第七章 显示锁

相对于以前的版本，Java 5.0 引入了新的调节共享对象访问的机制，即重入锁（ReentrantLock）。重入锁可以在内部锁被证明受到局限时，提供可选择的高级特性。它具有与内在锁相同的内存语义、相同的锁定，但在争用条件下却有更好的性能。

同时提供了读写锁，与互斥锁相比，读取数据远大于修改数据的频率时能提升性能。

在第 3 章讲解 JDK 并发 API 时已经介绍过 ReentrantLock，本章做一些提升和补充。

7.1. Lock 和 ReentrantLock

Lock 接口定义了一组抽象的锁定操作。与内部锁定（intrinsic locking）不同，Lock 提供了无条件的、可轮询的、定时的、可中断的锁获取操作，所有加锁和解锁的方法都是显式的。这提供了更加灵活的加锁机制，弥补了内部锁在功能上的一些局限——**不能中断那些正在等待获取锁的线程，并且在请求锁失败的情况下，必须无限等待**。

Lock 接口主要定义了下面的一些方法：

1）void lock()：获取锁。如果锁不可用，出于线程调度目的，将禁用当前线程，并且在获得锁之前，该线程将一直处于休眠状态。

2）void lockInterruptibly() throws InterruptedException：如果当前线程未被中断，则获取锁。如果锁可用，则获取锁，并立即返回。如果当前线程在获取锁时被中断，并且支持对锁获取的中断，则将抛出 InterruptedException，并清除当前线程的已中断状态。

3）boolean tryLock()：如果锁可用，则获取锁，并立即返回值 true。如果锁不可用，则此方法将立即返回值 false。

4）boolean tryLock(long time, TimeUnit unit) throws InterruptedException：如果锁在给定的等待时间内空闲，并且当前线程未被中断，则获取锁。

5）void unlock()：释放锁。

6）Condition newCondition()：返回绑定到此 Lock 实例的新 Condition 实

例。调用 Condition.await() 将在等待前以原子方式释放锁，并在等待返回前重新获取锁。

ReentrantLock 实现了 Lock 接口。获得 ReentrantLock 的锁与进入 synchronized 块具有相同的语义，释放 ReentrantLock 锁与退出 synchronized 块有相同的语义。相比于 synchronized，ReentrantLock 提供了更多的灵活性来处理不可用的锁。下面具体来介绍一下 ReentrantLock 的使用。

1. 实现可轮询的锁请求

在内部锁中，死锁是致命的——唯一的恢复方法是重新启动程序，唯一的预防方法是在构建程序时不要出错。而可轮询的锁获取模式具有更完善的错误恢复机制，可以规避死锁的发生。

如果你不能获得所有需要的锁，那么使用可轮询的获取方式使你能够重新拿到控制权，它会释放你已经获得的这些锁，然后再重新尝试。可轮询的锁获取模式，由 tryLock()方法实现。此方法仅在调用时锁为空闲状态才获取该锁。如果锁可用，则获取锁，并立即返回值 true。如果锁不可用，则此方法将立即返回值 false。此方法的典型使用语句如下：

```
Lock lock = ...;
if (lock.tryLock()) {
    try {
        // manipulate protected state
    } finally {
        lock.unlock();
    }
} else {
    // perform alternative actions
}
```

2. 实现可定时的锁请求

当使用内部锁时，一旦开始请求，锁就不能停止了，所以内部锁给实现具有时限的活动带来了风险。为了解决这一问题，可以使用定时锁。当具有时限的活

动调用了阻塞方法，定时锁能够在时间预算内设定相应的超时。如果活动在期待的时间内没能获得结果，定时锁能使程序提前返回。可定时的锁获取模式，由tryLock(long, TimeUnit)方法实现。

3. 实现可中断的锁获取请求

可中断的锁获取操作允许在可取消的活动中使用。lockInterruptibly()方法能够使你获得锁的时候响应中断。

7.2. 对性能的考察

当 ReentrantLock 被加入到 Java 5.0 中时，它提供的性能要远远优于内部锁。如果有越多的资源花费在锁的管理和调度上，那用留给应用程序的就会越少。更好的实现锁的方法会使用更少的系统调用，发生更少的上下文切换，在共享的内存总线上发起更少的内存同步通信。耗时的操作会占用本应用于程序的资源。Java 6 中使用了经过改善的管理内部锁的算法，类似于 ReentrantLock 使用的算法，从而大大弥补了可伸缩性的不足。因此 ReentrantLock 与内部锁之间的性能差异，会随着 CPU、处理器数量、高速缓存大小、JVM 等因素的发展而改变。

下面具体的构造一个测试程序来具体考察 ReentrantLock 的性能。构造一个计数器 Counter，启动 N 个线程对计数器进行递增操作。显然，这个递增操作需要同步以防止数据冲突和线程干扰，为保证原子性，采用 3 种锁来实现同步，然后查看结果。

测试环境是双核酷睿处理器，内存 3G，JDK6。

第一种是内在锁，第二种是不公平的 ReentrantLock 锁，第三种是公平的 ReentrantLock 锁。

首先定义一个计数器接口。

```
package locks;
public interface Counter {
   public long getValue();
   public void increment();
}
```

下面是使用内在锁的计数器类：

```
package lockbenchmark;
public class SynchronizedCounter implements Counter {
    private long count = 0;
    public long getValue() {
        return count;
    }
    public synchronized void increment() {
        count++;
    }
}
```

下面是使用不公平 ReentrantLock 锁的计数器。

```
package lockbenchmark;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class ReentrantUnfairCounterLockCounter implements Counter {
    private volatile long count = 0;
    private Lock lock;
    public ReentrantUnfairCounterLockCounter() {
        // 使用非公平锁，true就是公平锁
        lock = new ReentrantLock(false);
    }
    public long getValue() {
        return count;
    }
    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }
}
```

下面是使用公平的 ReentrantLock 锁的计数器。

```
package lockbenchmark;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class ReentrantFairLockCounter implements Counter {
```

```
    private volatile long count = 0;
    private Lock lock;
    public ReentrantFairLockCounter() {
        // true就是公平锁
        lock = new ReentrantLock(true);
    }
    public long getValue() {
        return count;
    }
    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }
}
```

下面是总测试程序。

```
package lockbenchmark;
import java.util.concurrent.CyclicBarrier;
public class BenchmarkTest {
    private Counter counter;
    // 为了统一启动线程，这样好计算多线程并发运行的时间
    private CyclicBarrier barrier;
    private int threadNum;// 线程数
    private int loopNum;// 每个线程的循环次数
    private String testName;
    public BenchmarkTest(Counter counter, int threadNum, int loopNum,
            String testName) {
        this.counter = counter;
        barrier = new CyclicBarrier(threadNum + 1); // 关卡计数=线程数
+1
        this.threadNum = threadNum;
        this.loopNum = loopNum;
        this.testName = testName;
    }
    public static void main(String args[]) throws Exception {
        int threadNum = 2000;
        int loopNum = 1000;
        new BenchmarkTest(new SynchronizedCounter(), threadNum,
loopNum, "内部锁")
                .test();
```

```
        new BenchmarkTest(new ReentrantUnfairCounterLockCounter(),
threadNum,
                loopNum, "不公平重入锁").test();
        new BenchmarkTest(new ReentrantFairLockCounter(), threadNum,
loopNum,
                "公平重入锁").test();
    }
    public void test() throws Exception {
        try {
            for (int i = 0; i < threadNum; i++) {
                new TestThread(counter, loopNum).start();
            }
            long start = System.currentTimeMillis();
            barrier.await(); // 等待所有任务线程创建,然后通过关卡，统一执行
所有线程
            barrier.await(); // 等待所有任务计算完成
            long end = System.currentTimeMillis();
            System.out.println(this.testName + " count value:"
                    + counter.getValue());
            System.out.println(this.testName + " 花费时间:" + (end -
start) + "毫秒");
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
    class TestThread extends Thread {
        int loopNum = 100;
        private Counter counter;
        public TestThread(final Counter counter, int loopNum) {
            this.counter = counter;
            this.loopNum = loopNum;
        }
        public void run() {
            try {
                barrier.await();// 等待所有的线程开始
                for (int i = 0; i < this.loopNum; i++)
                    counter.increment();
                barrier.await();// 等待所有的线程完成
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

对三种锁分别设置两个不同的参数：不同线程数和每个线程数的循环次数。最后记录每种锁的运行时间（单位：ms），形成下表。

循环次数 1000

循环次数 1000	线程数 200	线程数 500	线程数 1000	线程数 2000
内在锁	62	313	406	875
非公平锁	31	94	250	859
公平锁	4641	17610	44671	57391

循环次数 200

循环次数 200	线程数 200	线程数 500	线程数 1000	线程数 2000
内在锁	47	94	109	265
非公平锁	16	32	125	906
公平锁	781	3031	8671	13625

分析统计结果，在线程数小于 2000 的情况下，非公平可重入锁的性能要优于内部锁。公平可重入锁的性能最差。同时发现内部锁其实也是一个非公平锁。

7.3 Lock 与 Condition

当使用 synchronized 进行同步时，可以在同步代码块中使用 wait 和 notify 等方法。

在使用显示锁的时候，通过将 Condition 对象与任意 Lock 实现组合使用，为每个对象提供多个等待方法。其中，Lock 替代了 synchronized 方法和语句的使用，Condition 替代了 Object 监视器方法的使用。

条件（Condition，也称为条件队列或条件变量）为线程提供了一个含义，以便在某个状态条件现在可能为 true、另一个线程通知它之前，一直挂起该线程（即让其“等待”）。因为访问此共享状态信息发生在不同的线程中，所以它必须受保

护，因此要将某种形式的锁与该条件相关联。等待提供一个条件的主要属性是：以原子方式释放相关的锁，并挂起当前线程，就像 Object.wait 做的那样。

Condition 实例实质上被绑定到一个锁上。要为特定 Lock 实例获得 Condition 实例，请使用其 newCondition() 方法。

下面使用可重入锁与 Condition 替代 synchronized 实现生产者-消费者模式。

生产者-消费者问题一般是，有一个缓冲区，它支持 put 和 take 方法。如果试图在空的缓冲区上执行 take 操作，则在某一个项变得可用之前，线程将一直阻塞；如果试图在满的缓冲区上执行 put 操作，则在有空间变得可用之前，线程将一直阻塞。可以在单独的等待集合中保存 put 线程和 take 线程，这样就可以在缓冲区中的项或空间变得可用时利用最佳规划，一次只通知一个线程。可以使用两个 Condition 实例来做到这一点。

下面是缓冲区类 LockedBuffer，在这个类的 put 和 take 方法中使用了可重入锁与条件变量：

```
package conditionlock;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class LockedBuffer {
    // 可重入锁
    final Lock lock = new ReentrantLock();
    // 两个条件对象
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();
    // 缓冲区
    final Object[] items = new Object[10];
    int putptr, takeptr, count;// 计数器
    // 放数据操作，生产者调用该方法
    public void put(Object x) throws InterruptedException {
        lock.lock();
        try {
            // 如果缓冲区满了，则线程等待
            while (count == items.length)
                notFull.await();
            items[putptr] = x;
            if (++putptr == items.length)
                putptr = 0;
            ++count;
            // 向消费者线程发送通知
```

```
                notEmpty.signal();
            } finally {
                lock.unlock();
            }
        }
        // 消费者线程调用该方法
        public Object take() throws InterruptedException {
            lock.lock();
            try {
                // 如果缓冲区空，则等待
                while (count == 0)
                    notEmpty.await();
                Object x = items[takeptr];
                if (++takeptr == items.length)
                    takeptr = 0;
                --count;
                // 通知其他生产者线程
                notFull.signal();
                return x;
            } finally {
                lock.unlock();
            }
        }
    }
```

生产者：

```
package conditionlock;
//生产者
class Producer implements Runnable {
    LockedBuffer buffer;
    public Producer(LockedBuffer buf) {
        buffer = buf;
    }
    public void run() {
        char c;
        for (int i = 0; i < 20; i++) {
            c = (char) (Math.random() * 26 + 'A');
            try {
                // 向缓冲区放入数据
                buffer.put(c);
            } catch (InterruptedException e1) {
                e1.printStackTrace();
            }
            System.out.println("Produced: " + c);
```

```
            try {
                Thread.sleep((int) (Math.random() * 100));
            } catch (InterruptedException e) {
            }
        }
    }
}
```

消费者类

```
package conditionlock;
//消费者
class Consumer implements Runnable {
    LockedBuffer buffer;
    public Consumer(LockedBuffer buf) {
        buffer = buf;
    }
    public void run() {
        Object c = null;
        for (int i = 0; i < 20; i++) {
            try {
                // 取数据
                c = buffer.take();
            } catch (InterruptedException e1) {
                e1.printStackTrace();
            }
            System.out.println("Consumed: " + c);
            try {
                Thread.sleep((int) (Math.random() * 1000));
            } catch (InterruptedException e) {
            }
        }
    }
}
```

测试类

```
package conditionlock;
public class LockConditionTest {
    public static void main(String args[]) {
        LockedBuffer stack = new LockedBuffer();
        // 创建生产者，消费者
        int count = 3;
        Producer[] producers = new Producer[count];
        Consumer[] consumers = new Consumer[count];
        for (int i = 0; i < count; i++) {
            producers[i] = new Producer(stack);
```

```
            consumers[i] = new Consumer(stack);
        }
        for (int i = 0; i < count; i++) {
            new Thread(producers[i]).start();
            new Thread(consumers[i]).start();
        }
    }
}
```

程序运行结果如下：

```
Produced: Z
Consumed: Z
Produced: X
Consumed: X
…..
Produced: D
Produced: N
Produced: L
Produced: U
Produced: G
Produced: V
Consumed: Q
Produced: Q
Produced: U
Consumed: M
Produced: I
Consumed: D
….
Consumed: U
Produced: M
Consumed: G
Produced: P
Consumed: V
```

```
Produced: N
Consumed: Q
Produced: J
Consumed: U
Produced: L
……
Produced: Y
Consumed: O
Produced: E
Consumed: M
Produced: I
Consumed: P
```

7.4. 在内部锁和重入锁之间进行选择

重入锁（ReentrantLock）与内部锁在加锁和内存语义上是相同的。从性能上看，重入锁的性能看起来胜过内部锁。在 Java 5.0 中，两者性能之间的差距比较大；而在 Java 6 中，这种差距变得比较小。与重入锁相比，内部锁仍然具有很大的优势，比如内部锁更为人们所熟悉，也更简洁，而且很多现有的程序已经在使用内部锁了。重入锁是很危险的同步工具，程序员在使用重入锁时，容易产生错误。因此，**只有在内部锁不能满足需求，才需要使用重入锁。**

在 Java 5.0 中，内部锁还具有另外一个优点：线程转储能够显示哪些调用框架获得了哪些锁，并能够识别发生了死锁的线程。但 Java 虚拟机并不知道哪个线程持有重入锁，因此在调试使用了重入锁的线程时，无法从中获得帮助信息。这个问题在 Java 6 中得到了解决，它提供了一个管理和调试接口，锁可以使用这个接口进行注册，并通过其他管理和调试接口，从线程转储中得到重入锁的加锁信息。

由于内部锁是内置于 Java 虚拟机中的，它能够进行优化，因此未来的性能改进可能更倾向于内部锁，而不是重入锁。综上所述，除非你的应用程序需要发

布在 Java 5.0 上，或者需要使用重入锁的可伸缩性，否则就应该选择内部锁。

总之，ReentrantLock 锁与 Java 内在锁相比有下面的特点：

1)ReentrantLock 必须在 finally 块中释放锁，而使用 synchronized 同步，JVM 将确保锁会获得自动释放。

2）与目前的 synchronized 实现相比，争用下的 ReentrantLock 实现更具可伸缩性。

3）对于 ReentrantLock ，可以有不止一个条件变量与它关联。

4）允许选择想要一个公平锁，还是一个不公平锁。

5）除非您对 Lock 的某个高级特性有明确的需要，或者有明确的证据表明在特定情况下，同步已经成为可伸缩性的瓶颈，否则还是应当继续使用 synchronized。

6）Lock 类只是普通的类，JVM 不知道具体哪个线程拥有 Lock 对象。而且，几乎每个开发人员都熟悉 synchronized，它可以在 JVM 的所有版本中工作。

7.5. 读-写锁

读-写锁可以提高应用程序的并发度。在很多情况下，数据是“频繁被读取”的——它们是可变的，有的时候会被改变，但多数访问只进行读操作。此时，如果能够允许多个读线程同时访问数据就非常好了。而标准的互斥锁一次最多只允许一个线程能够持有相同的锁，这为保护数据一致性加了很强的约束，因此过分地限制了并发性。互斥是保守的加锁策略，避免了“写/写”和“写/读”的重叠，但是同样避开了“读/读”的重叠。只要每个线程保证能够读到最新的数据，并且在读线程读取数据的时候没有其他线程修改数据，就不会发生问题。读-写锁允许的情况是：一个资源能够被多个读线程访问，或者被一个写线程访问，但两者不能同时进行。读-写锁的定义如表所示。

表 7.1 ReadWriteLock 接口

```
public interface ReadWriteLock{
    Lock readLock();//返回用于读取操作的锁
    Lock writeLock();//返回用于写入操作的锁。
}
```

引入读-写锁是用来进行性能改进的，使得在特定情况下能够有更好的并发性。在实践中，当多处理器系统中频繁的访问主要是读取数据的时候，读-写锁能够改进性能；在其他情况下，运行的情况比互斥锁要稍差一些，这归因于读-写锁更大的复杂性。使用读-写锁究竟能否带来改进，最好通过对系统进行剖析来判断。

读写锁一般可用于缓存设计。

ReadWriteLock 维护了一对相关的锁，一个用于只读操作，另一个用于写入操作。只要没有 writer，读取锁可以由多个 reader 线程同时保持。写入锁是独占的。

在内容管理系统、新闻发布系统等网站的开发设计中，文档的分类（ArticleCategory）一般是极少变化的，并且数据量比较小，读取非常的频繁，可以一次性的把这些文档分类数据从数据库中一次读取出，放入缓存，减少数据库服务器的压力，当数据有变化时，则使缓存失效，然后从新从数据库读取数据。

未使用缓存时，主要包含下面的几个类：

ArticleCategory：表示文档分类的实体类。

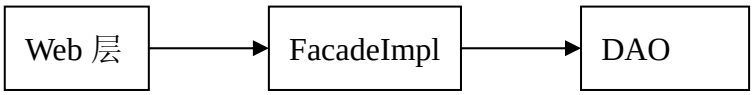
CategoryDao：定义访问数据库操作的接口。

CategoryDaoImpl：具体访问数据库操作的类，实现了 CategoryDao 接口。

Façade：定义了可以使用的业务方法的接口。

FacadeImpl：实现了 Façade 接口中的方法。

基本工作流程是：客户程序调用 Façade 中定义的业务方法 a，业务方法 a 如果需要访问数据库，调用 CategoryDao 中定义的访问数据库的方法，DAO 中定义的是操作 ArticleCategory 实体的方法。



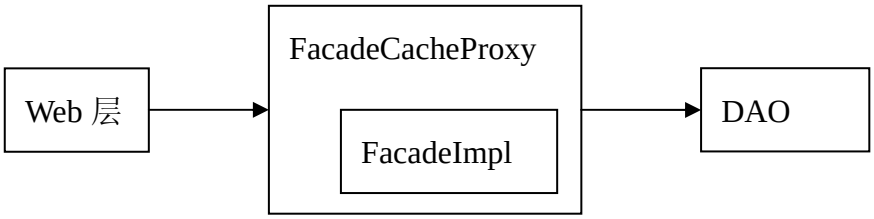
在原有系统基础上进行改造，增加对缓存的支持：

FullCache：缓存类，管理缓存数据。

FacadeCacheProxy：实现了 Façade 接口，其方法的实现又委托给 FacadeImpl 去完成。实现了代理设计模式。

FullCacheTest：缓存程序测试类。

因为 FacadeCacheProxy 也实现了 Façade 接口，使用缓存后，客户调用业务方法的代码无需改变。这样客户程序无需修改。FacadeCacheProxy 中关于读取文档分类的方法直接从缓存读取，执行其他需要更新数据库的方法时，使缓存失效，从新读取数据库更新后的数据填充缓存。



下面是主要类的代码，详细程序请参考光盘上的代码。

下面是 DAO 类，具体负责访问数据库的操作，使用了 Thread.sleep(1)模拟操作数据库的时延。

```
package lockcache;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
//DAO 实现类负责具体的访问数据库操作
public class CategoryDaoImpl implements CategoryDao {
    // 这里用内存的一个 HashMap 对象模拟数据库
    private static Map db = new HashMap();
    @Override
    public void create(ArticleCategory category) {
        // 暂停一毫秒模拟数据库的访问时间
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        db.put(category.getId(), category);
    }
    @Override
    public List queryArticleCategories() {
        System.out.println("从数据库读取数据！");
```

```
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return new ArrayList(db.values());
        }
        @Override
        public ArticleCategory queryArticleCategory(Serializable id) {
            if (db.containsKey((id))) {
                try {
                    Thread.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                return (ArticleCategory) db.get(id);
            }
            return null;
        }
        @Override
        public void update(ArticleCategory category) {
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            db.put(category.getId(), category);
        }
    }
```

管理缓存的类。

```
    package lockcache;
    import java.util.List;
    import java.util.concurrent.locks.Lock;
    import java.util.concurrent.locks.ReadWriteLock;
    import java.util.concurrent.locks.ReentrantReadWriteLock;
    //管理缓存的类
    public abstract class FullCache {
        // 读写锁
        private final ReadWriteLock lock = new ReentrantReadWriteLock();
        private final Lock readLock = lock.readLock(); // 读锁
        private final Lock writeLock = lock.writeLock(); // 写锁
        private List cachedList = null; // 持有缓存的数据，若为 null，表示缓存失
效
        public List getCachedList() {
```

```
            // 获得读锁:
            readLock.lock();
            System.out.println("读取缓存数据！");
            try {
                if (cachedList == null) {
                    // 在获得写锁前，必须先释放读锁:
                    readLock.unlock();
                    writeLock.lock();
                    try {
                        System.out.println("重新填充缓存数据！");
                        cachedList = doGetList(); // 获取真正的数据
                    } finally {
                        // 在释放写锁前，先获得读锁:
                        readLock.lock();
                        writeLock.unlock();
                    }
                }
                return cachedList;
            } finally {
                // 确保读锁在方法返回前被释放:
                readLock.unlock();
            }
        }
        // 子类重写该法，实现具体的获取数据填充缓存的方式
        abstract protected List doGetList();
        public void clearCache() {
            writeLock.lock();
            cachedList = null;
            System.out.println("缓存失效！");
            writeLock.unlock();
        }
    }
```

代理类

```
    package lockcache;
    import java.io.Serializable;
    import java.util.List;
    public class FacadeCacheProxy implements Facade {
        private Facade target;
        public void setFacadeTarget(Facade target) {
            this.target = target;
        }
        private FullCache cache = new FullCache() {
```

```
            // 实现了父类中定义的填充缓存数据的方法
            protected List doGetList() {
                return target.queryArticleCategories();
            }
        };
        public List queryArticleCategories() {
            return cache.getCachedList();
        }
        public void createArticleCategory(ArticleCategory category) {
            target.createArticleCategory(category);
            cache.clearCache();
        }
        public void updateArticleCategory(ArticleCategory category) {
        }
        @Override
        public ArticleCategory queryArticleCategory(Serializable id) {
            return null;
        }
        public void setCategoryDao(CategoryDao categoryDao) {
        }
    }
```

下面是总的缓存测试程序，定义了读取线程和写线程，其中读取频率要远大于写的频率。

```
package lockcache;
//测试缓存的类
public class FullCacheTest {
    // 定义Facade的变量，便于调用业务方法
    static Facade facade = new FacadeImpl();
    // 实现了缓存的Facade
    static Facade proxy = new FacadeCacheProxy();
    static CategoryDao dao = new CategoryDaoImpl();
    public static void main(String[] args) {
        // 设置需要的DAO
        facade.setCategoryDao(dao);
        // 把业务功能委托给FacadeImpl类
        ((FacadeCacheProxy) proxy).setFacadeTarget(facade);
        // 创建更新分类的线程
        Thread t1 = new Thread(new Updater());
        t1.start();
        // 创建读取分类数据的线程
        for (int i = 0; i < 5; i++) {
            new Thread(new Querier(), "t+i").start();
        }
```

```
    }
    // 更新数据的线程体
    static class Updater implements Runnable {
        @Override
        public void run() {
            for (;;) {
                ArticleCategory category = new ArticleCategory();
                double d = Math.random();
                category.setId("" + (int) (d * 10));
                category.setName("name" + d);
                // 创建一个
                proxy.createArticleCategory(category);
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
    // 读取数据的线程体
    static class Querier implements Runnable {
        @Override
        public void run() {
            for (;;) {
                // 打印读取的数据
                System.out.println(proxy.queryArticleCategories());
                try {
                    Thread.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

下面是程序运行结果的片段：

```
.....
读取缓存数据！
[3/name0.32385907946157355,                    2/name0.21873239178259907,
1/name0.10581430507095557,                     0/name0.048790763893907685,
```

7/name0.7092783397751578, 6/name0.6450949775582443, 5/name0.5813937994250811, 4/name0.44179702332968607, 9/name0.9571487587444742, 8/name0.8377456665152162]

读取缓存数据!

[3/name0.32385907946157355, 2/name0.21873239178259907, 1/name0.10581430507095557, 0/name0.048790763893907685, 7/name0.7092783397751578, 6/name0.6450949775582443, 5/name0.5813937994250811, 4/name0.44179702332968607, 9/name0.9571487587444742, 8/name0.8377456665152162]

缓存失效!

读取缓存数据!

重新填充缓存数据!

从数据库读取数据!

[3/name0.32385907946157355, 2/name0.21873239178259907, 1/name0.16459410374370664, 0/name0.048790763893907685, 7/name0.7092783397751578, 6/name0.6450949775582443, 5/name0.5813937994250811, 4/name0.44179702332968607, 9/name0.9571487587444742, 8/name0.8377456665152162]

读取缓存数据!

[3/name0.32385907946157355, 2/name0.21873239178259907, 1/name0.16459410374370664, 0/name0.048790763893907685, 7/name0.7092783397751578, 6/name0.6450949775582443, 5/name0.5813937994250811, 4/name0.44179702332968607, 9/name0.9571487587444742, 8/name0.8377456665152162]

……

Linux公社（LinuxIDC.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

LinuxIDC.com提供包括Ubuntu，Fedora，SUSE技术，以及最新IT资讯等Linux专业类网站。

并被收录到Google 网页目录-计算机 > 软件 > 操作系统 > Linux 目录下。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

包括:

Ubuntu专题

Fedora专题

RedHat专题

SUSE专题

红旗Linux专题

Android专题