EF-ORM使用手册 2014版

(原名:25个案例精通Easy Frame-ORM - 2011)

1. EF-ORM概览 8

1.1. 名词解释 8

1.2. 主要特点和特性 9

1.3. 选择EF的理由 12

1.3.1. API方式和xxQL(查询语言)之间的平衡 12

1.3.2. 比某H框架更灵活 13

1.3.3. 比○Batis框架更高效 14

1.3.4. 性能 14

2. 入门 Getting started 15

1.1. 第一个案例 15

1.1.1. Install plug-in for Eclipse 15

1.1.2. 示例工程搭建 16

1.1.3. Hello World！ 18

1.2. 正式开始EF-ORM之旅 20

1.2.1. EF-ORM的原生Entity 20

1.2.2. POJO or Non POJO 23

1.2.3. 实体的增强 24

1.2.3.1. 使用Maven构建时 25

1.2.3.2. 在Eclipse中运行单元测试时 25

1.2.3.3. 在开发调试时 25

1.2.3.4. 手工增强 27

1.3. 创建实体 27

1.3.1. 从数据库创建实体 27

1.3.2. 从PDM文件导入实体 28

1.3.3. 手工编辑和常用JPA Annotation 30

1.3.3.1. 注解的使用 30

1.3.3.2. 映射关系和注解的省略 32

3. 基本操作与对象映射 32

3.1. 基本单表操作 32

3.1.1. 复合主键 33

3.1.2. 主键自增 35

3.1.2.1. 自增值自动回写 35

3.1.2.2. 四种JPA主键生成方式 36

3.1.2.3. 配置的简化 37

3.1.2.4. Sequence或Table下的其他配置 37

3.1.2.5. Sequence或Table的性能优化 38

3.1.2.6. 配置方法和总结 40

3.1.3. 查询操作 41

3.1.3.1. 按主键或模板查询 41

3.1.3.2. 更复杂的条件 42

3.2. 你不是在操作‘对象池’ 43

3.2.1. 查询条件与字段排序 46

3.2.2. 更新主键列 47

3.2.3. 特殊条件: AllRecordsCondition 47

3.2.4. QueryBuilder的使用 48

3.2.5. Criteria API和查询语言 50

4. Criteria API单表操作 51

4.1. 查询 52

4.1.1. 查询条件的用法 52

4.1.1.1. And Or Not 52

4.1.1.2. 使用函数或表达式作为条件 53

4.1.1.3. 使用JPQLExpression 54

4.1.1.4. 使用SqlExpression 55

4.1.2. 使用Selects对象 55

4.1.2.1. 定义查询的列 55

4.1.2.2. 使用Distinct 56

4.1.2.3. 使用Group和max/min 56

4.1.2.4. 使用Having 57

4.1.2.5. count的用法 58

4.1.2.6. 使用数据库函数 59

4.1.2.7. 在查询项中使用SQL表达式 60

4.1.3. 分页 61

4.1.3.1. 限定结果范围 61

4.1.3.2. 使用PagingIterator 62

4.1.4. 小结 64

4.2. 更新 64

4.2.1. 基本操作 64

4.2.2. 更新操作Query的构成 65

4.2.3. 更多的更新用法 66

4.2.3.1. 并发环境下原子操作的更新 66

4.2.3.2. 使用prepareUpdate方法 68

4.2.4. UpdateValueMap的一些特性 69

4.2.4.1. 回写 69

4.2.4.2. 自动清空 69

4.2.4.3. stopUpdate和startUpdate 69

4.2.4.4. 通过对比形成updateValueMap 69

4.3. 删除记录 71

4.3.1. 概述 71

4.3.2. 用法示例 71

4.3.2.1. 基本操作 71

4.3.2.2. 使用Query对象 72

5. 级联操作 72

5.1. 基本操作 73

5.1.1. 使用注解描述级联关系 73

5.1.2. 单表操作方式的保留 74

5.1.3. 级联操作的效果 76

5.2. 使用注解定义级联行为 78

5.2.1. 仅引用级联对象的单个字段 78

5.2.2. @JoinDescription 80

5.2.2.1. 定义与作用 80

5.2.2.2. 控制关联对象列表的排序和数量 81

5.2.2.3. 在FilterCondition中使用变量 82

5.2.3. 其他JPA注解的支持 82

5.2.3.1. 延迟加载 82

5.2.3.2. 外连接获取 83

5.2.3.3. 级联方向 84

5.2.4. 定制级联行为 85

5.2.4.1. 四种级联关系的行为 85

5.2.4.2. 使用注解限制级联关系的使用场合 89

5.2.5. 级联条件CascadeCondition 89

6. Criteria API多表操作 90

6.1. Join查询 92

6.1.1. 基本用法 92

6.1.2. 使用Selects对象 93

6.1.3. 分页和总数 94

6.1.4. 条件容器的问题 95

6.2. Union查询 97

6.3. Exists条件和NotExists条件 98

7. 本地化查询 99

7.1. 本地化查询是什么 99

7.2. 使用本地化查询 101

7.2.1. NamedQuery和NativeQuery 101

7.2.2. API和用法 102

7.2.3. 命名查询的配置 105

7.2.3.1. 配置在named-queries.xml中 105

7.2.3.2. 配置在数据库中 105

7.2.3.3. 数据源绑定 106

7.2.3.4. 动态更新命名查询配置 107

7.3. NativeQuery特性使用 108

7.3.1. Schema重定向 108

7.3.2. 数据库方言——语法格式整理 109

7.3.3. 数据库方言——函数转换 111

7.3.3.1. 示例 111

7.3.3.2. 函数支持 112

7.3.3.3. 方言扩展 114

7.3.4. 绑定变量占位符 116

7.3.5. 动态SQL——表达式省略 117

7.3.6. 动态SQL片段 119

7.3.7. 分页查询 120

7.3.8. 在命名查询中编写多数据库SQL 120

7.3.9. 对Oracle Hint的支持 121

7.3.10. 对Start with ... Connect by的有限支持 122

7.4. 存储过程调用 123

7.4.1. 使用存储过程 123

7.4.2. 存储过程传出参数 124

7.4.3. 存储过程传出游标 125

7.4.4. 使用匿名过程（匿名块） 125

7.5. 原生SQL使用 126

7.5.1. 使用原生SQL查询 126

7.5.2. SqlTemplate 128

7.6. 无表查询 129

8. 高级查询特性 130

8.1. 查询结果的转换 130

8.1.1. 返回简单对象 130

8.1.2. 返回和查询表匹配的对象 131

8.1.3. 返回任意容器对象 131

8.1.4. 返回Var /Map 133

8.1.5. 多个对象以数组形式返回 134

8.1.6. 多个列以数组形式返回 135

8.2. Transformer的使用 136

8.2.1. 直接指定返回类型 136

8.2.2. 区分两种返回数据的规则 137

8.2.3. 忽略@Column注解 137

8.2.4. 自定义返回结果转换 138

8.3. 流式操作接口 142

9. 事务控制与批操作 144

9.1. 编程式事务控制 144

9.2. 批操作 145

9.2.1. 基本用法 145

9.2.2. 不仅仅是操作实体 147

9.2.3. 性能的极限——Extreme模式 149

10. 分库分表与数据路由 150

10.1. 数据分片概述 150

10.1.1. 什么是数据分片 150

10.1.2. EF-ORM支持的分片功能 151

10.1.3. 和具有同类功能的产品比较 152

Hibernate Shards / HiveDB 152

Alibaba TDDL 154

10.2. 分库分表规则配置 155

10.2.1. 水平拆分——分表 155

10.2.2. 水平拆分——分库 158

10.2.3. 垂直拆分 159

10.3. 分库分表后的操作和行为 159

10.3.1. 前提与原则 159

10.3.2. 按需建表 160

10.3.3. 自增主键生成行为 160

10.3.4. 手动建表 161

10.3.5. 过滤不存在的表 162

10.3.6. 如果没有命中表 163

10.3.7. 垂直拆分的场景 163

10.3.8. 在分库分表条件基本确定的场景下 164

10.3.9. Batch模式和分库分表 166

10.3.10. 当路由结果为多库多表时 168

10.3.11. 使用SQL语句插入、更新和删除时的路由 169

10.3.12. 使用SQL语句进行查询时的路由 171

10.4. 分库规则加载器 173

11. 与Spring集成 173

11.1. 配置和使用 174

11.1.1. EntityManagerFactory的配置 174

11.1.2. 事务配置 175

11.1.3. 编写DAO 177

继承GenericDaoSupport 177

继承BaseDao 178

11.1.4. 常用API：CommonDao 179

11.1.5. POJO操作支持 182

11.2. 分布式事务控制 184

11.2.1. JPA事务（多重） 185

11.2.2. JTA事务支持 185

12. DB元数据与DDL操作 185

12.1. 数据库基本信息 185

12.1.1. JDBC版本 185

12.2. 获得数据库对象 185

12.2.1. 表和表结构 186

12.2.2. 约束与外键 186

12.2.3. 索引 186

12.2.4. 函数 186

12.3. DDL操作 186

12.3.1. Create table 186

12.3.2. Drop table 186

12.3.3. Create index 186

12.3.4. Create sequence 186

12.4. 其他功能 186

12.4.1. 执行sql的脚本文件 186

13. 动态表支持 186

13.1. 动态表支持 186

13.1.1. 动态表的返回（特殊） 186

13.2. 扫描现存的数据库表作为动态表操作 187

14. 其他功能与特性 187

14.1. 记录时间戳自动维护 187

14.2. 内置连接池 188

15. JMX监控 188

15.1. DbClientInfo 189

15.2. ORMConfig 190

16. 保持数据库移植性的实践 191

16.1. 数据库结构维护 191

16.2. 数据库关键字处理 191

16.3. 自增主键返回 191

16.4. 分页处理 191

16.5. 在NativeQuery中编写“通用SQL” 191

数据库函数 191

表达式 191

16.6. 方言扩展 191

16.7. 其他 191

17. 性能调优指南 191

17.1. 性能日志 191

17.2. 级联性能 192

17.3. 一级缓存与二级缓存 193

17.4. 结果集加载调优 193

17.4.1. Fetch-size 193

17.4.2. max-results 194

17.4.3. 使用CachedRowSet 194

17.5. 查询超时控制 195

17.6. 自增值获取性能问题 195

18. 常见问题 (FAQ) 195

18.1.1. JDBC驱动问题 195

18.1.2. JDK7编译后的ASM兼容 195

18.1.3. 数据库存储的口令加密 196

18.1.4. Oracle RAC环境下的数据库连接 196

18.1.5. 某些正确的SQL语句解析错误怎么办 196

19. 附录一 配置参数一览 196

20. 附录二 数据库兼容性说明 200

# EF-ORM概览

## 名词解释

本文中出现的一些术语的解释。其中，知识领域一列指出了对该词条进行定义的知识领域。凡是知识领域为EF-ORM的词条，都是在本文中有特殊定义的词汇。

|  |  |  |  |
| --- | --- | --- | --- |
| 中文名 | 英文名 | 知识领域 | 解释 |
| 关系数据库管理系统 | RDBMS | 数据库常识 | 即各种数据库软件本身，如Oracle, DB2,Postgresql等。 |
| -- | DML | 数据库常识 | Data manipulation language：指SELECT、UPDATE、INSERT、DELETE四种操作数据的语句。 数据库的事务控制只针对DML。 |
| -- | DDL | 数据库常识 | Data definition language：DDL。命令有CREATE、ALTER、DROP等，DDL主要是用在定义或改变表（TABLE）的结构，数据类型等初始化工作上。除了四句DML外其他所有语句一般都认为是DDL，如TRUNCATE也是。 |
| -- | JPA | Java常识 | Java EE规范之一，由JCP组织定义的Java持久层(适用于关系数据库)操作规范。JPA有1.0和2.0两个版本，2.0是1.0的补充。 |
| -- | JTA | Java常识 | Java事务API，由JCP组织定义的Java EE标准之一。使用JTA可以实现分布式事务的一致性。 |
| 注解 | Annotation | Java常识 | JavaSE特性之一，编程时在代码上增加“@xxx”这样的标注来描述代码的行为。 |
| 简单对象 | POJO | Java常识 | 只有字段和get/set方法，不继承任何类或接口的java bean。 |
| 实体 | Entity | Jave EE  JPA | 一个实体就是用@Entity注解标注的一个类。这个类是一张数据库表的java映射。 |
| 元模型 | Meta-model | Jave EE  JPA | 元数据模型，描述数据库表以及其对应的Java类的结构、映射关系等信息的对象模型。在Java中有对应的对象和类。 |
| 增强 | Enhancement | Jave EE  JPA | 使用字节码技术对java类文件进行代码植入，从而实现Enity类一些高级行为的技术手段。 |
| 查询 | Query | EF-ORM | 一项对数据库操作请求。请不要从字面含义理解为是select语句，查询也可以是一项数据update操作。  在EF-ORM中Criteria 和 本地化查询都是查询的主要形式，最终都对应某项数据库操作。 |
| -- | Criteria API | EF-ORM | 原指JPA 2.0规范制定的一套用对象来实现复杂数据库查询的API。本文中指EF-ORM自身实现的类似功能的API。使用该API可以构造出查询对象（Criteria Query）来执行数据库查询。 |
| 本地化查询 | -- | EF-ORM | 使用SQL（或JPQL）进行的数据库操作。本地化查询包含NamedQuery、NativeQuery、NativeCall三种。 |
| -- | NativeQuery | EF-ORM | 基于增强型SQL或JPQL语句的查询。 |
| 命名查询 | NamedQuery | EF-ORM | 将NativeQuery的SQL语句和变参等信息预先配置并取名，在使用时按名称来调用查询。 |
| 存储过程调用 | NativeCall | EF-ORM | 一个调用存储过程的查询。 |
| 数据库方言 | Dialect | EF-ORM | 由于不同RDBMS语法和功能的差异，EF-ORM将这些行为差异集中到一个接口上，并为每个RDBMS单独提供一个实现类，用这种方法来支持不同的RDBMS。每个实现类就是一种方言。 |
|  | Session | EF-ORM | 和某H框架相似，主要的数据库操作方法都集中在这个对象上。Session表示一个在事务状态下,或者非事务状态下的数据库操作句柄。  本文中 最常见的DbClient对象，是Session的子类。Session的另一个子类是Transaction类。 |

## 主要特点和特性

EF-ORM是一个轻量，便捷的Java ORM框架。在经历了金融和电信等项目的锤炼后，具备了众多的企业级特性。

**主要特点**

|  |  |
| --- | --- |
| 特点 | 描述 |
| 轻量 | 该框架对应用环境、连接池、 是否为J2EE应用等没有特殊要求。可以和EJB集成，也可与Spring集成，也可以单独使用。整个框架模块和功能都较为轻量。 |
| 依赖少 | 整个框架只有两个jar包。间接依赖仅有commons-lang, slf4j等7个通用库，作为一个ORM框架，对第三方依赖极小。 |
| 简单直接的API | 框架的API设计直接面向数据库操作，不绕弯子，开发者只需要数据库基本知识，不必学习大量新的操作概念即可使用API完成各种DDL/DML操作。 |
| 最大限度利用编译器减少编码错误的可能性 | API设计和元数据模型（meta-model）的使用，使得常规的数据库查询都可以直接通过Criteria API来完成，无需使用任何JPQL/HQL/SQL。可以让避免用户犯一些语法、拼写等错误。 |
| JPA2规范兼容 | 使用JPA 2.0规范的标准注解方式来定义和操作对象。（但整个ORM不是完整的JPA兼容实现） |
| 更高的性能 | 依赖于ASM等静态字节码技术而不是CGlib，使得改善了代理性能；依赖于动态反射框架，内部数据处理上的开销几乎可以忽略。操作性能接近JDBC水平。对比某H开头的框架，在写入操作上大约领先30%，在大量数据读取上领先50%以上。 |
| 更多的性能调优手段 | debug模式下提供了大量性能日志，帮您分析性能瓶颈所在。同时每个查询都可以针对batch、fetchSize、maxResult、缓存、级联操作类型等进行调整和开关，可以将性能调到最优。 |
| 可在主流数据库之间任意切换 | 支持Oracle、MySQL、Postgres、GBase、SQLite、HSQL、Derby等数据库。 除了API方式下的操作能兼容各个数据库之外，就连SQL的本地化查询也能使之兼容。 |
| JMX动态调节 | 可以用JMX查看框架运行统计。框架的debug开关和其他参数都可以使用JMX动态调整。 |
| 企业级特性支持 | SQL分析，性能统计，分库分表，Oracle RAC支持，读写分离支持，动态字段支持 |

一谈ORM很多朋友都会自然性的和某H框架进行比较，从而发现这些ORM很多都只是功能简单的“玩具”。但是EF不是仅仅是一个玩具。某H框架支持的功能EF-ORM几乎都能支持，或者有替代的支持手段。

**和某H框架支持特性对比**

|  |  |  |
| --- | --- | --- |
|  | 某H框架 | EF-ORM |
| 兼容主流数据库 | 几乎所有数据库。一些过于简单的数据库不支持，如SQLite。 | 支持所有主流数据库，包括SQLite |
| 数据库移植性 | 通过配置数据库方言和调整一些映射配置，可以实现移植。 HQL 中的部分语法和函数支持跨库操作。SQL不支持跨库 | 自动检测数据库，自动兼容本地数据库方言，无需人工干预。 SQL/JPQL都支持自动跨库改写。 后续还将支持单Session同时支持多个异构数据库并发操作。 |
| 级联 | 支持JPA等规范要求的One-toOne, One-to-many, many-to-on, many-to-many下的CRUD级联操作。 | 除了支持JPA等规范要求四种级联关系外，还保留单表操作的方式，可以使用API在级联操作和单表操作中灵活变化。 |
| 延迟加载 | 支持级联场景下的各种延迟加载。 | 除了支持级联场景下的延迟加载外，还支持LOB字段延迟加载。 |
| 投影操作 | 支持 | 支持 |
| Criteria API | 支持JPA的Criteria API和框架自己的Criteria API | 支持框架自己的Criteria API。 |
| 自生成主键策略 | 支持Sequence,Native，Identity，GUID,Hilo,table, Froigen等。一旦配置为非Native方式，会降低可移植性 | 支持Identity、Sequence、Table、Auto(Native)、GUID等策略，Hilo作为辅助算法可以和Sequence或Table随意组合。  Sequence名称可定制，支持自动创建Sequence，支持Sequence步长检测。可根据DBMS自动切换自增方案。 |
| 通过Transformer定义查询结果转换 | 支持Map，支持用Object[]返回多个对象，支持用ResultTransformer类自定义转换逻辑。 | 支持Map，支持用Object[]返回多个对象，支持用Transformer自定义转换逻辑。 |
| 一级缓存 | 支持。作为Session的核心概念和实现手段，无法关闭。当影响结果正确性时，需通过flush 等API来手动操作。  HQL或SQL等操作不会被缓存，也不会刷新一级缓存中的的数据。 | 支持。可关闭。SQL语句和API操作都会刷新缓存中的数据。 |
| 动态表支持 | 在进行一些配置和定制后可以支持动态表的单表操作。 | 原生支持，所有动态表和静态表都会被提取为元模型，后台处理完全一致。因静态表支持的功能均能在动态表上使用，甚至包括动态表和静态表之间的级联操作。  还支持直接扫描数据库结构动态生成元模型。 |
| Oracle ROWID支持 | 支持 | 目前支持得还不太好，待改进。 |
| JTA事务 | 支持 | 支持 |
| Spring集成 | Spring原生支持。方便。支持Spring全部的7种事务传播行为、隔离级别。 | 按JPA规范和Spring集成。方便。提供JPADialect来支持7种传播行为、事务隔离级别、事务超时（标准JPA不支持） |
| 约束，所有表必须有主键 | ORM的基本约束之一 | 无需此约束 |
| 支持数据库逆向工程 | 通过第三方插件可从数据库生成对象映射 | EF Eclipse插件可从数据库生成对象映射。还可以从PDM文件中生成对象映射。 |
| 自增值也可以手工指定 | 不支持 | 支持 |
| 存储过程 | 不支持 | 支持存储过程，支持存储过程返回值的转换和封装 含匿名存储过程，即动态执行一段存储过程。 |
| 命名查询 | 配置在注解上 | 集中配置。可配在XML文件中和数据库中 |
| 动态SQL | 不支持 | 可根据条件自动省略SQL语句中的条件、函数等SQL片段。 SQL中可指定部分片段为变量，(如表名、条件等)，在运行时动态形成SQL。 |
| DDL | 支持hbm2ddl的配置方式刷新数据库结构。 | 使用实体扫描器可以验证和刷新数据库结构。同时所有的DDL操作也以API的形式直接提供，用户可随意操作。 |
| 分库分表 | 不支持 | 支持 |
| 针对Oracle RAC的优化 | 不支持 | 可根据分库路由结果或读写分离策略，单独操作RAC的节点。针对RAC的读写争用问题进行优化。 |
| Oracle Hint支持 | 不支持 | 原生支持Oracle的 /\*+ APPEND \*/等性能优化的手段。 |
| SQL分页改写 | 不支持 | 通过内置分析器，直接改写原SQL查询形成count语句。原生支持分页操作。 |
| 绑定变量 | HQL中支持。API操作均使用绑定变量。 | SQL/JPQL中支持。API操作均使用绑定变量。(部分方言中禁止绑定变量的情况除外) |
| 数据库关键字判断 | 部分数据库支持 | 建立了完整的常用数据库关键字列表，对用到的关键字自动添加引号或`符号，确保带有数据库关键字的操作也都能正常执行。 |

上面是EF-ORM和H框架的一个简单对比。总结下来，EF-ORM的支持功能基本覆盖了大家耳熟目详的H框架，甚至在很多项上要更进一步。但EF-ORM和传统的如H框架,T框架，OpenJPA框架的设计思想是不同的，这些思想上的差异体现在——

* 更多考虑逆向工程数据库的各种场景。
* 要求操作者面向数据库来思维，面向对象的操作手段更多只是带来开发效率的提升和开发难度的降低。
* 追求更高的性能。
* 同时保留多表级联的关系型操作和单表的操作。能保留在关系型数据库的ER模型的基础上完整的操作数据库，而不是要求用户必须使用对象模型来操作数据库。
* 兼容和保留以SQL为主的传统数据库操作方式，并使之能兼容多种不同的DBMS。在金融和电信领域，还有大量传统的以SQL（甚至是存储过程）为主的数据库使用方式。我们不可能让这些领域的应用完全改弦易帜到纯粹的对象型数据操作上。
* 不但封装了DML操作，也封装了建表、建索引、ALTER TABLE、TRUNCATE等DDL操作。

除此之外EF-ORM还有很多同类框架中很少具备的企业级应用特性

* 分库分表
* 分布式事务 JTA
* 多数据源与路由
* 多数据源支持
* Oracle RAC FCF支持，Oracle RAC读写分离

综上所述，EF-ORM可以是一个小玩具，但也是支撑大型项目中的企业级框架。

## 选择EF的理由

### API方式和xxQL(查询语言)之间的平衡

EF-ORM是以轻量、易用为目的设计的开源关系型数据库ORM框架。其特点之一是：

* 尽可能通过编译器检查数据库操作的正确性

EF-ORM通过原生的元模型和内建的Criteria API来完成类型**安全**的动态查询。 这种查询优于传统的基于字符串的 Java Persistence Query Language (JPQL/HQL) 查询。本文假设您具备基础的 Java 语言编程知识，并了解常见的 JPA 使用，比如 EntityManagerFactory 或 EntityManager。首先我们来看JPQL / HQL 查询有什么缺陷？

某H框架引入了HQL；JPA 1.0也引进了 JPQL，它们都是强大的查询语言，它在很大程度上导致了 各自框架的流行。不过，基于字符串并使用有限语法的 JPQL 存在一些限制。要理解 JPQL 的主要限制之一，请查看清单 1 中的简单代码片段，它通过执行 JPQL 查询选择年龄大于 20 岁的 Person 列表：

清单 1. 一个简单（并且错误）的 JPQL 查询

|  |
| --- |
| EntityManager em = ...;  String jpql = "select p from Person where p.age > 20";  Query query = em.createQuery(jpql);  List result = query.getResultList(); |

但是这个简单的例子有一个验证的错误。该代码能够顺利通过编译，但将在运行时失败，因为该 JPQL 查询字符串的语法有误。清单 1 的第 2 行的正确语法为：

|  |
| --- |
| String jpql = "select p from Person p where p.age > 20"; |

不幸的是，Java 编译器不能发现此类错误。在运行时，该错误才会出现并提示。

这个例子几乎描述出了一切xxQL（包括SQL）带来的困境。因为我们总是无法轻易发现查询语句中的错误。无论是在语句初次编写后，还是在数据库结构变化后的维护过程中。

而使用Criteria API来执行动态查询，将会安全得多，几乎任何错误都将会被编译器发现并无情的指出，码农可以放心的重构他们的代码乃至数据库，因为编译器会将一切笔误和疏忽指出。不要小看这样一个小小的进步，因为当您考量一个软件的成本会发现，绝大部分的软件成本都是在后期的维护和测试当中。

* 内建的元模型

JPA引入了meta-model这个概念。OpenJPA可以使用编译的处理器生成名为元模型的类。简单说来，元模型就是将数据表结构用Java类、字段、枚举等形式表现出来的静态结构。我们在代码使用元模型的常量来表示表和列，也就相当于将我们的查询让编译器来校验一遍。

OpenJPA的元模型是依赖于编译时生成的一个奇特的类，比如Entity类A.java，那么生成一个名为A\_.java的原模型类。EF-ORM的元模型则是利用了java的内部类机制。它需要在您的Entity中生成一个 枚举类型

|  |
| --- |
| public enum Field implements jef.database.Field {  field1, field2, ….. } |

这个枚举就成为了该Entity的元模型描述。可以在构建查询的时候使用。

* Say ‘No’ to XML

EF-ORM希望尽可能少的配置，以及由此带来的开发效率的提升。因此EF-ORM使用标准JPA注解方式来描述一个Entity。整个过程实现了零配置（如果您不把注解当做配置的话）。

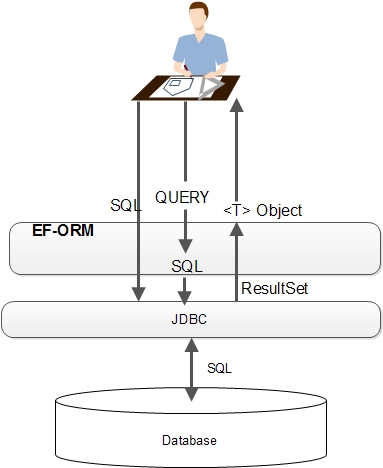
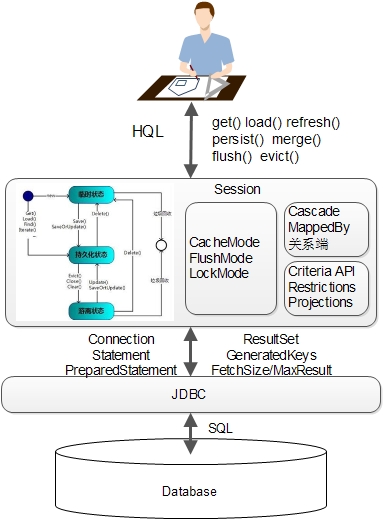
### 比某H框架更灵活

围绕H框架一直以来都有用性能和效率和灵活性的争论。人们在徘徊于这些问题的同时，往往忽视了H框架这些年来的巨大改进和性能优化。H框架其实提供了让人眼花缭乱的特性，以及优秀的性能。大多数围绕H框架的性能诟病，其实多是使用不当造成的。个人感觉，很多人在H框架的使用中存在以下的问题：

1 对于一级缓存等特性认识并不深刻，甚至有人为了避免麻烦，动辄flush，evict，性能不佳。

2 对于HQL过度依赖，最终走回到拼凑HQL语句的路子上去，就像当初拼凑SQL一样。这十多年前那些拼凑SQL的代码有什么不同？ 除了开发者需要学习一种形似SQL的新查询语言以外，代码的可维护性和开发效率的提升又在哪里？

EF-ORM的出发点之一是更简单和直白的使用数据库。并不希望使用者认为自己是在操作一个“对象库”。而是将JDBC用更简练，更方便的手段来封装给用户使用。



EF-ORM干的事情很简单，将SQL语句，或者和SQL语句直观对应的Query对象传给数据库执行操作；同时将查询结果的JDBC封装对象转换成用户需要的各种Object，至于不顾关系型数据库的特色把它封装成一个按主键操作的对象库更是想都没想过。EF-ORM提供的大部分API都可以直观的映射到传统JDBC的操作方式上，并没有虚拟出一个“Session”的对象库。也不存在什么“游离对象”“临时对象””持久对象”之类的概念；无需人工flush、evict。将数据库的一切特性和用法明明白白的展现在开发者面前。对于熟悉SQL的开发者和DBA来说，既保留了某H框架操作的高效，又没有“隔靴搔痒”的困惑。

正是这个特点，使得EF-ORM对于熟悉SQL语句的开发者来说，学习成本非常低。

EF-ORM原汁原味的保留了JDBC的高效特性，这会不会造成开发效率上的降低呢？众所周知，JDBC的开发效率和维护效率要低于各种ORM框架。事实上EF-ORM也汲取了JPA等持久化框架的优点，“拟态”出了某H框架能支持的几乎全部行为，提供了高效而实用的API。在表面上，EF-ORM像是一个完全的面向对象进行操作的ORM框架，实际上这些都是假象。EF-ORM本质上不支持任何对象操作，操作数据库只有两种方式——Query API、SQL。前者可以利用编译器发现大多数操作错误，减少开发者犯错误的可能。后者可以灵活的操作数据库，但保留了对象映射和多数据库兼容等特色。

### 比○Batis框架更高效

很多朋友在发现某H框架难以适应自己的项目后，最终都选择了○Batis框架作为项目的首选。○Batis也是一个非常优秀的框架。

使用动态SQL语句是这个框架的大亮点，这给了开人人员灵活而强有力的武器来操作数据库。但是这也使在这个框架上开发的应用存在数据库移植性上的问题。此外，对象和表的映射关系维护过于繁琐等问题，也困扰着○Batis的使用者。○Batis使用者还需要记忆特定的一系列XML标签来帮助他们实现动态SQL的特性。而且这些也需要纯手工完成。

综上所述○Batis的开发效率，和H框架还是有明显的差距的，相对于某H框架这个全自动框架，大多数人认为○Batis更像是一个“半自动”的框架。

但是我认为最大的问题不在于开发效率，而在维护效率上。其过于原生的数据库操作方式，难以避免项目维护过程中的巨大成本。当数据库字段变化带来的修改工作虽然可以集中到少数几个XML文件中，但是依然会分散在文件的各处，并且你无法依靠Java编译器帮助你发现这些修改是否有错漏。在一个复杂的使用○Batis的项目中，变更数据库结构往往带来大量的Code Review和测试工作，否则难以保证项目的稳定性。

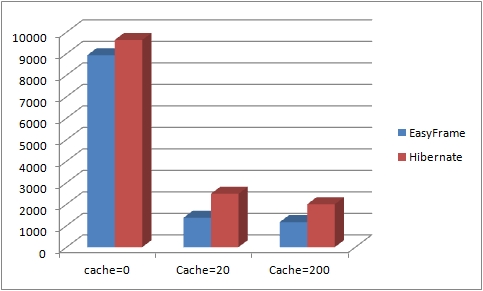
EF-ORM也有动态SQL的特性，但在一般性的项目中，使用Criteria API和SQL的比例大约是8:2。能够不写SQL就能解决的问题，为什么还要写SQL语句呢。

### 性能

得益于EF内部的动态反射框架和字节码技术，EF-ORM拥有优秀的性能，几乎接近于直接操作JDBC API。

在操作大量数据读写的的场合下，这一优势发挥得更为明显。

用EF-ORM和某H框架进行的读写性能测试表明



EF的性能优势来源于

* 1. 使用ASM而不是反射来构造对象
  2. 类的增强在编译期就已经完成。
  3. 大量后期的优化和策略模式的应用。

# 入门 Getting started

我们在这一章，将帮助您搭建一个简单的测试工程，完成您EF-ORM使用的第一步。

## 第一个案例

### Install plug-in for Eclipse

请先安装EF-ORM for eclipse插件。

1、在Eclipse的Update Site中增加站点<http://geequery.github.io/plugins/1.3.x/>，

2、选择help \ Install new Software来安装JEF插件。

### 示例工程搭建

**直接下载**

您可以直接下载本文的示例工程，也可以按后面的说明一步一步搭建此工程。

|  |
| --- |
| <https://github.com/GeeQuery/ef-orm/tree/master/orm-tutorial> |

该示例工程中，包含了本文中的所有代码清单

**自行创建**

新建测试工程，然后将EF-ORM的包加入到工程。

* 本文中的测试都采用Apache derby来进行，为了运行本文的案例，你需要下载一个derby.jar放到你的工程目录下。
* 未使用Maven场合下

可以从Git直接构建

|  |
| --- |
| git clone <https://github.com/geequery/ef-orm.git> cd ef-orm mvn install –Dmaven.test.skip=true |

### Hello World！

代码示例

src/main/java/org/easyframe/tutorial/lesson1/entity/Foo.java

|  |
| --- |
| **package** org.easyframe.tutorial.lesson1.entity;  **import** java.util.Date;  **import** javax.persistence.Id;  **public** **class** Foo {  @Id  **private** **int** id;  **private** String name;  **private** Date created;  **public** **int** getId() {  **return** id;  }  **public** **void** setId(**int** id) {  **this**.id = id;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** Date getCreated() {  **return** created;  }  **public** **void** setCreated(Date created) {  **this**.created = created;  }  } |

src/main/resources/jef.properties

|  |
| --- |
| db.type=derby  db.filepath=./  db.name=db  #调试开启，所有SQL和参数都会打印到控制台上  db.debug=true |

src/main/java/org/easyframe/tutorial/lesson1/Case1.java

|  |
| --- |
| **package** org.easyframe.tutorial.lesson1;  **import** java.sql.SQLException;  **import** java.util.Date;  **import** java.util.List;  **import** jef.database.DbClient;  **import** jef.database.jpa.JefEntityManagerFactory;  **import** jef.tools.reflect.BeanUtils;  **import** org.easyframe.enterprise.spring.CommonDao;  **import** org.easyframe.enterprise.spring.CommonDaoImpl;  **import** org.easyframe.tutorial.lesson1.entity.Foo;  **import** org.junit.Assert;  **import** org.junit.Test;  **public** **class** Case1 {  @Test  **public** **void** simpleTest() **throws** SQLException{  DbClient db=**new** DbClient();  JefEntityManagerFactory emf=**new** JefEntityManagerFactory(db);  CommonDao dao=**new** CommonDaoImpl();  //模拟Spring自动注入  BeanUtils.*setFieldValue*(dao, "entityManagerFactory", emf);    //1.创建表  dao.getNoTransactionSession().createTable(Foo.**class**);    Foo foo=**new** Foo();  foo.setId(1);  foo.setName("Hello,World!");  foo.setCreated(**new** Date());  dao.insert(foo); //2.插入一条记录    //3.从数据库查询这条记录  Foo loaded=dao.loadByKey(Foo.**class**, "id", foo.getId());  System.*out*.println(loaded.getName());    //4. 更新这条记录  loaded.setName("EF-ORM is very simple.");  dao.update(loaded);    //5.删除这条记录  dao.removeByKey(Foo.**class**, "id", foo.getId());  //6.查所有记录，查不到  List<Foo> allrecords=dao.find(**new** Foo());  Assert.*assertTrue*(allrecords.isEmpty());    //7. 删除表  dao.getNoTransactionSession().dropTable(Foo.**class**);  } } |

上面这个案例，运行以后，注意观察控制台，将会打印出所有执行的SQL语句和耗时。您将会注意到，EF-ORM在这个案例中，顺序进行了——

建表、插入记录、查出记录、按主键更新记录、按主键删除记录、查所有记录、删除表等7次数据库操作。这就是EF-ORM的“Hello,World”。希望您一切顺利。

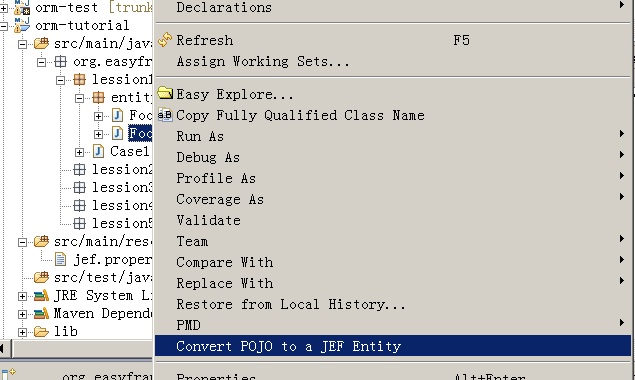
## 正式开始EF-ORM之旅

### EF-ORM的原生Entity

在上面的Case1.java中，我们将会发现操作数据库是如此轻松的一件事。如果您有兴趣可以尝试在jef.properties中配置别的数据源，目前EF-ORM能完全支持的RDBMS参见附录二。

在上面例子其实并不是EF-ORM的基本功能，其中使用到了EF-ORM的一个特殊功能，POJO-Support。事实上，EF-ORM中，会希望您用一个更复杂的对象来描述数据库实体而不是上例中的POJO，这样才能使用EF-ORM的全部特性和API。

我们再来一次。将刚才的Foo.java拷贝一份，名叫Foo2.java。除了类名变化外内容无需修改。然后我们在资源树上，文件上点右键，选择“Convert POJO to a JEF Entity ”。



您将发现，类的代码变成了下面所示的内容

|  |
| --- |
| **package** org.easyframe.tutorial.lesson1.entity;  **import** java.util.Date;  **import** javax.persistence.Id;  **import** javax.persistence.Entity;  @Entity  **public** **class** Foo2 **extends** jef.database.DataObject {  @Id  **private** **int** id;  **private** String name;  **private** Date created;  **public** **int** getId() {  **return** id;  }  **public** **void** setId(**int** id) {  **this**.id = id;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** Date getCreated() {  **return** created;  }  **public** **void** setCreated(Date created) {  **this**.created = created;  }  **public** **enum** Field **implements** jef.database.Field {  *id*, *name*, *created*  } } |

插件会自动在类中插入一些代码，包括JPA实体定义的Annonation，由enum构成的EF-ORM数据元模型(Meta Model)。然后我们可以用更简洁的代码，来重复上面Case1.java的案例。

|  |
| --- |
| **package** org.easyframe.tutorial.lesson1;  **import** java.sql.SQLException;  **import** java.util.Date;  **import** java.util.List;  **import** jef.codegen.EntityEnhancer;  **import** jef.database.DbClient;  **import** org.easyframe.tutorial.lesson1.entity.Foo2;  **import** org.junit.Assert;  **import** org.junit.Test;  **public** **class** Case2 {  @Test  **public** **void** simpleTest() **throws** SQLException{  DbClient db=**new** DbClient();  **new** EntityEnhancer().enhance("org.easyframe.tutorial");    //创建表  db.createTable(Foo2.**class**);    Foo2 foo=**new** Foo2();  foo.setId(1);  foo.setName("Hello,World!");  foo.setCreated(**new** Date());  db.insert(foo); //插入一条记录    //从数据库查询这条记录  Foo2 loaded=db.load(foo);  System.*out*.println(loaded.getName());    //更新这条记录  loaded.setName("EF-ORM is very simple.");  db.update(loaded);    //删除这条记录  db.delete(loaded);  List<Foo2> allrecords=db.selectAll(Foo2.**class**);  Assert.*assertTrue*(allrecords.isEmpty());    //删除表  db.dropTable(Foo2.**class**);  }  } |

在Case2.java中，我们可以发现变化——所有的数据库操作都可以直接在DbClient对象上进行了。

细心的同学可能会发现，update语句从原来的

update FOO set CREATED = ?, NAME = ? where ID=?

变成了

update FOO2 set NAME = ? where ID=?

这才是原生的EF-ORM的实现。事实上EF-ORM并不支持直接操作POJO类型的实体。所有EF-ORM中操作的Entity都必须实现jef.database.IQueryableEntity接口。这个接口提供了更多的方法，将各种复杂查询，更新操作等功能集成起来，比如上例中的update语句之所以被优化，就是因为EF-ORM检测到执行过setName方法，而其他字段并未执行过set方法。

上面的例子中，将POJO转换为原生的实体是使用JEF插件进行的，如果您熟练了，也可以自己编码。当然，更多的场合，您的实体是从数据库或PDM文件里导入的，这些场合下都会生成原生的实体。

### POJO or Non POJO

同学们可能会有很多吐槽，我们这里用问答的形式来讲述。

Q: 为什么要实现IQueryableEntity接口或继承DataObject，有什么用处？

A: EF的设计的一个主要目的是提高开发效率，减少编码工作。为此，可以让开发者“零配置”“少编码”的操作数据库大部分功能。而数据库查询条件问题是所有框架都不能回避的一个问题，所以我经常在想——既然我们可以用向DAO传入一个Entity来实现插入操作，为什么就不能用同样的方法来描述一个不以主键为条件的update/select/delete操作？为什么DAO的接口参数老是变来变去？为什么很多应用中，自行设计开发类来描述各种业务查询条件才能传入DAO？为什么我们不能在数据访问层上花费更少的时间和精力?

JPA1.0和早期的H框架，其思想是将关系型数据库抽象为对象池，这极大的限制了本来非常灵活的SQL语句的发挥空间。而本质上，当我们调用某H框架的session.get、session.load、session.delete时，我们是想传递一个以对象形式表达的数据库操作请求。只不过某H框架要求（并且限制）我们将其视作纯粹的“单个”对象而已。JPA 2.0为了弥补JPA1.0的不足，才将这种Query的思想引入为框架中的另一套查询体系——Criteria API。事实上针对单个对象的get/load/persist/save/update/merge/saveOrUpdate API和Criteria API本来就为一体，只不过是历史的原因被人为割裂成为两套数据库操作API罢了。

因此，我认为对于关系型数据库而言——Entity和Query是一体两面的事物，所谓Query，可以包含各种复杂的查询条件，甚至可以作为一个完整的SQL操作请求的描述。为此，EF彻底将Entity和Query绑在了一起。这种思想，使得——

1. 开发人员需要编写的类更少。开发人员无需编写其他类来描述复杂的SQL查询条件。也无需编写代码将这些查询条件转换为SQL/HQL/JPQL。DAO层也不会有老要改来改去的接口和API，几乎可以做到零编码。
2. 对单个对象进行CRUD的操作API现在和Criteria API合并在一起。Session对象可以直接提供原本要Criteria API才能提供实现的功能。API大大简化。
3. IQueryableEntity允许你将一个实体直接变化为一个查询（Query），在很多时候可以用来完成复杂条件下的数据查询。比如 ‘in (?,?,?)’， ‘Between 1 and 10’之类的条件。  
   xxQL有着拼装语句可读性差、编译器无法检查、变更维护困难等问题，但是却广受开发人员欢迎。这多少有历史原因，也有Criteria API设计上过于复杂的因素。两者一方是极端灵活但维护困难，一方是严谨强大而学习和编写繁琐，两边都是极端。事实上JPA的几种数据查询方式存在青黄不接的问题。选择查询语言xxQL，项目面临后续维护困难，跨数据库移植性差；选择Criteria API，代码臃肿，操作繁琐，很多人望而却步。EF的设计思想是使人早日摆脱拼装SQL/HQL/JPQL的困扰，而是用（更精简易用的）Criteria API来操作数据库。

不再执着于POJO、还带来了另外一些方面的好处，比如继承DataObject类后可以实现对set方法调用的记录，判断一个字段有没有被赋值，用来优化查询、更新等操作。还有延迟加载特性需要每个entity中记录延迟加载的相关信息。

Q：哎~~我以前用过H框架和My框架都是用POJO映射记录的啊。

A：基于API的功能和易用性之间平衡，EF-ORM最终还是决定让所有的实体实现一个接口，好在这个接口倒无需编写任何代码，直接继承jef.database.DataObject就可以了。从使用者编写的代码来看，倒也能算是”伪POJO”，并不会增加开发的复杂性。

Q: 继承DataObject对象后，实体变重了，会不会影响性能？

A: DataObject中的所有字段都是默认为null的，构造方法中不会对其进行初始化，因此对性能几乎没有影响。事实上和H框架的对比测试结果表明了这一点。

Q: Java的单继承结构决定了类只能继承一个父类，这会不会影响开发设计的灵活性？

A: 多个Entity依然可以互相继承，只不过是最顶层的Entity需要继承DataObject类。对灵活性的影响多少有一点，但并不明显。

Q: 如果我要将Entity使用反射拷贝到别的对象，或者转换为JSON/XML。那么这些父类中的属性不是会干扰我功能的正确性吗?

A: 这个问题也考虑过了。父类DataObject中设计的方法，都是和POJO的getter/setter的格式有区别的，反射中一般不会认错。其次这些属性都是标记为transient的，大部分序列化框架（包含JDK内部序列化、Google Gson、FastJSON、JAXB等在序列化时都会忽略这些属性）。

Q：但是前面也演示了直接使用POJO操作数据库，这样用不行吗？

A: 在小型项目中可以这样用，但这样做并没有发挥出EF-ORM的特色，事实上这样做性能较差，而且EF-ORM特色Criteria API也无法使用了。POJO的支持更多是为了兼容性的考虑，以及降低一些特定条件下的使用门槛设计的。支持POJO的API都集中在CommonDao上，这是在Spring集成下的一个通用DAO实现。实际上EF-ORM的核心API中都是不支持POJO的。

总之，POJO的问题就这样了，如果您有耐心继续向后阅读，就会明白为什么在这个地方做出如此的取舍，根本原因在于EF-ORM框架和JPA体系框架的设计思想的不同。

### 实体的增强

细心的同学还会问，从上面的例子看，Foo2.java中setName()方法的代码中只有一个简单的赋值语句，没有任何监听器之类的设计，EF-ORM怎么可能监测到这个赋值语句被执行呢？

这就必须解释，在OpenJPA和H框架中都有的增强（Enhancement）概念了。实际上在上面的例子中，我们执行了

|  |
| --- |
| **new** EntityEnhancer().enhance("org.easyframe.tutorial"); |

这个调用实际上对我们编译后的Foo2.class文件进行了代码植入，这种植入称为增强。EF-ORM遵循Java EE 中的JPA规范，允许为实体类进行增强。增强的意思就是说在编译或者运行时修改实体类的一些行为，使得实体能支持更好的ORM特性。

关于实体增强，有兴趣的朋友可以阅读一下OpenJPA中的介绍：<http://openjpa.apache.org/entity-enhancement.html>

H框架中也有增强，以前H框架总是要依赖一个CGlib，现在则是离不开javassist，就是这个道理。

和H框架在运行时动态执行增强不同，EF-ORM为了更好的性能，直接使用ASM(以前也用过Javassist)直接对磁盘上的静态类做增强。我们分三种情况描述EF-ORM是如何让增强操作不再困扰您的。

#### 使用Maven构建时

可以配置Maven-Plugin，使其在编译完后自动扫描编译路径并执行增强操作。jef-maven-plugin-1.9.0.RELEASE.jar在之前的下载包中可以下载到。

|  |
| --- |
| <plugin>  <groupId>jef</groupId>  <artifactId>jef-maven-plugin</artifactId>  <version>1.9.0.RELEASE</version>  <executions>  <execution>  <goals>  <goal>enhance</goal>  </goals>  </execution>  </executions>  </plugin> |

#### 在Eclipse中运行单元测试时

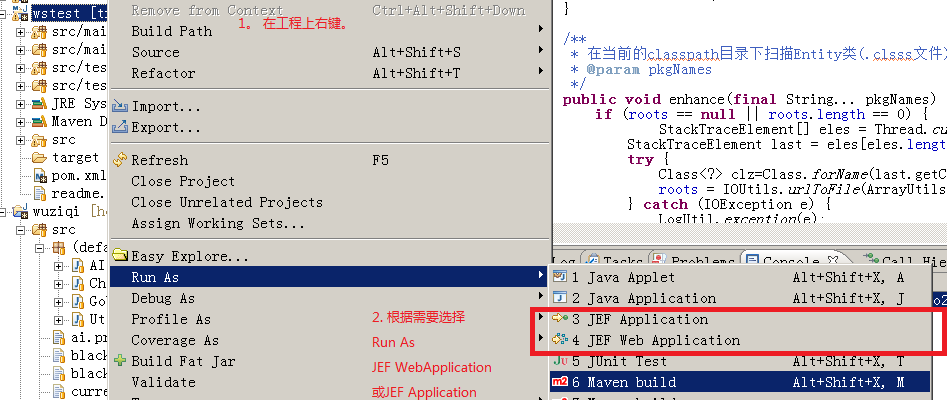
Eclipse会经常自动构建，因此即使我们执行了手工增强，也不能保证当运行单元测试时，类已经做过增强，比较直接的办法就是和上面的例子一样，在单元测试的开始(@BeforeClass)，执行一次

|  |
| --- |
| **new** EntityEnhancer().enhance("org.easyframe.tutorial"); //参数是要增强的类的包名。可传入多个。 |

#### 在开发调试时

JEF插件可以支持在开发时动态增强实体，其原理和前面的三种方式不同，是动态的。通过使用自定义的ClassLoader，在类加载时自动增强类。

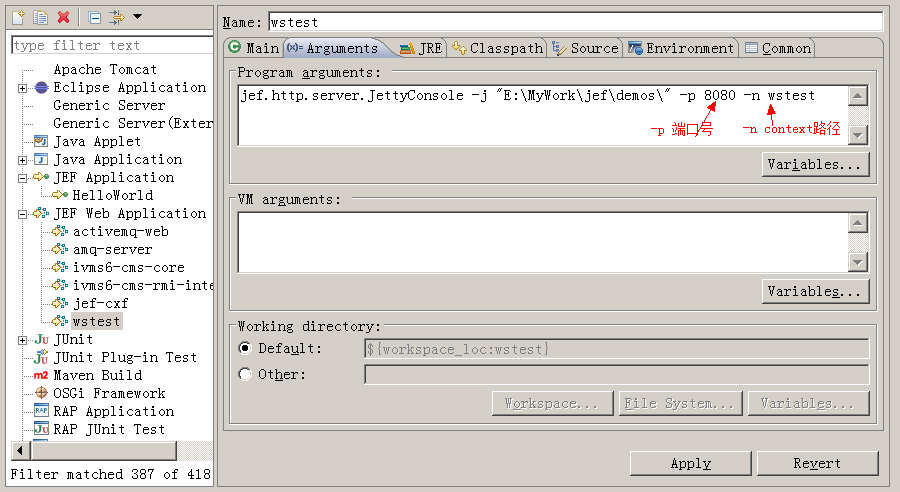
操作方式如下：



JEF Application是运行指定类的Main方法。JEF Web Application则是启动一个内置的Jetty，然后自动查找WEB-INF目录，并按Java的Web开发规范，将工程发布出来。

这两种方式都十分简单。使用这种方式运行的程序中的Entity都不需要手工增强，过程对开发人员透明，有利于编码时提高效率。

其中，Run As / JEF WebApplication中，您可以在Run Configuration界面中调整Context path和Web发布端口。



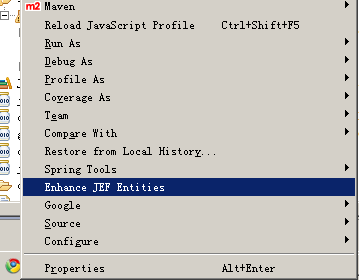
如果要用调试模式运行，启动时选择 ‘Debug As’即可。

三种场景下，EF-ORM都提供了相应的增强操作支持。除了Eclipse中运行单元测试外，你都无需去关注增强操作的存在。

#### 手工增强

最后，还有一种手动方式，可以修改编译目录下的class文件进行增强。

我们在**工程**上点右键，上下文菜单中就有“Enhance JEF Entites“



使用这里的Enhance EF-ORM Entities功能，默认会选中编译输出class文件夹，不用作任何修改直接点确定。

然后可以看到控制台输出：

|  |
| --- |
| enhanced class:org.easyframe.tutorial.lesson1.entity.Foo2  1 classes enhanced. |

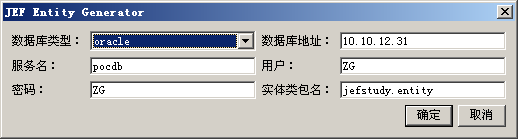
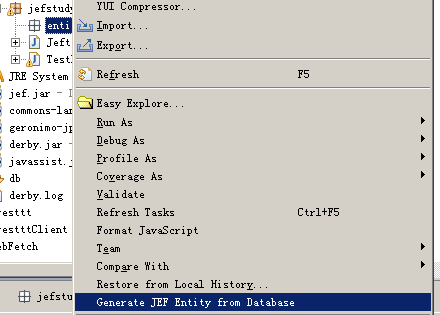
这就说明class文件已经被修改。

|  |
| --- |
| *可能有同学会吐槽，说用某H框架这么久，从来也不需要考虑什么增强不增强的问题。*  *我要说，同学你out了。请你看看某H框架最近在干什么——*  *对性能的追求是无止境的，当你接触的项目规模越大，对性能的考虑就越重要。某H框架也不例外。*  *EF-ORM是在2010年参考Open-JPA的做法，将增强设计成静态的。现在看来，静态增强代替动态增强是大势所趋。* |

## 创建实体

### 从数据库创建实体

先选中我们要生成映射类的包。右键上下文菜单中有“Generate JEF Entity from Database“.



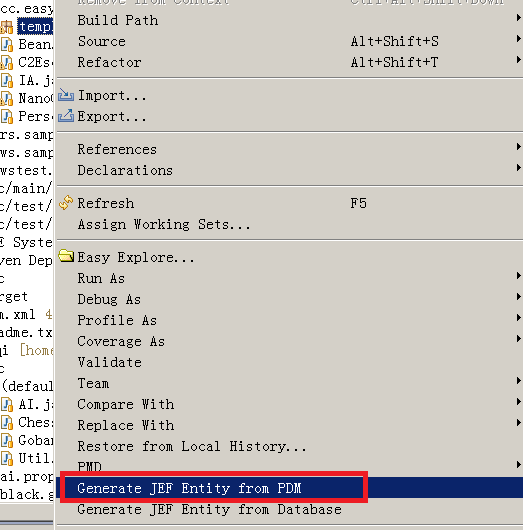
点确定后，出现检测到的数据库表名称，选择其中要映射的表。最后点击‘确定‘。

|  |
| --- |
| jdbc:oracle:thin:@10.10.12.31:1521:pocdb  Generating Class for table:ACCP\_STAT\_MONTH\_ADJUST....  Generating Class for table:ACCP\_TOPUP\_BOOK....  Generating Class for table:ACCP\_TOPUP\_FREE\_RES.... |

正常情况下EF-ORM生成的实体会自动判断数据库主键和主键生成策略。但不会对外键和多表关联操作进行生成，要使用级联功能，您需要按第四章进行操作。

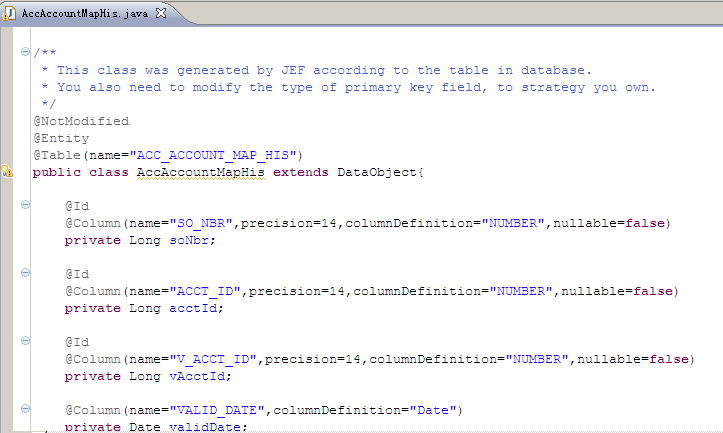
### 从PDM文件导入实体

先选中我们要生成映射类的包。右键上下文菜单中有“Generate JEF Entity from PDM“.



然后按提示步骤操作即可。

一个自动生成的实体可能如下所示：正常情况下所有的主键、表名、字段长度定义都已经以Annotation的形式标注出来。这意味着你可以用这个实体直接进行表的操作了。



自动生成的实体中，关于自增主键的生成规则‘GeneratedValue’还有各种多表关系一般需要手工调整，手工修改JPA注解等介绍参见后文。

### 手工编辑和常用JPA Annotation

#### 注解的使用

除了上述两种导入方式外，您也可以直接编码来创建实体。请看下面这个代码例子中的注释信息

|  |
| --- |
| **package** jef.orm.onetable.model;  **import** java.util.Date;  **import** javax.persistence.Column;  **import** javax.persistence.Entity;  **import** javax.persistence.GeneratedValue;  **import** javax.persistence.GenerationType;  **import** javax.persistence.Id;  **import** javax.persistence.Table;  **import** jef.database.annotation.Index;  **import** jef.database.annotation.Indexed;  **import** jef.database.annotation.Indexes;  **import** jef.database.DataObject;  @Entity  @Table(schema = "ad", name = "ca\_asset") //这里定义表所在的schema和名称，schema可不写  @Indexes(  @Index(name = "IDX\_DATE\_TYPE", definition = "unique", fields = { "thedate", "assetType" })  ) //EF-ORM特有注解，**可以定义该表上的复合索引**。建表时会自动创建索引。  **public** **class** CaAsset **extends** DataObject {  /\*\*  \* Asset ID  \*/  @Id //说明这个字段是主键字段  @GeneratedValue(strategy = GenerationType.*IDENTITY*) //使用列自增生成值，不支持列自增再Sequence  @Column(name = "asset\_id", precision = 6, columnDefinition = "NUMBER", nullable = **false**)  //定义该字段在数据库中的列名，number长度。是否可为null。  @SequenceGenerator(sequenceName="ca\_asset\_seq",name="ca\_asset\_seq")  //可以指定Sequence名称，**但不建议定义**，EF-ORM支持全局配置一个模板，来生成各个表的Sequence名称。  **private** **int** assetId;  /\*\*  \* A unique identifier of account.  \*/  @Id //重要,EF-ORM允许一个对象中有多个@Id字段，即复合主键。  //在某些关系表上，业务键要比物理键实用的多。这个与标准JPA的做法不同。  //**EF-ORM更倾向支持传统的数据库设计，而不是用面向对象来代替数据库设计。**  //因此，如果您正在使用EF-ORM，请在该用业务键的时候大胆的用业务键，  //不需要也不建议为每个表都生成一个物理主键。让更专业的DBA来设计数据库吧。  @Column(name = "acct\_id", precision = 14, columnDefinition = "NUMBER")  **private** Long acctId;  /\*\*  \* Asset type: 0- fund(account book), 1- credit limit, 2- bonus, 3- bank  \* capital, 4- free resource, 5 - cheque.  \*/  @Column(name = "ASSET\_TYPE", precision = 8, columnDefinition = "NUMBER")  @Indexed //EF-ORM特有的注解，在建表时**可以为这个列创建B树索引**。  **private** Integer assetType;  @Column(name = "COMMENTS", length=512, columnDefinition = "varchar")  **private** String normal;  @Column(name = "CONTENT", columnDefinition=”clob”)  @Lob //CLOB字段一般映射为String，也可以映射为File, char[]等  **private** String content;  @Lob //byte[]构成的Lob会映射为BLOB（在某些数据库上为BYTEA）。BLOB在java中还可以映射为  //String, File等。  **private** **byte**[] photo;  @Column(name = "PRICE", precision =12,scale=8, columnDefinition = "number")  //对于小数，precision=12 scale=8的意思是整数部分最多4位，小数部分最多8位。(和Oracle定义一致）  //理解为整数部分最多12位的同学都去面壁！  **private double** price;    @Column(name = "VALID\_DATE",columnDefinition="Date")  //注意，这里定义为Date时，精度为年月日，不含时分秒。定义为Timestamp时，精度到时分秒乃至毫秒。  //操作Oracle数据库也遵守相同的规律。  //Oracle同时具有Date和Timestamp两种类型，但和别的数据库不一样，其Date精度到秒。  //此处我们沿用JDBC标准，Date精度到天。确保实现的可移植性。  **private** Date thedate;    **public** **enum** Field **implements** jef.database.Field {  *acctId*, *assetId*, *assetType*,*thedate,normal,content*  }  //getters and setters  } |

总结一下，

1、上面提到了常见数据类型的定义方法。

而常见的数据类型不外乎varchar, blob,clob number, integer, nclob, nvarchar, double, float, bigint,Boolean等。

值得一提的是，您可以用java当中的Boolean类型。EF-ORM会根据方言的不同，支持原生boolean类型的数据库中，映射为Boolean，不支持原生Boolean的则映射为char(1)。

2、索引的定义方法

3、复合主键的定义方法

4、自增键的定义方法。需要指出的是，如果数据类型为String，而GenerationType=AUTO或IDENTITY时，会使用GUID生成主键，此时要求字段长度36位。

另外，上例中@Table(schema = "ad", name = "ca\_asset") 中指定了表所在的Schema为ad，这项配置是可以在运行环境中被重新映射的，相关特性参见7.3.1节Schema重定向。

#### 映射关系和注解的省略

许多时候注解可以省略，省略时，EF-ORM会根据默认的数据类型计算相应的数据库存储类型。事实上，columnDefinition也可以写成int, double, integer,varchar2等各种标准的SQL类型。因此，如果不指定Column类型，EF-ORM会默认的生成Column的类型如下

|  |  |
| --- | --- |
| **Java类型** | **数据库类型** |
| String + @Lob | CLOB |
| String | Varchar(255) |
| Int / Integer | Integer / Number(8) |
| double/Double | Double / Number(16,6) |
| float/Float | Float / Number(16,6) |
| boolean /Boolean | Boolean / char(1) |
| Long / Long | Bigint / Number(16) |
| java.util.Date | timestamp / datetime |
| java.sql.Date | Date |
| java.sql.Timestamp | timestamp / datetime |
| byte[] | BLOB |
| Enum | Varchar(32) |

表 1-1 映射关系

上表列举了您未定义Column注解时，java字段到数据库的映射关系。这也是建议您在实际使用注解时进行的映射方式。在实体加载时，EF-ORM会适配实体与数据库的映射关系，如果发现属于无法支持的映射（EF-ORM能兼容大部分不同数据类型的映射，包括从String到number等），那么会抛出异常，此时您需要修改java的映射字段类型。

值得注意的是EF-ORM支持您使用enum对象和varchar列发生映射，使用枚举类型有助于提高您应用系统的可理解性。

# 基本操作与对象映射

## 基本单表操作

本节介绍几种常用的单表查询方式，EF-ORM还支持更多复杂的查询，后面几节中还会继续描述。

在Hello,World!案例中，我们已经了解了实现单表基于主键进行增删改查操作的方法。我们来回顾一下，在此处和后面的代码清单中，我们都在session对象上操作数据库，这里的session对象可以理解为就是DbClient对象。，

|  |
| --- |
| Foo foo=**new** Foo();  foo.setId(1);  foo.setName("Hello,World!");  foo.setCreated(**new** Date());  //1.插入一条记录  session.insert(foo);    //2.从数据库查询这条记录  Foo loaded=session.load (foo);  System.*out*.println(loaded.getName());    //3. 更新这条记录  loaded.setName("EF-ORM is very simple.");  session.update(loaded);    //4.删除这条记录  session.delete(loaded); |

在上面的Foo对象中，其Id属性被标记为主键。当id属性有值时，load方法,update方法、delete方法都是按主键操作的。上面的操作，归纳起来就是插入记录、按主键加载、按主键更新、按主键删除。

### 复合主键

当有复合主键的场景下，情况是完全一样的：

在所有作为主键的列上增加@Id注解。

代码src/main/java/org/easyframe/tutorial/lesson2/entity/StudentToLesson.java

|  |
| --- |
| **package** org.easyframe.tutorial.lesson2.entity;  **import** java.util.Date;  **import** javax.persistence.Entity;  **import** javax.persistence.Id;  @Entity  //描述学生和课程映射关系的表  **public** **class** StudentToLesson **extends** jef.database.DataObject {  @Id  **private** **int** studentId; //复合主键一  @Id  **private** **int** lessonId; //复合主键二  **private** Date createTime;  **public** **enum** Field **implements** jef.database.Field {  *studentId*, *lessonId*, *createTime*  }  //getter setter省略  } |

**调整复合主键顺序**

在上例中，我们执行

|  |
| --- |
| *db*.createTable(StudentToLesson.**class**); |

建表时的定义是

|  |
| --- |
| *create table STUDENT\_TO\_LESSON(*  *CREATETIME timestamp,*  *STUDENTID int not null,*  *LESSONID int not null,*  *constraint PK\_STUDENTTOLESSON primary key(STUDENTID,LESSONID)*  *)* |

在复合主键中studentId在前，lessonId在后，怎么样其调整为lessonId在前，studentId在后呢？

对于1.8.0以下版本，方法是将lessonId的定义移动到studentId之前。

1.8.0以后的版本，是在枚举的enum Field中将lesseionId移动到studentId之前。

|  |
| --- |
| *1.8.0之前，EF-ORM是采用class.getDeclaredFields()方法来获取字段顺序的。而JDK也确实按照定义的顺序返回了这些字段，但是java虚拟机规范上说：“The elements in the array returned are not sorted and are not in any particular order.”也就是说，虚拟机规范不保证字段返回的顺序，因此这种调节方式可能无效。*  *1.8.0之后，改为使用enum中出现的顺序来排序字段。* |

有同学可能会问，你搞这个功能有什么意义呢？复合主键字段顺序的无论怎么变化其产生的约束效果是一样的。

实际上这不一样。数据库会为复合主键创建一个同名的复合索引。在上例中，我们如果要查询某个学生(studentId)的课程选修情况，那么studentId列在前可以让我们无需再为studentId创建单独的索引。事实上Oracle DBA们往往会称其为“复合索引前导列”。精确控制复合主键的顺序可以改善你的数据库设计中的索引规划，而无故增加索引则会造成每次CRUD时无谓的开销。

**其他**

使用过JPA的同学可能会说，EF-ORM这里的处理不规范啊，JPA规范中碰到复合主键都是必须定义一个@IdClass，要么作为一个独立类要么用@EmbeddedI嵌入对象中，你这种支持方法不是标准JPA的支持方式。

我的回答是：JPA的规范在实际使用时问题很多。由于凭空增加了一个对象的关系，在实际编码时需要增加大量的非null判断、主键对象必须实现hashCode和equals方法、主键值设置时是替换主键对象还是更新主键对象方法不统一，操作起来问题很多。这种坏味道甚至影响了代码的健壮性。实际情况下，大部分JPA使用者和H框架使用者会主动的给表增加一个物理键，避免使用业务键（复合主键），但操作中又基本上使用业务键来增删改查，物理键几乎无用。这样的因素也将影响DBA对数据库的设计发挥。

因此，在EF-ORM中，不支持标准JPA的复合主键方式，而是采用上文所述的方式。就像传统关系型RDBMS表结构定义那样，为平铺的属性增加两个主键的标识即可，类似于在PowerDesigner中定义主键那样。我还是那么想，既然用了关系数据库，就要让关系数据库的优势体现出来。

### 主键自增

#### 自增值自动回写

正常情况下，执行插入操作后，数据库生成的自增值主键值都会被设回对象中，我们看下面的示例。

src/main/java/org/easyframe/tutorial/lesson2/entity/Student.java

|  |
| --- |
| **package** org.easyframe.tutorial.lesson2.entity;  **import** javax.persistence.GeneratedValue;  **import** javax.persistence.GenerationType;  **import** javax.persistence.Id;  **import** javax.persistence.Entity;  @Entity  **public** **class** Student **extends** jef.database.DataObject {  /\*\*  \* 学号  \*/  @Id  @GeneratedValue(strategy=GenerationType.*AUTO*) //使用GeneratedValue表示自增键值  **private** **int** id;  /\*\*  \* 姓名  \*/  **private** String name;  /\*\*  \* 年级  \*/  **private** String grade;  /\*\*  \* 性别  \*/  **private** String gender;  **public** **enum** Field **implements** jef.database.Field {  *id*, *name*, *grade*, *gender*  }  //getter setter省略  } |

src/main/java/org/easyframe/tutorial/lesson2/Case1.java

|  |
| --- |
| @Test  **public** **void** studentAutoIncreament() **throws** SQLException{  *db*.createTable(Student.**class**);  Student s=**new** Student();  s.setName("Jhon Smith");  s.setGender("M");  s.setGrade("2");  *db*.insert(s);    Assert.*assertTrue*(s.getId()>0);  System.*out*.println("自增键值为："+s.getId());  } |

从上例可以发现，当insert方法执行后，自增值即被回写到对象中。每次调用Session中的操作方法，都是即时操作数据库的。再说一次，EF-ORM是JDBC的封装，目的是帮助用户更精确和更简便的操作关系数据库。所以不会有一个“对象池”的概念，将您所有操作记录下来，到最后再执行。

此外，无论在何种数据库下、无论在单条还是batch模式下（Batch模式以后会讲到），插入操作生成的值都会被回写到对象中。

对于自增值的生成有很多用法，后文详细讲述。

#### 四种JPA主键生成方式

EF-ORM支持JPA的主键生成方式。以下是JPA注解的实现方式

|  |
| --- |
| @Id  @GeneratedValue(strategy = GenerationType.*IDENTITY*)  **private** String id; |

JPA注解中的GenerationType有四种，其用法如下

|  |  |  |
| --- | --- | --- |
| 注解类型 | 含义 | 在EF-ORM中的用法 |
| AUTO | 自动，根据当前数据库等自动选择自增实现 | 1.为int或long类型时，根据数据库自动选择 Identity > Sequence > Table 。(由于数据库多少都支持前两个特性，所以实际上Table默认不会被使用 )  2.如果字段是String类型，使用GUID策略(即32位guid)。 |
| IDENTITY | 使用列自增特性 | 1. 为int或long类型时，使用表的列自增特性  2.如果字段是String类型，使用GUID策略(即32位guid)。 |
| SEQUENCE | 使用Sequence。 | 使用Sequence生成自增值 |
| TABLE | 使用Table模拟Sequence。 | 使用Table生成自增值 |

如果不配置，那么会隐式使用Annotation的默认值AUTO。

下面列举几种规则的用法和效果，嫌麻烦的可以直接跳转到3.1.2.6章看结论。

#### 配置的简化

上述四种配置的基础上，EF-ORM进行了一些配置的简化。形成了一套独有的规则。

**1 . 识别为GUID**

当字段类型为String时，如果自增方式为Identity或Auto那么使用guid策略。 但如果为Sequence和Table，EF-ORM还是认为该字段属于自增值，数据库中会使用Int等数值类型，只不过在java类中自动转换为String。

1. **识别为AUTO**

当EF-ORM获得一个自增值的注解配置后，默认情况下会将Sequence和Identity两种方式忽略，理解为AUTO方式。因为Identity和Sequence方式都有跨数据库移植性差的问题，为了更好的支持数据库Native的自增实现方式， EF-ORM默认Identity和Sequence方式都等同于Auto。除非用户禁用Native支持功能，即在jef.properties中配置

|  |
| --- |
| db.autoincrement.native=false |

**3、按本地数据库能支持的方式实现自增**

对于AOTU类型，按 数据库列自增 > Sequence > Table这样的优先级，使用数据库能支持的方式实现自增。

对于Sequence类型，按 Sequence > Table这样的优先级，使用数据库能支持的方式实现自增。

对于Table类型，使用Table模拟Sequence方式实现自增。由于默认情况下数据库多少都会支持优先度更高的两种特性之一。所以Table方式会很少用到。

4、**不使用数据库列自增方式的场合**

以下两种情况下，EF-ORM会认为列自增特性不可使用

* 数据库方言中指定不支持的，如Oracle，这是大家容易想到的。
* 对象开启了分库分表功能的，由于分库分表后列自增值不能保证唯一性，因此会认为不支持。

#### Sequence或Table下的其他配置

可以用JPA注解@SequenceGenerator或@TableGenerator两者，为自增值设置Sequence名或表名。

例如：

|  |
| --- |
| @GeneratedValue(strategy = GenerationType.*TABLE*)  @TableGenerator(name = "AA1", initialValue = 1, allocationSize = 10, valueColumnName = "TABLE", pkColumnValue = "SEQ\_VALUE", table = "T\_SEQ\_1")  **private** **int** tableSeq; //指定生成Sequence的表名为T\_SEQ\_1  @GeneratedValue(strategy = GenerationType.*SEQUENCE*)  @SequenceGenerator(name = "VVVV", sequenceName = "SEQ\_A1", initialValue = 1000)  **private** **int** seq; //指定SEQUENCE的名称为SEQ\_A1 |

通过这种方式，您也可以让多张表共享一个Sequence或Table。

需要注意的是，@SequenceGenerator仅在GenerateType为SEQUENCE时生效，@TableGenerator仅在GenerateType为TABLE时生效。这里的GenerateType不是指你注解里写的是什么，而是按照3.1.2.2的简化规则计算后生效。 也就是说，这两个注解不会影响EF-ORM用什么方式来实现自增，只会在确定自增方式之后，调整自增的实现参数。

默认情况下，您都不需要配置这两项内容，EF-ORM全局配置参数中可以配置命名模板，会根据模板来计算Sequence的名称。如果您单独配置了Sequence的名称，那么会覆盖全局配置。但从项目实践看，我们更建议您使用模板的方式，全局性的命名Sequence。

默认情况下，全局Sequence的名称为“%\_SEQ”,其中%是表的名称。也就是说，名为”TABLE1”的表，其使用Sequence为”TABLE\_SEQ”。

如果表的名称大于26个字符，会截取前26个字符。如果你有两张表名称达到或超过26个字符，并且前26个字符都是一样的，那么这两张表会公用同一个SEQUENCE。这是为了防止出现SEQUENCE名称大于30个字符，在Oracle下你无法创建大于30个字符的数据库对象。

前面既然说了Sequence名称模板默认为”%\_SEQ”，那也意味着您可以修改名称模板，比如您可以把它改为”S\_%” 。修改方法是jef.properties中。

|  |
| --- |
| sequence.name.pattern=S\_% |

对于Table模拟Sequence，也可以全局配置， 可以是全局一张表实现所有Table Sequence值。例如创建一张名为GLOBAL\_SEQ\_TABLE的表，要启用此功能，可以在jef.properties中

|  |
| --- |
| db.public.sequence.table=GOOBAL\_SEQ\_TABLE |

综上所述，一般情况下，我们不需要使用@SequenceGenerator 和 @TableGenerator注解。

#### Sequence或Table的性能优化

**Hilo算法**

熟悉数据库的人都知道，大量数据并发插入时，频繁到数据库中去获取Seqence性能是很差的。

某H框架支持hilo算法来优化Sequence的取值性能，原理是对获得到的值乘以系数，将值的范围放大后自动填充范围内的值，从而一次访问数据库客户端能得到多个不重复的序列值。

在EF-ORM中，将Hilo算法作为一种修饰器。当使用Table和Sequence两种方式，都可以添加Hilo算法的修饰。

比如

|  |
| --- |
| @Id  @GeneratedValue(strategy = GenerationType.*SEQUENCE*)  @HiloGeneration(maxLo=20)  **private** **int** idSeqHilo; |

或者

|  |
| --- |
| @Id  @GeneratedValue(strategy = GenerationType.*TABLE*)  @HiloGeneration(maxLo=20)  **private** **int** id; |

其中maxLo的含义和H框架相同。每次都是生成value \* (maxLo + 1) 到 value \* (maxLo + 2)范围内的值，算法此处不解释，请自行百度。

@HiloGeneration仅对SEQUENCE和TABLE两种算法有效，如果按上一节的规则，最终使用数据库列自增方式，那么@HiloGenerator注解会失效。比如我们可以配置下面的自增主键：

|  |
| --- |
| @Id  @GeneratedValue(strategy = GenerationType.*AUTO*)  @HiloGeneration(maxLo=20)  @SequenceGenerator(name = "S1", sequenceName = "S\_TABLE1", initialValue = 1000)  @TableGenerator(name = "S2", valueColumnName = "TABLE", pkColumnValue = "SeqValue", table = "TABLE\_SEQ\_GOLBAL")  **private** **int** id; |

上面的注解配置，在不同数据库下会产生不同的表现——

在MySQL下，@HiloGeneration、@SequenceGenerator、@TableGenerator都不会生效。（使用自增列，故这些生成规则都无效）

在Oracle下，@HiloGenerator和@SequenceGenerator会生效。（Oracle下默认使用SEQUENCE）

如果这张表启用了分库分表功能，那么在Oracle下情况不变。而在MySQL下，因为MySQL不支持Sequence，最终实际上变为TABLE模式，此时@HiloGenerator和@TableGenerator即会生效。

**步长跳跃**

这是另一种优化方式，步长跳跃的意思就是让Sequence或Table中的值不是每次自增1，而是一次性跳好几个数值，然后由EF-ORM自动填补这中间的空白整数。也可以起到一次访问数据库获得多个序列值。

使用Hilo的情况，如果随意修改Hilo的参数，很有可能造成主键冲突。此外数据库中的序列值和表中的ID之间也很难对应起来。从数据库维护角度来看很不直观，因此EF-ORM更为推荐步长跳跃这种方式来优化序列值的获取。

事实上，在使用TABLE方式时，EF-ORM已经用了这个方法一次性将表中的Value值加上较大的一个值。这个值就是 sequence.batch.size，默认20，也就是说一次访问数据库就获得20个序列值。

而如果使用数据库原生的Sequence，那么如何优化呢？

对于Oracle数据库，你可以直接将Sequence的步长调节为更大的值，例如40.

|  |
| --- |
| alter sequence TABLE1\_SEQ increment by 40 cache 20; |

EF-ORM会在首次使用Sequence前，检查Sequence的步长，这是自动的。非Oracle下，EF-ORM不会主动检查步长。

如果在非Oracle数据下使用Sequence跳跃，例如Postgres，可以使用jef.properties参数

|  |
| --- |
| *#当设置为-1时，强制检查Sequence步长；0时仅检查Oracle步长；1时步长固定为1；其他>0数值：步长固定为配置的值*  *db.sequence.step=-1* |

当配置为>0的数值时，Sequence步长固定为指定值（不推荐这样做），因为这样配置时如果和数据库中的Sequence实际步长不匹配，可能出现错误。

正常情况下，如果步长为5，select X\_SEQ.nextVal from dual如果返回结果=10，那么实际上EF-ORM会获得10,11,12,13,14这样五个值。再次调用后，数据库端的Sequence值=15，实际使用序列号到19为止。

由于非Oracle数据库没有有效的Sequence步长检测机制，在非oracle上检测步长将消耗掉一个Sequence值。

在Oracle上，如果数据库用户权限被严格限制，可能会无权限访问user\_sequences视图，此时您需要将db.sequence.step配置为一个正数来避免自动检查步长。

#### 配置方法和总结

对前面的种种规则比较含糊的同学，直接看这一节就行了。

|  |  |  |  |
| --- | --- | --- | --- |
|  | **H框架中的用法** | **EF-ORM对应** | **EF-ORM下特点** |
| identity | 数据库列自增特性，不支持Oracle。 | @GeneratedValue(strategy = GenerationType.IDENTITY) | 在Oracle下自动转换为Sequence |
| sequence | 仅在支持SEQUENCE的数据库上可用，必须指定Name。如果没有指定则默认为全局唯一的’hibernate\_sequence’。 | @GeneratedValue(strategy = GenerationType.SEQUENCE) | Oracle等支持Sequence的数据库下Sequence会自动命名、自动创建。  MySQL等不支持的数据库下自动转换为Identity或Table。 |
| hilo(table hilo) | 使用高/低位算法生成数值型，给定一个表和字段作为高位值的来源，默认的表是hibernate\_unique\_key，默认的字段是next\_hi。它将id的产生源分成两部分，DB+内存，然后按照算法结合在一起产生id值。目的是减少访问次数提高效率 | @HiloGeneration  @GeneratedValue(strategy = GenerationType.TABLE) | 可支持所有数据库。通过序列步长调整也可保持高性能。 |
| native | 对于 oracle 采用 Sequence 方式，对于MySQL 和 SQL Server 采用identity（自增主键生成机制） | @GeneratedValue(strategy = GenerationType.AUTO) | Sequence可以做到一表一Sequence。 自动命名和创建 |
| seqhilo | 由于局限于支持Seq的数据库，所以少用。 | @HiloGeneration  @GeneratedValue(strategy = GenerationType.SEQUENCE) | 默认情况下还可以使用数据库列自增。仅在Oracle下生效。 |
| uuid.hex uuid.string | 从H框架3.0开始已经不再支持uuid.string | @GeneratedValue(strategy = GenerationType.IDENTITY) | 数据类型为String时自动适配。  效果和H框架一样 |
| assigned | 人工指定 | 不配置 | EF-ORM可支持assigned，还支持手动和自增方式混用。 |
| foreign | 使用外部表的字段作为主键 | 不支持 | -- |

结论，对于大部分情况，大家只要对自增主键配置

|  |
| --- |
| @Id  @GeneratedValue(strategy = GenerationType.IDENTITY) |

就行了，别的啥都不用操心了。在常见数据库中，如果是在Oracle下发现获取Sequence不够快，那么可以按照3.1.2.5的方法进行优化。

### 查询操作

在了解了单表插入和主键自增后，我们来看其他操作。在类jef.databse.Session当中，有着

load(T entity)和select(T entity)方法。之前的例子中我们一直用这个方法来查询插入的记录。EF-ORM会按照对象中存在的主键进行查询。这是不是这个方法的唯一用法呢？除了这个方法之外，有没有其他方法按主键查询对象呢？

#### 按主键或模板查询

其实、前面用到的load方法和select方法并不仅仅是按照主键进行数据查询，它们其实是将传入对象当做一个模板来使用的，参见下面一组例子。

|  |
| --- |
| @Test  **public** **void** testLoadAndSelect() **throws** SQLException{  //创建一个模板对象,当模板对象主键有值时，按模板对象的主键查询。  {  Student query=**new** Student();  query.setId(3); //id是主键。  Student st=*db*.load(query); //按主键查询返回单条记录  }  //直接按主键查询,当对象为复合主键时,可传入多个键值。  //键值数量必须和复合主键的数量一致，顺序按field在emun Field中的出现顺序。  {  Student st=*db*.load(Student.**class**, 3);  StudentToLesson stl=*db*.load(StudentToLesson.**class**, 3, 1);  }  //load方法都是查询单值的，select方法可以查询多值  //创建一个模板对象,当模板对象的字段有值时，按这些字段查询  //类似于某H框架的 findByExample()。  {  Student query=**new** Student();  query.setGender("F");  query.setGrade("2");  List<Student> sts=*db*.select(query);//查出所有Gender='F' and grade='2'的记录。  }  //如果一个对象的复合主键没有全部赋值，那么也当做普通字段对待  //最终效果和findByExample()一样。  {  StudentToLesson query=**new** StudentToLesson();  query.setLessonId(1);  List<StudentToLesson> sts=*db*.select(query);//查出所有lessonId='1'的记录。    }  //如果一个对象的主键都赋了值，非主键字段也赋值。那么非主键字段不会作为查询条件  //因为框架认为主键字段足够定位记录，所以非主键不用作查询条件。  {  Student query=**new** Student();  query.setGrade("1");  query.setId(12);  List<Student> sts=*db*.select(query); //查询条件为 id=12。grade不用作查询条件  }  } |

从上面的例子，我们可以发现

1. load和select区别在于一个返回单条记录，一个返回多条记录的List。
2. load和select对于传入的条件，判断其主键是否就绪，有则按主键查询，没有则将传入对象当做一个模板查询。
3. 另外有一个load(Class, Serializable…)的方法，可以直接传入对象类和主键值来加载单条记录。
4. 复合主键必须全部都赋值才认为主键就绪，否则也是将传入对象当做模板查询。

#### 更复杂的条件

按主键或模板查询能适应不少查询场景，但还远远不够。比如我们要查询所有姓名中带有”Jhon”字样的学生。用SQL语句表达的话，就是

|  |
| --- |
| where name like ‘%Jhon%’ |

对象Student看似无法描述这种带有运算符的条件。但不要忘了我们的Student已经实现了jef.database.IqueryableEntity接口，这使Student对象携带复杂的条件成为可能。

代码清单： src/main/java/org/easyframe/tutorial/lesson2/Case1.java

|  |
| --- |
| @Test  **public** **void** testSelect\_Like() **throws** SQLException{  Student query=**new** Student();  //在Student对象中添加Like条件  query.getQuery().addCondition(  Student.Field.*name*,Operator.*MATCH\_ANY*, "Jhon"  );  List<Student> sts=*db*.select(query);    Assert.*assertEquals*(sts.size(), *db*.count(query.getQuery()));  } |

在上例中，我们从Student对象中通过getQuery()得到一个Query对象，我们可以在这个Query对象上添加各种复杂条件。我们在后面的章节可以了解到，这里可以放入几乎SQL语句所能写出的所有查询条件和修饰，例如Order By等等。Query对象的出现，让我们意识到不仅仅是复杂条件可以被对象所描述。它会带来更多的灵活用法。

在Query对象中，我们可以添加条件，每个条件中可以包含运算符（Operator），在jef.database.Condition.Operator类中，定义了以下几种运算符

|  |  |  |  |
| --- | --- | --- | --- |
| **运算符** | **对应SQL** | **含义** | **要求参数类型** |
| **EQUALS** | = | 等于 | 文本、数值、日期等 |
| **GREAT** | > | 大于 | 文本、数值、日期等 |
| **LESS** | < | 小于 | 文本、数值、日期等 |
| **GREAT\_EQUALS** | >= | 大于等于 | 文本、数值、日期等 |
| **LESS\_EQUALS** | <= | 小于等于 | 文本、数值、日期等 |
| **MATCH\_ANY** | like ‘%*param*%’ | 匹配字符串任意位置 | 文本 |
| **MATCH\_START** | like ‘*param*%’ | 匹配字符串头部 | 文本 |
| **MATCH\_END** | like ‘%*param*’ | 匹配字符串尾部 | 文本 |
| **IN** | in (*param,param*…) | 属于（列表） | 数组、List、Set、Collection、逗号分隔多值的文本 |
| **NOT\_IN** | not in(*param,param*…) | 不属于（列表） | 数组、List、Set、Collection、逗号分隔多值的文本 |
| **NOT\_EQUALS** | != 或者 <> | 不等于 | 文本、数值、日期等 |
| **BETWEEN\_L\_L** | between *param* and *param* | 介于xx和xx之间 | 数组、List、Set、Collection、逗号分隔多值的文本 (长度为2) |
| **IS\_NULL** | is null | 为NULL | 无 |
| **IS\_NOT\_NULL** | is not null | 不为NULL | 无 |

这些运算符都可以通过addCondition方法加入到条件中。在传入参数时，注意部分运算符是只针对集合操作进行的，必须传入集合对象。

部分操作仅针对文本，如果传入非String对象，将被强制转换为String对象处理。

## 你不是在操作‘对象池’

前面的例子中EF-ORM表现得和H框架非常相似，但是前面已经讲过，EF-ORM是基于数据库和SQL的，它的API是对JDBC的封装。而H框架则是视图让你面对一个“对象池”（一级缓存），要求你**按主键**来操作每一个对象。而EF-ORM完全没有这个限制，getQuery()带来的是概念上的变化。

下面是一个例子

代码清单： src/main/java/org/easyframe/tutorial/lesson2/Case1.java

|  |
| --- |
| @Test  **public** **void** testUpdateAndDelete\_WithLike() **throws** SQLException{  Student s=**new** Student();  s.setGender("F");  s.getQuery().addCondition(Student.Field.*name*,Operator.*MATCH\_ANY*, "Mary");    *db*.update(s);  //相当于执行  //update STUDENT set GENDER = 'F' where NAME like '%Mary%'    s.getQuery().clearQuery();//清除查询条件  s.getQuery().addCondition(Student.Field.*id*,Operator.*IN*, **new** **int**[]{2,3,4});  *db*.delete(s);  //相当于执行  //delete from STUDENT where ID in (2, 3, 4)  } |

上例中，两次数据库操作对应的SQL语句已经写在注释代码里了。

我们可以发现，同样是我们前几章介绍的update方法和delete方法，在有了Query对象后产生了不同于之前的效果。它们并不仅仅只能按主键更新或删除某条记录。事实上在上例中，Student这个对象并不代表任何一个“学生”，它就是一条完整SQL语句，一次完整的数据库操作请求。

现在明白了IQueryableEntity的含义吗？每个Entity不仅仅是一条记录的载体，通过getQuery()方法，Entity同时也可以表现为一个SQL操作，这个操作并不仅仅限于单条记录。

默认情况下，Entity并不对应任何Query，但如果您将其用于查询、更新、删除操作，或者调用其 getQuery()方法，就可以让一个Query对象和Entity绑定。通过getQuery()方法和Query的getInstance()方法，你可以在互相绑定的两个对象之间转来转去。如下图：



图3-1 Entity和Query的关系

这对对象，在不同的操作场景下，用法是这样的——

|  |  |
| --- | --- |
| **场合** | **处理规则** |
| **insert** | Query对象无效。总是将Entity当做一条待插入的记录。 |
| **update** | Query部分是where条件，Entity中被设值的数据是update语句的set部分。  如果没有Query部分，那么Entity主键作为where条件，其他字段作为set部分。  如果Query部分和主键都没有，那么不构成一个update请求(抛出异常) |
| **select** | Query部分是where条件  如果Query部分不存在，那么Entity主键作为where条件。  如果主键没有设值，那么Entity作为模板，按其他设了值的字段查询。  如果Entity和Query都没有设置任何值，那么不构成一个select请求（抛出异常） |
| **delete** | 和select规则相同 |

也就是说，前面的例子中，我们传入对象，并且按主键实现update、delete操作时，其实并不是像H框架那样在操作对象池，而是传入了一个SQL请求，是在操作一个关系型数据库。

看到这里大家应该已经明白为什么说EF-ORM和其他ORM不同的根本原因。EF-ORM的API，封装的其实是SQL操作。前面的基于对象主键的CRUD操作，其实是用SQL规则在模拟对象操作的一种“拟态”。

这是EF-ORM和H框架的根本性不同。也正是因为这些不同，我们可以解决很多其他ORM中不够灵活的问题，可以高效的操作关系型数据库。比如

|  |
| --- |
| Student s=new Student();’  s.getQuery().addCondition(Student.Field.*name*,Operator.*EQUALS,* "Mary");  s.setGender("F");  *db*.update(s); |

等效于update set gender=’f’ where name=’Mary’。而在某H框架的API中，先不谈非主键是否可用作update条件的问题，即便是用主键做update条件，H框架也会提示你这是一个游离对象，不能直接做update。也就是说它规定了你只能先按主键查出对象(放入一级缓存)，再按主键写入到数据库里。试想如果我们要批量更新多条记录，对于这种场景循环逐条处理，效率会怎么样。

当然了某H框架还有法宝，用一种Query Language(HQL)来封装另一种Query Language(SQL)，这种想法也是挺有“创意”的。不过也不能老抓着人家的黑历史不放。事实上业界对于使用API和xQL来封装数据库操作的优劣早有定论。JSR 317 JPA 2.0规范中增加了Criteria API，其实是比xQL更方便并能提高软件可维护性的手段。H框架除了实现JPA 2.0规范以外，本身也有非常强大的Criteria API。个人认为H框架仍然是业界最好的ORM，没有之一。不过是觉得EF-ORM更适应在性能和数据规模有较高要求的行业而已。

回归正题，在理解了Query的作用之后，我们可以看一下Query的简单用法。

### 查询条件与字段排序

查询免不了Order by, group by, having等等。这里先说order by。

代码清单：src/main/java/org/easyframe/tutorial/lesson2/Case1.java

|  |
| --- |
| @Test  **public** **void** testSelect\_LikeAndEtc() **throws** SQLException {  Student s = **new** Student();  s.getQuery()  .addCondition(Student.Field.*name*, Operator.*MATCH\_ANY*, "Jhon") //name like ‘%Jhon%’  .addCondition(Student.Field.*id*, Operator.*LESS*, 100) // id < 100  .orderByDesc(Student.Field.*grade*); //设置Order By  List<Student> sts = *db*.select(s);  Assert.*assertEquals*(sts.size(), *db*.count(s.getQuery()));  } |

对应的SQL语句如下

|  |
| --- |
| select t.\* from STUDENT t where t.NAME like ? escape '/' and t.ID<? order by t.GRADE DESC |

那么，在这个案例中，我们如果增加一个条件，要求 student的grade=’3’，如何写呢？

代码清单：src/main/java/org/easyframe/tutorial/lesson2/Case1.java

|  |
| --- |
| @Test  **public** **void** testSelect\_LikeAndEtc() **throws** SQLException{  Student s=**new** Student();  s.setGrade("3"); //希望增加一个条件  s.getQuery().addCondition(Student.Field.*name*,Operator.*MATCH\_ANY*, "Jhon"); //name like ‘%Jhon%’  s.getQuery().addCondition(Student.Field.*id*,Operator.*LESS*, 100); // id < 100  s.getQuery().orderByDesc(Student.Field.*grade*); //设置Order By  List<Student> sts=*db*.select(s);    Assert.*assertEquals*(sts.size(), *db*.count(s.getQuery()));  } |

上面的代码实际执行后，会发现无法达成预期目的。grade=’3’这个不会成为条件。

从前面的“拟态“规则中我们可以推出

*当Query对象中有条件时，对象中的所有字段值将不再作为条件。*

因此，在这个例子中，要增加一个条件，您必须这样写

|  |
| --- |
| @Test  **public** **void** testSelect\_LikeAndEtc2() **throws** SQLException{  Student s=**new** Student();  //s.setGrade("3"); //在已经使用了Query对象中的情况下，此处设值不作为查询条件  //添加 grade='3'这个条件。当运算符为 = 时，中间的运算符可以省略不写。  s.getQuery().addCondition(Student.Field.*grade*,"3");    s.getQuery().addCondition(Student.Field.*name*,Operator.*MATCH\_ANY*, "Jhon");  s.getQuery().addCondition(Student.Field.*id*,Operator.*LESS*, 100);  List<Student> sts=*db*.select(s);    Assert.*assertEquals*(sts.size(), *db*.count(s.getQuery()));  } |

### 更新主键列

在其他的基于主键管理Entity的ORM中，使用API去更新对象的主键是不可能的，但是在基于SQL的EF-ORM中，这成为可能

代码清单： src/main/java/org/easyframe/tutorial/lesson2/Case1.java

|  |
| --- |
| @Test  **public** **void** testUpdatePrimaryKey() **throws** SQLException{  Student q=**new** Student();  q.setId(1);  q=*db*.load(q);    q.getQuery().addCondition(Student.Field.*id*, q.getId());  q.setId(100);    *db*.update(q); //将id（主键）修改为100  //SQL：update STUDENT set ID = 100 where ID= 1  } |

上例中展示了旧的主键值作为条件，去替换新的主键值的用法。这是基于SQL封装框架的思想体现。

### 特殊条件: AllRecordsCondition

最简单的，我们可以这样查询出所有的学生

|  |
| --- |
| //查出全部学生  List<Student> allStudents=*db*.selectAll(Student.**class**); |

那么如果我们需要学生按照学号正序排列，这个selectAll的API就无法提供了。实际上，selectAll这个方法是可有可无的，它可以用普通的select()方法来代替。按照以前掌握的技巧，您可能会这样写

|  |
| --- |
| //按学号顺序查出全部学生  Student st=**new** Student();  st.getQuery().orderByAsc(Student.Field.*id*);  List<Student> all=*db*.select(st); |

不过您会收到一个错误消息

|  |
| --- |
| Illegal usage of Query object, must including any condition in query. |

您可能会问，我就是要查全部啊，怎么还要放条件到Query里去呢？

在EF-ORM中，有一个特殊的条件，名叫 AllRecordsCondition，它是这样使用的。

代码清单：src/main/java/org/easyframe/tutorial/lesson2/Case2.java

|  |
| --- |
| @Test  **public** **void** testAllRecords() **throws** SQLException{  Student st=**new** Student();  st.getQuery().setAllRecordsCondition(); //设置为查全部学生  st.getQuery().orderByAsc(Student.Field.*id*); //按学号排序  List<Student> all=*db*.select(st);  System.*out*.println("共有学生"+all.size());  } |

这里，通过setAllRecordsCondition()。指定了查询条件。

为什么这么设计？不传入任何条件，就查出全部记录，这不是很方便吗？

这个设计的本意是为了防止用户出现误用，我们假定这样一种场景：  
用户在查询前，通过若干if分支对条件进行赋值。如果用户没有考虑到的一条分支下，没有产生任何有效条件，而此时数据库中有大约一千万条记录。

大家可以想一想，是用户该次操作失败好一些呢，还是整个服务器因为内存溢出而退出要好一些？正是因为这种场景，我们希望开发人员在使用框架时想清楚，能不能承受将全部数据加载到内存的开销。

这个API的设计有人喜欢有人不喜欢。不过您可以在jef.properties中加上一行，让无条件查询等效于查询全部记录。

|  |
| --- |
| allow.empty.query=true |

### QueryBuilder的使用

在使用EF-ORM的查询API进行数据操作时，您必然会接触到一个工具类QueryBuilder。

QueryBuilder提供了很多基础方法，用来生成Condition、Query等。

比如，前面我们都是用 一个Student对象来构造一个SQL操作的描述的。在了解了这个Student对象的本质，其实是一个SQL查询操作之后，我们可以换一个角度来看问题。

这是前面出现过的代码（使用QueryBuilder之前）

代码清单： src/main/java/org/easyframe/tutorial/lesson2/Case1.java

|  |
| --- |
| @Test  **public** **void** testSelect\_LikeAndEtc2() **throws** SQLException{  Student s=**new** Student();  s.getQuery().addCondition(Student.Field.*grade*,"3");  s.getQuery().addCondition(Student.Field.*name*,Operator.*MATCH\_ANY*, "Jhon");  s.getQuery().addCondition(Student.Field.*id*,Operator.*LESS*, 100);  s.getQuery().orderByDesc(Student.Field.*grade*);  List<Student> sts=*db*.select(s);    Assert.*assertEquals*(sts.size(), *db*.count(s.getQuery()));  } |

当改为使用QueryBuilder之后——

代码清单： src/main/java/org/easyframe/tutorial/lesson2/Case1.java

|  |
| --- |
| @Test  **public** **void** testSelect\_LikeAndEtc3() **throws** SQLException {  Query<Student> query = QueryBuilder.*create*(Student.**class**);  query.addCondition(QueryBuilder.*eq*(Student.Field.*grade*, "3"));  query.addCondition(QueryBuilder.*matchAny*(Student.Field.*name*, "Jhon"));  query.addCondition(QueryBuilder.*lt*(Student.Field.*id*, 100));    query.orderByDesc(Student.Field.*grade*);  List<Student> sts = *db*.select(query);  Assert.*assertEquals*(sts.size(), *db*.count(query));  } |

这样整个查询操作都以Query作为主体了。

之前的所有条件运算符，都有在QueryBuilder中生成条件的方法

|  |  |  |
| --- | --- | --- |
| **条件方法** | **对应Operator中的运算符** | **备注** |
| eq() | ***EQUALS*** | 等于 |
| gt() | ***GREAT*** | 大于 |
| lt() | ***LESS*** | 小于 |
| ge() | ***GREAT\_EQUALS*** | 等于 |
| le() | ***LESS\_EQUALS*** | 小于等于 |
| matchAny() | ***MATCH\_ANY*** | 匹配任意位置 |
| matchStart() | ***MATCH\_START*** | 匹配尾部 |
| matchEnd() | ***MATCH\_END*** | 匹配尾部 |
| in() | ***IN*** | 属于 |
| notin() | ***NOT\_IN*** | 不属于 |
| ne() | ***NOT\_EQUALS*** | 不等于 |
| between() | ***BETWEEN\_L\_L*** | 介于..之间 |
| isNull() | ***IS\_NULL*** | 为NULL |
| notNull() | ***IS\_NOT\_NULL*** | 不为NULL |
| like() | ***无，可用SqlExpression()组合成条件*** | LIKE |
| or() | ***无*** | 将多个条件以OR相连 |
| and() | ***无*** | 将多个条件以AND相连 |
| not() | ***无*** | 一个条件修饰为NOT |

QueryBuilder有13个字符，所以做了一个别名，我们可以将QueryBuilder简写为QB。QB和QueryBuilder使用上没有任何不同。例如，上面的示例还可以写成——

|  |
| --- |
| Query<Student> query = QB.*create*(Student.**class**);  query.addCondition(QB.*eq*(Student.Field.*grade*, "3"));  query.addCondition(QB.*matchAny*(Student.Field.*name*, "Jhon"));  query.addCondition(QB.*lt*(Student.Field.*id*, 100)); |

### Criteria API和查询语言

前面讲过，EF-ORM的API封装的其实是SQL操作。有朋友问：ORM框架顾名思义是将E-R的关系型结果转换为对象模型来操作，最好是映射成JDO一般，把大家从写SQL语句中解放出来。你怎么能把框架设计成这样？

我的回答是：关系型数据库的设计有其经典的理论支持，也有众多的约束和性能局限，这些都是必须使用者去面对去解决的。如果完全像一个对象数据库操作，那么我们项目中根本不需要使用MySQL或是Oracle。NoSQL的数据存储满大街都是，对内存的使用率更高性能也更好。只满足主键操作的情况下，ORM根本连存在的意义都没有，redis,mongodb等大堆流行技术爱谁用谁。

正是因为SQL和E-R思想的必要性，在实际项目中，使用H框架的人最后不过是从SQL中跳出，又掉进HQL里去。有什么被解放了？使用RDBMS的人，谁能不学习SQL，谁能不了解SQL?

SQL的思想（而不是语法）才是关系型数据存储的精髓，要在RDBMS上做出性能优秀的应用，不可能不掌握这种思想。正是因为这样，EF-ORM试图在保留SQL的思想、淡化SQL的语法的方向上封装JDBC。

这包括了用JavaAPI代替查询语言解决SQL编译检查和开发效率的问题；用Java反射等特性解数据库行、列到数据对象的自动转换；使使用者更简单（而符合E-R关系）的方式操作数据库。

因此，EF-ORM封装后提供给开发者的API，不是面向对象操作的API，依然是一套基于SQL思想的API，只不过其中面向转换等操作被自动化了。用通俗一点的说法，EF是披着某H框架的外皮，但骨子里是IBatis的思想。

上面我们讲这套用于操作数据库的API，其实就是一直在说的Criteria API（非JPA标准的）。其实严格来说EF-ORM中只有两种数据库操作方式，即Criteria API和NativeQuery。之前所有模仿H框架的单记录操作都是用Criteria API实现的，正如前文说的——“拟态”。

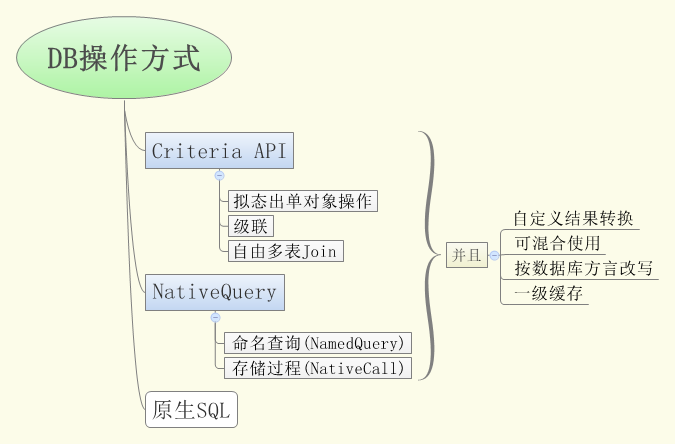


图3-2 EF-ORM的操作方式

另外的一个操作体系是NativeQuery，这将在第7章讲到。NativeQuery其实就是对SQL的直接封装和改良。用户直接提供SQL语句，EF-ORM负责对其进行管理、解析、动态改写、参数化等工作。

原生SQL则是可以让用户直接将SQL语句提交给数据库，EF-ORM不作任何处理，只不过允许其和其他操作处于同一个事务当中。这算不上是一种操作方式。

以JPA为代表的ORM应用， 最后都增加了Criteria API和xxQL的功能，来支持直接操作数据库表，这些都是与时俱进下不得不做出的改变，只是Criteria API来的有点晚，并且给人太多选择，实际上很多人已经走了弯路。而且个人觉得，JPA Criteria API设计得有点不接地气，虽然类型校验能力是很强，但是对开发者过于繁琐。比如——查询一张表里所有记录的条数：

**SQL**

|  |
| --- |
| *select count(\*) from product* |

评价：简单直接。

**JPA (Criteria API)**

|  |
| --- |
| EntityManagerFactory emf = get EntityManagerFactory();  //使用Spring得到当前线程所在事务的EntityManager.  EntityManager em= EntityManagerFactoryUtils.getTransactionalEntityManager(emf);  //构造查询  CriteriaBuilder qb=emf.getCriteriaBuilder();  CriteriaQuery<Long> q=qb.createQuery(Long.**class**);  Root<Product> root=q.from(Product.**class**);  q.select(qb.count(root));  //查询  **return** em.createQuery(q).getSingleResult().intValue(); |

评价：充满了专家组的风格，极其严谨的API，每个对象每个概念都一清二楚，显式声明数据类型和可校验。但是世界上大部分人都是懒人，不是专家。这样的API是不足以诱使人们放弃xxQL的。

某H框架的写法(Criteria API)就好了很多

|  |
| --- |
| Criteria crit = getSession().createCriteria(Users.class);  crit.setProjection(Projections.rowCount());  List list= crit.list();  Long count = (Long)list.get(0);  return count.intValue(); |

过渡到H框架的Criteria API也不错，不过对某H框架不熟的的同学请先学习什么叫“投影统计“。

EF-ORM写法则更为简单：

|  |
| --- |
| session.count(QueryBuilder.create(Users.class)); |

没有完全按照JPA来设计API的重要原因之一就是，大部分开发人员对J2EE规范都是选择性遵循的，EJB的普及比起Spring来在国内已经落后得太多。更接地气的东西才更有存在的意义。

# Query API单表操作

前面的许多查询和更新示例，其实就是在使用Criteria API。这套API涵盖了实体操作。囊括了单表的各种SQL操作和多表下的操作。我们先了解还有哪些单表下的操作用法。

注意：下面的例子都是在某些场景下，需要达到某个目的的API用法。

但这些用法不是一成不变是，是可以随意组合的。此外，列出的也不是实现目的的唯一方法。

请在理解其含义后灵活使用。

## 查询

### 查询条件的用法

我们先从一组示例来理解和使用Creteria API的单表查询操作。

#### And Or Not

在SQL条件中，我们经常会碰到这三种组合的场景。下面这个案例中组合使用了And Or Not三种条件连接。可以发现，Condition作为一个条件描述，可以互相嵌套，从而描述出And Or Not之间的顺序关系。

代码示例: src/main/java/org/easyframe/tutorial/lesson3/Case1.java

|  |
| --- |
| @Test  **public** **void** testSelect\_AndOrNot() **throws** SQLException {  Student st = **new** Student();    //三个Like条件，或  Condition isLastNam\_ZhaoQianSun =QueryBuilder.*or*(  QueryBuilder.*matchStart*(Student.Field.*name*, "赵"),  QueryBuilder.*matchStart*(Student.Field.*name*, "钱"),  QueryBuilder.*matchStart*(Student.Field.*name*, "孙")  );    //或条件前面加上 NOT。  Condition isNot\_ZhaoQianSun = QueryBuilder.*not*(isLastNam\_ZhaoQianSun);    //最终条件: 不姓'赵钱孙' 的女生。  st.getQuery().addCondition(QueryBuilder.*and*(  isNot\_ZhaoQianSun,  QueryBuilder.*eq*(Student.Field.*gender*, "F")  )  );    List<Student> students=db.select(st);  System.*out*.println("不姓'赵钱孙'三姓的女生:" + students.size());    *assertEquals*(db.count(st.getQuery()), students.size());  } |

对应SQL语句为：

|  |
| --- |
| select count(\*) from STUDENT t  where (not (t.NAME like ? escape '/' or t.NAME like ? escape '/' or t.NAME like ? escape '/' )  and t.GENDER=?) |

这里要指出的是，当使用Like语句时，为了支持传入的参数中存在 % \_等SQL字符的场景，EF-ORM会自动进行转义。同时也能防止注入攻击。

上面的写法还是稍显累赘，前面已经说过，QueryBuilder这个类可以省略写作QB。

此外上例中的最外层的And嵌套可以省略。因为Query对象中可以填入多个Condition，这些Condition互相之间就是And关系。省略最外层的嵌套后，写法为——

|  |
| --- |
| @Test  **public** **void** testSelect\_AndOrNot() **throws** SQLException {  Student st = **new** Student();  Condition isLastNam\_ZhaoQianSun =QB.*or*(  QB.*matchStart*(Student.Field.*name*, "赵"),  QB.*matchStart*(Student.Field.*name*, "钱"),  QB.*matchStart*(Student.Field.*name*, "孙")  );  st.getQuery().addCondition(QB.*not*(isLastNam\_ZhaoQianSun));  st.getQuery().addCondition(QB.*eq*(Student.Field.*gender*, "F"));  List<Student> students=db.select(st); |

#### 使用函数或表达式作为条件

使用函数运算作为查询条件，对数据库字段计算函数

下面例子中，

|  |
| --- |
| @Test  **public** **void** testSelect\_Function() **throws** SQLException {  Student st = **new** Student();  st.getQuery().addCondition(**new** FBIField("concat(lower(gender) , grade)"),"f2");  List<Student> students=db.select(st);    *assertEquals*(db.count(st.getQuery()), students.size());  } |

用FBIField可以对数据库中的字段进行函数运算。上例的条件实际执行SQL如下：

|  |
| --- |
| select count(\*) from STUDENT t where lower(gender)||grade= ‘f2’ |

FBIField这个词可能会引起吐槽。其实这个词和美国的FBI没有什么关系。由来是Oracle的Function based index。即函数列索引。许多数据库都不支持将用函数值来创建索引，而Oracle支持并且将这类索引命名为FBI。

由于在where条件中使用函数会造成数据库索引无效，是无法在数据量较大的表中使用的。即便是Oracle上也需要专门为此创建一个函数索引。这里用了FBIField的名称即来源于此。上例的这种用法仅限于数据量小的表使用。

仔细的同学还会发现，concat函数被转化为了lower(gender)||grade 这样的表达式。这其实是EF-ORM后文要提到的一个特点——方言转换功能，EF-ORM会将用户填入的JPQL或SQL表达式用当前数据库方言重写，以适应不同数据库的语法差异。这一功能主要针对NativeQuery查询，不过在这里也能生效。

在上面的例子中还要注意，FBIField中的表达式是用java模型的字段名来描述的。这里的gender、grade都是java字段名，不是数据库列名。 EF-ORM在实际查询前会使用真正的数据库列名替换这些Java属性名。这个替换的规则需要FBIFIeld对象绑定到一个Entity上。在单表查询中这不成问题，框架能自动完成绑定，但是如果在多表查询中，可能就无法准确的绑定到特定的对象上，这时候就需要用到FBIField的另两个构造函数。

|  |
| --- |
| *FBIField(String, Query<?>)*  或者  *FBIField(String, IQueryableEntity)* |

这两个种构造函数可以让FBIField显式的绑定到一个查询Entity上。确保各个java属性能被正确的转换为SQL语句中的列名。

#### 使用JPQLExpression

在Condition的Value部分，也允许自行编写表达式。EF-ORM提供了JpqlExpression和SqlExpression的类。

JpqlExpression和SqlExpression都是指表达式。但两者用法上还是有所不同的。这两个表达式的作用都是将制定的文本包装为数据库的原生SQL片段，将这部分SQL嵌入到我们的Criteria API查询中去，可以更为灵活和方便。

下面是JPQLExpression的两个例子

src/main/java/org/easyframe/tutorial/lesson3/Case1.java

|  |
| --- |
| @Test  **public** **void** testSelect\_JpqlExpression() **throws** SQLException {  Student st = **new** Student();  {  //案例一  st.getQuery().addCondition(Student.Field.*gender*, **new** JpqlExpression("upper(nvl('f','m'))"));  List<Student> students=db.select(st);  *assertEquals*(db.count(st.getQuery()), students.size());  }  st.getQuery().clearQuery(); //清除上一个查询条件  {  //案例二: 查出出生日期最晚的学生  st.getQuery().addCondition(Student.Field.*dateOfBirth*, **new** JpqlExpression("(select max(dateOfBirth) from student)",st.getQuery()));  List<Student> students=db.select(st);  *assertEquals*(db.count(st.getQuery()), students.size());  }  } |

上面的代码演示了两个例子，第一个例子很简单，使用数据库的upper和nvl函数。

|  |
| --- |
| select count(\*) from STUDENT t where t.GENDER=upper(coalesce('f','m')) |

Nvl函数是Oracle专用的，Derby不支持，这里EF-ORM将其自动改写为coalesce函数。

第二个例子复杂一些了

|  |
| --- |
| select t.\* from STUDENT t where t.DATE\_OF\_BIRTH=(select max(t.DATE\_OF\_BIRTH) from student) |

这严格意义上是一个多表查询。不过我们也能看出其特点

* 整个表达式被嵌入到where条件中的值的位置。
* 表达式中的java属性名’dateOfBirth’，被替换为了数据库列名 ‘DATE\_OF\_BIRTH’。

其实就这是JPQLExpression的特点，和前面介绍的一样，EF-ORM会对JPQLExpression表达式进行解析和改写处理。这包括数据库方言适配和字段名称别名匹配。

从上面例子可以看出，表达式是一个强力的工具，灵活使用表达式能让EF-ORM简单的API发挥出预想意外的强大功能。

#### 使用SqlExpression

上一节的案例二，也可以用SqlExpression来编写，写法上稍有不同

|  |
| --- |
| @Test  **public** **void** testSelect\_SqlExpression() **throws** SQLException {  Student st = **new** Student();  //案例: 查出出生日期最晚的学生  st.getQuery().addCondition(Student.Field.*dateOfBirth*, **new** SqlExpression("(select max(date\_of\_birth) from student)"));  List<Student> students=db.select(st);  *assertEquals*(db.count(st.getQuery()), students.size());  } |

生成的SQL语句和上例是一样的。

SqlExpression相比JpqlExpression，是更接近SQL底层的表达式。EF-ORM不会对SqlExpression进行改写。也就是说，SqlExpression对象中的字符被直接输出到SQL语句当中，为此你需要保证其中的内容符合数据库的语法，并且用作参数的是数据库列名，而不是对象的属性名称。这里的 列名date\_of\_birth 必须直接写成数据库里的字段名。如果写成 dataOfBirth那么肯定是要出错的。

SqlExpression可以提供更为灵活的功能。比如SqlExpression可以直接作为Condition使用。

|  |
| --- |
| @Test  **public** **void** testSelect\_SqlExpression2() **throws** SQLException {  Student st = **new** Student();  st.getQuery().addCondition(  **new** SqlExpression("{fn timestampdiff(SQL\_TSI\_DAY,date\_of\_birth,current\_timestamp)} > 100")    );  List<Student> students=db.select(st);    *assertEquals*(db.count(st.getQuery()), students.size());  } |

当然，使用SqlExpression要兼容各种数据库，对开发者的SQL功底有一定的要求。

### 使用Selects对象

#### 定义查询的列

经常有DBA告诫开发人员说，在Select语句中，不要总是查t.\*，而是需要那几个列就查哪几个列。

因此，我们在使用查询时，也可以指定要查的列。

src/main/java/org/easyframe/tutorial/lesson3/Case2.java

|  |
| --- |
| @Test  **public** **void** testSelect\_selectFrom() **throws** SQLException{  Query<Student> query=QueryBuilder.*create*(Student.**class**);  query.addCondition(QueryBuilder.*eq*(Student.Field.*gender*, "F"));    Selects selects=QueryBuilder.*selectFrom*(query);  selects.column(Student.Field.*id*);  selects.column(Student.Field.*name*);    List<Student> students=db.select(query); //查询所有女生的学号和名字  } |

当查询指定了若干列时，返回的对象中未指定选出的列都是null。只有选定的几个列的属性是有值的，因此查出的是一组“不完整”对象。

#### 使用Distinct

刚才说到，QueryBuilder可以从一个查询请求中提取出 Selects对象，Selects对象是一个操作SQL语句select部分的工具。它对应到SQL语句的select部分，是select部分的封装。

使用selects工具，我们可以完成各种诸如列选择、distinct、group by、having、为列指定别名等各种操作。例如

src/main/java/org/easyframe/tutorial/lesson3/Case2.java

|  |
| --- |
| @Test  **public** **void** testSelect\_selectDistinct() **throws** SQLException{  Query<Student> query=QueryBuilder.*create*(Student.**class**);  Selects selects=QueryBuilder.*selectFrom*(query);  selects.setDistinct(**true**);  selects.column(Student.Field.*name*);    //相当于 select distinct t.NAME from STUDENT t  List<Student> students=db.select(query);  } |

上例中，通过在selects中设置distinct=true，实现distinct的查询。

#### 使用Group和max/min

src/main/java/org/easyframe/tutorial/lesson3/Case2.java

|  |
| --- |
| @Test  **public** **void** testSelect\_selectGroup() **throws** SQLException{  Query<Student> query=QueryBuilder.*create*(Student.**class**);  Selects selects=QueryBuilder.*selectFrom*(query);    selects.column(Student.Field.*gender*).group(); //按性别分组  selects.column(Student.Field.*id*).count(); //统计人数  selects.column(Student.Field.*id*).max(); //最大的学号  selects.column(Student.Field.*id*).min(); //最小的学号    //上述查询的结果。无法再转换为Student对象返回了，这里将各个列按顺序形成数组返回。  List<String[]> stat=db.selectAs(query,String[].**class**);  //相当于执行查询select t.GENDER, count(t.ID), max(t.ID), min(t.ID) from STUDENT t group by t.GENDER  **for**(Object[] result : stat){  System.*out*.print("M".equals(result[0])?"男生":"女生");  System.*out*.print(" 总数:"+result[1]+" 最大学号:" + result[2] + " 最小学号"+ result[3]);  System.*out*.println();  }  } |

上例是Selects对象的进一步用法。通过Selects对象，可以指定SQL语句的group部分，同时可以执行count、sum、avg、max、min等通用的统计函数。

上例中还要注意一个问题，即查询对象的返回问题。在这例中，查询对象无法转换为Student对象了，因此我们需要设置一个能存放这些字段的数据类型作为返回记录的容器。最基本的容器当然就是Map和数组了。此处我们用一个String[]作为返回值的容器。

关于如何传入合适的返回结果容器，以及结果转换过程的干预等等，可以参见第8章，高级查询特性。

有朋友问，Oracle分析函数 partition by能不能用Query对象写出来。这个 是不支持的，因为这样的SQL语句只能在Oracle上使用，且在别的数据库上难以写出替代语句，因此此类语句建议您使用NativeQuery 来写。同时NativeQuery也无法自动将其改写为其他数据库兼容的语句，因此多数据库SQL方言也需要人工编写。

#### 使用Having

上面的例子稍微改写一下，可以产生Having的语句

src/main/java/org/easyframe/tutorial/lesson3/Case2.java

|  |
| --- |
| @Test  **public** **void** testSelect\_groupHaving() **throws** SQLException{  Query<Student> query=QueryBuilder.*create*(Student.**class**);  Selects selects=QueryBuilder.*selectFrom*(query);    selects.column(Student.Field.*grade*).group(); //按年级分组  //查出人数，同时添加一个Having条件，count(id)>2  selects.column(Student.Field.*grade*).count().having(Operator.*GREAT*, 2);    List<String[]> stat=db.selectAs(query,String[].**class**);  **for**(Object[] result : stat){  System.*out*.print("年级:"+result[0]+" 人数:"+result[1]);  System.*out*.println();  }  } |

上面的查询产生的SQL语句如下

|  |
| --- |
| select t.GRADE,count(t.GRADE) from STUDENT t group by t.GRADE having count(t.GRADE)>2 |

在count()后面，还可以用havingOnly。其效果是 count(id)将不作为select的一项出现，仅作为having子句的内容。

#### count的用法

在Session类当中，还有一个常用的方法 count();

count方法和select方法经常被拿在一起比较，当传入相同的查询请求时，select方法会查出全部数据，而count方法会将请求改写，用数据库的count函数去计算这个查询将返回多少条结果。

比如下面这个例子——

src/main/java/org/easyframe/tutorial/lesson3/Case2.java

|  |
| --- |
| @Test  **public** **void** testSelect\_countDistinct() **throws** SQLException {  Query<Student> q = QB.*create*(Student.**class**);  Selects items = QB.*selectFrom*(q);  items.column(Student.Field.*name*);  items.setDistinct(**true**);  q.setMaxResult(1);  **long** total = db.count(q);// select count(distinct t.NAME) from STUDENT t  List<String> result=db.selectAs(q,String.**class**); // select distinct t.NAME from STUDENT t  System.*out*.println("总数为:"+ total +" 查出"+ result.size()+"条");  } |

这个案例使用了同一个Query请求去执行count方法和select方法，前者使用了count distinct函数，查询不重复的名称数量，返回15。后者去查询，但只返回了一条结果，因为在请求中要求最多返回一条记录。

从上面的例子来看，count方法总是通过改写一个“查询数据内容”的请求来得到数量。

再来看下面这个有点相似的例子

src/main/java/org/easyframe/tutorial/lesson3/Case2.java

|  |
| --- |
| @Test  **public** **void** testSelect\_countDistinct2() **throws** SQLException {  Query<Student> q = QB.*create*(Student.**class**);  Selects items = QB.*selectFrom*(q);  items.column(Student.Field.*name*).countDistinct();  Integer total=db.loadAs(q,Integer.**class**); // select count(distinct t.NAME) from STUDENT t  System.*out*.println("Count:"+ total);  } |

就count而言，两种方法产生的SQL语句是完全一样的，但两种查询的含义是不同的。第二个例子中，count语句是由使用者自行指定，作为一个返回的列上的函数操作出现的。这种方式是可以和group混用的。相当于使用者自定义了一条查count的SQL语句，同时这个Query对象也只能返回数值类型的结果。

因此上面两端代码，虽然都实现了count distinct功能，但是使用的机制是不一样的，请仔细体会。

API说明：将传入的普通查询请求转换为count语句的查询方法

|  |  |
| --- | --- |
| 方法 | 作用 |
| jef.database.Session.count(ConditionQuery) | 将传入的查询请求（单表/多表）改写为count语句，然后求满足查询条件的记录总数。 |
| jef.database.Session.count(IQueryableEntity, String) | 根据传入的查询请求（单表），求符合条件记录总数。第二个参数可以强制指定表名。一般传入null即可。 |

#### 使用数据库函数

刚才我们已经在group中了解了常见统计函数的用法。我们在另一个案例中稍稍回顾一下

src/main/java/org/easyframe/tutorial/lesson3/Case2.java

|  |
| --- |
| @Test  **public** **void** testSelect\_function1() **throws** SQLException {  Query<Student> q = QB.*create*(Student.**class**);  Selects items = QB.*selectFrom*(q);  items.column(Student.Field.*id*).min().as("min\_id");  items.column(Student.Field.*id*).max().as("max\_id");  items.column(Student.Field.*id*).sum().as("sub\_id");  items.column(Student.Field.*id*).avg().as("avg\_id");    **for**(Map<String,Object> result:db.selectAs(q,Map.**class**)){  System.*out*.println(result);  }  }  //select min(t.ID) AS MIN\_ID, max(t.ID) AS MAX\_ID, sum(t.ID) AS SUB\_ID, avg(t.ID) AS AVG\_ID from STUDENT t |

上例中对统计列添加了别名，同时返回结果改用Map封装。

那么是不是就只能用几个基本的统计函数了呢？再看下面的例子

src/main/java/org/easyframe/tutorial/lesson3/Case2.java

|  |
| --- |
| @Test  **public** **void** testSelect\_function2() **throws** SQLException {  Query<Student> q = QB.*create*(Student.**class**);  Selects items = QB.*selectFrom*(q);  //对姓名统一转大写  items.column(Student.Field.*name*).func(Func.*upper*);  //性别进行函数转换，decode是Oracle下的函数，注意观察其在Derby下的处理。有兴趣的可以换成MySQL试一下。  items.column(Student.Field.*gender*).func(Func.*decode*, "?", "'M'" ,"'男'","'F'" ,"'女'");  //先对日期转文本，然后截取前面的部分  items.column(Student.Field.*dateOfBirth*).func(Func.*str*).func(Func.*substring*,"?","1","10");    **for**(String[] result:db.selectAs(q,String[].**class**)){  System.*out*.println(Arrays.*toString*(result));  }  } |

在上面这个例子中，使用了三个不同的函数，对结果进行处理。

此外，如果不是EF-ORM内部的标注函数，您可以可以直接输入函数名，比如在mySQL上，您可以这样用

|  |
| --- |
| items.column(Student.Field.*dateOfBirth*).func("date\_format","%Y-%M-%D"); |

上面的方法都是使用API创建的函数，还有一种用法，直接传入SQL表达式，这也是可以的。

看下面的例子。

#### 在查询项中使用SQL表达式

整个查询部分使用SQL表达式。

src/main/java/org/easyframe/tutorial/lesson3/Case2.java

|  |
| --- |
| @Test  **public** **void** testSelect\_sqlExpression() **throws** SQLException {  Query<Student> q = QB.*create*(Student.**class**);    Selects select = QB.*selectFrom*(q);  select.columns("name,decode(gender,'F','女','M','男') as gender");;  **for**(Student result:db.select(q)){  System.*out*.println(result.getName()+" "+result.getGender());  }  //实际执行SQL:  // select t.NAME,  CASE WHEN GENDER = 'F' THEN '女' WHEN GENDER = 'M' THEN '男' END AS GENDER  from STUDENT t  } |

上面是最接近SQL的写法。相当于整个select部分都直接用SQL片段表达了。

还可以这样写——单个字段的函数使用表达式。

src/main/java/org/easyframe/tutorial/lesson3/Case2.java

|  |
| --- |
| @Test  **public** **void** testSelect\_sqlExpression2() **throws** SQLException {  Query<Student> q = QB.*create*(Student.**class**);    Selects select = QB.*selectFrom*(q);  select.column(Student.Field.*name*);  select.sqlExpression("str(add\_months(Date\_Of\_Birth,24))").as("BIRTH\_ADD\_24");  **for**(String[] result:db.selectAs(q,String[].**class**)){  System.*out*.println(Arrays.*toString*(result));  }  } |

单个字段使用表达式，为每个学生的出生日期增加24个月，SQL表达式后面也可以添加列别名等修饰。

上面两个例子中传入的都是SQL表达式。因此注意列按数据库的名称。

上面两例都对传入的SQL表达式进行了解析和改写。某些时候如不希望EF-ORM进行改写，可以使用rawExpression()方法

|  |
| --- |
| //select.sqlExpression("str(add\_months(Date\_Of\_Birth,24))");  select.rawExpression("str(add\_months(Date\_Of\_Birth,24))"); //不带重写功能的表达式 |

### 分页

一般来说，分页查询包含两方面的内容，一是获取总数，二是查询限定范围的结果数据。为了准确的限定结果范围，排序条件必不可少。

EF-ORM对于分页的两步操作，总体上来将遵循以下建议。

|  |
| --- |
| *用户传入的Query对象或者是SQL语句中不要带有 count / limit/ offset操作。可以传入不分页的普通的查询，由EF-ORM进行转化，根据需要自动转变为count语句、限定行范围的查询语句。*  *这样做的原因*   1. *希望应用开发者专注于业务本身，而不是分页细节。* 2. *EF-ORM生成的分页语句能对开发人员屏蔽不同的数据库语法差异。* |

关于总数的获取，我们可使用前文介绍的几种count方法（参见 4.1.2.5）。

关于结果范围的限定，我们可以使用EF-ORM提供的几个API （参见4.1.3.1）。

EF-ORM还提供了将上述分页行为封装在一起的操作对象，可以——

* 直接获取总数、总页数等分页信息。
* 可从头到尾进行顺序翻页或跳转，获取当页数据。

该对象名为PagingIterator的类。(参见4.1.3.2)。

#### 限定结果范围

Session类中有以下几个方法，可以传入类型为IntRange的参数，这里的IntRange就可以用来限定结果范围。

|  |  |
| --- | --- |
| **方法** | **用途说明** |
| **Session.select(T, IntRange)** | 传入Entity形态的查询(单表/级联)，限定返回条数在IntRange区间范围内。 |
| **Session.select(ConditionQuery, IntRange)** | 传入Query形态的查询(单表、Union、Join均可)，限定返回条数在IntRange区间范围内。 |
| **Session.select(ConditionQuery, Class<T>, IntRange)** | 传入Query形态的查询(单表、Union、Join均可)，结果转换为指定类型，限定返回条数在IntRange区间范围内。 |
| **Session.selectForUpdate(Query<T>, IntRange)** | 传入Query形态的单表查询，可在结果集上直接更新记录。 |
| **Session.iteratedSelect(T, IntRange)** | 传入Entity形态的查询(单表/级联)，限定返回条数在IntRange区间范围内。将游标封装为返回结果遍历器。 |
| **Session.iteratedSelect(TypedQuery<T>, IntRange)** | 传入Query形态的查询(单表/级联/Union)，限定返回条数在IntRange区间范围内。将游标封装为返回结果遍历器。 |
| **Session.iteratedSelect(ConditionQuery, IntRange)** | 传入Query形态的查询(单表、Union、Join均可)，限定返回条数在IntRange区间范围内。将游标封装为返回结果遍历器。 |
| **Session.iteratedSelect(ConditionQuery, Class<T>, IntRange)** | 传入Query形态的查询(单表、Union、Join均可)，限定返回条数在IntRange区间范围内。将游标封装为返回结果遍历器。 |

可以发现，EF-ORM查询接口高度集中。主要分为几个系列： select系列是查询出List结果。iteratedSelect是查询出游标待遍历。还有一个load系列的方法是查出单条记录的。

IntRange表示的是一个含头含尾的区间(闭区间)，和Java Collection中常见的前闭后开区间有所不同。比如表示第 1到10条记录。不是用new IntRange(0, 10)，而是用new IntRange(1, 10)。来表示，更为接近我们日常的口头语法。其实Java用习惯的人会更偏好前闭后开区间，以后可能会再考虑向下兼容的前提下支持。

示例，查询返回11~20条记录

src/main/java/org/easyframe/tutorial/lesson3/Case3.java

|  |
| --- |
| @Test  **public** **void** test\_IntRange() **throws** SQLException{  Query<Student> q = QB.*create*(Student.**class**);  List<Student> results=db.select(q,**new** IntRange(11, 20));//查询，返回第11到20条  **int** count=db.count(q);  *assertEquals*(count-10, results.size());  } |

使用上述方法后，在不同的数据库下，框架会生成不同的分页语句。实现数据库分页。

#### 使用PagingIterator

Session的API方法中，除了select系列、iteratedSelect系列、load系列外，还有一套pageSelect系列的方法。

|  |  |
| --- | --- |
| **方法** | **作用** |
| **Session.pageSelect(T, int)** | 传入Entity形态的查询(单表/级联) 和 分页大小 |
| **Session.pageSelect(ConditionQuery, int)** | 传入Query形态的查询(单表、Union、Join均可) |
| **Session.pageSelect(ConditionQuery, Class<T>, int)** | 传入Query形态的查询(单表、Union、Join均可)， 并指定返回结果类型 |
| **Session.pageSelect(String, Class<T>, int)** | 传入NativeQuery形态的查询，并指定返回结果类型。 |
| **Session.pageSelect(String, ITableMetadata, int)** | 传入NativeQuery形态的查询，并指定返回结果类型元数据（一般用来描述动态表的模型）。 |
| **Session.pageSelect(NativeQuery<T>, int)** | 传入NativeQuery形态的查询，查询结果类型已经在NativeQuery中指定。一般为传入命名查询（NamedQuery）. |

上面的后三个方法涉及了NativeQuery和NamedQuery，可参见第7章。

另外，还有pageSelectBySQL的方法，但该方法支持的是原生SQL，即EF-ORM不作任何解析和改写处理的SQL。此处先不介绍。

pageSelect系列API的特点是——

* 都返回PagingIterator对象，其中封装了分页逻辑
* 最后一个int参数表示分页的大小
* 都只需要传入查询数据的请求，框架会自动改写出count语句来查总数，当然传入 SQL或Criteria Query的对象的改写实现是不同的，不过这被封装在框架内部。

当得到PagingIterator后，我们可以通过以下的API来实现分页操作

|  |  |
| --- | --- |
| **方法** | **作用** |
| **jef.database.PagingIterator.getTotalAsInt()** | 获得总记录数（int） |
| **jef.database.PagingIterator.getTotal()** | 获得总记录数 (long) |
| **jef.database.PagingIterator.getTotalPage()** | 获得总页数。无记录时为0。例如每页10条，总记录数11时算2页。 |
| **jef.database.PagingIterator.setOffset(int)** | 设置跳过的记录数。例如如果从第1条记录开始返回，传入0；要从第11条记录开始返回，传入10即可。 |
| **jef.database.PagingIterator.getRowsPerPage()** | 得到每页的记录数 |
| **jef.database.PagingIterator.getPageData()** | 得到Page对象，其中存放了当前页中的数据和总记录数。 |
| **jef.database.PagingIterator.recalculatePages()** | 强制重新计算页数。正常情况下，只会查询一次总数。基于数据库的不可重复读特性（在处理过程中有新的数据插入或被删除），如需刷新总数需要调用此方法。 |
| **jef.database.PagingIterator.hasNext()** | 当前页后面是否还有数据。（实现Iterator接口） |
| **jef.database.PagingIterator.next()** | 返回当前页数据，同时页码向后翻一页。（实现Iterator接口） |

我们用举例来看分页查询的用法

src/main/java/org/easyframe/tutorial/lesson3/Case3.java

|  |
| --- |
| @Test  **public** **void** test\_PageSelect() **throws** SQLException{  Student st=**new** Student();  st.getQuery().setAllRecordsCondition();  st.getQuery().orderByAsc(Student.Field.*id*);  **int** start=20;  **int** limit=10;    //10是每页大小，20是记录偏移。记录偏移从0开始。下面的语句相当于查询21~30条记录  Page<Student> pagedata= db.pageSelect(st, limit).setOffset(start).getPageData();  } |

注意PagingIterator对象是一个重量级的懒加载对象，其数据只有在被请求时才会去数据库查询。因此不适合作为DTO传递数据。要获得其实际信息，可用jef.database.PagingIterator.getPageData()方法。得到的Page对象较为适合作为DTO传输。

上面的例子中，我们直接从pageSelect返回的PagingIterator对象中得到了Page这个对象。

Page是一个适合于传输的POJO对象，其中只有两个属性

* 总数
* 当页的记录内容

事实上，分页经常用在Web界面显示上，需要从数据库获得的也就是这两个信息。

使用PagingIterator一个特点是，其对于范围的限定和目前大多数Web前端框架一致，都是从0开始的。

PagingIterator对象要重量得多，其中封装了很多分页逻辑等。但最后发现Web服务一般是无状态服务，不可能持有PagingIterator很久 ，所以从实际业务看PagingIterator对象使用频率不高。需要了解PagingIterator还提供了哪些功能的，可以自行阅读API-DOC。

### 小结

总而言之，Criteria API各种组合下，用法十分灵活，在上面这些案例中，请体会每个方法的用法。更多的API，请参阅API-DOC

单表的Criteria API，能将一般项目中90%以上的常见SQL语句都表达出来。在多表情况下大约75%左右的SQL语句也都能表达出来。更复杂的SQL语句可能就要使用EF-ORM中的另一操作体系，NativeQuery了。

要设计一套易于使用，并且含义明确的查询API是相当困难的。JPA 标准Criteria API集合了众多专家的智慧才能这样严谨。EF-ORM中的这套API比较随意，很大程度上来自于众多用户的意见和想法而设计。所以——

如果有什么用法，你不确定是否可以支持，可联系作者或尝试阅读源码。

如果有什么用法，你决定应该支持但却没能很支持，请联系作者。

## 更新

### 基本操作

前面我们已经讲过，update请求用到了Entity-Query这对对象。Entity中的值描述更新后的值，Query描述更新的Where条件。

在3.2.2中，我们甚至更新了对象的主键列。那么我们先回顾一下基本的更新操作用法。

src/main/java/org/easyframe/tutorial/lesson3/CaseUpdate.java

|  |
| --- |
| **public** **void** testUpdate\_Basic() **throws** SQLException{  {  //查出一条对象,更新该条记录.  Student st= db.load(Student.**class**, 1);  st.setGender("M");  **int** updated=db.update(st);  //SQL: update STUDENT set GENDER = 'M' where ID=1  }  {  //按主键随意更新一条记录  Student st=**new** Student();  st.setId(3);  st.setGrade("M");  **int** updated=db.update(st);  //SQL: update STUDENT set GRADE = 'M' where ID=3  }  {  //按条件更新多条记录  Student st=**new** Student();  st.getQuery().addCondition(Student.Field.*name*,Operator.*MATCH\_ANY*,"张");  st.setGender("M");  **int** updated=db.update(st);  //SQL: update STUDENT set GENDER = 'M' where NAME like '%张%' escape '/'  }  } |

上面列出了三种更新的场景。第一种，先从数据库读出再更新。第二种，指定主键后直接更新指定的记录。第三种，按条件更新记录。

这里比较有用的是每次update方法都返回一个数值，表示update操作影响到的记录数。如果为0，那么就没有记录被更新。

### 更新操作Query的构成

EF的更新操作有点“神奇”的地方是，它总是知道哪些字段被设过值、哪些字段没有设值。EF是如何做到这一点的呢，我们来看这段代码。

src/main/java/org/easyframe/tutorial/lesson3/CaseUpdate.java

|  |
| --- |
| @Test  **public** **void** testUpdate\_QueryAndUpdateMap() **throws** SQLException{  Student st=**new** Student();  st.setId(1);  st.setGender("M");    Map<Field,Object> updateMap=st.getUpdateValueMap();  System.*out*.println(updateMap);  //updateMap.clear();  **int** updated=db.update(st);  } |

运行代码后，输出

|  |
| --- |
| {gender=M, id=1} |

这就是实现的机制。

每个实体都隐式扩充一个UpdateValueMap,用来存放需要更新的字段。每当我们调用方法startUpdate后，我们对实体的每个set操作都会被记录下来，放入到updateMap中去，当我们调用DbClient.update(T)方法时，只有这些修改过的字段才会被更新到数据库中去。

当我们调用set方法， updateMap中记录下当前设置的值。由此也可以判断出哪些字段被赋过值。

EF在处理时，如果某个字段（主键）被挪作Where条件使用，那么它就被从updateMap中去除。因此你就不会看到这种SQL语句了——

|  |
| --- |
| update student set *id=1*, gender=’M’ where *id=1* |

我们来试一下，如果将这个Map清除会怎么样，我们将上面代码中注释掉的 updateMap.clear()语句重新恢复。然后运行，发现日志中输出下面的文字，然后什么也没发生。

|  |
| --- |
| Student Update canceled... |

这就说明，由于updateMap中没有值，因此这就成了一个无效的update请求，框架自动忽略了这个请求。所以事实上，框架所做的——

不是在将数据库中的值更新为Entity中的值，而是将其更新为updateMap中的记录的值。

利用这一点，我们可以写出更多灵活的更新语句来。

### 更多的更新用法

#### 并发环境下原子操作的更新

这是一种基于数据库频繁操作的业务更新用法。

业务是这样的，有一张用户账户余额表，用户可以消费从余额中扣款，也可以向其中充值。这些操作可能同时发生。

我们在充值时，可以这样做。

src/main/java/org/easyframe/tutorial/lesson3/CaseUpdate.java

|  |
| --- |
| @Test  **public** **void** testUpdate\_concurrent\_error() **throws** SQLException { // 准备  UserBalance u = **new** UserBalance();  u.setAmout(100); // 一开始用户账上有100元钱  db.insert(u);    //开始  UserBalance ub=db.load(UserBalance.**class**,1);//第一步,先查出用户账上有多少钱  //... do some thing.  ub.setAmout(ub.getAmout()-50);//扣款50元  db.update(ub);//将数值更新为扣款后的数值。  } |

这看上去没什么问题。但我们想一想，如果在load出数据，到update开始之间，用户同时充值了100元并提交了。这会发生什么事？很显然，用户会杯具地发现，他充值的100元消失了。因为我们在load记录时没有锁表，update的时候数据被更新为充值前再扣款50元。基于性能考虑，即便在事务中，一般也不会在一个普通的select语句中锁表。

何况锁表并不是解决问题的最好方法，在业务繁忙的系统中，我们应当尽可能提高系统的并行度，让充值和扣款能同时发生显然并不是一个坏主意。

src/main/java/org/easyframe/tutorial/lesson3/CaseUpdate.java

|  |
| --- |
| @Test  **public** **void** testUpdate\_Cas1() **throws** SQLException {  // 准备  UserBalance u = **new** UserBalance();  u.setAmout(100); // 一开始用户账上有100元钱  db.insert(u);    //开始  **int** updated;  **do**{  UserBalance ub=db.load(UserBalance.**class**,1);//第一步,先查出用户账上有多少钱  ub.getQuery().addCondition(UserBalance.Field.*id*,ub.getId());  ub.getQuery().addCondition(UserBalance.Field.*amout*,ub.getAmout());  ub.setAmout(ub.getAmout()-50);//扣款50元  updated=db.update(ub);  }**while**(updated==0);  } |

基于较小的调整，我们可以将更新请求改成这样。显然，只有当扣款前的数值和预期的一样的时候，扣款才会发生，否则会再到数据库中查询出最新的值，重新扣款。

这种算法实际上是基于Compare And Swap(CAS)的一种乐观锁实现。但是很少人会在实际项目中这么写。何况我们码农处理多变的业务规则已经够烦心的了，还要考虑在开发时考虑设计乐观锁来?

他们有更简单的理由。他们说，用一个很简单的SQL语句就能准确的扣款50元。

|  |
| --- |
| update userbalance set amount = amount - 50 where id=1 |

没有比这更简单高效的用法了，只要操作一次数据库就可以完成任务。但是这给ORM提出了挑战，怎么样，能够用对象模型完成么？

我们可以这样完成一个扣款的原子操作。

src/main/java/org/easyframe/tutorial/lesson3/CaseUpdate.java

|  |
| --- |
| @Test  **public** **void** testUpdate\_atom() **throws** SQLException {  UserBalance ub = **new** UserBalance();  ub.setId(1);  ub.prepareUpdate(UserBalance.Field.*amout*, **new** JpqlExpression("amout-50"));  **int** updated = db.update(ub);  } |

上例中，prepareUpdate()方法的作用，就是向updateValueMap中标记一个更新的表达式。而updateValueMap的存在，使得我们的Update请求可不仅仅将数据更新为一些字面常量，还可以更新为更多的SQL函数和表达式。

这种写法产生的SQL语句中的参数“50”没有使用绑定变量。对SQL语句绑定变量有严格要求的同学肯定会有意见。对于这些同学，可以这样写——

|  |
| --- |
| @Test  **public** **void** testUpdate\_atom() **throws** SQLException {  UserBalance ub = **new** UserBalance();  ub.setId(1);  ub.prepareUpdate(UserBalance.Field.*amout*, **new** JpqlExpression("amout-:cost")); //扣款额为绑定变量  ub.getQuery().setAttribute("cost", 50);  **int** updated = db.update(ub);  } |

这里我们可以看到，表达式中可以使用绑定变量占位符，这个也是EF-ORM查询语言的特点（参见7 本地化查询）。绑定变量的具体值可以通过Query对象中的Attribute属性传入。

#### 使用prepareUpdate方法

prepareUpdate还可以产生更多的灵活用法。比如，将一个字段的值更新为另外一个字段，或者一个已知的数据库函数。

src/main/java/org/easyframe/tutorial/lesson3/CaseUpdate.java

|  |
| --- |
| @Test  **public** **void** testUpdate\_MoreValues() **throws** SQLException {  UserBalance ub = **new** UserBalance();  ub.getQuery().setAllRecordsCondition();  // 将一个字段更新为另一个字段的值  ub.prepareUpdate(UserBalance.Field.*todayAmount*, UserBalance.Field.*amout*);  // 将updateTime更新为现在的值  ub.prepareUpdate(UserBalance.Field.*updateTime*, db.func(Func.*now*));  // 更新为另外两个字段相加  ub.prepareUpdate(UserBalance.Field.*totalAmount*, **new** JpqlExpression("todayAmount + amout"));  db.update(ub);  } //SQL: update USERBALANCE set TODAYAMOUNT = AMOUT, TOTALAMOUNT = TODAYAMOUNT + AMOUT, UPDATETIME = current\_timestamp |

上例中，我们将todayAmount更新为表的另一个字段值相加，并将updateTime更新为数据库当前的时间。这体现了prepareUpdate的灵活用法。

当然，update中也可以使用JpqlExpression和SqlExpression。这两个对象在前面的Select中已经演示过了。

update的Query部分则可以用前面select中演示过的大部分语法，包括And or等复杂条件的组合。

最终，使用上述办法Entity能够表达出大部分的Update SQL语句的，体现了这个框架的目标——更简单的操作SQL，而不是用对象关系去代替数据库的E-R关系。

### UpdateValueMap的一些特性

#### 回写

src/main/java/org/easyframe/tutorial/lesson3/CaseUpdate.java

|  |
| --- |
| @Test  **public** **void** testUpdate\_Writeback() **throws** SQLException{  Student st=db.load(Student.**class**,1);  st.prepareUpdate(Student.Field.*id*, 199);  **int** oldId=st.getId();  db.update(st);  System.*out*.println("Student的id从"+ oldId +" 更新为" + st.getId());  } |

上例代码运行后，出现信息

|  |
| --- |
| Student的id从1 更新为199 |

这说明，在执行update操作的时候，更新的值会回写到对象中，覆盖旧值。这也使的对象具备同时描述更新前与更新后的值的能力。

同时上述案例也提供了另一种更新主键的操作方式。即使用对象中的主键值作为where条件，使用updateValueMap作为更新的语句。

#### 自动清空

在更新操作完成后，updateValueMap将被清空，对象又重新开始记录赋值操作。

#### stopUpdate和startUpdate

在每个实体的基类DataObject当中，提供了两个方法

|  |  |
| --- | --- |
| **方法** | **用途说明** |
| **startUpdate()** | 让对象开始记录每次的赋值操作。调用此方法后，向对象赋值会被记录到updateValueMap中。 |
| **stopUpdate()** | 让对象停止记录每次的赋值操作。调用此方法后，向对象赋值不会更新到updateValueMap中。 |

要提到的是，对于从数据库中查询出来出来的实体，默认都已经执行了startUpdate’方法，此时对其做任何set操作都会被记录。因此我们可以直接在查出的数据上修改字段，然后直接更新到数据库。新构建的对象，也是出于startUpdate阶段的。因此大部分时候我们无需调用上述两个方法。

#### 通过对比形成updateValueMap

ER-ORM提供了一些其他的方法来生成更新请求。首先、update之前可以和数据库中的旧对象进行比较来生成updateValueMap。

src/main/java/org/easyframe/tutorial/lesson3/CaseUpdate.java

|  |
| --- |
| @Test  **public** **void** testUpdate\_Compare1() **throws** SQLException{  //获得新的st对象  Student newSt=**new** Student();  newSt.setId(1);  newSt.setDateOfBirth(**new** Date());  newSt.setGender("M");  newSt.setName("王五");  //从数据库中获得旧的st对象  Student oldSt=db.load(Student.**class**,1);    //把修改过的值记录到oldSt的updateValueMap中,相等的值不记入  DbUtils.*compareToUpdateMap*(newSt, oldSt);    //如果需要记录字段修改记录，可以直接获取oldSt.getUpdateValueMap()来记录。  db.update(oldSt); //只有数值不同的字段被更新。  } |

通过DbUtils.*compareToUpdateMap()*方法可以比较两个对象的修改内容。另外有一个DbUtils.*compareToNewUpdateMap（）*方法。其区别是，前者是将对比后的updateValueMap生成在old对象中，后者生成在new对象中。生成在new对象中的update操作更为适合在级联场景下，将多个表的数据都update成新的状态。

此外，当得到一个对象时，还可以将除了主键之外的全部数据都主动放置到updateValueMap中去。例如

src/main/java/org/easyframe/tutorial/lesson3/CaseUpdate.java

|  |
| --- |
| @Test  **public** **void** testUpdate\_fillValues() **throws** SQLException {  Student newSt = **new** Student();  newSt.stopUpdate();//不记录赋值操作  newSt.setId(1);  newSt.setDateOfBirth(**new** Date());  newSt.setGender("F");  newSt.setName("张三");  db.update(newSt);//由于未记录赋值变更，此处update操作无效。    DbUtils.*fillUpdateMap*(newSt);//将主键以外所有字段都记录为变更。  db.update(newSt);//update有效  } |

某些时候，对象中的updateValueMap是空的。原因可能有——

* 对象刚刚被执行过update操作，updateValueMap被清空
* 直接从数据库中查出的对象，updateValueMap是空的。
* 类不增强，失去记录赋值变更功能。
* 调用过了stopUpdate()方法。

如同上例演示一般，第一次调用update方法是无效的。  
对于updateValueMap信息缺失的情况，可以用一个DbUtils.*fillUpdateMap()*方法，将对象中除主键外所有属性都标记到updateValueMap中去。因此，第二个update语句相当与更新除主键外的所有字段，就成功操作了。

## 删除记录

### 概述

删除记录和查询记录是基本一样的。都是将Entity-Query当中的数据作为条件。因此几乎所有在单表select中的Query用法，都可以直接在delete查询中使用。反过来，delete中的用法也几乎都可以在select中使用。

区别是select查询返回数据本身，delete操作则返回被删除的记录条数。

### 用法示例

#### 基本操作

在学习了Select的用法以后，我们可以轻易的写出这样的delete用法。

代码： src/main/java/org/easyframe/tutorial/lesson3/CaseDelete.java

|  |
| --- |
| @Test  **public** **void** testDelete\_Basic() **throws** SQLException {  {// Case1. 删除从数据库加载出来的对象  Student st = db.load(Student.**class**, 1);  db.delete(st);  }  { // Case2. 删除所有女生  Student st = **new** Student();  st.setGender("F");  db.delete(st);  }  {//Case3. 删除所有1980年以前出生的学生  Student st = **new** Student();  st.getQuery().addCondition(Student.Field.*dateOfBirth*,Operator.*LESS*,DateUtils.*getDate*(1980, 1, 1));  db.delete(st);  }  } |

#### 使用Query对象

在查询时，对Query对象的赋值操作对应于select语句中的where条件。以此类推，在删除时同样可以通过Query对象传递where条件。

|  |
| --- |
| **public** **void** testDelete\_Basic2() **throws** SQLException {  {// Case1. Between条件,删除账户余额amout在-100到0之间的所有记录。  UserBalance ub=**new** UserBalance();  ub.getQuery().addCondition(QB.*between*(UserBalance.Field.*amout*, -100, 0));  db.delete(ub);  }  { // Case2. 两个字段比较，删除todayAmount和 totalAmout相等的记录  UserBalance ub=**new** UserBalance();  ub.getQuery().addCondition(UserBalance.Field.*todayAmount*, UserBalance.Field.*totalAmount*);  db.delete(ub);  }  {//Case3. 删除按表达式条件删除  UserBalance ub=**new** UserBalance();  ub.getQuery().addCondition(**new** SqlExpression("todayAmount + 100< totalAmount"));  db.delete(ub);  }  } |

上面例子都是将where条件写入到Query对象中，可对比等效的SQL语句。

# 级联操作

级联操作，是指在多个对象建立固定的关联关系，EF-ORM在维护对象的记录时，会将这些关联表中的记录一起维护起来。 在查询数据的时候，也会将这些关联表的中相关记录一起查询出来。

当然上面说的是一般情况下，为了防止使用中通过关联查询获得过多的数据影响性能，现代的ORM框架大多都具备延迟加载特性（又名懒加载）。即只有当使用者请求到这部分数据时，才去数据库中查询。

级联操作除了广泛用于查询以外，还可用于插入、修改、删除等场景。

关联关系根据JPA的定义，一般分为以下四种

|  |  |  |
| --- | --- | --- |
| 类型 | 注解 | java对象定义 |
| 一对一 | @ OneToOne | T |
| 一对多 | @ OneToMany | List<T> / Set<T> /Collection<T>/ T[] |
| 多对一 | @ ManyToOne | T |
| 多对多 | @ ManyToMany | List<T> / Set<T> /Collection<T>/ T[] |

在JPA中，这些关系都使用Annotation来标注。除此之外，EF-ORM还扩展了几个标注，用来支持一些常用的数据库操作的场景。

## 基本操作

### 使用注解描述级联关系

我们通过案例来看，首先创建如下几个实体

EF-ORM支持JPA所定义的几个多表关联和级联操作（部分支持），这些定义包括：@OneToOne,@ManyToOne, @OneToMany, @ManyToMany。使用注解，我们可以在类上描绘出级联关系。例如

src/main/java/org/easyframe/tutorial/lesson4/entity/School.java

|  |
| --- |
| /\*\*  \* 实体:学校  \* 描述一个学校的ID和名称  \*/  **public** **class** School **extends** DataObject{  @Column  @Id  @GeneratedValue(strategy=GenerationType.*AUTO*)  **private** **int** id;    @Column(length=64,nullable=**false**)  **private** String name;  //元模型  **public** **enum** Field **implements** jef.database.Field{  *id*,*name*  }  **public** School(){}  **public** School(String name){  setName(name);  }  Getter Setter此处省略  } |

src/main/java/org/easyframe/tutorial/lesson4/entity/Person.java

|  |
| --- |
| /\*\*  \* 实体:个人  \* 描述一个学校的ID和名称  \*/  @Entity  @Table(name = "t\_person")  **public** **class** Person **extends** DataObject {  @Id  @GeneratedValue(strategy = GenerationType.*AUTO*)  @Column  **private** Integer id;  @Column(name = "person\_name", length = 80, nullable = **false**)  @Indexed(definition = "unique")  **private** String name;  /\*\*  \* 当前学校ID  \*/  @Column(name="CURRENT\_SCHOOL\_ID",columnDefinition="integer")  **private** **int** currentSchoolId;  /\*\*  \* 将currentSchoolId字段和School表的id字段进行关联。从而扩展出关联对象school。  \*/  @ManyToOne(targetEntity = School.**class**)  @JoinColumn(name = "currentSchoolId", referencedColumnName = "id")  **private** School currentSchool;  **public** **enum** Field **implements** jef.database.Field {  *id*, *name*, *currentSchoolId*  }  **public** Person() {}  **public** Person(**int** id) {  **this**.id=id;  }  Getter Setter此处省略  } |

接下来我们在后面的案例中演示这种配置的效果。

### 单表操作方式的保留

在使用这个模型进行级联操作之前，我们先看一下，这个模型依然可以进行单表的操作。这意味着前面所看到的基于单表的Criteria API等各种单表用法在这个场景上依然可用。

src/main/java/org/easyframe/tutorial/lesson4/Case1.java

|  |
| --- |
| /\*\*  \* 级联描述是一种可以后续随时添加删除的扩展描述，原先的单表操作模型保持不变。  \*  \* **@throws** SQLException  \*/  @Test  **public** **void** testNonCascade() **throws** SQLException {  Person p = **new** Person();  {  p.setName("玄德");  p.setCurrentSchoolId(1);  // 虽然Person对象中配置了级联关系。  // 但在EF-ORM中，级联关系是在保证了单表模型完整可用的基础上,补充上去的一种附属描述  // 因此非级联操作一样可用.  db.insert(p);  }  //查出记录  p = db.load(p);  //单表更新  p.setCurrentSchoolId(2);  p.setName("云长");  db.update(p);    //单表删除  db.delete(p);  } |

执行上面的代码，可以看到执行的操作和前面的单表几乎一样。这个案例顺序执行了以下的SQL语句

|  |
| --- |
| insert into T\_PERSON(CURRENT\_SCHOOL\_ID,ID,PERSON\_NAME) values(?,DEFAULT,?)  (1)currentSchoolId: [1]  (2)name: [玄德]  select T1.CURRENT\_SCHOOL\_ID AS T1\_\_CURRENTSCHOOLID,  T1.ID AS T1\_\_ID,  T1.PERSON\_NAME AS T1\_\_NAME,  T2.NAME AS T2\_\_NAME,  T2.ID AS T2\_\_ID  from T\_PERSON T1  left join SCHOOL T2 ON T1.CURRENT\_SCHOOL\_ID=T2.ID  where T1.ID=?  (1)id: [1]  update T\_PERSON set CURRENT\_SCHOOL\_ID = ?, PERSON\_NAME = ? where ID=?  (1)currentSchoolId: [2]  (2)name: [云长]  (3)id: [1]  delete from T\_PERSON where ID=?  (1)id: [1] |

可以发现，除了在查询对象之外，Insert,update,delete操作都没有去维护级联关系。 而查询对象时默认会尝试用左外连接的方式查出用户所属学校，相当于开启了级联功能。

当然，增删改操作下的级联关系是可以维护的，这在后面会讲到。

这里我们阐述EF-ORM对级联关系的理解和处理。在EF-ORM中，级联操作途径不会破坏原有单表操作途径。

级联关系是一种可以灵活的修改和变更的**附加关系模型**。之所以说它是“附加的”，是因为EF-ORM中，级联关系是在保证了单表模型完整可用的基础上，补充的一种关系描述。

上例可以看出，补充上去的模型表现为这一段定义。无论类上有没有定义这个关系，都不影响这个这个类的单表操作。

|  |
| --- |
| /\*\*  \* 将currentSchoolId字段和School表的id字段进行关联。从而扩展出关联对象school。  \*/  @ManyToOne(targetEntity = School.**class**)  @JoinColumn(name = "currentSchoolId", referencedColumnName = "id")  **private** School currentSchool; |

这种设计的补充描述如下：

|  |
| --- |
| **级联不破单表**  *这种做法简单来说就是，“级联模型不破单表模型”，而是依附于单表模型。这和某H框架不一样，这类纯O-R映射试图抹去对象的单表设计，用关系属性直接代替数据库表的外键列。*  *EF为什么要这样设计？因为级联关系在数据库设计时往往表现为表之间的外键。和数据库设计一样，这种外键关系是后来附加的，在用PowerDesign设计数据库的过程中，这表现为两个Entity的一根连线。在数据库设计中，我们可以在不改动Entity本身的情况下，变更它们之间的关系（连线）。为什么到了java代码中，变更关系就要以修改java字段属性对列的映射的方式来实现？*  *这种反传统的设计，目的还是为了还关系数据库以本来面目，正向前面多次提到的那样。为业务操作提供更大的灵活性——即要照顾到级联场合下的高效，也要照顾到一些简单功能时的便捷。*  *此外，随着应用规模的扩大，应用设计可能会扩展为使用分区、分表、分库等大数据用法。在分库分表后，级联操作几乎就无法使用了。考虑到适应数据规模伸缩性的因素，作了这样的设计。* |

### 级联操作的效果

刚才说了单表操作方式得以保留的特点，现在来看级联模型下的用法。

src/main/java/org/easyframe/tutorial/lesson4/Case1.java

|  |
| --- |
| @Test  **public** **void** testCascade() **throws** SQLException{  School school=db.load(**new** School("天都中学"));  **int** personId;  {  Person p = **new** Person();  p.setName("玄德");  p.setCurrentSchool(school);  db.insertCascade(p);  personId=p.getId();  }  {  //查出记录  Person p=db.load(**new** Person(personId));  System.*out*.println(p.getCurrentSchoolId()+":"+p.getCurrentSchool().getName());  //可以看到级联对象：School被查出来了  *assertEquals*("天都中学",p.getCurrentSchool().getName());    //更新为另一学校  p.setCurrentSchool(**new** School("天都外国语学校"));  //外国语学校是新增的，在更新语句执行之前会先做插入School表操作。  db.updateCascade(p);  System.*out*.println("天都外国语学校 = "+p.getCurrentSchoolId()+"= "+p.getCurrentSchool().getId());  //再使用单表更新，更新回原来的学校  p.setCurrentSchoolId(school.getId());  db.update(p);    //删除该学生  db.deleteCascade(p);  }  } |

这一次发现被执行的SQL语句多了不少，由于最后记录被删除了，所以下面解说一下每个步骤对应的SQL操作。

|  |
| --- |
| *select t.\* from SCHOOL t where t.NAME=?*  *(1)name: [天都中学]*  查出“天都中学”。对应代码中的: School school=db.load(**new** School("天都中学"));  *select t.\* from SCHOOL t where t.ID=?*  *(1)id: [1]*  *insert into T\_PERSON(CURRENT\_SCHOOL\_ID,ID,PERSON\_NAME) values(?,DEFAULT,?)*  *(1)currentSchoolId: [1]*  *(2)name: [玄德]*  **级联插入，**上面两句，对应代码中的*db.insertCascade(p)*;，第一个select语句是在级联操作之前先检查“天都中学”是否存在，如果不存在会补充插入。  *select T1.CURRENT\_SCHOOL\_ID AS T1\_\_CURRENTSCHOOLID,*  *T1.ID AS T1\_\_ID,*  *T1.PERSON\_NAME AS T1\_\_NAME,*  *T2.NAME AS T2\_\_NAME,*  *T2.ID AS T2\_\_ID*  *from T\_PERSON T1*  *left join SCHOOL T2 ON T1.CURRENT\_SCHOOL\_ID=T2.ID*  *where T1.ID=?*  *(1)id: [1]*  **级联查询**，对应代码中的*Person p=db.load(****new*** *Person(personId))*;  *insert into SCHOOL(NAME,ID) values(?,DEFAULT)*  *(1)name: [天都外国语学校]*  *update T\_PERSON set CURRENT\_SCHOOL\_ID = ? where ID=?*  *(1)currentSchoolId: [4]*  *(2)id: [1]*  **级联更新**，对应第一个db.updateCascade(p);//外国语学校是新增的，在更新语句执行之前会先做插入School表操作。  *update T\_PERSON set CURRENT\_SCHOOL\_ID = ? where ID=?*  *(1)currentSchoolId: [1]*  *(2)id: [1]*  **非级联更新**，对应代码中的db.update(p); 直接将schoolId更新为1.  *select t.\* from T\_PERSON t where t.ID=?*  *(1)id: [1]*  *delete from T\_PERSON where ID=?*   1. *id: [1]*   **级联删除**，看似除了做了一次不需要的查询以外别的啥也没做。实际上，查询操作的目的是为了得到当前数据库中的Person记录的各个键值。如果这些键值中有需要去删除别的表中记录的键（例如 OneToMany）那么就会去删除别的表中的记录。  本例中，Person到School是ManyToOne关系。这意味着School中的对象可能被其他记录所使用，因此不会去删除School表中的对象。 |

上面的例子，演示了级联操作的用法。还包括了混合了一次非级联更新操作。

由于在Person中，同时存在 currentSchoolId和currentSchool两个字段，这两个字段将分别在单表操作和级联操作中使用。（即单表字段和级联描述字段同时存在）。

某些场合下，人为操作可能使Person对象的currentSchoolId和currentSchool可能指向不同的关联记录。为了明确的区分当前用户是使用级联模型操作，还是单表模型操作，EF-ORM将级联插入、更新、删除的API和非级联下的API显式的分离。

|  |  |  |
| --- | --- | --- |
|  | 非级联操作 | 级联操作 |
| 插入记录 | insert() | insertCascade() |
| 更新记录 | update() | updateCascade() |
| 删除记录 | delete() | deleteCascade() |

表5-1 级联和非级联操作API的对比

我们再看例子中的这几句。

|  |
| --- |
| //更新为另一学校  p.setCurrentSchool(**new** School("天都外国语学校"));  //外国语学校是新增的，在更新语句执行之前会先做插入School表操作。  db.updateCascade(p);  System.*out*.println("天都外国语学校 = "+p.getCurrentSchoolId()+"= "+p.getCurrentSchool().getId()); |

显然，在设置新的School对象到Person上之后，person.getCurrentSchoolId() 属性依然指向旧的School记录。但是更新操作完成后。除了刚刚被插入数据库的School对象中的id字段更新为数据库主键之外， Person对象中的外键值也被正确的更新了。

也就是说，虽然过程中Person对象短暂的出现指向School对象不一致的问题，但是在级联操作完成后，ID指向将会被正确的维护。

通过分离的级联操作API，我们应该可以清楚的知道，插入数据库的值来自单表模型还是级联模型。

级联操作API分离的另外一个好处是，由于级联下不支持分库分表。因此API的分离更容易在分库分表时被掌握和控制。

最后，可能有人会问，select和load操作默认都是级联的。这能关闭吗？看下面这个例子

|  |
| --- |
| Person query=**new** Person();  query.setId(personId);  //设置为非级联查询  query.getQuery().setCascade(**false**);  p= db.load(query);  System.*out*.println(p.getCurrentSchool());  *assertNull*(p.getCurrentSchool()); //关闭级联开关后不做级联查询，所以School对象得不到了 |

因此，级联查询开关是可以关闭的。

## 使用注解定义级联行为

这里说的注解即(Annotation)。

### 仅引用级联对象的单个字段

我们考虑这样一个场景——

Person类中有一个gender的字段，描述用户性别，M表示男性，F表示女性。

另外有一张DATA\_DICT表，其中记录了M=男性 F=女性的对应关系。

在查询时，我们希望查出”男”,”女”的属性。而不关心DATA\_DICT表中的其他字段。

查询时，很多时候我们希望**只引用目标对象中的个别字段**，并不希望引用整个对象。实际上这种情况在业务中很常见。

@FieldOfTargetEntity注解就是为这种场景设计的。

我们定义的DATA\_DICT对象如下：

|  |
| --- |
| @Entity  @Table(name="data\_dict")  **public** **class** DataDict **extends** DataObject {  @Id  @GeneratedValue  **private** **int** id;    @Column(name="dict\_type")  **private** String type;  @Column(name="value")  **private** String value;  @Column(name="text")  **private** String text;  **public** **enum** Field **implements** jef.database.Field {  *id*, *type*, *value*, *text*  }  //Getter setter方法略  } |

然后我们在Person对象中。定义一个“单字段”的引用字段。

|  |
| --- |
| /\*\*  \* 性别的显示名称“男”“女”  \*/  @ManyToOne(targetEntity=DataDict.**class**)  @JoinColumn(name="gender",referencedColumnName="value")  @FieldOfTargetEntity("text")  **private** String genderName; |

上例中，我们指定了“只引用目标对象中的 text字段”。

每次查询时，就像是访问Person表本身的属性一样。可以查出存储在另一张表中的用户性别的“男”，“女”这样的字样。

这种方式具有以下特点

* 1. 由于引用的字段较少，性能会比引用整个对象高很多。
  2. 因为引用描述的不是一个完整对象。因此这种引用方式只对查询生效，对插入、更新、删除不会产生影响。

本节总结如下

@FieldOfTargetEntity

用于指定单字段引用。

|  |  |
| --- | --- |
| 属性 | 作用 |
| value | String类型。字段名，表示仅引用目标entity中的指定字段。 |

我们建议在EF-ORM中使用单字段引用，不再引用整个对象，意味着查询效率的提高。此外，如果相同的引用关系（比如引用目标对象的两个字段），EF-ORM会合并处理（在SQL语句中指定引用的两个字段），因此不会造成多余的数据库查询。

### @JoinDescription、@OrderBy

#### 定义与作用

但是事实情况要比理想中的更为复杂，我们不会就为了存储一个 {M=男, F=女}这样的对应关系去设计一张表，现实中，往往会是这种情况——

DATA\_DICT表中另外有一个type字段，当type=’USER.GENDER’时，才表示性别的对应关系。当 type等于别的值时，这些记录表示别的对应关系。（比如 0=’在线’ 1=’离线’，这样的关系）。

所以在orm-tutorial\src\main\java\org\easyframe\tutorial\lesson4\Case1.java中，示例要更复杂一些。对genderName的定义是这样的——

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson4/ntity/Person.java

|  |
| --- |
| /\*\*  \* 性别的显示名称“男”“女”  \*/  @ManyToOne(targetEntity=DataDict.**class**)  @JoinColumn(name="gender",referencedColumnName="value")  @JoinDescription(filterCondition="type='USER.GENDER'") //在引用时还要增加过滤条件 @FieldOfTargetEntity("text")  **private** String genderName; |

通过增加@JoinDescription这样的注解，为SQL中的Join关系增加了一个过滤条件。

实际上查询的SQL语句变为(示意)

|  |
| --- |
| select person.\*, data\_dict.text  from person  left join data\_dict on person.gender = data\_dict.value and type=’USER.GENDER’  where  …. |

这样就起到了过滤其他类型的对应关系的效果。

实际示例代码如下

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson4\Case1.java

|  |
| --- |
| **public** **void** testGetFieldFromManyToOne() **throws** SQLException{  //准备数据  Person p1=**new** Person();  p1.setName("孟德");  p1.setGender('M');  Person p2=**new** Person();  p2.setName("貂蝉");  p2.setGender('F');  *db*.insert(p1);  *db*.insert(p2);    //查出数据  Query<Person> query=QB.*create*(Person.**class**);  query.addCondition(QB.*notNull*(Person.Field.*gender*));  query.orderByAsc(Person.Field.*gender*);  List<Person> p=*db*.select(query);  *assertEquals*("女人", p.get(0).getGenderName());  *assertEquals*("男人", p.get(1).getGenderName());  } |

上面介绍了@JoinDescription组合的用法。@JoinDescription用来描述多表关联关系时一些额外的特征与特性。

虽然上面的例子中同时使用了@FieldOfTargetEntity和@JoinDescription，但两者各有各的作用，并不一定要组合使用。@JoinDescription和@FieldOfTargetEntity使用上没有必然联系。

@ JoinDescription

|  |  |
| --- | --- |
| 属性 | 作用 |
| type | 枚举常量jef.database.annotation.JoinType。  LEFT 左外连接  RIGHT 右外连接  INNER 内连接  FULL 全外连接 |
| filterCondition | JPQL表达式，表示Join时额外的条件。表达式中可以包含参数变量。 |
| maxRows | 当对多连接时，限制结果记录数最大值。 |

#### 控制级联对象的排序和数量

上表中提到了JoinDescription中的另外属性，我们举例说明

|  |
| --- |
| @OneToMany(targetEntity = ExecutionLog.**class**, fetch = FetchType.*EAGER*, cascade = CascadeType.*REFRESH*)  @JoinColumn(name = "id", referencedColumnName = "taskId")  @JoinDescription(maxRows = 10)  @OrderBy("executeTime desc")  **private** List<ExecutionLog> lastExecutionLog; //最后10条Execution Log |

上例表示一个指向ExecutionLog的一对多引用。但是对应的ExecutionLog有很多条，我们只取**执行时间最近的10条**。之前的记录不会查出来。

JPA注解@OrderBy可以用于控制级联关系的排序。@OrderBy("execute\_time desc")表示对于executeTime字段进行倒序排序。

如果maxRow=1，那么这个映射将只对应最后一条ExecutionLog。那么数据类型可以进一步简化——

|  |
| --- |
| @OneToMany(targetEntity = ExecutionLog.**class**, fetch = FetchType.*EAGER*, cascade = CascadeType.*REFRESH*)  @JoinColumn(name = "id", referencedColumnName = "taskId")  @JoinDescription(maxRows = 1)  @OrderBy("executeTime desc")  **private** ExecutionLog lastExecutionLog; //最后一条Execution Log，不使用集合类型。 |

最后一条ExecutionLog是单值的，可以不使用集合类型。

#### 在FilterCondition中使用变量

某些时候，我们希望FilterCondition中的表达式中出现的不是固定的常量，而是运行时得到的变量。

比如前面那个 “M=男 F=女”的转换例子，如果数据字段中的映射名称并不总是” USER.GENDER”，而是一个变化的值。那该怎么办呢？

|  |
| --- |
| /\*\*  \* 性别的显示名称“男”“女”  \*/  @ManyToOne(targetEntity=DataDict.**class**)  @JoinColumn(name="gender",referencedColumnName="value")  @JoinDescription(filterCondition="type=:dictType") //将条件设置为变量  @FieldOfTargetEntity("text")  **private** String genderName; |

在执行查询时——

|  |
| --- |
| Query<Person> query=QB.*create*(Person.**class**);  query.addCondition(QB.*notNull*(Person.Field.*gender*));  query.orderByAsc(Person.Field.*gender*);  query.setAttribute("dictType", "USER.GENDER"); //为FileterCondition中的变量赋值  List<Person> p=*db*.select(query); |

在运行时就会将其作为绑定变量处理。

这种用法适用于以下两种场景——

* Join过滤条件中存在不确定的值时
* SQL语句中必须使用绑定变量时

### 其他JPA注解的支持

这一节，我们不介绍用法，而是针对级联时的操作行为进行一些分析

#### 延迟加载

我们修改一下Person.java，将Person.java中的currentSchool定义改成下面这样。

|  |
| --- |
| /\*\*  \* 学校映射  \*/  @ManyToOne(targetEntity = School.**class**,fetch=FetchType.*LAZY*) //显式指定该字段为延迟加载的。  @JoinColumn(name = "currentSchoolId", referencedColumnName = "id")  **private** School currentSchool; |

再次运行下面的测试案例

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson4\Case1.java

|  |
| --- |
| @Test  **public** **void** testLazyLoad() **throws** SQLException{  Person query=**new** Person();  query.setId(*firstId*);  Person p=*db*.load(query);  System.*out*.println("接下来观察调用get方法后，才会输出加载School的SQL语句。");  p.getCurrentSchool();  //请观察输出的SQL语句，  } |

通过输出可以观察到，启用了Lazy的Fetch方法之后，Person对象从数据库中查出时，School对象是未赋值的。只有当调用了getCurrentSchool()之后，才会去数据库中加载School对象。

在JPA定义中，fetch属性用于指定级联加载的行为。

|  |  |
| --- | --- |
| fetch属性取值 | 效果 |
| FetchType.EAGER | 饥渴的。查询时立刻加载级联对象 |
| FetchType.LAZY | 懒惰的。只有当属性被使用时（调用get方法），才会去加载级联对象。（延迟加载） |

JPA的四种级联关系中，默认的加载行为是不同的。

|  |  |  |
| --- | --- | --- |
| 类型 | 注解 | 缺省的fetch行为 |
| 一对一 | @ OneToOne | EAGER |
| 一对多 | @ OneToMany | LAZY |
| 多对一 | @ ManyToOne | EAGER |
| 多对多 | @ ManyToMany | LAZY |

如果不使用延迟加载，EF-ORM也会使用“外连接获取”的方式来减少数据库操作次数。见下一节。

#### 外连接获取

从5.1.3的例子中，我们可以从日志观察到：当Person对象中有两个多对一的引用时，在查询Person时，实际SQL语句如下

|  |
| --- |
| select T1.ID as T1\_\_ID,  T1.PERSON\_NAME as T1\_\_NAME,  T1.CURRENT\_SCHOOL\_ID as T1\_\_CURRENTSCHOOLID,  T1.GENDER as T1\_\_GENDER,  T2.TEXT as T2\_\_TEXT,  T3.NAME as T3\_\_NAME,  T3.ID as T3\_\_ID  from T\_PERSON T1  left join DATA\_DICT T2 ON T1.GENDER=T2.VALUE and T2.DICT\_TYPE='USER.GENDER'  left join SCHOOL T3 ON T1.CURRENT\_SCHOOL\_ID=T3.ID  where T1.ID=? |

也就是说，在查询Person表中的记录时，将对应的School对象和性别“男/女”的显示问题都一次性的查了出来。最后形成的是并不是一个简单的单表查询语句，而是一个多表联合的Join语句。为了防止Join无法连接的记录（比如未对应School的Person记录）被过滤，因此使用了左外连接(Left Join)。

一般来说，关联查询可以通过多次数据库查询完成，也可以用上面那样较为复杂的SQL语句一次性从数据库中查询得到，这种操作方式被称为外连接查询。

这样想，如果查询出10个Person。如果不使用外连接，那么我们还需要执行10次SQL语句去获得每个Person对象的School。但使用外连接后，一次数据库操作就能代替原先11次操作。

外连接查询只能应用与ManyToOne和OneToOne的级联关系中，并能有效的减少数据库访问次数，在某些场合下具有较好的性能。由于JPA注解中OneToOne和ManyToOne的缺省FetchType都是EAGER，即不使用延迟加载。此时单次SQL操作性能更好，因此这也是EF-ORM的默认实现方式。

如果你希望沿用原先执行多次单表查询的方式，可以使用setCascadeViaOuterJoin(false)方法。如下——

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson4\Case1.java

|  |
| --- |
| @Test  **public** **void** testNonOuterJoin() **throws** SQLException{  Person query=**new** Person();  query.setId(*firstId*);  query.getQuery().setCascadeViaOuterJoin(**false**); //改变默认行为，不使用外连接。  Person p=*db*.load(query);  p.getCurrentSchool();  p.getGenderName();  //请观察输出的SQL语句，  } |

除此之外，从5.2.3.1的内容也可以知道，使用JPA注解显式声明为LAZY，也可以使得级联关系从外连接转变为多次加载。

因此我们总结如下

1、当FetchType标记为LAZY时，所有级联 关系都通过多次的单表查询来实现。

2、当FetchType为EAGER时，OneToOne和ManyToOne可以优化为使用外连接单次查询。OneToMany和ManyToMany无法优化。

3、如果调用setCascadeViaOuterJoin(**false**)方法，或者配置全局参数 db.use.outer.join=false（见附录一），那么EF-ORM将放弃使用外连接优化。此时性能问题依赖于一级缓存和延迟加载来解决。

#### 级联方向

EF-ORM中，级联关系是针对单个对象的辅助描述，因此所有的级联关系维护都是单向的。上面的例子中，如果我们删除 School表中的数据，不会引起Person数据的任何变化。

如果我们想要在删除School时，级联删除该School的所有学生，那么需要修改School.java这个类。

修改后的School类如下

|  |
| --- |
| /\*\*  \* 实体:学校  \* 描述一个学校的ID和名称  \*/  **public** **class** School **extends** DataObject{  @Column  @Id  @GeneratedValue(strategy=GenerationType.*AUTO*)  **private** **int** id;    @Column(length=64,nullable=**false**)  **private** String name;  **public** **enum** Field **implements** jef.database.Field{  *id*,*name*  }  @OneToMany  @JoinColumn(name="id",referencedColumnName="currentSchoolId")  **private** List<Person> persons;  //getter setter省略  } |

当配置了从School到Person的关系后，如果级联删除School对象，则也会删除对应的Person对象。

上面的配置中， @JoinColumn注解可以省略。如果未配置Join的路径，EF-ORM会到目标对象中去查找反向关系。

因此最简的配法是

|  |
| --- |
| @OneToMany  **private** List<Person> persons; |

### 定制级联行为

#### 四种级联关系的行为

我们来了解EF-ORM在使用级联方法 (insertCascade()、 updateCascade() deleteCascade()等方法 )操作时，实际执行了什么动作。

在EF-ORM中，所有的四种级联关系，都是两表间的关系，不会出现第三张表，包括多对多（@ManyToMany）。事实上这也使ManyToMany的作用和OneToMany其实差不多，因此实用性不大。

这四种级联都对应一些简单的操作步骤，而且并没有提供太多的定制配置。对熟悉H框架的人来说，EF-ORM的级联行为和某H框架并不完全一致。EF-ORM希望设计出一些一刀切的简单规则来处理级联问题，从而降低ORM框架的使用难度。

一个典型的示例就是mappedby属性在EF-ORM中是无效的，EF-ORM也不会为OneToMany生成一张所谓中间表这样的设计。此外，下面的示例在某H框架中是不能删除子表对象的，但在EF-ORM中可以做到。

父表的定义

orm-tutorial/src/main/java/org/easyframe/tutorial/lesson5/entity/Catalogy.java

|  |
| --- |
| @Entity  **public** **class** Catalogy **extends** DataObject {  @Id  @GeneratedValue  **private** **int** id;  **private** String name;  **public** **enum** Field **implements** jef.database.Field {  *id*, *name*  }    @OneToMany(mappedBy="catalogyId")  //@JoinColumn(name="id",referencedColumnName="catalogyId")  **private** List<Item> items;  //getter setter省略  } |

这里顺便提一下——当使用主键和其他实体关联时，可以将@JoinColumn(name="id",referencedColumnName="catalogyId")简化为mappedBy="catalogyId"。两者是等效的。

子表的定义

orm-tutorial/src/main/java/org/easyframe/tutorial/lesson5/entity/Item.java

|  |
| --- |
| @Entity()  **public** **class** Item **extends** DataObject {  @Id  @GeneratedValue  **private** **int** id;  **private** String name;  **private** **int** catalogyId;    **public** **enum** Field **implements** jef.database.Field {  *id*, *name*, *catalogyId*  }  //getter setter省略  } |

运行下面的代码——

orm-tutorial/src/main/java/org/easyframe/tutorial/lesson5/Case1.java

|  |
| --- |
| **public** **void** testCascade() **throws** SQLException{  Catalogy c=**new** Catalogy(); //Catalogy表是父表  c.setId(1);  c=*db*.load(c);  *assertNotNull*(c);  *assertEquals*(4, c.getItems().size()); //Item表是子表  c.setItems(**null**);  System.*out*.println("设置为null，调用级联更新，会删除子表记录");  Transaction tx=*db*.startTransaction();  tx.updateCascade(c);  tx.rollback();  } |

在EF-ORM中，四种级联关系在增、删、改的时候执行的动作是固定的，下表列出了四种不同级联情况下框架执行的操作。

其中——当前对象指配置了级联关系的对象（如本例的Catalogy对象），关联对象指被引用的对象（本例的Item对象）。

|  |  |  |  |
| --- | --- | --- | --- |
| 关系类型 | insertCascade | updateCascade | deleteCascade |
| OneToOne | 1 插入当前对象。  2 检查关联对象—— 如为新对象则插入  如为旧对象且变化则更新。 | 1 更新当前对象  2 检查关联对象——  如为新对象则插入  如为旧对象且变化则更新。  如为不再使用则删除 | 1 删除当前对象  2 再删除关联对象。 |
| OneToMany | 1 插入当前对象。  2 检查关联对象—— 如为新对象则插入  如为旧对象且变化则更新。 | 1 更新当前对象  2 检查关联对象—— 如为新对象则插入  如为旧对象且变化则更新。  如为不再使用则删除 | 1 删除当前对象  2 再删除关联对象。 |
| ManyToOne | 1 检查关联对象——  如为新对象则插入  如为旧对象且变化则更新;  2 插入当前对象。 | 1 检查关联对象——  如为新对象则插入  如为旧对象且变化则更新  不做级联删除  2 再更新当前对象 | 1 删除当前对象  不做级联删除。 |
| ManyToMany | 1 插入当前对象。  2 检查关联对象—— 如为新对象则插入  如为旧对象且变化则更新。 | 1 更新当前对象  2 检查关联对象—— 如为新对象则插入  如为旧对象且变化则更新。  不做级联删除 | 1 删除当前对象  不做级联删除。 |

上表列出了四种不同的映射关系下的行为，可以发现基本上各个行为都只分为维护对象本身和维护级联关系两种。  
 上表中不同的行为特点是根据大量业务实际情况推导出的。例如：多对一这种关系不做级联删除，这也是考虑到大多数多对一的场景下，被引用的对象一般是公用的。

另外要注意，级联操作是递归的。比如——

A对象@OneToMany到B对象， B对象OneToOne到C对象，在对A对象级联操作的同时，会按上述规则一直维护到C对象为止。如果是更多的对象参与级联，那么以此类推。

上述行为中ManyToMany和某H框架差别较大，正常情况下，我们维护多对多关系总是需要一张中间表。因此某H框架提供了隐含的中间表实现。即两个实体、一个关系，总共三张表。EF-ORM早期几个版本也是这样设计的，但是后续版本中去除了这种设计。

|  |
| --- |
| *因为我们发现，自动维护的中间表不容易扩展。比如：多个考生参加多项课程的考试，获得分数*  *这样一个典型的多对多关系。*  *很显然，在关系表上我们需要增加 分数列、然后我们需要描述这个关系的创建时间——考试日期等等。*  *从业务实践看，单纯的只有两个实体的主键构成的关系实用性很低，而且不利于后续的扩展。*  *早期EF-ORM的设计还不是很清晰的时候，ManyToMany这种表面上是两表关系，实际上是三表关系的设计，极大的增加了框架的复杂度。为此，在权衡复杂度和实用性之后，EF-ORM在0.9版本中去除了这种设计。* |

那么，如果我们碰到有类似于E-R-E这样的多对多关系时，该怎么做呢？更倾向于设计三个实体，（例如 学生、课程、考试）分别维持以下关系——

|  |
| --- |
| class 学生{  @OneToMany(target = 考试)  }  class 考试{  @ManyToOne(target= 课程)  @ManyToOne(target= 学生)  }  class 课程{  @OneToMany(target=考试)  } |

也就是说，多对多关系被拆分为两个一对多关系。

这是不是说EF-ORM中的@ManyToMany就没有用了呢？不是这样的，再举一个例子。

某学校有10个班，每个班可以有多个学生和多个任课老师。

**教师任课表**

|  |  |  |
| --- | --- | --- |
| 教师ID | 班级ID | 课程ID |

**学生表**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 学生ID | 班级ID | 姓名 | 性别 | 年龄 |

在这两个实体中，我们就可以建立双向@ManyToMany的关系。从学生记录中，获得该学生所在班级的所有课程和任课教师。 从教师担任的课程的记录中，获得该班级所有的学生记录。

我们怎么区分什么时候该使用@OneToMany什么时候使用@ManyToMany呢？可以遵循这样的原则：

* 当目标实体依赖于当前实体（强关联）时使用OneToMany；
* 当目标实体相对独立 （弱关联）时使用ManyToMany。

弱关联情况下，级联操作将会使用一种较为保守的策略，来避免出现删除关联对象之类的误操作。

关于这个ManyToMany的模型，可以参阅orm-tutorial/src/main/java/org/easyframe/tutorial/lesson5/Case1.java 中的testManyToMany方法。本文不再赘述。

#### 使用注解限制级联关系的使用场合

上面提到了，级联关系是递归维护的。而且要注意，一个对象中可能存在多种级联关系，而其级联对象中又可能有多种级联关系。在递归情况下，情形可能会变得相当复杂。如果没有很好的设计你的E-R关系，你可能会发现，一个简单的数据库操作可能会衍生出大量的级联操作语句。

因此，JPA中可以限制级联关系的使用范围。使用JPA注解，使级联关系只在插入/更新/删除等一种或几种行为中生效。简而言之，我们可以让配置的级联关系只用于查询操作、或者只用于插入操作。

在@OneToOne @OneToMany @ManyToOne @ManyToMany中，可以设置cascade属性如下

|  |
| --- |
| @ManyToOne(targetEntity = School.**class**,cascade={CascadeType.*MERGE*,CascadeType.*REFRESH*}) |

cascade是一个多值属性，配置了以后，则表示此类操作要进行级联，否则就不作级联操作。例如上例配置该字段仅用于级联查询和级联插入更新，不用于级联删除。

|  |  |
| --- | --- |
| **CascadeType** | **对应** |
| CascadeType.PERSIST | 允许级联插入 |
| CascadeType.MERGE | 允许级联插入或更新 |
| CascadeType.REMOVE | 允许级联删除 |
| CascadeType.REFERSH | 允许级联查询 |
| CascadeType.ALL | 允许所有级联操作 |

### 级联条件CascadeCondition

在级联查询中，我们还能在级联查询的对象上添加条件。请看下面这个例子：

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson4\Case1.java

|  |
| --- |
| @Test  **public** **void** testCascadeCondition() **throws** SQLException{  Query<Person> query=QB.*create*(Person.**class**);  query.addCondition(QB.*eq*(Person.Field.*id*,*firstId*));  query.addCascadeCondition(QB.*matchAny*(School.Field.*name*, "清华"));  *db*.select(query);  } |

在这个例子中，使用addCascadeCondition()方法，在查询中增加了一个专门针对级联对象的过滤条件。在查询级联时中，只有带有”清华“字样的学校才会被查出。

另一个例子是在多对多时

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson5\Case1.java

|  |
| --- |
| @Test  **public** **void** testCascadeCondition() **throws** SQLException {  {//无级联条件时  Student s1 = *db*.load(Student.**class**,1);  **for**(TeacherLesson t:s1.getLessons()){  System.*out*.println(t);  }  }  {  Student st=**new** Student();  st.setId(1);  st.getQuery().addCascadeCondition(  QB.*in*(TeacherLesson.Field.*lessonName*, **new** String[]{"语文","化学"}));  **for**(TeacherLesson t:*db*.load(st).getLessons()){  System.*out*.println(t);  }  }    } |

可以发现，即使是在延迟加载的场景下，级联条件依旧会在查询时生效。

由于级联是递归的，即级联对象中我们还可以获得其他级联对象，因此过滤条件也不仅仅针对当前对象的，而是可以灵活的指定。我们再看复杂一点的例子

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson5\Case1.java

|  |
| --- |
| /\*\*  \* 级联过滤条件也可以用于间接的引用中，  \* 本例中， Cacalogy引用Item、Item引用ItemExtendIndo，通过指定引用字段，可以精确控制过滤条件要用于那个对象上。  \*/  @Test  **public** **void** testCascadeCondition2() **throws** SQLException {  Query<Catalogy> q = QB.*create*(Catalogy.**class**);  q.addCondition(QB.*eq*(Catalogy.Field.*id*, 1));  //指定过滤条件作用于 Catalogy的一个间接级联关系上  q.addCascadeCondition("items.itemExtInfos", QB.*eq*(ItemExtendInfo.Field.*key*, "拍摄地点"));  Catalogy c = *db*.load(q);  **for** (Item item : c.getItems()) {  System.*out*.print(item.getItemExtInfos());  }  //注意观察SQL语句  } |

Cascade条件一旦设置到一个查询中，无论级联操作是单次操作还是多次操作，都会生效。

# Criteria API多表操作

级联操作本质上是一种单表操作。所有的级联操作都可以以多次单表查询的方式来完成。

但前面的大多数级联操作都使用了自动外连接的功能，因此在查询时，实际上看到的是一个多表连接后的Join查询语句。这种SQL语句在EF-ORM内实现也是使用Criteria API来做到的。

前面的单表Criteria API中大家可以理解，一个单表操作SQL语句可以被抽象后用一个Query对象表示。多表操作Criteria API的概念也差不多。一个多表查询的SQL语句被java抽象后，可以用一个Join对象或者一个UnionQuery对象表示。

Join对象其实就是一个典型的Join的SQL语句在Java中的映射，一般来说标准的Join语句是

|  |
| --- |
| select t1.\*, t2.\* … from TABLE1 t1,  left join TABLE2 t2 on t1.xx=t2.xx  left join TABLE3 t3 on t1.xx=t3.xx and t2.xx=t3.xx  where t1.xxx=条件 and t2.xxx=条件 and t3.xxx=条件 |

可以发现，一个Join语句是针对多个表（或者视图或查询结果）。每个表都可以有自己的where条件（或者没有条件）。表和表之间用 ON 条件进行连接。

很多时候，我们在不太复杂的Join语句中，将On条件也放在where条件后面来写，这样写更方便，不过某些时候的计算顺序会和标准写法出现误差，造成一些计算结果不正确的问题。

在Java中，我们也将SQL语句映射为一个Join对象，其包含了多个Query对象，Query之间用 on 条件进行连接。on条件数量不定。



图6-1 Join的构成

上图可以看出，一个Join由多个单表的Query对象构成。我们在实际使用时，可以自由的组合各种Query，形成一个Join查询对象。这个模型实际上和我们编写的SQL是一样的。

## Join查询

### 基本用法

多表查询的主要场景都是Join查询。Join查询由多个单表的子查询构成，这点在前面已经叙述了。来看看实际用法。

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson6\Case1.java

|  |
| --- |
| @Test  **public** **void** testMultiTable() **throws** SQLException {  Query<Person> p = QB.*create*(Person.**class**);  Query<Item> i = QB.*create*(Item.**class**);  Join join = QB.*innerJoin*(p, i, QB.*on*(Person.Field.*name*, Item.Field.*name*));  // 不指定返回数据的类型时，Join查询默认返回Map对象。  Map<String, Object> o = *db*.load(join);  System.*out*.println(o);  // 如果指定返回“多个对象”，那么返回的Object[]中就包含了 Person对象和Item对象  {  Object[] objs = *db*.loadAs(join, Object[].**class**);  Person person = (Person) objs[0];  Item item = (Item) objs[1];  *assertEquals*(person.getName(), item.getName());  System.*out*.println(person);  System.*out*.println(item);  }  // 上面的join对象中只有两张表，还可以追加新的表进去  {  join.innerJoin(QB.*create*(Student.**class**),  QB.*on*(Person.Field.*name*, Student.Field.*name*));  Object[] objs = *db*.loadAs(join, Object[].**class**);  Person person = (Person) objs[0];  Item item = (Item) objs[1];  Student student = (Student) objs[2];  *assertEquals*(person.getName(), item.getName());  *assertEquals*(item.getName(), student.getName());  System.*out*.println(student);  }  } |

从上例中，我们可以发现

* 1. 可以用QueryBuilder.innerJoin()将两个Query对象拼接在一起形成内连接。形成Join对象后，join对象自带innerJoin方法，用于添加新的查询表。
  2. 使用QueryBuilder.on()可以指定Join时的连接键。
  3. 如果不指定Join查询的返回类型，那么会返回所有字段形成的Map，如果指定Join查询返回Object[]类型。那么会将所有参与查询的表的映射对象以数组的形式返回。  
     Join场景下，查询结果返回的最常用形式是这两种。但其实EF-ORM也支持其他很多种结果转换形式。相关内容请参见8.1节 查询结果的返回。

在QueryBuilder中，有以下几个方法用于创建Join连接

|  |  |
| --- | --- |
| 方法 | 效果 |
| **innerJoin(q1,q2)** | 创建内连接 |
| **leftJoin(q1,q2)** | 创建左外连接 |
| **rightJoin(q1,q2)** | 创建右外连接 |
| **outerJoin(q1,q2)** | 创建全外连接 |
| **innerJoinWithRef(q1,q2)** | 先根据左侧Query对象的级联关系创建级联Join，然后再将其和右侧的对象内连接。 |
| **leftJoinWithRef(q1,q2)** | 先根据左侧Query对象的级联关系创建级联Join，然后再将其和右侧的对象左外连接。 |
| **rightJoinWithRef(q1,q2)** | 先根据左侧Query对象的级联关系创建级联Join，然后再将其和右侧的对象右外连接。 |
| **outerJoinWithRef(q1,q2)** | 先根据左侧Query对象的级联关系创建级联Join，然后再将其和右侧的对象全外连接。 |

在Join对象中，也有类似的方法。在连接查询中增加新的表。

在上述方法中，入参包括可选择的On条件，该条件指出了新参与查询的表通过哪些键和旧的表连接。使用 QueryBuilder.on()方法，可以为两个Query指定连接条件。

### 使用Selects对象

其实就EF-ORM的API设计来说，并没有明显的区分单表操作和多表操作，前面单表查询的各种用法，也都可以直接在多表查询中使用。本例中的Selects使用来控制查出的列就是一例。

此外，单表查询中的 SQLExpression、JpqlExpression等特性都可以在多表查询中使用。

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson6\Case1.java

|  |
| --- |
| @Test  **public** **void** testSelectFromJoin() **throws** SQLException{  Query<Person> p = QB.*create*(Person.**class**);  Query<Item> i = QB.*create*(Item.**class**);  Join join = QB.*innerJoin*(p, i, QB.*on*(Person.Field.*name*, Item.Field.*name*));    Selects select=QB.*selectFrom*(join);  select.column(Person.Field.*id*).as("personid");  select.column(Item.Field.*name*).as("itemname");      List<Map<String,Object>> vars=*db*.select(join,**null**);  **for**(Map<String,Object> var:vars){  System.*out*.println(var);  //打印出 {itemname=张飞, personid=1}  }  } |

在上例中，我们可以发现，Query对象和Join对象都是JoinElement的子类，因此selectFrom方法对这两类对象都有效。

同样的，我们在之前单表查询中学习过的这些特性也都能够直接在Join查询中使用——

* Distinct
* Group by
* Having
* Max() min() avg() count（）等
* 函数的使用

这些都和在单表查询中没有什么区别。

### 分页和总数

同样的，分页和总数的实现方法和单表查询的API也是一样的，下面是和分页与总数相关的几个API的用法。虽然在单表查询中已经介绍过了。

**计算总数**

|  |  |
| --- | --- |
| 方法 | 作用 |
| **jef.database.Session.count(ConditionQuery)** | 将传入的查询请求（单表/多表）改写为count语句，然后求满足查询条件的记录总数。 |

**传入IntRange限制结果范围**

|  |  |
| --- | --- |
| **方法** | **用途说明** |
| **Session.select(ConditionQuery, IntRange)** | 传入Query形态的查询(单表、Union、Join均可)，限定返回条数在IntRange区间范围内。 |
| **Session.select(ConditionQuery, Class<T>, IntRange)** | 传入Query形态的查询(单表、Union、Join均可)，结果转换为指定类型，限定返回条数在IntRange区间范围内。 |
| **Session.iteratedSelect(TypedQuery<T>, IntRange)** | 传入Query形态的查询(单表/级联/Union)，限定返回条数在IntRange区间范围内。将游标封装为返回结果遍历器。 |
| **Session.iteratedSelect(ConditionQuery, IntRange)** | 传入Query形态的查询(单表、Union、Join均可)，限定返回条数在IntRange区间范围内。将游标封装为返回结果遍历器。 |
| **Session.iteratedSelect(ConditionQuery, Class<T>, IntRange)** | 传入Query形态的查询(单表、Union、Join均可)，限定返回条数在IntRange区间范围内。将游标封装为返回结果遍历器。 |

**使用pageSelect接口**，得到PageIterator对象。（参见4.1.3.2）

|  |  |
| --- | --- |
| **方法** | **作用** |
| **Session.pageSelect(ConditionQuery, int)** | 传入Query形态的查询(单表、Union、Join均可) |
| **Session.pageSelect(ConditionQuery, Class<T>, int)** | 传入Query形态的查询(单表、Union、Join均可)， 并指定返回结果类型 |

用例子来描述： 下面的例子里使用了两种方法来实现

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson6\Case1.java

|  |
| --- |
| **public** **void** testJoinWithPage()**throws** SQLException{  Query<Person> p = QB.*create*(Person.**class**);  Join join=QB.*innerJoin*(p, QB.*create*(Student.**class**), QB.*on*(Person.Field.*gender*, Student.Field.*gender*));  join.orderByDesc(Person.Field.*id*);  //方法1  {  **int** count=*db*.count(join);  List<Object[]> result=*db*.selectAs(join,Object[].**class**,**new** IntRange(4,8));//读取第4~第8条记录  System.*out*.println("总数:"+count);  **for**(Object[] objs:result){  System.*out*.println(Arrays.*toString*(objs));  }  }  //方法2  {  //使用分页查询，每页五条,从第四条开始读取  Page<Object[]> result=*db*.pageSelect(join, Object[].**class**,5).setOffset(3).getPageData();  System.*out*.println(result.getTotalCount());  **for**(Object[] objs:result.getList()){  System.*out*.println(Arrays.*toString*(objs));  }  }  } |

### 条件容器的问题

Join是由多个Query构成，而每个Query和Join对象本身都提供了addOrderBy等设置排序字段的方法。除此之外，每个Query对象都提供了addCondition方法，这产生了一个问题——即Join查询中的条件和排序应当设置到哪个对象容器中。如果我们在某个Query中设置了条件或者排序字段，然后再将其添加到一个Join对象中，还能正常生效吗？

从设计目的讲，目前无论是直接加入到Join对象上，还是任意Query对象上的条件，都会在查询中生效。

一种做法是，在构成Join的每个Query对象上添加条件和排序字段

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson6\Case1.java

|  |
| --- |
| @Test  **public** **void** testConditionAndOrder1()**throws** SQLException{  //两个Query对象，各自设置条件和顺序  Query<Person> p = QB.*create*(Person.**class**);  p.addCondition(Person.Field.*gender*, "M");  p.orderByAsc(Person.Field.*id*);    Query<Student> s=QB.*create*(Student.**class**);  s.addCondition(Student.Field.*dateOfBirth*,Operator.*IS\_NOT\_NULL*,**null**);  s.orderByDesc(Student.Field.*grade*);    Join join=QB.*innerJoin*(p,s , QB.*on*(Person.Field.*gender*, Student.Field.*gender*));  List<Map<String,Object>> result=*db*.select(join, **null**);  } |

另一种做法，在第一个Query上添加条件，在join中添加排序字段

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson6\Case1.java

|  |
| --- |
| **public** **void** testConditionAndOrder2()**throws** SQLException{  //把条件集中在第一个Query上。  Query<Person> p = QB.*create*(Person.**class**);  p.addCondition(Person.Field.*gender*, "M");  p.orderByAsc(Person.Field.*id*);  p.addCondition(Student.Field.*dateOfBirth*,Operator.*IS\_NOT\_NULL*,**null**);    Join join=QB.*innerJoin*(p, QB.*create*(Student.**class**) , QB.*on*(Person.Field.*gender*, Student.Field.*gender*));  //join上也可以直接设置排序字段。  p.orderByDesc(Student.Field.*grade*);    List<Map<String,Object>> result=*db*.select(join, **null**);  } |

上面两种代码实现，运行后可以发现最终效果是一样的。

也就是说：一般情况下，排序和Where条件可以放在Join的任意Query对象上。排序可以直接放在Join对象上。

为什么要限制“一般情况下”？我们需要了解一下在查询时实际发生了什么。我们先来看上面的例子所产生的实际SQL语句。

|  |
| --- |
| select T1.ID AS T1\_\_ID,  T1.GENDER AS T1\_\_GENDER,  T1.PERSON\_NAME AS T1\_\_NAME,  T1.CURRENT\_SCHOOL\_ID AS T1\_\_CURRENTSCHOOLID,  T2.GRADE AS T2\_\_GRADE,  T2.NAME AS T2\_\_NAME,  T2.ID AS T2\_\_ID,  T2.DATE\_OF\_BIRTH AS T2\_\_DATEOFBIRTH,  T2.GENDER AS T2\_\_GENDER  from T\_PERSON T1  inner join STUDENT T2 ON T1.GENDER = T2.GENDER  where T1.GENDER = ?  and T2.DATE\_OF\_BIRTH is not null  order by T1.ID ASC, T2.GRADE DESC |

首先，每个查询列（排序列）在查询语句中，都会添加该列所在表的别名（如T1.ID,中的T1为表T\_PEERSON的别名）。这是因为不同的表可能有相同名称的列，所以必须用表别名来限定。因此我们可以理解，每个条件中的Field对象最终是必须要绑定到一个Query上去的。如果我们在每个Query上设置条件，那么这个绑定是由开发人员显式完成的。如果将条件和排序设置到其他Query中或者Join对象中，那么绑定的过程是由框架内部完成的。

我们将由用户完成的绑定称为显式绑定，由框架内部自动完成的称为隐式绑定。隐式绑定是根据每个Field所在的Entity类型来判定的，很显然某种情况下隐式绑定是不准确的（也是危险的），那就是在同一张表多次参与Join的时候。

因此，如果在Join中同一张表参与多次，我们更建议显式的指定每个条件和排序对应的Query。否则的话两种方法都可以，可以按个人喜好使用。

## Union查询

多表查询中，除了Join以外，还有一种特殊的请求类型，即Union Query。

和SQL一样，多组查询（无论是Join的还是单表的， 只要选出的列的类型一样，就可以使用Union关系拼合成一个更大的结果集）。

UnionQuery具有特定的使用意义。我们之所以用数据库来union两个结果集，而不是在java代码中拼合两个结果集，最大的原因是——

* Union语句可以过滤重复行  
  使用union关键字，让数据库在拼合两个结果集时，去除完全相同的行。这会造成数据库排序等一系列复杂的操作，因此DBA们经常告诫开发人员，在能使用union all的场合就不要使用union。
* Union语句可以设置独立的排序条件  
  除了构成Union查询的子查询可以对结果排序外，Union/Union All语句还可以对汇总后的结果集进行排序，确保汇总后的结果集按特定的顺序列出。

因此，综上所述，我们在日常业务处理中，很难不使用union。

一个union查询的例子是这样的——

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson6\Case1.java

|  |
| --- |
| @Test  **public** **void** testUnion() **throws** SQLException {  JoinElement p = QB.*create*(Person.**class**);  p=QB.*innerJoin*(p, QB.*create*(School.**class**));    Selects select = QB.*selectFrom*(p);  select.clearSelectItems();  select.sqlExpression("upper(name) as name");  select.column(Person.Field.*gender*);  select.sqlExpression("'1'").as("grade");  select.column(School.Field.*name*).as("schoolName");    Query<Student> s=QB.*create*(Student.**class**);  select = QB.*selectFrom*(s);  select.column(Student.Field.*name*);  select.column(Student.Field.*gender*);  select.column(Student.Field.*grade*);  select.sqlExpression("'Unknown'").as("schoolName");    List<Map<String,Object>> result=*db*.select(QB.*union*(Map.**class**,p,s), **null**);  System.*out*.println(result);  } |

上例中，在两个对象的查询语句中，补齐了各自缺少的字段以后，两个查询具有相同的返回结果格式。此时就可以用union语句将两个查询结果合并为一个。(Derby的Union实现较为特殊，结果集之间是按列序号而不是列名进行匹配的，因此要注意列的输出顺序)

QueryBuilder可以用于生成union 查询。

在SQL语句中，有 union 和 union all 两种合并方式。对应到QueryBuilder中的以下几个方法上。

|  |  |
| --- | --- |
| 方法 | 作用 |
| QueryBuilder.union(TypedQuery<T>...) | 将多个Query用 union结合起来。 |
| QueryBuilder.union(Class<T>, ConditionQuery...) | 将多个Query用 union结合起来，查询的返回结果为指定的class |
| QueryBuilder.union(ITableMetadata, ConditionQuery...) | 将多个Query用 union结合起来，查询的返回结果为指定元模型对应的实体 |
| QueryBuilder.unionAll(TypedQuery<T>...) | 将多个Query用 union all结合起来。 |
| QueryBuilder.unionAll(Class<T>, ConditionQuery...) | 将多个Query用 union all结合起来，查询的返回结果为指定的class |
| QueryBuilder.unionAll(ITableMetadata, ConditionQuery...) | 将多个Query用 union all结合起来，查询的返回结果为指定元模型对应的实体 |

在使用union或unionAll方法时，需要传入一个类型，该类型为union查询最终要返回的结果容器。可以使用Map，也可以是任意对象。

关于排序

UnionQuery可以有单独的排序列，这个排序列属于union query独有，和各个Query对象的排序之间没有关系。这也是union语句的固有特点。再看下面的示例——

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson6\Case1.java

|  |
| --- |
| @Test  **public** **void** testUnion2() **throws** SQLException {  Query<Person> p = QB.*create*(Person.**class**);  p.orderByDesc(Person.Field.*currentSchoolId*);  Selects select = QB.*selectFrom*(p);  select.column(Person.Field.*name*).as("name");  select.column(Person.Field.*gender*);    Query<Student> s=QB.*create*(Student.**class**);  s.orderByAsc(Student.Field.*grade*);  select = QB.*selectFrom*(s);  select.column(Student.Field.*name*);  select.column(Student.Field.*gender*);  List<Student> result=*db*.select(QB.*union*(Student.**class**,p,s).orderByAsc(Student.Field.*name*));  **for**(Student st:result){  System.*out*.println(st.getName()+":"+st.getGender());  }  } |

执行上述代码，观察输出的SQL语句可以看到

|  |
| --- |
| (select t.PERSON\_NAME AS NAME,  t.GENDER  from T\_PERSON t  order by t.CURRENT\_SCHOOL\_ID DESC)  union  (select t.NAME,  t.GENDER  from STUDENT t  order by t.GRADE ASC) order by NAME ASC |

从SQL语句观察，可以发现各个子查询的排序列（Order By）和整个union语句的排序列其作用和定义是不同的。

## Exists条件和NotExists条件

我们还有可能会需要写带有exists条件的SQL语句。一个例子是——

Q: 查出所有姓名在Student表中出现过的Person。

A:

|  |
| --- |
| select t.\*  from T\_PERSON t  where exists (select 1 from STUDENT et where t.PERSON\_NAME = et.NAME) |

在这个例子中，exists带来了一个子查询，并且和父表发生了内连接。

EF-ORM中，也能生成这样的查询——

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson6\Case1.java

|  |
| --- |
| @Test  **public** **void** testExists() **throws** SQLException{  Query<Person> p=QB.*create*(Person.**class**);    //级联功能生效的情况下，查询依然是正确的。此处为了输出更简单的SQL语句暂时关闭级联功能。  //您可以尝试开启级联功能进行查询  p.setCascade(**false**);  p.addCondition(QB.*exists*(QB.*create*(Student.**class**),  QB.*on*(Person.Field.*name*, Student.Field.*name*)));  System.*out*.println(*db*.select(p));  } |

上面这个查询，将Exists作为一个特殊的条件进行处理。因此严格意义上面的查询语句是一个“单表查询”，因此级联功能依然会生效。为了使得生成的SQL更为简单，我们通过p.setCascade(**false**);语句关闭了级联功能。如果去掉这句语句，您将可以看到级联功能和exists共同生效的场景。

那么现在我们把问题变一下——

Q: 查出所有姓名未出现在Student表中的Person。

问题很简单，把exists改为not exists就可以了。

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson6\Case1.java

|  |
| --- |
| @Test  **public** **void** testNotExists() **throws** SQLException{  Query<Person> p=QB.*create*(Person.**class**);    p.addCondition(QB.*notExists*(QB.*create*(Student.**class**),  QB.*on*(Person.Field.*name*, Student.Field.*name*)));  System.*out*.println(*db*.select(p));  } |

您可能已经发现，多表查询下是不支持级联功能的。而上面的Exists和Not Exists中都出现了级联功能，所以实际上这两个例子都是单表查询。放在这一章仅仅是因为人们的习惯而已。

多表下的API查询就介绍到这里。关于多表查询下返回结果有哪些格式、用什么容器返回结果，本章没有介绍。因为EF-ORM无论在单表/多表/NativeQuery下，都有统一的查询结果返回规则。所以这部分内容将在学习完NativeQuery以后的第八章再介绍。

# 本地化查询

## 本地化查询是什么

这里本地化查询即’NativeQuery’，是指利用SQL（或者JPQL）进行的数据库操作——不仅限于select，也可以执行insert、update、甚至create table as、truncate等DDL。

本地化查询让用户能根据临时拼凑的或者预先写好的SQL语句进行数据库查询，查询结果将被转换为用户需要的类型。

在EF-ORM中，NativeQuery也有SQL和JPQL两种语法。其中JPQL是JPA规范定义的查询语言。但JPQL因为模型差距较大，一直没有完全支持，目前提供的名称为JPQL的若干方法仅为向下兼容而保留，不推荐大家使用。

因此**本地化查询就是用SQL语句操作数据库的方法**。

您可能会问，如果是用SQL，那么我们直接用JDBC就好了，还用ORM框架做什么？

事实上，NativeQuery中用的SQL，是被EF-ORM增强过的SQL，在语法特性上作了很多的补充。下面的表格中列出了所有在SQL上发生的增强，我们可以在下面的表格中查看到这些激动人心的功能。某种意义上讲，增强过的SQL是一种新的查询语言。我们也可以将其称为E-SQL (Enhanced SQL)。

E-SQL语法作了哪些改进呢？

|  |  |
| --- | --- |
| 特性 | 说明 |
| Schema重定向 | 在Oracle,PG等数据库下，我们可以跨Schema操作。Oracle数据库会为每个用户启用独立Schema，在SQL语句或对象建模中，我们可以指定Entity所属的Schema。 但是实际部署时schema名称如果发生变化，事先写好的程序就不能正常工作。  schema重定向不仅仅应用于CriteriaAPI中，在SQL语句中出现的Schema也会在改写过程中被替换为当前实际的Schema。 |
| 数据库方言： 语法格式整理 | 对于SQL语句进行重写，将其表现为适应本地数据库的写法，比如 || 运算符的使用。  又比如 as 关键字的使用。 |
| 数据库方言： 函数转换 | 对于translate、delete、sysdate、decode、nullif等数据库函数，能自动转换为当前数据库能够支持的表达形式。 |
| 增强的绑定变量 | 1. 允许在语句中用占位符来描述变量。绑定变量占位符可以用 :名称 也可以用 ?序号 的形式。即JPQL语法的占位符。 2. 绑定变量占位符中可以指定变量数据类型。这样，当传入String类型的参数时，能自动将其转换为绑定变量应当使用的类型，如java.sql.Date,、java.lang.Number等。 |
| 动态SQL功能： 自动省略 | 在SQL中，会自动扫描每个表达式的入参（占位符）是否被用到。如果某个参数未使用，那么该参数所在的表达式会被省略。例如  *select \* from t where id=:id and name=:name*  中，如果name参数未传入，则and后面的整个条件表达式被略去。 |
| 动态SQL功能： SQL片段参数 | 在SQL占位符中可以声明一个占位符是SQL片段。这个片段可以在运行时根据传入的SQL参数自动应用。 |

表 7-1 E-SQL的语法特性

EF-ORM会对用户输入的SQL进行解析，改写，从而使得SQL语句的使用更加方便，EF-ORM将不同数据库DBMS下的SQL语句写法进行了兼容处理。 并且提供给上层统一的SQL写法。

除了在SQL语法上的增强以外，通过EF-ORM NativeQuery操作和直接操作JDBC相比，还有以下优势：

|  |  |  |
| --- | --- | --- |
| **特点** | **NativeQuery** | **JDBC** |
| **对象返回** | 转换为需要的对象。转换规则和Criteria API一致。 | ResultSet对象 |
| **自定义返回对象转换映射** | 可以自定义ResultSet中字段返回的Mapping关系。 | -- |
| **性能调优** | 可以指定fetch-size , max-result 等参数，进行性能调优 | JDBC提供各种性能调优的参数 |
| **复用** | 一个NativeQuery可携带不同的绑定变量参数值，反复多次使用。 | PreparedStatment对象可以反复使用 |
| **一级缓存** | 会刷新和维护一级缓存中的数据。  比如用API插入一个对象，一级缓存中即缓存了这个对象。  虽然用SQL语句去update这条记录。一级缓存中的该对象会被自动刷新。 | 无此功能 |
| **SQL自动选择** | SQL改写功能不能解决一切跨库移植问题。用户可以对不兼容跨库的SQL写成多个版本，运行时自动选择。 | 无此功能 |
| **性能** | SQL解析和改写需要花费0.3~0.6毫秒。其他操作基本和JDBC直接操作保持一致。  对象结果转换会额外花费一点时间，但采用了策略模式和ASM无反射框架，性能优于大多数同类框架。 | 原生方式，性能最佳 |

表 7-2 使用NativeQuery和直接使用JDBC的区别

|  |
| --- |
| *关于JPQL支持*  *EF-ORM中，也可以用”JPQL“来构造NativeQuery，但并不推荐。因为EF并未实现JPQL的大部分功能。目前提供的JPQL功能其实只有将Java字段名替换为数据库列名的功能，离JPA规范的JPQL差距较大，而且由于设计理念等差异，要完整支持JPQL基本不可能。*  *现有若干伪JPQL功能是早期遗留的产物，后来在对SQL的特性作了大量改进后，E-SQL成为EF-ORM主要的查询语言。JPQL方面暂无改进计划，因此不建议使用。* |

## 使用本地化查询

### NamedQuery和NativeQuery

NativeQuery的用法可以分为两类。一类是在java代码中直接传入E-SQL语句的；另外一类是事先将E-SQL编写在配置文件或者数据库中，运行时加载并解析，使用时按名称进行调用。这类SQL查询被称为NamedQuery。对应JPA规范当中的“命名查询”。

命名查询也就是Named-Query，在某H框架和JPA中都有相关的功能定义。简单来说，命名查询就是将查询语句(SQL,HQL,JPQL等)事先编写好， 然后为其指定一个名称。在使用ORM框架时，取出事先解析好的查询，向其中填入绑定变量的参数，形成完整的查询。

EF-ORM的命名查询和OpenJPA以及某H框架中的命名查询用法稍有些不同。

* 命名查询默认定义在配置文件 named-queries.xml中。不支持使用Annotation等方法定义
* 命名查询也可以定义在数据库表中，数据库表的名称可由用户配置
* 命名查询可以支持 E-SQL和JPQL两种语法（后者特性未全部实现）
* 由于支持E-SQL，命名查询可以实现动态SQL语句的功能，可以模拟出与IBatis相似的用法。

为什么不使用JPA规范中的基于Annotation的方式来注册命名查询呢？因为考虑到ORM中一般只有跨表的复杂查询才会使用命名查询。而将一个多表的复杂查询注解在任何一个DAO上都是不合适的。分别注解在DAO上的SQL语句除了语法受限之外，还有以下缺点：

* 归属不明确，很难正确评判某个SQL语句应当属于某个DAO。而且不能被其他DAO使用？
* Java代码中写SQL涉及转义问题
* DAO太分散，不利于SQL语句的统一维护。

EF-ORM默认设计了两种方式来配置命名查询

* classpath下创建一个名为named-queries.xml的配置文件
* 存放在数据库中，表名可自定义，默认JEF\_NAMED\_QUERIES

NativeQuery是在EF-ORM 1.05开始增加的功能。在1.6开始支持数据库配置，在1.6.7开始支持动态改写和SQL片段。

### API和用法

我们分别来看Named-Query和Native Query的使用

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson7\Case1.java

|  |
| --- |
| @Test  **public** **void** testNativeQuery() {  // 方法1 NamedQuery  {  NativeQuery<ResultWrapper> query = *db*.createNamedQuery("unionQuery-1", ResultWrapper.**class**);  List<ResultWrapper> result = query.getResultList();  System.*out*.println(result);  }  // 方法2 直接传入SQL  {  String sql = "select \* from((select upper(t1.person\_name) AS name, T1.gender, '1' AS GRADE,"  + "T2.NAME AS SCHOOLNAME from T\_PERSON T1 inner join SCHOOL T2 ON T1.CURRENT\_SCHOOL\_ID=T2.ID"  + ") union ( select t.NAME,t.GENDER,t.GRADE,'Unknown' AS SCHOOLNAME from STUDENT t )) a";  NativeQuery<ResultWrapper> query = *db*.createNativeQuery(sql, ResultWrapper.**class**);  List<ResultWrapper> result = query.getResultList();  System.*out*.println(result);  }  } |

上面的例子中，两次查询使用的SQL语句是一样的，区别在于前者配置在named-queries.xml中，后者直接写在代码中。

|  |
| --- |
| *实践建议：*  *EF-ORM的SQL语句已经解决了动态化的问题（这点可在以后的例子中发现），在我看来，在代码中编写SQL语句带来的灵活性，远不如拼凑SQL带来的可读性和可维护性上的损失来得大。因此建议在使用时，尽可能多的使用命名查询，而少用拼凑SQL的查询。*  *上面的方法一相比方法二还有一个性能上的优势。EF-ORM会对传入的SQL进行解析和重写，如果是命名查询，解析只进行一次，解析结果会缓存下来，而传入的SQL语句则必须每次解析和重写。* |

在上面的例子中，无论是createNamedQuery方法还是createNativeQuery方法，返回的对象都是名为NativeQuyery的对象。其行为和功能也完全一样。

下面是用于获得NativeQuery的方法API

|  |  |
| --- | --- |
| 方法 | 用途 |
| Session.createNamedQuery(String) | 构造一个命名查询对象。不指定返回类型。 |
| Session.createNamedQuery(String, Class<T>) | 构造一个命名查询对象。指定返回类型为某class。 |
| Session.createNamedQuery(String, ITableMetadata) | 构造一个命名查询对象。指定返回类型为某表元模型所对应的类。 |
| Session.createNativeQuery(String) | 构造一个SQL查询对象。不指定返回类型。 |
| Session.createNativeQuery(String, Class<T>) | 构造一个SQL查询对象。指定返回类型为某class。 |
| Session.createNativeQuery(String, ITableMetadata) | 构造一个SQL查询对象。指定返回类型为某表元模型所对应的类。 |
| Session.createQuery(String) | 构造一个JPQL查询对象。不指定返回类型。 |
| Session.createQuery(String,Class<T>) | 构造一个JPQL查询对象。指定返回类型为某class。 |

上述方法都可以返回NativeQuery对象。

NativeQuery并不一定就是select语句。在NativeQuery中完全可以使用update delete insert等语句，甚至是create table等DDL语句。（当执行DDL时会造成事务被提交，需谨慎）。显然，在指定非Select操作时，传入一个返回结果类型是多此一举，可以指定NativeQuery返回的结果类型，也可以不指定。

在得到了NativeQuery以后，我们有很多种用法去使用这个Query对象。这里先列举一下主要的方法。

|  |  |
| --- | --- |
| 方法 | 用途 |
| ***执行查询动作*** | |
| NativeQuery.getResultList() | 查询出由多行记录转换的对象列表 |
| NativeQuery.getSingleResult() | 查询出单行记录转换的对象，如果有多行返回第一行 |
| NativeQuery.getResultCount() | 将SQL改为count语句后查出结果总数 |
| NativeQuery.getResultIterator() | 查询出多行记录，并用遍历器方式返回 |
| NativeQuery.executeUpdate() | 执行SQL语句，返回影响的记录行数 |
| NativeQuery.getSingleOnlyResult() | 和getSingleResult的区别是如果有多行则抛出异常 |
| ***性能参数*** | |
| NativeQuery.setMaxResults(int) | 设置查询最大返回结果数 |
| NativeQuery.getMaxResults() | 获取查询最大返回结果数 |
| NativeQuery.getFetchSize() | 获取结果集加载批次大小 |
| NativeQuery.setFetchSize(int) | 设置结果集加载批次大小 |
| ***结果范围限制（分页相关）*** | |
| NativeQuery.setFirstResult(int) | 设置查询结果的偏移，0表示不跳过记录。 |
| NativeQuery.getFirstResult() | 返回查询结果的偏移，0表示不跳过记录 |
| NativeQuery.setRange(IntRange) | 设置查询区间（含头含尾） |
| ***绑定变量参数*** | |
| NativeQuery.setParameter(String, Object) | 设置绑定变量参数 |
| NativeQuery.setParameter(int, Object) | 设置绑定变量参数 |
| NativeQuery.setParameterByString(String, String) | 设置绑定变量参数，传入String后根据变量类型自动转换 |
| NativeQuery.setParameterByString(String, String[]) | 设置绑定变量参数，传入String[]后根据变量类型自动转换 |
| NativeQuery.setParameterByString(int, String) | 设置绑定变量参数，传入String后根据变量类型自动转换 |
| NativeQuery.setParameterByString(int, String[]) | 设置绑定变量参数，传入String[]后根据变量类型自动转换 |
| NativeQuery.setParameterMap(Map<String, Object>) | 设置多组绑定变量参数 |
| NativeQuery.getParameterValue(String) | 获得绑定变量参数 |
| NativeQuery.getParameterValue(int) | 获得绑定变量参数 |
| NativeQuery.containsParam(Object) | 检查某个绑定变量是否已经设置了参数 |
| NativeQuery.clearParameters() | 清除目前已经设入的绑定变量参数。当需要重复使用一个NativeQuery对象进行多次查询时，建议每次清空旧参数。 |
| NativeQuery.getParameterNames() | 获得查询中所有的绑定变量参数名 |
| ***返回结果定义*** |  |
| NativeQuery.getResultTransformer() | 得到ResultTransformer对象，可定义返回结果转换动作。 |

根据上述API，我们简单的使用如下——

|  |
| --- |
| @Test  **public** **void** testQueryParams(){  String sql="select distinct(select grade from student s where s.name=person\_name) grade,person\_name,gender from t\_person where id<:id";  NativeQuery<Map> query = *db*.createNativeQuery(sql,Map.**class**);  query.setParameter("id", 12);  //自动改写为count语句进行查询  System.*out*.println("预计查出"+query.getResultCount()+"条结果");  //查询多条结果  System.*out*.println(query.getResultList());    //重新设置参数  System.*out*.println("=== 重新设置参数 ===");  query.setParameter("id", 2);  System.*out*.println("预计查出"+query.getResultCount()+"条结果");  System.*out*.println(query.getResultList());  //查出第一条结果  System.*out*.println(query.getSingleOnlyResult());  } |

注意观察输出的SQL语句，上面的案例中，演示了

1. 绑定变量参数用法
2. Count语句在一些复杂情况下的转换逻辑
3. 通过重置参数，可以复用NativeQuery对象。
4. 仅返回单条结果的场景

### 命名查询的配置

上一节中，我们基本了解了NativeQuery对象的构造和使用。本节来介绍命名查询如何配置。

前面说过，命名查询的配置方法有两种。我们先来看配置在文件中的场景。

#### 配置在named-queries.xml中

在classpath下创建一个名为named-queries.xml的文件。

named-queries.xml

|  |
| --- |
| <queries>  <query name = *"getUserById"*>  <![CDATA[  select \* from t\_person where person\_name=:name<string>  ]]>  </query>  <query name = *"testIn"* type=*"sql"* fetch-size=*"100"* >  <![CDATA[  select count(\*) from Person\_table where id in (:names<int>)  ]]>  </query>  </queries> |

上面就配置了两个命名查询，名称分别为“getUserById”以及”testIn”。

其中每个查询的SQL中，都有一个参数，参数在SQL中用绑定变量占位符表示。后面的参数使用in条件，使用时可以传入int数组。

在query元素中，可以设置以下属性

|  |  |  |
| --- | --- | --- |
| 属性名 | 作用 | 备注 |
| name | 指定查询的名称 | 必填 |
| type | sql或jpql，表示语句的类型 | 可选，默认为SQL |
| fetch-size | 指定结果集每次获取批次大小 | 可选，默认0即JDBC驱动默认值 |
| tag | 当DbClient连接到多数据源时，可以指定该查询默认使用哪个数据源连接 | 可选 |
| remark | 备注信息，可不写 | 可选 |

最后，当classpath下有多个named-queries.xml时，所有配置均会生效。如果多个文件中配置的同名的查询，那么后面加载的会覆盖前面的。当覆盖现象发生时，日志中会输出警告。

#### 配置在数据库中

你可以将命名查询配置在指定的数据库表中。要启用此功能，需要在jef.properties配置

|  |
| --- |
| db.query.table.name=表名 |

当EF-ORM初始化时，会自动检测这张表并加载数据，如果没有则会自动创建。表的结构是固定的(数据类型在不同的数据库上会自动转换为可支持的类型)

命名查询数据表的结构是固定的，结构如下——

|  |  |  |
| --- | --- | --- |
| **Column** | **Type** | **备注** |
| NAME | varchar2(256) not null primary key | 指定查询的名称 |
| SQL\_TEXT | varchar2(4000) | SQL/JPQL语句 |
| TAG | varchar2(256) | 当DbClient连接到多数据源时，可以指定该查询默认使用哪个数据源连接 |
| TYPE | number(1) default 0 | 语句类型。0:SQL 1:JPQL |
| FETCH\_SIZE | number(6) default 0 | 指定结果集每次获取批次大小 |
| REMARK | varchar2(256) | 备注 |

表7-3 命名查询的数据库表结构

您可以同时使用文件配置和数据库配置命名查询。但如果出现同名的查询，数据库的配置会覆盖文件的配置。

#### 数据源绑定

由于EF-ORM是一种支持多数据源自动路由的ORM框架。因此在命名查询中，还可以在tag属性中指定偏好的数据源ID。这样，如果你在query://getUserByName这样的例子中请求数据时，即使不通过\_dsname参数来指定数据源id，也可以到正确的数据库中查询数据。

我们先配置两个命名查询，区别是一个指定了数据源，另一个未指定数据源。

orm-tutorial\src\main\resources\named-queries.xml

|  |
| --- |
| <query name=*"getUserById"* tag=*"datasource2"*>  <![CDATA[  select \* from t\_person where person\_name=:name<string>  ]]>  </query>  <query name=*"getUserById-not-bind-ds"*>  <![CDATA[  select \* from t\_person where person\_name=:name<string>  ]]>  </query> |

然后编写代码如下——

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson7\Case2.java

|  |
| --- |
| @Test(expected=PersistenceException.**class**)  **public** **void** testDataSourceBind() **throws** SQLException{  //Person对象所在的数据源为DataSource2  //由于配置中指定默认数据源为datasource2，因此可以正常查出  List<Person> persons=*db*.createNamedQuery("getUserById",Person.**class**)  .setParameter("name", "张三").getResultList();  System.*out*.println(persons);    //这个配置未指定绑定的数据源，因此会抛出异常  **try**{  List<Person> p2=*db*.createNamedQuery("getUserById-not-bind-ds",Person.**class**)  .setParameter("name", "张三").getResultList();  }**catch**(RuntimeException e){  e.printStackTrace();  **throw** e;  }  } |

从日志上可以观察到，第一次操作时，操作是在datasource2数据库上执行的，因此能正确查出。第二次操作时，操作在datasource1上执行，由于表不存在，因此抛出异常。

#### 动态更新命名查询配置

考虑到性能因素，NamedQuery在首次使用时从配置中加载SQL语句并解析，之后解析结果就缓存在内存中了。作为服务器运行来说，这一点没什么问题。但在Web应用开发中，用户如果修改了SQL语句，不得不重启服务才能调试。这就会给开发者造成一定困扰。为了解决配置刷新的问题，框架中增加了配置变更检测的功能。

在jef.properties中配置

|  |
| --- |
| db.named.query.update=true |

开启更新检测功能后，每次获取命名查询都会检查配置文件是否修改，如果配置来自于数据库，也会加载数据库中的配置。在开发环境可以开启这个参数；考虑到性能，在生产环境建议关闭该参数。

当大量并发请求使用命名查询时，为了避免短时间重复检查更新，一旦检测过一次，10秒内不会再次检测。

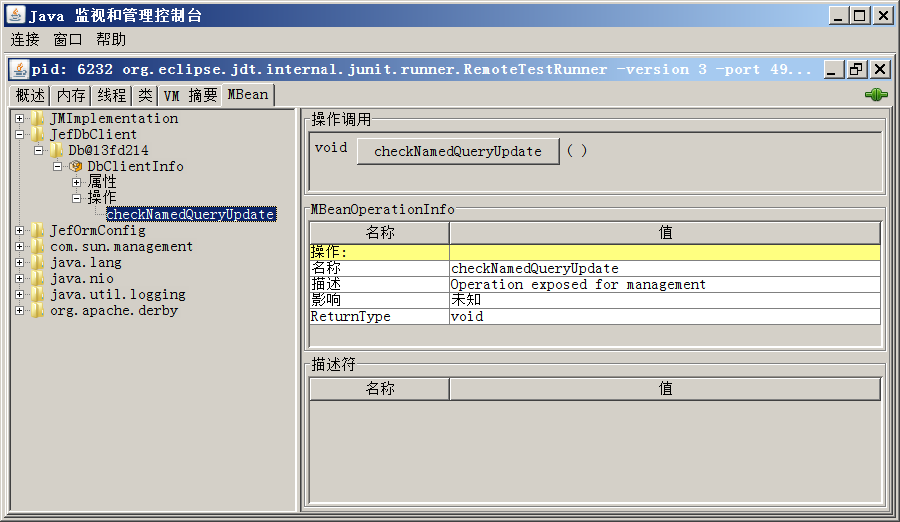
一般情况下，我们无需配置该参数。因为该参数的默认值会保持和 db.debug 一致，一般来说开发时我们肯定会设置db.debug=true。而在对性能要求较高的生产环境，需要指定该参数为false。

命名查询自动更新的配置项可以在JMX的ORMConfigMBean中开启或关闭。JMX相关介绍请参阅13章。

除了自动检测更新机制外，还有手动刷新命名查询配置的功能。比如在生产环境，关闭自动检测配置更新功能后，可以手工进行更新检测。在DbClient中可以强制立刻检查命名查询的更新。

|  |
| --- |
| DbClient.checkNamedQueryUpdate() |

强制检测命名查询功能也可以在JMX的Bean中调用。DbClientInfoMBean中有checkNamedQueryUpdate()方法。



## NativeQuery特性使用

### Schema重定向

Schema重定向多使用在Oracle数据库上。在Oracle上，数据库操作可以跨用户(Schema)访问。当跨Schema访问时，SQL语句中会带有用户名的前缀。（这样的应用虽然不多，但是在电信和金融系统中还是经常看到）。

例如USERA用户下和USERB用户下都有一张名为TT的表。 我们可以在一个SQL语句中访问两个用户下的表

|  |
| --- |
| select \* from USERA.TT  union all  select \* from USERB.TT |

当使用ORM进行此类映射时，一般用如下方式指定

|  |
| --- |
| @Entity  @Table(schema="USERA",name="TT")  **public** **class** TT {  ....  } |

这样就带来一个问题，在某些场合，实际部署的数据库用户是未定的，在编程时开发人员无法确定今后系统将会以什么用户部署。如果将“USERA”硬编码到程序中，实际部署时数据库就只能建在USERA用户下，部署时缺乏灵活性。

EF-ORM的Schema重定向功能对Query模型和SQL语句都有效。在开发时，用户根据设计中的虚拟Schema名编写代码，而在实际部署时，可以配置文件jef.properties指定虚拟schema对应到真实环境中的schema上。

例如，在jef.properties中配置

|  |
| --- |
| schema.mapping=USERA:APP, USERB:APP |

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson7\Case1.java

|  |
| --- |
| /\*  \* SQL语句中的usera userb都是不存在的schema，通过jef.properties中的配置，被重定向到APP schema下  \* **@throws** SQLException  \*/  @Test  **public** **void** testSchemaMapping() **throws** SQLException{  String sql="select \* from usera.t\_person union all select \* from userb.t\_person";  *db*.createNativeQuery(sql).getResultList();  } |

上例中的SQL语句本来是无法执行的，但是因为Schema重定向功能SQL语句在实际执行时，就变为

|  |
| --- |
| select \* from app.t\_person  union all  select \* from app.t\_person |

这样就能正常执行了。

使用schema重定向功能，可以解决开发和部署的 schema耦合问题，为测试、部署等带来更大的灵活性。

一个特殊的配置方式是

|  |
| --- |
| schema.mapping=USERA:, USERB: |

即不配置重定向后的schema，这个操作实际上会将原先制定的schema信息去除，相当于使用数据库连接的当前Schema。

最后，正如前文所述，重定向功能并不仅仅作用于本地化查询中，如果是在类的注解上配置了Schema，那么其映射会在所有Criteria API查询中也都会生效。

### 数据库方言——语法格式整理

根据不同的数据库语法，EF-ORM会在执行SQL语句前根据本地方言对SQL进行修改，以适应当前数据库的需要。

**例1**

|  |
| --- |
| select t.id||t.name as u from t |

在本例中||表示字符串相连，这在大部分数据库上执行都没有问题，但是如果在MySQL上执行就不行了，MySQL中||表示或关系，不表示字符串相加。 因此，EF-ORM在MySQL上执行上述E-SQL语句时，实际在数据库上执行的语句变为

|  |
| --- |
| select concat(t.id, t.name) as u from t //for MySQL |

这保证了SQL语句按大多数人的习惯在MYSQL上正常使用。

**例2**

|  |
| --- |
| select count(\*) total from t |

这句SQL语句在Oracle上是能正常运行的，但是在postgresql上就不行了。因为postgresql要求每个列的别名前都有as关键字。对于这种情况EF-ORM会自动为这样的SQL语句加上缺少的as关键字，从而保证SQL语句在Postgres上也能正常执行。

|  |
| --- |
| select count(\*) as total from t //for Postgresql |

上述修改过程是全自动的，无需人工干涉。EF-ORM会为所有传入本地化查询进行语法修正，以适应当前操作的数据库。

这些功能提高了SQL语句的兼容性，能对用户屏蔽数据库方言的差异，避免操作者因为使用了SQL而遇到数据库难以迁移的情况。

我们看一下orm-tutorial中的例子——

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson7\Case1.java

|  |
| --- |
| /\*\*  \* concat(person\_name, gender) 在实际使用时会改写为 person\_name||gender  \*/  @Test  **public** **void** testRewrite1() **throws** SQLException{  String sql="select concat(person\_name, gender) from usera.t\_person";  *db*.createNativeQuery(sql).getResultList();  } |

众所周知，Derby数据库中无法使用concat(a,b)的函数。所以经过方言转换会将其表示为

|  |
| --- |
| select person\_name||gender from app.t\_person |

**注意**

并不是所有情况都能实现自动改写SQL，比如有些Oracle的使用者喜欢用+号来表示外连接，写成仅有Oracle能识别的外连接格式。

|  |
| --- |
| select t1.\*,t2.\* from t1,t2 where t1.id=t2.id(+) |

目前EF-ORM还**不支持**将这种SQL语句改写为其他数据库支持的语法(今后可能会支持)。 因此如果要编写能跨数据库的SQL语句，还是要使用‘OUTER JOIN’这样标准的SQL语法。

### 数据库方言——函数转换

#### 示例

在EF-ORM对SQL的解析和改写过程中，还能处理SQL语句当中的数据库函数问题。EF-ORM在为每个数据库建立的方言当中，都指定了常用函数的支持方式。在解析时，EF-ORM能够自动识别SQL语句中的函数，并将其转换为在当前数据库上能够使用的函数。

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson7\Case1.java

|  |
| --- |
| @Test  **public** **void** testRewrite2() **throws** SQLException{  String sql="select replace(person\_name,'张','王') person\_name,decode(nvl(gender,'M'),'M','男','女') gender from t\_person";  System.*out*.println(*db*.createNativeQuery(sql).getResultList());  } |

我们观察一下在Derby数据库时输出的实际SQL语句

|  |
| --- |
| select replace(person\_name, '张', '王') AS person\_name,  CASE  WHEN coalesce(gender, 'M') = 'M'  THEN '男'  ELSE '女'  END AS gender  from t\_person |

解说一下，上面作了处理的函数包括——

nvl函数被转换为coalesce函数

Decode函数被转换为 case ... When ... Then ... Else...语句。

Replace函数也是很特殊的——Derby本没有Replace函数，这里的replace函数其实是一个用户自定义的java函数。也是由EF-ORM自动注入的自定义函数。

再看一个例子

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson7\Case1.java

|  |
| --- |
| @Test  **public** **void** testRewrite3() **throws** SQLException{  //获得：当前日期减去1个月，和学生生日相差的天数。  String sql="select datediff(add\_months(sysdate, -1), DATE\_OF\_BIRTH),DATE\_OF\_BIRTH from student";  System.*out*.println(*db*.createNativeQuery(sql).getResultList());    //获得：在当前日期上加上1年  sql="select addDate(sysdate, INTERVAL 1 YEAR) from student";  System.*out*.println(*db*.createNativeQuery(sql).getResultList());  } |

上例涉及日期时间的计算，最终结果是一个日期天数。执行上面的代码，可以看到SQL语句变为

|  |
| --- |
| select {fn timestampdiff(SQL\_TSI\_DAY, DATE\_OF\_BIRTH, {fn timestampadd(SQL\_TSI\_MONTH, -1, current\_timestamp) }) },  DATE\_OF\_BIRTH  from student  select {fn timestampadd(SQL\_TSI\_YEAR, 1, current\_timestamp) } from student |

其中——

datediff函数被转换为JDBC函数 timestampdiff

add\_months和adddate函数被转换为JDBC函数timestampadd

Sysdate函数被转换为current\_timestamp。

通过上面的转换过程，EF-ORM尽最大努力的保证查询的跨数据库兼容性。

#### 函数支持

EF-ORM对函数支持的原则是，尽可能将常用函数提取出来，并保证其能在任何数据库上使用。

目前整理定义的常用函数被定义在两个枚举类中，其用法含义如下表所示

jef.database.query.Func

|  |  |  |
| --- | --- | --- |
| 函数 | 作用 | 参数格式 |
| abs | 单参数。获取数值的绝对值。 | (number) |
| add\_months | 双参数。在日期上增加月数。（来自于Oracle）。 | (date, number) |
| adddate | 在日期上增加天数，adddate(current\_date, 1)即返回明天  支持Interval语法。Adddate(current\_date, interval 1 month)即返回下月当天  (来自于MYSQL,和DATE\_ADD含义相同） | (date, numer)  (date, interval) |
| avg | 返回分组中数据的平均值 |  |
| cast | 类型转换函数，格式为 cast(arg as varchar) cast(arg as timestamp)等等 |  |
| ceil | 单参数。按大数取整 | (float) |
| coalesce | 多参数。可以在括号中写多个参数，返回参数中第一个非null值 |  |
| concat | 多参数。连接多个字符串，虽然大部分数据库都用 || 表示两个字符串拼接，但是MYSQL是必须用concat函数的 |  |
| count | 计算行数 |  |
| current\_date | 无参数。返回当前日期 |  |
| current\_time | 无参数。返回当前不含日期的时间 |  |
| current\_timestamp | 无参数。返回当前日期和时间 |  |
| date | 单参数。截取年月日 | (timestamp) |
| datediff | 返回第一个日期减去第二个日期相差的天数。 | (date,date) |
| day | 获得日，如 day(current\_date)返回当天是几号。 | (date) |
| decode | 类似于多个if分支。 (来自于Oracle) |  |
| floor | 按较小的整数取整 |  |
| hour | 获得小时数 |  |
| length | 文字长度计算。基于字符个数（英文和汉字都算1） | (varchar) |
| lengthb | 文字长度按实际存储大小计算，因此在UTF8编码的数据库中，汉字实际占3个字符。 | (varchar) |
| locate | 双参数,在参数2中查找参数1，返回匹配的序号（序号从1开始）。 | (varchar,varchar) |
| lower | 字符串转小写 | (varchar) |
| lpad | 在字符串或数字左侧添加字符，拼到指定的长度。 | (expr, varchar, int) |
| ltrim | 截去字符串左边的空格 | (varchar) |
| max | 返回分组中的最大值 |  |
| min | 返回分组中的最小值 |  |
| minute | 获得分钟数 |  |
| mod | 取模运算。前一个参数除以后一个时的余数。 | (int, int) |
| month | 获得月份数 |  |
| now | 无参数，和CURRENT\_TIMESTAMP含义相同。 |  |
| nullif | 双参数，nullif ( L, R )。两值相等返回null，否则返回第一个参数值。 | (expr,expr) |
| nvl | 双参数，返回第一个非null值 (来自于Oracle) |  |
| replace | 三参数，查找并替换字符 replace(sourceStr, searchStr, replacement) | (varchar,varchar,varchar) |
| round | 四舍五入取整 | (float) |
| rpad | 在字符串或数字右侧添加字符，拼到指定的长度。 | (expr, varchar, int) |
| rtrim | 单参数，截去字符串右边的空格 (来自于Oracle) | (varchar) |
| second | 单参数，获得秒数 | (timestamp) |
| sign | 单参数，判断数值符号。正数返回1，负数返回-1，零返回0 | (number) |
| str | 但参数，将各种类型变量转为字符串类型 (来自HQL) |  |
| subdate | 双参数，在日期上减去天数，支持Interval语法。 (来自于MYSQL,和DATE\_SUB含义相同) |  |
| substring | 取字符的子串 (第一个字符号为1) ，例如 substring('abcde',2,3)='bcd' | (source, startIndex, length) |
| sum | 返回分组中的数据的总和 |  |
| time | 单参数，截取时分秒 |  |
| timestampadd | 时间调整 第一个参数是时间调整单位，第二个参数是调整数值，第三个参数是日期时间表达式。例如 timestampadd( MINUTE, 10, current\_timestamp)  （来自MySQL，同时是JDBC函数） | (SQL\_TSI, number, timestamp) |
| timestampdiff | 返回两个时间差值多少，第一个参数为返回差值的单位取值范围是SQL\_TSI的枚举，后两个参数是日期1和日期2，返回日期2减去日期1（注意和datediff刚好相反）  例如Timestampdiff(MINUTE, date1, date2)返回两个时间相差的分钟数。 （来自MySQL，同时是JDBC函数） | (SQL\_TSI, time1, time2) |
| translate | 三参数，针对单个字符的批量查找替换。Translate(‘Hello,World’,’eo’,’oe’). 将字符中的e替换为o，o替换为e。 (来自于Oracle)。  注意：在部分数据库上使用多个replace语句来模拟效果，但由于模拟相当于执行多次函数，因此如果先替换字符列表中出现重复替换现象，结果可能和Oracle不一致。 | (varchar,varchar, varchar) |
| trim | 截去字符串两头的空格。 | (varchar) |
| trunc | 适用于数字，保留其小数点后的指定位数。在Oracle上trunc还可用于截断日期。此处不支持。 | (float) |
| upper | 字符串转大写 | (varchar) |
| year | 获得年份数 | (timestamp) |

jef.database.query.Scientific

|  |  |
| --- | --- |
| 函数 | 作用 |
| acos | 反余弦函数 |
| asin | 反正弦函数 |
| atan | 反正切函数 |
| cos | 余弦函数 |
| cosh | 双曲余弦函数 |
| cot | 余切函数 |
| degrees | 弧度转角度 即value/3.1415926\*180 |
| exp | 返回自然对数的底(e)的n次方。 |
| ln | 自然对数等同于log |
| log10 | 以10为底的对数 |
| power | 乘方 |
| radians | 角度转弧度 即value/180\*3.1415926 |
| rand | 随机数，返回0..1之间的浮点随机数. |
| sin | 正弦函数 |
| sinh | 双曲正弦函数 |
| soundex | 字符串发音特征。不是科学计算，但只对英语有用，对汉语几乎没有作用。 |
| sqrt | 平方根 |
| tan | 正切函数 |
| tanh | 双曲正切函数 |

常用函数中，某些函数并非来自于任何数据库的SQL函数中，而是EF-ORM定义的，比如str函数。

除了上述被提取的常用函数外，数据库原生的各种函数和用户自定义的函数仍然能够被使用。但是EF-ORM无法保证包含这些函数的SQL语句被移植到别的RDBMS后还能继续使用。

#### 方言扩展

函数的支持和改写规则定义是通过各个数据库方言来定义的。因此，要支持更多的函数，以及现有的一些不兼容的场景，可以通过扩展方言来实现。

方言扩展的方法是配置自定义的方言类。在jef.properties中，我们可以指定自定义方言配置文件，来覆盖EF-ORM内置的方言。

jef.properties

|  |
| --- |
| #the custom dialect mapping file  db.dialect.config=my\_dialects.properties |

上面定义了自定义方言映射文件的名称是my\_dialacts.properties，然后在my\_dialacts.properties中配置自定义的方言映射。

在见示例工程目录：orm-tutorial\src\main\resources\my\_dialects.properties

|  |
| --- |
| derby=org.easyframe.tutorial.lessonc.DerbyExtendDialect |

文件中，前面是要定义的数据库类型，后面是方言类。

我们编写的自定义方言类如下。

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson7\dialect\DerbyExtendDialect.java

|  |
| --- |
| **import** jef.database.dialect.DerbyDialect;  **public** **class** DerbyExtendDialect **extends** DerbyDialect{  **public** DerbyExtendDialect() {  **super**();  //编写自己的方言逻辑  //....  }  //覆盖父类方法  } |

自定义方言可以控制数据库各种本地化行为，包括分页实现方式、数据类型等。这些实现可以参考EF-ORM内置的方言代码。

**示例1：让Derby支持反三角函数TAN2**

DERBY数据库支持反三角函数TAN2函数。但是因为方言中没有注册这个函数，因此我们在E-SQL中是无法使用的。

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson7\CaseDialectExtend.java

|  |
| --- |
| @Test  **public** **void** testExtendDialact() **throws** SQLException{  String sql="select atan2(12, 1) from t\_person";  System.*out*.println(*db*.createNativeQuery(sql).getResultList());  ……  } |

上面的代码会抛出异常信息

|  |
| --- |
| java.lang.IllegalArgumentException: database derby doesn't support function: atan2. |

这个信息其实是不对的，Derby数据库支持该函数，而方言中遗漏了这个函数。现在我们可以在方言中补上注册

orm-tutorial\src\main\java\org\easyframe\tutorial\lessonc\DerbyExtendDialect.java

|  |
| --- |
| **public** DerbyExtendDialect() {  **super**();  registerNative(**new** StandardSQLFunction("atan2"));  } |

然后再运行上面的案例，可以发现案例能够正确运行。

**示例2：让Derby支持ifnull函数**

MySQL中有一个ifnull函数。返回参数中第一个非空值。

如果我们要在Derby上支持带有这个函数的SQL语句，要怎么写呢？我们可以在自定义方言的构造函数中，注册这样一个函数。

|  |
| --- |
| registerCompatible(**null**, **new** TemplateFunction("ifnull", "(CASE WHEN %1$s is null THEN %2$s ELSE %1$s END)"),"ifnull"); |

这个函数的作用是在实际运行时，用一组CASE WHEN... THEN... ELSE... 语句来代替原先的ifnull函数。

我们试一下

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson7\ DerbyExtendDialect.java

|  |
| --- |
| **public** **void** testExtendDialact() **throws** SQLException {  ……  sql = "select ifnull(gender, 'F') from t\_person";  System.out.println(db.createNativeQuery(sql).getResultList());  } |

可以发现，在注册ifnull函数之前，上面的SQL语句是无法运行的，而注册了函数之后，SQL语句就可以运行了。

EF-ORM中注册的函数有三种方式

|  |  |
| --- | --- |
| 方法 | 作用 |
| registerNative() | 注册一个函数的本地实现。本地实现就是数据库原生支持的函数。比如 count() sum() avg()等函数。 |
| registerAlias() | 注册一个函数的替换实现。替换实现就是数据库虽然不支持指定的函数，但是有其他函数能兼容或者包含需要的函数功能。在实际执行时，函数名将被替换为本地函数名，但对参数等不作处理。 |
| registerCompatible() | 注册一个函数的模拟实现。模拟实现就是完全改写整个函数的用法。从函数名称到参数，都会被重写。 |

### 绑定变量占位符

E-SQL中表示参数变量有两种方式 :

* :param-name　(如:id :name，用名称表示参数)
* ?param-index　 (如 ?1 ?2，用序号表示参数)。

上述绑定变量占位符是和JPA规范完全一致的。

E-SQL中，绑定变量的参数类型可以声明也可以不声明。比如上例的

|  |
| --- |
| select count(\*) from Person\_table where id in (:ids<int>) |

也可以写作

|  |
| --- |
| select count(\*) from Person\_table where id in (:ids) |

但是如果不声明类型，那么如果传入的参数为List<String>，那么数据库是否能正常执行这个SQL语句取决于JDBC驱动能否支持。（因为数据库里的id字段是number类型而传入了string）。

指定参数类型是一个好习惯，尤其是当参数来自于Web页面时，这个特性尤其实用。

很多时候我们从Web页面或者配置中得到的参数都是string类型的，而数据库操作的类型可能是int,date,boolean等类型。此时我们可以在Native Query语句中指定参数类型，使其可以自动转换。

参数的类型有：date,timestamp,int,string,long,short,float,double,boolean。

参数可以为数组，如上例，可以用数组表示in条件参数中的列表。

目前我们支持的参数类型包括(类型不区分大小写)：

|  |  |
| --- | --- |
| 类型名 | 效果 |
| DATE | 参数将被转换为java.sql.Date |
| TIMESTAMP | 参数将被转换为java.sql.Timestamp |
| INT | 参数将被转换为int |
| STRING | 参数将被转换为string |
| LONG | 参数将被转换为long |
| SHORT | 参数将被转换为short |
| FLOAT | 参数将被转换为float |
| DOUBLE | 参数将被转换为double |
| BOOLEAN | 参数将被转换为boolean |
| STRING$ | 参数将被转换为string,并且后面加上%，一般用于like xxx% 的场合 |
| $STRING$ | 参数将被转换为string,并且两端加上%，一般用于like %xxx% 的场合 |
| $STRING | 参数将被转换为string,并且前面加上%，一般用于like %xxx 的场合 |
| SQL | SQL片段。参数将直接作为SQL语句的一部分，而不是作为SQL语句的绑定变量处理（见后文例子） |

上面的STRING$、$STRING$、$STRING三种参数转换，其效果是将$符号替换为%，主要用于从WEB页面传输模糊匹配的查询条件到后台。使用该数据类型后，%号的添加交由框架自动处理，业务代码可以更为清晰简洁。看下面的例子

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson7\Case1.java

|  |
| --- |
| /\*\*  \* 绑定变量中使用Like条件  \*/  @Test  **public** **void** testLike(){  String sql ="select \* from t\_person where person\_name like :name<$string$>";  System.*out*.println(*db*.createNativeQuery(sql).setParameter("name", "张").getResultList());  } |

SQL类型是将参数作为SQL片段处理，该功能使用参见7.3.6节。

### 动态SQL——表达式省略

EF-ORM可以根据未传入的参数，动态的省略某些SQL片段。这个特性往往用于某些参数不传场合下的动态条件，避免写大量的SQL。有点类似于IBatis的动态SQL功能。

我们先来看一个例子

|  |
| --- |
| **public** **void** testDynamicSQL(){  //SQL语句中写了四个查询条件  String sql="select \* from t\_person where id=:id " +  "and person\_name like :person\_name<$string$> " +  "and currentSchoolId=:schoolId " +  "and gender=:gender";  NativeQuery<Person> query=*db*.createNativeQuery(sql,Person.**class**);  {  System.*out*.println("== 按ID查询 ==");  query.setParameter("id", 1);  Person p=query.getSingleResult(); //只传入ID时，其他三个条件消失  System.*out*.println(p.getId());  System.*out*.println(p);  }  {  System.*out*.println("== 由于参数'ID'并未清除，所以变为 ID + NAME查询 ==");  query.setParameter("person\_name", "张"); //传入ID和NAME时，其他两个条件消失  System.*out*.println(query.getResultList());  }  {  System.*out*.println("== 参数清除后，只传入NAME，按NAME查询 ==");  query.clearParameters();  query.setParameter("person\_name", "张"); //只传入NAME时，其他三个条件消失  System.*out*.println(query.getResultList());  }  {  System.*out*.println("== 按NAME+GENDER查询 ==");  query.setParameter("gender", "F"); //传入GENDER和NAME时，其他两个条件消失  System.*out*.println(query.getResultList());  }  {  query.clearParameters(); //一个条件都不传入时，整个where子句全部消失  System.*out*.println(query.getResultList());  }  } |

上面列举了五种场合，每种场合都没有完整的传递四个WHERE条件。

上述实现是基于SQL抽象词法树（AST）的。表达式省略功能的定义是，如果一个绑定变量参数条件（如 = > < in like等）一端无效，那么整个条件都无效。如果一个二元表达式（如and or等）的一端无效，那么就退化成剩余一端的表达式。基于这种规则，NativeQuery能够将未设值的条件从查询语句中去除。来满足动态SQL的常见需求。

|  |
| --- |
| **使用 绑定变量 + 动态SQL 开发Web应用的列表视图**  *这种常见需求一般发生在按条件查询中，比较典型的一个例子是用户Web界面上的搜索工具栏，当用户输入条件时，按条件搜索。当用户未输入条件时，该字段不作为搜索条件。使用动态SQL功能后，一个固定的SQL语句就能满足整个视图的所有查询场景，极大的简化了视图查询的业务操作。*  *在一些传统应用中，开发者不得不使用许多IF分支去拼装SQL/HQL语句。为此产生了大量的重复编码。除此之外，还有一个丑陋的 “1=1” 条件（写过这类代码的人应该知道我在说什么。）这种SQL除了消耗额外的数据库解析时间外，也令数据库优化变得更为困难。*  *配合上一节讲到的绑定变量类型自动转换功能，使用EF-ORM在开发此类Web应用时，只要一个SQL语句加上极少的代码，就能完成所需的业务逻辑。如果在Web控制层(Action)中进行少量封装后，基本可以做到后台逻辑开发零编码。* |

最后，还要澄清一点——什么叫“不传入参数”。实时上，不传入参数表示自从NativeQuery构造或上次清空参数之后，都没有调用过setParameter()方法来设置参数的值。将参数设置为””或者null并不表示不设置参数的值。下面的例子说明了这一点。

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson7\Case1.java

|  |
| --- |
| @Test  **public** **void** testDynamicSQL3(){  String sql="select \* from t\_person where id not in (:ids)";  NativeQuery<Person> query=*db*.createNativeQuery(sql,Person.**class**);  //将参数值设置为null，并不能起到清空参数的作用  query.setParameter("ids", **null**);  System.*out*.println(query.getResultList());  } |

目前动态表达式省略可以用于两种场景，一是where条件，二是update语句中的set部分（见下面例子）。其他场合，如Insert into语句中的列等不支持这种用法。

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson7\Case1.java

|  |
| --- |
| /\*\*  \* 动态表达式省略——不仅仅是where条件可以省略，update中的赋值表达式也可以省略  \*/  @Test  **public** **void** testDynamicSQL4(){  String sql="update t\_person set person\_name=:new\_name, current\_school\_id=:new\_schoold\_id,gender=:new\_gender where id=:id";  NativeQuery<Person> query=*db*.createNativeQuery(sql,Person.**class**);  query.setParameter("new\_name", "孟德");  query.setParameter("id", 1);  **int** count=query.executeUpdate();  } |

### 动态SQL片段

有一种特别的NativeQuery参数类型，<SQL>表示一个SQL片段。严格来说，这其实不是一种绑定变量的实现。凡是标记为<SQL>类型的变量，都是被直接加入到SQL语句中的。

比如：

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson7\Case1.java

|  |
| --- |
| @Test  **public** **void** testDynamicSqlExpression(){  String sql="select :columns<sql> from t\_person where " +  "id in (:ids<int>) and person\_name like :person\_name<$string$> " +  "order by :orders<sql>";    NativeQuery<Person> query=*db*.createNativeQuery(sql,Person.**class**);  //查询哪些列、按什么列排序，都是在查询创建以后动态指定的。  query.setParameter("columns", "id, person\_name, gender");  query.setParameter("orders", "gender asc");    System.*out*.println(query.getResultList());    //动态SQL片段和动态表达式省略功能混合使用  query.setParameter("ids", **new** **int**[]{1,2,3});  query.setParameter("columns", "person\_name, id + 1000 as id");  System.*out*.println(query.getResultList());  } |

上面的例子中 :columns :orderby中被设置的值，都是SQL语句的一部分，通过这些动态的片段来重写SQL。第一次的查询，在Where条件没有任何输入的情况下where子句被省略。最后实际执行的SQL语句是这样——

|  |
| --- |
| select id, person\_name, gender from t\_person order by gender asc |

第二次查询，select表达式发生了一些变化，最后执行的SQL语句是这样——

|  |
| --- |
| select person\_name, id + 1000 as id from t\_person where id IN (?,?,?)  order by gender asc |

前面说了，在查询语句中也可以省略参数类型，比如上例，我们写作

|  |
| --- |
| select :columns from t\_person order by :orders |

此时，我们还能将orderBy当做是SQL片段，而不是运行时的绑定变量参数处理吗？

答案是可能的，只要我们传入的参数是SqlExpression对象，那么就会被当做是SQL片段，直接添加到SQL语句中。

|  |
| --- |
| query.setParameter("orders", new SqlExpression("id desc")); |

### 分页查询

NativeQuery对象支持分页查询。除了之前7.2.2中我们了解到的getResultCount()方法和setRange()方法外，还有已经封装好的方法。在Session对象中，我们可以从NativeQuery对象得到PagingIterator对象。

为了简化操作Session中还提供两个方法，直接将传入的SQL语句包装为NativeQuery，再从NativeQuery得到PagingIterator对象。

|  |  |  |
| --- | --- | --- |
| 方法 | 返回值 | 用途 |
| Session.pageSelect(NativeQuery<T>, int) | PagingIterator<T> | 从NativeQuery得到PagingIterator对象。 |
| Session.pageSelect(String, Class<T>, int) | PagingIterator<T> | 将传入的SQL语句包装为NativeQuery，再从NativeQuery得到PagingIterator对象。 |
| Session.pageSelect(String, ITableMetadata, int) | PagingIterator<T> | 将传入的SQL语句包装为NativeQuery，再从NativeQuery得到PagingIterator对象。 |

上述API的实际用法如下

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson7\Case1.java

|  |
| --- |
| @Test  **public** **void** testNativeQueryPage() **throws** SQLException{  //SQL语句中写了四个查询条件  String sql="select \* from t\_person where id=:id " +  "and person\_name like :person\_name<$string$> " +  "and currentSchoolId=:schoolId " +  "and gender=:gender";  NativeQuery<Person> query=*db*.createNativeQuery(sql,Person.**class**);  query.setParameter("gender", 'F');    //每页5条，跳过最前面的2条记录  Page<Person> page=*db*.pageSelect(query, 5).setOffset(2).getPageData();  System.*out*.println("总共:"+page.getTotalCount()+" "+page.getList());  } |

PagingIteraor对象的用法在前面已经介绍过了。这个对象为所有不同的查询方式提供了统一的分页查询接口。

NativeQuery分页应用的要点和前文一样——开发者无需关心COUNT语句和分页的写法，将这部分工作交给框架自行完成。

### 为不同RDBMS分别编写SQL

为了让SQL语句能被各种数据库识别，EF-ORM会在SQL语句和数据库函数层面进行解析和重写，但这不可能解决一切问题。在命名查询中，还允许开发者为不同的RDBMS编写专门的SQL语句。在运行时根据当前数据库动态选择合适的SQL语句。

一个实际的例子是这样——

|  |
| --- |
| <query name = *"test-tree#oracle"* type=*"sql"*>  <![CDATA[  select \* from t\_person t  START WITH t.id IN (:value)  CONNECT BY PRIOR t.id = t.parent\_id  ]]>  </query>  <query name = *"test-tree"* type=*"sql"*>  <![CDATA[  select \* from t\_person  ]]>  </query> |

在上例中，名为”*test-tree*”的命名查询有了两句SQL，一句是Oracle专用的，一句是其他数据库通用的。

一般来说，我们配置命名查询的名称是这样的，

|  |
| --- |
| <query name = *"test-tree"* > |

在所有RDBMS上使用的语句

但我们可以为查询名后面加上一个修饰——

|  |
| --- |
| <query name = *"test-tree"#oracle* > |

优先在Oracle上使用的语句

这个查询的名称依然是叫“test-tree”，但这个配置只会在特定的数据库(Oracle)上生效。

|  |
| --- |
| @Test  **public** **void** testNativeQueryPage2() **throws** SQLException{  //查询名还是叫test-tree。#后面的是修饰，不是查询名称的一部分  NativeQuery<Person> query=*db*.createNamedQuery("test-tree");  **if**(query.containsParam("value")){ //检查SQL是否需要该参数  query.setParameter("value", 100);  }  System.*out*.println(query.getResultList());  } |

在运行时，如果当前数据库是oracle，那么会使用专用的SQL语句，如果当前数据库是其他的，那么会使用通用的SQL语句。由于编码时不确定会使用哪个SQL语句，所以在设置参数前可以用containsParam检查一下是否需要该参数。

一般情况下，我们的查询名称中不用带RDBMS类型。这意味着该SQL语句可以在所有数据库上生效。

当RDBMS下SQL写法差别较大时，开发者可以使用这种用法，针对性的为不同的数据库编写SQL语句。

### 对Oracle Hint的支持

正常情况下，解析并重写后的SQL语句中的注释都会被除去。但是在Oracle数据库上，我们可能会用Oracle Hint的方式来提示优化器使用特定的执行计划，或者并行查询等。一个Oracle Hint的语法可能如下所述

|  |
| --- |
| select */\* + all\_rows\*/* \* from dave; |

基于抽象词法树的解析器对注释默认是忽略的，但为了支持Oracle Hint的用法，EF-ORM作了特别的处理，在特殊处理后，只有紧跟着SELECT UPDATE DELETE INSERT四个关键字的SQL注释可以被保留下来。一般情况下，Oracle Hint也就在这个位置上。

### 对Limit m,n / Limit n Offset n的支持

在PostgresSQL和MySQL中，都支持这种限定结果集范围的写法。这也是最简单的数据库分页实现方式。

我们首先来回顾一下SQL中Limit Offset的写法和几种变体：

1. LIMIT n OFFSET m 跳过前m条记录，取n条记录。
2. LIMIT m,n 跳过前m条记录，取n条记录。(注意这种写法下 n,m的顺序是相反的)
3. LIMIT ALL OFFSET m 跳过前m条记录，取所有条记录。
4. OFFSET m 跳过前m条记录，取所有条记录。(即省略LIMIT ALL部分)
5. LIMIT n OFFSET 0 取n条记录。
6. LIMIT n 取n条记录。(即省略 OFFSET 0部分)

以上就是LIMIT的SQL语法。

在E-SQL中，如果用户传入的SQL语句是按照上述语法进行分页的，那么EF-ORM会将其改写成适合当前RDBMS的SQL语句。即——

在非Postgresql或MySQL上，也能正常进行结果集分页。

在支持LIMIT语句的RDBMS上（如MySQL/Postgresql）上，LIMIT关键字将出现在SQL语句中。

### 对Start with ... Connect by的有限支持

Oracle支持的递归查询是一个让其他数据库用户很“怨念”的功能。这种语法几乎无法在任何其他数据库上使用，然而其用途却无可替代，并且难以用其他函数模拟。除了Postgres也有类似的递归查询用法外，在其他数据库上只有通过复杂的存储过程了……这使得开发要支持多数据库的产品变得更为困难。

EF-ORM却可以在所有数据库上，在一定程度上支持这种操作。

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson7\Case1.java

|  |
| --- |
| /\*\*  \* 在非Oracle数据库上支持递归查询  \*/  @Test  **public** **void** testStartWithConnectBy() **throws** SQLException {  //准备一些数据  *db*.truncate(NodeTable.**class**);  List<NodeTable> data=**new** ArrayList<NodeTable>();  data.add(**new** NodeTable(1,0,"-Root"));  data.add(**new** NodeTable(2,1,"水果"));  data.add(**new** NodeTable(5,2," 西瓜"));  data.add(**new** NodeTable(10,2," 葡萄"));  data.add(**new** NodeTable(4,2," 苹果"));  data.add(**new** NodeTable(8,4," 青涩的苹果"));  data.add(**new** NodeTable(12,4," 我的小苹果"));  data.add(**new** NodeTable(3,1,"家电"));  data.add(**new** NodeTable(6,3," 电视机"));  data.add(**new** NodeTable(7,3," 洗衣机"));  data.add(**new** NodeTable(11,6," 彩色电视机"));  *db*.batchInsert(data);    String sql = "select \* from nodetable t START WITH t.id IN (4,6) CONNECT BY PRIOR t.id = t.pid";  NativeQuery<NodeTable> query = *db*.createNativeQuery(sql, NodeTable.**class**);  List<NodeTable> result=query.getResultList();  **for**(NodeTable p:result){  System.*out*.println(p);  }  } |

上面的语句看似是不可能在Derby数据库上执行的，然而运行这个案例，你可以看见正确的结果。为什么呢？

这是因为EF-ORM 1.9.2以后，开始支持查询结果”后处理“。所谓“后处理”就是指对查询结果在内存中再进行过滤、排序等处理。这一功能本来是为了满足多数据库路由操作下的排序、group by、distinct等复杂操作而设计的，不过递归查询也得以从这个功能中获益。对于一些简单使用递归查询的场合，EF-ORM可以在内存中高效的模拟出递归效果。当然，在递归计算过程中需要占用一定的内存。

为什么说是”一定程度上支持这种操作“呢？因为目前对此类操作的支持限制还非常多，当前版本下，要使用内存计算模拟递归查询功能，有以下条件。

1. start with ... Connect by条件必须在顶层的select中，不能作为一个嵌套查询的内层。
2. Connect by的键值只允许一对。
3. Start with条件和connect by的键值这些列都必须在查询的结果中。
4. Start with目前还只支持一个条件，不支持AND OR。

## 存储过程调用

使用EF-ORM封装后的存储过程调用，可以——

* 指定存储过程的入参出参，并帮助用户传递。
* 将存储过程传出的游标，映射为合适的java对象。
* 支持匿名存储过程
* 和其他操作处于一个事务中

只要是数据库能支持的存储过程，EF-ORM都可以调用。存储过程调用过程会封装为一个NativeCall对象。使用该对象即可传入/取出存储过程的相关参数。对于游标类的传出参数，还可以直接转换为java bean。

### 使用存储过程

我们还是先从一个例子开始。由于Derby在存储过程上功能较弱，我们这个例子需要在MySQL下运行。存储过程为——

|  |
| --- |
| CREATE PROCEDURE update\_salary (IN employee\_number CHAR(6), IN rating INT)  LANGUAGE SQL  BEGIN  CASE rating  WHEN 1 THEN  UPDATE employee  SET salary = salary \* 1.10, bonus = 1000  WHERE empno = employee\_number;  WHEN 2 THEN  UPDATE employee  SET salary = salary \* 1.05, bonus = 500  WHERE empno = employee\_number;  ELSE  UPDATE employee  SET salary = salary \* 1.03, bonus = 0  WHERE empno = employee\_number;  END CASE;  END @ |

数据准备和测试代码：

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson7\Case3.java

|  |
| --- |
| @BeforeClass  **public** **static** **void** setup() **throws** SQLException {  **new** EntityEnhancer().enhance("org.easyframe.tutorial");    *db* = **new** DbClient("jdbc:mysql://localhost:3307/test","root","admin",5);  ORMConfig.*getInstance*().setDebugMode(**false**);  *db*.dropTable(Employee.**class**);  *db*.createTable(Employee.**class**);  Employee e=**new** Employee();  e.setId("100000");  e.setName("刘备");  e.setSalary(10000.0);  *db*.insert(e);  e=**new** Employee();  e.setId("100001");  e.setName("关羽");  e.setSalary(8000.0);  *db*.insert(e);  e=**new** Employee();  e.setId("100002");  e.setName("张飞");  e.setSalary(7000.0);  *db*.insert(e);  ORMConfig.*getInstance*().setDebugMode(**true**);  DbMetaData meta=*db*.getMetaData(**null**);  **if**(!meta.existsProcdure(**null**, "update\_salary")){ //如果存储过程不存在，就创建存储过程  meta.executeScriptFile(Case3.**class**.getResource("/update\_salary.sql"),"@");  }  }    @Test  **public** **void** testProducre() **throws** SQLException{  System.*out*.println("调整工资前——");  System.*out*.println(*db*.selectAll(Employee.**class**));    NativeCall nc=*db*.createNativeCall("update\_salary", String.**class**,**int**.**class**);  nc.execute("100002",2);  nc.execute("100001", 1);  System.*out*.println("调整工资后——");  System.*out*.println(*db*.selectAll(Employee.**class**));  } |

在运行上面的案例前请先修改

*db* = **new** DbClient("jdbc:mysql://localhost:3307/test","root","admin",5);

行中的URL，将其配置为一个可以连接的MySQL地址。在testProducre（）方法中对100001和100002号雇员进行了不同的加薪处理。通过存储过程调用前后输出的雇员信息变化，可以看到存储过程的效果。

### 存储过程传出参数

假设有存储过程如下

|  |
| --- |
| create procedure check\_user(IN \_name varchar(40),OUT userCount int)  BEGIN  select count(\*) into userCount from D where name like \_name;  END  $$ |

该存储过程传出一个数值。

|  |
| --- |
| @Test  **public** **void** testProducre2() **throws** SQLException{  NativeCall call2 = db.createNativeCall("check\_user",  String.**class**, OutParam.*typeOf*(Integer.**class**)); //设置存储过程入参和出参  call2.execute("张三");  Object obj = call2.getOutParameter(2);  LogUtil.*show*(obj);  } |

在上例中，存储过程的出参设置，需要用一个名为OutParam的工具类。使用这个类，可以生成若干出参的类型表示。OutParam.*typeOf*(Integer.**class**)表示一个Integer类型的传出参数。

在存储过程执行完后，使用getOutParameter(2);方法可以得到存储过程的传出参数，这里的2表示出参是第二个参数，这个序号需要和之前定义的参数需要一致，一个存储过程可以传出多个参数，因此要用序号加以区别。得到的传出参数类型和之前定义的一致，是Integer类型。

### 存储过程传出游标

Oracle数据库上的存储过程可以传出游标。存储过程如下：

|  |
| --- |
| CREATE OR REPLACE PACKAGE TESTPACKAGE AS  TYPE TYPE\_PERSON IS REF CURSOR;  end TESTPACKAGE;  /  create or replace PROCEDURE GET\_ALL(p\_CURSOR out TESTPACKAGE.TYPE\_PERSON) IS  BEGIN  OPEN p\_CURSOR FOR SELECT \* FROM T\_PERSON;  END GET\_ALL\_USER;  / |

这里的游标类型就是t\_person表。因此该存储过程相当于返回了一个在t\_person表上的查询结果集。

EF-ORM可以将游标类型的结果集重新映射为java对象。

|  |
| --- |
| @Test  **public** **void** testProducre3() **throws** SQLException{  NativeCall call3 = db.createNativeCall("GET\_ALL", OutParam.*listOf*(Person.**class**));  call3.execute();  List<Person> obj = call3.getOutParameterAsList(1, Person.**class**);  call3.close();  Assert.*assertTrue*(obj.size() > 0);  } |

在上例中，存储过程的出参设置，需要用一个名为OutParam的工具类。使用这个类，可以生成若干出参的类型表示。OutParam.*listOf*(Person.**class**)表示传出的游标将被包装为Person类型的List。

在存储过程执行完后，使用getOutParameterAsList(1,Person.class);方法可以得到存储过程的传出参数。其中，1是传出参数在存储过程定义中的序号。

大部分数据库都不支持传出游标。这个案例仅对支持的数据库（如Oracle）有效。

最后，要注意的是由于游标的存在，需要显式的去关闭NativeCall对象，否则会发生游标泄露问题。

### 使用匿名过程（匿名块）

在Oracle中，还可以执行临时编写的未命名的匿名块。匿名块的语法和存储过程基本一致。

对应到EF-ORM可以这样操作——

|  |
| --- |
| @Test  **public** **void** testProducre4() **throws** SQLException{  String sql = "declare " +  " l\_line varchar2(255); " +  " l\_done number; " +  " l\_buffer long; " +  "begin " +  " loop " +  " exit when length(l\_buffer)+255 > :maxbytes OR l\_done = 1; " +  " dbms\_output.get\_line( l\_line, l\_done ); " +  " l\_buffer := l\_buffer || l\_line || chr(10); " +  " end loop; " +  " :done := l\_done; " +  " :buffer := l\_buffer; " +  "end;";  NativeCall call = db.createAnonymousNativeCall(sql, Integer.**class**,  OutParam.*typeOf*(Integer.**class**), OutParam.*typeOf*(String.**class**));  call.execute(1, 26000);  System.*out*.println(call3.getOutParameter(2) + " || " + call3.getOutParameter(3));  } |

在上例中，用户临时定义了一个匿名块。EF-ORM中调用的方法为createAnonymousNativeCall()。该方法允许用户得到一个匿名块构成的NativeCall对象。匿名块的入参设置和出参获取和存储过程完全一样。

## 原生SQL使用

### 使用原生SQL查询

前面我们已经了解了EF-ORM对SQL的封装和改进。基于这种改进，我们使用E-SQL来享受改进所带来的优点——让SQL在各种RDBMS上运行；和业务代码更好的集成等等。

但是，麻烦必然伴随而来。SQL解析和改写器并不是总能完美的工作——

* SQL解析和改写是一个复杂的过程，尽管经过很多努力优化，但是每个SQL的解析依然要花费0.05到1毫秒不等的时间。可能不满足追求性能极限的场合。
* 一些过于复杂的，或者我们开发时没有测试到的SQL写法可能会解析错误。（请将解析错误的SQL语句发给我们，谢谢。）

EF-ORM内置的SQL分析器能处理绝大多数数据库DDL和DML语句。包括各种建表、删表、truncate、Create Table as、Select嵌套、Oracle分析函数、Oracle树型关系选择语句等。但是RDBMS的多样性和SQL语句的复杂性使得完全解析多少有些难度，因此EF-ORM依然保留原生的，不经过任何改写的SQL查询方式，作为NativeQuery在碰到以下麻烦时的”逃生手段“。

原生SQL和NativeQuery不同，不进行解析和改写。直接用于数据库操作。

明显的影响，原生SQL中，绑定变量占位符和E-SQL不同，用一个问号表示，和我们直接操作JDBC时一样——

|  |
| --- |
| select \* from t\_person where id=? and name like ? |

首先我们可以用Session对象(DbClient)中的selectBySql方法进行查询。看这个例子

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson7\Case1.java

|  |
| --- |
| @Test  **public** **void** testRawSQL() **throws** SQLException{  String sql="select id, person\_name,gender from t\_person";  { //普通的原生SQL查询  List<Person> result=*db*.selectBySql(sql,Person.**class**);  System.*out*.println(result);  *assertEquals*(3, result.size());  }  { //限定结果范围——分页  List<Person> result=*db*.selectBySql(sql, **new** Transformer(Person.**class**), **new** IntRange(2,3));  System.*out*.println(result);  *assertEquals*(2, result.size());  }  { //使用绑定变量  sql="select \* from t\_person where person\_name like ? or gender=?";  List<Person> result=*db*.selectBySql(sql, Person.**class**,"刘","F");  System.*out*.println(result);  *assertEquals*(3, result.size());  }  { //执行原生SQL  sql="insert into t\_person(person\_name,gender) values(?,?)";  *db*.executeSql(sql, "曹操","M");  *db*.executeSql(sql, "郭嘉","M");  *assertEquals*(5, *db*.getSqlTemplate(**null**).countBySql("select count(\*) from t\_person"));  } } |

上面使用了Session对象中的selectBySql方法，在Session中，这个系列的方法是可以使用原生SQL的——

|  |  |
| --- | --- |
| 方法 | 说明 |
| Session.executeSql(String, Object...) | 执行指定的SQL语句 |
| Session.getResultSet(String, int, Object...) | 根据SQL语句获得ResultSet对象 |
| Session.selectBySql(String, Class<T>, Object...) | 根据SQL查询，返回指定的对象 |
| Session.loadBySql(String, Class<T>, Object...) | 根据SQL查询，返回指定的对象（单行） |
| Session.selectBySql(String, Transformer, IntRange, Object...) | 根据SQL查询，传入自定义的结果转换器和分页信息 |
| Session.getSqlTemplate(String) | 获得指定数据源下的SqlTemplate对象。SQLTempate是一个可以执行各种SQL和本地化查询的操作句柄。 |

Session对象中，凡是xxxxBySql()这样的方法，都是传入原生SQL语句的。同时这些方法都提供了可变参数，其中的Obejct... 对象就是绑定变量参数。使用时按顺序传入绑定变量就可以了。

最后一个方法，可以得到SqlTemplate对象，SqlTemplate对象是一个可以执行各种SQL和本地化查询的操作句柄。

**在拼装SQL时处理Schema映射和函数本地化问题**

使用原生SQL，意味着开发者要自行解决schema重定向和数据库函数本地化的问题。可以使用下面两个方法来帮助获得相关的信息。

|  |  |
| --- | --- |
| 方法 | 说明 |
| MetaHolder.getMappingSchema(String) | 传入开发时的虚拟schema，返回实际运行环境对应的schema。用于拼装到原生SQL中。 |
| Session.func(DbFunction, Object...) | 传入函数名和参数。返回该函数在当前数据库下的方言实现。 |
| SqlTemplate.func(DbFunction, Object...) | 效果同上，区别是使用了特定数据源的方言。 |

上述三个方法都可以返回String，供开发人员自行拼装SQL语句使用。

### SqlTemplate

由于EF-ORM支持多数据源，因此要在特定数据源上执行SQL操作时，都要先获得对应的SqlTemplate对象。前面的各种示例中，都是在Session上直接操作本地化查询和SQL的，这种操作方式只会在默认数据源上操作。因此SQLTemplate除了提供更多原生SQL的操作方法以外，还是操作多数据源时必须使用的一个对象。

要获得一个数据源的SqlTemplate对象，可以使用——

|  |
| --- |
| *db*.getSqlTemplate(**null**);//获得默认数据源的SqlTemplate  *db*.getSqlTemplate("datasource1");//获得datasource1的SqlTemplate |

得到了SqlTemplate

SqlTemplate对象的使用。SqlTemplate中有很多方法是和本地化查询有关的，也有使用原生SQL语句的查询。( 详情参阅API-DOC)

|  |  |
| --- | --- |
| 方法 | 作用 |
| getMetaData() | 获得数据源元数据操作句柄 |
| createNativeQuery(String, Class<T>) | 创建本地化查询 |
| createNativeQuery(String, ITableMetadata) | 创建本地化查询 |
| createNativeCall(String, Type...) | 创建存储过程调用。 |
| createAnonymousNativeCall(String, Type...) | 创建匿名过程调用。 |
| pageSelectBySql(String, Class<T>, int) | 按原生SQL分页查询 |
| pageSelectBySql(String, ITableMetadata, int) | 按原生SQL分页查询 |
| countBySql(String, Object...) | 按SQL语句查出Long型的结果 |
| executeSql(String, Object...) | 执行SQL语句 |
| loadBySql(String, Class<T>, Object...) | 按SQL语句查出指定类型结果（单条记录） |
| selectBySql(String, Class<T>, Object...) | 按SQL语句查出指定类型结果 |
| selectBySql(String, Transformer, IntRange, Object...) | 按SQL语句查出指定类型结果(带分页范围) |
| iteratorBySql(String, Transformer, int, int, Object...) | 按SQL语句查出指定类型结果，以遍历器形式返回。 |
| executeSqlBatch(String, List<?>...) | 执行SQL语句，可以传入多组参数并在一个批次内执行。 |

使用SqlTemplate的示例——

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson7\Case2.java

|  |
| --- |
| @Test  **public** **void** testSqlTemplate() **throws** SQLException{  //获得在datasource2上执行SQL操作的句柄  SqlTemplate t=*db*.getSqlTemplate("datasource2");  List<Person> person=t.selectBySql("select \* from t\_person where gender=?", Person.**class**, "F");  System.*out*.println(person);  } |

SqlTemplate中的其他方法以此类推，详见java-doc。

## 无表查询

无表查询是一类特殊的SQL查询。比如，查询当前数据库时间，在Oracle中，是通过一张虚拟表DUAL完成的——

|  |
| --- |
| select sysdate from dual |

在MySQL中，无表查询是这样的——

|  |
| --- |
| select current\_timestamp |

在Derby中，无表查询是这样的——

|  |
| --- |
| values current\_timestamp |

显然，不同的数据库的无表查询语法是不一样的。为了兼容不同数据库的无表查询场景，框架提供了相应的API。和无表查询相关的API主要有getExpressionValue()系列的方法，该方法可以执行一次无表查询。

API列表

|  |  |
| --- | --- |
| 方法 | 用途 |
| **Session中的方法** |  |
| getExpressionValue(String, Class<T>) | 传入SqlExpression对象，计算表达式的值。 |
| **SqlTempate中的方法** |  |
| getExpressionValue(String, Class<T>, Object...) | 传入String对象。得到指定的SQL表达式的查询结果(无表查询) |
| getExpressionValue(DbFunction, Class<T>, String...) | 得到指定的SQL函数的查询结果(无表查询) |

使用举例

|  |
| --- |
| @Test  **public** **void** testExpressionValue() **throws** SQLException{  //传入复杂表达式时，其函数和语法会被改写  String s="'今天是'||str(cast(year(sysdate)/100+1 as int))||'世纪'";  *assertEquals*("今天是21世纪", *db*.getExpressionValue(s, String.**class**));    //要在某个特定数据库上执行无表查询，可以用SqlTemplate  SqlTemplate t=*db*.getSqlTemplate(**null**);  //直接传入数据库函数  Date dbTime=t.getExpressionValue(Func.*current\_timestamp*, Date.**class**);  System.*out*.println("当前时间为:"+dbTime);  } |

Session中的无表查询方法，将在默认数据源上计算指定的数据库表达式；SqlTemplate中的同名方法可以在指定数据源上计算表达式。表达式中出现的函数会被解析和改写。此外SqlTemplate还提供了一个直接传入Func对象计算的函数。

# 高级查询特性

## 查询结果的转换

ORM框架最被人直观感受的功能就是： JDBC的查询结果ResultSet会被转换为用户需要的Java对象。没有之一。

在前面的例子中，我们已经了解到了一部分查询和其对应返回结果的规律。本章中，我们将全面的罗列这些规律，并介绍一些新的返回结果指定方式。EF-ORM提供了一套统一的查询返回结果定义方式，因此本章中提到的结果返回方式规则，适用于前面讲到的所有查询方式——Criteria API的或者是NativeQuery的。

简单说来，查询结果的返回场景分为以下几种——

1. 返回简单对象
2. 返回和查询表匹配的对象
3. 返回任意对象容器
4. 返回Var /Map
5. 多个对象以数组形式返回
6. 多个列以数组形式返回

除了上述几个情况以外，还有两种方式转换结果。

1. 动态表的返回：在查询动态表时返回结果的场景。所谓动态表，就是开发时不为数据库表编写映射类。而是在运行时直接按表结构生成元数据模型（ITableMetadata），用该模型当做映射类来操作数据表的方法。动态表查询将在后面章节描述。
2. 自定义ResultSet到Java对象的映射关系： 用户实现ResultSet到Java对象之间的映射器(Mapper)接口，用自行实现的代码来描述从ResultSet到java类之间的转换逻辑。

场景1~6将在这一章介绍；场景8的用法在8.2节介绍。现在我们分别看一下上述场景的例子——

### 返回简单对象

查询结果可以返回为一个简单对象，如String、Integer、Long、Boolean、Date这样的基本类型。

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson8\Case1.java

|  |
| --- |
| @Test  **public** **void** testResultType\_returnSimpleType() **throws** SQLException{  //返回String  Query<Person> query=QB.*create*(Person.**class**);  Selects select=QB.*selectFrom*(query);  select.column(Person.Field.*gender*);  List<String> result=*db*.selectAs(query,String.**class**);  System.*out*.println(result);    //返回data  NativeQuery<Date> nq=*db*.createNativeQuery("select created from t\_person",Date.**class**);  List<Date> results=nq.getResultList();  System.*out*.println(results);    //返回数字  List<Integer> result2=*db*.selectBySql("select count(\*) from t\_person group by gender", Integer.**class**);  System.*out*.println(result2);  } |

返回简单对象时，查询出的列只取第一列。该列的值被转换为需要的简单对象。

### 返回和查询表匹配的对象

这个是使用最广泛的，简单说来查询什么表，返回什么表的对象。这也是大多数人对框架的理解方式。也只有在这种模式下，级联、延迟加载等特性才能体现出来。

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson8\Case1.java

|  |
| --- |
| @Test  **public** **void** testResultType\_returnTableObject() **throws** SQLException{  List<Person> persons=*db*.select(QB.*create*(Person.**class**));  *assertEquals*(3,persons.size());  } |

基于Query对象的查询，因为Query在构造时就已经知道其对应的表和对象。因此默认情况下，Query就能返回其对应表的对象。多表查询如Join、还有SQL查询下是不会出现这种场景的。

### 返回任意容器对象

用户可以指定任意一个Bean作为返回结果的容器，只要SQL语句中查出的字段和该容器对象中的字段名称一致（忽略大小写），就会将查询结果列值注入。对于不能匹配的字段，查询结果将丢弃。

比如，我们用临时定义的一个类PersonResult来作为查询结果的容器。

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson8\Case1.java

|  |
| --- |
| @Test  **public** **void** testResultType\_otherDataObject1() **throws** SQLException {  Query<Person> q1 = QB.*create*(Person.**class**);  List<PersonResult> result=*db*.selectAs(q1, PersonResult.**class**);  PersonResult first=result.get(0);  //Person表中查出的current\_school\_id字段无用处，被丢弃。  //PersonResult中的字段birthday不存在，不赋值  System.*out*.println(ToStringBuilder.*reflectionToString*(first));    String sql="select t.\*,sysdate as birthday from t\_person t";  NativeQuery<PersonResult> q=*db*.createNativeQuery(sql,PersonResult.**class**);  PersonResult p=q.getSingleResult();  System.*out*.println(ToStringBuilder.*reflectionToString*(p));  }    **public** **static** **class** PersonResult{  @Column(name = "person\_name")  **private** String personName;  **private** String id;  **private** String gender;  **private** Date birthday;  //Getter Setter方法略。  } |

上例中，定义了一个返回结果容器类PersonResult。该类的字段和Person表并不完全一致。其中Person表中有的current\_school\_id字段该类没有，因此该列的值被丢弃。同时PersonResult类中的birthday字段Person表中没有，故该字段的值为null。

我们注意到，t\_person表中有一个名为created的日期型字段，能不能把birthday的值设置为这个列的值呢？是可以的，一个办法是在birthday上加上@Column注解——

|  |
| --- |
| **public** **static** **class** PersonResult{  @Column(name = "person\_name")  **private** String personName;  **private** String id;  **private** String gender;  @Column(name = "created") //增加@Column注解后，birthday就会去匹配数据库中的created列。  **private** Date birthday;  //Getter Setter方法略。  } |

也就是说，无论用于结果容器的类是不是为Entity，其字段上的@Column注解都有效。在ResultSet和结果类的字段进行匹配时，如果没有@Column注解，那么使用java field name和数据库列匹配；反之则用@Column中的name属性和数据库列匹配。均忽略大小写。

我们再看下面这个例子。这一次我们利用另一个数据库表的对象Student，来作为Person表的查询容器。

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson8\Case1.java

|  |
| --- |
| @Test  **public** **void** testResultType\_otherDataObject2() **throws** SQLException {  // 用Person表查出Student对象。  {  Query<Person> query = QB.*create*(Person.**class**);  Selects select = QB.*selectFrom*(query);  select.columns("id, name as name, gender, '3' as grade, created as dateOfBirth");  List<Student> result = *db*.selectAs(query, Student.**class**);  System.*out*.println(result);  }  // 用Person表查出Student对象。(这种写法虽然啰嗦，但是可以利用java编译器检查字段的正确性)  {  Query<Person> query = QB.*create*(Person.**class**);  Selects select = QB.*selectFrom*(query);  select.column(Person.Field.*id*);  select.column(Person.Field.*name*).as("name");  select.column(Person.Field.*gender*);  select.sqlExpression("'3'").as("grade");  select.column(Person.Field.*created*).as("dateOfBirth");  List<Student> result = *db*.selectAs(query, Student.**class**);  System.*out*.println(result);  }  // 用NativeSQL做到上一点  {  String sql = "select id,person\_name as name, gender, '3' as grade, created as date\_of\_birth from t\_person";  NativeQuery<Student> nq = *db*.createNativeQuery(sql, Student.**class**);  List<Student> result = nq.getResultList();  System.*out*.println(result);  }  } |

上例中，采用三种写法，使Person表的数据能被注入到Student对象中去。其中值得注意的是Student对象中有一个名为dateOfBirth，数据库列为"DATE\_OF\_BIRTH"的字段。

|  |
| --- |
| /\*\*  \* 出生日期  \*/  @Column(name="DATE\_OF\_BIRTH")  **private** Date dateOfBirth; |

然而在上面的示例中，前两处用的是java名，在SQL语句中用的是数据库列名。这是因为CriteriaAPI中的column(xxx)实现上有一个特殊规则。当使用as()方法指定列在SQL中的别名是dateOfBirth时，同时还指定了这个列要注入到对象的dateOfBirth字段内。因此书写时可以按java field名称。

|  |
| --- |
| **下划线不会被忽略**  *曾经有人询问，为什么不默认认为数据库中的PERSON\_NAME对应java中的personName。这过于想当然了。因为下划线在java和在数据端，都是有效的标示符。你可以在同一个类中同时定义——*  *personName*  *\_personName*  *person\_name*  *因此这种过度依赖java编程规范和个人命名习惯的“容错”方式，只会让框架变得越来越混乱。因此，在没有显式指定映射关系的情况下，系统不会认为DATE\_OF\_BIRTH会和dateofbirth匹配。下划线是名称中不可忽略的要素，为了避免混乱，EF-ORM不会去忽略名称中的下划线。* |

使用自定义对象来容纳结果，对于NativeSQL和原生SQL来说是特别方便的。我们可以用任何一个已经存在的Java Bean作为结果返回容器——只要在SQL语句中，将列的别名(alias)写的和java类的中字段名称一样就可以了。

比如，我们找了一个其他库中的类，这个类有 key, value两个属性。我们在编写SQL时候让返回的两个字段名称分别为key,value，查询结果就乖乖的进入到指定的对象里去了

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson8\Case1.java

|  |
| --- |
| @Test  **public** **void** testResultType\_otherDataObject3() **throws** SQLException{  String sql = "select gender as \"key\",count(\*) as value from t\_person group by gender";  List<Entry> results=*db*.selectBySql(sql, jef.common.Entry.**class**);  System.*out*.println(results);  } |

### 返回Var /Map

在没有合适的容器容纳返回结果时，一种通常的手段是使用Map。在前面的例子中，我们已经很多次的使用了Map容器。我们回过去看一下很久以前的例子——

src/main/java/org/easyframe/tutorial/lesson3/Case2.java

|  |
| --- |
| @Test  **public** **void** testSelect\_function1() **throws** SQLException {  Query<Student> q = QB.*create*(Student.**class**);  Selects items = QB.*selectFrom*(q);  items.column(Student.Field.*id*).min().as("min\_id");  items.column(Student.Field.*id*).max().as("max\_id");  items.column(Student.Field.*id*).sum().as("sub\_id");  items.column(Student.Field.*id*).avg().as("avg\_id");    **for**(Map<String,Object> result:db.selectAs(q,Map.**class**)){  System.*out*.println(result);  }  }  //select min(t.ID) AS MIN\_ID, max(t.ID) AS MAX\_ID, sum(t.ID) AS SUB\_ID, avg(t.ID) AS AVG\_ID from STUDENT t |

一个值得注意是问题是，不同的数据库对列的大小写处理是不同的。比如Oracle中，返回的数据库列名称都是大写的，MySQL正相反。而很不幸的，Map的Key是大小写敏感的，因此当我们从Map中获取数据时，我们要使用大写的列名，还是小写的呢？

为了解决这个问题，EF-ORM返回Map的时候，用的不是JDK中的HashMap或是其他Map。而是自行实现的一个类jef.script.javascript.Var，这个类实现了Map接口，是一个忽略Key大小写的Map。所有的key在被放入时都转为小写。开发者可以任意用大写或小写的方式获取Map中的数值。这种定义也和大部分SQL环境保持一致。

一些时候，不指定返回的结果类型，也会用Map作为默认的数据返回类型。例如NativeQuery的单参数构造、不指定查询返回结果类型的Join查询等。

当需要用Map作为返回类型时，只需指定Var.class/Map.class作为返回类型，无需指定到具体的HashMap等类型。

### 多个对象以数组形式返回

在Join查询中我们经常需要返回多个对象。我们看一下前面的示例

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson6\Case1.java

|  |
| --- |
| @Test  **public** **void** testMultiTable() **throws** SQLException {  Query<Person> p = QB.*create*(Person.**class**);  Query<Item> i = QB.*create*(Item.**class**);  Join join = QB.*innerJoin*(p, i, QB.*on*(Person.Field.*name*, Item.Field.*name*));  ...  {  Object[] objs = *db*.loadAs(join, Object[].**class**);  Person person = (Person) objs[0];  Item item = (Item) objs[1];  *assertEquals*(person.getName(), item.getName());  }  } |

在使用CriteriaAPI的查询中，我们可以使用Object[].class作为返回结果类型。因为参与查询的每张表，都是以java对象的形式传入的。

那么，在本地化查询中，我们直接编写的SQL语句也能以Object[]分别传回吗？框架默认不提供这种行为。但是可以通过ResultTransformer来定义从ResultSet到Object[]的转换逻辑。这将在下一节介绍。

### 多个列以数组形式返回

另一种方式下，我们依然可以返回数组。但这种方式返回的数组和返回多个对象不同。这是将查询出的多个列，直接以数组的方式返回。

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson8\Case1.java

|  |
| --- |
| @Test  **public** **void** testSlelectSimpleValueArray() **throws** SQLException {  NativeQuery<Object[]> objs = db.createNativeQuery("select 'Asa' as a ,'B' as b,1+1 as c, current\_timestamp as D from student", Object[].**class**);  Object[] result = objs.getSingleResult();  *assertTrue*(result[1].getClass() == String.**class**);  *assertTrue*(result[2].getClass() == Integer.**class**);  *assertTrue*(result[3].getClass() == Timestamp.**class**);  } |

复杂一些的查询，如多表查询，也是可以用这种方式返回结果的.

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson8\Case1.java

|  |
| --- |
| @Test  **public** **void** testResultType\_columnAsArray() **throws** SQLException {  String sql = "select t1.id,person\_name,gender,t2.name from t\_person t1," +  "school t2 where t1.current\_school\_id=t2.id";  {  NativeQuery<String[]> q1 = *db*.createNativeQuery(sql, String[].**class**);  // 每个列的值被转换为String，每行记录变为一个String[]。  **for** (String[] array : q1.getResultList()) {  System.*out*.println(Arrays.*toString*(array));  }  }  {  NativeQuery<Object[]> q2 = *db*.createNativeQuery(sql, Object[].**class**);  **for** (Object[] array : q2.getResultList()) {  // 每个列的值被保留了其原始类型，每行记录变为一个Object[]。  System.*out*.println(Arrays.*toString*(array));  }  }  {  // 单表查询API也是可以的  Query<Person> q = QB.*create*(Person.**class**);  Selects sel = QB.*selectFrom*(q);  sel.columns(Person.Field.*id*, Person.Field.*name*, Person.Field.*gender*);  List<Object[]> persons = *db*.selectAs(q, Object[].**class**);  **for** (Object[] array : persons) {  System.*out*.println(Arrays.*toString*(array));  }  }  } |

上例中三次查询，指定的返回类型都是数组。查询条结果就是按照select语句中各列的出现顺序形成数组。

* 如果指定为Object[]，那么每列的值将保留为默认的数据类型，如String、Integer等。
* 如果指定为String[]，那么每列的值都会被强转为Striung。
* 如果指定为Integer[]也会执行转换，不过如果有列不能转换为数字就会抛出异常。
* 其他类型同

从上面可以看到，无论用NativeQuery，或是单表查询的CriteriaAPI查询，都可以将结果列拼成数组返回。

有同学可能会有疑问，目前对于8.1.5和本节中，数组返回的指定方式是一样的，但结果集转换逻辑是有差异的。这是不是会造成混淆呢？请参见8.2章。

## Transformer的使用

在上面的例子中，我们都使用slectAs() loadAs()等方法，在传入Query对象的同时指定了要返回的class类型。实际上，即便不使用selectAs和loadAs方法，也一样能指定返回的结果类型。

我们看一下Session类中的loadAs的源代码：

代码清单：EF-ORM中Session类的源代码, loadAs方法

|  |
| --- |
| **public** <T> T loadAs(ConditionQuery queryObj,Class<T> resultClz) **throws** SQLException {  queryObj.getResultTransformer().setResultType(resultClz);  **return** load(queryObj);  } |

我们可以发现，其实数据返回类型就是直接设置在Query对象内的。

我们前面看到过的三种查询对象——Join / Query<T> /UnionQuery，其实都提供了一个方法。

|  |
| --- |
| getResultTransformer() |

使用这个方法就可以得到Transformer对象，这个对象就是描述结果转换的各种行为的对象。除了控制结果转换的类型以外，还提供了更多的相关参数。我们举几个典型的例子来观察其作用。

### 直接指定返回类型

首先，前面例子中的一部分使用selectAs loadAs的方法，我们可以直接用select、load等方法等效的来实现。

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson8\Case1.java

|  |
| --- |
| /\*\*  \* 使用ResultTransformer来指定返回类型  \* **@throws** SQLException  \*/  @Test  **public** **void** testResult\_transformer1() **throws** SQLException {  {  Query<Person> q = QB.*create*(Person.**class**);  Selects sel = QB.*selectFrom*(q);  sel.columns(Person.Field.*id*, Person.Field.*name*, Person.Field.*gender*);  q.getResultTransformer().setResultType(String[].**class**); //指定返回类型  String[] result=*db*.load(q);  System.*out*.println(Arrays.*toString*(result));  }  {  Query<Person> query = QB.*create*(Person.**class**);  Selects select = QB.*selectFrom*(query);  select.column(Person.Field.*id*);  select.column(Person.Field.*name*).as("name");  select.column(Person.Field.*gender*);  select.sqlExpression("'3'").as("grade");  select.column(Person.Field.*created*).as("dateOfBirth");  query.getResultTransformer().setResultType(Student.**class**); //指定返回类型  Student st=*db*.load(query);  }  } |

因此loadAs和selectAs方法不是必需的。不过这两类方法存在一是可以简化API，让人更容易理解和使用。二是让传入的Class泛型等同于传出的泛型，使用这两个方法可以利用java语法的校验，减少开发者搞错返回类型的机会。因此，一般在条件允许的情况下，您无需使用Transformer来指定返回结果。

### 区分两种返回数据的规则

在上一节，介绍了查询时返回数组的两种转换规则。

* 多表查询时每张表对应一个java bean，多个bean构成数组
* 查询的每个列的值对应到一个java值，多列的java值构成数组

大部分情况下，EF-ORM都能判断出用户传入数组的实际意图，并根据这个意图使用合适的转换规则。

但是也有一些特殊情况，用户传入的查询会具有二义性。比如前面的8.1.5 中Join查询场景，Join查询的Object[]返回格式被认为是“多个对象形成数组”返回，而不是“多列形成数组”返回。

这种情况下，也有办法，我们可以显式的提示EF-ORM，要采用后一种方式返回结果。

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson8\Case1.java

|  |
| --- |
| @Test  **public** **void** testResultType\_columnAsArray2() **throws** SQLException {  //多表查询的时候,Object[]的返回类型默认是一张表的多个列拼成的对象作为数组元素，  //而不是每个列作为数组元素……  Query<Person> q = QB.*create*(Person.**class**);  Join join=QB.*innerJoin*(q, QB.*create*(School.**class**));  List<Object[]> persons=*db*.selectAs(join,Object[].**class**);  **for** (Object[] array : persons) {  System.*out*.println("["+array[0].getClass()+","+array[1].getClass()+"]");  //该查询每条记录都转换为一个Person对象和一个School对象。  }  //可以这样写  join.getResultTransformer().setStrategy(PopulateStrategy.*COLUMN\_TO\_ARRAY*);  persons=*db*.selectAs(join,Object[].**class**);  **for** (Object[] array : persons) {  System.*out*.println(Arrays.*toString*(array));  }  //这样,多表查询也可以以String[]形式返回值了  join.getResultTransformer().setStrategy(PopulateStrategy.*COLUMN\_TO\_ARRAY*);  List<String[]> stringColumns=*db*.selectAs(join,String[].**class**);  **for** (String[] array : stringColumns) {  System.*out*.println(Arrays.*toString*(array));  }  } |

上例中join.getResultTransformer().setStrategy(PopulateStrategy.*COLUMN\_TO\_ARRAY*);就是给查询设置了一个结果转换策略。显式的指定了采用COLUMN\_TO\_ARRAY的方式转换结果。

### 忽略@Column注解

8.1.3节中介绍过，可以用任意java bean作为返回数据的容器。同时我们也提到对于自行编写的sql语句，我们只要将SQL语句中列的别名和java bean的属性名对应上，就可以简单的将数据库查询结果注入到这个Bean中。

那么我们可能写出这样一段代码

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson8\Case1.java

|  |
| --- |
| @Test  **public** **void** testResultType\_ignoreColumnAnnotation() **throws** SQLException {  String sql = "select id as id, name as name, gender as gender from student";  //将student表中查出的数据映射为Person对象。  NativeQuery<Person> nq = *db*.createNativeQuery(sql, Person.**class**);  Person person=nq.getSingleResult();  *assertNotNull*(person.getName());  } |

上面的代码看起来没什么问题，Person对象中也有id、name、gender三个属性。数据库中查出的三个列正好对应到这三个属性上。

但实际运行后，我们发现，Person对象中的name属性并没有被赋值。原因很简单，因为Person类中的name属性上有@Column注解，指定这个属性是和数据库列”person\_name”映射，而不是和”name”映射。

|  |
| --- |
| @Column(name = "person\_name", length = 80, nullable = **false**)  @Indexed(definition = "unique")  **private** String name; |

因此，当开发者希望查询结果直接和对象中的属性名发生映射，不受@Column注解影响时，可以这样

代码清单：修改后的查询写法:orm-tutorial\src\main\java\org\easyframe\tutorial\lesson8\Case1.java

|  |
| --- |
| @Test  **public** **void** testResultType\_ignoreColumnAnnotation() **throws** SQLException {  String sql = "select id as id, name as name, gender as gender from student";  //将student表中查出的数据映射为Person对象。  NativeQuery<Person> nq = *db*.createNativeQuery(sql, Person.**class**);  nq.getResultTransformer().setStrategy(PopulateStrategy.*SKIP\_COLUMN\_ANNOTATION*);    Person person=nq.getSingleResult();  *assertNotNull*(person.getName());  } |

上例中setStrategy(PopulateStrategy.*SKIP\_COLUMN\_ANNOTATION*); 给了结果转换器一个提示，使其忽略@Column注解。

### 自定义返回结果转换

这个可能是Transformer最为复杂也最为强大用法了。在本节中，您可以用自行实现的逻辑去转换ResultSet中返回值的处理。

我们还是分几种情况来介绍相关API和用法。

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson8\Case1.java

|  |
| --- |
| @Test  **public** **void** testResultMapper\_1() **throws** SQLException{  Query<Student> q=QB.*create*(Student.**class**);  q.getResultTransformer().addMapper(**new** Mapper<PersonResult>(){  @Override  **protected** **void** transform(PersonResult obj, IResultSet rs) **throws** SQLException {  obj.setBirthday(rs.getDate("DATE\_OF\_BIRTH"));  obj.setPersonName(rs.getString("NAME"));  **if**(obj.getBirthday()!=**null**){  //计算并设置年龄  **int** year=DateUtils.*getYear*(**new** Date());  obj.setAge(year-DateUtils.*getYear*(obj.getBirthday()));  }  }  });  PersonResult result = *db*.loadAs(q,PersonResult.**class**);  System.*out*.println(result.getPersonName()+"出生于"+result.getBirthday()+" 今年"+result.getAge()+"岁");  } |

上例中，为Student表中的数据转换到PersonResult对象提供了自定义的规则实现。

上面的例子中，自定义ResultSet到java Bean的转换。如果一个Java Bean的属性非常多，那么代码也会很繁琐。因此，考虑到很多类都是已经通过Entity类定义了数据库字段和对象关系的类，我们是不是可以利用这些Entity中固有的和数据库的映射关系呢？

正是基于这种考虑，EF-ORM中还提供了一个Mappers工具类,可以用Mappers工具直接生成Mapper映射器。

比如这个例子，用自行编写的SQL语句返回Person和school对象。其中school对象将被赋值到Person中的currentSchool字段里。

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson8\Case1.java

|  |
| --- |
| @Test  **public** **void** testResultMapper\_2() **throws** SQLException{  String sql="select t1.\* , t2.\* from t\_person t1,school t2 " +  "where t1.current\_school\_id=t2.id ";  NativeQuery<Person> query = *db*.createNativeQuery(sql,Person.**class**);  //Mappers可以提供一些默认的映射器。大部分情况下，field到column的映射都是正确的，  query.getResultTransformer().addMapper(Mappers.*toResultProperty*("currentSchool", School.**class**));  List<Person> result=query.getResultList();  **for**(Person person: result){  System.*out*.println(person.toString()+" ->"+person.getCurrentSchool());  }  } |

在上例中，使用Mappers.*aliasToProperty*方法，框架提供了一个默认的映射行为。该行为将属于School对象的所有字段组成一个Schoold对象，然后赋值到目标的currentSchool这个属性中去。

正常操作下，只有框架的级联查询才能产生Bean嵌套的结构作为结果返回。当然用上面的例子手工编码也可以做出同样的效果。但是，既然Person和School都是我们已经建模的Entity对象，为什么不能少些一点代码呢。所以Mappers工具里提供了若干方法，作用是生成一个和已知Enrtity进行映射的Mapper对象。

并且Mappers工具里也包含这样的动作，即生成的映射可以将已知的Entity注入到返回结果的属性中去，产生类似级联操作的嵌套结构。

|  |
| --- |
| query.getResultTransformer().addMapper(  Mappers.*toResultProperty*("currentSchool", School.**class**)); |

这里的*toResultProperty*的含义就是，由数据库列转换而成的对象，注入到目标的”currentSchool”属性中去。

非常不幸的，上面的案例在Derby虽然可以运行，但是隐藏了一个重大的问题。因为t\_person表和school表中都有同名列id，因此查询结果中只能得到一个名为id的列，最终返回的数据是错误的。（在某些数据库上这样的SQL语句会直接报错）。

为此，如果我们要想将id的问题解决，我们可以把方法改成下面这样——

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson8\Case1.java

|  |
| --- |
| @Test  **public** **void** testResultMapper\_2\_1() **throws** SQLException{  //由于两张表都有id列，这里必须重命名  String sql="select t1.\* , t2.id as schoolid, t2.name from t\_person t1,school t2 " +  "where t1.current\_school\_id=t2.id ";  NativeQuery<Person> query = *db*.createNativeQuery(sql,Person.**class**);    //不一致的field，可以使用adjust方法调整。  query.getResultTransformer().addMapper(  Mappers.*toResultProperty*("currentSchool", School.**class**).adjust("id", "schoolid"));  List<Person> result=query.getResultList();  } |

Mappers工具返回的BeanMapper对象中，提供了adjust方法，可以指定个别属性的列映射。第一个参数是java属性名，第二个参数是数据库列名。这样就能解决实际使用时，一个Entity大部分字段都和数据库列对应、而个别不对应的问题了。

Mappers工具类提供的方法如下

|  |  |
| --- | --- |
| 方法 | 效果 |
| Mappers.toResultProperty(String, Class<T>) | 将查询结果中的列转换为Class所指定的对象后，注入到结果类的属性中。 |
| Mappers.toResultProperty(String, Class<T>, String) | 将查询结果中的列转换为Class所指定的对象后，注入到结果类的属性中。 可以指定列的命名空间。（参见后文注解） |
| Mappers.toResultProperty(String, ITableMetadata, String) | 将查询结果中的列转换为ITableMetadata所指定的对象后，注入到结果类的属性中。 可以指定列的命名空间。（参见后文注解） |
| Mappers.toResultBean(Class<T>) | 将查询结果中的列按指定类型Entity映射关系，直接注入到结果类中。（比如DO对象中配置映射关系，VO对象字段几乎一致但未配置映射关系，此时可以按DO对象的规则将属性注入到VO对象中） |
| Mappers.toResultBean(Class<T>, String) | 将查询结果中的列按指定类型Entity映射关系，直接注入到结果类中。  可以指定列的命名空间。（参见后文注解） |
| Mappers.toResultBean(ITableMetadata, String) | 将查询结果中的列按指定类型ITableMetadata映射关系，直接注入到结果类中。  可以指定列的命名空间。（参见后文注解） |
| Mappers.toArrayElement(int, Class<T>) | 将查询结果中的列转换为Class所指定的对象后，注入到结果数组的指定位置上。（要求查询返回结果为数组） |
| Mappers.toArrayElement(int, Class<T>, String) | 将查询结果中的列转换为Class所指定的对象后，注入到结果数组的指定位置上。（要求查询返回结果为数组）  可以指定列的命名空间。（参见后文注解） |
| Mappers.toArrayElement(int, ITableMetadata, String) | 将查询结果中的列转换为Class所指定的对象后，注入到结果数组的指定位置上。（要求查询返回结果为数组）  可以指定列的命名空间。（参见后文注解） |

上面列举了9个生成BeanMapper对象的方法。

首先，需要解释“指定列别名的前缀。”究竟是什么意思。

注意观察的同学可能已经发现，EF-ORM在使用Criteria API进行多表查询时，对列名的默认处理方式是这样的——

|  |
| --- |
| select T1.CURRENT\_SCHOOL\_ID AS T1\_\_CURRENTSCHOOLID,  T1.ID AS T1\_\_ID,  T1.PERSON\_NAME AS T1\_\_NAME,  T2.NAME AS T2\_\_NAME,  T2.ID AS T2\_\_ID  from T\_PERSON T1  left join SCHOOL T2 ON T1.CURRENT\_SCHOOL\_ID = T2.ID |

也就是说，每个列的别名是通过增加了表别名的前缀来完成的，整个前缀共4个字符，其中两位是分隔符。实际上，就相当于将原来的ID、NAME等列放到了不同的命名空间下。上面这句SQL中，PERSON表的字段都在T1的命名空间下，SCHOOL表的字段都在T2命名空间下。

因此，当我们使用框架的Criteria API查询数据时，使用“命名空间”可以快速的区分出属于一张表的所有列。

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson8\Case1.java

|  |
| --- |
| @Test  **public** **void** testResultMapper\_3() **throws** SQLException{  Query<Person> t1 = QB.*create*(Person.**class**);  Query<Student> t2 = QB.*create*(Student.**class**);  Join join = QB.*innerJoin*(t1, t2, QB.*on*(Person.Field.*name*, Student.Field.*name*));  Transformer transformer=join.getResultTransformer();    //因为是两表查询，默认返回的数组长度为2，为了增加一个返回对象需要将数组长度调整为3  transformer.setResultTypeAsObjectArray(3);  transformer.addMapper(Mappers.*toArrayElement*(0, Student.**class**, "T2"));  transformer.addMapper(Mappers.*toArrayElement*(1, Person.**class**, "T1"));    //增加一个自定义的映射。  transformer.addMapper(**new** Mapper<Object[]>() {  @Override  **protected** **void** transform(Object[] obj, IResultSet rs) **throws** SQLException {  PersonResult result=**new** PersonResult();  result.setPersonName(rs.getString("T1\_\_NAME"));  result.setBirthday(rs.getDate("T1\_\_CREATED"));  obj[2]=result;  }  });  List<Object[]> result = *db*.select(join, **null**);  *assertNotNull*(result.get(0)[2]);  } |

上面的映射中，将Join查询默认返回两个对象的行为，修改成了返回3个对象，同时还调整了School和Person对象在结果中的位置。这看似和默认的返回行为差不多，但整个过程是充分定制的。并且这种定制是可以随心所欲的——包括T1和T2的映射对象可以变为别的Java Bean。

最后，ResultTransformer中提供了若干ignore系列的方法。这是由addMapper的动作衍生而来的。在一些查询中，addMapper并不会清除掉框架默认的结果转换规则，而是会并存。而某些时候我们并不希望发生框架默认的转换行为，因此可以用ignoreXXX方法，要求框架不处理返回结果中的指定列。

我们直接在上面的方法结束前增加一些代码——

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson8\Case1.java

|  |
| --- |
| //第二段案例: 清除映射器、忽略默认的映射规则  {  Transformer t=join.getResultTransformer();  t.setResultType(Holder.**class**);    //清除之前定义的映射器，因为之前已经在Transformer中添加了映射器。  t.clearMapper();  //忽略默认的映射规则  t.ignoreAll();  t.addMapper(**new** Mapper<Holder<PersonResult>>(){  @Override  **protected** **void** transform(Holder<PersonResult> obj, IResultSet rs) **throws** SQLException {  PersonResult result=**new** PersonResult();  result.setPersonName(rs.getString("T1\_\_NAME"));  result.setBirthday(rs.getDate("T1\_\_CREATED"));  obj.set(result);  }  });  List<Holder<PersonResult>> holders=*db*.select(join,**null**);  **for**(Holder<PersonResult> h: holders){  System.*out*.println(h.get());  }  } |

可以看到，在这个部分做了两个至关重要的操作——

* t.clearMapper();  
  清除Join中之前设置的映射器，之前的自定义映射器无法针对本次查询的返回类型操作，如果不清除而任由这些映射器生效，必然造成程序异常。
* t.ignoreAll();   
  使框架默认的映射规则忽略所有列。即禁用默认的映射规则。因为默认的映射规则完全不能识别这次的返回类型，因此如不清除，一样会抛出异常。

## 流式操作接口

当我们从数据库需要读取数百万记录时（比如报表和导出功能），常常会面临内存不够用的问题。因为如果一个用户就在内存中缓存百万的数据，那么整个系统都没有内存进行其他正常的工作了。

许多人会采用分页的方法，例如每次到数据库读取5000条。但是会受数据库的幻读问题困扰，因为多次分页期间，系统可能并发的删除或添加某些记录。因此在“不可重复读”的环境下，用分页实现高可靠的数据导出是困难的，某些记录可能被丢失，而某些记录可能会被查出两遍。

一个可行的办法，是使用JDBC的流式处理接口。这点尤其在Oracle上特别有用，因为Oracle的驱动实现得较好——可以返回一批数据到JDBC驱动中，随着一边处理一边将结果向后滚动，批量的将后续的数据传入到客户端来。

这种ResultSet实现使得数据库的查询响应变快，同时用户可以合理的调整每批加载的记录数（fetch-size），使得整体性能达到最优。MySQL 5.0以上Server和5.0以上JDBC驱动，多少也实现了这种流式加载的方法。这能有效的避免客户端出现OutOfMemoryError的信息。

到目前为止，并非所有的数据库JDBC驱动都支持流式操作模型。（也有些数据库是一次性拉取全部结果的），但不管怎么说支持流式操作确实是RDBMS在大数据时代面临的和必须解决的一个问题。

但是我们再来看之前使用数据库查询接口

|  |
| --- |
| **public** <T **extends** IQueryableEntity> List<T> select(T obj) **throws** SQLException; |

很显然，流式处理模型在这个方法上是走不通的，List<T>中会缓存整个结果集的全部数据。内存占用不是一般的大。

为此，EF-ORM中提供了支持流式操作模型的接口。

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson8\Case2.java

|  |
| --- |
| @Test  **public** **void** testIteratedSelect() **throws** SQLException{  Query<Person> p=QB.*create*(Person.**class**);  p.setFetchSize(100);  ResultIterator<Person> results=*db*.iteratedSelect(p, **null**);  **try**{  **for**(;results.hasNext();){  Person person=results.next();  //执行业务处理  System.*out*.println(person);  }  }**finally**{  results.close();  }  } |

在Session对象中，有iteratedSelect的几个方法，这个系列的方法都可以返回一个ResultIterator<T>对象，这个对象提供了流式处理的行为。按照上例的逻辑，每处理一个对象，就可以在内存中释放掉该对象（在取消了结果集缓存的场合下）。结果集向后滚动到底，内存中也只有最近处理的对象。

对于NativeQuery。前面已经提到过，NativeQuery提供了相同功能的查询方法——

|  |
| --- |
| **public** ResultIterator<X> NativeQuery.getResultIterator(); |

那么如果原生SQL，想要查询大量数据并用流式模型操作呢？在SqlTemplate类中也提供了类似方法——

|  |
| --- |
| **public** **final** <T> ResultIterator<T> iteratorBySql(String sql,  Transformer transformers, **int** maxReturn, **int** fetchSize, Object... objs) **throws** SQLException |

这个方法的参数有些复杂，详情请参阅API-DOC。

**注意事项**

1 请注意ResultIterator的关闭问题。正常情况下，当遍历完成后ResultIterator会自动关闭。但是我们希望在编程时，必须在finally块中显式关闭ResultIterator对象。因为这个对象不关闭，意味着JDBC ResultSet不关闭，数据库资源会很快耗尽的。

2 另外。在Session中有一个方法Session.getResultSet(String, int, Object...)。这个方法返回ResultSet对象，可能有人会误以为这就是JDBC驱动的原生结果集。可以实现流式操作。但实际上这个方法是对原生的Result的完整缓存。因此并不能用在超大结果集的返回上。

# 事务控制与批操作

## 编程式事务控制

本节描述EF-ORM的原生事务接口，不讲述在Spring下的声明式事务。声明式事务请参阅第11章《与Spring集成》。

在EF-ORM中，事务操作是在一个名为Transaction的类上进行的。Transaction和DbClient上都可以操作数据库。它们的关系如下图所示：



图9-1 DbClient和Transaction的关系

所有DML类型的数据库操作，都是由DbClient和Transaction的公共基类Session提供的。Transaction类上不能执行DDL操作，但提供了commit()和rollback()方法，用于提交或回滚事务。

DbClient上没有提交或回滚操作，但允许进行DDL操作。

这个设计是基于这样的一种模型：用户可以同时操作多个Session。一些Session是有事务控制的(Transaction)，可以回滚或者提交。一些Session则是非事务的、会auto-commit的 (DbClient)，用户无论执行DML还是DDL，都会被立刻提交。事实上，在EF-ORM内部实现中，这也是通过为用户分配不同的连接来实现的。

简单来说，DbClient是没有事务状态的Session(Non-Transactional Session)。Transaction则是有事务状态的Session(Transactional Session)。

从DbClient上，开启一个新的事务Session，可以使用 startTransaction()方法。

|  |
| --- |
| Transaction session = db.startTransaction(); //创建一个新的事务，类似于H框架的openSession()方法 |

从Transaction对象上，也可以得到初始的非事务Session (即DbClient)。

|  |
| --- |
| DbClient db=session.getNoTransactionSession(); //得到非事务Session。 |

下面的代码清单例子演示了数据库事务操作。

|  |
| --- |
| DbClient db=new DbClient()  Transaction transaction=db.startTransaction(); 、  **try**{  transaction.insert(entity1);  transaction.insert(entity2);  transaction.insert(entity3);  db.insert(entity4); //这个插入操作和上面三条记录不一样，是单独提交的  transaction.commit(); //提交上面三个实体的插入  }**catch**(SQLException e){  transaction.rollback(); //回滚  }  transaction.updateCascade(entity5);//使在事务上执行级联操作。  transaction.commit (); //提交事务  transaction.close(); |

上面的方式，即相当于操作编程式事务，您可以在一个方法中同时操作多个事务。事务之间可以保持互相隔离。（隔离级别取决于数据库设置）。

在现在大部分企业级开发项目中，基于AOP的声明式事务已经代替了编程式事务，EF-ORM也同样提供了JPA的事务接口。利用Spring Framework也提供了对JPA的事务支持可以实现AOP式的声明式事务。

因此在实际的企业级开发中事务管理支持事实上都是统一使用Spring Framework的事务模型的，Spring事务管理模型支持的四种数据库隔离级别和七种事务传播行为都可以在EF-ORM上实现。（标准JPA只支持六种传播行为）相关内容请参阅11节《与Spring集成》。

## 批操作

无论在任何场合下，相同数据结构的多条数据操作性能问题总是会引起人们的关注。解决SQL性能的重要手段就是批量操作，这一操作经常被用来进行大量数据的插入、更新、删除。

批操作是改善数据库写入性能最重要的手段。批操作的本质，简单来说就是——*单句SQL，多组参数，一次执行*

这三个特定决定了批操作的特点和限制

* 单句SQL：一个批次当中所有数据的写入都必须共用同一句SQL，一个批次当中不能有不同的SQL语句。
* 多组参数：批操作中的每个元素都被转换为和公用SQL语句相匹配的参数。显然，批次要有合适大小，性能才会提高。如果一批请求只有三五组参数，性能优势就不明显了。
* 一次执行：前面的NativeQuery中，一个Query也是能搭配多组参数反复运行的。批操作比普通绑定变量操作更快的原因是，批操作不是逐条执行的，而是一次性发送所有参数到数据库服务器，数据库也一次性的返回结果。通信次数的减少是批操作性能提升的关键。  
  当然一次执行也衍生了另一个特点，即一批数据中任意一组参数操作失败，那么整批数据操作都失败。这是要引起注意的。

### 基本用法

Batch模式的最基本用法，是使用Session中的batchXXX系列的方法

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson9\BatchOperate.java

|  |
| --- |
| \*/  @Test  **public** **void** testBatchOperates() **throws** SQLException{  List<Person> persons=**new** ArrayList<Person>();  **for**(**int** i=0;i<5;i++){  Person p=**new** Person();  RandomData.*fill*(p); //填充一些随机值  persons.add(p);  }    { //批量插入  *db*.batchInsert(persons);    //批量操作下，从数据库获得的自增键值依然会写回到对象中。  **for**(**int** i=0;i<5;i++){  *assertEquals*(Integer.*valueOf*(i+1), persons.get(i).getId());  }  }  { //批量更新(按主键)  **for**(**int** i=0;i<5;i++){  persons.get(i).setGender(i % 2==0?'M':'F');  }  *db*.batchUpdate(persons);  }  {//批量更新 (按模板)  **for**(**int** i=0;i<5;i++){  Person p=persons.get(i);  p.getQuery().clearQuery();  p.getQuery().addCondition(Person.Field.*name*,p.getName());  p.setName("第"+(i+1)+"人");  }  *db*.batchUpdate(persons);  }  {//按主键批量删除记录  *db*.batchDeleteByPrimaryKey(persons);  }  } |

上例中，我们首先创建了5个Person对象，用批操作一次性插入到数据表。在单表操作中，我们了解到自增键值会在插入执行后立刻回写到对象中。在批操作中，对于自增主键的生成获取特性仍然有效。

然后我们对每个元素进行了修改，执行批量更新操作，可以发现五个元素在数据上都按主键为条件执行了更新操作。这一规则和之前的单表操作一致——在没有任何查询条件时，使用主键作为操作的where条件。

显然，一个自由的update语句并不一定是按主键操作的。因此例子接下来的演示了按自定义的条件去执行update。接下来，指定了name作为更新的条件。这样的用法产生的SQL如下——

|  |
| --- |
| update T\_PERSON set PERSON\_NAME = ? where PERSON\_NAME=? |

其效果是按“姓名”去更新”姓名”。我们已经知道，条件和对象是在不同的位置上的，因此这样的SQL语句是可以在EF-ORM中使用的。

最后我们将全部对象传入，并按主键批量删除所有记录。

上述例子中，直接使用了Session对象中一个简单的批操作API。Session对象中，和批操作相关的API有——

|  |  |
| --- | --- |
| 方法 | 用途 |
| Session.batchDelete(List<T>) | 按传入的操作，在数据库中进行批量删除。 |
| Session.batchDelete(List<T>, Boolean) | 按传入的操作，在数据库中进行批量删除。指定是否分组（为了支持分库分表） |
| Session.batchDeleteByPrimaryKey(List<T>) | 按传入的对象的主键，在数据库中进行批量删除。 |
| Session.batchInsert(List<T>) | 将传入的实体批量插入到数据库。 |
| Session.batchInsert(List<T>,Boolean) | 将传入的实体批量插入到数据库。指定是否分组（为了支持分库分表） |
| Session.batchInsertDynamic(List<T>, Boolean) | 将传入的实体批量插入到数据库（dynamic=true）。指定是否分组（为了支持分库分表） |
| Session.batchUpdate(List<T>) | 按传入的操作，在数据库中进行批量更新。 |
| Session.batchUpdate(List<T>, Boolean) | 按传入的操作，在数据库中进行批量更新。指定是否分组（为了支持分库分表） |
| Session.startBatchDelete(T, String) | 高级接口。获得批操作的Batch对象。Batch对象可以让开发者更多的干预和调整batch操作的参数。 |
| Session.startBatchInsert(T, String, boolean) | 高级接口。获得批操作的Batch对象。Batch对象可以让开发者更多的干预和调整batch操作的参数。 |
| Session.startBatchUpdate(T, String) | 高级接口。获得批操作的Batch对象。Batch对象可以让开发者更多的干预和调整batch操作的参数。 |

上述方法中dynamic是指动态插入。简单来说，就是在形成INSERT语句时，那些未赋值的字段不出现在SQL语句中。（从而可以使用数据库的DEFAULT值）。

值得解释的另一个参数是”指定是否分组“，大家回想前面的批操作限制，必须是单句SQL。这意味着当使用分库分表后，不同表的插入和更新操作不能合并为一句SQL执行。

为了满足这种场景，EF-ORM采用的办法是，将所有在同一张表上执行的操作合并成一个批。而位于不同表上的操作还是区分开，分次操作。显然，分析每个对象操作所影响的表是复杂的计算，会带来一定的开销。因此，在业务开发人员能保证所有操作请求位于同一张表上时可以传入false，禁止Batch对请求重新分组，从而带来更高效的批量操作。

这一示例详见第10章。

最后提一下，目前批操作全部都是单表操作。不支持级联特性。

### 不仅仅是操作实体

正如前面单表Criteria API中介绍的，批操作并非只能按主键update/delete数据。事实上在update和delete批操作时，传入的每个Entity都是一个完整的单表操作，其内嵌的Query对象可以描述一个通用的查询，而不是仅仅针对单条记录的。

从JDBC角度看，批操作是将一条SQL语句用多组参数执行。因此实际上批操作的每个元素都应该是一个SQL语句，而不仅仅是对一个实体（单条记录）的操作。

我们看下面这个批量update的例子

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson9\BatchOperate.java

|  |
| --- |
| @Test  **public** **void** testBatchUpdate() **throws** SQLException{  doInsert(5);  Person p1=**new** Person();  p1.getQuery().addCondition(QB.*matchAny*(Person.Field.*name*, "a"));  p1.prepareUpdate(Person.Field.*created*,*db*.func(Func.*current\_timestamp*));    Person p2=QB.*create*(Person.**class**).  addCondition(QB.*matchAny*(Person.Field.*name*, "b")).getInstance();    Person p3=QB.*create*(Person.**class**).  addCondition(QB.*matchAny*(Person.Field.*name*, "cc")).getInstance();    *db*.batchUpdate(Arrays.*asList*(p1,p2,p3));  } |

这段代码将引起数据库执行以下的SQL

|  |
| --- |
| update T\_PERSON  set CREATED = current\_timestamp  where PERSON\_NAME like ? escape '/' |

然后有三组参数，使用这条SQL语句来运行。

该SQL是由整个Batch的第一个元素决定的。 对于整个Batch来说，第一个操作请求决定了整个Batch要执行什么样的SQL语句。后面的操作对象**仅仅起到了传入参数的作用**。也就是说，在后面的对象中，无论增加什么样的其他条件，都不会影响整个批次的运行SQL。

上面的示例运行结果在日志中是这样显示的——

|  |
| --- |
| Update Batch executed:3/on 2 record(s) Time cost([ParseSQL]:2199us, [DbAccess]:38ms) |

这里清晰的指出了执行效果——有3组执行参数，共计修改了2行记录。

因此，如果说9.2.1节中体现的是传统ORM中批量操作实体行为，那么本节中进一步澄清了批操作的本质：

|  |
| --- |
| *以第一个传入的操作请求为模板，形成对应的SQL语句。*  *从所有操作请求中获取该SQL需要的操作参数，整批一起执行。* |

这种处理概念，使得我们能在Batch中处理多次SQL操作，而不仅仅是操作多个实体。

用另一个API来执行批操作可以让我们更明显的看出这一点

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson9\BatchOperate.java

|  |
| --- |
| @Test  **public** **void** testBatchUpdate2() **throws** SQLException {  doInsert(5);  Person query = **new** Person();  query.getQuery().addCondition(Person.Field.*created*, Operator.*GREAT*, DateUtils.*getDate*(2000, 1, 1));    List<Person> persons = *db*.select(query);  **for** (Person person : persons) {  person.setCreated(**new** Date());  }  Batch<Person> batch = *db*.startBatchUpdate(persons.get(0),**null**);  batch.execute(persons);  } |

这个例子中，我们查出所有创建日期在2000-1-1日之后的Person对象，将其创建日期设置为今天。

然后，startBatchUpdate()方法就是传入操作模板，形成Batch操作任务。之后将所有的person对象作为参数传入并执行。

### 性能的极限——Extreme模式

使用上面所说的Batch模式，并没有达到批量操作的最大性能，为了满足大量数据导入或导出（如ETL）等特殊场景，EF-ORM还提供了Extreme模式，批量插入或更新记录。

批量插入记录中，有哪些操作影响了性能的最大化呢？

1. Sequence获取访问。为了让用户能在完成插入操作后的对象中获得到其Sequence ID值，框架采用先访问一次数据库获得ID值，再执行插入的做法。批操作下，这一动作非常损耗性能。
2. 自增值回写，在一些自带自增列功能的数据库上，框架要将数据库端生成的自增列值写回到对象中。
3. 各个数据库往往会设计一些特殊的语法，来满足高性能写入等场合的需要。

Extreme模式就是针对上述情况所设计的，Extreme模式下，会放弃主键回写等一些不必要的功能，同时会启用数据库方言中的特殊语法，从而实现最大效率的数据插入和更新。

orm-tutorial\src\main\java\org\easyframe\tutorial\lesson9\BatchOperate.java

|  |
| --- |
| **private** **void** doExtremeInsert(**int** max) **throws** SQLException {  List<Person> persons = **new** ArrayList<Person>(max);  **for** (**int** i = 0; i < max; i++) {  Person p = **new** Person();  RandomData.*fill*(p); // 填充一些随机值  persons.add(p);  }  *db*.extremeInsert(persons,**false**);  } |

Extreme模式的使用很简单，使用session类中的 extremeInsert 和 extremeUpdate方法即可。

对用户来说，就这么简单，但如果您想进一步了解什么是Extreme模式，可以继续看下去。

在Oracle上，Extreme模式的数据插入性能改善最为明显。因为在Oracle的Extreme模式下，INSERT语句中会直接引用Sequence。如

|  |
| --- |
| INSERT INTO FOO (ID, NAME, REMARK) VALUES (S\_FOO.NEXTVAL, ?, ?) |

从而免去单独访问Sequence的开销。除此之外，当启用Extreme模式后，在Oracle数据库上会增加SQL Hint，变为

|  |
| --- |
| INSERT */\*+ APPEND \*/* INTO FOO (ID, NAME, REMARK) VALUES (S\_FOO.NEXTVAL, ?, ?) |

目的是进一步提升插入效率。

在别的数据库上，Extreme模式的性能差距会小一些，由于省去了ID回写功能，性能还是会有一定改善。extremeUpdate也是如此。后续的其他一些性能优化手段也会继续添加到Extreme模式上。

EF-ORM的Extereme模式插入数据的速度是非常快的。实际测试表明：在网络环境较好的Oracle数据库上，针对7~8字段（无LOB）的表。Extreme模式可以达到插入8~10万条/秒。在一般的百兆局域网环境，每秒插入记录数量也可以达到 5~6万/秒。Oracle下这一数值是普通Batch模式的10倍以上；是普通单条插入的千倍以上！

在其他数据库上，Extreme模式的性能也会有一定提高。

# 分库分表与数据路由

## 数据分片概述

### 什么是数据分片

一个生产系统总会经历一个业务量由小变大的过程，可扩展性成为了考量系统高可用性的一个重要衡量指标。在大数据时代，数据规模的支持成为衡量系统能力的重要指标。

在这个时代，几乎data sharding成为每个系统架构时需要考虑的重要问题。实时上，远在互联网和“大数据”概念时代之前，银行和电信业早已在处理数十亿数百亿级的数据，并且采用了传统的分区、分表、分库等手段，因此数据分片并不是什么新鲜概念。Oracle 9以前就提供分区表的功能，并且提供管理了TB以上单表的生产案例。在后来若干年中，支持数据分区已经成为各种商业数据库的标配特性。

数据库软件提供的数据分片方案，一般都是分区方案，根据分区条件，可以让数据存储在不同的表空间中。在DBA配置完分区规则后，整个分区的动作对应用程序是透明的。开发人员可以用普通的SQL语句来操作分区表，无须考虑分区的规则和数据位置。

商业数据库提供的原生数据分区方案，有不可替代的优越性——使用简单，性能较好，而且数据还会提供进一步的优化特性（如分区索引等）。最大的特点是，数据库原生的分区方案基本都能做到**对开发者透明**。

尽管数据库的供应商们已经为我们设计了众多的数据分片sharding方式；尽管在大数据时代各种分布式数据存储系统都提供了海量的存储规模。然而在大量的系统中，基于分库分表的RDBMS仍然占用重要的地位。在对实时性和一致性有高度要求的系统中，分库分表依然是处理海量结构化数据最重要的手段，并且为此也开发出了众多的中间框架。

为什么系统的开发者们不使用数据库软件本身提供分区功能，而要自行开发分库分表功能？一般有以下的理由——

* 使用数据库分区功能对数据库DBA的人力和技能要求。分区表后期的维护复杂性。
* 分成不同的表，有利于数据的归档、清除、备份等维护性行为。例如对交易记录和日志等，按历史表归档既简单又符合业务需求。
* 需要用多台数据库服务器的场合。RDBMS一般只提供分区功能，无法将分片到不同的实例上。
* 成本因素。不愿意花费购买商业数据库的分区特性或相关服务。
* 还是成本因素。使用了一些不支持分区功能的低成本数据库。
* 希望保留跨数据库的移植性。用户希望系统将来能移植到其他数据库软件上，无论这些软件是否支持分区功能。
* 习惯和守旧。沿用过去的做法，不愿意学习和接受数据库软件的新功能。

对于上面的几种因素，除了最后一种以外，我们都有在应用中自行实现数据分片的必要。分库分表依然是数据库分片中，成本最低，适应性最广，对开发者要求最低的方案。因此这种做法到今天为止也有着广泛的市场。但无论任何用户自行开发的或是框架提供的分表方案，都对操作有着诸多的限制，（例如不支持Join等），因此不能完全做到对开发者透明。

### EF-ORM支持的分片功能

EF-ORM的分库分表功能，能够通过配置数据分片规则，在操作时自动定位到数据所在的库和表上去。

EF-ORM支持数据的水平拆分和垂直拆分，在水平拆分维度上能同时支持多库和多表。

|  |
| --- |
| **水平拆分和垂直拆分**  *水平拆分就是将原本一张表的数据，按每条记录的特性分别存储到不同的数据表或数据库中。例如——*   * *将用户表拆成男用户表和女用户表* * *将日志表拆成每个月一张表* * *将某数据备份表拆成周一到周日，七张表并循环使用* * *按手机尾号的数字将用户拆分到10张表上。*   *水平分表后，拆开的表可以分布在同一个数据库上，也可以分布到不同的数据库上。水平分区在大型网站，大型企业应用中经常采用。 目的是出于海量数据分散存储，分散操作，分散查询以便提高数据处理量和整体数据处理性能。*  *垂直拆分，就是将不同的数据表存在到不同的数据库中。起到均衡负载的作用。解决多个表之间的IO竞争问题。但是垂直拆分不解决单表数据库过大引起的查询和修改效率等问题。* |

在水平分片的情况下，数据分库分表以后对操作有着巨大的限制——不支持分区表和其他表关联（Join）。因此一旦设置成分区表，这张表上只能执行单表操作了。跨分区查询、排序等特性也是分库分表后的难点。

EF-ORM的功能特色

1. 按需建表——对于按时间分表的场合，由于时间范围是无穷的，因此不可能实现创建所有的数据表。EF-ORM可以在需要用到某个时间点表的时候，为其创建表。
2. 范围分析——水平分区后，如果分区条件完整，那么各种框架都可以简单的定位到需要操作的分区表上。但如果分区条件不完整、或者只是一个区间范围，那么EF能够给予区间运算等逻辑，准确的计算出需要操作的表的范围。并且将操作局限在数量最少的表上。
3. 跨表查询——水平分区后跨表操作的场合下，EF-ORM能够通过调度改写为union语句、count数累加、顺序查询、结果集合并、结果集重排序等一系列操作模块，在内存占用极小的情况下完成对多表、多库查询乃至排序的需求。
4. 跨库查询——水平分区后跨库操作的场合下，可以对多库的不同表同时操作，影响记录数累加，Select语句的场合结果集可自动合并。
5. 双解析引擎。无论是用CriteriaAPI方式操作数据库，还是SQL方式操作数据库，都支持解析Query/SQL语句来分析分表条件，并路由到不同的数据源上进行操作。
6. 全自动。 用户只需要配置分表分库规则。在执行非多表关联操作时，分库分表引擎即自动生效。路由、多库操作、结果集处理其他无需操心。在多表/多库，数据库自增键值不能再使用数据库列自增了。此时包括Sequence/自增键生成规则变更等，也是均自动完成。

众所周知：水平拆分下，数据操作面临相当复杂的挑战——

1) distinct

因为需要遍历所有shard分区，并进行合并判断重复记录。

2) order by

类似 1)

3) aggregation

count，sim，avg等聚合操作先分散到分区执行，再进行汇总。

4) 跨分区查询时的多表Join

分区表Join非分区表。目前大部分分库分表方案都没有解决这个问题

备注：EF-ORM能支持水平拆分下的跨表和跨库操作

|  |  |  |
| --- | --- | --- |
| 查询操作 | 支持的操作 | 不支持的操作 |
| 跨表情况下(不跨库) | count  Distinct  Order by  Group by  Group by+having  Order by+分页  Group by+having+分页 | Join操作 |
| 跨库情况下 | Count  Order by  Distinct(有数据量限制)  Order by+分页(有数据量限制)  Distinct(有数据量限制)  Group by(有数据量限制)  Group by+having(有数据量限制)  Order by+分页(有数据量限制)  Group by+having+分页(有数据量限制) | Join操作 |

上述“有数据量限制”的特性，是指查询返回的结果数量不能过大。（并非指数据库中的记录数量）。因为上述操作都是不得不进行内存操作的功能，所以不建议在太大的结果集上进行操作。一般来说，OLTP类应用单笔业务影响的记录行数不超过1000条，因此EF-ORM的这种支持方案是能满足大部分需求的。但是，不建议在OLAP类应用中尝试用EF-ORM来解决跨库数据合并的问题。

### 和具有同类功能的产品比较

这里比较笔者有了解的Hibernate Shards和TDDL两个框架， Ibatis-Sharding没用过，有机会再补充。下面的各种论点均为采集互联网资料了解而得，可能有错漏、陈旧之处，请读者包含。

#### Hibernate Shards / HiveDB

Hibernate Shards是Google的工程师贡献的开源代码，集成到Hibernate中使用后可以支持数据水平拆分。HiveDB是基于HibernateShard 用来横向切分mysql数据库的开源框架。

这个解决方案目前通过 Criteria 接口的实现对 聚合提供了较好的支持， 因为 Criteria 以API接口指定了 Projection 操作，逻辑相对简单。

EF-ORM中的shards功能和上述框架比较类似，

1. 还有很多hibernate的API没有实现

2. 不支持cross-shard的对象关系，比如A、B之间存在关联关系，而A、B位于不同的shard中。hibernate shards提供了CrossShardRelationshipDetectingInterceptor，以hibernate Interceptor的方式来检测cross-shard的对象关系问题，但这个拦截器有一定的开销（比如需要查映射的元数据、有可能过早的触发延迟加载行为等），可以用于测试环境来检测cross-shard关系的问题

3. hibernate shards本身不支持分布式事务，若要使用分布式事务需要采用其他解决方案

4. hql、criteria存在不少限制，相比于hql，criteria支持的特性更多一些

5. Session或者SessionFactory上面有状态的拦截器，在shard环境下面会存在一些问题。拿session来说，在hibernate中拦截器是对单个session上的多次sql执行事件进行拦截，而在shard情况下hibernate shards的ShardedSession会对应每个shard建立一个session，这时拦截器就是跨多个session了，因此hibernate shards要求有状态的拦截器必须通过实现StatefulInterceptorFactory来提供新的实例。如果拦截器需要使用到目标shard的session，则必须实现hibernate shards的RequiresSession接口

EF相较于这两个框架，其优势是——

1. 提供了较好的分区范围分析算法。而HibernateShard依赖于用户自行实现分区策略的计算。而当分区条件含糊时，用户很难编写出精确的路由算法。
2. 针对分表和分库的情况加以区分，在同个数据库上的时候能利用SQL操作实现排序和聚合计算，对服务器的CPU和内存压力较小。而HibernateShard不区分这两种情况。
3. 优化的多库排序： 在多库排序时，能分析分表规则，当分表条件和排序条件一致时，直接将各个结果集按排序条件拼合。免去了排序的性能开销。  
   在必须重排序时，利用每个库各自的顺序，使用了内存占用较小的排序算法。
4. EF-ORM中分区操作对用户基本透明，无需移植。而从hienrate移植到HibernateShard时的部分接口类需要调整。
5. hibernate shards没有对hql进行解析,因此hql中的count、sum、distinct等这样的操作还无法支持。而EF-ORM除了API层面外，对于传入的SQL语句也支持分库分表。
6. 配置更简单，业务侵入性更小，对开发人员透明度更高。EF-ORM中除了Entity上需要加入少量注解外，开发者无需关心任何分库分表相关的工作，也无需编写任何代码。hibernate shards等则需要开发者实现若干策略。
7. EF-ORM的主键生成算法也做到对用户透明，在支持Sequence下的数据库中，会使用Sequence生成自增序列。其他数据库上会使用具有一定步长的表来生成Sequence。
8. Hibernate shards不支持按时间或实际使用时建表。

EF-ORM其功能基本包含了Hibernate Shards / HiveDB。但还有一定不足——

1. 没有提供并行查询策略。目前多次查询的场合都是顺序操作。
2. virtual shards，虚拟分片的作用是将数据分得更细，然后多个虚拟片公用一个实际的表。这样的好处是virtual shards和实际shards之间的映射关系还可以后续调整。
3. Hibernate shards的诞生和发展周期更长，功能完善程度更高，包括相同维度切分的实体之间的级联关系等也都做了处理。

EF-ORM和上述框架共有的不足之处是——

1. 两者都绑定特定的ORM实现，前者绑定Hibernate，后者是EF-ORM的一部分。

#### Alibaba TDDL

TDDL（Taobao Distributed Data Layer）顾名思义，是淘宝网的分布式数据框架。它和众多的连接池一样，是一个封装为DataSource的中间框架，也能处理SQL语句的分析和路由。

应该说，TDDL和本框架之间关注内容和所属的层次是不同的，TDDL的处理是更为底层的：

1. 是在JDBC规范下，以DataSource方式进行封装的。TDDL是对SQL语句进行分析和处理的，而不是Criteria对象上进行处理的。这使得TDDL能拦截一切数据库操作，但也使得复杂场景下的分库分表路由支持变得困难。路由条件的传递除了解析SQL之外，TDDL中还为此开了不少“后门”来传递路由条件。
2. 数据路由只是TDDL的一部分功能，TDDL本身还提供了SQL重试、负载均衡等一系列提高可用性的模块。正如官方材料中说的，你可以使用TDDL的分库分表功能，也可以不使用。而仅仅使用其底层封装的高可用性模块。

TDDL目前最大问题是，其“开源程度”不高，所谓的“开源程度”问题，主要是指

1. Github上，该项目只有2012年更新过，仅有4次代码提交。而阿里内部，该软件的版本则一直在升级改进，但并未对外公开。
2. TDDL升级改进的需求几乎均来自阿里内部，其开发团队几乎未对外部用户和开源社区做出什么响应。
3. TDDL中最关键的分库分表层的模块代码一直没有在开源项目中出现。遭到不少网友的疑问。
4. 几乎没有社区支持，官方也没有文档支持。仅有的少量文档都是阿里内部流出，其对应的版本和代码不明。

TDDL 3.3.x的代码最终还是找到了，我也读过了，感觉官方开发团队为了解决JDBC庞大的API支持投注的精力十分可观，因此对分库分表等支持上多少不够专注。

从我的感觉来说，TDDL最实用的地方恰恰不是其分库分表功能，而是——

1、数据库主备和动态切换；

2 、带权重的读写分离；

3 、单线程读重试；

4 、集中式数据源信息管理和动态变更；

5 、可分析的日志打印 , 日志流控，动态变更；

这些提高应用可用性、可维护性、伸缩性方面的特性。

由于TDDL的功能和EF-ORM的功能仅仅有少量是重叠的（分库分表），因此我们仅能对重叠部分进行一些功能的比较。

TDDL的开源版本（两年前的版本）分库分表功能相较于目前EF-ORM其问题是——

1. TDDL 需要依赖 diamond 配置中心
2. TDDL专注于支持MySQL等内部需要使用的场景，对其他数据库下的解析和处理支持有限。
3. 不支持从Between……and中解析分库条件，不支持SQL中的NOT语义。
4. TDDL目前还不支持任何种类的注释。TDDL目前还不支持强制指定使用排他锁的方式。
5. 不支持从当前数据库已有的分表中扫描存在的分表作为决策。
6. 不支持按时间或实际使用时建表。
7. TDDL对聚合函数支持较差，如Count,avg,max,min等还不支持，称在以后的版本中会对出现在column列名字段的聚合函数予以支持。但不会对group by 中having后出现的聚合函数加以支持。
8. TDDL不支持 Having以及having内使用的其他聚合函数。
9. 事务的有限支持。支持基于单库的事务，但不支持跨库进行事务。

上述是从目前网上了解到的对开源版本的TDDL的一些对比。从流出的TDDL 3.3.x版本代码来看，上述许多功能应该是是已经支持的。但具体情况因缺少支持文档不明。

## 分库分表规则配置

本章，我们开始了解EF-ORM的分库分表功能使用。

### 水平拆分——分表

首先，我们需要了解分库分表的配置。默认情况下，分库分表规则可以通过Entity类上标注的注解来描述。

orm-tutorial\src\main\java\org\easyframe\tutorial\lessona\entity\OperateLog.java

|  |
| --- |
| @Entity  @PartitionTable(key = {  @PartitionKey(field = "created", function = KeyFunction.*YEAR\_MONTH*)  })  **public** **class** OperateLog **extends** DataObject {  @Id  @GeneratedValue(strategy = GenerationType.*IDENTITY*)  **private** **long** id;  @GeneratedValue(generator = "created-sys")  **private** Date created;  @Column(name="message",length=500)  **private** String message;  **public** **enum** Field **implements** jef.database.Field {  *id*, *created*, *message*  } |

上例中的注解@PartitionTable就是分库分表规则了。

每个@PartitionKey表示一个分表条件字段。上例中的配置表示——取对象的createDate字段，取其年+月（yyyyMM格式），作为表名的后缀。

对于上面这样配置的日志表。2014-1-31日的日志就会存放在 operatelog\_201401表中，2014-2-1日的日志就会存放在operatelog\_201402表中。

如果插入前createDate字段没有值，那么意味着无法决定记录要插到哪张表中，此时会抛出异常。

当分库条件不明确时，路由计算器会尝试计算一个范围来覆盖记录所在的表中，因此分库分表的条件含糊可能引起路由返回结果为多张表、甚至多个数据库。

对于几种操作情形，路由返回结果为多张表时，EF-ORM将会采取不同的策略来执行。

|  |  |  |  |
| --- | --- | --- | --- |
| 操作 | 单库单表 | 单库多表 | 多库多表 |
| insert | 支持 | 抛出异常 | 抛出异常 |
| update | 支持 | 在所有表上执行update | 在所有表上执行update |
| delete | 支持 | 在所有表上执行delete | 在所有表上执行delete |
| select | 支持 | 尝试转换为union all语句。配置一定结果集处理。 | 对每个库上的多表采用 union all。  对多个库的结果集进行混合计算。 |
| count | 支持 | 如果不涉及distinct、group by则在所有表上执行count查询，并将结果累加。  如果涉及distinct、group by则查询结果并计算将SQL修改为在union all外部嵌套distinct和group by，最后计算总数。 | 如果不涉及distinct、group by则在所有表上执行count查询，并将结果累加。  如果涉及distinct、group by则查询结果并汇总混合。最后得到结果的数量。（数量有限制） |

我们再看下面一个例子，有一张表，其中有两个字段name和created，我们希望以这两个字段为关键字进行分表操作。我们可以这样定义。

|  |
| --- |
| @PartitionTable(key = {  // 分区关键字1为name字段，取头3个字符  @PartitionKey(field = "name", length = 3),  // 分区关键字2为created字段(日期型，取其月份数，长度不足2则补充到2)，  @PartitionKey(field = "created", function = KeyFunction.*MONTH*, length = 2) })  @Table(name = "PEOPLE\_TABLE")  @Entity  **public** **class** PeopleTable **extends** DataObject {  **private** **int** id;  **private** String name;  **private** Date created; |

对于这种配置，其分表条件有两个。如果用以下代码插入记录到数据库——

|  |
| --- |
| PeopleTable entity=**new** PeopleTable();  entity.setId(1);  entity.setCreated(DateUtils.*getDate*(2011, 2, 1));  entity.setName("ZHANG");  db.insert(entity); |

实际上该记录会写入到表 *PEOPLE\_TABLE\_ZHA02*中。其中ZHA是第一个分表条件产生的，02是第二个分表条件产生的。多个分表条件之间默认没有任何分隔符，如需要可以用keySeparator参数配置。

当查询数据时：

|  |
| --- |
| PeopleTable entity=**new** PeopleTable();  entity.getQuery().addCondition(PeopleTable.Field.*name*, "WANG");  entity.getQuery().addCondition(PeopleTable.Field.*created*, **new** Date());//假设现在是3月份  List<PeopleTable> data= db.select(entity); |

查询时的表名也是根据条件自动计算的，实际上会到一张名为 PEOPLE\_TABLE\_WAN03的表中查询。

从上面的配置我们可以了解到通过@PartitionTable注解，我们可以配置一些规则，这些规则将在表名后面添加若干后缀，作为一张分表。

@PartitionTable的可配置项包括

|  |  |  |
| --- | --- | --- |
| 属性 | 作用和举例 | 缺省值 |
| appender | 描述原始表名和分区后缀之间的分隔符 | "\_" |
| keySeparator | 描述多个KEY之间的分隔符 | “” |
| key | 多个分区条件。多值@PartitionKey | 无 |
| dbPrefix | 如果有配置了数据库名的分区条件时，数据库名的前缀 | “” |

@PartitionTable的可配置项

@PartitionTable的key属性中，我们可以配置多个@PartitionKey。

@PartitionKey的可配置项包括

|  |  |  |
| --- | --- | --- |
| 属性 | 作用和举例 | 缺省值 |
| field | 指定对应的字段(分表分库规则的依据字段) | 无 |
| function | 指定字段上函数(一般是对日期进行处理或者数值取摸)  函数可以从jef.database.routing.function.KeyFunction类中枚举。  KeyFunction类中定义了各种日期时间函数和数字取模函数。 | RAW |
| functionClass | 指定一个自定义的字段函数。  例如，配置为ModulusFunction.class，便表示对指定的分表字段进行取模运算。  此参数经常需要和functionClassConstructorParams注解同时使用，用于提供构造处理函数所需的构造参数。  此注解不可与function注解同时使用。  一个例子是这样的：  @PartitionKey(  field ="amount", length = 2,  functionClass=ModulusFunction.class,  functionClassConstructorParams={"5"} )  这个例子表示，对amount字段，按5取模，然后补充到两位数后作为分表后缀名。 | 无 |
| functionConstructorParams | 和function或者functionClass同时使用，描述functionClass构造时的构造参数。配置时只能是文本，使用时会自动转换。 | {} |
| isDbName | 当此值设置为true时，当前ParitionKey计算出来的字符串不是作为分表名称的一部分，而是作为一个独立的数据库名。  这种用法用于当应用部署在多个独立的数据库上时，可以实现跨库的数据库操作  每次数据库操作可以通过这个对象得到其他的数据库的连接，从而实现跨库的数据库操作。 | false |
| defaultWhenFieldIsNull | 当指定字段无值时，使用的缺省值 | “” |
| length | 指定用作表名后缀的字符串长度，0表示不限制。当长度限定时，如果不足会填充。如果超过会截断 | 0 |
| filler | 当表名后缀的字符串长度不足时，在左侧填充的字符。 | ‘0’ |

@PartitionKey的可配置项

上述配置可以描述对象的分表规则。一旦配置了分表规则后，就可以像操作正常的实体一样进行CRUD操作。框架会自动解析对象中的数值和Query中的条件，来判断需要操作哪些分表。

### 水平拆分——分库

上面的配置除了可以实现分表配置外，还可以配置出水平分库的实体。下面例举了一个既分库又分表的实体。

customer表可以按客户编号(customerNo)，按3取模。然后补到两位作为数据库名。我们可以推测出根据customerNo的不同，记录会分别落到D0, D1,D2这样三个名称的数据库上。

同时，按照createDate取年+月进行分表。

|  |
| --- |
| @Entity  @PartitionTable(key = {  @PartitionKey(field = "createDate",function=KeyFunction.*YEAR\_MONTH*),  @PartitionKey(field = "customerNo",functionClass=ModulusFunction.**class**,  functionConstructorParams="3",isDbName=**true**,filler='D',length=2)  })  **public** **class** Customer **extends** DataObject {  /\*\*  \* 客户编号  \*/  **private** **int** customerNo;  /\*\*  \* 出生日期  \*/  **private** Date DOB;  /\*\*  \* 死亡日期  \*/  **private** Date DOD;  /\*\*  \* 名  \*/  **private** String firstName;  /\*\*  \* 姓  \*/  **private** String lastName;  /\*\*  \* 电子邮件  \*/  **private** String email;  /\*\*  \* 记录创建日期  \*/  **private** Date createDate;  **public** **enum** Field **implements** jef.database.Field {  *customerNo*, *DOB*, *DOD*, *firstName*, *lastName*, *email*, *createDate*  } |

包含分库分表条件的Entity。

根据上面的配置，我们可以推测出——

customerNo=1000，createDate=2014-1-3的记录，将会存放到D1数据库的customer\_201401表中。

customerNo=1001，createDate=2014-2-1的记录，将会存放到D2数据库的customer\_201402表中。

customerNo=1002，createDate=2014-2-1的记录，将会存放到D0数据库的customer\_201402表中。

可能会有这样的疑问：我在开发的时候怎么能知道部署的时候会有几个数据库呢？怎么能确定部署时的数据库叫什么名字呢？在这里配置的数据库名其实也是一种虚拟数据源。实际使用时，可以在jef.properties中配置映射关系。

|  |
| --- |
| db.datasource.mapping=D0:mysql01,D1:mysql02,D2:mysql03 |

上面的配置，可以将开发时指定的虚拟数据源D0,D1,D2映射到实际部署环境的mysql01,mysql02,mysql03三个数据库上。

### 垂直拆分

数据垂直拆分是一个更简单的做法，也就是某类数据（某些表）被单独放到别的数据源上。

orm-tutorial\src\main\java\org\easyframe\tutorial\lessona\entity\Person2.java

|  |
| --- |
| /\*\*  \* 垂直拆分的实体。  \* 所谓数据垂直拆分，意思是将一类表放到不同的数据库上。从而降低负载。  \*/  @Entity  @BindDataSource("datasource2")  **public** **class** Person2 **extends** DataObject {  @Id  @GeneratedValue(strategy=GenerationType.*IDENTITY*)  **private** **int** id;  @Column(length=64)  **private** String name;  @Column(name="DATA\_DESC",length=255)  **private** String desc;  @GeneratedValue(generator="created")  **private** Date created;  @GeneratedValue(generator="modified")  **private** Date modified; |

一旦进行了上述配置则Person2表将始终保存到数据源datasource2上。前面已经说过，在实际部署环境中，可以通过db.datasource.mapping参数将数据源名称映射为部署环境的真实数据库名。

## 分库分表后的操作和行为

### 前提与原则

分库分表以后，有以下几个最基本的限制和规则

1. 不支持多表查询。 所有的单表查询Query API都能支持分库分表。所有的多表查询则都会**忽略**分库分表的配置规则。  
    在NativeQuery中运行单表查询的SQL语句，都能支持分库分表规则。而多表的SQL语句则都不支持。
2. 原生SQL操作方式由于对SQL是不作解析的，因此也就不支持分库分表。
3. 目前分库分表功能支持多库多表下的常用聚合函数、Distinct、Order by、Start with... Connect by、分页。
4. 目前分库分表功能不能支持分析函数、不支持聚合函数中的avg函数。
5. 使用API方式，分库分表前后的数据库操作方式没有任何差异。
6. 使用NativeQuery方式，要启用分库分表功能。需要调用NativeQuery的withRouting()方法(见后文示例)。

在对分库分表的操作有了上述认识后，可以看下面的特性和示例。

### 按需建表

我们来看分库分表后的实际操作例子。

orm-tutorial\src\main\java\org\easyframe\tutorial\lessona\Case1.java

|  |
| --- |
| /\*\*  \* 测试在分库后的表中插入一条记录  \*/  @Test  **public** **void** createTest() **throws** SQLException {  OperateLog log = **new** OperateLog();  log.setCreated(DateUtils.*getDate*(2010, 3, 2));  log.setMessage("测试！！");  *db*.insert(log);  } |

在上例中，我们可以发现，实际运行SQL语句为

|  |
| --- |
| insert into OPERATELOG\_20100302(ID,CREATED,MESSAGE) values(?,?,?) |

但是在插入之前，表OPERATELOG\_20100302还根本不存在。此时框架会自动创建这张表，并将记录插入到这张表中。

### 自增主键生成行为

我们在仔细看上面的例子还会发现更多的东西。比如，在对象中，我们配置了@GeneratedValue(strategy=GenerationType.*IDENTITY*)。 正常情况下将会使用数据库表列的自增功能来生成主键。但是多表下问题就来了，不同的表可能会生成重复的ID。因此我们可以看到框架自动将主键生成的行为变更了，在Derby上，采用TABLE的方式来生成自增主键值……打印出的SQL日志如下——

|  |
| --- |
| SELECT V FROM global\_sequences WHERE T='OPERATELOG\_SEQ' | [derby:DB@1]  UPDATE global\_sequences SET V=? WHERE V=? AND T='OPERATELOG\_SEQ' |[derby:DB@1]  (1): [60]  (2): [40] |

显然到TABLE中逐个获取序号，效率是非常低下的，因此EF-ORM采用了以下两个手段来保证效率——

1. 一次获取多个序号，上例中，一次SQL操作直接将序号递增20。即一次获取20个序号。
2. CAS(ComapreAndSwap)操作。上例的UpdateSQL语句是支持并发的，一旦有多个SQL同时操作该表，能保证每个SQL都能正确获取序号值。

EF-ORM中，用于调节Sequence获取行为的参数还有不少。比如TABLE方式，可以每个表一个TABLE，也可以全局公用一个TABLE。TABLE下的Sequence等也都可以调节。详见附录一 配置参数一览。

### 手动建表

DbClient中提供了若干数据库表维护的API，使用这些API，可以在数据库中创建表。

对于启用了分库分表后的情况，我们可以用下面的测试代码试验一下。

orm-tutorial\src\main\java\org\easyframe\tutorial\lessona\Case2.java

|  |
| --- |
| //现在删表，删表时会自动扫描目前存在的分表。  *db*.dropTable(Customer.**class**, Device.**class**, OperateLog.**class**, Person2.**class**);  System.*err*.println("现在开始为Customer对象建表。Customer对象按“年\_月”方式分表，并且按customerNo除以3的余数分库");  *db*.createTable(Customer.**class**);  System.*out*.println();  System.*err*.println("现在开始为Device对象建表。Device对象按“IndexCode的头两位数字”分表，当头两位数字介于10~20时分布于ds1；21~32时分布于ds2；33~76时分布于ds3；其他情形时分布于默认数据源");  *db*.createTable(Device.**class**);  System.*out*.println();    System.*err*.println("现在开始为OperateLog对象建表。OperateLog对象按“年\_月\_日”方式分表，不分库");  *db*.createTable(OperateLog.**class**);  System.*out*.println();    System.*err*.println("现在开始为Person2对象建表。Person2对象是垂直拆分，因此所有数据都位于datasource2上。不分表");  *db*.createTable(Person2.**class**);  System.*out*.println();  System.*err*.println("======= 建表操作完成，对于分区表只创建了可以预测到的若干表，实际操作中需要用到的表会自动按需创建========="); |

**Drop时，扫描数据库中存在的表**

在不指定特定的分库分表时，dropTable方法会检索数据库中所有存在的分表加以删除。

**Create时，预测需要的表来创建**

而createTable时，框架会尽量预测需要创建的分表来尝试创建。

预测的方法是——

对于取模类函数，将取模可能的值进行遍历。如按10取模，那么会预测到0~9的所有数字。

对于日期类的函数，以当前日期为界，创建前后一段时期内的表（这个时期范围可以通过jef.properties中的全局配置参数），如果配置下面这样，表示选取过去三个月和未来三个月。

|  |
| --- |
| partition.date.span=-3,3 |

要注意的是，上述参数不仅仅用于建表时表名的推测，也用于执行数据库操作时的分表范围推测。比如用户查询OperateLog数据时，没有指定时间范围，那么框架默认就查询距今前后三个月的表。不会再查询三个月之前的表。

上述方法都是手工创建一个分区表的所有分表。我们也可以手工指定分库分表条件，来创建某张特定的分表。

orm-tutorial\src\main\java\org\easyframe\tutorial\lessona\Case2.java

|  |
| --- |
| @Test  **public** **void** ddlTest() **throws** SQLException{  Customer customer=**new** Customer();  customer.setCustomerNo(1234);  customer.setCreateDate(DateUtils.*getDate*(2016,12,10));  *db*.createTable(customer);  *db*.dropTable(customer);      Device d=**new** Device();  d.setIndexcode("123456");  *db*.createTable(d);  *db*.dropTable(d);  } |

上述方法中，由于指定了分表条件，因此不会再像上例一样，在所有可能的数据库上创建分表，而是会在特定的datasource上创建和删除 CUSTOMER\_201612、 DEVICE\_1表。

### 路由时过滤不存在的表

对于未明确指定分库分表条件的查询，框架会查询所有可能有此记录的表。这种“可能性”分析并不仅仅局限于函数和枚举操作，还包括对数据库现存的表的扫描。因此正常情况下，数据库中存在的表才会被查询到。数据库中不存在的表，将不会被查到。

orm-tutorial\src\main\java\org\easyframe\tutorial\lessona\Case2.java

|  |
| --- |
| /\*\*  \* 当分表结果计算后，对比数据库中实际存在的表。不存在的表不会参与查询。  \*  \* **@throws** SQLException  \*/  @Test  **public** **void** testTableFilter() **throws** SQLException {  Device d=**new** Device();  d.setIndexcode("665565");  *db*.createTable(d);  ThreadUtils.*doSleep*(1000);  System.*out*.println("第一次查询，表Device\_6存在，故会查询此表");  Query<Device> query = QB.*create*(Device.**class**);  List<Device> device=*db*.select(query);    *db*.dropTable(d);  ThreadUtils.*doSleep*(500);  System.*out*.println("第二次查询，由于表Device\_6被删除，因此不会查此表");  device=*db*.select(query);  } |

前面提到的按需建表功能也好，此处的过滤不存在的表功能也好，都是对现有数据库中表是否存在进行判断。显然这个操作是较为耗时的，不可能每次计算路由时临时计算。因此，数据库中现存的表的情况，是被缓存下来的。

您可以关闭不存在表过滤的功能。通过jef.properties中配置

|  |
| --- |
| partition.filter.absent.tables=false //关闭表过滤功能。 |

同时，分表数据缓存的有效期可以设置

|  |
| --- |
| db.partition.refresh=3600 //设置分表信息缓存的最大生存时间。 |

### 如果路由结果为空

如果分表计算发现数据库中不存在符合条件的表，那么实际上就无需查询任何表，像下面这种情况。

orm-tutorial\src\main\java\org\easyframe\tutorial\lessona\Case2.java

|  |
| --- |
| /\*\*  \* 当分表结果计算后，发现没有需要查询的表的时候，会直接返回  \*/  @Test  **public** **void** testNoMatchTables() **throws** SQLException {  Query<Device> d = QB.*create*(Device.**class**);  //当Device\_9表还不存在时，这个查询直接返回空  **if**(!*db*.getMetaData(**null**).existTable("DEVICE\_9")){  d.addCondition(Device.Field.*indexcode*, "9999999");  List<Device> empty=*db*.select(d);  *assertTrue*(empty.isEmpty());  }  } |

上例中，本来按分表条件应该查询DEVICE\_9表。但是该表不存在，因此路由模块会返回空。此时框架直接返回空列表，不作任何查询。

### 垂直拆分的场景

垂直拆分就是将Entity用@BindDataSource绑定到另外一个数据库上去。实际使用的演示可以参考示例代码。

orm-tutorial\src\main\java\org\easyframe\tutorial\lessona\Case2.java

|  |
| --- |
| /\*\*  \* 当采用垂直拆分时， Person2的所有操作都集中在DB2上。而不是在默认的DB上操作  \*/  @Test  **public** **void** test1() **throws** SQLException {  // 插入  Person2 p2 = **new** Person2();  p2.setName("Jhon smith");  *db*.insert(p2);  // 查询  Person2 loaded = *db*.load(Person2.**class**, p2.getId());  // 更新  loaded.setName("Kingstone");  *assertEquals*(1, *db*.update(loaded));  // 查询数量  *assertEquals*(1, *db*.count(QB.*create*(Person2.**class**)));  // 删除  *assertEquals*(1, *db*.delete(loaded));  // 删除后  *assertEquals*(0, *db*.count(QB.*create*(Person2.**class**)));    System.*out*.println("=============== In Native Query ============");  {  String sql="insert into person2(name,data\_desc,created,modified) values(:name,:data,sysdate,sysdate)";  NativeQuery query=*db*.createNativeQuery(sql, Person2.**class**).withRouting();  query.setParameter("name", "测试111");  query.setParameter("data", "备注信息");  query.executeUpdate();  }  {  String sql="select \* from person2";  NativeQuery<Person2> query=*db*.createNativeQuery(sql, Person2.**class**).withRouting();  query.getResultCount();    query.getResultList();  }  {  String sql="update person2 set name=:name where id=:id";  NativeQuery query=*db*.createNativeQuery(sql).withRouting();  query.setParameter("id", 1);  query.setParameter("name", "测试222");  query.executeUpdate();  }  {  String sql="delete person2 where id=:id";  NativeQuery query=*db*.createNativeQuery(sql).withRouting();  query.setParameter("id", 2);  query.executeUpdate();  }  } |

上例分为两部分。第一部分是基于API的数据库操作。第二部分是基于SQL语句的数据库操作。

运行上面的案例，可以发现基于垂直分区的配置，所有的数据库操作都在DB2上运行，而不是在默认的数据库上运行。

可以发现，在垂直拆分的情况下，API方式操作和以前没有任何变化。而使用SQL语句NativeQuery方式的时候，需要调用一下withRouting()方法。

### 在分库分表条件基本确定的场景下

我们以customer表为例，看一下在customer表上执行增删改查操作的情况。Customer对象按customerNo除以3的余数，分配到三个数据库上。同时customer还按照记录的创建时间进行了分表。

orm-tutorial\src\main\java\org\easyframe\tutorial\lessona\Case2.java

|  |
| --- |
| /\*\*  \* Customer对象按创建记录created字段时的“年\_月”方式分表，并且按customerNo除以3的余数分库  \*/  @Test  **public** **void** testCustomer() **throws** SQLException {  // 插入  Customer c = **new** Customer();  c.setLastName("Hession");  c.setFirstName("Obama");  c.setDOB(DateUtils.*getDate*(1904, 3, 4));  c.setDOD(DateUtils.*getDate*(1991, 7, 12));  c.setEmail("obama@hotmail.com");  *db*.insert(c);  /\*  \* 这个案例中，用作分库分表的customerNo和created这两个字段都没赋值，但是记录却成功的插入了。  \* 因为customerNo是自增值，created是系统自动维护的当前时间值。系统自动维护了这两个字段，  \*从而使得分库分表的路由条件充分了。  \* 此外，注意自增值的生成方式变化了。在本例中，使用TABLE来实现自增值。因为Derby不支持Sequence，  \*所以采用TABLE方式生成自增值。(实际上支持，但兼容性有些问题，故关闭了)  \*/  Customer c2 = **new** Customer();  c2.setLastName("Joe");  c2.setFirstName("Lei");  c2.setDOB(DateUtils.*getDate*(1981, 3, 4));  c2.setEmail("joe@hotmail.com");  c2.setCreateDate(DateUtils.*getDate*(2013, 5, 1));  System.*out*.println("====将会按需建表====");  *db*.insert(c2);  /\*  \* 这里指定记录创建时间为2013-5-1，也就是说该条记录需要插到表 CUSTOMER\_201305这张表中。  \* 这张表虽然现在不存在，但是EF-ORM会在需要用到时，自动创建这张表。  \*/  // 路由维度充分时的查询  System.*out*.println("当分表条件充分时，只需要查一张表即可...");  Customer loaded1 = *db*.load(c);    // 查询。  System.*out*.println("当分表条件不完整时，需要查询同一个库上的好几张表。");  Customer loaded = *db*.load(Customer.**class**, c.getCustomerNo());  // 更新  loaded.setLastName("King");  *assertEquals*(1, *db*.update(loaded));  // 查询总数，由于有没分库分表条件，意味着要查询所有库上，一定时间范围内的所有表  *assertEquals*(1, *db*.count(QB.*create*(Customer.**class**)));  // 删除  *assertEquals*(1, *db*.delete(loaded));  // 删除后  *assertEquals*(0, *db*.count(QB.*create*(Customer.**class**))); |

上述代码中，展现了对customer对象的基本操作，包括插入，加载、更新、删除、计算总数等。对照日志中输出的SQL语句，可以发现当对象中createDate和customerNo两个字段均有值时，操作会准确的定位到一个库的一张表上。

但是在按customerNo查询时，由于缺少createDate这个分表条件，框架不得不在前后几个月的表中就近进行查询。而当不带任何条件查询总数时，框架在所有库上的多张表中进行了查询。

### Batch模式和分库分表

分库分表对Batch模式也有一定的影响。由于分库分表后，不同库和表上的数据操作SQL语句是不一样的，因此框架必须对Batch中的对象进行分组，将同一张表里的记录划为一组，然后再对每组进行操作。

orm-tutorial\src\main\java\org\easyframe\tutorial\lessona\Case2.java

|  |
| --- |
| /\*\*  \* 演示Batch操作在分库分表后发生的变化  \*/  @Test  **public** **void** testCustomerBatch() **throws** SQLException {  List<Customer> list = **new** ArrayList<Customer>();  // 批量插入  Customer c = **new** Customer();  c.setLastName("Hession");  c.setFirstName("Obama");  c.setDOB(DateUtils.*getDate*(1904, 3, 4));  c.setDOD(DateUtils.*getDate*(1991, 7, 12));  c.setEmail("obama@hotmail.com");  list.add(c);  c = **new** Customer();  c.setLastName("Joe");  c.setFirstName("Lei");  c.setDOB(DateUtils.*getDate*(1981, 3, 4));  c.setEmail("joe@hotmail.com");  c.setCreateDate(DateUtils.*getDate*(2013, 5, 1));  list.add(c);  c = **new** Customer();  c.setCustomerNo(4);  c.setLastName("Yang");  c.setFirstName("Fei");  c.setDOB(DateUtils.*getDate*(1976, 12, 15));  c.setEmail("fei@hotmail.com");  c.setCreateDate(DateUtils.*getDate*(2013, 5, 1));  list.add(c);    System.*out*.println("当Batch操作遇到分库分表——\n所有记录会重新分组后，再进行批量插入。");  *db*.batchInsert(list);  // 跨表跨库的搜索  Query<Customer> query = QB.*create*(Customer.**class**);  query.addCondition(QB.*between*(Customer.Field.*createDate*, DateUtils.*getDate*(2013, 4, 30), DateUtils.*getDate*(2014, 12, 31)));  // 由于条件只有一个时间范围，因此会搜索所有可能出现记录的表。包括——三个数据库上所有时段在13年4月到14年12月的表。理论上所有可能的组合。  // 但由于大部分表不存在。EF-ORM只会查找实际存在的表，实际上不存在的表不会被查找。  list = *db*.select(query);  *assertEquals*(3, list.size());  // 批量更新  **for** (**int** i = 0; i < list.size(); i++) {  list.get(i).setEmail("mail" + i + "@hotmail.com");  }  // 虽然是Batch更新，但实际上所有记录分散在不同的库的不同的表中，重新分组后，每张表构成一个批来更新  *db*.batchUpdate(list);  // 批量删除  *db*.batchDelete(list);  } |

上面的测试案例，演示了分库分表后的Batch操作。一般来说，用户也无需关心分库分表后Batch操作上的变化。框架会自动按照重新分组的方式进行batch操作。

但是也有特殊情况。分库分表路由计算毕竟是个一定耗时的操作。某些场景下，用户能在自己的业务中保证batch中的所有数据都是在一个表上的，此时对每条记录进行路由计算就显得多余了。这时就可以使用session中另一个batch操作的方法，即传入一个Boolean值，表示是否需要按路由重新分组。

|  |
| --- |
| db.batchDelete(list,**true**); //需要对每个传入数据进行路由计算并重新分组  db.batchInsert(list,**false**); //不需要对每个传入数据进行路由计算  db.extremeInsert(list, **true**);//需要对每个传入数据进行路由计算并重新分组  db.extremeInsert(list, **false**);//不需要对每个传入数据进行路由计算 |

### 当路由结果为多库多表时

那么，现在我们来看一看分库分表查询中最为复杂的情况。即路由条件不足，造成多库多表查询的情况。该案例较为复杂，也非常长，因此代码不全部贴出了。可以参见示例工程下的java代码。

orm-tutorial\src\main\java\org\easyframe\tutorial\lessona\Case2.java

|  |
| --- |
| /\*\*  \* 演示若干复杂的跨库Cirteria操作。这些操作虽然会涉及多个数据库的多张表，但在框架封装下，几乎对开发者完全透明。  \*  \* 案例说明：<br>  \* 1、用批模式插入50条记录，这些记录将会分布在三个数据库的10张表中。由于分库维度和分表维度一致，因此实际情况为，表1,8,9位于DB，表2,3位于DB2，表4,5,6,7位于DB3。  \* 2、跨库查询总数 (看点:跨库记录数汇总)  \* 3、查询index 为4开头的记录 （看点:条件解析与精确路由）  \* 4、查询indexcode含0的记录，并按创建日期排序 (看点: 跨库排序)  \* 5、查询indexcode含0的记录，并按创建日期排序，每页10条，显示第二页(看点: 跨库排序+分页)  \* 6、所有记录,按type字段group by，并计算总数。（看点：跨库+聚合计算）  \* 7、所有记录,按type字段group by，并计算总数。并按总数排序（看点：跨库+聚合计算+排序）  \* 8、查询所有记录的type字段，并Distinct。 (看点：跨库 distinct)  \* 9、跨数据库查询所有记录，并排序分页 (看点：跨库+分页)  \* 10、跨数据库查询所有记录,按type分组后计算总数，通过having条件过滤掉总数在9以上的记录，最后排序 (看点：跨库+聚合+Having+排序)  \* 11、跨数据库查询所有记录,按type分组后计算总数，通过having条件过滤掉总数在9以上的记录，最后排序，取结果的第3~5条。 (看点：跨库+聚合+Having+排序+分页)  \*/  @Test  **public** **void** testDeviceSelect() **throws** SQLException { |

该案例归纳情况如下——

1. 无论有多复杂，在查询结果数量不大的情况下（OLTP系统的特点）。开发人员都无须关注分库分表做了什么。一切都由框架完成了，除非实在无法支持的情况框架会抛出异常。
2. 框架可以识别利用 >、<、 between、in、not in、like等一切有关的条件来缩小查询范围。
3. 框架可以对查询语句中的group、distinct、having、记录范围等因素识别并进行相应的处理。

总的来说，对于分库分表下的查询操作，开发人员应当自行确定一些操作限制。因为SQL语句的写法是无穷的，变化也是无穷的，而框架能支持的操作场景是有限的。这是所有具备分库分表功能的框架都面临的问题。

复杂到框架无法支持的多库查询，一般的SQL开发人员自己也很难编写出来正确的查询来。反过来说，一个熟练的DBA做不到的SQL查询，框架也很难做出来。多库多表下的查询功能都是在不停的完善的，对任何一个框架都是如此。如果有什么需要支持的合理场景，也请联系作者，可以考虑将您的想法改进到这个框架当中。

### 使用SQL语句插入、更新和删除时的路由

和上面的复杂查询一样，我们例举了使用SQL语句进行分库分表操作的场景。

orm-tutorial\src\main\java\org\easyframe\tutorial\lessona\Case2.java

|  |
| --- |
| @Test  **public** **void** testNativeQuery() **throws** SQLException{  List<Device> list = generateDevice(50);  ORMConfig.*getInstance*().setDebugMode(**false**);  *db*.batchInsert(list);  ORMConfig.*getInstance*().setDebugMode(**true**);  System.*err*.println("=====数据准备：插入50条记录完成=====");  /\*\*  \* 使用SQL语句插入记录，根据传入的indexcode每次操作都路由到不同数据库上  \*/  {  NativeQuery nq=*db*.createNativeQuery("insert into DeVice(indexcode,name,type,createDate) values(:indexcode, :name, :type, sysdate)").withRouting();;  nq.setParameter("indexcode", "122346");  nq.setParameter("name", "测试插入数据");  nq.setParameter("type", "办公用品");  nq.executeUpdate();    nq.setParameter("indexcode", "7822346");  nq.setParameter("name", "非官方的得到");  nq.setParameter("type", "大家电");  nq.executeUpdate();    nq.setParameter("indexcode", "452346");  nq.setParameter("name", "萨菲是方式飞");  nq.setParameter("type", "日用品");  nq.executeUpdate();  }  /\*\*  \* 使用SQL语句更新记录，使用Between条件，这意味着indexcode在1000xxx到6000xxx段之间的所有表会参与更新操作。  \*/  {  System.*out*.println("===Between条件中携带的路由条件,正确操作表: 1,2,3,4,5,6");  NativeQuery nq=*db*.createNativeQuery("update DeVice xx set xx.name='ID:'||indexcode,createDate=sysdate where indexcode between :s1 and :s2").withRouting();;  nq.setParameter("s1", "1000");  nq.setParameter("s2", "6000");  nq.executeUpdate();  }    /\*\*  \* 使用SQL语句更新记录，使用In条件，这将精确定位到这些记录所在的表上。  \*/  {  System.*out*.println("===用IN条件更新 ，正确操作表5,6和另外两个==");  NativeQuery nq=*db*.createNativeQuery("update Device set createDate=sysdate, name=:name where indexcode in (:codes)").withRouting();;  nq.setParameter("name", "Updated value");  nq.setParameter("codes", **new** String[]{"6000123","567232",list.get(0).getIndexcode(),list.get(1).getIndexcode(),list.get(2).getIndexcode()});  nq.executeUpdate();  }  /\*\*  \* 使用SQL语句更新记录。由于where条件中没有任何用来缩小记录范围的条件，因此所有的表上都将执行更新操作  \*/  {  System.*out*.println("===正确操作表： 2==");  NativeQuery nq=*db*.createNativeQuery("update Device set createDate=sysdate where type='办公用品' or indexcode='2002345'").withRouting();;  nq.executeUpdate();  }    /\*\*  \* 删除记录。根据indexcode可以准确定位到三张表上。  \*/  {  System.*out*.println("===正确操作表： 1,6,5==");  NativeQuery nq=*db*.createNativeQuery("delete Device where indexcode in (:codes)").withRouting();;  nq.setParameter("codes", **new** String[]{"6000123","567232","110000"});  nq.executeUpdate();  }  /\*\*  \* 删除记录，indexCode上的大于和小于条件以及OR关系勾勒出了这次查询的表,将会是DEVICE\_2,DEVICE\_3,DEVICE\_4、DEVICE\_7、DEVICE\_8。(5张表)  \*/  {  System.*out*.println("===正确操作表： 2,3,4,5,7,8==");  NativeQuery nq=*db*.createNativeQuery("delete Device where indexcode >'200000' and indexcode<'5' or indexcode >'700000' and indexcode <'8'").withRouting();;  nq.executeUpdate();  } |

上述代码也在教程工程中，可以运行一下。归纳特性如下：

1、在NativeQuery中使用路由功能，开发者也基本无需关心路由实现细节，框架基本都自动完成。

2、 框架可以识别利用 >、<、 between、in、not in、like等一切有关的条件来缩小查询范围。

### 使用SQL语句进行查询时的路由

和上文一样，在使用SQL语句进行查询时，如何路由也无需开发者关心。还是那句话，复杂的逻辑都封在框架内，用户像平常那样操作数据库就好。

orm-tutorial\src\main\java\org\easyframe\tutorial\lessona\Case2.java

|  |
| --- |
| /\*\*  \* 查询，将查询所有表，跨库排序  \*/  {  System.*out*.println("查询，所有表，跨库排序");  String sql="select t.\* from device t where createDate is not null order by createDate";  NativeQuery<Device> query=*db*.createNativeQuery(sql,Device.**class**).withRouting();;  **long** total=query.getResultCount();  System.*out*.println("预计查询结果总数Count:"+ total);  List<Device> devices=query.getResultList();  *assertEquals*(total, devices.size());  **int** n=0;  **for**(Device d: devices){  System.*out*.println(d);  **if**(n++>10)**break**;  }  }  /\*\*  \* 查询两个条件时，  \*/  {  System.*out*.println("======查询，Like条件，后一个条件无参数被省略，正确操作表 3==");  String sql="select t.\* from device t where createDate is not null and (t.indexcode like '3%' or t.indexcode in (:codes)) order by createDate";  NativeQuery<Device> query=*db*.createNativeQuery(sql,Device.**class**).withRouting();;  **long** total=query.getResultCount();  System.*out*.println("预期查询总数Count:"+ total);  **int** count=0;  //使用迭代器方式查询  ResultIterator<Device> devices=query.getResultIterator();  **for** (;devices.hasNext();) {  System.*out*.println(devices.next());  count++;  }  devices.close();  *assertEquals*(count, total);    query.setParameter("codes", **new** String[]{"123456","8823478","98765"});  System.*out*.println("======查询，Like条件 OR IN条件，正确操作表:1,3,8,9==");  total=query.getResultCount();  System.*out*.println("预期查询总数Count:"+ total);  List<Device> rst=query.getResultList();  *assertEquals*(total, rst.size());  **int** n=0;  **for**(Device d: rst){  System.*out*.println(d);  **if**(n++>10)**break**;  }  }    /\*\*  \* 查询记录、如果SQL语句中带有Group条件...  \* 跨库条件下的Group实现尤为复杂  \*/  {  System.*out*.println("查询——分组查询，正确操作表：4,1,6");  String sql="select type,count(\*) as count,max(indexcode) max\_id from Device tx where indexcode like '4%' or indexcode like '1123%' or indexcode like '6%' group by type ";  NativeQuery<Map> query=*db*.createNativeQuery(sql,Map.**class**).withRouting();;  System.*out*.println("预期查询总数Count:"+ query.getResultCount());  List<Map> devices=query.getResultList();  **for** (Map ss : devices) {  System.*out*.println(ss);  }  }    /\*\*  \* 查询,SQL语句中带有distinct条件  \*/  {  String sql="select distinct type from device";  NativeQuery<String> query=*db*.createNativeQuery(sql,String.**class**).withRouting();;  **long** total=query.getResultCount();  System.*out*.println("预期查询总数Count:"+ total);  List<String> devices=query.getResultList();  **for** (String ss : devices) {  System.*out*.println(ss);  }  *assertEquals*(6, total);  *assertEquals*(6, devices.size());    } |

需要指出的是，跨库场景下如果使用了group语句，那么count数将不能直接从数据库得出，而是需要在内存中合并计算后才能得出。从性能上讲多少得不偿失，因此这种操作场景并不建议出现。

## 分库规则加载器

EF-ORM中，分表分库规则默认是用Annotation配置在类上的。这可能引起一些不爱用Annotation开发者的不适。EF-ORM中其实默认还支持从json文件中读取分表分库规则。除此之外，您还可以自行编写分库分表规则的加载器，覆盖框架默认的分库分表规则加载器。

如果要使用json方式配置分库分表规则，可以在src/main/resources/目录下添加json文件，文件名为类的全限定名。后缀为json。例如——

org.easyframe.tutorial.lessona.entity.OperateLog.json

|  |
| --- |
| {"key" : [{  "field" : "dateField",  "length" : 2,  "function" : "MONTH"  }, {  "field" : "dbkey",  "length" : 0,  "isDbName" : **true**  }]  } |

即前面提到的@PartitionTable和@PartitionKey的各个字段值。使用json方式定义的分库分表规则将优先于类的Annotation加载。因此当两边都配置了分库分表规则时，json文件中定义的分库分表规则将会覆盖位于类Annotation上的分库分表规则。

如果您需要使用自行编写的分库分表规则的加载器。您需要编写一个类，实现jef.database.meta.PartitionStrategyLoader接口。然后在jef,properties中配置

|  |
| --- |
| partition.strategy.loader=*自定义加载器类名* |

使用上述方法后，启动框架时就会从开发者指定的加载器中加载分库分表规则。

# 与Spring集成

EF-ORM主要通过实现部分JPA接口和Spring集成。EF-ORM将被Spring认为是一个JPA实现一样提供支持。

## 典型配置（快速入门）

对于只想在自己的项目中快速使用EF-ORM，看本节就够了。后面的章节可以跳过。

本节内容适合于熟悉Spring事务管理机制的同学，在面对日常单个数据库连接时，可以直接使用下面的典型配置并自行修改。

|  |
| --- |
| <bean id="dataSource" class="jef.database.datasource.SimpleDataSource"  p:url="jdbc:mysql://127.0.0.1:3306/cms?useUnicode=true" p:user="root"  p:password="12345"/>  <bean id="entityManagerFactory" class="org.easyframe.enterprise.spring.SessionFactoryBean">  <property name="dataSource" ref="dataSource" />  </bean>  <bean id="commonDao" class="org.easyframe.enterprise.spring.CommonDaoImpl" />  <tx:annotation-driven transaction-manager="transactionManager" proxy-target-class="true" />  <aop:aspectj-autoproxy />  <context:component-scan base-package="com.company.my.application"/>    <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">  <property name="entityManagerFactory" ref="entityManagerFactory" />  <property name="jpaDialect">  <bean class="org.easyframe.enterprise.spring.JefJpaDialect" />  </property>  </bean> |

上面这段配置，数据源、SessionFactory、TransactionManager、基于注解的事务声明都有了。然后在编写一个自己的DAO——

|  |
| --- |
| **package com.company.my.application;**  **import org.easyframe.enterprise.spring.GenericDaoSupport;**  **@Repository**  **public** **class** StudentDaoImpl **extends** GenericDaoSupport<Student> {  **public** **void** gradeUp(Collection<Integer> ids) {  Student st=**new** Student();  st.getQuery().addCondition(Student.Field.id, Operator.IN, ids);  st.prepareUpdate(Student.Field.grade, **new** JpqlExpression("grade+1"));  **try** {  //super.getSession()可以得到EF的Session对象  getSession().update(st);  } **catch** (SQLException e) {  **throw** DbUtils.toRuntimeException(e);  }  }  } |

在Dao中，使用super.getSession()方法可以得到当前事务中的Session对象进行数据库操作。

在Service中，使用@Transactional注解进行事务控制。（请参阅Spring官方文档）

## 配置和使用

### SessionFactory的配置

前面的所有例子中，EF的核心对象是一个DbClient对象。DbClient里封装了所有的数据库连接和对应的ORM操作逻辑。

在JPA中，起到类似作用的对象是javax.persistence.EntityManagerFactory类，其地位就和某H框架的SessionFactory一样。

|  |
| --- |
| <!-- 配置数据源，可以带连接池也可以不带 -->  <bean id=*"dataSource"* class=*"jef.database.datasource.SimpleDataSource"*  p:url=*"${db.url}"* p:username=*"${db.user}"* p:password=*"${db.password}"* />  <bean id=*"entityManagerFactory"* class=*"org.easyframe.enterprise.spring.SessionFactoryBean"*>  <property name=*"dataSource"* ref=*"dataSource"* />  </bean>  <bean id=*"commonDao"* class=*"org.easyframe.enterprise.spring.CommonDaoImpl"* /> |

代码 11-1 单数据源的EntityManagerFactory配置

上面的配置中，用Spring的FactoryBean创建EntityManagerFactory对象。EntityManagerFactory是JPA规范中的数据操作句柄。其类似于某H框架和某batis的SessionFactory。

上面的配置，除了定义了EntityManagerFactory对象以外，还定义一个CommonDao对象，该对象实现了org.easyframe.enterprise.spring.CommonDao接口。其作用是仿照传统Dao习惯，提供一个能完成基本数据库操作的对象。具体用法参见11.1.4节。

org.easyframe.enterprise.spring.SessionFactoryBean这个Bean支持以下的配置参数：

|  |  |  |
| --- | --- | --- |
| 参数 | 用途 | 备注或示例 |
| dataSource | 指定一个数据源。 | javax.sql.DataSource的子类即可。 |
| dataSources | map类型，指定多个数据源 | <property name="dataSources">  <map>  <entry key="ds1" value-ref="ds1" />  <entry key="ds2" value-ref="ds2" />  </map>  </property> |
| defaultDatasource | 多数据源时，指定缺省数据源 | <property name="defaultDatasource" value="ds1"> |
| packagesToScan | 配置包名，启动时会扫描这些包下的所有实体类并加载。 | <property name="packageToScan"><list>  <value>org.easyframe.test</value>  <value>org.easyframe.entity</value>  </list></property> |
| annotatedClasses | 配置类名，启动时会扫描这些类并加载 | <property name="annotatedClasses"><list>  <value>org.easyframe.testp.jta.Product</value>  <value>org.easyframe.testp.jta.Users</value>  </list></property> |
| createTable | boolean类型。当扫描到实体后，如果数据库中不存在，是否建表 | 默认true，可以关闭  <property name="createTable" value="false" /> |
| alterTable | boolean类型。当扫描到实体后，如果数据库中存在，是否修改表 | 默认true，可以关闭  <property name="alterTable" value="false" /> |
| allowDropColumn | boolean类型。当扫描到实体后，如果数据库中存在并且需要修改时，是否可以删除列 | 默认false，可以开启  <property name="allowDropColumn" value="true" /> |
| enhancePackages | 配置包名，启动时先对指定包下的实体进行一次增强，多个包用逗号分隔。 | 扫描增强只能对目录下的class文件生效，对ear/war/jar包中class无效。由于大部分J2EE容器都支持包方式部署，此功能只建议在单元测试时使用，不建议发布class未增强的包。  <property name="enhancePackages" value="org.easyframe.tutorial" /> |
| dynamicTables | 配置数据库表名，启动时扫描这些表，生成动态表模型。表名之间逗号分隔 | 参见动态表相关功能说明。  <property name="dynamicTables" value="EF\_TABLE1,XX\_TABLE2,TABLE3" /> |
| registeNonMappingTableAsDynamic | 对比当前数据库中存在的表，如果数据库中的表并未被任何实体所映射，那么生成这张表的动态表模型。 | 该功能可以将所有未被映射的表当做动态表，建立对应的动态元模型，参见动态表相关功能说明。默认false  <property name="registeNonMappingTableAsDynamic" value="true" /> |

### 多数据源的配置

前面提到EF-ORM原生支持分库分表，分库分表意味着EF-ORM要能支持多个数据库实例。

最简单的多数据库下的配置如下

|  |
| --- |
| <!-- 配置多个数据源的DataSource，可以配置带连接池的DataSource，也可以配置不带连接池的DataSource  甚至可以混合使用（没必要的情况下不建议这么做） -->  <bean id=*"dataSource-1"* class=*"jef.database.datasource.SimpleDataSource"*  p:url=*"${db.url}"* p:username=*"${db.user}"* p:password=*"${db.password}"* />  <bean id=*"dataSource-2"* class=*"com.alibaba.druid.pool.DruidDataSource"*  destroy-method=*"close"*  p:driverClassName=*"${db.driver2}"*  p:url=*"${db.url2}"*  p:username=*"${db.user2}"*  p:password=*"${db.password2}"*  p:initialSize=*3*  p:minIdle=*1*  p:maxIdle=*20*  p:maxActive=*50*  />  <bean id=*"dataSource-3"* class=*"org.apache.commons.dbcp.BasicDataSource"*  destroy-method=*"close"*  p:driverClassName=*"${db.driver3}"*  p:url=*"${db.url3}"*  p:username=*"${db.user3}"*  p:password=*"${db.password3}"*  p:initialSize=*3*  p:minIdle=*1*  p:maxIdle=*20*  p:maxActive=*50*  />  <bean id=*"routingDS"* class=*"jef.database.datasource.RoutingDataSource"*>  <property name=*"dataSourceLookup"*>  <bean class=*"jef.database.datasource.SpringBeansDataSourceLookup"* />  </property>  </bean>  <bean id=*"entityManagerFactory"* class=*"org.easyframe.enterprise.spring.SessionFactoryBean"*>  <property name=*"dataSource"* ref=*"routingDS"* />  </bean>  <bean id=*"commonDao"* class=*"org.easyframe.enterprise.spring.CommonDaoImpl"* /> |

代码 11-2 多数据源的EntityManagerFactory配置

多数据库下，需要声明一个RoutingDataSource对象。而RoutingDataSource中，可以配置一个DataSourceLookup对象。DataSourceLookup对象提供多个真正的数据源，通过配置不同的DataSourceLookup，可以实现从不同的地方读取数据源。

上面的配置方法中，定义了*dataSource-1,dataSource-2,dataSource-3*三个原始数据源，存放在Spring的ApplicationContext中。而对应的DataSourceLookup对象是SpringBeansDataSourceLookup，该对象可以从Spring上下文中查找所有的DataSource对象。

框架提供了多个DataSourceLookup，用于在从不同的地方读取数据源配置。这些数据源获取器包括以下几种。

#### URLJsonDataSourceLookup

使用HTTP访问一个URL，从该URL中获得数据源的配置信息。返回的数据源信息用JSON格式表示。参见下面的例子。

如果我们HTTP GET <http://192.168.0.1/getdb> 可以返回如下报文

|  |
| --- |
| [{  "id": "ds1",  "url": "jdbc:mysql://localhost:3306/test",  "user": "root",  "password": "123456",  "driverClassName": "org.gjt.mm.mysql.Driver"  },  {  "id": "ds1",  "url": "jdbc:mysql://localhost:3306/test2",  "user": "root",  "password": "123456",  "driverClassName": "org.gjt.mm.mysql.Driver"  }  ] |
| <bean class=*"jef.database.datasource.URLJsonDataSourceLookup"*  p:datasourceKeyFieldName=*"id*  p:urlFieldName=*"url"*  p:userFieldName=*"user"*  p:passwordFieldName=*"password"*  p:driverFieldName=*"driverClassName"*  p:location=*"http://192.168.0.1/getdb"*>  <property name=*"passwordDecryptor"*>  <!-- 自定义的数据库口令解密器 -->  <bean class=*"org.googlecode.jef.spring.MyPasswordDecryptor"* />  </property>  </bean> |

使用上述配置，即可在需要数据库连接信息时，通过网络调用去获取数据库连接配置。

#### DbDataSourceLookup

到数据库里去取数据源的配置信息。以此进行数据源的查找。

如果利用一个配置库来维护其他各种数据库的连接信息，那么系统会到这个数据库中去寻找数据源。数据库中配置数据源的表和其中的列名也可以配置。参见下面的示例。

|  |
| --- |
| <bean class=*"jef.database.datasource.DbDataSourceLookup"*  p:configDataSource-ref=*"dataSource"*  p:configDbTable=*"DATASOURCE\_CONFIG"*  p:whereCondition=*"enable='1'"*  p:columnOfId=*"DATABASE\_NAME"*  p:columnOfUrl=*"JDBC\_URL"*  p:columnOfUser=*"DB\_USER"*  p:columnOfPassword=*"DB\_PASSWORD"*  p:columnOfDriver=*""*  p:datasourceIdOfconfigDB=*""*  p:defaultDsName=*""* >  <property name=*"passwordDecryptor"*>  <!-- 自定义的数据库口令解密器 -->  <bean class=*"org.googlecode.jef.spring.MyPasswordDecryptor"* />  </property>  </bean> |

使用上述配置，即可在需要数据库连接信息时，通过数据库查找去获取数据库连接配置。

#### JndiDatasourceLookup

到JNDI上下文去找寻数据源配置。参见下面的示例。

|  |
| --- |
| <bean class=*"jef.database.datasource.JndiDatasourceLookup"*>  </bean> |

使用上述配置，即可在需要数据库连接信息时，通过JNDI查找去获取数据库连接配置。

#### MapDataSourceLookup

从一个固定的Map对象中获取已经配置的数据源信息。在我们的一些示例代码中，有些直接就用Map来传入数据源配置。

|  |
| --- |
| // 准备多个数据源  Map<String, DataSource> datasources = **new** HashMap<String, DataSource>();  datasources.put("datasource1", **new** SimpleDataSource("jdbc:derby:./db;create=true", **null**, **null**));  datasources.put("datasource2", **new** SimpleDataSource("jdbc:derby:./db2;create=true", **null**, **null**));  datasources.put("datasource3", **new** SimpleDataSource("jdbc:derby:./db3;create=true", **null**, **null**));  MapDataSourceLookup lookup = **new** MapDataSourceLookup(datasources);  lookup.setDefaultKey("datasource1");// 指定datasource1是默认的操作数据源  // 构造一个带数据路由功能的DbClient  *db* = **new** DbClient(**new** RoutingDataSource(lookup)); |

在Spring配置中也可以用Map来传入多个数据源。

#### PropertiesDataSourceLookup

从一个classpath下的Properties文件中获取数据源配置信息。参见下面的示例。

示例

#### SpringBeansDataSourceLookup

从Spring上下文中获取所有数据源信息。配置如下。

|  |
| --- |
| <bean class=*"jef.database.datasource.SpringBeansDataSourceLookup"* /> |

上面提到的是EF提供的几种默认的DataSourceLookup，开发者也可以编写自己的DataSourceLookup。

示例

### JPA事务配置

大家都知道，Spring有七种事务传播级别。因为标准JPA只能支持其中六种，因此EF-ORM提供了相关的JPA方言以支持第七种。其中nested方式需要JDBC驱动支持SavePoint.

|  |
| --- |
| <bean id=*"transactionManager"* class=*"org.springframework.orm.jpa.JpaTransactionManager"*>  <property name=*"entityManagerFactory"* ref=*"entityManagerFactory"* />  <property name=*"jpaDialect"*>  <bean class=*"org.easyframe.enterprise.spring.JefJpaDialect"* />  </property>  </bean>  <tx:annotation-driven transaction-manager=*"transactionManager"* proxy-target-class=*"true"* />  <aop:aspectj-autoproxy /> |

代码 11-3 Spring基于注解的声明式事务配置方法

Spring的事务配置有好多种方法，上面这种是纯注解的声明式事务，另一种流行的AOP拦截器配置方法如下

|  |
| --- |
| <bean id=*"transactionManager"* class=*"org.springframework.orm.jpa.JpaTransactionManager"*>  <property name=*"entityManagerFactory"* ref=*"entityManagerFactory"* />  <property name=*"jpaDialect"*>  <bean class=*"org.easyframe.enterprise.spring.JefJpaDialect"* />  </property>  </bean>  <tx:advice id=*"tx-advice-default* transaction-manager=*"transactionManager"*>  <tx:attributes>  <tx:method name=*"\*"* propagation=*"REQUIRED"* />  </tx:attributes>  </tx:advice>  <tx:advice id=*"tx-always-new"* transaction-manager=*"transactionManager"*>  <tx:attributes>  <tx:method name=*"\*"* propagation=*"REQUIRES\_NEW"* />  </tx:attributes>  </tx:advice>  <aop:config>  <aop:pointcut id=*"interceptorPointCuts"*  expression=*"execution(\* com.company.product.dao.\*.\*(..))"* />  <aop:advisor advice-ref=*"tx-advice-defaul"*  pointcut-ref=*"interceptorPointCuts"* />  </aop:config> |

代码 11-4 Spring基于AOP拦截器的声明式事务配置方法

Spring事务配置方法还有很多种，但不管哪种配置方法，和ORM框架相关的就只有“*transactionManager”*对象。其他配置都只和Spring自身的事务实现机制有关。

上面的TransactionManager的配置方法和标准的JPA事务管理器配置方法区别之处在于，指定了一个jpaDialect对象，这是因为标准JPA实现因为接口和方法功能较弱，不足以实现Spring事务控制的所有选项。因此Spring提供了一种手段，由ORM提供一个事务控制的方言，Spring根据方言可以精确控制事务。JefJpaDialect的增加，使得EF-ORM能够支持Spring的事务管理的以下特性。（这些特性是标准JPA接口无法支持的)

|  |  |  |
| --- | --- | --- |
| **Spring配置** | **在Spring中的作用** | **效果** |
| **Propagation=“*nested*”** | Spring的7种事务传播级别之一,NESTED方法是启动一个和父事务相依的子事务，因为不是EJB标准的，因此JPA不支持。 | JPA接口中无SavePoint操作，因此无法支持NESTED传播行为，EF-ORM在JpaDIalect中支持了SavePoint，因此可以使用NESTED传播行为。  再加上JPA本身支持的其他6种传播行为，EF-ORM可以支持全部7种传播行为。 |
| **isolation** | 定义事务的四种隔离级别。 | JPA接口不提供对数据库事务隔离级别的动态调整。也就无法支持Spring的事务隔离级别。但EF-ORM可以支持。 |
| **read-only=*"true"*** | 指定事务为只读。该属性提示ORM框架和JDBC驱动进行优化，比如Hibernate下只读事务可以省去flush缓存操作。Oracle服务器原生支持readonly级别，可以不产生回滚段，不记录重做日志，甚至可以提供可重复读等特性。 | 在只读模式下，EF-ORM将对JDBC Connection进行readOnly进行设置，从而触发数据库和驱动的只读优化。当然并不是所有的数据库都支持只读优化。 |
| **timeout** | 事务超时时间，事务一旦超时，会被标记为rollbackOnly，抛出异常并终止处理。 | JPA原生接口不提供事务超时控制。EF-ORM可以通过方言支持。 |

### 编写DAO

通过上面两节，我们在Spring中提供了EntityFactoryManager和事务管理。接下来就是编写自己的Dao对象了。EF-ORM提供了一个泛型DAO实现。

#### 继承GenericDaoSupport

EF-ORM提供了一个泛型的DAO实现。

* 接口类为org.easyframe.enterprise.spring.GenericDao<T>
* 实现类为org.easyframe.enterprise.spring.GenericDaoSupport<T>

开发者的DAO可以直接继承GenericDaoSupport类。

orm-tutorial\src\main\java\org\easyframe\tutorial\lessonb\StudentDao.java

|  |
| --- |
| /\*\*  \* 这个类实现了GenericDao<T>接口  \*/  **public** **class** StudentDao **extends** GenericDaoSupport<Student>{  } |

继承GenericDaoSupport后，该DAO就已经有了各种基本的持久化操作方法。



如果需要自行添加方法，可以这样做

接口 orm-tutorial\src\main\java\org\easyframe\tutorial\lessonb\StudentDao.java

|  |
| --- |
| **public** **interface** StudentDao **extends** GenericDao<Student>{  /\*\*  \* 批量升级学生  \* **@param** ids  \*/  **public** **void** gradeUp(Collection<Integer> ids);  } |

实现类 orm-tutorial\src\main\java\org\easyframe\tutorial\lessonb\StudentDaoImpl.java

|  |
| --- |
| **public** **class** StudentDaoImpl **extends** GenericDaoSupport<Student> **implements** StudentDao{  **public** **void** gradeUp(Collection<Integer> ids) {  Student st=**new** Student();  st.getQuery().addCondition(Student.Field.*id*, Operator.*IN*, ids);  st.prepareUpdate(Student.Field.*grade*, **new** JpqlExpression("grade+1"));  **try** {  getSession().update(st);  } **catch** (SQLException e) {  **throw** DbUtils.*toRuntimeException*(e);  }  }  } |

对于自行实现的方法，可以使用继承自BaseDao类的方法获得Session对象。

一般来说，GenericDao中已经包含了绝大多数日常所需的数据库操作。如果没有特殊操作，我们甚至不需要为某个Bean创建DAO，而是使用后文的CommonDao即可。

#### 继承BaseDao

GenericDao继承了BaseDao。开发者也可以直接继承org.easyframe.enterprise.spring.BaseDao类来编写DAO。在DAO中，开发者可以使用标准的 JPA 方法来实现逻辑，也可以使用EF-ORM的Session对象来实现逻辑。

orm-tutorial\src\main\java\org\easyframe\tutorial\lessonb\MyDao.java

|  |
| --- |
| **public** **class** MyDao **extends** BaseDao{    /\*\*  \* 使用标准JPA的方法来实现DAO  \*/  **public** Student loadStudent(**int** id){  **return** getEntityManager().find(Student.**class**, id);  }    /\*\*  \* 使用EF-ORM的方法来实现DAO  \* **@param** name  \* **@return**  \*/  **public** List<Student> findStudentByName(String name){  Student st=**new** Student();  st.getQuery().addCondition(QB.*matchAny*(Student.Field.*name*, name));  **try** {  **return** getSession().select(st);  } **catch** (SQLException e) {  **throw** DbUtils.*toRuntimeException*(e);  }  }  } |

BaseDao基类中提供了以下方法

|  |  |  |
| --- | --- | --- |
| 方法 | 作用 | 说明 |
| getEntityManager() | 得到javax.persistence.EntityManager对象 | EntityManager是JPA中操作持久化对象的主要方式。 |
| getSession() | 得到jef.database.Session对象 | Session对象是EF-ORM操作数据库的基本类。前面所有例子中的DbClient和Transaction都是其子类。 |
| getDbClient() | 得到jef.database.DbClient对象 | 不建议使用。DbClient对象是无事务状态的Session。对其进行的任何操作都是直接提交到数据库的，不在Spring事务控制之下。 |

要注意的是 getEntityManager()中得到的JPA对象javax.persistence.EntityManager中，EF-ORM并没有实现其全部方法。其中CriteriaBuilder CriteriaQuery相关的功能都会抛出UnSupportedOperationException. 这部分功能请使用EF-ORM自己的Criteria API。

### 常用API：CommonDao

EF-ORM提供了CommonDao是基于Spring的Dao bean的通用接口，提供了以下方法(此处仅列举，详细请参阅API-DOC)

由于EF-ORM中的Entity可以携带Query对象，表示复杂的where条件和update条件，因此很多看似简单的接口，实际上能传入相当复杂的SQL查询对象，请不要低估其作用。

|  |  |  |  |
| --- | --- | --- | --- |
| **方法** | **备注** | | |
| ***基础的查询、插入、更新、删除方法*** | | | |
| T insert(T entity); | | 相当于session.insert | |
| void remove(Object entity); | | 相当于session.delete() 支持单表CriteriaAPI | |
| int update(T entity); | | 相当于session.update() 支持单表CriteriaAPI | |
| List<T> find(T data); | | 相当于session.select() 支持单表CriteriaAPI | |
| T load(T data); | | 相当于session.load() 支持单表CriteriaAPI | |
| <T> ResultIterator<T> iterate(T obj); | | 相当于session.iteratedSelect() 支持单表CriteriaAPI | |
| ***byProperty/ byKey系列*** | | | |
| void removeByProperty(ITableMetadata meta, String propertyName, List<?> values); | | 指定一个字段为条件，批量删除。 | |
| int removeByKey(ITableMetadata meta,String field,Serializable key); | | 指定一个字段为条件，单次删除 | |
| int removeByKey(Class<T> meta,String field,Serializable key); | | 指定一个字段为条件，单次删除 | |
| T loadByKey(Class<T> meta,String field,Serializable key); | | 指定一个字段为条件，加载单条 | |
| T loadByKey(ITableMetadata meta,String field,Serializable id); | | 指定一个字段为条件，加载单条 | |
| List<?> findByKey(ITableMetadata meta, String propertyName, Object value); | | 指定一个字段为条件，加载多条 | |
| ***ByExample系列*** | | | |
| List<T> findByExample(T entity,String... properties); | | 传入模板bean，可指定字段名，这些字段值作为where条件 | |
| int removeByExample(T entity,String... properties); | | 传入模板bean，可指定字段名，这些字段值作为where条件 | |
| ***By PrimaryKey*** | | | |
| T loadByPrimaryKey(ITableMetadata meta, Object id); | | 按主键加载 | |
| T loadByPrimaryKey(Class<T> entityClass, Object primaryKey); | | 按主键加载 | |
| ***保存方法*** | | | |
| void persist(Object entity); | | 对象存在时更新，不存在时插入 | |
| T merge(T entity); | | 对象存在时更新，不存在时插入 | |
| ***Update方法*** | | | |
| int updateByProperty(T entity,String... property); | | 可传入多个字段名，这些字段的值作为where条件 | |
| int update(T entity,Map<String,Object> setValues,String... property); | | 可传入多个字段名，这些字段的值作为where条件。可在map中指定要更新的值。 | |
| ***Remove方法*** | | | |
| void removeAll(ITableMetadata meta); | | 删除全表记录 | |
| ***批量操作系列*** | | | |
| int batchUpdate(List<T> entities); | | | 批量（按主键）更新 |
| int batchUpdate(List<T> entities,Boolean doGroup); | | | 批量（按主键）更新 |
| int batchRemove(List<T> entities); | | | 批量删除 |
| int batchRemove(List<T> entities,Boolean doGroup); | | | 批量删除 |
| int batchInsert(List<T> entities); | | | 批量插入 |
| int batchInsert(List<T> entities,Boolean doGroup); | | | 批量插入 |
| ***命名查询NamedQuery*** | | |  |
| List<T> findByNq(String nqName, Class<T> type,Map<String, Object> params); | | | 传入查询名称、返回类型、参数 |
| List<T> findByNq(String nqName, ITableMetadata meta,Map<String, Object> params); | | | 传入查询名称、返回类型、参数 |
| int executeNq(String nqName,Map<String,Object> params); | | | 执行命名查询操作，传入查询名称，参数。 |
| ***E-SQL操作系列*** | | | |
| List<T> findByQuery(String sql,Class<T> retutnType, Map<String, Object> params); | | | 传入E-SQL语句查询结果 |
| List<T> findByQuery(String sql,ITableMetadata retutnType, Map<String, Object> params); | | | 传入E-SQL语句，查询结果 |
| int executeQuery(String sql,Map<String,Object> param); | | | 传入E-SQL语句，执行 |
| <T> ResultIterator<T> iterateByQuery(String sql,Class<T> returnType,Map<String,Object> params); | | | 传入E-SQL语句。查询并以遍历器返回。 |
| <T> ResultIterator<T> iterateByQuery(String sql, ITableMetadata returnType, Map<String, Object> params); | | | 传入E-SQL语句。查询并以遍历器返回。 |
| ***分页查询方法*** | | |  |
| Page<T> findAndPage(T data,int start,int limit); | | | 传入单表Criteria对象。分页查询 |
| Page<T> findAndPageByNq(String nqName, Class<T> type,Map<String, Object> params, int start,int limit); | | | 传入命名查询名称，分页查询 |
| Page<T> findAndPageByNq(String nqName, ITableMetadata meta,Map<String, Object> params, int start,int limit); | | | 传入命名查询名称，分页查询 |
| Page<T> findAndPageByQuery(String sql,Class<T> retutnType, Map<String, Object> params,int start,int limit); | | | 传入E-SQL语句，分页查询 |
| Page<T> findAndPageByQuery(String sql,ITableMetadata retutnType, Map<String, Object> params,int start,int limit); | | | 传入E-SQL语句，分页查询 |
| ***其他*** | | | |
| Session getSession(); | 得到的EF-ROM Session对象 | | |
| DbClient getNoTransactionSession(); | 得到当前无事务的操作Session | | |

从上面的API可以看出，配置命名查询配置，仅凭CommonDao已经可以完成大部分的数据库DAO操作。

|  |
| --- |
| ***DAO轻量化实践***  *从个人开发实践看，随着ORM框架封装性的提升，DAO层越来越趋向轻量化。这里是个人的一点看法和建议。*  *轻量化表现在*   1. *在无需多种实现的情况下，DAO无须设计接口类* 2. *无需为每个Entity创建DAO，大多数数据库操作在Service中直接获取CommonDao进行操作即可。*   *为什么说，为每个Entity创建一个DAO这种做法过时了呢？*  *实践表明，局限单表的操作都可以继承GenericDao自动获得，子类中几乎无需任何编码，甚至泛型都是不需要的，ORM能够根据传入的对象类型绑定到对应的数据表上。所以泛型的DAO仅仅是起到了增加了一些Bean类型校验的作用。为此付出的代价是，开发时还要控制不同DAO的依赖注入、还要控制bean使用对号入座的DAO进行操作，这些都是多余的工作。*  *此外，如果我们为每个Entity创建DAO，那么多表关联的操作应该放在哪个DAO里呢？ 事实上无论放在哪一个DAO中都不是那么合理的。*  *所以，结合业务实践，个人建议在使用EF-ORM的时候，可以省略掉大部分Entity的对应的DAO。一些复杂数据库操作（基本上是涉及多表的），可以自行继承CommonDaoImpl，放在公共的DAO中。一般中小规模的应用，最后的DAO个数不会超过10个。* |

### POJO操作支持

CommonDao中的方法还有一个特点，那就是可以支持POJO Bean。我们在最初的1.1.3示例中可以发现，无需继承jef.database.DataObject，我们可以直接使用单纯的POJO对象创建表、删除表、执行CRUD操作。

POJO支持是为了进一步简化ORM使用而在CommonDao中进行的特殊处理。因此CommonDao中所有的泛型T，都无需继承jef.database.DataObject。

当我们定义POJO时，依然可以使用 @Id @Column @Table等基本的JPA注解。不过由于POJO Bean中不包含Query对象，因此在使用上基本只能按主键实现CRUD操作。

CommonDao中设计了xxxxByProperty、xxxxByKey等系列的方法，也正是考虑到POJO对象中，无法准备的记录用户设置过值的字段，因此提供一个手工指定的补救办法。使用这两个系列的方法，可以更方便的操作POJO对象。例如

我们定义一个POJO Entity

orm-tutorial\src\main\java\org\easyframe\tutorial\lessonb\entity\Foo.java

|  |
| --- |
| @Table(name="MY\_FOO")  **public** **class** Foo {  @Id  @GeneratedValue  **private** **int** id;  **private** String name;  **private** **int** age;  //Getter Setter省略  } |

便可以对其进行各种操作了

|  |
| --- |
| @Test  **public** **void** test1() **throws** SQLException{  commonDao.getNoTransactionSession().dropTable(Foo.**class**);  commonDao.getNoTransactionSession().createTable(Foo.**class**);  {  Foo foo=**new** Foo();  foo.setName("Hello!");  commonDao.insert(foo);  }  {  Foo foo=**new** Foo();  foo.setAge(3);  foo.setName("飞");  //update MY\_FOO set age=3 where name='Hello!'  commonDao.updateByProperty(foo, "name");  }  {  Foo foo=commonDao.loadByPrimaryKey(Foo.**class**, 1);  System.*out*.println(foo.getName());  }  {  //根据ID删除  commonDao.removeByKey(Foo.**class**, "id", 1);  }  } |

上面演示了对Foo对象进行建表、删表、增删改查操作。

最后，EF-ORM可以在一定程度上识别某H框架的配置文件，当做POJO Bean的注解来使用。这种做法可以在EF-ORM中直接使用某H框架的Bean定义。

比如我们创建不带任何注解的POJO类

orm-tutorial\src\main\java\org\easyframe\tutorial\lessonb\entity\PojoEntity.java

|  |
| --- |
| **public** **class** PojoEntity {  **private** String name;  **private** Integer id;  **private** String comments;  //Getter Setter  } |

然后我们配置一个XML文件去定义这个类的元数据

orm-tutorial\src\main\java\org\easyframe\tutorial\lessonb\entity\hbm\PojoEntity.hbm.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">  <hibernate-mapping>  <class name=*"org.easyframe.tutorial.lessonb.entity.PojoEntity"* table=*"Jef\_pojo\_table"*>  <id name=*"id"*>  <column name=*"id"* length=*"5"* />  <generator strategy=*"IDENTITY"* />  </id>  <property name=*"name"*>  <column name=*"name" /*>  </property>  <property name=*"comments"*>  <column name=*"comments" /*>  </property>  </class>  </hibernate-mapping> |

只要通过指定class和某xml文件存在关联，EF-ORM就能够识别某H框架中的主要标签来读取元数据配置。

orm-tutorial\src\main\java\org\easyframe\tutorial\lessonb\Case1.java

|  |
| --- |
| @Test  **public** **void** test2() **throws** SQLException{  //读取指定路径下的某H框架配置文件。 %s表示类的SimpleName。%c表示类的全名。  ORMConfig.*getInstance*().setMetadataResourcePattern("hbm/%s.hbm.xml");  commonDao.getNoTransactionSession().dropTable(PojoEntity.**class**);  commonDao.getNoTransactionSession().createTable(PojoEntity.**class**);    PojoEntity p=**new** PojoEntity();  p.setName("fsdfsfs");    commonDao.insert(p);  System.*out*.println(p.getId());  commonDao.insert(p);  System.*out*.println(p.getId());  commonDao.insert(p);  System.*out*.println(p.getId());      PojoEntity pojo=commonDao.load(p);  System.*out*.println(pojo);    pojo.setName("35677");  commonDao.update(pojo);    System.*out*.println("===========================");    PojoEntity cond=**new** PojoEntity();  cond.setId(12);  System.*out*.println(commonDao.find(cond));  commonDao.remove(cond);  } |

有一项名为MetadataResourcePattern的全局参数配置，用于指定Entity类和某个XML文件之间的关联关系。

例如，所有的XML文件位于class path下的 /hbm目录中，其名称和类的SimpleName一致。此时可以在jef.properties中配置——

|  |
| --- |
| metadata.resource.pattern=/hbm/%s.hbm.xml |

其中 %s表示类的SimpleName；%c表示类的全名；

还可以用 %\*表示匹配任意字符，一旦匹配为\*，那么EF-ORM会查找所有满足条件的XML，然后根据XML中配置的class属性反向匹配到Entity上。

目前此功能仅支持某H框架中一些基本的单表字段描述，级联等描述目前还不支持。

## 多数据源下的事务控制

在数据分库之后。下一个问题就接踵而至，这就是分布式事务的一致性问题。

如果我们依旧使用Spring的JPA事务控制器，正常情况下，如果所有数据库都成功提交，那么事务可以保持一致，如下图所示——



### JPA事务（多重）

如果考虑到提交可能失败的场景，我们如果继续使用JPA事务管理器，我们将需要承担一定的风险。

***中断提交***

当遭遇提交失败时，有两种行为策略。默认的是“中断提交”

因此，在默认情况下，当一个Spring事务结束时，EF会顺序提交A、B两个数据库的修改。如果发生提交失败，则中断提交任务。



从上例看，也就是说，如果先提交 A库失败，那么A、B库都不提交。如果先提交A库成功，B库提交失败，那么A库的修改将会生效，而B库的修改不生效。

***继续提交***

此外，这一策略还可以变更。在jef.properties中配置：

|  |
| --- |
| db.jpa.continue.commit.if.error =true |

开启上述配置后，那么在一个库提交失败后，整个提交过程将持续进行下去，直到所有能提交的变更都写入数据库位置。这种策略下，哪个连接先提交哪个后提交将不再产生影响。如下图所示



这种方式下，简单来说，如果我们的事务中用到了A、B两个数据库，事务提交时A、B数据库的修改单独提交，互不影响。

无论使用上述哪种策略，都有可能会出现某些数据库提交成功、某些数据库提交失败的可能。因此，在没有跨库事务一致性要求的场合，我们依然可以用JPATransactionManager来管理事务，虽然这可能会造成上述两种场景的数据不一致，但如果您的系统业务上本身就没有这种严格的一致性要求时，JPA事务不失为是最简单的使用方法。

在多库上使用JPA事务管理器时，每个数据库上的操作分别位于独立的事务中，相当于将Spring的事务划分为了多个独立的小型JPA事务。我们姑且用 “多重JPA事务”来称呼。

如果出现了某些数据库被提交，某些数据库出错或未提交。此时框架将会抛出*jef.database.innerpool.InconsistentCommitException类。*该异常类标识着多个数据库的提交状态出现了不一致。该异常类中，可以获得哪些数据源提交成功，哪位未提交成功的信息。供开发者自行处理。

### JTA事务支持

上面的问题是不是无法避免的呢？不是， SpringFramework还支持JTA事务。使用J2EE的JTA规范，我们可以让EF-ORM在多数据库下支持分布式事务。

JTA是JavaEE技术规范之一，JTA允许应用程序执行分布式事务处理——在两个或多个网络计算机资源上访问并且更新数据。EF-ORM可以借助一些数据库JDBC驱动本身的XA功能，或者第三方的开源JTA框架实现分布式事务。

使用JTA事务后，刚才的流程即可变为下图所示，因此任何一个数据库提交错误情况下，都能确保数据库数据一致性。



目前ef-orm推荐使用atomikos作为JTA的事务支持框架。  
关于JTA的介绍，可参见http://www.ibm.com/developerworks/cn/java/j-lo-jta/

关于atomikos的介绍，可参见<http://www.atomikos.com/>

下面我们举例，用Spring + atomikos + EF-ORM实现分布式事务。  
首先，我们在pom.xml中，引入atomikos的包以及jta的API包。

|  |
| --- |
| <dependency>  <groupId>com.atomikos</groupId>  <artifactId>transactions-jdbc</artifactId>  <version>3.9.3</version>  </dependency>  <dependency>  <groupId>com.atomikos</groupId>  <artifactId>transactions-jta</artifactId>  <version>3.9.3</version>  </dependency>  <dependency>  <groupId>javax.transaction</groupId>  <artifactId>jta</artifactId>  <version>1.1</version>  </dependency> |

由于使用了atomikos，在Spring bean配置中，需要配置XA的数据源

|  |
| --- |
| <bean id=*"ds1"* class=*"com.atomikos.jdbc.AtomikosDataSourceBean"*  init-method=*"init"* destroy-method=*"close"*>  <property name=*"uniqueResourceName"*><value>mysql/ds1</value> </property>  <property name=*"xaDataSourceClassName"*>  <value>com.mysql.jdbc.jdbc2.optional.MysqlXADataSource</value>  </property>  <property name=*"xaProperties"*>  <props>  <prop key=*"URL"*>jdbc:mysql://localhost:3307/test</prop>  <prop key=*"user"*>root</prop>  <prop key=*"password"*>admin</prop>  </props>  </property>  <property name=*"poolSize"*><value>3</value></property>  <property name=*"maxPoolSize"*><value>30</value> </property>  </bean>  <bean id=*"ds2"* class=*"com.atomikos.jdbc.AtomikosDataSourceBean"*  init-method=*"init"* destroy-method=*"close"*>  <property name=*"uniqueResourceName"*><value>oracle/ds2</value></property>  <property name=*"xaDataSourceClassName"*>  <value>oracle.jdbc.xa.client.OracleXADataSource</value>  </property>  <property name=*"xaProperties"*>  <props>  <prop key=*"URL"*>jdbc:oracle:thin:@pc-jiyi:1521:orcl</prop>  <prop key=*"user"*>pomelo</prop>  <prop key=*"password"*>pomelo</prop>  </props>  </property>  <property name=*"poolSize"*><value>3</value></property>  <property name=*"maxPoolSize"*><value>30</value> </property>  </bean> |

上例配置了两个JTA的数据源，一个是Oracle数据库,的一个是MySQL数据库。然后配置EF-ORM的SessionFactory

|  |
| --- |
| <bean id=*"sessionFactory"* class=*"org.easyframe.enterprise.spring.SessionFactoryBean"* destroy-method=*"close"*>  <property name=*"transactionMode"* value=*"jta"*></property>  <property name=*"dataSources"*>  <map>  <entry key=*"ds1"* value-ref=*"ds1"* />  <entry key=*"ds2"* value-ref=*"ds2"* />  </map>  </property>  <property name=*"packagesToScan"*>  <list>  <value>com.github.xuse.easyframe.test </value>  </list>  </property>  </bean> |

配置SessionFactoryBean，和前面的区别在于要将tranactionMode配置为”jta”。

然后配置Spring的声明式事务管理。

|  |
| --- |
| <!--事务管理器，需要使用JtaTransactionManager -->  <bean id=*"transactionManager"*  class=*"org.springframework.transaction.jta.JtaTransactionManager"*>  <property name=*"userTransaction"*>  <bean class=*"com.atomikos.icatch.jta.UserTransactionImp"*  p:transactionTimeout=*"300"* />  </property>  <property name=*"transactionManager"*>  <bean class=*"com.atomikos.icatch.jta.UserTransactionManager"*  init-method=*"init"* destroy-method=*"close"* p:forceShutdown=*"true"* />  </property>  </bean>  <!-- 事务AOP切面，和标准的Spring配置方法没有区别 -->  <tx:advice id=*"advice"* transaction-manager=*"transactionManager"*>  <tx:attributes>  <tx:method name=*"save\*"* propagation=*"REQUIRED"* />  <tx:method name=*"delete\*"* propagation=*"REQUIRED"* />  <tx:method name=*"\*"* read-only=*"false"* />  </tx:attributes>  </tx:advice>  <aop:config>  <aop:pointcut id=*"point"*  expression=*" execution(\* com.github.easyframe.testp.jta.dao.Biz\*.\*(..))"* />  <aop:advisor advice-ref=*"advice"* pointcut-ref=*"point"* />  </aop:config> |

上述是Spring的事务策略和AOP配置。其中atomikos的连接池，事务超时等控制参数也可以配置，详情可参阅atomikos的官方文档。

使用上述配置后，EF-ORM和Spring基本放弃了对事务控制，单个线程中的所有操作都在一个事务(UserTransaction)中。直到事务结束，连接关闭（被放回JTA连接池）时，所有数据才被提交。凡是位于上述切面中的save\*或者delete\*方法中，如果操作了多个数据库的数据，框架都会保证其数据一致性。

|  |
| --- |
| ***在JTA模式下,EF-ORM作了哪些机制来适应JTA***  *在启用JTA后，EF-ORM会禁用一些内部特性来满足JTA的要求，比如禁用内部连接池，禁用Postgres事务保持功能等。也不会在Connection上执行commit 、rollback、setReadOnly等操作。*  *此外，启用了JTA后，DDL语句将不能和业务操作在同一个线程中运行，因此凡是涉及到建表、删表、创建Sequence等DDL操作时，EF-ORM都会创建一个新的线程，在独立连接上操作数据库。这些都是JTA事务模式下的特殊处理。* |

## 共享其他框架的事务

如果您将EF-ORM和其他ORM框架混合使用，那么就会碰到共享事务的问题。我们一般会希望在一个服务(Service)方法中，无论使用哪个框架来操作数据库，这些操作都位于一个事务中。

为了适应这种场景，EF-ORM中存在一个共享事务的模式，一旦启用后，EF-ORM将会放弃自己的事务控制和连接管理，而是到Spring的上下文中去查找其他框架所使用的连接对象，然后在该连接上进行数据库操作，从而保证多个框架操作同一个事务。

目前EF-ORM可以和以下三种框架共享事务。

Hibernate

MyBatis

Spring JdbcTemplate。

下面具体说明具体的配置方法。

### Hibernate

下面例子中配置了事务共享

|  |
| --- |
| <!— 配置数据源 -->  <bean id=*"dataSource"* class=*"jef.database.test.jdbc.DebugDataSource"*>  </bean>  <!— 配置Hibenrate 3 Session Factory-->  <bean id=*"sessionFactory"*  class=*"org.springframework.orm.hibernate3.LocalSessionFactoryBean"*>  <property name=*"configLocation"* value=*"classpath:hibernate-perftest.cfg.xml"* />  <property name=*"dataSource"* ref=*"dataSource"* />  <property name=*"lobHandler"* ref=*"lobHandler"* />  </bean>  <bean id=*"lobHandler"* class=*"org.springframework.jdbc.support.lob.DefaultLobHandler"*  lazy-init=*"true"* />  <!-- 事务管理器配置 -->  <bean id=*"hibernateTxManager"*  class=*"org.springframework.orm.hibernate3.HibernateTransactionManager"*>  <property name=*"sessionFactory"* ref=*"sessionFactory"* />  <property name=*"dataSource"* ref=*"dataSource"* />  </bean>  <!-- JDBC TEMPLATE -->  <bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>  <property name=*"dataSource"* ref=*"dataSource"* />  </bean>  <!-- EF配置 -->  <bean id=*"entityManagerFactory"* class=*"org.easyframe.enterprise.spring.SessionFactoryBean"* destroy-method=*"close"*>  <property name=*"dataSource"* ref=*"dataSource"* />  <property name=*"transactionMode"* value=*"JDBC"* />  </bean>  <!— 此处仅介绍事务管理器配置，其他的事务策略、事务拦截器、事务切面等略，请自行百度 --> |

上述配置的要点是

1. 使用Hibernate的事务管理器。注意需要注入DataSource对象。
2. 必须用同一个DataSource对象初始化JdbcTemplate， EF-ORM SessionFactory。
3. EF-ORM的*transactionMode*参数必须设置为jdbc 。

### MyBatis / JdbcTemplate

|  |
| --- |
| <bean id=*"dataSource"* class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>  <!— 此处定义数据源 -->  </bean>  <!— Spring JdbcTempate配置 -->  <bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>  <property name=*"dataSource"* ref=*"dataSource"* />  </bean>  <!— MyBatis配置，其他略 -->  <bean id=*"sqlSessionFactory"* class=*"org.mybatis.spring.SqlSessionFactoryBean"*>  <property name=*"dataSource"* ref=*"dataSource"* />  </bean>  <!-- 事务管理器配置 -->  <bean id=*"transactionManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>  <property name=*"dataSource"* ref=*"dataSource"* />  </bean>    <!-- EF配置 -->  <bean id=*"entityManagerFactory"* class=*"org.easyframe.enterprise.spring.SessionFactoryBean"* destroy-method=*"close"*>  <property name=*"dataSource"* ref=*"dataSource"* />  <property name=*"transactionMode"* value=*"jdbc"* />  </bean>  <!— 此处仅介绍事务管理器配置，其他的事务策略、事务拦截器、事务切面等略，请自行百度 --> |

上述配置的要点是

1. 使用*org.springframework.jdbc.datasource.DataSourceTransactionManager*事务管理器。
2. 必须用同一个DataSource对象初始化JdbcTemplate，MyBatis SessionFactory，EF-ORM SessionFactory。
3. EF-ORM的*transactionMode*参数必须设置为jdbc。

按上述要点配置后，即可确保三个框架的操作处于同一个事务中。

|  |
| --- |
| ***扩展阅读：共享事务的原理***  *Spring在设计时，考虑到了JdbcTemplate和Hibenrate共享事务的问题，会将Hibernate事务所使用的连接暴露出来，用ThreadLocal保存在一个静态变量中，这就为共享事务提供了可能。*  *对于Spring来说，在Hibernate中暴露出来的连接，和在使用MyBatis/JdbcTemplate时存放的事务连接是相同的，因此Hibernate和JdbcTemplate/MyBatis之间就可以共享事务。*  *在启用JDBC的事务模式后，EF-ORM会禁用内部连接管理和事务管理。每次操作时，都去寻找Spring事务管理器所暴露出来的当前事务连接进行利用。相当与把自身的事务管理方式改得和 JdbcTemplate一样，因此也就能和上述两个框架一样，共享事务连接。* |

# 动态表支持

## 动态表支持

### 动态表的返回（特殊）

本节介绍不同数据返回的规则。下节介绍操作这些映射规则的API

API介绍

动态表返回对象VartObject

1.8.0以上版本支持

## 扫描现存的数据库表作为动态表操作

1.8.2以上版本支持。

其他功能

内置连接池

# DB元数据与DDL操作

DB元数据，即数据库中的schema、table、view、function、procudure等数据库内建的各种数据结构和代码。

在EF-ORM中，提供了名为DbMetadata的API类来操作数据库的结构信息，初步了解可参见下文的例子。详情请参阅API文档。

使用API来操作数据库结构的主要优点是，一般无需考虑不同的RDBMS的兼容性，因此可以使语句在各种数据库上通用。（但依然要受数据库本身支持的数据类型等限制）

JDBC中，提供了名为DatabaseMetadata的元数据访问接口类。通过这个接口类可以获得大部分的数据库元数据信息。元数据访问接口类中的很多方法，都是在JDBC3或JDBC4中添加的，要想访问这些方法，请尽量使用满足JDBC4接口的驱动。

## 访问数据库结构

### 获取RDBMS版本号等基本信息

下面的例子演示了如何获取数据库的名称、版本号、JDBC驱动名和版本号；还有数据库支持的函数、数据类型等信息。

src\main\java\org\easyframe\tutorial\lessonc\CaseDatabaseMetadata.java

|  |
| --- |
| @Test  **public** **void** testDbInfo() **throws** SQLException {  // 当有多个数据源时，需要指定数据源的名称，如果只有一个数据源，那么传入null就行了。  DbMetaData meta = db.getMetaData(**null**);  Map<String, String> version = meta.getDbVersion();  **for** (String key : version.keySet()) {  System.out.println(key + ":" + version.get(key));  }  System.out.println("schema:" + meta.getCurrentSchema());  System.out.println("=== Functions ===");  System.out.println(meta.getAllBuildInFunctions());  System.out.println("=== DATA TYPES ===");  System.out.println(meta.getSupportDataType());  } |

前面已经讲过EF-ORM中，一个session对象中可以封装多个数据源，因此在获取DbMetaData时，首先要指定数据源。如果传入数据源名称为null，那么就会获得第一个（默认的）数据源。

### 获得表和字段结构信息

下面的例子演示了如何获得数据库中有哪些表和视图，然后可以打印出一张表的各个列和每个列的类型信息。还有相关的索引、主键等信息。

src\main\java\org\easyframe\tutorial\lessonc\CaseDatabaseMetadata.java

|  |
| --- |
| @Test  **public** **void** testTableInfo() **throws** SQLException {  DbMetaData meta = db.getMetaData(**null**);  //查看数据库中的所有表  List<TableInfo> tables=meta.getTables();  System.out.println("=== There are "+ tables.size()+" tables in database. ===");  **for**(TableInfo info:tables){  System.out.println(info);  }    //查看数据库中的所有视图  List<TableInfo> views=meta.getViews();  System.out.println("=== There are "+ views.size()+" views in database. ===");  **for**(TableInfo info:views){  System.out.println(info);  }    //查看一张表的信息  **if**(tables.isEmpty())return;  String tableName=tables.get(0).getName();  List<Column> cs = meta.getColumns(tableName,**true**);  System.out.println("==== Table " + tableName + " has " + cs.size() + " columns. ====");  **for** (Column c : cs) {  System.out.println(c.getColumnName()+"\t"+c.getDataType() + "\t" + c.getColumnSize() + "\t" + c.getRemarks());  }  //表的主键  System.out.println("======= Table " + tableName + " Primary key ========");  System.out.println(meta.getPrimaryKey(tableName));  //一张表的索引  Collection<Index> is = meta.getIndexes(tableName);  System.out.println("==== Table " + tableName + " has " + is.size() + " indexes. ===");  **for** (Index i : is) {  System.out.println(i);  }  } |

运行上述案例，可以查看到数据库中第一张表的列、数据类型、索引、主键等详细信息。

### 删表和建表

前面的代码中，您可以已经看到了关于删除表和创建表的例子。在DbMetaData上也有相应的删表和建表方法。

删除表时会主动删除相关的所有主外键、索引、约束。

建表时会同时创建表上定义的索引(使用@Index 或者 @Indexed定义)，但不会创建外键。

src\main\java\org\easyframe\tutorial\lessonc\CaseDatabaseMetadata.java

|  |
| --- |
| @Test  **public** **void** testTableDropCreate() **throws** SQLException{  DbMetaData meta = db.getMetaData(**null**);  //删除Student表  meta.dropTable(Student.**class**);  //创建Student表  meta.createTable(Student.**class**);  } |

注意——

在删表时，如果表存在并成功删除则返回true；如果表不存在不会抛出异常，而是返回false。

在建表时，如果表不存在并成功创建返回true；如果表已存在不会抛出异常，而是返回false。

### 操作索引

下面的例子演示了查询索引、创建索引、删除索引。

src\main\java\org\easyframe\tutorial\lessonc\CaseDatabaseMetadata.java

|  |
| --- |
| @Test  **public** **void** testIndexes() **throws** SQLException{  DbMetaData meta = db.getMetaData(**null**);  System.out.println("=== Indexes on table Student ===");  **for**(Index index: meta.getIndexes(Student.**class**)){  System.out.println(index);  }    System.out.println("=== Now we try to create two indexes ===");  //创建单字段的unique索引  Index index1=meta.toIndexDescrption(Student.**class**, "name");  index1.setUnique(**true**);  meta.createIndex(index1);  //创建复合索引，其中Grade列倒序  Index index2=meta.toIndexDescrption(Student.**class**, "grade desc", "gender");  meta.createIndex(index2);  //打印出表上的所有索引  System.out.println("=== Indexes on table Student (After create)===");  **for**(Index index: meta.getIndexes(Student.**class**)){  System.out.println(index);  }    System.out.println("=== Now we try to drop all indexes ===");  //删除所有索引  **for**(Index index: meta.getIndexes(Student.**class**)){  **if**(index.getIndexName().startsWith("IDX")){  meta.dropIndex(index);  }  }  //打印出表上的所有索引  System.out.println("=== Indexes on table Student (After drop)===");  **for**(Index index: meta.getIndexes(Student.**class**)){  System.out.println(index);  }  } |

### 操作外键

我们可以访问数据库中表和表之间的外键，并且删除它们或者创建它们。下面的例子演示了创建外键、查询外键，删除外键等操作。

src\main\java\org\easyframe\tutorial\lessonc\CaseDatabaseMetadata.java

|  |
| --- |
| @Test  **public** **void** testForeignKeys() **throws** SQLException{  String tableName="STUDENT\_TO\_LESSON";  DbMetaData meta = db.getMetaData(**null**);    //在表上面创建两个外键，分别引用Student表和LessonInfo表。  meta.createForeignKey(StudentToLesson.Field.studentId, Student.Field.id);  meta.createForeignKey(StudentToLesson.Field.lessionId, LessonInfo.Field.id);    //打印目前表上的外键(应该看到两个外键)  System.out.println("=== Foreign keys on table ["+tableName+"] ===");  LogUtil.show(meta.getForeignKey(tableName));    //打印引用到表Student的外键（应该看到一个外键）  System.out.println("=== Foreign keys on table [STUDENT] ===");  LogUtil.show(meta.getForeignKeyReferenceTo("STUDENT"));    //打印引用到表LessonInfo的外键（应该看到一个外键）  System.out.println("=== Foreign keys on table [LESSON\_INFO] ===");  LogUtil.show(meta.getForeignKeyReferenceTo("LESSON\_INFO"));    //删除表上的所有外键  System.out.println("=== Drop Foreign keys ===");  meta.dropAllForeignKey(tableName);    //打印目前表上的外键(应该没有外键)  System.out.println("=== Foreign keys on table "+tableName+" ===");  LogUtil.show(meta.getForeignKey(tableName));  } |

### 访问自定义函数和存储过程

我们可以访问数据库中的自定义函数和存储过程。不过DbMetaData没有提供方法去修改这些对象。

下面的例子演示了如何查询数据库中的自定义函数和存储过程。（只能看到基本信息，无法看到具体的SQL代码）。

src\main\java\org\easyframe\tutorial\lessonc\CaseDatabaseMetadata.java

|  |
| --- |
| @Test  **public** **void** testFunctions() **throws** SQLException{  DbMetaData meta = db.getMetaData(**null**);  System.out.println("=== User Defined Functions ===");  //打印出所有函数  List<Function> functions=meta.getFunctions(**null**);  **for**(Function f: functions){  System.out.println(f);  }  //打印出所有存储过程  System.out.println("=== User Defined Procedures ===");  functions=meta.getProcedures(**null**);  **for**(Function f: functions){  System.out.println(f);  }  } |

要注意的是，查询存储过程和自定义函数是较高版本的JDBC驱动才能提供的方法。在一些不够主流的数据库上，getFunctions操作可能不支持。如果您使用的是主流数据库，请注意使用较高版本的JDBC驱动。

### 修改已存在的表(Alter table)

修改表——ALTER TABLE语句，可能是不同的RDBMS之间差别最大的SQL语句了。对于修改表，框架提供的接口也会稍微复杂一些。

修改表的功能被统一为refreshTable()方法，这个方法的作用的将数据库中的表结构和传入的表模型进行对比，并让数据库中的表和模型保持一致。

src\main\java\org\easyframe\tutorial\lessonc\CaseAlterTable.java

|  |
| --- |
| /\*\*  \* refreshTable的效果是检查并修改数据库中的表，使其和传入的实体模型保持一致。  \* 本例中，数据库中没有此表，因此变为建表操作。  \*/  @Test  **public** **void** testCreateTableSimple() **throws** SQLException{  db.dropTable(Student.**class**);  db.refreshTable(Student.**class**);  } |

那么，如果Student表已经存在的情况下，就需要使用ALTER TABLE的方式来维护数据库结构了。再看下面这个例子。

src\main\java\org\easyframe\tutorial\lessonc\CaseAlterTable.java

|  |
| --- |
| /\*\*  \* 先用SQL语句直接建立一张类似的表。  \* 然后通过refresh方法，修改已存在的表。  \*/  @Test  **public** **void** testAlterTableSimple() **throws** SQLException{  //准备一张结构不同的表  db.dropTable(Student.**class**);  String sql="create table STUDENT(\n"+  "ID int generated by default as identity not null,\n"+  "GENDER varchar(6),\n"+  "NAME varchar(255),\n"+  "DATE\_OF\_BIRTH timestamp,\n"+  "constraint PK\_STUDENT primary key(ID)\n"+  ")";  db.executeSql(sql);  //开始刷新表  System.out.println("=== Begin refresh table ===");  db.refreshTable(Student.**class**);  } |

运行上述代码，就可以看到，框架会使用ALTER TABLE语句来修改已经存在的表，使其和模型保持一致。

### 监听和控制表的修改过程

在修改表的过程中，我们可以传入一个事件监听器。事件监听器并不仅仅用于了解刷新表的过程，还可以直接控制这一过程。我们可以先尝试运行位于orm-tutorial中的例子，来观察这一事件。

src\main\java\org\easyframe\tutorial\lessonc\CaseAlterTable.java

|  |
| --- |
| /\*\*  \* 传入一个事件监听器，从而可以监测刷新操作的步骤  \* **@see** jef.database.support.MetadataEventListener  \*/  @Test  **public** **void** testAlterTableProgress() **throws** SQLException{  //准备一张结构不同的表  db.dropTable(Student.**class**);  String sql="create table STUDENT(\n"+  "ID int generated by default as identity not null,\n"+  "GENDER varchar(6),\n"+  "NAME varchar(128),\n"+  "REV\_NAME varchar(255),\n"+  "DATE\_OF\_BIRTH timestamp,\n"+  "constraint PK\_STUDENT primary key(ID)\n"+  ")";  db.executeSql(sql);    System.out.println("=== Begin refresh table ===");  db.refreshTable(Student.**class**,**new** ProgressSample());  } |

运行上述案例，使用者可以看到有一个桌面进度条弹出，进度条中显示了当前的工作进度，包括正在执行的比较工作或者数据库SQL语句。

事实上

|  |  |  |
| --- | --- | --- |
|  |  |  |
| 变更表之前 | （可以控制是否继续刷新操作） |  |
| 如果表不存在， | 创建表之前（可以控制是否建表） |  |
| 如果表存在，在比较表之前 | （可以控制是否继续刷新操作） |  |
| 数据库中的表和元数据比较完成后 |  |  |
| 比较完成并生成SQL后，每句SQL执行之前 |  |  |
| 在每句SQL执行成功后 |  |  |
| 在任何一句SQL执行失败后 |  |  |
| 在所有SQL都执行完成后 |  |  |

按事件的时间顺序如下

### 使用动态表模型来维护数据库结构

在EF-ORM中支持了动态表模型，因此可以用编程接口来构造一个表模型，并且用这个表模型来维护数据库。

src\main\java\org\easyframe\tutorial\lessonc\CaseAlterTable.java

|  |
| --- |
| @Test  **public** **void** testAlterTupleTable() **throws** SQLException{  TupleMetadata model=**new** TupleMetadata("MY\_TABLE");  model.addColumn("ID", **new** ColumnType.AutoIncrement(8));  model.addColumn("NAME", **new** ColumnType.Varchar(64));  model.addColumn("DATA", **new** ColumnType.Varchar(128));  model.addColumn("DOB", **new** ColumnType.TimeStamp().notNull()  .defaultIs(Func.current\_timestamp));  model.addColumn("MODIFIED", **new** ColumnType.TimeStamp().notNull());  model.addColumn("CONTANT", **new** ColumnType.Clob());    //第一次刷新，由于此时表还不存在，因此会创建 MY\_TABLE  db.refreshTable(model, **null**);    //修改模型字段  model.removeColumn("DATA");  model.updateColumn("NAME", **new** ColumnType.Varchar(128).notNull());  model.updateColumn("MODIFIED", **new** ColumnType.TimeStamp());  model.addColumn("DATA1", **new** ColumnType.Varchar(64));  model.addColumn("AGE", **new** ColumnType.Int(12));  //第二次刷新，修改MY\_TABLE的字段  db.refreshTable(model, **null**);  System.out.println("=== begin drop ===");  db.dropTable(model);  } |

## 其他功能

### 执行sql脚本文件

# 其他功能与特性

这里列举一些较难归类的，或者是框架最近新增的特性。这些特性往往能在针对性的环境下简化开发，解决开发中的疑难问题。

## 记录时间戳自动维护

即自动维护每条记录的create-Time字段和update-time字段。

我们可能有这种场景，当数据插入数据库时，希望在某个日期时间型字段中写入当前数据库时间。在每次数据更新时，希望在某个日期时间型字段中写入当前的数据库时间。这其实就是常用的日期时间维护操作。用对象关系映射很难实现这种功能。同时由于需要设置的是数据库时间而不是当前java环境时间，这也给功能的实现造成了一定困难。

EF-ORM 1.9.1.RELEASE开始，原生支持此类操作。简单说来，在日期字段的注解上加上@GeneratedValue，则EF就会自动维护这些日期字段。

|  |
| --- |
| /\*\*  \* 为一个日期列添加@GeneratedValue后，如果在插入时没有指定数据，  \* 将使用sysdate / current\_timestamp填入(具体函数取决于使用的数据库)  \*/  @GeneratedValue(generator="created")  @Column(name="create\_time")  **private** Date created;  /\*\*  \* 为一个日期列添加@GeneratedValue后，如果在插入或者更新时没有指定数据  \* 将使用sysdate / current\_timestamp填入(具体函数取决于使用的数据库)  \*/  @GeneratedValue(generator="modified")  @Column(name="last\_modified")  **private** Date modified; |

一旦进行了上述配置，就会发现在insert或update语句中，会出现sysdate或current\_timestamp等函数，直接获取数据库时间进行操作。

不过这种方式下，数据库时间不会回写到插入或更新的对象中。您需要再执行一次查询操作，才能得到刚刚写入数据库的记录的时间。

另外一种变通的方法是，在插入之前自动为字段赋值，此时的时间为当前java服务器的时间。这种方式下由于是先赋值再插入的，在操作完成后，可以从对象中获得刚才操作的时间。EF-ORM也支持这种方式。

|  |
| --- |
| @GeneratedValue(generator="created-sys") //creatd-sys的意思是插入时取当前系统时间  @Column(name="create\_time")  **private** Date created;    @GeneratedValue(generator="modified-sys") //creatd-sys的意思是插入时或者更新时取当前系统时间  @Column(name="last\_modified")  **private** Date modified; |

使用以上注解后，数据记录的插入和更新时间就可以交由框架自动维护了。

## 内置连接池

EF-ORM内置了一个稳定高效的连接池（但功能较弱）。正常情况下，EF-ORM会检测用户的DataSource是否使用了连接池。如果用户已经使用了连接池，那么EF-ORM不会启用内置连接池。如果检测到用户没有使用连接池，EF-ORM就会启用内置连接池。

内置连接池可以在额定连接数到最大连接数之间变化。定期关闭那些闲置的过多的连接。内置连接池可以定期检查空闲连接的可用性。在取出的连接使用中，如果捕捉到SQLException，并且数据库方言判定这个Exception是由于网络断线等IO错误造成的，会强行触发连接池立刻检查，清除无效连接。并且连接池会不断尝试重新连接，试图从故障中恢复。

EF-ORM启动时，都有一条info日志，如果启用内置连接池，日志如下

|  |
| --- |
| There is NO Connection-Pool detected in data source class {*DataSource类名*}, EF-Inner Pool was enabled. |

如果没有启用连接池，日志如下

|  |
| --- |
| There is Connection-Pool in data source {*DataSource类名*}, EF-Inner Pool was disabled. |

大部分情况下，不需要您关心内置连接池是否开启。自动判断连接池能准确的识别DBCP, C3p0, proxool,Driuid,Tomcat cp, BoneCp等众多的连接池。如果您使用了较为冷僻的连接池造成两个连接池都被启用，您可以使用jef.properties参数（见附录一）强行关闭EF的内置连接池。

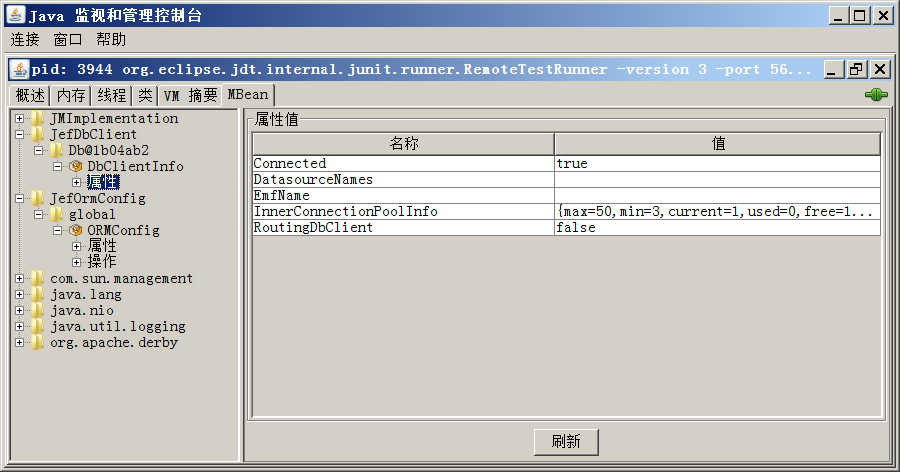
内置连接池相关参数配置可见附录一的“内置连接池相关”部分。

EF内置连接池采用了无阻塞算法，并发下安全，存取非常高效，有定期心跳，拿取检查，脏连接丢弃等连接池的基本功能，但没有PreparedStament缓存等功能。也曾在某大型电信中稳定使用，但现在是连接池层出不穷的年代，c3p0和proxool等老牌连接池都在druid、jboss等连接池面前相形见绌。而内置连接池不是EF-ORM今后发展的目标，您在小型项目中或者快速原型的项目用用没什么问题，大型商业项目建议您还是用tomcat cp或者druid吧。

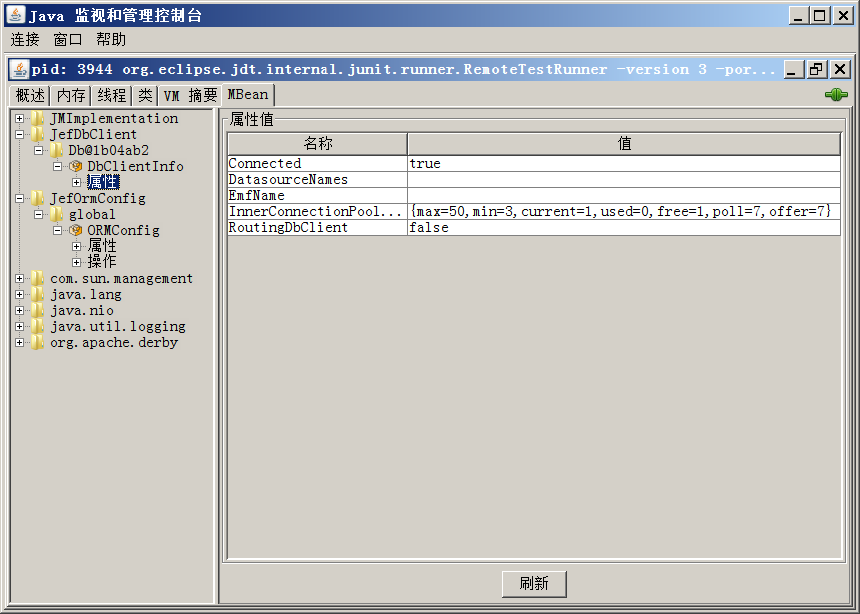
# JMX监控

EF\_ORM支持JMX监控。

目前EF-ORM提供两个JMX监控Bean，分别是DbClientInfo和ORMConfig。路径如下图。



## DbClientInfo

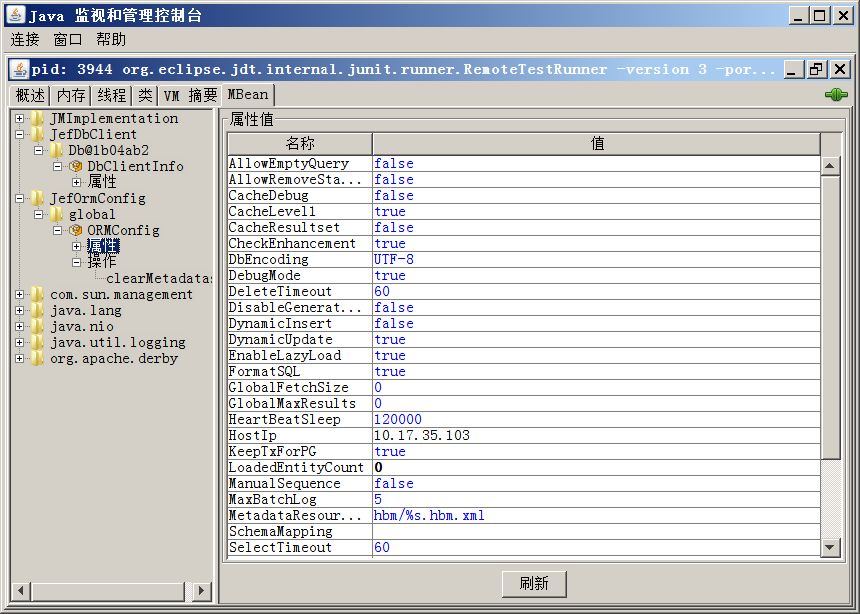


每个DbClient对象对应一个DbClientInfo的监控Bean。在一个进程中，如果有多个DbClient对象，那么也会有多个DbClientInfo的MXBean。

DbClientInfo的五个属性都是只读属性。记录了当前的一些运行情况信息。

|  |  |
| --- | --- |
| 属性 | 含义 |
| Connected | 是否连接中 |
| RoutingDbClient | 是否为多数据源DbClient |
| DataSourceNames | 如果是多数据源的DbClient，会列出所有已知数据源的名称 |
| EmfName | EntityManagerFactory的名称，如果有多个EntityManagerFactory可用于区分。 |
| InnerConnectionPoolInfo | 内置连接池信息。 Max：最大连接数  Min：最小连接数 Current:当前连接数  Used：使用中的连接数 Free:空闲连接数  Poll：连接取用累计次数 Offer:连接归还累计数 |

## ORMConfig



ORMConfig记录框架的各项配置信息，每个进程中仅有一个ORMConfig的MXBean。

**支持动态参数调整**

ORMConfig中的属性都是可读写的属性，即可以在运行过程中调整ORM的各项参数。包括调试日志、连接池大小等。这些参数大多数都和jef.properties中的参数对应。因此可以查看《附录一 配置参数一览》或者API-DOC。

# 保持数据库移植性的实践

本章内容暂缺，为保证章节号一致性，故保留本章标题。

# 性能调优指南

## 性能日志

要了解性能问题所在，首先要能看懂EF-ORM输出的性能日志。

一个查询语句输出的日志可能是这样的

|  |
| --- |
| select t.\* from LEAF t | [hsqldb:TESTHSQLDB@1]  Result Count:5 Time cost([ParseSQL]:0ms, [DbAccess]:1ms, [Populate]:0ms) max:0/fetch:0/timeout:60 |[hsqldb:TESTHSQLDB@1] |

在上面这段日志中，第一行打印出了SQL语句，竖线后面的是此时的环境描述。这部分信息包括三部分。

|  |
| --- |
| [hsqldb:TESTHSQLDB@1]  数据库类型：数据库名@线程号 |

上面是当SQL语句在无事务情况下执行时的环境，当语句在事务下执行时：

|  |
| --- |
| [Tx22953873@ORCL@1  事务编号@数据库名@线程号 |

事务编号中的数字是一个随机编号，用于和日志上下文核对，可以跟踪事务的情况。

而性能相关的统计信息都在这一行中显示

|  |
| --- |
| Result Count:5 Time cost([ParseSQL]:0ms, [DbAccess]:1ms, [Populate]:0ms) max:0/fetch:0/timeout:60  结果条数 耗时 （生成SQL语句耗时,执行SQL语句耗时，转换查询结果耗时） 性能参数信息 |

在上例中，查询出5条结果，耗时1ms，其中生成SQL语句和转换结果0ms，数据库查询1ms。

* ParseSQL:获取连接、生成SQL语句的时间。
* DbAccesss:数据库解析SQL，执行查询的时间。
* Populate: 将数据从ResultSet中逐条读出，形成Java对象的时间。

当然并非所有的数据库操作都有这三个时间记录。比如您自行编写的SQL语句（NativeQuery）中不会有ParseSQL的统计，非select语句不会有Populate的统计。

另外，上例中生成SQL语句和转换结果不可能真的不花费时间，因为统计是到毫秒的，因此500微秒以下的数值就被舍去了。

最后输出的是查询执行时都是三个性能相关的参数。

Max: 返回结果最大数限制。0表示不限制

Fetch:取ResultSet的fetch-size大小。该参数会严重影响大量数据返回时的性能。

Timeout: 查询超时时间。单位秒。

上面三个都是用户可以控制的性能相关参数，用来对照进行调优的。

所有查询类语句，都会输出结果条数（COUNT类语句直接输出COUNT结果）。而非查询语句则会显示影响的记录数。看懂上述日志，可以帮助用户统计一笔交易中，数据库操作的耗时情况，帮助用户分析和定位性能故障。

## 级联性能

前面已经讲过，EF-ORM在支持级联操作的基础上，还保留了单表操作的方式。此外还能控制单个查询语句中需要查询的字段等。首先我们可以考虑在应用场景上避免不当的数据操作。

此外EF-ORM中有若干参数可以辅助级联性能的调节。有以下两个全局参数。

|  |  |
| --- | --- |
| db.use.outer.join | 使用外连接加载多对一和一对一关系。这种情况下，只需要一次查询，就可以将对一关系查询出来。默认开启。如果关闭，那么一对一或多对一级联操作将会通过多次单表查询来实现。 |
| db.enable.lazy.load | 延迟加载开关，即关系数据只有当被使用到的时候才会去查询。由于默认情况下対一关系是一次性直接查出的，所以实际上会被延迟加载的只有一对多和多对多 关系。但如果关闭了外连接加载，那么一对一和多对一关系也会被延迟加载。 |

在查询数据时，我们可以精确控制每个查询是否采用外连接加载，是否要加载X対一关系，是否要加载X对多关系。下面的例子演示了这种用法。

|  |
| --- |
| Query<TestEntity> query = QB.*create*(TestEntity.**class**);  query.getResultTransformer().setLoadVsMany(**false**); //不加载X对多关系  query.getResultTransformer().setLoadVsOne(**true**); //加载X対一关系  query.setCascadeViaOuterJoin(**false**); //设置不使用外连接方式获取X対一关系 |

因此每个查询语句，都可以控制其级联加载的范围，级联加载的方式。

如果不希望级联操作，还可以这样

|  |
| --- |
| Query<TestEntity> query = QB.*create*(TestEntity.**class**);  query.setCascade(**false**); |

这和下面的操作是等效的

|  |
| --- |
| Query<TestEntity> query = QB.create(TestEntity.**class**);  query.getResultTransformer().setLoadVsMany(**false**);  query.getResultTransformer().setLoadVsOne(**false**); |

两个参数的用法上，延迟加载的开启是较为推荐的。这能有效防止你使用级联操作获取过多的数据。大部分情况下，外连接开启也能有效减少数据库操作的次数，提高性能。同时外连接查询能降低对一级缓存的依赖，因为在一些快速查询中，维护缓存数据也有一定的耗时。如果您关闭了外连接查询，那么推荐您开启一级缓存。因为此时级联操作对一级缓存的依赖性大大增加了。

## 一级缓存与二级缓存

EF-ORM设计了一级缓存。一级缓存是以每个事务Session为生命周期维护的缓存，这部分缓存会将您操作过的对象和查询过的数据缓存在内容中。（特别大的数据不会被缓存）一级缓存能有效的减少对相同对象的查询，尤其是在一对多的级联关系查询中。

一级缓存默认不开启，开启一级缓存的方法是在jef,properties中配置

|  |
| --- |
| cache.level.1=true |

使用JMX可以在ORMConfigMBean中，通过设置CacheDebug属性为true，从而在日志中输出一级缓存的命中和更新信息，用于细节上的调试和分析。

以下情况下，我们建议开启一级缓存：

* 使用较多的级联操作。
* db.use.outer.join=false时

相反，如果使用级联操作较少，同时也开启了db.use.outer.join的场合下，我们建议关闭一级缓存。因为基于SQL操作业务逻辑中，维护一级缓存反而会增加额外的内存和性能开销。

EF-ORM没有内置的二级缓存。你可以使用诸如EHCache的第三方缓存框架，并通过Spring AOP等手段集成，此处不再赘述。

## 结果集加载调优

### Fetch-size

即等同于JDBC中的fetch-size，描述了遍历结果集（ResultSet）时每次从数据库拉取的记录条数。设置为0则使用JDBC驱动默认值。过大则占用过多内存，过小则数据库通信次数很多，populate过程耗时很大。

如果您返回5000条以上数据，建议加大fetch-size。

Fetch-size的全局设置：在jef.properties中

|  |
| --- |
| #将全局的fetch-size设置为希望的值  db.fetch.size=0 |

针对单个查询设置fetch-size：所有的ConditionQuery对象，包括Query、Join、UnionQuery、NativeQuery都提供了setFetchSize(int)方法。

|  |
| --- |
| //设置NativeQuery的fetch-size  NativeQuery<Foo> nq=db.createNativeQuery(sql, Foo.**class**);  nq.setFetchSize(1000);    //设置Query对象的fetch-size  Query<Foo> q = QB.create(Foo.**class**)  q.setFetchSize(1000); |

### max-results

这个参数可以控制一个查询返回的最大结果数。事实上一个限制了最大结果数的查询逻辑上不一定正确，但是这能有效预防超出设计者预期数据规模时引起的OutOfMemory或者其他问题，而后者往往会影响整个系统中的所有交易，甚至引起服务器的故障。

因此全局性的max-result设置往往作为一个数据规模的约束条件来使用，而针对单个查询的max-result设置则可以根据应用场景而灵活控制。

|  |
| --- |
| #将全局的max-results设置为希望的值，0表示不限制  db.max.results.limit=0 |

|  |
| --- |
| //设置NativeQuery的max-result  NativeQuery<Foo> nq=db.createNativeQuery(sql, Foo.**class**);  nq.setMaxResults(200);    //设置Query对象的max-result  Query<Foo> q = QB.create(Foo.**class**)  q.setMaxResult(200); |

### 使用CachedRowSet

这个参数目前只支持全局设置。其作用是在查出结果后，先将ResultSet的所有数据放在JDBC的CachedRowSet中，释放连接（仅对非事务操作，因为事务操作下连接被事务专用，在提交/回滚前不会放回连接池），然后再转换为java对象，最后释放CachedRowSet。这种操作方式具有以下特点

* 它不能减少查询结果转换的总时间，因为原先转换结果该进行的操作一步也没有少。
* 在非事务下，连接能更快的被释放。供其他业务使用。
* 它会将从ResultSet中读取数据的时间计入DbAccess阶段，使得Populate阶段的时间仅剩下调用反射操作所耗的时间。此时用户可以更清楚的知道，转换结果操作的真实性能开销。也帮助用户了解在ResultSet上的IO通信是否值得增加fetch-size来优化。

调节是否开启此功能的 方法为，在jef.properties中

|  |
| --- |
| #开启结果集缓存  db.cache.resultset=true |

## 查询超时控制

查询超时控制可以让一个SQL操作在执行一段时间后，如果无返回则抛出异常。这虽然会造成当前业务的失败，但是可以帮助您从以下几个方面改善程序的性能：

* 避免让个别不佳的SQL语句或超出开发者规模预期的查询拖慢整个系统。
* 避免数据库崩溃
* 发现锁表现象。（个别查询是因为锁表而被卡住，不主动查询数据库往往发现不了）

控制超时时间的参数设置方法为，在jef.properties中

|  |
| --- |
| #设置查询操作超时时间（单位：秒）  db.select.timeout=0  #设置更新操作超时时间（单位：秒）  db.update.timeout=0  #设置删除操作超时时间（单位：秒）  db.delete.timeout=0 |

目前尚未提供针对单个查询设置timeout的方法，后续版本中会增加相关API。

## 自增值获取性能问题

很多时候插入不够快是因为Sequence自增值获取的性能开销造成，优化方法详见3.1.2.5。

# 常见问题 (FAQ)

### JDBC驱动问题

EF-ORM在很多DDL处理和函数改写等特性上，使用了JDBC4的一些功能，因此也要求尽可能使用支持JDBC4的数据库驱动，比如Oracle驱动请使用ojdbc5或者ojdbc6。其他驱动选择也请尽可能用最新的。

### JDK7编译后的ASM兼容

在1.8.0版本中，如果使用JDK7的编译版本编译了Entity，那么类在增强后无法在Java 7的版本下使用。这是因为JDK7使用了新版本的编译和类加载机制，将类型推断移到编译时进行，此时ASM修改后的类由于缺少类型推断信息故不能被JDK7加载。解决这个问题的办法是，在虚拟机启动参数中加上

|  |
| --- |
| -XX:-UseSplitVerifier |

EF-ORM 1.8.8以后的版本，在实体增强时会将类版本降低到50(对应JDK6)，因此不会发生此错误。

### 数据库存储的口令加密

可以配置加密后的数据库口令作为密码，EF-ORM在查询数据源时，会尝试解密。默认解密采用3DES算法，密钥可以在jef.properties中配置或者通过虚拟机启动参数指定。

### Oracle RAC环境下的数据库连接

默认在Oracle RAC下，使用Oracle驱动的FCF或者OCI驱动的TAF方式进行连接失效转移。（不明白什么是FCF和TAF的请自行百度）。这两种方式将会使用Oracle驱动的连接池，因此可以手工关闭EF-ORM的内建连接池。

|  |
| --- |
| Db.no.pool=true |

实际上，使用JTA事务等场景下，也要关闭内置连接池，不过此时关闭内建连接池是自动进行的。

### 某些正确的SQL语句解析错误怎么办

目前的词法分析器经过大量SQL语句的测试，都能正确解析。一个已知的问题是，JPQL参数名称不能取SQL关键字作为名称，比如”select \* from t where id=:top” 、“update t set name=:desc”等，这些语句都会解析失败，因为top，desc等是SQL关键字。因此，如果碰到解析错误的SQL，尝试先改变一下JPQL参数的名称，是不是用到了SQL关键字。任何复合词都不是SQL关键字。

如果确实有解析不了的SQL语句，请用7.5节的方法，直接使用原生SQL语句。当然原生SQL语句不具备分库分表等高级功能，您需要自行处理。

### 使用Jackjson做JSON序列化怎么办

Jackson是常见的JSON序列化工具，Jackson默认的序列化策略中会将DataObject中的Query，UpdateValueMap取出来进行序列化，这往往不是我们需要的。

为此，我们可以使用以下两种方法来避免出现这种情况——

方法一：只有Getter和Setter齐全的字段才被序列化。为ObjectMapper增加配置即可——

Jackson 1.x

|  |
| --- |
| objectMapper.configure(Feature.REQUIRE\_SETTERS\_FOR\_GETTERS, true); |

Jackson 2.x

|  |
| --- |
| objectMapper.configure(MapperFeature.REQUIRE\_SETTERS\_FOR\_GETTERS, true); |

方法二：忽略掉不希望被序列化的字段

在Entity类上增加注解

|  |
| --- |
| @JsonIgnoreProperties({"query","updateValueMap"})  public class MyClass{  … |

这两个方法任选其一，可以在Jackson下正常序列化Entity。

# 附录一 配置参数一览

jef.properties中可配置的参数说明。

|  |  |  |
| --- | --- | --- |
| **参数名** | **用途** | **默认值** |
| **各项数据库操作相关** | | |
| allow.empty.query | 允许为空的查询条件 | false |
| allow.remove.start.with | 当用户将oracle的 start... with... connect by这样的语句在其他不支持此特性的数据库上运行时，允许改写这部分语句以兼容其他数据库。  如果关闭此特性，此时会抛出异常，提示当前数据库无法支持 start with 这样的语法。 | false |
| db.dynamic.insert | 全局动态插入方式，不设值的字段不插入。 | false |
| db.dynamic.update | 全局动态更新方式，不设值的字段不更新。 | true |
| db.encoding | 数据库文本编码，当将string转为blob时，或者在计算字符串插入数据库后的长度时使用（java默认用unicode，中文英文都只占1个字符） | 系统defaultcharset |
| schema.mapping | 多Schema下：schema重定向映射表。 | - |
| db.datasource.mapping | 多数据源下:数据源名称重定向映射表。 | - |
| db.tables | 配置类名，数据库启动时扫描并创建这些类对应的表。 | - |
| db.init.static | 数据库启动时默认加载的类，用于加载一些全局配置 | - |
| db.force.enhancement | 检查每个加载的entity都必须增强，否则抛出异常。 | true |
| db.specify.allcolumn.name | 当选择t.\*时，是否指定所有的列名。有一种观点认为用t.\*的时候性能较差。 | false |
| db.password.encrypted | 配置为true，则对数据库密码做解密处理 | false |
| db.encryption.key | 数据库密码解密的密钥。会按3DES算法解密 | - |
| db.blob.return.type | string，配置blob的返回类型,支持 string/byte/stream/file 四种。当使用Object类型映射Blob，又或者在动态表中使用BLOB时，查询出的blob字段将被转换为指定类型进行处理。 | stream |
| db.enable.rowid | 启用oracle rowid支持。 目前支持不太好，请勿使用。 | false |
| db.keep.tx.for.postgresql | 通过记录savepoint的方式，为postgresql进行事务的保持。 | true |
| **分库分表功能相关** | | |
| db.single.datasource | 取消支持多数据源，原先配置的数据源绑定等功能都映射到当前数据源上 | false |
| partition.create.table.inneed | 按需建表开关 | true |
| partition\_filter\_absent\_tables | 过滤不存在的表开关 | true |
| partition\_inmemory\_maxrows | 当分库操作时，不得不进行内存排序和聚合计算时，限制最大操作的行数，防止内存溢出。一旦达到最大行数，该次操作将抛出异常。0表示不限制。 | 0 |
| partition.date.span | 当日期类型的分表分库条件缺失时，默认计算当前时间前后一段时间内的分表。 | -6,3 |
| db.partition.refresh | 当前数据库中存在哪些分表会被缓存下来。设置缓存的最大失效时间(单位秒)，默认3600秒，即一小时。 | 3600 |
| **开发和调试相关** | | |
| db.format.sql | 格式化sql语句，打印日志的时候使其更美观 | true |
| db.named.query.update | 是否自动更新namedqueries查询配置，开启此选项后，会频繁检查本地配置的（或数据库）中named-query配置是否发生变化，并加载这些变化。这是为了开发的时候能随时加载到最新的配置变动。 | 和db.debug一致 |
| db.log.format | sql语句输出格式。默认情况下面向开发期间的输出格式，会详细列出语句、参数，分行显示。但在linux的命令行环境下，或者系统试运行期间，如果要用程序统计SQL日志，那么分行显示就加大了统计的难度。为此可以配置为default / no\_wrap，no\_wrap格式下，一个SQL语句只占用一行输出。 | default |
| db.encoding.showlength | 在日志中，当文本过长的时候是不完全输出的，此时要输出文本的长度。 | false |
| db.max.batch.log | 在batch操作时，为效率计，不会打印出全部项的参数。 只会打印前几个对象的参数。配置为5的时候就打印前五个对象。 | 5 |
| db.query.table.name | 配置一张表名，启用数据表存放namedquery功能。一般配置为jef\_named\_queries | - |
| **自增主键和Sequence操作相关** | | |
| auto.sequence.offset | 在自动创建sequence (或模拟用的table)时，sequence的校准值。-1表示表示根据表中现存的记录自动校准(最大值+1)。当配置为正数时，为每张表的初始值+配置值。 | -1 |
| auto.sequence.creation | 允许自动创建数据库sequence (或模拟用的table)，一般用在开发时和一些小型项目中，不适用于对用户权限有严格规范的专业项目中。 | true |
| db.autoincrement.hilo | 如果自增实现实际使用了sequence或table作为自增策略，那么开启hilo优化模式。即sequence或table生成的值作为主键的高位，再根据当前时间等自动填充低位。这样做的好处是一次操作数据库可以生成好几个值，减少了数据库操作的次数，提高了性能。  如果实际使用的是Identity模式，那么此项配置无效。  false时仅有@HiloGeneration(always=true)的配置会使用hilo。  一旦设置为true，那么所有配置了@HiloGeneration()的自增键都会使用hilo. | false |
| db.autoincrement.native | jpa实现关于自增实现分为  identity: 使用数据库表本身的自增特性  sequence： 使用sequence生成自增  table： 使用数据库表模拟出sequence。  Auto: 根据数据库自动选择 Identity > Sequence > Table (由于数据库多少都支持前两个特性，所以实际上Table默认不会被使用 \* </ul>  开启此选项后，即便用户配置为Sequence或IDentity也会被认为采用Auto模式。(Table模式保留不变) | true |
| sequence.name.pattern | 全局的sequence名称模板。如果本身配置了sequence的名称，则会覆盖此项全局配置。 | %\_SEQ |
| sequence.batch.size | 如果自增实现实际使用了sequence或table作为自增策略，那么每次访问数据库时会取多个sequence缓存在内存中，这里配置缓存的大小。 | 50 |
| db.support.manual.generate | 是否允许手工指定的值代替自动生成的值 | false |
| db.sequence.step | 如果自增实现中实际使用了sequence作为自增策略，那么可以step优化模式。即sequence生成的值每次递增超过1，然后ORM会补齐被跳过的编号。  (对identity,table模式无效)当配置为  0: Sequence：对Oracle自动检测，其他数据库为1  -1:Sequence模式下：对所有数据库均自动检测。在非Oracle数据库上会消耗一次Sequence。  1: Sequence模式下：相当于该功能关闭，Sequence拿到多少就是多少  >1的数值:Sequence模式下：数据库不再自动检测，按配置值作为Sequence步长（如果和实际Sequence步长不一致可能出错） | 0 |
| db.public.sequence.table | 配置一个表名作为公共的sequence table名称。 | - |
| **默认数据库连接（空构造DbClient时连接数据）** | | |
| db.type | 当使用无参数构造时，所连接的数据库类型 | - |
| db.host | 当使用无参数构造时，所连接的数据库地址 | - |
| db.port | 当使用无参数构造时，所连接的数据库端口 | - |
| db.name | 当使用无参数构造时，所连接的数据库名称（服务名/sid） | - |
| db.user | 当使用无参数构造时，所连接的数据库登录 | - |
| db.password | 当使用无参数构造时，所连接的数据库登录密码 | - |
| db.filepath | 当使用无参数构造时，所连接的数据库路径（本地的文件数据库，如derby hsql sqlite等） | - |
| **数据库性能相关参数** | | |
| cache.level.1 | 是否启用一级缓存。当启用了db.use.outer.join参数后，一级缓存的作用和对性能的影响其实没有那么大了。但是如果关闭db.use.outer.join后，还是建议要开启一级缓存。 | false |
| db.cache.resultset | 将resultset的数据先放入缓存，再慢慢解析(iterated操作除外)。 使用这个操作可以尽快的释放连接，适用于当连接数不够用的情况。 | false |
| db.no.remark.connection | true后禁止创建带remark标记的oracle数据库连接。 带remark标记的oracle连接中可以获得metadata中的remarks，但是连接上的所有操作都变得非常慢。 | false |
| db.fetch.size | 数据库每次获取的大小。 0表示按JDBC驱动默认。 | 0 |
| db.max.results.limit | 数据库查询最大条数限制，0表示不限制。 | 0 |
| db.delete.timeout | 数据库删除超时,单位秒 | 60 |
| db.update.timeout | 数据库查询超时，单位秒。实际操作中各种存储过程和自定义的sql语句也大多使用此超时判断 | 60 |
| db.select.timeout | 数据库查询超时，单位秒。 | 60 |
| db.use.outer.join | 使用外连接的方式一次性从数据库查出所有对单关联的目标值。 | true |
| db.enable.lazy.load | 级联延迟加载特性，默认启用。由于默认情况下开启了db.use.outer.join特性，因此实际上会使用延迟加载的都是一对多和多对多。 | true |
| db.lob.lazy.load | LOB延迟加载。现还不够稳定，请谨慎使用 | false |
| **内置连接池相关** | | |
| db.no.pool | 禁用内嵌的连接池，当datasource自带连接池功能的时候使用此配置。默认情况下，EF-ORM能识别常见的dbcp,druid,c3p0,proxool,bonecp,tomcatcp等连接池。一旦发现使用了这些连接时就会自动禁用内嵌连接池。因此一般此参数无需修改。 | false |
| db.connection.live | 每个连接最小生存时间，单位毫秒。 | 60000 |
| db.connection.pool | jef内嵌连接池最小连接数,数字 | 3 |
| db.connection.pool.max | jef内嵌连接池最大连接数,数字 | 50 |
| db.heartbeat | jef内嵌连接池心跳时间，按此间隔对空闲的连接进行扫描检查，单位毫秒。 | 120000 |
| db.pool.debug | jef内嵌连接池调试开关，开启后输出连接池相关日志 | false |
| **其他行为** | | |
| table.name.translate | 自动转换表名(为旧版本保留，如果用户没有通过jpa配置对象与表名的关系，那么开启此选项后， userid -> user.id， 否则userid -> userid | False |
| custom.metadata.loader | 扩展点，配置自定义的元数据加载器 元数据默认现在都是配置在类的注解中的，使用自定义的读取器可以从其他更多的地方读取元数据配置。 | - |
| db.operator.listener | 配置一个类名，这个类需要实现jef.database.support.dboperatorlistener接口。 |  |
| partition.strategy.loader | 分表和路由规则加载器类.默认会使用基于class的annotation加载器，也可以使用资源文件 默认实现类：jef.database.meta.partitionstrategyloader$defaultloader 使用者可以实现jef.database.meta.partitionstrategyloader编写自己的分表规则加载器。 |  |
| partition.disabled | 尚未使用 |  |
| metadata.resource.pattern | 内建的metadata加载器可以从外部资源文件xml中读取元数据配置 如果此选项不配置任何内容，那么就取消外部元数据资源加载功能。 |  |
| db.dialect.config | 可以配置一个properties文件，文件中指定要覆盖哪些数据库方言和方言类的名称。通过此功能可以实现数据库方言覆盖。 |  |
| db.check.sql.functions | 默认在执行NativeQuery会对SQL语句中所有的function进行检查，如果认为数据库不支持则将抛出异常。设为false可以关闭此检查。 | true |

注一：所有EF-ORM中的参数，都可以在启动虚拟机时用虚拟机参数覆盖，如-Ddb.debug=true，将会覆盖jef.properties中的db.debug=false的配置。

# 附录二 数据库兼容性说明

完全支持以下数据库——

* Oracle 9.0以上版本、
* MySQL 5.0以上版本
* MariaDB 5.5测试通过。 10.x版本尚未测试。
* Postgresql 8.2以上版本
* Microsoft SQL Server 2003或以上
* HSQLDB 2.0以上版本
* Apache Derby 10.5以上版本
* SQL ite（部分DDL该DBMS无法支持）

部分支持——

* Gbase

计划支持——因时间或工作量因素，Dialect尚未编写的

* IBM DB2
* Sybase
* MongoDB -JDBC