# 05 数据库类型体系与 Java 类型体系之间的“爱恨情仇”

作为一个 Java 程序员，你应该已经具备了使用 JDBC 操作数据库的基础技能。在使用 JDBC 的时候，你会发现 JDBC 的数据类型与 Java 语言中的数据类型虽然有点对应关系，如下图所示，但还是无法做到一一对应，也自然无法做到自动映射。

| 数据库类型 | JAVA类型 |
| --- | --- |
| | |
| VARCHAR | java.lang.String |
| CHAR | java.lang.String |
| BLOB | java.lang.byte[] |
| VARCHAR | java.lang.String |
| | |
| INTEGER UNSIGNED | java.lang.Long |
| TINYINT UNSIGNED | java.lang.Integer |
| SMALLINT UNSIGNED | java.lang.Integer |
| MEDIUMINT UNSIGNED | java.lang.Integer |

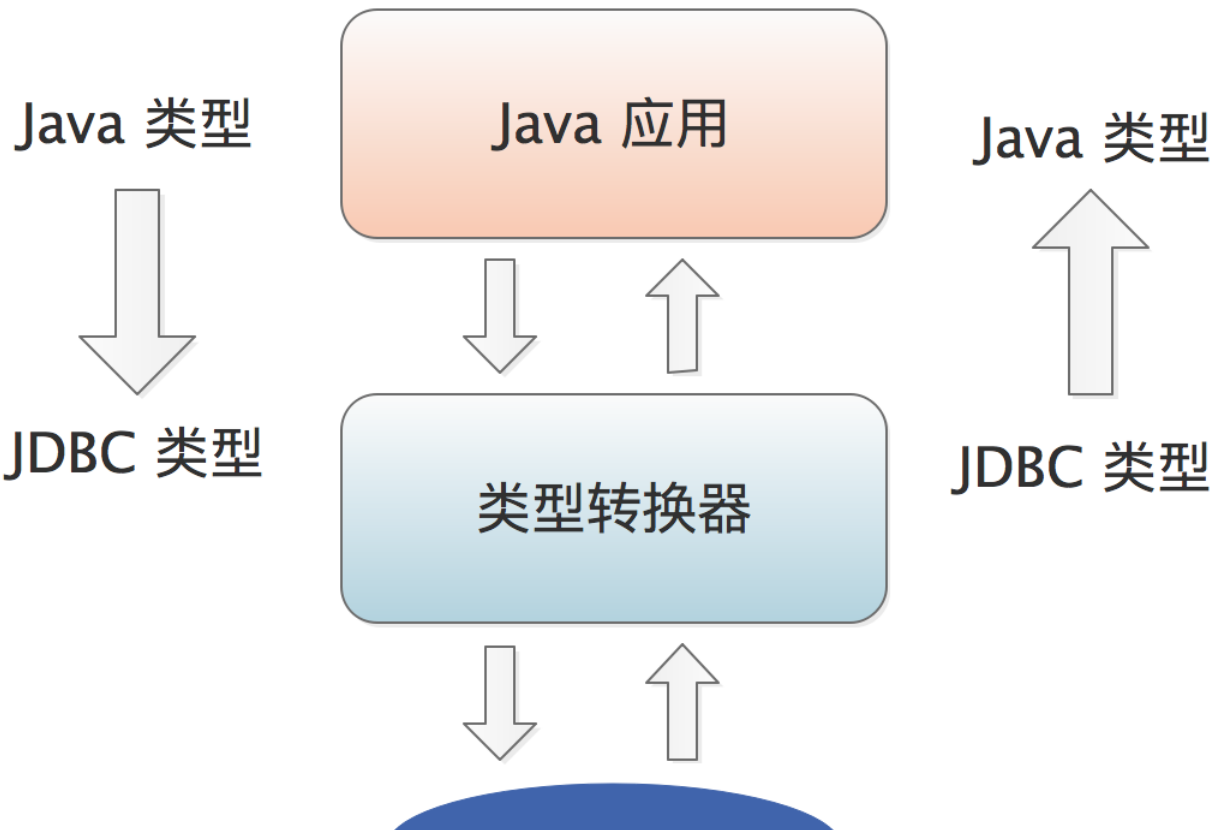
| | |
|---|---|
| BIT | java.lang.Boolean |
| BIGINT UNSIGNED | java.math.BigInteger |
| FLOAT | java.lang.Float |
| DOUBLE | java.lang.Double |
| DECIMAL | java.math.BigDecimal |

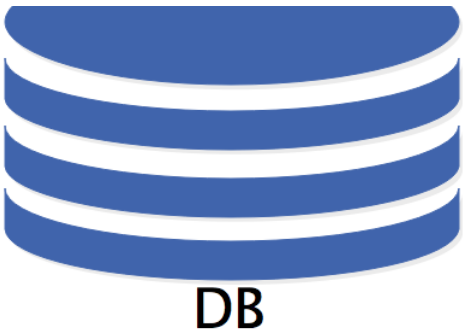


数据库类型与 Java 类型对应图表

在使用 PreparedStatement 执行 SQL 语句之前，都是需要手动调用 setInt()、setString() 等 set 方法绑定参数，这不仅仅是告诉 JDBC 一个 SQL 模板中哪个占位符需要使用哪个实参，还会将数据从 Java 类型转换成 JDBC 类型。当从 ResultSet 中获取数据的时候，则是一个逆过程，数据会从 JDBC 类型转换为 Java 类型。

可以使用 MyBatis 中的**类型转换器**，完成上述两次类型转换，如下图所示：







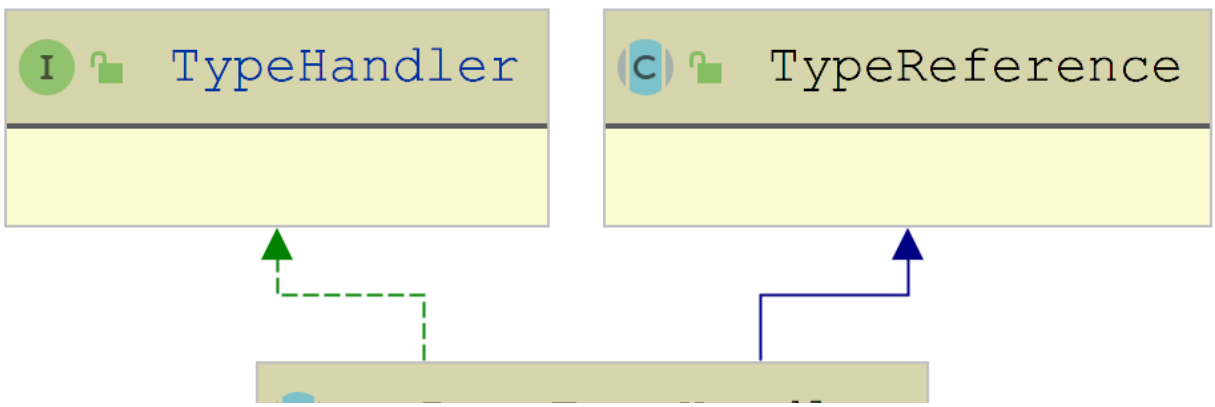

JDBC 类型数据与 Java 类型数据转换示意图

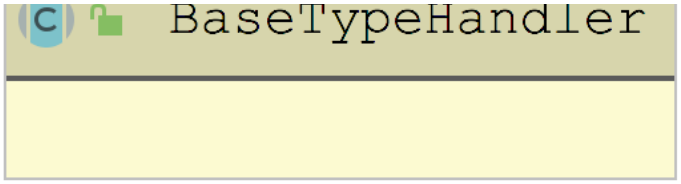
## 深入 TypeHandler

说了这么多，类型转换器到底是怎么定义的呢？其实，**MyBatis 中的类型转换器就是 TypeHandler 这个接口**，其定义如下：

```
public interface TypeHandler<T> {

  // 在通过PreparedStatement为SQL语句绑定参数时，会将传入的实参数据由JdbcType类型转换成J

  void setParameter(PreparedStatement ps, int i, T parameter, JdbcType jdbcType) th

  // 从ResultSet中获取数据时会使用getResult()方法，其中会将读取到的数据由Java类型转换成J

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

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

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

}
```

**MyBatis 中定义了 BaseTypeHandler 抽象类来实现一些 TypeHandler 的公共逻辑**，BaseTypeHandler 在实现 TypeHandler 的同时，还实现了 TypeReference 抽象类。其继承关系如下图所示：







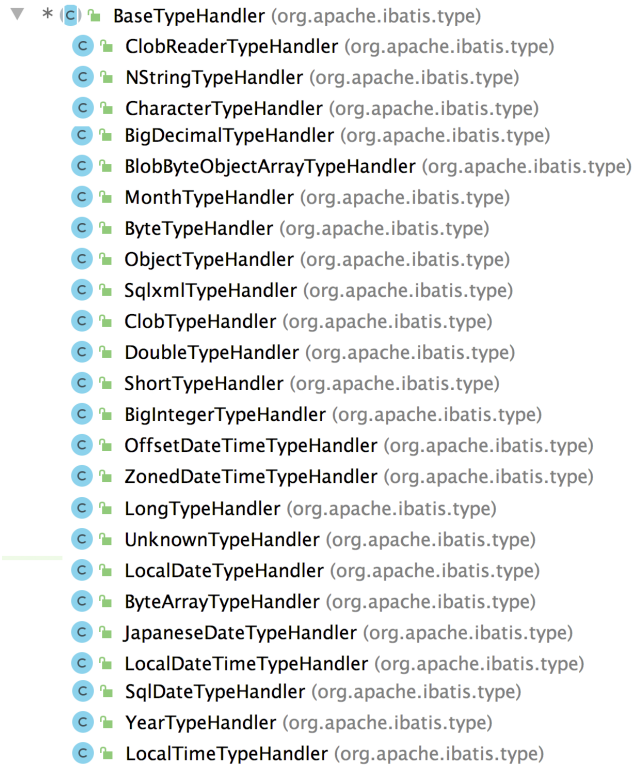

TypeHandler 继承关系图

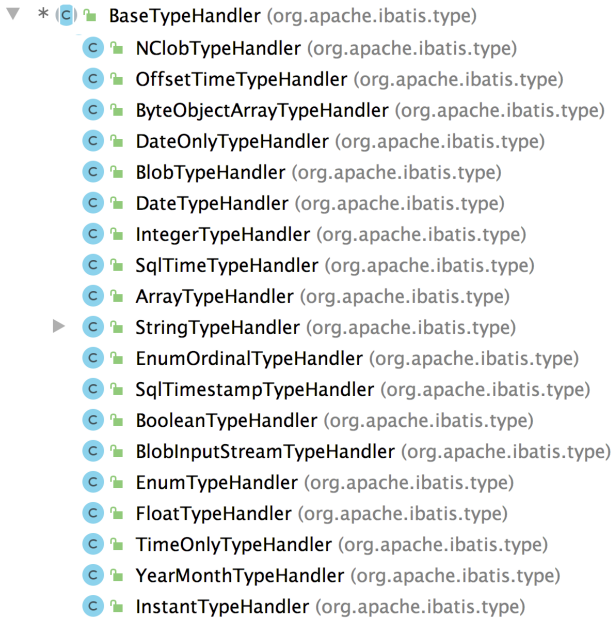
在 BaseTypeHandler 中，简单实现了 TypeHandler 接口的 setParameter() 方法和 getResult() 方法。

- 在 setParameter() 实现中，会判断传入的 parameter 实参是否为空，如果为空，则调用 PreparedStatement.setNull() 方法进行设置；如果不为空，则委托 setNonNullParameter() 这个抽象方法进行处理，setNonNullParameter() 方法由 BaseTypeHandler 的子类提供具体实现。
- 在 getResult() 的三个重载实现中，会直接调用相应的 getNullableResult() 抽象方法，这里有三个重载的 getNullableResult() 抽象方法，它们都由 BaseTypeHandler 的子类提供具体实现。

BaseTypeHandler 的具体实现比较简单，这里就不再展示，你若感兴趣的话可以参考源码进行学习。

下图展示了 BaseTypeHandler 的全部实现类，虽然实现类比较多，但是它们的实现方式大同小异。



BaseTypeHandler 实现类示意图

这里我们以 LongTypeHandler 为例进行分析，具体实现如下：

```
public class LongTypeHandler extends BaseTypeHandler<Long> {
    public void setNonNullParameter(PreparedStatement ps, int i, Long parameter, Jd
            throws SQLException {
        // 调用PreparedStatement.setLong()实现参数绑定
        ps.setLong(i, parameter);
    }
    public Long getNullableResult(ResultSet rs, String columnName)
            throws SQLException {
        // 调用ResultSet.getLong()获取指定列值
        long result = rs.getLong(columnName);
        return result == 0 && rs.wasNull() ? null : result;
    }
    public Long getNullableResult(ResultSet rs, int columnIndex)
            throws SQLException {
        // 调用ResultSet.getLong()获取指定列值
        long result = rs.getLong(columnIndex);
        return result == 0 && rs.wasNull() ? null : result;
    }
    public Long getNullableResult(CallableStatement cs, int columnIndex)
            throws SQLException {
        // 调用ResultSet.getLong()获取指定列值
        long result = cs.getLong(columnIndex);
        return result == 0 && cs.wasNull() ? null : result;
    }
}
```

可以看到：LongTypeHandler 的核心还是通过 PreparedStatement.setLong() 方法以及 ResultSet.getLong() 方法实现的。至于其他 BaseTypeHandler 的核心实现，同样也是依赖了 JDBC 的 API，这里就不再展开介绍了。

## TypeHandler 注册

了解了 TypeHandler 接口实现类的核心原理之后，我们就来解决下面两个问题：

- MyBatis 如何管理这么多的 TypeHandler 接口实现呢?
- 如何在合适的场景中使用合适的 TypeHandler 实现进行类型转换呢?

你若使用过 MyBatis 的话，应该知道我们可以在 mybatis-config.xml 中通过 标签配置自定义的 TypeHandler 实现，也可以在 Mapper.xml 配置文件定义 的时候指定 typeHandler 属性。无论是哪种配置方式，MyBatis 都会在初始化过程中，获取所有已知的 TypeHandler（包括内置实现和自定义实现），然后创建所有 TypeHandler 实例并注册到 TypeHandlerRegistry 中，**由 TypeHandlerRegistry 统一管理所有 TypeHandler 实例。** TypeHandlerRegistry 管理 TypeHandler 的时候，用到了以下四个最核心的集合。

- jdbcTypeHandlerMap（Map<JdbcType, TypeHandler<?>>类型）：该集合记录了 JdbcType 与 TypeHandler 之间的关联关系。JdbcType 是一个枚举类型，每个 JdbcType 枚举值对应一种 JDBC 类型，例如，JdbcType.VARCHAR 对应的就是 JDBC 中的 varchar 类型。在从 ResultSet 中读取数据的时候，就会从 JDBC_TYPE_HANDLER_MAP 集合中根据 JDBC 类型查找对应的 TypeHandler，将数据转换成 Java 类型。
- typeHandlerMap（Map<Type, Map<JdbcType, TypeHandler<?>>>类型）：该集合第一层 Key 是需要转换的 Java 类型，第二层 Key 是转换的目标 JdbcType，最终的 Value 是完成此次转换时所需要使用的 TypeHandler 对象。那为什么要有两层 Map 的设计呢？这里我们举个例子：Java 类型中的 String 可能转换成数据库中的 varchar、char、text 等多种类型，存在一对多关系，所以就可能有不同的 TypeHandler 实现。
- allTypeHandlersMap（Map<Class, TypeHandler>类型）：该集合记录了全部 TypeHandler 的类型以及对应的 TypeHandler 实例对象。
- NULL_TYPE_HANDLER_MAP（Map<JdbcType, TypeHandler<?>>类型）：空 TypeHandler 集合的标识，默认值为 Collections.emptyMap()。

在 MyBatis 初始化的时候，实例化全部 TypeHandler 对象之后，会立即调用 TypeHandlerRegistry 的 register() 方法完成这些 TypeHandler 对象的注册，这个注册过程的核心逻辑就是向上述四个核心集合中添加 TypeHandler 实例以及与 Java 类型、JDBC 类型之间的映射。

TypeHandlerRegistry.register() 方法有多个重载实现，这些重载中最基础的实现是三个参数的重载实现，具体实现如下：

```
private void register(Type javaType, JdbcType jdbcType, TypeHandler<?> handler) {
    if (javaType != null) { // 检测是否明确指定了TypeHandler能够处理的Java类型
        // 根据指定的Java类型，从typeHandlerMap集合中获取相应的TypeHandler集合
        Map<JdbcType, TypeHandler<?>> map = typeHandlerMap.get(javaType);
        if (map == null || map == NULL_TYPE_HANDLER_MAP) {
            map = new HashMap<>();
        }
        // 将TypeHandler实例记录到typeHandlerMap集合
        map.put(jdbcType, handler);
        typeHandlerMap.put(javaType, map);
    }
    // 向allTypeHandlersMap集合注册TypeHandler类型和对应的TypeHandler对象
    allTypeHandlersMap.put(handler.getClass(), handler);
}
```

除了上面的 register() 重载，在有的 register() 重载中会尝试从 TypeHandler 类中的 @MappedTypes 注解和 @MappedJdbcTypes 注解中读取信息。其中，**@MappedTypes 注解中可以配置 TypeHandler 实现类能够处理的 Java 类型的集合，@MappedJdbcTypes 注解中可以配置该 TypeHandler 实现类能够处理的 JDBC 类型集合。**

如下就是读取 @MappedJdbcTypes 注解的 register() 重载方法：

```
private <T> void register(Type javaType, TypeHandler<? extends T> typeHandler) {
    // 尝试从TypeHandler类中获取@MappedJdbcTypes注解
    MappedJdbcTypes mappedJdbcTypes = typeHandler.getClass().getAnnotation(MappedJd
    if (mappedJdbcTypes != null) {
        // 根据@MappedJdbcTypes注解指定的JDBC类型进行注册
        for (JdbcType handledJdbcType : mappedJdbcTypes.value()) {
```

```
            // 交给前面的三参数重载处理
            register(javaType, handledJdbcType, typeHandler);
        }
        // 如果支持jdbcType为null，也是交给前面的三参数重载处理
        if (mappedJdbcTypes.includeNullJdbcType()) {
            register(javaType, null, typeHandler);
        }
    } else {
        // 如果没有配置MappedJdbcTypes注解，也是交给前面的三参数重载处理
        register(javaType, null, typeHandler);
    }
}
```

下面是读取 @MappedTypes 注解的 register() 方法重载：

```
public <T> void register(TypeHandler<T> typeHandler) {
    boolean mappedTypeFound = false;
    // 读取TypeHandler类中定义的@MappedTypes注解
    MappedTypes mappedTypes = typeHandler.getClass().getAnnotation(MappedTypes.clas
    if (mappedTypes != null) {
        // 根据@MappedTypes注解中指定的Java类型进行注册
        for (Class<?> handledType : mappedTypes.value()) {
            // 交给前面介绍的register()重载读取@MappedJdbcTypes注解并完成注册
            register(handledType, typeHandler);
            mappedTypeFound = true;
        }
    }
    // 从3.1.0版本开始，如果TypeHandler实现类同时继承了TypeReference这个抽象类，
    // 这里会尝试自动查找对应的Java类型
```

```
        if (!mappedTypeFound && typeHandler instanceof TypeReference) {
            try {
                TypeReference<T> typeReference = (TypeReference<T>) typeHandler;
                // 交给前面介绍的register()重载读取@MappedJdbcTypes注解并完成注册
                register(typeReference.getRawType(), typeHandler);
                mappedTypeFound = true;
            } catch (Throwable t) {
            }
        }
        if (!mappedTypeFound) {
            register((Class<T>) null, typeHandler);
        }
    }
```

我们接下来看最后一个 register() 重载。**TypeHandlerRegistry 提供了扫描一个包下的全部 TypeHandler 接口实现类的 register() 重载**。在该重载中，会首先读取指定包下面的全部的 TypeHandler 实现类，然后再交给 register() 重载读取 @MappedTypes 注解和 @MappedJdbcTypes 注解，并最终完成注册。这个 register() 重载的具体实现比较简单，这里就不再展示，你若感兴趣的话可以参考源码进行学习。

最后，我们再来看看 TypeHandlerRegistry 的构造方法，其中会通过 register() 方法注册多个 TypeHandler 对象，下面就展示了为 String 类型注册 TypeHandler 的核心实现：

```
    public TypeHandlerRegistry() {
        // StringTypeHandler可以实现String类型与char、varchar、longvarchar类型之间的转换
        register(String.class, JdbcType.CHAR, new StringTypeHandler());
        register(String.class, JdbcType.VARCHAR, new StringTypeHandler());
        register(String.class, JdbcType.LONGVARCHAR, new StringTypeHandler());
        // ClobTypeHandler可以完成String类型与clob类型之间的转换
        register(String.class, JdbcType.CLOB, new ClobTypeHandler());
        // NStringTypeHandler可以完成String类型与NVARCHAR、NCHAR类型之间的转换
```

```
        register(String.class, JdbcType.NVARCHAR, new NStringTypeHandler());
        register(String.class, JdbcType.NCHAR, new NStringTypeHandler());
        // NClobTypeHandler可以完成String类型与NCLOB类型之间的转换
        register(String.class, JdbcType.NCLOB, new NClobTypeHandler());
        // 省略其他TypeHandler实现的注册逻辑
    }
```

## TypeHandler 查询

分析完注册 TypeHandler 实例的具体实现之后，我们接下来就来看看 MyBatis 是如何从 TypeHandlerRegistry 底层的这几个集合中查找正确的 TypeHandler 实例，**该功能的具体实现是在 TypeHandlerRegistry 的 getTypeHandler() 方法中。**

这里的 getTypeHandler() 方法也有多个重载，最核心的重载是 getTypeHandler(Type,JdbcType) 这个重载方法，其中会根据传入的 Java 类型和 JDBC 类型，从底层的几个集合中查询相应的 TypeHandler 实例，具体实现如下：

```
    private <T> TypeHandler<T> getTypeHandler(Type type, JdbcType jdbcType) {
        if (ParamMap.class.equals(type)) {
            return null; // 过滤掉ParamMap类型
        }
        // 根据Java类型查找对应的TypeHandler集合
        Map<JdbcType, TypeHandler<?>> jdbcHandlerMap = getJdbcHandlerMap(type);
        TypeHandler<?> handler = null;
        if (jdbcHandlerMap != null) {
            // 根据JdbcType类型查找对应的TypeHandler实例
            handler = jdbcHandlerMap.get(jdbcType);
            if (handler == null) {
                // 没有对应的TypeHandler实例，则使用null对应的TypeHandler
                handler = jdbcHandlerMap.get(null);
            }
```

```
            if (handler == null) {
                // 如果jdbcHandlerMap只注册了一个TypeHandler，则使用此TypeHandler对象
                handler = pickSoleHandler(jdbcHandlerMap);
            }
        }
        return (TypeHandler<T>) handler;
    }
```

在 getTypeHandler() 方法中会调用 getJdbcHandlerMap() 方法检测 typeHandlerMap 集合中相应的 TypeHandler 集合是否已经初始化。

- 如果已初始化，则直接使用该集合进行查询；
- 如果未初始化，则尝试以传入的 Java 类型的、已初始化的父类对应的 TypeHandler 集合作为初始集合；
- 如果该 Java 类型的父类没有关联任何已初始化的 TypeHandler 集合，则将该 Java 类型对应的 TypeHandler 集合初始化为 NULL_TYPE_HANDLER_MAP 标识。

getJdbcHandlerMap() 方法具体实现如下：

```
    private Map<JdbcType, TypeHandler<?>> getJdbcHandlerMap(Type type) {
        // 首先查找指定Java类型对应的TypeHandler集合
        Map<JdbcType, TypeHandler<?>> jdbcHandlerMap = typeHandlerMap.get(type);
        if (NULL_TYPE_HANDLER_MAP.equals(jdbcHandlerMap)) { // 检测是否为空集合标识
            return null;
        }
        // 初始化指定Java类型的TypeHandler集合
        if (jdbcHandlerMap == null && type instanceof Class) {
            Class<?> clazz = (Class<?>) type;
            if (Enum.class.isAssignableFrom(clazz)) { // 针对枚举类型的处理
                Class<?> enumClass = clazz.isAnonymousClass() ? clazz.getSuperclass() :
                jdbcHandlerMap = getJdbcHandlerMapForEnumInterfaces(enumClass, enumClas
```

```
            if (jdbcHandlerMap == null) {
                register(enumClass, getInstance(enumClass, defaultEnumTypeHandler))
                return typeHandlerMap.get(enumClass);
            }
        } else {
            // 查找父类关联的TypeHandler集合，并将其作为clazz对应的TypeHandler集合
            jdbcHandlerMap = getJdbcHandlerMapForSuperclass(clazz);
        }
    }
    // 如果上述查找皆失败，则以NULL_TYPE_HANDLER_MAP作为clazz对应的TypeHandler集合
    typeHandlerMap.put(type, jdbcHandlerMap == null ?
            NULL_TYPE_HANDLER_MAP : jdbcHandlerMap);
    return jdbcHandlerMap;
}
```

这里调用的 getJdbcHandlerMapForSuperclass() 方法会判断传入的 clazz 的父类是否为空或 Object。如果是，则方法直接返回 null；如果不是，则尝试从 typeHandlerMap 集合中获取父类对应的 TypeHandler 集合，但如果父类没有关联 TypeHandler 集合，则递归调用 getJdbcHandlerMapForSuperclass() 方法顺着继承树继续向上查找父类，直到查找到父类的 TypeHandler 集合，然后直接返回。

下面是 getJdbcHandlerMapForSuperclass() 方法的具体实现：

```
private Map<JdbcType, TypeHandler<?>> getJdbcHandlerMapForSuperclass(Class<?> clazz
    Class<?> superclass = clazz.getSuperclass();
    if (superclass == null || Object.class.equals(superclass)) {
        return null; // 父类为Object或null则查找结束
    }
    Map<JdbcType, TypeHandler<?>> jdbcHandlerMap = typeHandlerMap.get(superclass);
    if (jdbcHandlerMap != null) {
        return jdbcHandlerMap;
```

```
    } else {
        // 顺着继承树，递归查找父类对应的TypeHandler集合
        return getJdbcHandlerMapForSuperclass(superclass);
    }
}
```

## 别名管理

在《02 | 订单系统持久层示例分析，20 分钟带你快速上手 MyBatis》分析的 MyBatis 示例中，我们在 mybatis-config.xml 配置文件中使用 `<typeAlias>` 标签为 Customer 等 Java 类的完整名称定义了相应的别名，后续编写 SQL 语句、定义 `<resultMap>` 的时候，**直接使用这些别名即可完全替代相应的完整 Java 类名，这样就非常易于代码的编写和维护。**

**TypeAliasRegistry 是维护别名配置的核心实现所在**，其中提供了别名注册、别名查询的基本功能。在 TypeAliasRegistry 的 typeAliases 字段（Map<String, Class<?>>类型）中记录了别名与 Java 类型之间的对应关系，我们可以通过 registerAlias() 方法完成别名的注册，具体实现如下：

```
public void registerAlias(String alias, Class<?> value) {
    if (alias == null) { // 传入的别名为null，直接抛出异常
        throw new TypeException("The parameter alias cannot be null");
    }
    // 将别名全部转换为小写
    String key = alias.toLowerCase(Locale.ENGLISH);
    // 检测别名是否存在冲突，如果存在冲突，则直接抛出异常
    if (typeAliases.containsKey(key) && typeAliases.get(key) != null && !typeAliase
        throw new TypeException("...");
    }
    // 在typeAliases集合中记录别名与类之间的映射关系
    typeAliases.put(key, value);
}
```

在 TypeAliasRegistry 的构造方法中，会通过上述 registerAlias() 方法将 Java 的基本类型、基本类型的数组类型、基本类型的封装类、封装类型的数组类型、Date、BigDecimal、BigInteger、Map、HashMap、List、ArrayList、Collection、Iterator、ResultSet 等常用类型添加了别名，具体实现比较简单，这里就不再展示，你若感兴趣的话可以参考源码进行学习。

除了明确传入别名与相应的 Java 类型之外，TypeAliasRegistry 还提供了扫描指定包名下所有的类中的 @Alias 注解获取别名配置，并完成注册的功能，这个功能涉及两个 registerAliases() 方法的重载，相关实现如下：

```
public void registerAliases(String packageName, Class<?> superType) {
    ResolverUtil<Class<?>> resolverUtil = new ResolverUtil<>();
    // 查找指定包下所有的superType类型
    resolverUtil.find(new ResolverUtil.IsA(superType), packageName);
    Set<Class<? extends Class<?>>> typeSet = resolverUtil.getClasses();
    for (Class<?> type : typeSet) {
        // 过滤掉内部类、接口以及抽象类
        if (!type.isAnonymousClass() && !type.isInterface() && !type.isMemberClass(
            // 扫描类中的@Alias注解
            registerAlias(type);
        }
    }
}
public void registerAlias(Class<?> type) {
    // 获取类的简单名称，其中不会包含包名
    String alias = type.getSimpleName();
    // 获取类中的@Alias注解
    Alias aliasAnnotation = type.getAnnotation(Alias.class);
    if (aliasAnnotation != null) { // 获取特定别名
        alias = aliasAnnotation.value();
    }
```

```
        // 这里的@Alias注解指定的别名与type类型绑定
        registerAlias(alias, type);
    }
```

## 总结

在这一讲我们重点介绍了 MyBatis 中 JdbcType 与 Java 类型之间转换的相关实现。

- 首先，介绍了 JdbcType 与 Java 类型之间的常见映射关系，以及两种类型之间转换的基础知识；
- 然后，深入分析了 TypeHandler 接口及其核心实现，了解了两种类型转换的原理；
- 接下来，又讲解了 TypeHandler 的注册和查询机制，明确了 MyBatis 是如何管理和使用众多的 TypeHandler 实现；
- 最后，分析了 MyBatis 中的别名实现。

上一页 下一页