**一、**             **简介**

       Berkeley DB Java Edition (JE)是一个完全用JAVA写的，它适合于管理海量的，简单的数据。

l         能够高效率的处理1到1百万条记录，制约JE数据库的往往是硬件系统,而不是JE本身。

l         多线程支持，JE使用超时的方式来处理线程间的死琐问题。

l         Database都采用简单的key/value对应的形式。

l         事务支持。

l         允许创建二级库。这样我们就可以方便的使用一级key,二级key来访问我们的数据。

l         支持RAM缓冲，这样就能减少频繁的IO操作。

l         支持日志。

l         数据备份和恢复。

l         游标支持。

**二、**             **获取JE**

JE下载地址：

<http://www.oracle.com/technology/software/products/berkeley-db/je/index.html>

解开包后 把*JE\_HOME*/lib/je-<version>.jar 中的jar文件添加到你的环境变量中就可以使用je了。

相关帮助文档可以参考 *JE\_HOME*/docs/index.html

源代码见*JE\_HOME*/src/\*.\*

**三、**             **JE常见的异常**

DatabaseNotFoundException 当没有找到指定的数据库的时候会返回这个异常

DeadlockException 线程间死锁异常

RunRecoveryException 回收异常，当发生此异常的时候，你必须得重新打开环境变量。

**四、**             **关于日志文件必须了解的六项**

JE的日志文件跟其他的数据库的日志文件不太一样，跟C版的DBD也是有区别的

l         JE的日志文件只能APPEND，第一个日志文件名是 00000000.jdb，当他增长到一定大小的时候(默认是10M)，开始写第二个日志文件00000001.jdb，已此类推。

l         跟C版本有所不同，JE的数据日志和事务日志是放在一起的，而不是分开放的。

l         JE cleaner负责清扫没用到的磁盘空间，删除后，或者更新后新的记录会追加进来，而原有的记录空间就不在使用了，cleaner负责清理不用的空间。

l         清理并不是立即进行的，当你关闭你的数据库环境后，通过调用一个cleaner方法来清理。

l         清理也不是只动执行的，需要你自己手动调用cleaner 方法来定时清理的。

l         日志文件的删除仅发生在检查点之后。cleaner准备出哪些log 文件需要被删除，当检查点过后，删掉一些不在被使用的文件。每写20M的日志文件就执行一次检查点，默认下。

**五、**             **创建数据库环境**

JE要求在任何DATABASE操作前，要先打开数据库环境，就像我们要使用数据库的话必须得先建立连接一样。你可以通过数据库环境来创建和打开database，或者更改database名称和删除database.

可以通过Environments对象来打开环境，打开环境的时候设置的目录必须是已经存在的目录，否则会出错误。默认情况下，如果指定的database不存在则不会自动创建一个新的detabase,但可以通过设置setAllowCreate来改变这一情况。

**1.**         **打开database环境**

示例：

package je.gettingStarted;

import com.sleepycat.je.DatabaseException;

import com.sleepycat.je.Environment;

import com.sleepycat.je.EnvironmentConfig;

import java.io.File;

...

Environment myDbEnvironment = null;

try {

    EnvironmentConfig envConfig = new EnvironmentConfig();

    envConfig.setAllowCreate(true);//如果不存在则创建一个

    myDbEnvironment = new Environment(new File("/export/dbEnv"), envConfig);

} catch (DatabaseException dbe) {

    // 错误处理

}

**2.**         **关闭database环境**

可以通过Environment.close()这个方法来关闭database环境，当你完成数据库操作后一定要关闭数据库环境。

示例：

import com.sleepycat.je.DatabaseException;

import com.sleepycat.je.Environment;

...

try {

    if (myDbEnvironment != null) {

        myDbEnvironment.close();

    }

} catch (DatabaseException dbe) {

    // Exception handling goes here

}

**3.**         **清理日志**

通常在关闭数据库连接的时候，有必要清理下日志，用以释放更多的磁盘空间。我们可以在Environment.close前执行下Environment.cleanLog()来达到此目的。

示例：

import com.sleepycat.je.DatabaseException;

import com.sleepycat.je.Environment;

...

try {

    if (myDbEnvironment != null) {

        myDbEnvironment.cleanLog(); // 在关闭环境前清理下日志

        myDbEnvironment.close();

    }

} catch (DatabaseException dbe) {

    // Exception handling goes here

}

**4.**         **Database环境的配置**

可以通过EnvironmentConfig这个对象来配置database环境。如果想得到当前环境的配置信息则可以通过Environment.getConfig()方法得到当前环境的配置信息。

也可以使用EnvironmentMutableConfig来配置环境，其实 EnvironmentConfig是EnvironmentMutableConfig的子类，所以EnvironmentMutableConfig能够使用的设置，EnvironmentConfig也同样能够使用。

如果你要获取当前环境的使用情况，那么你可以通过使用EnvironmentStats.getNCacheMiss().来监视RAM cache命中率。EnvironmentStats可以由Environment.getStats()方法获取。

**EnvironmentConfig常见方法介绍**

l         EnvironmentConfig.setAllowCreate() ;

如果设置了true则表示当数据库环境不存在时候重新创建一个数据库环境，默认为false.

l         EnvironmentConfig.setReadOnly()

以只读方式打开，默认为false.

l         EnvironmentConfig.setTransactional()

事务支持,如果为true，则表示当前环境支持事务处理，默认为false，不支持事务处理。

**EnvironmentMutableConfig的介绍**

l         setCachePercent()

设置当前环境能够使用的RAM占整个JVM内存的百分比。

l         setCacheSize()

设置当前环境能够使用的最大RAM。单位BYTE

l         setTxnNoSync()

当提交事务的时候是否把缓存中的内容同步到磁盘中去。

true 表示不同步，也就是说不写磁盘

l         setTxnWriteNoSync()

当提交事务的时候，是否把缓冲的log写到磁盘上

true 表示不同步，也就是说不写磁盘

                            示例一：

package je.gettingStarted;

import com.sleepycat.je.DatabaseException;

import com.sleepycat.je.Environment;

import com.sleepycat.je.EnvironmentConfig;

import java.io.File;

...

Environment myDatabaseEnvironment = null;

try {

    EnvironmentConfig envConfig = new EnvironmentConfig();

    //当环境不存在的时候自动创建环境

       envConfig.setAllowCreate(true);

       //设置支持事务

       envConfig.setTransactional(true);

    myDatabaseEnvironment =

        new Environment(new File("/export/dbEnv"), envConfig);

} catch (DatabaseException dbe) {

   System.err.println(dbe.toString());

   System.exit(1);

}

                  示例二：

package je.gettingStarted;

import com.sleepycat.je.DatabaseException;

import com.sleepycat.je.Environment;

import com.sleepycat.je.EnvironmentMutableConfig;

import java.io.File;

...

try {

    Environment myEnv = new Environment(new File("/export/dbEnv"), null);

    EnvironmentMutableConfig envMutableConfig =

        new EnvironmentMutableConfig();

    envMutableConfig.setTxnNoSync(true);

    myEnv.setMutableConfig(envMutableConfig);

} catch (DatabaseException dbe) {

    // Exception handling goes here

}

                            示例三:

import com.sleepycat.je.Environment;

...

//没有命中的CACHE

long cacheMisses = myEnv.getStats(null).getNCacheMiss();

...

5.         **Database操作**

在BDB中，数据是以key/value方式成队出现的。

**打开database**

       可以通过environment.openDatabase()方法打开一个database,在调用这个方法的时候必须指定database的名称。和databaseConfig() (注：数据库设置)

       示例：

package je.gettingStarted;

import com.sleepycat.je.Database;

import com.sleepycat.je.DatabaseConfig;

import com.sleepycat.je.DatabaseException;

import com.sleepycat.je.Environment;

import com.sleepycat.je.EnvironmentConfig;

import java.io.File;

...

Environment myDbEnvironment = null;

Database myDatabase = null;

...

try {

    // 打开一个环境，如果不存在则创建一个

    EnvironmentConfig envConfig = new EnvironmentConfig();

    envConfig.setAllowCreate(true);

    myDbEnvironment = new Environment(new File("/export/dbEnv"), envConfig);

    // 打开一个数据库，如果数据库不存在则创建一个

**DatabaseConfig dbConfig = new DatabaseConfig();**

**dbConfig.setAllowCreate(true);**

**myDatabase = myDbEnvironment.openDatabase(null,**

**"sampleDatabase", dbConfig);** //打开一个数据库，数据库名为

                                   //sampleDatabase,数据库的配置为dbConfig

} catch (DatabaseException dbe) {

    // 错误处理

}

**关闭database**

       通过调用Database.close()方法来关闭数据库，但要注意，在关闭数据库前必须得先把游标先关闭。

       使用示例：

import com.sleepycat.je.DatabaseException;

import com.sleepycat.je.Database;

import com.sleepycat.je.Environment;

...

try {

        if (myDatabase != null) {

            myDatabase.close();

        }

        if (myDbEnvironment != null) {

            myDbEnvironment.close();

        }

} catch (DatabaseException dbe) {

    // 错误处理

}

**设置数据库属性**

       其实设置数据库属性跟设置环境属性差不多，JE中通过DatabaseConfig对象来设置数据库属性。你能够设置的数据库属性如下。

l         DatabaseConfig.setAllowCreate()

如果是true的话，则当不存在此数据库的时候创建一个。

l         DatabaseConfig.setBtreeComparator()

设置用于Btree比较的比较器，通常是用来排序

l         DatabaseConfig.setDuplicateComparator()

设置用来比较一个key有两个不同值的时候的大小比较器。

l         DatabaseConfig.setSortedDuplicates()

设置一个key是否允许存储多个值，true代表允许，默认false.

l         DatabaseConfig.setExclusiveCreate()

以独占的方式打开，也就是说同一个时间只能有一实例打开这个database。

l         DatabaseConfig.setReadOnly()

以只读方式打开database,默认是false.

l         DatabaseConfig.setTransactional()

如果设置为true,则支持事务处理，默认是false，不支持事务。

                            使用示例：

package je.gettingStarted;

import com.sleepycat.je.DatabaseException;

import com.sleepycat.je.Database;

import com.sleepycat.je.DatabaseConfig;

...

// Environment open omitted for brevity

...

Database myDatabase = null;

try {

    DatabaseConfig dbConfig = new DatabaseConfig();

    dbConfig.setAllowCreate(true);

    dbConfig.setSortedDuplicates(true);

    myDatabase =

        myDbEnv.openDatabase(null,

                             "sampleDatabase",

                             dbConfig);

} catch (DatabaseException dbe) {

    // Exception handling goes here.

}

**一些用来管理的方法**

l         Database.getDatabaseName()

取得数据库的名称

如：String dbName = myDatabase.getDatabaseName();

l         Database.getEnvironment()

取得包含这个database的环境信息

如：Environment theEnv = myDatabase.getEnvironment();

l         Database.preload()

预先加载指定bytes的数据到RAM中。

如：myDatabase.preload(1048576l); // 1024\*1024

l         Environment.getDatabaseNames()

返回当前环境下的数据库列表

如：

import java.util.List;

List myDbNames = myDbEnv.getDatabaseNames();

for(int i=0; i < myDbNames.size(); i++) {

    System.out.println("Database Name: " + (String)myDbNames.get(i));

}

l         Environment.removeDatabase()

删除当前环境中指定的数据库。

如：

String dbName = myDatabase.getDatabaseName();

myDatabase.close();

myDbEnv.removeDatabase(null, dbName);

l         Environment.renameDatabase()

给当前环境下的数据库改名

如：

String oldName = myDatabase.getDatabaseName();

String newName = new String(oldName + ".new", "UTF-8");

myDatabase.close();

myDbEnv.renameDatabase(null, oldName, newName);

l         Environment.truncateDatabase()

清空database内的所有数据，返回清空了多少条记录。

如：

Int numDiscarded= myEnv.truncate(null,

myDatabase.getDatabaseName(),true);

System.out.println("一共删除了 " + numDiscarded +" 条记录 从数据库 " + myDatabase.getDatabaseName());

**6.**         **Database 记录**

JE的记录包含两部分,key键值和value数据值，这两个值都是通过DatabaseEntry对象封装起来，所以说如果要使用记录，则你必须创建两个DatabaseEntry对象，一个是用来做为key，另外一个是做为value.

DatabaseEntry能够支持任何的能够转换为bytes数组形式的基本数据。包括所有的JAVA基本类型和可序列化的对象.

**使用记录**

 示例一：把字符串转换DatabaseEntry

package je.gettingStarted;

import com.sleepycat.je.DatabaseEntry;

...

String aKey = "key";

String aData = "data";

try {

 //设置key/value,注意DatabaseEntry内使用的是bytes数组

**DatabaseEntry theKey=new DatabaseEntry(aKey.getBytes("UTF-8"));**

**DatabaseEntry theData=new DatabaseEntry(aData.getBytes("UTF-8"));**

} catch (Exception e) {

    // 错误处理

}

 示例二：把DatabaseEntry里的数据转换成字符串

byte[] myKey = **theKey.getData();**

byte[] myData =**theData.getData();**

String key = new String(myKey, "UTF-8");

String data = new String(myData, "UTF-8");

**读和写database 记录**

    读和写database记录的时候大体是基本一样的，唯一有区别的是每个key写是否允许写多条记录，默认情况下是不支持多条记录的。

**a)**     **你可以使用如下方法向database 里添加记录**

l        Database.put()

    向database中添加一条记录。如果你的database不支持一个key对应多个data或当前database中已经存在该key了，则使用此方法将使用新的值覆盖旧的值。

l        Database.putNoOverwrite()

    向database中添加新值但如果原先已经有了该key，则不覆盖。不管database是否允许支持多重记录(一个key对应多个value),只要存在该key就不允许添加，并且返回perationStatus.KEYEXIST信息。

l        Database.putNoDupData()

    想database中添加一条记录，如果database中已经存在了相同的 key和value则返回 OperationStatus.KEYEXIST.

使用示例：

    package je.gettingStarted;

    import com.sleepycat.je.Database;

    import com.sleepycat.je.DatabaseEntry;

    ...

    String aKey = "myFirstKey";

    String aData = "myFirstData";

    try {

        DatabaseEntry theKey = new DatabaseEntry(aKey.getBytes("UTF-8"));

        DatabaseEntry theData = new DatabaseEntry(aData.getBytes("UTF-8"));

**myDatabase.put(null, theKey, theData);**

    } catch (Exception e) {

        // Exception handling goes here

    }

**b)**     **你可以使用如下方法从database 里读取记录**

**1.**         Database.get()

       基本的读记录的方法，通过key的方式来匹配，如果没有改记录则返回OperationStatus.NOTFOUND。

l         Database.getSearchBoth()

       通过key和value来同时匹配，同样如果没有记录匹配key和value则会返回OperationStatus.NOTFOUND。

使用示例：

package je.gettingStarted;

import com.sleepycat.je.Database;

import com.sleepycat.je.DatabaseEntry;

import com.sleepycat.je.LockMode;

import com.sleepycat.je.OperationStatus;

...

String aKey = "myFirstKey";

try {

    DatabaseEntry theKey = new DatabaseEntry(aKey.getBytes("UTF-8"));

    DatabaseEntry theData = new DatabaseEntry();

**if (myDatabase.get(null, theKey, theData, LockMode.DEFAULT) ==**

**OperationStatus.SUCCESS) {**

        byte[] retData = theData.getData();

        String foundData = new String(retData, "UTF-8");

        System.out.println("For key: '" + aKey + "' found data: '" +

                            foundData + "'.");

    } else {

        System.out.println("No record found for key '" + aKey + "'.");

    }

} catch (Exception e) {

    // Exception handling goes here

}

**c)**     **删除记录**

可以使用Database.delete()这个方法来删除记录。如果你的database支持多重记录，则当前key下的所有记录都会被删除，如果只想删除多重记录中的一条则可以使用游标来删除。

    当然你也可以使用Environment.truncateDatabase()这个方法来清空database 中的所有记录。

使用示例：

package je.gettingStarted;

import com.sleepycat.je.Database;

import com.sleepycat.je.DatabaseEntry;

...

try {

    String aKey = "myFirstKey";

    DatabaseEntry theKey = new DatabaseEntry(aKey.getBytes("UTF-8"));

**myDatabase.delete(null, theKey);**

} catch (Exception e) {

}

**d)**     **提交事务**

当你对database进行了写操作的时候，你的修改不一定马上就能生效，有的时候他仅仅是缓存在RAM中，如果想让你的修改立即生效，则可以使用Environment.sync()方法来把数据同步到磁盘中去。

**e)**     **不同类型的数据的处理**

**1.**     你可以使用DatabaseEntry来绑定基本的JAVA数据类型，主要有String、Character、Boolean、Byte、Short、Integer、Long、Float、Double.

       使用示例一：

              package je.gettingStarted;

              import com.sleepycat.bind.EntryBinding;

              import com.sleepycat.bind.tuple.TupleBinding;

              import com.sleepycat.je.DatabaseEntry;

              ...

              try {

                  String aKey = "myLong";

                  DatabaseEntry theKey = new

                  DatabaseEntry(aKey.getBytes("UTF-8"));

**Long myLong = new Long(123456789l);**

**DatabaseEntry theData = new DatabaseEntry();**

**EntryBinding myBinding =**

**TupleBinding.getPrimitiveBinding(Long.class);**

**myBinding.objectToEntry(myLong, theData);**

                  myDatabase.put(null, theKey, theData);

              } catch (Exception e) {

                  // Exception handling goes here

              }

       使用示例二：

       package je.gettingStarted;

       import com.sleepycat.bind.EntryBinding;

       import com.sleepycat.bind.tuple.TupleBinding;

       import com.sleepycat.je.Database;

       import com.sleepycat.je.DatabaseEntry;

       import com.sleepycat.je.LockMode;

       import com.sleepycat.je.OperationStatus;

       ...

       Database myDatabase = null;

       try {

           String aKey = "myLong";

           DatabaseEntry theKey = new

              DatabaseEntry(aKey.getBytes("UTF-8"));

**DatabaseEntry theData = new DatabaseEntry();**

**EntryBinding myBinding =**

**TupleBinding.getPrimitiveBinding(Long.class);**

           OperationStatus retVal = myDatabase.get(null, theKey, theData,

              LockMode.DEFAULT);

           String retKey = null;

           if (retVal == OperationStatus.SUCCESS) {

**Long theLong = (Long) myBinding.entryToObject(theData);**

               retKey = new String(theKey.getData(), "UTF-8");

               System.out.println("For key: '" + retKey + "' found Long: '" +

                            theLong + "'.");

           } else {

               System.out.println("No record found for key '" + retKey + "'.");

           }

       } catch (Exception e) {

           // Exception handling goes here

       }

**2.**         **可序列化的对象的绑定**

1.         首先你需要创建一个可序列化对象

2.         打开或创建你的database，你需要两个，一个用来存储你的数据，另外一个用来存储类信息。

3.         实例化catalog类，这个时候你可以使用com.sleepycat.bind.serial.StoredClassCatalog,来存储你的类信息。

4.         通过com.sleepycat.bind.serial.SerialBinding来绑定数据和类。

5.         绑定并存储数据。

示例：

l         **创建一个可序列化的对象**

package je.gettingStarted;

import java.io.Serializable;

**public class MyData implements Serializable {**

    private long longData;

    private double doubleData;

    private String description;

    MyData() {

        longData = 0;

        doubleData = 0.0;

        description = null;

    }

    public void setLong(long data) {

        longData = data;

    }

    public void setDouble(double data) {

        doubleData = data;

    }

    public void setDescription(String data) {

        description = data;

    }

    public long getLong() {

        return longData;

    }

    public double getDouble() {

        return doubleData;

    }

    public String getDescription() {

        return description;

    }

}

l         **存储数据**

package je.gettingStarted;

import com.sleepycat.bind.EntryBinding;

import com.sleepycat.bind.serial.StoredClassCatalog;

import com.sleepycat.bind.serial.SerialBinding;

import com.sleepycat.je.Database;

import com.sleepycat.je.DatabaseConfig;

import com.sleepycat.je.DatabaseEntry;

...

String aKey = "myData";

**MyData data2Store = new MyData();**

data2Store.setLong(123456789l);

data2Store.setDouble(1234.9876543);

data2Store.setDescription("A test instance of this class");

try {

    DatabaseConfig myDbConfig = new DatabaseConfig();

    myDbConfig.setAllowCreate(true);

    myDbConfig.setSortedDuplicates(true);

    Database myDatabase = myDbEnv.openDatabase(null, "myDb", myDbConfig);

    myDbConfig.setSortedDuplicates(false);

**//打开用来存储类信息的库**

**Database myClassDb = myDbEnv.openDatabase(null, "classDb", myDbConfig)**;

**// 3）创建catalog**

**StoredClassCatalog classCatalog = new StoredClassCatalog(myClassDb);**

**// 4）绑定数据和类**

**EntryBinding dataBinding = new SerialBinding(classCatalog,**

**MyData.class);**

    DatabaseEntry theKey = new DatabaseEntry(aKey.getBytes("UTF-8"));

**// 向DatabaseEntry里写数据**

**DatabaseEntry theData = new DatabaseEntry();**

**dataBinding.objectToEntry(data2Store, theData);**

    myDatabase.put(null, theKey, theData);

} catch (Exception e) {

    // 错误处理

}

l         **读数据**

package je.gettingStarted;

import com.sleepycat.bind.EntryBinding;

import com.sleepycat.bind.serial.StoredClassCatalog;

import com.sleepycat.bind.serial.SerialBinding;

import com.sleepycat.je.Database;

import com.sleepycat.je.DatabaseConfig;

import com.sleepycat.je.DatabaseEntry;

import com.sleepycat.je.LockMode;

...

// The key data.

String aKey = "myData";

try {

    DatabaseConfig myDbConfig = new DatabaseConfig();

    myDbConfig.setAllowCreate(false);

    Database myDatabase = myDbEnv.openDatabase(null, "myDb", myDbConfig);

       //用来存储类信息的库

**Database myClassDb = myDbEnv.openDatabase(null, "classDb", myDbConfig);**

**// 实例化catalog**

**StoredClassCatalog classCatalog = new StoredClassCatalog(myClassDb);**

**// 创建绑定对象**

**EntryBinding dataBinding = new SerialBinding(classCatalog,**

**MyData.class);**

    DatabaseEntry theKey = new DatabaseEntry(aKey.getBytes("UTF-8"));

    DatabaseEntry theData = new DatabaseEntry();

    myDatabase.get(null, theKey, theData, LockMode.DEFAULT);

    // Recreate the MyData object from the retrieved DatabaseEntry using

**// 根据存储的类信息还原数据**

**MyData retrievedData=(MyData)dataBinding.entryToObject(theData);**

} catch (Exception e) {

    // Exception handling goes here

}

**3.**         **自定义对象的绑定**

       使用tuple binding 来绑定自定义数据的步骤

①.     实例化你要存储的对象

②.     通过com.sleepycat.bind.tuple.TupleBinding class来创建一个tuple binding。

③.     创建一个database，跟序列化的对象不同，你只需要创建一个。

④.     通过继承第二步的类来创建一个entry binding 对象。

⑤.     存储和使用数据

使用示例：

l         创建要存储的对象

package je.gettingStarted;

public class MyData2 {

    private long longData;

    private Double doubleData;

    private String description;

    public MyData2() {

        longData = 0;

        doubleData = new Double(0.0);

        description = "";

    }

    public void setLong(long data) {

        longData = data;

    }

    public void setDouble(Double data) {

        doubleData = data;

    }

    public void setString(String data) {

        description = data;

    }

    public long getLong() {

        return longData;

    }

    public Double getDouble() {

        return doubleData;

    }

    public String getString() {

        return description;

    }

}

l         创建一个TupleBinding对象

package je.gettingStarted;

import com.sleepycat.bind.tuple.TupleBinding;

import com.sleepycat.bind.tuple.TupleInput;

import com.sleepycat.bind.tuple.TupleOutput;

public class MyTupleBinding extends TupleBinding {

    // 把对象转换成TupleOutput

    public void objectToEntry(Object object, TupleOutput to) {

        MyData2 myData = (MyData2)object;

**to.writeDouble(myData.getDouble().doubleValue());**

**to.writeLong(myData.getLong());**

**to.writeString(myData.getString());**

    }

   //把TupleInput转换为对象

    public Object entryToObject(TupleInput ti) {

**Double theDouble = new Double(ti.readDouble());**

**long theLong = ti.readLong();**

**String theString = ti.readString();**

        MyData2 myData = new MyData2();

        myData.setDouble(theDouble);

        myData.setLong(theLong);

        myData.setString(theString);

        return myData;

    }

}

l          读和写数据

package je.gettingStarted;

import com.sleepycat.bind.tuple.TupleBinding;

import com.sleepycat.je.DatabaseEntry;

...

**TupleBinding keyBinding = new MyTupleBinding();**

MyData2 theKeyData = new MyData2();

theKeyData.setLong(123456789l);

theKeyData.setDouble(new Double(12345.6789));

theKeyData.setString("My key data");

DatabaseEntry myDate = new DatabaseEntry();

try {

    // 把theKeyData 存储到DatabaseEntry里

**keyBinding.objectToEntry(theKeyData,**myDate**);**

    ...

    // Database 进行了一些读和写操作

    ...

    // Retrieve the key data

**theKeyData = (MyData2) keyBinding.entryToObject(**myDate**);**

} catch (Exception e) {

    // 错误处理

}

**f)**     **使用比较器**

JE是使用BTrees来组织结构的，这意味着当对database的读和写需要涉及BTrees间的节点比较。这些比较在key间是经常的发生的。如果你的database支持多重记录，那么也会存在data间的比较。

    默认的情况JE的比较器是按照字节的方式来进行比较的，这通常情况下能处理大多数的情况。但有的时候确实需要自定义比较器用于特殊的通途，比如说按照key来排序。

l        **创建自己的比较器**

    其实很简单，只要你重写Comparator class中的比较方法（compare）就可以了，通过Comparator.compare()会传递给你两个byte 数组形式的值，如果你知道结构，则可以根据你自己定义的方法来进行比较

示例：

package je.gettingStarted;

import java.util.Comparator;

public class MyDataComparator**implements Comparator {**

    public MyDataComparator() {}

    public int **compare**(Object d1, Object d2) {

        byte[] b1 = (byte[])d1;

        byte[] b2 = (byte[])d2;

        String s1 = new String(b1, "UTF-8");

        String s2 = new String(b2, "UTF-8");

**return** s1.compareTo(s2);

    }

}

l        **让database使用你自定义的比较器**

如果你想改变database中基本的排序方式，你只能重新创建database并重新导入数据。

①.    DatabaseConfig.setBtreeComparator()

    用于在database里两个key的比较

②.    DatabaseConfig.setOverrideBtreeComparator()

    如果为true则代表让database使用 DatabaseConfig.setBtreeComparator()设置的比较器来代替默认的比较器。

③.    DatabaseConfig.setDuplicateComparator()

    用于database可以使用多重记录的时候的data的    比较。

④.    DatabaseConfig.setOverrideDuplicateComparator()

    如果为true则代表让database使用 DatabaseConfig. setDuplicateComparator()设置    的比    较器来代替默认的比较器。

使用示例：

package je.gettingStarted;

import com.sleepycat.je.Database;

import com.sleepycat.je.DatabaseConfig;

import com.sleepycat.je.DatabaseException;

import java.util.Comparator;

...

try {

    DatabaseConfig myDbConfig = new DatabaseConfig();

    myDbConfig.setAllowCreate(true);

**// 设置要使用的比较器**

**myDbConfig.setDuplicateComparator(MyDataComparator.class);**

**// 使用自己定义的比较器**

**myDbConfig.setSortedDuplicates(true);**

    Database myDatabase = myDbEnv.openDatabase(null, "myDb", myDbConfig);

} catch (DatabaseException dbe) {

    // Exception handling goes here

}

**六、**             **游标的使用**

游标提供了遍历你database中记录的一种机制，使用游标你可以获取，添加，和删除你的记录。如果你的database支持多重记录，则可以通过游标访问同一个key下的每一个记录。

l         **打开和关闭游标**

       要想使用游标则你必须通过Database.openCursor()方法来打开一个游标，你可以通过CursorConfig来配置你的游标。

       可以通过Cursor.close()方法来关闭游标。请注意在关闭database和环境前一定要关闭游标，否则会带来错误。

       打开游标示例：

package je.gettingStarted;

import com.sleepycat.je.Cursor;

import com.sleepycat.je.CursorConfig;

import com.sleepycat.je.Database;

import com.sleepycat.je.DatabaseException;

import com.sleepycat.je.Environment;

import java.io.File;

...

Environment myDbEnvironment = null;

Database myDatabase = null;

**Cursor myCursor = null;**

try {

    myDbEnvironment = new Environment(new File("/export/dbEnv"), null);

    myDatabase = myDbEnvironment.openDatabase(null, "myDB", null);

**myCursor = myDatabase.openCursor(null, null);**

} catch (DatabaseException dbe) {

    // Exception handling goes here ...

}

关闭游标示例：

package je.gettingStarted;

import com.sleepycat.je.Cursor;

import com.sleepycat.je.Database;

import com.sleepycat.je.Environment;

...

try {

    ...

} catch ... {

} finally {

    try {

        if (myCursor != null) {

**myCursor.close();**

        }

        if (myDatabase != null) {

            myDatabase.close();

        }

        if (myDbEnvironment != null) {

            myDbEnvironment.close();

        }

    } catch(DatabaseException dbe) {

        System.err.println("Error in close: " + dbe.toString());

    }

}

l         **通过游标来获取记录**

       可以通过游标的Cursor.getNext()方法来遍历记录，Cursor.getNext()表示游标指针向下移动一条记录。同样的Cursor.getPrev()表示游标指针向上移动一条记录。

       使用示例一：

package je.gettingStarted;

import com.sleepycat.je.Cursor;

import com.sleepycat.je.Database;

import com.sleepycat.je.DatabaseEntry;

import com.sleepycat.je.DatabaseException;

import com.sleepycat.je.LockMode;

import com.sleepycat.je.OperationStatus;

...

Cursor cursor = null;

try {

    cursor = myDatabase.openCursor(null, null);

    DatabaseEntry foundKey = new DatabaseEntry();

    DatabaseEntry foundData = new DatabaseEntry();

    // 通过cursor.getNex方法来遍历记录

 while (**cursor.getNext(foundKey, foundData, LockMode.DEFAULT) ==**

**OperationStatus.SUCCESS**) {

***String keyString = new String(foundKey.getData(), "UTF-8");***

***String dataString = new String(foundData.getData(), "UTF-8");***

        System.out.println("Key | Data : " + keyString + " | " +

                       dataString + "");

    }

} catch (DatabaseException de) {

    System.err.println("Error accessing database." + de);

} finally {

    // 使用后必须关闭游标

    cursor.close();

}

使用示例二：

package je.gettingStarted;

import com.sleepycat.je.Cursor;

import com.sleepycat.je.Database;

import com.sleepycat.je.DatabaseEntry;

import com.sleepycat.je.DatabaseException;

import com.sleepycat.je.LockMode;

import com.sleepycat.je.OperationStatus;

...

Cursor cursor = null;

try {

    ...

    // Open the cursor.

    cursor = myDatabase.openCursor(null, null);

    DatabaseEntry foundKey = new DatabaseEntry();

    DatabaseEntry foundData = new DatabaseEntry();

    // 使用cursor.getPrev方法来遍历游标获取数据

**while (cursor.getPrev(foundKey, foundData, LockMode.DEFAULT)**

**== OperationStatus.SUCCESS) {**

        String theKey = new String(foundKey.getData(), "UTF-8");

        String theData = new String(foundData.getData(), "UTF-8");

        System.out.println("Key | Data : " + theKey + " | " + theData + "");

    }

} catch (DatabaseException de) {

    System.err.println("Error accessing database." + de);

} finally {

    // 使用后必须关闭游标

    cursor.close();

}

l         **搜索数据**

       你可以通过游标方式搜索你的database记录，你也可以通过一个key来搜索你的记录，同样的你也可以通过key和value组合在一起来搜索记录。如果查询失败，则游标会返回OperationStatus.NOTFOUND。

       游标支持都检索方法如下：

1)        Cursor.getSearchKey()

       通过key的方式检索，使用后游标指针将移动到跟当前key匹配的第一项。

2)      Cursor.getSearchKeyRange()

    把游标移动到大于或等于查询的key的第一个匹配key,大小比较是通过你设置的比较器来完成的，如果没有设置则使用默认的比较器。

3)      Cursor.getSearchBoth()

    通过key和value方式检索，然后把游标指针移动到与查询匹配的第一项。

4)      Cursor.getSearchBothRange()

    把游标移动到所有的匹配key和大于或等于指定的data的第一项。

 比如说database存在如下的key/value记录，,大小比较是通过你设置的比较器来完成的，如果没有设置则使用默认的比较器。

假设你的database存在如下的记录。

               Alabama/Athens

               Alabama/Florence

               Alaska/Anchorage

               Alaska/Fairbanks

                Arizona/Avondale

               Arizona/Florence

然后查询

|  |  |  |
| --- | --- | --- |
| 查询的key | 查询的data | 游标指向 |
| Alaska | Fa | Alaska/Fairbanks |
| Arizona | Fl | Arizona/Florence |
| Alaska | An | Alaska/Anchorage |

使用示例：

package je.gettingStarted;

import com.sleepycat.je.Cursor;

import com.sleepycat.je.Database;

import com.sleepycat.je.DatabaseEntry;

import com.sleepycat.je.DatabaseException;

import com.sleepycat.je.LockMode;

import com.sleepycat.je.OperationStatus;

...

String searchKey = "Alaska";

String searchData = "Fa";

Cursor cursor = null;

try {

    ...

    cursor = myDatabase.openCursor(null, null);

    DatabaseEntry theKey =

         new DatabaseEntry(searchKey.getBytes("UTF-8"));

    DatabaseEntry theData =

         new DatabaseEntry(searchData.getBytes("UTF-8"));

    cursor = myDatabase.openCursor(null, null);

**OperationStatus retVal = cursor.getSearchBothRange(theKey,**

**theData, LockMode.DEFAULT);**

**if (retVal == OperationStatus.NOTFOUND) {**

        System.out.println(searchKey + "/" + searchData +

                           " not matched in database " +

                           myDatabase.getDatabaseName());

    } else {

        String foundKey = new String(theKey.getData(), "UTF-8");

        String foundData = new String(theData.getData(), "UTF-8");

        System.out.println("Found record " + foundKey + "/" + foundData +

                           "for search key/data: " + searchKey +

                           "/" + searchData);

    }

} catch (Exception e) {

    // Exception handling goes here

} finally {

   cursor.close();

}

l         **使用游标来定位多重记录**

如果你的库支持多重记录，你可以使用游标来遍历一个key下的多个data.

1)      Cursor.getNext(), Cursor.getPrev()

       获取上一条记录或下一条记录

2)      Cursor.getSearchBothRange()

    用语定位到满足指定data的第一条记录。

3)      Cursor.getNextNoDup(), Cursor.getPrevNoDup()

    跳到上一个key的最后一个data或下一个key的第一个data,忽略     当前key多重记录的存在。

4)      Cursor.getNextDup(), Cursor.getPrevDup()

    在当前key中把指针移动到前一个data或后一个data.

5)      Cursor.count()

    获取当前key下的data总数。

使用示例：

package je.gettingStarted;

import com.sleepycat.je.Cursor;

import com.sleepycat.je.Database;

import com.sleepycat.je.DatabaseEntry;

import com.sleepycat.je.DatabaseException;

import com.sleepycat.je.LockMode;

import com.sleepycat.je.OperationStatus;

...

Cursor cursor = null;

try {

    ...

    // Create DatabaseEntry objects

    // searchKey is some String.

    DatabaseEntry theKey = new DatabaseEntry(searchKey.getBytes("UTF-8"));

    DatabaseEntry theData = new DatabaseEntry();

    cursor = myDatabase.openCursor(null, null);

**OperationStatus retVal = cursor.getSearchKey(theKey,**

**theData, LockMode.DEFAULT);**

    // 如果count超过一个，则遍历

    if (cursor.count() > 1) {

        while (retVal == OperationStatus.SUCCESS) {

            String keyString = new String(theKey.getData(), "UTF-8");

            String dataString = new String(theData.getData(), "UTF-8");

            System.out.println("Key | Data : " + keyString + " | " +

                               dataString + "");

**retVal = cursor.getNextDup(theKey, theData, LockMode.DEFAULT);**

        }

    }

} catch (Exception e) {

    // Exception handling goes here

} finally {

   // Make sure to close the cursor

   cursor.close();

}

l         **通过游标来添加数据**

       你可以通过游标来向database里添加数据

       你可以使用如下方法来向database里添加数据

1)        Cursor.put()

    如果database不存在key,则添加，如果database存在key但允许多重记录，则可以通过比较器在适当的位置插入数据，如果key已存在且不支持多重记录，则替换原有的数据。

2)      Cursor.putNoDupData()

    如果存在相同的key和data则返回OperationStatus.KEYEXIST.

       如果不存在key则添加数据。

3)      Cursor.putNoOverwrite()

    如果存在相同的key在database里则返OperationStatus.KEYEXIS，

    如果不存在key则添加数据。

package je.gettingStarted;

import com.sleepycat.je.Cursor;

import com.sleepycat.je.Database;

import com.sleepycat.je.DatabaseEntry;

import com.sleepycat.je.OperationStatus;

...

String key1str = "My first string";

String data1str = "My first data";

String key2str = "My second string";

String data2str = "My second data";

String data3str = "My third data";

Cursor cursor = null;

try {

    ...

    DatabaseEntry key1 = new DatabaseEntry(key1str.getBytes("UTF-8"));

    DatabaseEntry data1 = new DatabaseEntry(data1str.getBytes("UTF-8"));

    DatabaseEntry key2 = new DatabaseEntry(key2str.getBytes("UTF-8"));

    DatabaseEntry data2 = new DatabaseEntry(data2str.getBytes("UTF-8"));

    DatabaseEntry data3 = new DatabaseEntry(data3str.getBytes("UTF-8"));

    cursor = myDatabase.openCursor(null, null);

**OperationStatus retVal = cursor.put(key1, data1); // 添加成功**

**retVal = cursor.put(key2, data2); // 添加成功**

**retVal = cursor.put(key2, data3); // 如果允许多重记录则添加成功                                                                                      //否则添加失败**

} catch (Exception e) {

    // Exception handling goes here

} finally {

   // Make sure to close the cursor

   cursor.close();

}

l         **使用游标来删除记录**

       你可以通过调用Cursor.delete().方法来删除当前游标所指向的记录。删除后如果没有移动过指针这个时候调用Cursor.getCurrent()还是可以得到当前值的，但移动以后就不可以了。如果没有重设指针，对同一个位置多次调用删除方法，会返回OperationStatus.KEYEMPTY状态。

    使用示例：

package je.gettingStarted;

import com.sleepycat.je.Cursor;

import com.sleepycat.je.Database;

import com.sleepycat.je.DatabaseEntry;

import com.sleepycat.je.LockMode;

import com.sleepycat.je.OperationStatus;

...

Cursor cursor = null;

try {

    ...

    DatabaseEntry theKey = new DatabaseEntry(searchKey.getBytes("UTF-8"));

    DatabaseEntry theData = new DatabaseEntry();

    cursor = myDatabase.openCursor(null, null);

    OperationStatus retVal = cursor.getSearchKey(theKey, theData,                                        LockMode.DEFAULT);

    //如果date不是多重记录.

    if (cursor.count() == 1) {

            System.out.println("Deleting " +

                               new String(theKey.getData(), "UTF-8") +

                               "|" +

                               new String(theData.getData(), "UTF-8"));

**cursor.delete();//删除当前记录**

    }

} catch (Exception e) {

    // Exception handling goes here

} finally {

   // Make sure to close the cursor

   cursor.close();

}

l         **修改当前游标所在位置的值**

       可以通过Cursor.putCurrent()方法来修改，这个方法只有一个参数就是将要修改的值。这个方法不能用在多重记录。

    使用示例：

import com.sleepycat.je.Cursor;

import com.sleepycat.je.Database;

import com.sleepycat.je.DatabaseEntry;

import com.sleepycat.je.LockMode;

import com.sleepycat.je.OperationStatus;

...

Cursor cursor = null;

try {

    ...

    DatabaseEntry theKey = new DatabaseEntry(searchKey.getBytes("UTF-8"));

    DatabaseEntry theData = new DatabaseEntry();

    cursor = myDatabase.openCursor(null, null);

    OperationStatus retVal = cursor.getSearchKey(theKey, theData,

LockMode.DEFAULT);

    //将要被替换的值

    String replaceStr = "My replacement string";

    DatabaseEntry replacementData =

        new DatabaseEntry(replaceStr.getBytes("UTF-8"));

**cursor.putCurrent(replacementData);//把当前位置用新值替换**

} catch (Exception e) {

    // Exception handling goes here

} finally {

   // Make sure to close the cursor

   cursor.close();

}

**七、**             **二级database**

在JE中包含你需要的(主要)数据的database被叫做primary database.而通过某种关系关联起来的叫secondary database.通常secondary database和primary database的key是一样的，只不过是为了对应多条不同类型的数据。

       你可以通过SecondaryDatabase来创建二级库，而通过使用继承了SecondaryKeyCreator的 SecondaryConfig 来配置二级库。

       通常primary database和secondary database间是存在某种关联的，所以如果对其中一个做了一些变动，另外一个可能也需要跟着做相应的变动。

l         **打开和关闭二级库**

       你可以通过Environment.openSecondaryDatabase()这个方法来打开一个二级库，在打开前你必须指定二级库的名称和配置信息。

    需要与primary database和secondary database间绑定的就是索引。二级库通常能够提供额外的信息。

    如果想要使用二级库，primary database不能支持多重记录,他的key必须得保证唯一。不然你没办法进行关联。

所以说如果你要想打开和创建一个二级库需要如下步骤

①.     打开你的primary database.

②.     创建key creater实例。

③.     设置你的二级库的配置文件，也就是创建SecondaryConfig

④.     这个时候你就可以打开你的二级库了。

使用示例如下：

package je.gettingStarted;

import com.sleepycat.bind.tuple.TupleBinding;

import com.sleepycat.je.Database;

import com.sleepycat.je.DatabaseConfig;

import com.sleepycat.je.DatabaseException;

import com.sleepycat.je.Environment;

import com.sleepycat.je.SecondaryDatabase;

import com.sleepycat.je.SecondaryConfig;

import java.io.File;

...

DatabaseConfig myDbConfig = new DatabaseConfig();

//二级库的配置信息

**SecondaryConfig mySecConfig = new SecondaryConfig();**

myDbConfig.setAllowCreate(true);

**mySecConfig.setAllowCreate(true);**

// 通常二级库经常是允许多重记录的

mySecConfig.setSortedDuplicates(true);

// primary database

Environment myEnv = null;

Database myDb = null;

**SecondaryDatabase mySecDb = null;**

try {

    //打开primary库

       String dbName = "myPrimaryDatabase";

    myEnv = new Environment(new File("/tmp/JEENV"), null);

    myDb = myEnv.openDatabase(null, dbName, myDbConfig);

    //创建tuple binding

**TupleBinding myTupleBinding = new MyTupleBinding();**

    // 创建二级库的key创建器

**FullNameKeyCreator keyCreator = new        FullNameKeyCreator(myTupleBinding);**

    // 设置二级库的创建器

**mySecConfig.setKeyCreator(keyCreator);**

    //打开二级库

    String secDbName = "mySecondaryDatabase";

**mySecDb = myEnv.openSecondaryDatabase(null, secDbName,**

**myDb, mySecConfig);**

} catch (DatabaseException de) {

    // 错误处理

}

可以调用二级库的close()方法来关闭一个二级库，在关闭 primary database前你必须先关闭一个二级库。

try {

    if (mySecDb != null) {

        mySecDb.close();

    }

    if (myDb != null) {

        myDb.close();

    }

    if (myEnv != null) {

        myEnv.close();

    }

} catch (DatabaseException dbe) {

    // Exception handling goes here

}

l         **二级库的键创建器**

              在使用二级库的时候，你必须提供二级库的键创建器来为二级库创建二级。在程序中你可以使用SecondaryConfig.setKeyCreator()来指定一个键创建器。你可以使用任何的数据创建二级库的键，只要对你来说是需要的。

    自定义键创建器需要继承SecondaryKeyCreator类，并且要重写其中的createSecondaryKey方法。createSecondaryKey返回一个boolean形式的值，如果返回false,则表示二级库不存在这个key.

**示例：**

    假设你的primary database使用如下的结构来存储data.

package je.gettingStarted;

public class PersonData {

    private String userID;

    private String surname;

    private String familiarName;

    public PersonData(String userID, String surname, String familiarName) {

        this.userID = userID;

        this.surname = surname;

        this.familiarName = familiarName;

    }

    public String getUserID() {

        return userID;

    }

    public String getSurname() {

        return surname;

    }

    public String getFamiliarName() {

        return familiarName;

    }

}

 你可以使用如下的方法来创建键创建器

package je.gettingStarted;、

import com.sleepycat.bind.tuple.TupleBinding;

import com.sleepycat.je.SecondaryKeyCreator;

import com.sleepycat.je.DatabaseEntry;

import com.sleepycat.je.DatabaseException;

import com.sleepycat.je.SecondaryDatabase;

import java.io.IOException;

public class FullNameKeyCreator implements SecondaryKeyCreator {

**private TupleBinding theBinding;**

    public FullNameKeyCreator(TupleBinding theBinding1) {

            theBinding = theBinding1;

    }

    public boolean createSecondaryKey(SecondaryDatabase secDb,

                                      DatabaseEntry keyEntry,

                                      DatabaseEntry dataEntry,

                                      DatabaseEntry resultEntry) {

        try {

**PersonData pd =**

**(PersonData) theBinding.entryToObject(dataEntry);**

                String fullName = pd.getFamiliarName() + " " +

                    pd.getSurname();

**resultEntry.setData(fullName.getBytes("UTF-8"));**

        } catch (IOException willNeverOccur) {}

        return true;

    }

}

然后你可以向下面的方法使用你的键创建器

package je.gettingStarted;

import com.sleepycat.bind.tuple.TupleBinding;

import com.sleepycat.je.Database;

import com.sleepycat.je.DatabaseException;

import com.sleepycat.je.Environment;

import com.sleepycat.je.SecondaryDatabase;

import com.sleepycat.je.SecondaryConfig;

...

Environment myEnv = null;

Database myDb = null;

SecondaryDatabase mySecDb = null;

try {

...

**TupleBinding myDataBinding = new MyTupleBinding();**

**//创建键创建器**

**FullNameKeyCreator fnkc = new FullNameKeyCreator(myDataBinding);**

    SecondaryConfig mySecConfig = new SecondaryConfig();

       //设置键创建器

**mySecConfig.setKeyCreator(fnkc);**

    String secDbName = "mySecondaryDatabase";

    mySecDb = myEnv.openSecondaryDatabase(null, secDbName, myDb,

                                          mySecConfig);

} catch (DatabaseException de) {

    // Exception handling goes here

} finally {

    try {

        if (mySecDb != null) {

            mySecDb.close();

        }

        if (myDb != null) {

            myDb.close();

        }

        if (myEnv != null) {

            myEnv.close();

        }

    } catch (DatabaseException dbe) {

        // Exception handling goes here

    }

}

l         **二级库的配置设置**

可以通过SecondaryConfig来配置二级库，SecondaryConfig是 DatabaseConfig的子类。所以你可以使用DatabaseConfig的一些信息。

    SecondaryConfig支持如下方法。

n         SecondaryConfig.setAllowPopulate()

       如果设置为true，则表示二级库允许自动填充。当primary database中的内容加进来后自动也会把secondary里的数据也填充进来。

n         SecondaryConfig.setKeyCreator()

       设置要使用的键创建器

l         **读二级库的数据**

你可以通过SecondaryDatabase.get()或使用SecondaryCursor.来读取二级库的信息。与primary database不同的是data不是直接返回给你，而是返回给你对应的primary database中的key和data和二级库中对应的key.

如果二级库支持多重记录，则只返回第一条对应的数据。如果想看其他的只能通过SecondaryCursor(二级库游标)了。

    使用示例：

package je.gettingStarted;

import com.sleepycat.je.DatabaseEntry;

import com.sleepycat.je.LockMode;

import com.sleepycat.je.OperationStatus;

import com.sleepycat.je.SecondaryDatabase;

...

try {

   ...

    String searchName = "John Doe";

    DatabaseEntry searchKey =

        new DatabaseEntry(searchName.getBytes("UTF-8"));

    DatabaseEntry primaryKey = new DatabaseEntry();

    DatabaseEntry primaryData = new DatabaseEntry();

    // 通过searchKey查找primary database的key和value

**OperationStatus retVal = mySecondaryDatabase.get(null, searchKey,**

**primaryKey, primaryData, LockMode.DEFAULT);**

} catch (Exception e) {

    // Exception handling goes here

}

l         **删除二级库的记录**

       通常来讲你不能够直接修改二级库的记录，而只能通过修改primary database来达到修改二级库的目的。但是你可以直接通过econdaryDatabase.delete()来删除二级库的记录。但是如果你的库支持多重记录，那么只能删除匹配的第一条。

    使用示例：

package je.gettingStarted;

import com.sleepycat.je.DatabaseEntry;

import com.sleepycat.je.DatabaseException;

import com.sleepycat.je.OperationStatus;

import com.sleepycat.je.SecondaryDatabase;

...

try {

    ...

    String searchName = "John Doe";

    DatabaseEntry searchKey =

        new DatabaseEntry(searchName.getBytes("UTF-8"));

    // 删除匹配JOHN DOE键的第一个二级库的记录

**OperationStatus retVal = mySecondaryDatabase.delete(**

**null, searchKey);**

} catch (Exception e) {

    // Exception handling goes here

}

l         **使用二级库游标**

跟primary database一样你同样可以使用游标来操作数据。当你使用二级库游标的去记录的时候，都会包含primary的记录。SecondaryCursor.getSearchBoth()来搜索并不是搜索key/value对，而是搜索key和primary key.

    使用示例：

package je.gettingStarted;

import com.sleepycat.je.DatabaseEntry;

import com.sleepycat.je.LockMode;

import com.sleepycat.je.OperationStatus;

import com.sleepycat.je.SecondaryDatabase;

import com.sleepycat.je.SecondaryCursor;

...

try {

    ...

    String secondaryName = "John Doe";

    DatabaseEntry secondaryKey =

        new DatabaseEntry(secondaryName.getBytes("UTF-8"));

    DatabaseEntry foundData = new DatabaseEntry();

    SecondaryCursor mySecCursor =

        mySecondaryDatabase.openSecondaryCursor(null, null);

**OperationStatus retVal = mySecCursor.getSearchKey(secondaryKey,**

**foundData, LockMode.DEFAULT);**

    while (retVal == OperationStatus.SUCCESS) {

        mySecCursor.delete();

**retVal = mySecCursor.getNextDup(secondaryKey,**

**foundData, LockMode.DEFAULT);**

    }

} catch (Exception e) {

    // Exception handling goes here

}

l         **关联database**

如果你创建了两个或两个以上的二级库，你可以通过JoinCursor.来把他们关联起来，可以采用JoinCursor来实现多维度的查询，例如可以查询

String theColor = "red";

String theType = "minivan";

String theMake = "Toyota";

来实现直接查找这些指定数据的记录。与普通的游标比较起来是可以对data域进行查询，与单个次级数据库游标查询比较起来是可以实现多个条件的联立查询。

1)        使用Join Cursor

a)         打开两个或多个跟同一个primary database关联的二级库

b)        对每个二级库分别定义游标。

c)        创建secondary cursors数组

d)        通过Database.join()方法来建立关系

e)         通过JoinCursor.getNext()方法来遍历相关记录直到OperationStatus is notSUCCESS.为止。

f)         关闭你的游标

g)        关闭你所有的二级游标。

使用示例：

package je.gettingStarted;

import com.sleepycat.je.Database;

import com.sleepycat.je.DatabaseEntry;

import com.sleepycat.je.DatabaseException;

import com.sleepycat.je.JoinCursor;

import com.sleepycat.je.LockMode;

import com.sleepycat.je.OperationStatus;

import com.sleepycat.je.SecondaryCursor;

import com.sleepycat.je.SecondaryDatabase;

...

// Query strings:

String theColor = "red";

String theType = "minivan";

String theMake = "Toyota";

**SecondaryCursor colorSecCursor = null;**

**SecondaryCursor typeSecCursor = null;**

**SecondaryCursor makeSecCursor = null;**

**JoinCursor joinCursor = null;**

// 这些是用来做查询用的

DatabaseEntry foundKey = new DatabaseEntry();

DatabaseEntry foundData = new DatabaseEntry();

try {

    DatabaseEntry color = new DatabaseEntry(theColor.getBytes("UTF-8"));

    DatabaseEntry type = new DatabaseEntry(theType.getBytes("UTF-8"));

    DatabaseEntry make = new DatabaseEntry(theMake.getBytes("UTF-8"));

       // 创建二级库游标:

**colorSecCursor = automotiveColorDB.openSecondaryCursor(**

**null, null);**

**typeSecCursor = automotiveTypeDB.openSecondaryCursor(**

**null, null);**

**makeSecCursor = automotiveMakeDB.openSecondaryCursor(**

**null, null);**

    // 查询条件

**OperationStatus colorRet =**

**colorSecCursor.getSearchKey(color, foundData, LockMode.DEFAULT);**

**OperationStatus typeRet =**

**typeSecCursor.getSearchKey(type, foundData, LockMode.DEFAULT);**

**OperationStatus makeRet =**

**makeSecCursor.getSearchKey(make, foundData, LockMode.DEFAULT);**

    // If all our searches returned successfully, we can proceed

    if (colorRet == OperationStatus.SUCCESS &&

        typeRet == OperationStatus.SUCCESS &&

        makeRet == OperationStatus.SUCCESS) {

       // Get a secondary cursor array and populate it with our

        // 创建SecondaryCursor

**SecondaryCursor[] cursorArray = {colorSecCursor,**

**typeSecCursor, makeSecCursor, null};**

        // Create the join cursor

**joinCursor = automotiveDB.join(cursorArray, null);**

        // Now iterate over the results, handling each in turn

        while (joinCursor.getNext(foundKey, foundData, LockMode.DEFAULT) ==

                        OperationStatus.SUCCESS) {

            // Do something with the key and data retrieved in

            // foundKey and foundData

        }

    }

} catch (DatabaseException dbe) {

    // Error reporting goes here

} catch (Exception e) {

    // Error reporting goes here

} finally {

    try {

        // Make sure to close out all our cursors

        if (colorSecCursor != null) {

            colorSecCursor.close();

        }

        if (typeSecCursor != null) {

            typeSecCursor.close();

        }

        if (makeSecCursor != null) {

            makeSecCursor.close();

        }

        if (joinCursor != null) {

            joinCursor.close();

        }

    } catch (DatabaseException dbe) {

        // Error reporting goes here

    }

}

l         **配置JoinCursor**

你可以通过JoinConfig来配置joinCursor,JoinConfig只有一个方法。

    JoinConfig.setNoSort()，如果设置为true，则表示不使用自动排序，false的时候则按照关联少的到关联多的方式排序。

    使用示例：

JoinConfig config = new JoinConfig();

config.setNoSort(true);

JoinCursor joinCursor = myDb.join(cursorArray, config);

**八、**             **数据的备份和恢复**

在通常情况下你可以通过把你的database文件移动到安全的地方来保护你的数据。可能下面这些情况可能需要你额外需要考虑的

l         **数据库和日志**

              在了解数据备份和恢复前你有必要了解下JE database的内部数据结构、日志文件          和内存CACHE。

              JE以追加的方式向日志文件中添加数据，这意味着你日志文件永远不会被覆盖，          修改和删除。日志文件的命名是NNNNNNNN.jdb，NNNNNNNN是八进制的数字，第      一个日志文件名是00000000.jdb,以后每增加一个则加一。每个日志文件的最大大小默认    是10000000 bytes ,当日志文件超过这个大小后就会在生成一个日志文件，但你可以通          过修改je.log.fileMax来修改默认的日志文件的大小。

              因为JE使用不覆盖的方式来写日志文件，所以过一段时间必须得清理或压缩下日         志文件用来释放更多的磁盘空间。JE使用后台线程来清理任务。如果这个日志文件已             经不在使用，那么清理器就会自动帮你删除或给你加个删除标记。

              JE databases采用BTree的方式组织，当记录被添加、修改或删除，实际上这些操          作都是控制BTree的子节点，比如说更改一个节点到另外一个父节点下。

              数据库修改和同步，当对数据进行操作的时候，操作是直接操作内存中的BTree            子节点的，如果你的database不支持事务，则cache是唯一的，以保证相互间的修改不      会产生冲突。这种不时时把数据同步到磁盘其实是比较好的，因为他减少了频繁的IO             读取，从而提高了速度。但是如果你的数据要求十时性很强，你可以手动调用       Environment.sync()来时时的把数据同步到你的磁盘中，但这样可能会带来IO的繁忙。

              其实一但当环境被打开，JE就会尝试着从日志文件中恢复那些被丢失的节点

              你可以选择热备份和冷备份两种，但他都数据完全备份（也就是说所有的数据），        所以说不要把冷热备份与完全备份和增量备份相混淆。

              所谓的热备份就是在你的程序还在运行的时候把所有的\*.jdb文件COPY出去，这          这样的好处是你不必停止你的服务。但这样当前CACHE在内存中的数据还没有被备份     下来。

              离线备份比较完全，但他的缺点也是显而易见的，你必须终止你的服务。首先你要       终止你的写操作，然后调用Environment.sync()方法来把你内存中的数据同步      到硬盘中去，关闭你的游标，你的数据库,你的环境，这个时候把所有的.jdb 文件COPY出去就行了。

l         **使用备份帮助类**

        JE提供了DbBackup 这个类来协助你保护好你的数据。

       使用示例：

package je.gettingStarted;

...

import com.sleepycat.je.util.DbBackup;

...

    Environment env = new Environment(...);

    DbBackup backupHelper = new DbBackup(env);

    // 找到前一次备份的版本号

    // 检查保存状态

    long lastFileCopiedInPrevBackup = ...

    // 开始备份，检查你要备份那些文件

    backupHelper.startBackup();

    try {

**String[] filesForBackup =**

**backupHelper.getLogFilesInBackupSet(lastFileCopiedInPrevBackup);**

        // 复制文件

        myApplicationCopyMethod(filesForBackup)

        // 保存你本次备份的信息

        lastFileCopiedInPrevBackup = backupHelper.getLastFileInBackupSet();

        myApplicationSaveLastFile(lastFileCopiedInBackupSet);

    }

    finally {

        // Remember to exit backup mode, or all log files won't be cleaned

        // and disk usage will bloat.

**backupHelper.endBackup();**

   }

l         **灾难恢复**

1.         停止你的服务

2.         删除所有的在你环境所在目录的内容

3.         把你的备份COPY到你的目录中

4.         如果你是backup的增量方式备份你的环境目录，请COPY最新的哪个版本（翻译的不是十分准确原文如下：If you are using a backup utility that runs incremental backups of your environment directory, copy any log files generated since the time of your last full backup. Be sure to restore all log files in the order that they were written. The order is important because it is possible the same log file appears in multiple archives, and you want to run recovery using the most recent version of each log file）

5.         重新打开你的环境，恢复你的服务。

l         **热备份**

1.         从你的环境中复制所有的.jdb文件到你想要备份的地方，不管是热备份还是离线备份都要这么做。在备份前要确保你把内存中的数据也同步到了磁盘中去了。这个阶段属于完全备份。

2.         然后定期复制新生成的那些.jdb文件。

3.         删除那些已经做了删除标记的旧的jdb文件（通常是调用cleaner产生的）。

**九、**             **BDB针对应用的一些管理**

l         **用je.properties来初始化一些配置信息**

    je.properties必须位于你环境的主目录下，且名称必须是je.properties.

       je使用一些后台的线程来对你的环境进行预配置，这些后台线程包括如下：

l         Cleaner thread. 主要是进行一些清理性工作，当环境进行写操作的时候才启动。

**1)**        **je.cleaner.minUtilization**

主要是用来确保腾出一定的空间来存放记录。如果空间不够用了则移出一些旧的不在使用的记录，默认这个空间占50%。

**2)**     **je.cleaner.expunge**

主要是用来确认一个日志文件是否已经没用，可以被删除了，如果一个日志由原先的NNNNNNNN.jdb 名称被修改为NNNNNNNN.del名称那么你要对这进行一些处理。

l         Compressor thread 用来压缩一些已经删除的，被修改了的没用的BTree节点，当环境发生写操作的时候才触发。

l         Checkpointer thread，检查你的环境，本编程总是运行。

l         **Sizing the Cache内存中CACHE大小的设置**

       你可以使用 EnvironmentMutableConfig.setCachePercent()来修改

je.maxMemoryPercent属性来改变je能够使用的最大内存的占整个JVM虚拟机内存的百分比。

       你也可以通过使用EnvironmentConfig.setCacheSize().来改变je.maxMemory中设置的JE能够使用的最大内存。

       你也可以通过调用EnvironmentStats.getNCacheMiss()来取得当前有多少数据没有通过内存的方式来命中。

l         **命令行工具**

1)        DbDump，把database转换成一种可读的形式，通常用来导出数据。

       参数如下：

①.     –f ：指定把数据输出到那个文件中，如果没有指定则输出到控制台（也就是屏幕）。

②.     –h ：用来指定环境所在目录，这个参数必须指定。

③.     –l ：列出当前环境的数据库名，如果没有指定-s着此参数必选。

④.     –p ：以可读的方式输出数据库记录

⑤.     –r :这个操作通过不同的排序方式输出一些经过整理的那些有用的数据数据，通常如果指定你的数据库名为dbname,则他会把结果输出到dbname.dump(翻译的不是太准确原文如下：Salvage data from a possibly corrupt file. When used on a uncorrupted database, this option should return data equivalent to a normal dump, but most likely in a different order. This option causes the ensuing output to go to a file named dbname.dump where dbname is the name of the database you are dumping. The file is placed in the current working directory.)

⑥.     –R ：跟-r有所不同，他输出的是所有的数据包括被删除的和没有被删除的（原文如下：Aggressively salvage data from a possibly corrupt file. This option differs from the -r option in that it will return all possible data from the file at the risk of also returning already deleted or otherwise nonsensical items. Data dumped in this fashion will almost certainly have to be edited by hand or other means before the data is ready for reload into another database. This option causes the ensuing output to go to a file named dbname.dump where dbname is the name of the database you are dumping. The file is placed in the current working directory.）

⑦.     –s：指定数据库名称 如果没有指定-l参数，则此参数必选。

⑧.     –v：确定显示信息是按照-r模式还是按照-R模式

⑨.     –V:打印数据库版本后推出

使用示例：

**> java com.sleepycat.je.util.DbDump -h . -p -s VendorDB**

VERSION=3

format=print

type=btree

database=VendorDB

dupsort=false

HEADER=END

 Mom's Kitchen

 sr/01/01xpt/00/0d53 Yerman Ct.t/00/0c763 554 9200t/00/0bMiddle Townt/00

 /0eMaggie Kultgent/00/10763 554 9200 x12t/00/02MNt/00/0dMom's Kitchent/00

 /0555432

 Off the Vine

 sr/01/01xpt/00/10133 American Ct.t/00/0c563 121 3800t/00/0aCentennialt/00

 /08Bob Kingt/00/10563 121 3800 x54t/00/02IAt/00/0cOff the Vinet/00/0552002

 Simply Fresh

 sr/01/01xpt/00/1115612 Bogart Lanet/00/0c420 333 3912t/00/08Harrigant/00

 /0fCheryl Swedbergt/00/0c420 333 3952t/00/02WIt/00/0cSimply Fresht/00/0

 553704

 The Baking Pan

 sr/01/01xpt/00/0e1415 53rd Ave.t/00/0c320 442 2277t/00/07Dutchint/00/09

 Mike Roant/00/0c320 442 6879t/00/02MNt/00/0eThe Baking Pant/00/0556304

 The Pantry

 sr/01/01xpt/00/111206 N. Creek Wayt/00/0c763 555 3391t/00/0bMiddle Town

 t/00/0fSully Beckstromt/00/0c763 555 3391t/00/02MNt/00/0aThe Pantryt/00

 /0555432

 TriCounty Produce

 sr/01/01xpt/00/12309 S. Main Streett/00/0c763 555 5761t/00/0bMiddle Townt

 /00/0dMort Dufresnet/00/0c763 555 5765t/00/02MNt/00/11TriCounty Producet

 /00/0555432

DATA=END

>

2)        DbLoad 导入通过DbDump方式导出的数据

       常见的参数如下：

①.     –c: 可以指定两个参数 database 表示将要被导入的库名 dupsort 如果为true表示数据库支持多重记录。

②.     –f:指定将要导入的文件

③.     –n: 不覆盖已经存在的值(key/value),如果存在相同的则输出警告信息。

④.     –h:指定环境所在的目录。

⑤.     –s：指定数据库的名称

⑥.     –t:把指定的文本信息导入到库中，文本信息必须以key/value形式成对出现。

⑦.     –v：输出详细信息

⑧.     –V：输出数据库版本；

使用示例：

> java com.sleepycat.je.util.DbDump -h . -s VendorDB -f vendordb.txt

> java com.sleepycat.je.util.DbLoad -h . -f vendordb.txt

>

3)        DbVerify，检查数据库文件是否有错误。参数如下：

①.     –h:确定环境路径

②.     –q:禁止输出详细的调试信息，只输出成功还是失败。

③.     –s：确定数据库的名称。

④.     –v：输出每个节点信息。

⑤.     –V：输出数据库版本信息。

使用示例：

**> java com.sleepycat.je.util.DbVerify -h . -s VendorDB**

<BtreeStats>

<BottomInternalNodesByLevel total="1">

 <Item level="1" count="1"/>

</BottomInternalNodesByLevel>

<InternalNodesByLevel total="1">

 <Item level="2" count="1"/>

</InternalNodesByLevel>

<LeafNodes count="6"/>

<DeletedLeafNodes count="0"/>

<DuplicateCountLeafNodes count="0"/>

<MainTreeMaxDepth depth="2"/>

<DuplicateTreeMaxDepth depth="0"/>

</BtreeStats>

[Berkeley DB Java Edition 基本示例](http://blog.csdn.net/muyannian/article/details/1717468)

**package** test;

**import** com.sleepycat.je.\*;

**import** com.sleepycat.bind.EntryBinding;

**import** com.sleepycat.bind.tuple.TupleBinding;

**import** com.sleepycat.bind.serial.StoredClassCatalog;

**import** com.sleepycat.bind.serial.SerialBinding;

**import** java.io.File;

**public** **class** test {

**public** **static** **void** main(String[] args) {

    }

    /\*\*

     \* 打开和关闭环境，示例一

     \*

     \*/

**public** **void** eg1(){

        //----打开环境，如果不存在，则创建一个------------

        Environment myDbEnvironment=**null**;

**try** {

            EnvironmentConfig envConfig = **new** EnvironmentConfig();

            envConfig.setAllowCreate(**true**); //true不存在就创建,false如果不存在则打开环境失败

            //envConfig.setReadOnly(true); //true 以只读方式打开，如果是多进程应用，每个进程都要设置为true

            //envConfig.setTransactional(true);//true支持事务，false不支持，默认false。可以更改配置文件来设置此参数。

            myDbEnvironment = **new** Environment(**new** File(".//"), envConfig);//环境所在路径

            java.util.List myDbNames = myDbEnvironment.getDatabaseNames(); //得到所有的数据库的名字

**for**(**int** i=0; i < myDbNames.size(); i++) {

                System.*out*.println("Database Name: " + (String)myDbNames.get(i));

            }

        } **catch** (DatabaseException dbe) {

            // 错误处理

        }

        //----关闭环境----------------

**try** {

**if** (myDbEnvironment != **null**) {

            myDbEnvironment.cleanLog(); //在关闭前也最好执行一下cleaner，清理一下日志文件，因为delete操作会浪费一些空间

            myDbEnvironment.close();

            }

        } **catch** (DatabaseException dbe) {

            // 错误处理

        }

    }

    /\*\*

     \* 打开环境示例二

     \*

     \*/

**public** **void** eg2()

    {

        Environment myEnv = **null**;

**try**{

            myEnv=**new** Environment(**new** File("/export/dbEnv"), **null**);

            EnvironmentMutableConfig envMutableConfig = **new** EnvironmentMutableConfig();

            envMutableConfig.setCachePercent(50);//设置je的cache占用jvm 内存的百分比。

            //envMutableConfig.setCacheSize(123456);//设定缓存的大小为123456Bytes

            envMutableConfig.setTxnNoSync(**true**);//设定事务提交时是否写更改的数据到磁盘，true不写磁盘。

            //envMutableConfig.setTxnWriteNoSync(false);//设定事务在提交时，是否写缓冲的log到磁盘。如果写磁盘会影响性能，不写会影响事务的安全。随机应变。

            myEnv.setMutableConfig(envMutableConfig);

            EnvironmentStats envStats=myEnv.getStats(**null**);//调用 Environment.getStats() 返回一个EnvironmentStas对象。调用EnvironmentStats.getNCacheMiss()看命不中数。

**long** cacheMisses = envStats.getNCacheMiss(); //这个统计非常重要，尤其针对于长时间运行的应用。 它返回不能够在内存中命中的请求总数，这可以用来参考指定cache的大小。

            //myEnv.getMutableConfig();//得到当前的环境配置信息

        }**catch**(DatabaseException dbe){}

    }

    /\*\*

     \* 打开database,以及相关操作

     \*

     \*/

**public** **void** eg3(){

        Environment myDbEnvironment = **null**;

        Database myDatabase = **null**;

**try** {

            // Open the environment. Create it if it does not already exist.

            EnvironmentConfig envConfig = **new** EnvironmentConfig();

            envConfig.setAllowCreate(**true**);

            myDbEnvironment = **new** Environment(**new** File("/export/dbEnv"), envConfig); //也可能用DatabaseConfig参数来打开，这样就可以设置数据库的属性，比如是否允许在库不存在时创建它，是否只读打开，是否支持事务等。

            // Open the database. Create it if it does not already exist.

            DatabaseConfig dbConfig = **new** DatabaseConfig();

            dbConfig.setAllowCreate(**true**);

            //transaction为null，不支持事务

            myDatabase = myDbEnvironment.openDatabase(**null**,"sampleDatabase",dbConfig); //打开库，要提供一个数据库名字作为参数

        } **catch** (DatabaseException dbe) {

            // Exception handling goes here

        }

        //记录

        String aKey = "key";

        String aData = "data";

**try** {

            DatabaseEntry theKey = **new** DatabaseEntry(aKey.getBytes("UTF-8"));//最好指定编码方式，因为不指定编码会用系统的默认编码来转换，因为系统的默认编码可能会被人更改。

            DatabaseEntry theData = **new** DatabaseEntry(aData.getBytes("UTF-8"));

**byte**[] myKey = theKey.getData();

**byte**[] myData = theData.getData();

            //从byte数组转换为string的方法

            String key = **new** String(myKey, "UTF-8");

            String data = **new** String(myData, "UTF-8");

            //get和put用在非重复的数据存储，读写库时，注意一个小区别，就是数据库，是否允许重复的记录存在，两个记录公用一个key，这就是重复的记录，我们把重复的记录成为重复集合。或者叫多重。

            //游标用于重复数据存储put和get。

            //数据记录在内部是用Btree按照特定排序来存储的。一般是用key来排序的，key相同的多重数据是按照data来排序。

            //记录Using Database Records

            //记录是由key和data组成，即所熟悉的key->value，二者都被是有DatabaseEntry封装的。

            //这个之前也提过很多次了，DatabaseEntry可以封装原始类型和复杂的对象类型，二者都要被转换为byte array存储，转换可以使用Bind API来完成

            //写数据

            myDatabase.put(**null**, theKey, theData);//如果不是可重复数据库，put将会覆盖原有的记录。

            //myDatabase.putNoOverwrite(null, theKey, theData);//不允许覆盖，不管是否允许数据重复。

            //读数据

            //--myDatabase.getSearchBoth(null, theKey, theData, LockMode.DEFAULT);//查找key和data都匹配的记录

            //--查询出来的key和data都是byte数组形式。

**if** (myDatabase.get(**null**, theKey, theData, LockMode.*DEFAULT*) ==OperationStatus.*SUCCESS*)

            {

**byte**[] retData = theData.getData();

                String foundData = **new** String(retData, "UTF-8");

                System.*out*.println("For key: '" + aKey + "' found data: '" +foundData + "'.");

            }

            //删除数据

            myDatabase.delete(**null**, theKey); //删除数据

          } **catch** (Exception e) {}

        //关闭数据库

        //如果打开了游标，关闭时JE会发出警告，让你关闭他们先。活动状态的游标在关闭库的过程中会产生意想不到的结果，尤其是其他线程在写库的过程中。确定所有的访问都结束后再关闭库

**try** {

**if** (myDatabase != **null**) {

                myDatabase.close();

                myDbEnvironment.renameDatabase(**null**, "sampleDatabase", "test");//重命名,必须先关闭数据库

                myDbEnvironment.removeDatabase(**null**, "sampleDatabase");//删除数据库,必须先关闭数据库

                //myDbEnvironment.truncateDatabase(null, myDatabase.getDatabaseName(),true);//删除并回收数据库空间 ，true返回删除的记录的数量,false不返回删除的记录数量值

            }

**if** (myDbEnvironment != **null**) {

                myDbEnvironment.close();

            }

        } **catch** (DatabaseException dbe) {

            // Exception handling goes here

        }

    }

    /\*\*

     \* Deferred Write Databases 缓冲写库

     \*

     \* By默认，JE的数据库是持久化，意思就是说不缓存，写库的，这样多个应用都可以使用，把数据库设置成非持久化的，就成为缓冲写库。

     \* 缓冲写库本质上说就是内存级数据库，这适用于那些临时的操作。比如把oracle中的数据导入bdb，然后读bdbcache。

     \* 缓冲写库并不是总是请求磁盘I/O，很重要的一点，缓冲写库会在内存不够用的时候写磁盘。如果内存够大，用缓存写库那是最好不过了。

     \* 调用Database.sync()让修改过的cache数据写到磁盘

     \* 当重新打开一个缓冲写库的环境时，数据库的状态是和上一次sync时一致的。如果缓冲写库没有同步，就会当成是空库。问号，那么在数据关闭之前一定要sync？maybe yes。

     \* 程序员很容易就可以把缓冲写库的修改存储到磁盘上，对经常性的修改，增加，及删除等等的操作的应用也很有用处。通过延迟写库，延迟了磁盘IO， 这可以提高数据的流量。

     \* 注意，当没有调用sync的库关闭时而且环境也关闭的时候，cache的修改会丢失。如果没有关闭环境，所有的cache的修改还是会保留的。

     \* 设置库为defered or not，然后用DatabaseConfig的选项打开库。

     \*/

**public** **void** eg4(){

        Environment myDbEnvironment = **null**;

        Database myDatabase = **null**;

**try** {

            EnvironmentConfig envConfig = **new** EnvironmentConfig();

            envConfig.setAllowCreate(**true**);

            myDbEnvironment = **new** Environment(**new** File("/export/dbEnv"), envConfig);

            DatabaseConfig dbConfig = **new** DatabaseConfig();

            dbConfig.setAllowCreate(**true**); //设置允许创建与否，默认值是false，打开不存在的数据库会报错。true的时候，数据库不存在就创建。

            //dbConfig.setBtreeComparator();//设置B树比较器，用来比较两个记录的key是否相同。

            //dbConfig.setDuplicateComparator();//设置允许key重复的比较器。

            dbConfig.setSortedDuplicates(**false**);//设置为true，允许key重复，false的话，put一个存在key的记录会产生错误。如果使用了关联了多个索引则一定不能支持重复的记录。

            //dbConfig.setExclusiveCreate(false);//如果true，只能创建，如果存在，则打开失败If true, the database open fails if the database currently exists. That is, the open must result in the creation of a new database. Default is false.

            //dbConfig.setReadOnly(true);//设置是否只读

            //dbConfig.setTransactional(true);//设置是否支持事务

            dbConfig.setDeferredWrite(**true**);  //true为进行缓冲写库，false则不进行缓冲写库

            myDatabase = myDbEnvironment.openDatabase(**null**,"sampleDatabase",dbConfig);

            String dbName = myDatabase.getDatabaseName();//得到数据库的名字

            Environment theEnv = myDatabase.getEnvironment();//得到当前数据库环境

            myDatabase.preload(1024\*1024); //预先加载数据到内存，long型参数表示要装载到内存的数据的最大数

            //long类型的数据存储方法

**try** {

                String aKey = "myLong";

                DatabaseEntry theKey = **new** DatabaseEntry(aKey.getBytes("UTF-8"));

                Long myLong = **new** Long(123456789l);

                DatabaseEntry theData = **new** DatabaseEntry();

                EntryBinding myBinding = TupleBinding.*getPrimitiveBinding*(Long.**class**);

                myBinding.objectToEntry(myLong, theData);

                //存储long类型的数据

                myDatabase.put(**null**, theKey, theData);

                OperationStatus retVal = myDatabase.get(**null**, theKey, theData,LockMode.*DEFAULT*);String retKey = **null**;

**if** (retVal == OperationStatus.*SUCCESS*) {

                   //取得long类型的数据

                    Long theLong = (Long) myBinding.entryToObject(theData);

                    retKey = **new** String(theKey.getData(), "UTF-8");

                    System.*out*.println("For key: '" + retKey + "' found Long: '" +

                                        theLong + "'.");

                } **else** {

                    System.*out*.println("No record found for key '" + retKey + "'.");

                }

            } **catch** (Exception e) {

                // Exception handling goes here

            }

            //implements Serializable 的对象的存储

**try** {

                String aKey = "myLong";

                DatabaseEntry theKey = **new** DatabaseEntry(aKey.getBytes("UTF-8"));

                //写对象

                java.util.Vector data2Store = **new** java.util.Vector(); //假设他是implements Serializable

                StoredClassCatalog classCatalog = **new** StoredClassCatalog(myDatabase);

                EntryBinding dataBinding = **new** SerialBinding(classCatalog, java.util.Vector.**class**);//指定类型

                DatabaseEntry theData = **new** DatabaseEntry();

                dataBinding.objectToEntry(data2Store, theData);//绑定数据

                myDatabase.put(**null**, theKey, theData);

                //读对象

                myDatabase.get(**null**, theKey, theData, LockMode.*DEFAULT*);

                java.util.Vector retrievedData = (java.util.Vector) dataBinding.entryToObject(theData);

            } **catch** (Exception e) {

                // Exception handling goes here

            }

            //做一些处理

            myDatabase.sync(); //当写完一定量的数据以后，同步要硬盘中

        } **catch** (DatabaseException dbe) {

            // Exception handling goes here

        }

        //关闭数据库

**try** {

**if** (myDatabase != **null**) {

                myDatabase.close();

            }

**if** (myDbEnvironment != **null**) {

                myDbEnvironment.close();

            }

        } **catch** (DatabaseException dbe) {

            // Exception handling goes here

        }

    }

    /\*\*

     \* Data Persistence 持久化

     \* 如果是在内存中修了数据，需要写到磁盘。

     \* 怕因为系统错误引发数据丢失，可以使用transaction，每commit一次，修改都会被保存。

     \* 只是用来存放应用临时数据的话，就不需要用transaction了。

     \* 在每次关闭env的时候会执行，也可以在程序中调用执行。

     \*/

}

/\*\*

 \* 特殊对象间的自定义转换

 \*

 \*/

/\*

package je.gettingStarted;

import com.sleepycat.bind.tuple.TupleBinding;

import com.sleepycat.bind.tuple.TupleInput;

import com.sleepycat.bind.tuple.TupleOutput;

public class MyTupleBinding extends TupleBinding {

    // Write a MyData2 object to a TupleOutput

    public void objectToEntry(Object object, TupleOutput to) {

        MyData2 myData = (MyData2)object;

        // Write the data to the TupleOutput (a DatabaseEntry).

        // Order is important. The first data written will be

        // the first bytes used by the default comparison routines.

        to.writeDouble(myData.getDouble().doubleValue());

        to.writeLong(myData.getLong());

        to.writeString(myData.getString());

    }

    // Convert a TupleInput to a MyData2 object

    public Object entryToObject(TupleInput ti) {

        // Data must be read in the same order that it was

        // originally written.

        Double theDouble = new Double(ti.readDouble());

        long theLong = ti.readLong();

        String theString = ti.readString();

        MyData2 myData = new MyData2();

        myData.setDouble(theDouble);

        myData.setLong(theLong);

        myData.setString(theString);

        return myData;

    }

}

//==================================

 package je.gettingStarted;

import com.sleepycat.bind.tuple.TupleBinding;

import com.sleepycat.je.DatabaseEntry;

...

TupleBinding keyBinding = new MyTupleBinding();

MyData2 theKeyData = new MyData2();

theKeyData.setLong(123456789l);

theKeyData.setDouble(new Double(12345.6789));

theKeyData.setString("My key data");

DatabaseEntry myKey = new DatabaseEntry();

try {

    // Store theKeyData in the DatabaseEntry

    keyBinding.objectToEntry(theKeyData, myKey);

    ...

    // Database put and get activity omitted for clarity

    ...

    // Retrieve the key data

    theKeyData = (MyData2) keyBinding.entryToObject(myKey);

} catch (Exception e) {

    // Exception handling goes here

}

\*/

//比较器

/\*

package je.gettingStarted;

import java.util.Comparator;

public class MyDataComparator implements Comparator {

    public MyDataComparator() {}

    public int compare(Object d1, Object d2) {

        byte[] b1 = (byte[])d1;

        byte[] b2 = (byte[])d2;

        String s1 = new String(b1, "UTF-8");

        String s2 = new String(b2, "UTF-8");

        return s1.compareTo(s2);

    }

}

使用示例

package je.gettingStarted;

import com.sleepycat.je.Database;

import com.sleepycat.je.DatabaseConfig;

import com.sleepycat.je.DatabaseException;

import java.util.Comparator;

...

// Environment open omitted for brevity

try {

    // Get the database configuration object

    DatabaseConfig myDbConfig = new DatabaseConfig();

    myDbConfig.setAllowCreate(true);

    // Set the duplicate comparator class

    myDbConfig.setDuplicateComparator(MyDataComparator.class);

    // Open the database that you will use to store your data

    myDbConfig.setSortedDuplicates(true);

    Database myDatabase = myDbEnv.openDatabase(null, "myDb", myDbConfig);

} catch (DatabaseException dbe) {

    // Exception handling goes here

}

 \*/

本篇是在学习BDB JE的使用、阅读源码并参读文档后的笔记，涉及到内部实现细节，写的比较零散，也有很多疏漏，仅作为笔记待查。

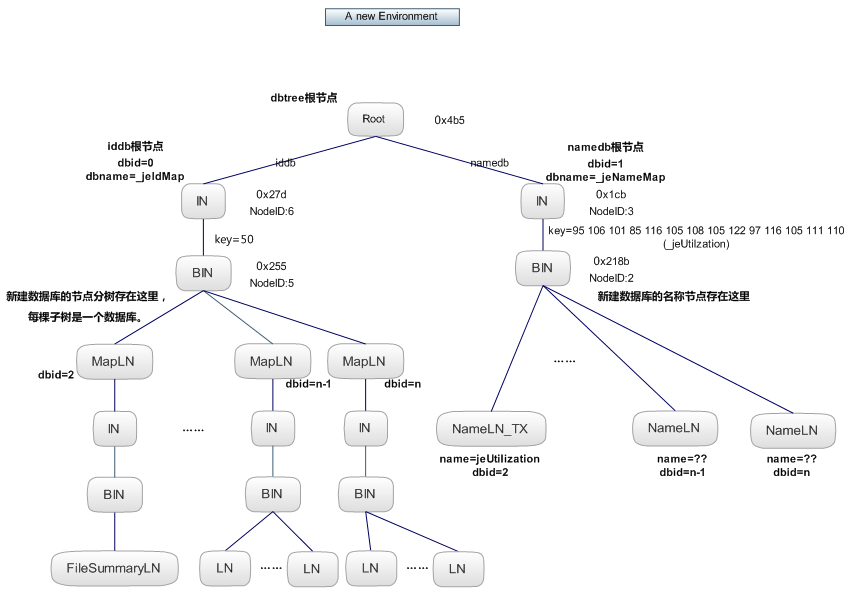
**Berkeley DB JE简介及主要特点**

Java版Berkeley DB，嵌入式数据库引擎，K/V存储，可提供高可用性（HA）。

 日志结构存储（LFS），采用B+树存储节点，Evictor进行内存回收，Cleaner释放磁盘空间。

**树的基本结构**

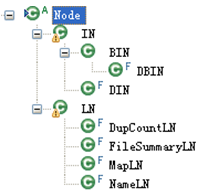
每个数据库环境在内存中会建立一棵DbTree。DbTree的结构如下所示。



dbid从2开始分配，0和1被作为保留的dbid序号分配给了iddb和namedb。分别建立iddb和namedb。Namedb中存储数据库名信息databaseName与数据库id作为NameLN，iddb中存储每个db对应的mapLN，mapLN下面存储这个db的数据tree。

内存中的逻辑结构主要通过com.sleepycat.je.tree里的类来实现。

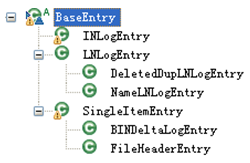
其中，主要的逻辑结构为节点Node，它实现了Loggable接口，这个接口定义了一组将Node的信息写入ByteBuffer的方法。Node分别对应于Tree中的各个逻辑节点，的层次关系如下：



物理存储即日志读写主要通过包com.sleepycat.je.log里的类来实现。

写日志主要用到LogManager的几个log方法，最终体现为调用其内部方法serialLogInternal，里面调用FileManagerd的writeLogBuffer方法。实现将内存中的数据写入文件的操作。

其中，把树中的节点变为日志的中间结构为Entry。在包中定义了3个接口INContainingEntry，LogEntry，NodeLogEntry（继承LogEntry），一共有7个不同的Entry类，根据功能实现不同的接口。Entry的层次关系如下：



在类LogEntryType中定义并初始化了30种不同的LogEntry类型，每个LogEntry包装的logClass类型都实现了Loggable接口。

每个LogEntry再经过包装成为LogItem，进行日志写入。

**日志文件**

数据文件类型在类FileManager中定义，如下所示：

[复制代码](javascript:void(0);)

    public static final String JE\_SUFFIX = ".jdb";  // regular log files  
    public static final String DEL\_SUFFIX = ".del";  // cleaned files  
    public static final String BAD\_SUFFIX = ".bad";  // corrupt files  
    private static final String LOCK\_FILE = "je.lck";// lock file  
  
    /\*  
     \* The suffix used to denote a file that is in the process of being  
     \* transferred during a network backup. The file may not have been  
     \* completely transferred, or its digest verified.  
     \*/  
    public static final String TMP\_SUFFIX = ".tmp";  
    /\*  
     \* The suffix used to rename files out of the way, if they are being  
     \* retained during a backup. Note that the suffix is used in conjunction  
     \* with a backup number as described in <code>NetworkBackup</code>  
     \*/  
     public static final String BUP\_SUFFIX = ".bup";

[复制代码](javascript:void(0);)

非数据日志的文件：EnvironmentImpl.java中定义了INFO文件je.info，LoggerUtils.java将程序日志（区别于dbd中的数据日志）写到这些文件里面：

private static final String INFO\_FILES = "je.info";

**节点回收部分（Evictor）**

这部分做的是内存回收的工作，主要目的是判断内存空间是否达到阈值，若达到阈值则应用策略回收cache中的节点以释放足够的空间。

节点回收的三部分组件：

″ 仲裁（Arbiter），查询内存预算空间判断是否需要回收。

″ 目标选择器（TargetSelector），选择目标节点。

″ 回收器（Evictor），执行节点回收。

节点回收的三种线程：

1. The application thread, in the course of doing critical eviction （执行）

2. Daemon threads, such as the cleaner or INCompressor, in the course of doing their respective duties （后台守护线程）

3. Eviction pool threads （线程池）

节点回收分为两种类型PrivateEvictor和SharedEvictor，分别对应单/多Environment的情况，但是逻辑上是一样的。

内存预算通过com.sleepycat.je.dbi.MemoryBudget实现。

Cache通过com.sleepycat.je.dbi.INList实现。

LRU策略。选择IN时的迭代顺序没有特别定义。◊脏数据◊目标选择器通过方法selectIN在Cache中取出IN，若使用LRU策略，则挑出最低generation的节点，否则策略优先级由高到低选择：最低tree level

对选出的IN，回收器使用方法evictRoot或evictIN回收内存。若IN是一个tree的root，若它的所有子节点都可以被回收，则它也可以被回收，若它是dirty，则需要写入磁盘。对非root的IN，找到其父节点，如果IN是dirty的，则写入磁盘并更新父节点的指针。其中，若IN是BIN，还要先做剪枝，然后将其引用的叶节点释放掉，若其中有数据需要写入磁盘，则将该BIN标记为dirty。

以下几种节点不能被回收：Mapping Tree的节点，内存中正在使用的节点的IN父节点，有游标指向的BIN节点。

**Cleaner**

Cleaner用来释放磁盘空间。

Cleaner的两个目标：

″ 使migration 的开销尽可能小

″ Migration操作与用户的其它线程同时来做，避免clean跟不上的情况

JE的树结构中存储了系统数据库\_jeUtilization，存储每个文件的利用率信息。

使用lazy migration和proactive migration策略移动节点。Lazy migration是说对LN不是立即移动，而是作以标记在下一次ckpt或者evictor的时候在写BIN时一起flush到日志里面，好处是在写一个BIN时可以尽可能多的将它引用的LN一次写入。

Cleaner做几件事情：1. 标记obsolete；2. 移动entry；3. 删除log文件。

日志中entry可能被obsolete的原因：1. 应用删除/更新了entry；2. Proactive migration主动移动了entry；3. trackDetail被置为false。

待删除文件的几种状态（为了保证文件删除时所有有效内容都被移动）：

″ TO\_BE\_CLEANED，FlieSelector选出的待clean的文件被标记为该状态

″ BEING\_CLEANED，FileProcessor选择处理的文件被标记为该状态

″ CLEANED，文件中所有entry被扫描和处理后文件被标记为该状态，这时entry有些未移动有些在pending队列，db也在pending队列

″ CHECKPOINTED，文件在一个ckpt开始时处于CLEANED状态，则在结束时会被标记为该状态，这时被标记为migrate的该移动的都移动了，但是pending的entry和db还没有处理好

″ FULLY\_PROCESSED，当pending队列处理完时文件被标记为该状态，但此时LN的BIN还没有被重写

″ SAFE\_TO\_DELETE，文件在一个ckpt开始时处于SAFE\_TO\_DELETE状态，则在结束时会被标记为该状态，这个时候就可以删除了

 几个记录文件利用率的类：

      类UtilizationProfile，记录所有log文件的利用率。

类PackedOffsets，记录一组LSN的偏移量，在summary里用来标记废弃的entrys。

类FileSummary，记录每个log文件的利用率信息，在UtilizationProfile里使用。（summary）

类TrackedFileSummary，记录每个tracked的log文件的增量信息，对应一个FileSummary，并存储obsolete的entry的偏移量列表。（detail）

类DbFileSummary，记录每个db的每个文件的利用率信息，在相应的DatabaseImpl里使用。

类DbFileSummaryMap，文件fileNum到DbFileSummary的映射。

    类LNInfo，记录被挂起或加锁的需要之后做migrate的叶节点，在FileProcessor里使用。

类FileProcessor执行清理操作，读取文件中所有entry，并决定entry是obsolete还是migration。LN的migration通过设置BIN实现，IN的migration通过标记dirty实现。简化流程如下：

1.        处理挂起的LN。

2.        选择文件（类FileSelector，按利用率最低文件最老的规则选择）。

3.        处理文件（对LN的处理-启用lazy migration时就标记它为migrate同时标记其bin为dirty，禁用lazy migration时就立即加锁执行migrate，若加锁失败则挂起；对IN的处理，标记为dirty。对每个要处理的LN和IN，都会再做一次obsolete验证）。

4.        如果还有待处理的文件，继续步骤1.

|  |
| --- |
|  |

**小结**

1. evictor和cleaner在处理时主要解决的问题：

″ 节点/文件选择策略

″ 内存树结构和日志系统的数据一致性

2. 期望达到的目的：

″ 内存中查找的命中率高

″ 磁盘的有效entry比例大