二

04 MyBatis 反射工具箱：带你领略不一样的反射设计思路

反射是 Java 世界中非常强大、非常灵活的一种机制。在面向对象的 Java 语言中，我们只能按照 public、private 等关键字的规范去访问一个 Java 对象的属性和方法，但反射机制可以让我们在运行时拿到任何 Java 对象的属性或方法。

有人说反射打破了类的封装性，破坏了我们的面向对象思维，我倒不这么认为。我觉得正是由于 Java 的反射机制，解决了很多面向对象无法解决的问题，才受到众多 Java 开源框架的青睐，也出现了有很多惊艳的反射实践，当然，这也包括 MyBatis 中的反射工具箱。

凡事都有两面性，越是灵活、越是强大的工具，用起来的门槛就越高，反射亦如此。这也是写业务代码时，很少用到反射的原因。反过来说，如果必须要用反射解决业务问题的时候，就需要停下来思考我们的系统设计是不是有问题了。

为了降低反射使用门槛，MyBatis 内部封装了一个反射工具箱，其中包含了 MyBatis 自身常用的反射操作，MyBatis 其他模块只需要调用反射工具箱暴露的简洁 API 即可实现想要的反射功能。

反射工具箱的具体代码实现位于 org.apache.ibatis.reflection 包中，下面我就带你一起深入分析该模块的核心实现。

Reflector

Reflector 是 MyBatis 反射模块的基础。要使用反射模块操作一个 Class，都会先将该 Class 封装成一个 Reflector 对象，在 Reflector 中缓存 Class 的元数据信息，这可以提高反射执行的效率。

1. 核心初始化流程

既然是涉及反射操作，Reflector 必然要管理类的属性和方法，这些信息都记录在它的核心字段中，具体情况如下所示。

- type（Class<?> 类型）：该 Reflector 对象封装的 Class 类型。

- readablePropertyNames、writablePropertyNames（String[] 类型）：可读、可写属性的名称集合。
- getMethods、setMethods（Map<String, Invoker> 类型）：可读、可写属性对应的 getter 方法和 setter 方法集合，key 是属性的名称，value 是一个 Invoker 对象。Invoker 是对 Method 对象的封装。
- getTypes、setTypes（Map<String, Class<?>> 类型）：属性对应的 getter 方法返回值以及 setter 方法的参数值类型，key 是属性名称，value 是方法的返回值类型或参数类型。
- defaultConstructor（Constructor<?> 类型）：默认构造方法。
- caseInsensitivePropertyMap（Map<String, String> 类型）：所有属性名称的集合，记录到这个集合中的属性名称都是大写的。

在我们构造一个 Reflector 对象的时候，传入一个 Class 对象，通过解析这个 Class 对象，即可填充上述核心字段，整个核心流程大致可描述为如下。

1. 用 type 字段记录传入的 Class 对象。
2. 通过反射拿到 Class 类的全部构造方法，并进行遍历，过滤得到唯一的无参构造方法来初始化 defaultConstructor 字段。这部分逻辑在 addDefaultConstructor() 方法中实现。
3. 读取 Class 类中的 getter方法，填充上面介绍的 getMethods 集合和 getTypes 集合。这部分逻辑在 addGetMethods() 方法中实现。
4. 读取 Class 类中的 setter 方法，填充上面介绍的 setMethods 集合和 setTypes 集合。这部分逻辑在 addSetMethods() 方法中实现。
5. 读取 Class 中没有 getter/setter 方法的字段，生成对应的 Invoker 对象，填充 getMethods 集合、getTypes 集合以及 setMethods 集合、setTypes 集合。这部分逻辑在 addFields() 方法中实现。
6. 根据前面三步构造的 getMethods/setMethods 集合的 keySet，初始化 readablePropertyNames、writablePropertyNames 集合。
7. 遍历构造的 readablePropertyNames、writablePropertyNames 集合，将其中的属性名称全部转化成大写并记录到 caseInsensitivePropertyMap 集合中。

2. 核心方法解析

了解了初始化的核心流程之后，我们再继续深入分析其中涉及的方法，这些方法也是 Reflector 的核心方法。

首先来看 addGetMethods() 方法和 addSetMethods() 方法，它们分别用来解析传入 Class 类中的 getter方法和 setter() 方法，两者的逻辑十分相似。这里，我们就以

addGetMethods() 方法为例深入分析，其主要包括如下三个核心步骤。

第一步，获取方法信息。 这里会调用 getClassMethods() 方法获取当前 Class 类的所有方法的唯一签名（注意一下，这里同时包含继承自父类以及接口的方法），以及每个方法对应的 Method 对象。

在递归扫描父类以及父接口的过程中，会使用 Map<String, Method> 集合记录遍历到的方法，实现去重的效果，其中 Key 是对应的方法签名，Value 为方法对应的 Method 对象。生成的方法签名的格式如下：

```
返回值类型#方法名称:参数类型列表
```

例如，addGetMethods(Class) 方法的唯一签名是：

```
java.lang.String#addGetMethods:java.lang.Class
```

可见，**这里生成的方法签名是包含返回值的，可以作为该方法全局唯一的标识。**

第二步，按照 Java 的规范，从上一步返回的 Method 数组中查找 getter 方法，将其记录到 conflictingGetters 集合中。这里的 conflictingGetters 集合（HashMap<String, List>() 类型）中的 Key 为属性名称，Value 是该属性对应的 getter 方法集合。

为什么一个属性会查找到多个 getter 方法呢？这主要是由于类间继承导致的，在子类中我们可以覆盖父类的方法，覆盖不仅可以修改方法的具体实现，还可以修改方法的返回值，getter 方法也不例外，这就导致在第一步中产生了两个签名不同的方法。

第三步，解决方法签名冲突。 这里会调用 resolveGetterConflicts() 方法对这种 getter 方法的冲突进行处理，**处理冲突的核心逻辑其实就是比较 getter 方法的返回值，优先选择返回值为子类的 getter 方法**，例如：

```
// 该方法定义在SuperClazz类中
public List getA();
// 该方法定义在SubClazz类中，SubClazz继承了SuperClazz类
public ArrayList getA();
```

可以看到，SubClazz.getA() 方法的返回值 ArrayList 是其父类 SuperClazz 中 getA() 方法返回值 List 的子类，所以这里选择 SubClazz 中定义的 getA() 方法作为 A 这个属性的 getter 方法。

在 resolveGetterConflicts() 方法处理完上述 getter 方法冲突之后，会为每个 getter 方法创建对应的 MethodInvoker 对象，然后统一保存到 getMethods 集合中。同时，还会在 getTypes 集合中维护属性名称与对应 getter 方法返回值类型的映射。

到这里了，addGetMethods() 的核心逻辑就分析清楚了。

我们接下来回到 Reflector 的构造方法中，在通过 addGetMethods() 和 addSetMethods() 方法，完成 Class 类中 getter/setter 方法的处理之后，会继续调用 addFields() 方法处理没有 getter/setter 方法的字段。

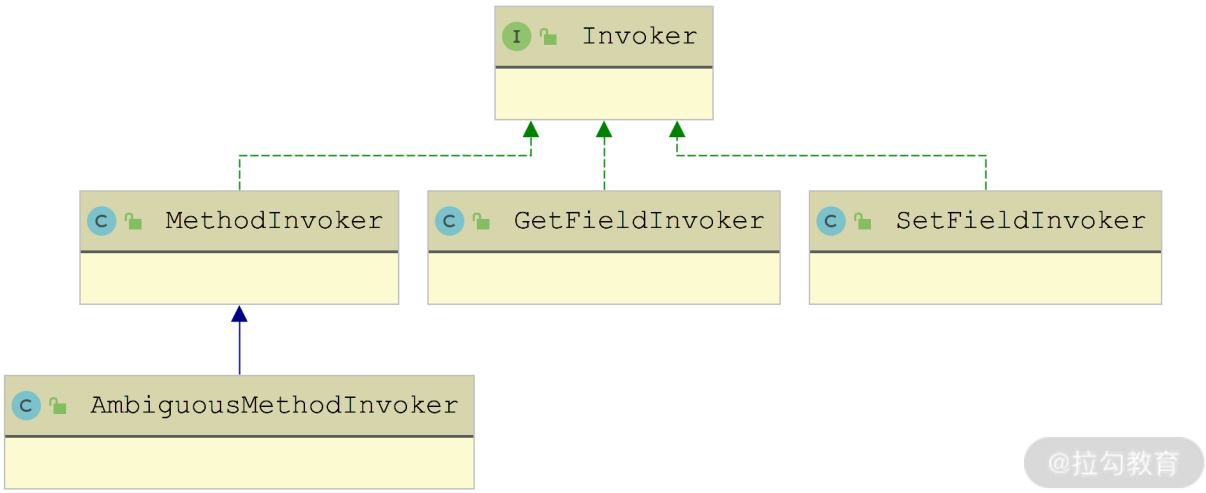
这里我们以处理没有 getter 方法的字段为例，addFields() 方法会为这些字段生成对应的 GetFieldInvoker 对象并记录到 getMethods 集合中，同时也会将属性名称和属性类型记录到 getTypes 集合中。处理没有 setter 方法的字段也是相同的逻辑。

3. Invoker

在 Reflector 对象的初始化过程中，所有属性的 getter/setter 方法都会被封装成 MethodInvoker 对象，没有 getter/setter 的字段也会生成对应的 Get/SetFieldInvoker 对象。下面我们就来看看这个 Invoker 接口的定义：

```
public interface Invoker {

    // 调用底层封装的Method方法或是读写指定的字段

    Object invoke(Object target, Object[] args);

    Class<?> getType(); // 返回属性的类型

}
```

Invoker 接口的继承关系如下图所示：



Invoker 接口继承关系图

其中，MethodInvoker 是通过反射方式执行底层封装的 Method 方法（例如，getter/setter 方法）完成属性读写效果的，Get/SetFieldInvoker 是通过反射方式读写底层封装的 Field 字段，进而实现属性读写效果的。

4. ReflectorFactory

通过上面的分析我们知道，Reflector 初始化过程会有一系列的反射操作，**为了提升 Reflector 的初始化速度，MyBatis 提供了 ReflectorFactory 这个工厂接口对 Reflector 对象进行缓存**，其中最核心的方法是用来获取 Reflector 对象的 findForClass() 方法。

DefaultReflectorFactory 是 ReflectorFactory 接口的默认实现，它默认会在内存中维护一个 ConcurrentHashMap<Class<?>, Reflector> 集合（reflectorMap 字段）缓存其创建的所有 Reflector 对象。

在其 findForClass() 方法实现中，首先会根据传入的 Class 类查询 reflectorMap 缓存，如果查找到对应的 Reflector 对象，则直接返回；否则创建相应的 Reflector 对象，并记录到 reflectorMap 中缓存，等待下次使用。

默认对象工厂

ObjectFactory 是 MyBatis 中的反射工厂，其中提供了两个 create() 方法的重载，我们可以通过两个 create() 方法创建指定类型的对象。

DefaultObjectFactory 是 ObjectFactory 接口的默认实现，其 create() 方法底层是通过调用 instantiateClass() 方法创建对象的。instantiateClass() 方法会通过反射的方式根据传入的参数列表，选择合适的构造函数实例化对象。

除了使用 DefaultObjectFactory 这个默认实现之外，我们还可以在 mybatis-config.xml 配置文件中配置自定义 ObjectFactory 接口扩展实现 类（在 MyBatis 提供的测试类中，就包含了自定义的 ObjectFactory 实现，可以参考我们的源码），完成自定义的功能扩展。

属性解析工具

在前面《02 | 订单系统持久层示例分析，20 分钟带你快速上手 MyBatis》介绍的订单系统示例中，我们在 orderMap 这个 ResultMap 映射中，如果要配置 Order 与 OrderItem 的一对多关系，可以使用 `<collection>` 标签进行配置；如果 OrderItem 个数明确，可以直接使用数组下标索引方式（即 ordersItems[0]）填充 orderItems 集合。

这里的 “.” 导航以及数组下标的解析，也都是在反射工具箱中完成的。下面我们就来介绍 reflection.property 包下的**三个属性解析相关的工具类**，在后面的 MetaClass、MetaObject 等工具类中，也都需要属性解析能力。

- PropertyTokenizer 工具类负责解析由“.”和“[]”构成的表达式。PropertyTokenizer 继承了 Iterator 接口，可以迭代处理嵌套多层表达式。
- PropertyCopier 是一个属性拷贝的工具类，提供了与 Spring 中 BeanUtils.copyProperties() 类似的功能，实现相同类型的两个对象之间的属性值拷贝，其核心方法是 copyBeanProperties() 方法。
- PropertyNamer 工具类提供的功能是转换方法名到属性名，以及检测一个方法名是否为 getter 或 setter 方法。

MetaClass

MetaClass 提供了获取类中属性描述信息的功能，底层依赖前面介绍的 Reflector，在 MetaClass 的构造方法中会将传入的 Class 封装成一个 Reflector 对象，并记录到 reflector 字段中，MetaClass 的后续属性查找都会使用到该 Reflector 对象。

MetaClass 中的 findProperty() 方法是实现属性查找的核心方法，它主要处理了“.”导航的属性查找，该方法会用前文介绍的 PropertyTokenizer 解析传入的 name 表达式，该表达式可能通过“.”导航多层，例如，order.deliveryAddress.customer.name。

MetaClass 会逐层处理这个表达式，首先通过 Order 类型对应的 Reflector 查找 deliveryAddress 属性，查找成功之后，根据 deliveryAddress 属性的类型（即 Address 类型）创建对应的 MetaClass 对象（以及底层的 Reflector 对象），再继续查找其中的 customer 属性，如此递归处理，直至最后查找到 Customer 中的 name 属性。这部分递归查找逻辑位于 MetaClass.buildProperty() 方法中。

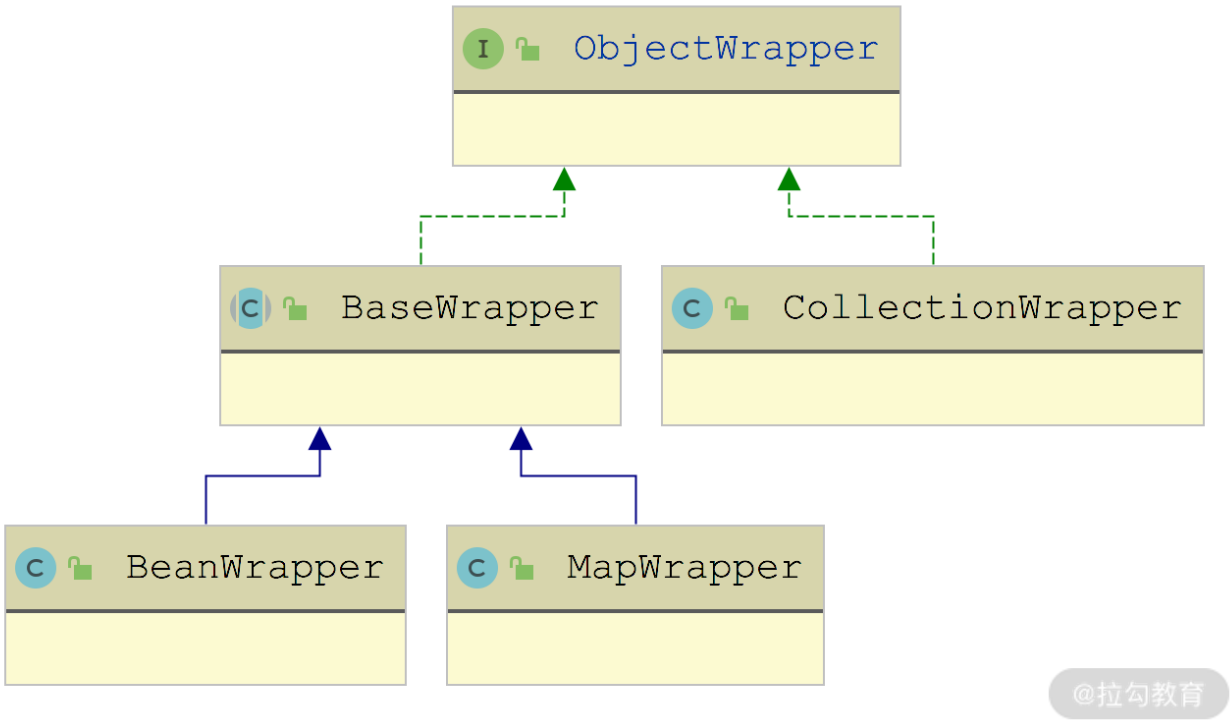
在上述 MetaClass 查找属性的过程中，还会调用 hasGetter() 和 hasSetter() 方法负责判断属性表达式中指定的属性是否有对应的 getter/setter 方法。这两个方法也是先通过 PropertyTokenizer 解析传入的 name 表达式，然后进行递归查询，在递归查询中会依赖 Reflector.hasGetter() 方法查找前文介绍的 getMethods 集合或 setMethods 集合，查找属性对应的 getter/setter 方法。

MetaClass 中的其他方法实现也都**大多是依赖 PropertyTokenizer 解析表达式，然后递归查找，查找过程会依赖 Reflector 的相关方法。**

ObjectWrapper

MetaClass 中封装的是 Class 元信息，ObjectWrapper 封装的则是对象元信息。在 ObjectWrapper 中抽象了一个对象的属性信息，并提供了查询对象属性信息的相关方法，以及更新属性值的相关方法。

ObjectWrapper 的实现类如下图所示：



ObjectWrapper 继承关系图

BaseWrapper 是 ObjectWrapper 接口的抽象实现，其中只有一个 MetaObject 类型的字段。BaseWrapper 为子类实现了 resolveCollection()、getCollectionValue() 和 setCollectionValue() 三个针对集合对象的处理方法。其中，resolveCollection() 方法会将指定属性作为集合对象返回，底层依赖 MetaObject.getValue()方法实现（后面还会详细介绍）。getCollectionValue() 方法和 setCollectionValue() 方法会解析属性表达式的下标信息，然后获取/设置集合中的对应元素，这里解析属性表达式依然是依赖前面介绍的 PropertyTokenizer 工具类。

BeanWrapper 继承了 BaseWrapper 抽象类，底层除了封装了一个 JavaBean 对象之外，还封装了该 JavaBean 类型对应的 MetaClass 对象，以及从 BaseWrapper 继承下来的 MetaObject 对象。

在 get() 方法和 set() 方法实现中，BeanWrapper 会根据传入的属性表达式，获取/设置相应的属性值。以 get() 方法为例，首先会判断表达式中是否含有数组下标，如果含有下标，会通过 resolveCollection() 和 getCollectionValue() 方法从集合中获取相应元素；如果不包含下标，则通过 MetaClass 查找属性名称在 Reflector.getMethods 集合中相应的

GetFieldInvoker，然后调用 Invoker.invoke() 方法读取属性值。

BeanWrapper 中其他方法的实现也大都与 get() 方法和 set() 方法类似，依赖 MetaClass、MetaObject 完成相关对象中属性信息读写，这里就不再一一介绍，你若感兴趣的话可以参考源码进行学习。

CollectionWrapper 是 ObjectWrapper 接口针对 Collection 集合的一个实现，其中封装了Collection `<Object>` 集合对象，只有 isCollection()、add()、addAll() 方法以及从 BaseWrapper 继承下来的方法是可用的，其他方法都会抛出 UnsupportedOperationException 异常。

MapWrapper 是针对 Map 类型的一个实现，这个实现就比较简单了，所以我就留给你自己去分析了，分析过程中可以参考下面将要介绍的 MetaObject。

MetaObject

通过对 ObjectWrapper 的介绍我们了解到，ObjectWrapper 实现了读写对象属性值、检测 getter/setter 等基础功能，在分析 BeanWrapper 等实现类时，我们可以看到其**底层会依赖 MetaObject**。在 MetaObject 中维护了一个 originalObject 字段指向被封装的 JavaBean 对象，还维护了该 JavaBean 对象对应的 ObjectWrapper 对象（objectWrapper 字段）。

MetaObject 和 ObjectWrapper 中关于类级别的方法，例如，hasGetter() 方法、hasSetter() 方法、findProperty() 方法等，都是直接调用 MetaClass 或 ObjectWrapper 的对应方法实现的。其他关于对象级别的方法，都是与 ObjectWrapper 配合实现，例如 MetaObject.getValue()/setValue() 方法等。

这里以 getValue() 方法为例，该方法首先根据 PropertyTokenizer 解析指定的属性表达式，如果该表达式是包含“.”导航的多级属性查询，则获取子表达式并为其对应的属性对象创建关联的 MetaObject 对象，继续递归调用 getValue() 方法，直至递归处理结束，递归出口会调用 ObjectWrapper.get() 方法获取最终的属性值。

在 MetaObject 中，setValue() 方法的核心逻辑与 getValue() 方法基本类似，也是递归查找。但是，其中有一个不同之处需要你注意：如果需要设置的最终属性值不为空时，在递归查找 setter() 方法的过程中会调用 ObjectWrapper.instantiatePropertyValue() 方法初始化递归过程中碰到的任意空对象，但如果碰到为空的集合元素，则无法通过该方法初始化。ObjectWrapper.instantiatePropertyValue() 方法实际上是依赖 ObjectFactory 接口的 create()方法（默认实现是 DefaultObjectFactory ）创建相应类型的对象。

了解了 MetaObject 和 BeanWrapper 配合使用的方式以及递归查找属性表达式指定的属性值的逻辑之后，MetaObject 剩余方法的实现就比较好分析了，这里我也就不再赘述了。

总结

这一讲我们重点介绍了 MyBatis 中的反射工具箱。首先，我们介绍了反射工具箱中最核心、最底层的 Reflector 类的核心实现；接下来介绍了反射工具箱在 Reflector 基础之上提供的各种工具类，其中包括 ObjectFactory 工厂类、ObjectWrapper 包装类以及记录元数据的 MetaClass、MetaObject 等。它们彼此联系紧密，希望你在学习过程中能将它们的各个知识点串联起来，灵活运用。

前面我们也说了，MapWrapper 是针对 Map 类型的一个实现，这个实现比较简单了，你可以试着去分析下。欢迎你在留言区分享你的分析过程。

上一页 下一页