第一篇：先导篇——开启Spring之旅，初识Spring Boot

目标： 了解Spring的设计目标及核心功能，通过Spring Boot初步体验Spring的高效开发模式。

第一章：Spring 简介

- Spring 是什么
- Spring设计的主要目的和适用场景
- Spring的核心功能简介

第二章：Spring Boot 初体验

- 搭建第一个Spring Boot项目
- 构建一个简单的RESTful API
- 打包与部署

第二篇：实战篇——掌握Spring在Web应用中的应用

目标： 通过Spring的核心功能，掌握Web应用开发的核心技能。

第三章：Spring 核心容器（Core Container）

- 依赖注入（DI）与控制反转（IoC）
- Bean管理、生命周期与作用域
- ApplicationContext、BeanFactory的区别与应用

第四章：Spring AOP（面向切面编程）

- AOP概念及Spring中的应用
- 使用AOP实现事务管理、日志记录等
- 动态代理的实现原理

第五章：Spring MVC（Model-View-Controller）

- Spring MVC架构与核心组件
- 路由映射与请求处理
- 表单处理与数据绑定
- 异常处理与错误页面定制

第六章：Spring Boot

- Spring Boot自动配置原理

- 延迟加载、懒加载
- 缓存机制（Redis、EhCache等）

说明：前两篇是初级人员必学内容。

    - 与前端（Vue、React）结合，实现前后端分离。
2. 企业级应用（Spring Boot / Spring Framework）
    - 适用于构建中大型企业级应用，如 ERP、CRM、OA 系统。
3. 微服务架构（Spring Cloud）
    - 适用于构建分布式微服务架构，如服务注册、配置中心、网关等。
4. 数据访问层开发（Spring Data）
    - 适用于与数据库交互，支持 JPA、Hibernate、MyBatis、MongoDB、Redis。
5. 安全管理（Spring Security）
    - 适用于身份认证、权限控制、OAuth2 授权等。
6. 定时任务与批处理（Spring Batch）
    - 适用于大数据处理、日志分析、ETL 任务。

1.3 Spring 的核心功能简介

Spring 具有以下核心模块：

模块	主要功能
Core Container	提供 IoC（控制反转）和 DI（依赖注入）容器，用于管理 Bean 生命周期。
Spring AOP	提供面向切面编程（AOP）支持，实现事务管理、日志等功能。
Spring MVC	提供基于 MVC 设计模式的 Web 框架，支持 RESTful API。
Spring Boot	提供开箱即用的 Spring 生态，简化配置，提高开发效率。
Spring Data	统一数据访问层，支持 JDBC、JPA、Hibernate、MongoDB、Redis。
Spring Security	提供身份认证、授权、安全防护，如 OAuth2、JWT 认证。
Spring Cloud	支持微服务架构，如服务发现（Eureka）、负载均衡（Ribbon）、网关Gateway）。
Spring WebFlux	提供响应式编程支持，适用于高并发非阻塞应用。
Spring Batch	适用于批处理任务，如大规模数据导入、报表生成等。

1.4.1 完整的后端服务解决方案 Spring 提供了一个全面的后端开发平台，涵盖了服务开发的各个层次，适用于不同的业务需求：

- 核心能力：
  - 依赖注入（IoC）：解耦应用组件，提升模块化和可测试性。
  - 数据访问（Spring Data）：通过统一接口，简化与数据库的交互，包括 JPA、Hibernate、JDBC 等技术。
  - 安全管理（Spring Security）：实现认证、授权、权限控制、CSRF 防护等安全功能。
  - 分布式架构（Spring Cloud）：提供微服务架构所需的组件，如服务注册与发现、分布式配置、负载均衡等。

这些核心能力使 Spring 成为开发复杂企业级应用的理想选择，从单体应用到微服务架构，都能轻松应对。

1.4.2 前端支持有限

Spring 本身并不提供独立的前端框架，但它依然可以支持前端开发：

- 模板引擎：通过 Thymeleaf 等模板引擎，Spring 提供了简单的服务端渲染能力。
- RESTful API：Spring 的 Web 框架（如 Spring MVC 和 Spring WebFlux）可以通过 RESTful API 与前端框架（如 React 或 Vue.js）进行无缝集成，支持前后端分离的开发模式。

1.4.3 模块化设计与丰富组件

Spring 是一个模块化的框架，采用松耦合的设计理念，将服务开发的各个方面分解为不同的组件，开发人员可以根据项目需求选择合适的模块进行组合。这种设计使得 Spring 非常灵活，可以适应各种开发需求，甚至可以与其他框架进行集成，如 Hibernate、MyBatis、Kafka、Redis 等。

总结

本章回顾

- Spring 是什么：Spring 是一个开源的 Java EE 开发框架，提供 IoC、AOP、MVC 等功能。
- Spring 的设计目的：简化 Java 开发，提高灵活性和扩展性，适用于 Web 开发、微服务、数据访问等场景。
- Spring 的核心功能：包括 Core、AOP、MVC、Boot、Data、Security、Cloud、WebFlux、Batch 等模块。
- Spring 的定位：Spring 是一个为服务开发量身定制的“瑞士军刀”，它通过模块化设计和丰富的功能组件，涵盖了服务端开发的方方面面，提供了完整的后端解决方案，并支持与前端框架的协作。

下一步

- 搭建第一个 Spring Boot 项目，体验 Spring 开发的便捷性！（第二章）

附录：Spring 生态系统架构图

第二章：Spring Boot 初体验

1. 搭建第一个 Spring Boot 项目

Spring Boot 的核心目标是简化 Spring 应用的初始搭建和开发过程。以下是搭建第一个 Spring Boot 项目的步骤：

- 使用 Spring Initializr 创建项目
  - 访问 Spring Initializr。
  - 选择项目类型（Maven 或 Gradle）。
  - 选择 Spring Boot 版本（推荐使用最新稳定版）。
  - 添加依赖（如 Spring Web、Spring Boot DevTools 等）。
  - 生成项目并下载到本地。
- 项目结构
  - src/main/java：Java 源代码目录。
  - src/main/resources：配置文件、静态资源等。
  - src/test/java：测试代码目录。
  - pom.xml 或 build.gradle：项目依赖管理文件。
- 运行项目
  - 使用 IDE（如 IntelliJ IDEA 或 Eclipse）导入项目。
  - 运行 Application 类中的 main 方法，启动 Spring Boot 应用。

2. 构建一个简单的 RESTful API

Spring Boot 提供了快速构建 RESTful API 的能力。以下是一个简单的示例：

- 创建 Controller

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api")
public class HelloController {

    @GetMapping("/hello")
    public String sayHello() {
        return "Hello, Spring Boot!";
    }
}
```

- 使用 Maven 命令：mvn clean package。
- 打包后会在 target 目录下生成一个可执行的 JAR 文件。
- 运行 JAR 文件：java -jar target/your-project-name.jar。

- 打包为 WAR 文件
  - 修改 pom.xml，将打包方式改为 war：

```
<packaging>war</packaging>
```

  - 修改 Application 类，继承 SpringBootServletInitializer：

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application)
{
        return application.sources(Application.class);
    }

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

  - 使用 Maven 命令打包：mvn clean package。
  - 将生成的 WAR 文件部署到 Tomcat 或其他 Servlet 容器中。

补充：常用注解速查表

第三章：Spring 核心容器（Core Container）

1. 依赖注入（DI）与控制反转（IoC）

- 控制反转（IoC）
  - IoC 是一种设计原则，将对象的创建和依赖关系的管理交给容器，而不是由开发者手动管理。
  - Spring 通过 IoC 容器（如 ApplicationContext）实现这一原则。
- 依赖注入（DI）
  - DI 是 IoC 的一种实现方式，容器负责将依赖关系注入到对象中。
  - 常见的注入方式：
    - 构造器注入：通过构造函数注入依赖。
    - Setter 注入：通过 Setter 方法注入依赖。
    - 字段注入：通过 @Autowired 注解直接注入字段（不推荐，因为不利于测试）。
- 代码示例

```
// 定义一个服务类
@Service
public class UserService {
    private final UserRepository userRepository;

    // 构造器注入
    @Autowired
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public void printUser() {
        System.out.println("User: " + userRepository.getUser());
    }
}

// 定义一个仓库类
@Repository
public class UserRepository {
    public String getUser() {
        return "John Doe";
    }
}
```

```java
@Component
@Scope("prototype") // 指定作用域为 Prototype
public class PrototypeBean {
    private static int count = 0;

    public PrototypeBean() {
        count++;
        System.out.println("PrototypeBean created: " + count);
    }
}

@Component
public class SingletonBean {
    private static int count = 0;

    public SingletonBean() {
        count++;
        System.out.println("SingletonBean created: " + count);
    }
}
```

3. ApplicationContext 与 BeanFactory 的区别与应用

- BeanFactory
  - 是 Spring 最基础的 IoC 容器，提供基本的依赖注入功能。
  - 懒加载（Lazy Loading）：Bean 在第一次使用时才会被创建。
- ApplicationContext
  - 是 BeanFactory 的扩展，提供了更多企业级功能（如国际化、事件发布、AOP 等）。
  - 立即加载（Eager Loading）：容器启动时就会创建所有单例 Bean。
- 常用实现类
  - AnnotationConfigApplicationContext：基于注解配置的 ApplicationContext。
  - ClassPathXmlApplicationContext：基于 XML 配置的 ApplicationContext。
- 代码示例


```java
// UserService.java
```

```
    @PreDestroy
    public void destroy() {
        System.out.println("PrototypeBean destroyed");
    }
}

// SingletonBean.java
@Component
public class SingletonBean {
    private static int count = 0;

    public SingletonBean() {
        count++;
        System.out.println("SingletonBean created: " + count);
    }

    @PostConstruct
    public void init() {
        System.out.println("SingletonBean initialized");
    }

    @PreDestroy
    public void destroy() {
        System.out.println("SingletonBean destroyed");
    }
}

// MainApp.java
public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);

        // 测试依赖注入
        UserService userService = context.getBean(UserService.class);
        userService.printUser();

        // 测试 Bean 作用域
        PrototypeBean prototypeBean1 = context.getBean(PrototypeBean.class);
        PrototypeBean prototypeBean2 = context.getBean(PrototypeBean.class);
```



```
        SingletonBean singletonBean1 = context.getBean(SingletonBean.class);
        SingletonBean singletonBean2 = context.getBean(SingletonBean.class);

        // 关闭容器，观察 Bean 销毁
        ((AnnotationConfigApplicationContext) context).close();
    }
}
```

练习总结

通过完成这个练习，你将：

1. 掌握 Spring 的依赖注入和控制反转机制。
2. 理解 Bean 的生命周期和作用域。
3. 学会使用 ApplicationContext 管理 Bean。

```
// 定义一个切面类
@Aspect
@Component
public class LoggingAspect {

    // 前置通知
    @Before("execution(* com.example.service.*.*(..))")
    public void beforeAdvice(JoinPoint joinPoint) {
        System.out.println("Before method: " + joinPoint.getSignature().getName());
    }

    // 返回通知
    @AfterReturning(pointcut = "execution(* com.example.service.*.*(..))", returning = "result")
    public void afterReturningAdvice(JoinPoint joinPoint, Object result) {
        System.out.println("After returning from method: " + joinPoint.getSignature().getName());
        System.out.println("Result: " + result);
    }

    // 异常通知
    @AfterThrowing(pointcut = "execution(* com.example.service.*.*(..))", throwing = "ex")
    public void afterThrowingAdvice(JoinPoint joinPoint, Exception ex) {
        System.out.println("After throwing from method: " + joinPoint.getSignature().getName());
        System.out.println("Exception: " + ex.getMessage());
    }

    // 环绕通知
    @Around("execution(* com.example.service.*.*(..))")
    public Object aroundAdvice(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
        System.out.println("Before proceeding method: " + proceedingJoinPoint.getSignature().getName());
        Object result = proceedingJoinPoint.proceed();
        System.out.println("After proceeding method: " + proceedingJoinPoint.getSignature().getName());
        return result;
    }
```


- getArgs(): 返回方法参数的数组。

1. 创建项目
   - 使用 Spring Initializr 创建一个新的 Spring Boot 项目，添加 Spring Web 和 Spring AOP 依赖。
2. 实现日志记录
   - 创建一个切面类 LoggingAspect，使用前置通知和后置通知记录方法的调用和返回结果。
3. 实现性能监控
   - 在 LoggingAspect 中添加环绕通知，计算方法的执行时间并输出。
4. 测试 AOP 功能
   - 创建一个 UserService 类，定义几个方法，并测试日志记录和性能监控功能。

参考代码


```
@Aspect
@Component
public class LoggingAspect {

    // 前置通知
    @Before("execution(* com.example.service.*.*(..))")
    public void beforeAdvice(JoinPoint joinPoint) {
        System.out.println("Before method: " + joinPoint.getSignature().getName());
    }

    // 后置通知
    @After("execution(* com.example.service.*.*(..))")
    public void afterAdvice(JoinPoint joinPoint) {
        System.out.println("After method: " + joinPoint.getSignature().getName());
    }

    // 环绕通知（性能监控）
    @Around("execution(* com.example.service.*.*(..))")
    public Object aroundAdvice(ProceedingJoinPoint proceedingJoinPoint) throws
Throwable {
        long startTime = System.currentTimeMillis();
        Object result = proceedingJoinPoint.proceed();
        long endTime = System.currentTimeMillis();
        System.out.println("Method " + proceedingJoinPoint.getSignature().getName() +
" executed in " + (endTime - startTime) + "ms");
        return result;
    }
}

// UserService.java
@Service
public class UserService {
    public void printUser() {
        System.out.println("User: John Doe");
    }

    public void simulateLongTask() throws InterruptedException {
        Thread.sleep(2000);
        System.out.println("Long task completed");
```

第五章：Spring MVC（Model-View-Controller）

1. 什么是 MVC?

- MVC 的定义
  - MVC（Model-View-Controller）是一种软件设计模式，用于将应用程序的逻辑分为三个核心组件：
    - Model（模型）：负责管理应用程序的数据和业务逻辑。
    - View（视图）：负责数据的展示和用户界面。
    - Controller（控制器）：负责接收用户输入，协调 Model 和 View 之间的交互。
- MVC 的组成
  - Model：
    - 封装应用程序的核心数据。
    - 提供数据的访问和修改方法。
    - 不直接与用户交互。
  - View：
    - 负责将数据渲染为用户可见的界面（如 HTML 页面）。
    - 从 Model 中获取数据，但不直接修改数据。
  - Controller：
    - 接收用户输入（如 HTTP 请求）。
    - 调用 Model 处理业务逻辑。
    - 选择适当的 View 来渲染数据。

---

2. MVC 解决了什么问题?

- 问题背景
  - 在传统的 Web 应用程序中，业务逻辑、数据访问和用户界面代码通常混杂在一起，导致代码难以维护和扩展。
  - 随着应用程序规模的增长，这种紧耦合的设计会导致代码重复、可读性差、测试困难等问题。
- MVC 的解决方案
  - 分离关注点：将应用程序分为 Model、View 和 Controller，每个部分只关注自己的职责。
  - 提高可维护性：通过清晰的职责划分，使代码更易于理解和修改。
  - 增强可扩展性：可以独立修改 Model、View 或 Controller，而不会影响其他部分。
  - 便于测试：Model 和 Controller 可以单独测试，而不需要依赖用户界面。

---

3. 一个 MVC 框架应该具备哪些功能?

- 路由映射
  - 将请求 URL 映射到对应的控制器方法。

  c. Controller 处理请求并返回 ModelAndView。
  d. DispatcherServlet 通过 ViewResolver 解析视图并渲染。

---

5. 路由映射与请求处理

- 路由映射
  - 使用 @RequestMapping 或 @GetMapping、@PostMapping 等注解将请求 URL 映射到控制器方法。
- 请求处理
  - 控制器方法可以接收请求参数、路径变量、请求体等数据，并返回视图名称或直接返回数据（如 JSON）。
- 代码示例

```
@Controller
@RequestMapping("/user")
public class UserController {

    // 处理 GET 请求，URL 为 /user/profile
    @GetMapping("/profile")
    public String profile(Model model) {
        model.addAttribute("name", "John Doe");
        return "profile"; // 返回视图名称
    }

    // 处理 POST 请求，URL 为 /user/submit
    @PostMapping("/submit")
    @ResponseBody
    public String submit(@RequestParam String name) {
        return "Hello, " + name;
    }
}
```

---

6. 表单处理与数据绑定

- 表单处理
  - 使用 @ModelAttribute 注解将表单数据绑定到 Java 对象。
  - 使用 th:object（Thymeleaf）或 form:form（JSP）标签渲染表单。

- 在 src/main/resources/templates 目录下创建自定义错误页面（如 error.html）。

- 代码示例

```
@ControllerAdvice
pul lic class GlobalExceptionHandler {

    // 处理特定异常
    @ExceptionHandler(RuntimeException.class)
    public String handleRuntimeException(RuntimeException  ex, Model model) {
        model.addAttribute("error", ex.getMessage());
        return "error";
    }
}
```

课后练习：Spring MVC 实战

练习目标

通过编写代码，深入理解 Spring MVC 的使用场景，并掌握如何实现路由映射、表单处理、异常处理等功能。

练习步骤

1. 创建项目
   - 使用 Spring Initializr 创建一个新的 Spring Boot 项目，添加 Spring Web 和 Thymeleaf 依赖。
2. 实现路由映射
   - 创建一个 UserController，定义多个路由映射方法，处理 GET 和 POST 请求。
3. 实现表单处理
   - 创建一个表单页面（使用 Thymeleaf），提交用户信息并显示提交结果。
4. 实现异常处理
   - 在 UserController 中抛出一个自定义异常，并使用 @ControllerAdvice 全局处理该异常。
5. 测试功能
   - 启动应用，访问不同路由，测试表单提交和异常处理功能。

参考代码


```
@Controller
@RequestMapping("/user")
public class UserController {

    @GetMapping("/profile")
    public String profile(Model model) {
        model.addAttribute("name", "John Doe");
        return "profile";
    }

    @GetMapping("/form")
    public String showForm(Model model) {
        model.addAttribute("user", new User());
        return "form";
    }

    @PostMapping("/submit")
    public String submitForm(@ModelAttribute User user, Model model) {
        if (user.getName().isEmpty()) {
            throw new RuntimeException("Name cannot be empty!");
        }
        model.addAttribute("message", "Form submitted successfully!");
        return "result";
    }
}

// User.java
public class User {
    private String name;
    private String email;

    // Getters and Setters
}

// GlobalExceptionHandler.java
@ControllerAdvice
public class GlobalExceptionHandler {

```

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Error</title>
</head>
<body>
    <h1>Error</h1>
    <p th:text="${error}"></p>
</body>
</html>
```

练习总结

通过完成这个练习，你将：

1. 掌握 Spring MVC 的基本架构和核心组件。
2. 学会实现路由映射、表单处理和异常处理。
3. 理解如何使用 Thymeleaf 渲染视图。

示例代码：一个简单的 Spring Boot 应用

```
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}

@RestController
class HelloWorldController {
    @GetMapping("/")
    public String hello() {
        return "Hello, Spring Boot!";
    }
}

```

运行应用：

- 运行 DemoApplication 类，访问 http://localhost:8080/，你会看到返回内容 Hello, Spring Boot!。

6.3 构建一个简单的 RESTful API

创建 REST 控制器：

6.5 Spring Boot 常用注解与应用

常用注解：

1. @SpringBootApplication：
   - 该注解是 Spring Boot 应用的入口点，包含了 @Configuration、@EnableAutoConfiguration 和 @ComponentScan，用于启用配置、自动配置和组件扫描。
2. @RestController 和 @RequestMapping：
   - @RestController 用于定义一个控制器，处理 HTTP 请求。
   - @RequestMapping 用于定义请求路径和方法类型（如 GET、POST）。

```
@RestController
@RequestMapping("/api")
public class UserController {
    @GetMapping("/hello")
    public String hello() {
        return "Hello, Spring Boot!";
    }
}

```

1. @Value：
   - 用于从配置文件中注入值。

```
@Value("${app.name}")
private String appName;

```

1. @ConfigurationProperties：
   - 将配置文件的属性绑定到一个 Java Bean 上，简化配置文件的使用。


```
<!DOCTYPE html>
<html>
<head>
    <title>Spring Boot Example</title>
</head>
<body>
    <h1>Welcome to Spring Boot!</h1>
</body>
</html>

```

6.7 Spring Boot 应用打包与部署

打包应用：Spring Boot 提供了 spring-boot-maven-plugin 来打包应用，将项目打包成一个可执行 JAR 文件。

在 pom.xml 中配置插件：

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>

```

执行 mvn clean package 打包项目。

部署与运行：

- 通过 java -jar target/demo-0.0.1-SNAPSHOT.jar 启动应用。

6.8 Spring Boot 的优势

Spring Boot 的主要优势：

1. 简化配置：

第七章：数据访问和集成

目标

- 理解并掌握 Spring Boot 提供的各种数据访问方式
- 学习如何通过 Spring Data JPA 操作数据库
- 掌握如何与外部系统进行集成，使用 REST、JMS 和其他协议

7.1 数据访问概述

在现代应用中，数据访问是核心任务之一。Spring Boot 提供了多种与数据库、消息队列、Web 服务等进行集成的方式。常见的数据库访问方式有：

- JDBC：通过 JDBC 与关系型数据库进行连接，执行 SQL 语句。
- JPA (Java Persistence API)：使用 JPA 进行 ORM（对象关系映射），简化数据库操作。
- Spring Data：提供了一套简化的 API，用于与关系型数据库、NoSQL 数据库等进行交互。

7.2 Spring Data JPA

Spring Data JPA 是 Spring Boot 提供的一种 ORM 解决方案，它使得数据持久化层的开发变得非常简便。通过 Spring Data JPA，我们可以通过声明式的方法来操作数据库，而无需编写复杂的 SQL 语句。

7.2.1 配置 Spring Data JPA

首先，在 pom.xml 中添加 Spring Data JPA 相关依赖：

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

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>

```

在 application.properties 中配置数据库连接信息：


```
package com.example.demo.repository;

import com.example.demo.model.User;
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {
    User findByName(String name);
}

```

7.2.4 使用 Repository

可以在服务层通过注入 UserRepository 来执行数据库操作。

```
package com.example.demo.service;

import com.example.demo.model.User;
import com.example.demo.repository.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public User findUserByName(String name) {
        return userRepository.findByName(name);
    }
}

```

7.2.5 数据访问方法

Spring Data JPA 提供了许多内置的方法，如 findAll()、save()、deleteById() 等。我们还可以定义自定义查询方法，Spring Data 会自动根据方法名称生成 SQL。

7.3 数据库迁移和版本控制

随着应用的持续开发，数据库的结构也会发生变化。Spring Boot 提供了两种常见的数据库版本管理工具：

1. Flyway
2. Liquibase

7.3.1 Flyway

Flyway 是一个数据库迁移工具，它通过执行版本化的 SQL 脚本来管理数据库的版本。

添加依赖：

```
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-core</artifactId>
</dependency>

```

配置 Flyway：

在 application.properties 中配置 Flyway：

```
spring.flyway.enabled=true
spring.flyway.locations=classpath:db/migration

```

在 src/main/resources/db/migration 文件夹中创建 SQL 脚本文件（如 V1__create_user_table.sql）：

```
CREATE TABLE user (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100)
);

```

当应用启动时，Flyway 会自动检查数据库版本并执行未执行的迁移脚本。


```
package com.example.demo.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Service;

@Service
public class UserService {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public void createUser(String name, String email) {
        String sql = "INSERT INTO user (name, email) VALUES (?, ?)";
        jdbcTemplate.update(sql, name, email);
    }
}

```

7.5 Spring 集成其他外部系统

Spring Boot 提供了很多方式来集成其他系统，比如 RESTful 服务、消息队列、外部服务等。

7.5.1 RESTful 集成

Spring Boot 支持通过 RestTemplate 或 WebClient 来集成外部的 RESTful 服务。

使用 RestTemplate 调用外部 RESTful 服务：

```java
package com.example.demo.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.web.reactive.function.client.WebClient;

@Service
public class ExternalService {

    @Autowired
    private WebClient.Builder webClientBuilder;

    public String getExternalData() {
        return webClientBuilder.baseUrl("https://api.example.com")
                .build()
                .get()
                .retrieve()
                .bodyToMono(String.class)
                .block();
    }
}
```

7.5.2 JMS 集成

Spring Boot 还支持通过 JMS（Java Message Service）来与消息队列进行集成。通过配置 spring-boot-starter-activemq 或其他 JMS 实现，可以轻松发送和接收消息。

7.6 课后练习

练习 1：使用 Spring Data JPA 创建一个 Product 实体类，并编写一个简单的 Repository 类来执行 CRUD 操作。

练习 2：使用 Flyway 管理数据库迁移，创建一个初始表并实现简单的数据插入操作。

练习 3：编写一个服务类，通过 JdbcTemplate 进行数据库查询，获取并显示用户信息。

练习 4：使用 RestTemplate 或 WebClient 调用外部 API，获取并处理返回的数据。

课后练习参考代码

第八章：Spring Security

目标

- 理解 Spring Security 的基本概念和功能
- 学会如何在 Spring Boot 项目中配置和使用 Spring Security
- 掌握常见的认证与授权机制，如表单登录、JWT、OAuth2 等
- 了解如何通过 Spring Security 强化应用的安全性

8.1 Spring Security 概述

Spring Security 是一个功能强大的安全框架，专门为 Java 应用程序提供身份验证和授权功能。它可以帮助开发者轻松实现安全的应用，保护应用免受未经授权的访问。

Spring Security 提供的主要功能包括：

- 认证（Authentication）：验证用户的身份。
- 授权（Authorization）：根据用户的角色或权限，决定是否允许访问特定的资源。
- CSRF 防护：防止跨站请求伪造攻击。
- 会话管理：支持会话并发控制、会话超时等。

8.2 Spring Security 的配置

8.2.1 添加依赖

首先，向 pom.xml 中添加 Spring Security 相关依赖：

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

```

添加该依赖后，Spring Boot 会自动配置 Spring Security，默认启用 HTTP 基本认证。

8.2.2 基本认证和自定义配置

默认情况下，Spring Security 会启用 HTTP 基本认证，需要提供用户名和密码才能访问任何受保护的资源。可以通过自定义配置类来修改默认行为。

例如，可以通过创建一个继承 WebSecurityConfigurerAdapter 的配置类来定制安全配置。

在实际项目中，通常会使用数据库来存储用户数据，但为了快速演示，我们可以使用内存中存储用户信息。

在这个示例中，我们使用 InMemoryUserDetailsManager 存储一个内存用户，并通过 withDefaultPasswordEncoder() 生成一个加密密码。

8.3.2 基于数据库的用户认证

在实际项目中，通常会通过数据库来存储用户信息。可以使用 JdbcUserDetailsManager 或 UserDetailsService 实现。

8.4.1 基于角色的访问控制

Spring Security 支持基于角色的访问控制。可以通过 hasRole() 或 hasAuthority() 方法进行配置。

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
            .antMatchers("/admin/**").hasRole("ADMIN")  // 只有 ADMIN 角色的用户才能访问
            .antMatchers("/user/**").hasAnyRole("USER", "ADMIN")  // 用户或管理员均可访问
            .anyRequest().authenticated();
}

```

8.4.2 基于权限的访问控制

除了角色外，Spring Security 还支持基于权限的访问控制。权限通常是细粒度的授权，使用 hasAuthority() 方法进行设置。

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
            .antMatchers("/admin/**").hasAuthority("ROLE_ADMIN")  // 基于权限控制
            .anyRequest().authenticated();
}

```

8.5 CSRF 防护

Spring Security 默认启用 CSRF（跨站请求伪造）防护，它会保护应用免受恶意请求的侵害。对于不需要 CSRF 防护的请求，可以通过禁用 CSRF 来绕过它。


```
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import java.util.Date;

public class JwtUtil {
    private String secretKey = "mySecretKey";

    public String generateToken(String username) {
        return Jwts.builder()
                .setSubject(username)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60 * 10))  // 10 hours
                .signWith(SignatureAlgorithm.HS256, secretKey)
                .compact();
    }
}

```

解析 JWT：

```
import io.jsonwebtoken.Jwts;

public class JwtUtil {

    public String extractUsername(String token) {
        return Jwts.parser()
                .setSigningKey("mySecretKey")
                .parseClaimsJws(token)
                .getBody()
                .getSubject();
    }
}

```

8.6.2 集成 JWT 到 Spring Security 中

```
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private JwtUtil jwtUtil;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {

        String token = request.getHeader("Authorization");
        if (token != null && token.startsWith("Bearer ")) {
            token = token.substring(7);
            String username = jwtUtil.extractUsername(token);
            // 进一步验证用户身份，并设置用户信息到 SecurityContext
        }
        filterChain.doFilter(request, response);
    }
}

```

在 Spring Security 配置中注册 JWT 过滤器：

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .addFilterBefore(jwtAuthenticationFilter(), UsernamePasswordAuthenticationFilter.class)
        .authorizeRequests()
            .anyRequest().authenticated();
}

@Bean
public JwtAuthenticationFilter jwtAuthenticationFilter() {
    return new JwtAuthenticationFilter(jwtUtil);
}

```


第九章：Spring Batch

目标

- 理解 Spring Batch 的核心概念和架构
- 掌握 Spring Batch 中的常见组件，如 Job, Step, ItemReader, ItemProcessor 和 ItemWriter
- 学会如何在 Spring Boot 项目中集成和配置 Spring Batch
- 了解批处理作业的异常处理、事务管理以及性能优化

9.1 Spring Batch 概述

Spring Batch 是一个专门为批处理应用设计的框架，提供了高效的批量数据处理和作业调度功能。它能够帮助开发者处理大量数据，支持事务管理、作业监控、作业恢复等功能。Spring Batch 使得开发批量处理应用变得更加简便，尤其适用于数据迁移、报表生成、ETL（提取、转换、加载）等任务。

Spring Batch 的主要特性包括：

- 作业和步骤：作业是批处理任务的顶层容器，包含一个或多个步骤。每个步骤代表一个独立的任务。
- 事务管理：Spring Batch 为每个步骤提供事务管理，确保数据一致性。
- 支持失败恢复：Spring Batch 能够自动处理失败并支持重试、恢复。
- 可扩展性：可以灵活地根据需求扩展 ItemReader、ItemProcessor 和 ItemWriter。

9.2 Spring Batch 的核心组件

Spring Batch 的主要构件是 **Job**、**Step**、**ItemReader**、**ItemProcessor** 和 **ItemWriter**。下面逐一介绍这些组件。

9.2.1 Job

Job 是一个批处理作业的容器，它由多个步骤（Step）组成。每个作业可以包含多个步骤，并按顺序执行。

```java
@Bean
public ItemReader<MyItem> reader() {
    return new JdbcCursorItemReaderBuilder<MyItem>()
            .dataSource(dataSource)
            .sql("SELECT * FROM my_table")
            .rowMapper(new MyItemRowMapper())
            .build();
}

```

9.2.4 ItemProcessor

ItemProcessor 用于处理读取到的数据。通常用于数据的转换、清洗、验证等操作。

```java
@Bean
public ItemProcessor<MyItem, MyProcessedItem> processor() {
    return item -> new MyProcessedItem(item.getName(), item.getValue() * 2);
}

```

9.2.5 ItemWriter

ItemWriter 用于将处理后的数据写入目标数据源。常见的目标数据源有数据库、文件、消息队列等。

```java
@Bean
public ItemWriter<MyProcessedItem> writer() {
    return new JdbcBatchItemWriterBuilder<MyProcessedItem>()
            .dataSource(dataSource)
            .sql("INSERT INTO processed_table (name, value) VALUES (:name, :value)")
            .beanMapped()
            .build();
}

```



```java
@Bean
public JobLauncherTestUtils jobLauncherTestUtils() {
    return new JobLauncherTestUtils();
}

```

9.4 Spring Batch 的事务管理

在 Spring Batch 中，每个 Step 都是一个事务。如果 Step 中的处理失败，Spring Batch 会自动回滚事务。你可以通过 @Transactional 注解来控制事务的边界。

9.4.1 配置事务管理器

```java
@Bean
public PlatformTransactionManager transactionManager(DataSource dataSource) {
    return new DataSourceTransactionManager(dataSource);
}

```

9.4.2 事务管理的优化

Spring Batch 提供了两种优化事务处理的方法：

- Chunk-Oriented Processing：每个 chunk 由一组数据组成，数据的读取、处理和写入都在一个事务内完成。
- Commit Interval：你可以设置处理数据的间隔（如每 100 条记录处理一次），通过 chunk() 方法设置。

```java
@Bean
public Step myStep(StepBuilderFactory stepBuilderFactory, ItemReader<MyItem> reader,
ItemProcessor<MyItem, MyItem> processor, ItemWriter<MyItem> writer) {
    return stepBuilderFactory.get("myStep")
            .<MyItem, MyItem> chunk(10)
            .reader(reader)
            .processor(processor)
            .writer(writer)
            .faultTolerant()
            .skip(Exception.class)
            .skipLimit(5)
            .retry(Exception.class)
            .retryLimit(3)
            .build();
}

```

9.5.2 作业重启

Spring Batch 支持作业重启。你可以在作业执行失败时进行重启，并从最后一次成功的步骤开始执行。

```java
@Bean
public Job myJob(JobBuilderFactory jobBuilderFactory, StepBuilderFactory
stepBuilderFactory) {
    return jobBuilderFactory.get("myJob")
            .start(myStep())
            .next(anotherStep())
            .preventRestart()  // 禁止重启
            .build();
}

```

9.6 性能优化

Spring Batch 提供了一些优化机制来提高批处理作业的性能，包括：

- 多线程处理（Multithreaded Processing）：在多个线程中并行处理数据。
- 流式处理（Streaming）：减少内存消耗，逐条处理数据。

```java
@Bean
public Step myStep(StepBuilderFactory stepBuilderFactory, ItemReader<MyItem> reader,
ItemProcessor<MyItem, MyItem> processor, ItemWriter<MyItem> writer) {
    return stepBuilderFactory.get("myStep")
            .<MyItem, MyItem> chunk(1000)
            .reader(reader)
            .processor(processor)
            .writer(writer)
            .taskExecutor(new SimpleAsyncTaskExecutor())  // 使用异步任务执行器
            .build();
}

```

9.7 小结

本章介绍了 Spring Batch 的核心概念和组件，包括作业（Job）、步骤（Step）、读取器（ItemReader）、处理器（ItemProcessor）和写入器（ItemWriter）。我们学习了如何配置和使用 Spring Batch 进行批量处理，如何管理事务，如何处理异常和恢复，以及如何优化批处理作业的性能。通过本章的内容，你将能够在 Spring Boot 项目中实现高效、可靠的批处理系统。

Spring Batch - 课后练习

练习 1：创建一个简单的 Spring Batch 作业

目标：通过本练习，你将实现一个简单的 Spring Batch 作业，其中包含一个步骤，该步骤读取数据，处理数据并将数据写入数据库。

步骤：

1. 创建一个 Spring Boot 项目并添加 Spring Batch 依赖。
2. 创建一个数据库表 employee，该表包含员工的 id、name 和 salary 字段。
3. 创建一个 Employee 类来表示数据模型。
4. 创建一个 ItemReader，从数据库中读取所有员工数据。
5. 创建一个 ItemProcessor，将员工薪资增加 10%。
6. 创建一个 ItemWriter，将处理后的数据写回数据库中的 employee_processed 表。

5. 测试作业执行时间，并与未优化的版本进行比较。

练习 5：作业重启和恢复

目标：通过本练习，你将学习如何在 Spring Batch 中配置作业的重启和恢复功能。

步骤：

1. 创建一个 Spring Batch 作业，其中包含两个步骤：一个读取操作和一个写入操作。
2. 在第二个步骤中，模拟处理失败（例如通过在 ItemProcessor 中抛出一个异常）。
3. 配置作业以支持重启功能，并设置作业恢复策略。
4. 运行作业，确保作业在失败后能够正确恢复，并从最后一个成功的步骤继续执行。
5. 通过作业日志，验证重启和恢复的过程。

练习 6：分区处理（Partitioned Processing）

目标：通过本练习，你将学习如何使用分区处理来优化批量数据的处理。

步骤：

1. 创建一个 Spring Batch 作业，读取大量员工数据（例如 10000 条数据）。
2. 使用分区处理（PartitionedStep）将数据划分为多个部分（如每个分区处理 2000 条数据）。
3. 配置多个子步骤来并行处理每个分区的数据。
4. 在 ItemProcessor 和 ItemWriter 中添加逻辑，以确保分区处理中的数据能够正确处理。
5. 测试分区处理和并行处理的性能，比较与未分区处理版本的性能差异。

练习 7：创建一个 ETL 作业

目标：通过本练习，你将使用 Spring Batch 实现一个完整的 ETL（提取、转换、加载）作业。

步骤：

1. 创建一个 Spring Batch 作业，读取一个 CSV 文件，其中包含多个不同的源数据。
2. 创建一个 ItemProcessor，将数据从 CSV 文件的格式转换为目标数据库所需的格式。
3. 创建一个 ItemWriter，将转换后的数据写入数据库。
4. 配置 Job 以实现 ETL 过程：首先读取数据（提取），然后转换数据，最后将数据写入数据库（加载）。
5. 运行 ETL 作业，确保数据能够正确地从文件提取并加载到数据库中。

- 在 application.properties 中启用 Eureka Server。

```
server.port=8761
spring.application.name=eureka-server
eureka.client.registerWithEureka=false
eureka.client.fetchRegistry=false

```

- 在启动类上加上 @EnableEurekaServer 注解。

2. 创建 Eureka 客户端：
   - 在微服务应用中添加 spring-cloud-starter-netflix-eureka-client 依赖。
   - 配置 Eureka 服务器地址。

```
spring.application.name=my-service
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/

```

- 在启动类上加上 @EnableEurekaClient 注解。

3. 服务发现：
   - 微服务在启动时会注册到 Eureka Server 上，其他微服务可以通过 Eureka 客户端来发现注册的服务。

4. 负载均衡（Ribbon）

Ribbon 是一个客户端负载均衡工具，可以与 Eureka 一起工作，帮助微服务实现对多个实例的负载均衡。

步骤：

1. 在微服务客户端中添加 spring-cloud-starter-netflix-ribbon 依赖。
2. 使用 @LoadBalanced 注解在 RestTemplate 上启用 Ribbon 负载均衡。

```
@Bean
@LoadBalanced
public RestTemplate restTemplate() {
    return new RestTemplate();
}
```

3. 使用 RestTemplate 发起请求时，Ribbon 会自动根据服务名称进行负载均衡。

```
@Autowired
private RestTemplate restTemplate;

public String callService() {
    return restTemplate.getForObject("http://my-service/endpoint", String.class);
}

```

5. 断路器（Hystrix）

Hystrix 是一种用于防止级联故障的断路器模式解决方案，它可以在服务调用失败时提供降级方案，防止故障蔓延到其他服务。

步骤：

1. 在微服务应用中添加 spring-cloud-starter-netflix-hystrix 依赖。
2. 在服务方法上使用 @HystrixCommand 注解，定义断路器和降级方法。

```
@HystrixCommand(fallbackMethod = "fallbackMethod")
public String callExternalService() {
    return restTemplate.getForObject("http://external-service/endpoint", String.class);
}

public String fallbackMethod() {
    return "Service is currently unavailable, please try again later.";
}

```

3. 在应用的启动类上启用 Hystrix：

在 application.properties 中配置 Git 存储库或本地文件系统。

```java
@EnableBinding(Sink.class)
public class MessageConsumer {
    @StreamListener(Sink.INPUT)
    public void handleMessage(String message) {
        System.out.println("Received: " + message);
    }
}

```

9. Spring Cloud Sleuth

Spring Cloud Sleuth 提供了分布式追踪功能，可以跟踪请求在微服务架构中的流转路径，帮助排查问题。

步骤：

1. 在微服务应用中添加 spring-cloud-starter-sleuth 依赖。
2. Sleuth 会自动为所有请求生成唯一的追踪 ID（trace ID）和跨度 ID（span ID）。
3. 可以通过日志或者集成 Zipkin 等工具来可视化追踪信息。

10. Spring Cloud Config Client

Spring Cloud Config Client 用于从 Spring Cloud Config Server 获取配置。

步骤：

1. 在客户端应用中添加 spring-cloud-starter-config 依赖。
2. 配置从 Spring Cloud Config Server 拉取配置。

```properties
spring.application.name=my-client
spring.cloud.config.uri=http://localhost:8888

```

小结

本章详细介绍了 Spring Cloud 的核心组件，包括服务注册与发现、负载均衡、断路器、API 网关、分布式配置等。通过这些组件，Spring Cloud 可以帮助我们构建和管理一个高度可扩展、可靠、易于维护的分布式系统。

路由器用于根据消息的内容将消息路由到不同的目标。常见的路由器有基于条件的路由和基于内容的路由。

8. Aggregators（聚合器）

聚合器将多个消息合并成一个单一消息。它用于在消息流中汇集数据，并在某个条件满足时将数据合并。

9. Splitter（拆分器）

拆分器用于将一个消息分解成多个消息，以便并行处理。

3. Spring Integration组件

Spring Integration 提供了丰富的组件，允许开发者使用消息通道、适配器、消息端点等构建集成应用程序。以下是一些常见的组件：

1. Channel（通道）
   - DirectChannel：消息在同一线程内同步传递。
   - QueueChannel：消息通过队列异步传递。
   - PublishSubscribeChannel：消息被广播给多个接收者。
2. Message Endpoints（消息端点）
   - Service Activator：将消息传递给服务方法进行处理。
   - Transformer：将消息的有效载荷转换为其他格式。
   - Router：根据消息的内容将其路由到不同的通道。
3. Adapters（适配器）
   - JMS Adapter：用于与 JMS 消息队列进行集成。
   - FTP Adapter：用于与 FTP 服务器进行集成。
   - File Adapter：用于与文件系统进行集成。
4. Messaging Gateway（消息网关）
   - 消息网关允许我们将消息从外部系统引入到 Spring Integration 流程中，或者将消息从 Spring Integration 流程输出到外部系统。它充当了连接外部系统和 Spring Integration 内部流程的桥梁。

4. Spring Integration流程示例

在 Spring Integration 中，应用程序的消息流通常遵循以下流程：

- 消息被发送到消息通道。
- 消息通道将消息传递给消息端点（如服务激活器、转换器或路由器）。
- 消息通过通道在集成系统中流动，可以经过过滤器、转换器等组件的处理。
- 最终，消息被发送到目标组件，可能是数据库、外部服务、文件系统等。

2. Message Filter（消息过滤器）

根据某些条件对消息进行过滤，决定是否允许消息继续传递。

3. Content-Based Router（基于内容的路由器）

根据消息的内容（如字段值、类型等）路由到不同的目标通道。

4. Splitter（拆分器）

将消息分解成多个子消息进行并行处理。

5. Aggregator（聚合器）

将多个消息合并成一个消息，以便进行汇总处理。

6. Service Activator（服务激活器）

将消息传递给外部服务或方法进行处理。

6. Spring Integration的配置方式

Spring Integration 提供了多种配置方式，包括基于 XML 配置和基于 Java 配置的方式。使用 XML 配置时，可以通过 <int:xxx> 标签来定义消息通道、端点、适配器等组件。使用 Java 配置时，可以通过注解和 Java 配置类来实现。

基于 Java 配置的示例:

小结

本章详细介绍了 **Spring Integration** 框架及其核心概念，包括消息、通道、适配器、消息端点等。通过使用 Spring Integration，开发者可以更容易地构建松耦合、可扩展的企业级集成解决方案。通过应用企业集成模式（EIP），Spring Integration 可以帮助开发者处理复杂的消息流转、转换、路由等集成问题。

5. 处理 HTTP 请求与响应

在 Spring WebFlux 中，处理 HTTP 请求和响应是通过 Mono 和 Flux 来实现的。可以使用这些类型来表示请求的输入和输出，并通过事件驱动机制进行异步处理。

示例 1：使用 Mono 处理单个请求

```
@RestController
public class ReactiveController {

    @GetMapping("/hello")
    public Mono<String> sayHello() {
        return Mono.just("Hello, Spring WebFlux!");
    }
}

```

在上面的代码中，Mono.just() 方法表示返回一个包含字符串的异步序列，响应会在 Mono 完成时返回。

示例 2：使用 Flux 处理多个请求

```
@RestController
public class ReactiveController {

    @GetMapping("/items")
    public Flux<String> getItems() {
        return Flux.just("Item1", "Item2", "Item3");
    }
}

```

在这个例子中，Flux.just() 方法返回一个包含多个项的异步序列，客户端会接收到这三个字符串的响应。

6. 响应式数据访问

Spring WebFlux 提供了对响应式数据库访问的支持，特别是通过 **Spring Data Reactive** 模块。它支持与响应式数据库进行交互，如 **MongoDB**、**Cassandra** 等，以及对传统关系型数据库（如 MySQL）提供响应式支持。

9. 测试响应式应用

Spring WebFlux 提供了对响应式应用程序进行测试的支持。可以使用 WebTestClient 来模拟客户端请求并验证响应。

示例 4：使用 WebTestClient 测试响应式端点

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class ReactiveControllerTests {

    @Autowired
    private WebTestClient webTestClient;

    @Test
    public void testSayHello() {
        webTestClient.get().uri("/hello")
                     .exchange()
                     .expectStatus().isOk()
                     .expectBody(String.class).isEqualTo("Hello, Spring WebFlux!");
    }
}

```

WebTestClient 是一个非阻塞的 HTTP 客户端，允许我们异步地发送请求并验证响应。

10. 小结

Spring WebFlux 是一种高效的异步和响应式编程框架，专门用于构建可扩展的 web 应用程序。它通过支持异步 I/O、反应式流（Mono 和 Flux）和响应式数据库访问，提供了比传统 Spring MVC 更高的并发性能和更低的资源消耗。WebFlux 对比传统编程模型的优势在于能够处理更多并发请求，尤其适用于高并发和低延迟的场景。

课后练习

1. 基本应用： 创建一个简单的 Spring WebFlux 应用，提供一个 /hello 路径，返回 "Hello, Spring WebFlux!"。
2. Flux 示例： 实现一个返回多个字符串项的端点，例如 /items，使用 Flux 返回一组数据。
3. 数据库访问： 使用 Spring Data Reactive 和 MongoDB 创建一个响应式应用，查询某个类别的商品信息。
4. 性能测试： 使用 WebTestClient 编写测试，验证你应用的响应性和正确性。

> **在 WebFlux 体系下，一切能非阻塞的，都应该选非阻塞的！**
> —— 否则就需要用 Schedulers.boundedElastic() 额外保护阻塞代码，不然破坏性能。

小提示：

- **HTTP调用**、**数据库访问**、**缓存读写**，是最最需要关注的三大重点。
- **RabbitMQ/Kafka消息中间件**：如果未来你有异步消息需求，也要选支持 Reactive 的库。
- **如果一时找不到替代方案**，记得用 .subscribeOn(Schedulers.boundedElastic()) 包住。

Spring Boot 中的一些配置可以通过配置类来优化，避免在每次应用启动时进行不必要的资源加载。通过精确地定义需要的配置，能够减少启动时的计算和资源开销。

示例 3：自定义配置类

```
@Configuration
public class CustomConfig {
    @Bean
    public SomeService someService() {
        return new SomeService();
    }
}

```

3. 内存优化

Spring 应用通常需要大量的内存来加载上下文和所有 Bean。在大规模应用中，内存管理和优化尤为重要。

3.1. 减少 Bean 的数量

每个 Bean 的创建都会占用一定的内存。因此，减少不必要的 Bean 和依赖项，可以有效降低内存占用。通过使用原型（@Scope("prototype")）模式，确保只创建实际需要的 Bean。

示例 4：减少不必要的 Bean

```
@Component
@Scope("singleton")
public class SingletonService {
    // Singleton Service
}

```

3.2. 使用对象池

对于一些需要频繁创建和销毁的对象，如数据库连接和线程池，使用对象池技术可以显著减少内存消耗和创建对象的开销。Spring Boot 提供了对 **HikariCP** 等连接池的集成，能够有效提升数据库连接的性能。

示例 5：配置数据库连接池

```
public Page<Order> findOrders(Pageable pageable);

```

5. 缓存优化

缓存可以显著减少频繁数据库查询或计算密集型操作的压力，提升应用性能。

5.1. 使用 Spring Cache

Spring 提供了一个统一的缓存抽象，支持多种缓存实现，如 **EhCache**、**Redis**、**Caffeine** 等。通过启用缓存功能，可以减少对数据库的重复查询，提高性能。

示例 9：启用缓存

```
@EnableCaching
@Configuration
public class CacheConfig {
    @Bean
    public CacheManager cacheManager() {
        return new ConcurrentMapCacheManager("items");
    }
}

```

5.2. 配置缓存策略

缓存策略（如过期时间、缓存清理等）对于保证缓存的有效性和性能至关重要。可以根据需求配置不同的缓存过期时间或清理策略。

示例 10：配置缓存过期时间

```
spring.cache.ehcache.config=classpath:ehcache.xml

```

6. 异步与并发优化

Spring 通过 @Async 注解为方法提供异步执行功能，这在处理时间较长的任务时非常有用。通过异步执行任务，可以提高系统的响应速度，避免阻塞线程。

6.1. 使用 @Async 注解

通过 @Async 注解，可以将某些任务放入后台执行，从而避免阻塞主线程。


```
@Around("execution(* com.example.service.*.*(..))")
public Object aroundMethod(ProceedingJoinPoint joinPoint) throws Throwable {
    return joinPoint.proceed();
}

```

8. Spring Web 性能优化

Web层是用户直接交互的部分，性能的优化对于用户体验至关重要。优化 Web 层主要集中在减少 HTTP 请求的延迟、优化响应时间以及减小传输的数据量。

8.1. 启用 HTTP 缓存

通过合理设置缓存策略，可以减少对服务器的重复请求，提高性能。

示例 14：启用 HTTP 缓存

```
@ResponseBody
@RequestMapping("/image")
public ResponseEntity<byte[]> getImage() {
    HttpHeaders headers = new HttpHeaders();
    headers.setCacheControl(CacheControl.maxAge(1, TimeUnit.DAYS));
    return new ResponseEntity<>(imageBytes, headers, HttpStatus.OK);
}

```

8.2. 使用压缩和内容协商

使用 GZIP 压缩可以减少响应数据的大小，尤其是当响应包含大量文本或 JSON 数据时。

示例 15：启用 GZIP 压缩

```
server.compression.enabled=true
server.compression.min-response-size=1024

```

9. 小结

Spring 框架为我们提供了丰富的功能和灵活的设计，但也可能带来性能开销。在构建高性能应用时，需要从多个方面进行优化：

- 内存优化：减少不必要的 Bean 数量，使用对象池和原型模式降低内存消耗。
- 数据库性能：采用批处理、分页查询等方式减少数据库压力。
- 缓存机制：利用 Spring Cache 提高性能，减少重复查询。

1. 启用Session：Spring MVC自动管理`HttpSession`。

java

```java
@PostMapping("/login")
public String login(User user, HttpSession session) {
    session.setAttribute("currentUser", user);
    return "dashboard";
}
```

2. 安全配置：

- 强制HTTPS：`server.ssl.enabled=true`（application.properties）。
- Cookie安全标志：

java

```java
Cookie cookie = new Cookie("JSESSIONID", sessionId);
cookie.setSecure(true);   // 仅HTTPS传输
cookie.setHttpOnly(true); // 禁止JS访问
cookie.setSameSite(SameSite.LAX); // 防止CSRF
```

RESTful API/SPA（如移动端、单页应用）

适用技术：JWT（JSON Web Token）

核心机制：

服务端生成签名Token，客户端存储（如LocalStorage）并在请求头中传递。

Token自包含用户信息（如ID、角色），服务端无需存储会话数据。

1. 生成Token：

java

```java
String token = Jwts.builder()
    .setSubject(user.getEmail())
    .claim("roles", user.getRoles())
    .setExpiration(new Date(System.currentTimeMillis() + 3600000)) // 1小时
    .signWith(SignatureAlgorithm.HS512, "secretKey")
    .compact();
```

2. 客户端存储：

javascript

```javascript
// 前端保存Token（登录成功后）
localStorage.setItem("token", token);
```

3. 请求鉴权：

java

```java
@GetMapping("/api/profile")
public ResponseEntity<?> profile(@RequestHeader("Authorization") String token) {
    Claims claims = Jwts.parser()
        .setSigningKey("secretKey")
        .parseClaimsJws(token.replace("Bearer ", ""))
        .getBody();
    String user = claims.getSubject();
    return ResponseEntity.ok("User: " + user);
}
```

跨域/分布式系统（如微服务架构）

适用技术：Spring Session + Redis

示例1：简单UserController（数据绑定学习）

1. 概述

- 目标：学习Spring MVC的数据绑定机制
- 特点：
  - 完全内存存储（无数据库）
  - 使用基础RESTful API实现
  - 演示请求/响应数据绑定

2. 核心功能

- POST /users：创建用户（JSON数据绑定）
- GET /users/{id}：路径参数绑定
- PUT /users/{id}：路径参数+请求体绑定
- 自动ID生成（AtomicLong实现）

3. 数据绑定关键点

注解/特性	说明	示例代码片段
@RequestBody	将请求体JSON自动绑定到Java对象	createUser(@RequestBody User user)
@PathVariable	绑定URL路径参数到方法参数	getUserById(@PathVariable Long id)
@RequestParam	绑定查询参数（本示例未使用，常见用法）	-
自动类型转换	Spring自动处理String到Long等基本类型的转换	id 参数自动转为Long类型
参数验证	可结合@Valid进行数据校验（本示例未包含）	-

4. 项目结构


```
package com.example.demo.model;

public class User {
    private Long id;
    private String name;
    private String email;

    // 构造方法、Getter 和 Setter
    public User() {}

    public User(Long id, String name, String email) {
        this.id = id;
        this.name = name;
        this.email = email;
    }

    // 省略 Getter 和 Setter（实际开发中建议用 Lombok @Data）
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }
}

```

（2）控制器 UserController.java

```
    // 更新用户
    @PutMapping("/{id}")
    public User updateUser(@PathVariable Long id, @RequestBody User updatedUser) {
        User user = getUserById(id);
        if (user != null) {
            user.setName(updatedUser.getName());
            user.setEmail(updatedUser.getEmail());
        }
        return user;
    }

    // 删除用户
    @DeleteMapping("/{id}")
    public void deleteUser(@PathVariable Long id) {
        users.removeIf(user -> user.getId().equals(id));
    }
}
```

（3）启动类 DemoApplication.java

```
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

---

3. 测试 API

操作	HTTP方法	URL	请求体示例（JSON）
获取所有用户	GET	http://localhost:8080/api/users	-
获取单个用户	GET	http://localhost:8080/api/users/1	-
创建用户	POST	http://localhost:8080/api/users	{"name": "Alice", "email": "alice@example.com"}
更新用户	PUT	http://localhost:8080/api/users/1	{"name": "Alice Updated", "email": "alice_new@example.com"}
删除用户	DELETE	http://localhost:8080/api/users/1	-

---

4. 关键点说明

1. **内存存储**：使用 List<User> 和 AtomicLong 模拟数据库。
2. **RESTful 设计**：
   - GET：查询
   - POST：新增
   - PUT：更新
   - DELETE：删除
3. **注解**：
   - @RestController：标记为REST控制器。
   - @RequestMapping：定义基础路径。
   - @PathVariable：获取URL中的参数。
   - @RequestBody：解析JSON请求体。

---

5. 扩展建议

- 添加输入验证（如用 @Valid）。
- 返回标准HTTP状态码（如 ResponseEntity）。
- 使用 Lombok 简化代码（如 @Data 自动生成Getter/Setter）。

示例2：MySQL分层架构示例

```
sequenceDiagram
    Client->>Controller: HTTP Request
    Controller->>Service: 调用业务方法
    Service->>Repository: 调用数据库操作
    Repository->>Database: 执行SQL
    Database-->>Repository: 返回数据
    Repository-->>Service: 返回Entity
    Service-->>Controller: 返回DTO
    Controller-->>Client: JSON Response
```

实现逻辑

一、数据库准备

```
-- 创建数据库
CREATE DATABASE springboot_db
    CHARACTER SET utf8mb4
    COLLATE utf8mb4_unicode_ci;

-- 使用数据库
USE springboot_db;

-- 创建用户表（与之前完全一致）
CREATE TABLE users (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

二、完整代码

1. 项目结构

3. 仓库接口 UserRepository.java

```
package com.example.demo.repository;

import com.example.demo.model.User;
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {
    User findByEmail(String email);
}
```

4. 服务层 UserService.java

```
package com.example.demo.service;

import com.example.demo.model.User;
import java.util.List;

public interface UserService {
    List<User> getAllUsers();
    User getUserById(Long id);
    User createUser(User user);
    User updateUser(Long id, User userDetails);
    void deleteUser(Long id);
}
```

5. 服务实现

```

import com.example.demo.model.User;
import com.example.demo.repository.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private UserRepository userRepository;

    @Override
    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

    @Override
    public User getUserById(Long id) {
        return userRepository.findById(id).orElse(null);
    }

    @Override
    public User createUser(User user) {
        return userRepository.save(user);
    }

    @Override
    public User updateUser(Long id, User userDetails) {
        User user = userRepository.findById(id).orElse(null);
        if (user != null) {
            user.setName(userDetails.getName());
            user.setEmail(userDetails.getEmail());
            return userRepository.save(user);
        }
        return null;
    }
```



```
package com.example.demo.controller;

import com.example.demo.model.User;
import com.example.demo.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import java.util.List;

@RestController
@RequestMapping("/api/users")
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping
    public List<User> getAllUsers() {
        return userService.getAllUsers();
    }

    @GetMapping("/{id}")
    public User getUserById(@PathVariable Long id) {
        return userService.getUserById(id);
    }

    @PostMapping
    public User createUser(@RequestBody User user) {
        return userService.createUser(user);
    }

    @PutMapping("/{id}")
    public User updateUser(@PathVariable Long id, @RequestBody User userDetails) {
        return userService.updateUser(id, userDetails);
    }

    @DeleteMapping("/{id}")
    public void deleteUser(@PathVariable Long id) {
        userService.deleteUser(id);
    }
```

```
curl -X POST -H "Content-Type: application/json" -d '{
    "name": "测试用户",
    "email": "test@example.com"
}' http://localhost:8080/api/users
```

1. 查询所有用户：

```
curl http://localhost:8080/api/users
```



```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>io.codescience</groupId>
    <artifactId>spring-practice</artifactId>
    <version>1.0.0</version>
    <packaging>pom</packaging>
    <name>Spring Practice Project</name>

    <modules>
        <!-- 子模块将自动添加到这里 -->
    </modules>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <java.version>17</java.version>
        <spring-boot.version>3.1.0</spring-boot.version>
    </properties>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-dependencies</artifactId>
                <version>${spring-boot.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
</project>

```

```
config/
service/
model/
```

2.

模块类型	模块名称	ArtifactId	主包路径
核心模块	Core Container	spring-core-practice	io.codescience.springpractice.core
AOP模块	Spring AOP	spring-aop-practice	io.codescience.springpractice.aop
Web模块	Spring MVC	spring-mvc-basic	io..springcodescience practice.web
数据模块	Spring Data JPA	spring-data-jpa	io.codescience.springpractice.data.jpa

3.

最终结构应如下（IDEA项目视图）：

```
spring-practice (root)
├── pom.xml
├── spring-core-practice
│   ├── src
│   │   ├── main
│   │   │   ├── java
│   │   │   │   └── io.codescience.springpractice.core
│   │   │   │       ├── config
│   │   │   │       ├── service
│   │   │   │       └── model
│   │   │   └── resources
│   │   └── test
│   └── pom.xml
└── spring-mvc-basic
    └── ... (类似结构)
```


- 【选学】spring-cloud-feign - 服务调用

批处理模块

- 【选学】spring-batch-practice - 基础批处理
- 【选学】spring-batch-file-import - 文件导入批处理