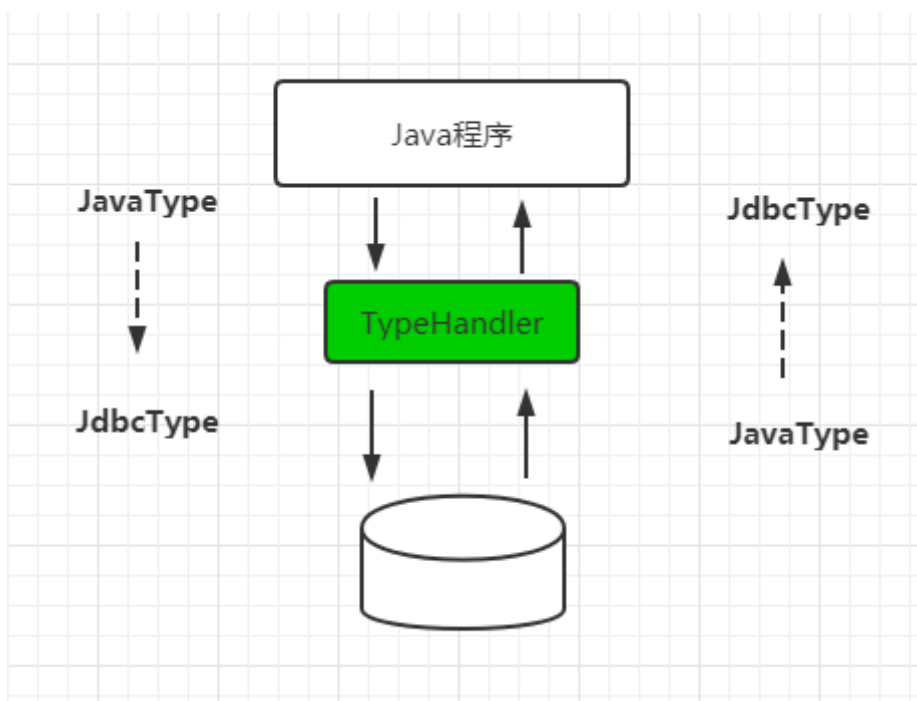


MyBatis基础模块-类型转换模块

1.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。



1.1 TypeHandler

MyBatis中的所有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 Column name, when configuration
    * <code>useColumnLabel</code> is <code>>false</code>
    */
    T getResult(ResultSet rs, String columnName) throws SQLException;

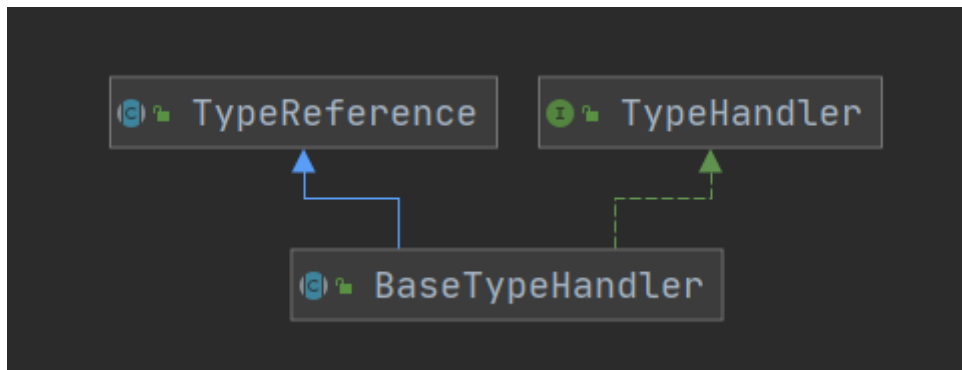
    T getResult(ResultSet rs, int columnIndex) throws SQLException;

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

```

1.2 BaseTypeHandler

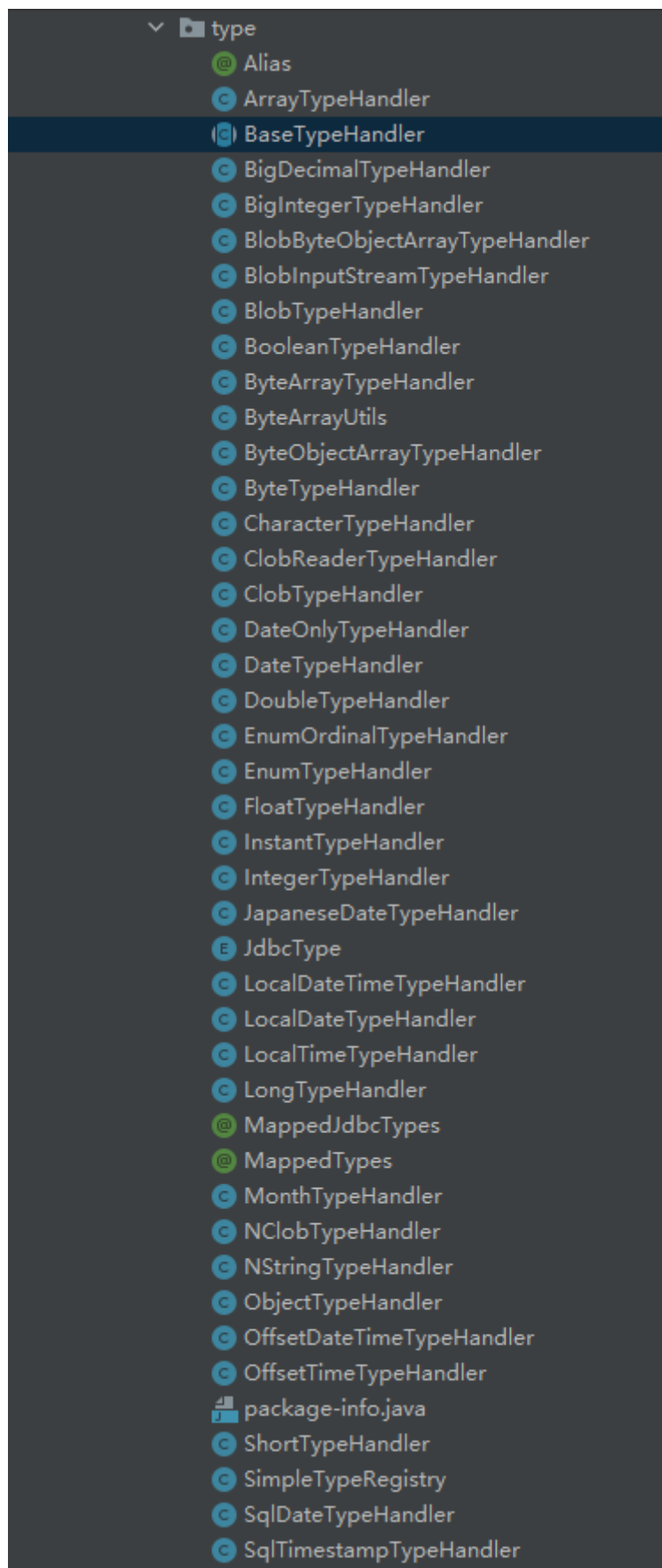
为了方便用户自定义TypeHandler的实现，在MyBatis中提供了BaseTypeHandler这个抽象类，它实现了TypeHandler接口，并继承了TypeReference类，



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

1.3 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;
}
}

```

1.4 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？实际上在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TypeHandler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TypeHandler
private final Map<Type, Map<JdbcType, TypeHandler<?>>> typeHandlerMap = new
ConcurrentHashMap<>();
private final TypeHandler<Object> unknownTypeHandler;
// 记录全部的TypeHandler类型及对应的TypeHandler对象
private final Map<Class<?>, TypeHandler<?>> allTypeHandlersMap = new HashMap<>
();

// 空TypeHandler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

```

public TypeHandlerRegistry(Configuration configuration) {
    this.unknownTypeHandler = new UnknownTypeHandler(configuration);

    register(Boolean.class, new BooleanTypeHandler());
    register(boolean.class, new BooleanTypeHandler());
    register(JdbcType.BOOLEAN, new BooleanTypeHandler());
    register(JdbcType.BIT, new BooleanTypeHandler());

    register(Byte.class, new ByteTypeHandler());
    register(byte.class, new ByteTypeHandler());
    register(JdbcType.TINYINT, new ByteTypeHandler());

    register(Short.class, new ShortTypeHandler());
    register(short.class, new ShortTypeHandler());
    register(JdbcType.SMALLINT, new ShortTypeHandler());

    register(Integer.class, new IntegerTypeHandler());
    register(int.class, new IntegerTypeHandler());
    register(JdbcType.INTEGER, new IntegerTypeHandler());

    register(Long.class, new LongTypeHandler());
    register(long.class, new LongTypeHandler());

    register(Float.class, new FloatTypeHandler());
    register(float.class, new FloatTypeHandler());
}

```

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 TypeHandler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TypeHandler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TypeHandler
 */
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TypeHandler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。

1.5 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

```

public class TypeAliasRegistry {

    // 保存 类型和别名的对应关系
    private final Map<String, Class<?>> typeAliases = new HashMap<>();

    public TypeAliasRegistry() {
        registerAlias(alias: "string", String.class);

        registerAlias(alias: "byte", Byte.class);
        registerAlias(alias: "long", Long.class);
        registerAlias(alias: "short", Short.class);
        registerAlias(alias: "int", Integer.class);
        registerAlias(alias: "integer", Integer.class);
        registerAlias(alias: "double", Double.class);
        registerAlias(alias: "float", Float.class);
        registerAlias(alias: "boolean", Boolean.class);

        registerAlias(alias: "byte[]", Byte[].class);
        registerAlias(alias: "long[]", Long[].class);
        registerAlias(alias: "short[]", Short[].class);
        registerAlias(alias: "int[]", Integer[].class);
        registerAlias(alias: "integer[]", Integer[].class);
        registerAlias(alias: "double[]", Double[].class);
        registerAlias(alias: "float[]", Float[].class);
        registerAlias(alias: "boolean[]", Boolean[].class);
    }
}

```

注册的方法逻辑也比较简单

```

public void registerAlias(String alias, Class<?> value) {
    if (alias == null) {
        throw new RuntimeException("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 RuntimeException("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

```

```

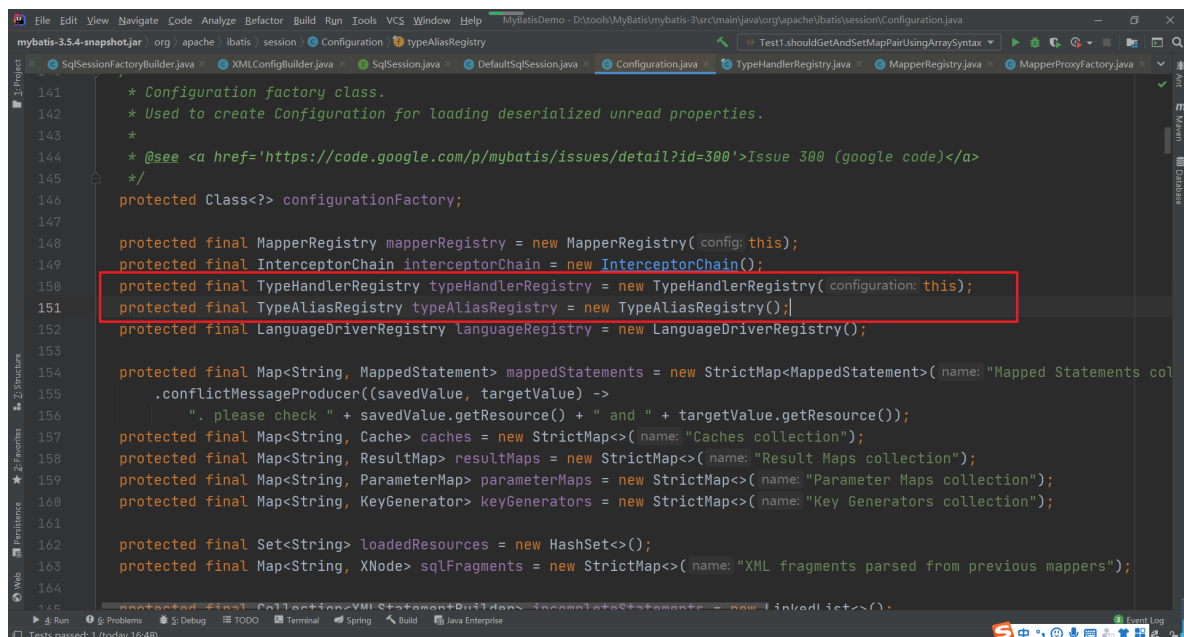
    * @param packageName
    * @param superType
    */
    public void registerAliases(String packageName, Class<?> superType) {
        ResolverUtil<Class<?>> resolverUtil = new ResolverUtil<>();
        resolverUtil.find(new ResolverUtil.IsA(superType), packageName);
        Set<Class<? extends Class<?>>> 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);
    }
}

```

1.6 TypeHandler的应用

1.6.1 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


```

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);

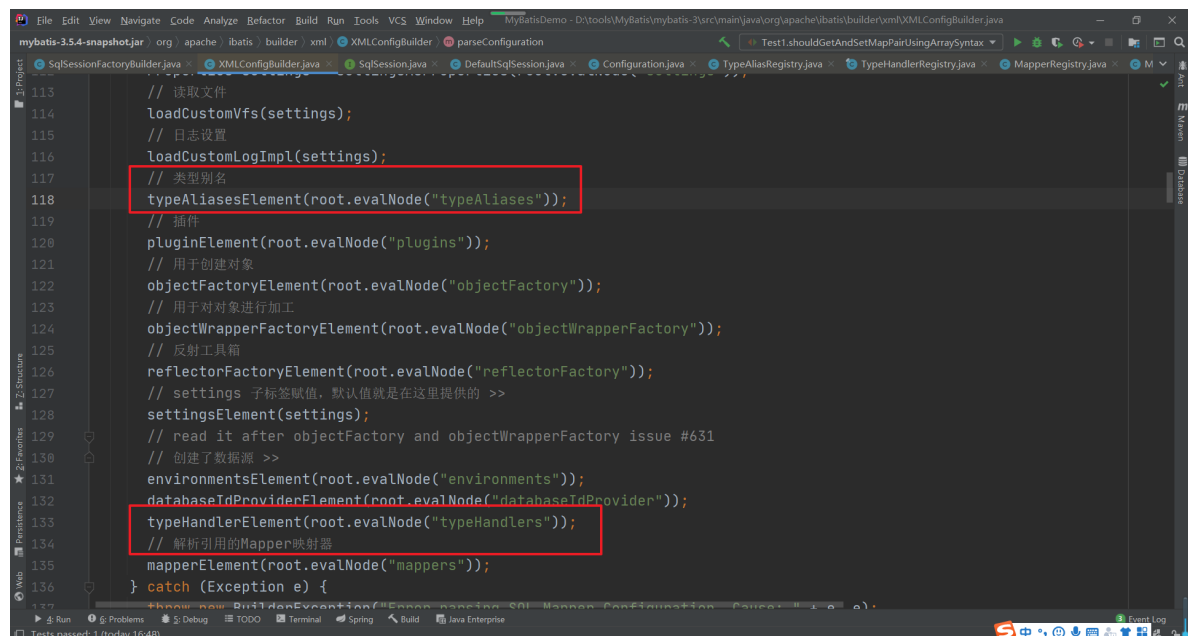
    typeAliasRegistry.registerAlias( alias: "XML", XMLLanguageDriver.class);
    typeAliasRegistry.registerAlias( alias: "RAW", RawLanguageDriver.class);

    typeAliasRegistry.registerAlias( alias: "SLF4J", Slf4jImpl.class);
    typeAliasRegistry.registerAlias( alias: "COMMONS_LOGGING", JakartaCommonsLoggingImpl.class);
    typeAliasRegistry.registerAlias( alias: "LOG4J", Log4jImpl.class);
    typeAliasRegistry.registerAlias( alias: "LOG4J2", Log4j2Impl.class);
}

```

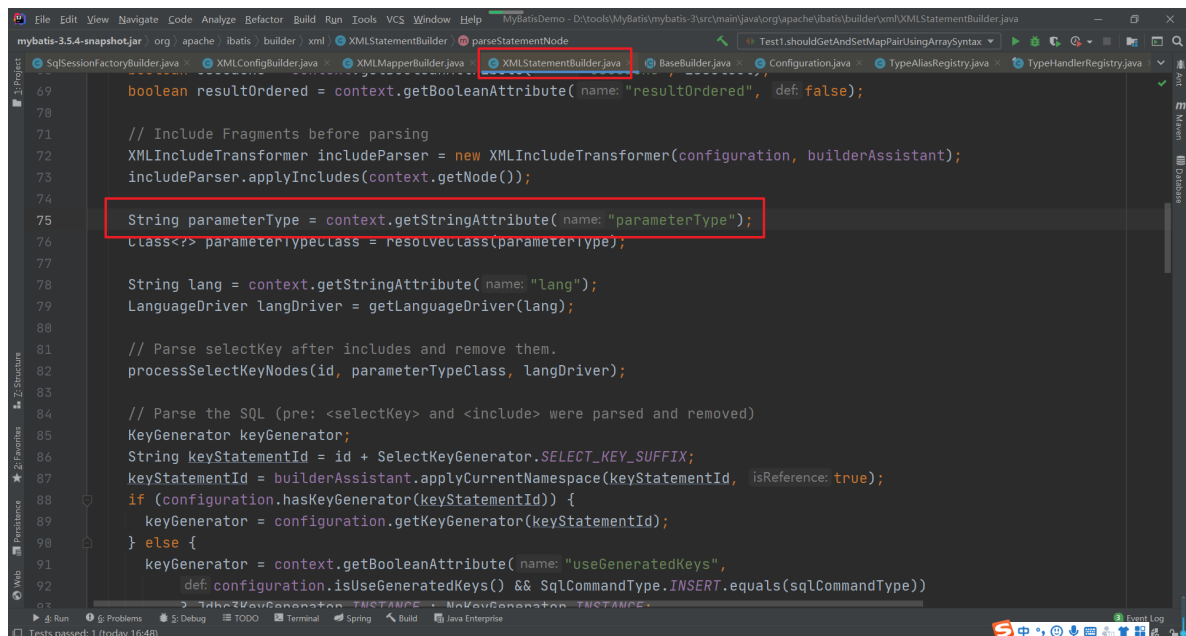
以上步骤完成了TypeHandlerRegistry和TypeAliasRegistry的初始化操作

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。



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
}
```

1.6.2 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

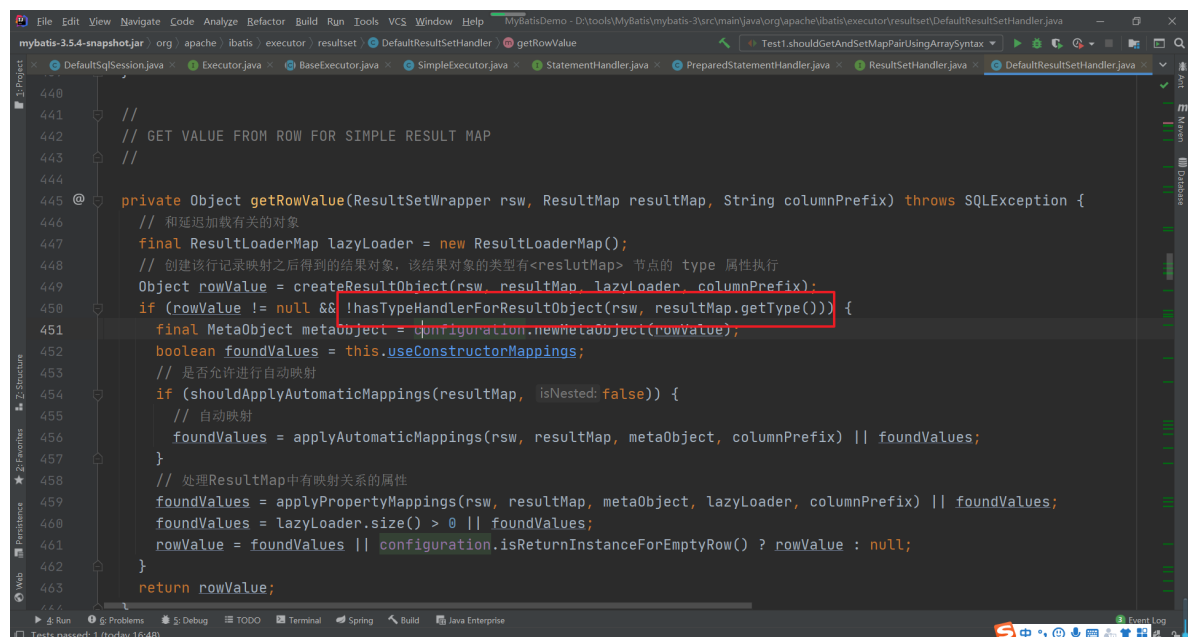
```

```

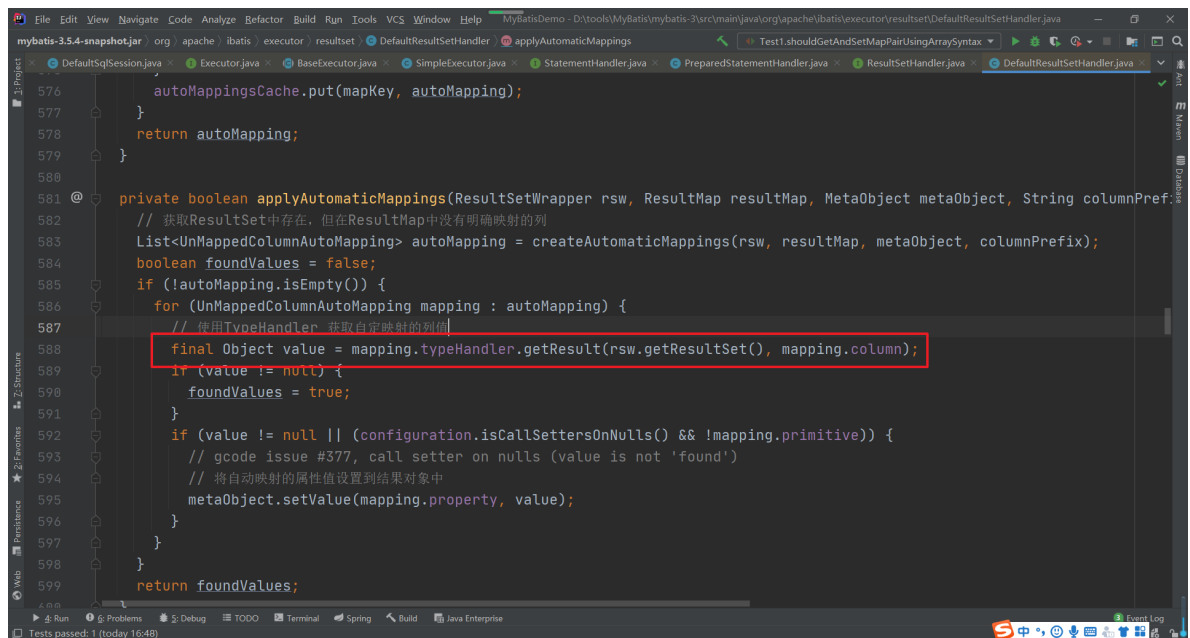
        Object value;
        String propertyName = parameterMapping.getProperty();
        if (boundSql.hasAdditionalParameter(propertyName)) { // issue #448 ask
first for additional params
            value = boundSql.getAdditionalParameter(propertyName);
        } else if (parameterObject == null) {
            value = null;
        } else if
(configurationHandlerRegistry.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



```
576     autoMappingsCache.put(mapKey, autoMapping);
577 }
578 return autoMapping;
579 }
580
581 @
582 private boolean applyAutomaticMappings(ResultSetWrapper rsw, ResultMap resultMap, MetaObject metaObject, String columnPrefix) {
583     // 获取ResultSet中存在, 但在ResultMap中没有明确映射的列
584     List<UnMappedColumnAutoMapping> autoMapping = createAutomaticMappings(rsw, resultMap, metaObject, columnPrefix);
585     boolean foundValues = false;
586     if (!autoMapping.isEmpty()) {
587         for (UnMappedColumnAutoMapping mapping : autoMapping) {
588             // 使用TypeHandler 获取自定义映射的列值
589             final Object value = mapping.typeHandler.getResult(rsw.getResultSet(), mapping.column);
590             if (value != null) {
591                 foundValues = true;
592             }
593             if (value != null || (configuration.isCallSettersOnNulls() && !mapping.primitive)) {
594                 // gcode issue #377, call setter on nulls (value is not 'found')
595                 // 将自动映射的属性值设置到结果对象中
596                 metaObject.setValue(mapping.property, value);
597             }
598         }
599     }
600     return foundValues;
601 }
```

根据对应的TypeHandler返回对应类型的值。