多线程

2016/11/20 10:57 by Vincent

一、基本概念

1.1 什么是线程？它与进程的区别？

进程	线程
1) 进程是指一段正在执行的程序。 2) 一个进程可以拥有多个线程。 3) 在操作系统中，程序的执行都是以进程为单位。 4) 每个进程通常都会有多个线程互不影响并发执行。	1) 线程被称为轻量级进程，他是程序执行的最小单元。 2) 各个线程之间共享程序的内存空间（代码段、数据段和推空间）及一些进程级的资源。 3) 各个线程拥有自己的栈空间。

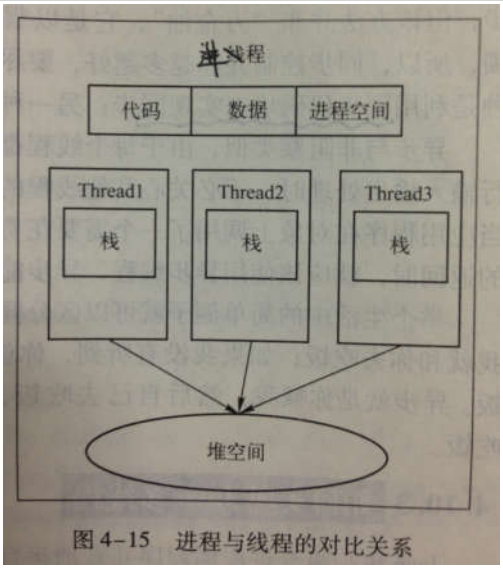


图 4-15 进程与线程的对比关系

1.2 为什么要使用多线程？

1. 使用多线程可以减少程序的响应时间。
2. 与进程相比线程的创建和切换开销要更小。
3. 多CPU或多核计算机本身具备执行线程的能力，因此多线程可以提高cpu的利用率。
4. 使用多线程能够简化程序的结构，使程序便于理解和维护。

1.3 Java中线程创建的方法？

1. 继承Thread类，重写run()方法

1) 继承Thread类，重写run()方法（线程体）
2) 创建Thread的子类对象，调用start()方法

以下程序为模拟龟兔赛跑:

```
/**通过继承thread来实现多线程
 * 1、自定义类继承Thread类并重现run()方法。
 * 2、创建自定义类的对象，并调用其start()方法。
 * @author Vincent
 *
 */
public class Race_thread extends Thread {
	@Override
	public void run() {
		for (int i = 0; i < 100; i++) {
			System.out.println("兔子在跑……"+i+"步");
		}
	}
}

class Tortoise_thread extends Thread {
	@Override
	public void run() {
		for (int i = 0; i < 100; i++) {
			System.out.println("乌龟在跑……"+i+"步");
		}
	}
}
```

```
public class RaceApp_thread {
	public static void main(String[] args) {
		//创建子类对象
		Race_thread rabbite=new Race_thread();
		Tortoise_thread tortoise=new Tortoise_thread();

		//调用子类对象的start方法
		rabbite.start();
		tortoise.start();

	for (int i = 0; i < 100; i++) {//主进程
		System.out.println("main---->"+i+"步");
	}
	}
}
```

2. 实现Runnable接口，实现run()方法

Java只支持单继承，如果继承了其他类，无法再继续继承Thread类，使用Runnable创建进程，既可以实现多线程也可以继承其它类。

1) 实现Runnable接口，实现run()方法

2) 创建Thread对象，自定义类类的对象为参数实例化Thread对象。
3) 调用Thread对象的start()方法。

模拟龟兔赛跑：

```
/**
 * 通过实现Runnable接口来实现多线程
 * 1、自定义类实现Runnable接口，实现run()。
 * 2、创建thread对象，以自定义类的对象为参数实例化thread对象。
 * 3、调用thread对象的start()方法。</br>
 *
 * 使用Runnable接口的优点：由于java只支持单继承，实现Runnable接口的同时还可以继承。
 * @author Vincent
 *
 */
public class Race_Runnable implements Runnable{
	@Override
	public void run() {
		for (int i = 0; i < 100; i++) {
			System.out.println("兔子在跑……"+i+"步");
		}
	}
}

class Tortoise_Runnable implements Runnable{
	@Override
	public void run() {
		for (int i = 0; i < 100; i++) {
			System.out.println("乌龟在跑……"+i+"步");
		}
	}
}
```

```
public class RaceApp_Runnable {
	public static void main(String[] args) {
		//创建自定义的对象
		Race_Runnable r=new Race_Runnable();
		Tortoise_Runnable t=new
		Tortoise_Runnable();

		//创建Thread类的对象
		Thread t1=new Thread(r);
		Thread t2=new Thread(t);

		//启动线程
		t1.start();
		t2.start();

		for(int i=0;i<100;i++){
			System.out.println("main--->"+i);
		}
	}
}
```

Thread类和Runnable接口之间是属于代理模式：

组成：
抽象角色：通过接口或抽象类声明真实角色实现的业务方法。
代理角色：实现抽象角色，是真实角色的代理，通过真实角色的业务逻辑方法来实现抽象方法，并可以附加自己的操作。
真实角色：实现抽象角色，定义真实角色所要实现的业务逻辑，供代理角色调用。

静态代理模式举例：

```
/**
 * 静态代理模式 设计模式
 * 1、真实角色， 相当于自定义类
 * 2、代理角色:要有真实角色的引用 , Thread
 * 3、二者要实现相同的接口, Runnable
 *
 * @author Vincent
 *
 */

// 抽象角色
interface Marry {
	public abstract void marry();
}

// 真实角色
class You implements Marry {

	@Override
	public void marry() {
		System.out.println("你结婚了……");
	}
}

// 代理角色
class WeddingCompany implements Marry {

	private Marry you;

	public WeddingCompany(Marry you) {
		this.you = you;
	}

	private void Before() {
```

抽象角色：

```
public
interface Runnable {
    /**
     * When an object implementing interface <code>Runnable</code> is used
     * to create a thread, starting the thread causes the object's
     * <code>run</code> method to be called in that separately executing
     * thread.
     * <p>
     * The general contract of the method <code>run</code> is that it may
     * take any action whatsoever.
     *
     * @see     java.lang.Thread#run()
     */
    public abstract void run();
}
```

代理角色：

```
public
class Thread implements Runnable {
    /* Make sure registerNatives is the first thing <clinit> does. */
    private static native void registerNatives();
    static {
        registerNatives();
    }
```

真实角色：

```
public class Race_Runnable implements Runnable{
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println("兔子在跑……"+i+"步");
        }
    }
}
```

```
    //创建自定义的对象
    Race_Runnable r=new Race_Runnable();
    Tortoise_Runnable t=new Tortoise_Runnable();
```

```
            System.out.println("布置结婚现场……");
        }

        private void After() {
            System.out.println("打扫结婚现场……");
        }

        @Override
        public void marry() {
            Before();
            you.marry();
            After();
        }

}

public class StaticProxy {
        public static void main(String[] args) {
            //创建真实角色
            You you=new You();
            //加入真实角色的引用
            WeddingCompany company=new WeddingCompany(you);
            //执行任务
            company.marry();
        }
}
```

```
        //创建Thread类的对象
        Thread t1=new Thread(r);
        Thread t2=new Thread(t);

        t1.start();
        t2.start();
```

3. 实现Callable接口，实现call()方法

使用Callable创建线程的好处：可以返回值+可以抛出异常。

1) 实现Callable接口，实现call()方法
2) ExecutorService threadPool=Executors.newFixedThreadPool(2); 利用该语句指明线程输。
3) 创建自定义类的对象
4) Future<Integer> future1=threadPool.submit(rabbit);启动线程
5) future1.get();获取返回值

模拟龟兔赛跑：

```
import java.util.concurrent.Callable;

/**创建线程类，模拟龟兔赛跑
 * 用Callable创建线程+重写call()方法，多用于服务器编程
 * 优点：
 * 1、线程体可以返回值
 * 2、线程体可以抛出异常
 * @author Vincent
 */
public class Race_Callable implements Callable<Integer>{

    private String name;
    private long time;
    private int step;
    private boolean flag=true;


    public Race_Callable(String name, long time) {
        super();
        this.name = name;
        this.time = time;
    }

    @Override
    public Integer call() throws Exception {
        while(flag){
            Thread.sleep(time);//线程延时
            step++;
        }
        return step;//线程返回值
    }

    /*省去get()/set()方法*/
}
```

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class RaceApp_Callable {
    public static void main(String[] args) throws InterruptedException, ExecutionException
    {
        //创建两个线程
        ExecutorService threadPool=Executors.newFixedThreadPool(2);

        Race_Callable rabbit=new Race_Callable("小兔子", 500);
        Race_Callable tortoise=new Race_Callable("小乌龟", 1000);

        //获取线程返回值
        Future<Integer> future1=threadPool.submit(rabbit);
        Future<Integer> future2=threadPool.submit(tortoise);

        Thread.sleep(3000);

        rabbit.setFlag(false);//停止线程
        tortoise.setFlag(false);

        int num1=future1.get();//获取返回值
        int num2=future2.get();

        System.out.println("小兔子跑了--->"+num1+"步");
        System.out.println("小乌龟跑了--->"+num2+"步");

        threadPool.shutdownNow();//停止服务
    }
}
```

```
<terminated> RaceApp_Callabl
小兔子跑了--->7步
小乌龟跑了--->4步
```

4、run()与start()方法的区别？

1) 通常，系统通过调用start()方法来启动线程，此时线程处于就绪状态，而非运行状态，意味着这个线程可以被JVM来调度执行。JVM通过调用线程类的run()方法来完成实际的操作，当run()方法结束后，线程就会终止。
2) 如果直接调用线程类的run()方法，run()方法会被当成一个普通方法被执行，无法达到多线程的效果。

5、线程信息的获取和设置

1）ID： 该属性表示每个线程的唯一标识；
- Thread对象.getId();

2）Name： 该属性存储每个线程的名称；
- Thread对象.getName()；
- Thread对象.setName(String name);

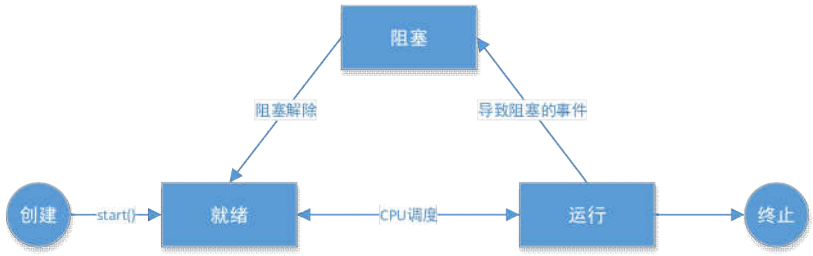
3）Priority： 该属性存储每个Thread对象的优先级。线程优先级分1到10十个级别，1表示最低优先级，10表示最高优先级。并不推荐修改线程的优先级，但是如果确实有这方面的需求，也可以尝试一下。
- Thread对象.setPriority(int newPriority)；
- Thread对象.getPriority()；

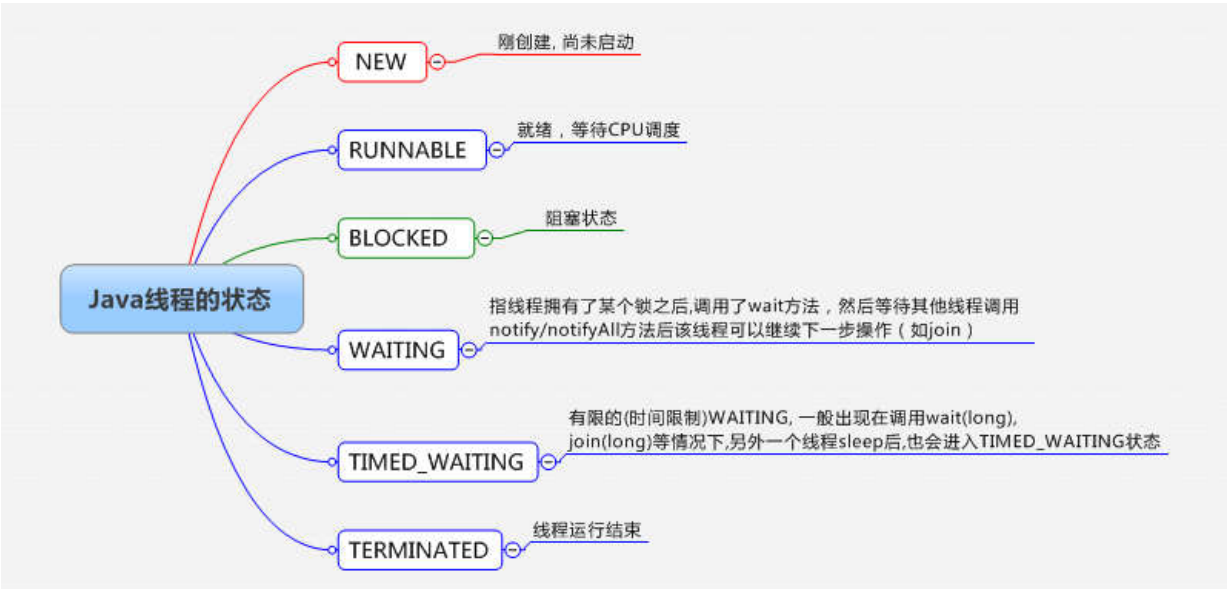
4)Status: 该属性存储线程的状态。线程共有六种不同的状态：新建（new）、运行（runnable）、阻塞（blocked）、等待（waiting）、限时等待（time waiting）或者终止（terminated）。线程的状态必定是其中一种。
- Java中的枚举类，public enum State { NEW, RUNNABLE,BLOCKED,WAITING,TIMED_WAITING, TERMINATED;}
- Thread对象.getState()；

二、线程管理

2.1 线程的状态？

1. 新建状态
2. 就绪状态（当调用了线程的start()方法之后，线程就处于就绪状态）
3. 运行状态
4. 阻塞状态（sleep(),等待I/O资源）
5. 终止状态





2.2 终止线程的方法有哪些？

1. 执行完run()方法之后线程即处于终止状态。
2. 通过设置flag标识来控制循环是否执行。
   1) 线程类中定义线程体使用的标识，flag=true。
   2) 线程体类使用while(flag)。
   3) 提供对外的方法，改变标识 public void stop(){this.flag=false;}。
   4) 外部根据条件调用stop()方法。
3. 调用interrupt()方法，抛出InterruptedException异常，使线程安全退出。

```
public class test {
        public static void main(String[] args) {
                Thread thread=new Thread(new Runnable() {
```

History Console
<terminated> test (3) [Java Appli

```
                @Override
                public void run() {
                        System.out.println("thread goes to sleep");
                        try {
                                Thread.sleep(5000);
                                System.out.println("thread finished");

                        } catch (InterruptedException e) {

                                System.out.println("thread is interupted");

                        }
                }
        });

        thread.start();
        thread.interrupt();
    }
}
```

```
thread goes to sleep
thread is interupted
```

2.3 阻塞

1. 当发生以下情况时，线程会进入阻塞状态：
   1) 调用sleep()方法主动放弃所占用的处理器资源。
   2) 线程调用了一个阻塞式的IO方法，该方法返回前，该线程被阻塞。
   3) 线程试图获得一个同步监视器，但该监视器正在被其它线程所持有。
   4) 线程在等待某个通知(notify)。
   5) 程序调用了线程的suspend方法将该线程挂起（此方法容易导致死锁，少用。）

2. 发生如下的情况可以解除上面的阻塞状态，让线程重新进入就绪状态：
   1) 调用sleep()方法的线程经过了指定的时间。
   2) 线程调用的阻塞式IO方法已经返回。
   3) 线程获得了视图取得的同步监视器。
   4) 线程正在等待某个通知，其它线程发出了一个通知。
   5) 处于挂起的线程被调用了resume()恢复方法。

2.4 控制线程

1. *join线程*

Thread提供了让一个线程等待另一个线程完成的方法：join()方法。当某个程序的执行流中调用其它线程的join()方法时，调用的线程将被阻塞，直到join方法加入的join线程完成为止（底层用wait实现）。

```
public class JoinThread extends Thread{

    //提供有参数构造器，用于设置该线程的名字
    public JoinThread(String name){
        super(name);
    }

    @Override
    public void run() {
        for(int i=0;i<10;i++){
            System.out.println(getName()+" "+i);
        }
    }

    public static void main(String[] args) throws InterruptedException {
        //启动子线程
        new JoinThread("新线程").start();
        for(int i=0;i<10;i++){
            if(i==5){
                JoinThread jt=new JoinThread("被Join的线程");
                jt.start();
                //main线程中调用了jt线程的join方法，main线程必须等待jt线程执行结束
                jt.join();
            }
            System.out.println(Thread.currentThread().getName()+" "+i);
        }
    }
}
```

```
History  Console
<terminated> JoinThread [Java Applic
新线程 0
新线程 1
main 0
main 1
main 2
main 3
main 4
新线程 2
新线程 3
新线程 4
新线程 5
新线程 6
新线程 7
新线程 8
新线程 9
被Join的线程 0
被Join的线程 1
被Join的线程 2
被Join的线程 3
被Join的线程 4
被Join的线程 5
被Join的线程 6
被Join的线程 7
被Join的线程 8
被Join的线程 9
main 5
main 6
main 7
main 8
main 9
```

join线程的三种重载形式：

1) join()：等待被join的线程执行完成。
2) join(long millis)：等待被join的线程最长时间为millis毫秒。如果在millis毫秒内，被join的线程还没有执行结束则等待。
3) join(long millis,int nanos)：等待被join的线程最长时间为millis毫秒+nanos微秒。

2. 线程休眠：sleep

如果我们想让当前正在执行的线程暂停一段时间，并进入阻塞状态（sleep不释放锁）。

1) 可以调用Thread类的静态sleep方法，sleep方法有两种重载的形式：
   - static void sleep(long millis)
   - static void sleep(long millis,int nanos)
2) 或者使用TimeUnit.SECONDS.sleep(long timeout);

如果休眠中的线程被中断（interrupt()）会立即抛出InterruptedException。

3. 线程让步：yield

yield()方法是Thread类的一个静态方法，它可以让当前正在执行的线程暂停（放弃当前分得的CPU时间），但线程不会进入阻塞状态，它只是将该进程转入就绪状态，所以使用yield()线程有可能又会立即变成运行状态。

实际上，当某个线程调用了yield()方法暂停后，只有优先级与当前优先级相同或者优先级高的就绪状态线程才有机会获得执行。

yield()方法通常只做调试时使用。

4. wait()/notify()/notifyAll()使用方法(http://www.cnphp6.com/archives/62258)

wait()、notify()、notifyAll()是三个定义在Object类里的方法，可以用来控制线程的状态。wait()会使线程进入阻塞状态，并且释放锁。

这三个方法最终调用的都是jvm级的native方法。随着jvm运行平台的不同可能有些许差异。

1) 如果对象调用了wait方法就会使持有该对象的线程把该对象的控制权交出去，然后处于等待状态。
2) 如果对象调用了notify方法就会通知某个正在等待这个对象的控制权的线程（如果有多个线程，随机通知）可以继续运行。
3) 如果对象调用了notifyAll方法就会通知所有等待这个对象控制权的线程继续运行。

其中wait方法有三个over load方法：

1) wait()
2) wait(long)
3) wait(long,int)

wait方法通过参数可以指定等待的时长。如果没有指定参数，默认一直等待直到被通知。

waite()和notify()因为会对对象的“锁标志”进行操作，所以它们必须在synchronized方法或synchronized代码块中进行调用。否则运行时会产生IllegalMonitorStateException的异常（但是可以通过编译，要注意）。

5. interrupt()中断

interrupt()意味着在该线程完成任务之前停止其正在进行的一切，有效地中止其当前的操作。线程是死亡、还是等待新的任务或是继续运行至下一步，就取决于这个程序。

Thread.interrupt()方法不会中断一个正在运行的线程。

这一方法实际上完成的是，在线程受到阻塞时抛出一个中断信号，这样线程就得以退出阻塞的状态。更确切的说，如果线程被Object.wait, Thread.join和Thread.sleep三种方法之一阻塞，那么它将接收到一个中断异常（InterruptedException），从而提早地终结被阻塞状态。

interrupt()方法通过修改了被调用线程的中断状态来告知该线程被中断。
1)非阻塞中断：改变了中断状态，并且this.isInterrupted()返回true。
2)阻塞中断：抛出InterruptedException，同时this.isInterrupted()状态设置为false(sleep,join,wait)。

6. 后台进程（守护进程）

有一种线程，它是在后台运行的，他的任务是为其他的线程提供服务，这种线程被称为“后台线程”（如JVM的垃圾回收线程）。如果所有的前台线程都死亡，则后台线程自动死亡。

调用Thread对象setDaemon(true)方法将线程指定为后台线程。当整个虚拟机中只剩下后台线程时，程序就没有执行下去的必要，于是虚拟机也就退出了。并且setDaemon()必须在start()方法被调用前设置，一旦线程开始执行，将不能再修改守护状态。

Thread还提供了isDaemon()方法，用于判断指定线程是否为守护线程。

7. 改变线程的优先级

每个线程执行时都具有一定的优先级，优先级高的线程获得较多的执行机会，而优先级较低的线程则获得较少的执行机会。（设置优先级只是概率，不是绝对的优先级）。
Thread提供setPriority(int newPriority)（参数范围是：1~10，或者是三个静态常量：MAX_PRIORITY 10，MIN_PRIORITY 1，NORM_PRIORITY 5）和getPriority()方法来设置和返回指定线程的优先级。

三、线程同步

同步——多个线程并发执行时，多个线程访问同一份资源，确保资源安全。

3.1 synchronized关键字

每一个用synchronized关键字声明的方法都是临界区，在Java中同一个对象的临界区在同一时间只允许一个线程被访问。但是静态方法则不同，被

synchronized关键字修饰的静态方法，同时只允许被一个线程访问，但是其他线程可以访问这个对象的非静态方法。（此时如果两个方法都改变了相同的数据，将会出现数据不一致的错误）

Synchronized关键字会降低程序的性能，因此只能在并发情景中需要修改共享数据的方法上使用。

1. synchronized同步代码块

```
synchronized(obj)
{
        …… //同步代码块
}
```

obj就是同步监视器，上面的代码含义就是：线程开始执行同步代码块之前，必须获得对同步监视器的锁定。并且JVM保证同一时间只有一个线程能够访问这个对象的同步代码块。

2. synchronized同步方法

```
public synchronized  void method(){ }
```

对于同步方法而言，同步方法的同步监视器是this。

3.2 同步锁（Lock）

通过显示的定义同步锁对象来实现同步，在这种机制下，同步所应该使用Lock对象来充当。Lock提供了比synchronized方法和synchronized代码块更广泛的锁定操作，Lock实现允许更灵活的结构。

1. 基于Lock接口和其实现类（如ReentrantLock等），相对于synchronized关键字提供了更多的优势：
   1) 支持更灵活的同步代码块结构，Lock接口允许实现更复杂的临界区结构，即控制的和释放不出现在同一个块结构中。
   2) Lock接口提供了更多的功能，如tryLock()，这个方法试图获取锁，如果锁被其他线程控制，它将返回false，并继续往下执行代码。通过tryLock()方法的返回值可以得知是否有其他线程正在使用这个锁保护的代码块。
   3) Lock接口允许读写分离操作，允许多个读线程和只有一个写线程。
   4) 相比synchronized关键字，lock接口具有更好的性能。

```
import java.util.concurrent.locks.ReentrantLock;

public class test {
        //定义锁对象
        private final ReentrantLock lock=new ReentrantLock();
        ……
        //定义需要保证线程安全的方法
        public void m(){
                //加锁
                lock.lock();
                try
                {
                        //需要保证线程安全的代码
                        // …… method body
                }
                //使用finally块来保证释放锁
                finally{
                        lock.unlock();
                }
        }
}
```

示例：

```
public class PrintQueue {
        private final Lock queueLock=new ReentrantLock();

        public void printJob(Object document){
                queueLock.lock();//通过调用lock()方法来获得锁对象
                try {
                        Long duration=(long)(Math.random()*10000);
                        System.out.printf("%s: PrintQueue: Printing a Job during %d seconds\n",Thread.currentThread().getName(),(duration/1000));
                        Thread.sleep(duration);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
                finally {
                        queueLock.unlock();//通过unlock()来释放锁，必须有否则会产生死锁
                }
        }
}
```

```
public class Job implements Runnable {
        private PrintQueue printQueue;

        public Job(PrintQueue printQueue){
                this.printQueue=printQueue;
        }
```

```
        @Override
        public void run() {
                System.out.printf("%s: Going to print a document\n",Thread.currentThread().getName());
                printQueue.printJob(new Object());
                System.out.printf("%s: The document has been printed\n",Thread.currentThread().getName());
        }
}
```

```
public class Main {
        public static void main (String args[]){
        PrintQueue printQueue=new PrintQueue();

        Thread thread[]=new Thread[2000];
        for (int i=0; i<2000; i++){
                thread[i]=new Thread(new Job(printQueue),"Thread "+i);
        }
        for (int i=0; i<2000; i++){
                thread[i].start();
        }
        }
}
```

运行结果：

```
Thread 2: Going to print a document
Thread 464: Going to print a document
……
Thread 487: Going to print a document
Thread 2: PrintQueue: Printing a Job during 9 seconds
Thread 1847: Going to print a document
……
Thread 1868: Going to print a document
Thread 2: The document has been printed//必须等Thread2打印完毕释放了锁其他线程才能开始打印，如果没有unlock，那么线程在此处产生死锁
Thread 464: PrintQueue: Printing a Job during 3 seconds
Thread 464: The document has been printed
Thread 465: PrintQueue: Printing a Job during 9 seconds
Thread 465: The document has been printed
Thread 463: PrintQueue: Printing a Job during 8 seconds
Thread 463: The document has been printed
Thread 462: PrintQueue: Printing a Job during 8 seconds
Thread 462: The document has been printed
Thread 461: PrintQueue: Printing a Job during 1 seconds
Thread 461: The document has been printed
Thread 457: PrintQueue: Printing a Job during 5 seconds
Thread 457: The document has been printed
Thread 456: PrintQueue: Printing a Job during 1 seconds
Thread 456: The document has been printed
Thread 480: PrintQueue: Printing a Job during 5 seconds
……
```

在PrintQueue类中的临界区开始时，必须通过lock方法获取对锁的控制，当线程A访问这个方法时，如果没有其他线程获取对这个锁的控制，lock()方法将让线程A 获得锁并且立刻执行邻接区代码。否则如果其他线程B正在执行这个锁保护的临界区代码，lock()方法将让线程A休眠直到线程B执行完临界区的代码（B释放锁后A才能执行临界区代码）。

2. 使用读写锁实现同步数据的访问（ReadWriteLock接口和ReenTrantReadWriteLock类）

   ReenTrantReadWriteLock类有两个锁，一个是读操作锁一个是写操作锁。使用读操作锁允许多个线程同时访问，但写操作锁只允许一个线程运行。
   - ReadWriteLock lock=new ReentrantReadWriteLock();
   - 读操作锁：lock.readLock().lock();
   - 写操作锁：lock.writeLock().lock();

   示例：

```
public class PricesInfo {
        private double price1;
        private double price2;

        private ReadWriteLock lock;

        public PricesInfo(){
                price1=1.0;
                price2=2.0;
                lock=new ReentrantReadWriteLock();
        }

        public double getPrice1() {
                lock.readLock().lock();
                double value=price1;
                lock.readLock().unlock();
                return value;
        }
```

```
        public double getPrice2() {
                lock.readLock().lock();
                double value=price2;
                lock.readLock().unlock();
                return value;
        }

        public void setPrices(double price1, double price2) {
                lock.writeLock().lock();
                this.price1=price1;
                this.price2=price2;
                lock.writeLock().unlock();
        }
}
```

```
public class Reader implements Runnable {
        private PricesInfo pricesInfo;

        public Reader (PricesInfo pricesInfo){
                this.pricesInfo=pricesInfo;
        }

        @Override
        public void run() {
                for (int i=0; i<10; i++){
                        System.out.printf("%s: Price 1: %f\n",Thread.currentThread().getName(),pricesInfo.getPrice1());
                        System.out.printf("%s: Price 2: %f\n",Thread.currentThread().getName(),pricesInfo.getPrice2());
                }
        }
}
```

```
public class Writer implements Runnable {
        private PricesInfo pricesInfo;

        public Writer(PricesInfo pricesInfo){
                this.pricesInfo=pricesInfo;
        }

        @Override
        public void run() {
                for (int i=0; i<3; i++) {
                        System.out.printf("Writer: Attempt to modify the prices.\n");
                        pricesInfo.setPrices(Math.random()*10, Math.random()*8);
                        System.out.printf("Writer: Prices have been modified.\n");
                        try {
                                Thread.sleep(2);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
        }
}
```

运行结果：使用读操作锁允许多个线程同时访问，但写操作锁只允许一个线程运行。

```
Thread-0: Price 1: 1.000000
Thread-4: Price 1: 1.000000
Thread-4: Price 2: 2.000000
Thread-3: Price 1: 1.000000
Thread-3: Price 2: 2.000000
Thread-3: Price 1: 1.000000
Thread-3: Price 2: 2.000000
Thread-3: Price 1: 1.000000
Thread-1: Price 1: 1.000000
Thread-1: Price 2: 2.000000
Thread-2: Price 1: 1.000000
Thread-2: Price 2: 2.000000
Writer: Attempt to modify the prices.
Thread-2: Price 1: 1.000000
Thread-1: Price 1: 1.000000
Thread-1: Price 2: 2.000000
Thread-3: Price 2: 2.000000
Thread-4: Price 1: 1.000000
Thread-4: Price 2: 7.226754 价格被线程4所修改
Thread-4: Price 1: 5.478235
Thread-0: Price 2: 2.000000
Thread-4: Price 2: 7.226754
Thread-3: Price 1: 5.478235
Thread-3: Price 2: 7.226754
```

```
Writer: Prices have been modified.
Thread-1: Price 1: 1.000000
Thread-1: Price 2: 7.226754
Thread-1: Price 1: 5.478235
Thread-1: Price 2: 7.226754
Thread-1: Price 1: 5.478235
Thread-2: Price 2: 2.000000
Thread-2: Price 1: 5.478235
Thread-2: Price 2: 7.226754
Thread-2: Price 1: 5.478235
Thread-2: Price 2: 7.226754
Thread-2: Price 1: 5.478235
Thread-2: Price 2: 7.226754
Thread-1: Price 2: 7.226754
Thread-3: Price 1: 5.478235
Thread-4: Price 1: 5.478235
Thread-4: Price 2: 7.226754
Thread-4: Price 1: 5.478235
Thread-0: Price 1: 5.478235
Thread-4: Price 2: 7.226754
Thread-3: Price 2: 7.226754
Writer: Attempt to modify the prices.
Thread-0: Price 2: 7.226754
Writer: Prices have been modified.
Thread-0: Price 1: 3.256387 价格被线程0所修改
Thread-0: Price 2: 6.667209
Thread-0: Price 1: 3.256387
Thread-0: Price 2: 6.667209
Thread-0: Price 1: 3.256387
Thread-0: Price 2: 6.667209
Writer: Attempt to modify the prices.
Writer: Prices have been modified.
```

3. ReentrantLock和ReenTrantReadWriteLock类的构造器都含有一个布尔参数fair，它允许控制这两个类的行为。
   1) fair=true，公平模式
      当有很多线程在等待锁时，锁将选择他们中的一个来访问临界区，而且选择等待时间最长的。
   2) fair=fale，非公平模式
      非公平模式下，这个选择没有任何约束。

```
public class Job implements Runnable {
	private PrintQueue printQueue;

	public Job(PrintQueue printQueue){
		this.printQueue=printQueue;
	}

	@Override
	public void run() {
		System.out.printf("%s: Going to print a job\n",Thread.currentThread().getName());
		printQueue.printJob(new Object());
		System.out.printf("%s: The document has been printed\n",Thread.currentThread().getName());
	}
}
```

```
public class PrintQueue {

	private final Lock queueLock=new ReentrantLock(false);

	public void printJob(Object document){
		queueLock.lock();

		try {
			Long duration=(long)(Math.random()*10000);
			System.out.printf("%s: PrintQueue: Printing a Job during %d seconds\n",Thread.currentThread().getName(),(duration/1000));
			Thread.sleep(duration);
		} catch (InterruptedException e) {
			e.printStackTrace();
		} finally {
			queueLock.unlock();
		}


		queueLock.lock();
		try {
			Long duration=(long)(Math.random()*10000);
			System.out.printf("%s: PrintQueue: Printing a Job during %d seconds\n",Thread.currentThread().getName(),(duration/1000));
			Thread.sleep(duration);
		} catch (InterruptedException e) {
```

```
                e.printStackTrace();
            } finally {
                queueLock.unlock();
            }
        }
}
```

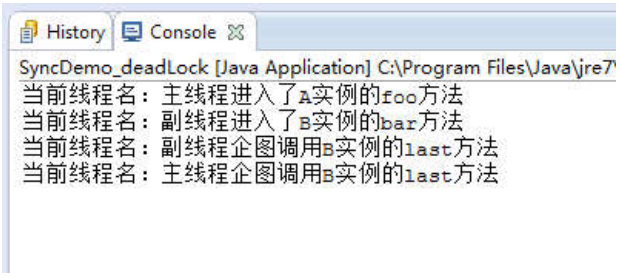
运行结果如下：

公平模式	非公平模式
Thread 0: Going to print a job	Thread 0: Going to print a job
Thread 0: PrintQueue: Printing a Job during 7 seconds	Thread 0: PrintQueue: Printing a Job during 2 seconds
Thread 1: Going to print a job	Thread 1: Going to print a job
Thread 2: Going to print a job	Thread 2: Going to print a job
Thread 3: Going to print a job	Thread 3: Going to print a job
Thread 4: Going to print a job	Thread 4: Going to print a job
Thread 1: PrintQueue: Printing a Job during 9 seconds	Thread 0: PrintQueue: Printing a Job during 0 seconds
Thread 2: PrintQueue: Printing a Job during 2 seconds	Thread 0: The document has been printed
Thread 3: PrintQueue: Printing a Job during 4 seconds	Thread 1: PrintQueue: Printing a Job during 5 seconds
Thread 4: PrintQueue: Printing a Job during 8 seconds	Thread 1: PrintQueue: Printing a Job during 2 seconds
Thread 0: PrintQueue: Printing a Job during 8 seconds	Thread 1: The document has been printed
Thread 0: The document has been printed	Thread 2: PrintQueue: Printing a Job during 1 seconds
Thread 1: PrintQueue: Printing a Job during 3 seconds	Thread 2: PrintQueue: Printing a Job during 1 seconds
Thread 1: The document has been printed	Thread 2: The document has been printed
Thread 2: PrintQueue: Printing a Job during 7 seconds	Thread 3: PrintQueue: Printing a Job during 6 seconds
Thread 2: The document has been printed	Thread 3: PrintQueue: Printing a Job during 8 seconds
Thread 3: PrintQueue: Printing a Job during 0 seconds	Thread 3: The document has been printed
Thread 3: The document has been printed	Thread 4: PrintQueue: Printing a Job during 9 seconds
Thread 4: PrintQueue: Printing a Job during 2 seconds	Thread 4: PrintQueue: Printing a Job during 2 seconds
Thread 4: The document has been printed	Thread 4: The document has been printed
每次释放锁时，都会选择等待队列中的等待时间最长的来执行，所以执行完代码段1之后，不会立即执行代码段2	

3.3 死锁

当两个线程互相等待对方释放同步监视器就会发生死锁。一旦出现死锁，整个程序既不会发生异常，也不会给出任何提示，只是所有的线程处于阻塞状态，无法继续。

```
package test.oop.thread.sync;
class A{
    public synchronized void foo(B b){//调用该方法会对A对象加锁
        System.out.println("当前线程
        名："+Thread.currentThread().getName()+"进入了A实例的foo
        方法");//1
        try {
            Thread.sleep(200);
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("当前线程
        名："+Thread.currentThread().getName()+"企图调用B实例的
        last方法");//3
        b.last();
    }

    public synchronized void last(){//调用该方法会对A对象加锁
        System.out.println("进入了A类的last方法内部");
    }
}

class B{
    public synchronized void bar(A a){//调用该方法会对B对象加锁
        System.out.println("当前线程
        名："+Thread.currentThread().getName()+"进入了B实例的bar
        方法");//2
        try {
            Thread.sleep(200);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("当前线程
        名："+Thread.currentThread().getName()+"企图调用A实例的
        last方法");//4
        a.last();
    }

    public synchronized void last(){//调用该方法会对B对象加锁
        System.out.println("进入了B类的last方法");
    }
```



该程序中A对象和B对象的方法都是同步方法，也就是说A对象和B对象都是同步锁。

主线程负责执行init()方法，调用了A对象的foo()同步方法，并且对A对象加锁（接着主线程暂停200ms），CPU执行副线程。……主线程醒来，于是主线程执行B对象的last()方法，执行last()方法之前需要对B对象加锁，但此时副线程保持着B对象的锁，主线程阻塞。

副线程负责执行run()方法，调用了B对象的bar()同步方法，并且对B对象加锁（接着副线程暂停200ms），CPU执行主线程。……副线程醒来，于是副线程执行A对象的last()方法，执行last()方法之前需要对A对象加锁，但此时主线程保持着A对象的锁，副线程阻塞。

上例中，主线程等待着副线程释放B对象的锁，处于阻塞状态，而副线程等待着主线程释放A对象的锁，也处于阻塞状态。因此产生了死锁。

```
}

public class SyncDemo_deadLock implements Runnable{
	A a=new A();
	B b=new B();

	public void init(){//主线程的方法
		Thread.currentThread().setName("主线程");
		a.foo(b);
		System.out.println("进入主线程之后");
	}

	public void run(){//副线程的方法体
		Thread.currentThread().setName("副线程");
		b.bar(a);
		System.out.println("进入副线程之后");
	}
	public static void main(String[] args) {
		SyncDemo_deadLock dl=new SyncDemo_deadLock();
		new Thread(dl).start();
		dl.init();
	}
}
```

3.4 生产者消费者示例

四、并发集合

Java提供了两类适用于并发应用的集合。

- 阻塞式集合：这集合包括添加和移除数据的方法。当集合已满或为空时，被调用的添加或者移除方法节不能被立即执行，那么调用这个方法的线程将被阻塞，一直到该方法可以被成功执行。
- 非阻塞式集合：这类集合也包括添加和移除数据的方法。如果方法不能立即执行，则返回null或抛出异常，但是调用这个方法的线程不会被阻塞。

4.1 非阻塞式线程安全列表 ConcurrentLinkedDeque

pollFirst()、pollLast()	分别返回列表中的第一个、最后一个元素并移除（返回并移除），列表为空是返回null
removeFirst()、removeLast()	分别返回列表中的第一个、最后一个元素并移除（返回并移除），列表为空时抛出NoSuchElementException
peekFirst()、peekLast()	分别返回列表中的第一个、最后一个元素（返回不移除），列表为空是返回null
getFirst()、getLast()	分别返回列表中的第一个、最后一个元素（返回不移除），列表为空时抛出NoSuchElementException
size()	返回列表中元素的数量，当有线程添加/删除列表中元素时，返回的数据可能不准确

示例：

```
public class AddTask implements Runnable {
	private ConcurrentLinkedDeque<String> list;

	public AddTask(ConcurrentLinkedDeque<String> list) {
		this.list=list;
	}

	@Override
	public void run() {
		String name=Thread.currentThread().getName();
		for (int i=0; i<10000; i++){
			list.add(name+": Element "+i);
		}
	}

}
```

```
public class PollTask implements Runnable {
	private ConcurrentLinkedDeque<String> list;

	public PollTask(ConcurrentLinkedDeque<String> list) {
		this.list=list;
	}

	@Override
	public void run() {
		for (int i=0; i<5000; i++) {
			list.pollFirst();
			list.pollLast();
		}
	}
}
```

```
public class Main {

        public static void main(String[] args) throws Exception {
                ConcurrentLinkedDeque<String> list=new
                ConcurrentLinkedDeque<>();
                Thread threads[]=new Thread[100];

        for (int i=0; i<100; i++){
                AddTask task=new AddTask(list);
                threads[i]=new Thread(task);
                threads[i].start();
        }
        System.out.printf("Main: %d AddTask threads have been
        launched\n",threads.length);

        for (int i=0; i<100; i++) {
                threads[i].join();
        }

        System.out.printf("Main: Size of the List: %d\n",list.size());

        for (int i=0; i<threads.length; i++){
                PollTask task=new PollTask(list);
                threads[i]=new Thread(task);
                threads[i].start();
        }
        System.out.printf("Main: %d PollTask threads have been launched
        \n",threads.length);

        for (int i=0; i<threads.length; i++) {
                threads[i].join();
        }

        System.out.printf("Main: Size of the List: %d\n",list.size());
        }
}
```

```
public class Main {

        public static void main(String[] args) throws Exception {
                ConcurrentLinkedDeque<String> list=new
                ConcurrentLinkedDeque<>();
                Thread threads[]=new Thread[100];

        for (int i=0; i<100; i++){
                AddTask task=new AddTask(list);
                threads[i]=new Thread(task);
                threads[i].start();
        }
        System.out.printf("Main: %d AddTask threads have been
        launched\n",threads.length);

        System.out.printf("Main: Size of the List: %d\n",list.size());

        for (int i=0; i<threads.length; i++){
                PollTask task=new PollTask(list);
                threads[i]=new Thread(task);
                threads[i].start();
        }
        System.out.printf("Main: %d PollTask threads have been launched
        \n",threads.length);

        System.out.printf("Main: Size of the List: %d\n",list.size());
        }
}
```

```
运行结果：
Main: 100 AddTask threads have been launched
Main: Size of the List: 1000000
Main: 100 PollTask threads have been launched
Main: Size of the List: 0
```

```
运行结果：
Main: 100 AddTask threads have been launched
Main: Size of the List: 969815
Main: 100 PollTask threads have been launched
Main: Size of the List: 760182
```

4.2 阻塞式线程安全列表LinkedBlockingDeque

put()、putFirst()、putLast()	添加数据，列表满会阻塞
add()、addFirst()、addLast()	添加数据，列表满会抛出IllegalStateException
take()、takeFirst()、takeLast()	返回数据并移除，列表空会阻塞
poll()、pollFirst()、pollFast()	返回数据并移除，列表空返回null
get()、getFirst()、getLast()	返回数据不移除，列表空抛出NoSuchElementException
peek()、peekFirst()、peekLast()	返回数据不移除，列表空返回null

示例：

```
public class Client implements Runnable{//向列表中加数据

        private LinkedBlockingDeque<String> requestList;

        public Client (LinkedBlockingDeque<String> requestList) {
                this.requestList=requestList;
        }

        @Override
        public void run() {
                for (int i=0; i<3; i++) {
                        for (int j=0; j<5; j++) {
                                StringBuilder request=new StringBuilder();
                                request.append(i);
                                request.append(":");
                                request.append(j);
                                try {
                                        requestList.put(request.toString());
                                } catch (InterruptedException e) {
                                        e.printStackTrace();
                                }
```

```
public class Main {//执行取数据

        public static void main(String[] args) throws Exception {

        LinkedBlockingDeque<String> list=new LinkedBlockingDeque<>
        (3);

        Client client=new Client(list);
        Thread thread=new Thread(client);
        thread.start();//执行加数据

        for (int i=0; i<5 ; i++) {
                for (int j=0; j<3; j++) {
                        String request=list.take();
                        System.out.printf("Main: Request: %s at %s.
                        \n",request,new Date());
                }
                TimeUnit.MILLISECONDS.sleep(300);
        }

        System.out.printf("Main: End of the program.\n");
```

```
                    System.out.printf("Client: %s at %s.
                    \n",request,new Date());
                }
                try {
                    TimeUnit.SECONDS.sleep(2);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.printf("Client: End.\n");
        }
    }
```

```
    }
}
```

运行结果：

```
Main: Request: 0:0 at Sun Nov 20 16:55:26 CST 2016.//列表为空，被阻塞
Client: 0:0 at Sun Nov 20 16:55:26 CST 2016.
Client: 0:1 at Sun Nov 20 16:55:26 CST 2016.
Main: Request: 0:1 at Sun Nov 20 16:55:26 CST 2016.
Client: 0:2 at Sun Nov 20 16:55:26 CST 2016.
Main: Request: 0:2 at Sun Nov 20 16:55:27 CST 2016.
Client: 0:3 at Sun Nov 20 16:55:27 CST 2016.
Client: 0:4 at Sun Nov 20 16:55:27 CST 2016.
Main: Request: 0:3 at Sun Nov 20 16:55:27 CST 2016.
Main: Request: 0:4 at Sun Nov 20 16:55:27 CST 2016.
Client: 1:0 at Sun Nov 20 16:55:29 CST 2016.
Client: 1:1 at Sun Nov 20 16:55:29 CST 2016.
Client: 1:2 at Sun Nov 20 16:55:29 CST 2016.
Client: 1:3 at Sun Nov 20 16:55:29 CST 2016.
Main: Request: 1:0 at Sun Nov 20 16:55:29 CST 2016.
Main: Request: 1:1 at Sun Nov 20 16:55:29 CST 2016.
Client: 1:4 at Sun Nov 20 16:55:29 CST 2016.
Main: Request: 1:2 at Sun Nov 20 16:55:29 CST 2016.
Main: Request: 1:3 at Sun Nov 20 16:55:29 CST 2016.
Main: Request: 1:4 at Sun Nov 20 16:55:29 CST 2016.
Client: 2:0 at Sun Nov 20 16:55:31 CST 2016.
Main: Request: 2:0 at Sun Nov 20 16:55:31 CST 2016.
Main: Request: 2:1 at Sun Nov 20 16:55:31 CST 2016.
Client: 2:1 at Sun Nov 20 16:55:31 CST 2016.
Client: 2:2 at Sun Nov 20 16:55:31 CST 2016.
Client: 2:3 at Sun Nov 20 16:55:31 CST 2016.
Client: 2:4 at Sun Nov 20 16:55:31 CST 2016.
Main: Request: 2:2 at Sun Nov 20 16:55:31 CST 2016.
Main: Request: 2:3 at Sun Nov 20 16:55:31 CST 2016.
Main: Request: 2:4 at Sun Nov 20 16:55:31 CST 2016.
Main: End of the program.
Client: End.
```

4.3 按优先级排序的阻塞式线程安全列表PriorityBlockingQueue

PriorityBlockingQueue是Java提供的一个阻塞式排序队列，其中存放的元素必须都要实现Comparable接口，在入队的时候会比较当前元素和队首元素的值大小（重写compareTo()方法，自定义要比较的元素）。

poll()	返回队首元素并移除
clear()	移除队列中的所有元素
take()	返回队首元素并移除，队列空阻塞
put(E e)	插入队列
peek()	返回队首元素不删除

4.4 带有延迟元素的线程安全列表DelayQueue

4.5 线程安全可遍历映射

public class ConcurrentSkipListMap<K,V> extends AbstractMap<K,V> implements ConcurrentNavigableMap<K,V>, Cloneable, Serializable

1) ConcurrentNavigableMap接口
2) ConcurrentSkipListMap类
   ConcurrentSkipListMap类基于SkipList来实现，SkipList是基于并发列表的数据结构，效率与二叉树相近，比有序列表在添加、收索或删除元素时消耗更少的访问时间，该接口类使用键值来排序所有的元素。

主要方法：

firstEntry()	返回map中第一个Map.Entry对象，如果map为空返回null
pollFirstEntry()	返回并移除map中第一个Map.Entry对象
subMap(K fromKey, K toKey)	返回从fromKey到toKey的ConcurrentSkipListMap子集合
headMap(K toKey)	返回所有小于toKey的ConcurrentSkipListMap子集合
tailMap(K fromKey)	返回所有大于fromKey的ConcurrentSkipListMap子集合

putIfAbsent(K key, V value)	如果ConcurrentSkipListMap中不存在key则加入
pollLastEntry()	返回并移除最后一个Map.Entry对象
replace(K key, V value)	如果ConcurrentSkipListMap中存在key，则替换相应的value

示例：

```
public class Contact {
	private String name;

	private String phone;

	public Contact(String name, String phone) {
		this.name=name;
		this.phone=phone;
	}

	public String getName() {
		return name;
	}

	public String getPhone() {
		return phone;
	}
}
```

```
public class Task implements Runnable {
	private ConcurrentSkipListMap<String, Contact> map;

	private String id;

	public Task (ConcurrentSkipListMap<String, Contact> map, String id) {
		this.id=id;
		this.map=map;
	}

	@Override
	public void run() {
		for (int i=0; i<1000; i++) {
			Contact contact=new Contact(id, String.valueOf(i+1000));
			map.put(id+contact.getPhone(), contact);
		}
	}
}
```

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

	ConcurrentSkipListMap<String, Contact> map=new ConcurrentSkipListMap<>();

	Thread threads[]=new Thread[25];

	for (char i='A'; i<'Z'; i++) {
		Task task=new Task(map, String.valueOf(i));
		threads[counter]=new Thread(task);
		threads[counter].start();
		counter++;
	}


	for (int i=0; i<threads.length; i++) {
		try {
			threads[i].join();
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}

	System.out.printf("Main: Size of the map: %d\n",map.size());

	Map.Entry<String, Contact> element;
	Contact contact;

	element=map.firstEntry();
	contact=element.getValue();
	System.out.printf("Main: First Entry: %s: %s\n",contact.getName(),contact.getPhone());

	element=map.lastEntry();
	contact=element.getValue();
	System.out.printf("Main: Last Entry: %s: %s\n",contact.getName(),contact.getPhone());
```

```
        System.out.printf("Main: Submap from A1996 to B1002: \n");
        ConcurrentNavigableMap<String, Contact> submap=map.subMap("A1996", "B1002");
        do {
                element=submap.pollFirstEntry();
                if (element!=null) {
                        contact=element.getValue();
                        System.out.printf("%s: %s\n",contact.getName(),contact.getPhone());
                }
        } while (element!=null);
        }
}
```

运行结果：

```
Main: Size of the map: 25000
Main: First Entry: A: 1000
Main: Last Entry: Y: 1999
Main: Submap from A1996 to B1002:
A: 1996
A: 1997
A: 1998
A: 1999
B: 1000
B: 1001
```

4.6 生成并发随机数

ThreadLocalRandom类用于在并发程序中生成伪随机数，它是线程本地变量，每一个生产随机数的线程都有一个不同的生成器，但是都在同一个类中被管理。相对于共享的Random对象为所有的线程生成随机数，该类拥有更好的性能。

示例：

```
public class TaskLocalRandom implements Runnable {

        public TaskLocalRandom() {
                ThreadLocalRandom.current();
        }

        @Override
        public void run() {
                String name=Thread.currentThread().getName();
                for (int i=0; i<5; i++){
                        System.out.printf("%s: %d\n",name,ThreadLocalRandom.current().nextInt(10));
                }
        }
}
```

```
public class Main {

        public static void main(String[] args) {

                Thread threads[]=new Thread[3];
                for (int i=0; i<threads.length; i++) {
                        TaskLocalRandom task=new TaskLocalRandom();
                        threads[i]=new Thread(task);
                        threads[i].start();
                }
        }
}
```

运行结果：

```
Thread-0: 6
Thread-1: 8
Thread-1: 7
Thread-1: 1
Thread-1: 5
Thread-1: 5
Thread-2: 8
Thread-2: 8
Thread-2: 8
Thread-2: 0
Thread-2: 7
Thread-0: 8
Thread-0: 2
Thread-0: 5
Thread-0: 5
```

其中ThreadLocalRandom.current();中的静态方法current实现如下：

```java
/** The common ThreadLocalRandom */
static final ThreadLocalRandom instance = new ThreadLocalRandom();
```

```java
/**
 * Returns the current thread's {@code ThreadLocalRandom}.
 *
 * @return the current thread's {@code ThreadLocalRandom}
 */
public static ThreadLocalRandom current() {
    if (UNSAFE.getInt(Thread.currentThread(), PROBE) == 0)
        localInit();
    return instance;
}
```

current方法是一个静态方法，返回与当前线程关联的ThreadLocalRandom对象，所以可以使用这个对象生成随机数。ThreadLocalRandom类也提供了方法来生成long、float、double以及boolean值，以及传入一个参数来生成0与该数字之间的随机数，还可以传入两个参数来生成介于两参数见的随机数。

4.7 原子操作

当多线程共享一个或多个对象时，为了避免数据不一致错误，需要实现同步机制（如锁或synchronized关键字）来保护共享属性的访问。但是这些共享机制存在如下问题：

- 死锁：一个线程被阻塞，并且试图获得的锁正在被其他线程所占有，但其他线程永远都不会释放该锁，这种情况会出现死锁。
- 即使只有一个线程访问共享对象，他任然需要执行必须的代码来获取和释放锁。

针对上上面的情况，为了提高更好的性能，Java引入比较和交换操作。这个操作使用以下三个步骤修改变量的值。

a. 取得变量的值，即变量的旧值。
b. 在一个临时变量中修改变量的值，即变量的新值。
c. 如果上面获得的变量旧值与当前变量的值相等，就用新值替换旧值。如果已有其他线程修改了这个变量的值，上面获得的变量的旧值就可能与当前变量的值不同，则重新执行前两步。

Java中通过compareAndSet()方法实现了上面的机制。

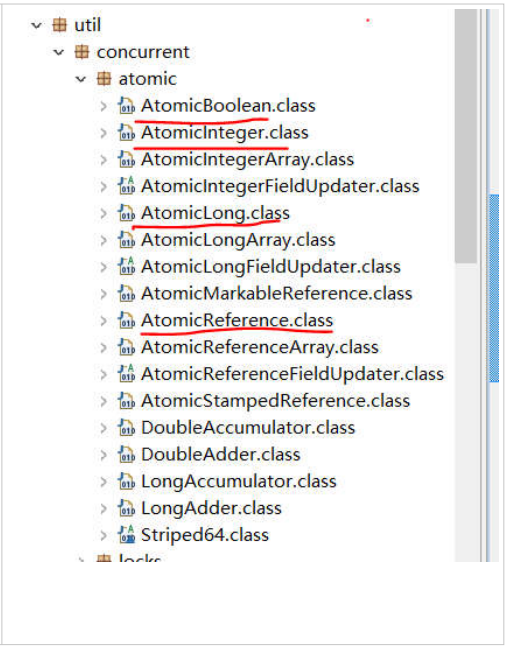
1. 原子变量

   AtomicBoolean、AtomicInteger、AtomicLong、AtomicReference<V>

   原子变量提供了单个变量上的原子操作。当一个线程在对原子变量操作时，如果其他线程也试图对同一个原子变量执行操作，原子变量的实现类提供了一套机制来检查操作是否在一步内完成。一般这个操作先获取变量的值，然后在本地改变变量的值，然后试图用这个改变的值去替换之前的值。如果之前的值没有被其他线程改变，就可以执行这个替换操作。否则方法将再次执行该操作。这种操作被称为CAS原子操作。

   原子变量不使用锁或者其他同步机制来保护对其值得并发访问。所有的操作都是原子的，它保证了多线程在同一时间操作一个原子变量而不会产生数据不一致的错误并且它的性能要优于使用同步机制保护的普通变量。

```java
public class Account {
        private AtomicLong balance;

        public Account(){
                balance=new AtomicLong();
        }

        public long getBalance() {
                return balance.get();
        }

        public void setBalance(long balance) {
                this.balance.set(balance);
        }

        public void addAmount(long amount) {
                this.balance.getAndAdd(amount);
        }

        public void subtractAmount(long amount) {
                this.balance.getAndAdd(-amount);
        }

}
```



2. 原子数组

   AtomicIntegerArray、AtomicLongArray