MyBatis 注解

一 mybaits 常见注解

> 此处忽略了其他配置,仅仅是对注解的使用类进行了编写,如果使用非 springboot 方式,请按照普通方式添加 xml 配置文件

1.1 mapper接口

```
@CacheNamespace(size = 512)
public interface TestMapper {

    /**
     *
     * @param id
     * @return
     */
    @SelectProvider(type = TestSqlProvider.class, method = "getSql")
    @Options(useCache = true, flushCache = false, timeout = 10000)
    @Results(value = {
            @Result(id = true, property = "id", column = "test_id", javaType =
String.class, jdbcType = JdbcType.VARCHAR),
            @Result(property = "testText", column = "test_text", javaType =
String.class, jdbcType = JdbcType.VARCHAR) })
    public TestBean get(@Param("id") String id);

    /**
     *
     * @return
     */
    @SelectProvider(type = TestSqlProvider.class, method = "getAllSql")
    @Options(useCache = true, flushCache = false, timeout = 10000)
    @Results(value = {
            @Result(id = true, property = "id", column = "test_id", javaType =
String.class, jdbcType = JdbcType.VARCHAR),
            @Result(property = "testText", column = "test_text", javaType =
String.class, jdbcType = JdbcType.VARCHAR) })
    public List<TestBean> getAll();

    /**
     *
     * @param testText
     * @return
```

```
     */
    @SelectProvider(type = TestSqlProvider.class, method = "getByTestTextSql")
    @Options(useCache = true, flushCache = false, timeout = 10000)
    @ResultMap(value = "getByTestText")
    public List<TestBean> getByTestText(@Param("testText") String testText);

    /**
     * insert a test bean into database.
     *
     * @param testBean
     */
    @InsertProvider(type = TestSqlProvider.class, method = "insertSql")
    @Options(flushCache = true, timeout = 20000)
    public void insert(@Param("testBean") TestBean testBean);

    /**
     *
     * @param testBean
     */
    @UpdateProvider(type = TestSqlProvider.class, method = "updateSql")
    @Options(flushCache = true, timeout = 20000)
    public void update(@Param("testBean") TestBean testBean);

    /**
     *
     * @param id
     */
    @DeleteProvider(type = TestSqlProvider.class, method = "deleteSql")
    @Options(flushCache = true, timeout = 20000)
    public void delete(@Param("id") String id);
}
```

1.2 注解解释

`@CacheNamespace(size = 512)` ： 定义在该命名空间内允许使用内置缓存，最大值为512个对象引用，读写默认是开启的，缓存内省刷新时间为默认3600000毫秒，写策略是拷贝整个对象镜像到全新堆（如同CopyOnWriteList）因此线程安全。

`@SelectProvider(type = TestSqlProvider.class, method = "getSql")` ： 提供查询的SQL语句，如果你不用这个注解，你也可以直接使用`@Select("select * from ....")`注解，把查询SQL抽取到一个类里面，方便管理，同时复杂的SQL也容易操作，`type = TestSqlProvider.class`就是存放SQL语句的类，而`method = "getSql"`表示get接口方法需要到`TestSqlProvider`类的`getSql`方法中获取SQL语句。

`@Options(useCache = true, flushCache = false, timeout = 10000)` ： 一些查询的选项开关，比如`useCache = true`表示本次查询结果被缓存以提高下次查询速度，`flushCache = false`表示下次查询时不刷新缓存，`timeout = 10000`表示查询结果缓存10000秒。

```
@Results(value = {
@Result(id = true, property = "id", column = "test_id", javaType = String.class,
jdbcType = JdbcType.VARCHAR),
@Result(property = "testText", column = "test_text", javaType = String.class,
jdbcType = JdbcType.VARCHAR) })
```

： 表示sql查询返回的结果集，`@Results`是以`@Result`为元素的数组，`@Result`表示单条属性-字段的映射关系，如：`@Result(id = true, property = "id", column = "test_id", javaType = String.class, jdbcType = JdbcType.VARCHAR)`可以简写为：`@Result(id = true, property = "id", column = "test_id")`，`id = true`表示这个`test_id`字段是个PK，查询时mybatis会给予必要的优化，应该说数组中所有的`@Result`组成了单个记录的映射关系，而`@Results`则单个记录的集合。另外还有一个非常重要的注解`@ResultMap`也和`@Results`差不多，到时会讲到。

`@Param("id")` ：全局限定别名，定义查询参数在sql语句中的位置不再是顺序下标`0,1,2,3....`的形式，而是对应名称，该名称就在这里定义。

`@ResultMap(value = "getByTestText")` ：重要的注解，可以解决复杂的映射关系，包括`resultMap`嵌套，鉴别器`discriminator`等等。注意一旦你启用该注解，你将不得不在你的映射文件中配置你的`resultMap`，而`value = "getByTestText"`即为映射文件中的`resultMap ID`(注意此处的`value = "getByTestText"`，必须是在映射文件中指定命名空间路径)。`@ResultMap`在某些简单场合可以用`@Results`代替，但是复杂查询，比如联合、嵌套查询`@ResultMap`就会显得解耦方便更容易管理。

1.3 result map 定义

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
PUBLIC "-//ibatis.apache.org//DTD Mapper 3.0//EN"
"http://ibatis.apache.org/dtd/ibatis-3-mapper.dtd">

<mapper namespace="com.wotao.taotao.persist.test.mapper.TestMapper">
    <resultMap id="getByTestText" type="TestBean">
        <id property="id" column="test_id" javaType="string" jdbcType="VARCHAR" />

        <result property="testText" column="test_text" javaType="string"
jdbcType="VARCHAR" />
    </resultMap>
</mapper>
```

1.4注意

注意文件中的namespace路径必须是使用@resultMap的类路径，此处是TestMapper，文件中id="getByTestText"必须和@resultMap中的value = "getByTestText"保持一致。

@InsertProvider(type = TestSqlProvider.class, method = "insertSql") ：用法和含义@SelectProvider一样，只不过是用来插入数据库而用的。

@Options(flushCache = true, timeout = 20000) ：对于需要更新数据库的操作，需要重新刷新缓存flushCache = true使缓存同步。

@UpdateProvider(type = TestSqlProvider.class, method = "updateSql") ：用法和含义@SelectProvider一样，只不过是用来更新数据库而用的。

@Param("testBean") ：是一个自定义的对象，指定了sql语句中的表现形式，如果要在sql中引用对象里面的属性，只要使用testBean.id，testBean.textText即可，mybatis会通过反射找到这些属性值。

@DeleteProvider(type = TestSqlProvider.class, method = "deleteSql") ：用法和含义@SelectProvider一样，只不过是用来删除数据而用的。

1.5 使用测试

> 开始写SQL语句了，因为我们不再使用映射文件编写SQL，那么就不得不在java类里面写，就像上面提到的，我们不得不在TestSqlProvider这个类里面写SQL，虽然已经把所有sql语句集中到了一个类里面去管理，但听起来似乎仍然有点恶心，幸好mybatis提供SelectBuilder和SqlBuilder这2个小工具来帮助我们生成SQL语句，SelectBuilder专门用来生成select语句，而SqlBuilder则是一般性的工具，可以生成任何SQL语句，我这里选择了SqlBuilder来生成，TestSqlProvider代码如下：

```
package xin.chenjunbo.test.sqlprovider;
```

```
import static org.apache.ibatis.jdbc.SqlBuilder.BEGIN;
import static org.apache.ibatis.jdbc.SqlBuilder.FROM;
import static org.apache.ibatis.jdbc.SqlBuilder.SELECT;
import static org.apache.ibatis.jdbc.SqlBuilder.SQL;
import static org.apache.ibatis.jdbc.SqlBuilder.WHERE;
import static org.apache.ibatis.jdbc.SqlBuilder.DELETE_FROM;
import static org.apache.ibatis.jdbc.SqlBuilder.INSERT_INTO;
import static org.apache.ibatis.jdbc.SqlBuilder.SET;
import static org.apache.ibatis.jdbc.SqlBuilder.UPDATE;
import static org.apache.ibatis.jdbc.SqlBuilder.VALUES;

import java.util.Map;
public class TestSqlProvider {

    /** 表明,这里是 test */
    private static final String TABLE_NAME = "test";

    /**
     * @param parameters
     * @return
     */
    public String getSql(Map<String, Object> parameters) {
        String uid = (String) parameters.get("id");
        BEGIN();
        SELECT("test_id, test_text");
        FROM(TABLE_NAME);
        if (uid != null) {
            WHERE("test_id = #{id,javaType=string,jdbcType=VARCHAR}");
        }
        return SQL();
    }

    /**
     * @return
     */
    public String getAllSql() {
        BEGIN();
        SELECT("test_id, test_text");
        FROM(TABLE_NAME);
        return SQL();
    }

    /**
     * @param parameters
     * @return
     */
    public String getByTestTextSql(Map<String, Object> parameters) {
        String tText = (String) parameters.get("testText");
```

```
        BEGIN();
        SELECT("test_id, test_text");
        FROM(TABLE_NAME);
        if (tText != null) {
            WHERE("test_text like #{testText,javaType=string,jdbcType=VARCHAR}");

        }
        return SQL();
    }

    /**
     * @return
     */
    public String insertSql() {
        BEGIN();
        INSERT_INTO(TABLE_NAME);
        VALUES("test_id", "#{testBean.id,javaType=string,jdbcType=VARCHAR}");
        VALUES("test_text", "#
{testBean.testText,javaType=string,jdbcType=VARCHAR}");
        return SQL();
    }

    /**
     * @return
     */
    public String updateSql() {
        BEGIN();
        UPDATE(TABLE_NAME);
        SET("test_text = #{testBean.testText,javaType=string,jdbcType=VARCHAR}");

        WHERE("test_id = #{testBean.id,javaType=string,jdbcType=VARCHAR}");
        return SQL();
    }

    /**
     * @return
     */
    public String deleteSql() {
        BEGIN();
        DELETE_FROM(TABLE_NAME);
        WHERE("test_id = #{id,javaType=string,jdbcType=VARCHAR}");
        return SQL();
    }
}
```

1.6 静态方法介绍

BEGIN();表示刷新本地线程，某些变量为了线程安全，会先在本地存放变量，此处需要刷新。

SELECT，FROM，WHERE等等都是sqlbuilder定义的公用静态方法，用来组成你的sql字符串。如果你在testMapper中调用该方法的某个接口方法已经定义了参数@Param()，那么该方法的参数Map<String, Object> parameters即组装了
@Param()定义的参数，比如testMapper接口方法中定义参数
为@Param("testId"),@Param("testText")，那么parameters的形态就
是：[key="testId",value=object1],[key="testText",value=object2]，如果接口方法没有定义@Param()，那么parameters的key就是参数的顺序小标：[key=0,value=object1],
[key=1,value=object2]，SQL()将返回最终append结束的字符串，sql语句中的形如
#{id,javaType=string,jdbcType=VARCHAR}完全可简写为#{id}，我只是为了规整如此写而已。另外，对于复杂查询还有很多标签可用，比如：JOIN，INNER_JOIN，GROUP_BY，ORDER_BY等等，具体使用详情，你可以查看源码。

1.7常见注解目录

注解	目标	响应XML	描述
@ CacheNamespace	类		为给定的命名空间（比如类）配置缓存。属性： implemetation,eviction, flushInterval , size 和 readWrite .
@CacheNamespaceRef	类		参照另外一个命名空间的缓存来使用 属性：value，也就是类的完全限定名
@ConstructorArgs	方法		收集一组结果传递给对象构造方法 属性：value，是形式参数的数组
@Arg	方法		单独的构造方法参数，是ConstructorArgs 集合的一部分。属性：id,column,javaType，typeHandler。 id属性是布尔值，来标识用于比较的属性，和XML 元素相似
@TypeDiscriminator	方法		一组实例值被用来决定结果映射的表现。 属性：Column, javaType , jdbcType typeHandler，cases。 cases属性就是实例的数组。
@Case	方法		单独实例的值和它对应的映射。 属性：value ， type , results 。 Results 属性是结果数组，因此这个注解和实际的ResultMap 很相似，由下面的 Results注解指定
@Results	方法		结果映射的列表，包含了一个特别结果列如何被映射到属性或字段的详情。属性：value ， 是Result注解的数组
@Result	方法		在列和属性或字段之间的单独结果映射。 属性：id ， column , property ， javaType , jdbcType , type Handler , one，many. id 属性是一个布尔值 ， 表示了应该被用于比较的属性。one 属性是单独的联系，和 相似，而many 属性是对集合而言的，和 相似。
@One	方法		复杂类型的单独属性值映射。 属性：select，已映射语句（也就是映射器方法）的完全限定名，它可以加载合适类型的实例。注意：联合映射在注解API中是不支持的。
@Many	方法		复杂类型的集合属性映射。 属性：select，是映射器方法的完全限定名，它可加载合适类型的一组实例。注意：联合映射在 Java注解中是不支持的。

@Options	方法	映射语句的属性	这个注解提供访问交换和配置选项的宽广范围，它们通常在映射语句上作为属性出现。而不是将每条语句注解变复杂，Options 注解提供连贯清晰的方式来访问它们。 属性：useCache=true, flushCache=false，resultSetType=FORWARD_ONLY, statementType=PREPARED，fetchSize= -1，timeout=-1 , useGeneratedKeys=false , keyProperty="id"。理解Java 注解是很重要的，因为没有办法来指定"null "作为值。因此，一旦你使用了 Options注解，语句就受所有默认值的支配。要注意什么样的默认值来避免不期望的行为
@Insert @Update @Delete	方法		这些注解中的每一个代表了执行的真实 SQL。它们每一个都使用字符串数组（或单独的字符串）。如果传递的是字符串数组，它们由每个分隔它们的单独空间串联起来。 属性：value，这是字符串数组用来组成单独的SQL语句
@InsertProvider @UpdateProvider @DeleteProvider @SelectProvider	方法		这些可选的SQL注解允许你指定一个类名和一个方法在执行时来返回运行的 SQL。基于执行的映射语句， MyBatis会实例化这个类，然后执行由 provider 指定的方法. 这个方法可以选择性的接受参数对象作为它的唯一参数，但是必须只指定该参数或者没有参数。 属性： type, method。type 属性是类的完全限定名。method 是该类中的那个方法名
@Param	参数	N/A	当映射器方法需多个参数，这个注解可以被应用于映射器方法参数来给每个参数一个名字。否则，多参数将会以它们的顺序位置来被命名。比如 #{1}，#{2} 等，这是默认的。使用@Param("person")，SQL中参数应该被命名为#{person}。