一、Spring概述

1.1 什么是Spring

- Spring是于2003 年兴起的一个**full-stack轻量级的Java 开源框架**，由Rod Johnson创建
- Spring以**IOC**（控制反转）和**AOP** （面向切面编程）为核心
- Spring提供了**展现层**Spring MVC、**持久层**Spring JDBC、**业务层**事务管理等众多的企业级应用技术
- Spring还能整合开源世界众多的第三方框架和类库，逐渐成为使用最多的Java EE企业应用开源框架

1.2 Spring的优势

①、方便解耦，简化开发

通过Spring提供的IoC容器，我们可以将对象之间的依赖关系交由Spring进行控制，避免硬编码所造成的过度程序耦合。有了Spring，用户不必再为单实例模式类、属性文件解析等这些很底层的需求编写代码，可以更专注于上层的应用。

②、AOP编程的支持

通过Spring提供的AOP功能，方便进行面向切面的编程，许多不容易用传统OOP实现的功能可以通过AOP轻松应付。

③、声明式事务的支持

在Spring中，我们可以从单调烦闷的事务管理代码中解脱出来，通过声明式方式灵活地进行事务的管理，提高开发效率和质量。

④、方便程序的测试

可以用非容器依赖的编程方式进行几乎所有的测试工作，在Spring里，测试不再是昂贵的操作，而是随手可做的事情。例如：Spring对Junit4支持，可以通过注解方便的测试Spring程序。

⑤、方便集成各种优秀框架

Spring不排斥各种优秀的开源框架，相反，Spring可以降低各种框架的使用难度，Spring提供了对各种优秀框架（如Struts、Hibernate、Hessian、Quartz）等的直接支持。

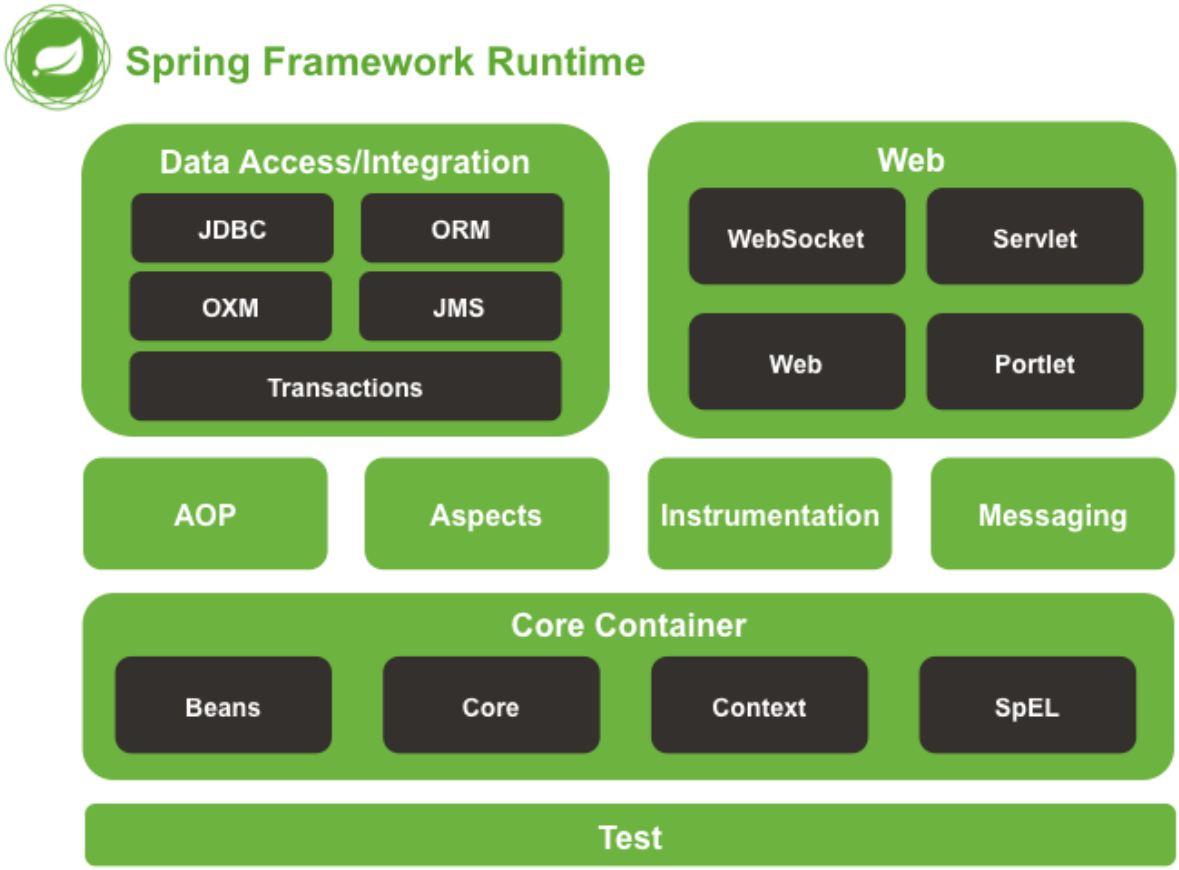
⑥、降低Java EE API的使用难度

Spring对很多难用的Java EE API（如JDBC，JavaMail，远程调用等）提供了一个薄薄的封装层，通过Spring的简易封装，这些Java EE API的使用难度大为降低。

⑦、Java 源码是经典学习范例

Spring的源码设计精妙、结构清晰、匠心独运，处处体现着大师对Java设计模式灵活运用以及对Java技术的高深造诣。Spring框架源码无疑是Java技术的最佳实践范例。

1.3 Spring的体系结构



面试直达:你能说出Spring是什么和它的主要功能吗?

二、认识IOC

IOC(控制反转)不是什么技术，而是一种设计思想。它的目的是指导我们设计出更加松耦合的程序。

- 控制：指的是控制权，在java中可以简单理解为 对象的控制权限(比如对象的创建、销毁等权限)
- 反转：指的是将对象的控制权由原来的 **程序员在类中主动控制** 反转到 **由Spring容器来控制。**

举个例子：找对象

- 传统方式：自己找，想要啥样的自己去大街上找（new）
- IOC方式 ： 婚介所，首先人们将自己的信息注册到婚介所。然后，等你想要对象的时候，婚介所就会帮你找到，然后给你送来。

以常见的service调用dao为例，说明IOC的核心思想

1. 分别提供dao和service接口以及其实现类

```
//dao接口
public interface UserDao {
    public void save();
}
//dao实现类
public class UserDaoImpl implements UserDao {
    @Override
    public void save() {
        System.out.println("保存成功了！");
    }
}
```

```
//service接口
public interface UserService {
    public void save() throws Exception;
}
//service实现类
public class UserServiceImpl implements UserService {
    private UserDao userDao;
    @Override
    public void save() throws Exception {
         userDao = new UserDaoImpl();
         userDao.save();
    }

}
```

代码问题：当前service对象和dao对象耦合度太高

2. 使用工厂解耦合

提供一个MyBeanFactory负责生产对象。

```
public static UserDao getBean(){
    return new UserDaoImpl();
}
```

修改service代码

```
public class UserServiceImpl implements UserService {

    private UserDao userDao;
    @Override
    public void save() throws Exception {
        userDao = MyBeanFactory.getBean();
        userDao.save();
    }

}
```

问题思考：这个工厂只能生产UserDao对象，是不是太low了?

3. 工厂升级

我们事先把所有要创建的对象都放入bean.xml配置文件中，让工厂一次全部创建出来，需要的时候直接拿。

```
<!--这里配置了所有需要创建的对象信息-->
<beans>
    <bean id="userDao" class="com.itheima.dao.impl.UserDaoImpl"></bean>
</beans>
```

```
//升级后的工厂对象
package com.itheima.factory;

import org.dom4j.Document;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;

import java.io.InputStream;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class MyBeanFactory {

    private static Map<String, Object> map = new HashMap<>();

    static {
        try {
            //把配置文件变成流
            InputStream is =
MyBeanFactory.class.getClassLoader().getResourceAsStream("beans.xml");
            //得到dom4j的操作对象
            SAXReader reader = new SAXReader();
            //把配置文件变成dom4j的文档对象
            Document document = reader.read(is);
            //获取根节点
            Element root = document.getRootElement();
            //得到根节点下所有的bean节点集合
            List<Element> list = root.elements("bean");
            for (Element element : list) {
                String id = element.attributeValue("id");
                map.put(id, Class.forName(element.attributeValue("class")).newInstance());
            }
        }catch (Exception e){
            e.printStackTrace();
        }
    }

    public static Object getBean(String beanId){

        return map.get(beanId);
```

```
    }
}
```

这时的工厂不仅可以生产任意对象，而且可以保证所有对象都是单例的，大大减轻了服务器压力！

4. IOC概念

其实升级后的MyBeanFactory就是一个简单的Spring的IOC容器所具备的功能。

通过一系列推导，现在可以告诉大家IOC的概念了：

IOC就是**控制反转**（Inversion of Control，缩写为**IoC**）。是面向对象编程中的一种设计原则，可以用来减低计算机代码之间的耦合度。

直白表述，就是之前我们需要一个对象是主动去new。而IOC思想则是让我们直接去IOC容器中去获取对象，此时对象的创建权已经反转给了IOC容器。

三、Spring入门案例(重点)

3.1 导入坐标

```
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.1.6.RELEASE</version>
        </dependency>
    </dependencies>
```

3.2 创建dao接口和实现类

```
//接口
public interface UserDao {
    public void save();
}
//实现类
public class UserDaoImpl implements UserDao {
    public UserDaoImpl() {
        System.out.println("对象创建了！");
    }
}
```

3.3 加入配置文件

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">

      <bean id="userDao" class="com.itheima.dao.impl.UserDaoImpl"/>

</beans>
```

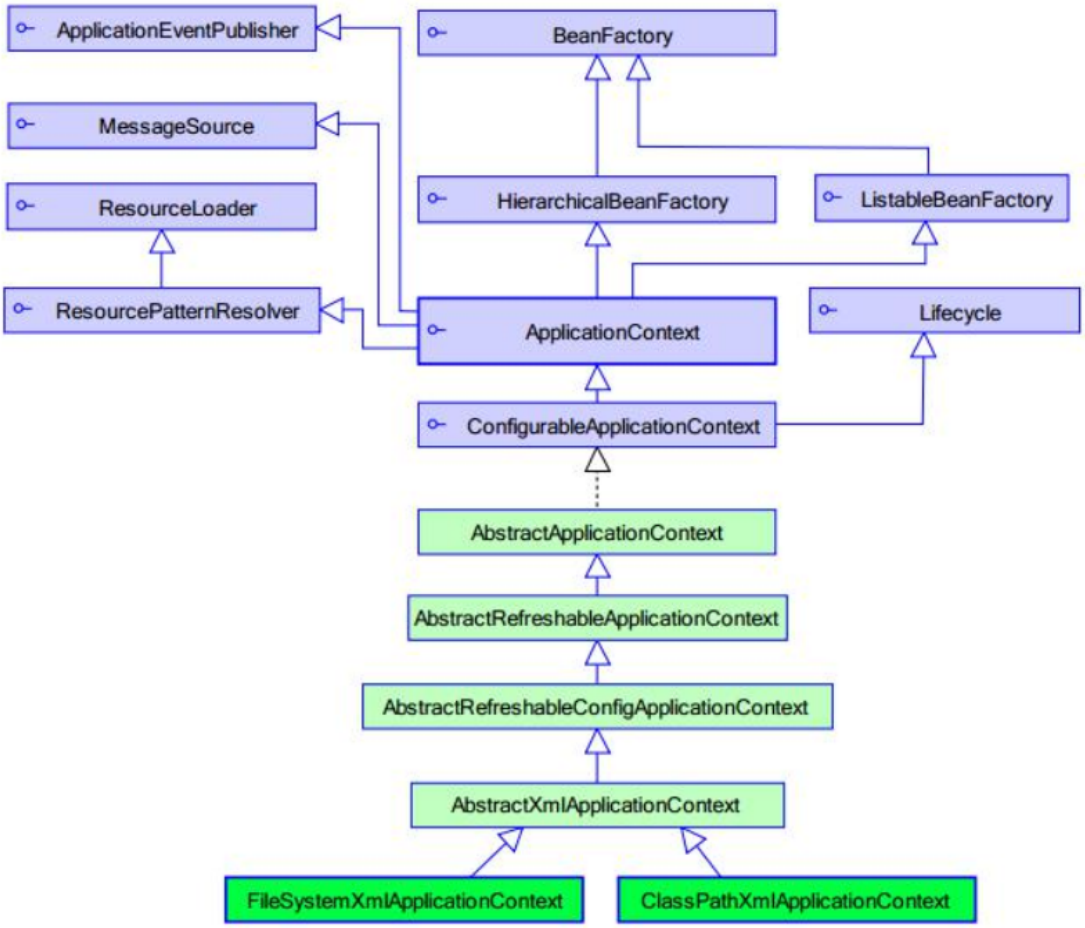
3.4 测试

```java
public class App {
    public static void main(String[] args) {

       //初始化Spring容器
       ApplicationContext applicationContext =
                new ClassPathXmlApplicationContext("applicationContext.xml");

        //从容器中获取对象
        UserDao userDao = (UserDao) ac.getBean("userDao");
        System.out.println(userDao);

    }
}
```

总结:Spring的IOC容器工作原理

1. 当Spring的IOC容器加载时，会读取配置文件中的诸多bean
2. 根据bean的class的值寻找对应的Class字节码文件，
3. 通过反射技术，创建出一个个对象，
4. 创建的对象会被存放到内部的一个Map结构中，等待被使用
5. 当我们需要使用具体的对象时就无须自己创建，而是直接从Spring的IOC容器中取。

四、spring的API介绍



Spring的API体系异常庞大，我们现在只关注两个BeanFactory和ApplicationContext:

BeanFactory

- BeanFactory是 Spring 的"心脏"。
- BeanFactory是 IOC 容器的核心接口，它定义了IOC的基本功能。
- Spring使用它来配置文档，管理bean的加载，实例化并维护bean之间的依赖关系，负责bean的声明周期。

ApplicationContext

- ApplicationContext由BeanFactory派生而来，可以比喻为Spring的躯体。
- ApplicationContext在BeanFactory的基础上添加了很多功能：
  - 支持了aop功能和web应用
  - MessageSource, 提供国际化的消息访问
  - 通过配置来实现BeanFactory中很多编码才能实现的功能
- ApplicationContext的常用实现类
  - ClassPathXmlApplicationContext：从classpath目录读取配置文件
  - FileSystemXmlApplicationContext：从文件系统或者url中读取配置文件
  - AnnotationConfigApplicationContext：当我们使用注解配置容器对象时，需要使用此类来创建 spring 容器。它用来读取注解。

两者区别:

- beanFactory主要是面向Spring框架的基础设施，也就是供spring自身内部调用，

  而Applicationcontext 主要面向Spring的使用者。

- BeanFactroy在第一次使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化，而ApplicationContext是在容器启动时，一次性创建并加载了所有的Bean。

五、创建Bean对象的不同方式

5.1 默认无参构造函数

它会根据默认无参构造函数来创建类对象。如果 bean 中没有默认无参构造函数，将会创建失败。

```
<bean id="userDao" class="com.itheima.dao.impl.UserDaoImpl"></bean>
```

5.2 工厂模式创建对象

在Spring中还可以通过工厂模式来创建对象。

```
工厂模式又分两种：
    静态工厂：不产生工厂的实例，直接调用工厂的静态方法创建对象。
    实例工厂：先产生工厂的实例，再调用工厂实例的方法创建对象。
```

使用工厂初始化对象，又可以根据工厂的性质分为使用静态工厂和实例化工厂，但是都很简单。

```
public class FactroyCreateBean {
    //静态工厂
    public static UserDao createUserDao(){
        return new UserDaoImpl();
    }
    //实例工厂
    public UserDao createUserDaoSimple(){
        return new UserDaoImpl();
    }
}
```

```
<!--使用静态工厂创建对象-->
<bean id="userDao1"
          class="com.itheima.factory.FactroyCreateBean" factory-method="createUserDao"/>

<!--使用实例工厂创建对象-->
<bean id="factroyCreateBean" class="com.itheima.factory.FactroyCreateBean"/>
<bean id="userDao2" factory-bean="factroyCreateBean" factory-method="createUserDaoSimple"/>
```

六、Bean对象的作用域和生命周期

所谓Bean的作用域其实就是指Spring给我们创建出的对象的存活范围。

可以在spring配置文件中通过bean标签中的scope属性来对当前bean对象的作用域进行指定。

```
<bean id="userDao" class="com.itheima.dao.impl.UserDaoImpl" scope="prototype"/>
```

scope属性有五个取值:

- singleton(默认)　创建出的实例为单例模式，IOC只创建一次，然后一直存在
- prototype　　创建出的实例为多例模式，每次获取bean的时候，IOC都给我们重新创建新对象
- request(web)　　web项目中,Spring 创建一个 Bean 的对象,将对象存入到 request 域中.
- session (web)　　web项目中,Spring 创建一个 Bean 的对象,将对象存入到 session 域中.
- globalSession (用于分布式web开发)　创建的实例绑定全局session对象

Bean的生命周期中的两个特殊方法

在这里所谓的Bean的生命周期其实指的是Bean创建到销毁的这么一段时间。

在Spring中可以通过配置的形式，指定bean在创建后和销毁前要调用的方法。

```
<!--通过init-method指定创建后要进行的动作,通过destroy-method指定销毁前要进行的动作-->
<bean id="userDao" class="com.itheima.dao.impl.UserDaoImpl" scope="prototype"
        init-method="init" destroy-method="destroy"/>
```

经测试得知:

- spring中单例对象的生命周期为:

  出生：IOC容器加载时出生

  存活：IOC容器运行时存活

  死亡：IOC容器销毁时死亡
- spring中多例对象的生命周期为:

  出生：使用对象时出生

  存活：一直存活

  死亡：由java垃圾回收器负责销毁

七、依赖注入

```
依赖注入：Dependency Injection。 它是 spring 框架核心 ioc 的具体实现。
我们的程序在编写时， 通过控制反转，把对象的创建交给了 spring，但是代码中不可能出现没有依赖的情况。
比如我们的Book中可能引入一个Publish类，在使用了Spring之后，它会为我们解决这些依赖对象的注入。
```

7.1 通过构造函数注入

```
public Book(String name, Float price, Date publishDate) {
      this.name = name;
      this.price = price;
      this.publishDate = publishDate;
}
```

```
<bean id="date" class="java.util.Date" />
<bean id="book" class="com.itheima.spring.Book">
      <!--
                index:指定参数在构造函数参数列表的索引位置，从0开始
                name:指定参数在构造函数中的名称
                type:指定参数在构造函数中的数据类型，可以通过反射拿到，不需要关系

                value:它能赋的值是基本数据类型和 String 类型
                ref:它能赋的值是其他 bean 类型，也就是说，必须得是在配置文件中配置过的 bean
       -->
        <constructor-arg name="name" value="小葵花" />
        <constructor-arg name="price" value="1" />
        <constructor-arg name="publishDate" ref="date" />
</bean>
```

7.2 通过setter属性注入

```
<bean id="book" class="com.itheima.spring.Book">
        <!--
            name:  找的是类中 set 方法后面的部分
            ref:   给属性赋值是其他 bean 类型的
            value: 给属性赋值是基本数据类型和string类型的
        -->
        <property name="name" value="小葵花" />
        <property name="price" value="1" />
        <property name="publishDate" ref="date" />
</bean>
```

做个扩展:p名称空间【了解即可】

```
 //xmlns:p="http://www.springframework.org/schema/p"
<bean id="book" class="com.itheima.ioc.Book" p:name="小葵花" p:price="1" />
```

7.3 注入集合属性

顾名思义，就是给类中的集合成员传值，它用的也是 set方法注入的方式，只不过变量的数据类型都是集合。 我们这里介绍注入数组、List、Set、Map、Properties。

```
<bean id="book" class="com.itheima.spring.Book">
        <!--List-->
        <property name="list">
```

```
            <list>
                <value>1</value>
                <value>2</value>
            </list>
        </property>

        <!--Set-->
        <property name="set">
            <set>
                <value>3</value>
                <value>4</value>
            </set>
        </property>

        <!--数组-->
        <property name="array">
            <array>
                <value>5</value>
                <value>6</value>
            </array>
        </property>

        <!--Map-->
        <property name="map">
            <map>
                <entry key="7" value="7-1" />
                <entry key="8" value="8-1" />
            </map>
        </property>

        <!--Properties-->
        <property name="properties">
            <props>
                <prop key="9">9-1</prop>
                <prop key="10">10-1</prop>
            </props>
        </property>
</bean>
```

八、配置文件模块化

我们现在的配置都集中配在了一个applicationContext.xml文件中，当开发人员过多时， 如果所有bean都配置到同一个配置文件中，会使这个文件巨大，而且也不方便维护。

针对这个问题，Spring提供了多配置文件的方式，也就是所谓的**配置文件模块化。**

配置文件模块化有两种形式:

1. 并列的多个配置文件

直接编写多个配置文件，比如说beans1.xml，beans2.xml……, 然后在创建ApplicationContext的时候，直接传入多个配置文件。

```
ApplicationContext act = new ClassPathXmlApplicationContext("beans1.xml","beans2.xml","...");
```

2. 主从配置文件

   先陪一个主配置文件，然后在里面导入其它的配置文件。

```
<import resource="beans1.xml" />
<import resource="beans2.xml" />
....
```

注意：

- 同一个xml文件中不能出现相同名称的bean,如果出现会报错
- 多个xml文件如果出现相同名称的bean，不会报错，但是后加载的会覆盖前加载的bean，所以企业开发中尽量保证bean的名称是唯一的。