MyBatis基础模块讲解与强化核心原理

课程目标

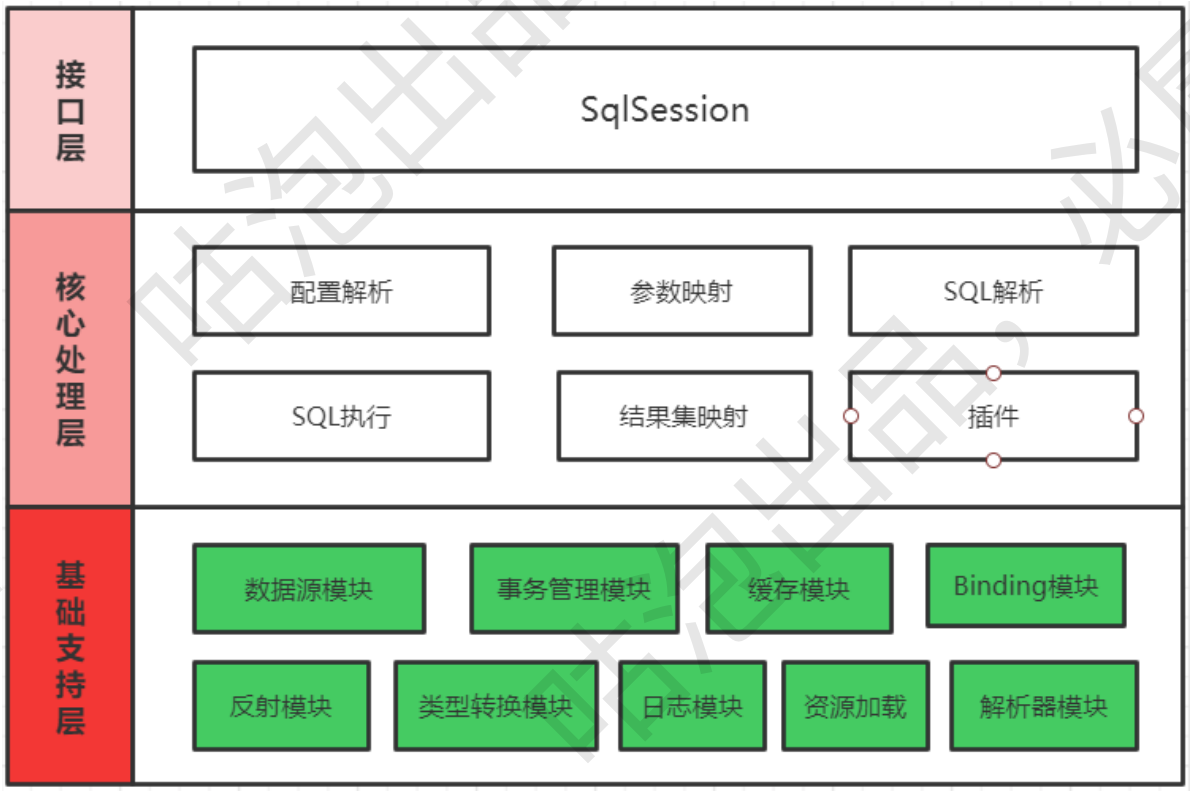
1. ORM框架的发展历史与MyBatis的高级应用
2. MyBatis的体系结构与核心工作原理分析
3. MyBatis基础模块讲解与强化核心原理
4. 探寻插件的原理与深究和Spring的集成
5. 通过手写MyBatis带你掌握自己写框架的秘诀

内容定位

1、适合已经掌握MyBatis框架结构的小伙伴
2、适合已经熟悉MyBatis核心流程的小伙伴
3、想要系统的掌握MyBatis的设计的小伙伴

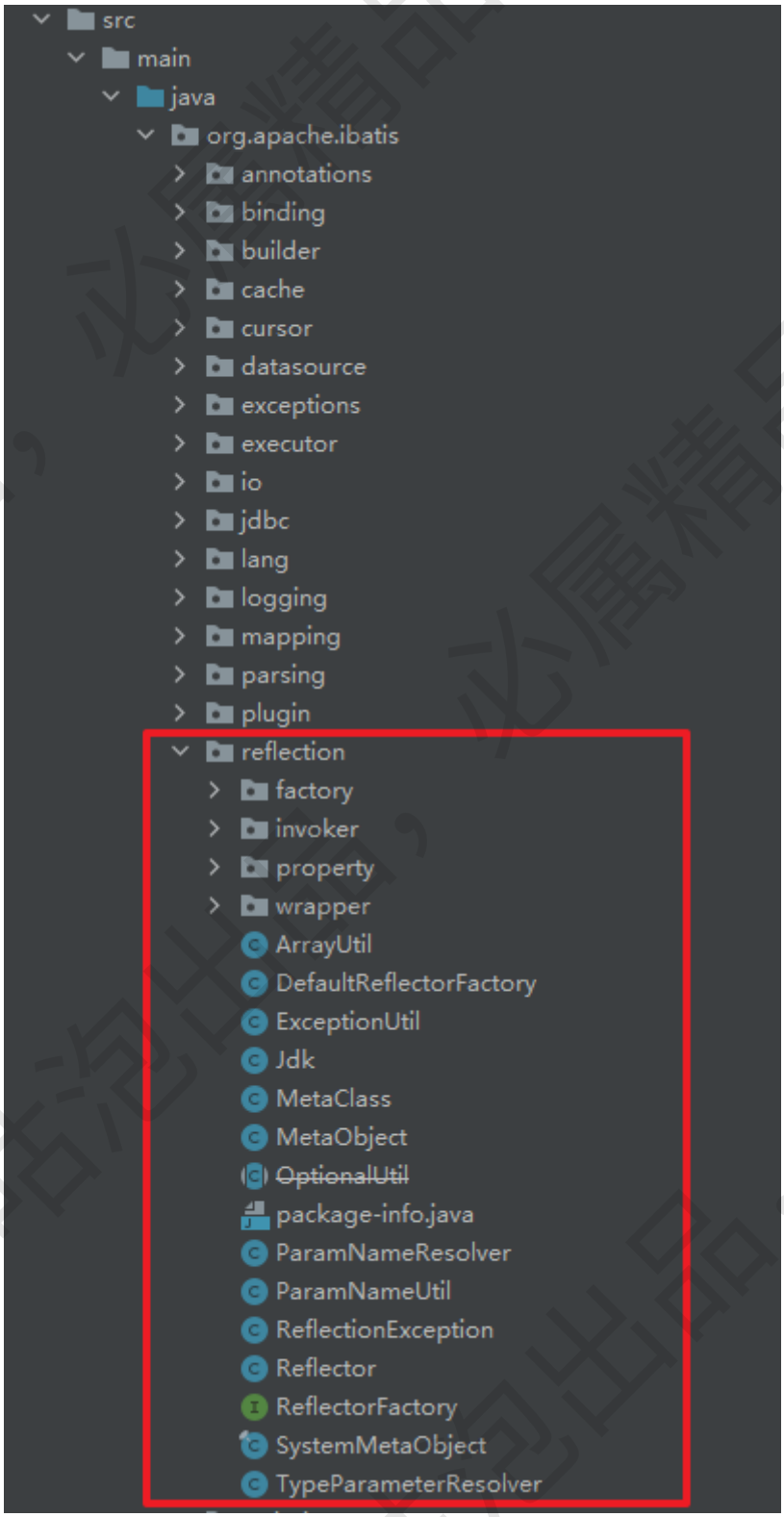
一、基础支持层

基础支持层位于MyBatis整体架构的最底层，支撑着MyBatis的核心处理层，是整个框架的基石。基础支持层中封装了多个较为通用的、独立的模块。不仅仅为MyBatis提供基础支撑，也可以在合适的场景中直接复用。



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。



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

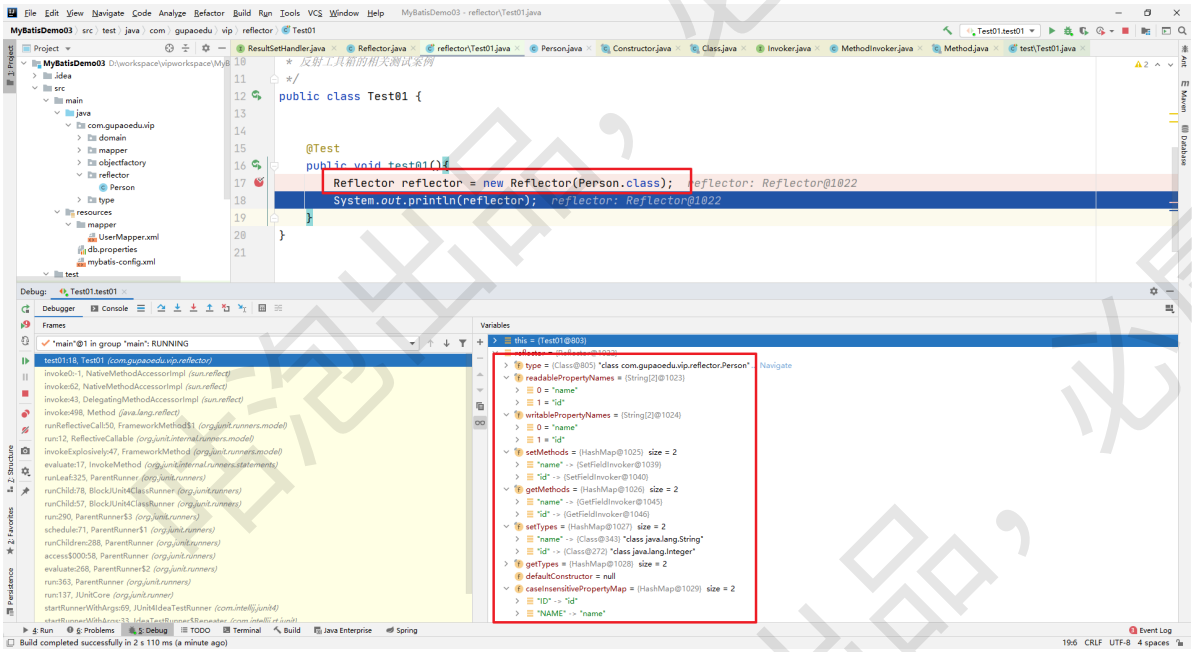
```
  // 解析指定的Class类型 并填充上述的集合信息
  public Reflector(Class<?> clazz) {
    type = clazz; // 初始化 type字段
    addDefaultConstructor(clazz);// 设置默认的构造方法
    addGetMethods(clazz);// 获取getter方法
    addSetMethods(clazz); // 获取setter方法
    addFields(clazz); // 处理没有getter/setter方法的字段
    // 初始化 可读属性名称集合
    readablePropertyNames = 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;
    }
}
```

测试



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

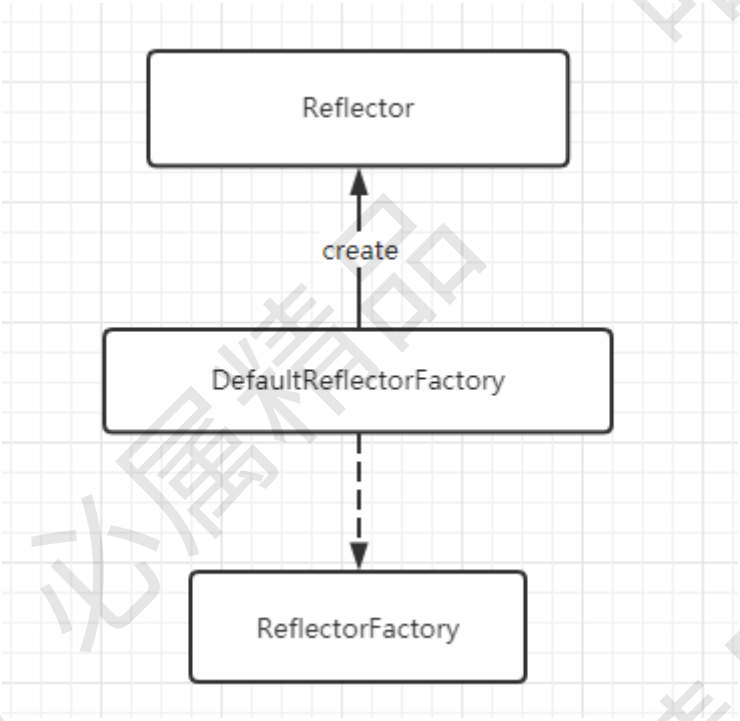
接口的定义如下

```
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);
    }
  }
}
```

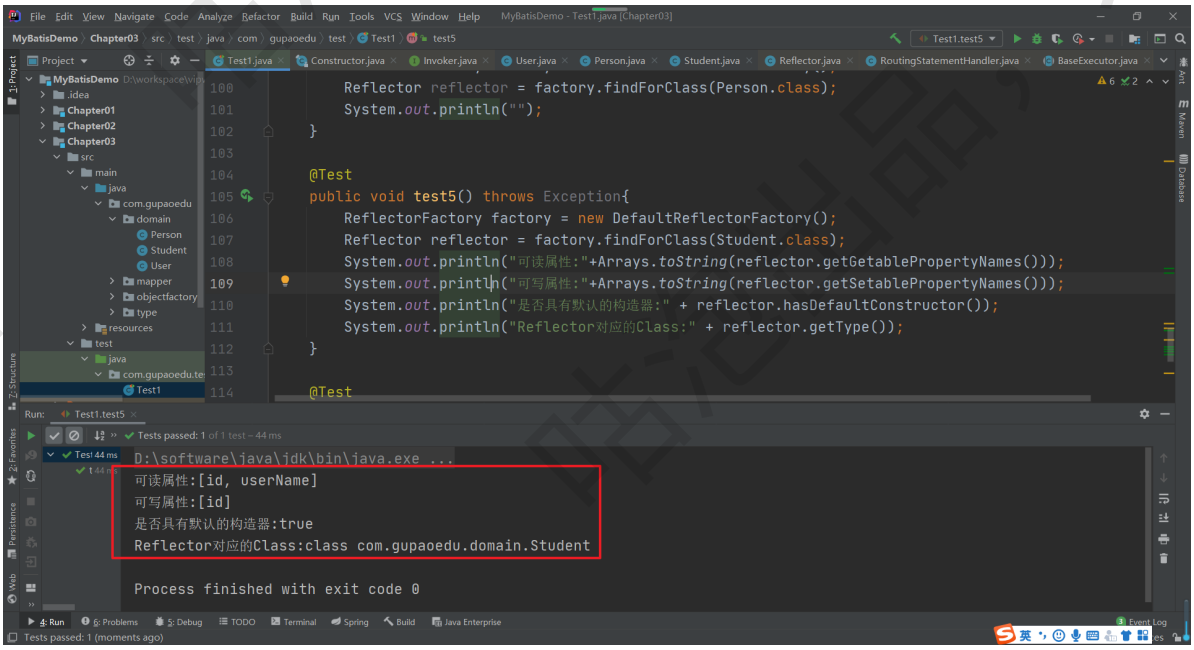
1.2.3 使用演示

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.gupaoedu.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());
    }
```



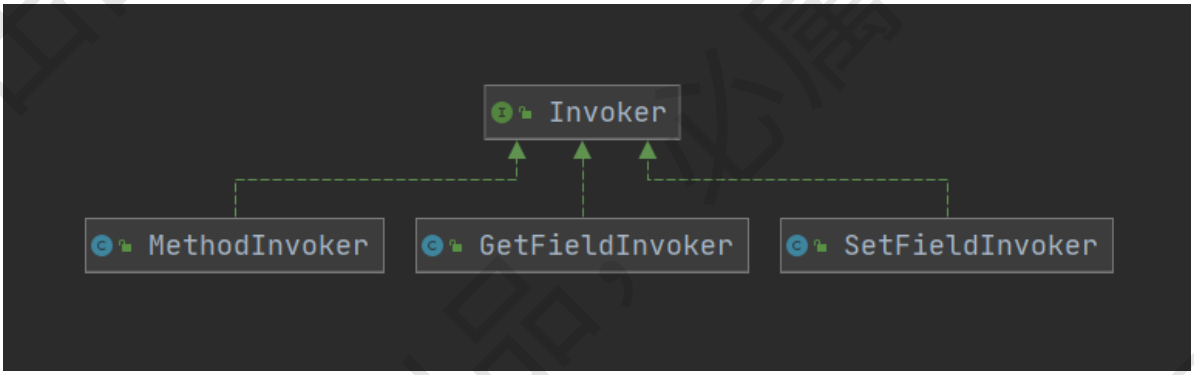
1.3 Invoker

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

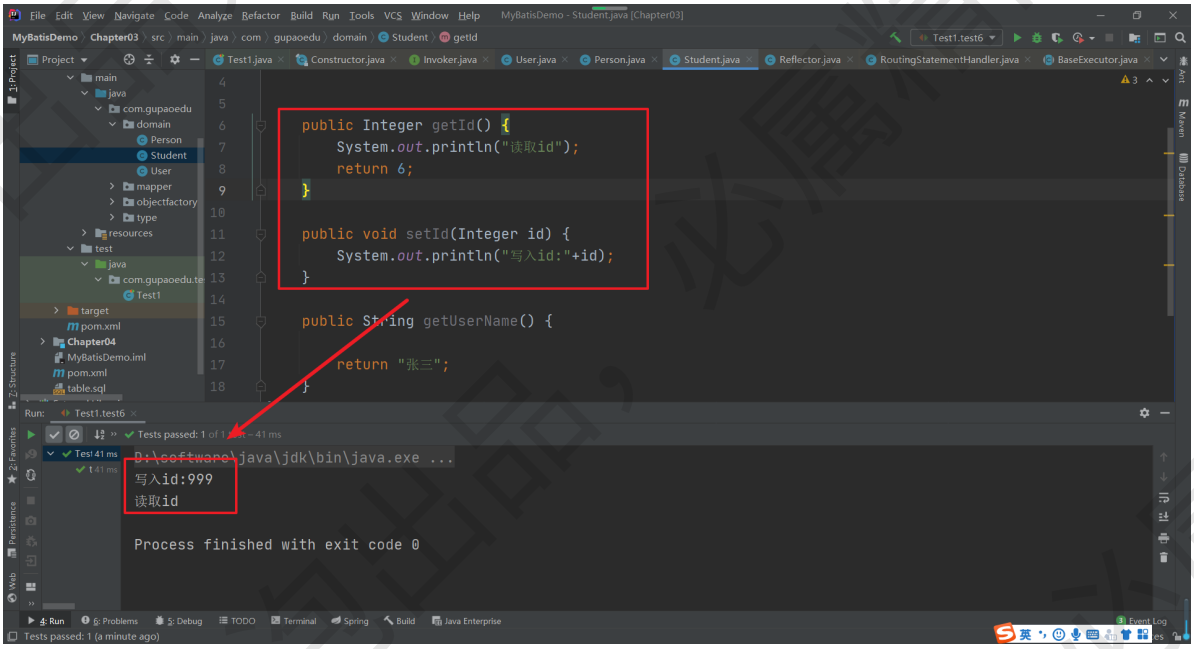
```
package com.gupaoedu.domain;

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);
    }
```

效果



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.gupaoedu.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

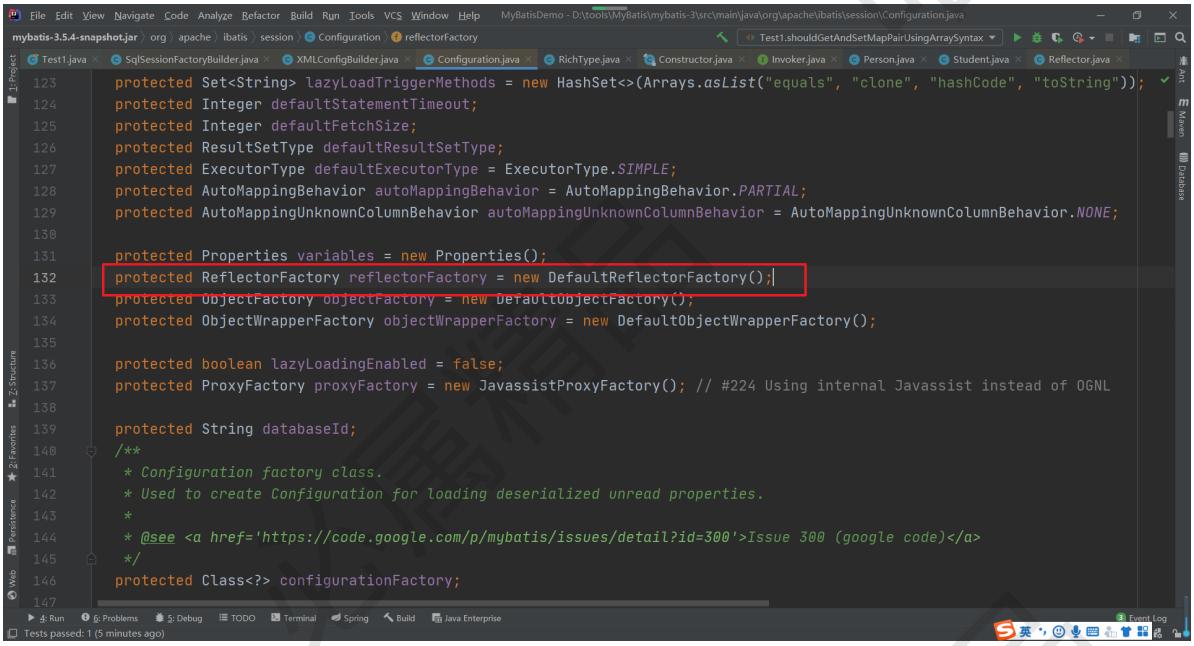
```
foo
```

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



```
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;
```

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

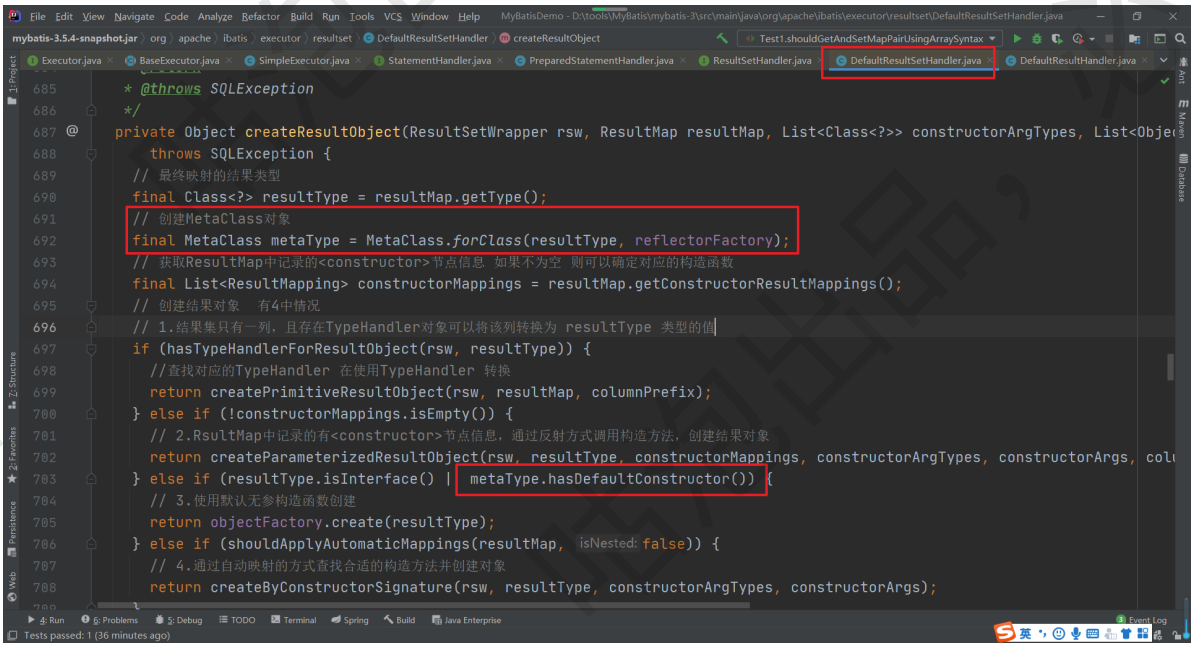
无相关操作

1.6.3 Mapper

无相关操作

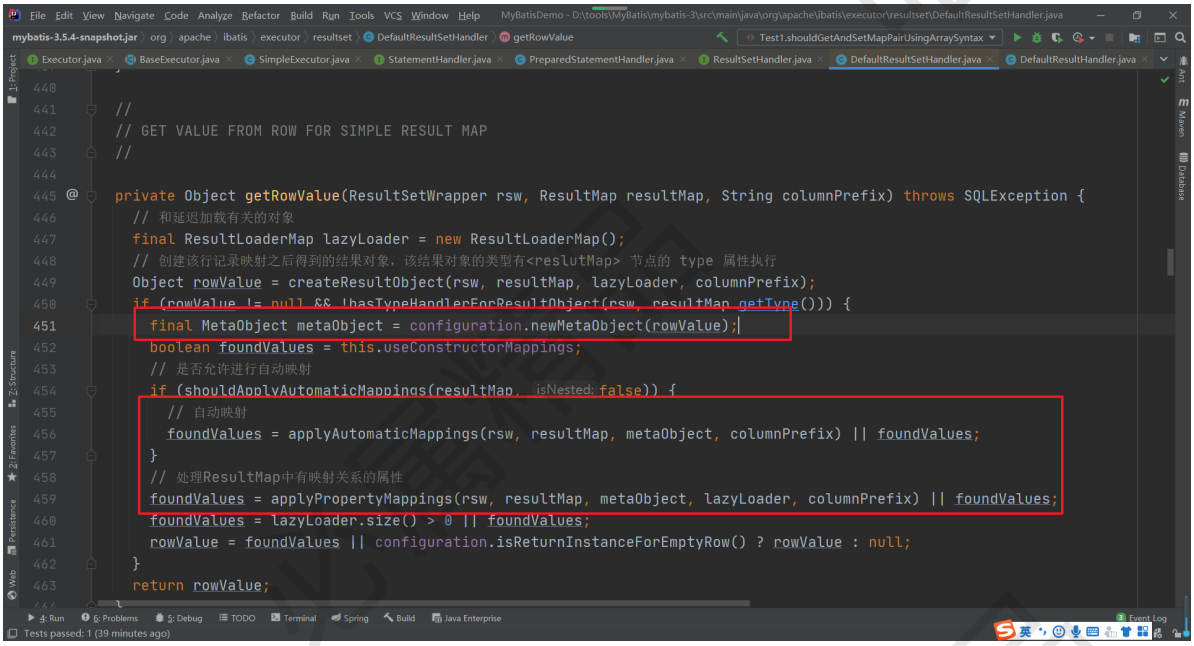
1.6.4 执行SQL

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。



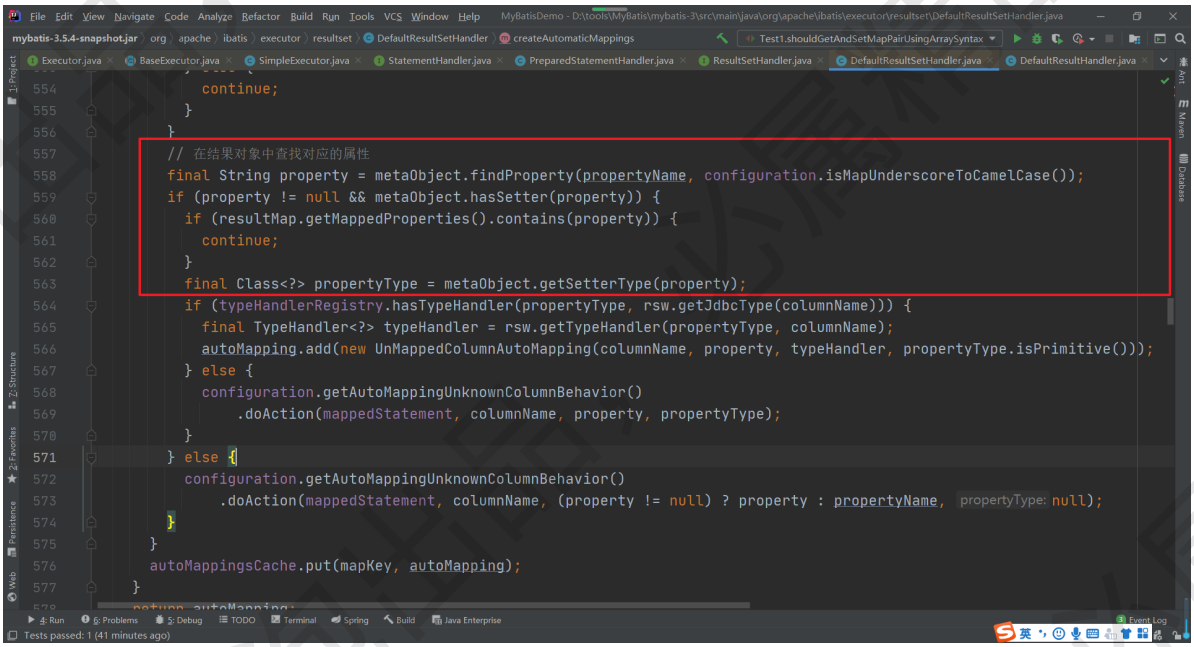
```
 * @throws SQLException
 */
private Object createResultObject(ResultSetWrapper rsw, ResultMap resultMap, List<Class<?>> constructorArgTypes, List<Obje
    throws SQLException {
  // 最终映射的结果类型
  final Class<?> resultType = resultMap.getType();
  // 创建MetaClass对象
  final MetaClass metaType = MetaClass.forClass(resultType, 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



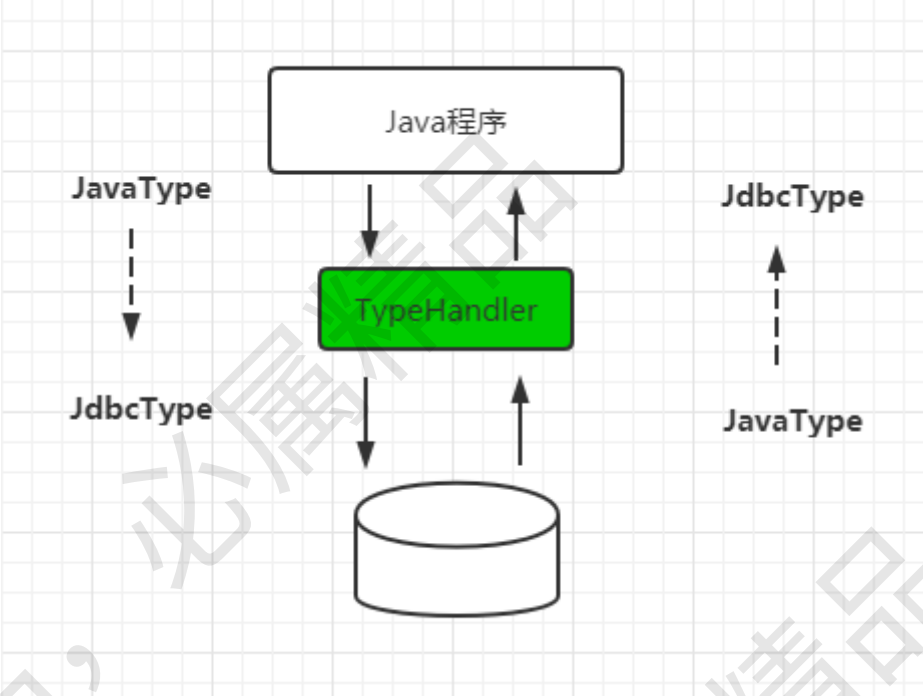
```
            continue;
          }
        }
        // 在结果对象中查找对应的属性
        final String property = metaObject.findProperty(propertyName, configuration.isMapUnderscoreToCamelCase());
        if (property != null && metaObject.hasSetter(property)) {
          if (resultMap.getMappedProperties().contains(property)) {
            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);
    }
```

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

2.类型转换模块

```
String sql = "SELECT id,user_name,real_name,password,age,d_id from t_user where
id = ? and user_name = ?";
ps = conn.prepareStatement(sql);
ps.setInt(1,2);
ps.setString(2,"张三");
```

MyBatis是一个持久层框架ORM框架，实现数据库中数据和Java对象中的属性的双向映射，那么不可避免的就会碰到类型转换的问题，在PreparedStatement为SQL语句绑定参数时，需要从Java类型转换为JDBC类型，而从结果集中获取数据时，则需要从JDBC类型转换为Java类型，所以我们来看下在MyBatis中是如何实现类型的转换的。



2.1 TypeHandler

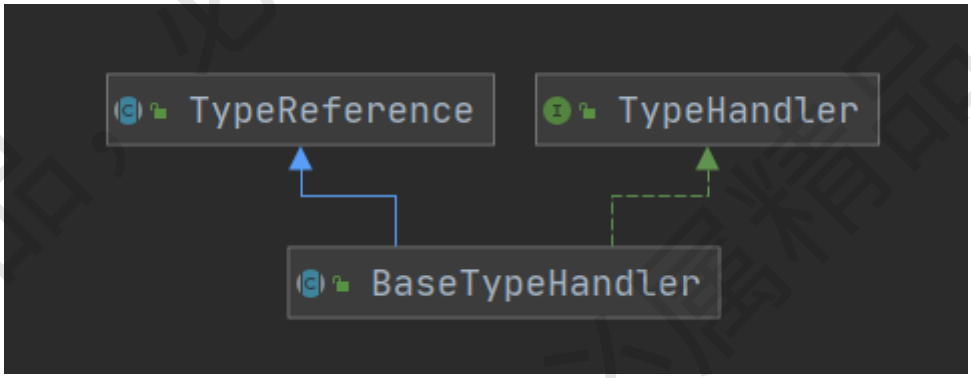
MyBatis中的所有的类型转换器都继承了TypeHandler接口，在TypeHandler中定义了类型转换器的最基本的功能。

```
/**
 * @author Clinton Begin
 */
public interface TypeHandler<T> {

  /**
   * 负责将Java类型转换为JDBC的类型
   *    本质上执行的就是JDBC操作中的 如下操作
   *        String sql = "SELECT id,user_name,real_name,password,age,d_id from
t_user where id = ? and user_name = ?";
   *        ps = conn.prepareStatement(sql);
   *        ps.setInt(1,2);
   *        ps.setString(2,"张三");
   * @param ps
   * @param i 对应占位符的 位置
   * @param parameter 占位符对应的值
   * @param jdbcType 对应的 jdbcType 类型
   * @throws SQLException
   */
  void setParameter(PreparedStatement ps, int i, T parameter, JdbcType jdbcType)
throws SQLException;

  /**
   * 从ResultSet中获取数据时会调用此方法，会将数据由JdbcType转换为Java类型
   * @param columnName Colunm name, when configuration
<code>useColumnLabel</code> is <code>false</code>
   */
```

```
    T getResult(ResultSet rs, String columnName) throws SQLException;

    T getResult(ResultSet rs, int columnIndex) throws SQLException;

    T getResult(CallableStatement cs, int columnIndex) throws SQLException;

}
```

2.2 BaseTypeHandler

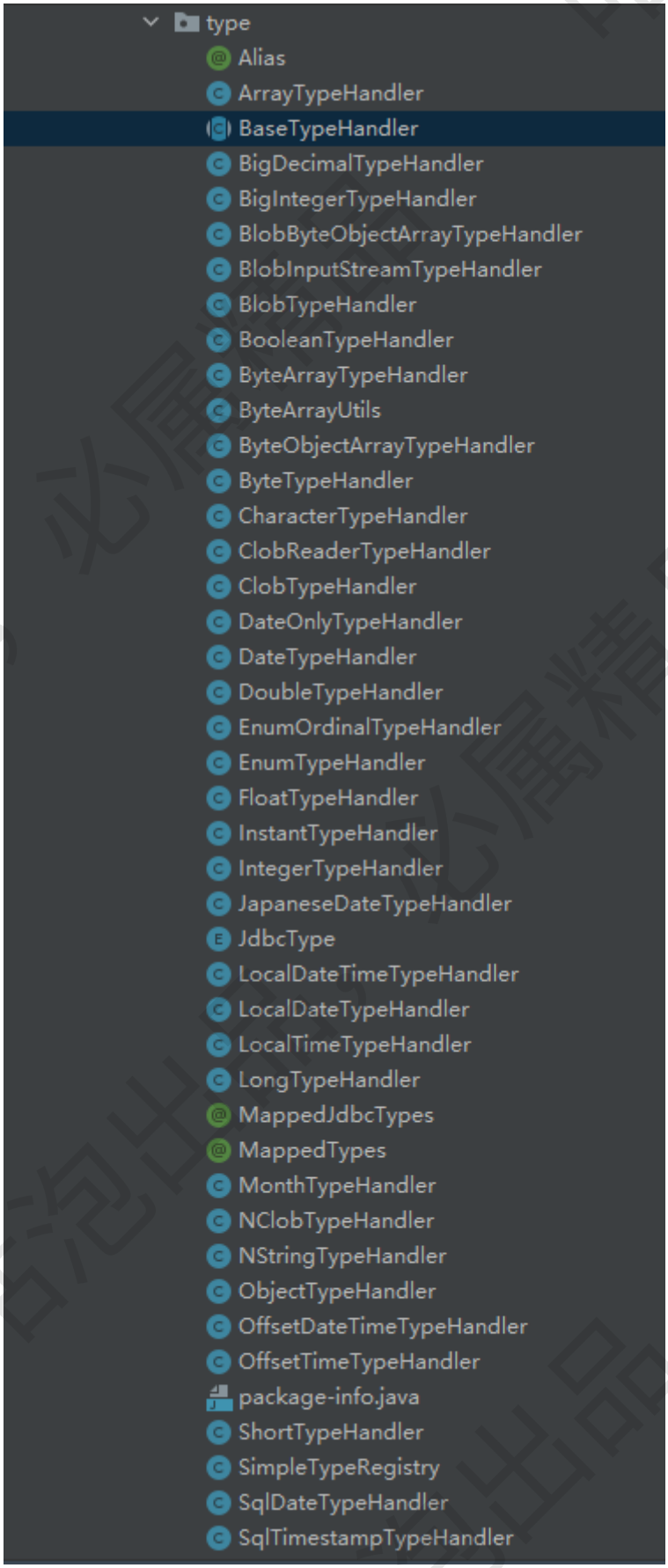
为了方便用户自定义TypeHandler的实现，在MyBatis中提供了BaseTypeHandler这个抽象类，它实现了TypeHandler接口，并继承了TypeReference类，



在BaseTypeHandler中的实现方法中实现了对null的处理，非空的处理是交给各个子类去实现的。这个在代码中很清楚的体现了出来

2.3 TypeHandler实现类

TypeHandler的实现类比较多，而且实现也都比较简单。



以Integer为例

```
/**
 * @author Clinton Begin
 */
public class IntegerTypeHandler extends BaseTypeHandler<Integer> {
```

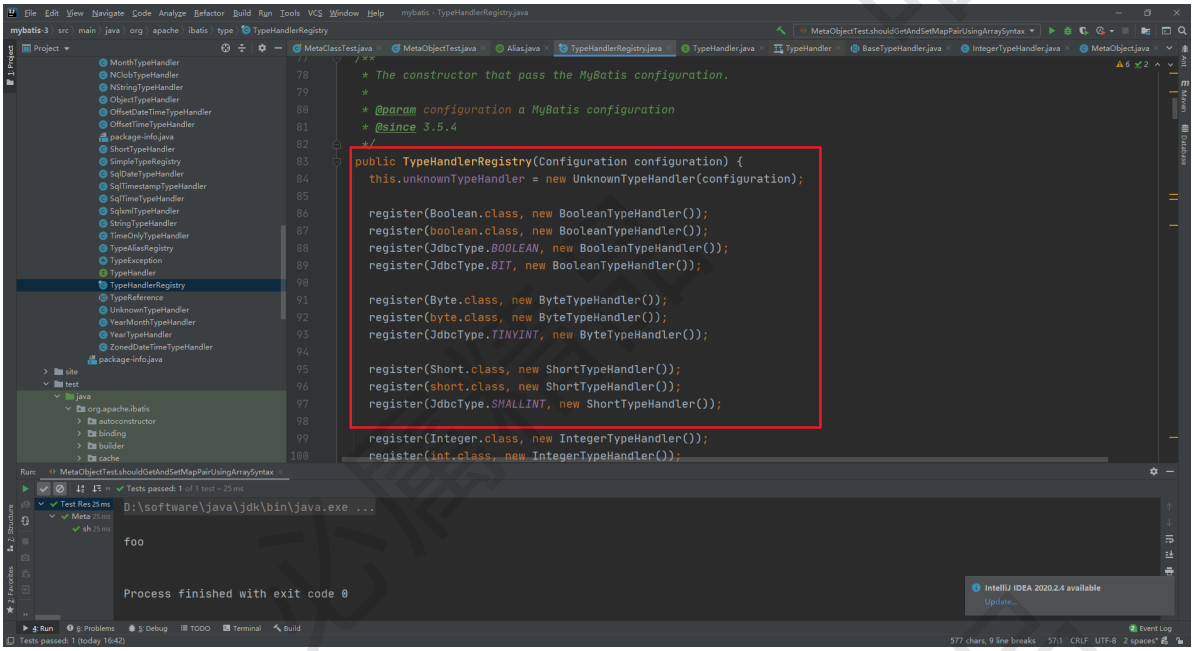
```
  @Override
  public void setNonNullParameter(PreparedStatement ps, int i, Integer
parameter, JdbcType jdbcType)
      throws SQLException {
    ps.setInt(i, parameter); // 实现参数的绑定
  }

  @Override
  public Integer getNullableResult(ResultSet rs, String columnName)
      throws SQLException {
    int result = rs.getInt(columnName); // 获取指定列的值
    return result == 0 && rs.wasNull() ? null : result;
  }

  @Override
  public Integer getNullableResult(ResultSet rs, int columnIndex)
      throws SQLException {
    int result = rs.getInt(columnIndex); // 获取指定列的值
    return result == 0 && rs.wasNull() ? null : result;
  }

  @Override
  public Integer getNullableResult(CallableStatement cs, int columnIndex)
      throws SQLException {
    int result = cs.getInt(columnIndex); // 获取指定列的值
    return result == 0 && cs.wasNull() ? null : result;
  }
}
```

2.4 TypeHandlerRegistry

通过前面的介绍我们发现在MyBatis中给我们提供的具体的类型转换器实在是太多了，那么在实际的使用时我们是如何知道使用哪个转换器类处理的呢？实际上再MyBatis中是将所有的TypeHandler都保存注册在了TypeHandlerRegistry中的。首先注意声明的相关属性

```
  // 记录JdbcType和TypeHandle的对应关系
  private final Map<JdbcType, TypeHandler<?>>  jdbcTypeHandlerMap = new
EnumMap<>(JdbcType.class);
  // 记录Java类型向指定的JdbcType转换时需要使用到的TypeHandle
  private final Map<Type, Map<JdbcType, TypeHandler<?>>> typeHandlerMap = new
ConcurrentHashMap<>();
  private final TypeHandler<Object> unknownTypeHandler;
  // 记录全部的TypeHandle类型及对应的TypeHandle对象
  private final Map<Class<?>, TypeHandler<?>> allTypeHandlersMap = new HashMap<>
();

  // 空TypeHandle的标识
  private static final Map<JdbcType, TypeHandler<?>> NULL_TYPE_HANDLER_MAP =
Collections.emptyMap();
```

然后在器构造方法中完成了系统提供的TypeHandler的注册



代码太长，请自行查阅。注意的是register()方法， 关键几个实现如下

```
  private <T> void register(Type javaType, TypeHandler<? extends T> typeHandler)
{
    // 获取@MappedJdbcTypes注解
    MappedJdbcTypes mappedJdbcTypes =
typeHandler.getClass().getAnnotation(MappedJdbcTypes.class);
    if (mappedJdbcTypes != null) {
      // 遍历获取注解中指定的 JdbcType 类型
      for (JdbcType handledJdbcType : mappedJdbcTypes.value()) {
        // 调用下一个重载的方法
        register(javaType, handledJdbcType, typeHandler);
      }
      if (mappedJdbcTypes.includeNullJdbcType()) {
        // JdbcType类型为空的情况
        register(javaType, null, typeHandler);
      }
    } else {
      register(javaType, null, typeHandler);
    }
  }

  private void register(Type javaType, JdbcType jdbcType, TypeHandler<?>
handler) {
    if (javaType != null) {// 如果不为空
      // 从 TypeHandle集合中根据Java类型来获取对应的集合
      Map<JdbcType, TypeHandler<?>> map = typeHandlerMap.get(javaType);
      if (map == null || map == NULL_TYPE_HANDLER_MAP) {
        // 如果没有就创建一个新的
        map = new HashMap<>();
      }
      // 把对应的jdbc类型和处理器添加到map集合中
      map.put(jdbcType, handler);
      // 然后将 java类型和上面的map集合保存到TypeHandle的容器中
      typeHandlerMap.put(javaType, map);
    }
    // 同时也把这个处理器添加到了 保存有所有处理器的容器中
    allTypeHandlersMap.put(handler.getClass(), handler);
  }
```

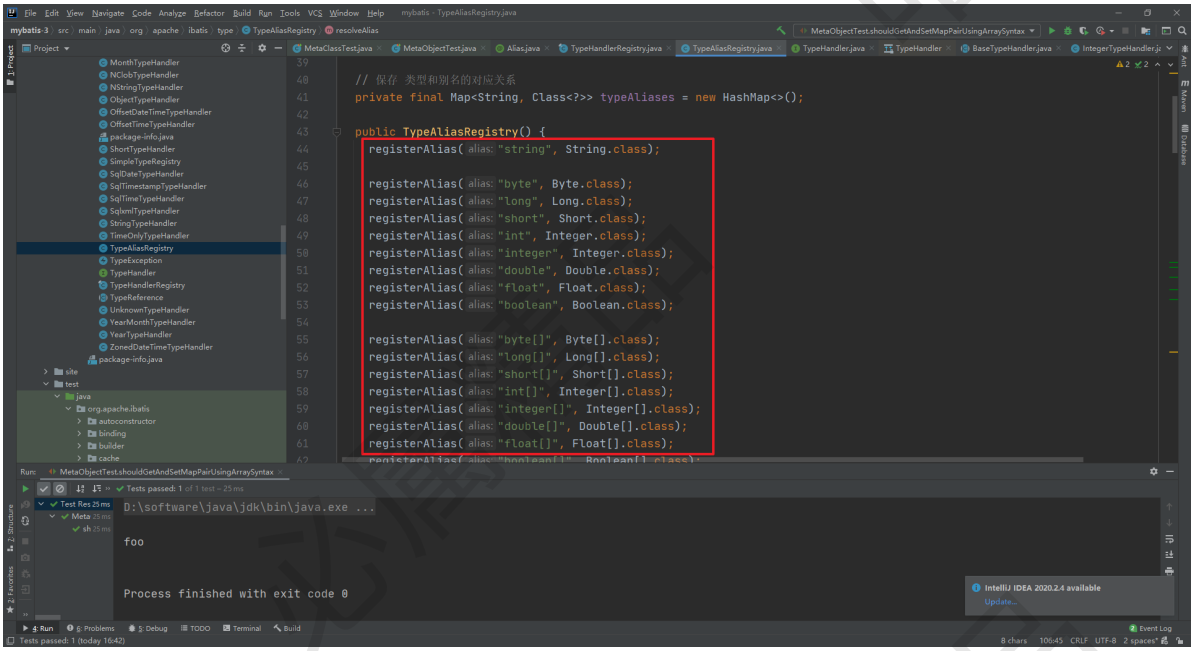
有注册的方法，当然也有从注册器中获取TypeHandler的方法，getTypeHandler方法，这个方法也有多个重载的方法，这里重载的方法最终都会执行的方法是

```
/**
 * 根据对应的Java类型和Jdbc类型来查找对应的TypeHandle
 */
private <T> TypeHandler<T> getTypeHandler(Type type, JdbcType jdbcType) {
  if (ParamMap.class.equals(type)) {
    return null;
  }
  // 根据Java类型获取对应的 Jdbc类型和TypeHandle的集合容器
  Map<JdbcType, TypeHandler<?>> jdbcHandlerMap = getJdbcHandlerMap(type);
  TypeHandler<?> handler = null;
  if (jdbcHandlerMap != null) {
    // 根据Jdbc类型获取对应的 处理器
    handler = jdbcHandlerMap.get(jdbcType);
    if (handler == null) {
      // 获取null对应的处理器
      handler = jdbcHandlerMap.get(null);
    }
    if (handler == null) {
      // #591
      handler = pickSoleHandler(jdbcHandlerMap);
    }
  }
  // type drives generics here
  return (TypeHandler<T>) handler;
}
```

当然除了使用系统提供的TypeHandler以外，我们还可以创建我们自己的TypeHandler了，之前讲解案例的时候已经带大家写过了，如果忘记可以复习下。

2.5 TypeAliasRegistry

我们在MyBatis的应用的时候会经常用到别名，这能大大简化我们的代码，其实在MyBatis中是通过TypeAliasRegistry类管理的。首先在构造方法中会注入系统常见类型的别名



注册的方法逻辑也比较简单

```
  public void registerAlias(String alias, Class<?> value) {
    if (alias == null) {
      throw new TypeException("The parameter alias cannot be null");
    }
    // issue #748 别名统一转换为小写
    String key = alias.toLowerCase(Locale.ENGLISH);
    // 检测别名是否存在
    if (typeAliases.containsKey(key) && typeAliases.get(key) != null &&
!typeAliases.get(key).equals(value)) {
      throw new TypeException("The alias '" + alias + "' is already mapped to
the value '" + typeAliases.get(key).getName() + "'.");
    }
    // 将 别名 和 类型 添加到 Map 集合中
    typeAliases.put(key, value);
  }
```

那么我们在实际使用时通过package指定别名路径和通过@Alisa注解来指定别名的操作是如何实现的呢？也在TypeAliasRegistry中有实现

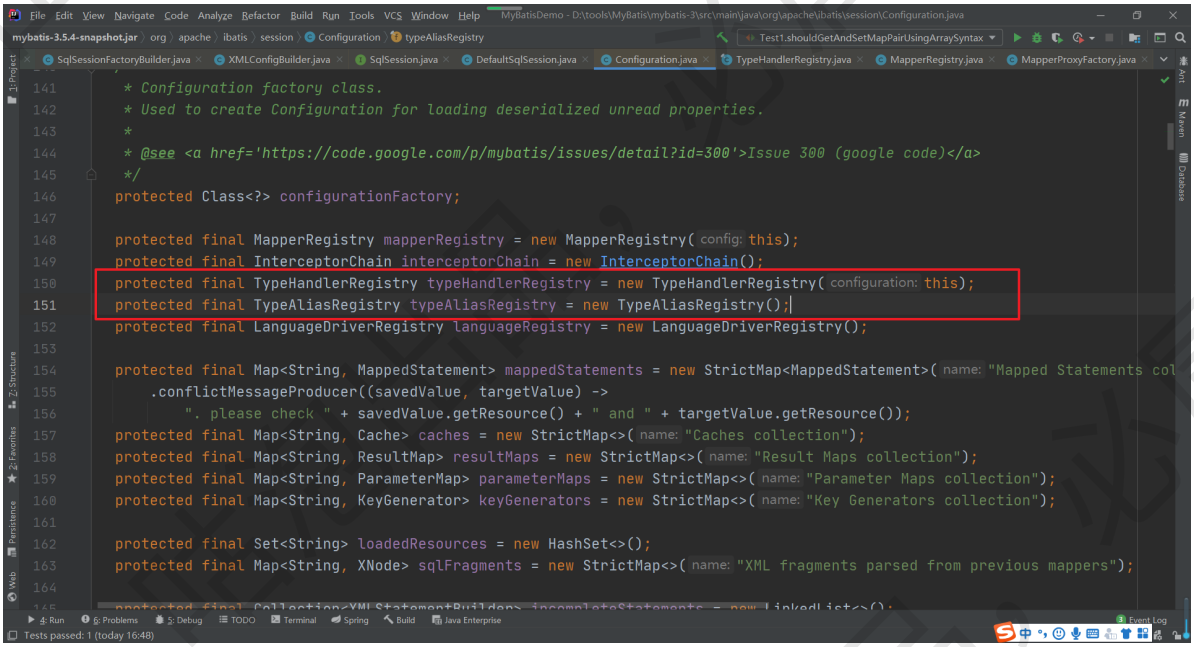
```
  /**
   * 根据 packagename 来指定
   * @param packageName
   * @param superType
   */
  public void registerAliases(String packageName, Class<?> superType) {
    ResolverUtil<Class<?>> resolverUtil = new ResolverUtil<>();
    resolverUtil.find(new ResolverUtil.IsA(superType), packageName);
    Set<Class<? extends Class<?>>> typeSet = resolverUtil.getClasses();
    for (Class<?> type : typeSet) {
      // Ignore inner classes and interfaces (including package-info.java)
      // Skip also inner classes. See issue #6
      if (!type.isAnonymousClass() && !type.isInterface() &&
!type.isMemberClass()) {
        registerAlias(type);
      }
    }
```

```
  }
  /**
   * 扫描 @Alias注解
   * @param type
   */
  public void registerAlias(Class<?> type) {
    String alias = type.getSimpleName();
    // 扫描 @Alias注解
    Alias aliasAnnotation = type.getAnnotation(Alias.class);
    if (aliasAnnotation != null) {
      // 获取注解中定义的别名名称
      alias = aliasAnnotation.value();
    }
    registerAlias(alias, type);
  }
```

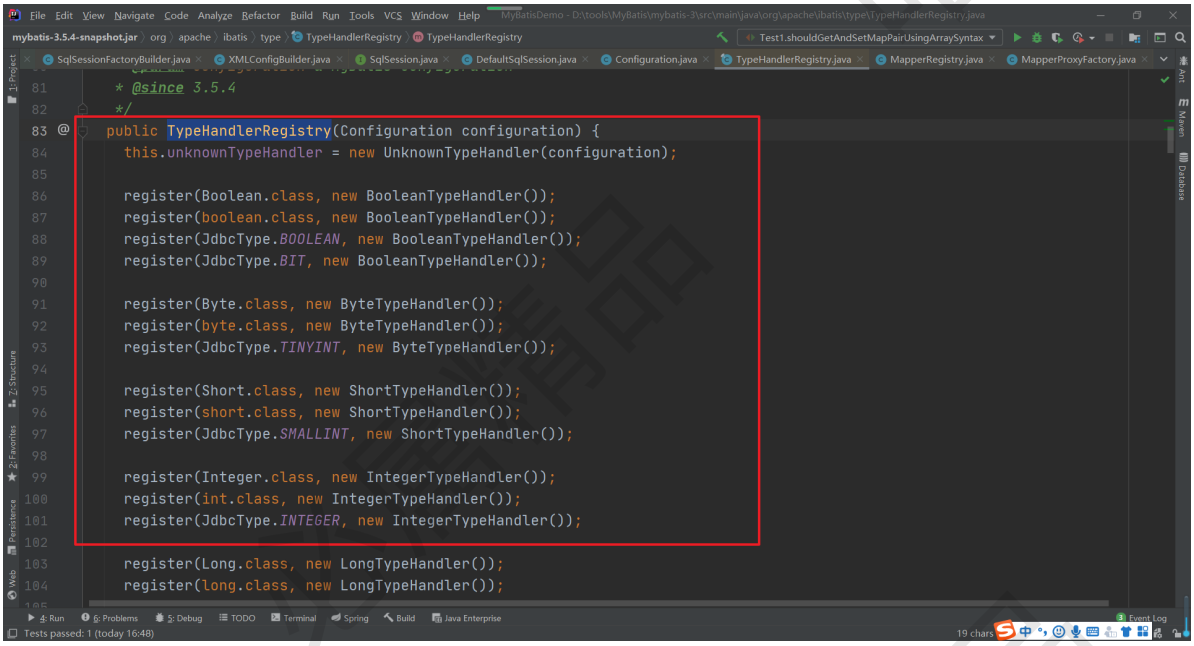
2.6 TypeHandler的应用

2.6.1 SqlSessionFactory

在构建SqlSessionFactory时，在Configuration对象实例化的时候在成员变量中完成了TypeHandlerRegistry和TypeAliasRegistry的实例化

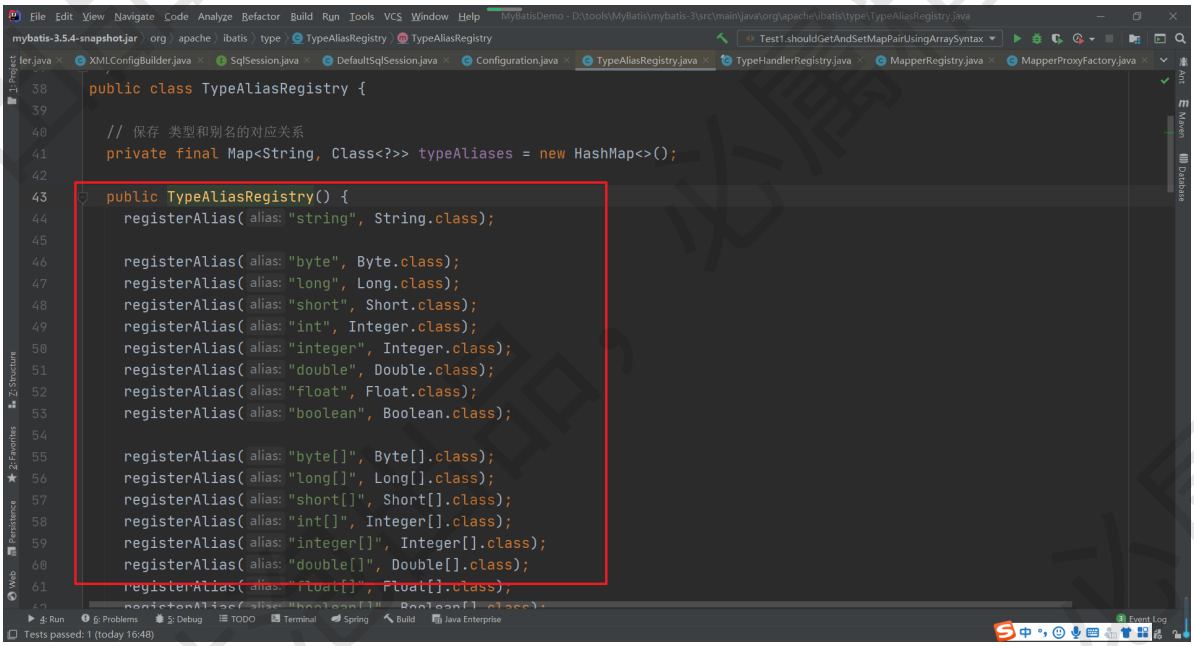


在TypeHandlerRegistry的构造方法中完成了常用类型的TypeHandler的注册



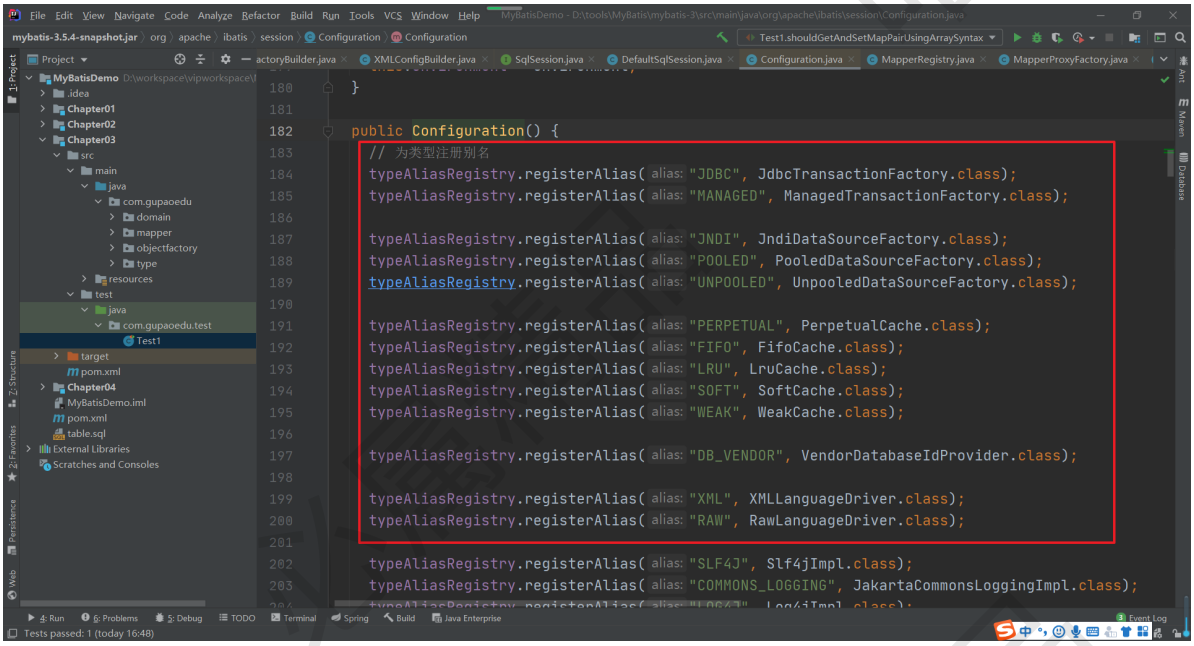
```
81      * @since 3.5.4
82      */
83     public TypeHandlerRegistry(Configuration configuration) {
84       this.unknownTypeHandler = new UnknownTypeHandler(configuration);
85
86       register(Boolean.class, new BooleanTypeHandler());
87       register(boolean.class, new BooleanTypeHandler());
88       register(JdbcType.BOOLEAN, new BooleanTypeHandler());
89       register(JdbcType.BIT, new BooleanTypeHandler());
90
91       register(Byte.class, new ByteTypeHandler());
92       register(byte.class, new ByteTypeHandler());
93       register(JdbcType.TINYINT, new ByteTypeHandler());
94
95       register(Short.class, new ShortTypeHandler());
96       register(short.class, new ShortTypeHandler());
97       register(JdbcType.SMALLINT, new ShortTypeHandler());
98
99       register(Integer.class, new IntegerTypeHandler());
100      register(int.class, new IntegerTypeHandler());
101      register(JdbcType.INTEGER, new IntegerTypeHandler());
102
103      register(Long.class, new LongTypeHandler());
104      register(long.class, new LongTypeHandler());
```

在TypeAliasRegistry中完成了常用Java类型别名的注册



```
38    public class TypeAliasRegistry {
39
40      // 保存 类型和别名的对应关系
41      private final Map<String, Class<?>> typeAliases = new HashMap<>();
42
43      public TypeAliasRegistry() {
44        registerAlias( alias: "string", String.class);
45
46        registerAlias( alias: "byte", Byte.class);
47        registerAlias( alias: "long", Long.class);
48        registerAlias( alias: "short", Short.class);
49        registerAlias( alias: "int", Integer.class);
50        registerAlias( alias: "integer", Integer.class);
51        registerAlias( alias: "double", Double.class);
52        registerAlias( alias: "float", Float.class);
53        registerAlias( alias: "boolean", Boolean.class);
54
55        registerAlias( alias: "byte[]", Byte[].class);
56        registerAlias( alias: "long[]", Long[].class);
57        registerAlias( alias: "short[]", Short[].class);
58        registerAlias( alias: "int[]", Integer[].class);
59        registerAlias( alias: "integer[]", Integer[].class);
60        registerAlias( alias: "double[]", Double[].class);
61        registerAlias( alias: "float[]", Float[].class);
```

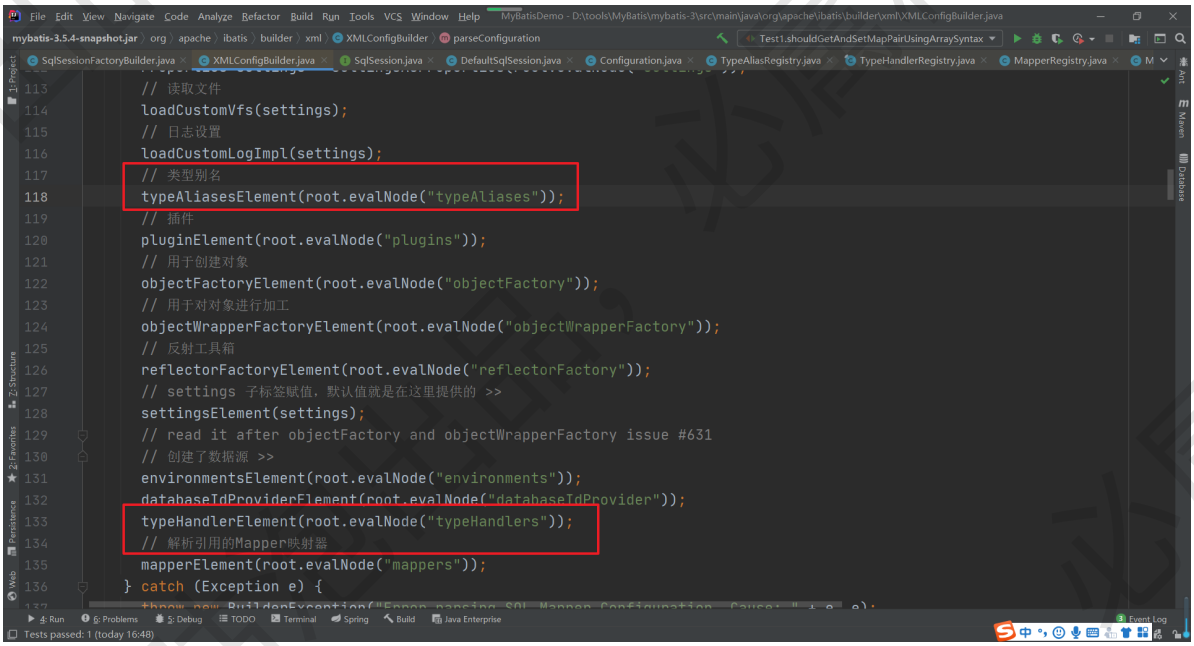
在Configuration的构造方法中会为各种常用的类型向TypeAliasRegistry中注册类型别名数据



```
public Configuration() {
    // 为类型注册别名
    typeAliasRegistry.registerAlias( alias: "JDBC", JdbcTransactionFactory.class);
    typeAliasRegistry.registerAlias( alias: "MANAGED", ManagedTransactionFactory.class);

    typeAliasRegistry.registerAlias( alias: "JNDI", JndiDataSourceFactory.class);
    typeAliasRegistry.registerAlias( alias: "POOLED", PooledDataSourceFactory.class);
    typeAliasRegistry.registerAlias( alias: "UNPOOLED", UnpooledDataSourceFactory.class);

    typeAliasRegistry.registerAlias( alias: "PERPETUAL", PerpetualCache.class);
    typeAliasRegistry.registerAlias( alias: "FIFO", FifoCache.class);
    typeAliasRegistry.registerAlias( alias: "LRU", LruCache.class);
    typeAliasRegistry.registerAlias( alias: "SOFT", SoftCache.class);
    typeAliasRegistry.registerAlias( alias: "WEAK", WeakCache.class);

    typeAliasRegistry.registerAlias( alias: "DB_VENDOR", VendorDatabaseIdProvider.class);

    typeAliasRegistry.registerAlias( alias: "XML", XMLLanguageDriver.class);
    typeAliasRegistry.registerAlias( alias: "RAW", RawLanguageDriver.class);

    typeAliasRegistry.registerAlias( alias: "SLF4J", Slf4jImpl.class);
    typeAliasRegistry.registerAlias( alias: "COMMONS_LOGGING", JakartaCommonsLoggingImpl.class);
```

以上步骤完成了TypeHandlerRegistry和TypeAliasRegistry的初始化操作

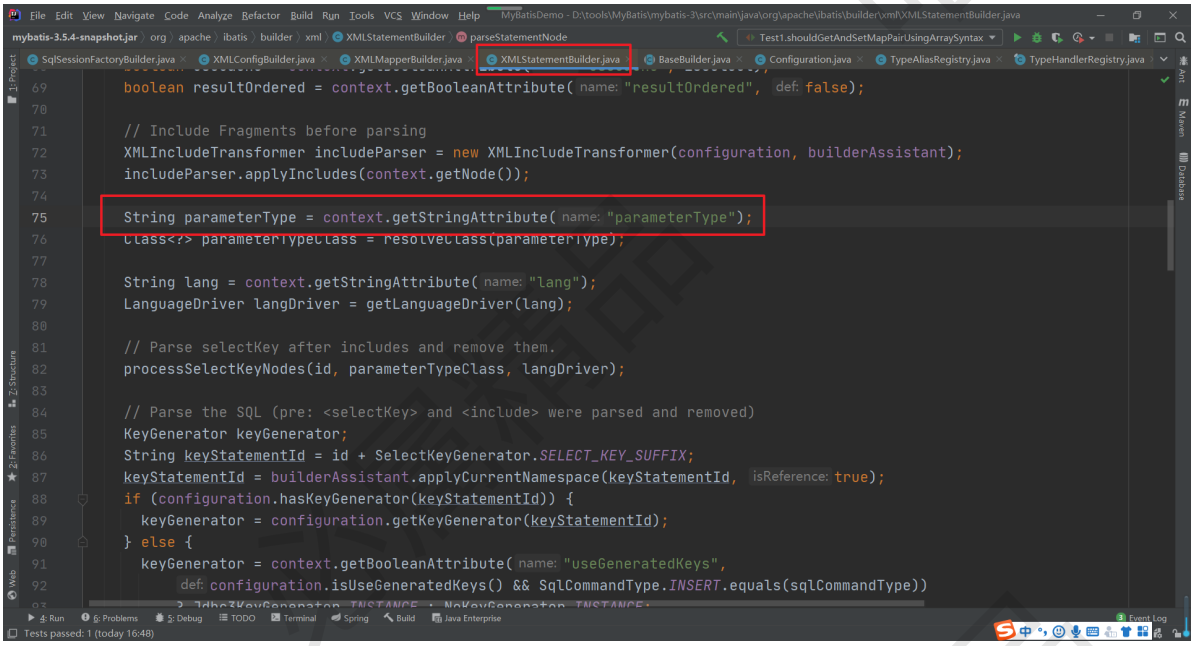
然后在解析全局配置文件时会通过解析<typeAliases>标签和<typeHandlers>标签，可以注册我们添加的别名和TypeHandler。



```
        // 读取文件
        loadCustomVfs(settings);
        // 日志设置
        loadCustomLogImpl(settings);
        // 类型别名
        typeAliasesElement(root.evalNode("typeAliases"));
        // 插件
        pluginElement(root.evalNode("plugins"));
        // 用于创建对象
        objectFactoryElement(root.evalNode("objectFactory"));
        // 用于对对象进行加工
        objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));
        // 反射工具箱
        reflectorFactoryElement(root.evalNode("reflectorFactory"));
        // settings 子标签赋值，默认值就是在这里提供的 >>
        settingsElement(settings);
        // read it after objectFactory and objectWrapperFactory issue #631
        // 创建了数据源 >>
        environmentsElement(root.evalNode("environments"));
        databaseIdProviderElement(root.evalNode("databaseIdProvider"));
        typeHandlerElement(root.evalNode("typeHandlers"));
        // 解析引用的Mapper映射器
        mapperElement(root.evalNode("mappers"));
    } catch (Exception e) {
```

具体解析的两个方法很简单，大家打开源码查看一下就清楚了。

因为我们在全局配置文件中指定了对应的别名，那么我们在映射文件中就可以简写我们的类型了，这样在解析映射文件时，我们同样也是需要做别名的处理的。在XMLStatementBuilder中



这个parameterType就可以是我们定义的别名，然后在 resolveClass中就会做对应的处理

```
  protected <T> Class<? extends T> resolveClass(String alias) {
    if (alias == null) {
      return null;
    }
    try {
      return resolveAlias(alias); // 别名处理
    } catch (Exception e) {
      throw new BuilderException("Error resolving class. Cause: " + e, e);
    }
  }

  protected <T> Class<? extends T> resolveAlias(String alias) {
    return typeAliasRegistry.resolveAlias(alias); // 根据别名查找真实的类型
  }
```

2.6.2 执行SQL语句

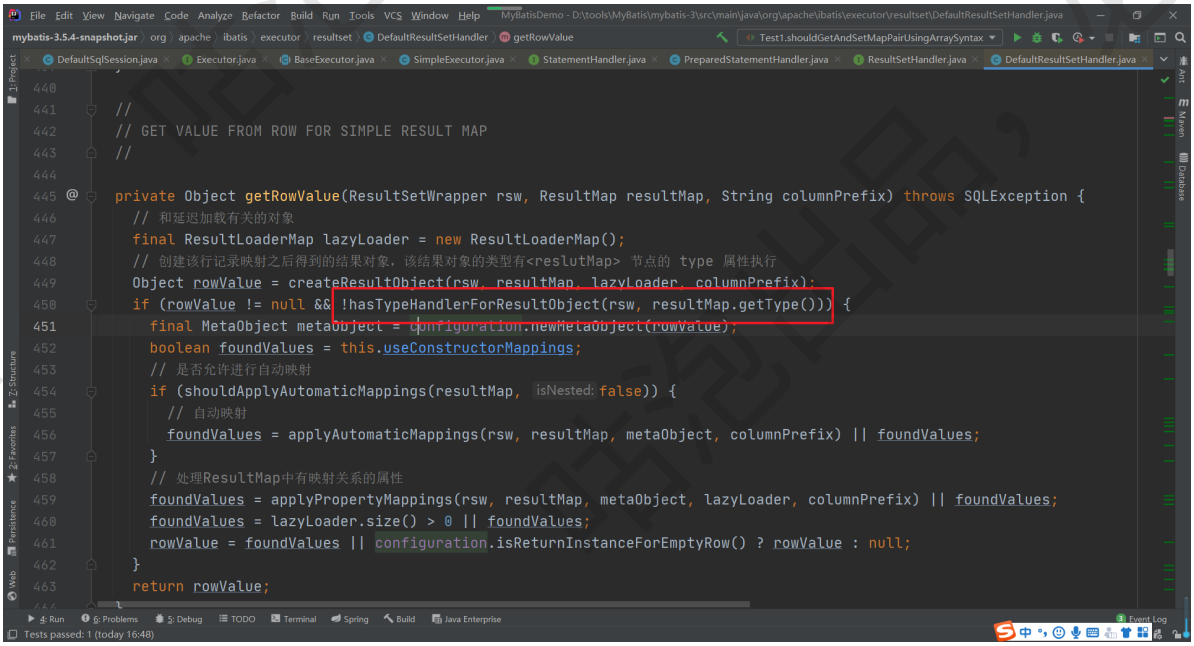
TypeHandler类型处理器使用比较多的地方应该是在给SQL语句中参数绑定值和查询结果和对象中属性映射的地方用到的比较多，

我们首先进入DefaultParameterHandler中看看参数是如何处理的

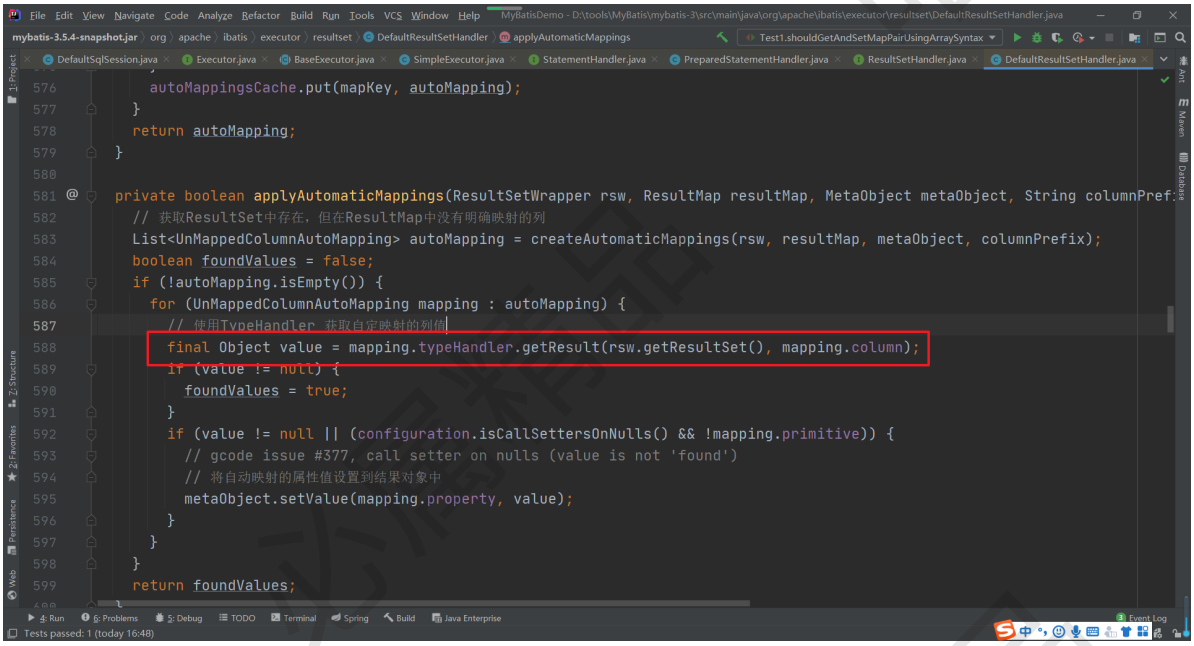
```
  /**
  * 为 SQL 语句中的 ？ 占位符 绑定实参
  */
  @Override
  public void setParameters(PreparedStatement ps) {
    ErrorContext.instance().activity("setting
parameters").object(mappedStatement.getParameterMap().getId());
      // 取出SQL中的参数映射列表
    List<ParameterMapping> parameterMappings = boundSql.getParameterMappings();
// 获取对应的占位符
    if (parameterMappings != null) {
      for (int i = 0; i < parameterMappings.size(); i++) {
        ParameterMapping parameterMapping = parameterMappings.get(i);
```

```
        if (parameterMapping.getMode() != ParameterMode.OUT) { // 过滤掉存储过程中的
输出参数
          Object value;
          String propertyName = parameterMapping.getProperty();
          if (boundSql.hasAdditionalParameter(propertyName)) { // issue #448 ask
first for additional params
            value = boundSql.getAdditionalParameter(propertyName);
          } else if (parameterObject == null) {
            value = null;
          } else if
(typeHandlerRegistry.hasTypeHandler(parameterObject.getClass())) { //
            value = parameterObject;
          } else {
            MetaObject metaObject =
configuration.newMetaObject(parameterObject);
            value = metaObject.getValue(propertyName);
          }
          // 获取 参数类型 对应的 类型处理器
          TypeHandler typeHandler = parameterMapping.getTypeHandler();
          JdbcType jdbcType = parameterMapping.getJdbcType();
          if (value == null && jdbcType == null) {
            jdbcType = configuration.getJdbcTypeForNull();
          }
          try {
            // 通过TypeHandler 处理参数
            typeHandler.setParameter(ps, i + 1, value, jdbcType);
          } catch (TypeException | SQLException e) {
            throw new TypeException("Could not set parameters for mapping: " +
parameterMapping + ". Cause: " + e, e);
          }
        }
      }
    }
  }
```

然后进入到DefaultResultSetHandler中的getRowValue方法中



然后再进入applyAutomaticMappings方法中查看



根据对应的TypeHandler返回对应类型的值。

3.日志模块

首先日志在我们开发过程中占据了一个非常重要的地位，是开发和运维管理之间的桥梁，在Java中的日志框架也非常多，Log4j,Log4j2,Apache Commons Log,java.util.logging,slf4j等，这些工具对外的接口也都不尽相同，为了统一这些工具，MyBatis定义了一套统一的日志接口供上层使用。首先大家对于适配器模式要了解下哦。

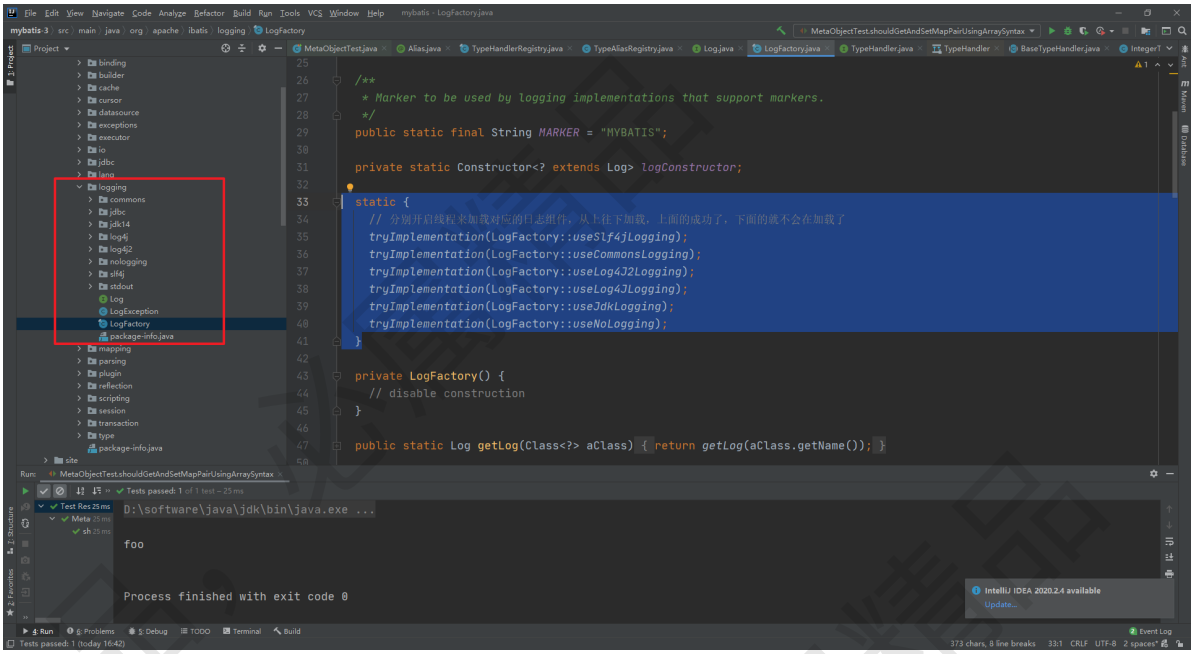
3.1 Log

Log接口中定义了四种日志级别，相比较其他的日志框架的多种日志级别显得非常的精简，但也能够满足大多数常见的使用了

```
public interface Log {

  boolean isDebugEnabled();

  boolean isTraceEnabled();

  void error(String s, Throwable e);

  void error(String s);

  void debug(String s);

  void trace(String s);

  void warn(String s);

}
```

3.2 LogFactory

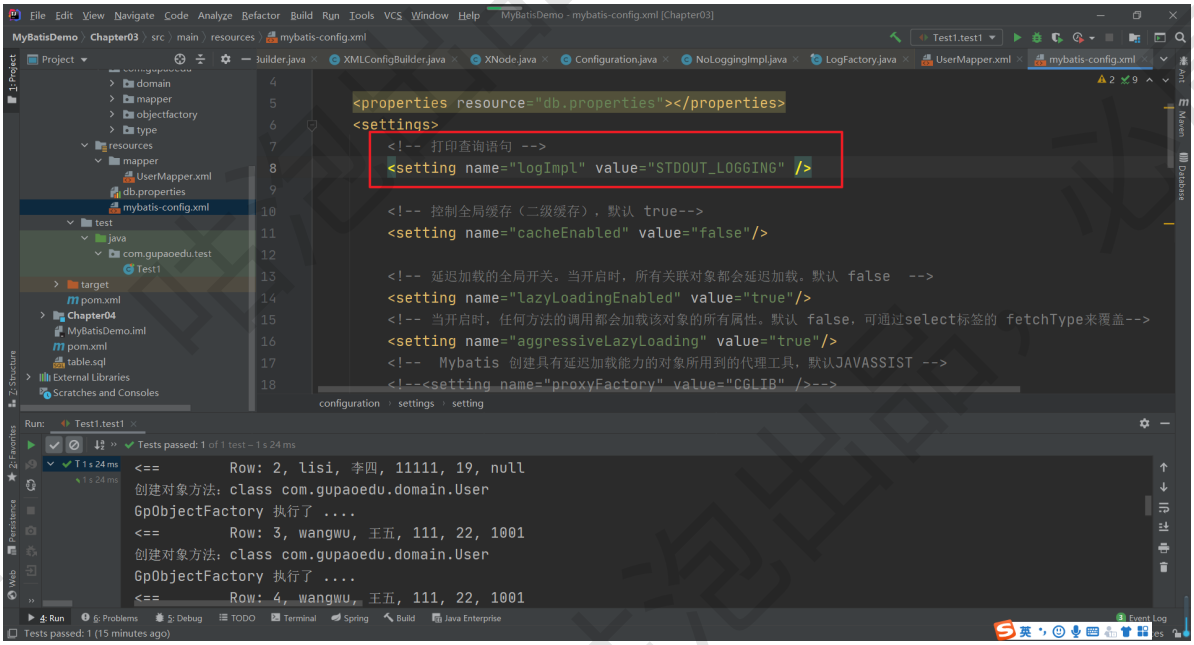
LogFactory工厂类负责创建日志组件适配器，



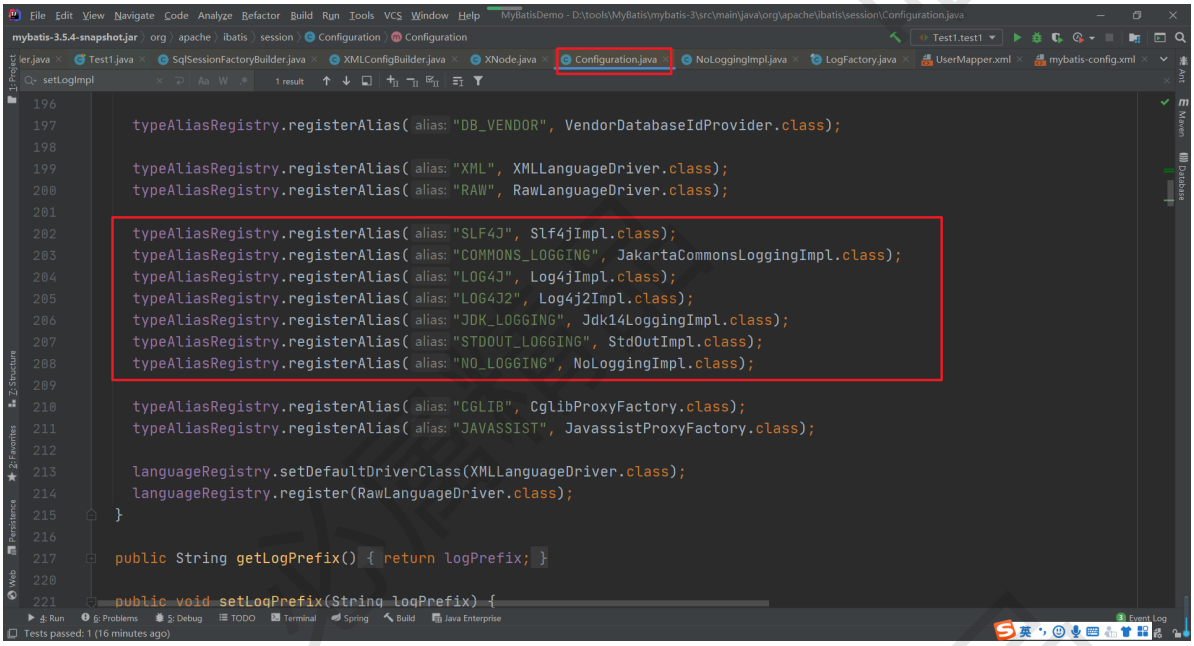
在LogFactory类加载时会执行其静态代码块，其逻辑是按序加载并实例化对应日志组件的适配器，然后使用LogFactory.logConstructor这个静态字段，记录当前使用的第三方日志组件的适配器。具体代码如下，每个方法都比较简单就不一一赘述了。

3.3 日志应用

那么在MyBatis系统启动的时候日志框架是如何选择的呢？首先我们在全局配置文件中我们可以设置对应的日志类型选择

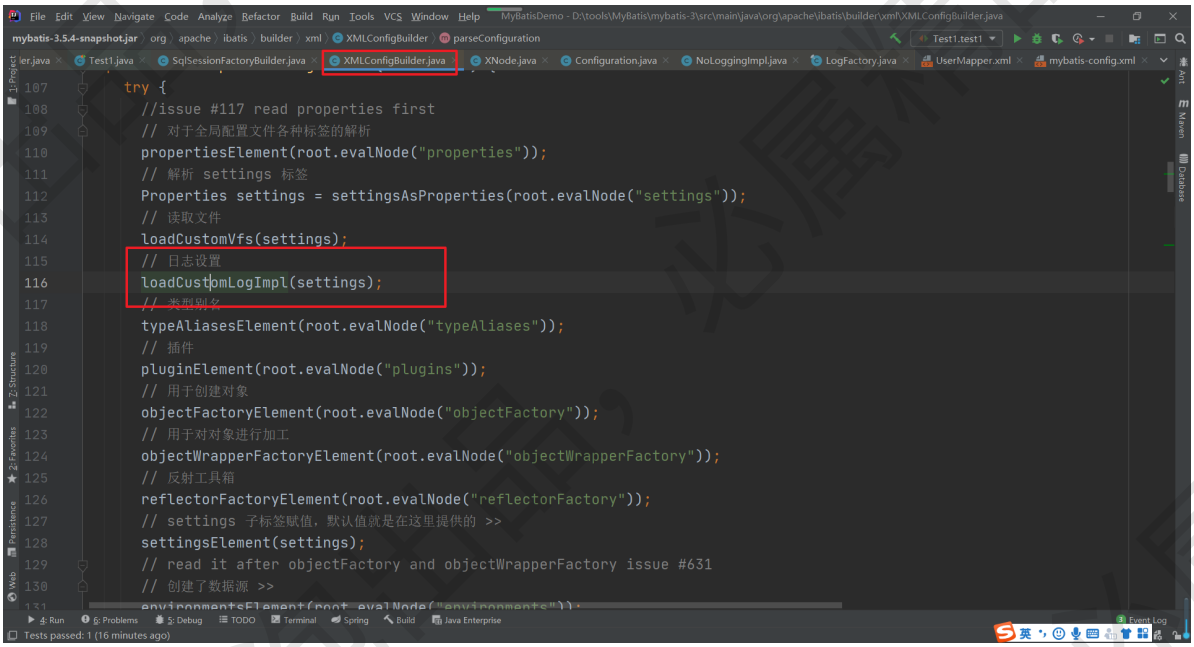


这个"STDOUT_LOGGING"是怎么来的呢？在Configuration的构造方法中其实是设置的各个日志实现的别名的



```
typeAliasRegistry.registerAlias( alias: "DB_VENDOR", VendorDatabaseIdProvider.class);

typeAliasRegistry.registerAlias( alias: "XML", XMLLanguageDriver.class);
typeAliasRegistry.registerAlias( alias: "RAW", RawLanguageDriver.class);

typeAliasRegistry.registerAlias( alias: "SLF4J", Slf4jImpl.class);
typeAliasRegistry.registerAlias( alias: "COMMONS_LOGGING", JakartaCommonsLoggingImpl.class);
typeAliasRegistry.registerAlias( alias: "LOG4J", Log4jImpl.class);
typeAliasRegistry.registerAlias( alias: "LOG4J2", Log4j2Impl.class);
typeAliasRegistry.registerAlias( alias: "JDK_LOGGING", Jdk14LoggingImpl.class);
typeAliasRegistry.registerAlias( alias: "STDOUT_LOGGING", StdOutImpl.class);
typeAliasRegistry.registerAlias( alias: "NO_LOGGING", NoLoggingImpl.class);

typeAliasRegistry.registerAlias( alias: "CGLIB", CglibProxyFactory.class);
typeAliasRegistry.registerAlias( alias: "JAVASSIST", JavassistProxyFactory.class);

languageRegistry.setDefaultDriverClass(XMLLanguageDriver.class);
languageRegistry.register(RawLanguageDriver.class);
}

public String getLogPrefix() { return logPrefix; }

public void setLogPrefix(String logPrefix) {
```

然后在解析全局配置文件的时候就会处理日志的设置



```
try {
  //issue #117 read properties first
  // 对于全局配置文件各种标签的解析
  propertiesElement(root.evalNode("properties"));
  // 解析 settings 标签
  Properties settings = settingsAsProperties(root.evalNode("settings"));
  // 读取文件
  loadCustomVfs(settings);
  // 日志设置
  loadCustomLogImpl(settings);
  // 类型别名
  typeAliasesElement(root.evalNode("typeAliases"));
  // 插件
  pluginElement(root.evalNode("plugins"));
  // 用于创建对象
  objectFactoryElement(root.evalNode("objectFactory"));
  // 用于对对象进行加工
  objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));
  // 反射工具箱
  reflectorFactoryElement(root.evalNode("reflectorFactory"));
  // settings 子标签赋值，默认值就是在这里提供的 >>
  settingsElement(settings);
  // read it after objectFactory and objectWrapperFactory issue #631
  // 创建了数据源 >>
```

进入方法

```
private void loadCustomLogImpl(Properties props) {
  // 获取 logImpl设置的 日志 类型
  Class<? extends Log> logImpl = resolveClass(props.getProperty("logImpl"));
  // 设置日志
  configuration.setLogImpl(logImpl);
}
```

进入setLogImpl方法中

```
public void setLogImpl(Class<? extends Log> logImpl) {
  if (logImpl != null) {
    this.logImpl = logImpl; // 记录日志的类型
    // 设置 适配选择
    LogFactory.useCustomLogging(this.logImpl);
  }
}
```

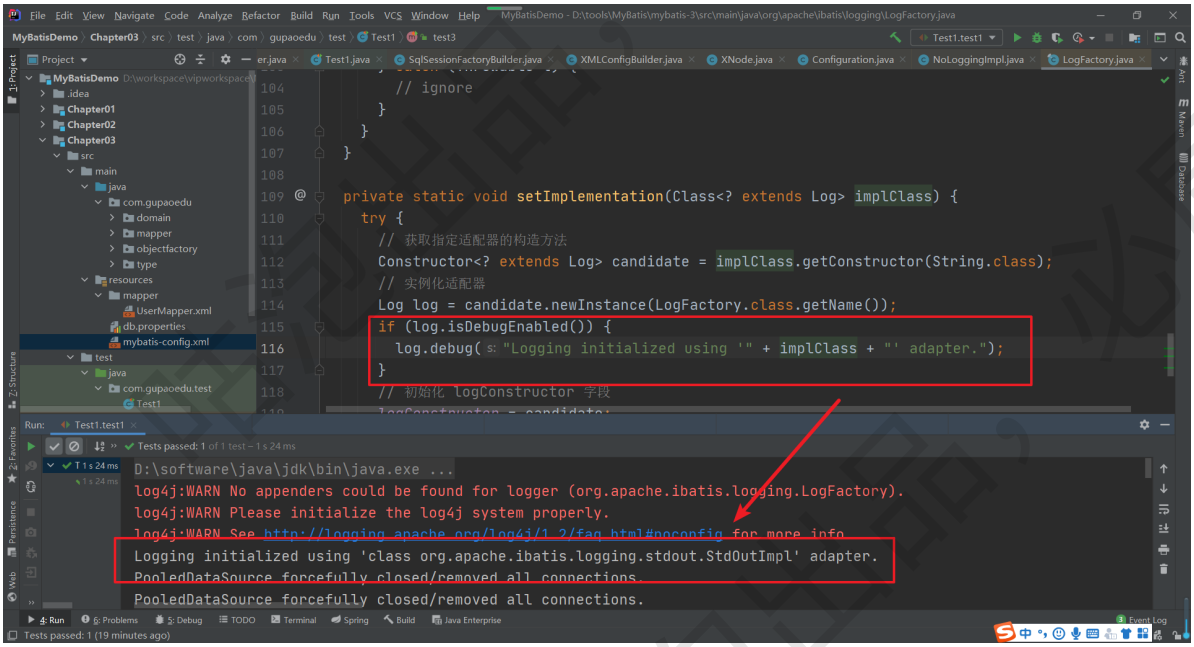
再进入useCustomLogging方法

```
  public static synchronized void useCustomLogging(Class<? extends Log> clazz) {
    setImplementation(clazz);
  }
```

再进入

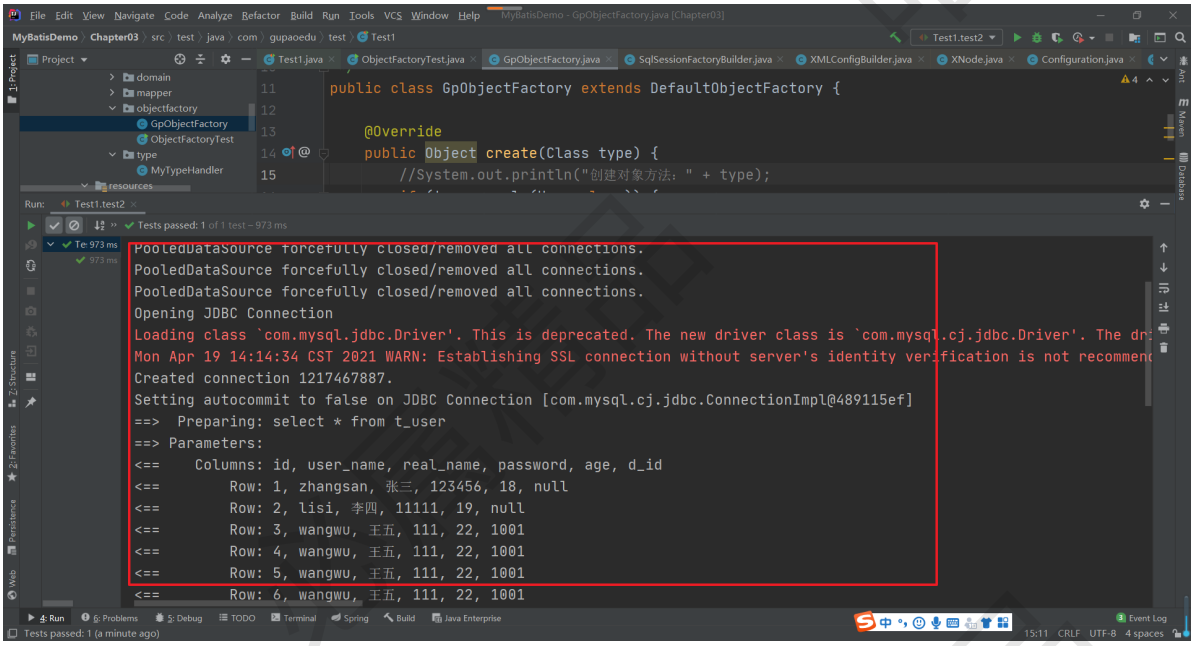
```
  private static void setImplementation(Class<? extends Log> implClass) {
    try {
      // 获取指定适配器的构造方法
      Constructor<? extends Log> candidate =
implClass.getConstructor(String.class);
      // 实例化适配器
      Log log = candidate.newInstance(LogFactory.class.getName());
      if (log.isDebugEnabled()) {
        log.debug("Logging initialized using '" + implClass + "' adapter.");
      }
      // 初始化 logConstructor 字段
      logConstructor = candidate;
    } catch (Throwable t) {
      throw new LogException("Error setting Log implementation.  Cause: " + t,
t);
    }
  }
```

这就关联上了我们前面在LogFactory中看到的代码，启动测试方法看到的日志也和源码中的对应上来了，还有就是我们自己设置的会覆盖掉默认的sl4j日志框架的配置



3.4 JDBC 日志

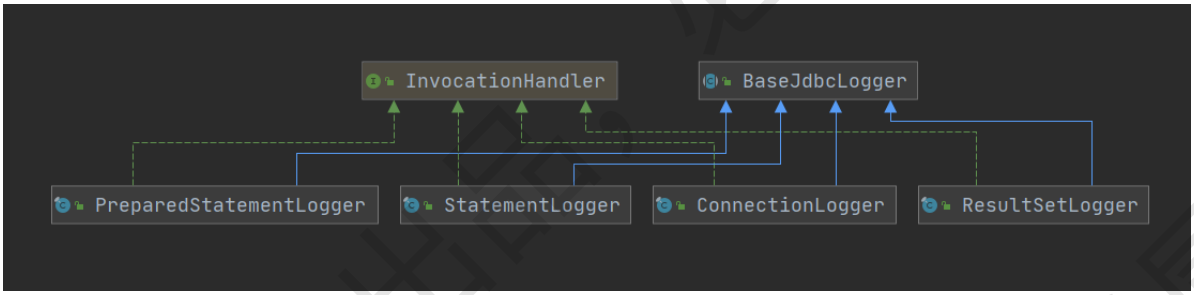
当我们开启了 STDOUT的日志管理后，当我们执行SQL操作时我们发现在控制台中可以打印出相关的日志信息



那这些日志信息是怎么打印出来的呢？原来在MyBatis中的日志模块中包含了一个jdbc包，它并不是将日志信息通过jdbc操作保存到数据库中，而是通过JDK动态代理的方式，将JDBC操作通过指定的日志框架打印出来。下面我们就来看看它是如何实现的。

3.4.1 BaseJdbcLogger

BaseJdbcLogger是一个抽象类，它是jdbc包下其他Logger的父类。继承关系如下



从图中我们也可以看到4个实现都实现了InvocationHandler接口。属性含义如下

```
// 记录 PreparedStatement 接口中定义的常用的set*() 方法
protected static final Set<String> SET_METHODS;
// 记录了 Statement 接口和 PreparedStatement 接口中与执行SQL语句有关的方法
protected static final Set<String> EXECUTE_METHODS = new HashSet<>();

// 记录了PreparedStatement.set*() 方法设置的键值对
private final Map<Object, Object> columnMap = new HashMap<>();
// 记录了PreparedStatement.set*() 方法设置的键 key
private final List<Object> columnNames = new ArrayList<>();
// 记录了PreparedStatement.set*() 方法设置的值 Value
private final List<Object> columnValues = new ArrayList<>();

protected final Log statementLog;// 用于日志输出的Log对象
protected final int queryStack;  // 记录了SQL的层数，用于格式化输出SQL
```

其他几个方法可自行观看

3.4.2 ConnectionLogger

ConnectionLogger的作用是记录数据库连接相关的日志信息，在实现中是创建了一个Connection的代理对象，在每次Connection操作的前后我们都可以实现日志的操作。

```
public final class ConnectionLogger extends BaseJdbcLogger implements
InvocationHandler {

  // 真正的Connection对象
  private final Connection connection;

  private ConnectionLogger(Connection conn, Log statementLog, int queryStack) {
    super(statementLog, queryStack);
    this.connection = conn;
  }

  @Override
  public Object invoke(Object proxy, Method method, Object[] params)
      throws Throwable {
    try {
      // 如果是调用从Object继承过来的方法，就直接调用 toString,hashCode,equals等
      if (Object.class.equals(method.getDeclaringClass())) {
        return method.invoke(this, params);
      }
      // 如果调用的是 prepareStatement方法
      if ("prepareStatement".equals(method.getName())) {
        if (isDebugEnabled()) {
          debug(" Preparing: " + removeBreakingWhitespace((String) params[0]),
true);
        }
        // 创建  PreparedStatement
        PreparedStatement stmt = (PreparedStatement) method.invoke(connection,
params);
        // 然后创建 PreparedStatement 的代理对象 增强
        stmt = PreparedStatementLogger.newInstance(stmt, statementLog,
queryStack);
        return stmt;
        // 同上
      } else if ("prepareCall".equals(method.getName())) {
        if (isDebugEnabled()) {
          debug(" Preparing: " + removeBreakingWhitespace((String) params[0]),
true);
        }
        PreparedStatement stmt = (PreparedStatement) method.invoke(connection,
params);
        stmt = PreparedStatementLogger.newInstance(stmt, statementLog,
queryStack);
        return stmt;
        // 同上
      } else if ("createStatement".equals(method.getName())) {
        Statement stmt = (Statement) method.invoke(connection, params);
        stmt = StatementLogger.newInstance(stmt, statementLog, queryStack);
        return stmt;
      } else {
        return method.invoke(connection, params);
      }
    } catch (Throwable t) {
```

```
      throw ExceptionUtil.unwrapThrowable(t);
    }
  }

  /**
   * Creates a logging version of a connection.
   *
   * @param conn - the original connection
   * @return - the connection with logging
   */
  public static Connection newInstance(Connection conn, Log statementLog, int
queryStack) {
    InvocationHandler handler = new ConnectionLogger(conn, statementLog,
queryStack);
    ClassLoader cl = Connection.class.getClassLoader();
    // 创建了 Connection的 代理对象 目的是 增强 Connection对象 给他添加了日志功能
    return (Connection) Proxy.newProxyInstance(cl, new Class[]
{Connection.class}, handler);
  }

  /**
   * return the wrapped connection.
   *
   * @return the connection
   */
  public Connection getConnection() {
    return connection;
  }

}
```

其他几个xxxxLogger的实现和ConnectionLogger几乎是一样的就不在次赘述了，请自行观看。

3.4.3 应用实现

在实际处理的时候，日志模块是如何工作的，我们来看看。

在我们要执行SQL语句前需要获取Statement对象，而Statement对象是通过Connection获取的，所以我们在SimpleExecutor中就可以看到相关的代码

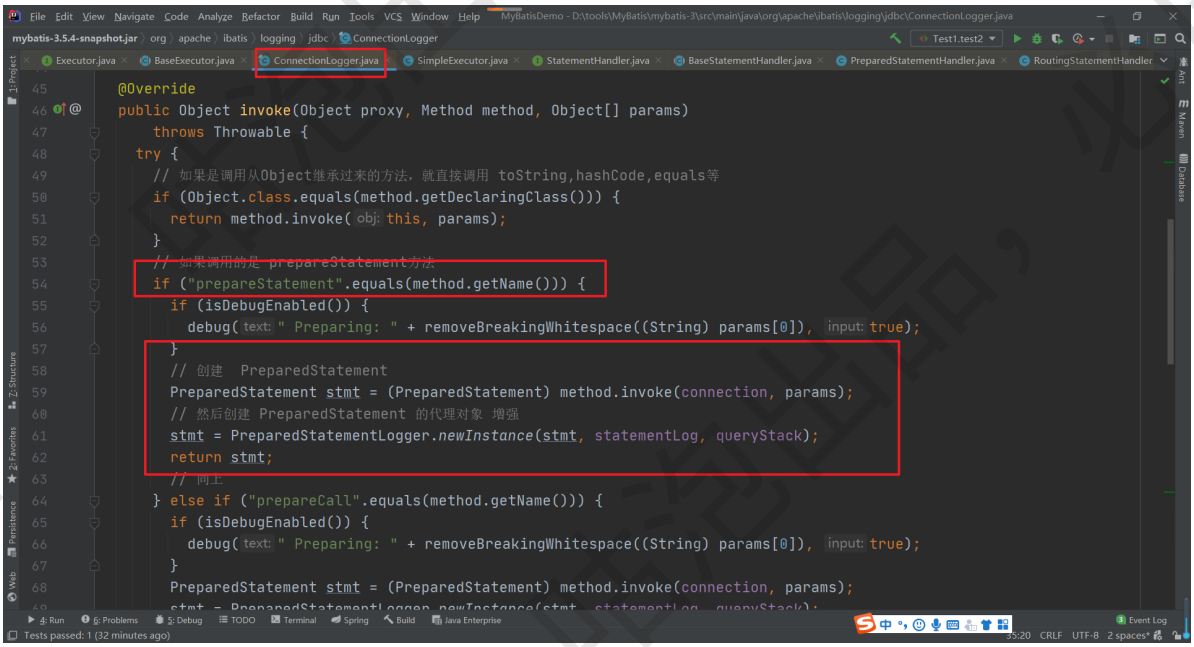
```
  private Statement prepareStatement(StatementHandler handler, Log statementLog)
throws SQLException {
    Statement stmt;
    Connection connection = getConnection(statementLog);
    // 获取 Statement 对象
    stmt = handler.prepare(connection, transaction.getTimeout());
    // 为 Statement 设置参数
    handler.parameterize(stmt);
    return stmt;
  }
```

先进入如到getConnection方法中

```
  protected Connection getConnection(Log statementLog) throws SQLException {
    Connection connection = transaction.getConnection();
    if (statementLog.isDebugEnabled()) {
      // 创建Connection的日志代理对象
      return ConnectionLogger.newInstance(connection, statementLog, queryStack);
    } else {
      return connection;
    }
  }
```

在进入到handler.prepare方法中

```
  @Override
  protected Statement instantiateStatement(Connection connection) throws
SQLException {
    String sql = boundSql.getSql();
    if (mappedStatement.getKeyGenerator() instanceof Jdbc3KeyGenerator) {
      String[] keyColumnNames = mappedStatement.getKeyColumns();
      if (keyColumnNames == null) {
        return connection.prepareStatement(sql,
PreparedStatement.RETURN_GENERATED_KEYS);
      } else {
        // 在执行 prepareStatement 方法的时候会进入进入到ConnectionLogger的invoker方法
中
        return connection.prepareStatement(sql, keyColumnNames);
      }
    } else if (mappedStatement.getResultSetType() == ResultSetType.DEFAULT) {
      return connection.prepareStatement(sql);
    } else {
      return connection.prepareStatement(sql,
mappedStatement.getResultSetType().getValue(), ResultSet.CONCUR_READ_ONLY);
    }
  }
```



在执行sql语句的时候

```
  @Override
  public <E> List<E> query(Statement statement, ResultHandler resultHandler)
throws SQLException {
    PreparedStatement ps = (PreparedStatement) statement;
    // 到了JDBC的流程
    ps.execute(); // 本质上 ps 也是 日志代理对象
    // 处理结果集
    return resultSetHandler.handleResultSets(ps);
  }
```

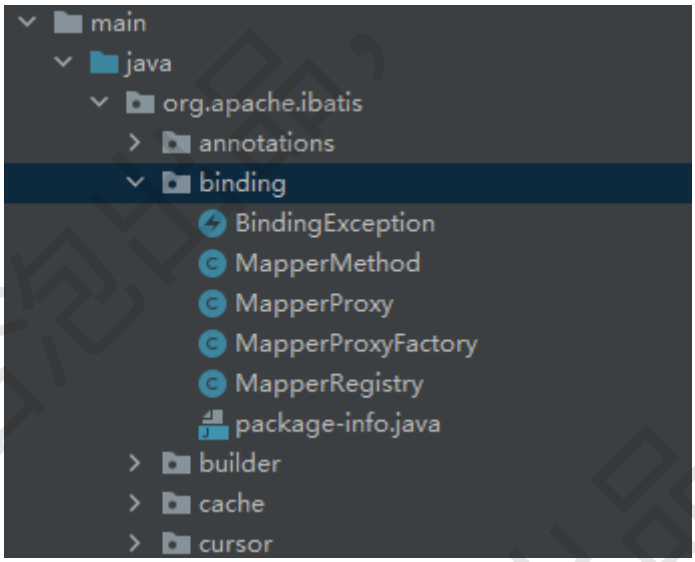
如果是查询操作，后面的ResultSet结果集操作，其他是也通过ResultSetLogger来处理的，前面的清楚了，后面的就很容易的。

4.binding模块

接下来我们看看在org.apache.ibatis.binding包下给我们提供的Binding模块，这个模块在我们前面使用的

```
// 3.根据SqlSessionFactory对象获取SqlSession对象
SqlSession sqlSession = factory.openSession();
// 4.通过SqlSession中提供的 API方法来操作数据库
UserMapper mapper = sqlSession.getMapper(UserMapper.class);
```

这种方式中发挥了很重要的作用，我们先来看这个包中给我们提供的工具类



大家会发现这个包里面提供的工具比较少，就几个，我们先来分别了解下他们的作用，然后在串联起来。

4.1 MapperRegistry

通过名称我们可以看出这显然是一个注册中心，这个注册中是用来保存MapperProxyFactory对象的，所以这个注册器中提供的功能肯定是围绕MapperProxyFactory的添加和获取操作，我们来看看具体的代码逻辑

成员变量

```
  private final Configuration config;
  // 记录 Mapper 接口和 MapperProxyFactory 之间的关系
  private final Map<Class<?>, MapperProxyFactory<?>> knownMappers = new
HashMap<>();
```

addMapper方法

```
  public <T> void addMapper(Class<T> type) {
    if (type.isInterface()) { // 检测 type 是否为接口
      if (hasMapper(type)) { // 检测是否已经加装过该接口
        throw new BindingException("Type " + type + " is already known to the
MapperRegistry.");
      }
      boolean loadCompleted = false;
      try {
        // ! Map<Class<?>, MapperProxyFactory<?>> 存放的是接口类型，和对应的工厂类的关
系
        knownMappers.put(type, new MapperProxyFactory<>(type));
        // It's important that the type is added before the parser is run
        // otherwise the binding may automatically be attempted by the
        // mapper parser. If the type is already known, it won't try.

        // 注册了接口之后，根据接口，开始解析所有方法上的注解，例如 @Select >>
        MapperAnnotationBuilder parser = new MapperAnnotationBuilder(config,
type);
        parser.parse();
        loadCompleted = true;
      } finally {
        if (!loadCompleted) {
          knownMappers.remove(type);
        }
      }
    }
  }
```

getMapper方法

```
  /**
   * 获取Mapper接口对应的代理对象
   */
  public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
    // 获取Mapper接口对应的 MapperProxyFactory 对象
    final MapperProxyFactory<T> mapperProxyFactory = (MapperProxyFactory<T>)
knownMappers.get(type);
    if (mapperProxyFactory == null) {
      throw new BindingException("Type " + type + " is not known to the
MapperRegistry.");
    }
    try {
      return mapperProxyFactory.newInstance(sqlSession);
    } catch (Exception e) {
      throw new BindingException("Error getting mapper instance. Cause: " + e,
e);
    }
  }
```

通过这个方法本质上获取的就是Mapper接口的代理对象。

4.2 MapperProxyFactory

MapperProxyFactory是一个工厂对象，专门负责创建MapperProxy对象。其中核心字段的含义和功能如下：

```
 /**
   * MapperProxyFactory 可以创建 mapperInterface 接口的代理对象
   *     创建的代理对象要实现的接口
   */
  private final Class<T> mapperInterface;
  // 缓存
  private final Map<Method, MapperMethodInvoker> methodCache = new
ConcurrentHashMap<>();

// ....

  /**
   * 创建实现了 mapperInterface 接口的代理对象
   */
  protected T newInstance(MapperProxy<T> mapperProxy) {
    // 1：类加载器:2：被代理类实现的接口、3：实现了 InvocationHandler 的触发管理类
    return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new
Class[] { mapperInterface }, mapperProxy);
  }

  public T newInstance(SqlSession sqlSession) {
    final MapperProxy<T> mapperProxy = new MapperProxy<>(sqlSession,
mapperInterface, methodCache);
    return newInstance(mapperProxy);
  }
```

4.3 MapperProxy

通过MapperProxyFactory创建的MapperProxy是Mapper接口的代理对象，实现了InvocationHandler接口，通过前面讲解的动态代理模式，那么这部分的内容就很简单了。

```
  private final SqlSession sqlSession; // 记录关联的 SqlSession对象
  private final Class<T> mapperInterface; // Mapper接口对应的Class对象
  // 用于缓存MapperMethod对象，key是Mapper接口方法对应的Method对象，value是对应的
MapperMethod对象。‘
  // MapperMethod对象会完成参数转换以及SQL语句的执行
  // 注意：MapperMethod中并不会记录任何状态信息，可以在多线程间共享
  private final Map<Method, MapperMethodInvoker> methodCache;
```

MapperProxy.invoke() 方法是代理对象执行的主要逻辑，实现如下：

```
  @Override
  public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
    try {
      // toString hashCode equals getClass等方法，无需走到执行SQL的流程
      if (Object.class.equals(method.getDeclaringClass())) {
        return method.invoke(this, args);
      } else {
        // 提升获取 mapperMethod 的效率，到 MapperMethodInvoker（内部接口） 的 invoke
        // 普通方法会走到 PlainMethodInvoker（内部类） 的 invoke
        return cachedInvoker(method).invoke(proxy, method, args, sqlSession);
      }
    } catch (Throwable t) {
      throw ExceptionUtil.unwrapThrowable(t);
    }
  }
```

然后在cachedInvoker中主要负责维护methodCache这个缓存集合

```
  private MapperMethodInvoker cachedInvoker(Method method) throws Throwable {
    try {
      // Java8 中 Map 的方法，根据 key 获取值，如果值是 null，则把后面Object 的值赋给 key
      // 如果获取不到，就创建
      // 获取的是 MapperMethodInvoker（接口） 对象，只有一个invoke方法
      // 根据method 去methodCache中获取 如果返回空 则用第二个参数填充
      return methodCache.computeIfAbsent(method, m -> {
        if (m.isDefault()) {
          // 接口的默认方法(Java8)，只要实现接口都会继承接口的默认方法，例如 List.sort()
          try {
            if (privateLookupInMethod == null) {
              return new DefaultMethodInvoker(getMethodHandleJava8(method));
            } else {
              return new DefaultMethodInvoker(getMethodHandleJava9(method));
            }
          } catch (IllegalAccessException | InstantiationException |
InvocationTargetException
              | NoSuchMethodException e) {
            throw new RuntimeException(e);
          }
        } else {
          // 创建了一个 MapperMethod
          return new PlainMethodInvoker(new MapperMethod(mapperInterface,
method, sqlSession.getConfiguration()));
        }
      });
    } catch (RuntimeException re) {
      Throwable cause = re.getCause();
      throw cause == null ? re : cause;
    }
  }
```

4.4 MapperMethod

MapperMethod中封装了Mapper接口中对应方法的信息，以及SQL语句的信息，我们可以把MapperMethod看成是配置文件中定义的SQL语句和Mapper接口的桥梁。

属性和构造方法

```
  // statement id （例如：com.gupaoedu.mapper.BlogMapper.selectBlogById） 和 SQL 类型
  private final SqlCommand command;
  // 方法签名，主要是返回值的类型
  private final MethodSignature method;

  public MapperMethod(Class<?> mapperInterface, Method method, Configuration config) {
    this.command = new SqlCommand(config, mapperInterface, method);
    this.method = new MethodSignature(config, mapperInterface, method);
  }
```

4.4.1 SqlCommand

SqlCommand是MapperMethod中定义的内部类，记录了SQL语句名称以及对应的类型(UNKNOWN,INSERT,UPDATE,DELETE,SELECT,FLUSH)

```
  public static class SqlCommand {

    private final String name; // SQL语句的的名称
    private final SqlCommandType type; // SQL 语句的类型

    public SqlCommand(Configuration configuration, Class<?> mapperInterface, Method method) {
      // 获取方法名称
      final String methodName = method.getName();
      final Class<?> declaringClass = method.getDeclaringClass();
      MappedStatement ms = resolveMappedStatement(mapperInterface, methodName, declaringClass,
          configuration);
      if (ms == null) {
        if (method.getAnnotation(Flush.class) != null) {
          name = null;
          type = SqlCommandType.FLUSH;
        } else {
          throw new BindingException("Invalid bound statement (not found): "
              + mapperInterface.getName() + "." + methodName);
        }
      } else {
        name = ms.getId();
        type = ms.getSqlCommandType();
```

```
        if (type == SqlCommandType.UNKNOWN) {
          throw new BindingException("Unknown execution method for: " + name);
        }
      }
    }

    public String getName() {
      return name;
    }

    public SqlCommandType getType() {
      return type;
    }

    private MappedStatement resolveMappedStatement(Class<?> mapperInterface,
String methodName,
        Class<?> declaringClass, Configuration configuration) {
      // statementId = Mapper接口全路径 + 方法名称 比
如:com.gupaoedu.mapper.UserMapper
      String statementId = mapperInterface.getName() + "." + methodName;
      if (configuration.hasStatement(statementId)) {// 检查是否有该名称的SQL语句
        return configuration.getMappedStatement(statementId);
      } else if (mapperInterface.equals(declaringClass)) {
        return null;
      }
      // 如果Mapper接口还有父类 就递归处理
      for (Class<?> superInterface : mapperInterface.getInterfaces()) {
        if (declaringClass.isAssignableFrom(superInterface)) {
          MappedStatement ms = resolveMappedStatement(superInterface,
methodName,
              declaringClass, configuration);
          if (ms != null) {
            return ms;
          }
        }
      }
      return null;
    }
  }
```

4.4.2 MethodSignature

MethodSignature也是MapperMethod的内部类，在其中封装了Mapper接口中定义的方法相关信息。

```
    private final boolean returnsMany; // 判断返回是否为 Collection类型或者数组类型
    private final boolean returnsMap; // 返回值是否为 Map类型
    private final boolean returnsVoid; // 返回值类型是否为 void
    private final boolean returnsCursor; // 返回值类型是否为 Cursor 类型
    private final boolean returnsOptional; // 返回值类型是否为 Optional 类型
    private final Class<?> returnType; // 返回值类型
    private final String mapKey; // 如果返回值类型为 Map  则 mapKey 记录了作为 key的
列名
    private final Integer resultHandlerIndex; // 用来标记该方法参数列表中
ResultHandler 类型参数的位置
    private final Integer rowBoundsIndex; // 用来标记该方法参数列表中 rowBounds 类型参
数的位置
    private final ParamNameResolver paramNameResolver; // 该方法对应的
ParamNameResolver 对象
```

构造方法中完成了相关信息分初始化操作

```
    /**
     * 方法签名
     * @param configuration
     * @param mapperInterface
     * @param method
     */
    public MethodSignature(Configuration configuration, Class<?>
mapperInterface, Method method) {
      // 获取接口方法的返回类型
      Type resolvedReturnType = TypeParameterResolver.resolveReturnType(method,
mapperInterface);
      if (resolvedReturnType instanceof Class<?>) {
        this.returnType = (Class<?>) resolvedReturnType;
      } else if (resolvedReturnType instanceof ParameterizedType) {
        this.returnType = (Class<?>) ((ParameterizedType)
resolvedReturnType).getRawType();
      } else {
        this.returnType = method.getReturnType();
      }
      this.returnsVoid = void.class.equals(this.returnType);
      this.returnsMany =
configuration.getObjectFactory().isCollection(this.returnType) ||
this.returnType.isArray();
      this.returnsCursor = Cursor.class.equals(this.returnType);
      this.returnsOptional = Optional.class.equals(this.returnType);
      this.mapKey = getMapKey(method);
      this.returnsMap = this.mapKey != null;
      // getUniqueParamIndex 查找指定类型的参数在 参数列表中的位置
      this.rowBoundsIndex = getUniqueParamIndex(method, RowBounds.class);
      this.resultHandlerIndex = getUniqueParamIndex(method,
ResultHandler.class);
      this.paramNameResolver = new ParamNameResolver(configuration, method);
    }
```

getUniqueParamIndex的主要作用是 查找指定类型的参数在参数列表中的位置

```
    /**
```

```
     * 查找指定类型的参数在参数列表中的位置
     * @param method
     * @param paramType
     * @return
     */
    private Integer getUniqueParamIndex(Method method, Class<?> paramType) {
      Integer index = null;
      // 获取对应方法的参数列表
      final Class<?>[] argTypes = method.getParameterTypes();
      // 遍历
      for (int i = 0; i < argTypes.length; i++) {
        // 判断是否是需要查找的类型
        if (paramType.isAssignableFrom(argTypes[i])) {
          // 记录对应类型在参数列表中的位置
          if (index == null) {
            index = i;
          } else {
            // RowBounds 和 ResultHandler 类型的参数只能有一个，不能重复出现
            throw new BindingException(method.getName() + " cannot have multiple
" + paramType.getSimpleName() + " parameters");
          }
        }
      }
      return index;
    }
```

4.4.3 execute方法

最后我们需要来看下再MapperMethod中最核心的方法execute方法，这个方法完成了数据库操作

```
public Object execute(SqlSession sqlSession, Object[] args) {
    Object result;
    switch (command.getType()) { // 根据SQL语句的类型调用SqlSession对应的方法
      case INSERT: {
        // 通过 ParamNameResolver 处理args[] 数组 将用户传入的实参和指定参数名称关联起来
        Object param = method.convertArgsToSqlCommandParam(args);
        // sqlSession.insert(command.getName(), param) 调用SqlSession的insert方法
        // rowCountResult 方法会根据 method 字段中记录的方法的返回值类型对结果进行转换
        result = rowCountResult(sqlSession.insert(command.getName(), param));
        break;
      }
      case UPDATE: {
        Object param = method.convertArgsToSqlCommandParam(args);
        result = rowCountResult(sqlSession.update(command.getName(), param));
        break;
      }
      case DELETE: {
        Object param = method.convertArgsToSqlCommandParam(args);
        result = rowCountResult(sqlSession.delete(command.getName(), param));
        break;
      }
      case SELECT:
        if (method.returnsVoid() && method.hasResultHandler()) {
          // 返回值为空 且 ResultSet通过 ResultHandler处理的方法
          executeWithResultHandler(sqlSession, args);
```

```
          result = null;
        } else if (method.returnsMany()) {
          result = executeForMany(sqlSession, args);
        } else if (method.returnsMap()) {
          result = executeForMap(sqlSession, args);
        } else if (method.returnsCursor()) {
          result = executeForCursor(sqlSession, args);
        } else {
          // 返回值为 单一对象的方法
          Object param = method.convertArgsToSqlCommandParam(args);
          // 普通 select 语句的执行入口 >>
          result = sqlSession.selectOne(command.getName(), param);
          if (method.returnsOptional()
              && (result == null ||
!method.getReturnType().equals(result.getClass()))) {
            result = Optional.ofNullable(result);
          }
        }
        break;
      case FLUSH:
        result = sqlSession.flushStatements();
        break;
      default:
        throw new BindingException("Unknown execution method for: " +
command.getName());
    }
    if (result == null && method.getReturnType().isPrimitive() &&
!method.returnsVoid()) {
      throw new BindingException("Mapper method '" + command.getName()
          + " attempted to return null from a method with a primitive return
type (" + method.getReturnType() + ").");
    }
    return result;
  }
```

在这个方法中所对应一些分支方法都还是比较简单的，可自行查阅。

4.4.4 核心流程串联

解析来我们看看在系统具体操作过程中解析器模块在哪些地方发挥了作用

首先在映射文件加载解析的位置。

```
  public void parse() {
    // 总体上做了两件事情，对于语句的注册和接口的注册
    if (!configuration.isResourceLoaded(resource)) {
      // 1、具体增删改查标签的解析。
      // 一个标签一个MappedStatement。 >>
      configurationElement(parser.evalNode("/mapper"));
      configuration.addLoadedResource(resource);
      // 2、把namespace（接口类型）和工厂类绑定起来，放到一个map。
      // 一个namespace 一个 MapperProxyFactory >>
      bindMapperForNamespace();
    }

    parsePendingResultMaps();
```

```
    parsePendingCacheRefs();
    parsePendingStatements();
  }
```

在bindMapperForNamespace中会完成Mapper接口的注册并调用前面介绍过的addMapper方法

然后就是在我们执行

```
// 4.通过SqlSession中提供的 API方法来操作数据库
UserMapper mapper = sqlSession.getMapper(UserMapper.class);
List<User> list = mapper.selectUserList();
```

这两行代码的内部逻辑，首先看下getMapper方法

```
  @Override
  public <T> T getMapper(Class<T> type) {
    return configuration.getMapper(type, this);
  }
```

继续

```
  public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
    // mapperRegistry中注册的有Mapper的相关信息 在解析映射文件时 调用过addMapper方法
    return mapperRegistry.getMapper(type, sqlSession);
  }
```

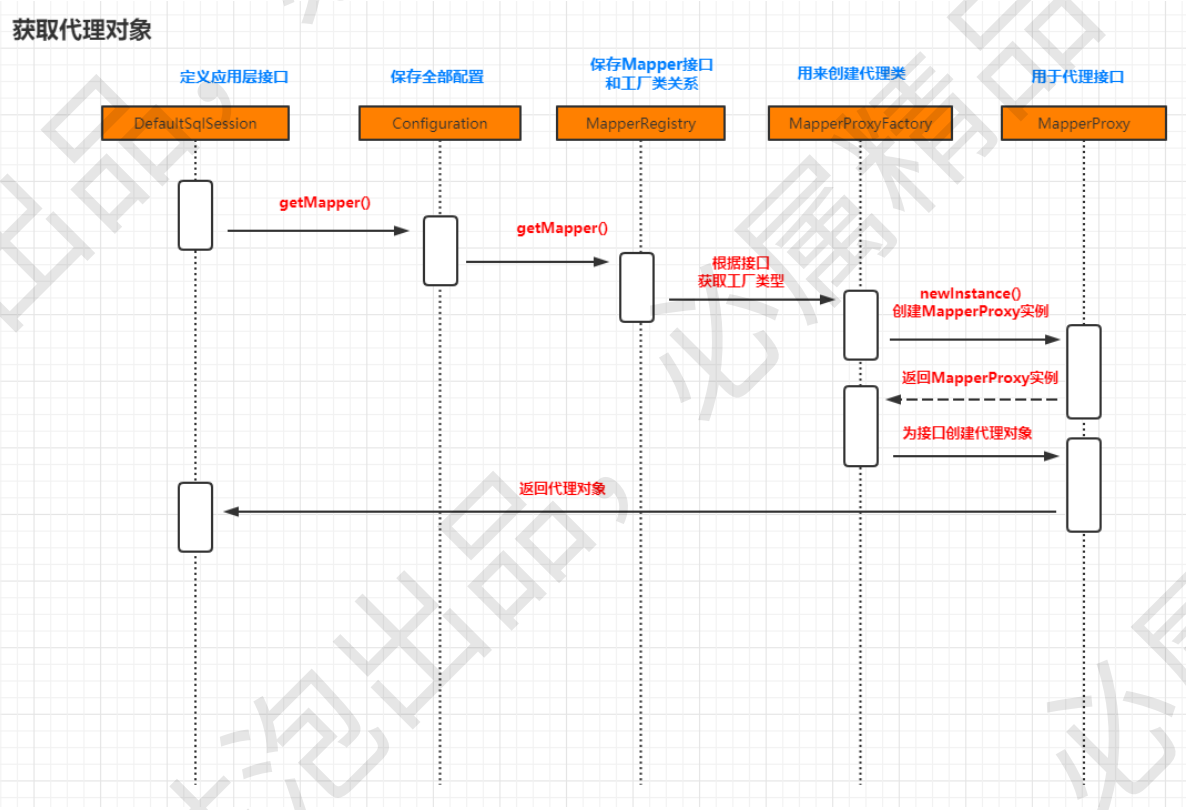
然后就是从MapperRegistry中获取对应的MapperProxyFactory对象。

```
  /**
   * 获取Mapper接口对应的代理对象
   */
  public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
    // 获取Mapper接口对应的 MapperProxyFactory 对象
    final MapperProxyFactory<T> mapperProxyFactory = (MapperProxyFactory<T>)
knownMappers.get(type);
    if (mapperProxyFactory == null) {
      throw new BindingException("Type " + type + " is not known to the
MapperRegistry.");
    }
    try {
      return mapperProxyFactory.newInstance(sqlSession);
    } catch (Exception e) {
      throw new BindingException("Error getting mapper instance. Cause: " + e,
e);
    }
  }
```

然后根据MapperProxyFactory对象获取Mapper接口对应的代理对象。

```
  /**
   * 创建实现了 mapperInterface 接口的代理对象
   */
  protected T newInstance(MapperProxy<T> mapperProxy) {
    // 1：类加载器;2：被代理类实现的接口、3：实现了 InvocationHandler 的触发管理类
    return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new
Class[] { mapperInterface }, mapperProxy);
  }

  public T newInstance(SqlSession sqlSession) {
    final MapperProxy<T> mapperProxy = new MapperProxy<>(sqlSession,
mapperInterface, methodCache);
    return newInstance(mapperProxy);
  }
```

上面简单的时序图为



然后我们再来看下调用代理对象中的方法执行的顺序

```
List<User> list = mapper.selectUserList();
```

会进入MapperProxy的Invoker方法中

```
  @Override
  public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
    try {
      // toString hashCode equals getClass等方法，无需走到执行SQL的流程
      if (Object.class.equals(method.getDeclaringClass())) {
        return method.invoke(this, args);
      } else {
```

```
        // 提升获取 mapperMethod 的效率，到 MapperMethodInvoker（内部接口） 的 invoke
        // 普通方法会走到 PlainMethodInvoker（内部类） 的 invoke
        return cachedInvoker(method).invoke(proxy, method, args, sqlSession);
      }
    } catch (Throwable t) {
      throw ExceptionUtil.unwrapThrowable(t);
    }
  }
```

然后进入PlainMethodInvoker中的invoke方法

```
    @Override
    public Object invoke(Object proxy, Method method, Object[] args, SqlSession
sqlSession) throws Throwable {
      // SQL执行的真正起点
      return mapperMethod.execute(sqlSession, args);
    }
```

然后会进入到 MapperMethod的execute方法中

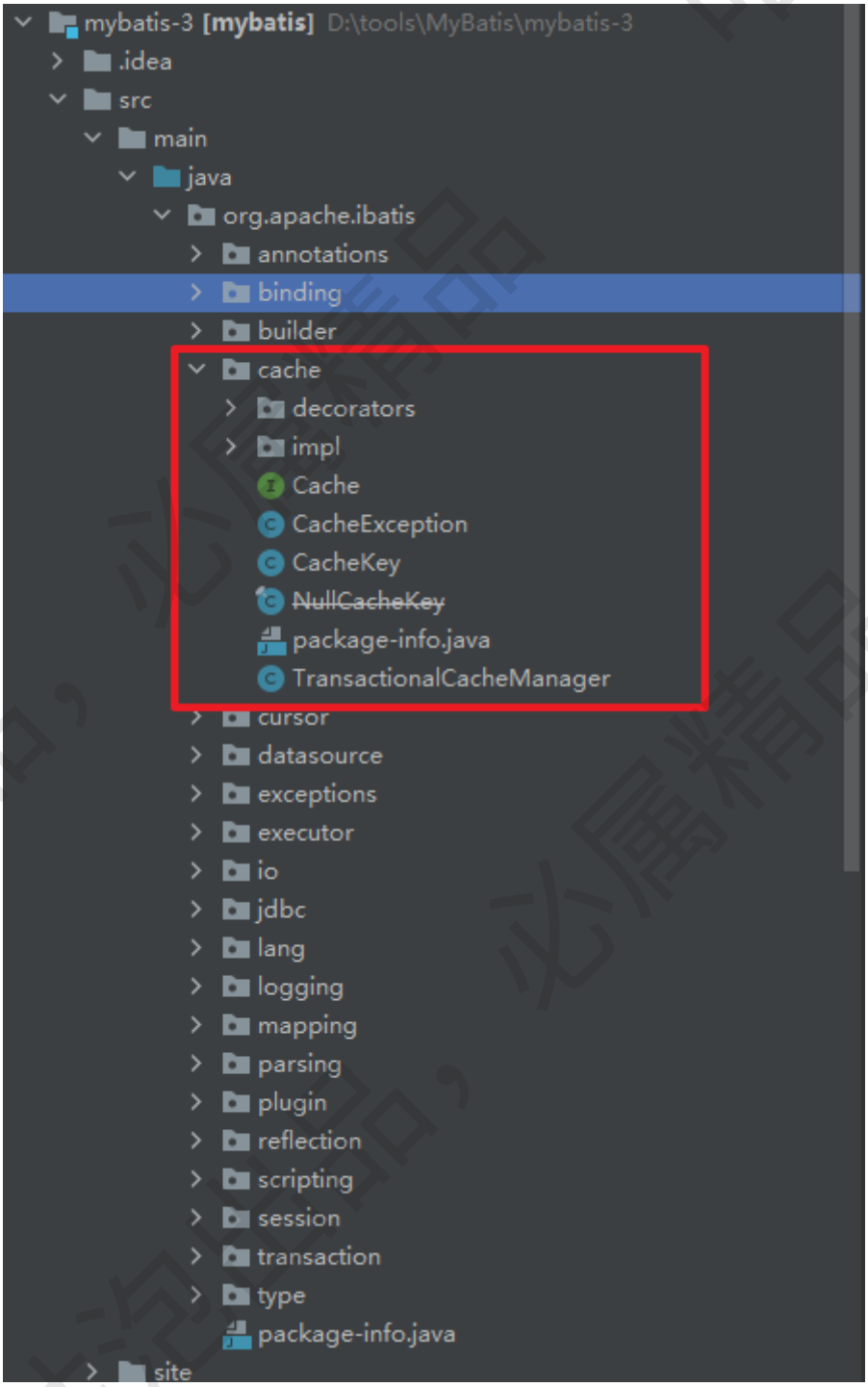
```
  public Object execute(SqlSession sqlSession, Object[] args) {
    Object result;
    switch (command.getType()) { // 根据SQL语句的类型调用SqlSession对应的方法
      case INSERT: {
        // 通过 ParamNameResolver 处理args[] 数组 将用户传入的实参和指定参数名称关联起来
        Object param = method.convertArgsToSqlCommandParam(args);
        // sqlSession.insert(command.getName(), param) 调用SqlSession的insert方法
        // rowCountResult 方法会根据 method 字段中记录的方法的返回值类型对结果进行转换
        result = rowCountResult(sqlSession.insert(command.getName(), param));
        break;
      }
      case UPDATE: {
        Object param = method.convertArgsToSqlCommandParam(args);
        result = rowCountResult(sqlSession.update(command.getName(), param));
        break;
      }
      case DELETE: {
        Object param = method.convertArgsToSqlCommandParam(args);
        result = rowCountResult(sqlSession.delete(command.getName(), param));
        break;
      }
      case SELECT:
        if (method.returnsVoid() && method.hasResultHandler()) {
          // 返回值为空 且 ResultSet通过 ResultHandler处理的方法
          executeWithResultHandler(sqlSession, args);
          result = null;
        } else if (method.returnsMany()) {
          result = executeForMany(sqlSession, args);
        } else if (method.returnsMap()) {
          result = executeForMap(sqlSession, args);
        } else if (method.returnsCursor()) {
          result = executeForCursor(sqlSession, args);
        } else {
          // 返回值为 单一对象的方法
          Object param = method.convertArgsToSqlCommandParam(args);
          // 普通 select 语句的执行入口 >>
```

```
          result = sqlSession.selectOne(command.getName(), param);
          if (method.returnsOptional()
              && (result == null ||
!method.getReturnType().equals(result.getClass()))) {
            result = Optional.ofNullable(result);
          }
        }
        break;
      case FLUSH:
        result = sqlSession.flushStatements();
        break;
      default:
        throw new BindingException("Unknown execution method for: " +
command.getName());
    }
    if (result == null && method.getReturnType().isPrimitive() &&
!method.returnsVoid()) {
      throw new BindingException("Mapper method '" + command.getName()
          + " attempted to return null from a method with a primitive return
type (" + method.getReturnType() + ").");
    }
    return result;
  }
```

之后就会根据对应的SQL类型而调用SqlSession中对应的方法来执行操作

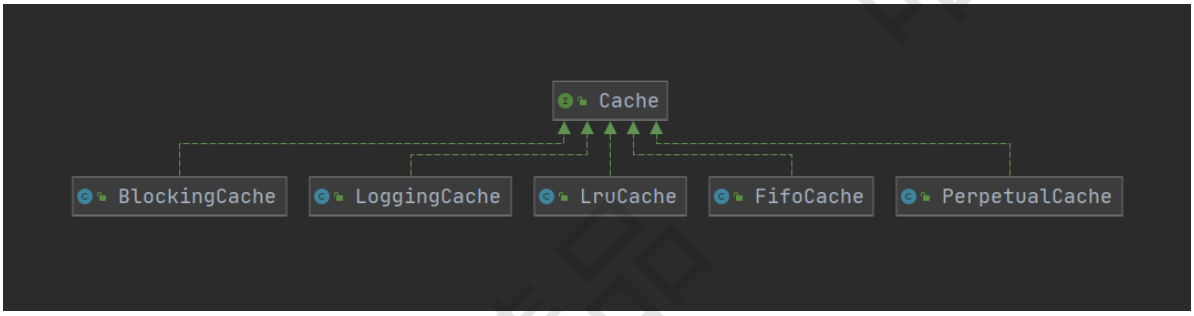
5. 缓存模块

MyBatis作为一个强大的持久层框架，缓存是其必不可少的功能之一，Mybatis中的缓存分为一级缓存和二级缓存。但本质上是一样的，都是使用Cache接口实现的。缓存位于 org.apache.ibatis.cache包下。



通过结构我们能够发现Cache其实使用到了装饰器模式来实现缓存的处理。首先大家需要先回顾下装饰器模式的相关内容哦。我们先来看看Cache中的基础类的API

```
// 煎饼加鸡蛋加香肠
“装饰者模式（Decorator Pattern）是指在不改变原有对象的基础之上，将功能附加到对象上，提供了比继承更有弹性的替代方案（扩展原有对象的功能）。”
```



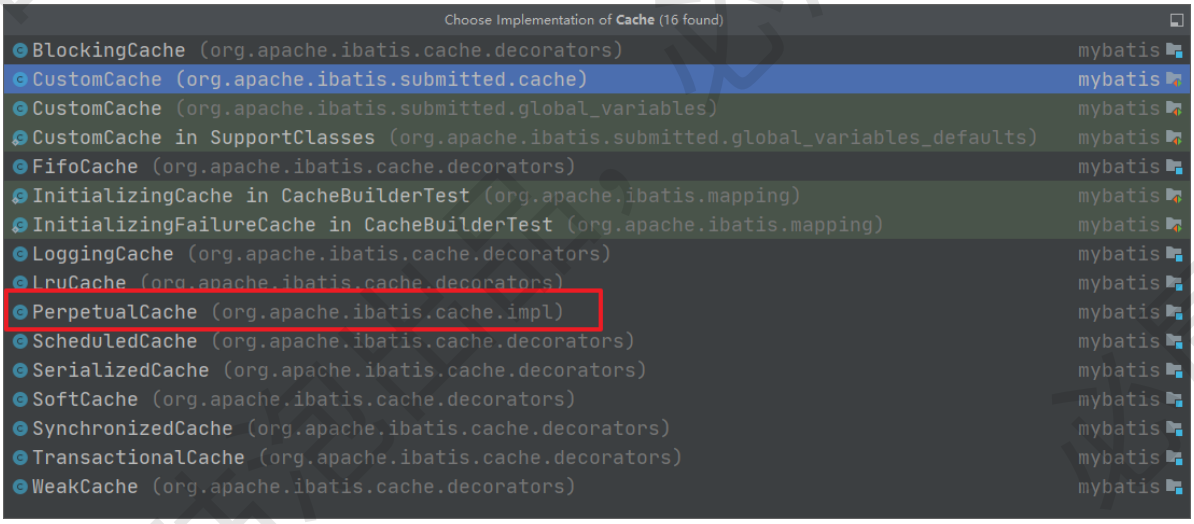
5.1 Cache接口

Cache接口是缓存模块中最核心的接口，它定义了所有缓存的基本行为，Cache接口的定义如下:

```
public interface Cache {

  /**
   * 缓存对象的 ID
   * @return The identifier of this cache
   */
  String getId();

  /**
   * 向缓存中添加数据，一般情况下 key是CacheKey  value是查询结果
   * @param key Can be any object but usually it is a {@link CacheKey}
   * @param value The result of a select.
   */
  void putObject(Object key, Object value);

  /**
   * 根据指定的key，在缓存中查找对应的结果对象
   * @param key The key
   * @return The object stored in the cache.
   */
  Object getObject(Object key);

  /**
   * As of 3.3.0 this method is only called during a rollback
   * for any previous value that was missing in the cache.
   * This lets any blocking cache to release the lock that
   * may have previously put on the key.
   * A blocking cache puts a lock when a value is null
   * and releases it when the value is back again.
   * This way other threads will wait for the value to be
   * available instead of hitting the database.
   *   删除key对应的缓存数据
   *
   * @param key The key
   * @return Not used
   */
  Object removeObject(Object key);

  /**
   * Clears this cache instance.
   * 清空缓存
   */
```

```
  void clear();

  /**
   * Optional. This method is not called by the core.
   * 缓存的个数。
   * @return The number of elements stored in the cache (not its capacity).
   */
  int getSize();

  /**
   * Optional. As of 3.2.6 this method is no longer called by the core.
   * <p>
   * Any locking needed by the cache must be provided internally by the cache
provider.
   *  获取读写锁
   * @return A ReadWriteLock
   */
  default ReadWriteLock getReadWriteLock() {
    return null;
  }

}
```

Cache接口的实现类很多，但是大部分都是装饰器，只有PerpetualCache提供了Cache接口的基本实现。



5.2 PerpetualCache

PerpetualCache在缓存模块中扮演了ConcreteComponent的角色，其实现比较简单，底层使用HashMap记录缓存项，具体的实现如下：

```
/**
 * 在装饰器模式用 用来被装饰的对象
 * 缓存中的  基本缓存处理的实现
 * 其实就是一个 HashMap 的基本操作
 * @author Clinton Begin
 */
public class PerpetualCache implements Cache {
```

```java
private final String id; // Cache 对象的唯一标识

// 用于记录缓存的Map对象
private final Map<Object, Object> cache = new HashMap<>();

public PerpetualCache(String id) {
  this.id = id;
}

@Override
public String getId() {
  return id;
}

@Override
public int getSize() {
  return cache.size();
}

@Override
public void putObject(Object key, Object value) {
  cache.put(key, value);
}

@Override
public Object getObject(Object key) {
  return cache.get(key);
}

@Override
public Object removeObject(Object key) {
  return cache.remove(key);
}

@Override
public void clear() {
  cache.clear();
}

@Override
public boolean equals(Object o) {
  if (getId() == null) {
    throw new CacheException("Cache instances require an ID.");
  }
  if (this == o) {
    return true;
  }
  if (!(o instanceof Cache)) {
    return false;
  }

  Cache otherCache = (Cache) o;
  // 只关心ID
  return getId().equals(otherCache.getId());
}

@Override
```

```
  public int hashCode() {
    if (getId() == null) {
      throw new CacheException("Cache instances require an ID.");
    }
    // 只关心ID
    return getId().hashCode();
  }

}
```

然后我们可以来看看cache.decorators包下提供的装饰器。他们都实现了Cache接口。这些装饰器都在PerpetualCache的基础上提供了一些额外的功能，通过多个组合实现一些特殊的需求。

5.3 BlockingCache

通过名称我们能看出来是一个阻塞同步的缓存，它保证只有一个线程到缓存中查找指定的key对应的数据。

```
public class BlockingCache implements Cache {

  private long timeout; // 阻塞超时时长
  private final Cache delegate; // 被装饰的底层 Cache 对象
  // 每个key 都有对象的 ReentrantLock 对象
  private final ConcurrentHashMap<Object, ReentrantLock> locks;

  public BlockingCache(Cache delegate) {
    // 被装饰的 Cache 对象
    this.delegate = delegate;
    this.locks = new ConcurrentHashMap<>();
  }

  @Override
  public String getId() {
    return delegate.getId();
  }

  @Override
  public int getSize() {
    return delegate.getSize();
  }

  @Override
  public void putObject(Object key, Object value) {
    try {
      // 执行 被装饰的 Cache 中的方法
      delegate.putObject(key, value);
    } finally {
      // 释放锁
      releaseLock(key);
    }
  }

  @Override
  public Object getObject(Object key) {
```

```
    acquireLock(key); // 获取锁
    Object value = delegate.getObject(key); // 获取缓存数据
    if (value != null) { // 有数据就释放掉锁，否则继续持有锁
      releaseLock(key);
    }
    return value;
  }

  @Override
  public Object removeObject(Object key) {
    // despite of its name, this method is called only to release locks
    releaseLock(key);
    return null;
  }

  @Override
  public void clear() {
    delegate.clear();
  }

  private ReentrantLock getLockForKey(Object key) {
    return locks.computeIfAbsent(key, k -> new ReentrantLock());
  }

  private void acquireLock(Object key) {
    Lock lock = getLockForKey(key);
    if (timeout > 0) {
      try {
        boolean acquired = lock.tryLock(timeout, TimeUnit.MILLISECONDS);
        if (!acquired) {
          throw new CacheException("Couldn't get a lock in " + timeout + " for
the key " +  key + " at the cache " + delegate.getId());
        }
      } catch (InterruptedException e) {
        throw new CacheException("Got interrupted while trying to acquire lock
for key " + key, e);
      }
    } else {
      lock.lock();
    }
  }

  private void releaseLock(Object key) {
    ReentrantLock lock = locks.get(key);
    if (lock.isHeldByCurrentThread()) {
      lock.unlock();
    }
  }

  public long getTimeout() {
    return timeout;
  }

  public void setTimeout(long timeout) {
    this.timeout = timeout;
  }
}
```

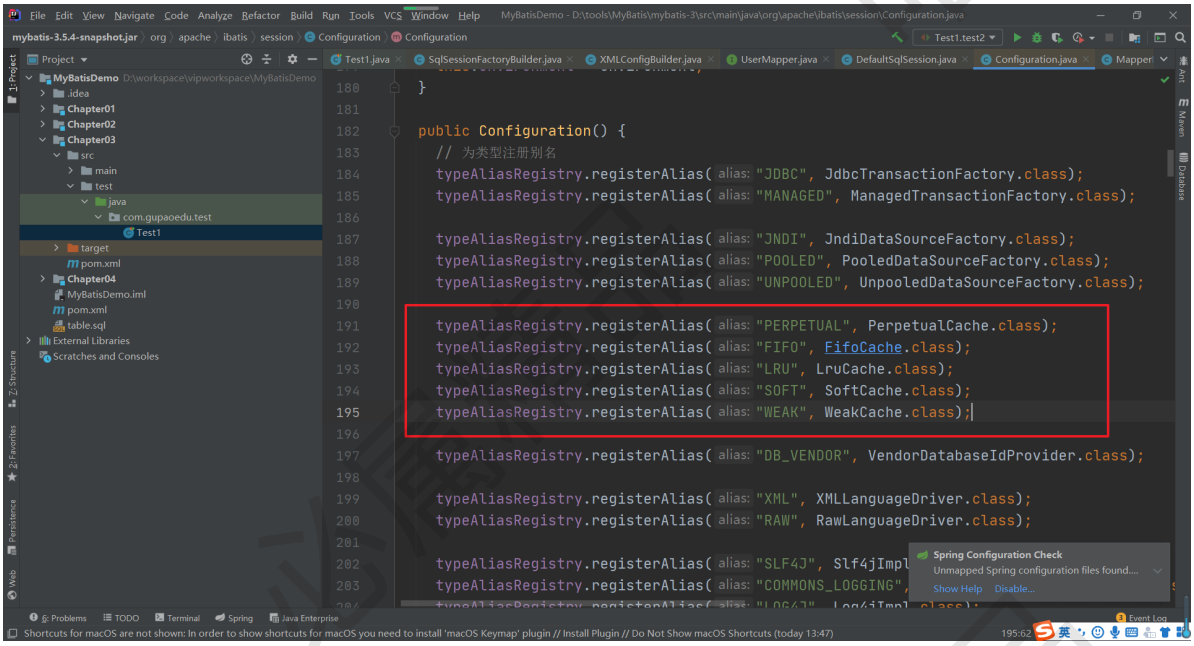
通过源码我们能够发现，BlockingCache本质上就是在我们操作缓存数据的前后通过ReentrantLock对象来实现了加锁和解锁操作。其他的具体实现类，大家可以自行查阅

缓存实现类	描述	作用	装饰条件
基本缓存	缓存基本实现类	默认是PerpetualCache，也可以自定义比如RedisCache、EhCache等，具备基本功能的缓存类	无
LruCache	LRU策略的缓存	当缓存到达上限时候，删除最近最少使用的缓存（Least Recently Use）	eviction="LRU"（默认）
FifoCache	FIFO策略的缓存	当缓存到达上限时候，删除最先入队的缓存	eviction="FIFO"
SoftCacheWeakCache	带清理策略的缓存	通过JVM的软引用和弱引用来实现缓存，当JVM内存不足时，会自动清理掉这些缓存，基于SoftReference和WeakReference	eviction="SOFT"eviction="WEAK"
LoggingCache	带日志功能的缓存	比如：输出缓存命中率	基本
SynchronizedCache	同步缓存	基于synchronized关键字实现，解决并发问题	基本
BlockingCache	阻塞缓存	通过在get/put方式中加锁，保证只有一个线程操作缓存，基于Java重入锁实现	blocking=true
SerializedCache	支持序列化的缓存	将对象序列化以后存到缓存中，取出时反序列化	readOnly=false（默认）
ScheduledCache	定时调度的缓存	在进行get/put/remove/getSize等操作前，判断缓存时间是否超过了设置的最长缓存时间（默认是一小时），如果是则清空缓存--即每隔一段时间清空一次缓存	flushInterval不为空
TransactionalCache	事务缓存	在二级缓存中使用，可一次存入多个缓存，移除多个缓存	在TransactionalCacheManager中用Map维护对应关系

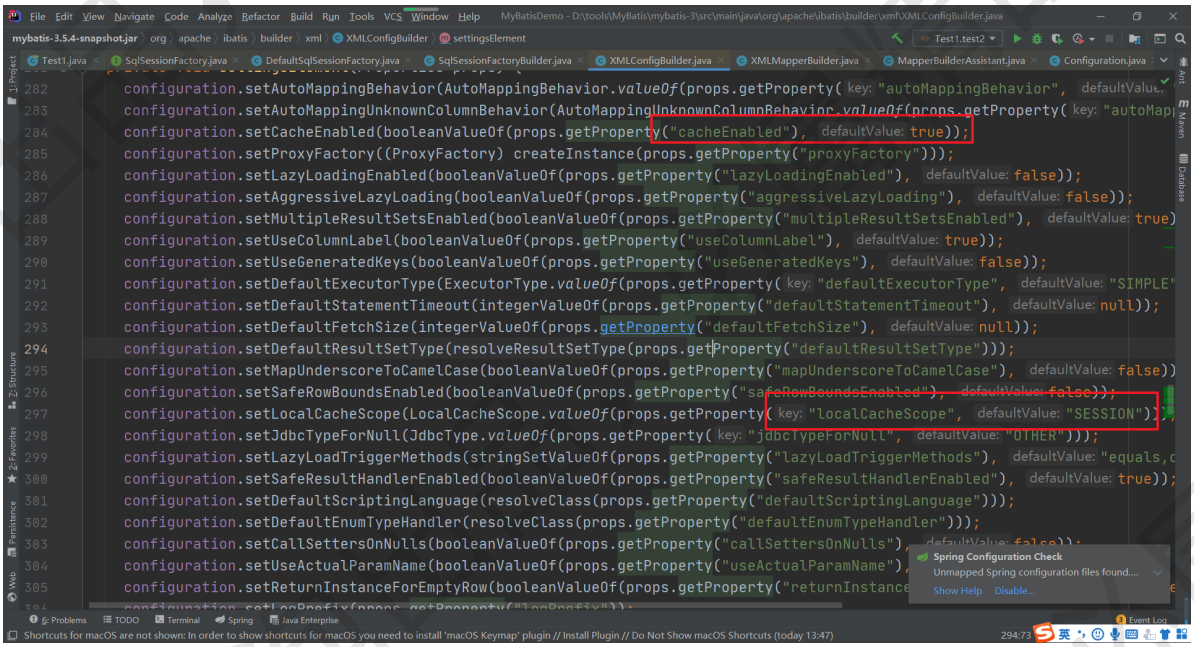
5.4 缓存的应用

5.4.1 缓存对应的初始化

在Configuration初始化的时候会为我们的各种Cache实现注册对应的别名

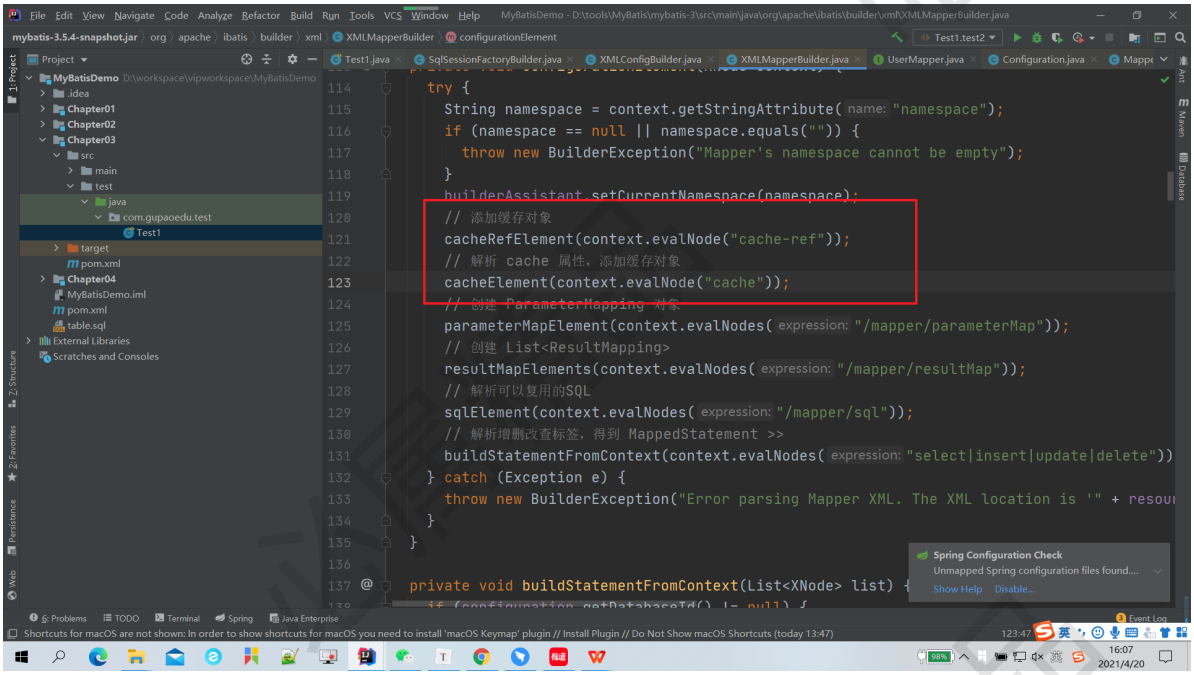


在解析settings标签的时候，设置的默认值有如下



cacheEnabled默认为true，localCacheScope默认为 SESSION

在解析映射文件的时候会解析我们相关的cache标签



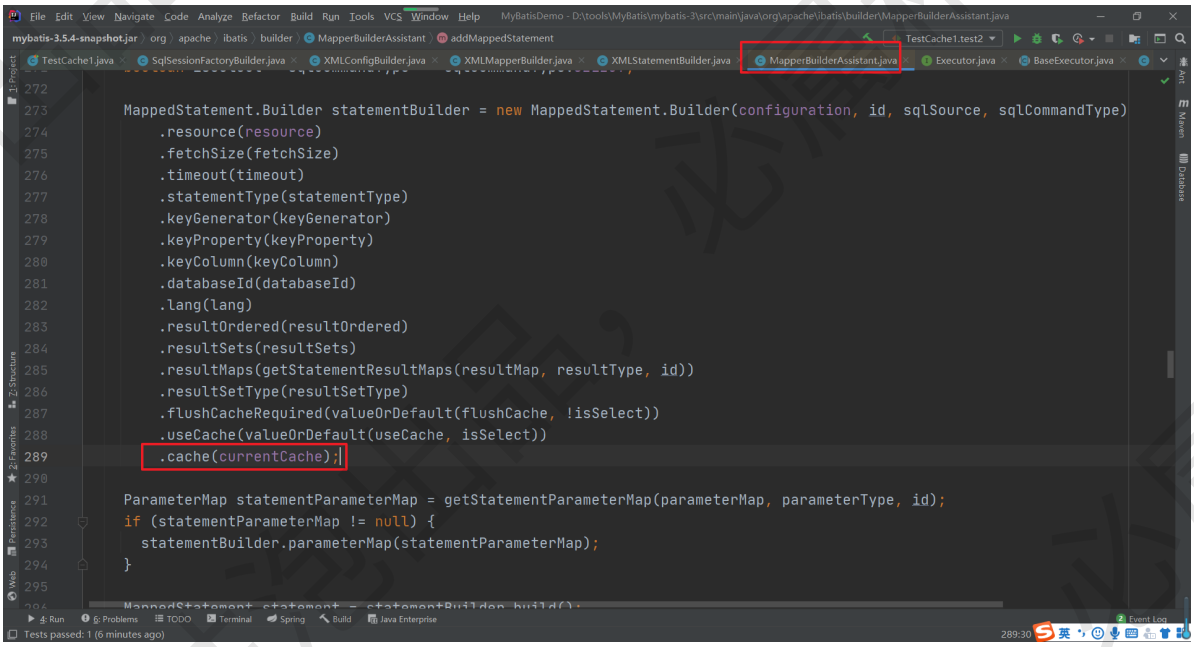
然后解析映射文件的cache标签后会在Configuration对象中添加对应的数据在

```
  private void cacheElement(XNode context) {
    // 只有 cache 标签不为空才解析
    if (context != null) {
      String type = context.getStringAttribute("type", "PERPETUAL");
      Class<? extends Cache> typeClass = typeAliasRegistry.resolveAlias(type);
      String eviction = context.getStringAttribute("eviction", "LRU");
      Class<? extends Cache> evictionClass =
typeAliasRegistry.resolveAlias(eviction);
      Long flushInterval = context.getLongAttribute("flushInterval");
      Integer size = context.getIntAttribute("size");
      boolean readWrite = !context.getBooleanAttribute("readOnly", false);
      boolean blocking = context.getBooleanAttribute("blocking", false);
      Properties props = context.getChildrenAsProperties();
      builderAssistant.useNewCache(typeClass, evictionClass, flushInterval,
size, readWrite, blocking, props);
    }
  }
```

继续

```
public Cache useNewCache(Class<? extends Cache> typeClass,
    Class<? extends Cache> evictionClass,
    Long flushInterval,
    Integer size,
    boolean readWrite,
    boolean blocking,
    Properties props) {
  Cache cache = new CacheBuilder(currentNamespace)
      .implementation(valueOrDefault(typeClass, PerpetualCache.class))
      .addDecorator(valueOrDefault(evictionClass, LruCache.class))
      .clearInterval(flushInterval)
      .size(size)
      .readWrite(readWrite)
      .blocking(blocking)
      .properties(props)
      .build();
  configuration.addCache(cache);
  currentCache = cache;
  return cache;
}
```

然后我们可以发现 如果存储 cache 标签，那么对应的 Cache对象会被保存在 currentCache 属性中。



进而在 Cache 对象 保存在了 MapperStatement 对象的 cache 属性中。

然后我们再看看openSession的时候又做了哪些操作，在创建对应的执行器的时候会有缓存的操作

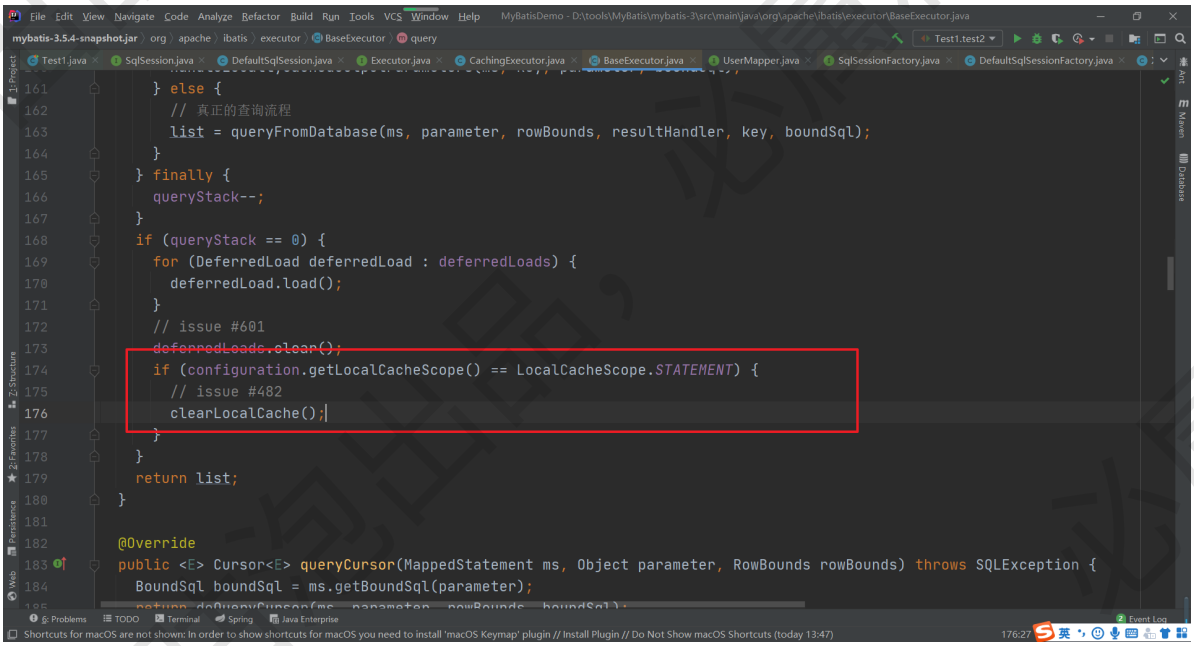
```
  public Executor newExecutor(Transaction transaction, ExecutorType
executorType) {
    executorType = executorType == null ? defaultExecutorType : executorType;
    executorType = executorType == null ? ExecutorType.SIMPLE : executorType;
    Executor executor;
    if (ExecutorType.BATCH == executorType) {
      executor = new BatchExecutor(this, transaction);
    } else if (ExecutorType.REUSE == executorType) {
      executor = new ReuseExecutor(this, transaction);
    } else {
      // 默认 SimpleExecutor
      executor = new SimpleExecutor(this, transaction);
    }
```

```
    // 二级缓存开关，settings 中的 cacheEnabled 默认是 true
    if (cacheEnabled) {
      executor = new CachingExecutor(executor);
    }
    // 植入插件的逻辑，至此，四大对象已经全部拦截完毕
    executor = (Executor) interceptorChain.pluginAll(executor);
    return executor;
  }
```

也就是如果 cacheEnabled 为 true 就会通过 CachingExecutor 来装饰executor 对象，然后就是在执行SQL操作的时候会涉及到缓存的具体使用。这个就分为一级缓存和二级缓存，这个我们来分别介绍

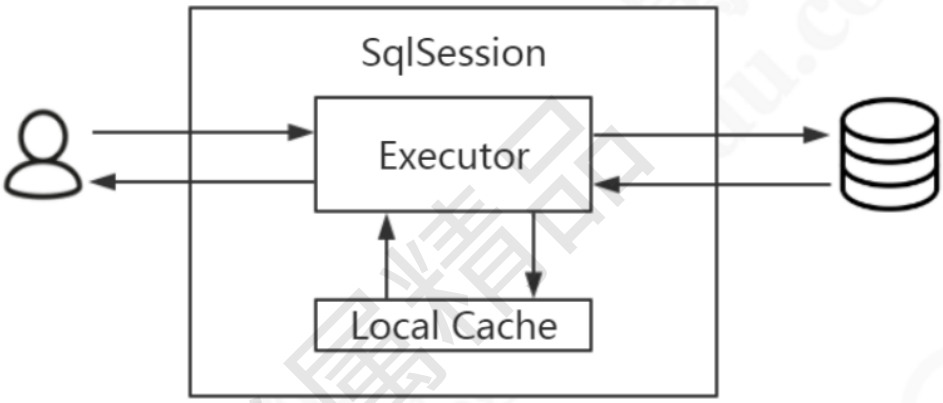
5.4.2 一级缓存

一级缓存也叫本地缓存（Local Cache），MyBatis的一级缓存是在会话（SqlSession）层面进行缓存的。MyBatis的一级缓存是默认开启的，不需要任何的配置（如果要关闭，localCacheScope设置为STATEMENT）。在BaseExecutor对象的query方法中有关闭一级缓存的逻辑



然后我们需要考虑下在一级缓存中的 PerpetualCache 对象在哪创建的，因为一级缓存是Session级别的缓存，肯定需要在Session范围呢创建，其实PerpetualCache的实例化是在BaseExecutor的构造方法中创建的

```
  protected BaseExecutor(Configuration configuration, Transaction transaction) {
    this.transaction = transaction;
    this.deferredLoads = new ConcurrentLinkedQueue<>();
    this.localCache = new PerpetualCache("LocalCache");
    this.localOutputParameterCache = new
PerpetualCache("LocalOutputParameterCache");
    this.closed = false;
    this.configuration = configuration;
    this.wrapper = this;
  }
```



一级缓存的具体实现也是在BaseExecutor的query方法中来实现的

```
public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds
rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql) throws
SQLException {
    // 异常体系之 ErrorContext
    ErrorContext.instance().resource(ms.getResource()).activity("executing a
query").object(ms.getId());
    if (closed) {
      throw new ExecutorException("Executor was closed.");
    }
    if (queryStack == 0 && ms.isFlushCacheRequired()) {
      // flushCache="true"时，即使是查询，也清空一级缓存
      clearLocalCache();
    }
    List<E> list;
    try {
      // 防止递归查询重复处理缓存
      queryStack++;
      // 查询一级缓存
      // ResultHandler 和 ResultSetHandler的区别
      list = resultHandler == null ? (List<E>) localCache.getObject(key) : null;
      if (list != null) {
        handleLocallyCachedOutputParameters(ms, key, parameter, boundSql);
      } else {
        // 真正的查询流程
        list = queryFromDatabase(ms, parameter, rowBounds, resultHandler, key,
boundSql);
      }
    } finally {
      queryStack--;
    }
    if (queryStack == 0) {
      for (DeferredLoad deferredLoad : deferredLoads) {
        deferredLoad.load();
      }
      // issue #601
      deferredLoads.clear();
      if (configuration.getLocalCacheScope() == LocalCacheScope.STATEMENT) {
        // issue #482
        clearLocalCache();
      }
```

```
    }
    return list;
  }
```

一级缓存的验证：

同一个Session中的多个相同操作

```
    @Test
    public void test1() throws  Exception{
        // 1.获取配置文件
        InputStream in = Resources.getResourceAsStream("mybatis-config.xml");
        // 2.加载解析配置文件并获取SqlSessionFactory对象
        SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(in);
        // 3.根据SqlSessionFactory对象获取SqlSession对象
        SqlSession sqlSession = factory.openSession();
        // 4.通过SqlSession中提供的 API方法来操作数据库
        List<User> list =
sqlSession.selectList("com.gupaoedu.mapper.UserMapper.selectUserList");
        System.out.println(list.size());
        // 一级缓存测试
        System.out.println("---------");
        list =
sqlSession.selectList("com.gupaoedu.mapper.UserMapper.selectUserList");
        System.out.println(list.size());
        // 5.关闭会话
        sqlSession.close();
    }
```

输出日志

```
Setting autocommit to false on JDBC Connection
[com.mysql.cj.jdbc.ConnectionImpl@477b4cdf]
==>  Preparing: select * from t_user
==> Parameters:
<==    Columns: id, user_name, real_name, password, age, d_id
<==        Row: 1, zhangsan, 张三, 123456, 18, null
<==        Row: 2, lisi, 李四, 11111, 19, null
<==        Row: 3, wangwu, 王五, 111, 22, 1001
<==        Row: 4, wangwu, 王五, 111, 22, 1001
<==        Row: 5, wangwu, 王五, 111, 22, 1001
<==        Row: 6, wangwu, 王五, 111, 22, 1001
<==        Row: 7, wangwu, 王五, 111, 22, 1001
<==        Row: 8, aaa, bbbb, null, null, null
<==        Row: 9, aaa, bbbb, null, null, null
<==        Row: 10, aaa, bbbb, null, null, null
<==        Row: 11, aaa, bbbb, null, null, null
<==        Row: 12, aaa, bbbb, null, null, null
<==        Row: 666, hibernate, 持久层框架, null, null, null
<==      Total: 13
13
---------
13
```

可以看到第二次查询没有经过数据库操作

不同Session的相同操作

```java
    @Test
    public void test2() throws  Exception{
        // 1.获取配置文件
        InputStream in = Resources.getResourceAsStream("mybatis-config.xml");
        // 2.加载解析配置文件并获取SqlSessionFactory对象
        SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(in);
        // 3.根据SqlSessionFactory对象获取SqlSession对象
        SqlSession sqlSession = factory.openSession();
        // 4.通过SqlSession中提供的 API方法来操作数据库
        List<User> list =
sqlSession.selectList("com.gupaoedu.mapper.UserMapper.selectUserList");
        System.out.println(list.size());
        sqlSession.close();
        sqlSession = factory.openSession();
        // 一级缓存测试
        System.out.println("---------");
        list =
sqlSession.selectList("com.gupaoedu.mapper.UserMapper.selectUserList");
        System.out.println(list.size());
        // 5.关闭会话
        sqlSession.close();
    }
```

输出结果

```
Setting autocommit to false on JDBC Connection
[com.mysql.cj.jdbc.ConnectionImpl@477b4cdf]
==>  Preparing: select * from t_user
==> Parameters:
<==    Columns: id, user_name, real_name, password, age, d_id
<==        Row: 1, zhangsan, 张三, 123456, 18, null
<==        Row: 2, lisi, 李四, 11111, 19, null
<==        Row: 3, wangwu, 王五, 111, 22, 1001
<==        Row: 4, wangwu, 王五, 111, 22, 1001
<==        Row: 5, wangwu, 王五, 111, 22, 1001
<==        Row: 6, wangwu, 王五, 111, 22, 1001
<==        Row: 7, wangwu, 王五, 111, 22, 1001
<==        Row: 8, aaa, bbbb, null, null, null
<==        Row: 9, aaa, bbbb, null, null, null
<==        Row: 10, aaa, bbbb, null, null, null
<==        Row: 11, aaa, bbbb, null, null, null
<==        Row: 12, aaa, bbbb, null, null, null
<==        Row: 666, hibernate, 持久层框架, null, null, null
<==      Total: 13
13
Resetting autocommit to true on JDBC Connection
[com.mysql.cj.jdbc.ConnectionImpl@477b4cdf]
Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@477b4cdf]
Returned connection 1199262943 to pool.
---------
Opening JDBC Connection
Checked out connection 1199262943 from pool.
Setting autocommit to false on JDBC Connection
[com.mysql.cj.jdbc.ConnectionImpl@477b4cdf]
==>  Preparing: select * from t_user
==> Parameters:
```

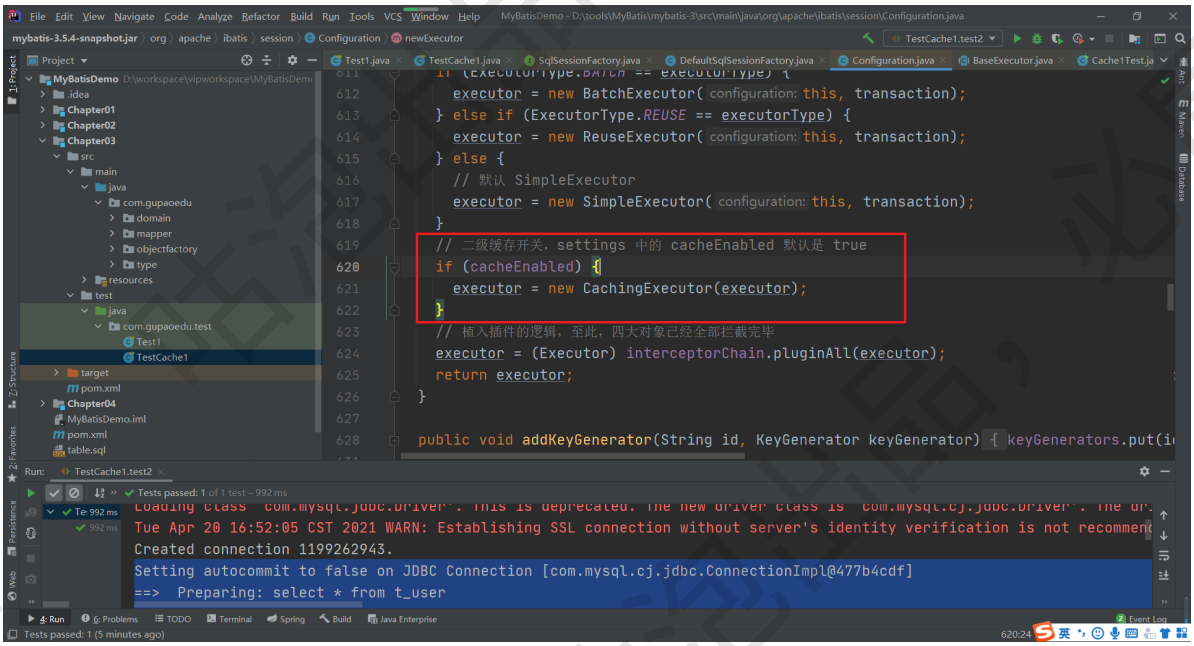
```
<==    Columns: id, user_name, real_name, password, age, d_id
<==        Row: 1, zhangsan, 张三, 123456, 18, null
<==        Row: 2, lisi, 李四, 11111, 19, null
<==        Row: 3, wangwu, 王五, 111, 22, 1001
<==        Row: 4, wangwu, 王五, 111, 22, 1001
<==        Row: 5, wangwu, 王五, 111, 22, 1001
<==        Row: 6, wangwu, 王五, 111, 22, 1001
<==        Row: 7, wangwu, 王五, 111, 22, 1001
<==        Row: 8, aaa, bbbb, null, null, null
<==        Row: 9, aaa, bbbb, null, null, null
<==        Row: 10, aaa, bbbb, null, null, null
<==        Row: 11, aaa, bbbb, null, null, null
<==        Row: 12, aaa, bbbb, null, null, null
<==        Row: 666, hibernate, 持久层框架, null, null, null
<==      Total: 13
13
Resetting autocommit to true on JDBC Connection
[com.mysql.cj.jdbc.ConnectionImpl@477b4cdf]
Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@477b4cdf]
Returned connection 1199262943 to pool.
```

通过输出我们能够发现，不同的Session中的相同操作，一级缓存是没有起作用的。

5.4.3 二级缓存

二级缓存是用来解决一级缓存不能跨会话共享的问题的，范围是namespace级别的，可以被多个SqlSession共享（只要是同一个接口里面的相同方法，都可以共享），生命周期和应用同步。

二级缓存的设置，首先是settings中的cacheEnabled要设置为true，当然默认的就是为true，这个步骤决定了在创建Executor对象的时候是否通过CachingExecutor来装饰。



那么设置了cacheEnabled标签为true是否就意味着 二级缓存是否一定可用呢？当然不是，我们还需要在 对应的映射文件中添加 cache 标签才行。

```
<!-- 声明这个namespace使用二级缓存 -->
<cache type="org.apache.ibatis.cache.impl.PerpetualCache"
       size="1024"  <!–最多缓存对象个数，默认1024-->
       eviction="LRU" <!–回收策略-->
       flushInterval="120000" <!–自动刷新时间 ms，未配置时只有调用时刷新-->
       readOnly="false"/> <!–默认是false（安全），改为true可读写时，对象必须支持序列化 -->
```

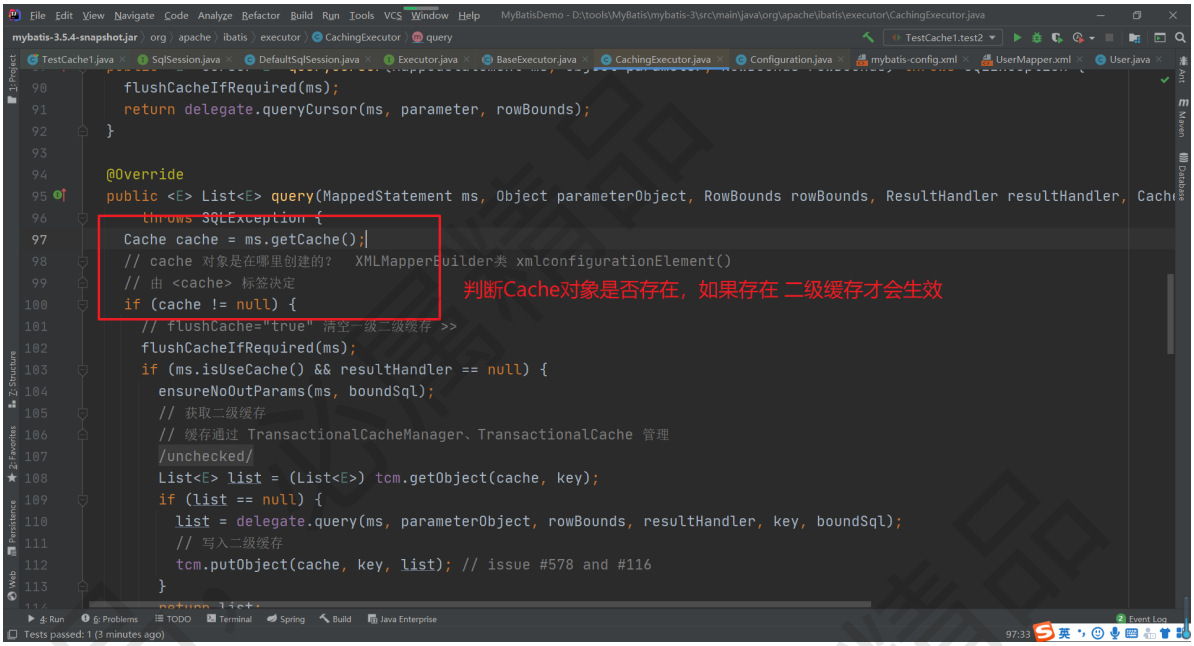
cache属性详解：

属性	含义	取值
type	缓存实现类	需要实现Cache接口，默认是PerpetualCache，可以使用第三方缓存
size	最多缓存对象个数	默认1024
eviction	回收策略（缓存淘汰算法）	LRU – 最近最少使用的：移除最长时间不被使用的对象（默认）。FIFO – 先进先出：按对象进入缓存的顺序来移除它们。SOFT – 软引用：移除基于垃圾回收器状态和软引用规则的对象。WEAK – 弱引用：更积极地移除基于垃圾收集器状态和弱引用规则的对象。
flushInterval	定时自动清空缓存间隔	自动刷新时间，单位 ms，未配置时只有调用时刷新
readOnly	是否只读	true：只读缓存；会给所有调用者返回缓存对象的相同实例。因此这些对象不能被修改。这提供了很重要的性能优势。false：读写缓存；会返回缓存对象的拷贝（通过序列化），不会共享。这会慢一些，但是安全，因此默认是 false。改为false可读写时，对象必须支持序列化。
blocking	启用阻塞缓存	通过在get/put方式中加锁，保证只有一个线程操作缓存，基于Java重入锁实现

再来看下cache标签在源码中的体现，创建cacheKey

```
  @Override
  public <E> List<E> query(MappedStatement ms, Object parameterObject, RowBounds
rowBounds, ResultHandler resultHandler) throws SQLException {
    // 获取SQL
    BoundSql boundSql = ms.getBoundSql(parameterObject);
    // 创建CacheKey：什么样的SQL是同一条SQL？ >>
    CacheKey key = createCacheKey(ms, parameterObject, rowBounds, boundSql);
    return query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);
  }
```

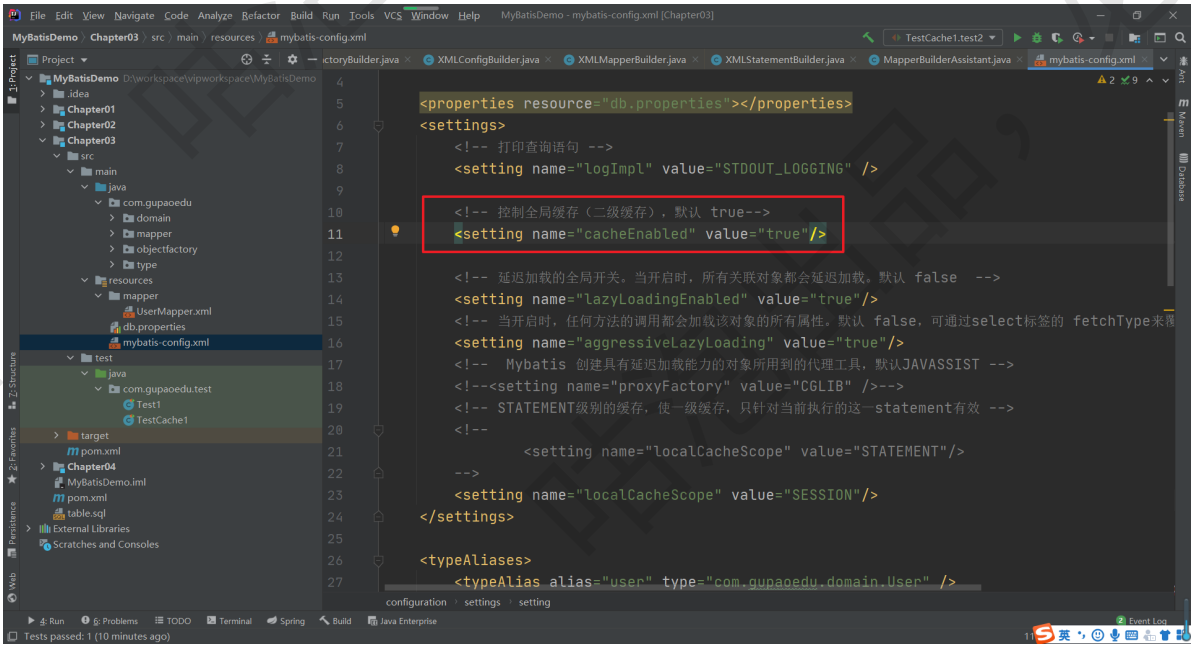
createCacheKey自行进去查看

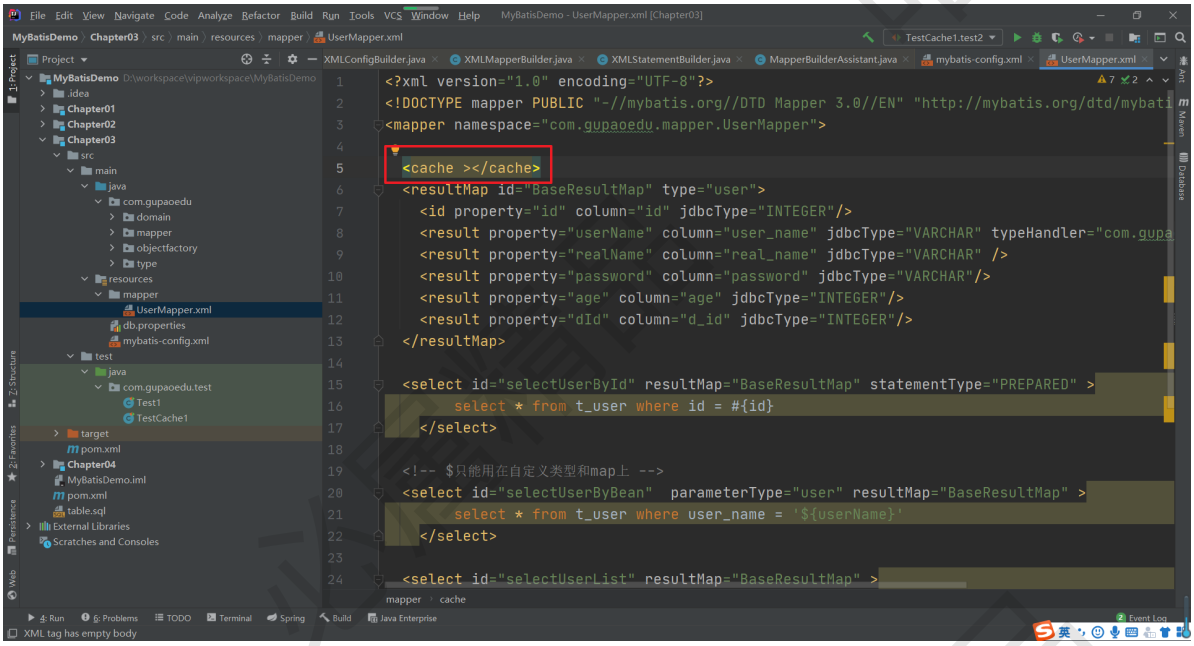



```
flushCacheIfRequired(ms);
return delegate.queryCursor(ms, parameter, rowBounds);
}

@Override
public <E> List<E> query(MappedStatement ms, Object parameterObject, RowBounds rowBounds, ResultHandler resultHandler, Cache
    throws SQLException {
Cache cache = ms.getCache();
// cache 对象是在哪里创建的？  XMLMapperBuilder类 xmlconfigurationElement()
// 由 <cache> 标签决定
if (cache != null) {
  // flushCache="true" 清空一级二级缓存 >>
  flushCacheIfRequired(ms);
  if (ms.isUseCache() && resultHandler == null) {
    ensureNoOutParams(ms, boundSql);
    // 获取二级缓存
    // 缓存通过 TransactionalCacheManager、TransactionalCache 管理
    /unchecked/
    List<E> list = (List<E>) tcm.getObject(cache, key);
    if (list == null) {
      list = delegate.query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);
      // 写入二级缓存
      tcm.putObject(cache, key, list); // issue #578 and #116
    }
    return list;
```

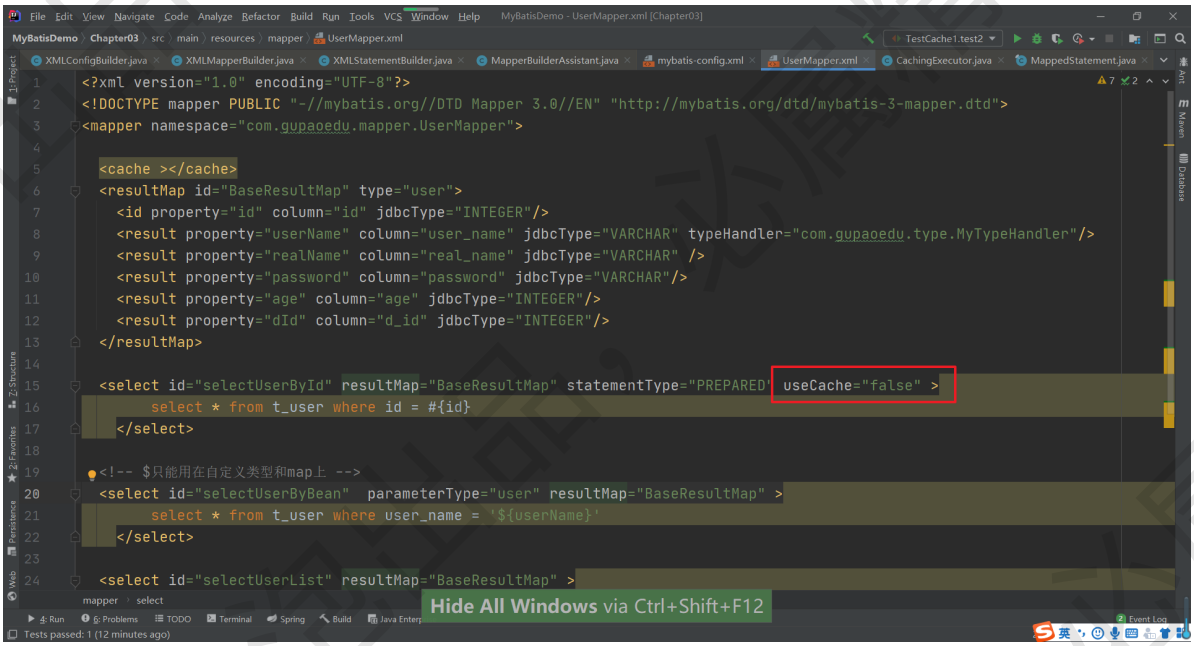
```java
*/
public final class MappedStatement {

  private String resource;
  private Configuration configuration;
  private String id;
  private Integer fetchSize;
  private Integer timeout;
  private StatementType statementType;
  private ResultSetType resultSetType;
  private SqlSource sqlSource;
  private Cache cache;
  private ParameterMap parameterMap;
  private List<ResultMap> resultMaps;
  private boolean flushCacheRequired;
  private boolean useCache;
  private boolean resultOrdered;
  private SqlCommandType sqlCommandType;
  private KeyGenerator keyGenerator;
  private String[] keyProperties;
  private String[] keyColumns;
  private boolean hasNestedResultMaps;
  private String databaseId;
```

而这看到的和我们前面在缓存初始化时看到的 cache 标签解析操作是对应上的。所以我们要开启二级缓存两个条件都要满足。





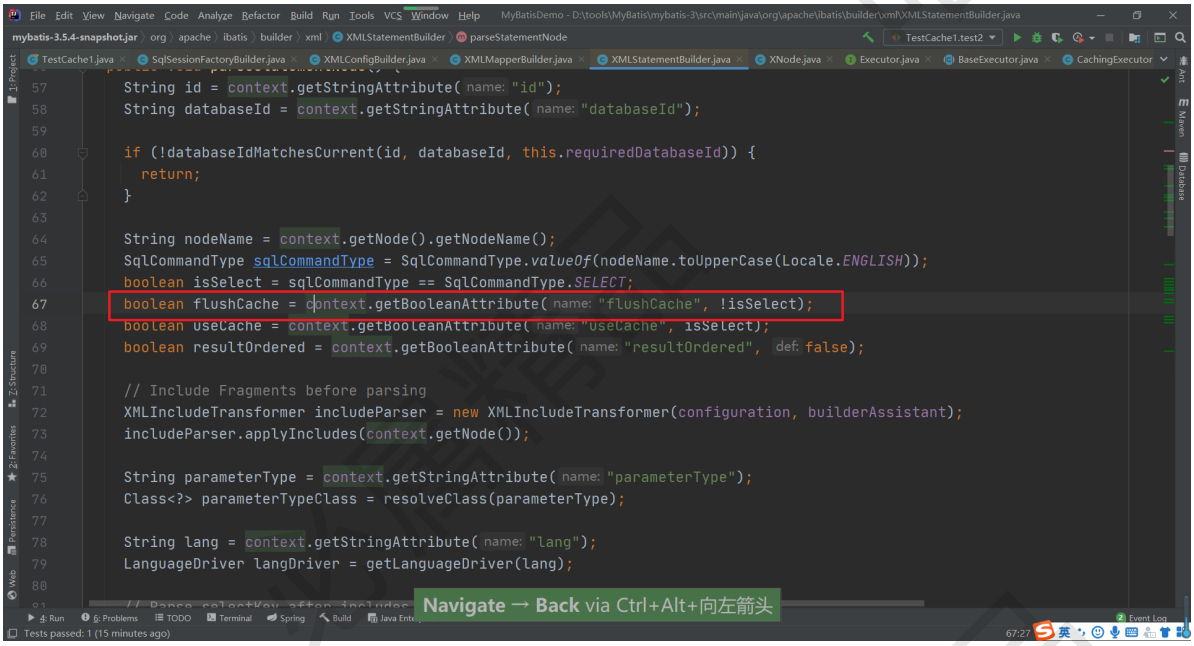
这样的设置表示当前的映射文件中的相关查询操作都会触发二级缓存，但如果某些个别方法我们不希望走二级缓存怎么办呢？我们可以在标签中添加一个 useCache=false 来实现的设置不使用二级缓存



还有就是当我们执行的对应的DML操作，在MyBatis中会清空对应的二级缓存和一级缓存。

```
private void flushCacheIfRequired(MappedStatement ms) {
  Cache cache = ms.getCache();
  // 增删改查的标签上有属性：flushCache="true" （select语句默认是false）
  // 一级二级缓存都会被清理
  if (cache != null && ms.isFlushCacheRequired()) {
    tcm.clear(cache);
  }
}
```

在解析映射文件的时候DML操作flushCacheRequired为true



5.4.4 第三方缓存

在实际开发的时候我们一般也很少使用MyBatis自带的二级缓存，这时我们会使用第三方的缓存工具Ehcache获取Redis来实现,那么他们是如何来实现的呢?

https://github.com/mybatis/redis-cache

添加依赖

```
<dependency>
   <groupId>org.mybatis.caches</groupId>
   <artifactId>mybatis-redis</artifactId>
   <version>1.0.0-beta2</version>
</dependency>
```

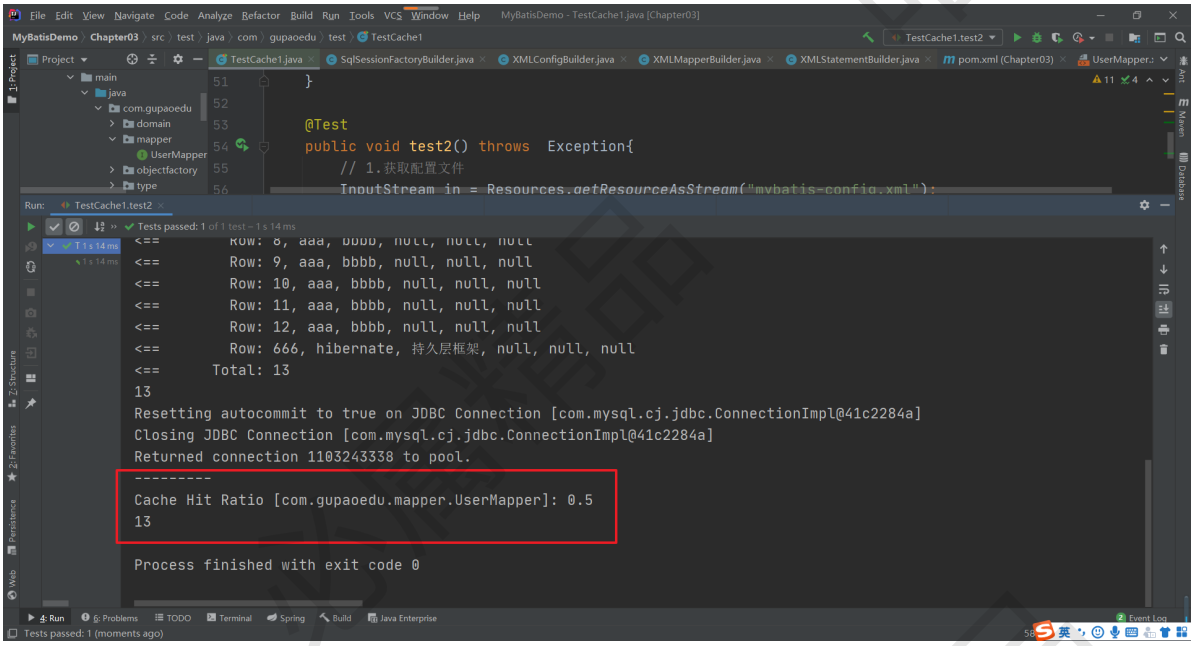
然后加上Cache标签的配置

```
  <cache type="org.mybatis.caches.redis.RedisCache"
         eviction="FIFO"
         flushInterval="60000"
         size="512"
         readOnly="true"/>
```

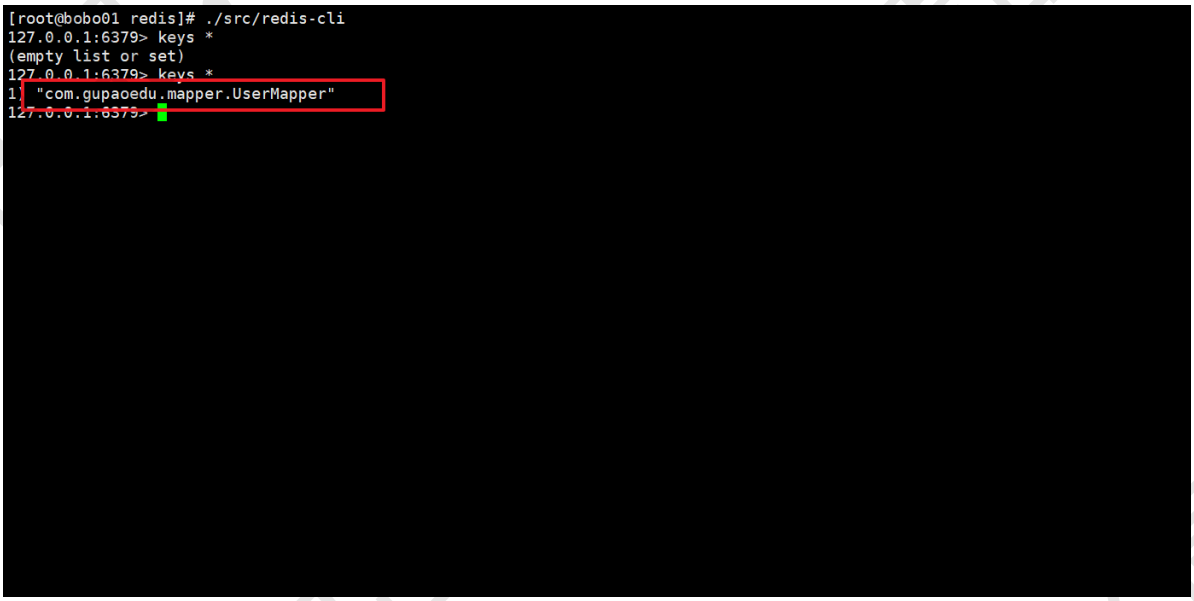
然后添加redis的属性文件

```
host=192.168.100.120
port=6379
connectionTimeout=5000
soTimeout=5000
database=0
```

测试效果

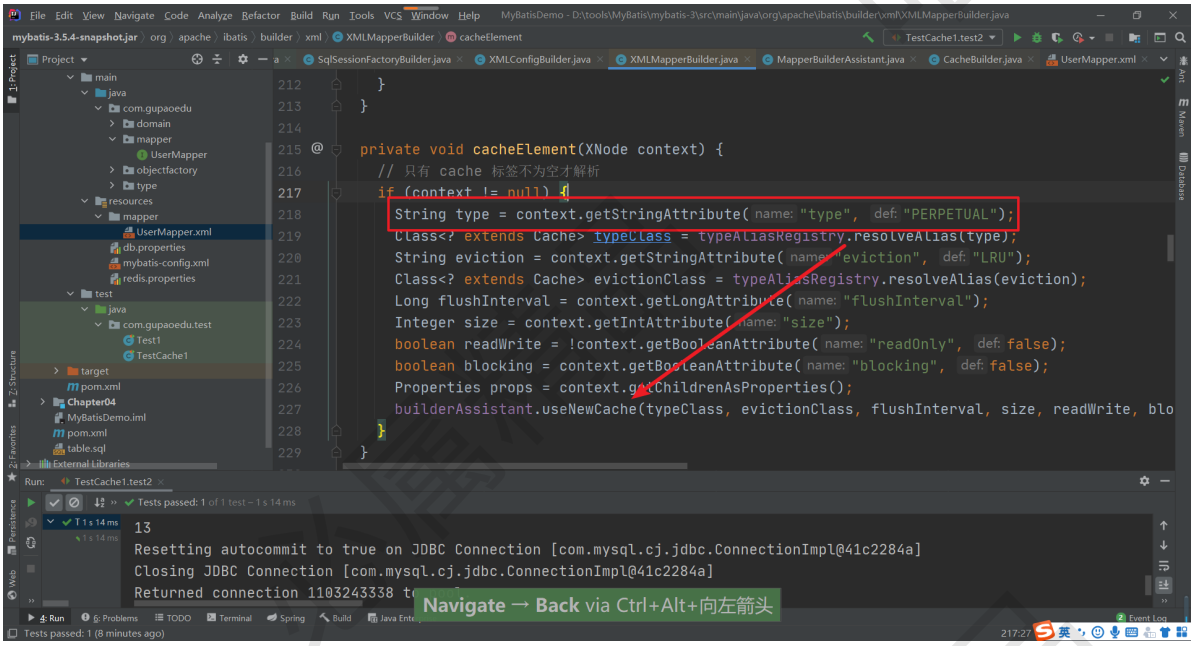


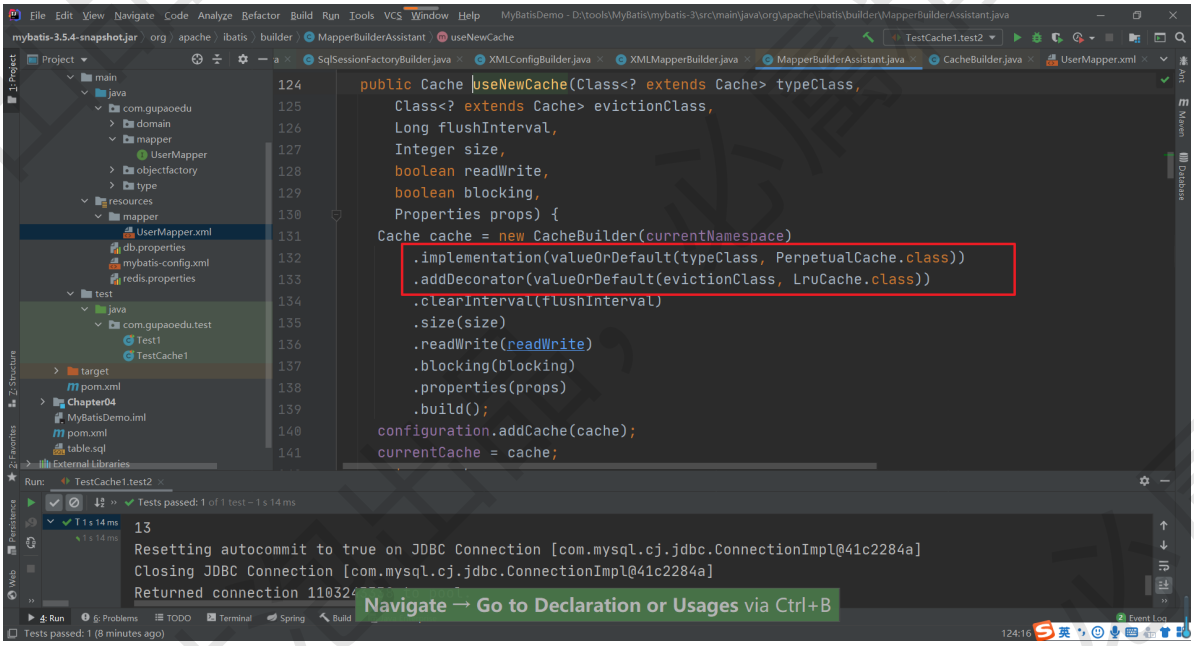
数据存储到了Redis中



然后大家也可以自行分析下第三方的Cache是如何替换掉PerpetualCache的，因为PerpetualCache是基于HashMap处理的，而RedisCache是基于Redis来存储缓存数据的。

提示





缓存模块我们就介绍到此。 然后大家可以基于我们上面所介绍的基础支持层，再系统的来梳理下核心处理层的流程