 不必再去搜寻样例代码中的依赖配置来复制使用， 只需要引入对应的模块包即可
  - 易于部署：
    - Spring Boot除了可以很好融入Docker之外， 其自身就支持嵌入式的 Tomcat、 Jetty 等容器。

    - 所以， 通过Spring Boot 构建的应用不再需要安装 Tomcat, 将应用打包成war, 再部署到Tomcat这样复杂的构建与部署动作
    - 只需将Spring Boot应用打成jar包， 并通过java -jar命令直接运行就能启动一个标准化的Web应用，

      这使得Spring Boot应用变得非常轻便
- 自动化配置、 快速开发、 轻松部署等， 非常适合用作微服务架构中各项具体微服务的开发框架
- 可以帮助快速地构建微服务， 还可以轻松简单地整合 Spring Cloud 实现系统服务化
- 整个SpringBoot的生态系统都使用到了Groovy, 很自然的，我们完全可以通过使用Gradle和Groovy来开发Spring Boot应用
- 项目构建的 build部分， 引入了Spring Boot的Maven插件， 该插件非常实用，可以帮助我们方便地启停应用，这样在开发时就不用每次去找主类或是打包成jar来运行微服务，只需要通过mvn spring-boot:run 命令就可以快速启动SpringBoot应用

RESTful

启动

- 作为一个 Java 应用程序， 可以直接通过运行拥有 `main` 函数的类来启动
- 执行`mvn spring-boot: run` 命令，或是直接单击 IDE 中对 Maven 插件的工具
- 在服务器上部署运行时， 通常先使用 `mvn install` 将应用打包成`jar` 包， 再通过`java -jar xxx. jar` 来启动应用。

配置

- `Spring Boot` 的默认配置文件位置为`src/main/resources/application.properties`
- YAML文件
    - 是 一 个可读性高， 用来表达资料序列的格式
    - 以类似大纲的缩进形式来表示
    - YAML 目前还有一 些不足，它无法通过 @PropertySource 注解来加载配置。 但是， YAML 将属性加载到内存中保存的时候是有序的
- 参数

```
# application.properties
# 自定义参数
book.name=SpringCloudinAction
# 在应用中可以通过@Value 注解来加载这些自定义的参数
# PlaceHolder 方式， 格式为${...} 或 SpEL 表达式 (Spring Expression Language), 格式为#{...}
@Value("${book.name}")
private String name;  # getter/setter

# 参数引用
book.name=SpringCloud
book.desc=he is writing《${book.name}》
```

```
# 随机数
# 随机字符串
com.didispace.blog.value=${random.value}
# 随机int ${random.int}
# 10以内的随机数 ${random.int(10)}
# 10-20的随机数 ${random.int[10,20]}
```

- 命令行参数
  - `java -jar xxx.jar --server.port= 8888`
    - 直接以命令行的方式来设置server.port属性， 并将启动应用的端口设为8888
    - 命令行方式启动 SpringBoot 应用时， 连续的两个减号--就是对application.properties中的属性值进行赋值的标识
- 多环境配置
  - 多环境配置的文件名需要满足 application-{profile}. properties的格式， 其中{profile}对应的环境标识
  - 至于具体哪个配置文件会被加载， 需要在application.properties文件中通过spring.profiles.active 属性来设置， 其值对应配置文件中的{profile}值
  - 在application.properties中配置通用内容，并设置spring.profiles.active= dev, 以开发环境为默认配置
  - 在application-{profile}.properties中配置各个环境不同的内容
  - 通过命令行方式去激活不 同 环境的配置
    - `java -jar xxx.jar --spring.profiles.active =test`
- 属性加载顺序（高-低）
  - 命令行中传入的参数
  - SPRING APPLICATION JSON中的属性。 SPRING_APPLICATION—JSON是以JSON格式配置在系统 环境变量中的内容
  - java:comp/env中的JNDI 属性
  - Java 的系统属性，可以通过System.getProperties()获得的内容
  - 操作系统的环境变量
  - 通过random.*配置的随机属性
  - 位于当前应用 jar 包之外，针对不同{profile}环境的配置文件内容，例如application-{profile}.properties或是YAML定义的配置文件
  - 位于当前应用 jar 包之内，针对不同{profile}环境的配置文件内容，例如 application-{profile}.properties或是YAML定义的配置文件
  - 位于当前应用jar包之外的application.properties和YAML配置内容
  - 位于当前应用jar包之内的application.properties和YAML配置内容
  - 在@Configuration注解修改的类中，通过@PropertySource注解定义的属性
  - 应用默认属性，使用SpringApplication.setDefaultProperties 定义的内容

监控与管理

- 一套自动化的监控运维机制，而这套机制的运行基础就是不间断地收集各个微服务应用的各项指标情况，并根据这些基础指标信息来制定监控和预警规则，更进一步甚至做到一些自动化的运维操作等
- 一套专门用于植入各个微服务应用的接口供监控 系统采集信息。 而这些接口往往有很大一 部分指标都是类似的， 比如环境变量、垃圾收集信息、内存信息、、线程池信息等
- 一个标准化的实现框架`spring-boot-starter-actuator`，引入该模块能够自动为 Spring Boot 构建的应用提供一系列用于监控的端点

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

- `management.security.enabled=false`
- 该模块根据应用依赖和配置自动创建出来的监控和管理端点。通过这些端点，可以实时获取应用的各项监控指标
- 原生端点
  - 应用配置类：获取应用程序中加载的应用配置、 环境变量、 自动化配置报告等与Spring Boot应用密切相关的配置类信息
    - SpringBoot为了改善传统Spring应用繁杂的配置内容，采用了包扫描和自动化配置的机制来加载原本集中于XML文件中的各项内容
    - 虽然这样让代码变得简洁，但是整个应用的实例创建和依赖关系等信息都被离散到了各个配置类的注解上，这使分析整个应用中资源和实例的各种关系变得非常困难。
    - 而这类端点可以帮助我们轻松获取一系列关于Spring应用配置内容的详细报告， 比如自动化配置的报告、 Bean创建的报告、 环境属性的报告等
    - `/autoconfig`：用来获取应用的自动化配置报告， 其中包括所有自动化配置的候选项。 同时还列出了每个候选项是否满足自动化配置的各个先决条件。可以方便地找到一些自动化配置为什么没有生效的具体原因
    - `/beans`：获取应用上下文中创建的所有Bean
    - `/configprops:`：获取应用中配置的属性信息报告，可看到各个属性的配置路径。通过使用 endpoints.configprops.enabled=false 来关闭
    - `/env`：获取应用所有可用的环境属性报告。 包括环境变量、JVM属性、应用的配置属性、命令行中的参数。它可以帮助我们方便地看到当前应用可以加载的配置信息， 并配合@ConfigurationProper巨es注解将它们引入到我们的应用程序中来进行使用
    - `/mappings:`返回所有Spring MVC的控制器映射关系报
    - `/info`：返回一些应用自定义的信息。 默认清况下， 该瑞点只会返回一个空的JSON内容。可以在application.properties配置文件中通过info前缀来设置一 些属性
  - 度最指标类：获取应用程序运行过程中用于监控的度量指标， 比如内存信息、 线程池信息、HTTP请求统计等
    - 应用配置类端点所提供的信息报告在应用启动的时候就已经基本确定了其返回内容， 可以说是一 个静态报告
    - 度量指标类端点提供的报告内容则是动态变化的，这些端点提供了应用程序在运行过程中的一些快照信息，比如内存使用情况、 HTTP请求统计、外部资源指标等

        - `/metrics`：返回当前应用的各类重要度量指标，比如内存信息、线程信息、垃圾回收信息等。可以提供应用运行状态的完整度量指标报告。可以通过/metrics/{name}接口来更细粒度地获取度量信息， 比 如可 以 通 过访 问/metrics/mem.free来获取当前可用 内存数量
        - `/health`：获取应用的各类健康指标信息
            - 实现一 个用来采集健康信息的检测器
                - 实现Healthindicator接口，@Component
                - 重写health ()函数可实现健康检查
                - 重新启动应用， 并访问/health接口， 在返回的JSON字符串中，将会包含
        - `/dump`：暴露程序运行中的线程信息
        - `/trace`：返回基本的 HTTP 跟踪信息，始终保留最近的100条请求记录
    - 操作控制类:提供了对应用的关闭等操作类功能
        - `/shutdown`：关闭应用

单元测试

```
//src/test目录下
import static org.hamcrest.Matchers.equalTo;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootApplication
@WebAppConfiguration
public class HelloServiceApplicationTests {
  private MockMvc mvc;
  @Before
    public void setUp() {
        mvc = MockMvcBuilders.standaloneSetup(new HelloController()).build();
    }
  @Test
  public void contextLoads() throws Exception {

 mvc.perform(MockMvcRequestBuilders.get("/hello").accept(MediaType.APPLICATION_
JSON))
                .andExpect(status().isOk())
                .andExpect(content().string(equalTo("Hello, World!")));
  }
}
```