Mybatis注解

| **注解** | **目标** | **相对应的 XML** | **描述** |
| --- | --- | --- | --- |
| @CacheNamespace | 类 | <cache> | 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 | <cacheRef> | 参照另外一个命名空间的缓存来使用。 属性:value,应该是一个名空间的字 符串值(也就是类的完全限定名) 。 |
| @ConstructorArgs | Method | <constructor> | 收集一组结果传递给一个劫夺对象的 构造方法。属性:value,是形式参数 的数组。 |
| @Arg | 方法 | <arg>  <idArg> | 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<idArg>XML 元素相似。 |
| @TypeDiscriminator | 方法 | <discriminator> | 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 | <case> | 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 | <resultMap> | 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, id。value 属性是 Result 注解的数组。 The id attribute is the name of the result mapping. |
| @Result | 方法 | <result>  <id> | 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(和在 XML 映射 中的<id>相似)的属性。one 属性是单 独 的 联 系, 和 <association> 相 似 , 而 many 属 性 是 对 集 合 而 言 的 , 和 <collection>相似。 它们这样命名是为了 避免名称冲突。 |
| @One | 方法 | <association> | 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。这是因为 Java 注解的 限制,不允许循环引用。 fetchType, which supersedes the global configuration parameterlazyLoadingEnabled for this mapping. |
| @Many | 方法 | <collection> | A mapping to a collection property of a complex type. Attributes: select, which is the fully qualified name of a mapped statement (i.e. mapper method) that can load a collection of instances of the appropriate types,fetchType, which supersedes the global configuration parameterlazyLoadingEnabled for this mapping. NOTE You will notice that join mapping is not supported via the Annotations API. This is due to the limitation in Java Annotations that does not allow for circular references. |
| @MapKey | 方法 |  | 复 杂 类 型 的 集合 属 性 映射 。 属 性 : select,是映射语句(也就是映射器方 法)的完全限定名,它可以加载合适类 型的一组实例。注意:联合映射在 Java 注解中是不支持的。这是因为 Java 注 解的限制,不允许循环引用。 |
| @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=FlushCachePolicy.DEFAULT , resultSetType=FORWARD\_ONLY , statementType=PREPARED , fetchSize=-1 , , timeout=-1 useGeneratedKeys=false , keyProperty=”id” , keyColumn=”” , resultSets=””。 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 * @Select | 方法 | <insert>  <update>  <delete>  <select> | 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。这就当用 Java 代码构 建 SQL 时避免了“丢失空间”的问题。 然而,如果你喜欢,也欢迎你串联单独 的字符串。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 | <insert>  <update>  <delete>  <select> | 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 允许创建动态 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. 该方法可以有选择地接受参数对象.(In MyBatis 3.4 or later, it’s allow multiple parameters) 属性: type,method。type 属性是类。method 属性是方法名。 注意: 这节之后是对 类的 讨论,它可以帮助你以干净,容于阅读 的方式来构建动态 SQL。 |
| @Param | Parameter | N/A | 如果你的映射器的方法需要多个参数, 这个注解可以被应用于映射器的方法 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 (不包括任何 RowBounds 参数) 比如。 #{param1} , #{param2} 等 , 这 是 默 认 的 。 使 用 @Param(“person”),参数应该被命名为 #{person}。 |
| @SelectKey | Method | <selectKey> | This annotation duplicates the <selectKey> functionality for methods annotated with @Insert, @InsertProvider, @Update or@UpdateProvider. It is ignored for other methods. If you specify a@SelectKey annotation, then MyBatis will ignore any generated key properties set via the @Options annotation, or configuration properties. Attributes: statement an array of strings which is the SQL statement to execute, keyProperty which is the property of the parameter object that will be updated with the new value, before which must be either true orfalse to denote if the SQL statement should be executed before or after the insert, resultType which is the Java type of the keyProperty, andstatementType=PREPARED. |
| @ResultMap | Method | N/A | This annotation is used to provide the id of a <resultMap> element in an XML mapper to a @Select or @SelectProvider annotation. This allows annotated selects to reuse resultmaps that are defined in XML. This annotation will override any @Results or @ConstructorArgs annotation if both are specified on an annotated select. |
| @ResultType | Method | N/A | This annotation is used when using a result handler. In that case, the return type is void so MyBatis must have a way to determine the type of object to construct for each row. If there is an XML result map, use the @ResultMap annotation. If the result type is specified in XML on the<select> element, then no other annotation is necessary. In other cases, use this annotation. For example, if a @Select annotated method will use a result handler, the return type must be void and this annotation (or @ResultMap) is required. This annotation is ignored unless the method return type is void. |
| @Flush | Method | N/A | If this annotation is used, it can be called theSqlSession#flushStatements() via method defined at a Mapper interface.(MyBatis 3.3 or above) |

**映射申明样例**

这个例子展示了如何使用 @SelectKey 注解来在插入前读取数据库序列的值：

@Insert("insert into table3 (id, name) values(#{nameId}, #{name})")

@SelectKey(statement="call next value for TestSequence", keyProperty="nameId", before=**true**, resultType=**int.class**)

**int** insertTable3(Name name);

这个例子展示了如何使用 @SelectKey 注解来在插入后读取数据库识别列的值：

@Insert("insert into table2 (name) values(#{name})")

@SelectKey(statement="call identity()", keyProperty="nameId", before=**false**, resultType=**int.class**)

**int** insertTable2(Name name);

This example shows using the @Flush annotation to call the SqlSession#flushStatements():

@Flush

List<BatchResult> flush();

These examples show how to name a ResultMap by specifying id attribute of @Results annotation.

@Results(id = "userResult", value = {

@Result(property = "id", column = "uid", id = **true**),

@Result(property = "firstName", column = "first\_name"),

@Result(property = "lastName", column = "last\_name")

})

@Select("select \* from users where id = #{id}")

User getUserById(Integer id);

@Results(id = "companyResults")

@ConstructorArgs({

@Arg(property = "id", column = "cid", id = **true**),

@Arg(property = "name", column = "name")

})

@Select("select \* from company where id = #{id}")

Company getCompanyById(Integer id);

This example shows solo parameter using the Sql Provider annotation:

@SelectProvider(type = UserSqlBuilder.class, method = "buildGetUsersByName")

List<User> getUsersByName(String name);

class UserSqlBuilder {

public String buildGetUsersByName(final String name) {

return new SQL(){{

SELECT("\*");

FROM("users");

if (name != null) {

WHERE("name like #{value} || '%'");

}

ORDER\_BY("id");

}}.toString();

}

}

This example shows multiple parameters using the Sql Provider annotation:

@SelectProvider(type = UserSqlBuilder.class, method = "buildGetUsersByName")

List<User> getUsersByName(

@Param("name") String name, @Param("orderByColumn") String orderByColumn);

class UserSqlBuilder {

// If not use @Param, you should be define same arguments with mapper method

public String buildGetUsersByName(

final String name, final String orderByColumn) {

return new SQL(){{

SELECT("\*");

FROM("users");

WHERE("name like #{name} || '%'");

ORDER\_BY(orderByColumn);

}}.toString();

}

// If use @Param, you can define only arguments to be used

public String buildGetUsersByName(@Param("orderByColumn") final String orderByColumn) {

return new SQL(){{

SELECT("\*");

FROM("users");

WHERE("name like #{name} || '%'");

ORDER\_BY(orderByColumn);

}}.toString();

}

}