南京大学

研 究 生 毕 业 论 文
（申请工程硕士学位）

论文题目	Theta ORM(TORM)-基于 MyBatis 的适应企业快速生产轻量级 ORM 扩展
作者姓名	陈望旭
学科、专业名称	计算机科学与技术系
研究方向	工程硕士
指导教师	胡昊 副教授

2014 年 11 月 13 日

学　　号：MP1333006

论文答辩日期：2015 年 01 月 17 日

指 导 教 师：　　（签字）

Theta ORM(TORM)
-基于 MyBatis 的适应企业快速生产轻量级 ORM 扩展

作　　者: 陈望旭

指导教师: 胡昊　副教授

南京大学研究生毕业论文

(申请工程硕士学位)

南京大学计算机科学与技术系

2014 年 11 月

Theta ORM(TORM)
A Lightweight MyBatis ORM Extension Adapted To Rapid Enterprise Production

Chen, Wangxu

Submitted in partial fulfillment of the requirements for the degree of Master of Engineering

Supervised by
Associate Professor **Hu, Hao**

Computer Science & Technology Department
NANJING UNIVERSITY
Nanjing, China
November, 2014

摘　要

面对繁多复杂的数据库访问层框架，企业在选择时需要在框架使用的灵活性与易用性上做出很多取舍，而 TORM 则是为了兼顾这种取舍而诞生的一款新型框架扩展。

单表操作是企业生产中数据库访问层最普遍的使用场景，因此 TORM 重点考虑对单表操作处理的支持。在此基础上，我们进行 TORM 的设计与开发时，第一步就是定义一个精简的关系运算子集，其中包含了单表操作所需要的单元关系运算以满足绝大多数的场景需要，与此同时在考虑到复杂情形时又能通过子集中已有的运算模拟并不常用的其他运算；第二步是定义具体的对象-关系映射和方法-操作映射规则，并在已经定义好的关系运算子集基础上，限定 TORM 采用的映射关系范围并尽量兼容 JPA 标准，从而在大部分企业应用场景中降低开发成本并提高映射执行效率；第三步是以面向切面编程技术基于 MyBatis 扩展并实现 TORM，保证 TORM 中的映射实现时完全出发于 MyBatis 原有执行链条的顶端，从而使得 TORM 的实现与 MyBatis 的动态 SQL 支持完全解耦，为以后针对这两个层次的扩展打下了良好的基础。

由此可见，TORM 是一层对 MyBatis 的非侵入式加盖，在继承了 MyBatis 原有处理机制的前提下，另外提供了一条全新的纯 ORM 式数据库访问层链路。同时，TORM 不仅仅是原有框架的一个简单扩展，在合理的设计模式下，它更可以做为一个独立框架应对绝大部分的企业级开发场景，并在大部分场景下比其他框架更简洁、高效。

关键词：数据库访问层框架，对象关系映射，面向切面编程，MyBatis，JPA

Abstract

In the face of the complexity of the database access layer (DAL) framework, enterprises need to make difficult choices between flexibility and accessibility. And TORM is a new framework extension in order to fix this problem.

The use of single-table operations is a universal of scenes in enterprise-level DAL, so that TORM shall be focus on single-table operations' supporting. When we consider the design and development of TORM, we can divide it into three steps. Step 1: Define a reduced relation-operation subset which contains some necessary single-relation-operations that can meet most of our needs, and at the same time, when more complex scene appears, it also can solve the problem by proper simulations. Step 2: Define the particular rules of Object-Relation mapping and Method-Operation mapping in order to ensure that some unusual or inefficiency mapping relations are abandoned to reduce the cost of enterprise-level development and increase the executions' efficiency of ORM. Step 3： Use AOP method to extend MyBatis framework and that means all of its operations are based on MyBatis' original execution-chain, what makes decoupling between TORM and MyBatis framework for more extendings of these two layers.

Till now, we can see that TORM is a non-invasive plugin for MyBatis and in terms of extending its own progress of MyBatis TORM provides another brand new DAL chain for the original framework. At the same time, we can't simply regard TORM as a framework extension, but an independent framework which can handle most enterprise-level development situations and do it simpler and more efficient.

Keywords：DAL framework, ORM,AOP, MyBatis, JPA

目 录

第一章 研究背景

1.1 研究动机

数据库是所有商业应用系统的核心所在,数据访问层的开发在系统开发中的重要性也不言而喻。由于几乎现在所有的生产中都采用面向对象语言,ORM(对象关系映射)就显得非常重要。ORM 并不仅仅提高了生产效率,降低了开发成本,同时也在保证了应用数据及数据操作与实际数据库数据及数据操作的一致性的基础上,将双方解耦。

由于关系型数据库已经发展得非常成熟,相关的数据访问层框架也非常丰富全面,通过针对各自系统的具体分析,完全可以轻易找出符合开发需要的 ORM 框架。很多优秀的 ORM 框架在映射上做了非常完备的处理,以满足传统关系型数据库的需要。但是在当前企业级开发过程的数据库设计中,有着与传统关系型数据库设计迥然不同的特点,导致“完备”有时候反而成为了缺点。学术上, 虽然满足 1NF 就是关系型数据库了,但一般讨论到数据库设计,都会要求其达到 3NF 甚至是 BCNF。 实际生产中,由于考虑到访问效率、数据迁移和开发维护成本等方面的因素,往往会对所有的关系 (表)加入逻辑主键以满足 2NF;同时,在非核心数据的设计上会使用大量的冗余,即使是在多关系的数据模型中,也往往会使用非物理形式外键形式(仅仅以属性来做关联标识)。这也就意味着,企业级的数据库设计其实绝大部分情况下是不满足也不需要满足 3NF 的,而是一种在 2NF 基础上的松散关联设计。

与此同时在企业级的应用开发中往往存在这样的情况:95%以上的数据库访问可能只是简单的 增删改查,余下的 5%却是核心业务逻辑中比较复杂的数据处理过程。面对这种情况,一套完整的 ORM 方案就更显得有些臃肿了。为此,我们需要的是一种既能满足快速开发、简单操作无配置化, 又能满足复杂操作可充分灵活定制的数据访问层框架,这也就是本文即将展开的内容。同时,我们将这个框架的解决方案命名 TORM(Theta ORM)。(因为 Java 是目前世界上使用最为广泛的面向对象语言,所以本文将以 Java 为例来探讨、解决研究背景中所讨论的问题,并进行 TORM 的设计与开发。)

1.2 业界概况

业界在数据库访问层框架的选择中有非常流行的两类框架：动态 SQL 型与完全映射型。其中动态 SQL 型以 MyBatis 为代表，完全映射型以 Hibernate 为代表，而 Hibernate 本身映射采用 JPA（Java Persistence API）标准。

根据我们上下文中的分析结果，业界的这两种方案都不能完全适应我们的需要。对此我们新的结论是：使用 MyBatis 强大的动态 SQL 能力，并以 JPA 的标准在 MyBatis 自身的插件机制上再建立一套简单的类 Hibernate 框架。

所以在分析业界相关技术及其背景时，我们将主要从 MyBatis 框架和 JPA 标准来进行参考和分析，而这两种技术的详细介绍将在本文的第二章中给出。

1.3 问题背景

1.3.1 现有解决方法

在目前使用 Java 语言的数据库访问层框架中大致分为两类：以动态 SQL 为方向,如 MyBatis。这一类框架支持普通 SQL 操作,大部分在执行前还会进行必要的优化、连接控制等,像 MyBatis 这类业界比较优秀的同时还能进行默认或自定义的 POJO(Plain Old Java Objects 的映射)。但由于其本质还是动态 SQL 框架,所以并没有直接实 现 SQL 本身的封装,开发者需要对 SQL 进行直接或者以模板为基础的定义； 以完整的 ORM 为方向,如 Hibernate。这一类框架则按照一些预定义的映射规则接口集合进行关系的属性与操作映射,当定义了 DMO(数据模型对象)之后,在相应资源中定义数据库关联的具体映射规则,就可以使用 DMO 进行数据访问,并不需要再另外编写 SQL。但是当其面对复杂的数据操作过程时,由于关系结构与对象结构本质上的不一致,往往会导致操作性能急剧下降。如果开发人员对此类情况经验不足或设计有所疏漏,实际应用中的数据模型与数据库中的数据模型不一致就变得很难避免且难以发现,大大增加了开发和维护的成本。

不论是哪一类框架，在应用系统中对应的抽象用例基本一致，只是在 SQL 执行前的若干步骤做出不同取舍，这些可以简单的归纳如下图：

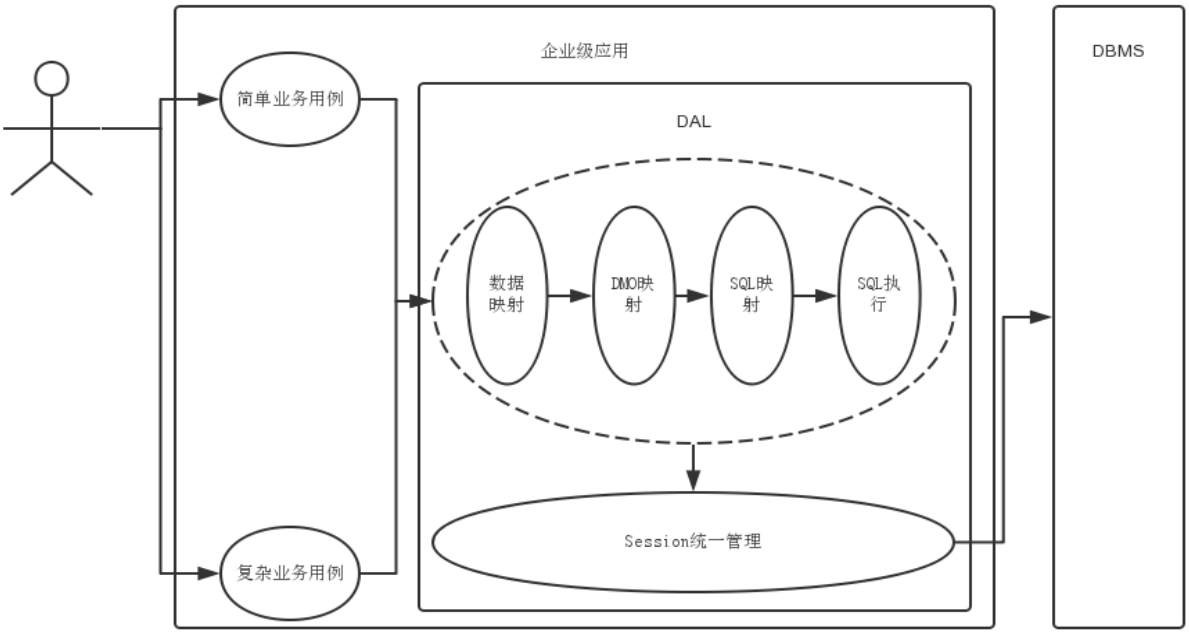

图 1-1

显然，对于动态 SQL 而言，则是将 SQL 映射这一模块开放给开发者自行管理，而纯 ORM 框架则也在其定义的规则下将 SQL 映射统一处理了。于此同时，我们还需要注意的是，这样的 DAL（数据库访问层）结构，所有的业务采用统一的 DAL 入口，这使得我们无法在机制上对简单业务用例与复杂业务用例采取区别对待从而实现最优的方案匹配。如果我们区分不同复杂度的业务用例，采取不同的 DAL 入口，并在 DAL 流程中采取可拆卸的适配形式，这两个问题将很容易推出我们将要用以解决问题的新用例模型：

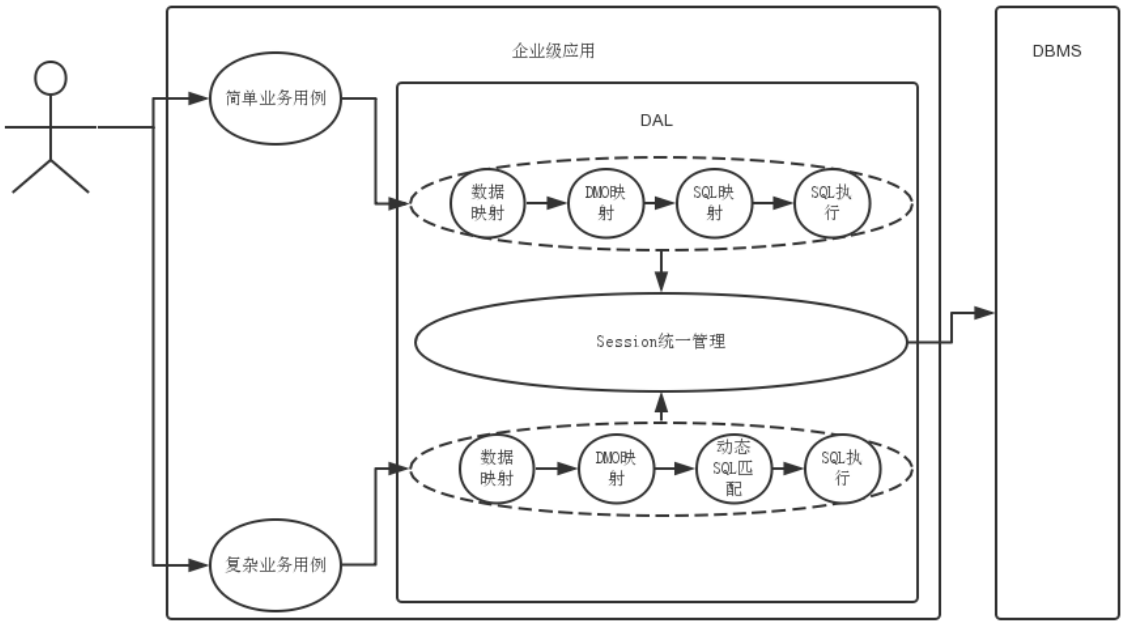

图 1-2

1.3.2 现有问题分析

如上一节中所述,一个矛盾产生了:

- 如果我使用动态 SQL 向框架,那么针对简单的大多数场景,为了实现数据操作,我们要对应用 中定义的 DMO 编写动态 SQL 生成实例,而这些实例在应用中生成出来的具体 SQL 却是大量类 似的基础 SQL,它们数量庞大、难以维护;

- 如果我们使用 ORM 向框架,那么针对复杂的较少数,我们很难让数据库最终执行高效(而且大 量的事实告诉我们很多时候这不仅不高效甚至不正确)的操作,虽然有些 ORM 框架,如 Hibernate,也集成了部分动态 SQL 功能,但并不是特别理想,无法和传统的动态 SQL 向框架相比。

所以目前摆在我们面前的事实是,虽然我们在两方面都有成熟而强大的解决方案,但是面对实际生产的时候我们却不能仅仅通过选择来实现我们的全部需求。又由于已有的良好基础,我们解决问题的方案很容易得出,就是针对其中一种框架,添加另一类框架的功能子集以满足我们的需求,而对此我们选择也有两条:

- 基于 Hibernate 添加动态 SQL 支持。Hibernate 自身带有其定义的 HQL,但 HQL 只是语法类似 SQL,本质上是基于面向对象进行设计的,所以必要路径还是在 Hibernate 的 ORM 上。那我 们扩展的方式只有通过对 Hibernate 底层

的调用执行 SQL,比如数据库会话或数据库连接。但这样我们除了 SQL 操作本身的封装外,还需要重新定义 POJO 的映射规则,并且无法利用到 Hibernate 本身的优化机制;

- 基于 MyBatis 添加 ORM 支持。由于 MyBatis 本身支持 POJO 的高级映射,所以从核心层面上讲我们需要的只是将 SQL 操作封装成对象操作,相比对 Hibernate 的扩展复杂度和成本都大大降低了。而且对 MyBatis 的扩展是添加 ORM 部分,而完成映射之后,再进行实际数据访问都可 以直接通过 MyBatis 已有的 SQL 加载和执行机制,在性能和安全性上都不需要额外的开销。

1.3.3 优化可能与结论

基于我们对现有问题的分析,在 MyBatis 现有框架基础之上加盖完整且可替换的 ORM 支持是目前可见的最为理想的解决案。

对于 Torm 中需要进行的加盖及改造也依据上文分析需遵循如下原则:

- 简单业务用例:大部分为单表操作,Torm 对其进行简单有效的封装;
- 复杂业务用例:可能涉及到复杂或性能要求较高的数据访问操作,Torm 为其提供 MyBatis 原有的实现接口。

1.3.4 需要解决的问题与挑战

确定了研究方向之后,也并不是就可以直接展开对 MyBatis 的扩展工作,由于 MyBatis 本身没有 ORM 功能,而上文中我们也简述了企业开发中数据访问层的特殊性,所以我们要做的不仅是实现, 更包含了关系运算子集的定制、对定制子集的适用性证明以及为一些不属于关系代数范畴的数据库 特性定制扩展的预留设计。按照研究进行的步骤,有一些核心的问题需要我们去解答:

- 如何定义满足企业需求的关系运算子集?
- 如何证明该集合在本文指定条件下与传统数据操作包含的关系运算集合等价?
- 如何在数据访问层定义满足上述关系运算子集的 ORM 映射规则?
- 如何设计并实现将 POJO 映射为封装 SQL 操作提供传参的关系对象?
- 如何在不过大地影响 Mybatis 原有性能的前提下加入 ORM 层封装?
- 如何使这种改造具有较好的扩展性以便解决以后可能需要处理的特殊情况?

1.4 本文结构

基于上文提出的问题以及我们研究的目的,本文的结构大致可以分为如下部分:

- 第一章:研究背景。该章节介绍了研究的大致背景、业界在该领域采用的一般解决方案以及我们对其初步的分析，并提出了需要解决的问题和可能的方案设想。

- 第二章:分析与定义。该章节对 TORM 的设计范围在数学上做出了推导，并对基本的映射规则做出了分析与定义。

- 第三章:实现与扩展。该章节包含了对 TORM 系统设计与实现的描述，从模型映射与 SQL 生成这两个主干分支介绍 TORM 的主要结构。

- 第四章:研究成果。该章节总结了 TORM 已实现功能，并对以后的相关扩展做出简单分析和展望。

第二章 相关工作

2.1 ORM 业界规则与技术框架

2.1.1 MyBatis

MyBatis 早年是 Apache 的一个开源项目 iBatis, 2010 年这个项目由 Apache Software Foundation 迁移到了 Google Code,并改名为 MyBatis。

MyBatis 是支持普通 SQL 查询,存储过程和高级映射的优秀持久层框架。MyBatis 封装了几乎所 有的 JDBC 代码和参数的手工设置以及结果集的检索。MyBatis 使用简单的 XML 和注解用于配置和原 始映射,将接口和 Java 的 POJOs 映射成数据库中的记录。这种 POJO 映射可以被 TORM 直接使用于 SQL 封装的过程中,而且几乎不需要引入新的改变,唯一可能要添加的过程只是对 POJO 属性映射 的重命名而已。大部分使用 MyBatis 的开发者都会使用 MyBatis 的 XML Mapper 映射定义形式,但这种传统方法并不能满足我们的需要,因为 MyBatis 在 SqlSessionFactoryBuilder 中加载 XML 定义是封闭的,无法满足我们对映射定义随时加载的需求。所幸的是,MyBatis 还提供了接口形式的映射定义(即上文所说基于注解的配置和原始映射),也就意味着如果我们能自行生成相关动态类的话, 我们就可以随时在 JVM ClassPool 中加载其映射定义接口,从而加载 TORM 中的映射实现。

同时,MyBatis 另外一个重要的特性就是它提供了丰富的插件接口,使得我们原本基于 MyBatis 的实现有了轻松扩展的可能。Mybatis 采用责任链模式,通过动态代理组织多个拦截器(插件),通 过这些拦截器可以改变 Mybatis 的默认行为。而我们的 ORM 实现最终是通过 MyBatis 的动态 SQL 生成 接口实现,也就是说,在拼接完成 SQL 之后的所有一系列应用动作和数据库动作我们都可以通过 MyBatis 插件的形式进行扩展。并且这些扩展都是可拆卸的,只要设计良好,甚至可以做到与 TORM 本身无耦合,那也就意味着相对而言,我们为 TORM 进行扩展甚至可以直接使用一些现有的 MyBatis 插件源码。

2.1.2 JPA

JPA 全称为 Java Persistence API。JPA 通过 JAVA 注解(或 XML)描述 ORM,并将运行期的实体对象持久化到数据库中。JPA 相对 JDBC 或其他形式 SQL 封装主要优势包括:

- 标准化:JPA 是 JCP 发布的 J2EE 标准之一,因此任何声称符合 JPA 标准的框架都遵循同样的架 构,提供相同的访问 API,这保证了基于 JPA 开发的企业应用能够经过少量的修改就能够在不同的 JPA 框架下运行;

- 容器级特性支持:在 JPA 框架中支持大数据集、事务、并发等容器级事务,这使 JPA 超越了简单持久化框架的局限,在企业级应用发挥更大的作用。JPA 的这个特点对 TORM 来说非常重要,在使用 JPA 定义映射的前提下,如果使用 TORM 的应用切换到其他支持 JPA 标准的数据库访问层框架上,几乎可以不改动任何业务代码,而只需简单更新一下部分框架配置;

- 简单方便:设计 JPA 的主要目标之一就是提供更加简单的编程模型。而事实上 JPA 本身可以说只是一个(或一组)标准,在企业应用的业务代码中,我们会使用符合 JPA 标准的注解或 XML 定义实体,而 TORM 只是对其进行实现,使得 TORM 不用过多的考虑特殊业务场景需求,简化了 TORM 本身需要的设计和开发流程。在 JPA 中创建实体和创建 Java 类一样简单, 没有任何的约束和限制,只要使用 javax.persistence.Entity 进行注释,JPA 的框架和接口也都非常简单,没有太多特别的规则和设计模式的要求,开发者可以很容易的掌握。JPA 基于非侵入式原则设计,因此可以很容易的和其它框架或者容器集成;

- 查询能力:JPA 的查询语言是面向对象而非面向数据库的,它以面向对象的自然语法构造查询语句,可以看成是 Hibernate HQL 的等价物。JPA 定义了独特的 JPQL(Java Persistence Query Language),JPQL 是 EJB QL 的一种扩展,它是针对实体的一种查询语言,操作对象 是实体,而不是关系数据库的表,而且能够支持批量更新和修改、JOIN、GROUP BY、 HAVING 等通常只有 SQL 才能够提供的高级查询特性,甚至还能够支持子查询。对于这一条 特性,实际上 TORM 是不会使用到的,因为我们设计 TORM 的初衷就是屏蔽简单且具有通用 性的查询及非查询 SQL。对于 HQL 或 JPQL 这种查询语言,我们完全可以弃之不用,而是在 面对复杂查询操作的时候直接使用 MyBatis 的动态 SQL 能力,执行我们需要的数据库操作;

面向对象的高级特性:JPA 中能够支持面向对象的高级特性,如类之间的继承、多态和类之间 的复杂关系,这样的支持能够让开发者最大限度的使用面向对象的模型设计企业应用,而不 需要自行处理这些特性在关系数据库的持久化。同 JPA 的查询能力一样,虽然这本身是 JPA 得一个诱人优点,但对于 TORM 存在的场景,这种特性也是不必要的。DMO 之间复杂的继 承、多态等关系会使得 TORM 的关系映射设计变得非常复杂,而 TORM 的实际场景绝大多数 是不会使用到这些关系的。而且对于 JAVA 而言,如果使用注解形式的 JPA 定义,也会存在不同版本 JDK 中对注解继承原则不同解释的问题,影响其 TORM 的兼容性。

2.2 Javassit-源码级别 AOP 技术

Javassist 是一款开源的分析、编辑和创建 Java 字节码的类库，它由东京工业大学的数学和计算机科学系的 Shigeru Chiba（千叶 滋）所创建。它已加入了开源代码 JBoss 应用服务器项目,通过使用 Javassist 对字节码操作为 JBoss 实现动态 AOP 框架。

关于 java AOP 或字节码的处理，目前有很多工具，如 asm、bcel 等，不过这些都需要直接跟虚拟机指令打交道。在考虑到要尽可能使得切面逻辑通俗易

懂且容易管理，则我们可以选择采用 javassist：Javassist 的主要优点就在于简单，而且快速；直接使用 Java 编码的形式，而不需要了解虚拟机指令，就能动态改变类的结构，或者动态生成类。

TORM 之所以能不使用自定义 SQL 模板文件或自动模板生成工具就可以执行简单常用的数据操作，是因为我们在设计 TORM 时，通过 AOP 技术将 SQL 生成与执行的实现逻辑事先拼接为 MyBatis 在 JVM 中可识别的类实例（文件）,而在 TORM 中实现该程度的功能时，我们选用了 Javassist 这种源码级别的 AOP 技术，而不是更普适的各种字节码技术，主要是出于两方面的考虑：

1、更简单的处理逻辑：源码的拼接比字节码容易理解的多，而我们要加塞的逻辑并不简单，更易理解的 AOP 方式对整体质量都是很好的保障；

2、尚可接受的性能：源码级别的 AOP 需要再经过预编译过程，性能自然会有所下降，但是在考虑过 TORM 的加载流程后，可以看到数据对象的懒加载模式使得这一开销较大的过程在整个应用的运行周期中只会在启动时执行一次，这种对日常使用并无影响的时间代价无疑是可以接受的。

第三章 分析与定义

3.1 关系运算子集定义

本文所讨论的关系运算限定为关系代数中定义的原始运算和域计算运算。而刚刚所指的关系代数不是 Augustus De Morgan 于 1860 年为代数逻辑提供的关系代数,而是计算机科学中一般所指的一阶逻辑分支,即闭合与运算下的关系的集合。而关系运算的本质就是其作用于一个或多个关系上来生成另一个关系。

在 1970 年 E.F. Codd 发表数据的关系模型之后,关系代数被广泛地用来设计和定义数据库查询语言或语言规范。由于我们只是希望利用其在数据库查询语言中的指导性作用来定义一组具有广泛 适用性并且正确合理的关系运算子集,从而定义我们需要的 ORM 子集,所以本文中我们并不关心关系代数的系统论证,将更多地去利用基础的关系代数定义去证明我们选取的运算子集能适用于一般的企业级开发,并在用所含运算模拟未包含运算时进行优化设计。

关系代数中定义的原始运算定义并不唯一。因为对于任何代数而言,一些运算是原始的,另一些运算则可以通过原始运算来定义。而这些运算的选取,除了需要符合常人的思维习惯之外,某种程度上来说是随机的。 Codd 对他的代数原始运算做出的选取为“选择”、“投影”、“笛卡尔积”、“并 集”、“差集”和“重命名”。

在此基础上,我们根据一般 SQL 操作定义的情况,将这 7 个运算分为两类：

- 传统集合运算:并集,差集(交集可用差集定义)；

- 关系专用运算:选择,投影,笛卡尔积,重命名;

而根据我们实际的设计需要,我们又将所有这 7 个运算按操作对象集合分为一元关系运算与多 元关系运算两类:

- 一元关系运算:选择,投影,重命名;
- 多元关系运算:并集,差集,笛卡尔积。

本章中,我们将对这七类原始运算进行分析,并根据我们对 TORM 的需求来取舍,并定义我们需要的运算子集。

3.1.1 一元关系运算(单表)集合定义

1)投影

定义:

投影是写为$\pi_{a_1,...,a_n}(R)$的一元运算,这里$a_1,...,a_n$是属性名字集合。这种投影的结果为当 前所有在R中的元组被限制为$a_1,...,a_n$的时候所获得的集合。

分析:

根据定义,投影可以简单理解为数据库中对表字段的选取,而这是所有非聚集查询运算中必然包含的操作之一,这些都分装在已经定义的 SQL 当中,不再需要我们处理。而当进行 ORM 时,我们也需要根据我们对象实体的属性在数据库表字段的基础上再进行一次投影,使得 DMO 中仅包含实体所需要的属性值。

结论:

投影运算作为关系运算的必要原始运算,将在我们的关系运算子集中定义,但我们不会在系统 中实现它,因为他已经被 SQL 本身完整地实现了。

2)选择

定义:

广义选择是写为$\sigma_{\varphi}(R)$的一元运算,这里的φ是由正常选择中所允许的原子和逻辑算子∧(与)、∨(或)和¬(非)构成的命题公式。这种选择选出R中使φ成立的所有元组。

分析:

选择运算是所有数据库查询操作的基础。从定义中可以看出,在关系代数中选择运算只包含逻辑谓词,而实际数据库操作中可能会包含一些业务需要的非逻辑谓词。这也是因为关系代数与实际数据库在查询结果上有本质的区别:关系代数的关系运算结果为集合,集合为无序的; 实际数据库中的操作结果为数据序列,这些序列或显式或隐式地包含了一定的有序关系。

结论:

考虑到实现的复杂度,在我们的关系运算子集定义中,不定义非逻辑谓词的选择操作(也无法定义)。非逻辑谓词的选择操作我们则会通过数据库访问层插件或为查询结果对象添加 AOP 切面来实现。

3)重命名

定义:

重命名是写为$\rho_{a/b}(R)$的一元运算,这里的结果同一于R,除了在所有元组中的b字段被重命名为a字段之外。它被简单的用来重命名关系的属性或关系自身。

分析:

重命名在 Codd 提出他定义的关系代数原始运算时被忽略了,而实际上在对不同的关系进行运算时,关系间的重名属性会导致运算的定义存在二义性,所以重命名在 ISBL 提出时被显著地包括了。同样,在实际数据库中在进行数据库操作时,为了避免表属性同名的问题,也会使用重命名操作。不过我们在进行 ORM 时,操作的表空间对象限定为与应用一一对应,而同一个表空间中不会有同名表,所以只需采用加上了表名的字段全路径就可以区别。

但实际上,关系属性其实没有关系的特征,换句话说关系属性本身和关系之间的关联是松散的,那也就意味着在数据库操作中的字段全路径其实本身也是经过了默认的重命名运算。所以我们在 ORM 时对 DMO 属性和数据库字段的映射其实也是为其做了一次重命名运算。

结论:

重命名运算也是关系运算的必要原始运算,将在我们的关系运算子集中定义,我们将在系统中提供其默认的实现,更复杂的重命名运算将作为扩展另作实现。

3.1.2 多元关系运算分析及在该场景下的处理方式

TORM 主要是针对企业级开发在单关系(单表)处理时的操作简化,并不需要对多元关系运算提供完整地支持。虽然在现有的 ORM 框架中,多元关系操作一直饱受诟病,但 TORM 作为一个标准 ORM 框架,并考虑到其今后的扩展性,我们需要在定义其关系运算集合时考虑到多元操作。即使我们目前的实现并不一定要包含这些运算,但我们需要保证,在一个可切入的扩展层面上 TORM 对 多元操作而言是具有良好扩展性的。

2.1.2.1 多元运算的应用场景分析

笛卡尔积:

关系代数中的笛卡尔积定义与集合论有所不同,这里的元组是平坦的、无子元组的(可参考 1NF 的定义)。集合论中n元组与m元组的笛卡尔积是 2 元组,而

关系代数中他们的笛卡尔积把这个2元组平展为$n+m$元组。形式上, $R \times S$被定义为：

$$R \times S = \{r \cup s | r \in R, s \in S\}$$

而在关系代数中, 我们一般不会使用到原始的笛卡尔积运算, 因为如上式定义的笛卡尔积运算中两个关系不能有公共属性, 那这样的运算结果对业务而言是没有意义的。绝大多数的场景下, 我们会选择具有至少一个公共属性的两个关系进行运算, 而这就是我们一般所说的连接(join), 记为$\bowtie$。 在关系代数中, 连接还有一种特殊形式。设θ是在集合中的二元关系, $\{a, b\}$是关系属性, v是常量, 当连接$R \bowtie S$满足逻辑谓词$a\theta b$或$a\theta v$时, 我们称这种连接为θ—连接, 写为$R \overset{\bowtie}{_{a\theta b}} S$或$R \overset{\bowtie}{_{a\theta v}} S$。但在实际数据库中并不需要也往往没有定义$\theta$—连接, 因为我们可以通过将连接退化为笛卡尔积再进行用等价谓词进行选择来模拟θ—连接, 记为：

$$R \bowtie_\theta S = \sigma_\theta(R \times S)$$

除此之外, 我们将连接按照算子运算对象所指关系的属性投影方式, 可将其分为连接(内连接)与外连接两类, 出去刚刚所说的自然连接, 其中包含：

- 半连接:分为左内连$R \ltimes S$和右内连$R \rtimes S$。半连接的结果只是在S中有在公共属性名字上相等的元组所有的R中的元组。其形式定义如：

  $$R \ltimes S = \{t : t \in R, s \in S, fun(t \cup s)\}_。$$

  半连接可被自然连接模拟, 取$a1, ..., an$是R的属性名, 则有：

  $$R \ltimes S = \prod_{a1,..,an} R \bowtie S$$

- 反连接:反连接记为$R \triangleright S$。反连接的结果是在 S 中没有在公共属性名字上相等的元组的R中的那些元组。反连接还可以定义为半连接的补集：

  $$R \triangleright S = R - R \ltimes S$$

  所以半连接也可以被自然连接加投影模拟。

- 外连接:外连接包括左外连、右外连、全外连, 它们是由内连接结果元组加上通过向一个操作 数的未匹配元组扩展上另一个操作数的每个属性的“填充”值而形成的元组组成。根据外连接的定义, 它也可以通过自然连接加投影进行模拟。

由此可见, 所有的连接运算都可以由基础的自然连接和基于自然连接的选择和投影来模拟, 也就是说, 如自然连接满足某条件, 那用该自然连接模拟其他连接满足某条件的步骤不会多于一次额外的选择及投影。那么我们在以后的讨论中将只分析自然连接的情况。

并、差集

关系代数中的并集和差集运算与集合论中的并集和差集运算定义一致,但添加了一个附加条件,即需要参与运算的两个操作数关系同构。所以,当仅仅讨论关系代数时,并、差操作在集合论中的运算律,在关系代数运算中存在包含投影的子式时,分配律和幂等律将不再适用。但实际数据库中的情况则较为理想化,因为数据库中需要做集合运算时,往往都是有人为的先天限制的,不同构的关系间的集合运算对业务本身没有意义,而且在存在并、差的运算中,其包含投影的运算子式也必然是对两个异构的关系进行同构映射,这时候的并、差运算就退化为集合论中的并、差运算。

而我们在 ORM 中,即使数据库中表同构,因为对象实体并不一定同构,所以我们也将其视为异 构,因此我们也不会对其进行并、差运算。这就意味着我们的集合运算算子永远出现在同一个关系 的两个选择之中,因此根据分配律能很简单的推导出:

$$1.\sigma_a(R) \cup \sigma_b(R) = \sigma_{a \vee b}(R)$$
$$2.\sigma_a(R) - \sigma_b(R) = \sigma_{a \nrightarrow b}(R)$$

所以,在本文讨论的所有关系运算中的集合运算,都可以通过合并过逻辑谓词的选择运算来模拟,则本文不再讨论并、差集等关系集合运算算子。

2.1.2.2 关系运算的取舍分析

上文在并、差集的讨论中我们已经分析了在特定情况下的使用一元关系运算对多元关系运算的可模拟性。但这还是代数层面的模拟，而在此层面上，笛卡尔积相应的各种关联操作无法通过纯粹的多次选择或投影来模拟。这是不是意味着我们的关系运算子集中必须包含关联这样的多元运算来满足我们的需求并保证日后的扩展性呢？

一次多元运算和多次一元运算的等价关系

事实上，除了代数层面的等价关系外，如果我们不去过分追求高效、简洁的逻辑表述和高性能的数据库实现，而是考虑到企业级应用中复杂关联查询的稀有性，我们还有另外一个方案：通过单次一元关系运算的运算结果来重组后继一元关系运算的逻辑谓词，并在最终结果中使用之前已运算的中间结果拼接成最后的关系结果。

除此之外，如果满足以下两点假设：

- 将关系R与关系S中最大属性元组$a_1, ..., a_n$及其任意非空真子集视为同一运算结果；
- 对于关系R与关系S的所有关联运算都可以通过θ—关联中不同的逻辑谓词来模拟实现。

则我们可以做出以下推导：

1. 定义一次运算

$$\forall(R,S), R \bowtie_\theta S = \{r \cup s : r \in R, s \in S, f_\theta(t \cup s)\}$$

2. 假设一种拆分

$$f_\theta(t \cup s) = f_{\theta_1}(t,s) \wedge f_{\theta_2}(t) \wedge f_{\theta_3}(s)$$

3. 并找到使其满足的对应选择

$$\begin{cases} f_{\theta_1}(t,s) = f_{\phi}(t,s) \\ f_{\theta_2}(t) = f_{\phi_1}(t) \\ f_{\theta_3}(s) = f_{\phi_2}(s) \end{cases}$$

4. 替换定义中的元素

$$\exists R \bowtie_{\theta} S = \{r \cup s : r \in R, s \in S, f_{\phi}(t,s) \wedge f_{\phi_1}(r) \wedge f_{\phi_2}(s)\} = \sigma_{\phi}(\sigma_{\phi_1}(R) \times \sigma_{\phi_2}(S))$$

5. 根据选择和叉积的分配律，并析取逻辑谓词，可得

$$R \bowtie_{\theta} S = \sigma_{\phi \wedge \phi_1}(R) \times \sigma_{\phi \wedge \phi_2}(S)$$

显然，当满足上文中两点假设时，连接操作可以分解为两个选择结果的叉积。对于关系型数据库而言，这样的模拟运算是无法完成的，因为有两点限制：第一，最终运算式中包含了叉积运算；第二，找出的替代逻辑谓词中同时包含两个关系的逻辑条件。但是我们可以采取非常简单的措施来规避这一问题。

- 对于逻辑谓词：先分别进行无耦合的选择运算，在运算结果的选择中，对另一组结果进行逐条遍历并判断，模拟叉积运算；
- 对于最终的叉积：
  - 内半连接：退化为单关系与常量关系的叉积，即不需要进行叉积运算；
  - 自然连接与外连接：可以暂时不进行叉积运算，而是在系统中使用时通过遍历来获取某一条结果元素。

多元运算和与其等价的一元运算的实际时空复杂度分析

时间复杂度：

因为所有复杂的关系运算最终都由有限的原始运算组成，所以分析其复杂度时，我们需要首先定义原始运算的复杂度，而数据库中的时间复杂度主要来自于选择与叉乘，因此不妨做出如下设定：

- 选择时间复杂度：$T_{\sigma}(R)$
- 笛卡尔积时间复杂度：$T_{\times}(R,S)$

则对于真正的连接而言其复杂度为：

$$T_{\bowtie(R,S)} = T_{\times}(R,S) + T_{\sigma}(R \times S)$$

而对于上文中使用多次选择模拟的连接运算中，设

$$m = count(\sigma(R)), n = count(\sigma(S))$$

则有对于上文中使用多次选择模拟的连接运算复杂度为：

$$T_{\bowtie'(R,S)} = nT_{\sigma}(R) + mT_{\sigma}(S)$$

可见，一般情况下，两种方式的复杂度并无明显可比性。但显然，当R、S有一个或者两者都较小时，有

$$T_{\bowtie(R,S)} < T_{\bowtie'(R,S)}$$

而当m和n，即选择结果集合规模远小于R、S，且原始关系规模很大时，反而有

$$T_{\bowtie(R,S)} > T_{\bowtie'(R,S)}$$

（如果生产经验比较丰富，可以看出这是一种很常见的数据库访问优化的背景）

一般而言，除了数据分析的场景之外，大部分应用场景都是在大量的数据中找出有限的条目，这种情况下，我们的模拟运算是可以接受甚至更优的。

空间复杂度：

同空间复杂度一样，由于选择操作的拆分，空间复杂度上同样会出现因为关系R与关系S本身的规模及其选择结果的规模差别而产生的复杂度差别，并这种差别与时间复杂度分析中是同趋的，即一般情况下模拟的开销略大，而对选择结果远小于原始关系规模的情况则模拟方法占优。

由于我们的应用场景并不适用大量数据的分析和筛选，而除去这种场景，现在计算机的存储空间一般可视为充足的，所以对于其空间复杂度这里不再多做分析。

3.1.3 小结

经过以上分析，在 TORM 的关系运算子集中，我们有如下结论：

- 保留一元关系运算，即投影、选择和重命名;
- 多元关系运算中的集合运算不保留，通过运算中算子的逻辑谓词合并来实现;
- 笛卡尔积及相关连接运算不保留，简单的连接运算通过一元关系运算模拟，或在 TORM 的扩展级中通过插件形式实现，复杂的连接运算则直接通过框架的源生动态 SQL 功能实现。

3.2 ORM 定义

本节中，我们将会对 TORM 的对象关系映射做出定义，而定义的基础是第一章的关系运算子集定义，定义的依据则是 JPA 现有的 ORM 标准，其中包括其注解、接口和解释。需要注意的是因为我们只是对一元运算进行了定义并将对其采取映射，所以只会用到 JPA 的一个较小子集，同时，我们采用 Apache 旗下的 OpenJPA 文档作为 JPA 的解释入口。

3.2.1 映射规则的定义

1、属性数据类型

依然是依据尽量广泛适用且简单的规则，在做类型映射时，我们原则上参考三条规则：

- 永远只考虑从对象属性类型向数据库数据类型的转化。相反的，对象属性类型中不存在的类型也不应该存在于 TORM 的映射规则中;
- 永远只使用应用框架语言的非基础类型。如果使用基础类型作为映射源，会使 Java 中的泛型处理极大的复杂化;
- 永远只使用原子对象属性类型进行映射。因为数据表中的字段根据 1NF 定义应该是不可拆分的数据内容，所以做对象属性映射时，我们需要将对象实

体的属性定义拆分至原子级别。如果业务中需要使用不同层次设计的对象实体，应当对映射源实体进行再次封装；如果数据库中字段数据内容具有特殊格式且含有有业务意义的拆分可能，应当在应用上层添加相关解析代码。

2、数据模型对象定义约束

根据 Java 本身的面向对象特性以及 ORM 的一些特点，我们对实际的映射数据模型对象进行定义时，我们需要保证映射的源端类满足：

- 非虚类，因为映射汇端的数据库表为存在实体，所以源端对象也必须可实例化；
- 继承链末端，因为 Java8 之前对注解继承的机制本身不完善，所以在使用 TORM 这样基于注解的数据库访问层解决方案时，考虑到不同 JVM 版本的兼容性问题，我们应该避免源端类的注解被显式或隐式的继承，又由于需要保证数据的原始性，所以源端类也不该被继承以注入不同的实现；

3、操作映射的定义与形式

操作映射其实本身并不属于 ORM 定义的一部分，但最终我们的 ORM 实现必须依赖数据库操作执行，所以我们需要定义一组从对象操作到数据库操作的映射，从而实现最终的数据库操作。数据库操作不与关系代数运算一一对应，实际上 SQL 中的查询与非查询操作都是根据关系代数的理论规则来定义而通过已经产品化的 RDBMS 来实现的。

因为在第一章的关系运算子集定义中只定义了基于单关系的运算，所以查询操作的结构相对于一般的极大的简化了。而对于包含查询从句的非查询操作，因为其在对象映射关系上的复杂性我们亦不予考虑，所以无论是查询或是非查询操作，我们都可以简单的以以下形式描述：

```
[操作符]+[属性集]+[介词集]+[谓词集]
```

具体的操作实现我们将在 TORM 的系统实现中详细描述。

3.2.2 属性映射定义

1、 实体

描述：

这里的实体指的是元数据(MetaData)中的实体。实体是 ORM 的核心，实体是对关系的业务抽象，他对应了在应用系统中的对象及其属性。而我们在 TORM 中针对实体本身不再进行查询操作，所以可以将应用中的 DMO 对应为我们需要的实体，反之亦然。同样，因为上述原因，实体本身将只做定义，而没有相关操作的实现，所以 TORM 中我们将其定义为 DMO 的逻辑标记，没有相关参数。

注解级别：

类级

注解属性：

无

注解形式：

```
@Entity
```

```
class public ClassName();
```

2、表

描述:

表处于对象关系映射的过渡节点，是关系在数据库中的最重要体现形式，是关系映射到对象过程中的实际访问客体，所以 TORM 中需要定义实体的表注解，以标明其映射关系，对应到相应的实际关系中去。

注解级别:

类级

注解属性:

- name：表名，标注该实体映射对应数据表，必需；
- schema：表空间命，标注数据表所在表空间，缺省为当前数据库连接使用表空间。

注解形式:

```
@Table(name="TableName",schema="SchemaName")
class public className();
```

3、列

描述:

列对应于关系中的关系属性，而其在对象中表型形式为对象的成员属性。列是元数据中存储数据的最小单位，而在 TORM 中的列注解将定义其数据内容相关标注，将关系中关系属性的特征，如主键，分离为其他注解，使对象成员属性与表字段有一般性的对应关系。

注解级别:

域级

注解属性:

- name：列名，标注该实体映射中对应数据字段名，必需；
- length：最大长度，该字段数据的最大内容长度；
- nullable:是否可为空，该字段值是否可为空值。

注解形式:

```
@Table(name="TableName",schema="SchemaName")
class public className(){
@Column(name="ColumnName",length="255",nullable=true)
private Object field;}
```

4、主键

描述:

注解注解用于描述对象中映射属性的主键描述，而非键值本身。所以该注解也是一种标识行注解，依赖于列注解，且不含任何参数。

注解级别:

域级

注解属性:

无

注解形式

```
@Table(name="TableName",schema="SchemaName")
class public className(){
@Id
@Column(name="ColumnName",length="255",nullable=true)
private Object field;}
```

5、键值生成策略

描述:

在执行对象持久化操作时，每个属性到数据库字段的映射会存在一个自身的生成策略。默认的持久化策略为直接获取对象中的属性值，这就不需要额外的定义。而有时，我们希望对部分字段的生成采用某种规则进行特殊处理，这时候就需要键值生成策略注解来标注这种规则。

注解级别:

域级

注解属性:

- strategy:生成策略，选取自预定义的生成策略集合，标注该域使用的生成策略;
- AUTO:默认策略，应用容器或其他调用者主动赋值;
- SEQUENCE:使用序列生成，需要 Oracle 等支持序列的数据库系统支持;
- TABLE:使用表字段生成，需预定义特殊生成所需表及生成字段;
- generator:生成器名，如生成策略需指定生成器生成，则需该属性标注。

注解形式:

```
@Table(name="TableName",schema="SchemaName")
class public className(){
@Id
@GeneratedValue(strategy=GenerationType.SEQUENCE,generator="Generator
Name")
@Column(name="ColumnName",length="255",nullable=true)
private Object field;
}
```

3.2.3 小结

本章根据已确定的关系运算子集，加之对 TORM 所面对的应用场景的考量，进行了在 TORM 中对象、关系两侧的属性定义，并对映射规则及操作映射的定义与形式进行了简单约束。

第四章 实现与扩展

4.1 扩展总体结构设计

既然我们的 ORM 是通过分析代码中的对象定义，根据定义生成 MyBatis 中的动态 SQL 映射，在系统运行时就可以通过一般的 MyBatis 数据访问流程将对象实例中域值映射到动态 SQL 中。所以我们需要解决的问题就很清晰了：

- 如何解析数据模型对象的结构
- 如何使用已解析的对象结构生成动态 SQL 映射生成 MyBatis 可使用的映射描述单元

参考这样的问题基础，我们可以给出 TORM 的总体结构，如图：

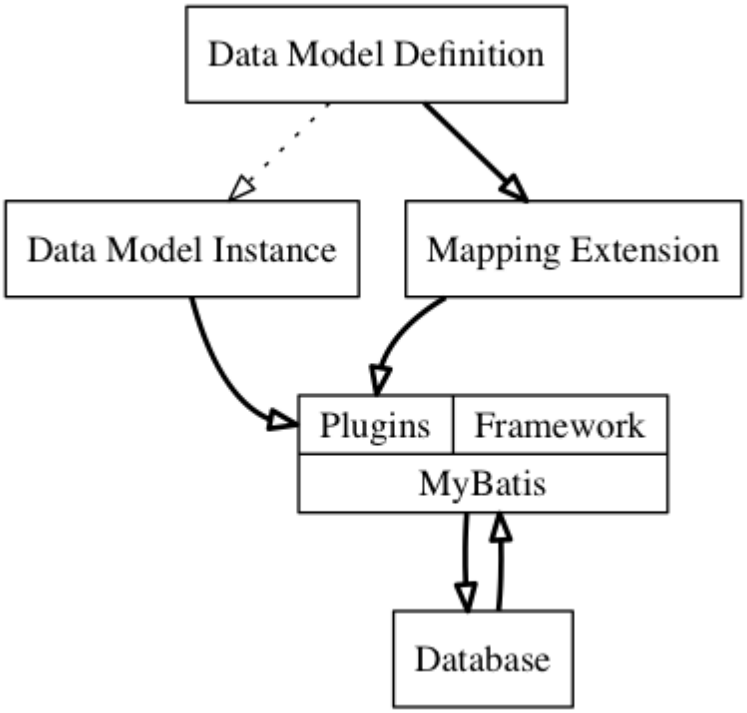

图 4-1

上图中可以看出，数据模型的定义是 Torm 本身与应用系统唯一相关的代码模块，则对应的 Torm 调用形式可以采取一般的 MyBatis 调用的形式，设计结构如下图：

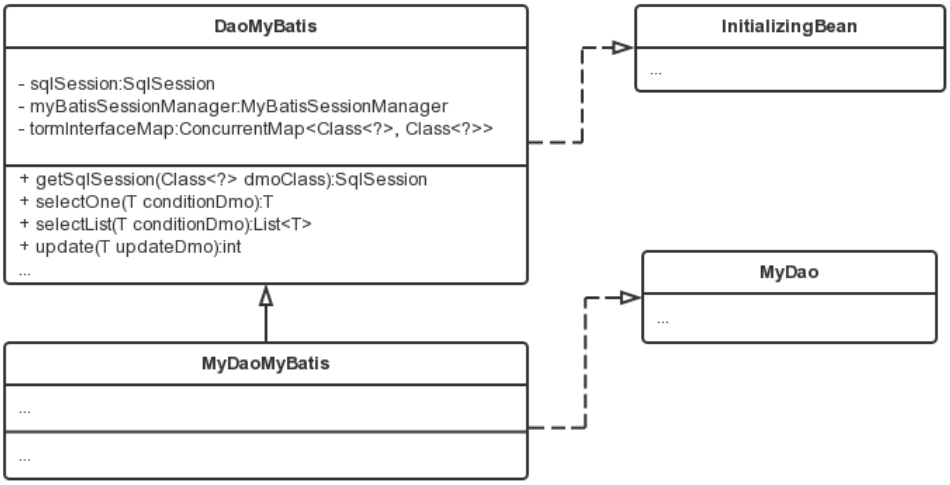

图 4-2

其中，Torm 的入口统一封装在 DaoMyBatis 类中，应用系统通过该类或该类子类传入定义好的 DMO，而 DaoMyBatis 再去调用 MyBatis 的 SqlSession 以执行相关操作。

4.2 数据模型对象相关

在 ORM 的定义中，我们其实已经对 DMO 的定义做出了描述，但作为数据库访问层框架，只有 DMO 的映射定义是不够的，因为对象与数据库中的关系毕竟是异构的，无法直接提供给应用的 SQL 生成来使用。所以，同时我们还要定义在应用中的符合数据库关系结构的对象定义，并提供上述两者对象之间转化的方法。

4.2.1 数据模型对象定义

DMO 的定义在属性映射定义中已经根据其相关注解做出了初步描述，所以这里不再描述注解的具体意义，而是只描述其类结构：

- DMO 定义需满足一般 POJO 定义形式；
- DMO 定义中，参与映射域需根据数据库中对应关系的特征给予相应注解描述，不参与映射域不应与映射域存在数据相关性；
- DMO 定义中，参与映射域需包含完整的 setter/getter 方法，但不限定为直接定义或框架切面添加；
- DMO 的域值中，剔除原有的 NULL 值意义，因为在数据库中引入 NULL 值会涉及到三值逻辑的判断，与对象表现不一致。

4.2.2 关系对象定义

这里所指的关系对象如上文所说，为对 DMO 进行数据访问操作时中间结构。在一次数据库访问操作中，对象实例的域值可以直接通过 MyBatis 的映射规则被 SQL 使用，所以关系对象中只需要通过类解析获取 DMO 对象结构，从而完成动态 SQL 结构的拼接。

同时，关系对象是处于对象至关系的映射过程之中，所以其本身也就代表一个关系，关系中的属性将通过在关系对象中聚合一个内部类，来描述属性（列）的结构，并存储其属性特征。根据现有注解的定义，关系属性的内容结构为：

- name：属性名
- length：字段最大长度
- isNullable：可否为空
- isPk：是否为主键
- strategy：主键策略
- generator：主键策略生成器
- dmoFieldName：对象域名
- dmoFieldClass：对象域类型

4.2.3 对象转化

从 DMO 结构到关系对象的转化比较简单，因为没有参与映射域的注解继承关系，所以只需遍历 DMO 定义中的所有定义域，并解析其注解，然后存储到关系对象中即可。

其转化过程可简单以下图描述：

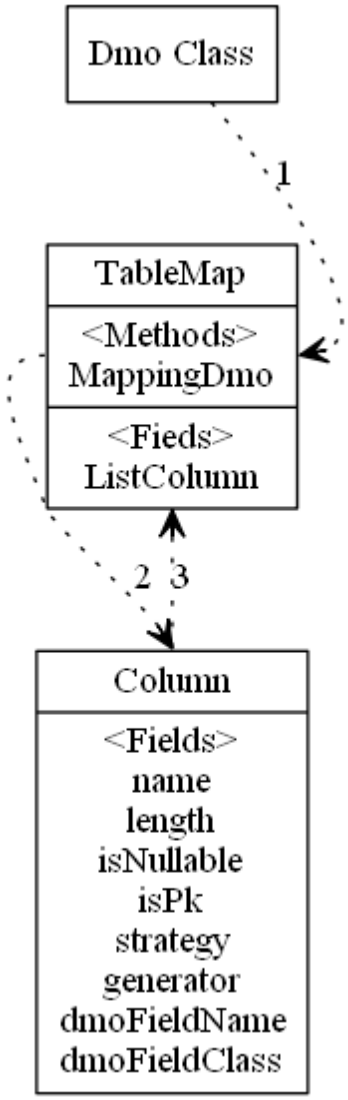

图 4-3

4.3 动态 SQL 定义、生成与加载

4.3.1 动态 SQL 的定义

MyBatis 的动态 SQL 定义分为两种形式，一种较为常见的是使用其特定语法定义的 XML Mapper 文件，另一种则是使用 Java 接口及其方法注解。

接口形式的定义是 MyBatis 的底层实现形式，XML 形式也是基于 Mapper 文件用 AOP 的方式在 JVM 中动态生成相应的类实例和对象实例（虽然和对开发者开放的接口形式有所不同）。对此我们可以模拟 MyBatis 自身的实现，以字节码或源码级别的结构拼接符合我们需求的动态 SQL 定义接口，将其加载到应用所在 JVM。

XML 形式为大多数 MyBatis 使用者采用，其特点是结构清晰、语法集合丰富，几乎能实现所有数据库相关的需求。但 XML 在 MyBatis 调用时，还是通过

MyBatis 使用的 AOP 方法生成相应的接口字节码及其实例，然后添加到 MyBatis 的 Mapper 集合中，也就意味着会带来的额外的成本。但这不是最不利于 TORM 的特征，XML 形式由于使用文件形式存储，如果我们需要动态地去管理这些 Mapper，也就需要相应的系统文件权限，同时 MyBatis 对 XML Mapper 的加载在其自身框架加载之中，我们也就无法根据需要选择我们希望的加载时间点。

所以，接口形式的定义是 TORM 比较理想的实现方式。而对于接口形式定义的实现中我们又有两套方案供选择：

- 纯接口，接口方法加以包含动态 SQL 定义的注解；
- 接口加 SQL 提供类，在接口中定义数据库访问方法，而在 SQL 提供类中实现动态 SQL 拼接。

由于纯接口形式的定义在MyBatis中提供的功能太少，只能完成最为基础的 CRUD 操作，也无法根据实例域值进行逻辑判断，也就无法实现我们根据 DMO 中域属性完成 ORM 定制的需求。因此我们选择接口加 SQL 提供类的形式来定义动态 SQL。

在这种形式下，我们定义数据操作接口及 SQL 提供类形式如下：

```
/** 数据操作接口**/
public interface ISomeOperation<T>{
@SelectProvider("SomeProvider.selectOne")
public T selectOne(T dmo);
}
```

```
/** SQL提供类 **/
public class SomeProvider{
public String selectOne(T dmo){
//基于MyBatis SqlBuilder提供拼接实现
return null;
}
}
```

4.3.2 动态 SQL 的生成

Javassit

MyBatis 对接口形式的具有比较简单清晰的定义形式，对此我们采取 Javassit 这样源码级的 AOP 工具来实现接口相关类实例的生成。Javassist 是一个分析、编辑和创建 Java 字节码的类库。这里我们用它来生成需要的动态 SQL 定义接口及 SQL 提供类。我们选取他的理由就是 Javassist 可以使用源码级的实现来完成 Java 字节码文件的构造，这大大降低了我们的开发难度。虽然 Javassist 在性能上比直接使用字节码拼接略低，但由于 TORM 的动态 SQL 定义加载只需在启动时或定义首次调用时触发，所以对于一个长时间稳定运行的系统而言，这些开销几乎可以忽略不计。

代码模板

使用 Javassit 拼接字节码文件时，过程是类似 Java 手动编写的。所以我们需要定义一组用于拼接的代码模板，包含了我们在 TORM 中会使用到的对象解析、属性判断和 SQL 拼接代码。其中必要模板如下：

- 类级定义模板，用于定义操作接口及 SQL 提供类的类级定义；
- 方法定义模板，用于定义操作接口及 SQL 提供类中的具体方法源码模板；
- 注解定义模板，用于定义操作接口中方法的 MyBatis 映射注解；
- 属性判断模板，用于在 SQL 提供类中添加动态 SQL 的属性判断功能代码；
- SQL 拼接模板，用于 SqlBuilder 中生成、拼接具体 SQL 的源码模板；

特殊规则

这里在使用 Javassist 的生成字节码时，根据 Torm 本身的情况，需要额外定义以下 3 条特殊规则：

- 泛型问题处理：在所有方法入口统一采取泛型参数，添加泛型签名（Generic Signature）；
- JVM 静态堆：加载 Torm 类实例时使用应用系统默认 JVM Class Pool，保证应用其他模块可以正常调用；
- 字节码路径：使用较为规范的命名规则，保证与应用系统中已有路径不冲突；
- SQL 拼接处理：采用 MyBatis 的 SqlBuilder 工具类，便于 MyBatis 自身 SQL 优化。

4.3.3 动态 SQL 的加载

环境

TORM 生成动态的接口类代码实例和 SQL 提供类代码示例时，没有特殊的环境限制，只需满足一条：在 JVM 中的类池里没有同路径下同名类文件即可。

步骤

由于一个应用系统中，DMO 数量可能会较多，如果在启动应用时一次加载会使启动时间存在一个虽然可以接受但不一样的延长。因此我们在 TORM 中使用懒加载机制，步骤如下：

- 在启动应用时使用一个 TORM 的加载池;
- 每次使用 DMO 进行数据访问时从该加载池中捞取对应接口;
- 如果存在则直接使用 MyBatis 中 Dao 层方法;
- 如果不存在则先生成 SQL 提供类，再生成数据访问接口类，再调用 MyBatis 中 Dao 层方法。

性能

每个 DMO 在 Torm 中对应 TormInterface 与 TormProvider 这 2 个动态类实例，随着应用 DMO 数量增长，其永久堆使用也呈线性增长，而 DMO 本身的定义数量也是由应用系统本身而定，且相对于每个 DMO 对应的业务模块而言，线性增长的 2 个动态类实例所占用的永久堆空间几乎是可以忽略不计的。

4.3.4 动态 SQL 的执行

关联到上述的生成与加载步骤，整个调用链的执行过程如下图所示：

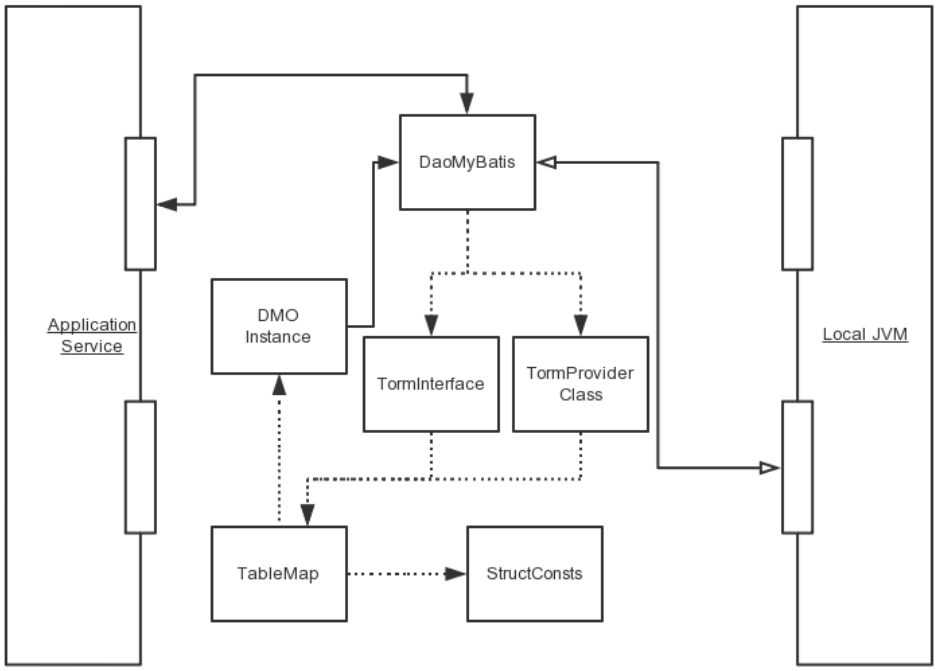

图 4-4

可见，与 JVM 实际执行器交互的只有 DaoMyBatis 的相关接口，所以可以说在 Torm 中，动态 SQL 的执行与普通 MyBatis 过程一样，在定义好的 DaoMyBatis 中直接使用 SqlSession 的相关方法进行数据访问。

4.4 ORM 功能扩展性提供

目前，TORM 已经拥有了实现我们关系运算子集所有运算的基础功能。但是作为一个企业级的应用扩展，需要适应企业不断变更的需求以及一些特殊业务场景的需要是必须做到的。对此，我们将展开在不同层次上对 TORM 的扩展性分析。

4.4.1 对象操作扩展

对象操作扩展是一种比较简单的扩展场景，由于特殊的业务需求，比如非全体的格式转化和缺省数据的初始化等（尤其是一些在应用对象中没有意义却在数据库中有意义的数据值，反之亦然）。对于数据模型对象的相关扩展，建

议使用 AOP 形式，对其构造函数及 setter/getter 进行拦截，并加入对应的业务逻辑，这样侵入性小、可配置性强，而且不需要修改 TORM 原有代码及结构。

4.4.2 TORM 源码扩展

虽然源码扩展是相对麻烦的一种场景，但某些情况下我们不得不采取这种方案，比如我们需要添加新的运算支持，常见的就是关系连接运算。

- 数据访问接口方法扩展
  - 当我们需要为 TORM 添加一个全新的数据库操作时，我们首先需要在数据库访问接口中为其添加新的对应方法。大致步骤如下：
    - 定义新的接口方法模板；
    - 定义新的数据操作实现相关模板；
    - 定义新的 SQL 提供类模板；
    - 应用中添加新的 MyBatis Dao 层调用方法。
- 查询谓词或谓词条件扩展
  - 所谓查询谓词扩展，对于 TORM 而言基本可以等同去添加新的非逻辑谓词，如 GroupBy、OrderBy 等，谓词条件扩展则是对谓词逻辑条件中的逻辑运算符进行扩展。而谓词与 SQL 对应的解释与拼接分处于 TORM 的两个部分：数据模型对象-关系对象映射、SQL 提供类动态 SQL 生成。而在这两步中我们对原始结构(也就是数据模型对象)的解析都是基于其属性域结构和属性域上的注解解析，所以查询谓词或谓词条件扩展也就可以理解为对 DMO 结构的扩展或 DMO 属性域注解扩展。

由于 TORM 的实现归根结底都是由其 Javassist 拼接源码模板，所以除了定义新的扩展注解之后，所有需要做的就只有在 TORM 的源码模板中添加新的解析及拼接代码，然而这是依赖于具体需求的，这里就不做展开了。

4.4.3 MyBatis 插件扩展

在开发不涉及到 ORM 特性的数据库访问层扩展时，如果我们希望既可以使用到 TORM 又不需要改动它，那么这时我们可以直接使用 MyBatis 的插件机制。

- MyBatis 插件对 TORM 执行的兼容性分析
  - TORM 是我们基于 MyBatis 进行的一次非插件级扩展，那么在使用它时，MyBatis 的插件是否能应用到其中呢？答案的分析非常简单：我们使用 TORM 无论其中间过程如何，最终都是做为 MyBatis 的动态 SQL 实现，而 MyBatis 的插件使用代理链机制，其代理链的出入口都处于 MyBatis 的动态 SQL Mapper，所以我们只需要保证使用的插件处于代理链中，那插件本身必然会对 TORM 的所有操作发生作用且不会发生冲突。
- 已有扩展的设计与描述
  - 现在的 TORM 本身还实现了一些常见的插件集合：
    - 数据库方言：基本的数据库方言在 MyBatis 中已经通过对 JDBC 的封装有了基本的实现，但是在面对自身的插件开发时，如果涉及到对特定 SQL 或关系结构的方言特性处理，则需要自行处理。在 TORM

中我们通过定义 Dialect 接口，针对不同的数据库定义公共的接口方法，在以后会使用到这些方言特性的 Plugin 中实现。

- 数据字段加解密：对数据库字段的加解密控制在只考虑数据加解密而非过程加解密的情况下，可以直接通过对 SQL 执行参数的加解密来实现。TORM 中定义了 CryptPlugin，对 MyBatis 代理链上的 Executor 和 ResultHandler 进行拦截，加密分别处理 CRUD 操作，解密处理 HandlerResultSets 中的查询结果。
- 分页查询：分页查询的扩展与数据字段加解密类似，对 Query 操作的 Executor 进行拦截，并在 SQL 执行前添加相应方言下的分页属性内容，查询结果即可实现分页控制。

4.5 小结

TORM 目前实现了在无非逻辑谓词的情况下所有单表操作集合，而这种实现是通过在 MyBatis 上层构筑与 ORM 规则对应的动态 SQL 模板拼接来实现，这样不仅对第二章中对包含逻辑谓词或简单的关联操作做出了扩展性预留，也为其他的可能特殊操作场景提供了与 MyBatis 原有扩展机制一致的扩展可能。在 TORM 中，对映射的具体实现采取 AOP 的形式动态载入应用所在 JVM 中，根据实测，在系统数据操作增加时，JVM 负荷也只是呈现一致的线性增长，系统空间压力完全可以接受。这些 MyBatis Mapper 的载入采用懒加载机制，当首次使用到对应数据操作时再进行加载，对于一般应用场景，该时间开销也完全可以忽略不计。

但由于关系代数运算子集的定义中并没有加入过多对负荷操作的分析，所以在需要定义以某前置操作结果为后置操作参数的聚合操作时，暂时只能通过业务代码进行中间步骤参数的转化，而无法直接利用数据库中的投影特性。这一点在 TORM 设计时事实上是被有意忽略的，因为对于小集合的聚合操作而言，TORM 目前的机制具有更大的灵活性，而对于中间参数集合过大的聚合操作而言，TORM 本身就不是为这种场景而设计的，面对它们，我们需要考虑采用其他的解决方案。

第五章 研究成果

5.1 已实现功能

TORM 目前的已实现的功能可以分为两大类：基础单表数据操作和特殊应用场景的扩展：

- 基础单表功能的实现：TORM 的设计基础就是基于单关系模型下的各类运算，所以对于常见的单表操作，TORM 的使用非常简单且高效，且目前已覆盖所有无非逻辑谓词的单表操作。
- 特殊应用场景的扩展：TORM 在映射层使用了 SQL 模板的设计方案，对于新增功能来说十分方便，只需添加模板即可；对于 SQL 执行和结果返回的扩展和优化则采用 MyBatis 的插件机制，具有很好的兼容性，能在数据访问的各个环节方便地进行切面。已实现的特殊功能有：
  - 数据库方言
  - 查询分页
  - 对象-关系类型匹配
  - 数据加解密
  - 数据访问路由（分库分表）

5.2 重要扩展思考

现有的 TORM 功能，在面对企业中常见的大部分数据库访问场景中都可以适用，但还有两种常见的情况没有囊括：批量 SQL 和简单连接查询。

- **SQL 批量处理**

在企业应用中，对业务数据往往会有大批量的 Non-Query 操作，尤其是 Insert。而 TORM 现在只是采用了 MyBatis 原生的 SQL 执行机制，在处理大量 SQL 时只是交给默认事务进行处理。当操作数量级达到某种程度时，这种操作的时间代价很难承受，而且频繁、庞大的操作提交容易造成操作异常。所以对于同一场景下大量相似的操作请求，TORM 应该提供其批量处理的方案，能在保证事务正确执行的情况下，并发提交 SQL 操作。为此我们可以联想到两种解决方案：代码层面的并发控制和可配性更强的事务控制。

而按照现在的角度观察这两种方案，他们各自有各自的不足。首先，共同的，这两种方案都会带来繁杂的额外配置，比如并发数、事务操作数等等，都需要开发者事先对应用的实际数据量有所估算，再定义具体的值，如果设置不合理，则可能导致整个应用的业务失败。对于代码层面的并发控制，相对简单明了，但面对集群部署的应用时就显得力不从心，如果要实现则需要在 TORM 中添加相关的集群环境支持，这显得有些不现实。对于提高事务 控制的可配性，则存在与业务中原有事务相互冲突的可能，如果开发者对 TORM 不熟悉则可能造成致命的错误，这明显有违 TORM 的初衷。

- **简单连接查询**

在第二章中的推导，其实很大一部分就是为了证明 TORM 这样的实现机制能满足以后对连接查询的扩展，这也是因为简单连接查询确实在企业的应用开发中比较常见。而第一章的结论也表明了，确实在代数层面上，通过对两个单关系的操作可以模拟对一个关联关系的操作。但与理论不同的是，在 TORM 这样的实际应用中，我们不仅仅需要考虑正确性还要考虑其简洁性与效率。

就简洁性而言，由于关联查询需要考虑到两种业务情况：

- 一个关系仅仅是另一个关系的辅助查询条件，在其对象模型中，两个关系代表了不同的对象。这种情况下，联查需要在数据访问方法的参数层面体现。
- 两个关系在对象模型中实际为同一对象的不同属性子集。这种情况下，联查需要在对象模型的定义层面体现。

既然同样地联查场景可能来自于完全不同的对象模型，那我们对这种场景的映射描述也将不是唯一的，这会使得映射的定义本身变得更为复杂。而如果使用强硬的关系类型描述，则削弱了 TORM 在此类场景中对对象关系的抽象能力。

既然同样地联查场景可能来自于完全不同的对象模型，那我们对这种场景的映射描述也将不是唯一的，这会使得映射的定义本身变得更为复杂。而如果使用强硬的关系类型描述，则削弱了 TORM 在此类场景中对对象关系的抽象能力。

效率方面，模拟方式进行的联查更难较好实现。对联查的模拟，本质上是对关系运算的拆分，而原来由关系代数定义的连接操作则有非关系型的对象模型处理来完成相应地筛选和新操作参数的拼接。因此，如果拆分的中间结果集合某一步过大，则会造成原始操作整体的效率极大下降，并对服务器内存造成极大的压力，当数据规模不可接受时，应用会产生系统或业务错误。

5.3 未来的展望

TORM 目前已经在一些成熟的 WEB 应用系统中使用了一段时间，经过这段时间的验证，其效率、安全相关的指标得到了充分的验证，而在首个 TORM 版本发布之后，也经历了几次重要功能的升级，在扩展性方面如同设计时所思考的一样提供了方便、快捷的接口与空间。而作为一个需要适应企业快速生产的 ORM 框架，TORM 离成熟还有很多路要走。对于未来 TORM 的发展而言，以下几点是我们期望 TORM 能完善和实现的：

- 不增加开发复杂度下的联查扩展：联查扩展比较重要，但对于 TORM 而言，同时也不能牺牲其简洁与效率。
- 非逻辑谓词相关功能的完善：现在可以通过在对象模型中的操作实现，但遇上复杂的，尤其是分页相关的操作，则无法实现该需求。
- 动态 Sql 模板的优化或重构：TORM 目前的 SQL 模板还未完全按照 SQL 规范结构组织，可能会影响到其扩展性。
- 批量操作的扩展：批量操作非常重要，但也很难实现，事实上一般企业开发中也是特殊场景特殊实现的，所以可以考虑 TORM 提供批量接口，但不提供批量实现，让开发者根据情况实配。

我们相信以后的版本更替中，这些问题会被逐步解决，而 TORM 也会成为一个成熟的框架，应用到更多的企业级系统中去。

图目录

致谢

首先感谢胡昊老师在本文撰写过程中对我的指导性帮助，其次感谢陈书元高工和袁明高工在 TORM 的研发过程中对框架设计及细节修正方面的巨大帮助。

谢谢。

版权及论文原创性说明

任何收存和保管本论文的单位和个人，未经作者本人授权，不得将本论文转借他人并复印、抄录、拍照或以任何方式传播，否则，引起有碍作者著作权益的问题，将可能承担法律责任。

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含其他个人或集体已经发表或撰写的作品成果。本文所引用的重要文献，均已在文中以明确方式标明。本声明的法律结果由本人承担。

作者签名：

日期： 年 月 日