China-pub.com
下载

第6章

同步问题

- 同步问题
- 数据讹误
- 死锁
- 资源缺乏
- 同步方法

在多个进程对共享数据结构进行处理时，如果不对数据保持适当的同步，将会出问题。

本章将介绍一些处理同步问题的技术。同步问题的根本原因可归结于共享或处理数据的进程之间动作不协调。

下面将介绍几个技术，它们说明了能够在 JavaSpaces环境中协调分布式进程的方法。

6.1 同步问题的类型

不进行同步访问可能会产生的问题有下列几种类型：

• 数据讹误。

• 进程死锁。

• 进程缺乏资源。

我们在本节中简略地介绍这些问题的含义和原因。下一节才介绍处理这些问题的方法。

China-pub.com 下载

6.1.1 数据讹误

两个或多个修改数据的进程互相不了解是数据讹误的根本原因。这种修改导致最终的数据处于一种混杂的状态，其值来自各个进程。图6-1示出了两个进程使一个共享结构产生混乱的例子。

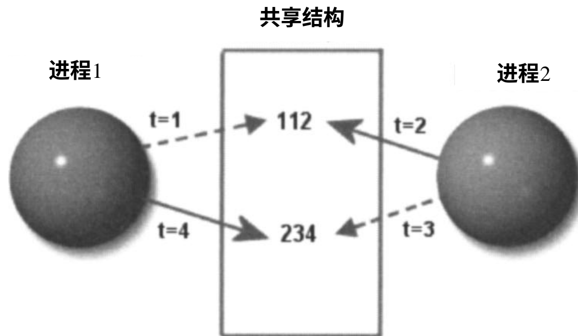

图6-1 混合更新导致的数据讹误

在时间t=1时，进程1将值456写到共享结构中。在时间t=2时，进程2将值112写入共享结构并覆盖了456。在时间t=3时，进程2写入值721作为结构的第二个属性。而这个值又在时间 t=4时被进程1将234作为结构的第二个属性写入所覆盖。在时间 t=5时，此结构保存了混合的值。进程 1希望值456和234在结构中。进程2希望值112和721在结构中。而结果状态既不是进程 1想要的也不是进程2想要的。

6.1.2 死锁问题

两个或多个争用资源的进程可能产生死锁问题。在死锁状态下，互相竞争的一组进程被阻塞而不能完成它们的任务。组中每个进程都在等待获得组中其他进程所占据（且没有释放）的资源。

图6-2示出死锁中所涉及的两个进程。进程 1占据资源A并等待获得资源B。进程2占据资源B并等待获得资源A。

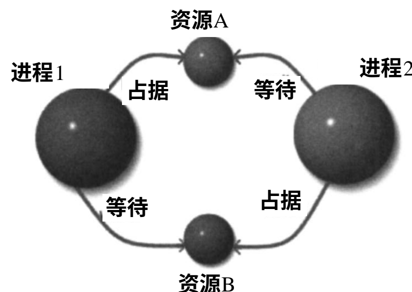

图6-2 死锁进程

China-pub.com
下载

因为每个进程都在等待其他进程所占据的资源，所以它们将会永远等待下去。

必须具备下列四个条件才可能出现死锁，它们分别是：

- 互斥：进程独自地占有资源。
- 非抢占（non-preemption）：在一个进程获得某个资源后，其他进程不能夺走此资源。
- 资源等待：在一个进程请求某个资源时，它将等待直到该资源可用。
- 部分分配：为完成任务，一个进程能够获得所需的某些资源，同时试图获得完成任务所需的其余资源。

在JavaSpaces的使用中，这些情况都可能发生。下面列出使用 JavaSpaces时可能出现这些情况的例子。

- 在一个进程进行take操作时，其他进程不能利用相应的项 —— 互斥。
- 从某个进程取回一个项的自动方法不存在 —— 非抢占。
- 如果一个进程对某个项执行 read，它将阻塞直到该项出现或者其租用期满。很长的租用期基本上可以说是无限的 —— 资源等待。
- 每次只能取一个项 —— 部分分配。

所有这些情况都可能发生，但好消息是有办法处理它们。还有更好的消息，如果能够保证这些条件至少有一个不发生，就不会发生死锁。

防止出现简单死锁的一种简捷办法似乎是对阻塞 JavaSpace方法使用合理的超时值。这将会防止调用者在相应操作上永久阻塞。但必须小心。如果你在没有获得试图访问的项时不能前进，事实上还是被阻塞了。不过，这是一种处理资源等待的机制。

事务处理（正如将要看到的那样）提供了一种有效地把项取回的办法。请考虑一下怎样完成此项工作。（别担心，后面会介绍的。）

6.1.3 资源缺乏问题

在一个进程等待某个重复可用的资源，但此资源永远不分配给等待进程时，出现资源缺乏问题。一般，这个问题由不公平的资源调度算法引起。

6.2 同步方法

在建立应用程序时，除了要处理与同步分布进程和资源有关的特定问题外，还可以使用大量的基础机制。这些机制在处理同步问题时可能会相当方便。

此为试读，需要完整PDF请访问：www.ertongbook.com

China-pub.com 下载

6.2.1 基本JavaSpace机制

JavaSpaces技术本身可以处理第一层次的同步问题。可以保证单项的更新（ take或write）是原子的。只有单个进程能够take空间中给定的项。如果在空间中有一个匹配项，两个或多个进程试图取该项，则只有一个进程能够取得该项。所有其他进程都将取得空。如果 n个进程试图写一个具有相同数据的项到空间中，则在空间中将出现该项的 n个拷贝。

同样，在读一个项时，总会接收到一个完整的项。你永远不会取回一个只有部分内容写入空间的项。这些机制的存在表示不必为个别项的讹误而担心。但是，正如在第 5章中所看到的那样，许多有用的分布式结构都是由多个项组成的。需要更为复杂的措施来防止多个项的数据结构讹误。

6.2.2 空间初始化

经常提出的一个问题是怎样把某样东西的第一个实例放入一个空间中。为了说明这里所涉及的问题，我们来仔细地考察一下 ID的生成。

可回忆一下，第5章中介绍过EntryID。在第5章中，利用了它的一种极为简单的实现。基本上，客户机只是取得一个串，这个串就是用来区别各个元素的 ID。

很显然，这不是生成惟一 ID的一种非常可靠的方法。程序清单 6-1中给出了Stamp（时间戳）类。这个类实现接口EntryID并可以用来建立一系列惟一的ID值。

Stamp类在几个方面很有意思。请注意， Stamp本身并不实现Entry。Stamp打算放在项内部。这表示Stamp需要可串行化，否则无需项类规则就可以实现。例如，它可以具有私有数据和由原始数据构成的数据。

程序清单6-1 Stamp.java

```
package jsbe.ch6;
import jsbe.util.EntryID;
import java.security.SecureRandom;
import java.net.InetAddress;

public class Stamp implements jsbe.util.EntryID {

    private static SecureRandom secRand = new SecureRandom();
    private static long longAddr = getAddress();
    private static final long RANDOMMASK  = 0xFFFFFFFF00000000L;
    private static final long LOWMASK     = 0x00000000FFFFFFFFL;
    private static final long ADDRMASK    = 0x00000000FFFFFFFFL;

    private long counter;
    private long high;
    private long low;
```

China-pub.com
下载


```
    // Default constructor
    public Stamp() {
    }

    public Stamp(long value) {
        set(value);
    }

    // copy constructor
    public Stamp(Stamp inStamp) {
        counter = inStamp.counter;
        high = inStamp.high;
        low = inStamp.low;
    }
```

构造Stamp实例的正常方法是调用取一个 long值作为其参数的构造函数。这个 long值给出用来生成惟一ID值流的一个计数器（counter）的初始值。

在静态初始化时（在装载 Stamp类时），建立两个数据。建立 SecureRandom的一个实例。SecureRandom是随Java技术一起提供的一个密码含义很浓的伪随机数生成器。这表示它所生成的数据看上去很无规律，不可预测（通过大量的随机统计测试）。换句话说，它是一个相当好的随机数生成器。

初始化的另一个静态数据是一个 long属性，longaddr。longaddr是Java虚拟机运行的机器的InetAddress的位表示。

程序清单6-1　Stamp.java（续1）

```
public void set(long value) {

    counter = value;

    byte[] secRandBuf = new byte[8];
    secRand.nextBytes(secRandBuf);

    // set the high field
    high = System.currentTimeMillis();

    // set the low field
    secRand.nextBytes(secRandBuf);

    low = 0;
    for (int i = 0; i < 8; i++) {
        low = (low << 8) | (secRandBuf[i] & 0xff);
    }

    low &= RANDOMMASK;
    low |= longAddr;
}
```

China-pub.com 下载

set方法设置counter属性为传入的值并把一个时间戳设置到 high属性中。然后用两个数据的组合设置low属性。用SecureRandom生成一个随机数，把此随机数与机器地址结合。一个时间戳、一个随机数和当前机器的地址共同提供了惟一 ID的良好的起点。

程序清单6-1 Stamp.java（续2）

```
    public void advance() {
            counter++;
            if (counter < 0) { // rollover
                set(0);
            }
        }

        public long getCount() {
            return counter;
        }

        public long getHigh() {
            return high;
        }

        public long getLow() {
            return low;
        }
```

advance方法用来对counter增量。如果advance检测到counter已经翻转过来，它将调用 set(0)重新初始化其他时间戳值并设置计数器为 0。

程序清单6-1 Stamp.java（续3）

```
        private static long getAddress() {
        byte[] inetAddress;
        long longAddr=0;

        try {
            inetAddress = InetAddress.getLocalHost().getAddress();
        } catch (java.net.UnknownHostException e) {
            inetAddress = new byte[]{0, 0, 0, 0};
        }

        // convert the byte array into a long
        for (int i = 0; i < 4; i++) {
            longAddr = (longAddr << 8) | (inetAddress[i] & 0xff);
        }

        longAddr = longAddr & ADDRMASK;
        return longAddr;
    }
}
```

getAddress方法找到本地机器的地址，然后把它填入一个长整数（ long）。

China-pub.com 下载

程序清单6-2给出StamEntry类。此类存放两个信息。这两个信息一个是时间戳，另一个是它自己的随机位数组。

程序清单6-2　StampEntry.java

```
package jsbe.ch6;
import net.jini.core.entry.Entry;
import jsbe.util.EntryID;

public class StampEntry implements Entry {

    public Stamp stamp;
    public byte[] randombits;

    // Default constructor
    public StampEntry() {
    }

    public StampEntry(Stamp stamp, byte[] randombits) {
        this.stamp = stamp;
        this.randombits = randombits;
    }
}
```

程序清单6-3中的StampGateEntry类是另一个项类。它有一个与StampEntry实例中的随机位相匹配的随机位数组，并且还提供了一个可用来获得查找自己对应的StampEntry实例的模板的辅助方法。

程序清单6-3　StampGateEntry.java

```
package jsbe.ch6;
import net.jini.core.entry.Entry;
import jsbe.util.EntryID;

public class StampGateEntry implements Entry {

    public byte[] randombits;

    // Default constructor
    public StampGateEntry() {
    }

    public StampGateEntry(byte[] randombits) {
        this.randombits = randombits;
    }

    public StampEntry getStampEntryTemplate() {
        return new StampEntry(null, randombits);
    }
}
```

此为试读，需要完整PDF请访问：www.ertongbook.com

China-pub.com 下载

程序清单6-4中的Stamper类提供了获得时间戳的驱动逻辑。

程序清单6-4　Stamper.java

```
import net.jini.core.lease.Lease;
import jsbe.util.SpaceUtil;

public class Stamper {

    private static final long TRANSACTIONDURATION = 1000 * 30;

    private static StampGateEntry sgTmpl = new StampGateEntry();
    private static SecureRandom secRand = new SecureRandom();
    private static TransactionManager tm =
        SpaceUtil.findTransactionManager();

    private JavaSpace space;
    private StampEntry seTmpl;

    public Stamper(JavaSpace space) throws WrappedException {
        this.space = space;
        try {
            StampGateEntry sge =
                (StampGateEntry)space.read(sgTmpl,
                                           null,
                                           Long.MAX_VALUE);
            seTmpl = sge.getStampEntryTemplate();
        } catch (Exception e) {
            throw new WrappedException(e);
        }
    }
```

在静态初始化时，Stamper建立一个ScureRandom实例和一个用作模板的空值StampGateEntry。

在Stamper的构造函数中，试图从所提供的空间中读一个 StampGateEntry。在此构造函数运行时，应该有这样一个实例。在看了程序清单 6-5中的StampInit类时，就会明白为什么要这样了。在Stamper构造函数中，也可以看到一种可供替换的异常处理样式。因为 Stamper不能真正地根据个别的异常情况做什么事情，它只是捕获所有异常，把它们包装在另一个异常中，然后对客户机产生此异常。这样给客户机提供了处理包装起来的异常的机会，而不仅仅是取消它。

程序清单6-4　Stamper.java（续1）

```
public Stamp getStamp() throws WrappedException {

    Transaction tran = null;

    try {
        Transaction.Created tc =
            TransactionFactory.create(tm, TRANSACTIONDURATION);
```

China-pub.com
下载


```
            tran = tc.transaction;
        } catch (Exception e) {

            throw new WrappedException(e);
        }

        Stamp stamp = null;

        try {
            StampEntry se = (StampEntry)space.take(seTmpl,
                                                   tran,
                                                   Long.MAX_VALUE);

            stamp = new Stamp(se.stamp);
            se.stamp.advance();

            space.write(se, tran, Lease.FOREVER);
            tran.commit();
        } catch (Exception e) {
            try {
                tran.abort();
            } catch (Exception ea) {
            }

            throw new WrappedException(e);
        }

        return stamp;
    }
```

getStamp方法从空间中取时间戳项，对所取出的时间戳进行拷贝，增加原时间戳计数器，然后把它写回空间。它在一个事务处理的范围内做所有这些事情。此事务处理的租用值设置为30秒，因此，如果此过程中的某项操作所花时间超过30秒，事务处理将会超时，空间中时间戳的原值将不改变。

如果一切正常，将时间戳的拷贝送回调用者。调用者现在就得到一个惟一的ID了。

程序清单6-4 Stamper.java（续2）

```
    public static StampGateEntry createGate(JavaSpace space)
        throws WrappedException {

            byte[] secRandBuf = new byte[16];
            secRand.nextBytes(secRandBuf);

            StampGateEntry sg = new StampGateEntry(secRandBuf);

            StampEntry se = new StampEntry(new Stamp(0), secRandBuf);

            Transaction tran = null;
```

China-pub.com 下载

```
            try {
                Transaction.Created tc =
                    TransactionFactory.create(tm,
                      TRANSACTIONDURATION);

                tran = tc.transaction;
            } catch (Exception e) {

                throw new WrappedException(e);
            }

            try {
                space.write(se, tran, Lease.FOREVER);
                space.write(sg, tran, Lease.FOREVER);
                tran.commit();
            } catch (Exception e) {
                try {
                    tran.abort();
                } catch (Exception ea) {
                }
                throw new WrappedException(e);
            }

            return sg;
        }
    }
```

Stamper的静态createGate方法用来在空间中建立StampGateEntry和StampEntry对。它以一种相当直观的方式完成这项工作。不寻常的是，只打算在 StampInit类中调用它。

程序清单6-5中的StampInit类提供了怎样放置一个初始值到空间中的答案。将想要初始化的空间名传递给StampInit，然后StampInit再调用creatGate方法。需要保证完成下列两件事情：

• 必须在任何人使用空间前调用 StampInit初始化空间。

• 对一个给定的空间必须只运行 StampInit一次。

因此，怎样把一个初始值放入一个空间的不寻常的答案就是，你必须人工完成它。不存在执行初始指令的基本机制。

程序清单6-5 StampInit.java

```
package jsbe.ch6;

import net.jini.space.JavaSpace;
import jsbe.util.SpaceUtil;

public class StampInit {
    public static void main(String[] args) {
        try {
```

China-pub.com 下载


```
            String spaceName = "JavaSpaces";

            if (args.length > 0) {
                spaceName = args[0];
            }

            JavaSpace space = SpaceUtil.findSpace(spaceName);

            Stamper.createGate(space);

        } catch (WrappedException ce) {
            Exception wrappedException = ce.getWrappedException();
            ce.printStackTrace();
        }
    }
}
```

程序清单6-6中的WrappedException类提供了容纳一般异常的一个简单的包装辅助类。

程序清单6-6 WrappedException.java

```
package jsbe.ch6;

public class WrappedException extends Exception {

    private Exception wrappedException;

    public WrappedException(Exception exIn) {
        wrappedException = exIn;
    }

    public Exception getWrappedException() {
        return wrappedException;
    }
}
```

虽然空间初始化的答案可能有点令人感到意外，不过会发现在引入更为自动的方法前在空间中只需建立几个关键的结构。

一个基本的结构是提供惟一的 ID。如果能够保证空间中不出现名称冲突（实际上，空间很不愿意通知你有这样的事情发生），则多个分布式进程的工作可以变得更容易。

6.2.3 信号量

信号量（semaphore）是处理并发性问题的一个经典机制。它们是 Edsger Dijkstra在1968年提出来的。本质上，信号量是一种以结构化的方式管理有限资源的机制。

信号量的一般实现是作为一个共享的整数值。此整数值代表可使用的资源量。这个值通过两个简单的操作来处理。

此为试读，需要完整PDF请访问：www.ertongbook.com

China-pub.com 下载

第一个操作用来尝试获取所代表的资源。它的语义如下：

• 检查此值看它是否大于0。如果此值大于0，对其减1。此测试必须作为单个操作自动执行。将会发现大多数用来建立操作系统信号量的现代处理器都提供这种操作。但是，如果信号量中的值为0，则阻塞调用进程直到此值变为大于0为止。在进程不阻塞时，计数器将相应地减值。

第二个操作释放所代表的资源。它的语义如下：

• 给值增加1。如果有进程正等待此信号量，随机选择一个，让它继续执行。

这两个操作有许多名称。 P和V、上和下以及获得和释放等是其中几个称呼。我喜欢称第一个操作为获得，第二个操作为释放，因为我认为这相当准确地反映了其含义。

在JavaSpaces中，信号量的一个明显的实现是建立一个保存此整数值的共享项。不过，这种办法存在几个问题。正如第 5章中所述，在任何时候使用一个单一的项时，它就有可能成为一个瓶颈源。而且它也会成为未决的，因为如果一个进程在获得信号量后未释放它就死了，则相应的值会变旧。

请注意，此整数值只是用来代表许多资源的。在 JavaSpaces中，可把每个资源表示为空间中一个独立的项。程序清单6-7给出了恰好可以起这种作用的SemaphoreEntry类。它是一个简单的项类，只有一个作为区分元素的ID。如果此信号量代表n个资源，将建立和使用n个SemaphoreEntry实例。

这说明了实现JavaSpaces的并行特性和已经与JavaSpaces技术一起存在的同步的固有机制的技术。

程序清单6-7 SemaphoreEntry.java

```
package jsbe.ch6;
import net.jini.core.entry.Entry;
import jsbe.util.EntryID;

public class SemaphoreEntry implements Entry {

    public EntryID id;
    // Default constructor
    public SemaphoreEntry() {
    }

    // helper constructor
    public SemaphoreEntry(EntryID id) {
        this.id = id;
    }
}
```

程序清单6-8中的DistributedSemaphore类提供了信号量的逻辑控制。

China-pub.com
下载

程序清单6-8　DistributeSemaphore.java

```
Package jsbe.ch6;

import net.jini.space.*;
import net.jini.core.entry.Entry;
import net.jini.core.transaction.server.TransactionManager;
import net.jini.core.transaction.TransactionFactory;
import net.jini.core.transaction.Transaction;
import net.jini.core.transaction.Transaction.Created;
import net.jini.core.lease.Lease;
import jsbe.util.EntryID;
import jsbe.util.SpaceUtil;

public class DistributedSemaphore implements java.io.Serializable{
    private static TransactionManager tm =
        SpaceUtil.findTransactionManager();

    private EntryID id;
    private JavaSpace space;
    private SemaphoreEntry semTmpl;

   // Access an existing semaphore
    public DistributedSemaphore(JavaSpace space,
                                EntryID id) {
        this.id = id;
        this.space = space;
        semTmpl = new SemaphoreEntry(id);
    }

 // create a new semaphore with a unique ID
    public DistributedSemaphore(JavaSpace space, int count)
    throws WrappedException {
        this.space = space;
    Stamper stamper = new Stamper(space);
    id = stamper.getStamp();

    semTmpl = new SemaphoreEntry(id);
    createEntries();
}

// create a new semaphore with the given ID
public DistributedSemaphore(JavaSpace space,
                            int count,
                            EntryID id)
Throws WrappedException {
    this.space = space;
    this.id = id;

    semTmpl = new SemaphoreEntry(id);
    createEntries();
}
```

China-pub.com 下载

```
private void createEntries(int count) throws WrappedException {
        Transaction tran = null;

        try {
            Transaction.Created tc =
                TransactionFactory.create(tm, 1000*20*count);

            tran = tc.transaction;
        } catch (Exception e) {
            throw new WrappedException(e);
        }

        // Create the elements
        try {
            for (int i=0; i< count; i++) {
                space.write(semTmpl, tran, Lease.FOREVER);
            }
            tran.commit();
        } catch (Exception e) {
            throw new WrappedException(e);
        }
    }
```

DistributedSemaphore提供了三个构造函数。第一个构造函数用来访问一个已经存在的信号量并传入空间和要访问的信号量的ID。

第二个构造函数用来建立SemaphoreEntry实例的初始集合。它从Stamper获得一个惟一的ID，然后在空间中建立所需数目的项。

因为此ID值是惟一的，并且是生成的，为了使用此信号量，必须在信号量的用户之间通信。可以用多种方式安排这件工作。目前应该考虑把为什么 DistributedSemaphore是可串行化的作为一个方向性的线索。第7章将介绍某些通信技术。

第三个构造函数使信号量的创建者能够指定ID。然后它也建立所需数目的项。

程序清单6-8 DistributeSemaphore.java（续1）

```
    public EntryID getID() {
        return id;
    }

public Transaction acquire(long transactionDuration)
    throws WrappedException {
        Transaction tran = null;

        try {
            Transaction.Created tc =
                TransactionFactory.create(tm,
                  transactionDuration);
```

China-pub.com 下载


```
            tran = tc.transaction;
            space.take(semTmpl, tran, Long.MAX_VALUE);
        } catch (Exception e) {
            throw new WrappedException(e);
        }
        return tran;
    }
```

acquire方法在空间上对信号量项执行一个take方法。如果当前空间中有项，则take取得成功，且取进程继续执行。如果没有任何项，则取进程将等待直到某个项可用。

这个方法的一个非常重要的方面是它返回一个Transaction实例。此事务处理用来帮助保证毁损的进程不会长久地减少信号量项实例的数目。

acquire方法建立一个具有由调用者传入的长度的租用期的事务处理。然后将此事务处理返回给调用者。

如果一切正常，则将此transaction实例传递到release方法，并且在release结束时提交此事务处理。release还把一个新SemaphoreEntry实例写回空间。这使一个正等待acquire的进程能够继续执行。

如果acquire方法的调用者不能在此事务处理期满前结束，将对空间恢复所取出的Semaphore Entry。如果后面调用者能够调用release，write将会失败，因为此事务处理不再有效。

程序清单6-8 DistributeSemaphore.java（续2）

```
        Public void release(Transaction tran) throws
          WrappedException {
            try {
                space.write(semTmpl, tran, Lease.FOREVER);
                tran.commit();
            } catch (Exception e) {
                throw new WrappedException(e);
            }
        }
    }
```

当拥有信号量时在空间上所进行的所有活动都应该利用相同的事务处理来完成。

6.2.4 乐器店的例子

为了把上述内容结合起来，有必要举一个小例子。想像有一个乐器店，它的业主已经变得很狂躁了。似乎在这个乐器商店中有两个特别奇妙的钹（cymbal）。更奇妙的是，来自四面八方的钹迷们都来这个商店使用它们。不幸的是，他们是那么狂热，他们立即冲到店中抓起了最近的钹。

因为每个拔迷都想同时拥有两个钹，如果一个钹迷抓住一个钹，另一个钹迷抓住另一个钹，

China-pub.com 下载

他们互相都拒绝把自己的钹让与另一人。乐器店老板已经让许多钹迷饿死在店中了，他们都不肯放弃自己持有的钹。

为了解决这个问题，乐器店老板不得不实行一些相当严格的规定：

- 一次只有一个钹迷能够进入商店。
- 在进商店之前，一个钹迷必须答应只呆一小会。
- 在进入后，钹迷可以持有两个钹。
- 在退出时，钹迷必须放弃两个钹。
- 如果某个钹迷呆的时间超过规定时间或拒绝放弃钹，将要从该钹迷的手中把钹夺走。

可以断定，这确实是一个资源分配方面的问题。两个互相死锁的钹迷事实上模拟了图 6-2中的情形。这个问题可以用本章中给出的工具很好地模拟和解决。

在程序清单6-9中可以看到StoreEntry类。这个类在JavaSpace中代表了乐器店。它包含乐器店的名称和一个共享信号量的ID。

程序清单6-9 StoreEntry.java

```
package jsbe.ch6;
import net.jini.core.entry.Entry;
import jsbe.util.EntryID;

public class StoreEntry implements Entry {

    public String storeID;
    public EntryID semaphore;

    // Default constructor
    public StoreEntry() {
    }

    // helper constructor
    public StoreEntry(String id,
                      DistributedSemaphore sem) {
        this.storeID = id;
        if (sem != null) {
            semaphore = sem.getID();
        }
    }
}
```

程序清单6-10包含CymbalEntry类。这个类用来代表钹。它只包含店名。两个奇妙的钹是完全相同的。

China-pub.com
下载

程序清单6-10 CymbalEntry.java

```
package jsbe.ch6;
import net.jini.core.entry.Entry;
import jsbe.util.EntryID;

public class CymbalEntry implements Entry {

    public String storeID;

    // Default constructor
    public CymbalEntry() {
    }

    // helper constructor
    public CymbalEntry(String id) {
        this.storeID = id;
    }
}
```

程序清单6-11是一个包装类，用来给出一个乐器顾客（钹迷）的符号。请注意，除了钹以外，这个类还包含一个Transaction实例。

程序清单6-11 Cymbals.java

```
package jsbe.ch6;

import net.jini.core.transaction.Transaction;

public class Cymbals {

    public CymbalEntry cymbal0;
    public CymbalEntry cymbal1;
    public Transaction tran;

    // Default constructor
    public Cymbals(CymbalEntry c0,
                   CymbalEntry c1,
                   Transaction tran) {
        cymbal0 = c0;
        cymbal1 = c1;

        this.tran = tran;
    }
}
```

程序清单6-12中的MusicStore类提供了乐器店表示的驱动逻辑。它有一个构造函数，此函数取乐器店所在的JavaSpace以及店名。

程序清单6-12 MusicStore.java

```
package jsbe.ch6;
```

此为试读，需要完整PDF请访问：www.ertongbook.com