4.5 Spring之Spring IOC

4.5.1 概述

Spring是全栈式轻量级开源框架

两大核心：

1. 反转控制IOC：把对象的创建交给了Spring
2. 面向切片编程AOP：在不修改源码的情况下，对方法进行增强

4.5.2 Spring优势

耦合：程序之间的依赖

解耦合：降低程序之间的依赖

解决思路：配置文件+反射

4.5.4 IOC概念

主要作用：解耦合

控制：对象的创建，销毁

反转：把对象的控制权交给了Spring来完成，而不是开发者

4.5.5 ～ 7 自定义IOC容器

利用反射获取对象，解决编译期依赖的问题。

步骤：

1. 准备一个配置文件
2. 编写一个工厂工具类，工具类使用dom4j来解析配置文件，获取类全路径
3. 使用反射生成对应类的对象，存到map中（IOC容器）

4.5.9 Spring相关API

BeanFactory

Spring的底层接口，核心接口。

加载配置文件时，不会创建Bean对象；只有调用getBean方法时，才会真正创建Bean对象，存入IOC容器中

ApplicationContext

代表应用上下文。

加载配置文件时就会创建Bean对象，并存入IOC容器中

常用的实现类：

1. ClassPathXmlApplicationContext：它是从类的根路径下加载配置文件 推荐使用这种。
2. FileSystemXmlApplicationContext：它是从磁盘路径上加载配置文件，配置文件可以在磁盘的任意位置。
3. AnnotationConfigApplicationContext：当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。

4.5.10 bean标签配置及作用范围

默认情况下，调用无参构造函数

Bean标签的范围设置

取值范围	说明
singleton	默认值，单例的
prototype	多例的
request	WEB项目中，Spring创建一个Bean的对象，将对象存入到request域中
session	WEB项目中，Spring创建一个Bean的对象，将对象存入到session域中
global session	WEB项目中，应用在Portlet环境，如果没有Portlet环境那么globalSession 相当

1. 当scope的取值为singleton时

   Bean的实例化个数：1个

   Bean的实例化时机：当Spring核心文件被加载时，实例化配置的Bean实例

   Bean的生命周期：

   对象创建：当应用加载，创建容器时，对象就被创建了

   对象运行：只要容器在，对象一直活着

   对象销毁：当应用卸载，销毁容器时，对象就被销毁了
2. 当scope的取值为prototype时

   Bean的实例化个数：多个

   Bean的实例化时机：当调用getBean()方法时实例化Bean

   Bean的生命周期：

   对象创建：当使用对象时，创建新的对象实例

对象运行：只要对象在使用中，就一直活着

对象销毁：当对象长时间不用时，被 Java 的垃圾回收器回收了

4.5.11 Bean生命周期

```
<bean id="" class="" scope="" init-method="" destroy-method=""></bean>

\* init-method：指定类中的初始化方法名称

\* destroy-method：指定类中销毁方法名称
```

4.5.12 Bean实例化的三种方式

无参构造方式实例化

工厂静态方法实例化

工厂普通方法实例化

使用场景：

依赖的jar包中有个A类，A类中有个静态方法m1，m1方法的返回值是一个B对象。如果我们频繁使用B对象，此时我们可以将B对象的创建权交给spring的IOC容器，以后我们在使用B对象时，无需调用A类中的m1方法，直接从IOC容器获得。

4.5.13 依赖注入

DI：dependency infection是IOC的具体实现；就是通过框架把持久层对象传入业务层，而不用我们自己去获取。

4.5.14 有参构造方法注入

```
<bean id="userDao" class="com.zichen.dao.UserDaoInterfaceImp"></bean>
<bean id="userService" class="com.zichen.service.UserServiceInterfaceImp">
<!--<constructor-arg index="0" type="com.zichen.dao.UserDaoInterface"
ref="userDao"/>-->
     <constructor-arg name="userDao" ref="userDao"></constructor-arg>
</bean>
```

4.5.15 set方式注入

```
<bean id="userDao" class="com.zichen.dao.UserDaoInterfaceImp"></bean>
<bean id="userService" class="com.zichen.service.UserServiceInterfaceImp">
  <property name="userDao" ref="userDao"></property>
</bean>
```

name中的值为set后面的首字母小写单词；ref/value。set方法常用

4.5.16 普通数据类型注入

基本数据类型+String；property中的value标签对普通数据类型进行注入

```
<bean id="userDao" class="com.zichen.dao.UserDaoInterfaceImp">
  <property name="username" value="zichen"></property>
  <property name="age" value="28"></property>
</bean>
```

4.5.17 集合注入

set/array/list/map/properties：属性集，属性列表中每个键及其对应值都是一个字符串。

```
<bean id="userDao" class="com.zichen.dao.UserDaoInterfaceImp">
        <property name="list">
            <list>
                <value>zichen</value>
                <ref bean="user"></ref>
            </list>
        </property>
        <property name="set">
            <set>
                <value>zichen</value>
                <ref bean="user"></ref>
            </set>
        </property>
        <property name="array">
            <array>
                <value>zichen</value>
                <ref bean="user"></ref>
            </array>
        </property>
        <property name="map">
            <map>
                <entry key="key1" value="zichen"></entry>
                <entry key="key2" value-ref="user"></entry>
            </map>
```

```
        </property>
        <property name="properties">
            <props>
                <prop key="key1">zichen</prop>
            </props>
        </property>
</bean>
```

4.5.18 配置文件模块化

1. 标签：创建对象并放到spring的IOC容器

   id属性:在容器中Bean实例的唯一标识，不允许重复

   class属性:要实例化的Bean的全限定名

   scope属性:Bean的作用范围，常用是Singleton(默认)和prototype

2. 标签：属性注入

   name属性：属性名称

   value属性：注入的普通属性值

   ref属性：注入的对象引用值

3. 标签：属性注入

   name属性：属性名称

   value属性：注入的普通属性值

   ref属性：注入的对象引用值

4. 标签:导入其他的Spring的分配置文件

4.5.19 DBUtils回顾

DBUtils是Apache的一款简化Dao代码的工具类，它底层封装了JDBC技术。

核心对象

```
QueryRunner queryRunner = new QueryRunner(DataSource dataSource);
```

核心方法

```
int update(); 执行增、删、改语句

T query(); 执行查询语句
ResultSetHandler<T> 这是一个接口，主要作用是将数据库返回的记录封装到实体对象
```

4.5.20 ～24 IOC实战

4.5.25 Spring常用注解介绍

1. 相当于配置了bean标签，生成类的实例对象存到IOC容器当中。标注到哪个类上，就生成对应类的实例对象存到IOC容器中。

注解	说明
@Component	使用在类上用于实例化Bean
@Controller	使用在web层类上用于实例化Bean
@Service	使用在service层类上用于实例化Bean
@Repository	使用在dao层类上用于实例化Bean

2. 进行依赖注入，相当于配置property标签，但是利用反射实现的

注解	说明
@Autowired	根据类型完成依赖注入；注意：如果该类型匹配到多个实例对象，会报错
@Qualifier	结合@Autowired，根据名称进行依赖注入
@Resource	相当于@Autowired+@Qualifer，根据名称进行依赖注入；java11移除了@Resource，想要使用需手动添加dependency
@Value	注入普通属性

3. 相当于配置scope属性，指定bean的作用范围

注解	说明
@Scope	

4. init-method destory-method

注解	说明
@PostConstuct	
@PreDestory	

注意⚠️：

使用注解进行开发时，需要在applicationContext.xml中配置组件扫描，作用是指定哪个包及其子包下的Bean需要进行扫描以便识别使用注解配置的类、字段和方法

```
<!--注解的组件扫描-->
<context:component-scan base-package="com.zichen"></context:component-scan>
```

4.5.28 Spring新注解

使用上面的注解还不能全部替代xml配置文件，还需要使用注解替代的配置如下：

```
1. 非自定义的Bean的配置：<bean>

2. 加载properties文件的配置：<context:property-placeholder>

3. 组件扫描的配置：<context:component-scan>

4. 引入其他文件：<import>
```

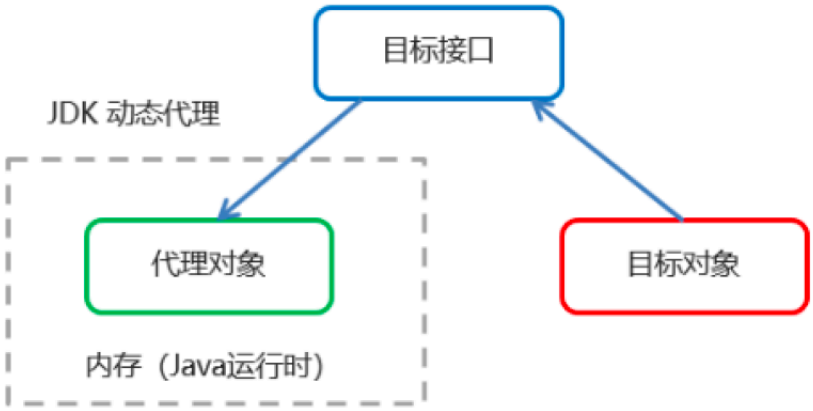
4.6 Spring之Spring AOP

4.6.8 ~ 10 使用动态代理技术

我们可以通过动态代理将业务代码和事务代码进行划分，降低耦合。通过调用代理对象来实现方法的增强。

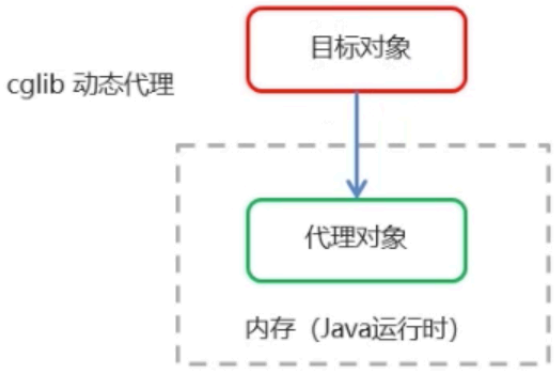
JDK动态代理：基于接口的动态代理(参考spring_transfer)

1. 目标类/目标对象（被代理类）-- 要动态增加方法的那个类
2. 目标对象至少要实现一个接口
3. java运行时，代理对象是根据目标接口生成的
4. 代理对象和目标对象是同级的，所以方法的返回值必须是接口类型
5. 采用JDK动态代理技术生成目标类的代理对象
6. 通过调用代理对象来实现方法的增强
7. 原理：利用拦截器（必须实现invocationHandler）加反射生成一个代理接口的匿名类，在调用具体方法**前**都会调用invocationHanlder的invoke方法，从而显示方法的增强



CGLIB动态代理：基于父类的动态代理

动态生成一个要代理的子类，子类重写要代理的类的所有不是final的方法。在子类中采用方法拦截技术拦截所有的父类方法的调用，顺势织入横切逻辑，对方法进行增强



4.6.11 AOP概念

面向切面编程 Aspect Oriented Programming：对业务逻辑的各个部分进行隔离

好处：

1. 在程序运行期间，在不修改源码的情况下对方法进行功能增强
2. 逻辑清晰，开发核心业务的时候，不必关注增强业务的代码
3. 减少重复代码，提高开发效率，便于后期维护

底层实现：

AOP的底层是通过Spring提供的动态代理技术实现的。在运行期间，Spring通过动态代理技术动态的生成代理对象，代理对象方法去调用目标对象的方法时，会进行增强功能的介入，从而完成功能的增强。

4.6.12 AOP相关术语

AOP的底层实现是**动态代理**；并通过**配置的方式**来完成指定目标的方法增强。

相关术语	解释
Target	目标对象/被代理类：需要增强方法的那个类
Proxy	代理：生成的代理对象
Joinpoint	连接点：**可以**被拦截增强的方法
Pointcut	切入点：**真正**被拦截增强的方法
Advice	通知：增强的业务逻辑。前置通知，后置通知，异常通知，最终通知，环绕通知
Aspect	切面：切入点和通知结合的过程
Weaving	织入：是把增强应用到目标对象来创建的代理对象的过程

4.6.13 AOP应用注意事项

开发阶段：我们做的

1. 编写核心业务代码-切入点
2. 把通用代码抽取出来，制成通知-通知
3. 在配置文件中，声明切入点和通知之间的关系-切面

运行阶段：Spring完成的

Spring监控切入点方法的执行，一旦监控到切入点被运行，使用代理机制，动态创建目标对象的代理对象，根据通知类型，在代理对象的的对应位置，将通知对应的功能织入，完成完整的逻辑运行。

底层实现：

1. 当bean实现接口，采用JDk动态代理
2. 当bean没有实现接口，采用cglib动态代理

4.6.14 基于XML方式的AOP开发

4.6.15 切点表达式详解

execution([修饰符] 返回值类型 包名.类名.方法名(参数类型))

```
<aop:before method="before" pointcut="execution(public void
com.zichen.service.impl.AccountServiceImpl.transfer
())">
</aop:before>
```

1. 访问修饰符可以省略
2. 返回值类型，包名，类名，方法名可以使用*来代替

3. 包名与类型之间一个点 . 代表当前包下的类；两个点 .. 表示当前包及其子包下的类
4. 参数列表可以使用两个点 .. 表示任意个数，任意类型的参数列表

```
<aop:before method="before" pointcut="execution(*
com.zichen.service.impl.AccountServiceImpl.*(..))"></aop:before>
```

切点表达式抽取：

```
<aop:config>
  <aop:pointcut id="myPointCut" expression="execution(*
com.zichen.service.impl.AccountServiceImpl.*(..))"/>
  <aop:aspect ref="myAdvice">
    <aop:before method="before" pointcut-ref="myPointCut"></aop:before>
    <aop:after-returning method="after" pointcut-ref="myPointCut"></aop:after-
returning>
  </aop:aspect>
</aop:config>
```

4.6.16 通知类型详解

```
<aop:通知类型 method=“通知类中方法名” pointcut=“切点表达式"></aop:通知类型>
```

aop:before	用于配置前置通知。指定增强的方法在切入点方法之前执行
aop:afterReturning	用于配置后置通知。指定增强的方法在切入点方法之后执行
aop:afterThrowing	用于配置异常通知。指定增强的方法出现异常后执行
aop:after	用于配置最终通知。无论切入点方法执行时是否有异常，都会执行
aop:around	用于配置环绕通知。开发者可以手动控制增强代码在什么时候执行；通常独立使用

4.6.17 基于注解的AOP开发

4.6.19 注解配置AOP详解_通知类型

注解配置有一个bug：@Before -> @After -> @AfterReturning (如果有异常：@AfterThrowing)

一般情况下还是采用xml配置或者使用@Around

4.7 Spring之JdbcTemplate&事务声明

4.7.1 JdbcTemplate概述

Spring框架中提供的一个模版对象，是对原始繁琐的Jdbc API对象的简单封装

核心对象

```
JdbcTemplate jdbcTemplate = new JdbcTemplate(DataSource dataSource);
```

核心方法

```
int update(); 执行增、删、改语句
List<T> query(); 查询多个
T queryForObject(); 查询一个
  new BeanPropertyRowMapper<>(); 实现ORM映射封装
```

4.7.8 Spring事务控制及PlatformTransactionManager

Spring事务控制可以分为**编程式事务控制**和**声明式事务控制**

~~编程式事务控制：开发者直接把事务代码和业务代码耦合在一起，在实际开发中不用~~

声明式事务控制：开发者采用配置的方式来实现事务的控制，业务代码和事务代码实现解耦合，使用AOP的思想

PlatformTransactionManager接口，是Spring的事务管理器，里面提供了我们常用的操作事务的方法。而且不同的Dao层技术有不用的实现类：

Dao层技术是jdbcTemplate或mybatis时：DataSourceTransactionManager，需要借助connection

Dao层技术是hibernate时：HibernateTransactionManager

Dao层技术是JPA时：JpaTransactionManager

4.7.9 编程式事务控制_TransactionDefinition

是一个接口，提供事务的定义信息（事务隔离级别，事务传播行为等）

1. 事务隔离级别：设置隔离级别可以解决事务并发的问题，如脏读，不可重复读，虚读（幻读）

问题	说明
脏读	事务A读取事务B尚未提交的数据，此时如果事务B发生错误并执行回滚，那么事务A读取到的数据就是脏数据
不可重复读	同一事务，前后多次读取数据，数据内容不一致
幻读	同一事务，前后多次读取，数据总量不一致

隔离级别	说明
isolation_default	使用数据库默认级别
isolation_read_uncommitted	读未提交(什么问题都解决不了)
isolation_read_committed	读已提交(解决脏读的问题)
isolation_repeatable_read	可重复读(解决脏读，不可重复读；mysql默认隔离级别)
isolation_serializable	串行化(脏读，不可重复读，虚读)

2. 事务传播行为值的是当一个业务方法被另一个业务方法调用时，应该如何控制事务

传播行为	说明
requrired	如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。一般的选择（默认值）；当前被调用的方法必须要进行事务控制。**增，删，改**
supports	支持当前事务，如果当前没有事务，就以非事务方式执行（没有事务）；当前调用的方法有没有事务都可以执行。**查**

3. read-only（是否只读）：建议查询时设为只读
4. timeout（超时时间）：默认是-1，没有超时时间。如果有，以秒为单位进行设置

4.7.10 编程式事务控制_TransactionStatus

TransactionStatus接口提供的事务具体的运行状态

总结：

事务管理器（PlatformTransactionManager）通过读取**事务定义参数（TransactionDefinition）**进行事务管理，然后产生一系列的**事务状态（TransactionStatus）**

4.7.11 基于XML的声明试事务控制

底层是AOP，通过配置来控制事务。

切入点，通知，切面

4.7.12 事务配置参数详解

```
<tx:method name="transfer" isolation="REPEATABLE_READ" propagation="REQUIRED"
timeout="-1" read-only="false"/>
* name：切点方法名称
* isolation:事务的隔离级别
* propogation：事务的传播行为
* timeout：超时时间
* read-only：是否只读
```

*为通配符；

```
<tx:attributes>
  <tx:method name="save*" propagation="REQUIRED"/>
  <tx:method name="delete*" propagation="REQUIRED"/>
  <tx:method name="update*" propagation="REQUIRED"/>
  <tx:method name="find*" read-only="true"/>
  <tx:method name="*"/>
</tx:attributes>
```

4.7.14 基于纯注解的声明式事务控制（spring_transfer_tx）

```
* 平台事务管理器配置（xml、注解方式）
* 事务通知的配置（@Transactional注解配置）
* 事务注解驱动的配置 <tx:annotation-driven/>、@EnableTransactionManagement
```

4.7.15 - 16 Spring集成web环境（重看）

应用上下文对象是通过 new ClasspathXmlApplicationContext(spring配置文件) 方式获取的，但是每次从容器中获得Bean时都要编写 new ClasspathXmlApplicationContext(spring配置文件) ，这样的弊端是配置文件加载多次，应用上下文对象创建多次。

解决思路分析：

在Web项目中，可以使用ServletContextListener监听Web应用的启动，我们可以在Web应用启动时，就加载Spring的配置文件，创建应用上下文对象ApplicationContext，在将其存储到最大的域servletContext域中，这样就可以在任意位置从域中获得应用上下ApplicationContext对象了

上面的分析不用手动实现，Spring提供了一个监听器ContextLoaderListener就是对上述功能的封装，该监听器内部加载Spring配置文件，创建应用上下文对象，并存储到ServletContext域中，提供了一个客户端工具WebApplicationContextUtils供使用者获得应用上下文对象。

所以我们需要做的只有两件事：

1. 在web.xml中配置ContextLoaderListener监听器（导入spring-web坐标）
2. 使用WebApplicationContextUtils获得应用上下文对象ApplicationContext