Berkeley DB Java 版
直接持久层基础

Oracle 白皮书
2006 年 9 月

Berkeley DB Java 版
直接持久层基础

概览

Berkeley DB Java 版是一套纯 Java 语言实现的嵌入式数据库。它提供的事务存储引擎不仅显著的减少了对象持久化开销，而且保持了对象-关系映射（ORM）解决方案的灵活性，速度和扩展性。Berkeley DB Java 版 3.0 引入了直接持久层（DPL），旨在提供和与企业 Java Bean 3.0 (EJB3) 持久性相同好处：对象持久化时，不需要将对象转换成关系表。

关系数据库是开发人员可用来进行数据存储和分析的最复杂的工具之一。应用程序大多将持久化的数据还原成 Java 对象使用，并不需要 SQL 来做分析。对于这种基本的对象还原任务来说，一个复杂的分析引擎是不必要的，即关系模型的分析功能对于有效地进行 Java 对象持久化操作不是必须的。因而在大多数情况下，这是不必要的开销。相比之下，Berkeley DB Java 版不提供类似于 SQL 的查询语言，所以不会有这层开销，其好处是更快的存储，较低的 CPU 和内存的要求，和更有效的开发过程。即使没有专门的查询语言，Berkeley DB Java 版依然可以专用方式访问 Java 对象，并提供和任何其他数据库相同的快速，可靠和可扩展的数据存储。所不同的是它是实现在一个小型，高效和易于管理的包内。

Java 开发者现在可以使用直接持久层来高效地持久化以及还原相互关联的 Java 对象。对比于相应的 ORM 解决方案，直接持久层的复杂性和开销只相当于其一部分。

这份白皮书将通过一个简单的例子程序介绍这项新的 API 及其主要概念。

导言

直接持久层是设计来满足下列要求的。

提供类型安全和方便的API访问持久化对象。您可以选择使用Java的通用类型（generic types）来提供类型安全。例如：

```
PrimaryIndex<Long,Employer> employerById = ...;
long employerId = ...;
Employer employer = employerById.get(employerId);
```

所有Java类型都是可持久化的，且不需要实现特别的接口。持久化字段的访问类型可以是private，package-private（缺省访问类型），protected，或public。您不需要手工编码将实例字段和持久化数据进行绑定，只需在每个持久化类中提供一个缺省的构造函数。例如：

```
@Persistent
class Address {
    String street;
    String city;
    String state;
    int zipCode;
    private Address() {}
}
```

容易定义主键和次键。不需要使用外部模式而用Java的注释（annotation）即可定义所有的元数据。继承从其他来源获的元数据可用来作元数据扩展。例如，下面的Employer类定义了一个持久化实体，一个主键字段id和一个次键字段name：

```
@Entity
class Employer {

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

    @SecondaryKey(relate=ONE_TO_ONE)
    String name;

    Address address;

    private Employer() {}
}
```

通过 Java 集合框架与外部组件互操作。任何主键或次键索引可以使用标准 java.util 集合来访问。例如：

```
java.util.SortedMap<String,Employer> map =
                    employerByName.sortedMap();
```

明确支持类的演变。兼容的变化（加入字段和类型扩大）会自动执行且是透明的。举例来说，向Address类中添加street2字段、zipCode从int改为long等操作均不需要作任何特殊的配置：

```
@Persistent
class Address {
    String street;
    String street2;
    String city;
    String state;
    long zipCode;
    private Address() {}
}
```

许多不兼容的改变，如重命名字段或优化单个的类（如：使用通用类型，模块复用等改变），可以用 Mutations。Mutations 操作是延迟的：只在存取数据时自动改变，故避免了软件升级时大型数据库转换导致的长时间停机。

复杂的类优化可能涉及到多个类，使用 ConversionStore 进行。因而，无论持久化类作出何种改变，直接持久层都始终提供可靠数据存取。

Berkeley DB事务引擎提供了性能调优选项：操作直接映射到了引擎内部的API，轻量的对象绑定，而且所有的调整参数均可用。例如， “脏读”可通过LockMode参数实现：

```
Employer employer = employerByName.get(null, "Gizmo Inc",
                                       LockMode.READ_UNCOMMITTED);
```

对于高性能应用，DatabaseConfig参数可以用来调整Berkeley DB引擎的性能。举例来说，可

指定内部B树节点的大小来调整性能，通过如下方式来指定：

```
DatabaseConfig config = store.getPrimaryConfig(Employer.class);
config.setNodeMaxEntries(64);
store.setPrimaryConfig(config);
```

实体模型

普通的 Java 类通过直接持久层可以方便地和持久化数据关联。如果将 Java 类的某个字段定义为主键（primary key），并通过基于该主键字段上的索引（primary index）进行储存和读取操作，该 Java 类称为实体类。实体类只能有一个主键和任意数量的次键（secondary key），直接持久层会自动为每个次键字段建立索引，实体数据也可以按次键索引（secondary index）储存和读取。

实体类可以用 Entity 注释定义。对于任意的实体类，其唯一的主键可使用 PrimaryKey 注释来定义；使用 SecondaryKey 注释可以定义任意多的次键。

在下面的例子中，Person.ssn（社会安全号码）字段是主键，Person.employerIds 字段是一个多对多映射的次键。

```
@Entity
class Person {

    @PrimaryKey
    String ssn;

    String name;
    Address address;

    @SecondaryKey(relate=MANY_TO_MANY,
                  relatedEntity=Employer.class)
    Set<Long> employerIds = new HashSet<Long>();

    private Person() {} // For bindings
}
```

一套实体集合便构成了实体模型。此外，除了孤立的实体，实体模型中其余实体之间还有对应关系。关系可以使用 SecondaryKey 注释定义。支持的关系有：多对一，一对多，多对多，一对一以及外键约束。

在上面的例子中，Person 通过 Person.employerIds 字段建立和 Employer 的关系。那个 relatedEntity = employer.class 注释确立了外键约束，以保证每一个 employerIds 都是一个 Employer 的主键。

更多实体模型信息，请参阅 AnnotationModel 注释和 Annotation 注释。

实体库（EntityStore）是直接持久层的根对象。一个实体库管理着在实体模型中定义的任意数目的实体类。对于每个实体类，实体库为每个实体对象提供了到主/次键索引的访问。例如：

```
EntityStore store = new EntityStore(...);

PrimaryIndex<String,Person> personBySsn =
    store.getPrimaryIndex(String.class, Person.class);
```

一个简短的例子

下面的例子演示了如何定义一个实体模型，以及如何存储和访问持久化对象（异常处理在此不做赘述）。

```
import java.io.File;
import java.util.HashSet;
import java.util.Set;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;
import com.sleepycat.persist.EntityCursor;
import com.sleepycat.persist.EntityIndex;
import com.sleepycat.persist.EntityStore;
import com.sleepycat.persist.PrimaryIndex;
import com.sleepycat.persist.SecondaryIndex;
import com.sleepycat.persist.StoreConfig;
import com.sleepycat.persist.model.Entity;
import com.sleepycat.persist.model.Persistent;
import com.sleepycat.persist.model.PrimaryKey;
import com.sleepycat.persist.model.SecondaryKey;
import static com.sleepycat.persist.model.DeleteAction.NULLIFY;
import static com.sleepycat.persist.model.Relationship.ONE_TO_ONE;
import static com.sleepycat.persist.model.Relationship.ONE_TO_MANY;
import static com.sleepycat.persist.model.Relationship.MANY_TO_ONE;
import static com.sleepycat.persist.model.Relationship.MANY_TO_MANY;

// An entity class.
//
@Entity
class Person {

    @PrimaryKey
    String ssn;

    String name;
    Address address;

    @SecondaryKey(relate=MANY_TO_ONE, relatedEntity=Person.class)
    String parentSsn;

    @SecondaryKey(relate=ONE_TO_MANY)
    Set<String> emailAddresses = new HashSet<String>();

    @SecondaryKey(relate=MANY_TO_MANY, relatedEntity=Employer.class,
                                       onRelatedEntityDelete=NULLIFY)
    Set<Long> employerIds = new HashSet<Long>();

    Person(String name, String ssn, String parentSsn) {
        this.name = name;
        this.ssn = ssn;
        this.parentSsn = parentSsn;
    }
```

```
    private Person() {} // For bindings
}

// Another entity class.
//
@Entity
class Employer {

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

    @SecondaryKey(relate=ONE_TO_ONE)
    String name;

    Address address;

    Employer(String name) {
        this.name = name;
    }

    private Employer() {} // For bindings
}

// A persistent class used in other classes.
//
@Persistent
class Address {
    String street;
    String city;
    String state;
    int zipCode;
    private Address() {} // For bindings
}

// The data accessor class for the entity model.
//
class PersonAccessor {

    // Person accessors
    //
    PrimaryIndex<String,Person> personBySsn;
    SecondaryIndex<String,String,Person> personByParentSsn;
    SecondaryIndex<String,String,Person> personByEmailAddresses;
    SecondaryIndex<Long,String,Person> personByEmployerIds;

    // Employer accessors
    //
    PrimaryIndex<Long,Employer> employerById;
    SecondaryIndex<String,Long,Employer> employerByName;

    // Opens all primary and secondary indices.
    //
    public PersonAccessor(EntityStore store)
        throws DatabaseException {

        personBySsn = store.getPrimaryIndex(
            String.class, Person.class);
```

```
        personByParentSsn = store.getSecondaryIndex(
            personBySsn, String.class, "parentSsn");

        personByEmailAddresses = store.getSecondaryIndex(
            personBySsn, String.class, "emailAddresses");

        personByEmployerIds = store.getSecondaryIndex(
            personBySsn, Long.class, "employerIds");

        employerById = store.getPrimaryIndex(
            Long.class, Employer.class);

        employerByName = store.getSecondaryIndex(
            employerById, String.class, "name");
    }
}

// Open a transactional Berkeley DB engine environment.
//
EnvironmentConfig envConfig = new EnvironmentConfig();
envConfig.setAllowCreate(true);
envConfig.setTransactional(true);
Environment env = new Environment(new File("/my/data"), envConfig);

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

// Initialize the data access object.
//
PersonAccessor dao = new PersonAccessor(store);

// Add a parent and two children using the Person primary index.
// Specifying a non-null parentSsn adds the child Person to the
// sub-index of children for that parent key.
//
dao.personBySsn.put(new Person("Bob Smith", "111-11-1111", null));
dao.personBySsn.put(new Person("Mary Smith", "333-33-3333", "111-11-
1111"));
dao.personBySsn.put(new Person("Jack Smith", "222-22-2222", "111-11-
1111"));

// Print the children of a parent using a sub-index and a cursor.
//
EntityCursor<Person> children =
    dao.personByParentSsn.subIndex("111-11-1111").entities();
try {
    for (Person child : children) {
        System.out.println(child.ssn + ' ' + child.name);
    }
} finally {
    children.close();
}
```

```
// Get Bob by primary key using the primary index.
//
Person bob = dao.personBySsn.get("111-11-1111");
assert bob != null;

// Create two employers. Their primary keys are assigned from
// a sequence.
//
Employer gizmoInc = new Employer("Gizmo Inc");
Employer gadgetInc = new Employer("Gadget Inc");
dao.employerById.put(gizmoInc);
dao.employerById.put(gadgetInc);

// Bob has two jobs and two email addresses.
//
bob.employerIds.add(gizmoInc.id);
bob.employerIds.add(gadgetInc.id);
bob.emailAddresses.add("bob@bob.com");
bob.emailAddresses.add("bob@gmail.com");

// Update Bob's record.
//
dao.personBySsn.put(bob);

// Bob can now be found by both email addresses.
//
bob = dao.personByEmailAddresses.get("bob@bob.com");
assert bob != null;
bob = dao.personByEmailAddresses.get("bob@gmail.com");
assert bob != null;

// Bob can also be found as an employee of both employers.
//
EntityIndex<String,Person> employees;
employees = dao.personByEmployerIds.subIndex(gizmoInc.id);
assert employees.contains("111-11-1111");
employees = dao.personByEmployerIds.subIndex(gadgetInc.id);
assert employees.contains("111-11-1111");

// When an employer is deleted, the onRelatedEntityDelete=NULLIFY
// for the employerIds key causes the deleted ID to be removed from
// Bob's employerIds.
//
dao.employerById.delete(gizmoInc.id);
bob = dao.personBySsn.get("111-11-1111");
assert !bob.employerIds.contains(gizmoInc.id);

store.close();
env.close();
```

前面的例子阐述直接持久层的如下几个特征：

持久化数据具体来说就是实例的字段。为简洁起见，该例子并未列出getter和setter方法，虽然这些方法通常会在类封装时存在。直接持久层通过对象序列化（serialization）及反序列化

（deserialization）来访问字段而不是通过调用getter/setter方法，从而您的业务方法可以自由的，执行任意的验证规则。例如：

```
@Persistent
public class ConstrainedValue {

    private int min;
    private int max;
    private int value;

    private ConstrainedValue() {} // For bindings

    public ConstrainedValue(int min, int max) {
        this.min = min;
        this.max = max;
        value = min;
    }

    public setValue(int value) {
        if (value < min || value > max) {
            throw new IllegalArgumentException("out of range");
        }
        this.value = value;
    }
}
```

上述 setValue 方法如果是在对象反序列化时调用是行不通的，因为字段设置的秩序是任意的。该 min 和 max 有可能不是在 value 值被设置前而设置。

该例子创建了一个事务实体库，因此，所有操作都有事务性保障。因为没有明确的事务声明，所有事务操作均隐含是自动提交（auto-commit）。

一个事务也可能包含了多个操作，并提供可选的事务调整参数。例如，以下事务包含了两个操作：

```
Transaction txn = env.beginTransaction(null, null);
dao.employerById.put(txn, gizmoInc);
dao.employerById.put(txn, gadgetInc);
txn.commit();
```

为了提供最好的性能，直接持久层的操作直接映射为 Berkeley DB 引擎的对于 B 树的操作。与其它持久化的办法不同的是，Berkeley DB 的键及索引是可直接访问的且性能是可调整的。

查询操作可通过调用主/次键索引的相关方法来实现。并且，通过 EntityJoin 类还可进行等值连接（equality join）操作。举例来说，以下代码查询所有 Bob 的孩子中为 gizmo 公司工作的员工：

```
EntityJoin<String,Person> join = new EntityJoin(dao.personBySsn);

join.addCondition(dao.personByParentSsn, "111-11-1111");
join.addCondition(dao.personByEmployerIds, gizmoInc.id);

ForwardCursor<Person> results = join.entities();
try {
```

```
    for (Person person : results) {
        System.out.println(person.ssn + ' ' + person.name);
    }
} finally {
    results.close();
}
```

对象的关系是基于键的。当一个Person与某一特定雇主编号（ID）通过employerIds联系上，这个人便成为所有为那个雇主工作的所有雇员的一员了。对这些雇员的集合的访问可以使用SecondaryIndex.subindex与雇主编号来得到，如下所示：

```
EntityCursor<Person> employees =
    dao.personByEmployerIds.subIndex(gizmoInc.id).entities();
try {
    for (Person employee : employees) {
        System.out.println(employee.ssn + ' ' + employee.name);
    }
} finally {
    employees.close();
}
```

请注意，当 Bob 的雇主被删除时，该 Person 对象会被重新获取，对 employerIds 的改变也可以看到。这是因为对象是通过值来访问的，而非引用。换言之，直接持久层没有对象的高速缓存或"持久化上下文"。而正是嵌入式 Berkeley DB 引擎的底层的缓存机制加上轻量的对象绑定，提供了最高的性能。

使用哪个 API？

Berkeley DB Java 版引擎有一个基础 API，一个集合 API 和一个直接持久层。如果您不能确定使用哪个的话，可遵守以下指引：

- 当 Java 类是用来代表应用中的域对象（domain objects），也就是说，该模式是相对稳定的，建议用直接持久层。
- 当在 Berkeley DB 和 Berkeley DB Java 版之间移植应用程序时，或当实现自己的动态模式（举例来说，一个 LDAP 服务器），那么建议用基础 API。您也可能喜欢使用这个基础 API 如果您有极少数域类（domain class）。
- 集合 API 有利于和外部组件交互，因为它遵从 Java 集合框架标准。继而，和基础 API 以及直接持久层结合后会很有用。您可能会喜欢这个 API，因为它提供了熟悉的 Java 集合接口。

性能

Berkeley DB Java 版可以非常好地执行在 Java 环境中。其底层架构是不同于其他数据库的，它的设计就是要尽量减少磁盘访问次数。而当它访问磁盘时，它尽量做到最小的磁头移动（寻道），从而提高访问速度。

相比于 EJB 和其他 ORM 式的解决办法，Berkeley DB Java 版及其直接持久层是一个更有效率存储组合。它只要求有一个数据高速缓存而 EJB 要求至少两个：一个由关系数据库管理，一个由 EJB 层管理。对与相同的操作而言，后者将导致效率降低和需要更多的内存。

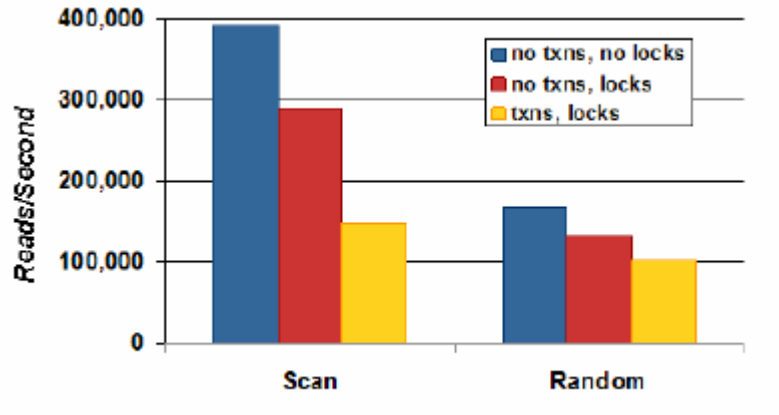
当使用 ORM 解决方案，即使你的关系数据库和应用运行于同一个 JVM 内，并通过 JDBC Type 4 驱动程序（纯 java 的驱动直接连到数据库）互联，但三层架构之间的额外开销仍然是有的。首先

，动态生成 SQL（ORM 层）时需要读取所需的实例数据；其次，在 JDBC 层处理也存在额外开销。SQL 在抵达的关系数据库后，它必须经过分析，生成执行计划并执行。关系数据（如行和列）再途经 JDBC 返回，然后由 ORM 还原成各种对象。而 Berkeley DB Java 版则避免了这些中间步骤，它直接将 java 对象储存进它的数据库！ 从而从对象持久化和还原的角度来讲，两者是同等的，只是使用 Berkeley DB Java 版的好处是显而易见的：相同的操作下，它使用更少的内存和需要更短的执行时间。

测试结果

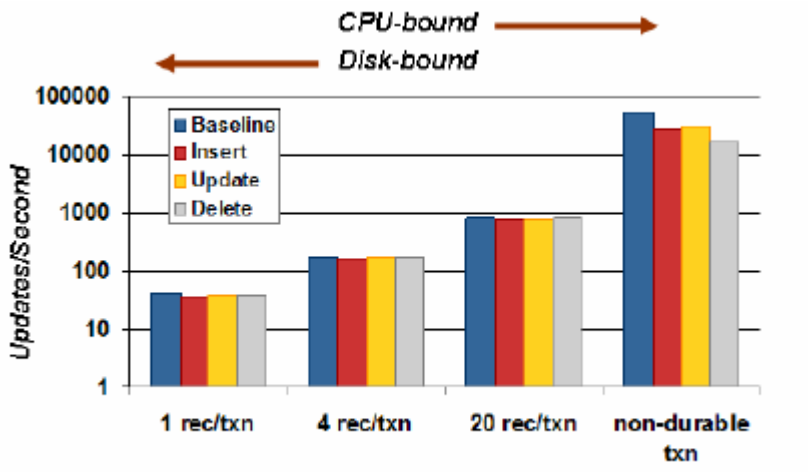
以下测试均是在一台有双 AMD Opteron 1.8 GHz 处理器，配备 2GB 533 MHz RAM，Solaris 10 操作系统和使用的 J2SE 1.5.05 的 JVM 的平台上做出的。结果演示了 Berkeley DB Java 版读取和写入操作的速度。

下图（图一）是以顺序和随机方式读取数据时的系统吞吐量，单位：操作数/秒。



图一

下图（图二）为写操作的吞吐量，单位：操作数/秒。结果表明，当“记录数/事务”提高时，决定更新操作的吞吐量的因素由磁盘限制（disk bound）转变为 CPU 限制（CPU bound）。



图二

结论

正如你从例子可以看到，Berkeley DB Java 版直接持久层提供了 Java 对象持久化的完整的解决方案。这是一个集 ORM 和事务存储为一体的解决方案，可用于一些不需要 SQL 的，基于 ORM 的 EJB 式的解决方案中。

如需有关 Berkeley DB Java 版和完整的产品包括源代码，文档，示例代码和测试代码可从以下地址下载：
http://www.oracle.com/technology/products/berkeley-db/je/index.html。

Berkeley DB Java 版直接持久层基础
2006 年 9 月

甲骨文公司
全球总部
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

全球咨询
电话: +1.650.506.7000
传真: +1.650.506.7200
oracle.com