MyBatis

[1. 执行流程 1](#_102)

[2. #{}和${} 1](#_101)

[3. 核心对象 1](#_106)

[4. 分页方式 2](#_104)

[4.1. 逻辑分页 3](#_107)

[4.2. 物理分页 3](#_108)

[5. 批量注册Mapper 3](#_105)

[6. 执行器 3](#_109)

[7. 与Spring整合 4](#_103)

[8. 配置解析 4](#_111)

[8.1. properties 5](#_112)

[8.2. setting 5](#_113)

[8.3. typeAlias 5](#_114)

[8.4. plugin 5](#_115)

[8.5. typeHandler 5](#_116)

[8.6. mapper解析 5](#_123)

[9. 执行过程 5](#_117)

[9.1. 为mapper生成接口类 6](#_118)

[9.2. 根据配置信息生成SQL，并将运行时参数设置到SQL 6](#_119)

[9.3. 一二级缓存的实现 6](#_120)

[9.4. 插件机制 6](#_121)

[9.5. 数据库连接和获取 6](#_122)

[9.6. 查询结果处理和延迟加载 6](#_124)

# 执行流程

1. 创建SqlSessionFactory：SqlSessionFactory f = new SqlSessionFactoryBuilder().build(reader)

2. 创建SqlSession：f.openSession()

3. 使用动态代理获得Mapper：mapper = sqlSession.getMapper(XXX.class)

4. 执行Mapper接口方法：xxxMapper.select...()

# #{}和${}

#{}在动态SQL解析后变为?，然后在JDBC的预编译中变成相应参数。

${}在动态SQL解析后就会变成相应参数，可能会有SQL注入的风险。

SQL预编译指的是数据库驱动在发送SQL语句和参数给DBMS之前对SQL进行编译，这样DBMS执行SQL时，就不需要编译了。

预编译能够防止SQL注入是因为：SQL语句在程序运行前已经进行了预编译，在程序运行时第一次操作数据库之前，SQL语句已经被数据库分析，编译和优化，对应的执行计划也会缓存下来并允许数据库以参数化的形式进行查询，当运行时动态地把参数传给PreprareStatement时，即使参数里有敏感字符如 or '1=1'。数据库也会将其作为一个字段的属性值来处理而不会作为一个SQL指令。

# 核心对象

ParameterHandler：处理SQL的参数对象

ResultSetHandler：处理SQL的返回结果集

StatementHandler：数据库的处理对象，用于执行SQL语句

有三种：SimpleStatementHandler、PreparedStatementHandler和CallableStatementHandler。分别对应于JDBC的Statement、PreparedStatement和CallableStatement。

RoutingStatementHandler则根据statementType创建不同的StatementHandler代理。

Executor：MyBatis的执行器，用于执行增删改查

有三种：SimpleExecutor，ReuseExecutor，BatchExecutor。

# 分页方式

1. 数组分页

通过对所有查询结果进行自主分页

2. SQL分页

在SQL语句上传入limit参数

MyBatis接口：List<Student> queryStudentsBySql(Map<String,Object> data);

XML文件：

<select id="queryStudentsBySql" parameterType="map" resultMap="studentmapper">

select \* from student limit #{currIndex} , #{pageSize}

</select>

实现：

public List<Student> queryStudentsBySql(int currPage, int pageSize) {

Map<String, Object> data = new HashedMap();

data.put("currIndex", (currPage-1)\*pageSize);

data.put("pageSize", pageSize);

return studentMapper.queryStudentsBySql(data);

}

}

3. 拦截器分页

通过在生成PrepareStatement前改变SQL语句，从而实现分页。实际上拦截StatementHandler的prepare方法。

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

public class MyInterceptor implements Interceptor {

@Override

public Object intercept(Invocation invocation) throws Throwable {

RoutingStatementHandler handler = (RoutingStatementHandler) invocation.getTarget();

StatementHandler delegate = (StatementHandler) getFieldValue(handler, "delegate");

MappedStatement mappedStatement = (MappedStatement) getFieldValue(delegate, "mappedStatement");

if (mappedStatement.getId().matches(".+ByPage")) {

BoundSql boundSql = handler.getBoundSql();

String sql = (String) getFieldValue(boundSql, "sql");

setFieldValue(boundSql, "sql", sql + " limit 2, 2");

}

return invocation.proceed();

}

...

}

4. RowBound分页

List<Student> students = session.selectList("com.twd.mapper.StudentMapper.selectAll", null, new RowBounds(0, 5));

## 逻辑分页

将所有数据读入内存中，再进行分页。

## 物理分页

只向数据库读取定量数据。

# 批量注册Mapper

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">

<property name="basePackage" value="com.twd.mapper" />

</bean>

# 执行器

MyBatis有三种基本的执行器

1. SimpleExecutor：每执行一次update或select，就开启一个Statement对象，用完立刻关闭Statement。

2. ReuseExecutor：执行update或select，以SQL作为key查找Statement对象，存在就使用，不存在就创建。用完后不关闭Statement对象，而是放在map中，供下一次使用。

3. BatchExecutor：执行update（没有select），将所有SQL都添加到批处理中，等待统一执行（executeBatch()），它缓存了多个Statement对象。每个Statement对象都是addBatch()完毕后，等待逐一执行批处理。Batch执行器不仅重用语句，还会批量更新。

其中CachingExecutor是装饰器，给上述三个具体的Executor增加缓存功能。

# 与Spring整合

首先都需要配置dataSource和sqlSessionFactoryBean

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql://172.16.1.127:3306/twd"/>

<property name="username" value="twd"/>

<property name="password" value="123456"/>

</bean>

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

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

</bean>

然后有两种方法配置Mapper

1. 在xml文件中sqlSessionFactory构造Mapper。

<bean id="studentMapper" class="org.mybatis.spring.mapper.MapperFactoryBean">

<property name="mapperInterface" value="com.twd.mapper.StudentMapper"/>

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

</bean>

2. 在xml文件中配置SqlSessionTemplate，然后构造Mapper

<!-- SqlSessionTemplate是一种线程安全的SqlSession，能供多个Mapper使用，可以替代DefaultSqlSession -->

<bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">

<constructor-arg index="0" ref="sqlSessionFactory" />

</bean>

<bean id="studentMapper" class="org.mybatis.spring.mapper.MapperFactoryBean">

<property name="mapperInterface" value="com.twd.mapper.StudentMapper"/>

<property name="sqlSessionTemplate" ref="sqlSession"/>

</bean>

# 配置解析

通过构建xml树，从根节点configuration依次往下进行配置。

## properties

1. 先将子节点内容转化为属性对象Properties，并设置为defaults。

2. 通过当前<properties>的属性url或resource加载并解析属性文件，添加到defaults中。

3. 最后将configuration原本的属性添加进行，并设置成为configuration的最终属性变量。

## setting

首先调用settingsAsProperties方法

1. 解析setting子节点内容，并将解析结果转化为Properties对象。

2. 为Configuration创建元信息对象

3. 通过MetaClass检测Configuration中是否存在某个属性的setter方法；不存在则抛出异常。

4. 若通过MetaClass的检测，则返回Properties对象，方法逻辑结束。

实际原理是通过反射判断Configuration.class中是否有对应属性的set方法。

## typeAlias

其实就是构建一个别名到class对象的映射，存在typeAliasRegistry中。

## plugin

会对插件进行实例化，存储在Configuration中。

## typeHandler

类型处理器，解决jdbcType和JavaType之间的转换关系，可以使用自动扫描（需要在类型转换器上使用@MappedType和@MappedJdbcType分别指明Java类型和Jdbc类型）和xml配置。

## mapper解析

对于mapper的xml文件或是注解类进行解析，然后根据命令空间绑定mapper接口。

# 执行过程

## 为mapper生成接口类

Mapper的实际代理MapperProxyFactory已经存在，但是代理对象都是即时生成的。

MapperMethod也是懒加载的，但是生成后就会进行缓存。

## 根据配置信息生成SQL，并将运行时参数设置到SQL

MyBatis会将映射文件中的SQL解析成一组SQL片段。如果某个片段中也包含SQL相关标签，那么MyBatis会对该片段进行再次分片。最终，一个SQL配置将会被解析成一个SQL片段树。

根据SQL配置中是否包含${}，<if>和<where>结点，会选择是否使用DynamicSqlSource去存储SQL片段；否则使用RawSqlSource存储。#{}则被设置为？

## 一二级缓存的实现

## 插件机制

## 数据库连接和获取

分为池化的和非池化的。这里主要讲下PooledDataSource。

PooledDataSource会维护两个池，一个是空闲池，一个是活跃池。

当需要获得连接时，会经历以下步骤：

1. 查看空闲池是否有连接，有的话直接用。

2. 查看活跃池的连接数是否达到上限，没有的话创建一个。

3. 活跃池中最老的连接是否超时，超时的话回滚当前事务，复用该老连接创建一个新连接。

4. 若未超时，继续等待。

## 查询结果处理和延迟加载