版本: 2010-08-09

Java并发程序设计教程

温绍锦

邮箱:szujobs@hotmail.com
旺旺:shaojinwensj
QQ: 1420452
Blog:http://www.cnblogs.com/jobs/

旧时王谢堂前燕，飞入寻常百姓家。

Alibaba Group

内容列表

1、使用线程的经验：设置名称、响应中断、使用ThreadLocal
2、Executor ： ExecutorService和Future ☆ ☆ ☆
3、阻塞队列 : put和take、offer和poll、drainTo
4、线程间的协调手段：lock、condition、wait、notify、notifyAll ☆ ☆ ☆
5、Lock-free: atomic、concurrentMap.putIfAbsent、CopyOnWriteArrayList ☆ ☆ ☆
6、关于锁使用的经验介绍
7、并发流程控制手段：CountDownlatch、Barrier
8、定时器: ScheduledExecutorService、大规模定时器TimerWheel
9、并发三大定律：Amdahl、Gustafson、Sun-Ni
10、神人和图书、相关网络资源
11、业界发展情况: GPGPU、OpenCL
12、复习题

学习的过程，着重注意红星标识☆的内容，学完之后，要求能够回答复习题。

Alibaba Group

启动线程的注意事项

```
Thread thread = new Thread("thread name") {
        public void run() {
                // do xxx
        }
};
thread.start();
```


```
public class MyThread extends Thread {
        public MyThread() {
                super("thread name");
        }
        public void run() {
                // do xxx
        }
}
MyThread thread = new MyThread ();
thread.start();
```


```
Thread thread = new Thread() {
        public void run() {
                // do xxx
        }
};
thread.setName("thread name");
thread.start();
```


```
Thread thread = new Thread(task); // 传入任务
thread.setName("thread name");
thread.start();
```


```
Thread thread = new Thread(task, "thread name");
thread.start();
```

无论何种方式，启动一个线程，就要给它一个名字！这对排错诊断系统监控有帮助。否则诊断问题时，无法直观知道某个线程的用途。

Alibaba Group

要响应线程中断

thread.interrupt();

```
Thread thread = new Thread("interrupt test") {
        public void run() {
                for (;;) {
                        doXXX();
                        if (Thread.interrupted()) {
                                break;
                        }
                }
        }
};
thread.start();
```

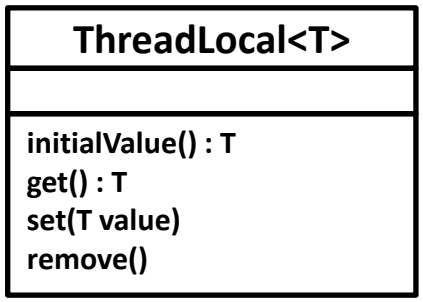

```
public void foo() throws InterruptedException {
        if (Thread.interrupted()) {
                throw new InterruptedException();
        }
}
```


```
Thread thread = new Thread("interrupt test") {
        public void run() {
                for (;;) {
                        try {
                                doXXX();
                        } catch (InterruptedException e) {
                                break;
                        } catch (Exception e) {
                                // handle Exception
                        }
                }
        }
};
thread.start();
```

程序应该对线程中断作出恰当的响应。

Alibaba Group

ThreadLocal



顾名思义它是**local variable**（线程局部变量）。它的功用非常简单，就是为每一个使用该变量的线程都提供一个变量值的副本，是每一个线程都可以独立地改变自己的副本，而不会和其它线程的副本冲突。从线程的角度看，就好像每一个线程都完全拥有该变量。

使用场景

To keep state with a thread (user-id, transaction-id, logging-id)

To cache objects which you need frequently

隐式传参

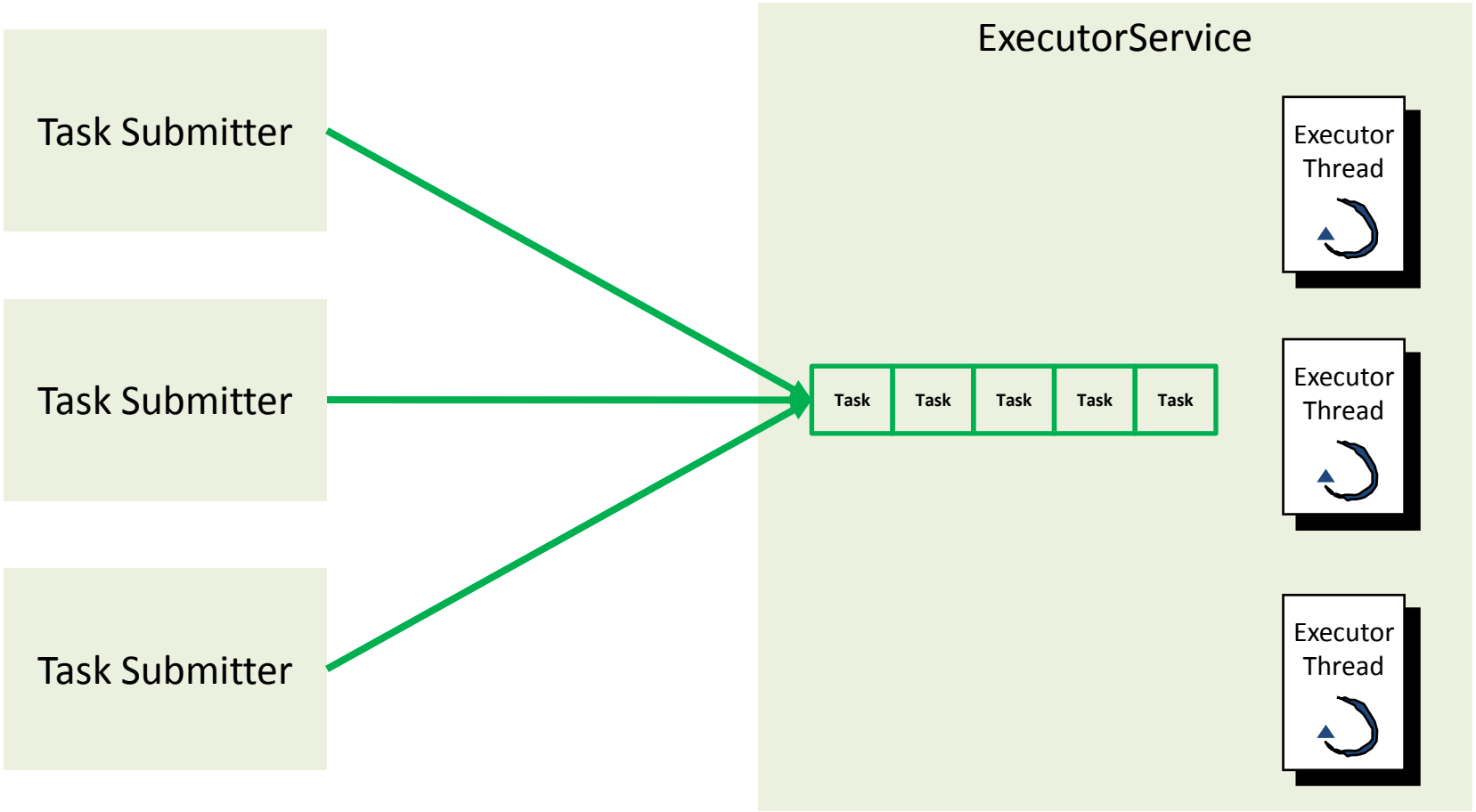
注意：使用ThreadLocal，一般都是声明在静态变量中，如果不断的创建ThreadLocal而且没有调用其remove方法，将会导致内存泄露。

同时请注意，如果是static的ThreadLocal，一般不需要调用remove。

Alibaba Group

任务的提交者和执行者

为了方便并发执行任务，出现了一种专门用来执行任务的实现，也就是Executor。
由此，任务提交者不需要再创建管理线程，使用更方便，也减少了开销。



java.util.concurrent.Executors是Executor的工厂类，通过Executors可以创建你所需要的Executor。

任务的提交者和执行者之间的通讯手段

```
ExecutorService executor = Executors.newSingleThreadExecutor();

Callable<Object> task = new Callable<Object>() {
        public Object call() throws Exception {
                Object result = "...";
                return result;
        }
};

Future<Object> future = executor.submit(task);
future.get(); // 等待至完成
```

Future<T>
cancel(boolean) : boolean isCancelled() : boolean isDone() : boolean get() : T get(long, TimeUnit) : T

有两种任务：

Runnable

Callable

Callable是需要返回值的任务

Task Submitter

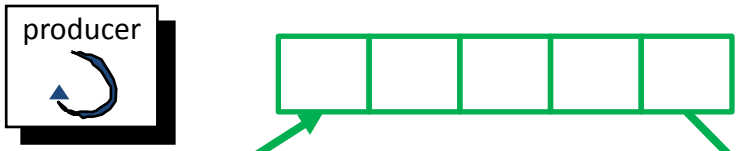
```
Future<Object> future = executor.submit(task);

// 等待到任务被执行完毕返回结果
// 如果任务执行出错，这里会抛ExecutionException
future.get();

//等待3秒，超时后会抛TimeoutException
future.get(3, TimeUnit.SECONDS);
```

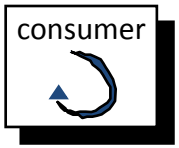
Task Executor

```
Callable<Object> task = new Callable<Object>() {
        public Object call() throws Exception {
                Object result = ...;
                return result;
        }
};
```

Task Submitter把任务提交给Executor执行，他们之间需要一种通讯手段，这种手段的具体实现，通常叫做Future。Future通常包括get（阻塞至任务完成），cancel，get(timeout)（等待一段时间）等等。Future也用于异步变同步的场景。

阻塞队列






```
// 如果队列满则阻塞
blockingQ.put(object);
```

```
for (;;) {
        blockingQ.take(); // 如果队列空则阻塞
}
```

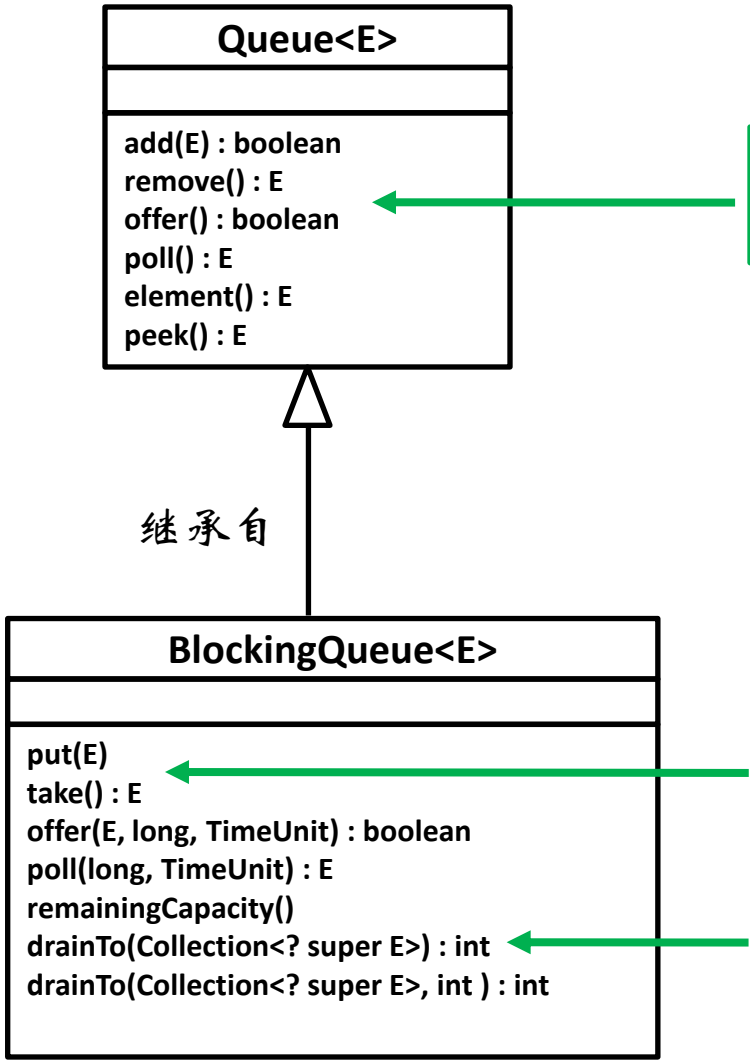
阻塞队列，是一种常用的并发数据结构，常用于生产者-消费者模式。

在Java中，有很多种阻塞队列：

- **ArrayBlockingQueue** ← 最常用
- **LinkedBlockingQueue** ← 不会满的
- **SynchronousQueue** ← size为0
- **PriorityBlockingQueue**
- **CompletionService (BlockingQueue + Executor)**
- **TransferQueue (JDK 7中更快的SynchronousQueue)**

Alibaba Group

使用阻塞队列



使用BlockingQueue的时候，尽量不要使用从Queue继承下来的方法，否则就失去了Blocking的特性了。

在BlockingQueue中，要使用put和take，而非offer和poll。如果要使用offer和poll，也是要使用带等待时间参数的offer和poll。

使用drainTo批量获得其中的内容，能够减少锁的次数。

Alibaba Group

使用阻塞队列

```
final BlockingQueue<Object> blockingQ = new ArrayBlockingQueue<Object>(10);
Thread thread = new Thread("consumer thread") {
        public void run() {
                for (;;) {
                        Object object = blockingQ.poll(); // 杯具，不等待就会直接返回
                        handle(object);
                }
        }
};
```

```
final BlockingQueue<Object> blockingQ = new ArrayBlockingQueue<Object>(10);
Thread thread = new Thread("consumer thread") {
        public void run() {
                for (;;) {
                        try {
                                Object object = blockingQ.take(); // 等到有数据才继续
                                handle(object);
                        } catch (InterruptedException e) {
                                break;
                        } catch (Exception e) {
                                // handle exception
                        }
                }
        }
};
```

使用阻塞队列

```
final BlockingQueue<Object> blockingQ = new ArrayBlockingQueue<Object>(10);
Thread thread = new Thread("consumer thread") {
        public void run() {
                for (;;) {
                        try {
                                Object object = blockingQ.poll(1, TimeUnit.SECONDS); //防止死等
                                if (object == null) {
                                        continue;  // 或者做其他处理
                                }
                        } catch (InterruptedException e) {
                                break;
                        } catch (Exception e) {
                                // handle exception
                        }
                }
        }
};
```

Alibaba Group

实现一个简单的阻塞队列(1)

通过实现简单的阻塞队列来学习并发知识

```
class BlockingQ {
        private Object notEmpty = new Object();
        private Queue<Object> linkedList = new LinkedList<Object>();

        public Object take() throws InterruptedException {
                synchronized (notEmpty) {
                        if (linkedList.size() == 0) {
                                notEmpty.wait();
                        }
                        return linkedList.poll();
                }
        }

        public void offer(Object object) {
                synchronized (notEmpty) {
                        if (linkedList.size() == 0) {
                                notEmpty.notifyAll();
                        }
                        linkedList.add(object);
                }
        }
}
```

要执行wait操作，必须先取得该对象的锁。

执行wait操作之后，锁会释放。

被唤醒之前，需要先获得锁。

要执行notify和notifyAll操作，都必须先取得该对象的锁。

未取得锁就直接执行wait、notfiy、notifyAll会抛异常

Alibaba Group

实现一个简单的阻塞队列(2)

通过实现简单的阻塞队列来学习并发知识

```
class BlockingQ {
        private Object notEmpty = new Object();
        private Object notFull = new Object();
        private Queue<Object> linkedList = new LinkedList<Object>();
        private int maxLength = 10;

        public Object take() throws InterruptedException {
                synchronized (notEmpty) {
                        if (linkedList.size() == 0) {
                                notEmpty.wait();
                        }
                        synchronized (notFull) {
                                if (linkedList.size() == maxLength) {
                                        notFull.notifyAll();
                                }
                                return linkedList.poll();
                        }
                }
        }
        public void offer(Object object) throws InterruptedException {
                synchronized (notEmpty) {
                        if (linkedList.size() == 0) {
                                notEmpty.notifyAll();
                        }
                        synchronized (notFull) {
                                if (linkedList.size() == maxLength) {
                                        notFull.wait();
                                }
                                linkedList.add(object);
                        }
                }
        }
}
```

分别需要对notEmpty和notFull加锁

分别需要对notEmpty和notFull加锁

Alibaba Group

实现一个简单的阻塞队列(3)

通过实现简单的阻塞队列来学习并发知识

```
class BlockingQ {
        private Lock lock = new ReentrantLock();
        private Condition notEmpty = lock.newCondition();
        private Condition notFull = lock.newCondition();
        private Queue<Object> linkedList = new LinkedList<Object>();
        private int maxLength = 10;
        public Object take() throws InterruptedException {
                lock.lock();
                try {
                        if (linkedList.size() == 0) {
                                notEmpty.await();
                        }
                        if (linkedList.size() == maxLength) {
                                notFull.signalAll();
                        }
                        return linkedList.poll();
                } finally {
                        lock.unlock();
                }
        }
        public void offer(Object object) throws InterruptedException {
                lock.lock();
                try {
                        if (linkedList.size() == 0) {
                                notEmpty.signalAll();
                        }
                        if (linkedList.size() == maxLength) {
                                notFull.await();
                        }
                        linkedList.add(object);
                } finally {
                        lock.unlock();
                }
        }
}
```

一个锁可以创建多个Condition

要执行await操作，必须先取得该Condition的锁。

执行await操作之后，锁会释放。

被唤醒之前，需要先获得锁。

要执行signal和signalAll操作，都必须先取得该对象的锁。

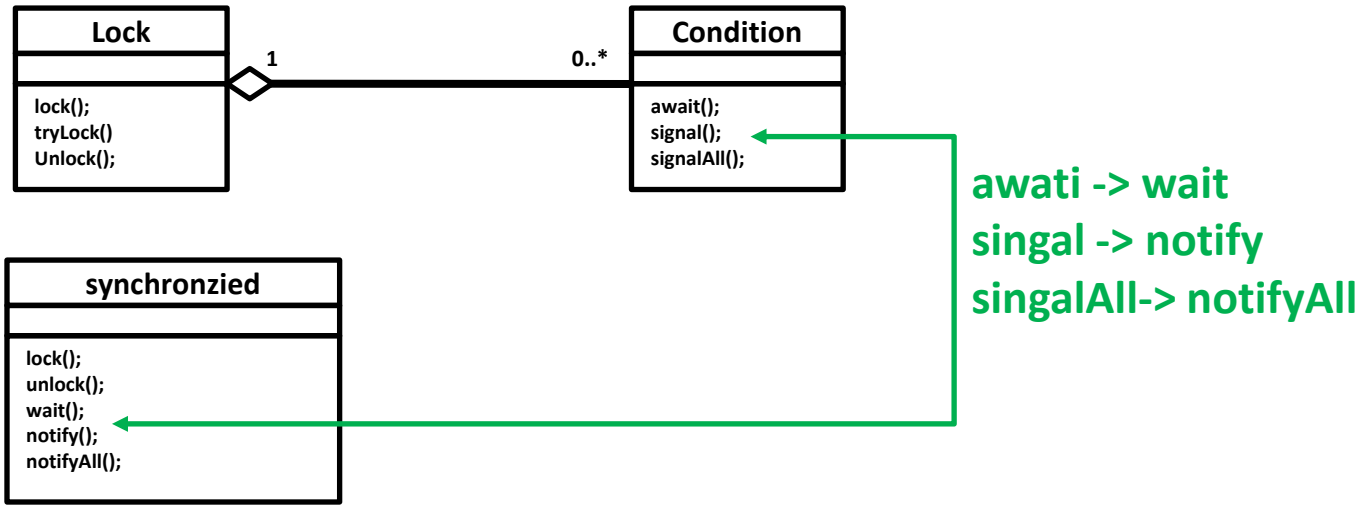
注意：未锁就直接执行await、signal、siganlAll会抛异常

ReentrantLock和Synchronized

Synchronized是Lock的一种简化实现，一个Lock可以对应多个Condition，而synchronized把Lock和Condition合并了，一个synchronized Lock只对应一个Condition，可以说Synchronized是Lock的简化版本。

在JDK 5，Synchronized要比Lock慢很多，但是在JDK 6中，它们的效率差不多。

不要在Lock和Condition上使用wait、notiffy、notifyAll方法！

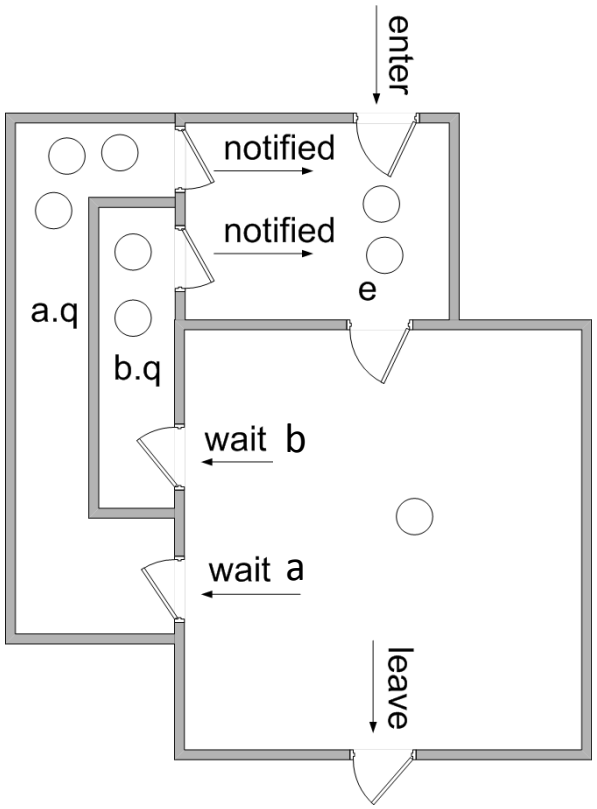


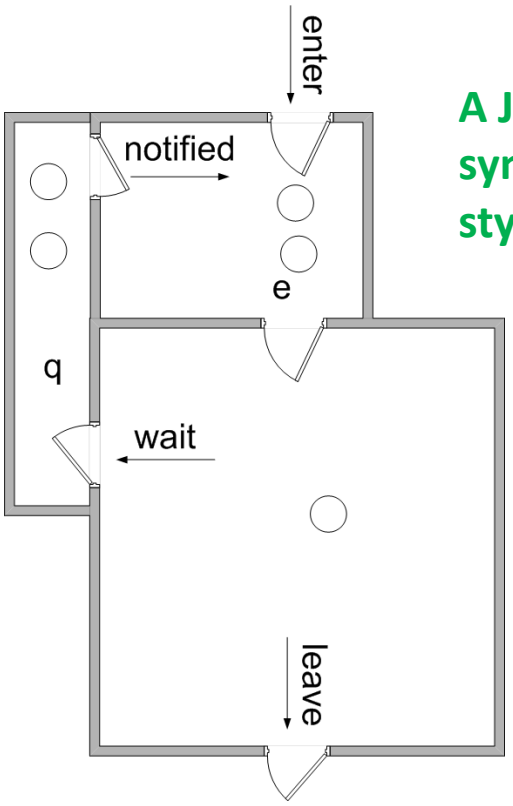
awati -> wait
singal -> notify
singalAll-> notifyAll

Alibaba Group

Monitor的理论模型

A Mesa style monitor with two condition variables a and b





A Java synchronized style monitor

http://en.wikipedia.org/wiki/Monitor_(synchronization)

In concurrent programming, a monitor is an object intended to be used safely by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion. That is, at each point in time, at most one thread may be executing any of its methods. This mutual exclusion greatly simplifies reasoning about the implementation of monitors compared with code that may be executed in parallel.

Monitors also provide a mechanism for threads to temporarily give up exclusive access, in order to wait for some condition to be met, before regaining exclusive access and resuming their task. Monitors also have a mechanism for signaling other threads that such conditions have been met.

Monitors were invented by **C.A.R. Hoare** [1] and **Per Brinch Hansen**, [2] and were first implemented in Brinch Hansen's Concurrent Pascal language.

使用AtomicInteger

```
class Counter {
        private volatile int count = 0;

        public synchronized void increment() {
                count++;
        }

        public int getCount() {
                return count;
        }
}
```

若要线程安全执行执行count++，需要加锁

1

```
class Counter {
        private AtomicInteger count = new AtomicInteger();

        public void increment() {
                count.incrementAndGet();
        }

        public int getCount() {
                return count.get();
        }
}
```

使用AtomicInteger之后，不需要加锁，也可以实现线程安全。

2

这是由硬件提供原子操作指令实现的。在非激烈竞争的情况下，开销更小，速度更快。java.util.concurrent中实现的原子操作类包括：

AtomicBoolean、AtomicInteger、AtomicLong、AtomicReference

使用Lock-Free算法

```
class Counter {
        private volatile int max = 0;
        public synchronized void set(int value) {
                if (value > max) {
                        max = value;
                }
        }
        public int getMax() {
                return max;
        }
}
```

若要线程安全，需要加锁

1

```
class Counter {
        private AtomicInteger max = new AtomicInteger();

        public void set(int value) {
                for (;;) {
                        int current = max.get();
                        if (value > current) {
                                if (max.compareAndSet(current, value)) {
                                        break;
                                } else {
                                        continue;
                                }
                        } else { break; }
                }
        }
        public int getMax() {
                return max.get();
        }
}
```

循环

CAS

回退

LockFree算法，不需要加锁。

通常都是三个部分组成：

① 循环

② CAS (CompareAndSet)

③ 回退

2

Alibaba Group

使用Lock-Free算法

```
class Counter {
        private AtomicInteger max = new AtomicInteger();

        public void set(int value) {
                int current;
                do {
                        current = max.get();
                        if (value <= current) {
                                break;
                        }
                } while (!max.compareAndSet(current, value));
        }

        public int getMax() {
                return max.get();
        }
}
```

方案3和方案2的实现是一样的，只不过把for(;;)换成了do…while。

非阻塞型同步(Non-blocking Synchronization)

如何正确有效的保护共享数据是编写并行程序必须面临的一个难题，通常的手段就是同步。同步可分为阻塞型同步（Blocking Synchronization）和非阻塞型同步（ Non-blocking Synchronization）。

阻塞型同步是指当一个线程到达临界区时，因另外一个线程已经持有访问该共享数据的锁，从而不能获取锁资源而阻塞，直到另外一个线程释放锁。常见的同步原语有 mutex、semaphore 等。如果同步方案采用不当，就会造成死锁（deadlock），活锁（livelock）和优先级反转（priority inversion），以及效率低下等现象。

为了降低风险程度和提高程序运行效率，业界提出了不采用锁的同步方案，依照这种设计思路设计的算法称为非阻塞型算法，其本质特征就是停止一个线程的执行不会阻碍系统中其他执行实体的运行。

当今比较流行的 Non-blocking Synchronization 实现方案有三种：

Wait-free

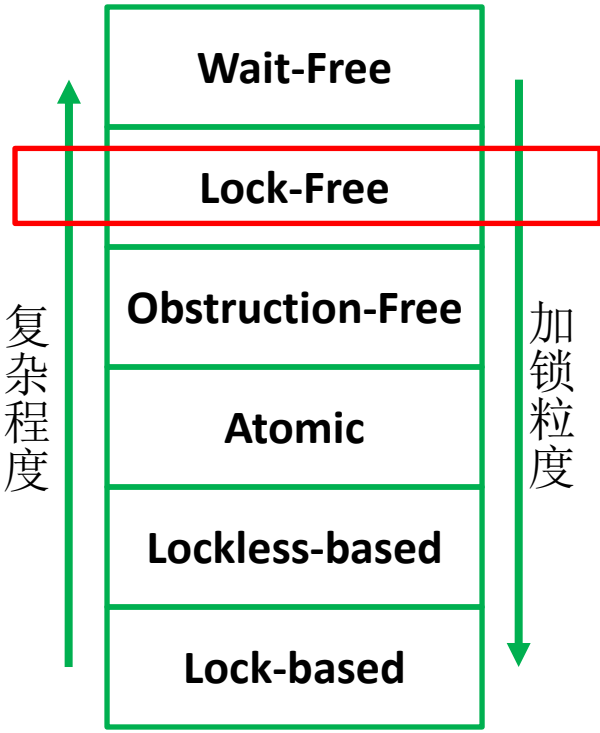
Wait-free 是指任意线程的任何操作都可以在有限步之内结束，而不用关心其它线程的执行速度。 Wait-free 是基于 per-thread 的，可以认为是 starvation-free 的。非常遗憾的是实际情况并非如此，采用 Wait-free 的程序并不能保证 starvation-free，同时内存消耗也随线程数量而线性增长。目前只有极少数的非阻塞算法实现了这一点。

Lock-free

Lock-Free 是指能够确保执行它的所有线程中至少有一个能够继续往下执行。由于每个线程不是 starvation-free 的，即有些线程可能会被任意地延迟，然而在每一步都至少有一个线程能够往下执行，因此系统作为一个整体是在持续执行的，可以认为是 system-wide 的。所有 Wait-free 的算法都是 Lock-Free 的。

Obstruction-free

Obstruction-free 是指在任何时间点，一个孤立运行线程的每一个操作可以在有限步之内结束。只要没有竞争，线程就可以持续运行。一旦共享数据被修改，Obstruction-free 要求中止已经完成的部分操作，并进行回滚。 所有 Lock-Free 的算法都是 Obstruction-free 的。



进一步使用Lock-Free数据结构

```
class BeanManager {
        private Map<String, Object> map = new HashMap<String, Object>();

        public Object getBean(String key) {
                synchronized (map) {
                        Object bean = map.get(key);
                        if (bean == null) {
                                map.put(key, createBean());
                                bean = map.get(key);
                        }
                        return bean;
                }
        }
}
```

```
class BeanManager {
        private ConcurrentMap<String, Object> map = new ConcurrentHashMap<String, Object>();

        public Object getBean(String key) {
                Object bean = map.get(key);
                if (bean == null) {
                        map.putIfAbsent(key, createBean());
                        bean = map.get(key);
                }
                return bean;
        }
}
```

使用ConcurrentMap，避免直接使用锁，锁由数据结构来管理。

ConcurrentHashMap并没有实现Lock-Free，只是使用了分离锁的办法使得能够支持多个Writer并发。 ConcurrentHashMap需要使用更多的内存。

同样的思路用于更新数据库-乐观锁

```
public class SequenceDao extends SqlMapClientDaoSupport {
        public boolean compareAndSet(String name, int value, int expect) {
                Map<String, Object> parameters = new HashMap<String, Object>();
                parameters.put("name", name);
                parameters.put("value", value);
                parameters.put("expect", expect);

                // UPDATE t_sequence SET value = #value# WHERE name = #name# AND value = #expect#
                int updateCount = getSqlMapClientTemplate().update("Sequence.compareAndSet", parameters);

                return updateCount == 1;
        }

}
```

通过UpdateCount来实现CompareAndSet

```
public class SequenceService {
        @Transactional(propagation = Propagation.NOT_SUPPORTED)
        public synchronized void increment(String sequenceName) {
                for (;;) {
                        int value = dao.getValue(sequenceName);
                        if (dao.compareAndSet(sequenceName, value + 1, value)) {
                                break;
                        }
                }
        }

}
```

三个部分：
① 循环
② CAS (CompareAndSet)
③ 回退

注意，乐观锁时必须使用：`@Transactional(propagation = Propagation.NOT_SUPPORTED)`

对比，使用悲观锁版本

```
public class SequenceDao extends SqlMapClientDaoSupport {
        public int getValueForUpdate(String name) {
                // SELECT value FROM t_sequence WHERE name = #name# FOR UPDATE
                return (Integer) getSqlMapClientTemplate().queryForObject("Sequence.getValueForUpdate", name);
        }

        public void set(String name, int value) {
                Map<String, Object> parameters = new HashMap<String, Object>();
                parameters.put("name", name);
                parameters.put("value", value);

                // UPDATE t_sequence SET value = #value# WHERE name = #name#
                getSqlMapClientTemplate().update("Sequence.set", parameters);
        }
}
```

```
public class SequenceService {
        @Transactional(propagation = Propagation.REQUIRED)
        public synchronized void increment2(String sequenceName) {
                int value = dao.getValueForUpdate(sequenceName);
                dao.set(sequenceName, value + 1);
        }
}
```

读取时，就开始加锁。

Lock-Free算法，可以说是乐观锁，如果非激烈竞争的时候，不需要使用锁，从而开销更小，速度更快。

使用CopyOnWriteArrayList

```
class Engine {
        private List<Listener> listeners = new ArrayList<Listener>();

        public boolean addListener(Listener listener) {
                synchronized (listeners) {
                        return listeners.add(listener);
                }
        }

        public void doXXX() {
                synchronized (listeners) {
                        for (Listener listener : listeners) {
                                listener.handle();
                        }
                }
        }
}
```

1

```
class Engine {
        private List<Listener> listeners = new CopyOnWriteArrayList <Listener>();

        public boolean addListener(Listener listener) {
                return listeners.add(listener);
        }

        public void doXXX() {
                for (Listener listener : listeners) {
                        listener.handle();
                }
        }
}
```

2

COW是一种很古老的技术，类似的并发数据结构有：

- ConcurrentSkipListMap
- ConcurrentSkipListSet
- CopyOnWriteArrayList
- CopyOnWriteArraySet

适当使用CopyOnWriteArrayList，能够提高读操作时的效率。

锁的使用

① 使用支持CAS的数据结构，避免使用锁，如：
AtomicXXX、ConcurrentMap、CopyOnWriteList、ConcurrentLinkedQueue

② 一定要使用锁的时候，注意获得锁的顺序，相反顺序获得锁，就容易产生死锁。

③ 死锁经常是无法完全避免的，鸵鸟策略被很多基础框架所采用。

④ 通过Dump线程的StackTrace，例如linux下执行命令 kill -3 <pid>，或者jstack –l <pid>，或者使用Jconsole连接上去查看线程的StackTrace，由此来诊断死锁问题。

⑤ 外部锁常被忽视而导致死锁，例如数据库的锁

⑥ 存在检测死锁的办法

⑦ 存在一些预防死锁的手段，比如Lock的tryLock，JDK 7中引入的Phaser等。

并发流程控制-使用CoutDownLatch

```
final int COUNT = 10;
final CountDownLatch completeLatch = new CountDownLatch(COUNT);

for (int i = 0; i < COUNT; ++i) {
        Thread thread = new Thread("worker thread " + i) {
                public void run() {
                        // do xxxx
                        completeLatch.countDown();
                }
        };
        thread.start();
}

completeLatch.await();
```

当你启动了一个线程，你需要等它执行结束，此时，CountDownLatch也许是一个很好的选择。

```
final CountDownLatch startLatch = new CountDownLatch(1);

for (int i = 0; i < 10; ++i) {
        Thread thread = new Thread("worker thread " + i) {
                public void run() {
                        try {
                                startLatch.await();
                        } catch (InterruptedException e) {
                                return;
                        }
                        // do xxxx
                }
        };
        thread.start();
}
// do xxx
startLatch.countDown();
```

当你启动很多线程，你需要这些线程等到通知后才真正开始，CountDownLatch也许是一个很好的选择。

Alibaba Group

为什么要使用CountDownLatch

```
final CountDownLatch latch = new CountDownLatch(1);
Thread thread = new Thread(“worker-thread") {
        public void run() {
                // do xxx
                latch.countDown();
        }
};
thread.start();
latch.await();
```

1

```
final Object completeSignal = new Object();
Thread thread = new Thread("worker-thread") {
        public void run() {
                // do xxx
                synchronized (completeSignal) {
                        completeSignal.notifyAll();
                }
        }
};

synchronized (completeSignal) {
        thread.start();
        completeSignal.wait();
}
```

2

```
final Lock lock = new ReentrantLock();
final Condition completeSignal = lock.newCondition();
Thread thread = new Thread("worker-thread") {
        public void run() {
                // do xxx
                lock.lock();
                try {
                        completeSignal.signalAll();
                } finally {
                        lock.unlock();
                }
        }
};

lock.lock();
try {
        thread.start();
        completeSignal.await();
} finally {
        lock.unlock();
}
```

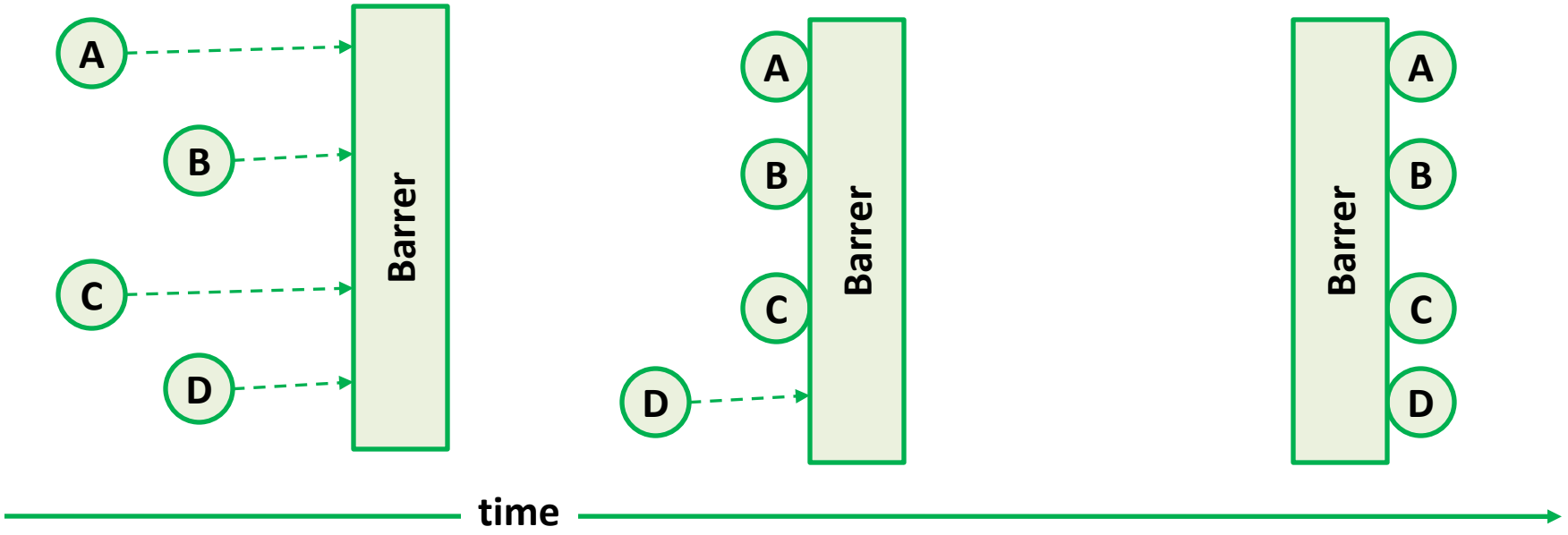
3

以上三种实现功能是一样的

方案1使用CountDownLatch最简单

Alibaba Group

Barrier



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.

并发流程控制-使用CycliBarrier

```
class PerformaceTest {
        private int threadCount;
        private CyclicBarrier barrier;
        private int loopCount = 10;

        public PerformaceTest(int threadCount) {
                this.threadCount = threadCount;
                barrier = new CyclicBarrier(threadCount, new Runnable() {
                        public void run() {
                                collectTestResult();
                        }
                });
                for (int i = 0; i < threadCount; ++i) {
                        Thread thread = new Thread("test-thread " + i) {
                                public void run() {
                                        for (int j = 0; j < loopCount; ++j) {
                                                doTest();
                                                try {
                                                        barrier.await();
                                                } catch (InterruptedException e) {
                                                        return;
                                                } catch (BrokenBarrierException e) {
                                                        return;
                                                }
                                        }
                                }
                        };
                        thread.start();
                }
        }
        private void doTest() { /* do xxx */ }
        private void collectTestResult() { /* do xxx */ }
}
```

使用Barrier来实现并发性能测试的聚合点。

JDK 7是中包括一个类似的流程控制手段Phaser

Alibaba Group

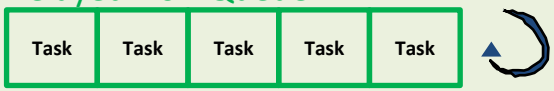
使用定时器

ScheduledExecutorService
schedule(Runnable command, long delay, TimeUnit unit) : ScheduledFuture schedule(Callable<V> callable, long delay, TimeUnit unit) : ScheduledFuture scheduleAtFixedRate(Runnable comand, long initDelay, long period, TimeUnit unit) : ScheduledFuture scheduleWithFixedDelay(Runnable command, long initDelay, long delay, TimeUnit unit) : ScheduledFuture

ScheduledTask Submitter

```
ScheduleFuture<Object> future = scheduler.schedule(task, 1, TimeUnit.SECONDS);

// 等待到任务被执行完毕返回结果
// 如果任务执行出错，这里会抛ExecutionException
future.get();

//取消调度任务
future.cancel();
```

ScheduledExecutorService

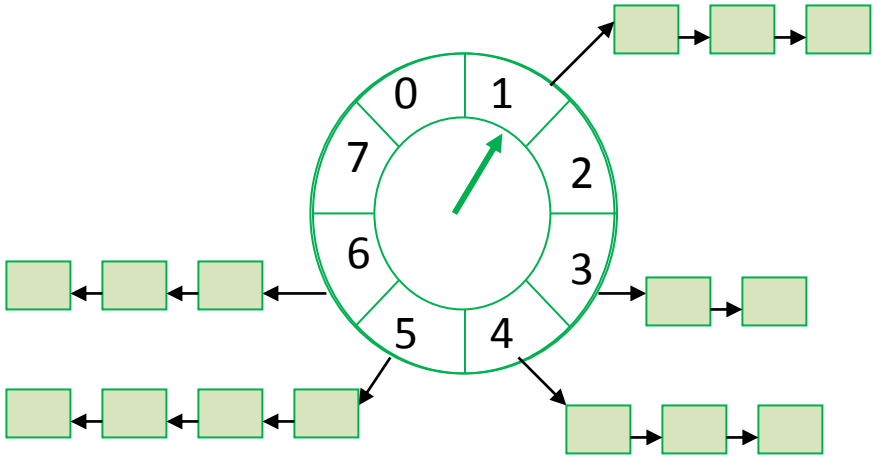


java.util.concurrent.Executors是ScheduledExecutorService的工厂类，通过Executors，你可以创建你所需要的ScheduledExecutorService 。

JDK 1.5之后有了ScheduledExecutorService，不建议你再使用java.util.Timer，因为它无论功能性能都不如ScheduledExecutorService。

大规模定时器TimerWheel

存在一种算法 **TimerWheel**，适用于大规模的定时器实现。这个 算法最早是被设计用来实现 **BSD** 内核中定时器的，后来被广泛 移植到诸如 **ACE** 等框架中，堪称 **BSD** 中经典算法之一，能针 对定时器的各类常见操作提供接近常数时间的响应，且能根据需 要很容易进行扩展。



在网络通讯领域，常见有TimerWheel的实现，包括ACE、Microsoft Unified Comunication Managed API 等，因为大规模定时器目前只能使用TimerWheel。

Alibaba Group

JDK 7 任务分解工具Fork/Join

硬件的发展趋势非常清晰；Moore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边界（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——在这种环境下短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，否则将面临即便有许多工作需要去做而处理器却仍处于空闲的风险。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包含一种框架，用于表示某种更细粒度级别的并行算法：fork-join框架。**

Fork-join融合了分而治之技术；获取问题后，递归地将它分成多个子问题，直到每个子问题都足够小，以至于可以高效地串行地解决它们。递归的过程将会把问题分成两个或者多个子问题，然后把这些问题放入队列中等待处理（fork步骤），接下来等待所有子问题的结果（join步骤），把多个结果合并到一起。

任务分解	不同的行为分配给不同的线程
数据分解	多个线程对不同的数据集执行同样的操作
数据流分解	一个线程的输出是第二个线程的输入

Fork/Join使用示例

```
public static void main(String[] args) throws Exception {
        final ForkJoinPool mainPool = new ForkJoinPool();

        int len = 1000 * 1000 * 10;
        int[] array = new int[len];

        mainPool.invoke(new SortTask(array, 0, len - 1));
}

public static class SortTask extends RecursiveAction {
        private int[] array;
        private int fromIndex;
        private int toIndex;
        private final int chunksize = 1024;

        public SortTask(int[] array, int fromIndex, int toIndex) {
                this.array = array;
                this.fromIndex = fromIndex;
                this.toIndex = toIndex;
        }

        @Override
        protected void compute() {
                int size = toIndex - fromIndex + 1;
                if (size < chunksize) {
                        Arrays.sort(array, fromIndex, toIndex);
                } else {
                        int leftSize = size / 2;
                        SortTask leftTask = new SortTask(array, fromIndex, fromIndex + leftSize);
                        SortTask rightTask = new SortTask(array, fromIndex + leftSize + 1, toIndex);
                        this.invokeAll(leftTask, rightTask);
                }
        }
}
```

任务分解

并发三大定律

- **Amdahl 定律**
  - Gene Amdahl 发现在计算机体系架构设计过程中，某个部件的优化对整个架构的优化和改善是有上限的。这个发现后来成为知名的 Amdahl 定律。

即使你有10个老婆，也不能一个月把孩子生下来。

- **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。

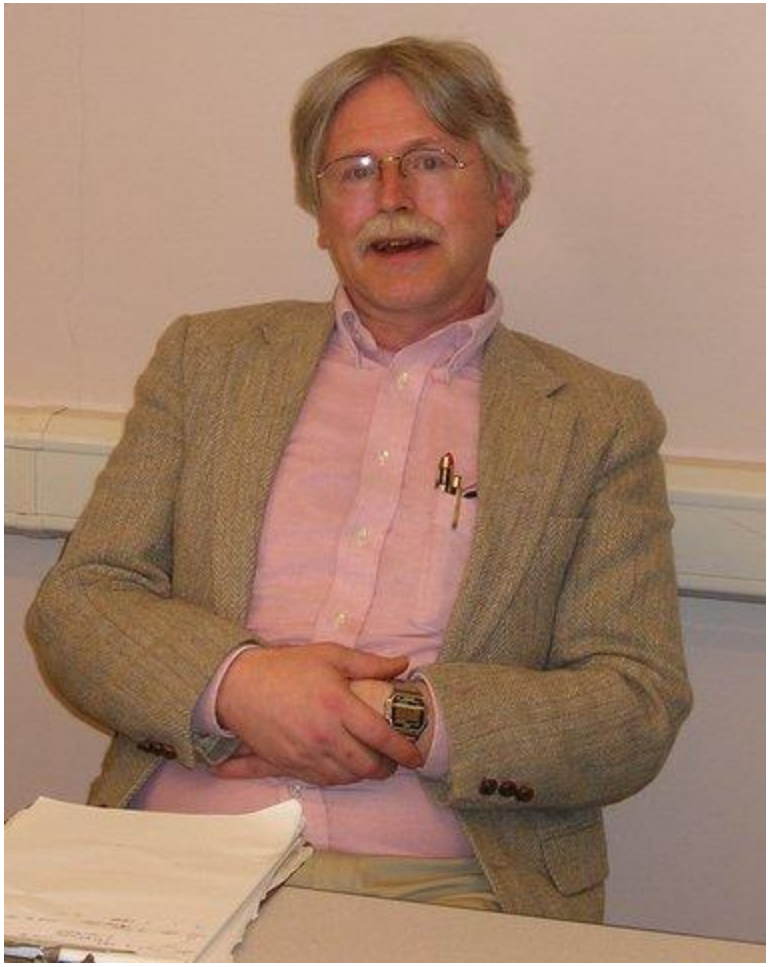
当你有10个老婆，就会要生更多的孩子。

- **Sun-Ni 定律**
  - 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。

你要设法让每个老婆都在干活，别让她们闲着。

Alibaba Group

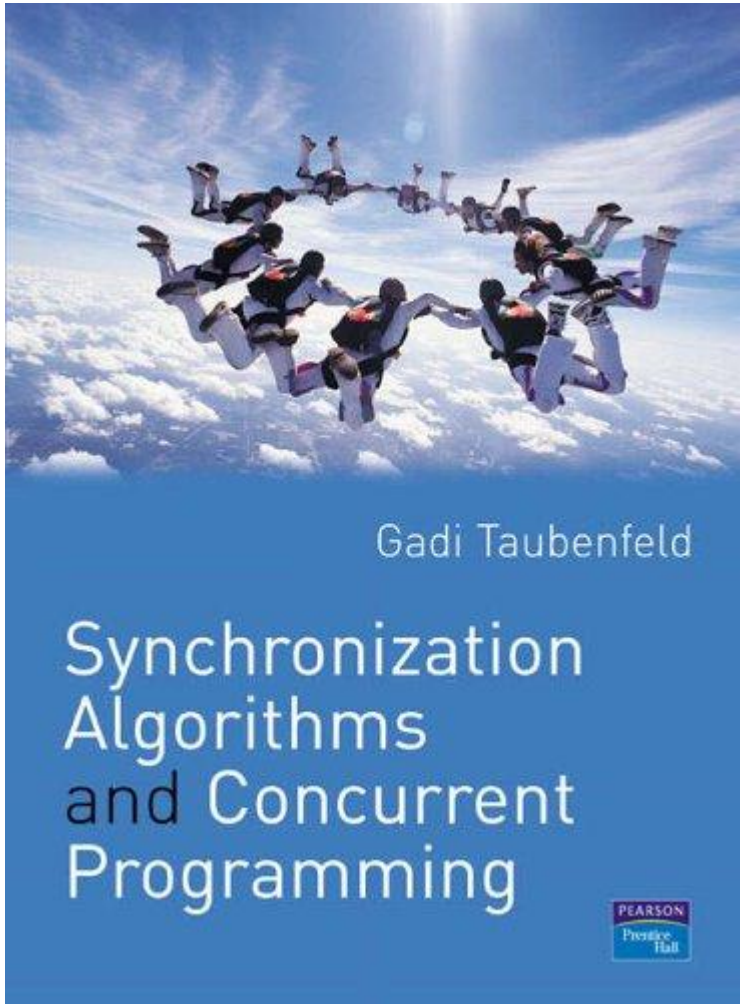
拜神



Doug Lea - 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规范的制定者。图书著作《Concurrent Programming in Java: Design Principles and Patterns》。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

推荐图书


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


Gadi Taubenfeld
Synchronization Algorithms and Concurrent Programming
PEARSON
Prentice Hall

并发相关网络资源

维基百科并发控制专题
http://en.wikipedia.org/wiki/Category:Concurrency_control

维基百科并行计算专题
http://en.wikipedia.org/wiki/Parallel_computing

维基百科非阻塞同步专题
http://en.wikipedia.org/wiki/Non-blocking_synchronization

Herb Sutter的个人主页
http://www.gotw.ca

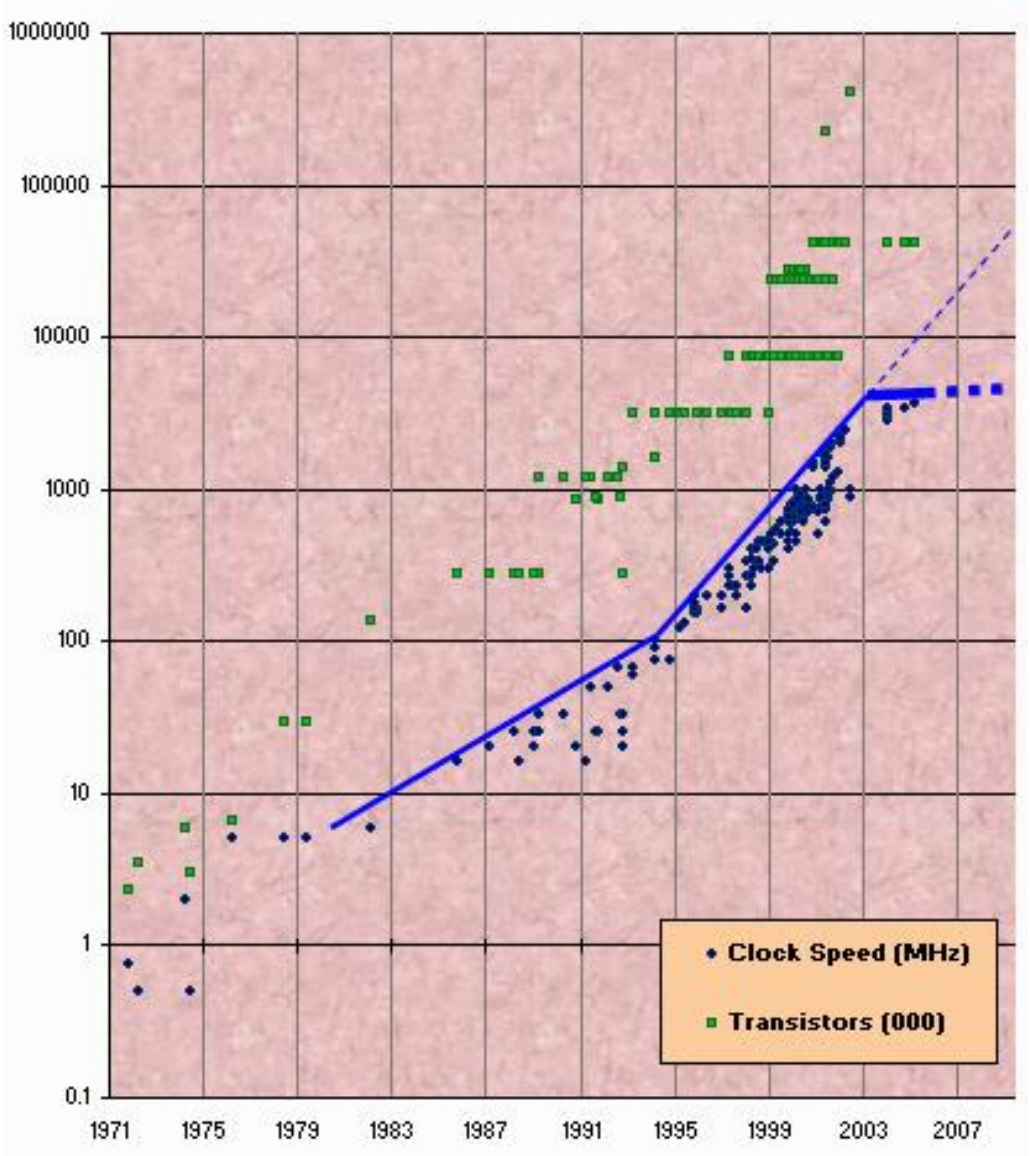
Doug Lea的个人主页
http://g.oswego.edu/

非阻塞同步算法论文
http://www.cs.wisc.edu/trans-memory/biblio/swnbs.html

ACE关于并发和网络的指南
http://www.cs.wustl.edu/~schmidt/tutorials-patterns.html

透过 Linux 内核看无锁编程
http://www.ibm.com/developerworks/cn/linux/l-cn-lockfree/

http://www.khronos.org/opencl/ OpenCL官方网站



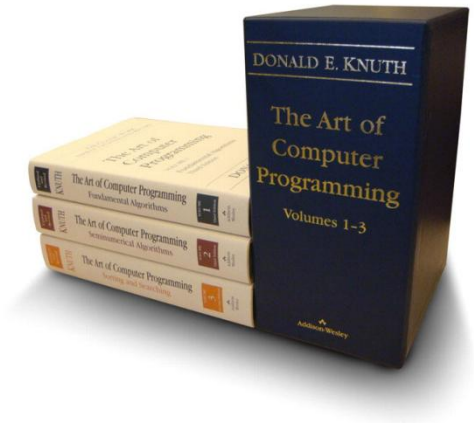
我们今天没有10GHz芯片！

在IDF05（Intel Developer Forum 2005）上，Intel首席执行官Craig Barrett就取消4GHz芯片计划一事，半开玩笑当众单膝下跪致歉。

Donald Knuth
世界顶级计算机科学家

在我看来，这种现象(并发)或多或少是由于硬件设计者已经无计可施了导致的，他们将Moore定律失效的责任推脱给软件开发者。

Donald Knuth 2008年7月接受Andrew Binstock访谈



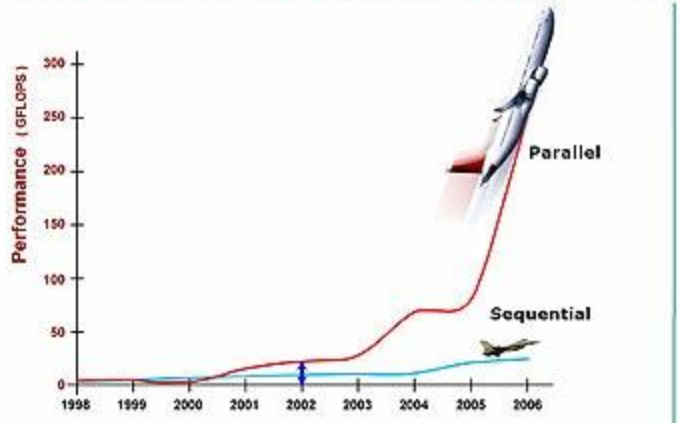
A Brief History of Time

	Introduced	Adopted in mainstream
GUIs	1973 (Xerox Alto)	~1984-89 (Mac) ~1990-95 (Win3.x)
Objects	1967 (Simular)	~1993-98 (C++, Java)
Garbage Collection	1958 (Lisp)	~1995-2000 (Java)
Generic Types	1967 (Strachey)	~198x (US DoD, Ada) ~1995-2000 (C++)
Internet	1967+ (ARPAnet)	~1995-2000
Concurrency	1964 (CDC 6600)	~2007-2012

2010年最快的CPU

主要参数	POWER 7	皓龙6100	至强7500
首次发布	2010-2-8	2010-3-29	2010-3-31
生产工艺	45nm	45nm	45nm
晶体管数量	12亿	19亿	23亿
晶体面积	567mm²	692mm²	684mm²
最多核心数量	8	12	8
每核心最大线程数量	4	1	2
CPU最大核心/线程数量	8/32	12/12	8/16
系统最高支持插槽数量	32	4	8
系统最大核心/线程数量	256核/1024线程	48核/48线程	64核/128线程
最高主频	3.8G（8核）	2.3G（12核）	2.26G（8核）
最高主频时CPU浮点性能	243.2 GFLOPS	105.6 GFLOPS	73.32 GFLOPS
CPU最高内存带宽	68.2GB/s	43GB/s	34.1GB/s
平均每核内存带宽	8.5GB/s	3.58GB/s	4.26GB/s
平均每内核IO带宽	42GB/s	8.13GB/s	12.8GB/s

GPU Performance = End of the CPU? NO!



For Enterprise IT: Petascale Processing for the Masses

TOP500 SUPERCOMPUTER SITES

Over half of the top 500 supercomputers today use over 1000 processors*

48 GPU Pipes
x 8 Flops/cycle
x 3 GHz

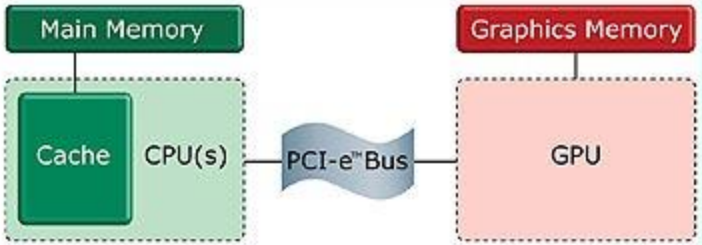
Power and bandwidth unconstrained

socket
= 1 PetaFLOP

Ask.com Google

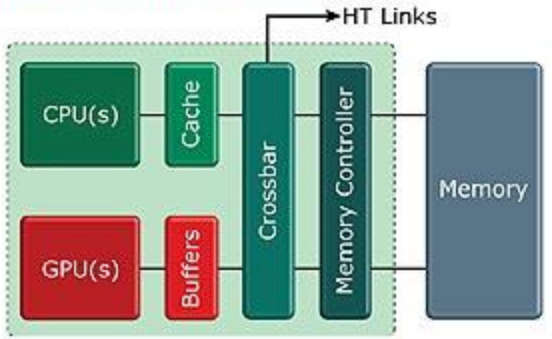
* Source: top500.org

The Data Efficiency Benefits of Silicon-Level Integration



Maximum power usage

The Data Efficiency Benefits of Silicon-Level Integration



Planned step-function improvement in power/performance with cost reduction opportunities

中国的超级计算机：天河1号 1.206P，星云3P，都是异构计算体系。

AMD Radeon HD 5970

单精度浮点处理能力	4.64 TFLOPS
双精度浮点处理能力	928 GFLOPS
核心频率	725 MHz
处理器核心数量	3200
内存类型	GDDR5
显存容量	4 GB
最大功耗	294 W
显存带宽	256 GB/sec

中国的超级计算机

天河一号

CPU　: 3072x2颗 Intel Quad Core Xeon E5540 3.0GHz
GPU　: 2560颗 ATI Radeon 4870x2 575MHz
内存　: 98TB
世界排名第七

星云

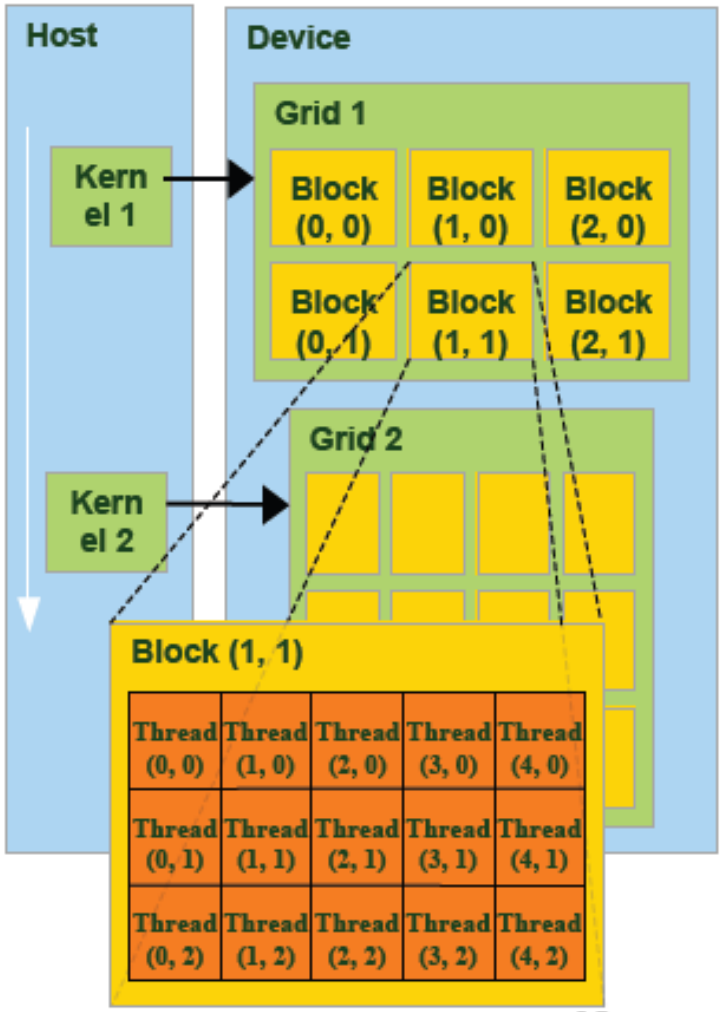
CPU　: 9280颗 Intel X5650 2.6GHz
GPU　: 4640颗 Nvidia C2050
内存　: 98TB
世界排名第二

“星云”在Linpack测试中的效能表现为每秒1.271 petaflops，略逊于美国能源部旗下橡树岭国家实验室(Oak Ridge Lab)所有的超级计算机“Jaguar”的每秒1.75 petaflops。不过由美国厂商Cray所打造、采用AMD六核心处理器Istanbul的“Jaguar”，理论峰值为每秒2.3 petaflops，逊于“星云”每秒2.98 petaflops的理论峰值。星云”功耗仅2.55MW，低于“Jaguar”的7MW。

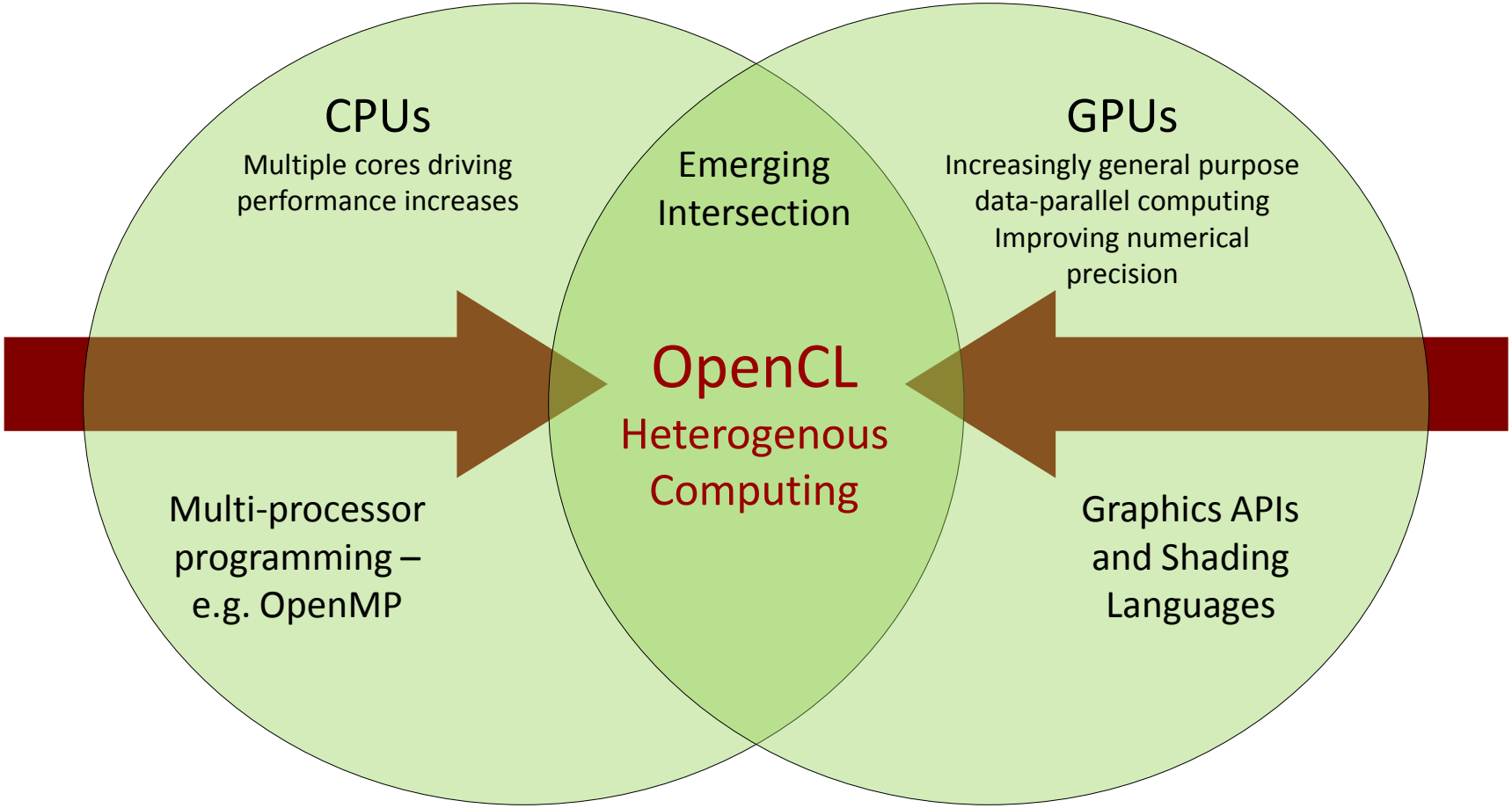
GPU大规模并行计算

并行线程组织结构

- Thread: 并行的基本单位
- Thread block: 互相合作的线程组
  - Cooperative Thread Array (CTA)
  - 允许彼此同步
  - 通过快速共享内存交换数据
  - 以1维、2维或3维组织
  - 最多包含512个线程
- Grid: 一组thread block
  - 以1维或2维组织
  - 共享全局内存
- Kernel: 在GPU上执行的核心程序
  - One kernel ↔ one grid



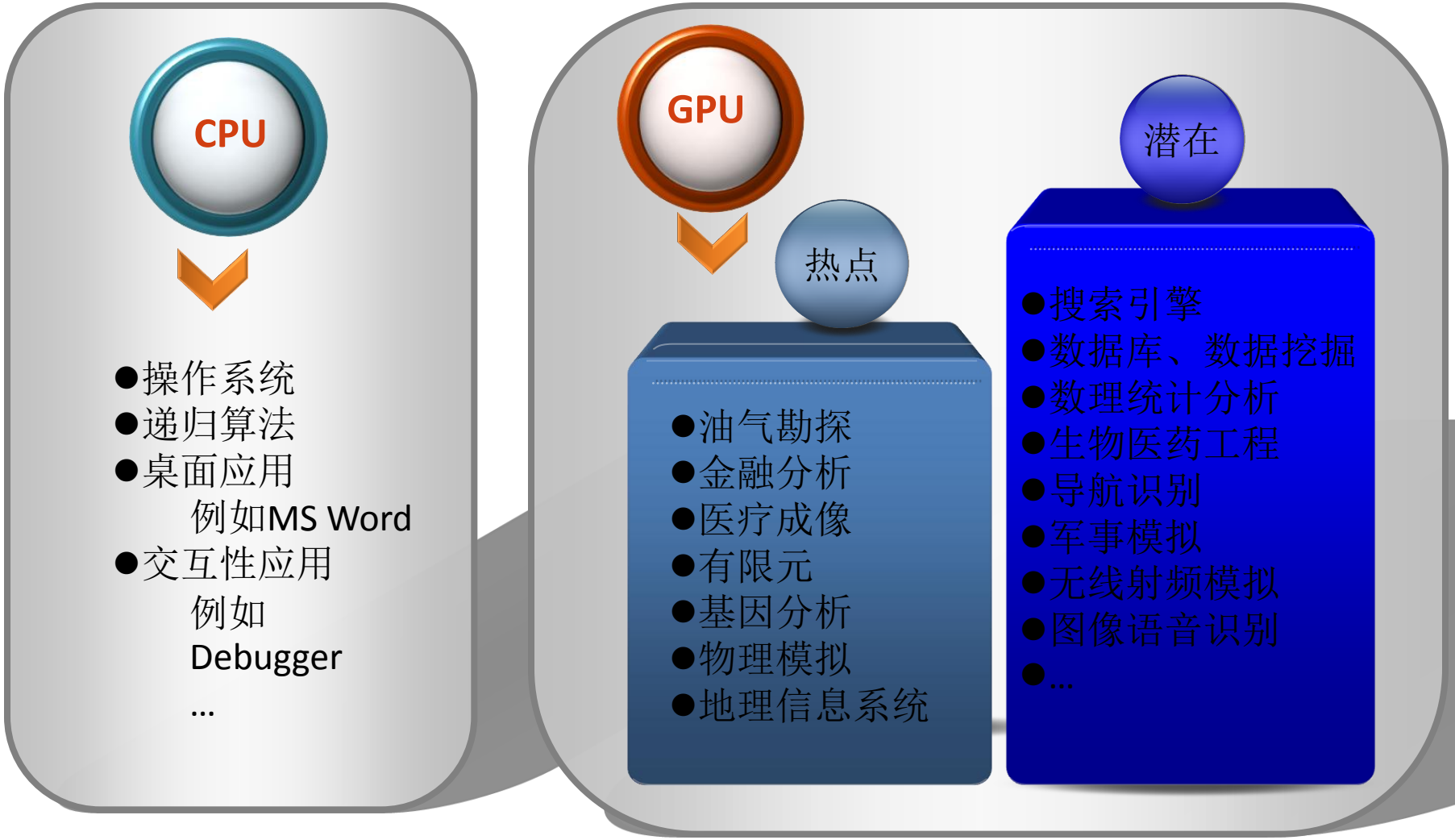
Processor Parallelism



OpenCL – Open Computing Language

Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

适用的应用范围


CPU
●操作系统
●递归算法
●桌面应用
例如MS Word
●交互性应用
例如
Debugger
…
GPU
热点
●油气勘探
●金融分析
●医疗成像
●有限元
●基因分析
●物理模拟
●地理信息系统
潜在
●搜索引擎
●数据库、数据挖掘
●数理统计分析
●生物医药工程
●导航识别
●军事模拟
●无线射频模拟
●图像语音识别
●…

内容回顾

1、使用线程的经验：设置名称、响应中断、使用ThreadLocal
2、Executor ： ExecutorService和Future ☆ ☆ ☆
3、阻塞队列 : put和take、offer和poll、drainTo
4、线程间的协调手段：lock、condition、wait、notify、notifyAll ☆ ☆ ☆
5、Lock-free: atomic、concurrentMap.putIfAbsent、CopyOnWriteArrayList ☆ ☆ ☆
6、关于锁使用的经验介绍
7、并发流程控制手段：CountDownlatch、Barrier、Phaser
8、定时器: ScheduledExecutorService、大规模定时器TimerWheel
9、JDK 7的Fork/Join介绍
10、并发三大定律：Amdahl、Gustafson、Sun-Ni
11、神人和图书
12、业界发展情况: GPGPU、OpenCL

复习题

请回答以下问题：

1. Future是做什么用的？

2. Lock和synchronized的区别是什么？

3. 什么是CAS？

4、Lock-Free算法的三个组成部分是什么？

谢谢！