第三章 使用 JDK 并发包构建程序

第三章 使用JDK并发包构建程序 1
3.1 java.util.concurrent概述 2
3.2 原子量 2
3.2.1 锁同步法 3
3.2.2 比较并交换 4
3.2.3 原子变量类 6
3.2.4 使用原子量实现银行取款 8
3.3 并发集合 12
3.3.1 队列Queue与BlockingQueue 12
3.3.2 使用 ConcurrentMap 实现类 19
3.3.3 CopyOnWriteArrayList和CopyOnWriteArraySet 20
3.4 同步器 21
3.4.1 Semaphore 21
3.4.2 Barrier 24
3.4.3 CountDownLatch 27
3.4.4 Exchanger 29
3.4.5 Future和FutureTask 31
3.5 显示锁 33
3.5.1 ReentrantLock 33
3.5.1.1 ReentrantLock的特性 34
3.5.1.2 ReentrantLock性能测试 38
3.5.2 ReadWriteLock 42
3.6 Fork-Join框架 46
3.6.1 应用Fork-Join 47
3.6.2 应用ParallelArray 51
参考文献 52

3.1 java.util.concurrent 概述

JDK5.0 以后的版本都引入了高级并发特性，大多数的特性在 java.util.concurrent 包中，是专门用于多线并发编程的，充分利用了现代多处理器和多核心系统的功能以编写大规模并发应用程序。主要包含原子量、并发集合、同步器、可重入锁，并对线程池的构造提供了强力的支持。

原子量是定义了支持对单一变量执行原子操作的类。所有类都有 get 和 set 方法，工作方法和对 volatile 变量的读取和写入一样。

并发集合是原有集合框架的补充，为多线程并发程序提供了支持。主要有：BlockingQueue，ConcurrentMap，ConcurrentNavigableMap。

同步器提供了一些帮助在线程间协调的类，包括 semaphores,barriers,latches,exchangers 等。

一般同步代码依靠内部锁（隐式锁），这种锁易于使用，但是有很多局限性。新的 Lock 对象支持更加复杂的锁定语法。和隐式锁类似，每一时刻只有一个线程能够拥有 Lock 对象，通过与其相关联的 Condition 对象，Lock 对象也支持 wait 和 notify 机制。

线程完成的任务（Runnable 对象）和线程对象（Thread）之间紧密相连。适用于小型程序，在大型应用程序中，把线程管理和创建工作与应用程序的其余部分分离开更有意义。线程池封装线程管理和创建线程对象。线程池在第一章已经讲过，不再赘述。

3.2 原子量

近来关于并发算法的研究主要焦点是无锁算法（nonblocking algorithms），这些无锁算法使用低层原子化的机器指令，例如使用compare-and-swap（CAS）代替锁保证并发情况下数据的完整性。无锁算法广泛应用于操作系统与JVM中，比如线程和进程的调度、垃圾收集、实现锁和其他并发数据结构。

在 JDK5.0 之前，如果不使用本机代码，就不能用 Java 语言编写无等待、无锁定的算法。在 java.util.concurrent 中添加原子变量类之后，这种情况发生了变化。本节了解这些新类开发高度可伸缩的无阻塞算法。

要使用多处理器系统的功能，通常需要使用多线程构造应用程序。但是正如任何编写并发应用程序的人可以告诉你的那样，要获得好的硬件利用率，只是简单地在多个线程中分割工作是不够的，还必须确保线程确实大部分时间都在工作，而不是在等待更多的工作，或等待锁定共享数据结构。

如果线程之间不需要协调，那么几乎没有任务可以真正地并行。以线程池为例，其中执行的任务通常相互独立。如果线程池利用公共工作队列，则从工作队列中删除元素或向工作队列添加元素的过程必须是线程安全的，并且这意味着要协调对头、尾或节点间链接指针所进行的访问。正是这种协调导致了所有问题。

3.2.1 锁同步法

在 Java 语言中，协调对共享字段访问的传统方法是使用**同步**，确保完成对共享字段的所有访问，同时具有适当的锁定。通过同步，可以确定（假设类编写正确）具有保护一组访问变量的所有线程都将拥有对这些变量的独占访问权，并且以后其他线程获得该锁定时，将可以看到对这些变量进行的更改。弊端是如果**锁定**竞争太厉害（线程常常在其他线程具有锁定时要求获得该锁定），会损害吞吐量，因为竞争的同步非常昂贵。对于现代 JVM 而言，无竞争的同步现在非常便宜。

基于锁的算法的另一个问题是：如果延迟具有锁的线程（因为页面错误、计划延迟或其他意料之外的延迟），则没有要求获的锁的线程可以继续运行。

还可以使用 **volatile** 变量来以比同步更低的成本存储共享变量，但它们有局限性。虽然可以保证其他变量可以立即看到对 **volatile** 变量的写入，但无法呈现原子操作的读-修改-写顺序，这意味着 **volatile** 变量无法用来可靠地实现互斥（互斥锁定）或计数器。

下面以实现一个计数器为例。通常情况下一个计数器要保证计数器的增加，减少等操作需要保持原子性，使类成为线程安全的类，从而确保没有任何更新信息丢失，所有线程都看到计数器的最新值。使用内部锁实现的同步代码一般如下：

```
package jdkapidemo;
public class SynchronizedCounter {
    private int value;
    public synchronized int getValue() {
        return value;
    }
    public synchronized int increment() {
```

```
        return ++value;
    }
    public synchronized int decrement() {
        return --value;
    }
}
```

increment() 和 decrement() 操作是原子的读-修改-写操作，为了安全实现计数器，必须使用当前值，并为其添加一个值，或写出新值，所有这些均视为一项操作，其他线程不能打断它。否则，如果两个线程试图同时执行增加，操作的不幸交叉将导致计数器只被实现了一次，而不是被实现两次。(注意，通过使值变量成为 volatile 变量并不能可靠地完成这项操作。)

计数器类可以可靠地工作，在竞争很小或没有竞争时都可以很好地执行。然而，在竞争激烈时，这将大大损害性能，因为 JVM 用了更多的时间来调度线程，管理竞争和等待线程队列，而实际工作（如增加计数器）的时间却很少。

使用锁，如果一个线程试图获取其他线程已经具有的锁，那么该线程将被阻塞，直到该锁可用。此方法具有一些明显的缺点，其中包括当线程被阻塞来等待锁时，它无法进行其他任何操作。如果阻塞的线程是高优先级的任务，那么该方案可能造成非常不好的结果（称为优先级倒置的危险）。

使用锁还有一些其他危险，如死锁（当以不一致的顺序获得多个锁时会发生死锁）。甚至没有这种危险，锁也仅是相对的粗粒度协调机制，同样非常适合管理简单操作，如增加计数器或更新互斥拥有者。如果有更细粒度的机制来可靠管理对单独变量的并发更新，则会更好一些；在大多数现代处理器都有这种机制。

3.2.2 比较并交换

大多数现代处理器都包含对多处理的支持。当然这种支持包括多处理器可以共享外部设备和主内存，同时它通常还包括对指令系统的增加来支持多处理的特殊要求。特别是，几乎每个现代处理器都有通过可以检测或阻止其他处理器的并发访问的方式来更新共享变量的**指令**。

现在的处理器（包括 Intel 和 Sparc 处理器）使用的最通用的方法是实现名为“比较并交换（Compare And Swap）”或 CAS 的原语。（在 Intel 处理器中，比较并交换通过 cmpxchg 系列指令实现。PowerPC 处理器有一对名为“加载并保留”和“条件存储”的指令，它们实现相同的目地；MIPS 与 PowerPC 处理器相似，除了第一个指令称为“加载链

接”。）

CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值。否则，处理器不做任何操作。无论哪种情况，它都会在 CAS 指令之前返回该位置的值。（在 CAS 的一些特殊情况下将仅返回 CAS 是否成功，而不提取当前值。）CAS 有效地说明了“我认为位置 V 应该包含值 A；如果包含该值，则将 B 放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。”

通常将 CAS 用于同步的方式是从地址 V 读取值 A，执行多步计算来获得新值 B，然后使用 CAS 将 V 的值从 A 改为 B。如果 V 处的值尚未同时更改，则 CAS 操作成功。

类似于 CAS 的指令允许算法执行读-修改-写操作，而无需害怕其他线程同时修改变量，因为如果其他线程修改变量，那么 CAS 会检测它（并失败），算法可以对该操作重新计算。下面的程序说明了 CAS 操作的行为（而不是性能特征），但是 CAS 的价值是它可以在硬件中实现，并且是极轻量级的（在大多数处理器中）。后面我们分析 Java 的源代码可以知道，JDK 在实现的时候使用了本地代码。下面的代码说明 CAS 的工作原理（为了便于说明，用同步语法表示）。

```
package jdkapidemo;
public class SimulatedCAS {
    private int value;
    public synchronized int getValue() {
        return value;
    }
    public synchronized int compareAndSwap(int expectedValue, int
newValue) {
        if (value == expectedValue)
            value = newValue;
        return value;
    }
}
```

基于 CAS 的并发算法称为“无锁定算法”，因为线程不必再等待锁定（有时称为互斥或关键部分，这取决于线程平台的术语）。无论 CAS 操作成功还是失败，在任何一种情况中，它都在可预知的时间内完成。如果 CAS 失败，调用者可以重试 CAS 操作或采取其他适合的操作。下面的代码显示了重新编写的计数器类来使用 CAS 替代锁定：

```
package jdkapidemo;
public class CasCounter {
```

```
    private SimulatedCAS value;
    public int getValue() {
        return value.getValue();
    }
    public int increment() {
        int oldValue = value.getValue();
        while (value.compareAndSwap(oldValue, oldValue + 1) !=
oldValue)
            oldValue = value.getValue();
        return oldValue + 1;
    }
}
```

如果每个线程在其他线程任意延迟（或甚至失败）时都将持续进行操作，就可以说该算法是 **“无等待”**的。**“无锁定算法”**要求某个线程总是执行操作。（无等待的另一种定义是保证每个线程在其有限的步骤中正确计算自己的操作，而不管其他线程的操作、计时、交叉或速度。这一限制可以是系统中线程数的函数；例如，如果有 10 个线程，每个线程都执行一次 CasCounter.increment() 操作，最坏的情况下，每个线程将必须重试最多九次，才能完成增加。）

再过去的 15 年里，人们已经对无等待且无锁算法（也称为**无阻塞算法**）进行了大量研究，许多人通用数据结构已经发现了无阻塞算法。无阻塞算法被广泛用于操作系统和 JVM 级别，进行诸如线程和进程调度等任务。虽然它们的实现比较复杂，但相对于基于锁的备选算法，它们有许多优点：可以避免优先级倒置和死锁等危险，竞争比较便宜，协调发生在更细的粒度级别，允许更高程度的并行机制等等。

3.2.3 原子变量类

java.util.concurrent.atomic 包中添加原子变量类。所有原子变量类都公开**“比较并设置”原语**（与比较并交换类似），这些原语都是使用平台上可用的最快本机结构（比较并交换、加载链接/条件存储，最坏的情况下是旋转锁）来实现的。 java.util.concurrent.atomic 包中提供了原子变量的 9 种风格（AtomicInteger、AtomicLong、AtomicReference、AtomicBoolean、原子整型、长型、引用、及原子标记引用和戳记引用类的数组形式，其原子地更新一对值）。

原子变量类可以认为是 volatile 变量的泛化，它扩展了 volatile 变量的概念，来支持原子条件的比较并设置更新。读取和写入原子变量与读取和写入对 volatile 变量的访问具有相同的存取语义。

虽然原子变量类表面看起来与 SynchronizedCounter 例子一样，但相似仅是表面的。在表面之下，原子变量的操作会变为平台提供的用于并发访问的硬件原语，比如比较并交换。

调整具有竞争的并发应用程序的可伸缩性的通用技术是降低使用的锁对象的粒度，希望更多的锁请求从竞争变为不竞争。从锁转换为原子变量可以获得相同的结果，通过切换为更细粒度的协调机制，竞争的操作就更少，从而提高了吞吐量。

下面的程序是使用原子变量后的计数器：

```
package jdkapidemo;
import java.util.concurrent.atomic.AtomicInteger;
public class AtomicCounter {
    private AtomicInteger value = new AtomicInteger();
    public int getValue() {
        return value.get();
    }
    public int increment() {
        return value.incrementAndGet();
    }
    public int increment(int i) {
        return value.addAndGet(i);
    }
    public int decrement() {
        return value.decrementAndGet();
    }
    public int decrement(int i) {
        return value.addAndGet(-i);
    }
}
```

下面写一个测试类：

```
package jdkapidemo;
public class AtomicCounterTest extends Thread {
    AtomicCounter counter;
    public AtomicCounterTest(AtomicCounter counter) {
        this.counter = counter;
    }
    @Override
    public void run() {
        int i = counter.increment();
        System.out.println("generated number:" + i);
    }
    public static void main(String[] args) {
        AtomicCounter counter = new AtomicCounter();
        for (int i = 0; i < 10; i++) {//10个线程
```

```
            new AtomicCounterTest(counter).start();
        }
    }
}
```

运行结果如下：

```
generated number:1
generated number:2
generated number:3
generated number:4
generated number:5
generated number:7
generated number:6
generated number:9
generated number:10
generated number:8
```

会发现 10 个线程运行中，没有重复的数字，原子量类使用本机 CAS 实现了值修改的原子性。

3.2.4 使用原子量实现银行取款

下面再看一个银行取款的例子，下面定义了一个帐户类 Account，重点关注其中的取款方法 withdraw()，取款前先判断余额是否足够支付，然后把余额减去取款额，为了更好的模拟线程并发的情况，在其中增了一个休眠语句。

```
package jdkapidemo.bank;
public class Account {
    private double balance;
    public Account(double money) {
        balance = money;
        System.out.println("Totle Money: " + balance);
    }
    public void deposit(double money) {
        balance = balance + money;
    }
    public void withdraw(double money, int delay) {
        if (balance >= money) {
            try {
                Thread.sleep(delay);
                balance = balance - money;
                System.out.println(Thread.currentThread().getName()
                        + " withdraw  " + money + " successful!" +
balance);
            } catch (InterruptedException e) {
```

```
            }
        } else
            System.out.println(Thread.currentThread().getName()
                    + " balance is not enough, withdraw failed!" +
balance);
    }
}
```

为了测试帐户类，定义一个测试类

```
package jdkapidemo.bank;
public class AccountThread extends Thread {
    Account account;
    int delay;
    public AccountThread(Account acount, int delay) {
        this.account = acount;
        this.delay = delay;
    }
    public void run() {
        account.withdraw(100, delay);
    }
    public static void main(String[] args) {
        Account acount = new Account(100);
        AccountThread acountThread1 = new AccountThread(acount, 1000);
        AccountThread acountThread2 = new AccountThread(acount, 0);
        acountThread1.start();
        acountThread2.start();
    }
}
```

运行结果如下：

```
Totle Money: 100.0
Thread-1 withdraw    100.0 successful!0.0
Thread-0 withdraw    100.0 successful!-100.0
```

从运行结果可以看出，总额 100 元，使用两个线程同时取钱，都成功，最后帐户余额为 -100 元，表现为透支，这样破坏了数据的完整性。

从程序可以看出 withdrawal 方法包含了余额判断语句，为什么还会发生数据的一致性被破坏呢？因多线程并发，当执行“`balance = balance – money`”这条语句时，`balance` 的实际值已经不是先前的值。

按照正确的业务逻辑，需要保证在一个取款操作结束时，不能执行另一个取款操作，需要把 withdraw 同步起来，我们可以使用 synchronized 关键字。修改如下：

```
public synchronized void withdraw(double money, int delay)
```

运行修改后的程序,结果如下：

```
Totle Money: 100.0
Thread-1 withdraw  100.0 successful!0.0
Thread-0balance is not enough, withdraw failed!0.0
```

前面我们讲过了原子量的使用，现在修改 balance 为原子量。用原子量的特性实现取款操作的原子性。

把 Account 类修为 AtomicAccount，把 balance 定义为 AtomicLong 类型，然后修改 withdraw 方法，把原来方法的修改语句“`balance = balance – money`”修改为“**`balance.compareAndSet(oldvalue, oldvalue - money)`**”，这个方法在执行的时候是原子化的，首先比较所读取的值是否和被修改的值一致，如果一致则执行原子化修改，否则失败。如果帐余额在读取之后，被修改了，则 **compareAndSet** 会返回 FALSE，则余额修改失败，不能完成取款操作。

```
package jdkapidemo.bank;
import java.util.concurrent.atomic.AtomicLong;
public class AtomicAccount {
    AtomicLong balance;
    public AtomicAccount(long money) {
        balance = new AtomicLong(money);
        System.out.println("Totle Money: " + balance);
    }
    public void deposit(long money) {
        balance.addAndGet(money);
    }
    public void withdraw(long money, int delay) {
        long oldvalue = balance.get();
        if (oldvalue >= money) {
            try {
                Thread.sleep(delay);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            if (balance.compareAndSet(oldvalue, oldvalue - money)) {
                System.out.println(Thread.currentThread().getName()
                        + " withdraw  " + money + " successful!" +
balance);
            } else {
                System.out.println(Thread.currentThread().getName()
                        + "thread concurrent, withdraw failed!" +
balance);
```

```
            }
        } else {
            System.out.println(Thread.currentThread().getName()
                    + " balance is not enough,withdraw failed!" +
balance);
        }
    }
    public long get() {
        return balance.get();
    }
}
```

重新定义测试类：

```
package jdkapidemo.bank;
public class AtomicAccountTest extends Thread {
    AtomicAccount account;
    int delay;
    public AtomicAccountTest(AtomicAccount account, int delay) {
        this.account = account;
        this.delay = delay;
    }
    public void run() {
        account.withdraw(100, delay);
    }
    public static void main(String[] args) {
        AtomicAccount account = new AtomicAccount(100);
        AtomicAccountTest accountThread1 = new
AtomicAccountTest(account, 1000);
        AtomicAccountTest accountThread2 = new
AtomicAccountTest(account, 0);
        accountThread1.start();
        accountThread2.start();
    }
}
```

运行结果如下：

```
    Totle Money: 100
    Thread-1 withdraw  100 successful!0
    Thread-0 thread concurrent, withdraw failed!0
```

从运行结果可以看出，两个线程在执行 withdraw 方法时，开始余额比较都是成功的，随后在更新余额是我们使用了 balance.compareAndSet(oldvalue, oldvalue - money)原子方法，这个方法在修改余额值之前还要**比较所读取的值是否和被修改的值一致**，如果一致则修改，

如果不一致则修改失败，返回 false。并且保证在修改的过程是原子性的，不会被中断。

大多数用户都不太可能自己使用原子变量开发无阻塞算法,他们更可能使用 java.util.concurrent 中提供的版本，如 ConcurrentLinkedQueue。但是万一您想知道对比以前 JDK 中的相类似的功能，这些类的性能是如何改进的，可以使用通过原子变量类公开的细粒度、硬件级别的并发原语。

开发人员可以直接将原子变量用作共享计数器、序号生成器和其他独立共享变量的高性能替代，否则必须通过同步保护这些变量。

通过内部公开新的低级协调原语，和提供一组公共原子变量类，现在用 Java 语言开发无等待、无锁定算法首次变为可行。然后， java.util.concurrent 中的类基于这些低级原子变量工具构建，为它们提供比以前执行相似功能的类更显著的可伸缩性优点。虽然您可能永远不会直接使用原子变量，还是应该为它们的存在而欢呼。

3.3 并发集合

我们将探讨集合框架中新的 Queue 接口、这个接口的非并发和并发实现、并发 Map 实现和专用于读操作大大超过写操作这种情况的并发 List 和 Set 实现。

3.3.1 队列 Queue 与 BlockingQueue

java.util 包为集合提供了一个新的基本接口：java.util.Queue。虽然肯定可以在相对应的两端进行添加和删除而将 java.util.List 作为队列对待，但是这个新的 Queue 接口提供了支持添加、删除和检查集合的更多方法。

1) boolean add(Object e)：将指定的元素插入此队列（如果立即可行且不会违反容量限制），在成功时返回 true，如果当前没有可用的空间，则抛出 IllegalStateException。

2) public boolean offer(Object element)：将指定的元素插入此队列（如果立即可行且不会违反容量限制），当使用有容量限制的队列时，此方法通常要优于 add(E)，后者可能无法插入元素，而只是抛出一个异常。

3) public Object remove()：获取并移除此队列的头。

4) public Object poll()；获取并移除此队列的头，如果此队列为空，则返回 null。

5) public Object element()；获取但是不移除此队列的头。此队列为空时将抛出一个异常。

6) public Object peek()；获取但不移除此队列的头；如果此队列为空，则返回 null。

基本上，一个队列就是一个先入先出（FIFO）的数据结构。一些队列有大小限制，因此如果想在一个满的队列中加入一个新项，多出的项就会被拒绝。这时新的 offer 方法就可以起作用了。它不是对调用 add()方法抛出一个 unchecked 异常，而只是得到由 **offer()方法**返回的 false。 remove() 和 poll() 方法都是从队列中删除第一个元素（head）。 remove() 的行为与 Collection 接口的版本相似，但是新的 **poll()方法**在用空集合调用时不是抛出异常，只是返回 null。因此新的方法更适合容易出现异常条件的情况。后两个方法 element() 和 peek()用于在队列的头部查询元素。与 remove() 方法类似，在队列为空时， element() 抛出一个异常，而 peek() 返回 null。

在 JDK 中有两组 Queue 实现：实现了新 BlockingQueue 接口的和没有实现这个接口的。我将首先分析那些没有实现的。

在最简单的情况下，原来有的 java.util.LinkedList 实现已经改造成不仅实现 java.util.List 接口，而且还实现 java.util.Queue 接口。可以将 LinkedList 集合看成这两者中的任何一种。下面的程序将显示把 LinkedList 作为 Queue 的使用方法：

```
package queuedemo;
import java.util.LinkedList;
import java.util.Queue;
public class QueueTest {
    public static void main(String[] args) {
        Queue queue = new LinkedList();
        queue.offer("One");
        queue.offer("Two");
        queue.offer("Three");
        queue.offer("Four");
        System.out.println("Head of queue is: " + queue.poll());
    }
}
```

输出结果为：

```
Head of queue is: One
```

PriorityQueue 和 ConcurrentLinkedQueue 类在 Collection Framework 中加入两个具体集合实现。PriorityQueue 类实质上维护了一个有序列表。加入到 Queue 中的元素根据它们的天然排序（通过其 java.util.Comparable 实现）或者根据传递给构造函数的 java.util.Comparator 实现来定位。将上面程序中的 LinkedList 改变为 PriorityQueue 将会打印出 Four 而不是 One，因为按字母排列，即字符串的天然顺序，Four 是第一个。 ConcurrentLinkedQueue 是

基于链接节点的、线程安全的队列。并发访问不需要同步。因为它在队列的尾部添加元素并从头部删除它们，所以只要不需要知道队列的大小， ConcurrentLinkedQueue 对公共集合的共享访问就可以工作得很好。收集关于队列大小的信息会很慢，需要遍历队列。

```
package queuedemo;
import java.util.PriorityQueue;
import java.util.Queue;
public class PriorityQueueDemo {
    public static void main(String[] args) {
        Queue<String> queue = new PriorityQueue<String>();
        queue.offer("One");
        queue.offer("Two");
        queue.offer("Three");
        queue.offer("Four");
        System.out.println("Head of queue is: " + queue.poll());
    }
}
```

输出结果如下：

```
Head of queue is: Four
```

新的 java.util.concurrent 包可用的具体集合类中加入了 BlockingQueue 接口和五个阻塞队列类。阻塞队列实质上就是一种带有一点扭曲的 FIFO 数据结构，不是立即从队列中添加或者删除元素，线程执行操作被阻塞，直到有空间或者元素可用。 BlockingQueue 接口的 Javadoc 给出了阻塞队列的基本用法，生产者中的 put() 操作会在没有空间可用时阻塞，而消费者的 take() 操作会在队列中没有任何东西时阻塞。

五个队列所提供的各有不同：

ArrayBlockingQueue ： 一个由数组支持的有界队列。

LinkedBlockingQueue ： 一个由链接节点支持的可选有界队列。

PriorityBlockingQueue ： 一个由优先级堆支持的无界优先级队列。

DelayQueue ： 一个由优先级堆支持的、基于时间的调度队列。

SynchronousQueue ： 一个利用 BlockingQueue 接口的简单聚集（rendezvous）机制。

下面以 `ArrayBlockingQueue` 为例写一个程序，表示生产者-消费者问题。生产者向阻塞队列中放入字符，消费者从阻塞队列中移除字符。

```
package queuedemo;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
public class BlockingQueueDemo {
```

```
        public static void main(String[] args) {
            BlockingQueue<String> queue = new ArrayBlockingQueue<String>(5);
            Producer p = new Producer(queue);
            Consumer c1 = new Consumer(queue);
            Consumer c2 = new Consumer(queue);
            new Thread(p).start();
            new Thread(c1).start();
            new Thread(c2).start();
        }
    }
    class Producer implements Runnable {
        private final BlockingQueue<String> queue;
        Producer(BlockingQueue<String> q) {
            queue = q;
        }
        public void run() {
            try {
                for (int i = 0; i < 100; i++) {
                    queue.put(produce());
                }
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
        String produce() {
            String temp = "" + (char) ('A' + (int) (Math.random() * 26));
            System.out.println("produce " + temp);
            return temp;
        }
    }
    class Consumer implements Runnable {
        private final BlockingQueue<String> queue;
        Consumer(BlockingQueue<String> q) {
            queue = q;
        }
        public void run() {
            try {
                for (int i = 0; i < 100; i++) {
                    consume(queue.take());
                }
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
```

```
        void consume(String x) {
            System.out.println("consume " + x);
        }
    }
```

输出结果如下：

```
produce W
produce S
produce D
produce Q
consume S
consume W
consume Q
consume D
produce V
produce J
produce P
produce A
consume V
consume P
produce I
consume J
consume I
produce C
...
```

前两个类 ArrayBlockingQueue 和 LinkedBlockingQueue 几乎相同，只是在后备存储器方面有所不同，LinkedBlockingQueue 并不总是有容量界限。无大小界限的 LinkedBlockingQueue 类在添加元素时永远不会有阻塞队列的等待（至少在其中有 Integer.MAX_VALUE 元素之前不会）。

PriorityBlockingQueue 是具有无界限容量的队列，它利用所包含元素的 Comparable 排序顺序来以逻辑顺序维护元素。可以将它看作 TreeSet 的可能替代物。例如，在队列中加入字符串 One、Two、Three 和 Four 会导致 Four 被第一个取出来。对于没有天然顺序的元素，可以为构造函数提供一个 Comparator 。不过对 PriorityBlockingQueue 有一个技巧。从 iterator() 返回的 Iterator 实例不需要以优先级顺序返回元素。如果必须以优先级顺序遍历所有元素，那么让它们都通过 toArray() 方法并自己对它们排序，像 Arrays.sort(pq.toArray()) 。

新的 DelayQueue 实现可能是其中最有意思（也是最复杂）的一个。加入到队列中的元素必须实现新的 Delayed 接口（只有一个方法 long getDelay(java.util.concurrent.TimeUnit

unit))。因为队列的大小没有界限，使得添加可以立即返回，但是在延迟时间过去之前，不能从队列中取出元素。如果多个元素完成了延迟，那么最早失效/失效时间最长的元素将第一个取出。实际上没有听上去这样复杂。下面的程序演示了这种新的阻塞队列集合的使用：

```
package queuedemo;
import java.util.Random;
import java.util.concurrent.DelayQueue;
import java.util.concurrent.Delayed;
import java.util.concurrent.TimeUnit;
public class DelayQueueDemo {
    static class NanoDelay implements Delayed {
        long trigger;
        NanoDelay(long i) {
            trigger = System.nanoTime() + i;
        }
        public boolean equals(Object other) {
            return ((NanoDelay) other).trigger == trigger;
        }
        public boolean equals(NanoDelay other) {
            return ((NanoDelay) other).trigger == trigger;
        }
        public long getDelay(TimeUnit unit) {
            long n = trigger - System.nanoTime();
            return unit.convert(n, TimeUnit.NANOSECONDS);
        }
        public long getTriggerTime() {
            return trigger;
        }
        public String toString() {
            return String.valueOf(trigger);
        }
        @Override
        public int compareTo(Delayed o) {
            long i = trigger;
            long j = ((NanoDelay) o).trigger;
            if (i < j)
                return -1;
            if (i > j)
                return 1;
            return 0;
        }
    }
    public    static    void    main(String    args[])    throws
```

```
InterruptedException {
          Random random = new Random();
          DelayQueue<NanoDelay> queue = new DelayQueue<NanoDelay>();
          for (int i = 0; i < 5; i++) {
              queue.add(new NanoDelay(random.nextInt(1000)));
          }
          long last = 0;
          for (int i = 0; i < 5; i++) {
              NanoDelay delay = (NanoDelay) (queue.take());
              long tt = delay.getTriggerTime();
              System.out.println("Trigger time: " + tt);
              if (i != 0) {
                  System.out.println("Delta: " + (tt - last));
              }
              last = tt;
          }
      }
  }
```

运行结果如下：

```
Trigger time: 5629057839457
Trigger time: 5629057894502
Delta: 55045
Trigger time: 5629057925948
Delta: 31446
Trigger time: 5629057938107
Delta: 12159
Trigger time: 5629057948783
Delta: 10676
```

这个例子首先是一个内部类 NanoDelay ，它实质上将暂停任意纳秒（nanosecond）数，这里利用了 System 的新 nanoTime() 方法。然后 main() 方法只是将 NanoDelay 对象放到队列中并再次将它们取出来。如果希望队列项做一些其他事情，就需要在 Delayed 对象的实现中加入方法，并在从队列中取出后调用这个新方法。（请随意扩展 NanoDelay 以试验加入其他方法做一些有趣的事情。）显示从队列中取出元素的两次调用之间的时间差。如果时间差是负数，可以视为一个错误，因为永远不会在延迟时间结束时，在一个更早的触发时间从队列中取得项。

SynchronousQueue 类是最简单的。它没有内部容量。它就像线程之间的手递手机制。在队列中加入一个元素的生产者会等待另一个线程的消费者。当这个消费者出现时，这个元素就直接在消费者和生产者之间传递，永远不会加入到阻塞队列中。

3.3.2 使用 ConcurrentMap 实现类

java.util.concurrent.ConcurrentMap 接口和 ConcurrentHashMap 实现类只能在键不存在时将元素加入到 map 中，只有在键存在并映射到特定值时才能从 map 中删除一个元素。主要定义了下面几个方法（K 表示键的类型，V 表示值的类型）：

V putIfAbsent(K key,V value)：如果指定键已经不再与某个值相关联，则将它与给定值关联。

boolean remove(Object key,Object value)：只有目前将键的条目映射到给定值时，才移除该键的条目。

boolean replace(K key,V oldValue,V newValue)：只有目前将键的条目映射到给定值时，才替换该键的条目。

V replace(K key,V value)：只有目前将键的条目映射到某一值时，才替换该键的条目。

putIfAbsent() 方法用于在 map 中进行添加。这个方法以要添加到 ConcurrentMap 中的键的值为参数，就像普通的 put() 方法，但是只有在 map 不包含这个键时，才能将键加入到 map 中。如果 map 已经包含这个键，那么这个键的现有值就会保留。 putIfAbsent() 方法是原子的。等价于下面的代码（除了原子地执行此操作之外）：

```
if (!map.containsKey(key))
      return map.put(key, value);
  else
    return map.get(key);
```

像 putIfAbsent() 方法一样，重载后的 remove() 方法有两个参数：键和值。在调用时，只有当键映射到指定的值时才从 map 中删除这个键。如果不匹配，那么就不删除这个键，并返回 false。如果值匹配键的当前映射内容，那么就删除这个键，**这个方法是原子性的**。这种操作的等价源代码（除了原子地执行此操作之外）：

```
if (map.containsKey(key) && map.get(key).equals(value)) {
      map.remove(key);
      return true;
} else return false;
```

总之，ConcurrentMap 中定义的方法是原子性的。

3.3.3 CopyOnWriteArrayList 和 CopyOnWriteArraySet

这两个集合对对 copy-on-write 模式作了比较好的支持。这个模式说明了，为了维护对象的一致性快照，要依靠不可变性（immutability）来消除在协调读取不同的但是相关的属性时需要的同步。

对于集合，这意味着如果有大量的读（即 get() ）和迭代，不必进行同步操作以照顾偶尔的写（即 add() ）调用。对于新的 CopyOnWriteArrayList 和 CopyOnWriteArraySet 类，所有可变的（mutable）操作都首先取得后台数组的副本，对副本进行更改，然后替换副本。这种做法保证了在遍历自身可更改的集合时，永远不会抛出 ConcurrentModificationException。遍历集合会用原来的集合完成，而在以后的操作中使用更新后的集合。

这些新的集合最适合于读操作通常大大超过写操作的情况。集合的使用与它们的非 copy-on-write 替代物完全一样。只是创建集合并在其中加入或者删除元素。即使对象加入到了集合中，原来的 Iterator 也可以进行，继续遍历原来集合中的项。

下面是使用 copy-on-write 集合和一般类型集合进行遍历的例子：

```
package copyonwrite;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.ConcurrentModificationException;
import java.util.Iterator;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;
public class CopyOnWriteDemo {
    @SuppressWarnings("unchecked")
    public static void main(String args[]) {
        String[] ss = { "aa", "bb", "cc" };
        List list1 = new CopyOnWriteArrayList(Arrays.asList(ss));
        List list2 = new ArrayList(Arrays.asList(ss));
        Iterator itor1 = list1.iterator();
        Iterator itor2 = list2.iterator();
        list1.add("New");
        list2.add("New");
        try {
            printAll(itor1);
        } catch (ConcurrentModificationException e) {
```

```
            System.err.println("Shouldn't get here");
        }
        try {
            printAll(itor2);
        } catch (ConcurrentModificationException e) {
            System.err
                    .println("Will get
here.ConcurrentModificationException occurs!");
        }
    }
    @SuppressWarnings("unchecked")
    private static void printAll(Iterator itor) {
        while (itor.hasNext()) {
            System.out.println(itor.next());
        }
    }
}
```

运行结果如下：

```
Will get here.ConcurrentModificationException occurs!
aa
bb
cc
```

这个示例程序创建 CopyOnWriteArrayList 和 ArrayList 这两个实例。在得到每一个实例的 Iterator 后，分别在其中加入一个元素。当 ArrayList 迭代因一个 ConcurrentModificationException 问题而立即停止时，CopyOnWriteArrayList 迭代可以继续，不会抛出异常，因为原来的集合是在得到 iterator 之后改变的。如果这种行为（比如通知原来一组事件监听器中的所有元素）是您需要的，那么最好使用 copy-on-write 集合。如果不使用的话，就还用原来的，并保证在出现异常时对它进行处理。

3.4 同步器

3.4.1 Semaphore

类 java.util.concurrent.Semaphore 提供了一个计数信号量，从概念上讲，信号量维护了一个许可集。如有必要，在许可可用前会阻塞每一个 acquire()，然后再获取该许可。每个 release() 添加一个许可，从而可能释放一个正在阻塞的获取者。但是，不使用实际的许可对象，

Semaphore 只对可用许可的号码进行计数，并采取相应的行动。

Semaphore 通常用于限制可以访问某些资源（物理或逻辑的）的线程数目。一般操作系统的进程调度中使用了 PV 原语，需要设置一个信号量表示可用资源的数量，P 原语就相当于 acquire()，V 原语就相当于 release()。

例如，下面的类使用信号量控制对内容池的访问，内容池的大小作为 Semaphore 的构造参数传递初始化*许可*的数目，每个线程获取数据之前必须获得许可，这样就限制了访问内容池的线程数目：

```
package synchronizer;
import java.util.concurrent.Semaphore;
class PoolSemaphoreDemo {
    private static final int MAX_AVAILABLE = 5;
    private final Semaphore available = new Semaphore(MAX_AVAILABLE, true);
    public static void main(String args[]) {
        final PoolSemaphoreDemo pool = new PoolSemaphoreDemo();
        Runnable runner = new Runnable() {
            @Override
            public void run() {
                try {
                    Object o;
                    o = pool.getItem();
                    System.out.println(Thread.currentThread().getName()
                            + " acquire " + o);
                    Thread.sleep(1000);
                    pool.putItem(o);
                    System.out.println(Thread.currentThread().getName()
                            + " release " + o);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
        for (int i = 0; i < 10; i++)// 构造 10 个线程
        {
            Thread t = new Thread(runner, "t" + i);
            t.start();
        }
    }
    //获取数据，需要得到许可
    public Object getItem() throws InterruptedException {
        available.acquire();
        return getNextAvailableItem();
```

```
        }
        //放回数据，释放许可
        public void putItem(Object x) {
            if (markAsUnused(x))
                available.release();
        }
        protected Object[] items = { "AAA", "BBB", "CCC", "DDD", "EEE" };
        protected boolean[] used = new boolean[MAX_AVAILABLE];
        protected synchronized Object getNextAvailableItem() {
            for (int i = 0; i < MAX_AVAILABLE; ++i) {
                if (!used[i]) {
                    used[i] = true;
                    return items[i];
                }
            }
            return null;
        }
        protected synchronized boolean markAsUnused(Object item) {
            for (int i = 0; i < MAX_AVAILABLE; ++i) {
                if (item == items[i]) {
                    if (used[i]) {
                        used[i] = false;
                        return true;
                    } else
                        return false;
                }
            }
            return false;
        }
    }
```

运行结果如下：

```
t0 acquire AAA
t1 acquire BBB
t4 acquire EEE
t5 acquire DDD
t2 acquire CCC
t0 release AAA
t3 acquire AAA
t8 acquire BBB
t1 release BBB
t5 release DDD
t6 acquire DDD
t4 release EEE
t7 acquire EEE
```

```
t2 release CCC
t9 acquire CCC
t3 release AAA
t8 release BBB
t6 release DDD
t7 release EEE
t9 release CCC
```

获得一项前，每个线程必须从信号量获取许可，从而保证可以使用该项。该线程结束后，将项返回到池中并将许可返回到该信号量，从而允许其他线程获取该项。

从程序的运行结果，我们可以看出，池的大小是 5，先前有 5 个线程可以使用池中的内容，后面的线程调用 acquire()获得池的许可时，被阻塞。直到前面的线程释放已经获得的许可，后面的线程才可以使用池中的内容。

注意，调用 acquire()时无法保持同步锁，因为这会阻止将数据项返回到池中。信号量封装所需的同步，以限制对池的访问，这同维持该池本身一致性所需的同步是分开的。

将信号量初始化为 1，使得它在使用时最多只有一个可用的许可，从而可用作一个相互排斥的锁。这通常也称为二进制信号量，因为它只能有两种状态：一个可用的许可，或零个可用的许可。按此方式使用时，二进制信号量具有某种属性（与很多 Lock 实现不同），即可以由线程释放“锁”，而不是由所有者（因为信号量没有所有权的概念）。在某些专门的上下文（如死锁恢复）中这会很有用。

3.4.2 Barrier

在实际应用中，有时候需要多个线程同时工作以完成同一件事情，而且在完成过程中，往往会等所有线程都到达某一个阶段后再统一执行。

比如有几个旅行团需要途经深圳、广州、最后到达重庆。旅行团中有自驾游的、有徒步的、有乘坐旅游大巴的；这些旅行团同时出发，并且每到一个目的地，都要等待其他旅行团到达此地后再同时出发，直到都到达终点站重庆。

这时候 **java.util.concurrent.CyclicBarrier** 就可以派上用场。一个同步辅助类，它允许一组线程互相等待，直到到达某个公共屏障点 (common barrier point)。在涉及一组固定大小的线程的程序中，这些线程必须不时地互相等待，此时 CyclicBarrier 很有用。

因为该 barrier 在释放等待线程后可以重用，所以称它为循环的 barrier。CyclicBarrier 最重要的属性就是参与者个数，另外最要方法是 await()。当所有线程都调用了 await()后，就

表示这些线程都可以继续执行，否则就会等待。

CyclicBarrier 支持一个可选的 Runnable 命令，在一组线程中的最后一个线程到达之后（但在释放所有线程之前），该命令只在每个屏障点运行一次。若在继续所有参与线程之前更新共享状态，此屏障操作有用。

上面提到的旅行团问题，可以用下面的程序实现，在程序中，某一个旅行团先到达某一个中转站后，调用 await()方法等待其他旅行团，都到齐后，执行 Runnable。

```
package synchronizer;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class CyclicBarrierDemo {
    // 徒步需要的时间: Shenzhen, Guangzhou, Chongqing
    private static int[] timeForWalk = { 5, 8, 15 };
    // 自驾游
    private static int[] timeForSelf = { 1, 3, 4 };
    // 旅游大巴
    private static int[] timeForBus = { 2, 4, 6 };
    static String nowTime() {//时间格式化
        SimpleDateFormat sdf = new SimpleDateFormat("HH:mm:ss");
        return sdf.format(new Date()) + ": ";
    }
    static class Tour implements Runnable {
        private int[] timeForUse;
        private CyclicBarrier barrier;
        private String tourName;
        public Tour(CyclicBarrier barrier, String tourName, int[] timeForUse)
{
            this.timeForUse = timeForUse;
            this.tourName = tourName;
            this.barrier = barrier;
        }
        public void run() {
            try {
                Thread.sleep(timeForUse[0] * 1000);
                System.out.println(nowTime() + tourName + " Reached
Shenzhen");
                barrier.await();//到达中转站后等待其他旅行团
                Thread.sleep(timeForUse[1] * 1000);
```

```
                    System.out.println(nowTime() + tourName + " Reached
Guangzhou");
                    barrier.await();//到达中转站后等待其他旅行团
                    Thread.sleep(timeForUse[2] * 1000);
                    System.out.println(nowTime() + tourName + " Reached
Chongqing");
                    barrier.await();//到达中转站后等待其他旅行团
                } catch (InterruptedException e) {
                } catch (BrokenBarrierException e) {
                }
            }
        }
        public static void main(String[] args) {
            // 三个旅行团都到到达某一个站点后，执行下面的操作，表示都到齐了。
            Runnable runner = new Runnable() {
                @Override
                public void run() {
                    System.out.println("we all are here.");
                }
            };
            CyclicBarrier barrier = new CyclicBarrier(3, runner);
            //使用线程池
            ExecutorService exec = Executors.newFixedThreadPool(3);
            exec.submit(new Tour(barrier, "WalkTour", timeForWalk));
            exec.submit(new Tour(barrier, "SelfTour", timeForSelf));
            exec.submit(new Tour(barrier, "BusTour", timeForBus));
            exec.shutdown();
        }
    }
```

运行结果如下：

```
17:13:18: SelfTour Reached Shenzhen
17:13:19: BusTour Reached Shenzhen
17:13:22: WalkTour Reached Shenzhen
we all are here.
17:13:25: SelfTour Reached Guangzhou
17:13:26: BusTour Reached Guangzhou
17:13:30: WalkTour Reached Guangzhou
we all are here.
17:13:34: SelfTour Reached Chongqing
17:13:36: BusTour Reached Chongqing
17:13:45: WalkTour Reached Chongqing
we all are here.
```

3.4.3 CountDownLatch

类 **java.util.concurrent.CountDownLatch** 是一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。

用给定的数字作为计数器初始化 CountDownLatch。一个线程调用 await()方法后，在当前计数到达零之前，会一直受阻塞。其他线程调用 countDown() 方法，会使计数器递减，所以，计数器的值为 0 后，会释放所有等待的线程。其他后续的 await 调用都将立即返回。这种现象只出现一次，因为计数无法被重置。如果需要重置计数，请考虑使用 CyclicBarrier。

CountDownLatch 作为一个通用同步工具，有很多用途。使用“1”初始化的 CountDownLatch 用作一个简单的开/关锁存器，或入口：在通过调用 countDown() 的线程打开入口前，所有调用 await 的线程都一直在入口处等待。用 N 初始化的 CountDownLatch 可以使一个线程在 N 个线程完成某项操作之前一直等待，或者使其在某项操作完成 N 次之前一直等待。

下面给出了两个类，其中一组 worker 线程使用了两个倒计数 **CountDownLatch**：

第一个类是一个启动信号，在 driver 为继续执行 worker 做好准备之前，它会阻止所有的 worker 继续执行。

第二个类是一个完成信号，它允许 driver 在完成所有 worker 之前一直等待。

```
package synchronizer;
import java.util.concurrent.CountDownLatch;
public class LatchDriverDemo {
    public static final int N = 5;
    public static void main(String[] args) throws InterruptedException
{
        // 用于向工作线程发送启动信号
        CountDownLatch startSignal = new CountDownLatch(1);
        // 用于等待工作线程的结束信号
        CountDownLatch doneSignal = new CountDownLatch(N);
        for (int i = 0; i < N; ++i)
            // 创建启动线程
            new Thread(new LatchWorker(startSignal, doneSignal), "t"
+ i)
                    .start();
        // 得到线程开始工作的时间
        long start = System.nanoTime();
        // 主线程，递减开始计数器，让所有线程开始工作
        startSignal.countDown();
```

```
        // 主线程阻塞，等待所有线程完成
        doneSignal.await();
        long end = System.nanoTime();
        System.out.println("all worker take time（ms）:" + (end - start)
                / 1000000);
    }
}
class LatchWorker implements Runnable {
    // 用于等待启动信号
    private final CountDownLatch startSignal;
    // 用于发送结束信号
    private final CountDownLatch doneSignal;
    LatchWorker(CountDownLatch startSignal, CountDownLatch
doneSignal) {
        this.startSignal = startSignal;
        this.doneSignal = doneSignal;
    }
    public void run() {
        try {
            startSignal.await();// 阻塞，等待开始新信号
            doWork();
            doneSignal.countDown();// 发送完成信号
        } catch (InterruptedException ex) {
        }
    }
    void doWork() {
        System.out.println(Thread.currentThread().getName() + " is
working...");
        int sum = 0;
        for (int i = 0; i < 10000000; i++) {
            sum += i;
        }
    }
}
```

运行结果如下：

```
t0 is working...
t4 is working...
t1 is working...
t3 is working...
t2 is working...
all worker take time（ms）:65
```

另一种典型用法是，将一个问题分成 N 个部分，用执行每个部分并让 **CountDownLatch**

倒计数的 Runnable 来描述每个部分，然后将所有 Runnable 加入到 Executor 队列。当所有的子部分完成后，协调线程就能够通过 await。（当线程必须用这种方法反复倒计数时，可改为使用 CyclicBarrier。）

这个做法请大家在实验中完成。

3.4.4 Exchanger

类 java.util.concurrent.Exchanger 提供了一个同步点，在这个同步点，一对线程可以交换数据。每个线程通过 exchange()方法的入口提供数据给他的伙伴线程，并接收他的伙伴线程提供的数据，并返回。

当在运行不对成的活动时很有用，比如当一个线程填充了 buffer，另一个线程从 buffer 中消费数据；这些线程可以用 Exchanger 来交换数据。当两个线程通过 Exchanger 交互了对象，这个交换对于两个线程来说都是安全的。

下面给出了两个线程：一个生产者生产数据，通过 Exchanger 与另外一个消费者交换数据。

```
package synchronizer;
import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;
import java.util.List;
import java.util.concurrent.Exchanger;
public class ExchangerDemo {
    private static final Exchanger ex = new Exchanger();
    class DataProducer implements Runnable {
        private List list = new ArrayList();
        public void run() {
            for (int i = 0; i < 5; i++) {
                System.out.println("生产了一个数据，耗时1秒");
                list.add(new Date());
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            try {
```

```
                // 将数据准备用于交换，并返回消费者的数据
                list = (List) ex.exchange(list);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            for (Iterator iterator = list.iterator();
iterator.hasNext();) {
                System.out.println("Producer " + iterator.next());
            }
        }
    }
    class DataConsumer implements Runnable {
        private List list = new ArrayList();
        public void run() {
            for (int i = 0; i < 5; i++) {
                // 消费者产生数据，后面交换的时候给生产者
                list.add("这是一个收条。");
            }
            try {
                // 进行交换数据，返回生产者的数据
                list = (List) ex.exchange(list);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            for (Iterator iterator = list.iterator();
iterator.hasNext();) {
                Date d = (Date) iterator.next();
                System.out.println("consumer:" + d);
            }
        }
    }
    public static void main(String args[]) {
        ExchangerDemo et = new ExchangerDemo();
        new Thread(et.new DataProducer()).start();
        new Thread(et.new DataConsumer()).start();
    }
}
```

运行结果如下：

```
生产了一个数据，耗时 1 秒
生产了一个数据，耗时 1 秒
生产了一个数据，耗时 1 秒
生产了一个数据，耗时 1 秒
生产了一个数据，耗时 1 秒
Producer 这是一个收条。
```

```
Producer 这是一个收条。
Producer 这是一个收条。
Producer 这是一个收条。
Producer 这是一个收条。
consumer:Wed Feb 25 12:08:10 CST 2009
consumer:Wed Feb 25 12:08:11 CST 2009
consumer:Wed Feb 25 12:08:12 CST 2009
consumer:Wed Feb 25 12:08:13 CST 2009
consumer:Wed Feb 25 12:08:14 CST 2009
```

从运行结果可以看出，使用 Exchanger 完成了两个线程的数据交换。

3.4.5 Future 和 FutureTask

接口 public interface Future<V> 表示异步计算的结果。它提供了检查计算是否完成的方法，以等待计算的完成，并获取计算的结果。计算完成后只能使用 get() 方法来获取结果，如有必要，计算完成前可以**阻塞**此方法。取消则由 cancel 方法来执行。还提供了其他方法，以确定任务是正常完成还是被取消了。一旦计算完成，就不能再取消计算。如果为了可取消性而使用 Future 但又不提供可用的结果，则可以声明 Future<?> 形式类型、并返回 null 作为底层任务的结果。

Future 主要定义了 5 个方法：

1) boolean cancel(boolean mayInterruptIfRunning)：试图取消对此任务的执行。如果任务已完成、或已取消，或者由于某些其他原因而无法取消，则此尝试将失败。当调用 cancel 时，如果调用成功，而此任务尚未启动，则此任务将永不运行。如果任务已经启动，则 mayInterruptIfRunning 参数确定是否应该以试图停止任务的方式来中断执行此任务的线程。

2) boolean isCancelled()：如果在任务正常完成前将其取消，则返回 true。

3) boolean isDone()：如果任务已完成，则返回 true。 可能由于正常终止、异常或取消而完成，在所有这些情况中，此方法都将返回 true。

4) V get() throws InterruptedException,ExecutionException：如有必要，等待计算完成，然后获取其结果。

5) V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException：如有必要，最多等待为使计算完成所给定的时间之后，获取其结果（如果结果可用）。

FutureTask 类是 Future 的一个实现，并实现了Runnable，所以可通过Executor(线程池）来执行。也可传递给Thread对象执行。

如果在主线程中需要执行比较耗时的操作时，但又不想阻塞主线程时，可以把这些作业交给 Future 对象在后台完成，当主线程将来需要时，就可以通过 Future 对象获得后台作业的计算结果或者执行状态。

下面的例子模拟一个会计算账的过程，主线程已经获得其他帐户的总额了，为了不让主线程等待 PrivateAccount 类的计算结果的返回而启用新的线程去处理，并使用 FutureTask 对象来监控，这样，主线程还可以继续做其他事情，最后需要计算总额的时候再尝试去获得 PrivateAccount 的信息。

```
import java.util.Random;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;
public class FutureTaskDemo {
    public static void main(String[] args) {
        // 初始化一个Callable对象和FutureTask对象;
        Callable pAccount = new PrivateAccount();
        FutureTask futureTask = new FutureTask(pAccount);
        // 使用FutureTask创建一个线程
        Thread pAccountThread = new Thread(futureTask);
        System.out.println("future task starts at " +
System.nanoTime());
        // 启动线程
        pAccountThread.start();
        // 主线程执行自己的任务
        System.out.println("main thread doing something else here. ");
        // 从其他帐户获取总金额
        int totalMoney = new Random().nextInt(100000);
        System.out.println(" You have  " + totalMoney
                + "  in your other Accounts. ");
        System.out.println(" Waiting for data from Private Account ");
        // 测试后台的就计算线程是否完成，如果未完成，等待
        while (!futureTask.isDone()) {
            try {
                Thread.sleep(5);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("future task ends at " +
```

```
System.nanoTime());
        Integer privataAccountMoney = null;
        // 如果后台的FutureTask计算完成，则返回计算结果
        try {
            privataAccountMoney = (Integer) futureTask.get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
        System.out.println(" The total moeny you have is  "
                + (totalMoney + privataAccountMoney.intValue()));
    }
}
// 创建一个Callable类，模拟计算一个私有帐户中的金额
class PrivateAccount implements Callable {
    Integer totalMoney;
    @Override
    public Integer call() throws Exception {
        // 为了延长计算时间，这里暂停几秒
        Thread.sleep(5000);
        totalMoney = new Integer(new Random().nextInt(10000));
        System.out.println(" You have  " + totalMoney
                + "  in your private Account. ");
        return totalMoney;
    }
}
```

运行结果如下：

```
future task starts at 8081043630405
main thread doing something else here.
 You have   10802   in your other Accounts.
 Waiting for data from Private Account
 You have   6771   in your private Account.
future task ends at 8086046077923
 The total moeny you have is   17573
```

从运行结果可以看出，使用 FutureTask 后，主线程可以获得异步线程的计算结果了。

3.5 显示锁

3.5.1 ReentrantLock

java.util.concurrent.lock 中的类 ReentrantLock 被作为 Java 语言中 synchronized 功能

的替代，它具有相同的内存语义、相同的锁定，但在争用条件下却有更好的性能，此外，它还有 synchronized 没有提供的其他特性。

Java 是第一个直接把跨平台线程模型和正规的内存模型集成到语言中的主流语言。核心类库包含一个 Thread 类，可以用它来构建、启动和操纵线程，Java 语言包括了跨线程传达并发性约束的构造 —— synchronized 和 volatile 。在简化与平台无关的并发类的开发的同时，它决没有使并发类的编写工作变得更繁琐，只是使它变得更容易了。

把代码块声明为 synchronized，有两个重要后果，通常是指该代码具有原子性（atomicity）和可见性（visibility）。原子性意味着一次只能有一个线程执行一个指定监控对象（lock）保护的代码，从而防止多个线程在更新共享状态时相互冲突。可见性则更为微妙；它要对付内存缓存和编译器优化的各种反常行为。一般来说，线程以某种不必让其他线程立即可以看到的方式（不管这些线程在寄存器中、在处理器特定的缓存中，还是通过指令重排或者其他编译器优化）不受缓存变量值的约束，但是如果开发人员使用了同步，如下面的代码所示，那么运行库将确保某一线程对变量所做的更新先于对现有 synchronized 块所进行的更新，当进入由同一监控器（lock）保护的另一个 synchronized 块时，将立刻可以看到这些对变量所做的更新。类似的规则也存在于 volatile 变量上。

使用 synchronized 进行同步的典型方法如下：

```
synchronized (lockObject) {
   //更新对象状态
}
```

实现同步操作需要考虑安全更新多个共享变量所需的一切，不能有争用条件，不能破坏数据（假设同步的边界位置正确），而且要保证正确同步的其他线程可以看到这些变量的最新值。通过定义一个清晰的、跨平台的内存模型，通过遵守下面这个简单规则，构建“一次编写，随处运行”的并发类是有可能的：不论什么时候，只要您将编写的变量接下来可能被另一个线程读取，或者您将读取的变量最后是被另一个线程写入的，那么您必须进行同步。

Synchronized 虽然能够实现同步，但是他有一些限制，比如：它无法中断一个正在等候获得锁的线程，也无法通过投票得到锁，如果不想等下去，也就没法得到锁。

3.5.1.1ReentrantLock 的特性

java.util.concurrent.lock 中的 Lock 框架是锁定的一个抽象，它允许把锁定的实现作为

Java 类，而不是作为语言的特性来实现。这就为 Lock 的多种实现留下了空间，各种实现可能有不同的调度算法、性能特性或者锁定语义。 ReentrantLock 类实现了 Lock ，它拥有与 synchronized 相同的并发性和内存语义，但是添加了类似锁投票、定时锁等候和可中断锁等候的一些特性。此外，它还提供了在激烈争用情况下更佳的性能。（换句话说，当许多线程都想访问共享资源时，JVM 可以花更少的时候来调度线程，把更多时间用在执行线程上。）

ReentrantLock（可重入锁）有一个与锁相关的获取计数器，如果拥有锁的某个线程再次得到锁，那么获取计数器就加 1，然后锁需要被释放两次才能获得真正释放。这模仿了 synchronized 的语义；如果线程进入由线程已经拥有的监控器保护的 synchronized 块，就允许线程继续进行，当线程退出第二个（或者后续）synchronized 块的时候，不释放锁，只有线程退出它进入的监控器保护的第一个 synchronized 块时，才释放锁。

ReentrantLock 锁的使用方法如下：

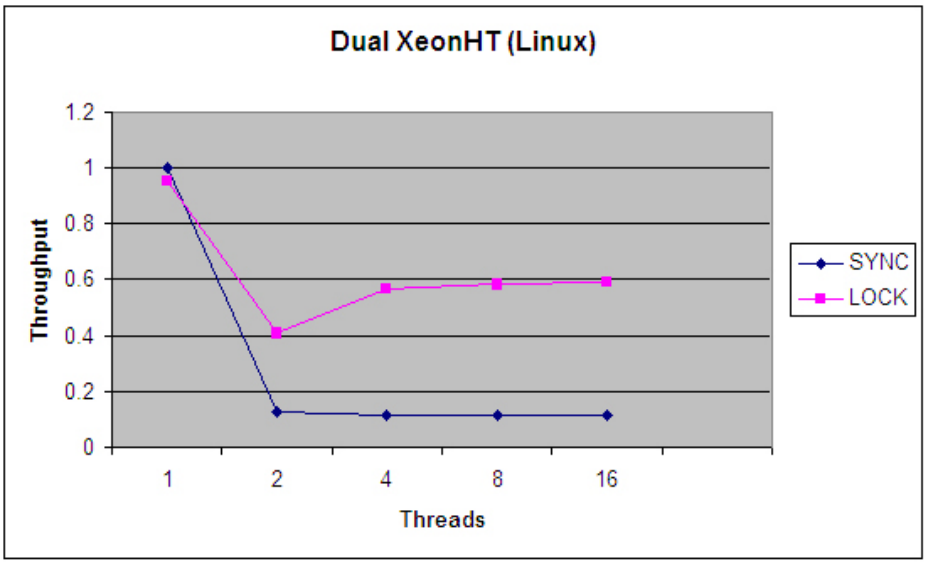
```
Lock lock = new ReentrantLock();
lock.lock();
try {
   // 更新对象状态
}
finally {
   lock.unlock();
}
```

Lock 和 synchronized 有一点明显的区别 —— lock 必须在 finally 块中释放。否则，如果受保护的代码将抛出异常，锁就有可能永远得不到释放。这一点区别看起来可能没什么，但是实际上，它极为重要。忘记在 finally 块中释放锁，可能会在程序中留下一个定时炸弹，当有一天炸弹爆炸时，您要花费很大力气才有找到源头在哪。而使用 synchronized 同步，JVM 将确保锁会获得自动释放。

除此之外，与目前的 synchronized 实现相比，争用下的 ReentrantLock 实现更具可伸缩性。

国外学者 Tim Peierls 用一个简单的线性全等伪随机数生成器（PRNG）构建了一个简单的评测，用它来测量 synchronized 和 Lock 之间相对的可伸缩性。这个示例很好，因为每次调用 nextRandom() 时，PRNG 都确实在做一些工作，所以这个基准程序实际上是在测量一个合理的、真实的 synchronized 和 Lock 应用程序，而不是测试纯粹纸上谈兵或者什么也不做的代码（就像许多所谓的基准程序一样。）

在这个基准程序中，有一个 PseudoRandom 的接口，它只有一个方法 nextRandom(int bound) 。该接口与 java.util.Random 类的功能非常类似。因为在生成下一个随机数时，PRNG 用最新生成的数字作为输入，而且把最后生成的数字作为一个实例变量来维护，其重点在于让更新这个状态的代码段不被其他线程抢占，所以要用某种形式的锁来确保这一点。（ java.util.Random 类也可以做到这点。）为 PseudoRandom 构建了两个实现；一个使用 synchronized，另一个使用 java.util.concurrent.ReentrantLock 。驱动程序生成了大量线程，每个线程都疯狂地争夺时间片，然后计算不同版本每秒能执行多少轮。下面的图总结了不同线程数量的结果。这个评测并不完美，而且只在两个系统上运行了（一个是双 Xeon 运行超线程 Linux，另一个是单处理器 Windows 系统），但是，应当足以表现 synchronized 与 ReentrantLock 相比所具有的伸缩性优势了。



根类 Object 包含某些特殊的方法，如：wait() 、 notify() 和 notifyAll() 在线程之间进行通信。这些是高级的并发性特性，许多开发人员从来没有用过它们 —— 这可能是件好事，因为它们相当微妙，很容易使用不当。幸运的是，随着 JDK 5.0 中引入 java.util.concurrent ，开发人员几乎更加没有什么地方需要使用这些方法了。

通知与锁定之间有一个交互 —— 为了在对象上 wait 或 notify ，您必须持有该对象的锁。就像 Lock 是同步的概括一样， Lock 框架包含了对 wait 和 notify 的概括，这个概括叫做条件（Condition）。Lock 对象则充当绑定到这个锁的条件变量的工厂对象，与标准的 wait 和 notify 方法不同，对于指定的 Lock ，可以有不止一个条件变量与它关联。这样就简化了许多并发算法的开发。例如， 条件（Condition） 的 Javadoc 显示了一个有界缓冲区实现的示例，该示例使用了两个条件变量，“not full”和“not empty”，它比每个 lock

只用一个 wait 设置的实现方式可读性要好一些(而且更有效)。Condition 的方法与 wait 、notify 和 notifyAll 方法类似，分别命名为 await 、 signal 和 signalAll ，因为它们不能覆盖 Object 上的对应方法。

ReentrantLock 构造器的一个参数是 boolean 值，它允许选择想要一个公平（fair）锁，还是一个不公平（unfair）锁。公平锁使线程按照请求锁的顺序依次获得锁；而不公平锁则允许讨价还价，在这种情况下，线程有时可以比先请求锁的其他线程先得到锁。

为什么我们不让所有的锁都公平呢？毕竟，公平是好事，不公平是不好的，不是吗？在现实中，公平保证了锁是非常健壮的锁，有很大的性能成本。要确保公平所需要的记帐（bookkeeping）和同步，就意味着被争夺的公平锁要比不公平锁的吞吐率更低。作为默认设置，应当把公平设置为 false ，除非公平对您的算法至关重要，需要严格按照线程排队的顺序对其进行服务。

那么同步又如何呢？内置的监控器锁是公平的吗？答案令许多人感到大吃一惊，它们是不公平的，而且永远都是不公平的。但是没有人抱怨过线程饥渴，因为 JVM 保证了所有线程最终都会得到它们所等候的锁。确保统计上的公平性，对多数情况来说，这就已经足够了，而这花费的成本则要比绝对的公平保证的低得多。所以，默认情况下 ReentrantLock 是“不公平”的，这一事实只是把同步中一直不公平的东西表面化而已。如果您在同步的时候并不介意这一点，那么在 ReentrantLock 时也不必为它担心。

虽然 ReentrantLock 是个非常动人的实现，相对 synchronized 来说，它有一些重要的优势，但是急于把 synchronized 视若敝屣，绝对是个严重的错误。 java.util.concurrent.lock 中的锁定类是用于高级用户和高级情况的工具 。一般来说，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（而不是仅仅是怀疑）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。

为什么我在一个显然“更好的”实现的使用上主张保守呢？因为对于 java.util.concurrent.lock 中的锁定类来说，synchronized 仍然有一些优势。比如，在使用 显示锁的时候，可能忘记用 finally 块释放锁，这对程序非常有害。您的程序能够通过测试，但会在实际工作中出现死锁，那时会很难指出原因（这也是为什么根本不让初级开发人员使用 Lock 的一个好理由。）但在退出 synchronized 块时，JVM 会为您做这件事。

另一个原因是因为，当 JVM 用 synchronized 管理锁定请求和释放时，JVM 在生成线程转储时能够包括锁定信息。这些对调试非常有价值，因为它们能标识死锁或者其他异常行为的来源。 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。在 JDK 5.0 成为标准（从现在开始可能需要两年）之前，使用 Lock 类将意味着要利用的特性不是每个 JVM 都有的，而且不是每个开发人员都熟悉的。

既然如此，我们什么时候才应该使用 ReentrantLock 呢？答案非常简单 —— 在确实需要一些 synchronized 所没有的特性的时候，比如时间锁等候、可中断锁等候、无块结构锁、多个条件变量或者锁投票。 ReentrantLock 还具有可伸缩性的好处，应当在高度争用的情况下使用它，但是请记住，大多数 synchronized 块几乎从来没有出现过争用，所以可以把高度争用放在一边。我建议用 synchronized 开发，直到确实证明 synchronized 不合适，而不要仅仅是假设如果使用 ReentrantLock “性能会更好”。请记住，这些是供高级用户使用的高级工具。（而且，真正的高级用户喜欢选择能够找到的最简单工具，直到他们认为简单的工具不适用为止。）。一如既往，首先要把事情做好，然后再考虑是不是有必要做得更快。

3.5.1.2ReentrantLock 性能测试

下面的例子是一个计数器，启动 N 个线程对计数器 Counter 进行递增操作，显然，这个递增操作需要同步以保证原子性，采用不同的锁来实现同步，然后查看结果。实验环境是 Windows XP with SP2，双核酷睿处理器。通过查看输出结果可以比较一下不同锁的性能。

计数器接口：

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

内部锁：

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

不公平重入锁

```
package locks;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class ReentrantLockUnfairBeanchmarkDemo implements Counter {
    private volatile long count = 0;
    private Lock lock;
    public ReentrantLockUnfairBeanchmarkDemo() {
        // 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();
        }
    }
}
```

公平重入锁

```
package locks;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class ReentrantLockFairBeanchmarkDemo implements Counter {
    private volatile long count = 0;
    private Lock lock;
    public ReentrantLockFairBeanchmarkDemo() {
        // 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();
        }
    }
```

```
    }
```

总测试程序

```
package locks;
import java.util.concurrent.CyclicBarrier;
public class BenchmarkTest {
    private Counter counter;
    private CyclicBarrier barrier;
    private int threadNum;
    private int loopNum;
    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 = 5000;
        int loopNum = 100;
        new BenchmarkTest(new SynchronizedBenchmarkDemo(),
threadNum, loopNum,
                "内部锁").test();
        new BenchmarkTest(new ReentrantLockUnfairBeanchmarkDemo(),
threadNum,
                loopNum, "不公平重入锁").test();
        new BenchmarkTest(new ReentrantLockFairBeanchmarkDemo(),
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);
            }
        }
    }
}
```

从程序中可以看出两路 threadNum 和 loopNum 的值分别为 5000 和 100，就是创建 5000 个线程，每个线程循环 100 次。运行结果如下：

```
内部锁  count value:500000
内部锁  花费时间:1406 毫秒
不公平重入锁  count value:500000
不公平重入锁  花费时间:704 毫秒
公平重入锁  count value:500000
公平重入锁  花费时间:22796 毫秒
```

可以看出不公平重入锁需要的时间小于内部锁，公平重入锁需要的时间最多。

把 threadNum 修改为 500，loopNum=100；

运行结果如下：

```
内部锁  count value:50000
内部锁  花费时间:47 毫秒
不公平重入锁  count value:50000
不公平重入锁  花费时间:47 毫秒
公平重入锁  count value:50000
```

```
公平重入锁  花费时间:953 毫秒
```

threadNum=2000, loopNum=100；运行结果如下

```
内部锁  count value:200000
内部锁  花费时间:484 毫秒
不公平重入锁  count value:200000
不公平重入锁  花费时间:125 毫秒
公平重入锁  count value:200000
公平重入锁  花费时间:7500 毫秒
```

threadNum=2000, loopNum=1000；运行结果如下

```
内部锁  count value:2000000
内部锁  花费时间:921 毫秒
不公平重入锁  count value:2000000
不公平重入锁  花费时间:750 毫秒
公平重入锁  count value:2000000
公平重入锁  花费时间:57813 毫秒
```

从上面的运行结果可以看出，非公平重入锁的性能最好，公平重入锁的性能最差。在线程数比较少的情况下，内部锁和非公平重入锁的性能相当。

ReentrantLock 还有两个比较重要的方法是：**tryLock**()和 tryLock(long timeout, TimeUnit unit)。tryLock()仅在调用时锁未被另一个线程保持的情况下，才获取该锁。后者如果锁在给定等待时间内没有被另一个线程持有，且当前线程未被中断，则获取该锁。其他方法详细看 JDK 文档。

3.5.2 ReadWriteLock

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。

所有 ReadWriteLock 实现都必须保证 writeLock 操作的内存同步效果也要保持与相关 readLock 的联系。也就是说，成功获取读锁的线程会看到写入锁之前版本所做的所有更新。

与互斥锁相比，读-写锁允许对共享数据进行更高级别的并发访问。虽然一次只有一个线程（writer 线程）可以修改共享数据，但在许多情况下，任何数量的线程可以同时读取共享数据（reader 线程），读-写锁利用了这一点。从理论上讲，与互斥锁相比，使用读-写锁所允许的并发性增强将带来更大的性能提高。在实践中，只有在多处理器上并且只在访问模式适用于共享数据时，才能完全实现并发性增强。

与互斥锁相比，使用读-写锁能否提升性能则取决于读写操作期间读取数据相对于修改数据的频率，以及数据的争用——即在同一时间试图对该数据执行读取或写入操作的线程数。例如，某个最初用数据填充并且之后不经常对其进行修改的 collection，因为经常对其进行搜索（比如搜索某种目录），所以这样的 collection 是使用读-写锁的理想候选者。但是，如果数据更新变得频繁，数据在大部分时间都被独占锁，这时，就算存在并发性增强，也是微不足道的。更进一步地说，如果读取操作所用时间太短，则读-写锁实现（它本身就比互斥锁复杂）的开销将成为主要的执行成本，在许多读-写锁实现仍然通过一小段代码将所有线程序列化时更是如此。最终，只有通过分析和测量，才能确定应用程序是否适合使用读-写锁。

下面是一个使用读写锁的例子，创建几个写线程和读线程对 HashMap 中数据进行操作。读线程的个数多于写线程，也就是说读取数据的频率高于修改数据的频率。使用读写锁比合适。

```
package locks.readwritelock;
import java.util.Calendar;
import java.util.Map;
import java.util.TreeMap;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantReadWriteLock;
public class ReadWriteLockDemo {
    // 可重入读写锁
    private ReentrantReadWriteLock lock = null;
    private Lock readLock = null;// 读锁
    private Lock writeLock = null;// 写锁
    public int key = 100;
    public int index = 100;
    public Map<Integer, String> dataMap = null;// 线程共享数据
    public ReadWriteLockDemo() {
        lock = new ReentrantReadWriteLock(true);
        readLock = lock.readLock();
        writeLock = lock.writeLock();
        dataMap = new TreeMap<Integer, String>();
    }
    public static void main(String[] args) {
        ReadWriteLockDemo tester = new ReadWriteLockDemo();
        // 第一次获取锁
        tester.writeLock.lock();
        System.out
                .println(Thread.currentThread().getName() + " get writeLock.");
```

```
            // 第二次获取锁，应为是可重入锁
            tester.writeLock.lock();
            System.out
                    .println(Thread.currentThread().getName() + " get writeLock.");
            tester.readLock.lock();
            System.out.println(Thread.currentThread().getName() + " get readLock");
            tester.readLock.lock();
            System.out.println(Thread.currentThread().getName() + " get readLock");
            tester.readLock.unlock();
            tester.readLock.unlock();
            tester.writeLock.unlock();
            tester.writeLock.unlock();
            tester.test();
        }
        public void test() {
            // 读线程比写线程多
            for (int i = 0; i < 10; i++) {
                new Thread(new reader(this)).start();
            }
            for (int i = 0; i < 3; i++) {
                new Thread(new writer(this)).start();
            }
        }
        public void read() {
            // 获取锁
            readLock.lock();
            try {
                if (dataMap.isEmpty()) {
                    Calendar now = Calendar.getInstance();
                    System.out.println(now.getTime().getTime() + " R "
                            + Thread.currentThread().getName()
                            + " get key, but map is empty.");
                }
                String value = dataMap.get(index);
                Calendar now = Calendar.getInstance();
                System.out.println(now.getTime().getTime() + " R "
                        + Thread.currentThread().getName() + " key = " + index
                        + " value = " + value + " map size = " + dataMap.size());
                if (value != null) {
                    index++;
                }
            } finally {
                // 释放锁
                readLock.unlock();
```

```
            }
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        public void write() {
            writeLock.lock();
            try {
                String value = "value" + key;
                dataMap.put(new Integer(key), value);
                Calendar now = Calendar.getInstance();
                System.out.println(now.getTime().getTime() + " W "
                        + Thread.currentThread().getName() + " key = " + key
                        + " value = " + value + " map size = " + dataMap.size());
                key++;
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            } finally {
                writeLock.unlock();
            }
        }
    }
    class reader implements Runnable {
        private ReadWriteLockDemo tester = null;
        public reader(ReadWriteLockDemo tester) {
            this.tester = tester;
        }
        public void run() {
            Calendar now = Calendar.getInstance();
            System.out.println(now.getTime().getTime() + " R "
                    + Thread.currentThread().getName() + " started");
            for (int i = 0; i < 10; i++) {
                tester.read();
            }
        }
    }
    class writer implements Runnable {
        private ReadWriteLockDemo tester = null;
        public writer(ReadWriteLockDemo tester) {
```

```
            this.tester = tester;
        }
        public void run() {
            Calendar now = Calendar.getInstance();
            System.out.println(now.getTime().getTime() + " W "
                    + Thread.currentThread().getName() + " started");
            for (int i = 0; i < 10; i++) {
                tester.write();
            }
        }
    }
```

运行结果如下：

```
main get writeLock.
main get writeLock.
main get readLock
main get readLock
1235978402187 R Thread-3 started
1235978402187 W Thread-12 started
1235978402187 R Thread-8 started
1235978402187 R Thread-7 started
1235978402187 R Thread-5 started
1235978402187 R Thread-4 started
1235978402187 W Thread-10 started
1235978402187 R Thread-2 started
1235978402187 R Thread-6 started
1235978402187 R Thread-1 started
1235978402187 R Thread-0 started
1235978402187 W Thread-11 started
1235978402187 R Thread-9 started
1235978402187 R Thread-3 get key, but map is empty.
1235978402187 R Thread-3 key = 100 value = null map size = 0
1235978402187 W Thread-12 key = 100 value = value100 map size = 1
1235978402687 R Thread-5 key = 100 value = value100 map size = 1
1235978402687 R Thread-4 key = 100 value = value100 map size = 1
1235978402687 R Thread-7 key = 100 value = value100 map size = 1
1235978402687 R Thread-8 key = 102 value = value100 map size = 1
1235978402687 W Thread-10 key = 101 value = value101 map size = 2
.......
```

3.6 Fork-Join 框架

在 JDK 7 中，java.util.concurrent 包的新增功能之一是一个 fork-join 风格的并行分解框架。fork-join 概念提供了一种分解多个算法的自然机制，可以有效地应用硬件并行性。[12,13]

JDK7 中还未正式发布，目前提供的开发版本中还为包含相关 API，是 JSR166y 的一部分。主要参考 IBM DWs 上面的文章，相关代码仅供参考。

3.6.1 应用 Fork-Join

语言、库和框架形成了我们编写程序的方式。Alonzo Church 早在 1934 年就曾表明，所有已知的计算性框架对于它们所能表示的程序集都是等价的，程序员实际编写的程序集是由特定语言形成的，而编程模型（由语言、库和框架驱动）可以简化这些语言的表达。

另一方面，一个时代的主流硬件平台形成了我们创建语言、库和框架的方法。Java 语言从一开始就能够支持线程和并发性；该语言包括像 synchronized 和 volatile 这样的同步原语，而类库包含像 Thread 这样的类。然而，1995 年流行的并发原语反映了当时的硬件现状：大多数商用系统根本没有提供并行性，甚至最昂贵的系统也只提供了有限的并行性。当时，线程主要用来表示异步，而不是并发，而这些机制已足够满足当时的需求了。

随着多处理器系统价格降低，更多的应用程序需要使用这些系统提供的硬件并行性。而且程序员们发现，使用 Java 语言提供的低级原语和类库编写并发程序非常困难且容易出错。在 Java 5 中，java.util.concurrent 包被添加到 Java 平台，它提供了一组可用于构建并发应用程序的组件：并发集合、队列、信号量、锁存器（latch）、线程池等等。这些机制非常适合用于粗任务粒度的程序；应用程序只需对工作进行划分，使并发任务的数量不会持续少于可用的处理器数量。通过将对单个请求的处理用作 Web 服务器、邮件服务器或数据库服务器的工作单元，应用程序通常能满足这种需求，因此这些机制能够确保充分利用并行硬件。

技术继续发展，硬件的趋势非常清晰；摩尔定律表明不会出现更高的时钟频率，但是每个芯片上会集成更多的内核。很容易想象让十几个处理器繁忙地处理一个粗粒度的任务范围，比如一个用户请求，但是这项技术不会扩大到数千个处理器。在很短一段时间内流量可能会呈指数级增长，但最终硬件趋势将会占上风。当跨入多内核时代时，我们需要找到更细粒度的并行性，否则将面临处理器处于空闲的风险，即使还有许多工作需要处理。如果希望

跟上技术发展的脚步，软件平台也必须配合主流硬件平台的转变。最终，Java 7 将会包含一种框架，用于表示某种更细粒度并行算法的类：fork-join 框架。

如今，大多数服务器应用程序将用户请求-响应处理作为一个工作单元。服务器应用程序通常会运行比可用的处理器数量多很多的并发线程或请求。这是因为在大多数服务器应用程序中，对请求的处理包含大量 I/O，这些 I/O 不会占用太多的处理器（所有网络服务器应用程序都会处理许多的套接字 I/O，因为请求是通过套接字接收的；也会处理大量磁盘（或数据库）I/O）。如果每个任务的 90% 的时间用来等待 I/O 完成，您将需要 10 倍于处理器数量的并发任务，才能充分利用所有的处理器。随着处理器数量增加，可能没有足够的并发请求保持所有处理器处于繁忙状态。但是，仍有可能使用并行性来改进另一种性能度量：用户等待获取响应的时间。

一个典型网络服务器应用程序的例子是，考虑一个数据库服务器。当一个请求到达数据库服务器时，需要经过一连串的处理步骤。首先，解析和验证 SQL 语句。然后必须选择一个查询计划；对于复杂查询，数据库服务器将会评估许多不同的候选计划，以最小化预期的 I/O 操作数量。搜索查询计划是一种 CPU 密集型任务；在某种情况下，考虑过多的候选计划将会产生负面影响，但是如果候选计划太少，所需的 I/O 操作肯定比实际数量要多。从磁盘检索到数据之后，也许需要对结果数据集进行更多的处理；查询可能包含聚合操作，比如 SUM、AVERAGE，或者需要对数据集进行排序。然后必须对结果进行编码并返回到请求程序。

就像大多数服务器请求一样，处理 SQL 查询涉及到计算和 I/O。虽然添加额外的 CPU 不会减少完成 I/O 的时间（但是可以使用额外的内存，通过缓存以前的 I/O 操作结果来减少 I/O 数量），但是可以通过并行化来缩短请求处理的 CPU 密集型部分（比如计划评估和排序）的处理时间。在评估候选的查询计划时，可以并行评估不同的计划；在排序数据集时，可以将大数据集分解成更小的数据集，分别进行排序然后再合并。这样做会使用户觉得性能得到了提升，因为会更快收到结果（即使总体上可能需要更多工作来服务请求）。

合并排序是分治（ divide-and-conquer） 算法的一个例子，在这种算法中将一个问题递归分解成子问题，再将子问题的解决方案组合得到最终结果。并行分解方法常常称作 fork-join，因为执行一个任务将首先分解（fork）为多个子任务，然后再合并（join）（完成后）。

fork-join 框架支持几种风格的 ForkJoinTasks，包括那些需要显式完成的，以及需要循环执行的。下面程序是一个从大型数组中选择最大值的问题，使用的 RecursiveAction 类直

接支持 non-result-bearing 任务的并行递归分解风格；RecursiveTask 类解决 result-bearing 任务的相同问题（其他 fork-join 任务类包括 CyclicAction、AsyncAction 和 LinkedAsyncAction；要获得关于如何使用它们的更多细节，请查阅 Javadoc）。

下面的程序仅供参考，不一定能运行。

```
package forkjoin;
public class SelectMaxProblem {
    private final int[] numbers;
    private final int start;
    private final int end;
    public final int size = 1000;
    public SelectMaxProblem(int[] numbers2, int i, int j) {
        this.numbers = numbers2;
        this.start = i;
        this.end = j;
    }
    public int solveSequentially() {
        int max = Integer.MIN_VALUE;
        for (int i = start; i < end; i++) {
            int n = numbers[i];
            if (n > max)
                max = n;
        }
        return max;
    }
    public SelectMaxProblem subproblem(int subStart, int subEnd)
{
        return new SelectMaxProblem(numbers, start + subStart, start
+ subEnd);
    }
}
```

```
package forkjoin;
import jsr166y.ForkJoinPool;
public class MaxWithFJ {
    private final int threshold;
    private final SelectMaxProblem problem;
    public int result;
    public MaxWithFJ(SelectMaxProblem problem, int threshold) {
        this.problem = problem;
        this.threshold = threshold;
    }
    protected void compute() {
```

```
        if (problem.size < threshold)
            result = problem.solveSequentially();
        else {
            int midpoint = problem.size / 2;
            MaxWithFJ left = new MaxWithFJ(problem.subproblem(0,
midpoint),
                    threshold);
            MaxWithFJ right = new
MaxWithFJ(problem.subproblem(midpoint + 1,
                    problem.size), threshold);
            coInvoke(left, right);
            result = Math.max(left.result, right.result);
        }
    }
    public static void main(String[] args) {
        SelectMaxProblem problem = ….;
        int threshold = 500;
        int nThreads = 10;
        MaxWithFJ mfj = new MaxWithFJ(problem, threshold);
        ForkJoinExecutor fjPool = new ForkJoinPool(nThreads);
        fjPool.invoke(mfj);
        int result = mfj.result;
    }
}
```

使用传统的线程池来实现 fork-join 具有挑战性，因为 fork-join 任务将线程生命周期的大部分时间花费在等待其他任务上。这种行为会造成线程饥饿死锁（thread starvation deadlock），除非小心选择参数以限制创建的任务数量，或者池本身非常大。传统的线程池是为相互独立的任务设计的，而且设计中也考虑了潜在的阻塞、粗粒度任务。 fork-join 解决方案不会产生这两种情况。对于传统线程池的细粒度任务，也存在所有工作线程共享的任务队列发生争用的情况。

fork-join 框架通过一种称作工作窃取（work stealing） 的技术减少了工作队列的争用情况。每个工作线程都有自己的工作队列，这是使用双端队列（或者叫做 deque）来实现的（Java 6 在类库中添加了几种 deque 实现，包括 ArrayDeque 和 LinkedBlockingDeque）。当一个任务划分一个新线程时，它将自己推到 deque 的头部。当一个任务执行与另一个未完成任务的合并操作时，它会将另一个任务推到队列头部并执行，而不会休眠以等待另一任务完成（像 Thread.join() 的操作一样）。当线程的任务队列为空，它将尝试从另一个线程的 deque 的尾部 窃取另一个任务。

fork-join 方法提供了一种表示可并行化算法的简单方式，而不用提前了解目标系统将提供多大程度的并行性。所有的排序、搜索和数字算法都可以进行并行分解（以后，像 Arrays.sort() 这样的标准库机制将会使用 fork-join 框架，允许应用程序免费享有并行分解的益处）。随着处理器数量的增长，我们将需要在程序内部使用更多的并行性，以有效利用这些处理器；对计算密集型操作（比如排序）进行并行分解，使程序能够更容易利用未来的硬件。

3.6.2 应用 ParallelArray

随着处理器数量的增加，为了有效利用可用的硬件，我们需要识别并利用程序中更细粒度的并行性。最近几年中，选择粗粒度的任务边界（例如在 Web 应用程序中处理单一请求）和在线程池中执行任务，通常能够提供足够的并行性，实现可接受的硬件利用效率。但是如果要再进一步，就必须深入挖掘更多的并行性，以让硬件全速运转。一个成熟的并行领域就是大数据集中的排序和搜索。用 fork-join 可以很容易地表示这类问题。但是由于这些问题非常普遍，所以该类库提供了一种更简单的方法 — ParallelArray。

在主流服务器应用程序中，最适合更细粒度并行性的地方是数据集的排序、搜索、选择和汇总。其中的每个问题都可以用 divide-and-conquer 轻松地并行化，并能轻松地表示为 fork-join 任务。例如，要将对大数据集求平均值的操作并行化，可以递归地将大数据集分解成更小的数据集 — 就像在合并排序中做的那样 — 对子集求均值，然后在合并步骤中求出各子集的平均值的加权平均值。

对于排序和搜索问题，fork-join 库提供了一种表示可以并行化的数据集操作的非常简单的途径：ParallelArray 类。其思路是：用 ParallelArray 表示一组结构上类似的数据项，用 ParallelArray 上的方法创建一个对分解数据的具体方法的描述。然后用该描述并行地执行数组操作（幕后使用的是 fork-join 框架）。这种方法支持声明性地指定数据选择、转换和后处理操作，允许框架计算出合理的并行执行计划，就像数据库系统允许用 SQL 指定数据操作并隐藏操作的实现机制一样。ParallelArray 的一些实现可用于不同的数据类型和大小，包括对象数组和各种原语组成的数组。

ParallelArray 支持以下基本操作：

1） 筛选：选择计算过程中包含的元素子集。

2） 应用：将一个过程应用到每个选中的元素。

3） 映射：将选中的元素转换为另一种形式（例如从元素中提取数据字段）。

4） 替换：将每个元素替换为由它派生的另一个元素，创建新的并行数组。

此技术与映射类似，但是形成新的 ParallelArray，可以在其上执行进一步查询。替换的一种情况是排序，将元素替换为不同的元素，从而对其进行排序（内置的 sort() 方法可用于此操作）。另一种特殊情况是 cumulate() 方法，该方法根据指定的组合操作用累积值替换每个元素。替换操作也可用于组合多个 ParallelArray，例如创建一个 ParallelArray，其元素为对并行数组 a 和 b 执行 a[i]+b[i] 操作得到的值。

5） 汇总：将所有值组合为一个值，例如计算总和、平均值、最小值或最大值。

ParallelArray 并不是一种通用的内存中数据库，也不是一种指定数据转换和提取的通用机制；它只是用于简化特定范围的数据选择和转换操作的表达方式，以将这些操作轻松、自动地并行化。所以，它存在一些局限性；例如，必须在映射操作之前指定筛选操作。（允许多个筛选操作，但是将它们组合成一个复合筛选操作通常会更有效）。它的主要目的是使开发人员不用思考如何将工作并行化；如果能够用 ParallelArray 提供的操作表示转换，那么就能轻松实现并行化。

ParallelArray 提供了一种不错的方法，可用于声明性地指定数据集上的筛选、处理和聚合操作，还方便自动并行化。但是，尽管它的语法比使用原始的 for-join 库更容易表达，但还是有些麻烦；每个筛选器、映射器、reducer 通常被指定为内部类。Java 7 可能会在 Java 语言中加入闭包；支持闭包的一种说法是：闭包使得小段代码 — 例如 ParallelArray 中的筛选器、映射器、reducer —的表示更加紧凑。

随着可用的处理器数量增加，我们需要发现程序中更细粒度的并行性来源。最有吸引力候选方案之一是聚合数据操作——排序、搜索和汇总。JDK 7 中将引入的 fork-join 库提供了一种 “轻松表示” 某类可并行化算法的途径，从而让程序能够在一些硬件平台上有效运行。通过声明性地描述想要执行的操作，然后让 ParallelArray 确定具体的执行方法，fork-join 库的 ParallelArray 组件使并行聚合操作的表示变得更加简单。

由于 JDK 还未发布，没有编写能够实际运行的程序。

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