单体架构和 SOA

传统的单体架构，就是在一台服务器上，把 java web 程序打包成 war 包给 tomcat 去运行。一旦服务器要处理更多的并发，就采取水平扩展的方法，开出更多的服务器，用一个负载均衡器来重定向请求。

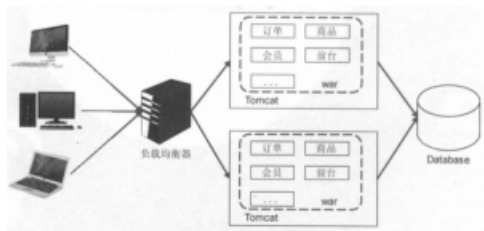


图1-2 传统单体应用架构

因为把所有模块都集成在 war 包中，这样使得更新和维护非常困难。同样的，在扩展的时候我们必须要扩展包含所有模块的 war 包，这样使得不需要扩展的模块也进行了扩展，占用了多余的资源。

针对于传统单体架构，有了 SOA（面向服务的架构）来解决问题。它的思路就是把应用中相同的功能聚合在一起，提供一个服务，基于 SOA 的架构可以理解为一批服务的组合。可以理解为一个 Tomcat 中运行了很多不同的 web 应用。

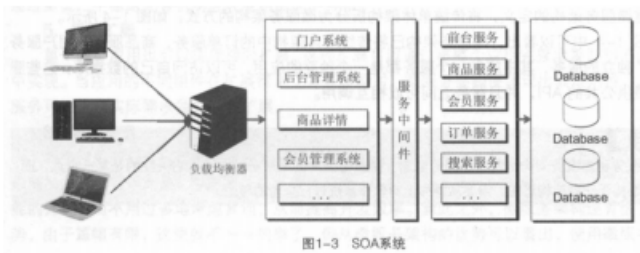


图1-3 SOA系统

由于 SOA 依旧在服务变多的时候，依旧非常复杂，所以就有了微服务架构。

微服务架构

微服务架构是一种架构风格和架构思想，它倡导我们在传统软件应用架构的基础上，将系统业务按照功能拆分为更加细粒度的服务，所拆分的每一个服务都是一个独立的应用，这些应用对外提供公共的 API，可以独立承担对外服务的职责，通过此种思想方式所开发的软件服务实体就是“微服务”，而围绕着微服务思想构建的一系列体系结构（包括开发、测试、部署等），我们可以将它称之为“微服务架构”。

微服务架构的组件

- 服务注册中心：注册系统中所有服务的地方。
- 服务注册：服务提供方将自己调用地址注册到服务注册中心，让服务调用方能够方便地找到自己。
- 服务发现：服务调用方从服务注册中心找到自己需要调用服务的地址。
- 负载均衡：服务提供方一般以多实例的形式提供服务，使用负载均衡能够让服务调用方连接到合适的服务节点。
- 服务容错：通过断路器（也称熔断器）等一系列的服务保护机制，保证服务调用者在调用异常服务时快速地返回结果，避免大量的同步等待。
- 服务网关：也称为 API 网关，是服务调用的唯一入口，可以在这个组件中实现用户鉴权、动态路由、灰度发布、负载限流等功能。
- 分布式配置中心：将本地化的配置信息（properties、yml、yaml 等）注册到配置中心，实现程序包在开发、测试、生产环境的无差别性，方便程序包的迁移。

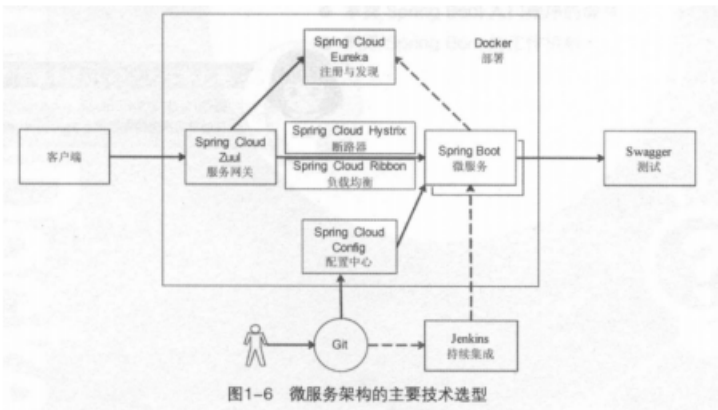


图1-6 微服务架构的主要技术选型

本书选型如上图所示，我们使用 Spring Boot 实现微服务实例的开发，使用 Spring Cloud Eureka 来实现服务的注册和发现，使用 Spring Cloud Hystrix 的断路器功能来实现服务容错，使用 Spring Cloud Ribbon 来实现服务间的负载均衡，使用 Spring Cloud Zuul 实现服务网关，使用 Spring Cloud Config 作为分布式配置中心，使用 Swagger 对微服务进行测试，并使用 Jenkins 的持续集成功能来实现自动化部署。

2.2 Spring Boot 入门

按照书上的写上 HelloController, HelloApplication, application.properties, HelloApplicationTests。然后 import 的红色部分全部靠 IDEA 的 add maven dependencies，最终需要修改一个包的依赖为：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.2.0.RELEASE</version>
</dependency>
```

就能在 8081 端口开出一个处理/hello 请求的 hello, world!

2.3

文件 2-1　HelloApplication.java

```
1  package com.itheima.web;
2  import org.springframework.boot.SpringApplication;
3  import org.springframework.boot.autoconfigure.SpringBootApplication;
4  @SpringBootApplication
5  public class HelloApplication {
6      public static void main(String[] args) {
7          SpringApplication.run(HelloApplication.class, args);
8      }
9  }
```

@SpringBootApplication 是 Spring Boot 的核心注解，它是一个复合注解，用于开启组件扫描和自动配置。在 main()方法中，通过调用 SpringApplication 类的 run()方法将业务委托给了 Spring Boot 的 SpringApplication 类，SpringApplication 类将引导应用程序启动 Spring，并且相应地启动被自动配置的 Tomcat 服务器。只需要将 HelloApplication.class 作为参数传递给 run()方法，以此来通知 SpringApplication 谁是主要的 Spring 组件，并传递给 args 数组作为参数。

@SpringBootApplication 可以分解成以下三个注解：

@SpringBootConfiguration	该注解和@Configuration 的作用相同，它表示其标注的类是 IoC 容器的配置类。
@EnableAutoConfiguration	用于将所有符合自动配置的 Bean 加载到当前 Spring Boot 创建并使用的 IoC 容器中。
@ComponentScan	用于自动扫描和加载符合条件的组件或 Bean，并将 Bean 加载到 IoC 容器中。

Spring IOC容器基本原理

2.2.1 IOC容器的概念

IOC容器就是具有依赖注入功能的容器，IOC容器负责实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。应用程序无需直接在代码中new相关的对象，应用程序由IOC容器进行组装。在Spring中BeanFactory是IOC容器的实际代表者。

Spring IOC容器如何知道哪些是它管理的对象呢？这就需要配置文件，Spring IOC容器通过读取配置文件中的配置元数据，通过元数据对应用中的各个对象进行实例化及装配。一般使用基于xml配置文件进行配置元数据，而且Spring与配置文件完全解耦的，可以使用其他任何可能的方式进行配置元数据，比如注解、基于java文件的、基于属性文件的配置都可以。

那Spring IOC容器管理的对象叫什么呢？

2.2.2 Bean的概念

由IOC容器管理的那些组成你应用程序的对象我们就叫它Bean，Bean就是由Spring容器初始化、装配及管理的对象，除此之外，bean就与应用程序中的其他对象没有什么区别了。那IOC怎样确定如何实例化Bean、管理Bean之间的依赖关系以及管理Bean呢？这就需要配置元数据，在Spring中由BeanDefinition代表，后边会详细介绍，配置元数据指定如何实例化Bean、如何组装Bean等。概念知道的差不多了，让我们来做个简单的例子。

在 Spring Boot 项目的 main()方法中，SpringApplication.run(HelloApplication.class, args)是唯一执行的方法体，该方法体的执行过程可分为两部分来看，具体如下。

1. 创建 SpringApplication 对象

在 SpringApplication 实例初始化时，它会做如下几项工作。

（1）根据 classpath 内是否存在某个特征类来判断是否为 Web 应用，并使用 webEnvironment 标记是否为 Web 应用。

（2）使用 SpringFactoriesLoader 在 classpath 中的 spring.factories 文件查找并加载所有可用的 ApplicationContextInitializer。

（3）使用 SpringFactoriesLoader 在 classpath 中的 spring.factories 文件查找并加载所有可用的 ApplicationListener。

（4）推断并设置 main()方法的定义类。

2. 调用实例的 run()方法

run()方法是 Spring Boot 执行流程的主要方法，该方法执行时，主要做了如下工作。

（1）查找并加载 spring.factories 中配置的 SpringApplicationRunListener，并调用它们的 started()方法。告诉 SpringApplicationRunListener，Spring Boot 应用要执行了。

（2）创建并配置当前 Spring Boot 应用要使用的 Environment，然后遍历调用所有 SpringApplicationRunListener 的 environmentPrepared()方法。告诉 SpringBoot 应用使用的环境准备好了。

（3）如果 SpringApplication 的 showBanner 属性被设置为 true，则打印 banner。

（4）根据用户是否明确设置了 applicationContextClass 类型，以及初始化阶段（创建 SpringApplication 对象的第（1）步）的推断结果来决定当前 Spring Boot 应用创建什么类型的 ApplicationContext。

（5）创建故障分析器。故障分析器用于提供错误和诊断信息。

（6）对 ApplicationContext 进行后置处理。对所有可用的 ApplicationContextInitializer 遍历执行 initialize()方法；遍历调用所有 SpringApplicationRunListener 的 environmentPrepared()方法；将之前通过@EnableAutoConfiguration 获取的所有配置以及其他形式的 IoC 容器配置加载到已经准备完毕的 ApplicationContext；遍历调用所有 SpringApplicationRunListeners 的 contextLoaded()方法。

（7）调用 refreshContext()方法执行 applicationContext 的 refresh()方法。

（8）查找当前 ApplicationContext 中是否有 ApplicationRunner 和 CommandLineRunner，如果有，则遍历执行它们。

（9）正常情况下，会遍历执行所有 SpringApplicationRunListener 的 finished()方法；如果出现异常，也会调用该方法，只不过这种情况下会将异常信息一并传入处理。

经过上述步骤后，一个完整的 Spring Boot 项目就已经启动完成。

图2-7 Spring Boot的整个启动流程示意图

3.1 Spring Boot 和 Mybatis 的集成

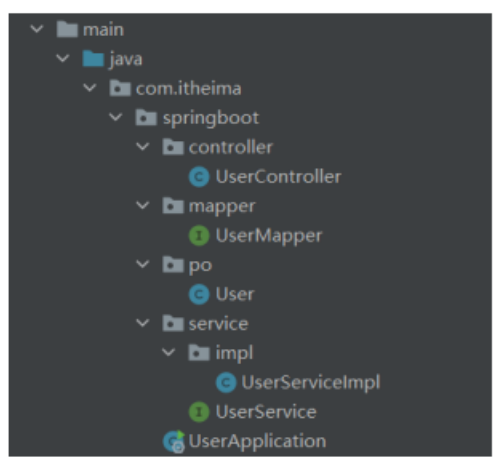
```
<!--          MyBatis启动器 -->
<!--        mysql-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.46</version>
</dependency>
<!--druid连接池-->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.0.29</version>
</dependency>
<!--mybatis-->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.2</version>
</dependency>
```

引入对应的依赖后运行出现报错

Failed to introspect Class [org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration]

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.1.3.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-autoconfigure</artifactId>
    <version>2.1.3.RELEASE</version>
</dependency>
```

把上述两个依赖修改为 2.1.3.RELEASE 后，运行正常。结构如下图所示：



在实现书籍提供的前端时，遇到了跨域问题，添加@CrossOrigin(max=3600) 前端在访问 URL 前添加 http 解决。

多学一招：使用 YAML 配置外部属性

YAML 是 JSON 的一个超集，可以非常方便地将外部配置以层次结构形式存储起来。当项目的类路径中有 SnakeYAML 库（spring-boot-starter 中已经被包含）时，SpringApplication 类将自动支持 YAML 作为 properties 的替代。

如果将项目中的 application.properties 文件修改为 YAML 文件（后缀为.yml 或 yaml）的形式，则其配置信息如文件 3-8 所示。

文件 3-8 application.yml

```
#DB Configuration
spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/microservice
    username: root
    password: root
#logging
logging:
  level:
    com.itheima.springboot: debug
```

从上述配置文件中可以看出，yml 文件是一个树状结构的配置，它与 properties 文件相比，有很大的不同，在编写时需要注意以下几点。

（1）在 properties 文件中是以“.”进行分割的，在 yml 中是用“:”进行分割的。

（2）yml 的数据格式和 json 的格式很像，都是 K-V 格式，并且通过“:”进行赋值。

（3）每个 k 的冒号后面一定都要加一个空格，例如 driver-class-name 后面的“:”之后，需要有一个空格，否则文件会报错。

由于在 Spring Boot 官方文档中，主要使用的是 properties 文件，而 Spring Cloud 官网文档以及一些开源的项目中，大多数使用的是 yml 文件，所以本书在 Spring Boot 部分将使用 properties 文件，而在后面的 Spring Cloud 部分将使用 yml 文件。

Spring Boot 和 Redis 集成

Redis 的安装：https://www.runoob.com/redis/redis-install.html

Redis 配置环境变量：https://blog.csdn.net/qq_42773229/article/details/88745259

配置完成后可以如上图所示启动。

Redis 是一个完全开源免费的、遵守 BSD 协议的、内存中的数据结构存储，它既可以作为数据库，也可以作为缓存和消息代理。因其性能优异等优势，目前已被很多企业所使用，但通常在企业中我们会将其作为缓存来使用。Spring Boot 对 Redis 也提供了自动配置的支持，接下来本小节将讲解如何在 Spring Boot 项目中使用 Redis。

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
    <version>2.1.3.RELEASE</version>
</dependency>
```

添加对应 redis 依赖即可。
具体版本应当选取兼容的，可以参考
https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-data-redis

Redis和Mysql的区别

1. mysql是关系型数据库，redis是NOSQL，非关系型数据库。mysql将数据持久化到硬盘，读取数据慢，而redis数据先存储在缓存中，读取速度快
2. mysql作为持久化数据库，频繁访问数据库会在反复连接数据库上花费大量时间。redis则会在缓存区存储大量频繁访问的数据，即先访问缓存。

2. 添加缓存注解

（1）在引导类 Application.java 中，添加@EnableCaching 注解开启缓存，添加后的代码如下所示：

```
@SpringBootApplication
@EnableCaching  //开启缓存
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

（2）在业务逻辑类 UserServiceImpl 的 getAllUsers()方法上添加@Cacheable 注解来支持缓存，添加后的实现代码如下：

```
// 查询所有用户
@Cacheable(value="UserCache",key="'user.getAllUsers'")
public List<User> getAllUsers() {
    return this.userMapper.getAllUsers();
}
```

需要注意的是，@Cacheable 注解中的 key 属性值除了需要被英文双引号引用外，还需要加入英文单引号，否则系统在执行缓存操作时将出错。

3. 使实体类实现可序列化接口

为了便于数据的传输，需要将实体类 User 实现序列化接口 Serializable，具体代码如下：

```
import java.io.Serializable;
public class User implements Serializable{
    private static final long serialVersionUID = 1L;
    private Integer id;
    private String username;
    private String address;
    ...
}
```

为了实现缓存，我们需要执行以上步骤。