Java 并发编程

龙浩

一个计算的问题

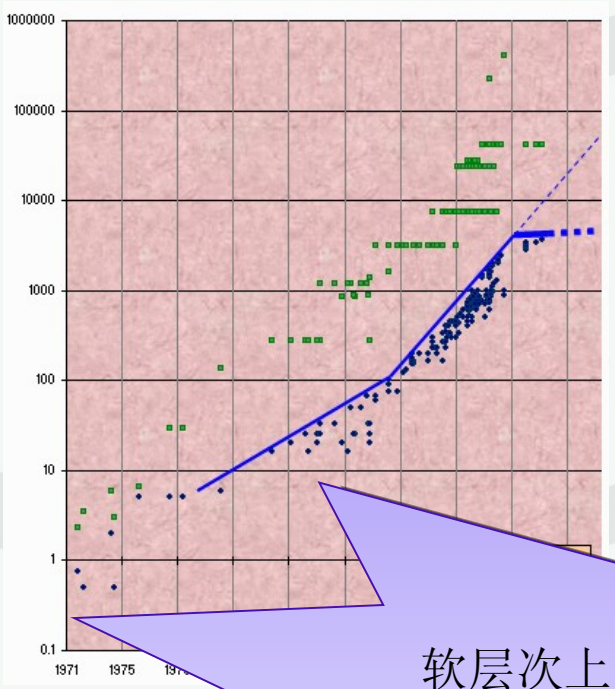
在一个 list 中有过亿条的 Integer 类型的值，如何更快的计算这些值的总和？

简单的方法：更快的 CPU 来遍历

靠谱的方法：分而治之来处理

进一步的方法： Fork/jion

简单的方法靠谱么？

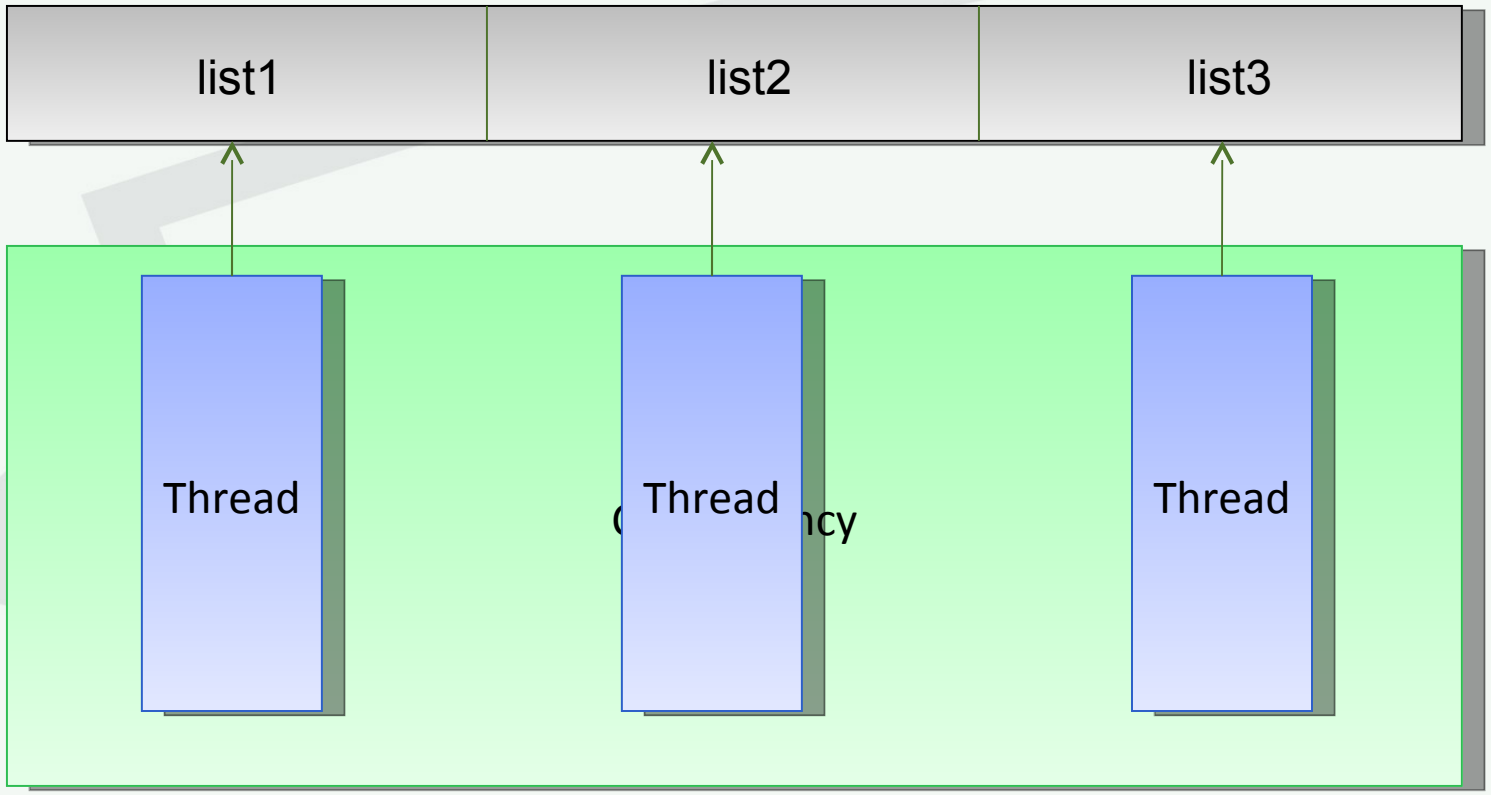


软层次上：遍历是不靠谱的
，for 小学生了！

免费午 靠拢

http:// 880.html

靠谱的方法简单么？（分而治之）



那帮 Java 大神在他们书中说：

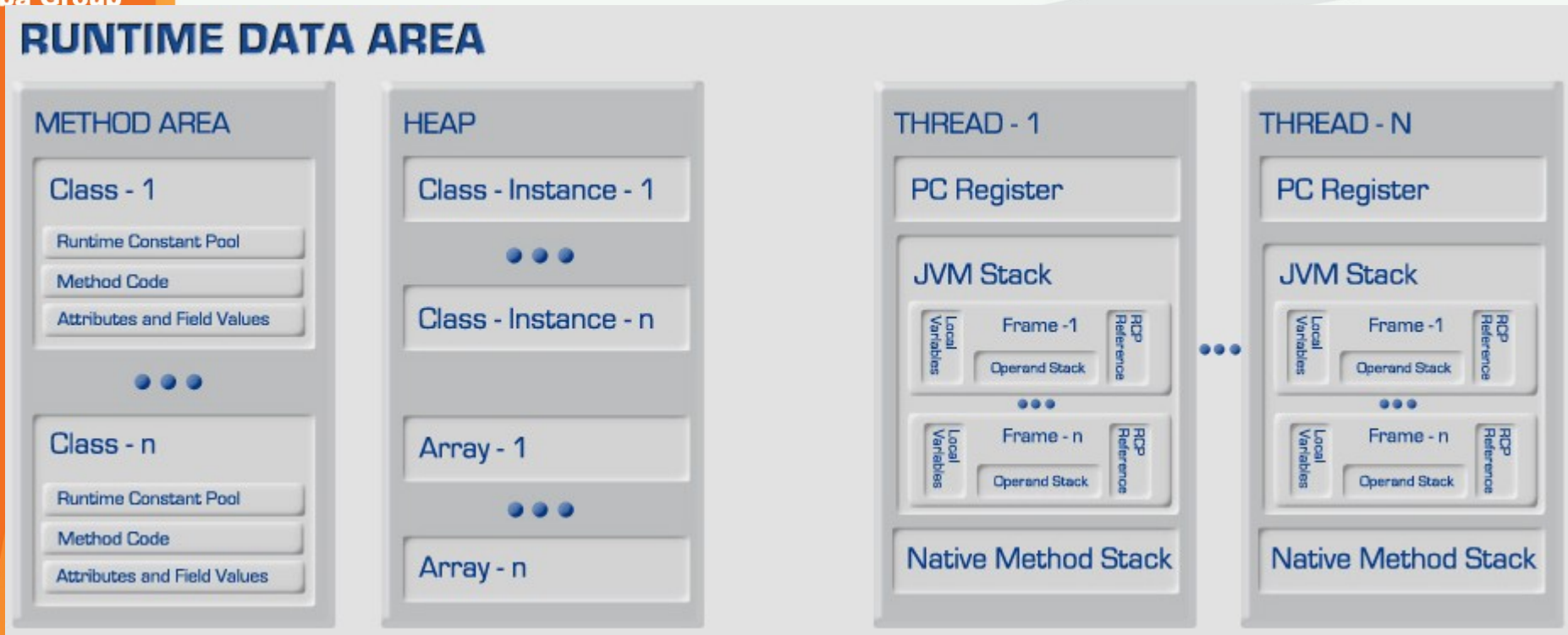
在对性能的追求很可能是并发 bug 唯一最大的来源！

So：

同样不是免费的午餐，需要学习和批量实践。

目录

线程：先让路给内存模型



Visibility：通过并发线程修改变量值，必须将线程变量同步回主存后，其他线程才能访问到。

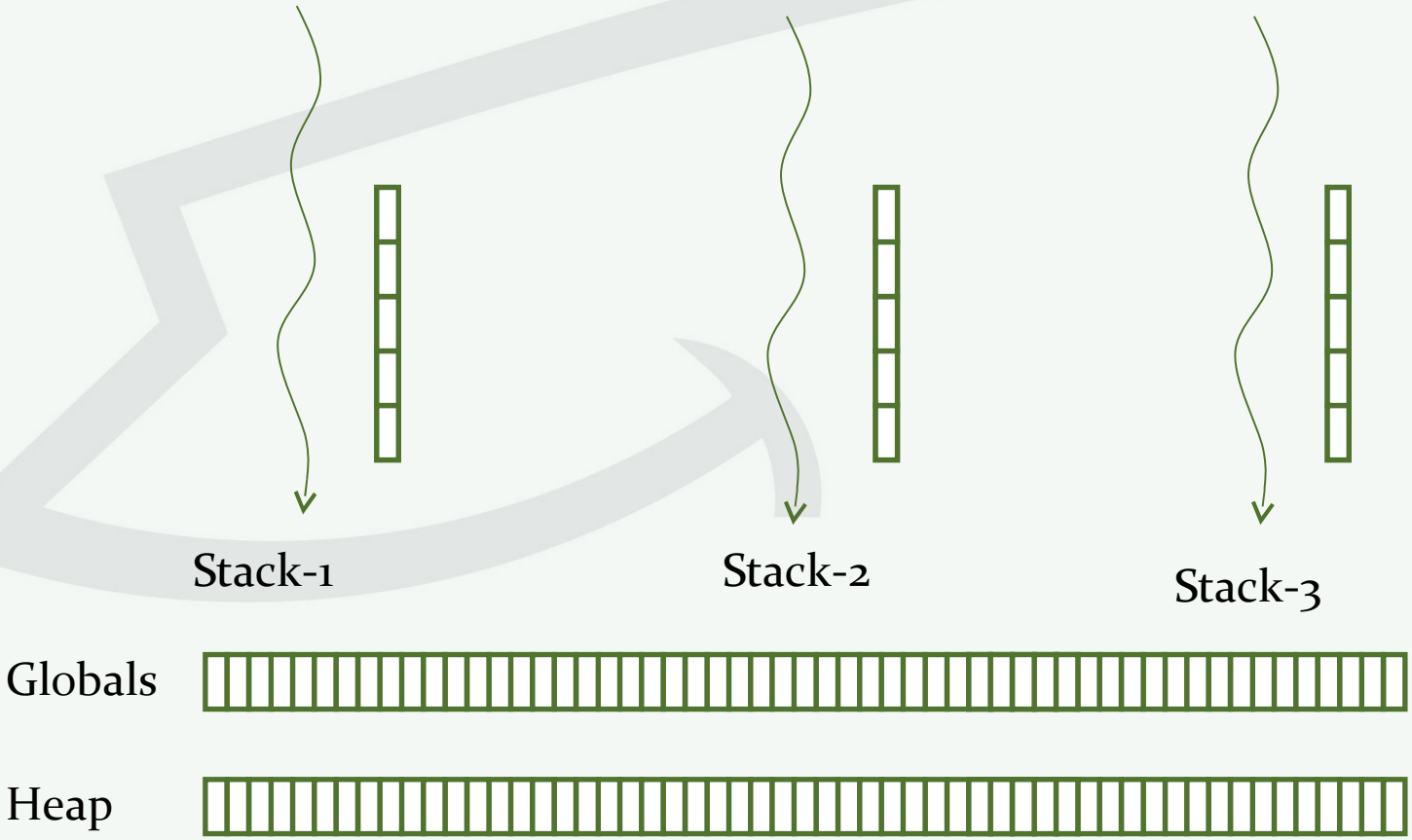
Ordering：通过 java 提供的同步机制或 volatile 关键字，来保证内存的访问顺序。

Cache coherency ：它是一种管理多处理器系统的高速缓存区结构，其可以保证数据在高速缓存区到内存的传输中不会丢失或重复。

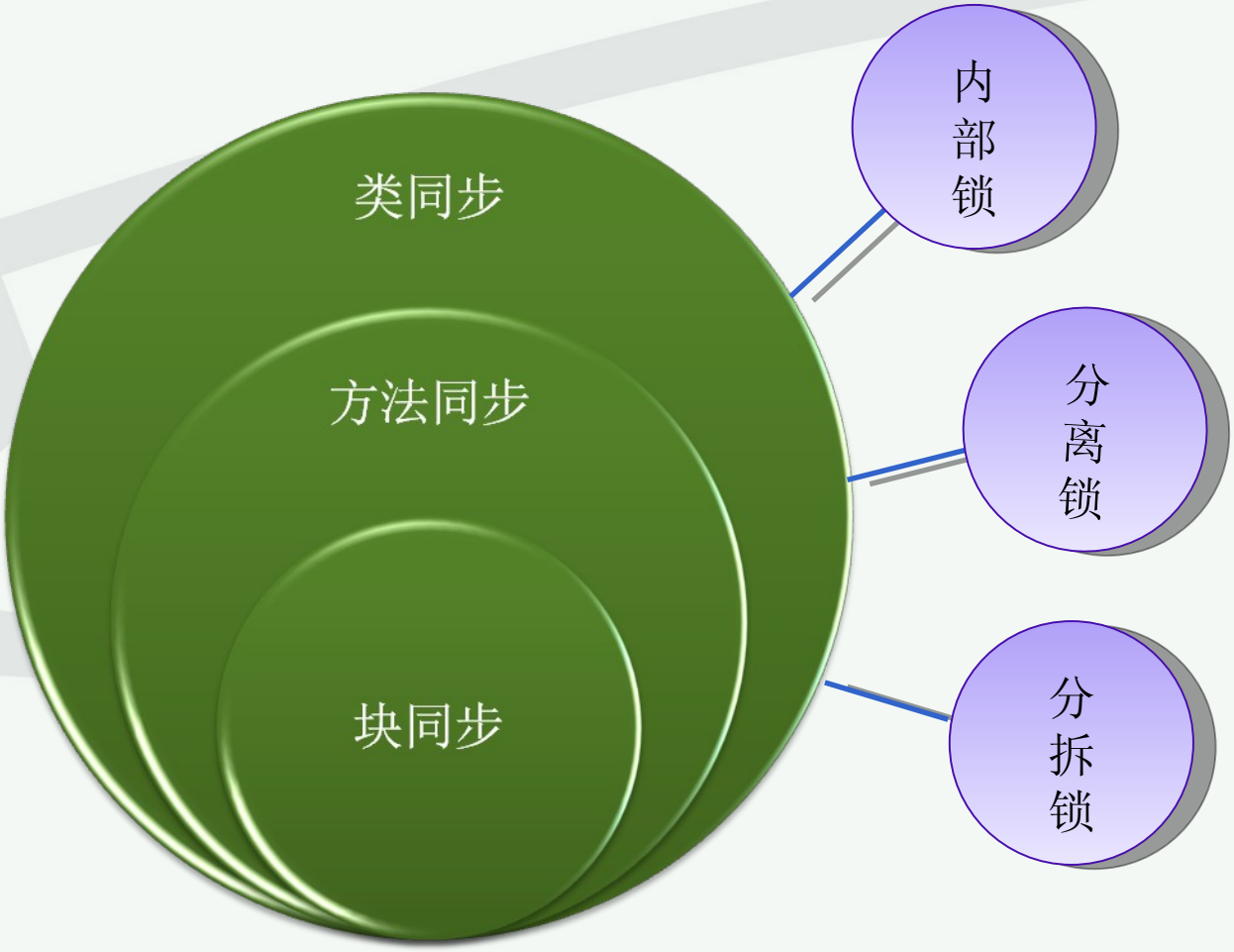
Happens-before ordering： synchronized,volatile,final,java.util.concurrent.lock|atomic

这里有详述： http://is.gd/c8fhE (别迷恋哥，哥只是传说！)

内存中的可见部分



线程： synchronized



保证原子性和可见性

线程： Java Monitors

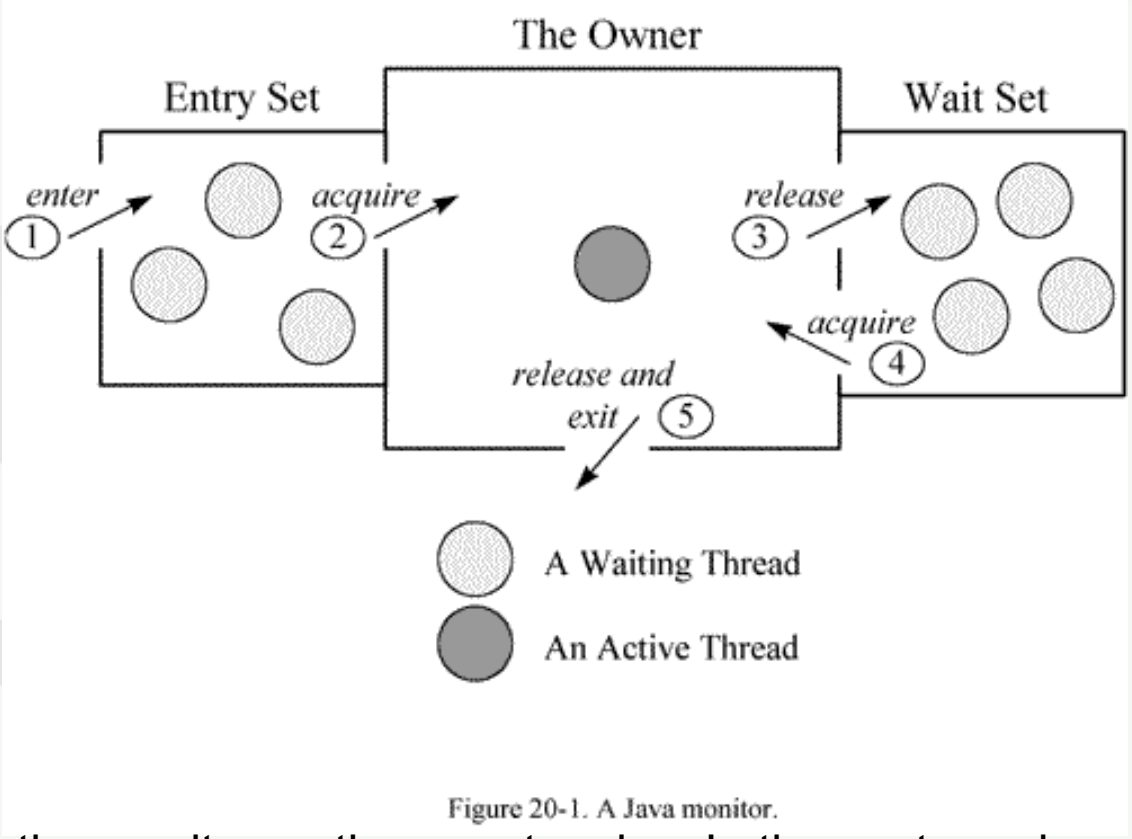


Figure 20-1. A Java monitor.

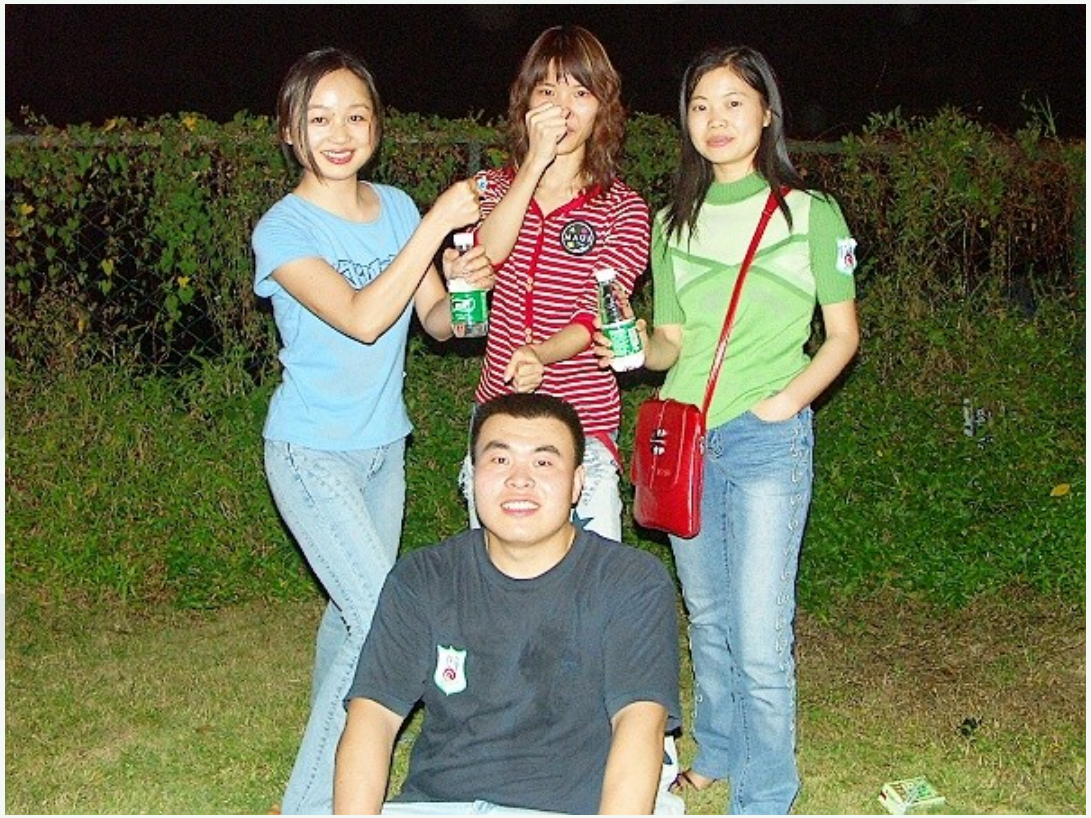
This figure shows the monitor as three rectangles. In the center, a large rectangle contains a single thread, the monitor's owner. On the left, a small rectangle contains the entry set. On the right, another small rectangle contains the wait set. Active threads are shown as dark gray circles. Suspended threads are shown as light gray circles.

线程：独占锁（ synchronized ）

- 非方法修饰符，注意方法覆写的时候需要加上 synchronized ；
- 经典的顺序锁问题（两个线程安全的方法放在一起线程安全么？ ）
- getClass 的问题。
- ……

```
Object a = new Object();
Object b = new Object();
public void order(){
    synchronized(a){
        //do something
        synchronized(b){
            //do something
        }
    }
}
public void order1(){
    synchronized(b){
        //do something
        synchronized(a){
            //do something
        }
    }
}
```

分拆前：思考问题，顺便教你一招！



分拆不了人，工具还不能分拆么？对，买 3 个手机去……

线程：分拆锁

```
public class ServerStatus{
    public final Set<String> users = new HashSet<String>();
    public final Set<String> queries = new HashSet<String>();

    public synchronized void addUser(String u) { users.add(u); }
    public synchronized void addQuery(String q) { queries.add(q); }

    public synchronized void removeUser(String u) {users.remove(u);}
    public synchronized void removeQuery(String q) {queries.remove(q);}
}
```

```
public class SpinOffServerStatus {
    public final Set<String> users = new HashSet<String>();
    public void addUser(String u) {
        synchronized(users){
            users.add(u);
        }
    }

    public void removeUser(String u) {
        synchronized(users){
            users.remove(u);
        }
    }
}
```

线程：分离锁

```
private final Object[] locks;
private static final int N_LOCKS = 4;
private final String [] share ;
private int opNum;
private int N_ANUM;

public StrippingLock(int on, int anum) {
    opNum = on;
    N_ANUM = anum;
    share = new String[N_ANUM];
    locks = new Object[N_LOCKS];
    for (int i = 0; i<N_LOCKS; i++)
    locks[i] = new Object();
}

public synchronized void put1(int indx, String k) {
    share[indx] = k;    //acquire the object lock
}
public void put2(int indx, String k) {
    synchronized (locks[indx%N_LOCKS]) {
        share[indx] = k;    // acquire the corresponding lock
    }
}
```

分离锁负面作用：对容器加锁，进行独占访问更加困难，并且更加昂贵了。

内存使用的问题： sina 就曾经因为在 Action 层使用 ConcurrentHashMap 而导致内存使用过大，修改 array 后竟然单台服务器节省 2G。

线程： static 的案例

```
public class StaticThreadTest {
        // 线程避免调用这个；
        public static Tree tree = new Tree(“jizi”,“2”);
        public static void createTree(Tree trees){
            Tree t = tree;
            if(trees.getName().equals("pg")){t.setName("ceshi");}
        }
        public static void main(String[] args) throws
InterruptedException{
                    ExecutorService exec =
                    Executors.newFixedThreadPool(10);
        for(int i=0;i<10;i++){
        exec.execute(new TreeThread(i));
Thread.sleep(50);
        }
        exec.shutdown();
        exec.awaitTermination(1, TimeUnit.SECONDS);
        }
}
```

线程：可见性

volatile 关键字：

1：简化实现或者同步策略验证的时候来使用它；
2：确保引用对象的可见性；
3：标示重要的生命周期的事件，例如：开始或者关闭。

脆弱的 volatile 的使用条件：

1：写入变量不依赖变量的当前值，或者能够保证只有单一的线程修改变量的值；
2：变量不需要和其他变量共同参与不变约束；
3：访问变量时不需要其他原因需要加锁。

```
private volatile boolean isInterrupted = false;
```

任务的取消和线程超时

```
class TimeoutThread extends Thread {
    private long timeout;
    volatile boolean isCancel = false;
    private TimeoutException timeoutException;
    public TimeoutThread(long timeout,TimeoutException timeoutException){
        super();
        this.timeout = timeout;
        this.timeoutException = timeoutException;
        this.setDaemon(true);
    }
    public void run() {
        try{
            sleep(timeout);
            if(!isCancel){
                throw timeoutException;
            }
        }catch(InterruptedException ie){

        }
    }
    public synchronized void isCancled(){
        isCancel = true;
    }
}
class TimeoutException extends RuntimeException{
    private static final long serialVersionUID = 1389067892664758L;

}
```

线程中断

```
private final BlockingQueue<String> queue = new LinkedBlockingQueue<String>();
private volatile boolean isCanceled = false;
public void run(){
    try {
        String str = "feed";
        while(!isCanceled){
                queue.put(str);
        }
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
    }
}
public void cancle(){
    isCanceled = true;
}
```

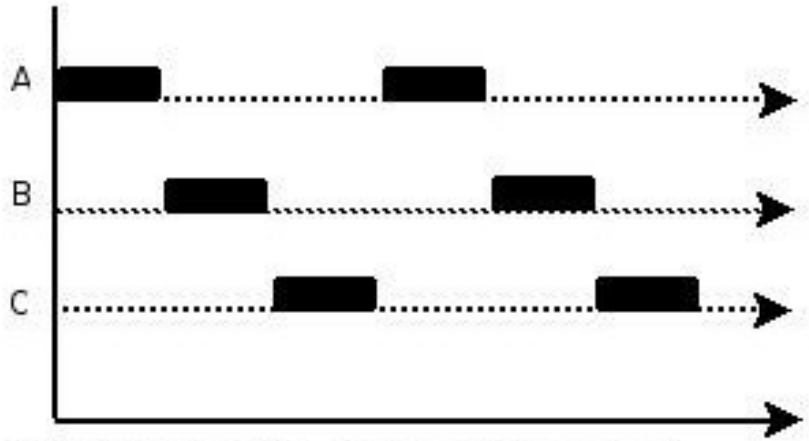
```
private final BlockingQueue<String> queue = new LinkedBlockingQueue<String>();
public void run(){
    try {
        String str = "feed";
        while(!Thread.currentThread().isInterrupted()){
                queue.put(str);
        }
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
    }
}
public void cancle(){
    this.interrupt();
}
```

教父 Joshua Bloch 说线程：

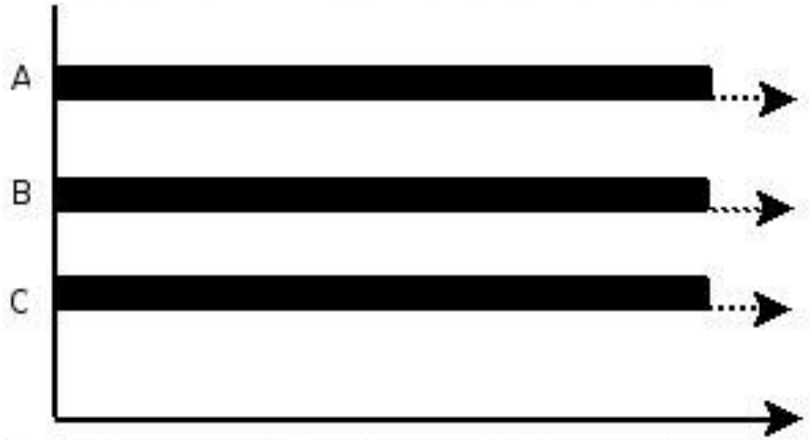
1. 对共享可变数据同步访问；
2. 避免过多的同步；
3. 永远不要在循环外面调用 wait ；
4. 不要依赖于线程调度器；
5. 线程安全的文档化；
6. 避免使用线程组。

目录

开始并发编程了



Concurrency : 1. Single Processor
2. logically simultaneous processing



Parallelism : 1. Multiprocessores, Multicore
2. Physically simultaneous processing

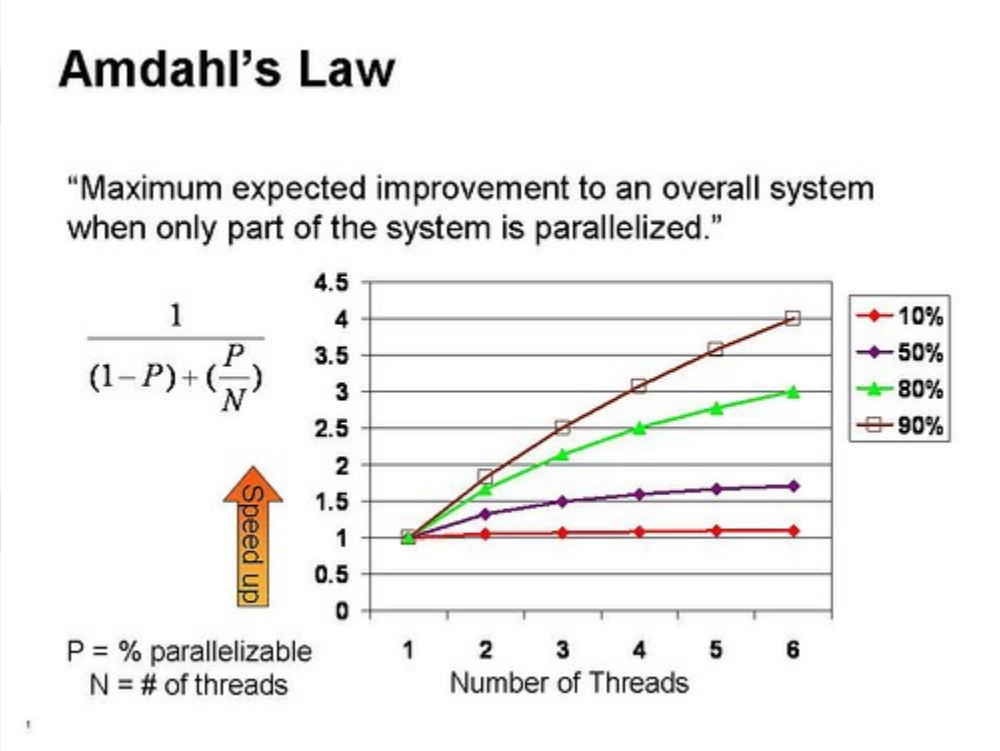
行动之前，拜神先

Doug Lea

1. Mr. concurrency ，当今世界上并发程序设计领域的先驱，著名学者。他是 util.concurrent 包的作者， JSR166 规范的制定。
2. 图书著作《 Concurrent Programming in Java: Design Principles and Patterns 》。
3. 其” A Scalable Elimination-based Exchange Channel” 和 ” Scalable Synchronous Queues” 两篇论文列为非阻塞同步算法的经典文章。
4. A fork/jion framework 同样影响着 java7 。

并发编程：三大定律（1）

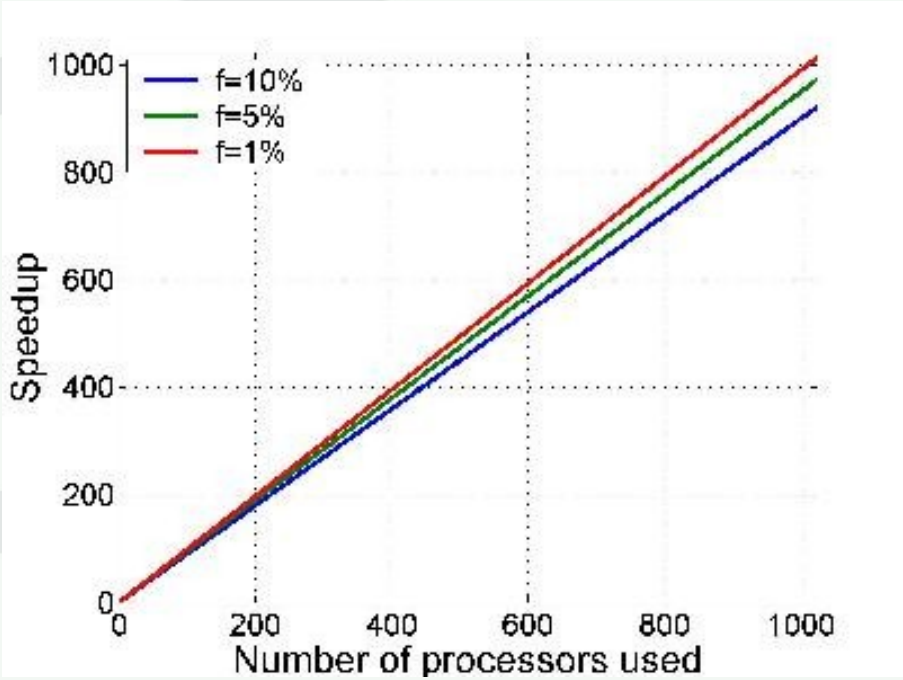
Amdahl定律



讨论的是加速比（speedup）的问题

并发编程：三大定律（2）

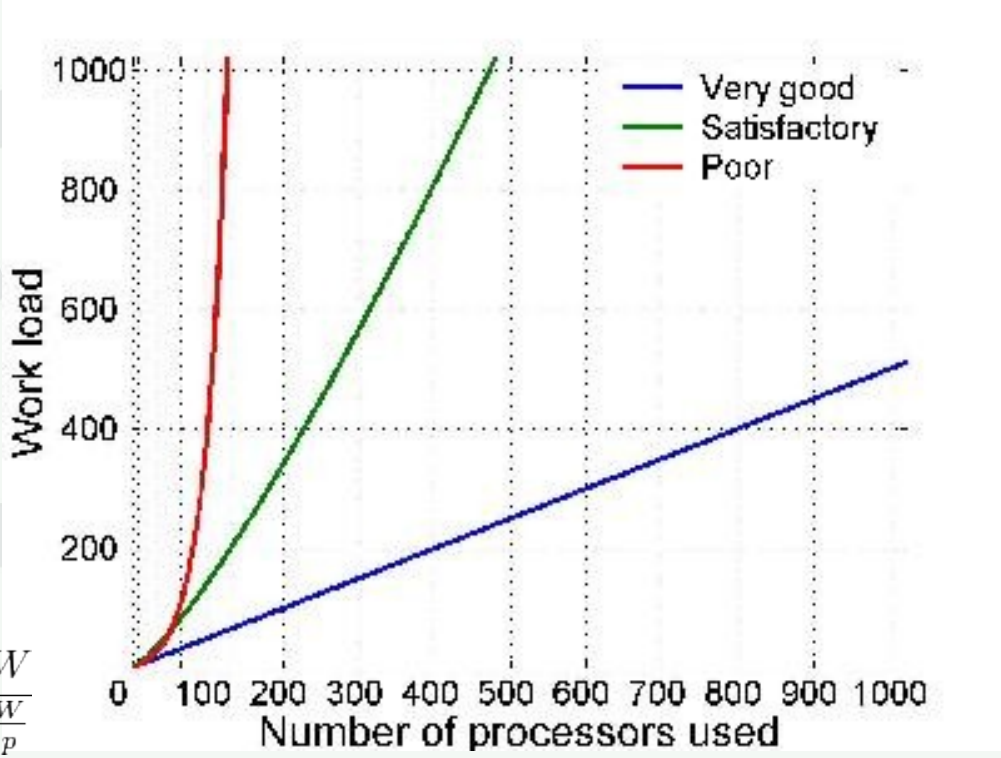
Gustafson 定律



Gustafson 假设随着处理器个数的增加，并行与串行的计算总量也是可以增加的。 Gustafson 定律认为加速系数几乎跟处理器个数成正比，如果现实情况符合 Gustafson 定律的假设前提的话，那么软件的性能将可以随着处理个数的增加而增加。

并发编程：三大定律（3）

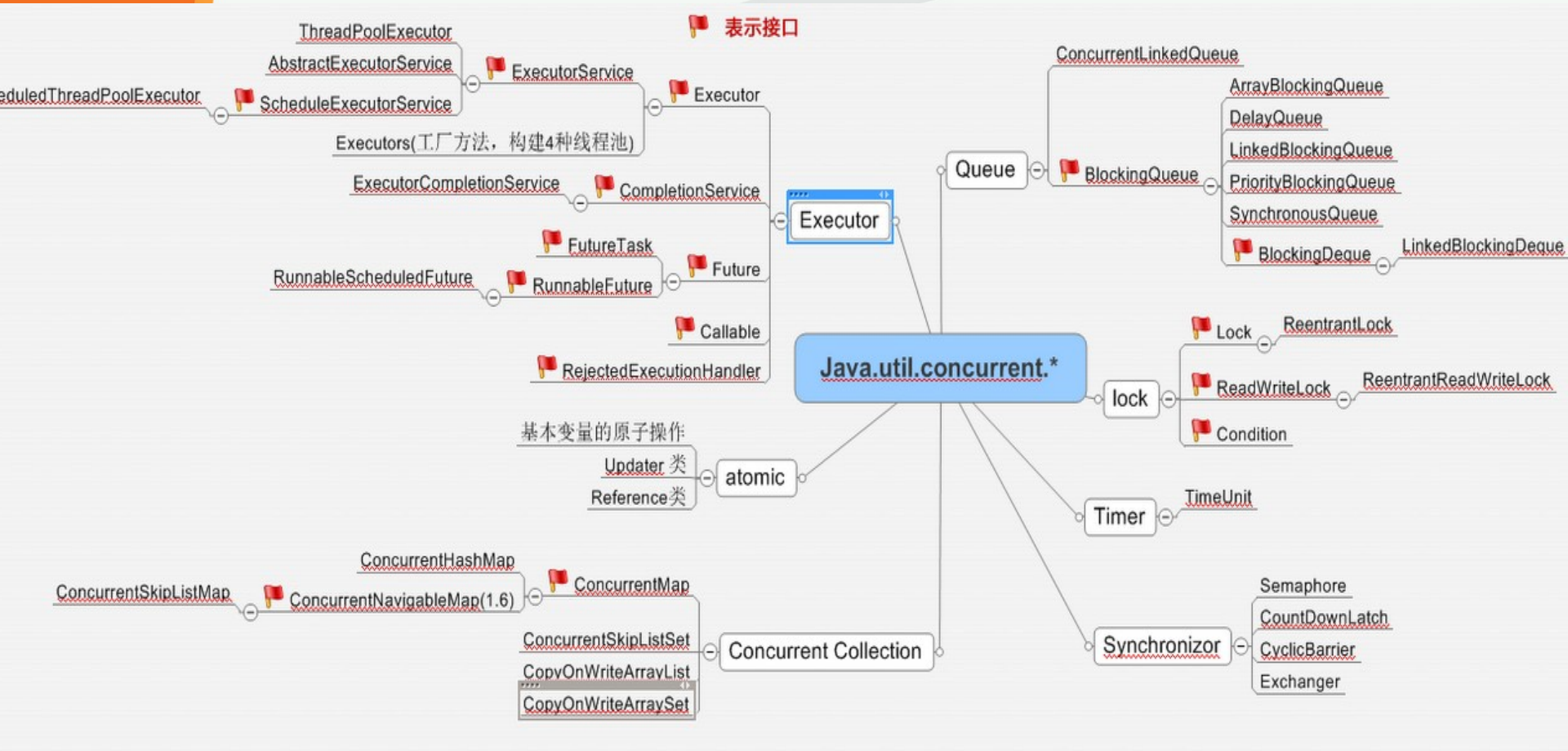
Sun-Ni定律



$$S = \frac{W_s + (1-f)G(p)W}{W_s + (1-f)G(p)\frac{W}{p}}$$

充分利用存储空间等计算资源，尽量增大问题规模以产生更好 / 更精确的解。

总结不是 API，是寂寞！


Java.util.concurrent.*
表示接口
Executor
ThreadPoolExecutor
AbstractExecutorService
ExecutorService
ScheduleExecutorService
eduledThreadPoolExecutor
Executors(工厂方法，构建4种线程池)
ExecutorCompletionService
CompletionService
FutureTask
RunnableScheduledFuture
RunnableFuture
Future
Callable
RejectedExecutionHandler
Queue
ConcurrentLinkedQueue
BlockingQueue
ArrayBlockingQueue
DelayQueue
LinkedBlockingQueue
PriorityBlockingQueue
SynchronousQueue
BlockingDeque
LinkedBlockingDeque
lock
Lock
ReentrantLock
ReadWriteLock
ReentrantReadWriteLock
Condition
Timer
TimeUnit
Synchronizor
Semaphore
CountDownLatch
CyclicBarrier
Exchanger
atomic
基本变量的原子操作
Updater 类
Reference类
Concurrent Collection
ConcurrentHashMap
ConcurrentMap
ConcurrentSkipListMap
ConcurrentNavigableMap(1.6)
ConcurrentSkipListSet
CopyOnWriteArrayList
CopyOnWriteArraySet

来个高清无码版

Executors
- *Executor*
- *ExecutorService*
- *ScheduledExecutorService*
- *Callable*
- *Future*
- *ScheduledFuture*
- *Delayed*
- *CompletionService*
- ThreadPoolExecutor
- ScheduledThreadPoolExecutor
- AbstractExecutorService
- Executors
- FutureTask
- ExecutorCompletionService

Queues
- *BlockingQueue*
- ConcurrentLinkedQueue
- LinkedBlockingQueue
- ArrayBlockingQueue
- SynchronousQueue
- PriorityBlockingQueue
- DelayQueue

Concurrent Collections
- *ConcurrentMap*
- ConcurrentHashMap
- CopyOnWriteArray{List,Set}

Synchronizers
- CountDownLatch
- Semaphore
- Exchanger
- CyclicBarrier

Timing
- TimeUnit

Locks
- *Lock*
- *Condition*
- *ReadWriteLock*
- AbstractQueuedSynchronizer
- LockSupport
- ReentrantLock
- ReentrantReadWriteLock

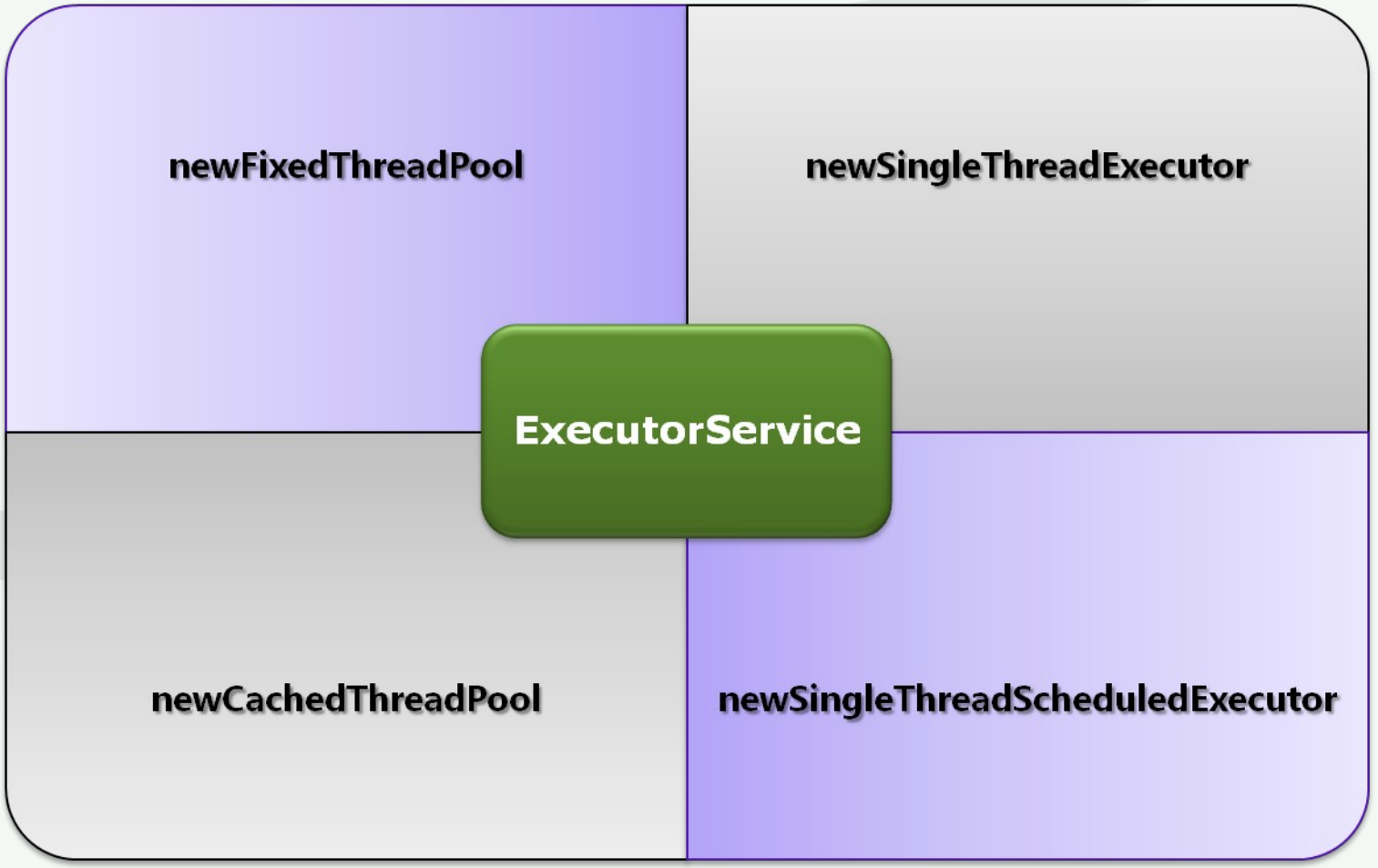
Atomics
- Atomic[Type], Atomic[Type]Array
- Atomic[Type]FieldUpdater
- Atomic{Markable,Stampable}Reference

ThreadPoolExecutor：自己动手，丰衣足食！

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

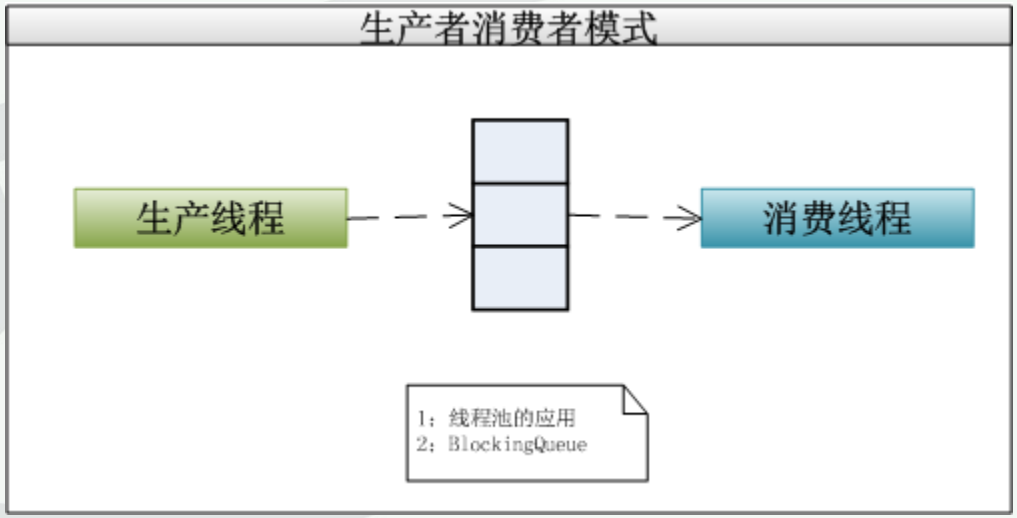
1：线程池的大小最好是设定好，因为 JDK 的管理内存毕竟是有限的；
2：使用结束，需要关闭线程池；
3： Runtime.*getRuntime().addShutdownHook(hook);* *对不能正常关闭的线程做好相关的记录。*

Executors： ExecutorService



严重注意：别设置线程池无限大小

入门版： CompletionService



生产者消费者模式的简要实现版本。

双剑合璧： Future+Callable

```
interface ArchiveSearcher { String search(String target); }
class App {
  ExecutorService executor = ...
  ArchiveSearcher searcher = ...
  void showSearch(final String target)
      throws InterruptedException {
    Future<String> future
      = executor.submit(new Callable<String>() {
        public String call() {
            return searcher.search(target);
        }});
    displayOtherThings(); // do other things while searching
    try {
      displayText(future.get()); // use future
    } catch (ExecutionException ex) { cleanup(); return; }
  }
}
```

任务池： ScheduledExecutorService

```
private ScheduledThreadPoolExecutor scheduler; // 任务执行者
private List<String> taskIdentities = new ArrayList<String>(); //简单防止任务重复提交

public DelayRetryTaskCentrel(int threadPoolSize) {
    scheduler = (ScheduledThreadPoolExecutor) Executors.newScheduledThreadPool(threadPoolSize);
    // 任务满载时，把任务丢给主线程继续执行
    scheduler.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy());
```

计划任务执行相关的操作，**使用 java 真幸福，选择多多！**

阻塞队列： BlockingQueue

	抛出异常	*特殊值*	*阻塞*	*超时*
插入	add(e)	offer(e)	put(e)	offer(e, time
移除	remove()	poll()	take()	poll(time, un
检查	element()	peek()	*不可用*	*不可用*

Kaopuability：插入（ offer ）；移除（ poll ）

BlockingQueue 的诸侯领地

- ArrayBlockingQueue: 一个由数组支持的有界阻塞队列。此队列按 FIFO （先进先出）原则对元素进行排序。
- Delayed 元素的一个无界阻塞队列，只有在延迟期满时才能从中提取元素 .
- LinkedBlockingDeque 一个基于已链接节点的、任选范围的阻塞双端队列。
- LinkedBlockingQueue 一个基于已链接节点的、范围任意的 blocking queue 。此队列按 FIFO （先进先出）排序元素
- PriorityBlockingQueue 一个无界阻塞队列，它使用与类 PriorityQueue 相同的顺序规则，并且提供了阻塞获取操作。
- SynchronousQueue 一种阻塞队列，其中每个插入操作必须等待另一个线程的对应移除操作 ，反之亦然。

BlockingDeque: 双端队列

第一个元素（头部）				
	抛出异常	*特殊值*	*阻塞*	*超时期*
插入	addFirst(e)	offerFirst(e)	putFirst(e)	offerFirst(e, tir
移除	removeFirst()	pollFirst()	takeFirst()	pollFirst(time,
检查	getFirst()	peekFirst()	*不适用*	*不适用*
最后一个元素（尾部）				
	抛出异常	*特殊值*	*阻塞*	*超时期*
插入	addLast(e)	offerLast(e)	putLast(e)	offerLast(e, tin
移除	removeLast()	pollLast()	takeLast()	pollLast(time,
检查	getLast()	peekLast()	*不适用*	*不适用*

并发集合：你值得拥有

```
public void putInfoA(String name){
    Map<String,String> map = Collections.synchronizedMap(new HashMap<String,String>());
    map.put(name, name);
    //do something
    Set<String> s = map.keySet();
    Iterator<String> i = s.iterator();
    while (i.hasNext()){
        //do something(i.next());
    }
}
```

同步的不是 **Map**, 是凤姐！

```
public void putInfoB(String name){
    Map<String,String> map = Collections.synchronizedMap(new HashMap<String,String>());
    map.put(name, name);
    //do something
    Set<String> s = map.keySet();  // Needn't be in synchronized block
    synchronized(map) {  // Synchronizing on m, not s!
      Iterator<String> i = s.iterator(); // Must be in synchronized block
      while (i.hasNext()){
          //do something(i.next());
      }
    }
}
```

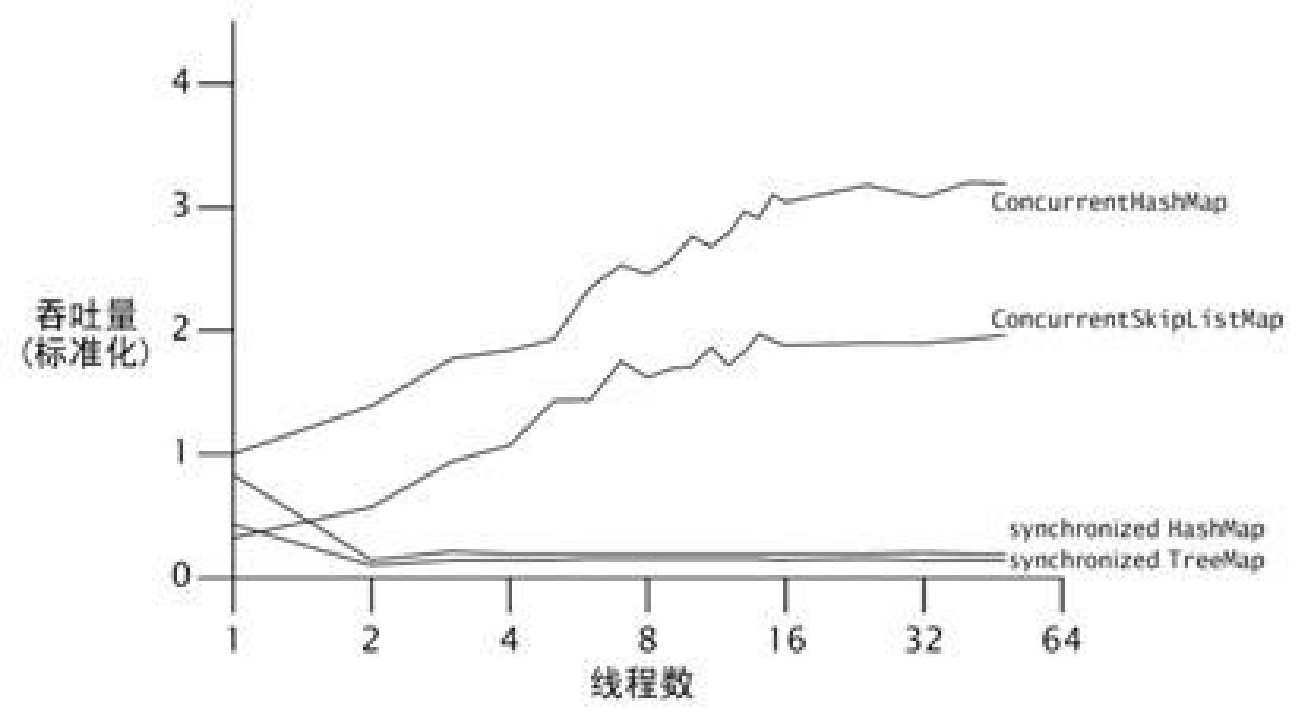
ConcurrentHashMap: 解放军 38 军

```
public void putInfoC(String name){
    Map<String,String> map = new ConcurrentHashMap<String,String>();
    map.put(name, name);
    //do something
    Set<String> s = map.keySet();
    Iterator<String> i = s.iterator();
    while (i.hasNext()){
        //do something(i.next());
    }
}
```

你那飘逸的同步，分离锁的设计，再 HASH 算法以及游离于多个 SEGMENT 的耍心跳的各种操作，都深深的吸引了我。

详细设计细节：
http://www.ibm.com/developerworks/java/library/j-jtp08223/

比较 Map 的性能


4
3
2
1
0
吞吐量
(标准化)
ConcurrentHashMap
ConcurrentSkipListMap
synchronized HashMap
synchronized TreeMap
1
2
4
8
16
32
64
线程数

CopyOnWriteArray{List,Set}

当读操作远远大于写操作的时候，考虑用这个并发集合。例如：维护监听器的集合。注意：其频繁写的效率可能低的惊人。

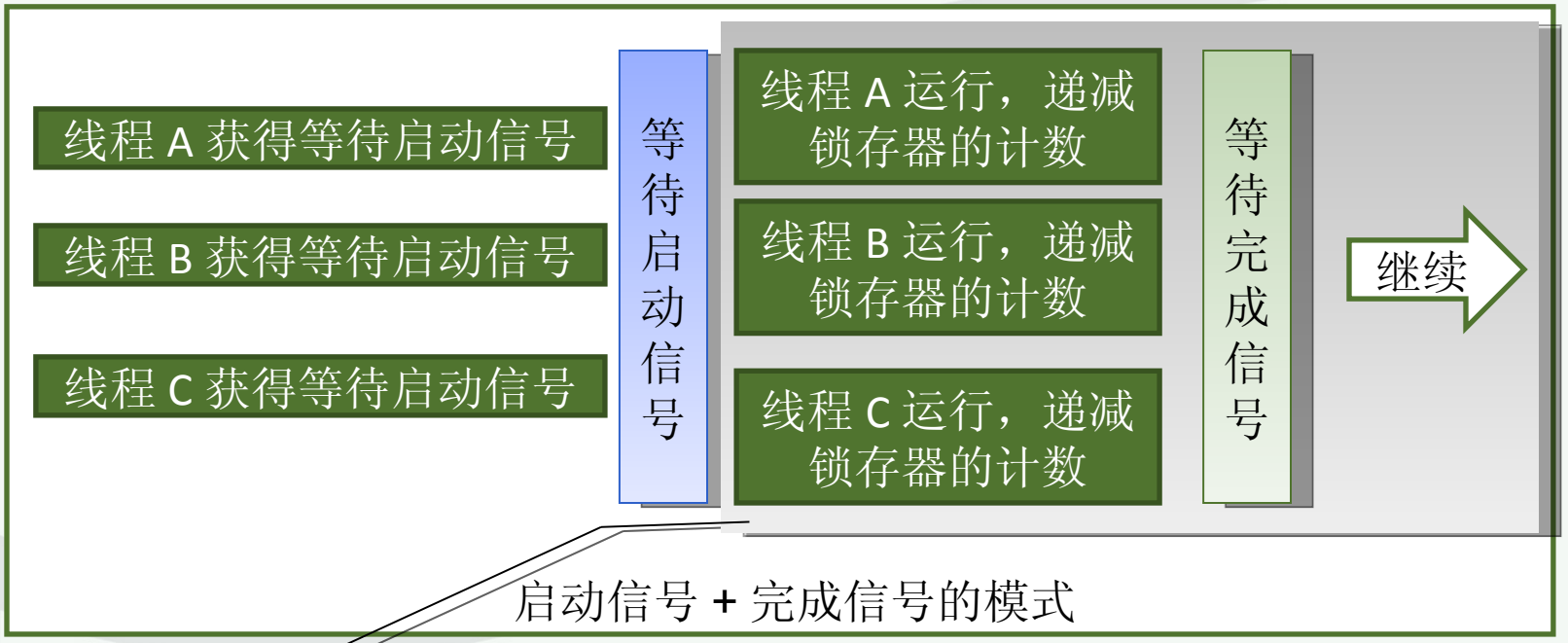
```
public boolean add(E e) {
final ReentrantLock lock = this.lock;
lock.lock();
try {
    Object[] elements = getArray();
    int len = elements.length;
    Object[] newElements = Arrays.copyOf(elements, len + 1);
    newElements[len] = e;
    setArray(newElements);
    return true;
} finally {
    lock.unlock();
}
}
```

奇技淫巧：屏蔽 add 带来的数组拷贝；

```
public List<String> array = new ArrayList<String>();
public List<String> list = new CopyOnWriteArrayList<String>(array);
```

同步器：四大金刚

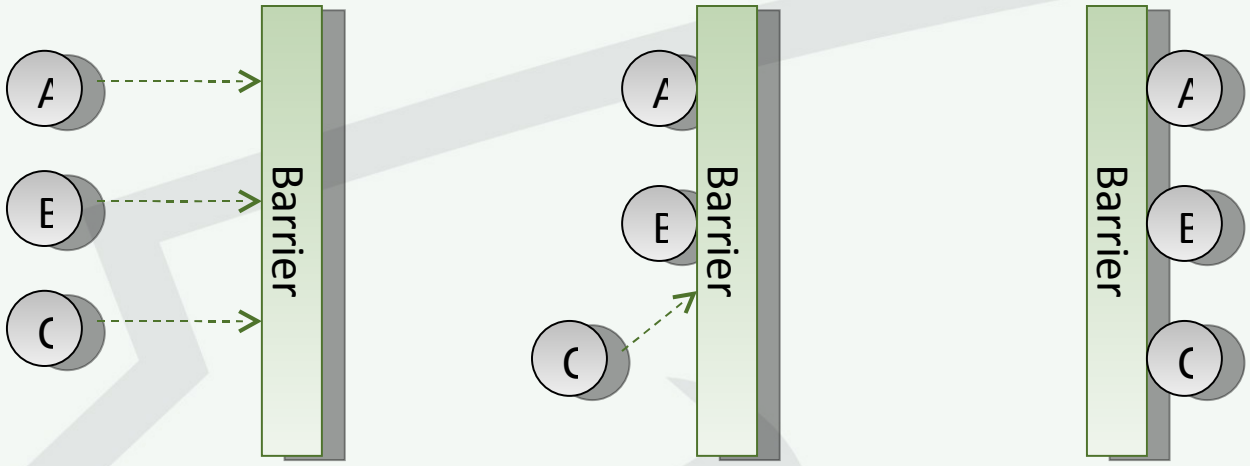
闭锁： CountDownLatch



N 部分锁存器倒计数模式；
当线程必须用这种方法反复倒计数时，可改为使用 CyclicBarrier

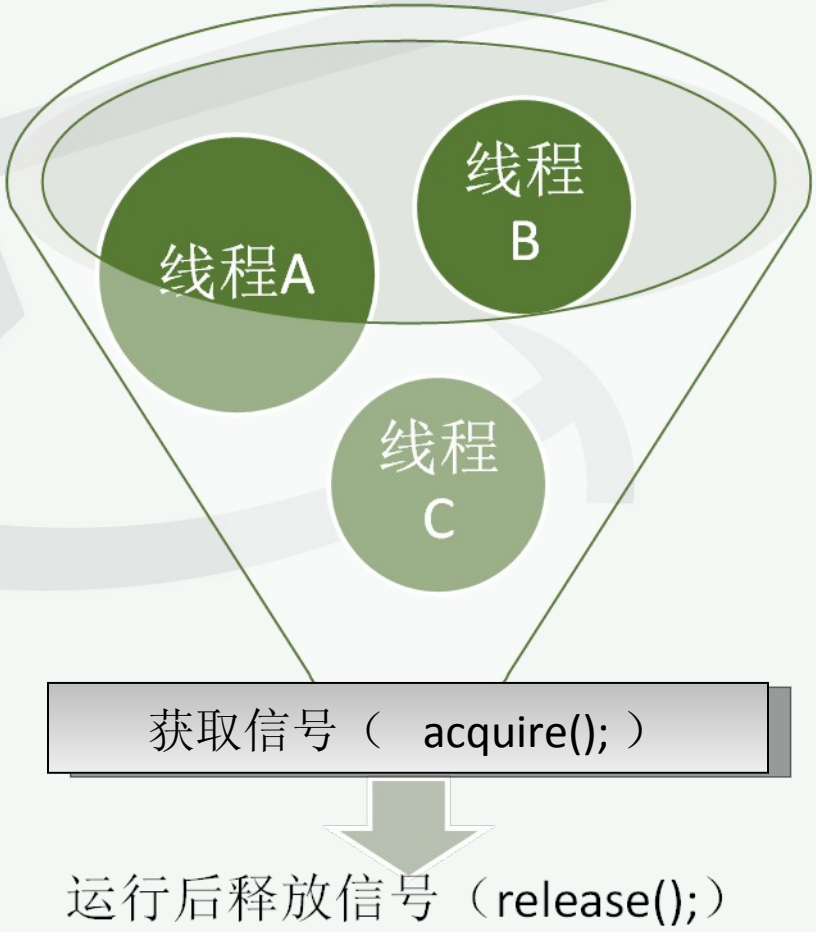
典型应用：手动控制事务，从数据库读取多份数据做初始化；

关卡： CyclicBarrier



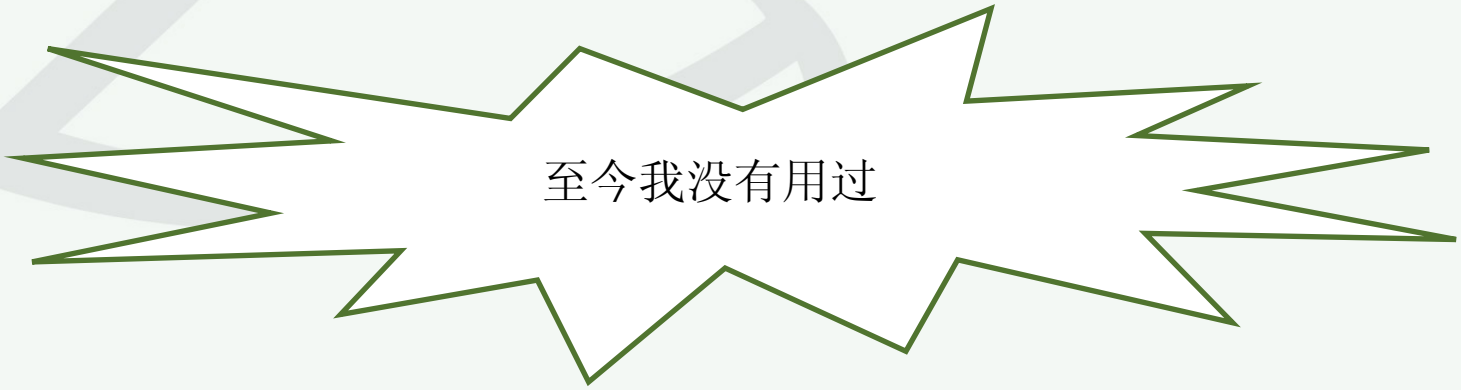
A barrier: A barrier is a coordination mechanosm (an algorithm) that forces process which participate in a concurrent (or distributed) algorithm to wait until each one of them has reached a certain point in its program. The collection of these coordination points is called the barrier. Once all the processes have reached the barrier, they are all permitted to continue past the barrier.

信号量： Semaphore



交换器： Exchanger

1. 数据分解和数据流分解的一种技巧，可被视为 SynchronousQueue 的双向形式。
2. JDK5 时支持 2 个线程之间交换数据， JDK6 后支持多个。



锁云：你的柔情我永远不懂

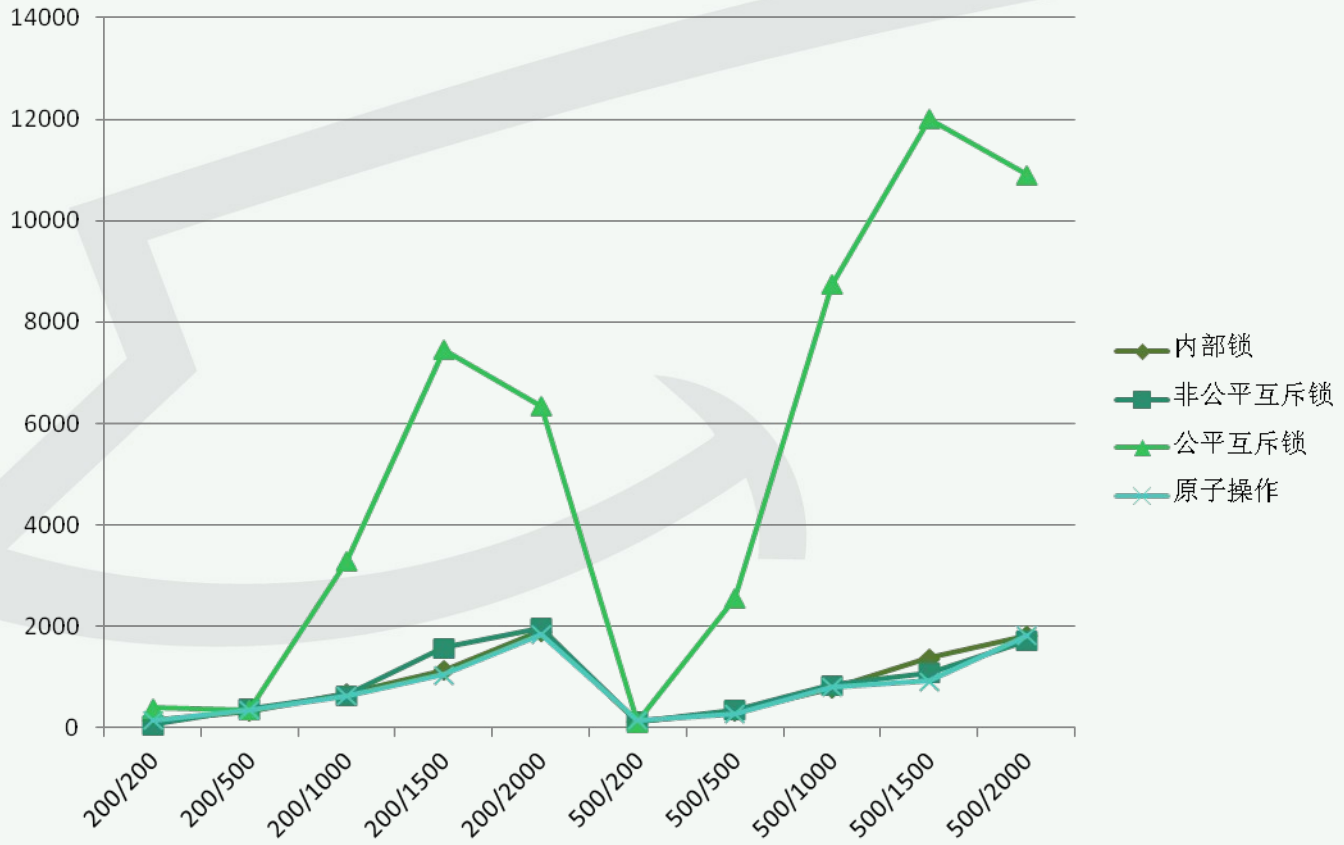
互斥锁： ReentrantLock

- Lock 更加灵活，性能更好

```
interface Lock {
        void lock();
        void lockInterruptibly() throws InterruptedException;
        boolean tryLock();
        boolean tryLock(long timeout, TimeUnit unit)
                                        throws InterruptedException;
        void unlock();
        Condition newCondition();
}
```

- 支持多个 Condition
- 可以不以代码块的方式上锁
- 可以使用 tryLock，并指定等待上锁的超时时间
- 调试时可以看到内部的 owner thread，方便排除死锁
- RenntrantLock 支持 fair 和 unfair 两种模式

互斥锁（公平与非公平）和内部锁性能对比图

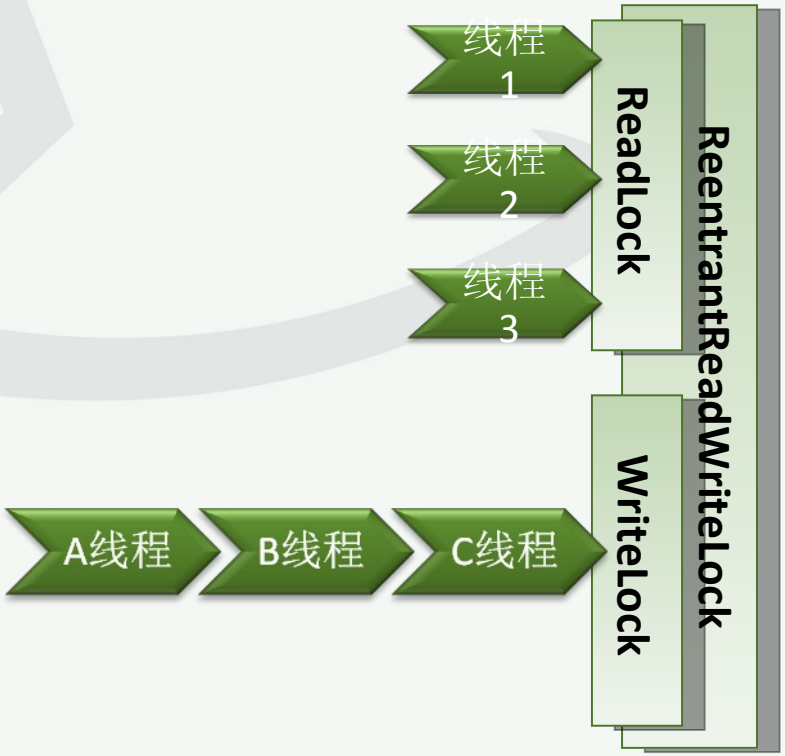


结论：非公平互斥锁和内部锁性能在 jdk6_17 下性能基本一样。

测试机：酷睿 2.66 双核， 2G 内存， win7

读写锁： 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。

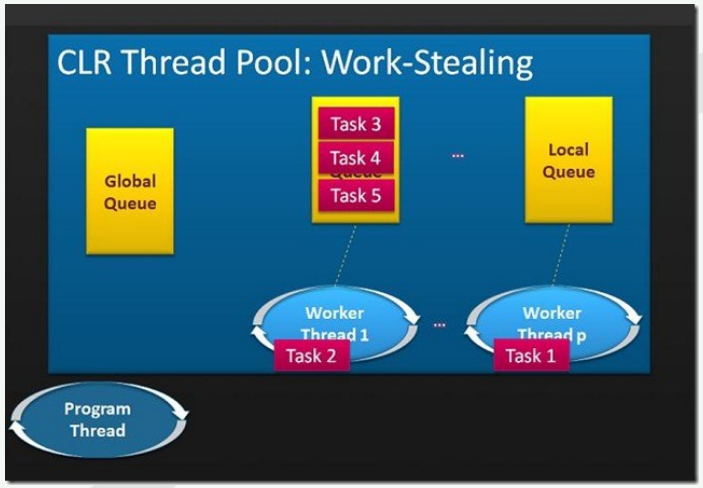


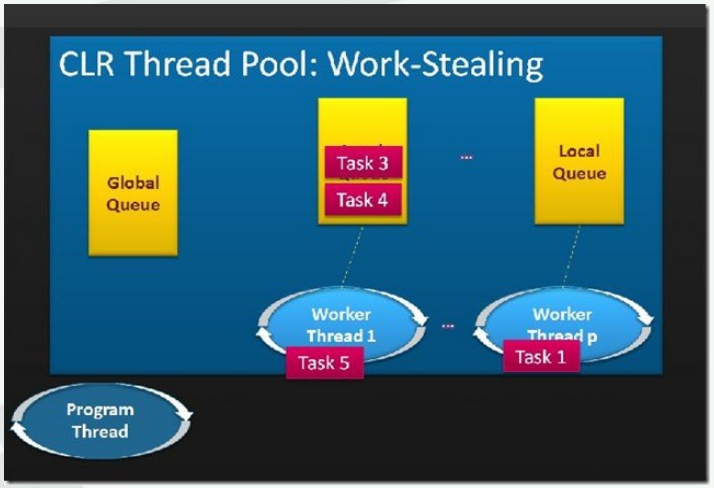
高深：AbstractQueuedSynchronizer

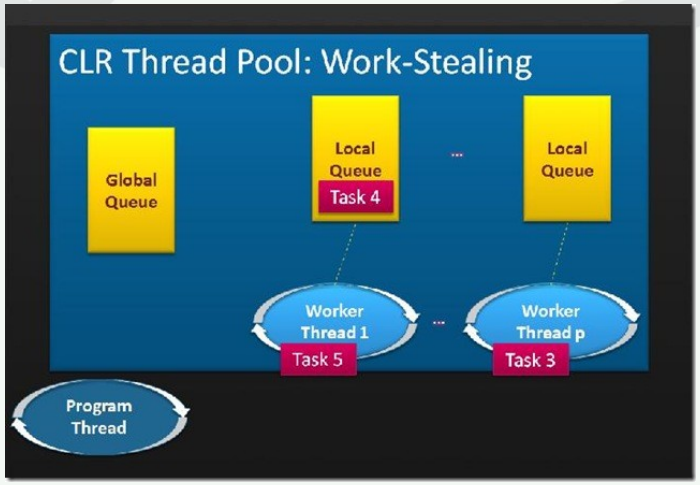
- 为实现依赖于先进先出 (FIFO) 等待队列的阻塞锁和相关同步器（信号量、事件，等等）提供一个框架；
- 基于 JDK 底层来操控线程调度， LockSupport.park 和 LockSupport.unpark ；
- 此类支持默认的*独占* 模式和*共享* 模式之一，或者二者都支持；
- 如果想玩玩它，请看 CountDownLatch 的源码。
- 当然，也可以去 FutureTask 这个类看看。

目录

Fork/Join Framework (1)







Work Stealing

Fork/Join Framework

- FJTask 框架是 Cilk 用到的设计方案的变体，相关系统依赖于轻量级可执行任务。这些框架都是像操作系统映射线程到 CPU 一样来映射任务到线程，从而开发单纯，有规律，有约束力的 fork/join 程序来执行映射动作。
- 工作线程池是确定的，每个工作线程（ Thread 子类 FJTaskRunner 的一个实例）是处理队列中任务的标准线程。正常的，工作线程的数量和系统的 CPU 的数量是一致的。任何合乎情理的映射策略将把这些线程映射到不同的 cpu 。
- 所有的 fork/join 任务都是一个轻量级可执行类的实例，不是一个线程实例。在 java 中，独立的可执行任务必须实现 Runnable 接口并定义一个 run() 方法。在 FJTask 框架中，这些任务是 FJTask 的子类而不是 Thread 的子类，它们都实现了 Runnable 接口
- 一个简单的控制和管理设施（这里讲的是 FJTaskRunnerGroup ）设置工作池，从一个正常的线程（例如 java 语言中的 main() 方法）调用启动执行提供的 fork/join 任务。

Doug Lea 论文： **http://gee.cs.oswego.edu/dl/papers/fj.pdf**

Fork/Join 的案例：求数组中的最大值

```
public class MaxWithFJ extends RecursiveAction {
    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 = ...
        int nThreads = ...
        MaxWithFJ mfj = new MaxWithFJ(problem, threshold);
        ForkJoinExecutor fjPool = new ForkJoinPool(nThreads);

        fjPool.invoke(mfj);
        int result = mfj.result;
    }
}
```

Fork/Join 运行测试结果

	阈值 = 500k	阈值 = 50k	阈值 = 5k	阈值 = 500	阈值 = -50
Pentium-4 HT （2 个线程）	1.0	1.07	1.02	.82	.2
Dual-Xeon HT （4 个线程）	.88	3.02	3.2	2.22	.43
8-way Opteron （8 个线程）	1.0	5.29	5.73	4.53	2.03
8-core Niagara （32 个线程）	.98	10.46	17.21	15.34	6.49

fork-join 池中的线程数量与可用的硬件线程（内核数乘以每个内核中的线程数）相等

Fork/Join 有用的资源

jsr166 ：

http://gee.cs.oswego.edu/dl/concurrency-interest/index.html

Java 理论与实践：应用 fork-join 框架： Brian Goetz

http://www.ibm.com/developerworks/cn/java/jjp11137.html

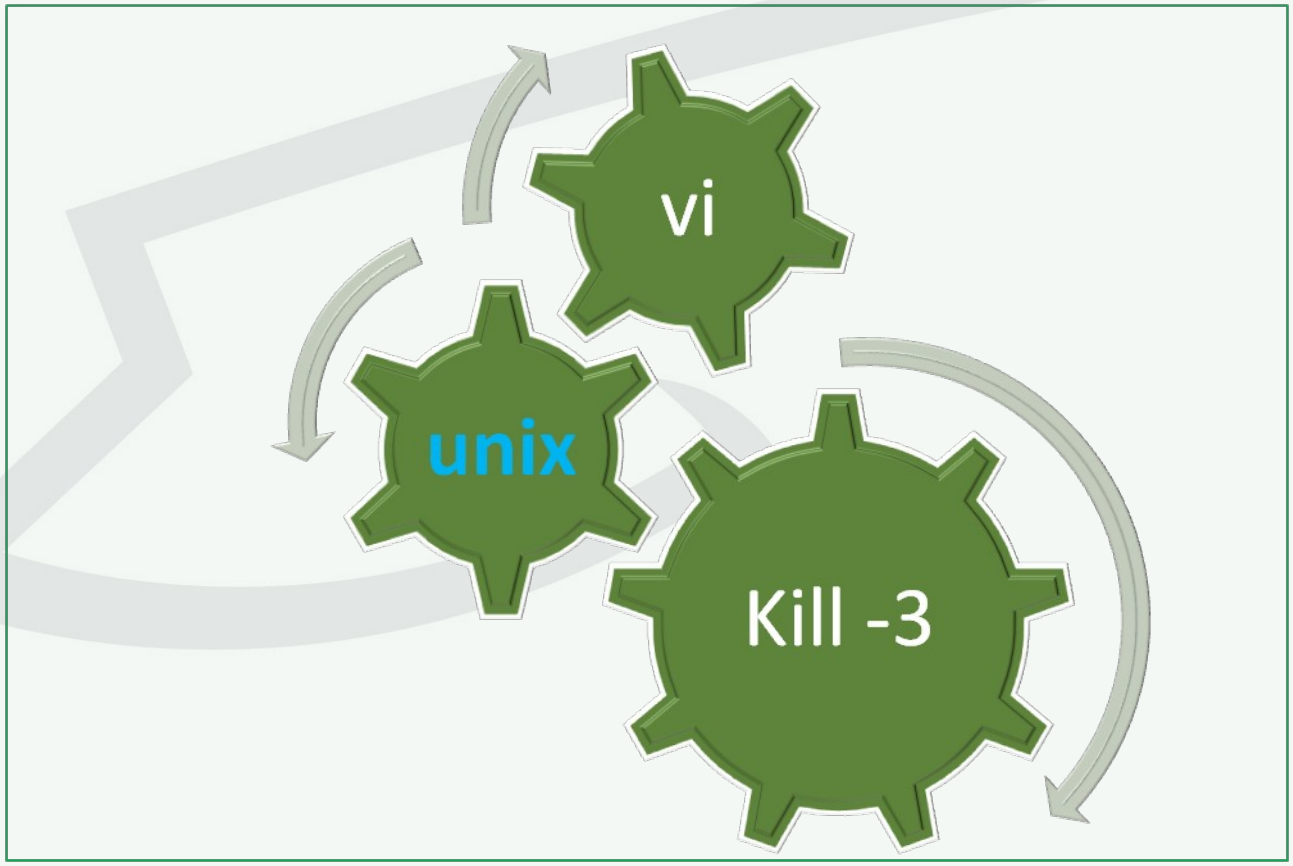
http://www.ibm.com/developerworks/cn/java/jjp03048.html

JDK 7 中的 Fork/Join 模式

http://www.ibm.com/developerworks/cn/java/j-lo-forkjoin/index.html

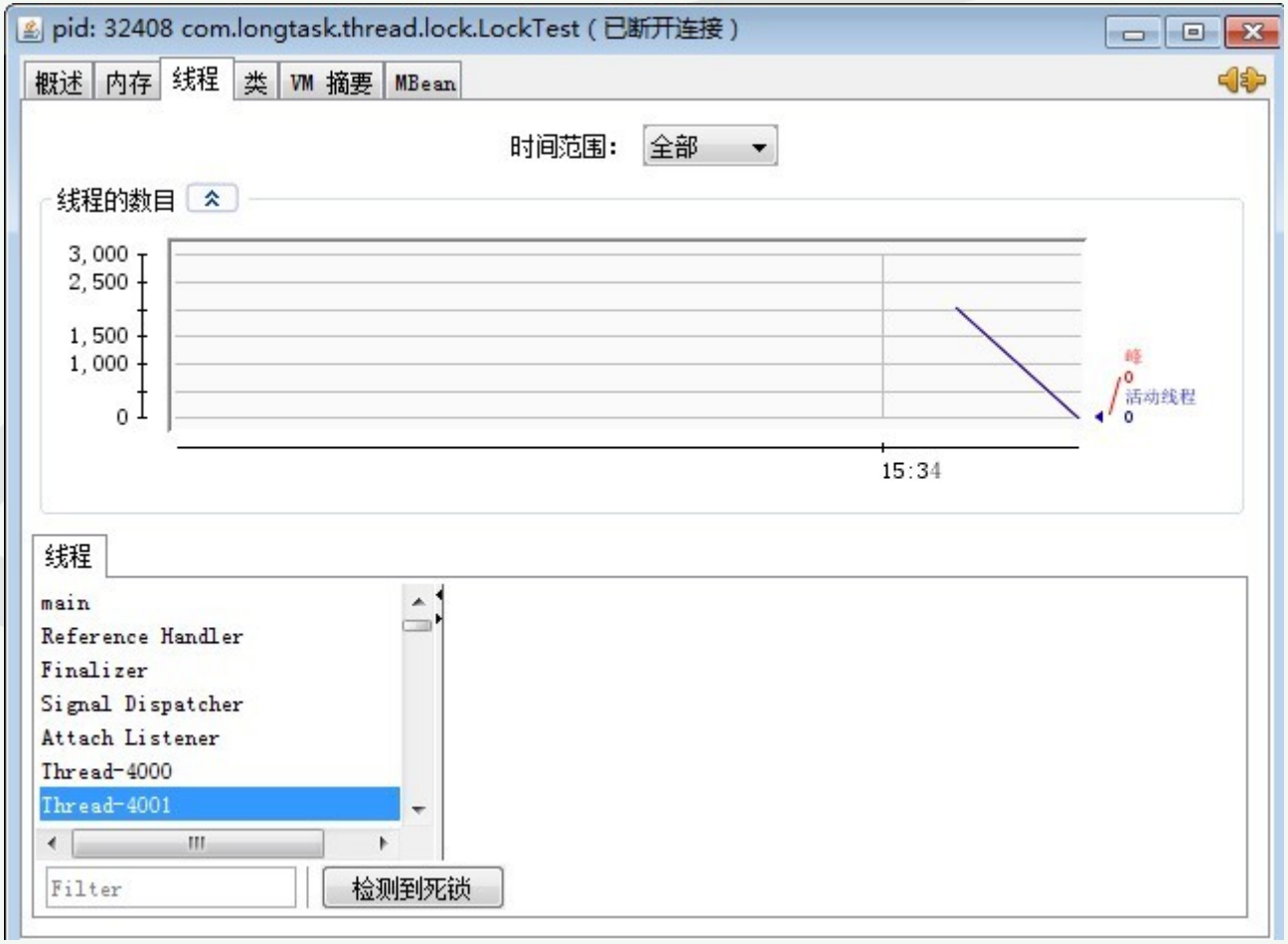
目录

线程监控： DIY

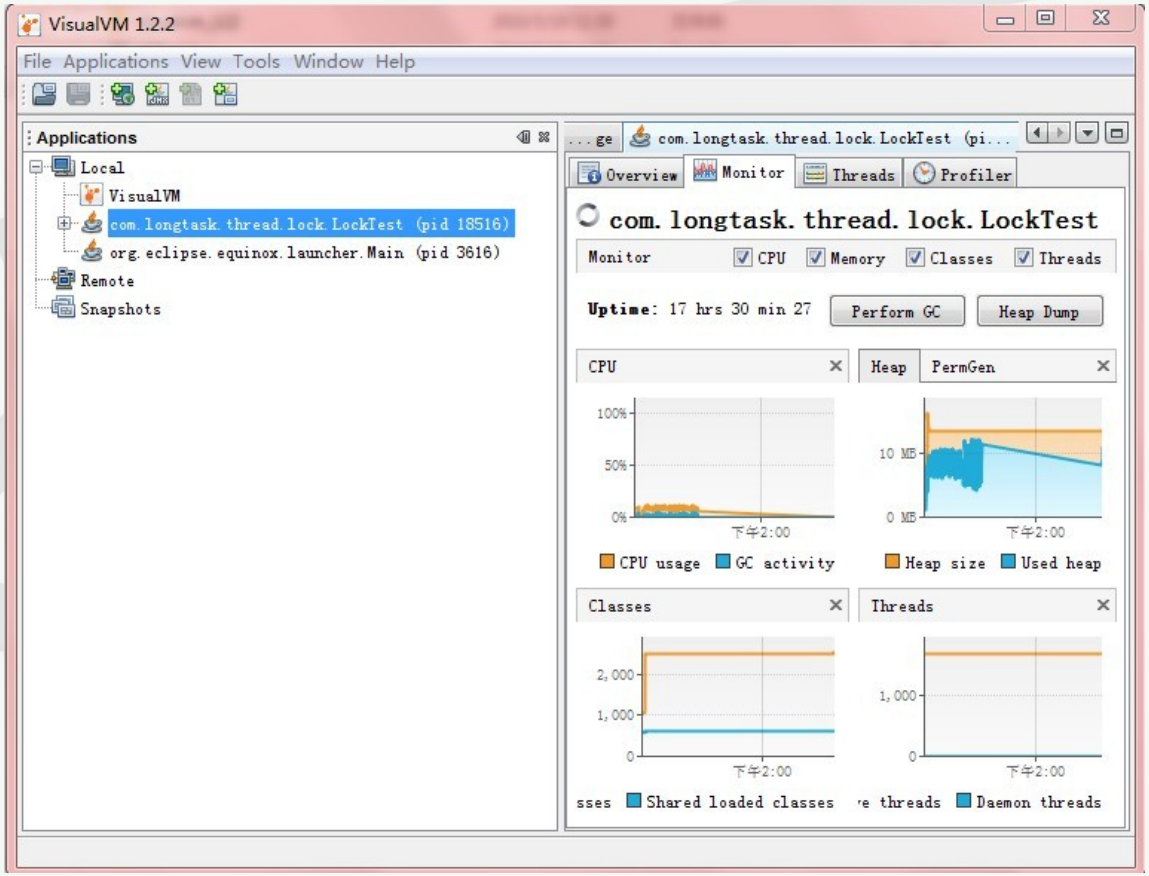


Windows 下面 DIY 方式很多， txt 编辑器都是好东西

线程监控： Jconsole

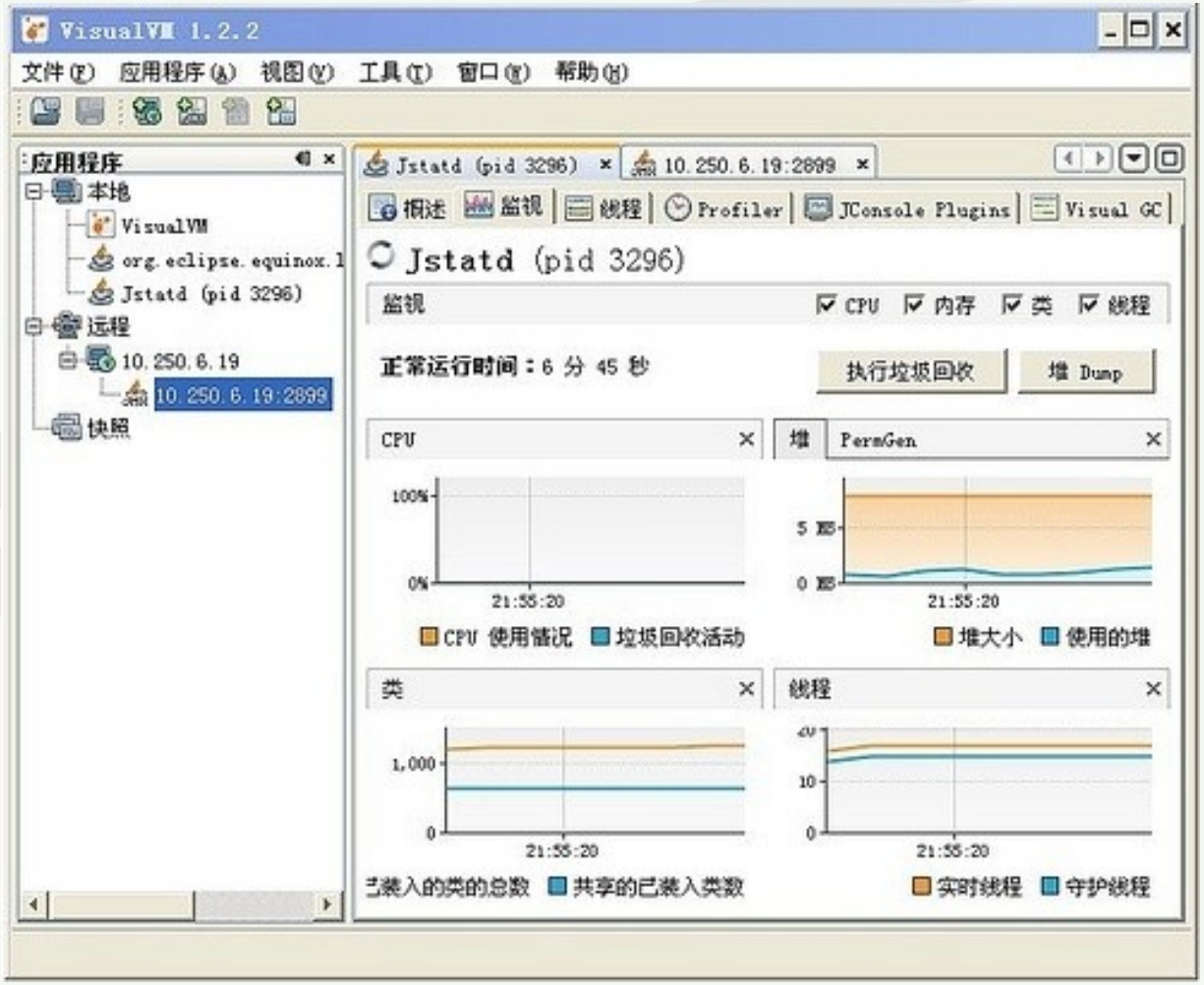


线程监控： VisualVm 本地

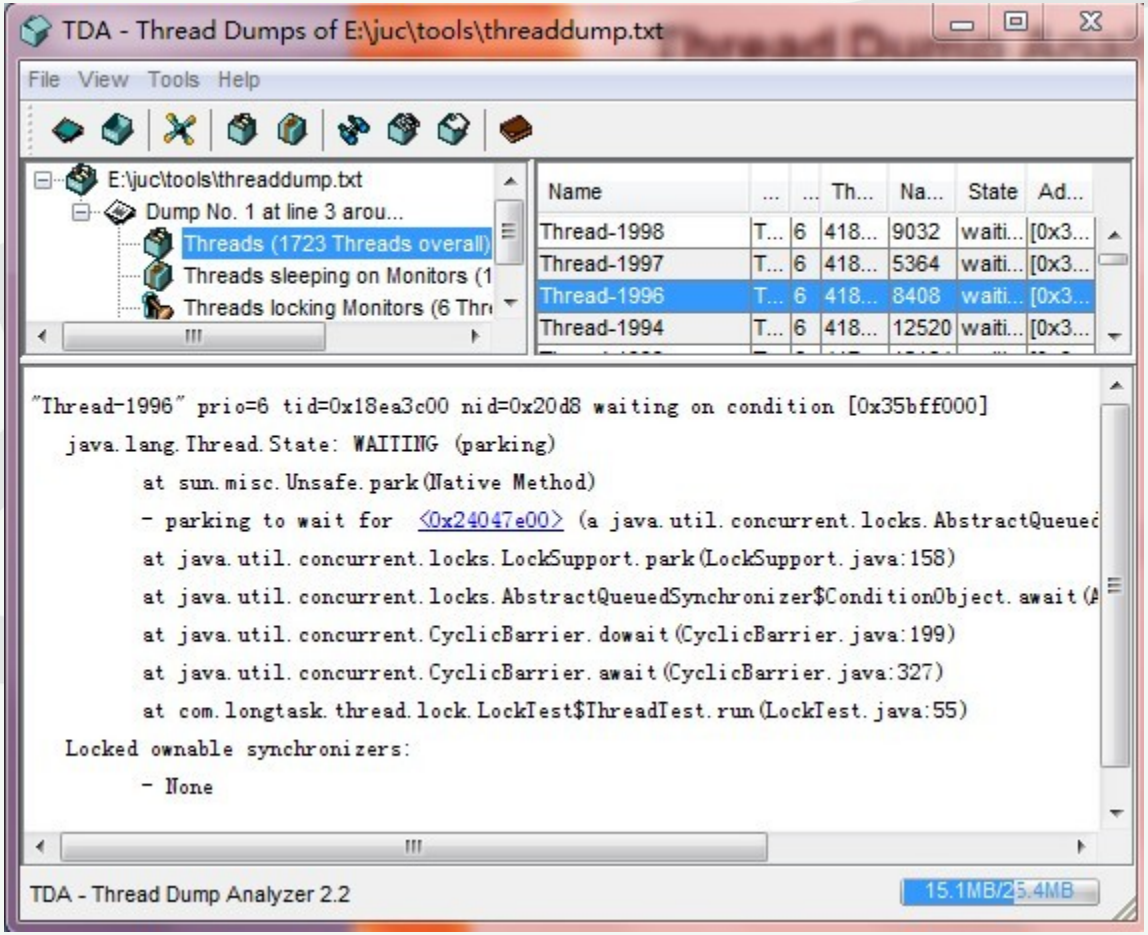


下载地址： https://visualvm.dev.java.net/download.html
JDK1.6 之后自带了这个攻击，在 java_home/bin 下面；
喜新厌旧的 GGDDJJMM 去上述地址下载吧！

内存监控： VisualVm 远程



Thread Dump Analyzer



https://tda.dev.java.net/

目录

基本思想： CAS 操作

- 处理器指令，全称 Compare and swap，原子化的读 - 该 - 写指令
- 处理竞争策略：单个线程胜出，其他线程失败，但不会挂起

基本思想： Atomic 原子类

- 基于硬件 CAS 实现 Atomic 类
- 分四组：计量器、域更新器、数组、复合变量
- 更佳的 volatile
- 目标 1：实现复杂算术运算：incrementAndGet 、 getAndIncrement 等
- 目标 2：支持 Java 类型对象的 CAS 操作： compareAndSet

基本思想：非阻塞算法

- 一个线程的失败或挂起不影响其他线程
- J.U.C 的非阻塞算法依赖 Atomic
- 算法里一般采用回退机制处理 Atomic 的 CAS 竞争
- 对死锁免疫的同步机制
- 目标：相对阻塞算法，减少线程切换开销、减少锁的竞争等。
- 也是 Lock-free ，即无锁编程

无锁栈算法： Treiber 算法

```
1 public class ConcurrentStack<E> {
2    AtomicReference<Node<E>> head = new AtomicReference<Node<E>>();
3    public void push(E item) {
4       Node<E> newHead = new Node<E>(item);
5       Node<E> oldHead;
6       do {
7          oldHead = head.get();
8          newHead.next = oldHead;
9       } while (!head.compareAndSet(oldHead, newHead));
10   }
11   public E pop() {
12      Node<E> oldHead;
13      Node<E> newHead;
14      do {
15         oldHead = head.get();
16         if (oldHead == null)
17            return null;
18         newHead = oldHead.next;
19      } while (!head.compareAndSet(oldHead,newHead));
20      return oldHead.item;
21   }
22   static class Node<E> {
23      final E item;
24      Node<E> next;
25      public Node(E item) { this.item = item; }
26   }
27 }
```

改进版： http://research.microsoft.com/en-us/um/cambridge/projects/terminator/popl09.pdf

无锁队列算法： MS-Queue 算法

MS-queue 算法是 1996 年由 Maged. M .Michael and M. L. Scott 提出的，是最为经典的并发 FIFO 队列上的算法，目前很多对并发 FIFO 队列的研究都是基于这个算法来加以改进的。

MS-queue 算法的队列用一个单链表来实现，包含两个基本的操作， enquene() 和 dequene() ，新节点总是从队尾最后一个元素后面加入队列，节点元素总是从队头删除。包含两个指针， head 和 tail ， head 总是自相链表头部的节点，指向的这个节点被当作是哑节点或哨兵节点，它保存的值是多少并无意义； tail 总是指向链表中的一个节点，不一定是队尾元素。每个节点包含两个数据域值信息，即存放的数值信息和指向下一个节点的指针。每个指针对象，除了包含一个指向节点的指针外，还包含一个时间戳，初试时时戳为零，每修改一次指针，时戳增加一，在 64 位系统中，无需考虑时戳溢出的影响。

论文地址：
http://www.research.ibm.com/people/m/michael/podc-1996.pdf

无锁队列算法： Optitmistic 算法

Optimistic 算法对于上面提到的 MS-queue 算法的改进就在于使用普通的 store 指令代替代价昂贵的 CAS 指令。

Optimistic 算法的高效性在于使用双向链表表示队列，并且入队和出队操作都只需要一次成功的 CAS 操作。该算法保证链表总是连接的，next 指针总是一致的，当 prev 指针出现不一致时通过调用 fixList 方法能够恢复到一致性状态。

同 MS-queue 算法一样， optimistic 算法也用到了原子化的指令 Compare-and-swap(CAS)， CAS(a， p， n)，原子化的将内存地址 a 中的值与 p 进行比较，如果二者相等，就将 n 写入地址 a 中并返回 true，否则返回 false 。由于 optimistic 算法使用了 CAS 指令，所以经典的 ABA 问题同样会出现，解决方案同 MS-queue 相同，即使用标签机制。

论文地址： http://nedko.arnaudov.name/soft/L17_Fober.pdf

Atomic 实现

```
public final int incrementAndGet() {
        for (;;) {
            int current = get();
            int next = current + 1;
            if (compareAndSet(current, next))
                return next;
        }
    }

public final boolean compareAndSet(int expect, int update) {

        return unsafe.compareAndSwapInt(this, valueOffset, expect,
update);

    }
```

当 import sun.misc.Unsafe; 这个的时候，就因为各种问题（例如：专利）看不到源码了。

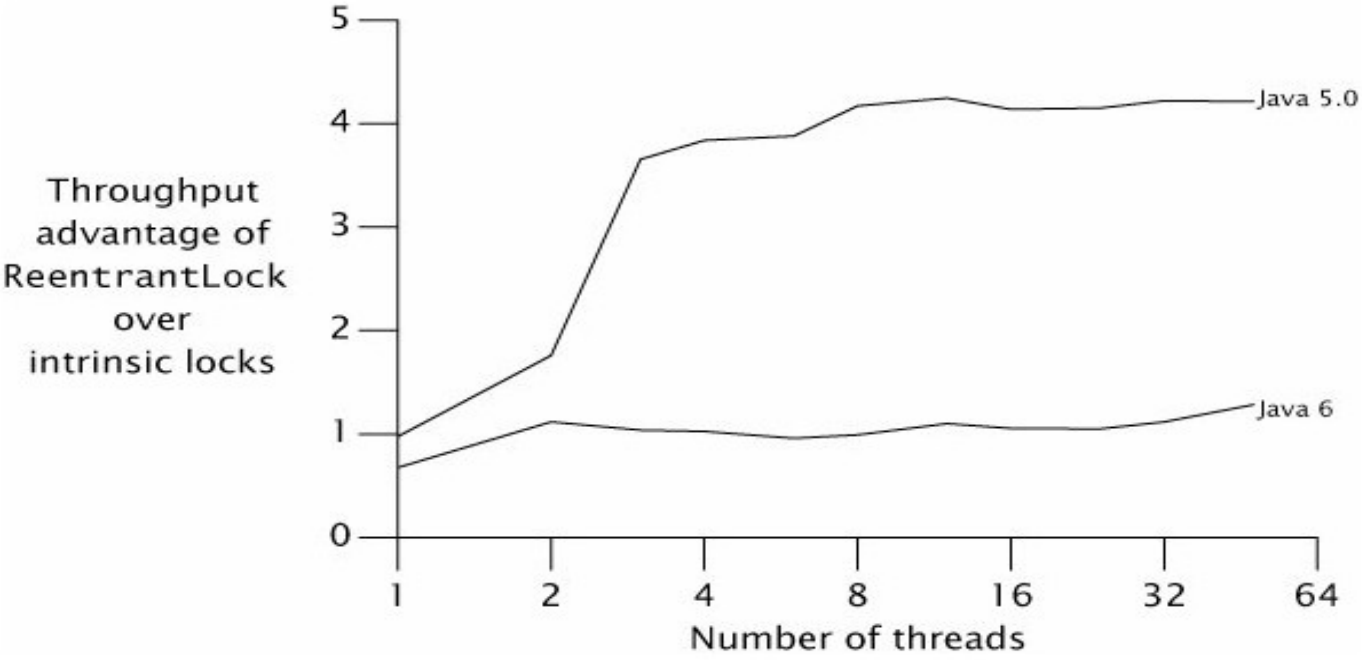
好东东： AtomicReference

Lock-free 的数据结构就靠这个了，无论你喜欢与否，玩无锁编程，你都绕不开这个类。看 amino 框架的源码，你会发现这个妞无处不在。

当然，还是 AtomicInteger 比较实用，多线程计数的时候，你会喜欢的。

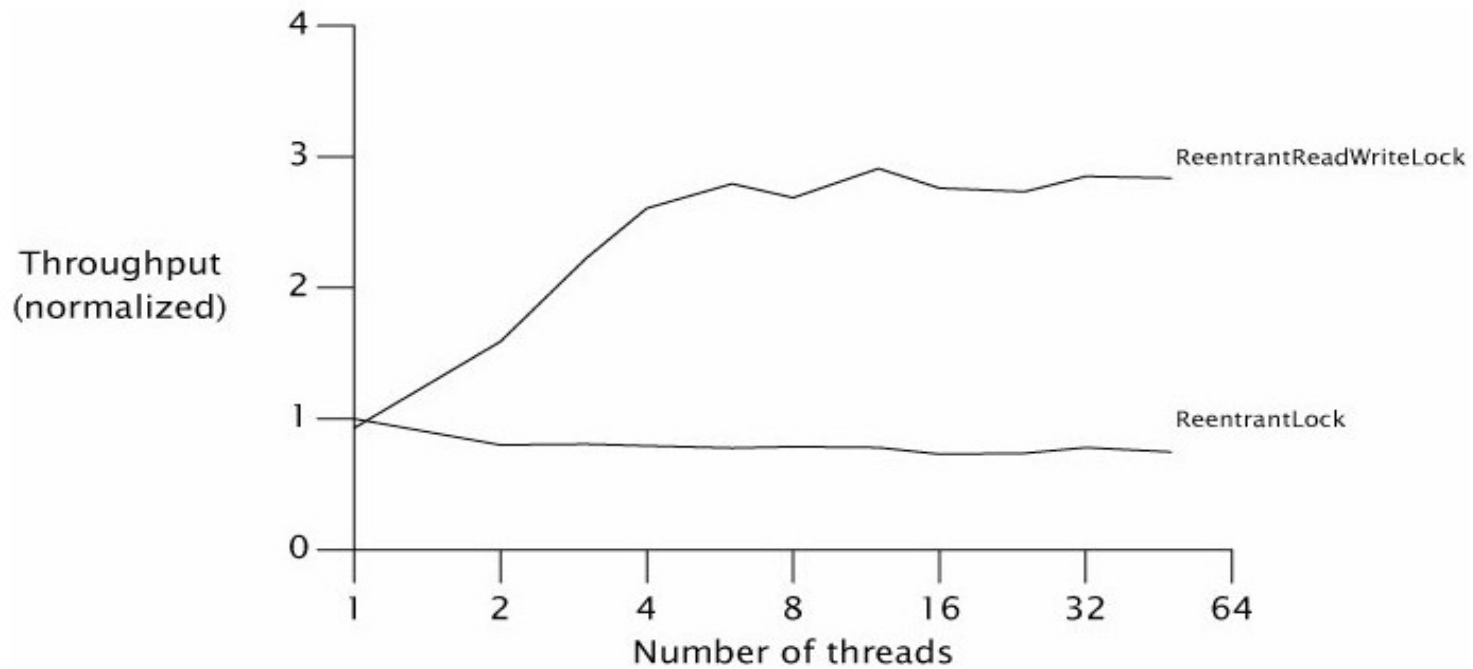
编程实践：使用更优的锁

编程实践：使用更优的锁



ReentrantLock 比较内部锁在 JDK 的性能提升对比

编程实践：使用更优的锁



读一写锁对比互斥锁 ReentrantLock 的性能

编程实践：缩小锁的范围

- 减少线程把持锁的时间
- 避免在临界区进行耗时计算
- 常见做法 1 ： 缩小同步快
- 常见做法 2 ： 把一个同步块分拆成多个
- 需要保证业务逻辑正确为前提

编程实践：避免热点域

- 热点域：每次操作，都需要访问修改的 fields
- eg. ConcurrentHashMap 的 size 计算问题

编程实践：使用不变和 Thread Local 的数据

- 不变数据，即 Immutable 类型数据、在其生命周期中始终保持不变，可以安全地在每个线程中复制一份以便快速读取。
- ThreadLocal 数据，只被线程本身使用，因此不存在不同线程之间的共享数据的问题。

编程实践：使用高并发容器

- J.U.C 的高效地线程安全的并发容器
- Amino 提供更多非阻塞的容器

编程实践：高速缓存计算结果

- 利用空间换时间
- 避免了重复计算相同的数据

编程实践：文档化对外接口的同步策略

- 如果一个类没有明确指明，就不要假设它是线程安全的

编程实践：安全发布

```
@NotThreadSafe
public class UnsafeLazyInitialization {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null)
            resource = new Resource();  // unsafe publication
        return resource;
    }
}
@ThreadSafe
public class SafeLazyInitialization {
    private static Resource resource;

    public  synchronized  static Resource getInstance() {
        if (resource == null)
            resource = new Resource();
        return resource;
    }
}
```

编程实践：安全发布

```
@ThreadSafe
public class EagerInitialization {
    private static Resource resource  = new Resource();

    public static Resource getResource() { return resource; }
}
@ThreadSafe
public class ResourceFactory {
     private static class ResourceHolder {
         public static Resource resource = new Resource();
     }

     public static Resource getResource() {
         return  ResourceHolder.resource ;
     }
}
```

编程实践：安全发布

```
@NotThreadSafe
public class DoubleCheckedLocking {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null) {
            synchronized (DoubleCheckedLocking.class) {
                if (resource == null)
                    resource = new Resource();
            }
        }
        return resource;
    }
}
```

编程实践：利用成熟的框架

- J.U.C
- Amino

http://amino-cbbs.sourceforge.net

(1)、 Data Structures

(论文： http://www.research.ibm.com/people/m/michael/spaa2002.pdf

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.104.8774&rep=rep1&type=pdf)

LockFreeList、 LockFreeSet、 LockFreeOrderedList、 LockFreeDeque、 LockFreeBlockQueue、 ParallelRBTree

(2)、 Parallel Patterns & functions

MasterWorker(类似 ExecutorService)、 WorkStealingScheduler

编程实践：利用成熟的框架

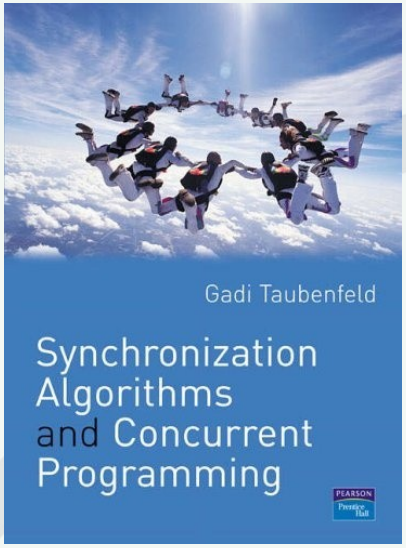
(3)、 Parallel functions

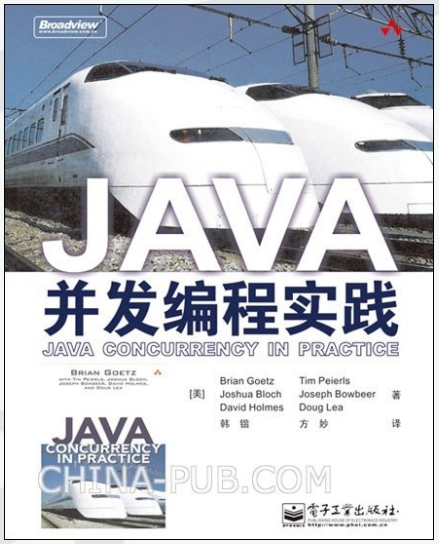
GraphAlg、 ParallelPrefix、 ParallelScanner、 QuickSorter

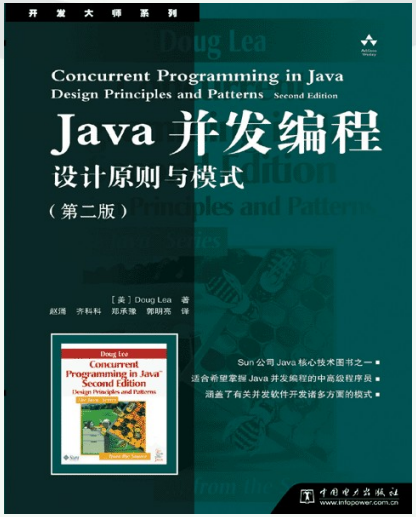
(4)、 Atomics and STM

MultiCAS、 LockFreeBSTree

书籍推荐：


Gadi Taubenfeld
Synchronization Algorithms and Concurrent Programming
PEARSON Prentice Hall


Broadview
JAVA
并发编程实践
JAVA CONCURRENCY IN PRACTICE
BRIAN GOETZ
[美] Brian Goetz Tim Peierls Joshua Bloch Joseph Bowbeer David Holmes Doug Lea 著
韩 锴 方 妙 译
JAVA CONCURRENCY IN PRACTICE
CHINA-PUB.COM
電子工業出版社


开 发 大 师 系 列
Concurrent Programming in Java
Design Principles and Patterns Second Edition
Java 并发编程
设计原则与模式
（第二版）
[美] Doug Lea 著
Sun 公司 Java 核心技术图书之一
适合希望掌握 Java 并发编程的中高级程序员
涵盖了有关并发软件开发诸多方面的模式
www.infopower.com.cn


Design and Verification of Lock-free Parallel Algorithms
Hui Gao
Department of Mathematics and Computing Science
University of Groningen

Q&A

Alibaba Group

HARVARD BUSINESS REVIEW CHINA

Best Practice

寻找行动的巨人

阿里巴巴 Alibaba Group