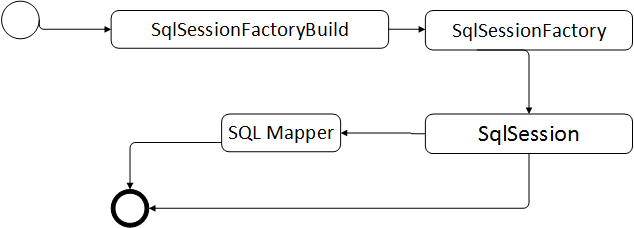
Hibernate屏蔽了SQL,意味着只能全表映射,如果一张表有几十到上百个字段,而你使用的只有2个,Hiberbate则无法适应,会引发性能问题.在复杂的场景需要关联多张表时,Hibernate全表逐级取对象的方法同样不适用.

Hibernate缺点:

1. 全表映射带来的不便,比如更新是需要发送所有的字段.
2. 无法根据不同的条件组装不同的SQL
3. 对多表关联和复杂SQL查询支持较差,需自己写SQL,返回后,需要自己将数据组装为POLO.
4. 不能有效支持存储过程
5. 虽然有HQL,但性能较差.大型互联网系统往往需要优化SQL,而Hibernate做不到.

Mybatis拥有动态列\动态表名,存储过程都支持,同时提供了简易的缓存\日志\级联.但是它的缺陷是需要你提供映射规则和SQL,所以它的开发工作量比Hibernate略大一些.



构建SqlSessionFactory两种方式

1. 使用XML方式构建
2. 使用代码方式构建 Configuration类

创建SqlSession接口类似于JDBC中的Connection接口对象

sqlSessionFactory.openSession()

SqlSession用途:

1. 获取映射器
2. 直接通过命名信息去执行SQL返回结果

映射器由Java接口和XML文件共同组成

一个没有实现类的接口怎么能够运行,其实是用到了Java语言的动态代理去实现的.

会在Mybatis上下文中描述这个接口,而Mybatis会为这个接口生成代理类对象,代理类对象会根据”接口全路径+方法名”去匹配,找到对应的XML文件去完成它所需要完成的任务,返回结果.

2.3生命周期

1.SqlSessionFactoryBuilder:作为构造器,一旦构建了SqlSessionFactory就失去了存在的意义.它的生命周期只存在于方法的局部.

2.SqlSessionFactory:每次应用程序需要访问数据库,就要通过SqlSessionFactory创建sqlsession,所以sqlsessionFactory应该在mybatis应用的整个生命周期.其责任是唯一的,每次创建sqlsessionFactory会打开更多的数据库连接资源,为避免资源耗尽,采用单例模式. 全局单例

3.SqlSession是一个会话,相当于JDBC的一个Connection对象,它的生命周期应该是在请求数据库处理事务的过程中. 及时关闭 线程不安全 在涉及多线程时,操作数据库需要注意其隔离级别\数据库锁等高级特性.它存活于一个应用的请求和操作,可以执行多条SQL,保证事务的一致性.

4.Mapper:它的最大范围和SqlSession相同.

3.配置:

Mybatis配置XML文件的层次结构 p37

3.1properties元素<properties resource= ”jdbc.properties ”><property name=”” value=””/></properties>

3.2 设置 <settings> <setting name=”” value=”” /></settings>

3.3 别名 系统定义别名在 TypeAliasRegisTry()中

自定义别名 <typeAliases><tyoeAlias alias=”” type=””/></typeAliases>

扫描注解 <typeAliases><package name=” “/></typeAliases> @Alias(“”)

3.4 typeHandle类型处理器

MyBatis在预处理语句(PreparedStatement)中设置一个参数时,或者从结果集(ResultSet)中取出一个值时,都会用注册了的typeHandler进行处理.

typeHandle常用的配置为Java类型(javaType)\JDBC类型(jdbcType).typeHandler的作用就是将参数从javatype转化为jdbctype,或者从数据库取出结果时把jdbcType转化为javatype.

1.系统定义的typeHandler TypeHandlerRegistry()里注册了系统定义的typeHandler

2.自定义typeHandler

注册自定义typeHandler. <typeHandlers><typeHandler jdbcType=”VARCHAR” javaType=”string” handler=” MyStringTypeHandler”/></typeHandlers>

MyStringTypeHandler 继承TypeHandler接口 注解配置jdbctype\javatype

@MappedTypes({String.class}) @MappedJdbcTypes(JdbvType.VARCHAR)

3种typeHandler的使用方法

1. 在配置文件里配置,在映射集上定义jdbcType和javaType
2. 在映射集里面直接定义具体的typeHandler
3. 在参数中定义typeHandler.
4. 枚举类型typeHandler

EnumTypeHandler是使用枚举字符串名称作为参数传递的,EnumOrdinalTypeHandler是使用整数下标作为参数传递的.

自定义枚举类的typeHandler

3.5ObjectFactory 当MyBatis在构建一个结果返回的时候,都会使用ObjectFactory(对象工厂)去构建POJO,在Mybatis中可以定制自己的对象工厂.

<objectFactory type=”MyObjectFactory><property name=”name” value=”MyObjectFactory”/></objectFactory>

要实现ObejctFactory接口.也可以继承DefaultObjectFactory类.

3.7 environments配置环境 数据库事务 数据源

3.8 databaseProvider数据库厂商标识.

3.9 引入映射器的方法

1.用文件路径<mappers><mapper resource=”com/learn/mapper/roleMapper.xml”/></mappers>

2.用包名 <mappers><package name=”com.learn.mapper”/></mappers>

3.用类注册引入 <mappers><mapper class=”com.learn.mapper.UserMapper”/></mappers>

4.用userMapper.xml引入. <mappers><mapper url=<file:///var/mappers/com/learn/mapper/roleMapper.xml>/></mappers>

4. 映射器

Mybatis支持自动绑定JavaBean,只要让SQL返回的字段名和JavaBean的属性名保持一致(或者采用驼峰式命名),便可以省掉这些繁琐的映射配置.

4.2 select元素

4.2.3自动映射

autoMappingBehavior属性: NULL:取消自动映射 PARTIAL:只会自动映射,没有定义嵌套结果集映射的结果集. FULL:会自动映射任意复杂的结果集(无论是否嵌套).

数据库规范命名的,每个单词都用下划线分割,POJO采用驼峰式命名方法,那么可以设置mapUnderscoreToCamelCase为true,实现从数据库到POJO的自动映射.

4.2.4传递多个参数

1.使用Map传递参数 2.使用注解方式传递参数 3.使用JavaBean传递参数

4.2.5使用resultMap映射结果集

* 1. insert元素

4.3.2 主键回填和自定义 使用keyProperty属性指定哪个是主键字段,同时使用useGeneratedKeys属性告诉MyBatis这个主键是否使用数据库内置策略生成.

可以使用selectKey元素进行自定义处理

<selectKey keyProperty=”id” resultType=”int” order=”BEFORE”>……</selectKey>

在插入数据时,主键的值也可以使用自定义序列设置, nextval(‘ ‘)

* 1. update元素和delete元素

4.5参数

4.5.1参数配置 #{price, javaType=double, jdbcType=NUMERIC, numericScale=2, typeHandler=MyTypeHandler}

4.5.2 存储过程支持

4.5.3 特殊字符串替换和处理(#和$)

4.6 sql元素

可以定义一串SQL语句的组成部分,其他的语句可以通过引用来使用它

<sql id=”” >……</sql> <include refid=" “/>

也可以制定参数来使用,如下

<sql id=”role\_columns”>

#{prefix}.role\_no, #{prefix}.role\_name, #{prefix}.note

</sql>

<select parameterType=”string” id=”getRole” resultMap=”roleResultMap”>

Select <include refid=”role\_columns”><property name=”prefix” value=”r”/></include>

From t\_role r where role\_no=#{roleNo}

</select>

我们还可以给refid一个参数值,由程序制定引入SQL

<sql id=”someinclude”>select \* from <include refid=”${tableName}”/>

4.7 resultMap结果映射集

它的作用是定义映射规则/级联的更新/定制类型转化器等.

4.7.1 resultMap元素的构成

假设RoleBean不存在没有参数的构造方法,它的构造方法声明为public RoleBean(Integer id, String roleName),那么配置这个结果集:

<resultMap ……>

<constructor ><idArg column=”id” javaType=”int”/><arg column=”role\_name” javaType=”string”/>

</constructor>

</resultMap>

4.7.2使用map存储结果集 4.7.3使用POJO存储结果集

4.7.4 级联

MyBatis中级联分为3种:

Association:一对一关系 collection:一对多关系

Discriminator:鉴别器,可以根据实际选择采用哪个类作为实例,允许根据特定的条件去关联不同的结果集.

<resultMap><discriminator javaType=”int” column=”sex”>

<case value=”1” resultMap=”maleStudentMap”/>

<case value=”2” resultMap=”femaleStudentMap”/>

</discriminator></resultMap>

<resultMap id=”maleStudentMap” type=”com.learn.po.MaleStudentBean” extends=”studentMap”>

<collection property=”studentHealthMaleList” select=”com.learn.chapter4.mapper.StudentHealthMaleMapper.findStudentHealthMaleByStuId” column=”id”/>

</resultMap>

多层关联时,建议超过三层关联时尽量少用级联

为了处理N+1的问题,MyBatis引入了延迟加载的功能,延迟加载功能的意义在于,一开始并不取出级联数据,只有当使用了它了才发送SQL去取回数据

在MyBatis的配置中有两个全局的参数lazyLoadingEnable:是否开启延迟加载功能.

AggressiveLazyLoading的含义是对任意延迟属性的调用会使带有延迟加载属性的对象完整加载;反之,每种属性将按需加载.

在默认的情况下,MyBatis是按层级延迟加载的,aggressiveLazyLoading为true时,MyBatis的内容按层级加载,否则就按我们调用的要求加载.

局部延迟加载功能.可以在association和collection元素上加入属性值fetchType,它有两个取值:eager和lazy

另一种级联 没有N+1的问题 尽量通过左连接找到其他信息.

ofType属性定义的是collection里面的泛型是什么Java类型,Mybatis会拿你定义的Java类和结果集做映射.

4.8缓存cache

在默认情况下,mybatis只开启一级缓存(一级缓存只是相对于同一个sqlsession而言)

SqlSessionFactory层面上的二级缓存是不开启的,二级缓存的开启需要进行配置,实现二级缓存的时候,Mybatis要求返回的pojo必须是可序列化的,也就是要求实现serializable接口,配置的方法很简单,只需在映射XML文件配置就可以开启缓存.<cache/>

<cache eviction=”LRU” flushInterval=”100000” size=”1024” readOnly=”true”/>

Eviction:代表的是缓存回收策略.----LRU:最近最少使用的 FIFO:先进先出 SOFT:软引用,移除基于垃圾回收器状态和软引用规则的对象. WEAK:弱引用,更积极地移除基于垃圾收集器状态和弱引用规则的对象.

flushInterval:刷新间隔时间 readOnly:只读,意味着缓存数据只能读取而不能修改

自定义缓存 <cache type=”com.learn.MyCache”/>

也可以配置SQL层面上的缓存规则 <select ......flushCache=”false” useCache=”true”/>

<insert…….flushCache=”true”/>

1. 动态SQL

5.2 if元素 5.3choose、when、otherwise元素

5.4 trim、where、set元素

<where><if test=”….”> ….. </if></where> 这样当where元素里面的条件成立的时候,才会加入where这个SQL关键字到组装的SQL里面,否则不加入.

<trim prefix=”where” prefixOverrides=”and”><if …>……</if></trim>

Trim意味着需要去掉一些特殊的字符串,prefix代表的是语句的前缀,prefixOverrides代表的是你需要去掉的那种字符串.

<update ….> update t\_role <set> …</set> </update> set元素遇到了逗号,它会把对应的逗号去掉

等同于<trim prefix=”SET” suffixOverrides=”,”> …..</trim>

* 1. foreach元素

它的作用是遍历集合,可以很好的支持数组、List、Set接口的集合

<foreach item=”sex” index=”index” collection=”sexList” open=”(“ separator=”,” close=”)”>

#{sex} </foreach>

* 1. test的属性
  2. bind元素 其作用是通过OGNL表达式去自定义一个上下文变量

<bind name=”pattern” value=” ’%’ + \_parameter + ‘%’ ” />

1. Mybatis的解析和运行原理

Mybatis运行分为两大部分，第一部分是读取配置文件缓存到Configuration对象，用以创建SqlSessionFactory，第二部分是SqlSession的执行过程。

6.1涉及的技术难点简介

所谓的代理模式就是在原有的服务上多加一个占位，通过这个占位去控制服务的访问。

一般而言，动态代理分为两种，一种是JDK反射机制提供的代理，另一种是CGLIB代理。在JDK提供的代理，我们必须要提供接口，而CGLIB则不需要提供接口

6.1.1反射技术

反射调用的最大好处是配置性大大提高，就如同spring ioc容器一样，可以给很多配置设置参数，使得java应用程序能够顺利运行起来，大大提高了Java的灵活性和可配置性，降低模块之间的耦合。

6.1.2 JDK动态代理

1.编写服务类和接口 2.编写代理类，提供绑定和代理方法

代理类的要求是实现InvocationHandler接口的代理方法，当一个对象被绑定后，执行其方法的时候就会进入到代理方法里。 bind(target) invoke(Object proxy,Method method, args)

Proxy.newProxyInstance(target.getClass().getClassLoadeer(),target.getClass().getInterfaces(), this)

一旦绑定后，在进入代理对象方法调用的时候就会到代理类的代理方法上，代理方法有三个参数：第一个是proxy是代理对象，第二个是当前调用的那个方法，第三个是方法的参数。

6.1.3 CGLIB动态代理

实现MethodInterceptor接口

创建代理对象： 通过Enhancer对象的create方法。

回调方法 intercept(Object obj, Method method, Object[] args, MethodProxy proxy)

在mybatis中通常在延迟加载的时候才会用到CGLIB的动态代理。

6.2构建SqlSessionFactory过程

通过SqlSessionFactoryBuilder去构建SqlSessionFactory分为两步：

1. 通过XMLConfigBuilger解析配置的XML文件，读出配置参数，并将读取的数据存入Configuration类中。Mybatis几乎所有的配置都是存在这里的。
2. 使用Configuration对象去创建SqlSessionFactory

6.2.1构建Configuration

它会做如下初始化：properties全局参数、settings设置、typeAliases别名、typeHandler类型处理器、ObjectFactory对象、plugin插件、environment环境、DatabaseIdProvider数据库标识、mapper映射器。

6.2.2映射器的内部组成

MappedStatement:它保存映射器的一个节点（select|insert|delete|update）。包括许多我们配置的SQL、SQL的id、缓存信息、resultMap、parameterType、resultType、languageDriver等重要配置内容。

SqlSource，它是提供BoundSql对象的地方，它是MappedStatement的一个属性。

BoundSql：它是建立SQL和参数的地方。它有3个常用的属性：SQL、parameterObject、parameterMappings。

ParameterObject为参数本身：传递简单对象、POJO、Map或者@Param注解的参数。

如果传递多个参数、没有@param注解，那么MyBatis就会把parameterObject变为一个Map<String, Object>对象，其键值的关系是按顺序来规划的，类似于{‘1’:p1,”2”:p2……},所以在编写的时候可以使用#{1}去引用第一个参数。 如果使用了@param注解，也会变为Map<String,Object>对象，类似为{“key1”:p1,”key2”:p2,……}

parameterMappings,是一个List，每一个元素都是ParameterMapping的对象，这个对象会描述我们的参数，包括属性、名称、表达式、javaType、jdbcType、typeHandler等重要信息

6.3 SqlSession运行过程

6.3.1映射器的动态代理

Mybatis为什么只用Mapper接口便能够运行SQL，因为映射器的XML文件的命名空间对应的便是这个接口的全路径，那么它根据全路径和方法名便能够绑定起来，通过动态代理技术，让这个接口跑起来。而后采用命令模式，最后还是使用SqlSession接口的方法使得它能够执行查询，有了这层封装我们便可以使用接口编程。

6.3.2SqlSession下的四大对象

映射器就是一个动态代理对象，进入到了MapperMethod的execute方法。它经过简单判断就进入了SqlSession的删除、更新、插入、选择等方法。

Mapper执行的过程是通过Executor、StatementHandler、ParameterHandler和ResultHandler来完成数据库操作和结果返回的。

1.Execute代表执行器，由它来调度StatementHandler、ParameterHandler、ResultHandler等来执行对应的SQL.

2.StatementHandler的作用是使用数据库的statement（preparedstatemet）执行操作，它是四大对象的核心，起到承上启下的作用

3.parameterHandler用于SQL对参数的处理

4．ResultHandler是进行最后数据集（ResultSet）封装返回处理的

执行器

SIMPLE:简易执行器，默认。REUSE:是一种执行器重用预处理语句。

BATCH:执行器重用语句和批量更新，它是针对批量专用的执行器。

interceptorChain。pluginAll（executor）；

Mybatis根据Configuration来构建StatementHandler，然后使用prepareStatement方法，对SQL编译并对参数进行初始化，在它的实现过程，它调用了StatementHandler的prepare（）进行了预编译和基础设置，然后通过StatementHandler的parameterize()来设置参数并执行，resultHandler再组装查询结果返回给调用者来完成一次查询。

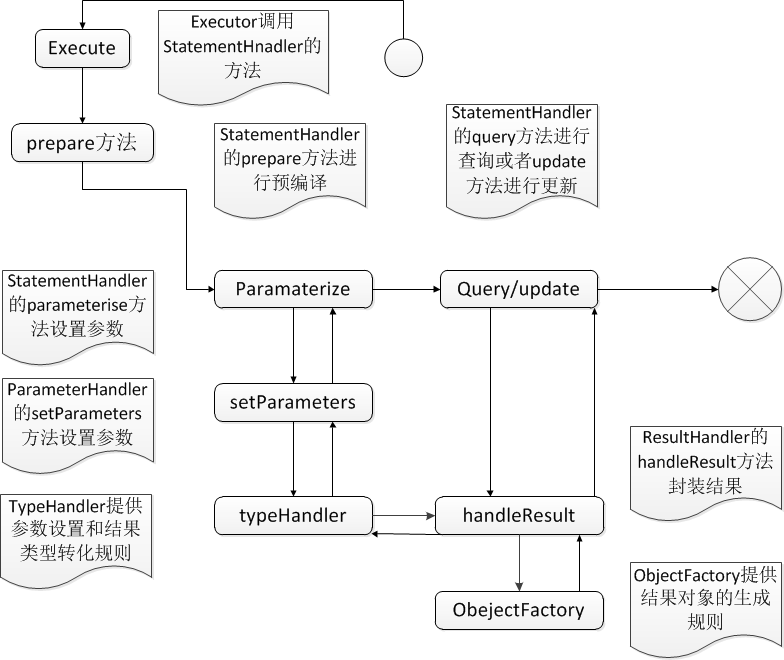
数据库会话器（statementHandler）专门处理数据库会话

RoutingStatementHandler实现了接口StatementHandler，通过适配器模式找到对应的StatementHandler来执行， 一般分为三种：SimpleStatementHandler、PresparedStatementHandler、CallableStatementHandler。对应了三种执行器。

Executor会先调用StatementHandler的prepare()方法预编译SQL语句，同时设置一些基本运行的参数。然后用parameterize()方法启用ParameterHandler设置参数，完成预编译，跟着就是执行查询。

6.3.3SqlSession运行总结

SqlSession是通过Executor创建StatementHandler来运行的，而StatementHandler要经过下面三步：prepared预编译SQL;parameterize设置参数；query/update执行SQL



7.插件

7.1插件接口。在Mybatis中使用插件，必须实现接口Interceptor。其中的

1.intercept方法将直接覆盖你所拦截对象原有的方法。它的参数Invocation对象可以用来反射调度原来对象的方法。

2.plugin方法：target是被拦截对象，它的作用是给被拦截对象生成一个代理对象，并返回它

3.setProperties方法：允许在plugin元素中配置所需参数，方法在插件初始化的时候就被调用了一次，然后把插件对象存入到配置中，以便后面再取出。

7.3插件的代理和反射设计

插件用的是责任链模式。 由interceptorChain去定义的。

Executor = （Executor） interceptorChain.pluginAll(executor)

Plugin方法生成代理对象，当它取出插件的时候是从Configuration对象中去取出的。

Plugin类用来生成代理对象，实现了InvocationHandler接口。其中的wrap方法用来生成动态代理对象。

在Invoke方法中，如果存在签名的拦截方法，插件的intercept方法就会被在这里调用，然后返回结果。如果不存在签名方法，将直接反射调度我们要执行的方法。

Proceed()方法推动代理对象运行，调度proceed（）方法时，mybatis总是从最后一个代理对象运行到第一个代理对象，最后是真实被拦截的对象方法被运行。

插件实例

@Intercepts说明它是一个拦截器，@Signature是注册拦截器签名的地方，只有签名满足条件才能拦截，type可以是四大对象中的一个，method代表要拦截四大对象的某一种接口方法，而args表示该方法的参数，需要根据拦截对象的方法参数进行设置。

（避免数据量过大，配置查询50条数据）思路，拦截StatementHandler对象，在预编译前修改语句来满足需求。

@Intercepts（{Signature(type=StatementHandler.class, method=”prepare”, args={Connection.class}) }）

Public class QueryLimitPlugin implements Interceptor {

Private int limit;

Private String dbtype;

Private static final String LMT\_TABLE\_NAME=”limit\_Table\_Name\_xxx”;

@override

Public Object intercept(Invocation invocation) throws Throwable {

StatementHandler stmtHandler = (StatementHandler) invocation.getTarget();

MetaObject metaStmHandler = SystemMetaObject.forObject(stmtHandler);

//分离代理对象，从而形成多次代理，通过两次循环最原始的被代理类，mybatis使用的是jdk代理

While （metaStmtHandler.hasGetter(“h”)） {

Object object = metaStmtHandler.getValue(“h”);

metaStmtHandler = SystemMetaObject.forObject(object);

}

//分离最后一个代理对象的目标类

While(metaStmtHandler.hasGetter(“target”)) {

Object object = metaStmtHandler.getValue(“target”);

metaStmtHandler = SystemMetaObject.forObject(object);

}

//取出即将要执行的SQL

String sql = (String) metaStmtHandler.getValue=(“delegate.boundSql.sql”);

String limitSql;

//判断参数是不是Mysql数据库且sql有没有被插件重写过。

If(“mysql”.equals(this.dbtype) && sql.indexOf(LMT\_TABLE\_NAME) == -1) {

//去掉前后空格

Sql = sql.trim();

//将参数写入SQL

limitSql = ”select \* from (“+sql + “) ”+LMT\_TABLE+NAME+”limit ”+limit;

//重写要执行的SQL

metaStmtHandler.setValue(“deletgate.boundSql.sql”, limitSql);

}

//调用原来对象的方法，进入责任链的下一层级

Return invocation.proceed();

}

@Override

Public Object plugin(Object target) {

//使用默认的mybatis提供的类生成代理对象

Return Plugin.wrap(target, this);

}

@Overrider

Public void setProperties(Properties props) {

String strLimit = (String)props.getProperty(“limit”, “50”);

This.limit = Integer.parseInt(strLimit);

This.dbType=(String)props.getProperty(“dbtype”,”mysql”);

}

}

这个插件的intercept方法会覆盖掉StatementHandler的prepare方法，我们先从代理对象分离出真实对象，然后根据需要修改sql，来达到限制返回行数的需求。

<plugins>

<plugin interceptor=”com.learn.plugin.QueryLimitPlugin”>

<property name=”dbtype” value=”mysql”/>

<property name=”limit” value=”50”/>

</plugin>

</plugins>

注意：尽量不要用插件，

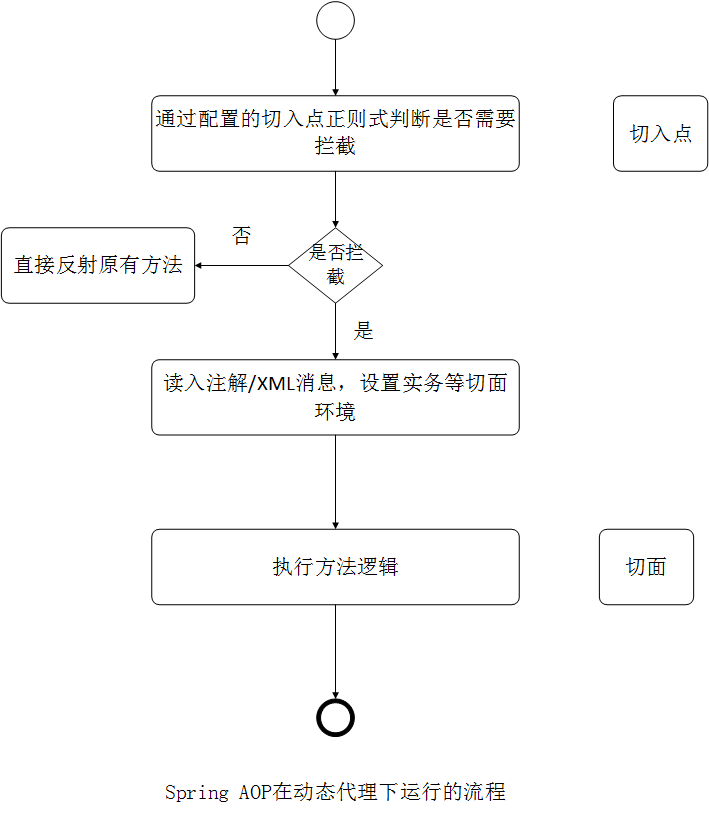
1. 它将修改Mybatis底层设计。
2. 责任链模式，层层代理，性能消耗高
3. 了解四大对象及其方法的作用，准确判断要拦截什么对象，什么方法，参数是什么，才能确定签名如何编写。

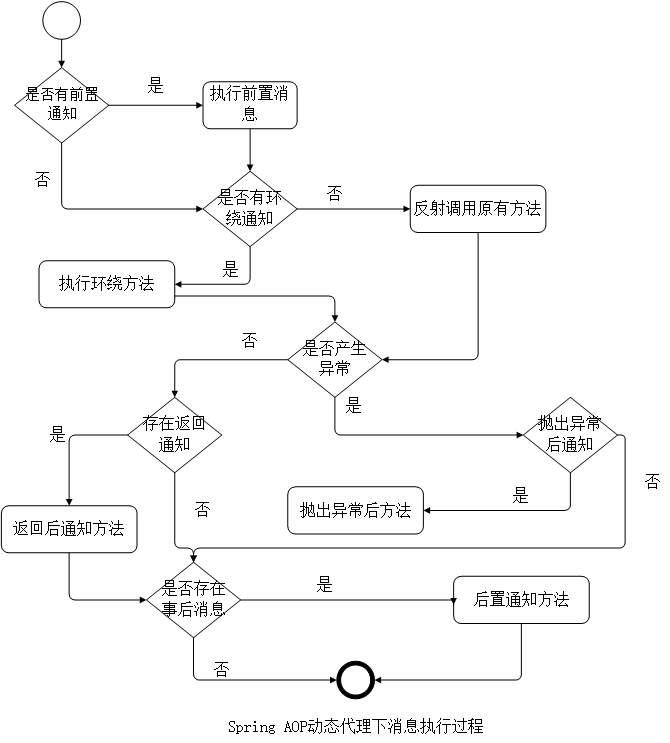
8、MyBatis-Spring

1.spring会判断到底要不要拦截这个方法，这是一个切入点的配置问题

2.切面是插入角色的，里面包含事务，而事务就是整个方法的一个切面

3.连接点是在程序运行中根据不同的通知来实现的程序段。前置、后置、正常返回后、抛出异常后、环绕通知。





8.1.3 Spring事务管理

事务隔离级别

脏读：一个事务能够读取另外一个事务未提交的数据。

不可重复读是针对同一条记录，而幻读是针对删除和插入记录的。

序列化就意味着所有的操作都会按顺序执行，不会出现脏读、不可重复读和幻读。

|  |  |  |  |
| --- | --- | --- | --- |
| 项目 | 脏读 | 不可重复读 | 幻读 |
| 脏读 | √ | √ | √ |
| 读写提交 | × | √ | √ |
| 可重复读 | × | × | √ |
| 序列化 | × | × | × |

性能 从脏读→读写提交→可重复读→序列化直线下降

7种传播行为

8.2 MyBatis-Spring应用

9.2批量更新

<settings><setting name=”defaultExecutorType” value=”BATCH”/></settings>

一旦使用了批量执行器，那么在默认的情况下，它在cmmit后才发送SQL到数据库。

如果不想提交事务，只要执行SqlSession的flushStatements方法就可以将当前缓存的SQL发送给数据库执行。

9.3 调用存储过程

存储过程说白了就是一堆SQL的合并，中间加了点逻辑控制。

存储过程只在创造时进行编译，以后每次执行都不需再重新编译，提高运行效率。将复杂操作封装在一起，只需连接一次数据库。

9.3.1存储过程in和out参数的使用

CREATE OR REPLACE PROCEDURE count\_role(p\_role\_name in varchar, count\_total out int, exec\_date out date)

IS

BEGIN

Select count(\*) into count\_total from t\_role where role\_name like ‘%’ || p\_role\_name || ‘%’;

Select sysdate into exec\_date from dual;

END

In参数是一个输入的参数，out参数则是一个输出的参数。首先把模糊查询的结果保存到count\_total这个out参数中，并且将当前日期保存在out\_date这个参数中。

<select id=”count” parameterType=”com.learn.pojo.ProcedurePojo” statementType=”CALLABLE”>

{call count\_role ( #{roleName, mode=IN, jdbcType=VARCHAR},

#{result, mode=OUT, jdbcType=INTEGER},

#{execDate, mode=OUT, jdbcType=DATE}

)}

</select>

这里的statementType=”CALLABLE”告诉MyBatis我们将用存储过程的方式去执行它。

Public interface ProcedureMapper {

Public void count (ProcedurePojo pojo); }

9.3.2存储过程游标

在存储过程中往往还需要返回游标。MyBatis对存储过程的游标提供了一个JdbcType=CURSOR的支持，它可以智能地把游标读到的数据通过配置的映射关系映射到某个类型的POJO上。

Create or replace procedure find\_role( p\_role\_name in varchar, p\_start in int, p\_end in int, r\_count out int, ref\_cur out sys\_refcursor) AS

BEGIN

Select count(\*) into r\_count from t\_role where role\_name like ‘%’ || p\_role\_name || ‘%’;

Open ref\_cur for

Select role\_no, role\_name, note, create\_date from (SELECT role\_no, role\_name,note,create\_date,rownum as row1 FROM t\_role a where a.role\_name like ‘%’ || p\_role\_name || ‘%’ and rownum <=p\_end) where row1>p\_start;

End find\_role;

定义游标返回的映射规则

<result id=”roleMap” type=”com.learn.pojo.TRole”>

<id property=”roleName” column=”ROLE\_NAME”/>

<result property=”roleNo” column=”USER\_NAME”/>

<result property=”note” column=”NOTE”/>

<result property=”createDate” column=”CREATE\_DATE”/>

</resultMap>

<select id=”findRole” parameterType=”com.learn.pojo.PageRole” statementType=”CALLABLE”>

{call find\_role( #{roleName, mode=IN, jdbcType=VARCHAR}, #{start, mode=IN,jdbcType=INTEGER}, #{end, mode=IN,jdbcType=INTEGER}, #{count, mode=OUT, jdbcType=INTEGER}, #{roleList,mode=OUT,jdbcType=CURSOR,javaType=ResultSet,resultMap=roleMap} )}

</select>

9.4分表

表名也是参数 <select id=”getBill” resultType=”com.learn.pojo.Bill”> select id,bill\_name as billName, note from t\_bill\_${year} where id=#{id} </select>

${year}的含义是直接让参数加入到SQL中。总之，我们可以使用这样一个规则：让SQL的任何部分都可以被参数改写，包括列名，以此来满足不同的需求。

9.5分页

9.5.1RowBounds分页

Mybatis中的类RowBounds会在一条SQL中查询所有的结果出来，然后根据从第几条到第几条取出数据返回 如果返回很多数据则很容易抛出内存溢出。

RowsBounds分页更适合在一些返回数据结果较少的查询中使用。

9.5.2插件分页

9.7 在映射中使用枚举

定义一个自定义的typeHandler

Public class ColorEnumTypeHandler implements TypeHandler<Color> {

@Override

Public void setParameter(PreparedStatement ps, int I, Color color, JdbcType jdbcType) throws SQLException {

Ps.setInt(i, color.getCode());

}

@Override

Public Color getResult(ResultSet rs, String name) throws SQLException {

Int result = rs.getInt(name);

Return Color.getEnumByCode(result);

}

@Override

Public Color getResult(ResultSet rs, int i) throws SQLException {

Int result = rs.getInt(i);

Return Color.getEnumByCode(result);

}

@Override

Public Color getResult(CallableStatement cs, int i)throws SQLException {

Int result= cs.getInt(i);

Return Color.getEnumByCode(result);

}

}

<resultMap id=”colorResultMapper” type=”com.learn.pojo.ColorBean”>

<result property=”color” column=”color” jdbcType=”INTEGER” jdbcType=”com.learn.typeHandler.Color” typeHandler=”com.learn.typeHandler.ColorEnumTypeHandler”/>

</resultMap>

9.8多对多级联

拆解为两个一对多的关联 把关联的那一项设为延迟加载fetchType=lazy