嵌入式 Java 持久性

Oracle Berkeley DB Java 版（以下简称 JE）是一个卓越的 Java 对象持久化方案。JE 是纯 Java 语言实现的嵌入式数据库。它充分利用了 Java 平台的强大功能，实现了与 Berkeley DB C 语言版（以下简称 Core）同样成功的底层机制，延续了 Berkeley DB 产品家族一贯的优良特性。

本文介绍了 JE 及其直接持久层（以下简称 DPL），并通过一个网上购物车例子演示了如何使用 DPL。在此过程中，您会发现 DPL 为 Java 对象的存储和检索提供了一套优秀易用的接口。

JE 简介

JE 不是一个传统意义上的关系数据库，而是用 B 树结构存储键-值对（key-data pair）的数据库。B 树是高效且经过良好验证的数据存取结构。

这种 B 树数据存储方式来源于 Core。Core 由 C 语言实现，诞生于上世纪 90 年代初，并迅速成为世界上使用最广泛的数据库之一。Core 是 Linux 和 Unix 每个版本都自带的数据库，且广泛地应用于各种应用程序中，从移动电话、路由器到主要的零售、搜索站点，都有成功使用 Core 的案例。

JE 的 B 树键-值存储方式并不局限于基本数据类型，如整型数或字符串。其值可以是任意的字节序列，例如表示一个 Java 类的字节序列。同样的，其键也可以是任意的用来唯一标识该类对象的字节序列。

JE 能够直接嵌入到 Java 应用程序中，并且与应用程序运行于同一个 Java 虚拟机。它没有上下文交换的开销，没有与远程数据库服务器交互的开销，也没有查询语法分析的开销（JE 不支持 SQL）。

直接持久层（DPL）

使用 JE 存储 Java 对象的关键是将 Java 对象持久化，即转换成二进制形式存储。当需要还原对象时，可以在读到的字节序列基础上重构该对象。恩，有点麻烦，不是吗？不要紧，DPL 能处理这些琐碎的工作。由于 JE 将对象以整体形式存储，而不是将其分开存储于不同的列和表，所以这里不存在任何的对象-关系映射（ORM）概念。

DPL 的使用非常简单，只需在定义持久化类时，指定少量的注释（annotations，JDK5.0 的新特性）就可以完成这些工作。如果不想使用注释，可以创建一个 EntityModel 类的子类以提供那些相应的元数据。本文不讨论该方法的细节，您可以从链接地址[1]获得更多相关信息。

实体类（Entity Classes）

JE 嵌入在网上购物车应用程序中使用，共同运行于应用程序服务器端。DPL 负责应用程序对象的存储和读取，您只需关注如何实现业务逻辑即可。

---

[1] http://www.oracle.com/technology/documentation/berkeley-db/je/index.html

```
@Entity
public class Product {
    @PrimaryKey(sequence="ID")
    private long id;
    @SecondaryKey(relate=ONE_TO_ONE)
    private String name;

    public Product(String name) {
        this.name = name;
    }

    /** A default constructor is needed by DPL for deserialization.*/
    private Product() {
    }
...
...
}
```

代码列表 1：创建一个实体类

现在从定义实体类开始构造示例程序。在 DPL 术语中，实体类是指其对象直接存储在 JE 数据库的类。列表 1 创建了一个名为 Product 的实体类，有四处代码是需要特别注意的。首先是出现在类定义之前的@Entity 注释的使用：

```
@Entity
public class Product {
```

Java 编译器会将@Entity 注释转换成该类的一个属性。DPL 会读到这一类属性，并利用其引导 DPL 执行底层的数据库操作。

接下来需要注意主键的定义：

```
@PrimaryKey(sequence="ID")
private long id;
```

DPL 使用@PrimaryKey 注释定义某个类字段是该类的主键。在示例代码中，该键由一个名为 ID 的序列（sequence）自动产生。当存储一个新的 Product 对象时，DPL 会将 ID 的当前值赋给 id 字段。DPL 也会在存储第一个 Product 对象时，自动产生 ID 序列。

您可以选择是否创建次键。示例代码在 Product 类的 name 字段上定义了一个次键：

```
@SecondaryKey(relate=ONE_TO_ONE)
private String name;
```

当您定义次键时，一定会联想到关系数据库外键关系的定义。JE 用次键创建一个辅助 B 树，根据主/次键之间的关系，辅助 B 树可以支持或不支持重复的记录。在这里，ONE_TO_ONE 关系的意思可以理解为：一个 Product 记录有且仅有一个 name 与之相对应。

最后需要注意的是默认构造函数：

```
private Product() { }
```

DPL 要求每个实体类必须有一个默认构造函数。当还原对象时，DPL 从数据库中读到该对象的字节流，反序列化（deserialize）这个字节流，并最终调用默认构造函数实例化该对象。

User 和 ShoppingCart 也是购物车应用程序用到的实体类。在 ShoppingCart 类中，您将可以看到 DPL 通过次键（@SecondaryKey）定义了两个实体类之间的联系：

```
@Entity
public class ShoppingCart {

    @PrimaryKey(sequence="ID")
    private long id;

    @SecondaryKey(relate=MANY_TO_ONE,
                  relatedEntity=User.class,
                  onRelatedEntityDelete=CASCADE)
    private String userName;
    ...
    ...
```

代码在 userName 字段上指定了一个包含三个参数的@SecondaryKey 注释。relatedEntity 参数定义了目标实体（即与该字段相关联的实体）。因此，可以将 MANY_TO_ONE 理解为“一个 User 对应多个 ShoppingCart”。

@SecondaryKey 的最后一个参数定义了在删除该对象时，与其关联的对象需要采取的相应操作。这里的 CASCADE 表示 DPL 会在删除某个用户的同时删除该用户的所有购物车。由于篇幅有限，在这里没有列出类定义的所有细节。如果需要完整的示例代码，请链接地址[2]下载。

持久类（Persistent Classes）

一个类的内部成员（类域或内部类）可能也是类。在 DPL 中，一个持久（persistent）类是可以被用在实体类中的类。实体类内部成员类需要注释为持久类。除此之外，一个实体类的子类和超类必须注释为@Persistent，并且 DPL 也支持继承和多态。

购物车应用程序使用了一个名为 LineItem 的持久类。列表 2 是它的定义（为节省空间，本文去掉了 getter 和 setter 方法）。持久化一个类非常简单，只需使用@Persistent 注释并定义一个默认构造函数。

---

[2] http://www.oracle.com/technology/oramag/oracle/07-mar/o27berk.zip

```
    @Persistent
    public class LineItem {
        private long productId;
        private int quantity;

        public LineItem(long productId, int quantity) {
            this.productId = productId;
            this.quantity = quantity;
        }

        /** A default constructor is needed by the DPL for deserialization.
*/
        private LineItem() {
        }
    ...
    ...
    }
```

代码列表 2：创建一个持久类

数据库实例（Database Environment）

在定义实体类和持久类之后，接下来需要创建一个数据库实例并打开一个实体库（entity store）。数据库实例是一个封装并定义实体库集合默认属性的抽象。在应用程序中，通过实体库存取对象。

```
    /* Open a transactional Oracle Berkeley DB Environment. */
    EnvironmentConfig envConfig = new EnvironmentConfig();
    envConfig.setAllowCreate(true);
    envConfig.setTransactional(true);
    env = new Environment(dataDirectory, envConfig);

    /* Open a transactional EntityStore. */
    StoreConfig storeConfig = new StoreConfig();
    storeConfig.setAllowCreate(true);
    storeConfig.setTransactional(true);
    store = new EntityStore(env, "MyStore", storeConfig);
```

代码列表 3：打开一个数据库实例和一个实体库

列表 3 中的代码执行下列步骤打开一个数据库实例，若该实例不存在，则创建一个新的数据库实例：

1. 实例化一个 `EnvironmentConfig` 对象；

2. 调用 EnvrionmentConfig 类的 setAllowCreate(boolean)函数，将其参数设置为 true，使得第一次打开数据库实例时，创建所有必需的底层文件；
3. 调用 EnvironmentConfig 类的 setTransactional(boolean)函数，将其参数设置为 true，这样数据库实例就能运行事务；
4. 将 EnvironmentConfig 对象传给 Environment 构造函数。如果该数据库实例存在，则打开这个数据库实例；如果数据库实例不存在，则创建一个新的数据库实例。

在物理层，数据库实例与磁盘上某个特定的文件夹相联系。当您运行这个示例的时候，请确保 Enironment 构造函数的第一个参数是一个合法存在的文件夹。DPL 不会自动创建该文件夹。

在打开（或新建）数据库实例之后，列表 3 用基本相同的流程打开和配置一个实体库。您可以任意命名实体库，实体库名是由实体库创造的数据库名的一部分。事实上，在同一实体模式下创建不同的实体库，就能构成独立的数据集。如果运行两个在线购物程序，就需要为每个程序创建独立的实体库。

存储对象

在打开数据库实例和实体库之后，让我们来存储一些对象吧。JE 是一个 B 树结构的数据库，对象被直接加入到基于其主键值的 B 树结构，即主键索引。因此，为了运行应用程序，需要创建一个主键索引（Primary Index）。下列代码创建了 User 和 Product 类的主键索引：

```
private PrimaryIndex<String, User> userByName;
private PrimaryIndex<Long, Product> productById;

userByName = store.getPrimaryIndex(String.class, User.class);
productById = store.getPrimaryIndex(Long.class, Product.class);
```

您可以通过 userByName，productById 主键索引存储 User 和 Product 类的对象。您可以选择使用事务保证操作的原子性：

```
Transaction txn = env.beginTransaction(null, null);
```

然后创建和存储一个 User 对象：

```
User user = new User("Mr. Shooper");
userByName.put(txn, user);
```

接着创建和存储一个 Product 实例：

```
Product camera = new Product("Ultrazoom Camera");
productById.put(txn, camera);
```

最后提交事务：

```
txn.commit();
```

您可以根据应用程序的要求决定使用或不使用事务。如果没有显示的声明一个事务，JE 将会自动为每个存储、查询和删除操作创建独立的事务。当显示声明一个事务时，由 beginTransaction(Transaction, TransactionConfig)函数的第二个参数配置该事务（例如，事务隔离的级别）。

查询，更新和删除

通过主键查询对象非常简单。例如，下列代码查询一个名为“Mr.Shopper”的 User：

```
User user = userByName.get(txn, "Mr.Shopper", null);
```

修改并存储一个对象同样简单：

```
user.setPassword("secret");
userByName.put(txn, user);
```

通过主键删除对象只需调用 PrimaryIndex 的删除方法：

```
userByName.delete(txn, "Mr.Shopper");
```

在示例中，删除一个用户会级联删除其拥有的所有购物车。这是次键定义的购物车和用户之间的联系："onRelatedEntityDelete=CASCADE"规则作用的结果。

次键查询

在某些情况下，您可能不知道被检索对象的主键。Product 对象就是一个很好的例子。您使用 name 进行查找的几率要比使用自动生成的 ID 号（当然，对用户来说，ID 号也是不可见的）的大得多。幸运的是，Product 类定义了一个在 name 字段上的次键。

我们需要通过访问类的主键索引来访问该类的次键索引。下面是进行次键查询所需遵守的规范：

1. 得到主键索引；
2. 由主键索引得到次键索引；
3. 次键索引通过检索次键得到对象。

代码列表 4 通过检索 name 字段为“Old Cassette”的 Product 对象，详细描述了上述过程。

```
public PrimaryIndex<Long, Product> productById;
public SecondaryIndex<String, Long, Product> productByName;

productById = store.getPrimaryIndex(Long.class, Product.class);
productByName =
     store.getSecondaryIndex(productById, String.class, "name");
Product product = productByName.get("Old Cassette Tape");
```

代码列表 4：通过次键读取数据

基于游标的检索

JE 同样支持基于游标的检索。使用游标能够遍历一个对象集合。例如，如果需要打印数据库中的所有 Product 记录并以 ID 字段排序，可以通过主键索引创建一个游标：

```
EntityCursor<Product> products = productById.entities();
```

如果以 name 排序，可以通过 name 字段上的次键索引创建一个游标：

```
EntityCursor<Product> products = productByName.entities();
```

您还可以游标查询指定范围内的数据，比如说 name 以字母 O 到 P 开头但不包括 P 的记录，entities 方法中的 true 和 false 参数表明相应的边界参数是否被包括在内：

```
EntityCursor<Product> products = productByName.entities("O", true, "P", false);
```

现在，您仅需一个简单的 for 循环，就可以遍历游标包含的结果集：

```
try {
    for (Product product : products) {
        System.out.println(product);
    }
} finally {
    products.close();
}
```

finally 代码块能够保障最终关闭游标，从而防止潜在的内存泄漏。同样的，确保在不再使用时关闭实体库，并最终提交或回退事务，也是防止内存泄露的重要手段。如果下载本文的完整例子，您会看到示例代码在很多地方使用了 try/finally 代码块以保障资源被恰当的关闭和释放。

使用 DPL 的好处

JE DPL 是一个优秀易用且功能强大的 Java 对象持久化解决方案。在符合下列情形时，请考虑使用 JE 和 DPL：

1. 需要以键-值对形式存储和检索 Java 对象；
2. 需要一个嵌入式存储解决方案；
3. 良好定义的数据库操作；
4. 应用程序对低延迟和高可靠性有较高要求。

如果您是对 JE 和 DPL 感兴趣的 Java 开发人员，请从链接[3]下载并运行完整的示例程序，从链接[4]下载并阅读"Getting Started Guides"，尝试着在项目中使用 DPL，我相信您会喜欢上 JE 和 DPL 的。

感谢 Gregory Burd 和 Mark Hayes 对本文的审核，特别感谢 Mark Hayes 提供了本文示例的所有代码。

[3] http://www.oracle.com/technology/oramag/oracle/07-mar/o27berk.zip
[4] http://www.oracle.com/technology/documentation/berkeley-db/je/index.html