The Open For Business Project: 实体引擎指南

Written By: David E. Jones, jonesde@ofbiz.org
Minor corrections by: Pawel H. Debski
Last Updated: January 28, 2003
Translated by: Lei1 Xiao, lei1@linux.net
Translated Date: November 02, 2003

内容目录

相关文档

实体指南配置指南
核心配置指南
核心 JavaDoc(包括实体引擎)
实体模型 DTD
实体组 DTD
域类型模型 DTD

导言

实体引擎是一组工具和模式(patterns)的集合，用来建模和管理实体数据。在本文中，实体是由一组字段(field)、以及与其他各种实体之间的关系(relation)所构成的数据片段。此定义来自于关系数据库管理系统的标准实体-关系建模观点。实体引擎的目标是简化实体数据的企业级运用。它包含了定义、维护、质量保证、以及实体相关功能的开发等。

实体引擎运用了大量的业务层和集成层模式。在 OFBiz 项目中也应用了大量的展示层模式，但只在 servlet controller 中，而不是在实体引擎中。实体引擎中用到的模式包括：业务

代理(Business Delegate)、值对象(Value Object)、组合实体(Composite Entity)、值对象装配器(Value Object Assembler)、服务定位器(Service Locator)、和数据访问对象(Data Access Object)等。其他模式的实现以及上面这些模式更完整的实现已纳入计划中

这些模式的描述可以从 sun 公司出版的书中获得：“Core J2EE Patterns” by Alur，Crupi，and Malks。用户也可以从此书作者的站点上找到相关信息。参见 http://developer.java.sun.com/developer/restricted/patterns/J2EEPatternsAtAGlance.html。注意，为了浏览这些信息，用户需要一个 Java 开发人员连接帐号。

除此之外，还运用了大量来自 TheServerSide.com 的 Floyd Marinescu 的书“EJB Design Pattern”中的模式。这些包括 Data Transfer HashMap 和通用属性访问(Generic Attribute Access)等等。其中的唯一主键生成(unique primary key generation)模式，用到了最初的模式形式，我们更喜欢把它叫着“Ethernet Key Generation”模式，因为此模式中用到了冲突检测机制，为的是确保多个服务器能够利用单个数据库来获得唯一的键值。而和数据库之间无关。

实体引擎的基本目标是在各种事务性应用领域中尽可能消除实体对持久化代码的特殊需求。当然对报表和类似系统来说，这可能不是其关注点，但对于事务性系统来说，它们是每日运转着的所有业务的基础。实体引擎能节约大量的开发工作量，并能奇迹般的减少系统中与持久化相关的各种 bug。这些应用包括了商业、会计、库存和仓储管理、人力资源管理等等。其实实体引擎对报表和分析系统也是非常有用的，但事实上并不意味着能够满足在这些工具中经常发生的大量定制化查询。

为了使代码和特定的实体尽量不相关，所有的值对象都是通用的，运用了 Map 来按名字存储实体实例的字段值。字段的 get 和 set 方法接收一个包含字段名的字符串(字段名用来验证字段是属于此实体的)，然后就可以获得或设置期望的值。在实体引擎和应用程序之间使用 contract 时，降低了其灵活性所带来的危险，这一点包含在特殊的 XML 文件中。

不用直接写出特定实体的代码，而是从 XML 文件中读取出实体定义，然后实体引擎用它在应用程序和 data source(加入它是数据库或其他的 source)之间加上一组规则。在这些 XML 实体定义中指定了每个实体的名字、各种字段、对应着哪一张数据库表、以及每个字段在表中对应哪一列等。同时也为每个字段指定了其类型，然后用来在给定数据源(data source)的字段类型文件中查找对应的 Java 和 SQL 数据类型。实体之间的关系也定义在此 XML 文件中。通过指定相关表(related table)、关系类型(one 或 many)和关系的各种键映射(keymap)来定义关系。也可以给关系加上一个 title，然后此 title 成了关系名字的一部分，用来区分各种关系。

把实体引擎作为抽象层，特定实体的代码就很容易创建和修改。利用实体引擎 API 来与实体之间交互的代码就能够以各种方式部署，因此实体持久化可以各不相同，并且在上层没有改变与这些实体交互的代码。这种抽象方式好处的一个典型例子是：通过改变配置文件，使用实体引擎的应用程序能够从数据库的 JDBC 直接访问方式切换到 EJB 服务器(即通过实体 Bean 来持久化)。同样的代码也能够用于定制化的数据源，例如在同样的框架内通过实现很少的代码，基于 HTTP 或消息服务的遗留系统。

实体建模

实体建模文件和其路径

开始一个新实体的第一件事是定义或建模此实体。在 OFBiz 的实体引擎中通过两个 XML 文件来完成，一个是关于实体建模的，另一个是关于字段类型建模的。它们都链接到上面相关文档中的的相应 XML DTDs 文档。这两个文件被分开的主要原因是字段类型的定义是和特定数据库相关的。换句话说，对给定的实体模型定义，不同的数据库有不同的字段类型定义。当定义了持久化服务器时，也定义了此服务器要用的字段类型模型(field type model)XML 文

件。

OFBiz 的主要实体模型 XML 文件能够在 ofbiz/commonapp/entitydef/下找到。最初的所有实体都在文件 entitymodel.xml 中，但是现在它们被拆分到了不同的文件中(包括 entitymodel.xml 中)，以下列模式命名：entitymodel_*.xml.

OFBiz 的 MySQL 字段类型模型 XML 文件在文件 ofbiz/commonapp/entitydef/fieldtypemysql.xml 中能够找到。也有其他数据库的字段类型文件，例如：PostgresSQL、Hypersonic、Oracle 等等。根据实体模型文件和字段类型文件，就能通过 GenericHelper 接口上的 CheckDataSource 例程(routine)自动创建数据库表。这一步在启动的时候或者通过 WebTools 中的工具自动执行。

加载种子数据(Loading Seed Data)

当自动创建完表的同时，必需从数据文件中加载数据。这些数据文件要么是 SQL 脚本，要么是 XML 实体引擎文件。所有类型信息以及其他预加载的信息，例如 status、enumeration、geo 数据等等，都位于 ofbiz/commonapp/db 下的 XML 实体引擎文件中。这些文件能够通过 WebTools 中的 install.jsp 页面来自动定位和自动加载。此页面在 entityengine.xml 文件中给定的实体组名字(entity group name)指定的目录下查找，并且查找所有的.xml 和.sql 文件。找到的文件被列表出来，并且请求用户的确认。当点击 Yes 时，Load Now 链接将加载这些文件。错误信息将同时出现在此页面和控制台上。在加载单个文件的 Form 中，通过指定.sql 或.xml 文件的全路径，数据文件也可以一次加载一个。通过 WebTools 中的导入和导出页面，也能够导入和导出 XML 实体引擎文件。

实体定义的一个例子

正如上面提到的，实体包含了一组字段以及一组与其他各种实体的关系。除此之外，在实体的 XML 定义中还包含实体自身的属性，例如实体名、对应的表名、实体对应的包名，以及关于实体的元数据，例如作者、版权声明、描述等。这里是一个 XML 实体定义的例子:

```
<entity title="Sample Entity"
copyright="Copyright (c) 2001 John Doe Enterprises"
            author="John Doe" version="1.0"

            package-name="org.ofbiz.commonapp.sample"
            entity-name="SampleEntity"
            table-name="SAMPLE_ENTITY">
       <field name="primaryKeyFieldOne" col-name="PRIMARY_KEY_FIELD_ONE" type="id-
ne"></field>

      <field name="primaryKeyFieldTwo" type="id-ne"></field>
      <field name="fieldOne" type="long-varchar"></field>
      <field name="fieldTwo" type="long-varchar"></field>

      <field name="foreignKeyOne" type="id"></field>
      <prim-key field="primaryKeyFieldOne" />
      <prim-key field="primaryKeyFieldTwo" />

      <relation type="one" rel-entity-name="OtherSampleEntity">
        <key-map field-name="foreignKeyOne" rel-field-name="primaryKeyOne" />
      </relation>

      <relation type="one" title="Self" rel-entity-name="SampleEntity">
        <key-map field-name="primaryKeyFieldOne" />
```

```
        <key-map field-name="primaryKeyFieldTwo" />

      </relation>
      <relation type="many" title="AllOne" rel-entity-name="SampleEntity">
        <key-map field-name="primaryKeyFieldOne" />

      </relation>
    </entity>
```

上面是一个演示了部分信息的实体例子。在顶部的所有元数据都是可选的，如果没有指定，则将使用实体模型文件中的相应默认值。例如，没有指定“description”元素，那么ModelReader将用XML文件顶部指定的描述信息(如果存在的话)。Package-name用来组织实体。当实体模型中有成百上千个实体时将变的非常有用。

实体和域定义

注意到如果字段primaryKeyFieldOne有一个指定的列名时，那么其他的字段都不能有指定的列名。Col-name和table-name元素可选。它们能够按照通常的使用约定而从字段名或实体名字中继承过来。这些使用约定能够戏剧性的简化实体定义。

表名和列名以下划线来分开单词(in caps)，类似SAMPLE_ENTITY。实体名和字段(field)名类似Java中类名与域(field)名的约定，即所有的字母都是小写，除了每个单词的第一个字母是大写外。实体名对应Java中类名，即第一个字母是大写。而字段名对应类的成员域，因此第一个字母是小写。例如：SampleEntity和fieldOne分别对应SAMPLE_ENTITY和FIELD_ONE。

通过多个<prim-key>(指定了主键字段的名字)标签能够指定多个主键列。

通过类型字符串来指定字段类型，这些类型字符串在字段类型模型(fieldtypemodel) XML文件中，而字段类型模型XML文件又遵循fieldtypemodel.dtd。每种类型映射到一种Java类型和SQL类型。不同的数据库对应有不同的字段类型模型XML文件，因此一个实体模型XML文件能够工作于各种数据库。除此之外，能为字段类型或任何字段指定validators，表示当有数据输入时，指定的validators应该被调用。这些validators定义在类org.ofbiz.core.util.UtilValidate中，并遵循定义模式：[boolean isValid (String in);]。

实体关系

每个实体能够有多个关系。每个关系类型要么是“one”、“one-nofk”、或“many”，倚赖于关系的基数(cardinality)以及是否需要有外键。如果类型是“one”或“one-nofk”，那么key-map元素必需全部(fully)指定相关实体(related entity)的主键。如果类型是“many”，key-map元素不会受到相关实体主键的限制。

外键和基于外键上的索引能够通过实体引擎来自动创建。但只针对类型是“one”的关系，不包括“one-nofk”和“many”关系。

如果给定实体的多个关系指向同一个相关实体，那么为了使关系名唯一，必需指定一个title。按照这种约定，关系名定义为[title][rel-table-name]。例如：对于两个SampleEntity关系，它们的名字是SelfSampleEntity和AllOneSampleEntity。对OtherSampleEntity则没有title，因此关系名只是简单的OtherSampleEntity，或者是相关实体的名字。

键映射(Key Maps)用来定义怎么查询相关实体。每个键映射指定了一个字段名(field-

name)和一个相关字段名(rel-field-name)。如果两列的名字相同，则不用指定 rel-field-name。

实体元素引擎

entity

属性名	是否必需?	描述
entity-name	Y	用在实体引擎的 Java API 和其他地方时，所引用的实体名
table-name	N	和实体对应的数据库表名字。此属性可选，如果没有指定，则数据库表名将从实体名字派生出来
package-name	Y	实体所在包的名字。在大型数据模型中，有成百上千个实体，因此包用来组织各种实体定义
dependent-on	N	用来指定父实体或此实体倚赖的实体。现在已经不再使用，因为在实体引擎中都已自动实现，但是保留在这里，主要是用来指定层次化的实体结构
enable-lock	N	指定此实体是否需要使用优化锁(optimistic locking)。在此实体上必需存在 lastUpdatedStamp 字段，用来跟踪实体实例上一次被更新的时间。如果当前被更新的实例和 lastUpdatedStamp 不匹配，则将抛出 EntityLockedException 异常。必需是 true 或 false。默认是 false
never-cache	N	如果设置为 true，则不允许 cache 实体。也不会调用自动 cache 清理来提高效率，并且任何试图调用实体上的 cache 方法都将抛出异常。必需是 true 或 false。默认是 false
title	N	实体的 title。如果没有指定，默认是实体所在文件的全局设置
copyright	N	实体版权。如果没有指定，默认是实体所在文件的全局设置
author	N	实体作者。如果没有指定，默认是实体所在文件的全局设置
version	N	实体版本。如果没有指定，默认是实体所在文件的全局设置

子元素名	多少	描述
description	0 or 1	实体描述。如果没有指定，默认是实体所在文件的全局设置。此元素没有属性并且只应该包含一个简单字符串
field	1 to many	用来声明域是实体的一部分
prim-key	0 to many	用来声明哪些域是主键
relation	0 to many	用来声明实体之间的关系

field

属性名	是否必需?	描述
name	Y	字段名，在 Java 代码和其他地方用来引用它
col-name	N	对应的数据库列名。可选属性，如果没有指定，它将从字段名派生出来

type	Y	字段类型。在运行时(run-time)用来在当前数据源的字段类型文件中查找，并决定字段和数据库列的 Java 和 SQL 类型

子元素名	多少	描述
validate	0 to many	每个 validate 元素都有一个简单的 name 属性，此属性指定了要调用的验证方法的名字。这些方法并不是在所有的实体操作中都调用，而是仅用于通用用户接口，例如 WebTools 中的实体数据维护页面

prim-key

属性名	是否必需?	描述
field	Y	字段名，是主键的一部分

relation

属性名	是否必需?	描述
type	Y	指定了包含关系（单向）基数(cardinality)的关系(relationship)类型，并且是否对基数为 1 的单关系(one relationship)创建外键。必需是“one”、“one-nofk”、或“many”
title	N	对单个实体，用户可能需要多个关系，因此此属性允许用户指定一个(前)附加到 rel-entity-name 上的 title，用来构造关系名。如果没有指定，则 rel-entity-name 独自用着关系名
rel-entity-name	Y	相关实体名字。关系是从本实体到相关实体
fk-name	N	能够根据关系名自动创建外键，但有两个理由不推荐这样做对外键和索引，很多数据库有一个很小的最大长度限制(大约 18 个字符)，并且很多数据库需要 FK 名字在整个数据库中是唯一的，而不是只针对外健的来源表

子元素名	多少	描述
key-map	1 to many	key-map 用来指定此实体的字段对应到相关实体中的哪个字段。此元素有两个属性：field-name 和 rel-field-name。这些属性用来指定此实体的字段名和相关实体上的字段名。

视图实体建模

除了直接映射到简单关系型表的实体之外，用户也能够创建映射到多个实体的“virtual”或“view”实体。视图实体和 Oracle、Access、以及其他流行数据库管理系统中视图的概念相同。视图实体能够让用户结合或联合多个实体来生成一个新的实体。通过创建一组包含key-maps 的 view-links，把视图的成员实体链接起来，类似上面描述的关系。

视图实体的元数据几乎和正常实体的相同。它也有 name、package、description、copyright、author、version 等等。它没有表名，因为在数据源中它并不对应到一个简单的关系型表，视图实体也可以分配给实体组 XML 文件中的一个组，类似任何正常实体那样。视图实体中的所有成员实体必需在同一个数据库中，但未必在同一实体组中。

```
   <view-entity title="Sample View Entity"
           copyright="Copyright (c) 2001 John Doe Enterprises"
```

```
            author="John Doe" version="1.0"

            package-name="org.ofbiz.commonapp.sample"
            entity-name="SampleViewEntity">
      <member-entity entity-alias="SE" entity-name="SampleEntity" />

      <member-entity entity-alias="OSE" entity-name="OtherSampleEntity" />
      <alias entity-alias="SE" name="primaryKeyFieldOne" />

      <alias entity-alias="SE" name="primaryKeyFieldTwo" />
      <alias entity-alias="SE" name="fieldOne" />

      <alias entity-alias="SE" name="fieldTwo" />
      <alias entity-alias="OSE" name="primaryKeyOne" />

      <alias entity-alias="OSE" name="otherFieldOne" field="fieldOne" />
      <view-link entity-alias="SE" rel-entity-alias="OSE">

        <key-map field-name="foreignKeyOne" rel-field-name="primaryKeyOne" />
      </view-link>
      <relation type="one" rel-entity-name="OtherSampleEntity">

        <key-map field-name="primaryKeyOne" />
      </relation>
      <relation type="many" title="AllOne" rel-entity-name="SampleEntity">

        <key-map field-name="primaryKeyFieldOne" />
      </relation>
    </view-entity>
```

成员实体

视图实体中的正常实体被作为成员实体来引用。每个成员实体有一个别名，然后在剩下的定义中使用此别名来引用它。这样就允许一个简单实体能够被链接多次。

字段别名(Field Aliases)

在视图实体中为字段指定别名，不如这种方式：每个别名都当着字段来使用，并且按照映射到别名成员实体上的字段的方式来定义。如果字段名和别名相同，就无须再指定它。换句话说，如果没有指定字段名，那么别名将用着字段名并映射到 entity-alias 指定的成员实体上。(译者注：这一段比较费解，请参考原文)

视图链接(View links)

视图链接用来指定视图的成员实体之间的链接。它们把一个 entity-alias 链接到另一个 entity-alias，这里的 key-maps 正好类似于关系。这里 field-name 指定了别名实体上的字段名，同时 rel-field-name 指定相关别名实体上的字段名。如果 rel-field-name 和 field-name 一样，则 rel-field-name 可选。

为了表示 outer join，用户能够在 view-link 元素中用 rel-optional 属性来指定相关实体是可选的，rel-optional 要么是“true”，要么是“false”。依靠 entityengine.xml 文件中的设置，实体引擎将产生 ANSI、Oracle Theta、或 MS SQL Server Theta 样式的 join 代码。更多的信息请参看实体配置指南。

主键

类似一般实体，视图实体也有主键。通过查找别名字段，实体引擎自动决定哪些别名是主键。

视图实体的主键应该包括每个成员实体的所有主键字段。这就意味着它的所有主键字段都必需是别名化的，但用来链接实体的字段只需要被别名一次。例如，如果 OrderHeader 实体有一个 orderId 主键，OrderLine 实体也有一个 orderId 主键和 orderLineSeqId 主键，那么视图实体将有两个主键：orderId 和 orderLineSeqId。这种情况下，orderId 将只被别名一次，因为按照定义，来自每个实体的 orderId 将总是相同，这时由于这些实体是被链接(或 joined)起来的。

关系

视图实体中的关系类似一般实体中的关系。Key-map 的属性还是叫 field-name 和 rel-field-name，但是在视图实体中，字段名实际上是别名。

分组和概要数据(Grouping and Summary Data)

在视图实体中另一个有用的特征就是分组和概要数据特征。这是通过 alias 元素上的两个属性来完成的：group-by 和 function。处理方式与 SQL 中的分组和函数相同。

Group-by 属性能够设置成“true”或“false”，自然默认值就是“false”了。当设置为 true 时，查询结构将按照给定的别名字段进行分组。这就意味着所有的值被组织在一起。

Function 属性用来概要(summarize)在结果组(result group)中的数据。此结果组是按照上面表述的 group-by 来指定。在结果组中的值将按照指定的 function 来概要化。可用的 function 有：min | max | sum | avg | count | count-distinct | upper | lower 等。这些和 SQL 中通常使用的函数平行。

如果不需要这些，而希望事情更简单一些，那么推荐使用：只查询要么是按照 group-by 别名、要么是按照 function 别名的别名字段(aliased fields)。同样，并不是所有指定的别名字段都可以包含在要查询的字段列表中，因为这将过度限制查询。

分组和概要数据对某些定制的报表或数据迁移于清洗非常有用。

提示(Tips)

注意到视图实体很容易变的非常复杂。对熟悉 SQL 的人来说，他们能够查看生成的 SQL 代码以确保能达到预期目的。对每个人来说，都有很多可能有用的提示。

首先注意到视图链接(view-links)的排序，以及某些情况下的字段别名排序。在 view-links 中，除了第一个外，确保在 entity-alias 属性中引用的成员实体在前面的 view-link 中已经引用过(这就是为什么 link 树能够保持干净)。

视图实体元素引用(Reference)

view-entity

属性名	是否必需?	描述
entity-name	Y	用在实体引擎的 Java API 和其他地方时，所引用的实体名

package-name	Y	此实体所属包名字。在一个大的数据模型中有成百上千个实体，通过包来组织实体定义。
dependent-on	N	用来指定父实体或此实体倚赖的实体。现在已经不再使用，因为在实体引擎中都已自动实现，但是保留在这里，主要是用来指定层次化的实体结构
never-cache	N	指定了此实体是否需要使用优化锁(optimistic locking)。在此实体上必需存在lastUpdatedStamp字段，用来跟踪实体实例上一次被更新的时间。如果当前被更新的实例和lastUpdatedStamp不匹配，则将抛出EntityLockedException异常。必需是true或false。默认是false
title	N	实体的title。如果没有指定，默认是实体所在文件的全局设置
copyright	N	实体版权。如果没有指定，默认是实体所在文件的全局设置
author	N	实体作者。如果没有指定，默认是实体所在文件的全局设置
version	N	实体版本。如果没有指定，默认是实体所在文件的全局设置

子元素名	多少	描述
description	0 or 1	实体描述。如果没有指定，默认是实体所在文件的全局设置。此元素没有属性并且只应该包含一个简单字符串
member-entity	1 to many	用来声明视图实体的成员实体，并且用什么别名来引用它们
alias	1 to many	用来声明来自成员实体的别名字段(将是视图实体的一部分)
view-link	1 to many	在视图中，用来声明各个成员实体怎样被联接起来
relation	0 to many	用于声明实体之间的关系

member-entity

属性名	是否必需?	描述
entity-alias	Y	成员实体别名。在所有的成员实体当中必需是唯一的。在视图实体的定义中，实体别名用来引用此成员实体
entity-name	Y	成员实体对应的实体名字。同一实体在同一视图实体中可以以不同的别名使用多次

alias

属性名	是否必需?	描述
entity-alias	Y	此别名字段(aliased field)所对应成员实体的实体别名
name	Y	字段别名(field alias)的名字。和视图实体交互时使用，同样，字段名是用来和实体交互的。
field	N	对应于实体别名的成员实体的字段名字。如果没有指定，则别名默认是在名字属性中指定
prim-key	N	用来指定此别名是否是视图实体主键。如果指定了，则要么是true要么是false。如果没有指定，那么实体引擎将从此字段的来源查询此字段的定义，看它是否是原来实体的主键。
group-by	N	用来指定别名字段应该用于分组结果，主要和其他别名字段上的功能属性(function attribute)联合使用。完整讨论请参看正文。要么是true要么是false。默认是false.

function	N	指定此字段上的功能，用于计算概要信息(summary information)。和 group-by 属性联合起来指定怎么组织概要信息。更完整的讨论请看正文。

view-link

属性名	是否必需?	描述
entity-alias	Y	link 来源实体的别名
rel-entity-alias	Y	link 目的实体的别名
rel-optional	N	用来指定相关实体是否是可选的。如果是 true，它将影响到相关实体的外联合(outer join)。必需是 true 或 false，默认是 false。

子元素名	多少	描述
key-map	1 to many	Key-map 用来指定此实体中字段对应到相关实体中的哪个字段。此元素有两个属性：field-name 和 rel-field-name。这些属性用来指定此实体上的字段名和相关实体上的字段名。

relation

属性名	是否必需?	描述
type	Y	指定关系(relationship)类型，包含关系基数(单向)，以及是否要为基数是 one 的关系创建外键。必需是“one”、“one-nofk”或者“many”。
title	N	因为用户可能需要针对单个实体的多个 one 关系，此属性允许用户指定一个附加到 rel-entity-name 上的 title，用来构建关系名字。如果没有指定，则 rel-entity-name 独自用着关系名
rel-entity-name	Y	相关实体名字。关系是从当前实体到相关实体。
fk-name	N	根据关系名字，外键名字能够自动创建，但不推荐这样做，原因有二：对外键和索引名字，很多数据库有一个很小的最大长度限制(大约是 18 个字符)，并且很多数据库需要 FK 名字对整个数据库是唯一的，而不是仅仅针对此外键所在的表。

子元素名	多少	描述
key-map	1 to many	Key-map 用来指定此实体中的字段对应到相关实体中的哪个字段。此元素有两个属性：field-name 和 rel-field-name。这些属性用来指定此实体上的字段名和相关实体上的字段名。

实体引擎 API

在包 org.ofbiz.core.entity 中的实体引擎类定义了与实体数据交互的 API。从用户的观点来看，只需要真正理解三个类。它们是：GenericDelegator、GenericValue 和 GenericPK。GenericDelegator 类，通常和实例名字“delegator”一齐使用，它通常用于 GenericValue 对象的创建、查询、存储以及其他的操作。一旦创建了 GenericValue 对象，它将包含一个对创建它自己的 delegator 的引用，并且通过此引用，它就知道怎么存储、删除以及其他操作，而不需程序去调用 delegator 自身上的方法。

更进一步的了解请参考 org.ofbiz.core.entity 包的 JavaDoc，或浏览 eCommerce 和

其他的应用程序。

工厂方法

不应该试着去构造一个 GenericValue 或 GenericPK 对象自己，而应该通过 GenericDelegator 的 makeValue 和 makePK 方法。这些方法将创建一个还没被持久化的对象，因此允许用户在后面用它们自己的创建或保存方法、或调用 delegator 对象上的创建或保存方法，去增加、创建或存储此对象。

创建、保存和删除

为了向数据库中创建(或插入)值，使用 GenericValue 或 GenericDelegator 对象上的 create 方法。为了保存(或更新)现存的值，使用 GenericValue 或 GenericDelegator 对象上的 store 方法。

对多实体的保存，GenericDelegator 类有一个叫 storeAll 的方法。此 方法接受多个 GenericValue 实例，然后在同一个事务中保存它们。实际上，说此方法保存这些实例是不正确的。StoreAll 首先会查看它们是否存在，如果存在则更新之；如果不存在，则插入之。这一步在将来还可以优化速度，基于实体先前的信息，允许用户指定是否是插入还是更新。注意，storeAll 方法和 store 方法有不同的行为，store 方法仅仅是更新。

要么是通过 delegator 的 remove 方法、要么是通过 GenericValue 的 remove 方法来实现实体的删除。

查询

Value 实例能够通过 findByPrimaryKey 方法从数据库中提取出来，或者能够通过 findAll 或 findByAnd 方法来返回一个 collection。

FindByAnd 方法有两种主要类型。每种都有很多变种，它们可能包含 cache 的使用、也可能接受 orderBy 字段列表(orderBy field list)。这两种类型都接受不同的字段列表。一种接受字段的 Map，然后按照 anding 的方式来查询，其中每个命名字段必需等于 map 中对应的值。FindByAnd 的另一种类型则接受一个 EntityExpr 对象(用来指定一个很小的表达式，然后这些表达式被 and 起来)的列表。每个 EntityExpr 指定了一个字段名、一个操作、以及一个字段值。

EntityCondition 对象

最初的时候，EntityExpr 对象原打算是可嵌套的，这样就允许更灵活的查询，但决不是完备的。当然，即使能够嵌套它，用户能够运行的查询类型也将非常有限，因为用户不能在一组刮号里有两个 ANDs(例如)。为了解决这些问题，并能够完善 EntityExpr 的实现，引进了 EntityCondition 抽象对象，以及 EntityConditionList 和 EntityFieldMap 对象，其中后面两个对象扩展了 EntityCondition 对象。EntityExpr 对象也已经改为扩展 EntityCondition.

EntityConditionList 和 EntityFieldMap 对象都非常简单，它们分别被创建为 List 或 Map，并且，EntityOperator 用来指定把这些容器成员联合(join)起来的操作符(operator)，通常是 AND 或 OR。

EntityExpr 类现在有两个主要构造方法：一个是把一个字段和一个值(the String, EntityOperator, Object constructor)比较，而另一个是比较两个 EntityCondition 对象(the EntityCondition, EntityOperator, EntityCondition constructor).

FindByCondition 方法现在也是可用的，它接受一个 EntityCondition 参数(当然也有一些其他有用参数)并且此 EntityCondition 能够是一个 EntityExpr、EntityConditionList 或 EntityFieldMap 等。

在实体引擎中的代码稍微有点变动，因为这些 EntityCondition 对象现在都是自己创建它们的 WHERE 语句。这一点非常好，因为开发人员能够按照自己的意愿来定制化 EntityCondition.

已经完成了 EntityCondition 抽象类的便捷实现，用来把 SQL WHERE 语句片断或完整语句结合到 EntityCondition 的查询框架中去。这个类名叫 EntityConditionString，用一个代表 SQL(将最终插入到最终生成的 SQL 中)的 String 参数来构造之。如果有其他可选方式时，一般不推荐使用这里的方式，但对某些实体引擎不支持的功能性来说还是很有必要的。

EntityListIterator 对象

考虑到方便的缘故，EntityListIterator 类实现了 ListIterator interface 接口，但是也有其他的一些必需的方法，诸如完成时的 close()方法等。

通过维护一个到查询结果中的结果集(ResultSet)的引用，此对象允许用户很方便的从两个方向遍历查询结果。使得在数据库中能够使用光标的特征，特别是针对大型查询，使得内存的使用有效的多。此对象一次只构造很少(on the fly)的 GenericValue 对象而不是一下就构造太多，因此，如果用户需要导出大量查询结果到文件或其他某个地方时，它不需要大量的内存就可以实现。(译者注：这一段加红色标记的是意译，可能不太准确，请参考原文)

EntityListIterator 也有一些很有用的方法来得到查询结果的一个子集，从一个起始索引值开始，给定想要的结果数量，立即就能得到想要的而不用去创建一个循环来完成此工作。

实体引擎的 cache

因为从数据库中检索数据的效率代价非常高，并对应用程序(或应用程序的组件)的整个性能很严重的影响，因此对来自数据库的数据具备缓存能力非常重要。GenericDelegator 中很多查询方法都有对应的 cache 方法(函数名以后缀“Cache”结尾)。对按照主键查询回来的单条值，以及按照 and 或按照 all 方式查询回来的值列表，都很容易 cache 起来。

实体引擎 cache 机制利用了 OFBiz 的 UtilCache 类来实现实际的 cache。UtilCache 有很多特点，例如：有限 cache 大小，软引用(soft references)等，因此需要更多内存时，垃圾回收器能够从大的 cache 中回收内存。

配置 UtilCache 有两种方式。第一种是修改 cache.properties 文件(详悉描述在核心配置指南里)，这时一种永久性改变。第二种是通过 WebTools webapp 中的 Cache 管理页面来改变，不过这时暂时性改变。这些页面也能用来查看(view)统计情况、清理 cache 值、以及完成其他 cache 相关维护。

这种 cache 方式(它不直接绑定到数据库)有一个很大的问题是它自动清理“脏” cache。当通过实体引擎来执行创建、保存或删除操作等时，正常情况下它将能够自动清理任何要被更新的数据。分布式 cache 清理也被实现了，因此能够通过多种方式的配置以利用服务引擎的灵活性。

也有几种情况下 cache 的数据不能被自动清理。最主要的问题是视图实体，由于更新实体时，它只是视图实体的一部分，因此它仅有主键中的部分数据，或者仅有 and set(用于从 cache 中查询和索引数据)中的部分数据。当然也有方法能实现自动清理，但目前手动清理视图实体 cache 还是必要的。

JTA 支持

实体引擎 JTA 支持使用起来很简单，但是配置起来有点复杂。JTA 支持是通过一个 API 和一个 Factory 类来实现的，因此不需要知道 JTA 的具体实现。能通过 TransactionFactory 类得到 JTA 需要的两个主要对象：UserTransaction 和 TransactionManager。目前实现是简单的支持 Tyrex JTA/JTS 的实现。如果需要用别的实现，只不过是简单的改变 TransactionFactory 类而已，但这里的配置比较富有技巧性。对于 Tyrex，必需确保一个叫 tyrexdomain.xml 的 domain 配置文件在 classpath 下。

为了划分事务界线，用户能够使用 TransactionUtil 类。此类包装了 UserTransaction 类，然后仅仅抛出 GenericEntityExceptions 和 runtime 异常。需要的基本方法是 begin()、commit()、和 rollback()，但也包含了其他的方法，在有的时候可能非常有用。事务是附加(attached)在当前线程上，因此没必要把它传递过去传递过来的。在事务开始后，确保它要么是提交、要么是回滚，而没有其他的选择。一般是在 try 块结束时提交而在每个 catch 块中回滚。用户也能够使用标准的 UserTransaction 对象(通过 TransactionFactory 得到)。

核心 Web 工具

WebTools 的 web 应用中包含了大量有用的实体引擎工具。这些工具中包括了对实体定义之间的交叉链接引用(cross linked reference)、编辑实体和关系的工具、用着 XML 模板的 JSP(此 JSP 也负责把 XML 实体的定义保存到它们对应的文件)。

也有 JSP 被用着子例程(routines)的前端，这些子例程用来检查数据表定义(definitions)的当前状态并报告任何差异情况。例如，可能缺失的表或列能够加载到数据库中。也有子例程，用于服务器装载时运行，然后创建缺失的表和列。但此子例程和创建动作都是可选的。

实体代码生成器已从项目中删除了，或已过时了(如果用户需要用的话)，因为已经有了下一代生成实体工具：实体引擎。但还有某些情况下，需要使用模板来创建一些特殊的实体代码或其他文本，事实上这也是很有必要的。一个例子是为实体定义创建 XML 文件的 JSP，并且用于实体定义(definitions)变化后输出 XML。模板的其他用途包括手动生成特定数据库用来创建表的 SQL 语句，然后快速启动 JSPs 和事件处理器，以允许查询、查看和编辑特定实体数据。这些能够用于有特殊任务的应用程序的出发点。

如果实体数据编辑(entity data editing)不是特殊任务的而是特殊实体的，那么就可以使用 WebTools 中实体数据维护页面。它们是一些动态页面，而这些动态页面倚赖于内存中的实体定义(definitions)来创建数据输入的表单，倚赖于处理输入数据的事件(包括在实体定义中的 validators)，倚赖于把实体字段之间 and 起来而查询出的特定实体，或者查询给定实体实例的任何关系实体实例。例如，当查看 OrderHeader 实体时，也能够查看链接到此实体上的所有关系，包括编辑和查看它们的链接。这些相关实体将包含 OrderType(one 关系)、OrderLine (many 关系)和很多其他的关系。

在导入导出页面中，数据库中的数据能导出到实体引擎 XML 文件中，反之也能把 XML 文件中的数据导入到数据库中。导入数据将引起对应实体实例要么被创建要么被更新，倚赖于它们是否已存在。导出页面通过下拉列表框允许用户指定它们想导出的实体。