课程主题

Spring整体介绍&手写实现IoC模块的V1和V2版本

课程目标

1. 搞清楚spring全家桶包含哪些常用的技术？
2. 搞清楚spring框架是主要担任的职责是什么？
3. 搞清楚spring框架和springboot的区别？
4. 搞清楚spring框架中每个模块的作用？
5. 搞清楚spring中的核心概念，比如基础容器、高级容器、ioc容器、BeanFactory&FactoryBean、ioc、aop、di、bop、循环依赖等
6. 可以实现ioc模块手写V2版本

课程内容

一、spring整体介绍

```
* spring全家桶
    * spring是一个轻量级一站式企业开发解决方案。
        * spring framework（就是我们ssm中的spring）：基石
            * ioc、aop
            * springmvc：针对浏览器/服务器这种BS结构开发的框架。
            * spring JdbcTemplate：针对底层数据库进行CRUD操作的框架。
        * spring boot：尽量减少框架本身的学习成本以及开发成本，而让程序员更多的去关注和开发业务代码。
            * 纯注解
            * 零配置（无spring配置文件）
            * 自动装配
            * 起步依赖
            * 内嵌Tomcat、Jetty服务
            * 可以以jar方式去部署启动

        * spring cloud：分布式服务/微服务一站式开发框架
        * spring data：解决的是数据交互（MySQL、redis、elasticsearch、mongodb、ORM等）
        * spring security(oauth2)：主要针对登录安全、权限认证等
        * spring oauth2：解决单点登录、用户鉴权等问题
    * spring framework
        * 认识spring framework六大模块集合，重点要理解核心容器模块和AOP&AspectJs模块
        * 参考架构图
    * 核心概念
        * OOP：面向对象编程
        * EJB：Spring就是为了替代EJB
        * BOP：面向Bean的编程，在Spring体系里面，一切都是针对Bean这个东西进行设计和实现。
        * 容器：存储单例Bean的实例
```

```
            * 基础容器：BeanFactory
            * 高级容器：ApplicationContext
        * 基础容器和高级容器的区别
            * 基础容器产生Bean实例的时机，是第一次被调用的时候，才产生
            * 高级容器是服务启动的时候，就会创建所有单例的bean实例。

        * BeanFactory和FactoryBean的区别
            * BeanFactory是spring中的顶级接口，是spring中容器的底层实现接口。BeanFactory实现类管理着所有的单例bean实例。
            * FactoryBean是spring容器中管理的一个特殊的bean。因为该bean还能产生指定类型的bean实例。
        * BeanFactoryPostProcessor和BeanPostProcessor的区别
            * BeanFactoryPostProcessor主要是在Bean【实例创建】之前，BeanDefinition初始化前后去调用。
            * BeanPostProcessor主要是在Bean【实例初始化】前后，对Bean实例进行一些后置操作。

        * IoC：控制反转，指的是创建Bean实例的角色发生了反转，由程序员主动new变为框架为你new，它离不开DI
        * DI：依赖注入，就是设置成员变量。
        * AOP：面向切面编程，和OOP一样都是一种开发思想。
            * Spring AOP
            * AspectJ
            * Spring 整合 AspectJ

        * 循环依赖
            * 先搞清楚依赖注入有几种方式（setter和构造方法）
            * A-->B B-->A形成闭环
            * 构造方法的循环依赖（无法解决的）
            * setter方法的循环依赖（Spring使用三级缓存技术解决）
```

二、手写ioc模块分析

2.1 需求

> 实现用户查询功能

- 业务层
  - UserService
  - UserServiceImpl
- 持久层
  - UserDao
  - UserDaoImpl
- PO层
  - User

2.2 V1版本实现

```
public class TestSpringV1 {
```

```
    @Test
    public void test() throws Exception {
        // 创建UserServiceImpl对象
        UserServiceImpl userService = new UserServiceImpl();
        // 创建UserDaoImpl对象
        UserDaoImpl userDao = new UserDaoImpl();
        // 创建BasicDataSource对象
        BasicDataSource dataSource = new BasicDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://111.231.106.221:3306/kkb");
        dataSource.setUsername("kkb");
        dataSource.setPassword("kkb111111");

        // 对UserDaoImpl对象依赖注入BasicDataSource实例
        userDao.setDataSource(dataSource);
        // 对UserServiceImpl对象依赖注入UserDaoImpl实例
        userService.setUserDao(userDao);

        // 入参对象
        Map<String, Object> param = new HashMap<>();
        param.put("username", "王五");
        // 根据用户名称查询用户信息
        List<User> users = userService.queryUsers(param);
        System.out.println(users);
    }
}

```

2.3 V2版本实现（面向过程）

思路分析

采取Map集合来存储单例Bean实例，通过beanName来获取该实例

- 需要通过xml来配置bean信息（直接借鉴于spring的xml配置）
  参考xml配置
  - 将XML中配置的每个Bean的信息封装到一个Java对象里面保存（BeanDefinition），最终将bean的名称和BeanDefinition对象封装到Map集合中

代码实现

```
public class TestSpringV2 {

    // K:BeanName
    // V:Bean实例对象
    private Map<String, Object> singletonObjects = new HashMap<String, Object>();
    // K:BeanName
    // V:BeanDefinition对象
    private Map<String, BeanDefinition> beanDefinitions = new HashMap<String, BeanDefinition>();

```

```
    @Test
    public void test() {
        // 从XML中加载配置信息，先完成BeanDefinition的注册
        registerBeanDefinitions();

        // 根据用户名称查询用户信息
        UserService userService = (UserService) getBean("userService");

        // 入参对象
        Map<String, Object> param = new HashMap<>();
        param.put("username", "王五");
        // 根据用户名称查询用户信息
        List<User> users = userService.queryUsers(param);
        System.out.println(users);
    }

    public void registerBeanDefinitions() {
        // XML解析，将BeanDefinition注册到beanDefinitions集合中
        String location = "beans.xml";
        // 获取流对象
        InputStream inputStream =
this.getClass().getClassLoader().getResourceAsStream(location);
        // 创建文档对象
        Document document = createDocument(inputStream);

        // 按照spring定义的标签语义去解析Document文档
        parseBeanDefinitions(document.getRootElement());
    }

    @SuppressWarnings("unchecked")
    public void parseBeanDefinitions(Element rootElement) {
        // 获取<bean>和自定义标签（比如mvc:interceptors）
        List<Element> elements = rootElement.elements();
        for (Element element : elements) {
            // 获取标签名称
            String name = element.getName();
            if (name.equals("bean")) {
                // 解析默认标签，其实也就是bean标签
                parseDefaultElement(element);
            } else {
                // 解析自定义标签，比如mvc:interceptors标签回去
                parseCustomElement(element);
            }
        }
    }

    @SuppressWarnings("unchecked")
    private void parseDefaultElement(Element beanElement) {
        try {
            if (beanElement == null)
                return;
            // 获取id属性
            String id = beanElement.attributeValue("id");

            // 获取name属性
            String name = beanElement.attributeValue("name");
            // 获取class属性
            String clazzName = beanElement.attributeValue("class");
```

```
            if (clazzName == null || "".equals(clazzName)) {
                return;
            }

            // 获取init-method属性
            String initMethod = beanElement.attributeValue("init-method");
            // 获取scope属性
            String scope = beanElement.attributeValue("scope");
            scope = scope != null && !scope.equals("") ? scope :
"singleton";

            // 获取beanName
            String beanName = id == null ? name : id;
            Class<?> clazzType = Class.forName(clazzName);
            beanName = beanName == null ? clazzType.getSimpleName() :
beanName;
            // 创建BeanDefinition对象
            // 此次可以使用构建者模式进行优化
            BeanDefinition beanDefinition = new BeanDefinition(clazzName,
beanName);
            beanDefinition.setInitMethod(initMethod);
            beanDefinition.setScope(scope);
            // 获取property子标签集合
            List<Element> propertyElements = beanElement.elements();
            for (Element propertyElement : propertyElements) {
                parsePropertyElement(beanDefinition, propertyElement);
            }

            // 注册BeanDefinition信息
            this.beanDefinitions.put(beanName, beanDefinition);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    private void parsePropertyElement(BeanDefinition beanDefination,
Element propertyElement) {
        if (propertyElement == null)
            return;

        // 获取name属性
        String name = propertyElement.attributeValue("name");
        // 获取value属性
        String value = propertyElement.attributeValue("value");
        // 获取ref属性
        String ref = propertyElement.attributeValue("ref");

        // 如果value和ref都有值，则返回
        if (value != null && !value.equals("") && ref != null &&
!ref.equals("")) {
            return;
        }

        /**
         * PropertyValue就封装着一个property标签的信息
         */
        PropertyValue pv = null;

```

```
        if (value != null && !value.equals("")) {
            // 因为spring配置文件中的value是String类型，而对象中的属性值是各种各样的，所以需要存储类型
            TypedStringValue typeStringValue = new TypedStringValue(value);

            Class<?> targetType = getTypeByFieldName(beanDefination.getClazzName(), name);
            typeStringValue.setTargetType(targetType);

            pv = new PropertyValue(name, typeStringValue);
            beanDefination.addPropertyValue(pv);
        } else if (ref != null && !ref.equals("")) {

            RuntimeBeanReference reference = new RuntimeBeanReference(ref);
            pv = new PropertyValue(name, reference);
            beanDefination.addPropertyValue(pv);
        } else {
            return;
        }
    }

    private Class<?> getTypeByFieldName(String beanClassName, String name) {
        try {
            Class<?> clazz = Class.forName(beanClassName);
            Field field = clazz.getDeclaredField(name);
            return field.getType();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }

    private void parseCustomElement(Element element) {
        // TODO Auto-generated method stub

    }

    private Document createDocument(InputStream inputStream) {
        Document document = null;
        try {
            SAXReader reader = new SAXReader();
            document = reader.read(inputStream);
            return document;
        } catch (DocumentException e) {
            e.printStackTrace();
        }
        return null;
    }

    private Object getBean(String beanName) {
        // 先从一级缓存中获取单例Bean的实例
        Object singletonObject = singletonObjects.get(beanName);

        if (singletonObject != null) {
            return singletonObject;
```

```
        }
        // 懒汉式，等你getBean的时候，并且singletonObjects没有该实例的时候，才去创建该实例
        // 当缓存中没有找到该Bean实例，则需要创建Bean，然后将该Bean放入一级缓存中
        // 要创建Bean，需要知道该Bean的信息（这个信息是配置到XML中的）

        // 根据beanName去beanDefinitions获取对应的Bean信息
        BeanDefinition beanDefinition = this.beanDefinitions.get(beanName);
        if (beanDefinition == null || beanDefinition.getClazzName() == null) {
            return null;
        }

        // 根据Bean的信息，来判断该bean是单例bean还是多例（原型）bean
        if (beanDefinition.isSingleton()) {
            // 根据Bean的信息去创建Bean的对象
            singletonObject = createBean(beanDefinition);
            // 将Bean的对象，存入到singletonObjects
            this.singletonObjects.put(beanName, singletonObject);
        } else if (beanDefinition.isPrototype()) {
            // 根据Bean的信息去创建Bean的对象
            singletonObject = createBean(beanDefinition);
        } else {
            // TODO 。。。
        }

        return singletonObject;
    }

    private Object createBean(BeanDefinition beanDefinition) {
        // TODO 第一步：new对象（初始化）
        // TODO 第二步：依赖注入
        // TODO
        // 第三步：初始化，就是调用initMethod指定的初始化方法，或者实现了InitializingBean接口的afterPropertiesSet方法----
        return null;
    }
}
```