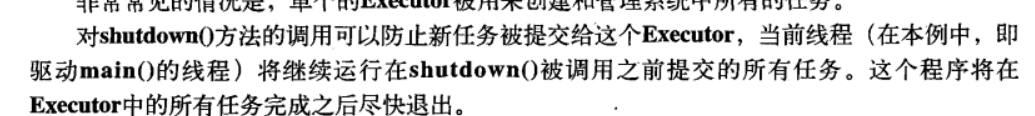
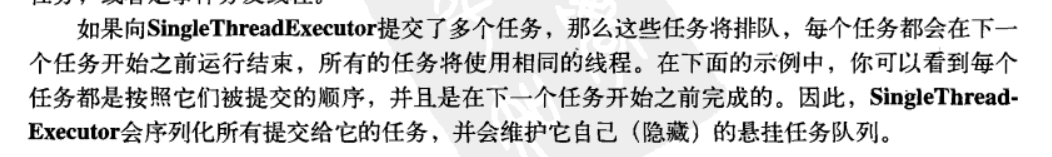
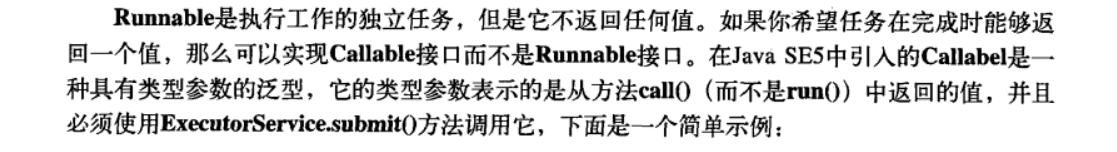


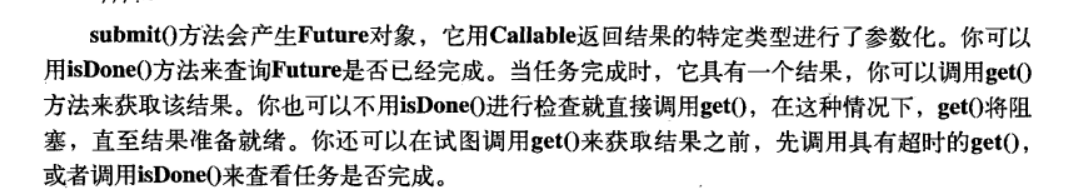
线程池创建线程

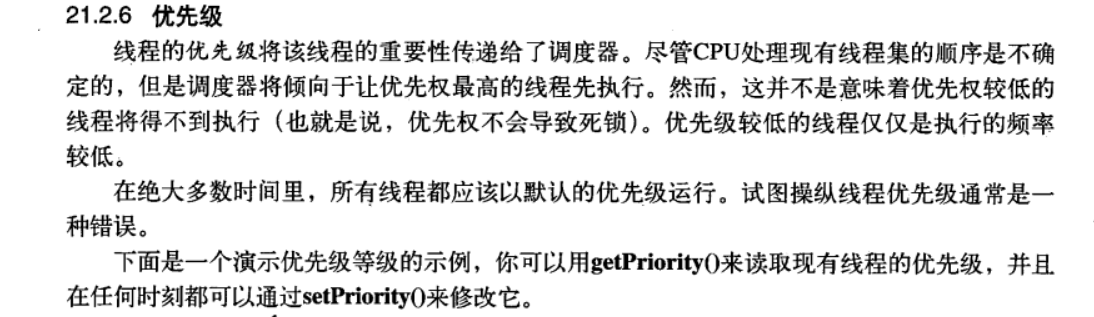
package com.cxy.rabbitmq.thread;  
  
import java.util.ArrayList;  
import java.util.Date;  
import java.util.List;  
import java.util.concurrent.\*;  
  
*/\*\*\*  
 \* Created by CaoXingYun on 2018/11/01  
 \*\*/*public class ExecutorServiceTest {  
 public static void main(String[] args) throws ExecutionException, InterruptedException {  
 System.*out*.println("----程序开始运行----");  
 Date time1 = new Date();  
 int taskSize = 5;  
 //创建一个线程池  
 */\*\*\*  
 \* Executor executor = Executors.newFixedThreadPool(10);  
 \* Runnable task = new Runnable() {  
 \** ***@Override*** *\* public void run() {  
 \* System.out.println("task over");  
 \* }  
 \* };  
 \* executor.execute(task);  
 \* ExecutorService 继承接口 Executor ,Executor的execute方法  
 \* 并发编程的一种编程方式是把任务拆分为一些列的小任务，即Runnable，然后在提交给一个Executor执行，  
 \* Executor.execute(Runnalbe)  
 \* 。Executor在执行时使用内部的线程池完成操作  
 \* 上述代码中Executors类，提供了一系列工厂方法用于创先线程池，返回的线程池都实现了ExecutorService接口。  
 \* public static ExecutorService newFixedThreadPool(int nThreads)   
 \* 创建固定数目线程的线程池。  
 \* public static ExecutorService newCachedThreadPool()   
 \* 创建一个可缓存的线程池，调用execute 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。  
 \* public static ExecutorService newSingleThreadExecutor()   
 \* 创建一个单线程化的Executor。  
 \* public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)   
 \* 创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代Timer类。  
 \*/* ExecutorService executorService = Executors.*newFixedThreadPool*(taskSize);  
 List<Future> futures = new ArrayList<>();  
 for (int i = 0; i < taskSize; i++) {  
 */\*\*\*  
 \* 可返回值的任务必须实现Callable接口，类似的，无返回值的任务必须Runnable接口。  
 \* 执行Callable任务后，可以获取一个Future的对象，在该对象上调用get就可以获取到Callable任务返回的Object了，  
 \* 再结合线程池接口ExecutorService就可以实现传说中有返回结果的多线程了。  
 \*/* Callable callable = new MyCallable(i+"");  
 // 执行任务并获取Future对象  
 Future future = executorService.submit(callable);  
 */\*\*\*  
 \* ExecutorService submit callable 后，最终会调用Executor 的execute来执行，使用内部的线程池来完成操作  
 \* ExecutoreService提供了submit()方法，传递一个Callable，或Runnable，返回Future。  
 \* 如果Executor后台线程池还没有完成Callable的计算，这调用返回Future对象的get()方法，会阻塞直到计算完成。  
 \*/* futures.add(future);  
 }  
 // 获取所有并发任务的运行结果  
 for(Future msg:futures){  
 System.*out*.println(""+msg.get().toString());  
 }  
 Date time2 = new Date();  
 System.*out*.println("程序结束"+"时间为"+ (time2.getTime() - time1.getTime())+"毫秒");  
 }  
 */\*\*\*  
 \* 运行结果  
 \* ----程序开始运行----  
 \* ---->1任务开始  
 \* ---->0任务开始  
 \* ---->2任务开始  
 \* ---->3任务开始  
 \* ---->4任务开始  
 \* ---->1任务终止  
 \* ---->3任务终止  
 \* ---->2任务终止  
 \* ---->4任务终止  
 \* ---->0任务终止  
 \* 0任务返回运行结果,当前任务时间【1000毫秒】  
 \* 1任务返回运行结果,当前任务时间【1000毫秒】  
 \* 2任务返回运行结果,当前任务时间【1000毫秒】  
 \* 3任务返回运行结果,当前任务时间【1000毫秒】  
 \* 4任务返回运行结果,当前任务时间【1000毫秒】  
 \* 程序结束时间为1046毫秒  
 \*/  
 /\*\*\*  
 \* 线程池中的任务不是按提交到线程池的顺序来顺序执行的，不同线程竞争CUP资源，谁先竞争到谁先执行。  
 \* 各线程利用多核CPU并行执行  
 \* 主线程只负责将线程添加到线程池中，线程就结束了，其它线程各自执行任务，与主线程无关。  
 \*/*}  
  
class MyCallable implements Callable<Object> {  
 private String taskNum;  
  
 public MyCallable(String taskNum){  
 this.taskNum = taskNum;  
 }  
  
 @Override  
 public Object call() throws Exception {  
 System.*out*.println("---->"+taskNum+"任务开始");  
 Date now = new Date();  
 Thread.*sleep*(1000);  
 Date t = new Date();  
 long time = t.getTime() - now.getTime();  
 System.*out*.println("---->"+taskNum+"任务终止");  
 return taskNum + "任务返回运行结果,当前任务时间【" + time + "毫秒】";  
 }  
}

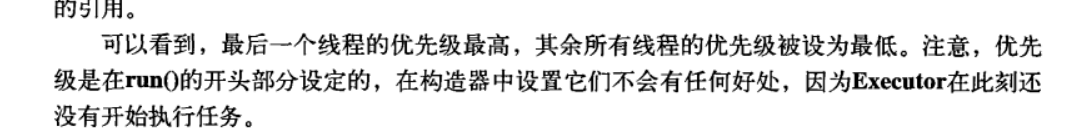


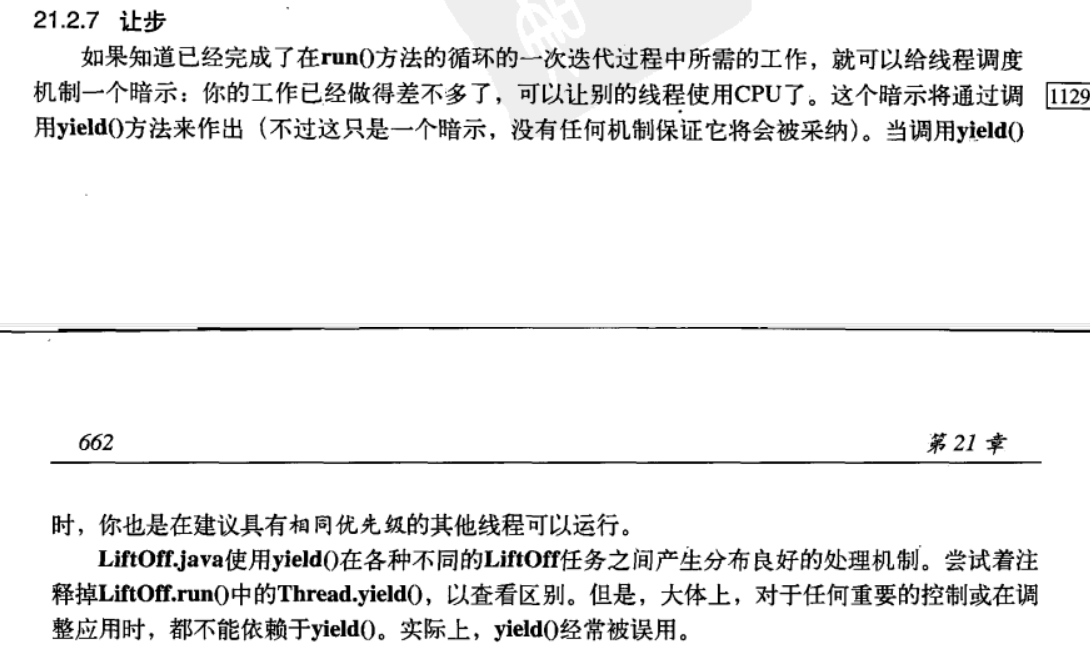


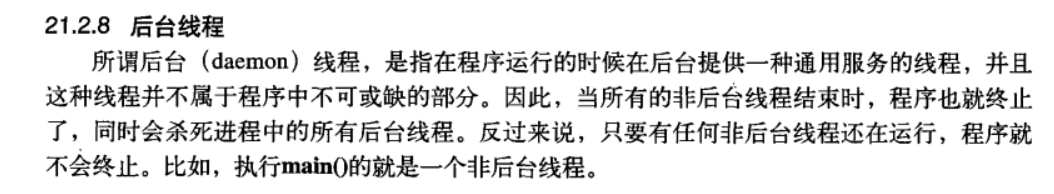












有非守护线程，程序就不能停止，只有当所有非守护线程都结束后，不管你有没有守护线程，程序都会结束



**sleep()，wait()，yield()和join()方法的区别**

sleep()

　　sleep()方法需要指定等待的时间，它可以让当前正在执行的线程在指定的时间内暂停执行，进入阻塞状态，该方法既可以让其他同优先级或者高优先级的线程得到执行的机会，也可以让低优先级的线程得到执行机会。但是sleep()方法不会释放“锁标志”，也就是说如果有synchronized同步块，其他线程仍然不能访问共享数据。

wait()

　　wait()方法需要和notify()及notifyAll()两个方法一起介绍，这三个方法用于协调多个线程对共享数据的存取，所以必须在synchronized语句块内使用，也就是说，调用wait()，notify()和notifyAll()的任务在调用这些方法前必须拥有对象的锁。注意，它们都是Object类的方法，而不是Thread类的方法。

　　wait()方法与sleep()方法的不同之处在于，wait()方法会释放对象的“锁标志”。当调用某一对象的wait()方法后，会使当前线程暂停执行，并将当前线程放入对象等待池中，直到调用了notify()方法后，将从对象等待池中移出任意一个线程并放入锁标志等待池中，只有锁标志等待池中的线程可以获取锁标志，它们随时准备争夺锁的拥有权。当调用了某个对象的notifyAll()方法，会将对象等待池中的所有线程都移动到该对象的锁标志等待池。

　　除了使用notify()和notifyAll()方法，还可以使用带毫秒参数的wait(long timeout)方法，效果是在延迟timeout毫秒后，被暂停的线程将被恢复到锁标志等待池。

　　此外，wait()，notify()及notifyAll()只能在synchronized语句中使用，但是如果使用的是ReenTrantLock实现同步，该如何达到这三个方法的效果呢？解决方法是使用ReenTrantLock.newCondition()获取一个Condition类对象，然后Condition的await()，signal()以及signalAll()分别对应上面的三个方法。

yield()

　　yield()方法和sleep()方法类似，也不会释放“锁标志”，区别在于，它没有参数，即yield()方法只是使当前线程重新回到可执行状态，所以执行yield()的线程有可能在进入到可执行状态后马上又被执行，另外yield()方法只能使同优先级或者高优先级的线程得到执行机会，这也和sleep()方法不同。

join()

　　join()方法会使当前线程等待调用join()方法的线程结束后才能继续执行，例如：

package concurrent;

public class TestJoin {

public static void main(String[] args) {

Thread thread = new Thread(new JoinDemo());

thread.start();

for (int i = 0; i < 20; i++) {

System.out.println("主线程第" + i + "次执行！");

if (i >= 2)

try {

// t1线程合并到主线程中，主线程停止执行过程，转而执行t1线程，直到t1执行完毕后继续。

thread.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

class JoinDemo implements Runnable {

@Override

public void run() {

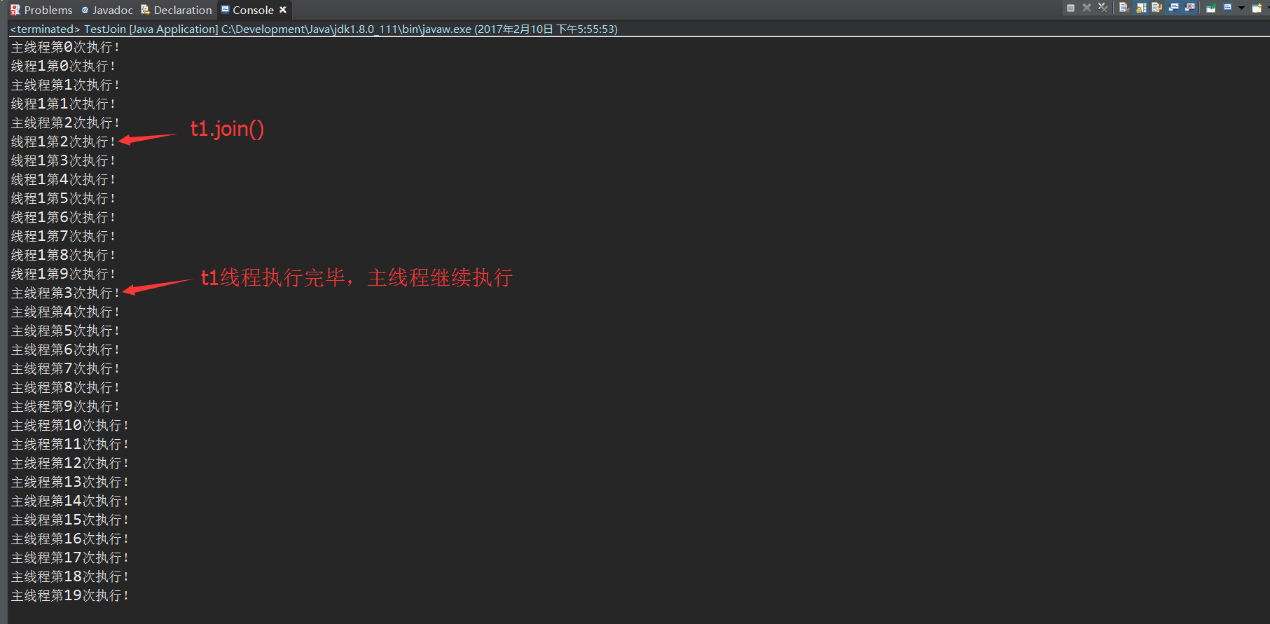
for (int i = 0; i < 10; i++) {

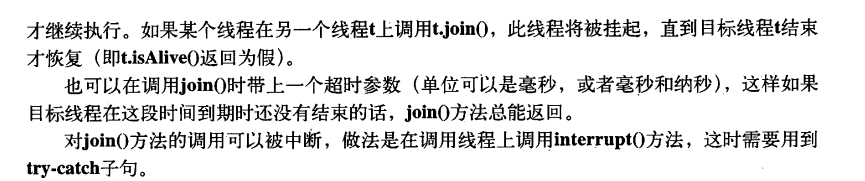
System.out.println("线程1第" + i + "次执行！");

}

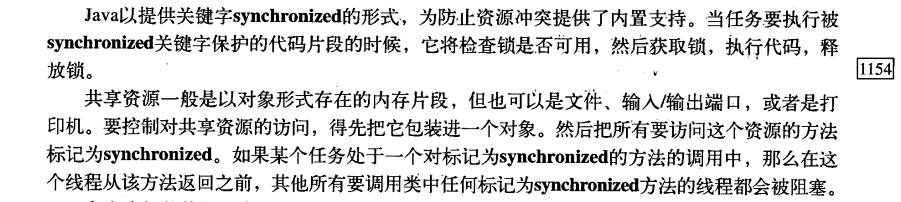
}

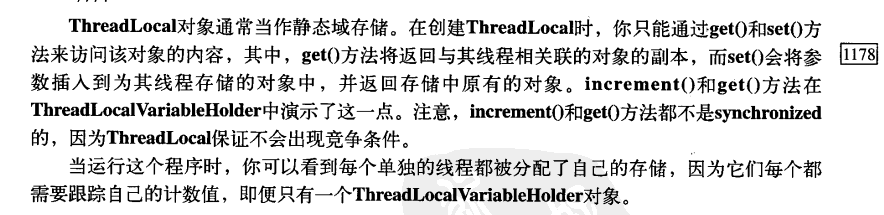
}





资源同步





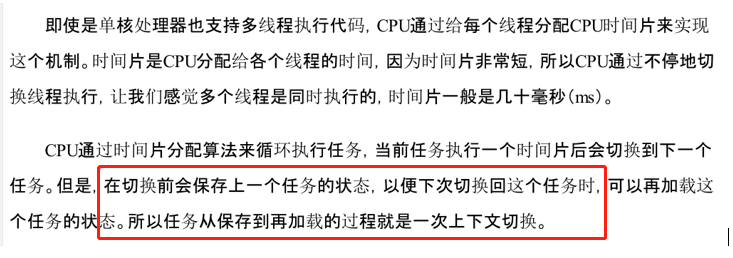
生产者消费者模型

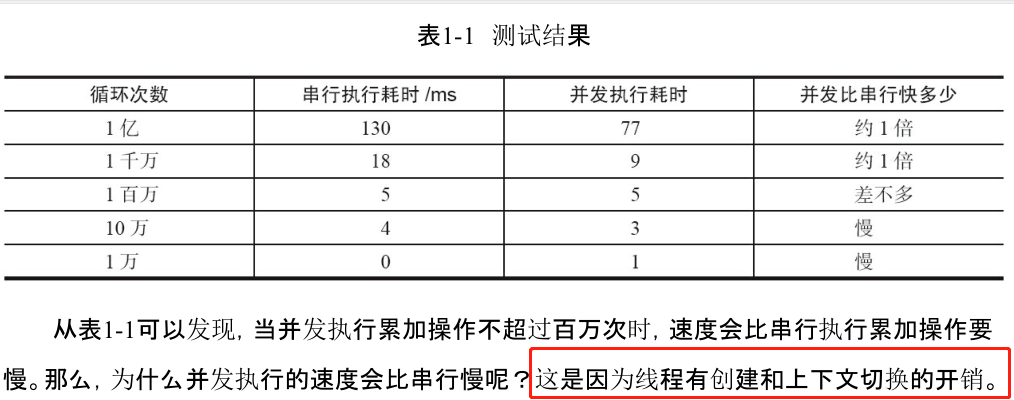
开启生产消费者线程

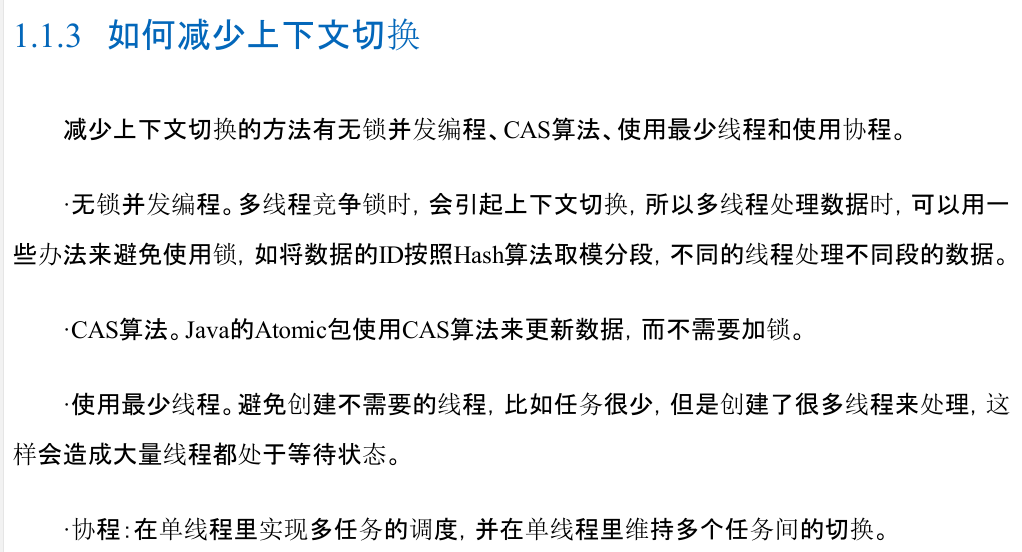
传入共享的对象

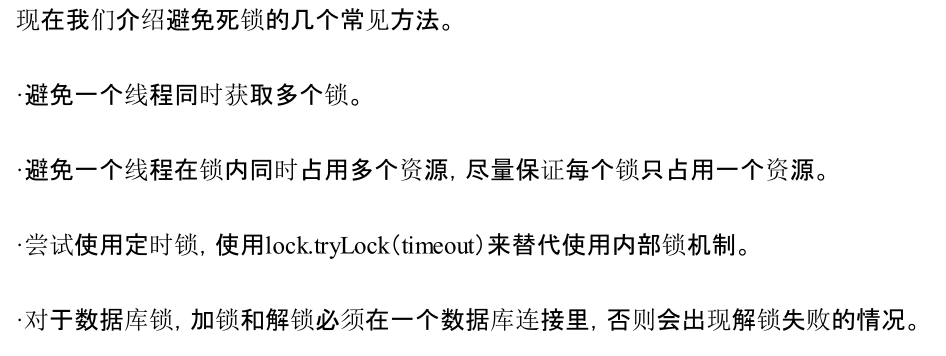
线程的上下文切换

上下文的切换会影响多线程的执行效率。



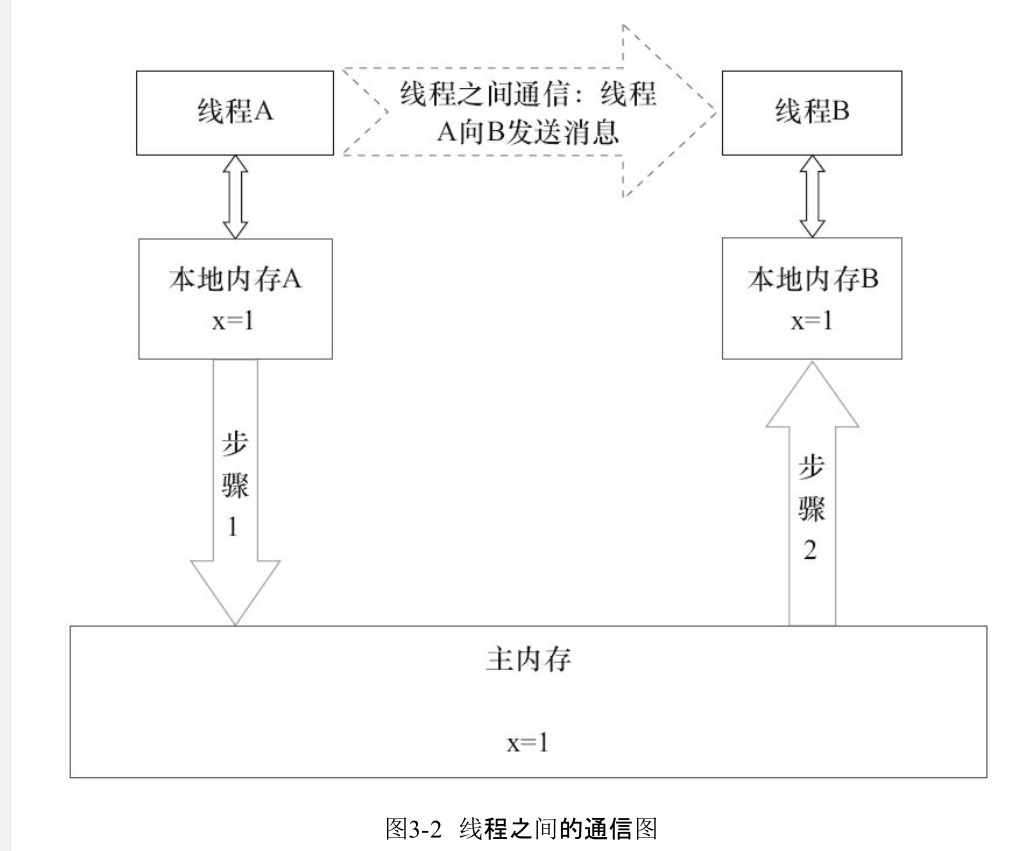


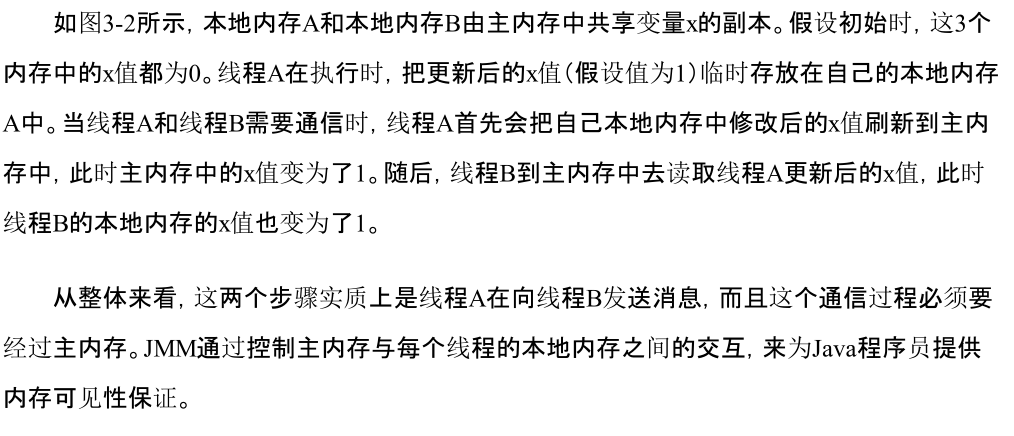


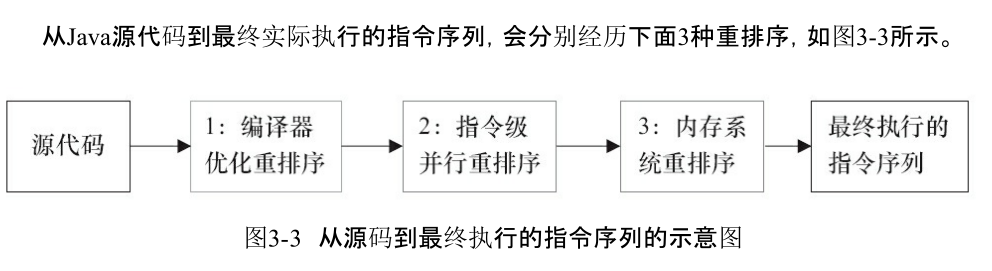












Volatile

# 

# 

# 

# 

# Synchronized

# 

# 

# 

# 

# 重入锁

# 

# 

# 线程的初始化

# 

# 线程同步

# 对于同步块的实现使用了monitorenter和monitorexit指令，而同步方法则

# 是依靠方法修饰符上的ACC\_SYNCHRONIZED来完成的。无论采用哪种方式，其本质是对一

# 个对象的监视器（monitor）进行获取，而这个获取过程是排他的，也就是同一时刻只能有一个

# 线程获取到由synchronized所保护对象的监视器。

# 任意一个对象都拥有自己的监视器，当这个对象由同步块或者这个对象的同步方法调用

# 时，执行方法的线程必须先获取到该对象的监视器才能进入同步块或者同步方法，而没有获

# 取到监视器（执行该方法）的线程将会被阻塞在同步块和同步方法的入口处，进入BLOCKED

# 状态。

# 

# 等待/通知机制，是指一个线程A调用了对象O的wait()方法进入等待状态，而另一个线程B

# 调用了对象O的notify()或者notifyAll()方法，线程A收到通知后从对象O的wait()方法返回，进而

# 执行后续操作。上述两个线程通过对象O来完成交互，而对象上的wait()和notify/notifyAll()的

# 关系就如同开关信号一样，用来完成等待方和通知方之间的交互工作。

# 使用wait()、notify()和notifyAll()时需要先对调用对象加锁。

# 

# 

# 

# 等待超时模式

# 调用一个方法时等待一段时间（一般来说是给

# 定一个时间段），如果该方法能够在给定的时间段之内得到结果，那么将结果立刻返回，反之，

# 超时返回默认结果。

# 

# 

# 锁 lock

# 

# 读写锁

# 

# 读写锁实例

# 

# ConcurrentHashMap是线程安全且高效的HashMap

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 线程池

# 

# 

# 

# ThreadPoolExecutor采取上述步骤的总体设计思路，是为了在执行execute()方法时，尽可能

# 地避免获取全局锁（那将会是一个严重的可伸缩瓶颈）。在ThreadPoolExecutor完成预热之后

# （当前运行的线程数大于等于corePoolSize），几乎所有的execute()方法调用都是执行步骤2，而

# 步骤2不需要获取全局锁。

# 