- Spring基础
  - 基础
    - 什么是Spring？ — Spring的一个最大的目的就是JAVA EE开发更加容易的一个框架
    - Spring的特性和优势
      - 特性
        - 非侵入式：基于Spring开发的应用中的对象可以不依赖于Spring的API
        - 控制反转：IOC——Inversion of Control，指的是将对象的创建权交给 Spring 去创建。使用 Spring 之前，对象的创建都是由我们自己在代码中new创建。而使用 Spring 之后，对象的创建都是给了 Spring 框架。
        - 依赖注入：DI——Dependency Injection，是指依赖的对象不需要手动调用 setXX 方法去设置，而是通过配置赋值。
        - 面向切面编程：Aspect Oriented Programming——AOP
        - 容器：Spring 是一个容器，因为它包含并且管理应用对象的生命周期
        - 组件化：Spring 实现了使用简单的组件配置组合成一个复杂的应用 — 在 Spring 中可以使用XML和Java注解组合这些对象。一站式：在 IOC 和 AOP 的基础上可以整合各种企业应用的开源框架和优秀的第三方类库（实际上 Spring 自身也提供了表现层的 SpringMVC 和持久层的 Spring JDBC）
      - 好处
        - Spring 可以使开发人员使用 POJOs 开发企业级的应用程序
        - Spring 在一个单元模式中是有组织的。即使包和类的数量非常大
        - 轻量级的 IOC 容器往往是轻量级的，例如，特别是当与 EJB 容器相比的时候。这有利于在内存和 CPU 资源有限的计算机上开发和部署应用程序
        - Spring 提供了一致的事务管理接口，可向下扩展到 （使用一个单一的数据库，例如）本地事务并扩展到全局事务
    - Spring Bean
      - 概念 — Spring框架管理这些Bean的创建工作，即由用户管理Bean转变为框架管理Bean，这个就叫控制反转
      - IoC的好处 — 把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，所以对象与对象之间是松散耦合，这样也方便测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活
      - 概念 — Spring里面的bean就类似是定义的一个组件，而这个组件的作用就是实现某个功能的，这里所定义的bean就相当于给了你一个更为简便的方法来调用这个组件去实现你要完成的功能
      - 存放位置 — Spring 框架托管创建的Bean放在 IoC Container
      - Spring 框架既然能管了Bean的生成，必然需要管理整个Bean的生命周期 — 整个生命周期可以大致分为3个大的阶段，分别是：创建、使用、销毁。还可以进一步分为5个小的阶段：实例化、初始化、注册Destruction回调、Bean的正常使用以及Bean的销毁
      - Spring 框架为了更好让用户配置Bean，必然会引入不同方式来配置Bean（IoC配置的三种方式）
        - xml 配置 — 顾名思义，就是将bean的信息配置.xml文件里，通过Spring加载文件为我们创建bean。
          - 优点 — ：这种方式出现很多早前的SSM项目中，将第三方类库或者一些配置工具类都以这种方式进行配置，主要原因是由于第三方类不支持Spring注解。
          - 优点 — ：可以使用于任何场景，结构清晰，通俗易懂
          - 缺点 — ：配置繁琐，不易维护，枯燥无味，扩展性差
        - Java 配置 — 将类的创建交给我们配置的JavcConfig类来完成。Spring只负责维护和管理，采用纯Java创建方式。其本质上就是把在XML上的配置声明转移到Java配置类中
          - 优点 — ：适用于任何场景，配置方便，因为是纯Java代码，扩展性高，十分灵活
          - 缺点 — ：由于是采用Java类的方式，声明不明显，如果大量配置，可读性比较差
        - 注解配置 — 通过在类上加注解的方式，来声明一个类交给Spring管理，Spring会自动扫描带有@Component，@Controller，@Service，@Repository这四个注解的类，然后帮我们创建并管理，前提是需要先配置Spring的注解扫描器
          - 优点 — 开发便捷，通俗易懂，方便维护。
          - 缺点 — 具有局限性。对于一些第三方资源，无法添加注解。只能采用XML或JavaConfig的方式配置
        - 目前的主流方式是 注解 + Java 配置
      - Spring Bean的创建是典型的工厂模式
        - BeanFactory：工厂模式定义了IOC容器的基本功能规范 — BeanFactory和FactroyBean的关系
          - BeanFactory是Spring IoC容器的一个接口，用来获取Bean以及管理Bean的依赖注入和生命周期
          - FactoryBean是一个接口，用于定义一个工厂Bean，它可以产生某种类型的对象
        - BeanRegistry：向IOC容器手工注册 BeanDefinition 对象的方法
      - Spring Bean的循环依赖问题
        - 循环依赖是指两个或多个bean之间相互依赖，形成了一个循环引用的情况，会导致应用程序启动失败
        - 解决循环依赖的方式就是引入了三级缓存
          - singletonObjects是一级缓存，存储的是完整创建好的单例bean对象
          - earlySingletonObjects是二级缓存，存储的是尚未完全创建好的单例bean对象
          - singletonFactories是三级缓存，存储的是单例bean的创建工厂
        - 为啥能解决 — 当Spring容器创建一个Bean时，如果这个Bean有依赖其他的Bean
          - 容器首先尝试从一级缓存中获取所需的依赖Bean。
          - 如果依赖Bean不在一级缓存中，容器会继续检查二级缓存。
          - 如果依赖Bean的早期引用在二级缓存中，就使用这个早期引用来满足当前Bean的依赖需求
          - 如果依赖Bean连早期引用都还没有，容器会尝试获取二级缓存中的Bean工厂对象，使用这个工厂对象创建Bean的早期引用，并将其存放到二级缓存中然后使用这个早期引用来满足当前Bean的依赖需求。
          - 容器继续完成当前Bean的剩余初始化工作，并将完全初始化好的Bean存放到一级缓存中
        - Spring解决循环依赖是有一定限制的
          - 首先就是要求互相依赖的Bean必须要是单例的Bean，多例没有三级缓存
          - 另外就是依赖注入的方式不能都是构造函数注入的方式
      - Spring 中的 Bean 是线程安全的吗
        - Spring的Bean是否线程安全，这个要取决于他的作用域
        - 默认情况下，Spring Bean 是单例的（Singleton），这意味着在整个 Spring 容器中只存在一个 Bean 实例如果将 Bean 的作用域设置为原型的（Prototype），那么每次从容器中获取 Bean 时都会创建一个新的实例
          - 对于Prototypei这种作用域的Bean，他的Bean 实例不会被多个线程共享，不存在线程安全的问题
          - 对于Singleton的Bean，可能存在线程安全问题，要看这个Bean中是否有共享变量
        - Spring 中的 Bean 作用域有哪些
          - Spring的Bean的作用域，描述的就是这个Bean在哪个范围内可以被使用，不同的作用域决定了了 Bean 的创建、管理和销毁的方式
          - 常见的作用域有
            - 单例（Singleton）：
              - 默认作用域。
              - 对于整个 Spring IoC 容器，只创建一个 Bean 实例。
              - 适用于无状态和全局的对象。
            - 原型（Prototype）：
              - 每次获取都会创建一个新的 Bean 实例。
              - 适用于有状态的对象或者需要频繁创建的对象。
            - 请求（Request）：
              - 仅在 Web 应用程序中有效。
              - 每个 HTTP 请求都会创建一个新的 Bean 实例。
              - 用于请求级别的数据共享和处理。
            - 会话（Session）：
              - 仅在 Web 应用程序中有效。
              - 每个 HTTP 会话都会创建一个新的 Bean 实例。
              - 适用于会话级别的数据共享和状态管理。
            - 应用（Application）：
              - 仅在 Web 应用程序中有效。
              - 在 ServletContext 的生命周期内，只创建一个 Bean 实例。
              - 适用于全应用级别的数据共享和资源管理。
            - WebSocket：
              - 仅在 Web 应用程序中有效。
              - 在 WebSocket 的生命周期内，只创建一个 Bean 实例。
              - 适用于WebSocket长期的交互场景。
    - 控制反转Inversion of Control (IoC)
      - 定义 — 应用程序代码从Ioc Container中获取依赖的Bean，注入到应用程序中的这个过程
      - 和Ioc关系 — Ioc是通过DI实现的，其实它们是同一个概念的不同角度描述。通俗来说就是IoC是设计思想，DI是实现方式
      - 依赖注入(Dependency Injection，DI)
        - setter方式
          - 在XML配置方式中，property都是setter方式注入
            - 第一步，需要new UserServiceImpl()创建对象，所以需要默认构造函数
            - 第二步，调用setUserDao()函数注入userDao的值，所以需要setUserDao()函数
          - 在注解和Java配置方式下
        - 构造方法注入（Construct注入） — 为什么推荐构造器注入方式
          - 通过final保证依赖不可变
          - 确保需要的依赖不为空（省去了我们对其检查）
          - 总是能够在返回客户端（组件）代码的时候保证完全初始化的状态
        - DI的三种方式
          - 注解注入
            - 常见注解为@Autowired和@Resource以及@Inject
              - 以@Autowired（自动注入）注解注入为例
                - 修饰符有三个属性：Constructor，byType，byName。默认按照byType注入
                - 将@Autowired写在被注入的成员变量上，setter或者构造器上，就不用再xml文件中配置了
              - 三者区别
                - @Autowired是Spring自带的，@Resource是JSR250规范实现的，@Inject是JSR330规范实现的
                - @Autowired、@Inject用法基本一样，不同的是@Inject没有required属性
                - @Autowired、@Inject是默认按照类型匹配的，@Resource是按照名称匹配的
                - @Autowired如果需要按照名称匹配需要和@Qualifier一起使用，@Inject和@Named一起使用，@Resource则通过name进行指定
            - Autowired和Resource的关系
              - 都可以将bean注入inject到对应的(corresponding) field中
              - 作用域不同
                1. Autowired可以作用在构造器，字段field，setter方法上
                2. Resource 只可以使用在field，setter方法上
              - 支持方不同
                1. Autowired是Spring提供的自动注入注解annotation，只有Spring容器会支持，如果做容器迁移，是需要修改代码的
                2. Resource是JDK官方提供的自动注入注解（JSR-250），它等于说是一个标准或者约定，所有的IOC容器都会支持这个注解。
              - byName和byType匹配顺序matching order不同 — Autowired在获取bean的时候，先是byType的方式，再是byName的方式
                Resource在获取bean的时候，和Autowired恰好相反(just opposite)，先是byName方式，然后再是byType方式
    - 面向切面编程Aspect Oriented Programming (AOP)
      - 定义 — Spring 框架通过定义切面, 通过拦截切点实现了不同业务模块的解耦 — AOP的本质也是为了解耦，它是一种设计思想
      - 如何理解
        - OOP面向对象编程，针对业务处理过程的实体及其属性和行为进行抽象封装，以获得更加清晰高效的逻辑单元划分
        - AOP则是针对业务处理过程中的切面进行提取，它所面对的是处理过程的某个步骤或阶段，以获得逻辑过程的中各部分之间低耦合的隔离效果
      - Spring框架是如何实现AOP的（AOP的配置方式）
        - 对XML模式
          - 定义目标类
          - 定义切面类
          - XML配置AOP
          - 测试类
        - AspectJ注解方式
          - AspectJ是一个java实现的AOP框架，它能够对java代码进行AOP编译（一般在编译期进行），让java代码具有AspectJ的AOP功能（当然需要特殊的编译器）
          - 使用了@AspectJ框架为AOP的实现提供了一套注解
          - 接口使用JDK代理
            - 定义接口
            - 实现类
            - 定义切面
            - 输出
          - 非接口使用Cglib代理
            - 类定义
            - 切面定义
            - 输出
        - Spring AOP和AspectJ是什么关系
          - AspectJ是更强的AOP框架，是实际意义的AOP标准；
          - 互补而不是竞争的关系
          - AOP在运行时仍旧是纯的Spring AOP，并不依赖于AspectJ的编译器或者织入器（weaver）
    - @Service 和 @Controller 互换可以吗？
      - 在Spring框架中，@Service和@Controller这两个注解并不是可以随意互换的。这两个注解都有它们特定的用途：
      - @Service：用于标注服务层的组件。服务层通常包含业务逻辑，处理数据访问层返回的数据，或者将数据传递给数据访问层进行存储。这个注解表明这是一个服务类，Spring会把它当作一个Bean进行管理。
      - @Controller：用于标注控制层的组件。控制层主要负责处理用户请求，调用服务层的方法，并返回响应给用户。在Spring MVC中，这个注解还常与@RequestMapping等注解一起使用，以定义URL路由和请求处理方法。
      - 互换这两个注解可能会导致以下问题：
      - 如果把@Service换成@Controller，那么该类可能会被错误地视为控制器，而实际上它可能不包含处理HTTP请求的逻辑。
      - 如果把@Controller换成@Service，那么该类可能不会被Spring MVC正确地识别为控制器，从而导致请求处理失败。
      - 因此，这两个注解不应该互换使用。
    - Spring中的事务
      - Spring中如何开启事务
        - 编程式事务 — 基于底层的API，如PlatformTransactionManager、TransactionDefinition 和 TransactionTemplate 等核心接口，开发者完全可以通过编程的方式来进行事务管理。
          - 编程式事务方式需要是开发者在代码中手动的管理事务
          - 另外的开启、提交、回滚等操作
        - 声明式事务 — 声明式事务管理方法允许开发者配置的帮助下来管理事务，而不需要依赖底层API进行硬编码。开发者可以只使用注解或基于配置的 XML 来管理事务
      - @Transaction 事务如何使用的，回滚操作如何实现的？
        - 在Spring中，@Transactional注解用于声明事务边界。它可以被添加到类或方法上，以指定该方法或类中的所有公共方法都应该在事务中执行。
        - 使用@Transactional的基本步骤如下：
          - 在Spring配置中启用事务管理，这通常通过配置一个事务管理器来完成。
          - 在需要执行事务的方法或类上添加@Transactional注解。
          - 在方法中执行数据库操作。
          - 如果事务中的操作因为某些原因（如异常）而失败，Spring会自动回滚该事务，即撤销事务中所做的所有更改，以确保数据的一致性。
          - 关于回滚操作，它是自动发生的，不需要显式调用。当在@Transactional注解的方法中抛出非检查异常（即运行时异常）时，Spring会默认触发事务回滚。如果需要在检查异常时也回滚事务，可以通过设置@Transactional(rollbackFor = Exception.class)来实现。
          - 另外，@Transactional注解还提供了其他属性，如propagation（传播行为），isolation（隔离级别），readOnly（只读），timeout（超时时间）等，以便更精细地控制事务的行为。
      - Spring的事务传播机制
        - REQUIRED，如果不存在事务则开启一个事务，如果存在事务则加入之前的事务，总是只有一个事务在执行
        - REQUIRES_NEW，每次执行新开一个事务
        - SUPPORTS，有事务则加入事务，没有事务则普通执行
        - NOT_SUPPORTED，有事务则暂停该事务，没有则普通执行
        - MANDATORY，强制有事务，没有事务则报异常
        - NEVER，有事务则报异常
        - NESTED，如果之前有事务，则创建嵌套事务，嵌套事务回滚不影响父事务，反之父事务影响嵌套事务
  - 核心
    - 初始化的主体流程
      - 初始化BeanFactory之obtainFreshBeanFactory
      - 初始化BeanFactory之loadBeanDefinitions
      - AbstractBeanDefinitionReader读取Bean定义资源
      - XmlBeanDefinitionReader加载Bean定义资源
      - DocumentLoader将Bean定义资源转换为Document对象
      - XmlBeanDefinitionReader解析载入Bean定义资源文件
      - DefaultBeanDefinitionDocumentReader对Bean定义的Document对象解析
      - BeanDefinitionParserDelegate解析Bean定义资源文件生成BeanDefinition
      - 解析过后的BeanDefinition在IoC容器中的注册
      - DefaultListableBeanFactory将Ioc容器注册解析后的BeanDefinition
    - Springboot
      - 和Spring的关系
        - Spring Boot 是 Spring Framework 的扩展：Spring Boot 是 Spring 项目的子项目，旨在简化 Spring 应用的开发和部署
        - 基于 Spring — 利用 Spring 的核心功能（如 IOC、AOP 等），并在此基础上提供自动配置和开箱即用的特性
        - 增强和简化 — Spring Boot 通过自动配置、嵌入式服务器、Starter POMs 等特性增强了 Spring 的功能，简化了开发过程
      - Spring MVC
        - 什么是SpringMVC — Spring MVC是Spring在AOP等技术基础上，遵循上述Web MVC的规范推出的web开发框架，目的是为了简化Java栈的web开发
        - 什么是MVC — 是一种基于Java 的实现了Web MVC 设计模式的请求驱动类型的轻量级Web 框架，将 web 层进行职责解耦，基于请求驱动指的是使用请求-响应模型
          - MVC英文是Model View Controller，是模型(model) - 视图(view) - 控制器(controller)的缩写，一种软件设计规范
            - Model（模型）是应用程序中用于处理应用程序数据逻辑的部分。通常模型对象负责在数据库中存取数据
            - View（视图）是应用程序中处理数据显示的部分。通常视图是依据模型数据创建的
            - Controller（控制器）是应用程序中处理用户交互的部分。通常控制器负责从视图读取数据，控制用户输入，并向模型发送数据
          - 用一种业务逻辑、数据、界面显示分离的方法，将业务逻辑聚集到一个部件里面，在改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑
      - SpringBoot 框架都做了什么来简化你的开发过程？
        - 自动配置：SpringBoot可以根据项目的类路径、构建方式和其他条件，自动配置并创建Spring容器中的Bean。这减少了开发人员需要手动进行的大量配置工作，使得项目搭建和配置变得更加快速和简单。
        - 内嵌服务器：SpringBoot应用程序可以直接打包成可执行的Jar包，内嵌了如Tomcat、Jetty或Undertow等服务器。这样，开发人员无需将应用程序部署到外部的Web容器，就可以直接运行测试，极大地简化了开发和测试流程。
        - Starter依赖：SpringBoot提供了一系列预配置的Starter依赖包，这些依赖包含了相关的库和框架的依赖。例如，Spring Boot Starter Web就包含了开发Web应用所需的所有依赖，如Spring MVC。这避免了开发人员需要手动导入和管理大量的依赖。
        - 简化配置：SpringBoot使用YAML格式进行配置，这种格式简洁明了，易于理解和维护，进一步降低了配置的复杂性。
        - 外部化配置：SpringBoot支持将配置文件放置在外部位置，如服务器上的目录或环境变量中，这使得配置更加灵活，避免了将配置硬编码到代码中。
      - SpringBoot的自动配置底层实现 — 它主要依赖于SpringFactoriesLoader类和@EnableAutoConfiguration注解。在启动时，SpringBoot会根据项目中添加的jar包、类路径中的jar包以及各种属性设置等来判断当前需要什么样的配置，并自动加载和配置相应的Bean
      - SpringBoot是怎么做到打包部署的？ — SpringBoot通过Maven或Gradle等构建工具进行打包部署。以Maven为例，通常会在项目的pom.xml文件中添加spring-boot-maven-plugin插件。这个插件会辅助对项目进行打包，生成一个可执行的jar包。这个Jar包包含了应用程序的所有依赖以及内嵌的服务器，因此可以直接运行。
        部署时，只需在服务器上执行java -jar target/jar包名称.jar命令即可启动应用程序。
      - 和Spring的区别
        - 配置复杂度：
          - Spring: — 需要手动配置大量的 XML 或 Java 类，配置过程复杂。
          - Spring Boot: — 灵活性高，但需要开发者具有较深的配置知识。
          - 提供自动配置，极大简化了配置过程。
          - 约定优于配置（Convention over Configuration），减少了手动配置的需求。
        - 启动和部署：
          - Spring: — 通常需要在外部应用服务器（如 Tomcat、JBoss）中部署。
          - 需要打包成 WAR 文件并部署到服务器中。
          - Spring Boot: — 提供嵌入式服务器（如 Tomcat、Jetty），可以直接运行 JAR 文件启动应用。
          - 支持更简便的部署和测试。
        - 依赖管理：
          - Spring: — 手动管理依赖，需要明确添加每个模块的依赖。
          - Spring Boot: — 提供 Starter POMs（启动器依赖），简化了依赖管理，自动引入相关依赖。
        - 开发和测试：
          - Spring: — 需要编写大量的配置代码和样板代码。
          - 测试配置相对复杂。
          - Spring Boot: — 减少了样板代码，快速开发和测试更为便捷。
          - 提供许多默认配置和最佳实践，支持更高效的开发。
        - 功能扩展：
          - Spring: — 提供核心的企业级应用功能，如数据访问、事务管理、MVC 等。
          - Spring Boot: — 在 Spring 的基础上提供更多的扩展功能，如监控、外部化配置、内嵌服务器等。
        - 适用场景：
          - Spring: — 适用于复杂、大型企业级应用，需要精细控制和自定义配置。
          - Spring Boot: — 适合快速开发和中小型应用，强调快速上手和简洁配置。
      - Spring Boot 配置过程
        - 项目初始化
          - 使用 Spring Initializr — 网站生成项目
          - 手动创建 — 选择构建工具、Spring Boot 版本、项目元数据、依赖
          - 创建 Maven 或 Gradle 项目
          - 添加 Spring Boot Starter 依赖
        - 项目结构 — 目录结构
          - src/main/java
          - src/main/resources
          - src/test/java
          - 构建文件（pom.xml 或 build.gradle）
        - 主应用程序类 — 包含 @SpringBootApplication 注解
          - 启动应用程序
        - 配置文件 — application.properties 或 application.yml
          - 设置服务器端口
          - 数据源配置
        - 自动配置 — 依赖注入和自动配置 — 使用 @Autowired 注解
        - 自定义配置 — 创建自定义配置类 — 使用 @Configuration 注解
        - Profile 配置
          - 多环境配置 — application-{profile}.properties 或 application-{profile}.yml
          - 指定激活的 Profile — 设置 spring.profiles.active
        - 运行和测试
          - 运行应用 — 运行主类中的 main 方法
          - 使用 Maven 或 Gradle 命令
          - 测试 — 使用 @SpringBootTest 注解进行集成测试
        - 部署
          - 打包应用 — 使用 Maven 或 Gradle 命令打包
          - 运行 JAR 文件 — 使用 java -jar 命令
      - Springboot核心注解
        - @SpringBootApplication：
          - 这是Spring Boot的最核心注解，用在Spring Boot主类上，标识这是一个Spring Boot应用。
          - 它实际上是@SpringBootConfiguration、@EnableAutoConfiguration、@ComponentScan这三个注解的组合。
        - @EnableAutoConfiguration：
          - 允许Spring Boot自动配置注解，开启后，Spring Boot能根据当前类路径下的包或者类来自动配置Spring Bean。
          - 例如，如果类路径下有Mybatis的JAR包，MybatisAutoConfiguration就能根据相关参数来自动配置Mybatis的各个Spring Bean。
        - @Configuration：
          - 用于定义一个配置类，可替换传统的applicationContext.xml配置文件。
          - 配置类中可以使用@Bean注解来定义Bean。
        - @SpringBootConfiguration：
          - 这个注解是@Configuration的一个特化，用于标识一个类是Spring Boot的配置类。
          - 通常用于修饰Spring Boot的主类，或者任何需要特定于Spring Boot配置的场景。
        - @ComponentScan：
          - 用于开启组件扫描，即自动扫描包路径下的@Component注解，并将它们注册为Spring的Bean。
          - 这可以简化Bean的注册过程，避免在配置文件中逐个声明。
        - @Conditional：
          - 用于标识一个Spring Bean或者Configuration配置文件，在满足特定条件时才进行配置。
          - 需要与Condition接口一起使用，通过实现该接口来定义满足何种条件时激活配置。
          - 除了上述核心注解外，Spring Boot还有许多其他有用的注解，如@RestController（用于创建RESTful Web服务），@Autowired（用于自动装配Bean）等。这些注解大大简化了Spring应用的开发和配置过程。
      - Springboot配置过程
        - application.properties 或 application.yml：
          - 这两个文件是Spring Boot项目的核心配置文件。你可以在其中配置各种属性，如数据库连接信息、服务器端口、日志级别等。例如，要修改服务访问端口，你可以在application.properties文件中添加或修改server.port=8081这样的配置。
        - pom.xml 或 build.gradle（如果使用Gradle构建）：
          - 这些是项目的构建配置文件。在这里，你可以添加项目依赖、配置插件等。例如，如果你想在项目中使用某个库，你需要在这些文件中添加相应的依赖。
        - 环境变量配置：
          - 有时，你可能还需要配置环境变量，特别是当你需要在不同的环境中使用不同的配置时（如开发环境、测试环境和生产环境）。这通常涉及到设置系统环境变量或使用Spring Boot的配置文件来指定不同环境的配置。
        - 日志配置：
          - 如果需要自定义日志行为，如日志级别、输出格式或输出目的地等，你可以修改或创建logback.xml、log4j2.xml等日志文件进行配置。
        - 数据源配置：
          - 如果项目涉及数据库操作，你需要在配置文件中指定数据源的相关配置，如数据库URL、用户名、密码等。
        - 安全性配置：
          - 对于需要安全性控制的项目，你可能还需要在配置文件中指定安全相关的设置，如CORS策略、CSRF保护、会话管理等。
  - 前端
    - 前端后端的数据传输
      - 前端： — 使用 Flutter 进行开发，用户在前端填表提交数据。
      - 通过 HTTP POST 请求，将数据传输到后端的 RESTful API。
      - 项目配置： — Spring Boot 应用启动配置 application.properties 或 application.yml，示例配置文件 application.properties：
      - 数据接收与处理
        - Controller 层： — 使用 @RestController 处理前端请求。
        - FormData 类： — 用于接收前端发送的数据。
      - 业务逻辑层 (Service) — Service 层： — 处理业务逻辑，调用数据访问层保存数据。
      - 示例代码：
      - 数据访问层 (Repository)
        - Repository 层： — 使用 Spring Data JPA 进行数据库操作。
        - FormDataEntity 类：
          - 映射到数据库中的表
          - FormDataEntity类映射到数据库中的表结构，包含用户ID、用户名、电子邮件、表单内容和提交时间等字段
      - 数据缓存 (Redis) — 使用 Redis 进行缓存：
        - 缓存配置： — 配置 RedisTemplate 和缓存管理器。
        - 缓存使用： — 在 Service 层引入 Redis 进行数据缓存。
      - 异步任务处理 (Kafka)
        - Kafka 配置： — 配置 Kafka 生产者和消费者。
        - 生产者发送消息： — 在需要处理异步任务时发送 Kafka 消息。
        - 消费者处理消息： — 处理 Kafka 消息的消费者
      - 数据库一致性 (Canal + RabbitMQ)
        - Canal 配置：
          - 用于监听 MySQL 数据库变更。
          - Canal 监听并发送消息： — 监听数据库变更并发送消息到 RabbitMQ
        - RabbitMQ 配置：
          - 配置 RabbitMQ 连接和队列。
          - RabbitMQ 消费者： — 从队列中消费消息并处理
        - 配置 RabbitMQ 连接和队列。
      - 数据库一致性 (Canal + RabbitMQ)
    - Kafka是在项目中是如何使用的？
      - 在番茄Plus项目中，Kafka的使用场景主要集中在处理用户设置的日期提醒功能。同时，通过日志收集功能，方便故障排查和性能监控。 — 通过Kafka的事件流处理，确保提醒消息的可靠传递和顺序处理，实现异步任务处理。避免阻塞主线程，提高系统的响应速度和性能。
      - 用户设置提醒
        - 用户通过前端界面设置提醒，包括提醒的时间和内容。
        - 前端操作：用户在前端应用中输入提醒时间和内容，点击保存按钮。
        - 发送请求：前端通过RESTful API将提醒请求发送到后端。
      - 2. 后端接收请求
        - Spring Boot应用中的REST控制器接收到提醒请求，并进行相应的处理：
        - 验证请求数据：验证用户的权限以及提醒时间的合法性等。
        - 持久化数据：将提醒的详细信息存储在数据库中。
        - 生成提醒事件：创建一个包含提醒ID、用户ID、提醒时间和提醒内容的提醒事件消息。
      - 3. 发布到Kafka
        - 使用Spring Kafka的模板（KafkaTemplate）将事件消息发送到Kafka的特定主题。
        - 设置消息key：消息的key可以是提醒ID，以保证相同提醒事件消息被顺序处理。
        - 发送消息：将提醒事件消息发送到Kafka主题。
      - 4. 消费者服务
        - 创建一个Kafka消费者服务，订阅提醒事件的Kafka主题，并处理提醒事件消息。
        - 订阅主题：消费者服务使用@KafkaListener注解订阅并处理Kafka消息。
        - 处理即时提醒：对于需要即时发送的提醒，通过HTTP响应直接发送提醒。
        - 安排定时任务：对于未来的提醒，安排定时任务，在提醒时间到达时标记为待发送。
      - 5. 消息处理
        - 根据不同的提醒类型（即时或未来）进行处理：
        - 即时提醒：直接通过HTTP响应发送提醒给用户。
        - 未来提醒：安排定时任务，在提醒时间接近时标记为待发送。
      - 6. 处理结果反馈
        - 提醒发送后，更新数据库中的提醒状态为已发送，并记录处理结果到日志系统。
        - 更新数据库：将提醒状态更新为已发送。
        - 记录日志：记录处理结果，以便后续分析和排查问题。
      - 7. 定期查询未发送提醒
        - 客户端定期发送HTTP请求，轮询检查是否有新的提醒。
        - 轮询请求：客户端定期向服务器发送请求，查询待发送的提醒。
        - 查询数据库：服务器查询数据库，返回待发送的提醒列表。
      - 8. 提醒触发
        - 更新提醒状态：调度任务触发时，将提醒状态更新为待发送。
        - 返回提醒数据：客户端轮询请求时，返回待发送的提醒，客户端显示提醒。
      - 9. 前端显示提醒
        - 客户端在接收到提醒数据时，更新界面显示提醒。
        - 显示提醒：在前端界面上显示提醒通知，可以考虑使用浏览器通知或声音提醒。
        - 用户交互：用户可以对提醒进行相应的操作，如标记为已读、再次提醒等