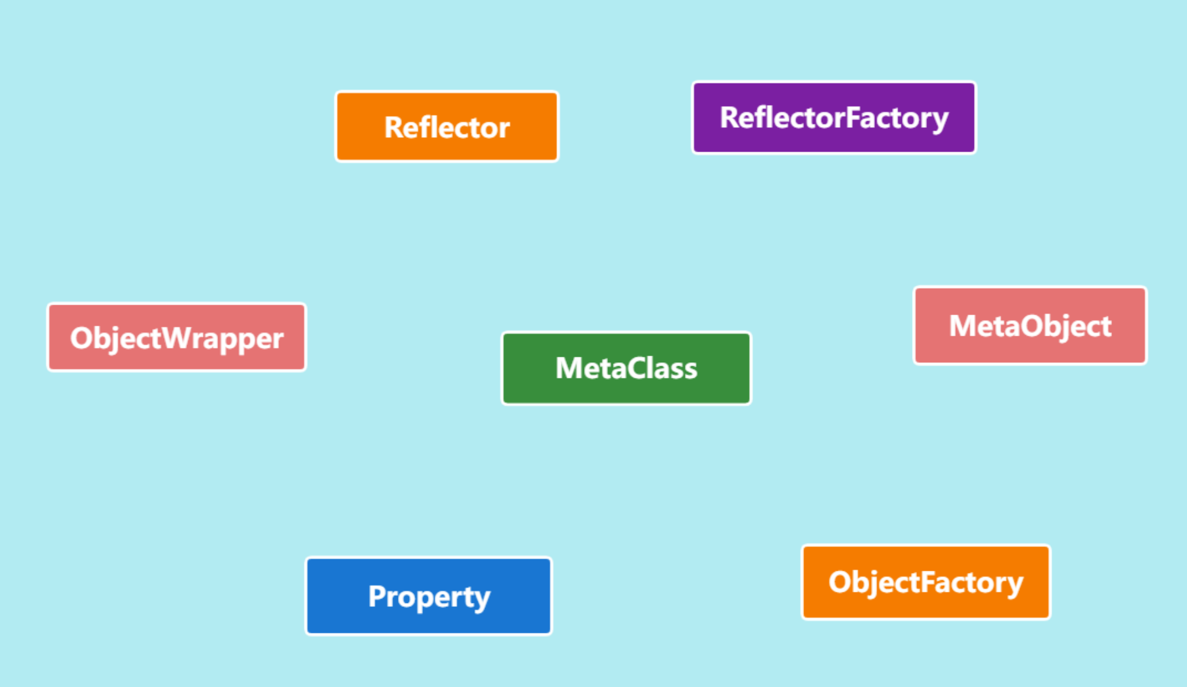
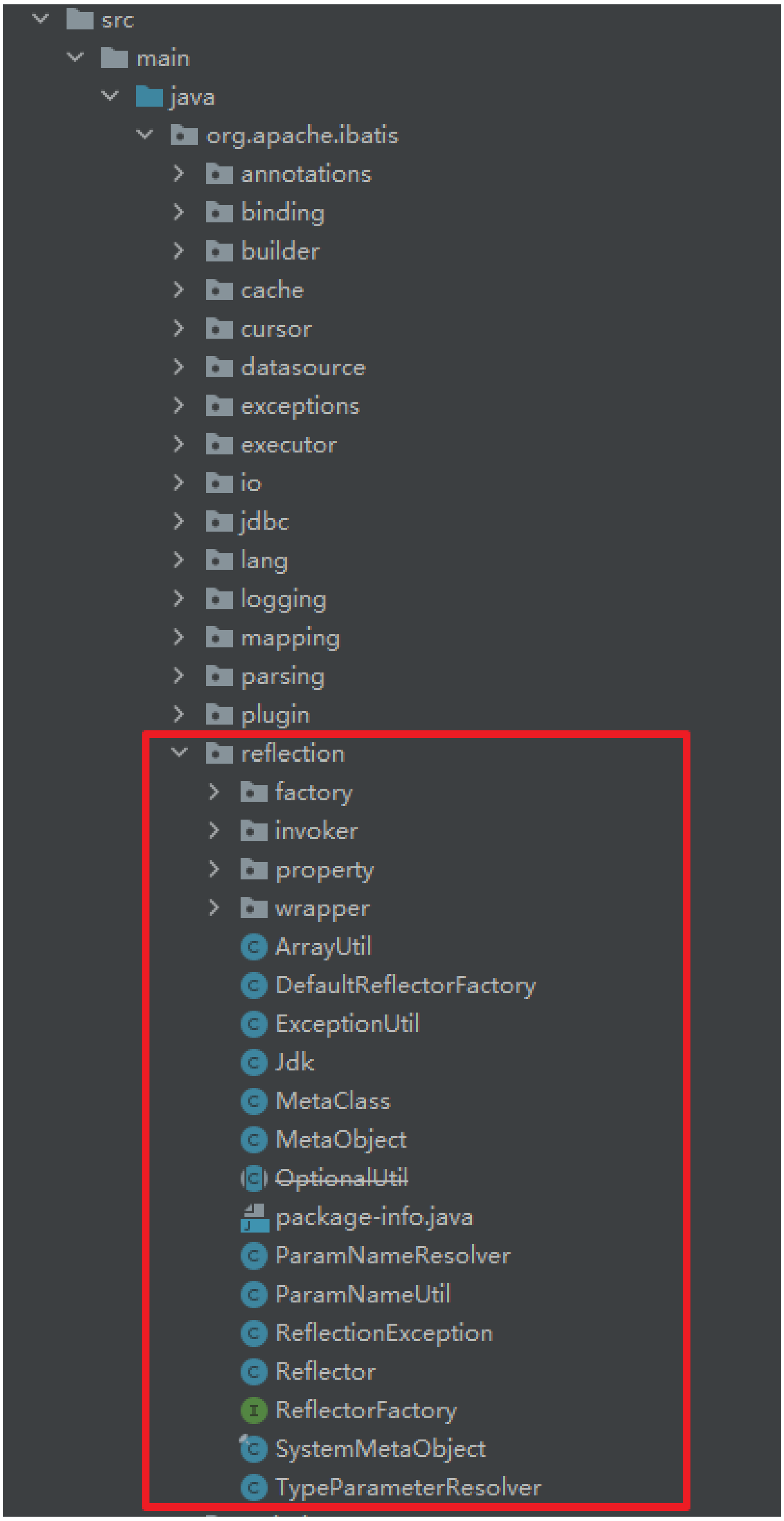
MyBatis基础模块-反射模块



https://www.processon.com/view/link/62ea3de3637689072efdd258

1.反射模块

MyBatis在进行参数处理、结果集映射等操作时会使用到大量的反射操作，Java中的反射功能虽然强大，但是代码编写起来比较复杂且容易出错，为了简化反射操作的相关代码，MyBatis提供了专门的反射模块，该模块位于org.apache.ibatis.reflection包下，它对常见的反射操作做了进一步的封装，提供了更加简洁方便的反射API。



- src
  - main
    - java
      - org.apache.ibatis
        - annotations
        - binding
        - builder
        - cache
        - cursor
        - datasource
        - exceptions
        - executor
        - io
        - jdbc
        - lang
        - logging
        - mapping
        - parsing
        - plugin
        - reflection
          - factory
          - invoker
          - property
          - wrapper
          - ArrayUtil
          - DefaultReflectorFactory
          - ExceptionUtil
          - Jdk
          - MetaClass
          - MetaObject
          - ~~OptionalUtil~~
          - package-info.java
          - ParamNameResolver
          - ParamNameUtil
          - ReflectionException
          - Reflector
          - ReflectorFactory
          - SystemMetaObject
          - TypeParameterResolver

1.1 Reflector

Reflector是反射模块的基础，每个Reflector对象都对应一个类，在Reflector中缓存了反射需要使用的类的元信息

1.1.1 属性

首先来看下Reflector中提供的相关属性的含义

```
// 对应的Class 类型
private final Class<?> type;
// 可读属性的名称集合 可读属性就是存在 getter方法的属性，初始值为null
private final String[] readablePropertyNames;
```

```
  // 可写属性的名称集合 可写属性就是存在 setter方法的属性，初始值为null
  private final String[] writablePropertyNames;
  // 记录了属性相应的setter方法，key是属性名称，value是Invoker方法
  // 他是对setter方法对应Method对象的封装
  private final Map<String, Invoker> setMethods = new HashMap<>();
  // 属性相应的getter方法
  private final Map<String, Invoker> getMethods = new HashMap<>();
  // 记录了相应setter方法的参数类型，key是属性名称 value是setter方法的参数类型
  private final Map<String, Class<?>> setTypes = new HashMap<>();
  // 和上面的对应
  private final Map<String, Class<?>> getTypes = new HashMap<>();
  // 记录了默认的构造方法
  private Constructor<?> defaultConstructor;

  // 记录了所有属性名称的集合
  private Map<String, String> caseInsensitivePropertyMap = new HashMap<>();
```

1.1.2 构造方法

在Reflector的构造器中会完成相关的属性的初始化操作

```
  // 解析指定的Class类型 并填充上述的集合信息
  public Reflector(Class<?> clazz) {
    type = clazz; // 初始化 type字段
    addDefaultConstructor(clazz);// 设置默认的构造方法
    addGetMethods(clazz);// 获取getter方法
    addSetMethods(clazz); // 获取setter方法
    addFields(clazz); // 处理没有getter/setter方法的字段
    // 初始化 可读属性名称集合
    readablePropertyNames = getMethods.keySet().toArray(new String[0]);
    // 初始化 可写属性名称集合
    writablePropertyNames = setMethods.keySet().toArray(new String[0]);
    // caseInsensitivePropertyMap记录了所有的可读和可写属性的名称 也就是记录了所有的属性名
称
    for (String propName : readablePropertyNames) {
      // 属性名称转大写
      caseInsensitivePropertyMap.put(propName.toUpperCase(Locale.ENGLISH),
propName);
    }
    for (String propName : writablePropertyNames) {
      // 属性名称转大写
      caseInsensitivePropertyMap.put(propName.toUpperCase(Locale.ENGLISH),
propName);
    }
  }
```

反射我们也可以在项目中我们直接拿来使用,定义一个普通的Bean对象。

```
/**
 * 反射工具箱
 *    测试用例
 */
public class Person {

    private Integer id;

    private String name;

    public Person(Integer id) {
        this.id = id;
    }

    public Person(Integer id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

1.1.3 公共的API方法

然后我们可以看看Reflector中提供的公共的API方法

方法名称	作用
getType	获取Reflector表示的Class
getDefaultConstructor	获取默认的构造器
hasDefaultConstructor	判断是否有默认的构造器
getSetInvoker	根据属性名称获取对应的Invoker 对象
getGetInvoker	根据属性名称获取对应的Invoker对象
getSetterType	获取属性对应的类型 比如： String name; // getSetterType("name") --> java.lang.String
getGetterType	与上面是对应的
getGetablePropertyNames	获取所有的可读属性名称的集合
getSetablePropertyNames	获取所有的可写属性名称的集合
hasSetter	判断是否具有某个可写的属性
hasGetter	判断是否具有某个可读的属性
findPropertyName	根据名称查找属性

了解了Reflector对象的基本信息后我们需要如何来获取Reflector对象呢？在MyBatis中给我们提供了一个ReflectorFactory工厂对象。所以我们先来简单了解下ReflectorFactory对象,当然你也可以直接new 出来，像上面的案例一样，

1.2 ReflectorFactory

ReflectorFactory接口主要实现了对Reflector对象的创建和**缓存**。

1.2.1 ReflectorFactory接口的定义

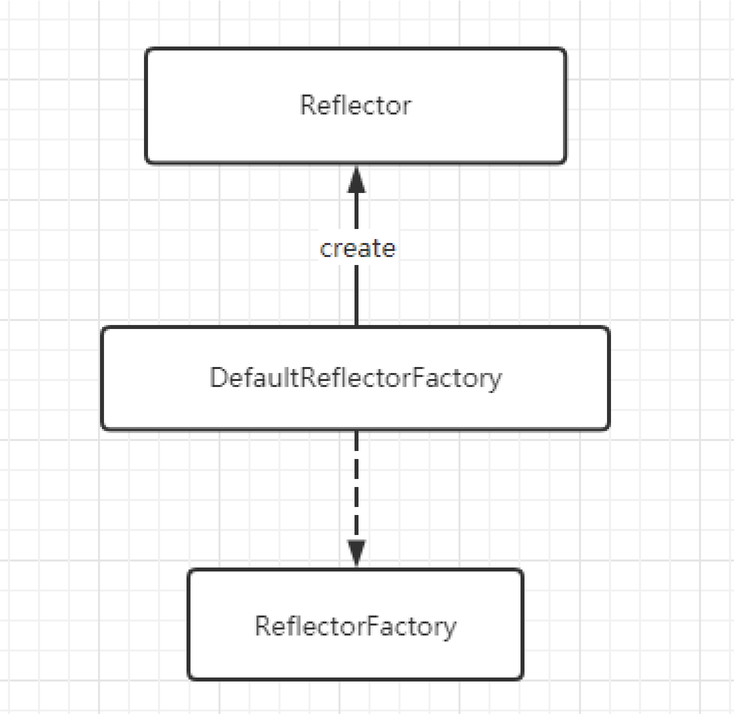
接口的定义如下

```
public interface ReflectorFactory {
  // 检测该ReflectorFactory是否缓存了Reflector对象
  boolean isClassCacheEnabled();
  // 设置是否缓存Reflector对象
  void setClassCacheEnabled(boolean classCacheEnabled);
  // 创建指定了Class的Reflector对象
  Reflector findForClass(Class<?> type);
}
```

然后我们来看看它的具体实现

1.2.2 DefaultReflectorFactory

MyBatis只为该接口提供了DefaultReflectorFactory这一个实现类。他与Reflector的关系如下：



DefaultReflectorFactory中的实现，代码比较简单，我们直接贴出来

```
public class DefaultReflectorFactory implements ReflectorFactory {
  private boolean classCacheEnabled = true;
  // 实现对 Reflector 对象的缓存
  private final ConcurrentMap<Class<?>, Reflector> reflectorMap = new
ConcurrentHashMap<>();

  public DefaultReflectorFactory() {
  }

  @Override
  public boolean isClassCacheEnabled() {
    return classCacheEnabled;
  }

  @Override
  public void setClassCacheEnabled(boolean classCacheEnabled) {
    this.classCacheEnabled = classCacheEnabled;
  }

  @Override
  public Reflector findForClass(Class<?> type) {
    if (classCacheEnabled) {// 开启缓存
      // synchronized (type) removed see issue #461
      return reflectorMap.computeIfAbsent(type, Reflector::new);
    } else {
      // 没有开启缓存就直接创建
      return new Reflector(type);
    }
  }
}
```

1.2.3 使用演示

通过上面的介绍，我们可以具体的来使用下，加深对其的理解,先准备一个JavaBean，

```
package com.boge.domain;

public class Student {

    public Integer getId() {
        return 6;
    }

    public void setId(Integer id) {
        System.out.println(id);
    }

    public String getUserName() {
        return "张三";
    }
}
```

这个Bean我们做了简单的处理

```
    @Test
    public void test02() throws Exception{
        ReflectorFactory factory = new DefaultReflectorFactory();
        Reflector reflector = factory.findForClass(Student.class);
        System.out.println("可读属
性:"+Arrays.toString(reflector.getGetablePropertyNames()));
        System.out.println("可写属
性:"+Arrays.toString(reflector.getSetablePropertyNames()));
        System.out.println("是否具有默认的构造器:" +
reflector.hasDefaultConstructor());
        System.out.println("Reflector对应的Class:" + reflector.getType());
    }
```

1.3 Invoker

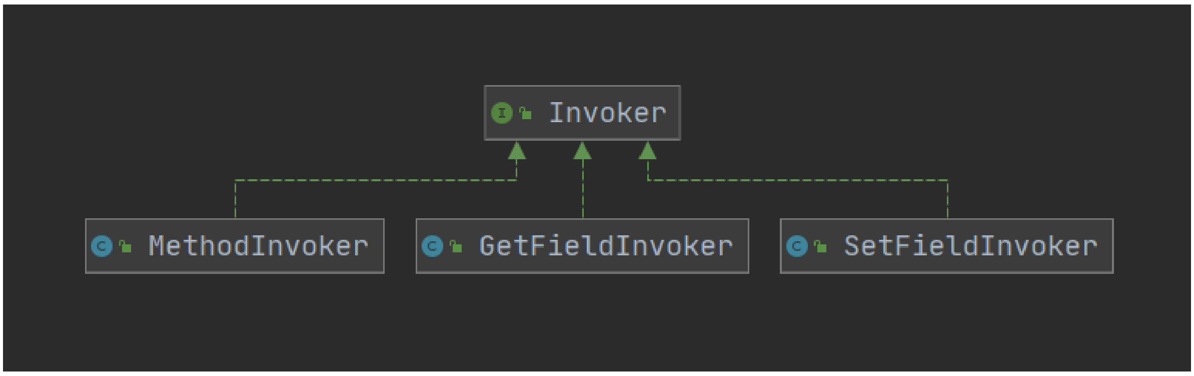
针对于Class中Field和Method的调用，在MyBatis中封装了Invoker对象来统一处理(有使用到适配器模式)

1.3.1 接口说明

```
/**
 * @author Clinton Begin
 */
public interface Invoker {
  // 执行Field或者Method
  Object invoke(Object target, Object[] args) throws IllegalAccessException,
InvocationTargetException;

  // 返回属性相应的类型
  Class<?> getType();
}
```

该接口有对应的三个实现



1.3.2 效果演示

使用效果演示，还是通过上面的案例来介绍

```
package com.boge.domain;

public class Student {


    public Integer getId() {
        System.out.println("读取id");
        return 6;
    }

    public void setId(Integer id) {
        System.out.println("写入id:"+id);
    }

    public String getUserName() {

        return "张三";
    }
}
```

测试代码

```
    public void test03() throws Exception{
        ReflectorFactory factory = new DefaultReflectorFactory();
        Reflector reflector = factory.findForClass(Student.class);
        // 获取构造器 生成对应的对象
        Object o = reflector.getDefaultConstructor().newInstance();
        MethodInvoker invoker1 = (MethodInvoker) reflector.getSetInvoker("id");
        invoker1.invoke(o,new Object[]{999});
        // 读取
        Invoker invoker2 = reflector.getGetInvoker("id");
        invoker2.invoke(o,null);
    }
```

1.4 MetaClass

在Reflector中可以针对普通的属性操作，但是如果出现了比较复杂的属性，比如 private Person person; 这种，我们要查找的表达式 person.userName.针对这种表达式的处理，这时就可以通过MetaClass来处理了。我们来看看主要的属性和构造方法

```
/**
 * 通过 Reflector 和 ReflectorFactory 的组合使用 实现对复杂的属性表达式的解析
 * @author Clinton Begin
 */
public class MetaClass {
  // 缓存 Reflector
  private final ReflectorFactory reflectorFactory;
  // 创建 MetaClass时 会指定一个Class reflector会记录该类的相关信息
  private final Reflector reflector;

  private MetaClass(Class<?> type, ReflectorFactory reflectorFactory) {
    this.reflectorFactory = reflectorFactory;
    this.reflector = reflectorFactory.findForClass(type);
  }
  // ....
}
```

效果演示，准备Bean对象

```
package com.boge.domain;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class RichType {

    private RichType richType;

    private String richField;

    private String richProperty;

    private Map richMap = new HashMap();

    private List richList = new ArrayList() {
        {
            add("bar");
        }
    };

    public RichType getRichType() {
        return richType;
    }

    public void setRichType(RichType richType) {
        this.richType = richType;
    }

    public String getRichProperty() {
        return richProperty;
    }

    public void setRichProperty(String richProperty) {
        this.richProperty = richProperty;
    }

    public List getRichList() {
        return richList;
    }

    public void setRichList(List richList) {
        this.richList = richList;
    }

    public Map getRichMap() {
        return richMap;
    }

    public void setRichMap(Map richMap) {
        this.richMap = richMap;
    }
}
```

测试代码

```
    @Test
    public void test7(){
        ReflectorFactory reflectorFactory = new DefaultReflectorFactory();
        MetaClass meta = MetaClass.forClass(RichType.class, reflectorFactory);
        System.out.println(meta.hasGetter("richField"));
        System.out.println(meta.hasGetter("richProperty"));
        System.out.println(meta.hasGetter("richList"));
        System.out.println(meta.hasGetter("richMap"));
        System.out.println(meta.hasGetter("richList[0]"));

        System.out.println(meta.hasGetter("richType"));
        System.out.println(meta.hasGetter("richType.richField"));
        System.out.println(meta.hasGetter("richType.richProperty"));
        System.out.println(meta.hasGetter("richType.richList"));
        System.out.println(meta.hasGetter("richType.richMap"));
        System.out.println(meta.hasGetter("richType.richList[0]"));
        // findProperty 只能处理 . 的表达式
        System.out.println(meta.findProperty("richType.richProperty"));
        System.out.println(meta.findProperty("richType.richProperty1"));
        System.out.println(meta.findProperty("richList[0]"));

        System.out.println(Arrays.toString(meta.getGetterNames()));
    }
```

输出结果

```
true
true
true
true
true
true
true
true
true
true
true
richType.richProperty
richType.
null
[richType, richProperty, richMap, richList, richField]
```

1.5 MetaObject

我们可以通过MetaObject对象解析复杂的表达式来对提供的对象进行操作。具体的通过案例来演示会更直观些

```
    @Test
    public void shouldGetAndSetField() {
        RichType rich = new RichType();
        MetaObject meta = SystemMetaObject.forObject(rich);
        meta.setValue("richField", "foo");
        System.out.println(meta.getValue("richField"));
    }

    @Test
    public void shouldGetAndSetNestedField() {
        RichType rich = new RichType();
        MetaObject meta = SystemMetaObject.forObject(rich);
        meta.setValue("richType.richField", "foo");
        System.out.println(meta.getValue("richType.richField"));
    }

    @Test
    public void shouldGetAndSetMapPairUsingArraySyntax() {
        RichType rich = new RichType();
        MetaObject meta = SystemMetaObject.forObject(rich);
        meta.setValue("richMap[key]", "foo");
        System.out.println(meta.getValue("richMap[key]"));
    }
```

以上三个方法的输出结果都是

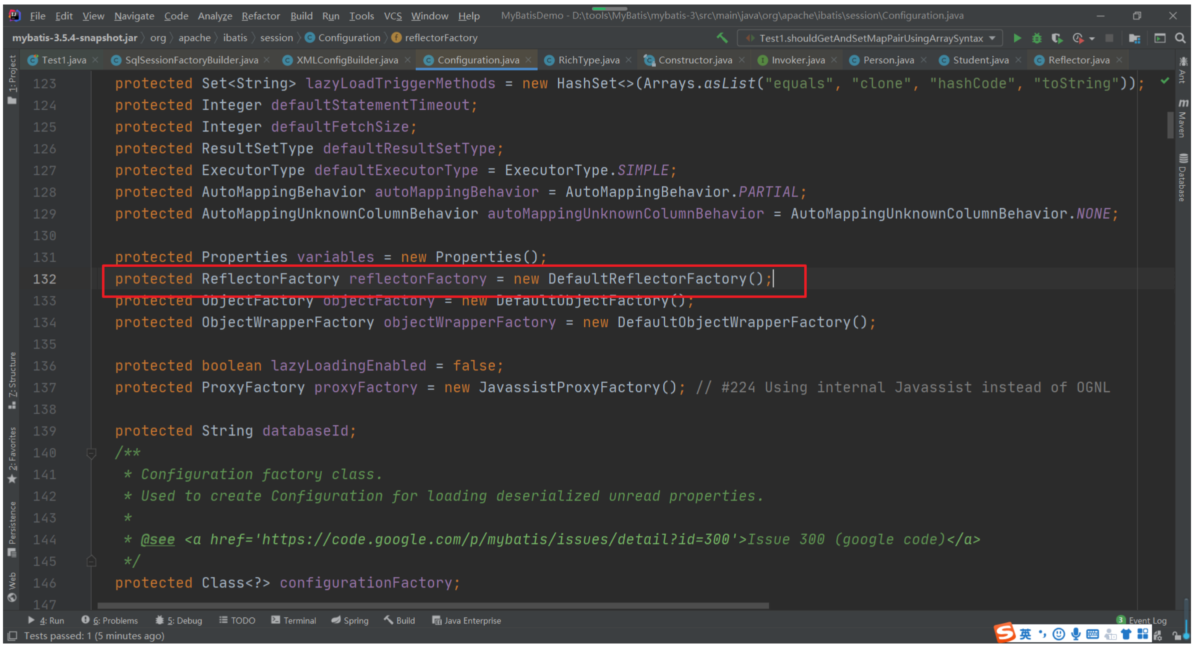
```
foo
```

1.6 反射模块应用

然后我们来看下在MyBatis的核心处理层中的实际应用

1.6.1 SqlSessionFactory

在创建SqlSessionFactory操作的时候会完成Configuration对象的创建，而在Configuration中默认定义的ReflectorFactory的实现就是DefaultReflectorFactory对象


File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
MyBatisDemo - D:\tools\MyBatis\mybatis-3\src\main\java\org\apache\ibatis\session\Configuration.java
mybatis-3.5.4-snapshot.jar 〉 org 〉 apache 〉 ibatis 〉 session 〉 Configuration 〉 reflectorFactory
Test1.shouldGetAndSetMapPairUsingArraySyntax
Test1.java
SqlSessionFactoryBuilder.java
XMLConfigBuilder.java
Configuration.java
RichType.java
Constructor.java
Invoker.java
Person.java
Student.java
Reflector.java
1: Project
Z: Structure
2: Favorites
Persistence
Web
Ant
Maven
Database
4: Run
6: Problems
5: Debug
TODO
Terminal
Spring
Build
Java Enterprise
Event Log

```
protected Set<String> lazyLoadTriggerMethods = new HashSet<>(Arrays.asList("equals", "clone", "hashCode", "toString"));
protected Integer defaultStatementTimeout;
protected Integer defaultFetchSize;
protected ResultSetType defaultResultSetType;
protected ExecutorType defaultExecutorType = ExecutorType.SIMPLE;
protected AutoMappingBehavior autoMappingBehavior = AutoMappingBehavior.PARTIAL;
protected AutoMappingUnknownColumnBehavior autoMappingUnknownColumnBehavior = AutoMappingUnknownColumnBehavior.NONE;

protected Properties variables = new Properties();
protected ReflectorFactory reflectorFactory = new DefaultReflectorFactory();
protected ObjectFactory objectFactory = new DefaultObjectFactory();
protected ObjectWrapperFactory objectWrapperFactory = new DefaultObjectWrapperFactory();

protected boolean lazyLoadingEnabled = false;
protected ProxyFactory proxyFactory = new JavassistProxyFactory(); // #224 Using internal Javassist instead of OGNL

protected String databaseId;
/**
 * Configuration factory class.
 * Used to create Configuration for loading deserialized unread properties.
 *
 * @see <a href='https://code.google.com/p/mybatis/issues/detail?id=300'>Issue 300 (google code)</a>
 */
protected Class<?> configurationFactory;

```

Tests passed: 1 (5 minutes ago)

然后在解析全局配置文件的代码中，给用户提供了ReflectorFactory的扩展，也就是我们在全局配置文件中可以通过

reflectorFactory标签来使用我们自定义的ReflectorFactory

1.6.2 SqlSession

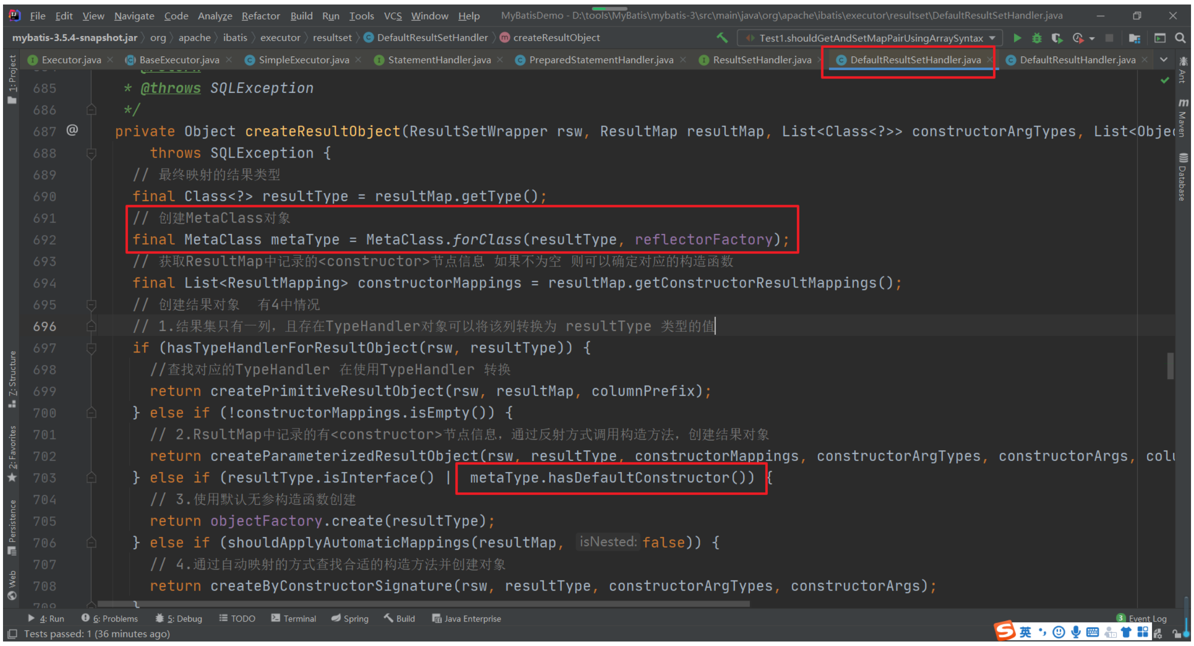
无相关操作

1.6.3 Mapper

无相关操作

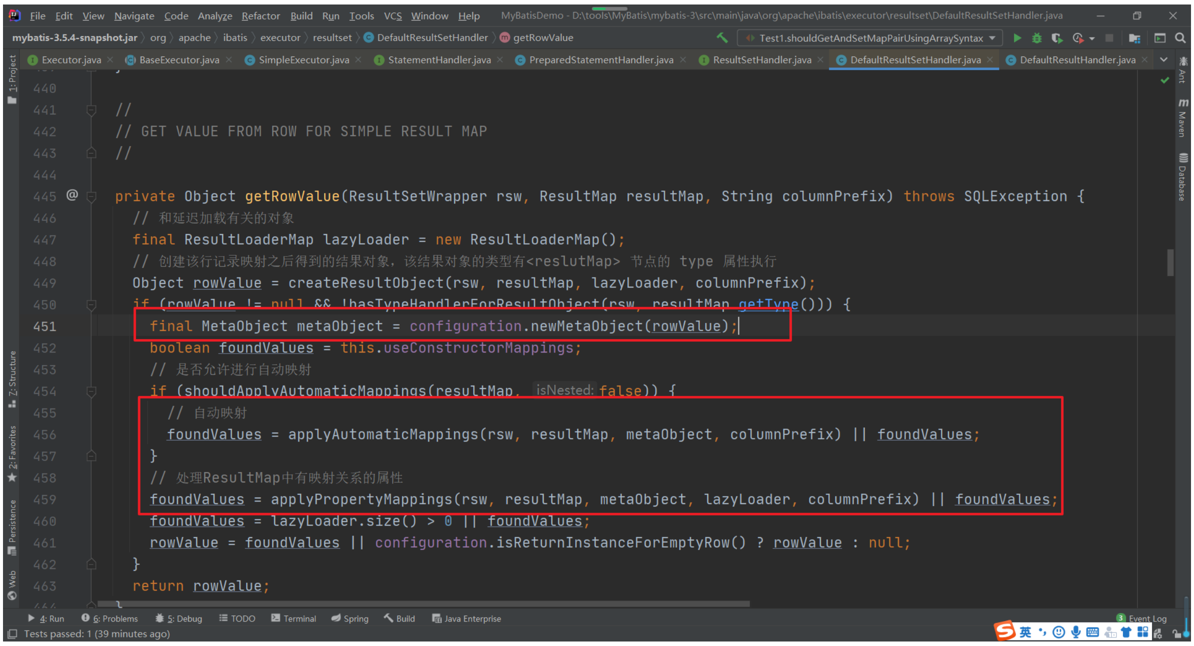
1.6.4 执行SQL

在Statement获取结果集后，在做结果集映射的使用有使用到，在DefaultResultSetHandler的createResultObject方法中。



```
    * @throws SQLException
    */
  private Object createResultObject(ResultSetWrapper rsw, ResultMap resultMap, List<Class<?>> constructorArgTypes, List<Obje
      throws SQLException {
    // 最终映射的结果类型
    final Class<?> resultType = resultMap.getType();
    // 创建MetaClass对象
    final MetaClass metaType = MetaClass.forClass(resultType, reflectorFactory);
    // 获取ResultMap中记录的<constructor>节点信息 如果不为空 则可以确定对应的构造函数
    final List<ResultMapping> constructorMappings = resultMap.getConstructorResultMappings();
    // 创建结果对象  有4中情况
    // 1.结果集只有一列，且存在TypeHandler对象可以将该列转换为 resultType 类型的值
    if (hasTypeHandlerForResultObject(rsw, resultType)) {
      //查找对应的TypeHandler 在使用TypeHandler 转换
      return createPrimitiveResultObject(rsw, resultMap, columnPrefix);
    } else if (!constructorMappings.isEmpty()) {
      // 2.RsultMap中记录的有<constructor>节点信息，通过反射方式调用构造方法，创建结果对象
      return createParameterizedResultObject(rsw, resultType, constructorMappings, constructorArgTypes, constructorArgs, colu
    } else if (resultType.isInterface() || metaType.hasDefaultConstructor()) {
      // 3.使用默认无参构造函数创建
      return objectFactory.create(resultType);
    } else if (shouldApplyAutomaticMappings(resultMap, isNested: false)) {
      // 4.通过自动映射的方式查找合适的构造方法并创建对象
      return createByConstructorSignature(rsw, resultType, constructorArgTypes, constructorArgs);
```

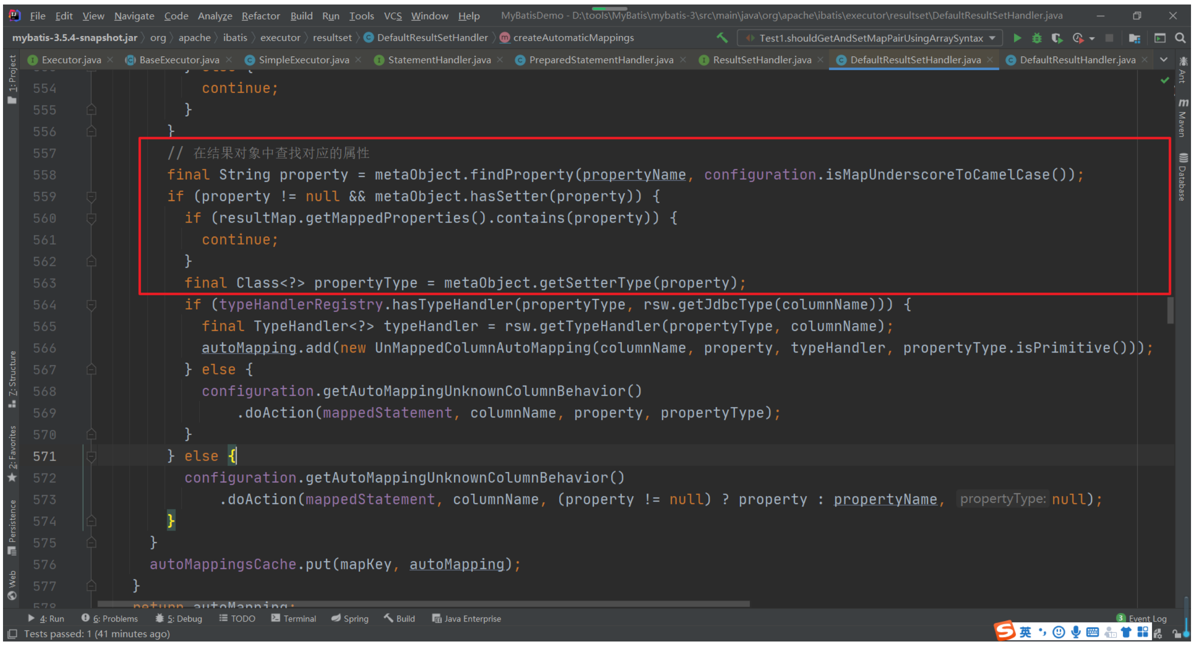
然后在DefaultResultSetHandler的getRowValue方法中在做自动映射的时候



```
  //
  // GET VALUE FROM ROW FOR SIMPLE RESULT MAP
  //

  private Object getRowValue(ResultSetWrapper rsw, ResultMap resultMap, String columnPrefix) throws SQLException {
    // 和延迟加载有关的对象
    final ResultLoaderMap lazyLoader = new ResultLoaderMap();
    // 创建该行记录映射之后得到的结果对象，该结果对象的类型有<reslutMap> 节点的 type 属性执行
    Object rowValue = createResultObject(rsw, resultMap, lazyLoader, columnPrefix);
    if (rowValue != null && !hasTypeHandlerForResultObject(rsw, resultMap.getType())) {
      final MetaObject metaObject = configuration.newMetaObject(rowValue);
      boolean foundValues = this.useConstructorMappings;
      // 是否允许进行自动映射
      if (shouldApplyAutomaticMappings(resultMap, isNested: false)) {
        // 自动映射
        foundValues = applyAutomaticMappings(rsw, resultMap, metaObject, columnPrefix) || foundValues;
      }
      // 处理ResultMap中有映射关系的属性
      foundValues = applyPropertyMappings(rsw, resultMap, metaObject, lazyLoader, columnPrefix) || foundValues;
      foundValues = lazyLoader.size() > 0 || foundValues;
      rowValue = foundValues || configuration.isReturnInstanceForEmptyRow() ? rowValue : null;
    }
    return rowValue;
```

继续跟踪，在createAutomaticMappings方法中



```
            continue;
          }
        }
        // 在结果对象中查找对应的属性
        final String property = metaObject.findProperty(propertyName, configuration.isMapUnderscoreToCamelCase());
        if (property != null && metaObject.hasSetter(property)) {
          if (resultMap.getMappedProperties().contains(property)) {
            continue;
          }
          final Class<?> propertyType = metaObject.getSetterType(property);
          if (typeHandlerRegistry.hasTypeHandler(propertyType, rsw.getJdbcType(columnName))) {
            final TypeHandler<?> typeHandler = rsw.getTypeHandler(propertyType, columnName);
            autoMapping.add(new UnMappedColumnAutoMapping(columnName, property, typeHandler, propertyType.isPrimitive()));
          } else {
            configuration.getAutoMappingUnknownColumnBehavior()
                .doAction(mappedStatement, columnName, property, propertyType);
          }
        } else {
          configuration.getAutoMappingUnknownColumnBehavior()
              .doAction(mappedStatement, columnName, (property != null) ? property : propertyName, propertyType: null);
        }
      }
      autoMappingsCache.put(mapKey, autoMapping);
    }
    return autoMapping;
```

当然还有很多其他的地方在使用反射模块来完成的相关操作，这些可自行查阅