# 1、Spring AOP设计原理及具体实践

大白话：面向切面编程（规则），开发时将代码拆分，运行时在将代码合并（解耦，无缝拼接）

AOP是OOP的延续，是Aspect Oriented Programming的缩写，意思是面向切面编程。可以通过预编译方法

**切面**：一个切面，代表N个Bean的一个集合，这N个Bean都拥有共同点，所以他们组成一个切面。在ApplicationContext中<aop:aspect>来配置。事务管理的时候，就用到了切面的定义，在配置中配置一个包的路径，这个包下面的所有类都是处理的事务的共同特性。

**连接点**：程序执行过程中的某一行为，报异常回滚

**通知**：“切面”对于某个“连接点”所产生的的动作。通知利用了IOC中的后置处理

**切入点**：切面中某一个具体的Bean中的某一个具体方法，由切入点表达式决定的

execution(\* com.spring.service.\*.\*(..))来决定的。

**目标对象**：被一个或者多个切面所通知的对象。就是代理对象所持有的引用。

**AOP代理**：持有被代理对象的引用，在调用被代理方法时，在调用之前加点东西，在调用之后加点东西，中间就用我们自己保存的对象引用去调用具体的方法

方法拦截器：

切面是所有类有一样的规则

连接点是所有方法调用有一样的规则

切入点是进入切面内部的一个入口

一旦调用过程中，满足连接点的规则，那么就会触发一个通知：调用代理写的代码

## 通知类型：

**前置通知**：在某连接点之前执行的通知，但这个通知不能阻止连接点前的执行。ApplicationContext中在<aop:aspect>里面使用<aop:before>元素进行声明。

**后置通知**：当某个连接点退出的时候执行的通知

**返回后通知**：在某连接点正常完成后执行的通知，不包括抛出异常的情况。

**环绕通知**：包围一个连接点的通知，类似Web中Servlet规范中的Filter的doFiler方法。可以在方法的调用前后完成自定义行为，也可以选择不执行。使用<aop:around>元素进行声明

**抛出异常后通知**：在方法抛出异常退出时执行的通知。

注：可以将多个通知应用到一个目标对象上，即可以将多个切面织入到同一个目标对象。

## 手写AOP例子

### 1、2.5以前的例子

切面类：

package main.java.vip.aop.aspect;

import org.aopalliance.intercept.Joinpoint;

import org.apache.log4j.Logger;

public class LogAspect {

private final static Logger LOG = Logger.getLogger(LogAspect.class);

public void before(Joinpoint joinpoint){

LOG.info("调用方法之前执行" + joinpoint);

}

public void after(Joinpoint joinpoint){

LOG.info("调用方法之后执行" + joinpoint);

}

public void afterReturn(Joinpoint joinpoint){

LOG.info("调用方法获得返回值之后执行" + joinpoint);

}

public void afterThrow(Joinpoint joinpoint){

LOG.info("抛出异常之后执行" + joinpoint);

}

}

被修饰的类：

package main.java.vip.aop.service;

import org.apache.log4j.Logger;

import org.springframework.stereotype.Service;

import main.java.vip.model.Member;

@Service

public class MemberManagerService {

private final static Logger LOG = Logger.getLogger(MemberManagerService.class);

public boolean add(Member member){

LOG.info("增加用户");

return true;

}

public boolean remove(long id) throws Exception{

LOG.info("删除用户");

throw new Exception("这是我们自己跑出的异常");

}

public boolean modify(Member member) {

LOG.info("修改用户");

return true;

}

public boolean query(String loginName) {

LOG.info("查询用户");

return true;

}

}

设置切面application-aop.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" xmlns:mvc="http://www.springframework.org/schema/mvc"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:aop="http://www.springframework.org/schema/aop" xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.2.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.2.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx-3.2.xsd ">

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

<!-- 声明一个需要织入到虚拟切面的逻辑（切面） -->

<bean id="logAspect" class="main.java.vip.aop.aspect.LogAspect"></bean>

<aop:config>

<aop:aspect ref="logAspect">

<!-- 切点（具体的方法声明） -->

<!-- Spring表达式的强大功能 -->

<aop:pointcut expression="execution(\* main.java.vip.aop.service..\*(..))" id="logPointcut"/>

<aop:before method="before" pointcut-ref="logPointcut" />

<aop:after-returning method="afterReturn" returning="boolean" pointcut-ref="logPointcut" />

<aop:after method="after" pointcut-ref="logPointcut" />

<aop:after-throwing method="afterThrow" pointcut-ref="logPointcut" />

</aop:aspect>

</aop:config>

</beans>

测试类：

package test.java.vip.aop.service;

import main.java.vip.aop.service.MemberManagerService;

import main.java.vip.model.Member;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.test.context.ContextConfiguration;

import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@ContextConfiguration(locations={"classpath\*:application-context.xml"})

@RunWith(SpringJUnit4ClassRunner.class)

public class MemberManagerServiceTest {

@Autowired MemberManagerService managerService;

@Test

public void testAdd(){

managerService.add(null);

}

public void testRemove(long id){

try {

managerService.remove(id);

} catch (Exception e) {

e.printStackTrace();

}

}

public void testModify(Member member) {

managerService.modify(member);

}

public void testQuery(String loginName) {

managerService.query(loginName);

}

}

### 2、2.5之后的注解开发

切点类：

package main.java.vip.aop.aspect;

import org.aopalliance.intercept.Joinpoint;

import org.apache.log4j.Logger;

import org.aspectj.lang.annotation.After;

import org.aspectj.lang.annotation.AfterReturning;

import org.aspectj.lang.annotation.AfterThrowing;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.aspectj.lang.annotation.Pointcut;

import org.springframework.stereotype.Component;

@Component //声明这个类是被SpringIOC容器来管理的，如果不声明，就无法做到自动化织入

@Aspect //这个类被生明为是一个需要动态织入到我们的虚拟切面中的类

public class AnnotationAspect {

private final static Logger LOG = Logger.getLogger(AnnotationAspect.class);

//声明切点

//应为要利用反射机制去读取这个切面的所有注解信息

@Pointcut("execution(\* main.java.vip.aop.service..\*(..))")

public void pointcutConfig(){}

@Before("pointcutConfig()")

public void before(Joinpoint joinpoint){

LOG.info("调用方法之前执行" + joinpoint);

}

@After("pointcutConfig()")

public void after(Joinpoint joinpoint){

LOG.info("调用方法之后执行" + joinpoint);

}

@AfterReturning("pointcutConfig()")

public void afterReturn(Joinpoint joinpoint){

LOG.info("调用方法获得返回值之后执行" + joinpoint);

}

@AfterThrowing("pointcutConfig()")

public void afterThrow(Joinpoint joinpoint){

LOG.info("抛出异常之后执行" + joinpoint);

}

}

测试类：

package test.java.vip.aop.service;

import main.java.vip.aop.service.MemberManagerService;

import main.java.vip.model.Member;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.test.context.ContextConfiguration;

import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@ContextConfiguration(locations={"classpath\*:application-context.xml"})

@RunWith(SpringJUnit4ClassRunner.class)

public class MemberManagerServiceTest {

@Autowired MemberManagerService managerService;

@Test

public void testAdd(){

managerService.add(null);

}

public void testRemove(long id){

try {

managerService.remove(id);

} catch (Exception e) {

e.printStackTrace();

}

}

public void testModify(Member member) {

managerService.modify(member);

}

public void testQuery(String loginName) {

managerService.query(loginName);

}

}

## execution表达式的含义

execution(\* main.java.vip.aop.service..\*(..))

权限修饰符（public/private/protected可省略） 返回值（void/自定义类型） 属于哪个包哪个类.方法名称(参数列表) throws 异常类型

\* 代表任意

. 可以省略包

execution(public void com.java.Test..\*(String, String))

## AOP到底能干啥

Authentication 权限

Caching 缓存

Debugging 调试

Logging 日志

Transaction Manager 事务管理

Mointer 监听

Intercepter 拦截器

Context Passing内容传递

监听的缓存结果，第一次调用的时候，就把结果存到缓存中，那么第二次调用，就不再调用那个实际的

# 2、Spring AOP源码解析

Spring AOP执行流程

AOP不用在执行IOC的操作了，只要能够拿到IOC容器的引用，直接从IOC容器中取出需要被二次操作的所有对象。

IOC和AOP的工厂并不是同一个工厂，他们做的事不同。

AOP依赖于IOC

AOP最核心的是代理：JDK代理和CGLib代理

Spring提供了两种方式来生成代理对象：JDKProxy和Cglib，具体使用哪种方式生成由AopProxyFactory根据AdviseSupport对象的配置来决定。默认的策略是如果目标类是接口，则使用JDK动态代理技术，否则使用Cglib来生成代理。下面我们来研究一下Spring如何使用JDK来生成代理对象，具体生成代码放在JdkDynamicAopProxy这个类中，直接上相关代码：

public Object getProxy(ClassLoader classLoader) {

if (logger.isDebugEnabled()) {

logger.debug("Creating JDK dynamic proxy: target source is " + this.advised.getTargetSource());

}

Class[] proxiedInterfaces = AopProxyUtils.completeProxiedInterfaces(this.advised);

findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);

return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);

}

那这个其实根明了，注释上已经写清楚了，不在赘述。

下面的问题是，代理对象生成了，那切面是如何织入的？

我们知道InvocationHandler是JDK动态代理的核心，生成的代理对象的方法调用都会委托到InvocationHandler，invoke()方法。而通过JdkDynamicAopProxy的签名我们可以看到这个类其实也实现了InvocationHandler，下面我们就通过分析这个类中实现的invoke()方法来具体看下Spring AOP是如何织入切面的：

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

MethodInvocation invocation;

Object oldProxy = null;

boolean setProxyContext = false;

TargetSource targetSource = this.advised.targetSource;

Class targetClass = null;

Object target = null;

try {

if (!this.equalsDefined && AopUtils.isEqualsMethod(method)) {

// The target does not implement the equals(Object) method itself.

return equals(args[0]);

}

if (!this.hashCodeDefined && AopUtils.isHashCodeMethod(method)) {

// The target does not implement the hashCode() method itself.

return hashCode();

}

if (!this.advised.opaque && method.getDeclaringClass().isInterface() &&

method.getDeclaringClass().isAssignableFrom(Advised.class)) {

// Service invocations on ProxyConfig with the proxy config...

return AopUtils.invokeJoinpointUsingReflection(this.advised, method, args);

}

Object retVal;

if (this.advised.exposeProxy) {

// Make invocation available if necessary.

oldProxy = AopContext.setCurrentProxy(proxy);

setProxyContext = true;

}

// May be null. Get as late as possible to minimize the time we "own" the target,

// in case it comes from a pool.

target = targetSource.getTarget();

if (target != null) {

targetClass = target.getClass();

}

// Get the interception chain for this method.

List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);

// Check whether we have any advice. If we don't, we can fallback on direct

// reflective invocation of the target, and avoid creating a MethodInvocation.

if (chain.isEmpty()) {

// We can skip creating a MethodInvocation: just invoke the target directly

// Note that the final invoker must be an InvokerInterceptor so we know it does

// nothing but a reflective operation on the target, and no hot swapping or fancy proxying.

retVal = AopUtils.invokeJoinpointUsingReflection(target, method, args);

}

else {

// We need to create a method invocation...

invocation = new ReflectiveMethodInvocation(proxy, target, method, args, targetClass, chain);

// Proceed to the joinpoint through the interceptor chain.

retVal = invocation.proceed();

}

// Massage return value if necessary.

Class<?> returnType = method.getReturnType();

if (retVal != null && retVal == target && returnType.isInstance(proxy) &&

!RawTargetAccess.class.isAssignableFrom(method.getDeclaringClass())) {

// Special case: it returned "this" and the return type of the method

// is type-compatible. Note that we can't help if the target sets

// a reference to itself in another returned object.

retVal = proxy;

} else if (retVal == null && returnType != Void.TYPE && returnType.isPrimitive()) {

throw new AopInvocationException("Null return value from advice does not match primitive return type for: " + method);

}

return retVal;

}

finally {

if (target != null && !targetSource.isStatic()) {

// Must have come from TargetSource.

targetSource.releaseTarget(target);

}

if (setProxyContext) {

// Restore old proxy.

AopContext.setCurrentProxy(oldProxy);

}

}

}

主流程可以简述为：获取可以应用到此方法上的通知链，如果有，则应用通知，并执行joinpoint；如果没有，则直接反射执行joinpoint。而这里的关键是通知链是如何获取的以及它又是如何执行的，

获取通知链的方法：

public List<Object> getInterceptorsAndDynamicInterceptionAdvice(Method method, Class targetClass) {

MethodCacheKey cacheKey = new MethodCacheKey(method);

List<Object> cached = this.methodCache.get(cacheKey);

if (cached == null) {

cached = this.advisorChainFactory.getInterceptorsAndDynamicInterceptionAdvice(

this, method, targetClass);

this.methodCache.put(cacheKey, cached);

}

return cached;

}

public List<Object> getInterceptorsAndDynamicInterceptionAdvice(

Advised config, Method method, Class targetClass) {

// This is somewhat tricky... we have to process introductions first,

// but we need to preserve order in the ultimate list.

List<Object> interceptorList = new ArrayList<Object>(config.getAdvisors().length);

boolean hasIntroductions = hasMatchingIntroductions(config, targetClass);

AdvisorAdapterRegistry registry = GlobalAdvisorAdapterRegistry.getInstance();

for (Advisor advisor : config.getAdvisors()) {

if (advisor instanceof PointcutAdvisor) {

// Add it conditionally.

PointcutAdvisor pointcutAdvisor = (PointcutAdvisor) advisor;

if (config.isPreFiltered() || pointcutAdvisor.getPointcut().getClassFilter().matches(targetClass)) {

MethodInterceptor[] interceptors = registry.getInterceptors(advisor);

MethodMatcher mm = pointcutAdvisor.getPointcut().getMethodMatcher();

if (MethodMatchers.matches(mm, method, targetClass, hasIntroductions)) {

if (mm.isRuntime()) {

// Creating a new object instance in the getInterceptors() method

// isn't a problem as we normally cache created chains.

for (MethodInterceptor interceptor : interceptors) {

interceptorList.add(new InterceptorAndDynamicMethodMatcher(interceptor, mm));

}

}

else {

interceptorList.addAll(Arrays.asList(interceptors));

}

}

}

}

else if (advisor instanceof IntroductionAdvisor) {

IntroductionAdvisor ia = (IntroductionAdvisor) advisor;

if (config.isPreFiltered() || ia.getClassFilter().matches(targetClass)) {

Interceptor[] interceptors = registry.getInterceptors(advisor);

interceptorList.addAll(Arrays.asList(interceptors));

}

}

else {

Interceptor[] interceptors = registry.getInterceptors(advisor);

interceptorList.addAll(Arrays.asList(interceptors));

}

}

return interceptorList;

}

切点：由切点进入切面，切点实际上就是Method，通知的动作。切点，转换为MethodInterceptor，保存到一个容器里面，这个容器一定是一个链表结构，一定是有顺序的，它知道他的上一个是谁，它的下一个是谁

这个方法执行完成后，Advised中配置能够应用到连接点或者目标类的Advisor全部被转化成了MethodInterceptor。

接下来我们在看下得到的拦截器链是怎么起作用的（在JdkDynamicAopProxy类中的invoke方法）。

List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);

if (chain.isEmpty()) {

如果没有可以引用到此方法的通知，此直接反射调用

retVal = AopUtils.invokeJoinpointUsingReflection(target, method, args);

}

else {

// 创建MethodInvocation

invocation = new ReflectiveMethodInvocation(proxy, target, method, args, targetClass, chain);

retVal = invocation.proceed();

}

从这段代码可以看出，如果得到的拦截器链为空，则直接反射调用目标方法，否则创建MethodInvocation，调用其proceed方法，触发器拦截器链的执行，来看下具体代码：

public Object proceed() throws Throwable {

// We start with an index of -1 and increment early.

if (this.currentInterceptorIndex == this.interceptorsAndDynamicMethodMatchers.size() - 1) {

return invokeJoinpoint();

}

Object interceptorOrInterceptionAdvice =

this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex);

if (interceptorOrInterceptionAdvice instanceof InterceptorAndDynamicMethodMatcher) {

// Evaluate dynamic method matcher here: static part will already have

// been evaluated and found to match.

InterceptorAndDynamicMethodMatcher dm =

(InterceptorAndDynamicMethodMatcher) interceptorOrInterceptionAdvice;

if (dm.methodMatcher.matches(this.method, this.targetClass, this.arguments)) {

return dm.interceptor.invoke(this);

}

else {

// Dynamic matching failed.

// Skip this interceptor and invoke the next in the chain.

return proceed();

}

}

else {

// It's an interceptor, so we just invoke it: The pointcut will have

// been evaluated statically before this object was constructed.

return ((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(this);

}

}

在这个通知链表中，前面节点如果执行错误，以后的节点就不执行了。

## 总结：

1. 加载配置信息，解析成AopConfig
2. 交给AopProxyFactory，调用一个createAopProxy的方法
   1. JdkDynamicAopProxy调用AdvisedSupport的getInterceptorsAndDynamicIntercaption

Advise方法得到的方法拦截器，并保存到一个容器（List，链表）

3、递归执行拦截器方法proceed

# 3、Spring JDBC设计原理及二次开发

它是封装了JDBC操作的一个框架，必须依赖Spring。Spring JDBC基于模板模式来开发的。

Mybatis是一个半自动的ORM框架

Hibernate是一个全自动的ORM框架

Spring JDBC是一个手动的ORM框架

SpringJDBC采用的是Template设计模式，指定义了一个RowMapper的接口mapping方法，这个方法是没有实现的

Mybatis半自动：mapping.xml文件如果你的类结构根数据表结构完全一致，他就能帮你用反射机制自动匹配成一个java对象

## jdbc执行流程：

1. 加载驱动类（基于Mysql的）
2. 获取连接（被封装到DataSource里面去了）
3. 创建语句集（预处理语句集合标准语句集）
4. 执行语句集（执行事务操作）
5. 获取结果集（如果是增删改，拿到一个int值，影响行数，如果查询，就会拿到一个Result）

手写JDBC框架，做两件事：

1. 实现单表操作实现NoSql
2. 把手动ORM编程自动ORM

Spring JDBC是依赖于Spring，而Spring是一个外能胶。

ORM：

将数据库查询的结果，映射成一个我们自己定义的类

## 手写一个简单的ORM：

import java.lang.reflect.Field;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.util.ArrayList;

import java.util.List;

public class JdbcTest {

public static void main(String[] args) {

//原生JDBC如何操作？

try {

//1、加载驱动类

Class.forName("com.mysql.jdbc.Driver");

//2、建立连接

Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/1232069494", "root", "");

//3、创建语句集

PreparedStatement pstm = con.prepareStatement("select \* from test");

//4、执行

ResultSet rs = pstm.executeQuery();

//5、获取结果集

int len = rs.getMetaData().getColumnCount();

List<Object> result = new ArrayList<Object>();

while(rs.next()){

Class clazz = Member.class;

Object obj = clazz.newInstance();

for(int i = 1; i <= len; i++){

String columnName = rs.getMetaData().getColumnName(i);

Field field = clazz.getDeclaredField(columnName);

field.setAccessible(true);

Object type = field.getType();

if(type == Long.class){

field.set(obj, rs.getLong(columnName));

} else if(type == String.class){

field.set(obj, rs.getString(columnName));

}

}

result.add(obj);

System.out.println(obj.toString());

}

for(Object o : result){

Member m = (Member)o;

m.toString();

}

rs.close();

pstm.close();

con.close();

} catch (Exception e) {

}

}

}

public class Member {

private int id;

private String pc;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getPc() {

return pc;

}

public void setPc(String pc) {

this.pc = pc;

}

@Override

public String toString() {

return "Member [id=" + id + ", pc=" + pc + "]";

}

}

## 使用Spring JDBC完成一个ORM

Hibernate优点：

1. API丰富，可以实现无SQL操作（HQL），为了兼容所有数据库（都会解释为HQL）
2. ORM全自动化

Mybatis优点：

1. 轻量级，性能好
2. Sql和Java代码分离（SqlMap，把每一条SQL语句起一个名字，作为Map的Key保存）

我们想要做的ORM特性：

第一，性能要好，是啥就是啥，不经过二次处理（不对SQL语句进行二次包装）

第二，单表操作实现NoSQL（最终生成的是一个字符串）

第三，ORM零配置实现自动化（利用反射机制，把字段和属性对应上，然后，自定实例化返回结果）

原则：约定优于配置（保证代码健壮性）

一个DAO只操作一张表

约定：做修改和删除的时候是根据主键来操作的

约定：尽量使用单表操作，如果实在要多表操作，可以先把数据查出来放到内存，然 后再内存中进行计算

约定：支持读写分离

约定：支持分库分表

约定：所有ORM支持的类型原则上只认java八大基本数据类型 + String（为了降低 复杂度）

对象状态：临时态、持久态、删除态、游离态

防止Sql注入：

Spring 框架防止Sql注入使用的是preparedStatement预处理语句集

Spring的JDBC是 JDBCTemplate类

# 4、Spring MVC

SpringMVC的流程：

初始化过程

1. web.xml配置一个DispatcherServlet（启动的入口）实现Awaer接口，能够得到ApplicationContext
2. 默认加载我们的IOC容器（ApplicationContext）

3、开始扫描SpringMVC的配置，一般来说（扫描注解），View的配置，插件（拦截器、转换器、视图解析器）

4、解析成一个HandlerMapping的List。主要是保存了url和具体的执行的对应关系

等待用户请求过程：

1、从浏览器中输入URL

2、统一拦截，400/500等

3、DispatcherServlet接收到请求，从上面初始化已经保存的数据中找到请求url对应方法然后调用

4、把响应结果输出

正常的三层架构：

Dao数据访问层

Service业务逻辑层

Web层（j2EE的内容）

Request和Response

MVC是Web针对Web层的框架  
M：Model

V：View

C：Controller

servlet的流程：

1、从web.xml文件开始在这个文件里面配置了N个Servlet。一般而言一个Servlet对用一个url以后要增加功能，增加url，每次都要去修改配置文件增加Servlet配置，导致配置膨胀，代码膨胀返回结果，在Servlet里面是直接输入HTML的代码。

启发：ORM框架（自动化和半自动化框架）

MVC：视图和Java逻辑分离（隔离配置）

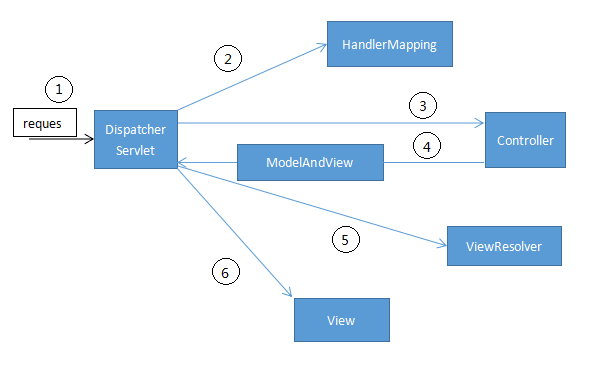
Jsp：归根到底还是一个Servlet，只不过是自动生成了HTML代码而已

MVC框架：

第一个MVC框架WebWork，增加需求，增加URL，就不需要频繁修改web.xml只需要增加class就行

支持模板化开发：渐渐地把JSP退出历史舞台FreeMark（自己的语法，不能写java代码，支持标签）

## 1、SpringMVC和核心图



1. DispatcherServlet是SpringMVC中的前端控制器（front controller），负责接收request并将request装发给对应的组件
2. HanderMapper是SpringMVC中完成url到controller映射的组件，DispatcherServlet接收request，然后从HandlerMapping查找处理request的controller
3. Controller处理request，并返回ModelAndView对象，Controller是springMVC中负责处理request的组件（类似struts2中的Action），ModelAndView是封装结果视图的组件
4. 下同
5. 下同
6. 视图解析器解析ModelAndView对象并返回对应的视图给客户端

## 2、SpringMVC的工作机制

在容器初始化时会建立所有url和controller的对应关系，保存到Map<url,controller>中，tomcat启动时，会通知Spring初始化容器（加载bean的定义信息和初始化所有单例bean），然后SpringMVC会遍历容器中的bean，获取每一个controller中的所有方法访问的url，然后将url和controller保存到一个Map中。

这样就可以根据request快速定位到controller，因为最终处理request的是controller中的方法，Map中只保留了url和controller中的对应关系，所以要根据request的url进一步确认controller中的method，这一步工作的原理就是拼接controller的url（controller上@RequestMapping的值）和方法的url（method上@RequestMapping的值），与request的url进行匹配，找到匹配的哪个方法；

确定处理请求的method后，解下来的任务就是参数绑定，把request中参数绑定到方法的形式参数上，这一步是整个请求处理中最复杂的一个步骤。SpringMVC提供了两种request参数和方法形参的绑定方法：

1. 通过注解进行绑定，@RequestParam
2. 通过参数名称进行绑定

使用注解进行绑定，我们只要在方法参数前面声明@RequestParam(“a”)，就可以将request中参数a的值绑定到方法的该参数上，使用参数名称进行绑定的前提是必须获取方法中参数的名称，Java反射只提供了获取方法的参数的类型，并没有提供获取参数名称的方法，SpringMVC解决了这个问题的方法是用asm框架读取字节码文件，来获取方法的参数名称，asm框架是一个字节码操作框架，关于asm更多介绍可以参考它的官网。

## 3、Spring MVC源码分析

程序查看入口为DispatcherServlet类

我们根据工作机制中三部分来分析springmvc的源代码

其一、ApplicationContext初始化建立所有url和controller类的对应关系（用Map保存）

其二、根据请求url找到对应的controller，并从controller中找到处理器请求的方法

其三、request参数绑定到方法的形参，执行方法处理请求，并返回结果视图

### 第一步、建立Map<urls,controller>的关系、初始化

我们首先看第一步骤，也就是建立Map<url,controller>关系的部分，第一部分的入口类为ApplicationObjectSupport的setApplicationContext方法，setApplicationContext方法中核心部分就是初始化容器initApplicationContext(context)，子类AbstractDetectingURLHandlerMapping实现了该方法，所以我们直接看子类中的初始化容器方法。

只要是IOC容器启动后，就会调用DispatcherServlet的onRefresh方法

protected void onRefresh(ApplicationContext context) {

initStrategies(context);

}

protected void initStrategies(ApplicationContext context) {

//请求解析

initMultipartResolver(context);

//多语言，国际化

initLocaleResolver(context);

//主题View层的

initThemeResolver(context);

//解析url和Method的关联关系

initHandlerMappings(context);

//适配器（匹配过程）

initHandlerAdapters(context);

//异常解析

initHandlerExceptionResolvers(context);

//视图转发（根据视图名字匹配到一个具体模板）

initRequestToViewNameTranslator(context);

//解析模板中的内容（拿到服务器传过来的数据，生成HTML代码）

initViewResolvers(context);

initFlashMapManager(context);

}

解析url和Method的关联关系：

private void initHandlerMappings(ApplicationContext context) {

this.handlerMappings = null;

if (this.detectAllHandlerMappings) {

// Find all HandlerMappings in the ApplicationContext, including ancestor contexts.

Map<String, HandlerMapping> matchingBeans =

BeanFactoryUtils.beansOfTypeIncludingAncestors(context, HandlerMapping.class, true, false);

if (!matchingBeans.isEmpty()) {

this.handlerMappings = new ArrayList<HandlerMapping>(matchingBeans.values());

// We keep HandlerMappings in sorted order.

OrderComparator.sort(this.handlerMappings);

}

}

else {

try {

HandlerMapping hm = context.getBean(HANDLER\_MAPPING\_BEAN\_NAME, HandlerMapping.class);

this.handlerMappings = Collections.singletonList(hm);

}

catch (NoSuchBeanDefinitionException ex) {

// Ignore, we'll add a default HandlerMapping later.

}

}

// Ensure we have at least one HandlerMapping, by registering

// a default HandlerMapping if no other mappings are found.

if (this.handlerMappings == null) {

this.handlerMappings = getDefaultStrategies(context, HandlerMapping.class);

if (logger.isDebugEnabled()) {

logger.debug("No HandlerMappings found in servlet '" + getServletName() + "': using default");

}

}

}

initHandlerMapping中Map去BeanFactoryUtils类中去取：

public static <T> Map<String, T> beansOfTypeIncludingAncestors(

ListableBeanFactory lbf, Class<T> type, boolean includeNonSingletons, boolean allowEagerInit)

throws BeansException {

Assert.notNull(lbf, "ListableBeanFactory must not be null");

Map<String, T> result = new LinkedHashMap<String, T>(4);

result.putAll(lbf.getBeansOfType(type, includeNonSingletons, allowEagerInit));

if (lbf instanceof HierarchicalBeanFactory) {

HierarchicalBeanFactory hbf = (HierarchicalBeanFactory) lbf;

if (hbf.getParentBeanFactory() instanceof ListableBeanFactory) {

Map<String, T> parentResult = beansOfTypeIncludingAncestors(

(ListableBeanFactory) hbf.getParentBeanFactory(), type, includeNonSingletons, allowEagerInit);

for (Map.Entry<String, T> entry : parentResult.entrySet()) {

String beanName = entry.getKey();

if (!result.containsKey(beanName) && !hbf.containsLocalBean(beanName)) {

result.put(beanName, entry.getValue());

}

}

}

}

return result;

}

HandlerMapping/HandlerAdapters都是通过扫描注解得到的

### doDispatch执行HandlerMapping映射

/\*\* 中央控制器,控制请求的转发 \*\*/

protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {

HttpServletRequest processedRequest = request;

HandlerExecutionChain mappedHandler = null;

boolean multipartRequestParsed = false;

WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

try {

ModelAndView mv = null;

Exception dispatchException = null;

try {

//1.检查是否是文件上传的请求

processedRequest = checkMultipart(request);

multipartRequestParsed = processedRequest != request;

// Determine handler for the current request.

// 2.取得处理当前请求的controller,这里也称为hanlder,处理器,第一个步骤的意义就在这里体现了.

//这里并不是直接返回controller,而是返回的HandlerExecutionChain请求处理器链对象,

//该对象封装了handler和interceptors.

mappedHandler = getHandler(processedRequest, false);

// 如果handler为空,则返回404

if (mappedHandler == null || mappedHandler.getHandler() == null) {

noHandlerFound(processedRequest, response);

return;

}

// Determine handler adapter for the current request.

//3. 获取处理request的处理器适配器handler adapter

HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

// Process last-modified header, if supported by the handler.

// 处理 last-modified 请求头

String method = request.getMethod();

boolean isGet = "GET".equals(method);

if (isGet || "HEAD".equals(method)) {

long lastModified = ha.getLastModified(request, mappedHandler.getHandler());

if (logger.isDebugEnabled()) {

String requestUri = urlPathHelper.getRequestUri(request);

logger.debug("Last-Modified value for [" + requestUri + "] is: " + lastModified);

}

if (new ServletWebRequest(request, response).checkNotModified(lastModified) && isGet) {

return;

}

}

// 4.拦截器的预处理方法

if (!mappedHandler.applyPreHandle(processedRequest, response)) {

return;

}

try {

// Actually invoke the handler.

// 5.实际的处理器处理请求,返回结果视图对象

mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

}

finally {

if (asyncManager.isConcurrentHandlingStarted()) {

return;

}

}

// 结果视图对象的处理

applyDefaultViewName(request, mv);

// 6.拦截器的后处理方法

mappedHandler.applyPostHandle(processedRequest, response, mv);

}

catch (Exception ex) {

dispatchException = ex;

}

processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);

}

catch (Exception ex) {

// 请求成功响应之后的方法

triggerAfterCompletion(processedRequest, response, mappedHandler, ex);

}

catch (Error err) {

triggerAfterCompletionWithError(processedRequest, response, mappedHandler, err);

}

finally {

if (asyncManager.isConcurrentHandlingStarted()) {

// Instead of postHandle and afterCompletion

mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);

return;

}

// Clean up any resources used by a multipart request.

if (multipartRequestParsed) {

cleanupMultipart(processedRequest);

}

}

}

## 4、手写Spring MVC：

### 第一步、将编写自定义注解

import java.lang.annotation.Documented;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documented

public @interface DCAutowired {

String value() default "";

}

import java.lang.annotation.Documented;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

public @interface DCController {

String value() default "";

}

import java.lang.annotation.Documented;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Target({ElementType.METHOD, ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

public @interface DCRequestMapping {

String value() default "";

}

import java.lang.annotation.Documented;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Target(ElementType.PARAMETER)

@Retention(RetentionPolicy.RUNTIME)

@Documented

public @interface DCRequestParam {

String value() default "";

boolean required() default true;

}

import java.lang.annotation.Documented;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

@Documented

public @interface DCResponseBody {

}

import java.lang.annotation.Documented;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

public @interface DCService {

String value() default "";

}

### 第二步、在web.xml中配置SpringMVC的入口文件

<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd" id="WebApp\_ID" version="2.5">

<display-name>011.myspringmvc</display-name>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

<welcome-file>index.htm</welcome-file>

<welcome-file>index.jsp</welcome-file>

<welcome-file>default.html</welcome-file>

<welcome-file>default.htm</welcome-file>

<welcome-file>default.jsp</welcome-file>

</welcome-file-list>

<servlet>

<servlet-name>DCSpringMVC</servlet-name>

<servlet-class>com.dc.mvc.framework.servlet.DCDispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>application.properties</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>DCSpringMVC</servlet-name>

<url-pattern>\*.json</url-pattern>

</servlet-mapping>

</web-app>

### 第三步、编写入口文件，进行创建ioc容器、解析注解

public class DCDispatcherServlet extends HttpServlet {

private static final String LOCATION = "contextConfigLocation";

private Map<String, Handler> handlerMapping = new HashMap<String, Handler>();

//初始化IOC容器

@Override

public void init(ServletConfig config) throws ServletException {

//IOC容器要初始化

DCApplicationContext context = new DCApplicationContext(config.getInitParameter(LOCATION));

initMultipartResolver(context);

initLocaleResolver(context);

initThemeResolver(context);

initHandlerMappings(context);

initHandlerAdapters(context);

initHandlerExceptionResolvers(context);

initRequestToViewNameTranslator(context);

initViewResolvers(context);

initFlashMapManager(context);

}

@Override

protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

this.doPost(req, resp);

}

//在这里调用自己写的Controller方法

@Override

protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

System.out.println("调用DCSpring MVC的doPost方法");

try {

doDispatch(req, resp);

} catch (Exception e) {

resp.getWriter().write("500 Exception, Msg:" + Arrays.toString(e.getStackTrace()));

}

}

}

真正的解析在DCApplicationContext类中完成：

public class DCApplicationContext {

private Map<String,Object> instanceMapping = new ConcurrentHashMap<String, Object>();

//类似于内部的配置信息（BeanDefintion），我们在外面是看不到的只有ioc容器，通过getBean方法间接访问的

private List<String> classCache = new ArrayList<String>();

public DCApplicationContext(String location) {

InputStream is = null;

//先加载配置文件 定位、载入、注册、初始化、注入

try {

//定位

is = this.getClass().getClassLoader().getResourceAsStream(location);

//载入

Properties config = new Properties();

config.load(is);

//注册，把所有的class找出来

String packageName = config.getProperty("scanPackage");

doRegister(packageName);

//初始化，只要循环class

doCreateBean();

//注入

populate();

} catch (Exception e) {

e.printStackTrace();

}

}

//把符合条件的class全部找出来，注册到缓存里面去

private void doRegister(String packageName) {

URL url = this.getClass().getClassLoader().getResource("/" + packageName.replaceAll("\\.", "/"));

File dir = new File(url.getFile());

for(File file : dir.listFiles()) {

//如果是一个文件夹，继续递归

if(file.isDirectory()) {

doRegister(packageName + "." + file.getName());

} else {

classCache.add(packageName + "." + file.getName().replace(".class", "").trim());

}

}

}

private void doCreateBean() {

//检查看有没有注册信息

if(classCache.size() == 0) { return ; }

try {

for(String className : classCache) {

Class clazz = Class.forName(className);

//只要加了 @Service @Controller注解的都要初始化

if(clazz.isAnnotationPresent(DCController.class)) {

//名字

String id = lowerFirstChar(clazz.getSimpleName());

instanceMapping.put(id, clazz.newInstance());

} else if(clazz.isAnnotationPresent(DCService.class)) {

DCService service = (DCService) clazz.getAnnotation(DCService.class);

//如果设置了自定义名字，就优先用他定义的名字

String id = service.value();

if("".equals(id.trim())) {

instanceMapping.put(id, clazz.newInstance());

continue;

}

//如果是空的，就默认规则

//1、类名首字母小写

//2、如果这个类是接口，可以根据类型类匹配

Class[] interfaces = clazz.getInterfaces();

//如果这个类实现了接口，就用接口的类型作为id

for(Class i : interfaces) {

instanceMapping.put(i.getName(), clazz.newInstance());

}

} else {

continue;

}

}

} catch (Exception e) {

e.printStackTrace();

}

}

private void populate() {

//首先要判断ioc容器中有没有东西

if(instanceMapping.isEmpty()) {return;}

for(Entry<String, Object> entry : instanceMapping.entrySet()) {

//把所有的属性都取出来

Field[] fields = entry.getValue().getClass().getDeclaredFields();

for(Field field : fields) {

if(!field.isAnnotationPresent(DCAutowired.class)) { continue; }

DCAutowired autowried = field.getAnnotation(DCAutowired.class);

String id = autowried.value().trim();

//如果id为空，也就是说，自己没有设置，默认根据类型来注入

if("".equals(id)) {

id = field.getType().getName();

} else {

}

field.setAccessible(true); //把私有变量开放访问权限

try {

field.set(entry.getValue(), instanceMapping.get(id));

} catch (Exception e) {

e.printStackTrace();

continue;

}

}

}

}

//将首字母小写

private String lowerFirstChar(String str) {

char[] chars = str.toCharArray();

chars[0] += 32;

return String.valueOf(chars);

}

public Object getBean(String name) {

return null;

}

public Map<String, Object> getAll() {

return instanceMapping;

}

}

# 5、Spring mvc总结

beans包定义了规范

core包定义了一些核心工具包

context工厂实现，工厂扩展

## 谈谈SpringMVC的优化

1. controller如果能保持单例，尽量使用单例，这样可以减少创建对象和回收对象的开销，也就是说，如果controller的类变量和实力变量可以以方法形参声明的尽量以方法的形参声明，不要以类变量和实例变量声明，这样可以避免线程安全问题。
2. 处理request的方法中的形参务必加上@RequestParam注解，这样可以避免Springmvc使用asm框架读取class文件获取方法名的过程，即便springmvc对读取出的方法参数名进行了缓存，如果不要读取class文件当然是更加好。
3. 阅读源码的过程中，发现springmvc并没有处理url的方法进行缓存，也就是说每次都要根据请求去匹配controller中的方法url，如果把url和method的关系缓存起来，会不会带来性能上的提升呢？有点恶心的是，负责解析url和method对应关系的ServletHandlerMethodResolver是一个private的内部类，不能直接继承类增强代码，必须要该代码后重新编译。当然，如果缓存起来，必须要考虑缓存的线程安全问题

# 6、Spring和SpringMVC面试问题

1、spring AOP中默认使用jdk的动态代理还是使用CGLIB？是不是在Spring 配置文件中加入

<aop:aspectj-autoproxy proxy-target-class=”true” />默认是false 改成true就会强制使用CGLIB？

IOC判断，如果被代理的类实现一个接口，那么默认用JDK代理

如果别代理的类没有实现任何一个接口，那么默认用CGLib

2、

jdk1.8和Spring的4.x以上版本可以一起使用

jdk1.7及以下可以和Spring的3.x使用

3、为什么使用Spring

Spring支持了JDBC，Dao层解决了

Spring支持AOP，多模块开发解耦的问题解决了

Spring支持MVC，页面交互的问题也解决了

JDBC代替了Hibernate、Mybatis

MVC代替了Struts、WebWork

4、拦截器和过滤器的区别

实现原理一样

只不过拦截器这个概念首先由Struts提出

然后Spring在其基础上进行了升华，优化

而过滤器只是我们的J2EE标准，在J2EE标准过程中是没有拦截器这个概念的

5、SpringMVC是否为线程安全，分布式情况下如何做到线程安全，如何理解SpringMVC，Spring IOC创建实例对象的生命周期，Spring Bean什么时候被垃圾回收。

6、SpringMVC的父子容器，怎么控制类的加载顺序，怎么使用Spring配置事务

7、Spring AOP动态代理有哪些实现方式：

jdk和cglib

8、SpringMVC的父子容器？

Spring中所有的容器首先都要实现BeanFactory

IOC容器（默认实现）、AOP容器、MVC容器（扩展关系）

9、Spring中哪些类用到了单例

Spring BeanFactory读取配置时候，只要你没有设置scope属性，默认就是单例的

10、Spring Bean什么时候被回收

Spring运行的时候，实际上都是不同的java类（包括代理以后的类，都要被我们ClassLoad加载进来），它都由我们的GC机制来统一回收

1. 类指针丢失，多一段事件自动回收
2. 手动置空，也会被回收（设置为null）

11、Spring如何实现分布式事务

# 7、SpringMVC与Struts对比

1、Struts2是类级别的拦截，一个类对应一个request上下文，SpringMVC是方法级别的拦截，一个方法对应一个request上下文，而方法同时又跟一个url对应，所以说从架构本身上SpringMVC就容易实现restful url，而Struts2的架构实现起来要费劲，因为Struts2中Action的一个方法可以对应一个url，而其类属性却被所有方法共享，这也就无法用注解或其他方式标识其所属方法了。

2、由上边原因，SpringMVC的方法之间基本上独立的，独享request response数据，请求数据通过参数获取，处理结果通过ModelMap交回给框架，方法之间不共享变量，而Struts搞得就比较乱，虽然方法之间也是独立的，但其所有Action变量是共享的，这不会影响程序运行，去给我们编码，读程序是带来了麻烦，每次来了请求就创建一个Action，一个Action对象对应一个request上下文。

3、由于Struts2需要针对每个request进行封装，把requst，session等servlet生命周期变量封装成一个一个Map，供给每个Action使用，并保证线程安全，所以原则上，是比较耗费内存的

4、拦截器实现机制上，Struts2有以自己的interceptor机制，SpringMVC用的是独立的AOP方式，这样导致Struts2的配置文件量还是比SpringMVC大。

5、SpringMVC的入口是servlet，而Struts2是filter（这里要指出，filter和servlet是不同的。以前认为filter是servlet的一种特殊），这就导致了二者的机制不同，这里就牵涉到servlet和filter的区别了。

6、SpringMVC继承了Ajax，使用非常方便，只需要一个注解@ResponseBody就可以实现，然后直接返回响应文本即可，而Struts2拦截器继承了Ajax，在Action中处理时一般必须安装插件或者自己写代码继承进去，使用起来也相对不方便。

7、SpringMVC验证了JSR303，处理起来相对更加灵活方便，而Struts2验证比较繁琐，感觉太烦乱。

8、Spring MVC和Spring是无缝的。从这个项目的管理和安全上也比Struts2高（当然Struts2也可以通过不同的目录结构和相关配置做到SpringMVC一样的效果，但是需要xml配置的地方不少）

9、设计思想上，Struts2更加符合OOP的编程思想，SpringMVC就比较谨慎，在servlet上扩展。

10、SpringMVC开发效率和性能高于Struts2

11、SpringMVC可以认为已经100%零配置

# 8、Spring总结

优点：

1. Spring是java开发中的集大成者，自从有了Spring，真的就像java的春天来了。
2. Spring将常用的设计模式运用的淋漓尽致
3. Spring可以说是万能胶，主流的java框架都可以与她集成（因为Spring是从java的基本元素Bean作文章）。
4. 自从有了Annotation，Spring几乎可以实现零配置编码
   1. 降低了业务对象替换的复杂性，属于低侵入，代码污染极低。
5. Spring已经形成了自身的生态链，例如Spring Boot、Spring Cloud、Spring Data（对大数据的支持等）真可谓与时俱进，又好又快发展。

当然，从唯物辩证法来说，任何东西都是一把双刃剑，Spring也有以下遗憾

1. Spring本身而言功能越来越强大，对于java基础较弱的人，想要弄懂其原理不是一件很容易的事。
2. Spring暂不支持分布式编程，这也是EJB还一直存在的原因（分布式只能借助其他框架来实现）

# Spring源码解析-事务

spring的事务不能自己处理异常，应该将异常抛出

## 1、什么是事务（Transaction）

事务（Transaction）是访问可能更新数据库中各种数据项的一个程序执行单元（unit）。

特点：

事务是回复和并发控制的基本单位。

事务应该具有4个属性：原子性、一致性、隔离性、持久性。这四个属性通常称为ACID特性。

原子性：一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做

一致性：事务必须是使数据库从一个一致性状态变换到另一个一致性状态。一致性和原子性是密切相关的。

隔离性：一个事务的执行不能被其他事务干扰。即一个事务内部操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。

持久性：持久性也称永久性，指一个事务一旦提交，他对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。

插入：

1. 先把要插入的数据放入临时表
2. 将临时表中数据插入实际表中去
3. 如果没问题，就复制一份到实际表中，并将临时表中的数据删除
4. 如果有问题，返回错误信息，临时表清空

删除：

1. 想根据条件从原始表中查询来满足条件的数据行
2. 将这些数据行复制一份到临时表
3. 执行删除，如果出现错误，原来的数据原封不动，清空临时表满足本次条件的记录，返回错误信息
4. 如果执行成功，真正的干掉原始表中的记录。返回影响行数

## 2、事务的基本原理

事务操作基本流程：

1. 事务开启（open）
2. 执行职务（execute）
3. 提交事务（自动提交AutoCommit/手动提交CustomCommit）

事务回滚（rollBack如果出现错误）

1. 关闭事务（close）

Spring事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，spring是无法提供事务功能的。对于纯JDBC操作数据库，想要用到事务，可以按照以下步骤进行：

1.获取连接 Connection conn= DriverManager.getConnection()

2.开启事务 conn.setAutoConmmit(true/false)

3.执行CRUD

4.提交事务/回滚事务 conn.commit() / conn.rollback();

5.关闭连接 conn.close();

使用Spring的事务管理功能后，我们可以不再写步骤2和4的代码，而是由Spring自动完成。那么Spring是如何在我们书写的CRUD之前和之后开启事务和关闭事务的呢？解决这个问题，也就可以从整体上理解Spring的事务管理实现原理了。下面简单的介绍下，注解方式为例子

1. 配置文件开启注解驱动，在相关的类和方法上通过注解@Transactional标识
2. spring在启动的时候会去解析生成相关的bean，这时候会查看拥有相关注解的类和方法，并且为这些类和方法生成代理，并根据@Transaction的相关参数进行相关配置注入，这样就在代理中为我们把相关的事务处理掉了（开启正常提交事务，异常回滚事务）
3. 真正的数据库层的事务提交和回滚是通过binlog或者redo log实现的。

## Spring事务传播属性

所谓spring事务的传播属性，就是定义在存在多个事务同时存在的时候，spring应该如何处理这些事务的行为。这些属性在TransactionDefinition中定义，具体常量的解析见下表：

|  |  |
| --- | --- |
| 常量名称 | 常量解析 |
| PROPAGATION\_REQUIRED  propagation\_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，也是Spring默认的事务的传播。 |
| PROPAGATION\_REQUIRES\_NEW  propagation\_requires\_new | 新建事务，如果当前存在事务，把当前事务挂起。新建的事务将和被挂起的事务没有任何关系，是两个独立的事务，外层事务失败回滚之后，不能回滚内层事务执行的结果，内层事务失败抛出异常，外层捕获，也可以不处理回滚操作 |
| PROPAGATION\_MANDATORY  propagation\_mandatory | 支持当前事务，如果当前没有事务，就抛出异常 |
| PROPAGATION\_NOT\_SUPPORTED  propagation\_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 |
| PROPAGATION\_NEVER  propagation\_never | 以非事务方式执行，如果当前存在事务，则抛出异常 |
| PROPAGATION\_NESTED  propagation\_nested | 如果一个活动的事务存在，则运行在一个嵌套的事务中。如果没有活动事务，则按REQUIRED属性执行。它使用了一个单独事务，这个事务拥有多个可以回滚的保存点。内部事务的回滚不会对外部事物造成影响。他只对DataSourceTransactionManager事务管理器起效 |

<!--

1、数据源：不管是哪个厂商都要是实现DataSource接口，拿到包含了Connection的对象

2、使用Spring给我们提供的工具类TransactionMagager 事务管理器，来管理所有的 事务操作(肯定要拿到连接对象)

-->

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"/>

</bean>

<tx:annotation-driven transaction-manager="transactionManager"/>

<!-- 3、利用切面编程来实现对某一类方法进行事务统一管理(声明式事务) -->

<!-- 属于AOP中的东西，比较熟悉了的 -->

<!-- <aop:config>

<aop:pointcut expression="execution(public \* com.gupaoedu.vip..\*.service..\*Service.\*(..))" id="transactionPointcut"/>

<aop:advisor pointcut-ref="transactionPointcut" advice-ref="transactionAdvice"/>

</aop:config> -->

<!-- 4、配置通知规则 -->

<!-- Transaction tx :NameSpace -->

<!-- <tx:advice id="transactionAdvice" transaction-manager="transactionManager">

<tx:attributes>

<tx:method name="add\*" propagation="REQUIRED" rollback-for="Exception,RuntimeException" timeout="-1"/>

<tx:method name="remove\*" propagation="REQUIRED" rollback-for="Exception,RuntimeException"/>

<tx:method name="modify\*" propagation="REQUIRED" rollback-for="Exception,RuntimeException"/>

<tx:method name="transfer\*" propagation="REQUIRED" rollback-for="Exception"/>

<tx:method name="login" propagation="REQUIRED"/>

<tx:method name="query\*" read-only="true"/>

</tx:attributes>

</tx:advice> -->

## 4、数据库隔离级别

|  |  |  |
| --- | --- | --- |
| 隔离级别 | 隔离级别的值 | 导致的问题 |
| Read-Uncommited | 0 | 导致脏读 |
| Read-Committed | 1（默认） | 导致脏读，允许不可重复读或幻读 |
| Repeatable-Read | 2 | 导致脏读，不可重复读，允许幻读 |
| Serializable | 3 | 串行化读，事务只能一个一个执行，避免了脏读、不可重复读、幻读。执行效率慢，使用时谨慎 |

脏读：一事务对数据进行了增删改，但未提交，另一事物可以读取到未提交的数据。如果第一个事务这时候回滚了，那么第二个事务就读到了脏数据（多读）

不可重复读：一个事务中发生了两次操作，第一次操作和第二次操作之间，另外一个事务对数据进行了修改，这时候两次读取的数据是不一致的。

幻读：第一个事务对一定范围的数据进行批量修改，第二个事务在这个范围增加一条数据，这时候第一个事务就会丢失新增数据的修改。（少读）

### 总结

隔离级别越高，越能保证数据的完整性和一致性，但是对并发性能的影响也越大

大多数的数据库默认隔离级别为Read Commited，比如SqlServer、Oracle

少数数据库默认隔离级别为Repeatable Read 比如：MySQL InnoDB

## 5、Spring中的隔离级别

|  |  |
| --- | --- |
| 常量 | 解释 |
| ISOLATION\_DEFAULT | 这个是 PlatfromTransactionManager 默认的隔离级别，使用数据库默认的事务隔离级别。另外四个与JDBC的隔离级别相对应 |
| ISOLATION\_READ\_UNCOMMITTED | 这是食物最低的隔离级别，他允许另外一个事务可以看到这个事务未提交的数据。这个隔离级别会产生脏读，不可重复读和幻读 |
| ISOLATION\_READ\_COMMITTED | 保证一个事务修改的数据提交后才能被另一个事务读取。另外一个事务不能读取该事物未提交的数据。 |
| ISOLATION\_REPEATABLE\_READ | 这种事务隔离级别可以防止脏读，不可重复读。但是可能出现幻读。 |
| ISOLATION\_SERIALIZABLE | 这是花费最高代价但是最可靠的事务隔离级别。事务被处理为顺序执行。 |

## 6、事务嵌套

## 7、事务源码解析

根据对事务的配置标签<tx:advice>...</tx:advice>分析

tx是TransactionNameSpace。对应的Handler是TxNamespaceHandler

这个类一个init方法

public void init() {

registerBeanDefinitionParser("advice", new TxAdviceBeanDefinitionParser());

registerBeanDefinitionParser("annotation-driven", new AnnotationDrivenBeanDefinitionParser());

registerBeanDefinitionParser("jta-transaction-manager", new JtaTransactionManagerBeanDefinitionParser());

}

这个方法是在DefaultNamespaceHandlerResolver的resolver中调用的。在为对应的标签寻找namespacehandler的时候，调用这个resolve方法。resolve方法先寻找namespaceUri对应的namespacehandler，如果找到了就先调用Init方法。

<tx:advice>对应的解析器也就注册了。那就是上面的代码里面的。现在这个parser的parse方法在NamespaceHandlerSupport的parse方法中调用了，下面我们来看看这个TxAdviceBeanDefinitionParser的parse方法吧，这个方法在TxAdviceBeanDefinitionParser的祖父类AbstractBeanDefinitionParser中：

public final BeanDefinition parse(Element element, ParserContext parserContext) {

AbstractBeanDefinition definition = parseInternal(element, parserContext);

if (definition != null && !parserContext.isNested()) {

try {

String id = resolveId(element, definition, parserContext);

if (!StringUtils.hasText(id)) {

parserContext.getReaderContext().error(

"Id is required for element '" + parserContext.getDelegate().getLocalName(element)

+ "' when used as a top-level tag", element);

}

String[] aliases = new String[0];

String name = element.getAttribute(NAME\_ATTRIBUTE);

if (StringUtils.hasLength(name)) {

aliases = StringUtils.trimArrayElements(StringUtils.commaDelimitedListToStringArray(name));

}

BeanDefinitionHolder holder = new BeanDefinitionHolder(definition, id, aliases);

registerBeanDefinition(holder, parserContext.getRegistry());

if (shouldFireEvents()) {

BeanComponentDefinition componentDefinition = new BeanComponentDefinition(holder);

postProcessComponentDefinition(componentDefinition);

parserContext.registerComponent(componentDefinition);

}

}

catch (BeanDefinitionStoreException ex) {

parserContext.getReaderContext().error(ex.getMessage(), element);

return null;

}

}

return definition;

}

parseInternal()方法是在TxAdviceBeanDefinitionParser的父类AbstractSingleBeanDefinitionPar

ser中实现，代码如下：

@Override

protected final AbstractBeanDefinition parseInternal(Element element, ParserContext parserContext) {

BeanDefinitionBuilder builder = BeanDefinitionBuilder.genericBeanDefinition();

String parentName = getParentName(element);

if (parentName != null) {

builder.getRawBeanDefinition().setParentName(parentName);

}

Class<?> beanClass = getBeanClass(element);

if (beanClass != null) {

builder.getRawBeanDefinition().setBeanClass(beanClass);

}

else {

String beanClassName = getBeanClassName(element);

if (beanClassName != null) {

builder.getRawBeanDefinition().setBeanClassName(beanClassName);

}

}

builder.getRawBeanDefinition().setSource(parserContext.extractSource(element));

if (parserContext.isNested()) {

// Inner bean definition must receive same scope as containing bean.

builder.setScope(parserContext.getContainingBeanDefinition().getScope());

}

if (parserContext.isDefaultLazyInit()) {

// Default-lazy-init applies to custom bean definitions as well.

builder.setLazyInit(true);

}

doParse(element, parserContext, builder);

return builder.getBeanDefinition();

}

使用了加载的类依赖AOP中的方法拦截器：

@Override

protected Class<?> getBeanClass(Element element) {

return TransactionInterceptor.class;

}

至此，这个标签解析的流程已经基本解析了。那就是：解析除了一个以TransactionInerceptor为classname的beandefinition并且注册这个bean。剩下来要看，就是这个TransactionInerceptor到地是什么：

看看这个类的接口定义，就明白了：

public class TransactionInterceptor extends TransactionAspectSupport implements MethodInterceptor, Serializable

这根本就是一个spring AOP的advice！现在明白为什么事务的配置能通过aop产生作用了吧？

public Object invoke(final MethodInvocation invocation) throws Throwable {

Class<?> targetClass = (invocation.getThis() != null ? AopUtils.getTargetClass(invocation.getThis()) : null);

return invokeWithinTransaction(invocation.getMethod(), targetClass, new InvocationCallback() {

public Object proceedWithInvocation() throws Throwable {

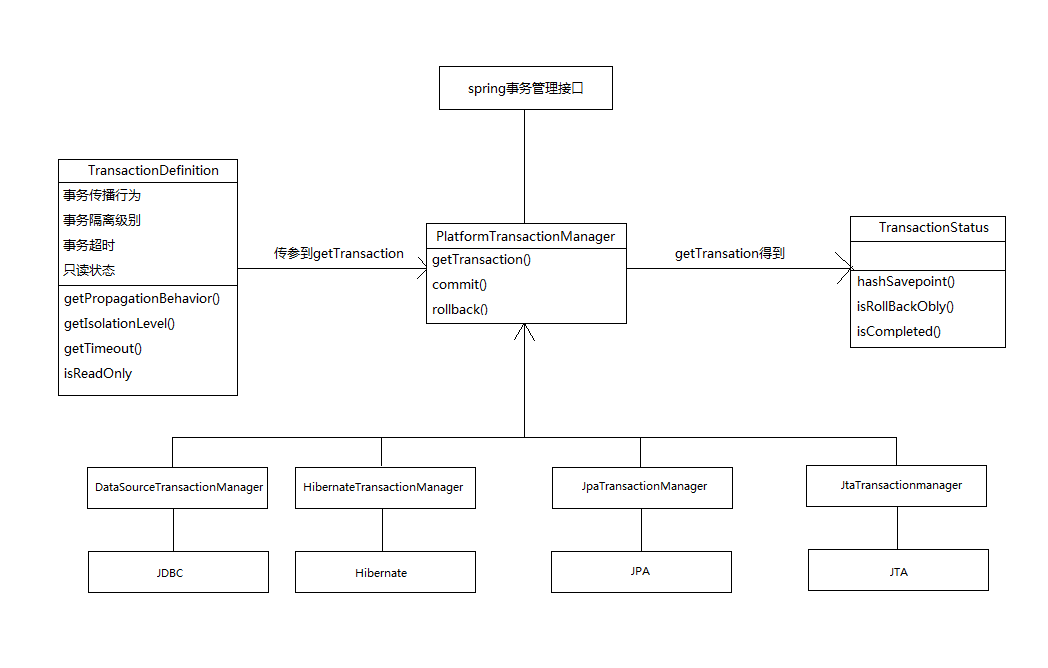
return invocation.proceed();

}

});

}

接下来再看看DataSourceTransactionManager是如何工作的



if (con.getAutoCommit()) {

txObject.setMustRestoreAutoCommit(true);

if (logger.isDebugEnabled()) {

logger.debug("Switching JDBC Connection [" + con + "] to manual commit");

}

con.setAutoCommit(false);

}

DoGetTraansaction() //从ThreadLocal中获取一个connection（相互独立的）

DoBegin() //开启事务

//执行业务逻辑

//根据业务逻辑的执行结果来判断是否要提交还是回滚

//doCommit() 提交

//dorollBack() 回滚

## 总结：

### 什么是事务？

就是一个整体的执行单元，只有两个结果，要么全失败，要么全成功。

### 2、事务的特性？

原子性、隔离性、持久性、一致性

原子性：一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做

一致性：事务必须是使数据库从一个一致性状态变换到另一个一致性状态。一致性和原子性是密切相关的。

隔离性：一个事务的执行不能被其他事务干扰。即一个事务内部操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。

持久性：持久性也称永久性，指一个事务一旦提交，他对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。

### 3、事务的基本原理

从数据库角度来说：就是提供了一种后悔机制（代码写错了，可以SVN、Git）它使用的是临时表才实现的后悔

执行增删改之前，先将满足条件的数据查询出来放入临时表中：

将数据操作现在临时表中完成，完成过程中如果没有任何问题，就将数据同步（剪切）到实际的数据表中，并返回影响行数

将数据操作现在临时表中完成，完成过程中如果一旦出现错误，那么就将满足条件的数据清掉，并返回错误码

### 4、Spring配置事务

AOP配置，配置哪些方法需要加事务

声明式事务配置，事务的传播属性、隔离级别、回滚条件

传播属性：DEFAUTL REQUIRED EQUIRES\_NEW SUPPORTS NESTED

隔离级别：DEFAULT READ\_UNCOMMITTED READ\_COMMITTED REPEATABLE\_READ SERIALIZABLE

### 5、源码

通过解析配置文件，得到TransactionDefinition，实际上就是AOP中的MethodInterceptor（方法代理）

就可以在满足条件的方法调用之前和调用之后加一些东西（PlatformTransactionManager中的方法）

getTransaction 调用了TransactionSynchronizationManager类的getResource方法，从ThreadLocal里面取值，Map<key:DataSource, value:ConnectionHolder(相当于获取一个连接对象Connection)>

Commit conn.commit()

Rollback conn.rollback()

# 8、事务的应用案例

银行转账模拟