

1. Spring

1.1. 什么是 spring

1、Spring 是一个开源框架，Spring 是于 2003 年兴起的一个轻量级的 Java 开发框架。

2、Spring 是一个一站式框架。

3、官方 jar 包下载网址：

<http://repo.springsource.org/libs-release-local/org/springframework/spring/>

轻量：Spring 是轻量级的，基本的版本大小为 2MB

控制反转：Spring 通过控制反转实现了松散耦合，对象们给出它们的依赖，而不是创建或查找依赖的对象们。

面向切面的编程 AOP:Spring 支持面向切面的编程，并且把应用业务逻辑和系统服务分开。

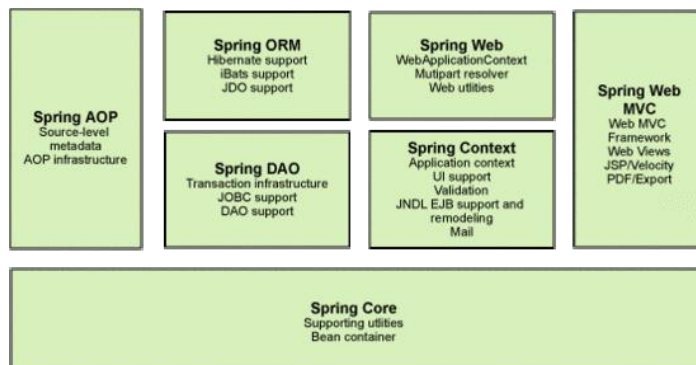
容器：Spring 包含并管理应用中对象的生命周期和配置

MVC 框架：Spring-MVC

事务管理:Spring 提供一个持续的事务管理接口,可以扩展到上至本地事务下至全局事务 JTA

异常处理：Spring 提供方便的 API 把具体技术相关的异常

1.2. spring 功能组成



Spring 由 7 个模块组成：

- Spring Core: 核心容器提供 Spring 框架的基本功能。核心容器的主要组件是 BeanFactory，它是工厂模式的实现。BeanFactory 使用控制反转（IOC）模式将应用程序的配置和依赖性规范与实际的应用程序代码分开。
- Spring 上下文：Spring 上下文是一个配置文件，向 Spring 框架提供上下文信息。Spring 上下文包括企业服务，例如 JNDI、EJB、电子邮件、国际化、校验和调度功能。
- Spring AOP: 通过配置管理特性，Spring AOP 模块直接将面向方面的编程功能集成到了 Spring 框架中。所以，可以很容易地使 Spring 框架管理的任何对象支持 AOP。Spring AOP 模块为基于 Spring 的应用程序中的对象提供了事务管理服务。通过使用 Spring AOP，不用依赖 EJB 组件，就可以将声明性事务管理集成到应用程序中。

- Spring DAO: JDBC DAO 抽象层提供了有意义的异常层次结构,可用该结构来管理异常处理和不同数据库供应商抛出的错误消息。异常层次结构简化了错误处理,并且极大地降低了需要编写的异常代码数量(例如打开和关闭连接)。Spring DAO 的面向 JDBC 的异常遵从通用的 DAO 异常层次结构。
- Spring ORM: Spring 框架插入了若干个 ORM 框架,从而提供了 ORM 的对象关系工具,其中包括 JDO、Hibernate 和 iBatis SQL Map。所有这些都遵从 Spring 的通用事务和 DAO 异常层次结构。
- Spring Web 模块: Web 上下文模块建立在应用程序上下文模块之上,为基于 Web 的应用程序提供了上下文。所以, Spring 框架支持与 Jakarta Struts 的集成。Web 模块还简化了处理多部分请求以及将请求参数绑定到域对象的工作。
- Spring MVC 框架: MVC 框架是一个全功能的构建 Web 应用程序的 MVC 实现。通过策略接口, MVC 框架变成高度可配置的, MVC 容纳了大量视图技术,其中包括 JSP、Velocity、Tiles、iText 和 POI。

1.3. Spring 容器

1. BeanFactory: (org.springframework.beans.factory.BeanFactory 接口定义)是最简单的容器,提供了基本的 DI 支持。最常用的 BeanFactory 实现就是 XmlBeanFactory 类,它根据 XML 文件中的定义加载 beans,该容器从 XML 文件读取配置元数据并用它去创建一个完全配置的系统或应用。

2. ApplicationContext 应用上下文: (org.springframework.context.ApplicationContext)基于 BeanFactory 之上构建,并提供面向应用的服务。

3. ApplicationContext 通常的实现

- ClassPathXmlApplicationContext: 从类路径下的 XML 配置文件中加载上下文定义,把应用上下文定义文件当做类资源。
- FileSystemXmlApplicationContext: 读取文件系统下的 XML 配置文件并加载上下文定义。
- XmlWebApplicationContext: 读取 Web 应用下的 XML 配置文件并装载上下文定义。

```
ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
```

1.3.1. BeanFactory 与 ApplicationContext 的区别

ApplicationContext 是由 BeanFactory 派生而来的,主要的实现是 ClassPathXmlApplicationContext, FileSystemXmlApplicationContext, XmlWebApplicationContext。面向的是框架开发者,几乎所有的应用都可以使用 ApplicationContext 而不是 BeanFactory。ClassPathXmlApplicationContext 默认从类路径下加载配置文件, FileSystemXmlApplicationContext 默认从系统文件中加载配置文件。XmlWebApplicationContext 默认从相对于 Web 根目录下加载配置文件 ApplicationContext 在初始化应用上下文容器时,会初始化所有单例 Bean。

BeanFactory 延迟加载 bean,当你使用 bean 的时候才去初始化,ApplicationContext 启动的容器的时候直接初始化。

1.4.IOC&DI

Inversion of Control, 一般分为两种类型: 依赖注入 DI(Dependency Injection)和依赖查找 (Dependency Lookup) .
依赖注入应用比较广泛。

Spring IOC 扶着创建对象, 管理对象 (DI), 装配对象, 配置对象, 并且管理这些对象的整个生命周期。

优点: 把应用的代码量降到最低。容器测试, 最小的代价和最小的侵入性使松散耦合得以实现。IOC 容器支持加载服务时的饿汉式初始化和懒加载。

DI 依赖注入是 IOC 的一个方面, 是个通常的概念, 它有多种解释。这概念是说你不用床架对象, 而只需要描述它如何被创建。你不在代码里直接组装你的组件和服务, 但是要在配置文件里描述组件需要哪些服务, 之后一个 IOC 容器辅助把他们组装起来。

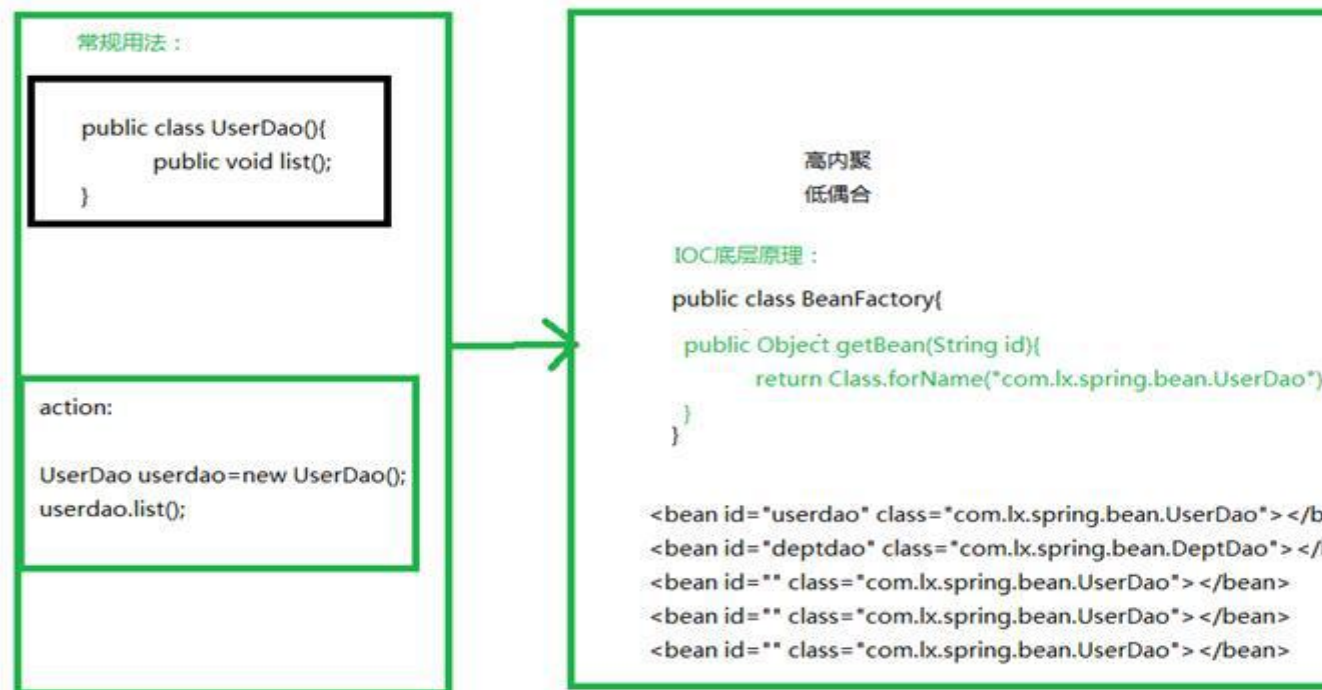
IOC 的注入方式: 1. 构造器依赖注入; 2. Setter 方法注入。

IOC (Inversion of Control)

DI(依赖注入, 建立在 IOC 的基础之上)

IOC (Inversion of Control)

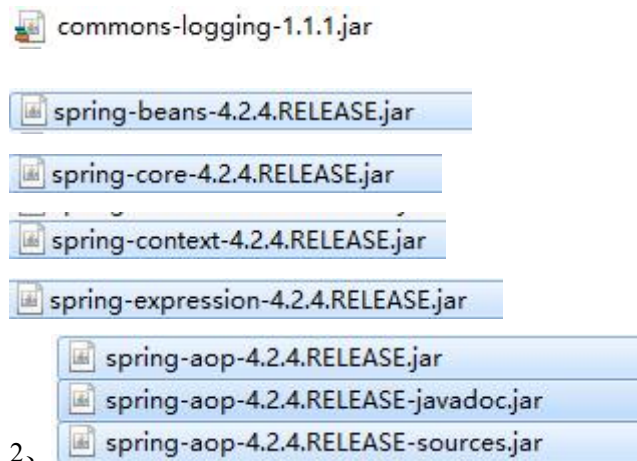
DI(依赖注入, 建立在 IOC 的基础之上)



1.5.IOC

1.5.1. 需要 jar 包

做 spring 基础功能的时候，只需要这 5 个核心 jar 包



1.5.2. 包的特征：

每一个包都有 3 个

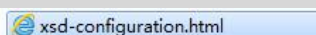


1.5.3. 核心配置文件

- 1、名字和位置 都不固定。
- 2、一般叫 applicationContext.xml，放在 src 下

1.5.3.1. xml 约束引用

spring-framework-4.2.4.RELEASE-dist\spring-framework-4.2.4.RELEASE\docs\spring-framework-reference\html



打开文件拉到最下面复制就可以。

```
<?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 http://www.springframework.org/schema/beans/spring-beans.xsd"

    <bean id="foo" class="x.y.Foo">
        <meta key="cacheName" value="foo"/>
        <property name="name" value="Rick"/>
    </bean>

</beans>
```

1.5.3.2. 约束参考

```
<?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
           http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="studentDao" class="com.hy.spring.dao.StudentDao" scope="singleton"></bean>

    <bean id="studentService" class="com.hy.spring.service.StudentService">
        <!-- 完成对你所需要的属性进行注入-->
        <property name="studentDao" ref="studentDao"></property>
    </bean>
</beans>
```

1.5.3.3. bean 配置（IOC）

```
<!-- id是唯一的，不能重复 -->
<bean id="userdao" class="com.Lx.spring.dao.UserDao"></bean>
```

1.5.3.4. bean 标签常用属性

Id

Name 也可以当做查找 bean 的标识，也要唯一。

class

init-method: Bean 实例化后会立刻调用的方法

destory-method: Bean 从容器移除和销毁前，会调用的方法

factory-method:运行我们调用一个指定的静态方法，从而代替构造方法来创建一个类的实例。

scope: Bean 的作用域，

① singleton

使用该属性定义 Bean 时，IOC 容器仅创建一个 Bean 实例，IOC 容器每次返回的是同一个 Bean 实例。

② prototype

使用该属性定义 Bean 时，IOC 容器可以创建多个 Bean 实例，每次返回的都是一个新的实例。

③ request

该属性仅对 HTTP 请求产生作用，使用该属性定义 Bean 时，每次 HTTP 请求都会创建一个新的 Bean，适用于 WebApplicationContext 环境。

④ session

该属性仅用于 HTTP Session，同一个 Session 共享一个 Bean 实例。不同 Session 使用不同的实例。

⑤ global-session

该属性仅用于 HTTP Session，同 session 作用域不同的是，所有的 Session 共享一个 Bean 实例。

autowired:自动装配 byName, byType, constructor, autodetect(首先阐释使用 constructor 自动装配，如果没有发现与构造器相匹配的 Bean 时，Spring 将尝试使用 byType 自动装配)

```
<bean id="useraction" class="com.lx.spring.action.UserAction" scope="prototype">
  <property name="service" ref="userservice"></property>
</bean>
```

1.5.3.5. Springbean 怎么创建的

反射模式（最常见的方式）

工厂方法模式(本文重点)

Factory Bean 模式

1.5.3.6. 加载核心配置文件

```
ApplicationContext app=new ClassPathXmlApplicationContext("applicationContext.xml");
UserDao dao=(UserDao)app.getBean("userdao");
dao.say();
```

```
//1 加载spring配置文件，根据创建对象
ApplicationContext context =
    new ClassPathXmlApplicationContext("bean1.xml");
//2 得到配置创建的对象
User user = (User) context.getBean("user");
```

1.5.3.7. Springbean 的生命周期

1.5.3.7.1. BeanFactory 创建 bean

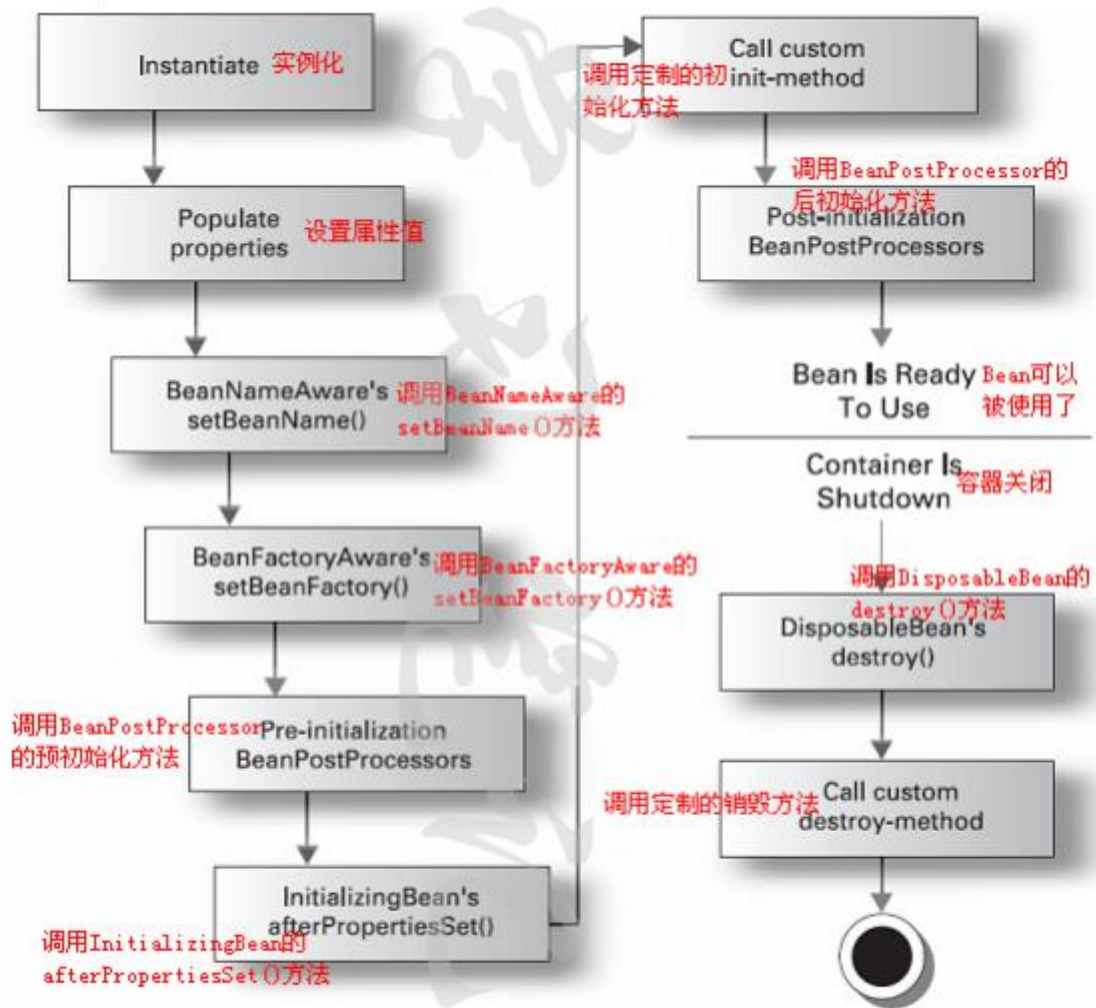


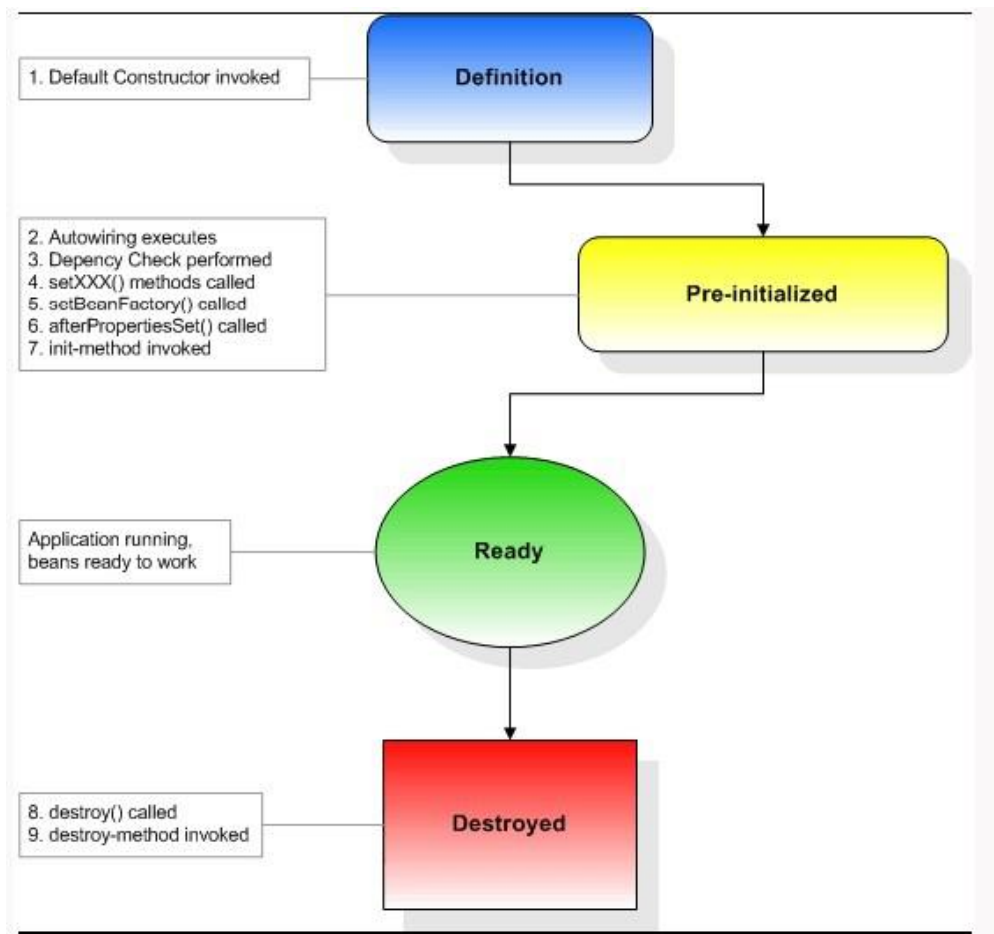
Figure 2.1 The life cycle of a bean within a Spring bean factory container

1. Bean 的构造
2. 调用 setXXX()方法设置 Bean 的属性
3. 调用 BeanNameAware 的 setBeanName()
4. 调用 BeanFactoryAware 的 setBeanFactory()方法
5. 调用 BeanPostProcessor 的 postProcessBeforeInitialization()方法
6. 调用 InitializingBean 的 afterPropertiesSet()方法

7. 调用自定义的初始化方法
8. 调用 BeanPostProcessor 类的 postProcessAfterInitialization()方法
9. working
10. 调用 DisposableBean 的 destroy()方法
11. 调用自定义的销毁方法

1.5.3.7.2. ApplicationContext

基本跟 beanfactory 流程一样。



1.5.3.8. Springbean 加载顺序

在加载某个包含 bean 的 xml 文件时，按照 bean 的类型

- 1) BeanFactoryPostProcessor 类的 bean;
- 2) BeanPostProcessor 类的 bean;
- 3) 普通 bean，包括 import 进来的（bean 标签和 scan 标签指定的）；的顺序进行加载。同类型的 bean 按照定义顺序加载。所有 bean 默认是单例的。

因此，对于 BeanFactoryPostProcessor 和 BeanPostProcessor 类型的 bean，即使被放置在最后面，也会先加载。

1.5.4. 依赖注入（DI）

1.5.4.1. 属性注入

最基本的对象创建方式，只需要有一个无参构造函数（类中没有写任何的构造函数，默认就是有一个构造函数，如果写了任何一个构造函数，默认是无参构造函数就不会自动创建哦!!）和字段的 setter 方法。



```
public class UserService implements IUserService{
    public IUserDao userdao;

    public IUserDao getUserdao() {
        return userdao;
    }

    public void setUserdao(IUserDao userdao) {
        this.userdao = userdao;
    }

    public List<String> queryAll(){
        return userdao.querylistList();
    }
}

<bean id="userservice" class="com.lx.spring.service.UserService">
    <!-- name:当前类的属性 ref:你要引用的类的id -->
    <property name="userdao" ref="userdao"></property>
</bean>

<bean id="userdao" class="com.lx.spring.dao.UserDao"></bean>
```

1.5.4.2. 构造注入

```
public class EmpServiceImp implements IEmpService{
    private IEmpDao empDao;
    public EmpServiceImp(IEmpDao empDao){
        this.empDao=empDao;
    }

    public void queryAll(){
        empDao.queryAll();
    }
}

<bean id="empService" class="com.lx.spring.service.EmpServiceImp">
    <constructor-arg name="empDao" ref="empDao"></constructor-arg>
</bean>
```

1.5.5. IOC 和 DI 的好处

- 1、我们无需关注对象的创建和销毁。
- 2、解决代码耦合度高的问题，利于代码的重用和利于维护。

总结：

DI 和 IOC 其实是一个思想，她们的好处是：如果依赖的类修改了，比如修改了构造函数，如果没有依赖注入，则需要修改依赖对象调用着，如果依赖注入则不需要。

spring IOC 的好处是，对象的构建如果依赖非常多的对象，且层次很深，外层在构造对象时很麻烦且不一定知道如何构建这么多层次的对象。IOC 帮我们管理对象的创建，只需要在配置文件里指定如何构建，每一个对象的配置文件都在类编写的时候指定了，所以最外层对象不需要关心深层次对象如何创建的，前人都写好了。

1.5.6. 业务委托模式

（要的是接口类型，注入的是实现类【为了解偶】）



```
public class EmpServiceImp implements IEmpService{
    private IEmpDao empDao;
    public EmpServiceImp(IEmpDao empDao){
        this.empDao=empDao;
    }

    public void queryAll(){
        empDao.queryAll();
    }
}
```

2. Spring 容器

Spring 的容器可以分为两种类型

2.1. BeanFactory

（org.springframework.beans.factory.BeanFactory 接口定义）是最简答的容器，提供了基本的 DI 支持。最常用的 BeanFactory 实现就是 XmlBeanFactory 类，它根据 XML 文件中的定义加载 beans，该容器从 XML 文件读取配置元数据并用它去创建一个完全配置的系统或应用。

2.2. ApplicationContext 应用上下文

(org.springframework.context.ApplicationContext) 基于 BeanFactory 之上构建，并提供面向应用的服务。

ApplicationContext 通常的实现

- **ClassPathXmlApplicationContext**: 从类路径下的 XML 配置文件中加载上下文定义，把应用上下文定义文件当做类资源。
- **FileSystemXmlApplicationContext**: 读取文件系统下的 XML 配置文件并加载上下文定义。
- **XmlWebApplicationContext**: 读取 Web 应用下的 XML 配置文件并装载上下文定义。

```
ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
```

3. AOP

3.1. AOP 概念

- 1、AOP 面向切面编程，扩展功能不需要修改源代码
- 2、AOP 采用横向抽取机制，取代了传统纵向继承体系重复性代码

3.2. AOP 原理



3.3.AOP 操作术语

<pre>public class User { public void add() { } public void update() { } public void delete() { } public void findAll() { } }</pre>	<ul style="list-style-type: none">* 连接点：类里面哪些方法可以被增强，这些方法称为连接点* 切入点：在类里面可以有非常多的方法被增强，比如实际操作中，只是增强了类里面add方法和update方法，实际增强的方法称为 切入点* 通知/增强：增强的逻辑，称为增强，比如扩展日志功能，这个日志功能称为增强<ul style="list-style-type: none">前置通知：在方法之前执行后置通知：在方法之后执行异常通知：方法出现异常最终通知：在后置之后执行环绕通知：在方法之前和之后执行* 切面：把增强应用到具体方法上面，过程称为切面<ul style="list-style-type: none">把增强用到切入点过程
--	--

3.4. AOP 常用术语：

3.4.1. 连接点(Joinpoint)

增强程序执行的某个特定位置(要在哪个地方做增强操作)。Spring 仅支持方法的连接点，既仅能在方法调用前，方法调用后，方法抛出异常时等这些程序执行点进行织入增强。

3.4.2. 切点 (Pointcut)

切点是一组连接点的集合。AOP 通过“切点”定位特定的连接点。通过数据库查询的概念来理解切点和连接点的关系再适合不过了：连接点相当于数据库中的记录，而切点相当于查询条件。

3.4.3. 增强 (Advice)

增强是织入到目标类连接点上的一段程序代码。表示要在连接点上做的操作。

3.4.4. 切面 (Aspect)

切面由切点和增强（引介）组成(可以包含多个切点和多个增强)，它既包括了横切逻辑的定义，也包括了连接点的定义，SpringAOP 就是负责实施切面的框架，它将切面所定义的横切逻辑织入到切面所指定的链接点中。

3.5.通知类型

before:前置通知,在目标方法被调用之前调用通知功能

after:后置通知,在目标方法完成之后调用通知,不关心目标方法的输出

after-returning:返回通知,在目标方法成功执行之后调用通知

after-throwing:异常通知,在目标方法抛出异常之后执行通知

around:环绕通知,在调用前后都会执行

注意：环绕通知需要在增强类的方法内接收一个参数：

ProceedingJoinPoint

3.6. Spring 的 AOP 操作







1 在 spring 里面进行 aop 操作，使用 aspectj 实现。

- (1) aspectj 不是 spring 一部分，和 spring 一起使用进行 aop 操作。
- (2) Spring2.0 以后新增了对 AspectJ 支持。

2 使用 aspectj 实现 aop 有两种方式。

- (1) 基于 aspectj 的 xml 配置。
- (2) 基于 aspectj 的注解方式。

3.6.1. 导入相应的 jar 包

 aopalliance-1.0.jar	2016-02-29 19:06	360压缩	5 KB
 aspectjweaver-1.8.7.jar	2016-02-29 19:06	360压缩	1,822 KB
 spring-aop-4.2.4.RELEASE.jar	2016-02-29 19:06	360压缩	362 KB
 spring-aspects-4.2.4.RELEASE.jar	2016-02-29 19:06	360压缩	58 KB

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aop</artifactId>
  <version>4.3.10.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aspects</artifactId>
  <version>4.3.10.RELEASE</version>
</dependency>

<dependency>
  <groupId>aopalliance</groupId>
  <artifactId>aopalliance</artifactId>
  <version>1.0</version>
</dependency>
```

第一和第二需要额外提供。

第三和第四可以从 spring 安装包得到。

3.6.2. 引入 xml 定义，导入 aop 的定义

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

  <bean id="user" class="com.lx.spring.bean.User"></bean>
```

```
</beans>
```

3.6.3. 使用表达式配置切入点

1 切入点：实际增强的方法

2 常用的表达式

execution(<访问修饰符>?<返回类型><方法名>(<参数>)<异常>)

(1) execution(* cn.itcast.aop.Book.add(..))

(2) execution(* cn.itcast.aop.Book.*(..))

(3) execution(* *.*(..))

```
<!-- execution(* com.lx.ssh.aop.User.save(..)) -->
<!-- execution(* com.lx.ssh.aop.User.*(..)) -->
<!-- execution(* com.lx.ssh.aop.*.*(..)) -->
```

3.6.4. Xml 方式配置 aop

```
<!-- 配置对象 -->
<bean id="user" class="com.lx.ssh.aop.User"></bean>
<bean id="strong" class="com.lx.ssh.aop.Strong"></bean>

<!-- 2配置AOP操作 -->
<aop:config>
  <!-- 2.1配置切入点 -->
  <aop:pointcut expression="execution(* com.lx.ssh.aop.User.save(..))" id="point1"/>
  <!-- 2.2配置切面
         把增强用到方法上
  -->
  <aop:aspect ref="strong">
    <!-- 配置增强类型
           method:增强类里面哪个方法作为前置
    -->
    <aop:around method="around" pointcut-ref="point1"/>
  </aop:aspect>
```

```

public class Strong {
    public void around(ProceedingJoinPoint pjp) throws Throwable{
        System.out.println("日语");
        System.out.println("英语");
        pjp.proceed();
        System.out.println("方言");
    }
}

```

环绕通知需要在增强类的方法内接收一个参数：**ProceedingJoinPoint**
其他通知类型在增强类里面可以不用此参数

3.6.5. 注解的方式配置 aop

常用注解：

- @aspect 定义切面
- @pointcut 定义切点
- @before 标注 Before Advice 定义所在的方法
- @afterreturning 标注 After Returning Advice 定义所在的方法
- @afterthrowing 标注 After Throwing Advice 定义所在的方法
- @after 标注 After(Finally) Advice 定义所在的方法
- @around 标注 Around Advice 定义所在的方法

1、配置切面

```

public class Student{
    public void say() {
        System.out.println("我是一个成年人学生");
    }
}

```



```

@Aspect
public class StrongStudent {
    @Pointcut("execution(* com.hy.spring.aop.Student.say(..))")
    public void say(){
    }

    //实现环绕增强
    @Around("say()")
    public void around(ProceedingJoinPoint joinPoint) throws Throwable{
        System.out.println("开始增强");
        joinPoint.proceed(); //执行目标方法
        System.out.println("结束增强");
    }
}

```

2、需要在 spring.xml 里面开启配置

```
<aop:aspectj-autoproxy proxy-target-class="true"/>
```

为了让 spring 能自动帮我们实现织入 我们还需要开启自动注入 在 spring 配置文件中: <aop:aspectj-autoproxy proxy-target-class="true"/> 这样 spring 就能在 IOC 容器找到所有要织入的方法 动态帮我们织入。

4. Spring 的 dao 实现

1、spring 是一站式框架


- (1) 针对 javaEE 三层, 每一层都有解决技术。
- (2) 在 dao 层, 使用 jdbcTemplate


2、spring 对不同的持久化层技术都进行了封装

ORM持久化技术	模板类
JDBC	org.springframework.jdbc.core.JdbcTemplate
Hibernate5.0	org.springframework.orm.hibernate5.HibernateTemplate
IBatis(MyBatis)	org.springframework.orm.ibatis.SqlMapClientTemplate
JPA	org.springframework.orm.jpa.JpaTemplate

4.1. Jdbctemplate

4.1.1. 导入 jar 包

 spring-jdbc-4.2.4.RELEASE.jar

 spring-tx-4.2.4.RELEASE.jar

4.1.2. 添加

```
// 设置数据库信息
DriverManagerDataSource dataSource = new DriverManagerDataSource();
dataSource.setDriverClassName("com.mysql.jdbc.Driver");
dataSource.setUrl("jdbc:mysql:///spring_day03");
dataSource.setUsername("root");
dataSource.setPassword("root");

// 创建jdbcTemplate对象，设置数据源
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

// 调用jdbcTemplate对象里面的方法实现操作
// 创建sql语句
String sql = "insert into user values(?,?)";
int rows = jdbcTemplate.update(sql, "lucy", "250");
```

4.1.3. 修改

```
//设置数据库信息
DriverManagerDataSource dataSource = new DriverManagerDataSource();
dataSource.setDriverClassName("com.mysql.jdbc.Driver");
dataSource.setUrl("jdbc:mysql:///spring_day03");
dataSource.setUsername("root");
dataSource.setPassword("root");

//创建jdbcTemplate对象
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

//调用jdbcTemplate里面的方法实现update方法
String sql = "update user set password=? where username=?";
int rows = jdbcTemplate.update(sql, "1314", "lucy");
System.out.println(rows);
```

4.1.4. 删除

```
//设置数据库信息
DriverManagerDataSource dataSource = new DriverManagerDataSource();
dataSource.setDriverClassName("com.mysql.jdbc.Driver");
dataSource.setUrl("jdbc:mysql:///spring_day03");
dataSource.setUsername("root");
dataSource.setPassword("root");

//创建jdbcTemplate对象
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

//调用update方法实现删除
String sql = "delete from user where username=?";
int rows = jdbcTemplate.update(sql, "lucy");
System.out.println(rows);
```

4.1.5. 查询

4.1.5.1. 查询返回某一个值

```
//调用方法得到记录数
String sql = "select count(*) from user";
//调用jdbcTemplate的方法
int count = jdbcTemplate.queryForObject(sql, Integer.class);
System.out.println(count);
```

4.1.5.2. 查询返回对象



```
class MyRowMapper implements RowMapper<User> {  
    @Override  
    public User mapRow(ResultSet rs, int num) throws SQLException {  
        // 1 从结果集里面把数据得到  
        String username = rs.getString("username");  
        String password = rs.getString("password");  
  
        // 2 把得到数据封装到对象里面  
        User user = new User();  
        user.setUsername(username);  
        user.setPassword(password);  
  
        return user;  
    }  
}  
  
//创建jdbcTemplate对象  
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);  
  
//写sql语句, 根据username查询  
String sql = "select * from user where username=?";  
//调用jdbcTemplate的方法实现  
//第二个参数是接口 RowMapper, 需要自己写类实现接口, 自己做数据封装  
User user = jdbcTemplate.queryForObject(sql, new MyRowMapper(), "lucy");  
System.out.println(user);
```

4.1.5.3. 查询返回 list 集合

```
class MyRowMapper implements RowMapper<User> {  
    @Override  
    public User mapRow(ResultSet rs, int num) throws SQLException {  
        // 1 从结果集里面把数据得到  
        String username = rs.getString("username");  
        String password = rs.getString("password");  
  
        // 2 把得到数据封装到对象里面  
        User user = new User();  
        user.setUsername(username);  
        user.setPassword(password);  
  
        return user;  
    }  
}  
  
//写sql语句  
String sql = "select * from user";  
//调用jdbcTemplate的方法实现  
List<User> list = jdbcTemplate.query(sql, new MyRowMapper());  
  
System.out.println(list);
```

4.2. spring 中连接池的配置

1、导入 jar 包

 c3p0-0.9.2.1.jar	2016-06-18 21:01	360压缩	414 KB
 mchange-commons-java-0.2.3.4.jar	2016-06-18 21:08	360压缩	568 KB

2、创建 spring 配置文件，配置连接池

原始用法：

```

ComboPooledDataSource dataSource = new ComboPooledDataSource();
dataSource.setDriverClass("com.mysql.jdbc.Driver");
dataSource.setJdbcUrl("jdbc:mysql:///spring_day03");
dataSource.setUser("root");
dataSource.setPassword("root");

```

```

<!-- 配置c3p0连接池 -->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <!-- 注入属性值 -->
    <property name="driverClass" value="com.mysql.jdbc.Driver"></property>
    <property name="jdbcUrl" value="jdbc:mysql:///spring_day03"></property>
    <property name="user" value="root"></property>
    <property name="password"></property>
</bean>

```

4.3. 配置多个数据源

在 spring 里面配置多个数据源，然后根据业务的不同随时进行切换。

4.4. Spring 加载 properties 文件

4.4.1. 配置根目录下创建 db.properties

```

dataSource=com.mchange.v2.c3p0.ComboPooledDataSource
driverClass=com.mysql.jdbc.Driver
jdbcUrl=jdbc:mysql://127.0.0.1:3306/student
user=root
password=huayu

```

4.4.2. 让 spring 加载 properties 文件

```

<!-- 加载 properties -->
<context:property-placeholder
location="classpath:dbproperties/db.properties"/>

```

4.4.3. 在数据源中引入 properties 文件中的值

```

<!-- 配置数据源 -->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="driverClass" value="${driverClass}"></property>
    <property name="jdbcUrl" value="${jdbcUrl}"></property>
    <property name="user" value="${user}"></property>
    <property name="password" value="${password}"></property>
</bean>

```

4.5. dao 中使用 jdbcTemplate

```
<!-- 创建service和dao对象, 在service注入dao对象 -->
<bean id="userService" class="cn.itcast.c3p0.UserService">
  <!-- 注入dao对象 -->
  <property name="userDao" ref="userDao"></property>
</bean>

<bean id="userDao" class="cn.itcast.c3p0.UserDao">
  <!-- 注入jdbcTemplate对象 -->
  <property name="jdbcTemplate" ref="jdbcTemplate"></property>
</bean>
<!-- 创建jdbcTemplate对象 -->
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
  <!-- 把dataSource传递到模板对象里面 -->
  <property name="dataSource" ref="dataSource"></property>
</bean>
```

5. Spring Data Jpa

5.1. 什么是 springdata

Spring Data JPA 是 spring data 项目下的一个模块。提供了一套基于 JPA 标准操作数据库的简化方案。底层默认的是依赖 Hibernate JPA 来实现的。

Spring Data JPA 框架，主要针对的就是 Spring 唯一没有简化到的业务逻辑代码，至此，开发者连仅剩的实现持久层业务逻辑的工作都省了，唯一要做的，就只是声明持久层的接口，其他都交给 Spring Data JPA 来帮你完成！

5.2. Springdata Jpa 技术特点

我们只需要定义接口并集成 Spring Data JPA 中所提供的接口就可以了。不需要编写接口实现类。

6. Spring 整合 web 项目

6.1. 整合思路



6.2. Spring 整合 struts2

把 struts2 action 的创建交给 spring 来创建

```
<bean id="userAction" class="com.lx.ssh.action.UserAction" scope="prototype">
    <property name="userService" ref="userServiceImpl"></property>
</bean>
```

注意：

需要额外提供 struts 跟 spring 整合的 jar
struts2-spring-plugin-2.3.32.jar

6.3. Spring 整合 hibernate

6.3.1. 提供 jar

Spring-orm.jar

6.3.2. 整合数据库连接配置

把 hibernate 核心配置文件里面的数据库连接配置交给 Spring 来配置

不管采用何种持久化技术，都需要定义数据源。Spring中提供了4种不同形式的数据源配置方式：

spring 自带的数据源(DriverManagerDataSource)，DBCP数据源，C3P0数据源，JNDI数据源。

我们采用比较流行的 C3P0（需要导入 C3P0 的 jar）

```
<!-- 数据源-->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="driverClass" value="${driverclass}"></property>
    <property name="jdbcUrl" value="${url}"></property>
    <property name="user" value="${user}"></property>
    <property name="password" value="${password}"></property>

    <!-- 连接池参数配置-->
    <property name="acquireIncrement" value="3"></property>
    <property name="initialPoolSize" value="20"></property>
    <property name="minPoolSize" value="2"></property>
    <property name="maxPoolSize" value="10"></property>
</bean>
```


6.3.3. 创建 sessionFactory

第一次访问 hibernate 很慢，因为要创建 sessionFactory
把 sessionFactory 对象交给 Spring

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <!-- 注入dataSource,需要知道用的是哪个数据库 -->
    <property name="dataSource" ref="dataSource"></property>
    <!-- 指定使用hibernate的核心配置文件 -->
    <property name="configLocations" value="classpath:hibernate.cfg.xml"></property>
</bean>
```

6.3.4. 在服务器启动的时候加载配置文件

在服务启动的时候加载 spring 配置文件，把压力交给服务器
底层用的技术是：

- 1、servletContext
- 2、ServletContextListener（监听器，监听 servletContext 的创建，在创建的时候去加载配置文件）

```
<!-- 指定Spring的配置文件 -->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
</context-param>
<!-- 配置监听器，监听servletcontext的创建，然后在监听器里面去加载spring的核心配置文件 -->
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

6.3.5. Spring 对不同持久化框架的支持

Spring对不同持久化技术的支持

- Spring为各种支持的持久化技术，都提供了简单操作的模板和回调

ORM持久化技术	模板类
JDBC	org.springframework.jdbc.core.JdbcTemplate
Hibernate5.0	org.springframework.orm.hibernate5.HibernateTemplate
iBatis(MyBatis)	org.springframework.orm.ibatis.SqlMapClientTemplate
JPA	org.springframework.orm.jpa.JpaTemplate

HibernateTemplate常用API

- Serializable save(Object entity)
- void update(Object entity)
- void delete(Object entity)
- <T> T get(Class<T> entityClass, Serializable id)
- <T> T load(Class<T> entityClass, Serializable id)
- List find(String queryString, Object... values)

```
<!-- 创建hibernateTemplate(封装了原来我们写hibernate的那7个步骤) -->
<bean id="hibernateTemplate" class="org.springframework.orm.hibernate5.HibernateTemplate">
    <property name="sessionFactory" ref="sessionFactory"></property>
</bean>
```

6.3.6. 基本查询、修改、删除

```
public class UserDao implements IUserDao{
    private HibernateTemplate htem;

    public void setHtem(HibernateTemplate htem) {
        this.htem = htem;
    }
    public void save(User user){
        htem.save(user);
    }
    public User getUser(Integer id){
        return htem.get(User.class, id);
    }
    @Override
    public void update(User user) {
        htem.update(user);
    }
    @Override
    public void delete(User user) {
        htem.delete(user);
    }
}
```

6.3.7. 查询所有及带条件

```
public List<User> findAll(){
    return (List<User>)htem.find("from User");
}
public List<User> findByName(String name,Integer id){
    return (List<User>)htem.find("from User u where u.name=? and u.id=?",name,id);
}
```

find(Class,Object...values); 【values 的位置是个可变参数】

```
public List<User> findByName(String name){
    return (List<User>)htem.find("from User u where u.name like ?",name);
}
```

6.3.8. 分页查询

Dao:

```
public List<User> findByName(DetachedCriteria deta,int first,int max){
    return (List<User>) htem.findByCriteria(deta, first, max);
}
```

Service:

```
public List<User> findByName(int first, int max,String name) {
    DetachedCriteria deta=DetachedCriteria.forClass(User.class);
    deta.add(Restrictions.eq("name", name));
    return userdao.findByName(deta, first, max);
}
```

6.3.9. HibernateTemplate 执行原生态 sql 语句

```
String sql = "select * from user";//原生态sql语句
Session session = htem.getSessionFactory().openSession();
List list = session.createQuery(sql).list();//用debug查看返回的list是怎么存储数据的
session.close();//注意要自行关闭session
```

6.3.10. Dao 里面需要 hibernateTemplate

```
public class UserDao implements IUserDao{
    private HibernateTemplate htem;

    public void setHtem(HibernateTemplate htem) {
        this.htem = htem;
    }
    /**
     * 保存
     * @param user
     * @void
     */
    public void save(User user){
        htem.save(user);
    }
}

<bean id="userDao" class="com.lx.ssh.dao.UserDao">
    <property name="htem" ref="hibernateTemplate"></property>
</bean>
```

6.3.11. Spring 整合 hibernate 其他方式

不需要 hibernate 核心配置文件

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessi
<property name="dataSource" ref="dataSource"></property>
<!-- 指定hibernate核心配置文件的位置 -->
<property name="configLocations" value="classpath:hibernate.cfg.xml"></property>
<property name="hibernateProperties">
    <props>
        <prop key="hibernate.hbm2ddl.auto">update</prop>
        <prop key="hibernate.show_sql">true</prop>
        <prop key="hibernate.format_sql">true</prop>
        <prop key="dialect">org.hibernate.dialect.Oracle10gDialect</prop>
    </props>
</property>
<property name="mappingResources">
    <list>
        <value>com.lx.ssh.bean/User.hbm.xml</value>
    </list>
</property>
```

```
<!--配置SessionFactory-->
<bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessi
<property name="dataSource" ref="dataSource"></property>
<property name="hibernateProperties">
    <props>
        <prop key="dialect">org.hibernate.dialect.MySQLDialect</prop>
        <prop key="show_sql">true</prop>
        <prop key="format_sql">true</prop>
    </props>
</property>
<property name="annotatedClasses">
    <list>
        <value>com.hy.ssh.bean.Student</value>
    </list>
</property>
</bean>

<!--配置hibernateTemplate-->
<bean id="hibernateTemplate" class="org.springframework.orm.hibernate5.HibernateTemplate">
```

总结：

用 xml 来映射：

mappingResources: 加载 xxx.hbm.xml

用注解来映射:

annotatedClasses: 加载实体类

6.4. Spring 核心配置文件约束引用

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

       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/tx
           http://www.springframework.org/schema/tx/spring-tx.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop.xsd">
```

6.5. Spring 事物管理

6.5.1. 初步理解

理解事务之前，先讲一个你日常生活中最常干的事：取钱。

比如你去 ATM 机取 1000 块钱，大体有两个步骤：首先输入密码金额，银行卡扣掉 1000 元钱；然后 ATM 出 1000 元钱。这两个步骤必须是要么都执行要么都不执行。如果银行卡扣除了 1000 块但是 ATM 出钱失败的话，你将会损失 1000 元；如果银行卡扣钱失败但是 ATM 却出了 1000 块，那么银行将损失 1000 元。所以，如果一个步骤成功另一个步骤失败对双方都不是好事，如果不管哪一个步骤失败了以后，整个取钱过程都能回滚，也就是完全取消所有操作的话，这对双方都是极好的。

事务就是用来解决类似问题的。事务是一系列的动作，它们综合在一起才是一个完整的工作单元，这些动作必须全部完成，如果有一个失败的话，那么事务就会回滚到最开始的状态，仿佛什么都没发生过一样。

事务有四个特性：ACID

- ### 6.5.2. 核心接口

Spring 事务管理涉及的接口的联系如下:



```
Public interface PlatformTransactionManager()...{  
    // 由TransactionDefinition得到TransactionStatus对象  
    TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;  
    // 提交  
    void commit(TransactionStatus status) throws TransactionException;  
    // 回滚  
    void rollback(TransactionStatus status) throws TransactionException;  
}
```

从这里可知具体的具体的事务管理机制对 **Spring** 来说是透明的，它并不关心那些，那些是对应各个平台需要关心的，所以 **Spring** 事务管理的一个优点就是为不同的事务 **API** 提供一致的编程模型，如 **JTA**、**JDBC**、**Hibernate**、**JPA**。下面分别介绍各个平台框架实现事务管理的机制。

6.5.2.1.1. JDBC 事物

6.5.2.1.2. Hibernate 事物

6.5.2.1.3. Java 持久化 api 事物 (jpa)

6.5.2.2. 基本事物属性的定义

6.5.2.2.1. 传播行为

事务的第一个方面是传播行为（**propagation behavior**）。当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。**Spring** 定义了七种传播行为：

传播行为	含义
PROPAGATION_REQUIRED	表示当前方法必须运行在事务中。如果当前事务存在，方法将会在该事务中运行。否则，会启动一个新的事务
PROPAGATION_SUPPORTS	表示当前方法不需要事务上下文，但是如果存在当前事务的话，那么该方法会在这个事务中运行
PROPAGATION_MANDATORY	表示该方法必须在事务中运行，如果当前事务不存在，则会抛出一个异常
PROPAGATION_REQUIRED_NEW	表示当前方法必须运行在它自己的事务中。一个新的事务将被启动。如果存在当前事务，在该方法执行期间，当前事务会被挂起。如果使用 JTATransactionManager 的话，则需要访问 TransactionManager
PROPAGATION_NOT_SUPPORTED	表示该方法不应该运行在事务中。如果存在当前事务，在该方法运行期间，当前事务将被挂起。如果使用 JTATransactionManager 的话，则需要访问 TransactionManager
PROPAGATION_NEVER	表示当前方法不应该运行在事务上下文中。如果当前正有一个事务在运行，则会抛出异常
PROPAGATION_NESTED	表示如果当前已经存在一个事务，那么该方法将会嵌套事务中运行。嵌套的事务可以独立于当前事务进行单独地提交或回滚。如果当前事务不存在，那么其行为与 PROPAGATION_REQUIRED 一样。注意各厂商对这种传播行为的支持是有所差异的。可以参考资源管理器的文档来确认它们是否支持嵌套事务

6.5.2.2.2. 隔离级别

事务的第二个维度就是隔离级别（**isolation level**）。隔离级别定义了一个事务可能受其他并发事务影响的程度。

（1）并发事务引起的问题

在典型的应用程序中，多个事务并发运行，经常会操作相同的数据来完成各自的任务。并发虽然是必须的，但可能会导致一下的问题。

- 脏读（Dirty reads）——脏读发生在一个事务读取了另一个事务改写但尚未提交的数据时。如果改写在稍后被回滚了，那么第一个事务获取的数据就是无效的。
- 不可重复读（Nonrepeatable read）——不可重复读发生在一个事务执行相同的查询两次或两次以上，但是每次都得到不同的数据时。这通常是因为另一个并发事务在两次查询期间进行了更新。
- 幻读（Phantom read）——幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录。

不可重复读与幻读的区别：

不可重复读的重点是修改：

同样的条件，你读取过的数据，再次读取出来发现值不一样了

例如：在事务 1 中，Mary 读取了自己的工资为 1000,操作并没有完成

```
con1 = getConnection();
select salary from employee empId ="Mary";
```

在事务 2 中，这时财务人员修改了 Mary 的工资为 2000,并提交了事务。

```
con2 = getConnection();
update employee set salary = 2000;
con2.commit();
```

在事务 1 中，Mary 再次读取自己的工资时，工资变为了 2000

```
//con1
select salary from employee empId ="Mary";
```

在一个事务中前后两次读取的结果并不一致，导致了不可重复读。

幻读的重点在于新增或者删除：

同样的条件，第 1 次和第 2 次读出来的记录数不一样

例如：目前工资为 1000 的员工有 10 人。事务 1,读取所有工资为 1000 的员工。

```
con1 = getConnection();
Select * from employee where salary =1000;
```

共读取 10 条记录

这时另一个事务向 employee 表插入了一条员工记录，工资也为 1000

```
con2 = getConnection();
Insert into employee(empId,salary) values("Lili",1000);
con2.commit();
```

事务 1 再次读取所有工资为 1000 的员工

```
//con1
select * from employee where salary =1000;
```

共读取到了 11 条记录，这就产生了幻像读。

从总的结果来看，似乎不可重复读和幻读都表现为两次读取的结果不一致。但如果你从控制的角度来看，两者的区别就比较大。

对于前者，只需要锁住满足条件的记录。

对于后者，要锁住满足条件及其相近的记录。

（2）隔离级别

隔离级别	含义
ISOLATION_DEFAULT	使用后端数据库默认的隔离级别
ISOLATION_READ_UNCOMMITTED	最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读
ISOLATION_READ_COMMITTED	允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生
ISOLATION_REPEATABLE_READ	对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生
ISOLATION_SERIALIZABLE	最高的隔离级别，完全服从 ACID 的隔离级别，确保阻止脏读、不可重复读以及幻读，也是最慢的事务隔离级别，因为它通常是通过完全锁定事务相关的数据库表来实现的

spring 默认使用：ISOLATION_DEFAULT

6.5.2.2.3. 只读事物

如果事务只进行读取的动作，则可以利用底层数据库在只读操作时发生的一些最佳化动作，由于这个动作利用到数据库在只读的事务操作最佳化，因而必须在事务中才有效，也就是说要搭配传播行为 PROPAGATION_REQUIRED、PROPAGATION_REQUIRES_NEW、PROPAGATION_NESTED 来设置。

“只读事务”并不是一个强制选项，它只是一个“暗示”，提示数据库驱动程序和数据库系统，这个事务并不包含更改数据的操作，那么 JDBC 驱动程序和数据库就有可能根据这种情况对该事务进行一些特定的优化，比方说不安排相应的数据库锁，以减轻事务对数据库的压力，毕竟事务也是要消耗数据库的资源的。

但是你非要在“只读事务”里面修改数据，也并非不可以，只不过对于数据一致性的保护不像“读写事务”那样保险而已。

因此，“只读事务”仅仅是一个性能优化的推荐配置而已，并非强制你要这样做不可

只读事物好处：

如果你一次执行单条查询语句，则没有必要启用事务支持，数据库默认支持 SQL 执行期间的读一致性；

如果你一次执行多条查询语句，例如统计查询，报表查询，在这种场景下，多条查询 SQL 必须保证整体的读一致性，否则，在前条 SQL 查询之后，后条 SQL 查询之前，数据被其他用户改变，则该次整体的统计查询将会出现读数据不一致的状态，此时，应该启用事务支持

read-only="true"表示该事务为只读事务，比如上面说的多条查询的这种情况可以使用只读事务，

由于只读事务不存在数据的修改，因此数据库将会为只读事务提供一些优化手段，例如 Oracle 对于只读事务，不启动回滚段，不记录回滚 log。

（1）在 JDBC 中，指定只读事务的办法为：connection.setReadOnly(true);

（2）在 Hibernate 中，指定只读事务的办法为：session.setFlushMode(FlushMode.NEVER);

此时，Hibernate 也会为只读事务提供 Session 方面的一些优化手段

（3）在 Spring 的 Hibernate 封装中，指定只读事务的办法为：bean 配置文件中，prop 属性增加“read-Only”

或者用注解方式@Transactional(readOnly=true)

Spring 中设置只读事务是利用上面两种方式（根据实际情况）

在将事务设置成只读后，相当于将数据库设置成只读数据库，此时若要进行写的操作，会出现错误。

6.5.2.2.4. 事物超时

有的事务操作可能延续很长一段时间，事务本身可能关联到数据表的锁定，因而长时间的事务操作会有效率上的问题，对于过长的事务操作，考虑 Roll back 事务并要求重新操作，而不是无限时的等待事务完成。可以设置事务超时期间，计时是从事务开始时，所以这个设置必须搭配传播行为 PROPAGATION_REQUIRED、PROPAGATION_REQUIRES_NEW、PROPAGATION_NESTED 来设置。

6.5.2.2.5. 回滚规则

事务五边形的最后一个方面是一组规则，这些规则定义了哪些异常会导致事务回滚而哪些不会。默认情况下，事务只有遇到运行期异常时才会回滚，而在遇到检查型异常时不会回滚（这一行为与 EJB 的回滚行为是一致的）

但是你可以声明事务在遇到特定的检查型异常时像遇到运行期异常那样回滚。同样，你还可以声明事务遇到特定的异常不回滚，即使这些异常是运行期异常。

6.5.3. 编程式事物和声明式事物区别

Spring 提供了对编程式事务和声明式事务的支持，编程式事务允许用户在代码中精确定义事务的边界，而声明式事务（基于 AOP）有助于用户将操作与事务规则进行解耦。简单地说，编程式事务侵入到了业务代码里面；而声明式事务由于基于 AOP，所以既能起到事务管理的作用，又可以不影响业务代码的具体实现。

总结：

显然声明式事务管理要优于编程式事务管理，这正是 spring 倡导的非侵入式的开发方式。声明式事务管理使业务代码不受污染，

6.5.4. 编程式事物（了解）

6.5.5. 声明式事物（配置）

声明式事务管理建立在 AOP 之上的。其本质是对方法前后进行拦截，然后在目标方法开始之前创建或者加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。声明式事务最大的优点就是不需要通过编程的方式管理事务，这样就不需要在业务逻辑代码中掺杂事务管理的代码，只需在配置文件中做相关的事务规则声明(或通过基于 @Transactional 注解的方式)，便可以将事务规则应用到业务逻辑中。

6.5.5.1. 基于 xml 配置实现

第一步：配置事物管理器

Jdbctemplate 事物管理器类：

```
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"></property>
  </bean>
```

Hibernate 事物管理器类：

```
<!-- 第一步配置事物管理器 -->
<bean id="transactionManager" class="org.springframework.orm.hibernate5.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory"></property>
</bean>
```


第二步：配置增强切面

```
<!-- 配置事务增强切面 -->
<tx:advice id="adviceTx" transaction-manager="transactionManager">
  <tx:attributes>
    <!--
      设置进行事务操作的方法匹配规则
      propagation: 传播机制-->
    <!-- * 匹配任意方法 -->
    <tx:method name="*" propagation="REQUIRED"/>
    <!-- 匹配以save开头的方法 -->
    <tx:method name="save*" propagation="REQUIRED"/>
  </tx:attributes>
</tx:advice>
```

第三步：配置增强

```
<!-- 配置spring 事务 -->
<aop:config>
  <!-- 切入点的配置 -->
  <aop:pointcut id="txdao" expression="execution(* com.hy.ssh.service.*(..))"></aop:pointcut>
  <!-- 引入切面和切入点 -->
  <aop:advisor advice-ref="adviceTx" pointcut-ref="txdao"></aop:advisor>
</aop:config>
```

在声明式的事务处理中，要配置一个切面，即一组方法，如

Java 代码 

```
<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
    <tx:method name="*" read-only="true" propagation="NOT_SUPPORTED" />
  </tx:attributes>
</tx:advice>
```

其中就用到了 propagation，表示打算对这些方法怎么使用事务，是用还是不用，其中 propagation 有七种配置，REQUIRED、SUPPORTS、MANDATORY、REQUIRES_NEW、NOT_SUPPORTED、NEVER、NESTED。默认是 REQUIRED。

2. 七种配置的意思

下面是 Spring 中 Propagation 类的事务属性详解：

REQUIRED：支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。

SUPPORTS：支持当前事务，如果当前没有事务，就以非事务方式执行。

MANDATORY: 支持当前事务, 如果当前没有事务, 就抛出异常。

REQUIRES_NEW: 新建事务, 如果当前存在事务, 把当前事务挂起。

NOT_SUPPORTED: 以非事务方式执行操作, 如果当前存在事务, 就把当前事务挂起。

NEVER: 以非事务方式执行, 如果当前存在事务, 则抛出异常。

NESTED: 支持当前事务, 如果当前事务存在, 则执行一个嵌套事务, 如果当前没有事务, 就新建一个事务。

3.注意.

这个配置将影响数据存储, 必须根据情况选择。

6.5.5.2. 基于注解方式实现

第一步:

```
<!-- 第一步配置事物管理器 -->
<bean id="transactionManager" class="org.springframework.orm.hibernate5.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory"></property>
</bean>
```

第二步:

```
<!-- 第二步 开启事物注解 -->
<tx:annotation-driven transaction-manager="transactionManager"/>
```

第三步:

在你的 service 实现类里面加上这个注解。

```
@Transactional
public class UserServiceImpl implements IUserService{
    private IUserDao userDao;
    public void setUserDao(IUserDao userDao) {
        this.userDao = userDao;
    }
    @Override
    public void save(User user) {
        userDao.save(user);
    }
}
```

@Transactional注解

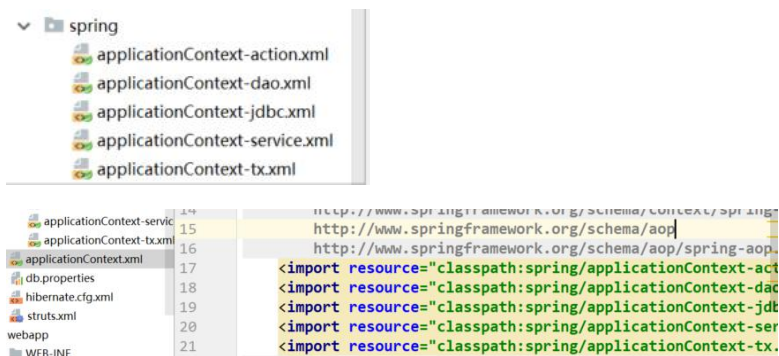
@Transactional属性

属性	类型	描述
value	String	可选的限定描述符，指定使用的事务管理器
propagation	enum: Propagation	可选的事务传播行为设置
isolation	enum: Isolation	可选的事务隔离级别设置
readOnly	boolean	读写或只读事务，默认读写
timeout	int (in seconds granularity)	事务超时时间设置
rollbackFor	Class对象数组，必须继承自Throwable	导致事务回滚的异常类数组
rollbackForClassName	类名数组，必须继承自Throwable	导致事务回滚的异常类名字数组
noRollbackFor	Class对象数组，必须继承自Throwable	不会导致事务回滚的异常类数组
noRollbackForClassName	类名数组，必须继承自Throwable	不会导致事务回滚的异常类名字数组

6.6. Spring 分模块开发

```
<!-- 用户管理模块 -->
<import resource="classpath:Springconfig/user_spring.xml"/>
```

6.6.1. 按照模块分成多个 xml，保留一个主配置文件



加载的时候就加载主配置文件

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:applicationContext.xml</param-value>
</context-param>
```

6.6.2. 引用的时候用*号

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:spring/applicationContext-*.xml</param-value>
</context-param>
```

7. 注解配置

7.1.1 注解方式管理 bean

7.1.1.1. 注解创建对象

```
<!-- 开启注解扫描 -->
<!-- 到包里面扫描类、属性、方法上面是否有注解 -->
<context:component-scan base-package="com.lx.ssh"></context:component-scan>
```

```
@Component(value="user") // <bean id="user" class=""/>
public class User {

    public void add() {
        System.out.println("add.....");
    }
}
```

Spring 中提供@Component 的三个衍生注解：(功能目前来讲是一致的)

- * @Controller :WEB 层
- * @Service :业务层
- * @Repository :持久层

这三个注解是为了让标注类本身的用途清晰，Spring 在后续版本会对其增强

- (1) @Component
- (2) @Controller
- (3) @Service
- (4) @Repository

目前这四个注解功能是一样的，都创建对象。

7.1.1.2. 多实例配置

```
@Scope(value="prototype")
public class User {
```


7.1.1.3. 注解注入属性

7.1.1.3.1. @Autowired

用来装配 bean，都可以写在字段上，或者方法上。

默认情况下必须要求依赖对象必须存在，如果要允许 null 值，可以设置它的 required 属性为 false，例如：@Autowired(required=false)

1 创建 service 类，创建 dao 类，在 service 得到 dao 对象
(1) 创建 dao 和 service 对象

```
@Component(value="userDao")  
public class UserDao {  
  
}
```

(2) 在 service 类里面定义 dao 类型属性

```
//得到dao对象  
//1 定义dao类型属性  
//在dao属性上面使用注解 完成对象注入  
@Autowired  
private UserDao userDao;  
// 使用注解方式时候不需要set方法
```

7.1.1.3.2. @Resource

```
// name属性值 写 注解创建dao对象 value值  
@Resource(name="userDao")  
private UserDao userDao;
```

Qualifier

@Qualifier

7.1.1.3.3. @qualifier

用于区分多个实现类

```
@Autowired
@Qualifier(value = "hibernateStudentDaoImpl")
private IStudentDao studentDao;
```

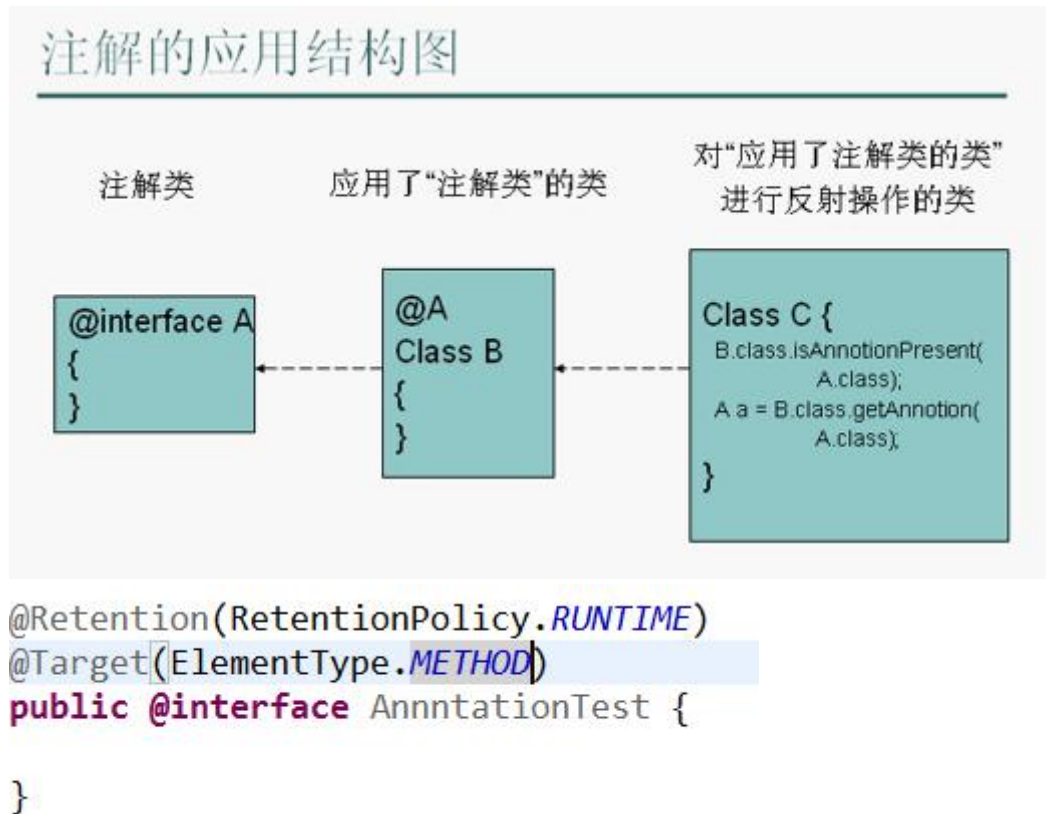
7.1.1.3.4. 注解和配置文件混合使用

```
1 创建对象操作使用配置文件方式实现
2 注入属性的操作使用注解方式实现
```

7.1.1.3.5. @AutoWired 的底层实现原理

1. 首先根据 byType 搜索，如果搜索到唯一就正常注入，
2. 如果根据 byType 结果大于 1，接着根据 ByName 搜索（name 值就是属性名）
3. 如果根据 byType 结果大于 1，也可以使用@Qualifier

7.1.2. 注解



8. Java 配置

Java 配置是 Spring4.x 推荐的配置方式，可以完全替代 xml 配置。

Spring 的 Java 配置方式是通过 `@Configuration` 和 `@Bean` 这两个注解实现的：

1、`@Configuration` 相当于 spring 的配置文件 XML

注解作为一个标示告知 Spring：这个类将包含一个或者多个 Spring Bean 的定义。这些 Bean 的定义是使用 `@Bean` 注解所标注的方法。

2、`@Bean` 用到方法上，表示当前方法的返回值是一个 bean

8.1. 注解配置和 Java 配置的区别

这两种方法的区别在于如果使用注解的方式，那么你需要在 Service 层，DAO 层的时候，需要在类上进行注解，就可获得 spring 的依赖注入：

```

package di;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
//注解配置
@Service
public class UseFunctionService {
    @Autowired
    FunctionService functionService;

    public String sayHello(String word) {
        return functionService.toHello(word);
    }
}

```

如果使用 java 配置的方式，那么就不需要在类上写注解了，直接在配置类里面进行申明即可：

```

package javaconfig;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class JavaConfig {
    //通过这种方式，获得spring的依赖注入
    @Bean
    public UseFunctionService useFunctionService () {
        return new UseFunctionService ();
    }
}

```

@Bean 告知 Spring 这个方法将返回一个对象，该对象应该被注册为 Spring 应用上下文中的一个 Bean。方法名作为该 Bean 的 ID。在该方法中所实现的所有逻辑本质上都是为了创建 Bean。上面例子中，该方法创建并返回一个 UseFunctionService 的实例对象，该对象被注册为 Spring 应用上下文中 ID 为 useFunctionService 的 Bean。

注意:方法名是作为返回对象的名字的，因此一般不带 get。

优点：

在XML配置中，Bean的类型和ID都是由Spring属性来表示的。Spring标识符的缺点是它们无法进行编译器检查。在Spring的基于Java的配置中并没有String属性。Bean的ID和类型都被视为方法签名的一部分。Bean的实际创建是在方法体中定义的。因为它们全都是Java代码，所以我们可以进行编译器检查确保Bean的类型是合法类型，并且Bean的ID是唯一的。

8.2. 例子

8.2.1. 一般用法

```
@Configuration
public class AppConfig {

    @Bean
    public EmpDaoImp empDao(){
        return new EmpDaoImp();
    }

    @Bean
    public EmpServiceImp empService(){
        EmpServiceImp empService=new EmpServiceImp();
        empService.setEmpDao(empDao());
        return empService;
    }
}
```

8.2.2. 指定 bean 的作用范围

```
@Configuration
public class AppConfig {
    @Bean
    @Scope("prototype")
    public Foo foo() {
        return new Foo();
    }
}
```

8.2.3. 做到 spring 零配置

```
/**
 * @Author: wangsq
 * @Date: 2018/11/22 16:29
 * @Description:
 */

@Configuration
@ComponentScan(basePackages = {"com.hy.ssh"})//扫描我们的业务bean
@EnableTransactionManagement
public class JavaConfig {

    @Bean
    public ComboPooledDataSource dataSource() throws PropertyVetoException {
```

```

        ComboPooledDataSource dataSource=new ComboPooledDataSource();

        dataSource.setDriverClass("com.mysql.jdbc.Driver");

        dataSource.setJdbcUrl("jdbc:mysql://127.0.0.1:3306/student");

        dataSource.setUser("root");

        dataSource.setPassword("lanxin");

        return dataSource;
    }

    @Bean

    public LocalSessionFactoryBean sessionFactory() throws PropertyVetoException {

        LocalSessionFactoryBean sessionFactory=new LocalSessionFactoryBean();

        sessionFactory.setDataSource(dataSource());

        Properties properties=new Properties();

        properties.setProperty("hibernate.show_sql","true");

        properties.setProperty("hibernate.format_sql","true");

        properties.setProperty("dialect","org.hibernate.dialect.MySQLDialect");

        sessionFactory.setHibernateProperties(properties);

        sessionFactory.setMappingResources("hbm/Student.hbm.xml");

        return sessionFactory;
    }

    @Bean

    public HibernateTemplate hibernateTemplate() throws PropertyVetoException {

        HibernateTemplate hibernateTemplate=new HibernateTemplate();

        hibernateTemplate.setSessionFactory(sessionFactory().getObject());

        return hibernateTemplate;
    }

    @Bean

    public HibernateTransactionManager hibernateTransactionManager() throws PropertyVetoException {

        HibernateTransactionManager hibernateTransactionManager=new HibernateTransactionManager();

        hibernateTransactionManager.setSessionFactory(sessionFactory().getObject());

        return hibernateTransactionManager;
    }

}

```

8.2.4. 常用注解说明

8.2.4.1. @Configuration

表示是一个 spring 的配置文件

8.2.4.2. @bean

标注一个方法的返回值是 spring 的一个 bean。

8.2.4.3. @ComponentScan

```
@ComponentScan(basePackages = {"com.hy.ssh"})
```

扫描我们的业务 bean

8.2.4.4. @EnableTransactionManagement

注解@EnableTransactionManagement 通知 Spring,@Transactional 注解的类被事务的切面包围。这样@Transactional 就可以使用了。

8.2.4.5. @ImportResource

`@ImportResource("classpath:applicationContext.xml")`加载 spring 的 xml 配置文件。

也可以使用通配符。`@ImportResource("classpath:applicationContext-*.xml")`

8.2.4.6. @Import

```
@Import({AppConfig.class})
```

加载 java 配置的其他的配置类。

ps:

- 1、一个 java 配置类加上另外一个配置类使用@Import(AppConfig1.class)，AppConfig1 可以不加@Configuration 注解
- 2、@ComponentScan(“com.springdemo.dao”)，AppConfig1 在 com.springdemo.dao 包下或者子包下，可以不使用@Import 注解，但是必须加@Configuration 注解

8.2.5. 测试

```
public class Test {  
  
    public static void main(String[] args) {  
  
        // 通过 Java 配置来实例化 Spring 容器
```



```

        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(SpringConfig.class);

        // 在 Spring 容器中获取 Bean 对象

        UserService userService = context.getBean(UserService.class);
    }

```

8.2.6. Java 配置和 xml 混用

```

@ImportResource("classpath:com/yiibai/core/applicationContext.xml")

@Configuration

@Import({ CustomerConfig.class, SchedulerConfig.class })

public class AppConfig {

}

```

8.2.7. 启动 web 项目直接加载配置类

在 web.xml 中直接加载类(带上@Configuration 注解的类)

```

<context-param>
    <param-name>contextClass</param-name>
    <param-value>
        org.springframework.web.context.support.AnnotationConfigWebApplicationContext
    </param-value>
</context-param>

<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>com.hy.ssh.spring.SpringConfig</param-value>
</context-param>

<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

```

8.2.8. 不要 web.xml

基于 servlet3.0 及以上版本。

只要实现 WebApplicationInitializer 接口，项目在启动的时候会自动加载此类。

```

public class AppConfig implements WebApplicationInitializer {

    public void onStartup(javax.servlet.ServletContext servletContext) throws ServletException {

        // 创建 Spring 的 root 配置环境

        AnnotationConfigWebApplicationContext rootContext =

            new AnnotationConfigWebApplicationContext();
    }
}

```

```

rootContext.register(JavaConfig.class);

// 将 Spring 的配置添加为 Listener
servletContext.addListener(new ContextLoaderListener(rootContext));

// 添加 struts2 的过滤器
FilterRegistration.Dynamic filter = servletContext.addFilter(
    StrutsPrepareAndExecuteFilter.class.getSimpleName(),
    StrutsPrepareAndExecuteFilter.class);
EnumSet<DispatcherType> dispatcherTypes = EnumSet
    .allOf(DispatcherType.class);
dispatcherTypes.add(DispatcherType.REQUEST);
dispatcherTypes.add(DispatcherType.FORWARD);

filter.addMappingForUrlPatterns(dispatcherTypes, true, "/*");

//springmvc 前端控制器
DispatcherServlet dispatcherServlet=new DispatcherServlet();
dispatcherServlet.setContextConfigLocation("classpath:springmvc.xml");

ServletRegistration.Dynamic
dynamic=servletContext.addServlet("dispatcherServlet",dispatcherServlet);
dynamic.addMapping("*.do");

}
}

```

8.3. 总结

这两种方式没有什么所谓的优劣之分，主要看使用情况，一般来说是这样：

涉及到全局配置的，例如数据库相关配置、MVC 相关配置等，就用 JAVA 配置的方式

涉及到业务配置的，就使用注解方式。

9. Spring 定时任务

9.1. 定时任务概述

在项目中开发定时任务应该是一种比较常见的需求了，在 Java 中开发定时任务主要有三种解决方案：1 使用 JDK 自带的 Timer 2 使用第三方组件 Quartz 3 使用 Spring Task。Timer 是 JDK 自带的定时任务工具，其简单易用，但是对于复杂的定时规则无法满足，在实际项目开发中也很少使用到，Quartz 功能强大但是使用起来相对笨重，而 Spring Task 则具备前两

者的优点（功能强大且简单易用），在这篇博客中我将介绍如何使用 Spring Task 进行定时任务开发。

9.2.Xml 配置

```
public interface TaskService {  
  
    public void taskOne();  
  
    public void taskTwo();  
  
}
```

```
public class TaskServiceImpl implements TaskService {  
  
    @Override  
    public void taskOne() {  
        System.out.println("这是我的第一个定时任务"+new Date());  
    }  
  
    @Override  
    public void taskTwo() {  
        System.out.println("这是我的第二个定时任务"+new Date());  
    }  
  
}
```

```
<!-- 配置定时规则 -->  
<task:scheduled-tasks>  
    <!-- 表示在服务器启动一秒后执行任务1 并且每隔1秒钟执行一次 -->  
    <task:scheduled ref="taskService" method="taskOne" initial-delay="1000" fixed-delay="1000" />  
    <!-- 表示在每天的13:40分0秒执行该任务 -->  
    <task:scheduled ref="taskService" method="taskTwo" cron="0 40 13 * * ?" />  
</task:scheduled-tasks>  
</beans>
```

9.3. 注解配置

9.3.1. 在需要进行定时任务的方法上添加注解

```
@Service
public class TaskServiceImpl implements TaskService {

    @Override//表示服务器启动两秒后启动该任务每隔一秒执行一次
    @Scheduled(initialDelay=2000,fixedDelay=1000)
    public void taskOne() {
        System.out.println("这是我的第一个定时任务"+new Date());
    }

    @Override
    @Scheduled(cron="*/5 * * * * ?")//每隔五秒钟执行一次
    public void taskTwo() {
        System.out.println("这是我的第二个定时任务"+new Date());
    }

}
```

9.3.2. 开启任务的注解扫描

```
<context:component-scan base-package="com.wistronits.service"></context:
component-scan>
<task:annotation-driven/>
</beans>
```

10. Spring 优点

Spring 能有效地组织你的中间层对象，无论你是否选择使用了 EJB。如果你仅仅使用了 Struts 或其他的包含了 J2EE 特有 APIs 的 framework，你会发现 Spring 关注了遗留下的问题。Spring 能消除在许多工程上对 Singleton 的过多使用。根据我的经验，这是一个主要的问题，它减少了系统的可测试性和面向对象特性。

Spring 能消除使用各种各样格式的属性定制文件的需要，在整个应用和工程中，可通过一种一致的方法来进行配置。曾经感到迷惑，一个特定类要查找迷幻般的属性关键字或系统属性，为此不得不读 Javadoc 乃至源代码吗？有了 Spring，你可很简单地看到类的 JavaBean 属性。

Spring 能通过接口而不是类促进好的编程习惯，减少编程代价到几乎为零。

Spring 被设计为让使用它创建的应用尽可能少的依赖于他的 APIs。在 Spring 应用中的大多数业务对象没有依赖于 Spring。所以使用 Spring 构建的应用程序易于单元测试。

Spring 能使 EJB 的使用成为一个实现选择，而不是应用架构的必然选择。你能选择用 POJOs 或 local EJBs 来实现业务接口，却不会影响调用代码。

Spring 帮助你解决许多问题而无需使用 EJB。Spring 能提供一种 EJB 的替换物，它们适于许多 web 应用。例如，Spring 能使用 AOP 提供声明性事务而不通过使用 EJB 容器，如果你仅仅需要与单个的数据库打交道，甚至不需要 JTA 实现。

Spring 为数据存取提供了一致的框架，不论是使用 JDBC 或 O/R mapping 产品（如 Hibernate）。

总结：

- 1.使用 Spring 的 IOC 容器，将对象之间的依赖关系交给 Spring，降低组件之间的耦合性，让我们更专注于应用逻辑
- 2.可以提供众多服务，事务管理，WS 等。
- 3.AOP 的很好支持，方便面向切面编程。
- 4.对主流的框架提供了很好的集成支持，如 Hibernate,Struts2,JPA 等
- 5.Spring DI 机制降低了业务对象替换的复杂性。
- 6.Spring 属于低侵入，代码污染极低。
- 7.Spring 的高度可开放性，并不强制依赖于 Spring，开发者可以自由选择 Spring 部分或全部