多线程---------------

# Thread

创建线程方法1：

Thread thread = new Thread(){

public void run(){

...

}

};

获取线程名

Thread.currentThread().getName();

this.getName(); ??

创建线程方法2：

Thread thread = new Thread(

new Runnable(){

public void run(){

...

}

}

);

多线程不一定能提高新能；

## Interrupt()

threadObj.interrupt() 中断

threadObj.isInterrupted() 判断是否被中断；Thread.currentThread().isInterrupted()

Thread.interrupted() 判断当前线程是否被中断，并清除当前中断状态

## suspend-resume

已废弃

suspend暂停期间不会释放任何锁资源

被挂起的线程status是Runnable状态；

Suspend需要在synchronous中吗？？

## join

等待指定对象结束

thread.join() === while(isAlive()){wait(0);}

# fork/join框架?

<http://www.blogjava.net/yongboy/>

# ThreadLocal？

put()

get()

remove()

JVM会自动清理THreadLocalmap

# ThreadLocal内部实现？

弱引用

# Callable/Future

## 关于Callable和Future

**Callable**是类似于Runnable的接口,实现Callable接口的类和实现Runable的类都是可被其他线程执行的任务, 区别如下:

1)Callable定义的方法是call,而Runnable定义的方法是run;

2)Callable的call方法可以有返回值,而Runnable的run方法不能有返回值;

3)Callable的call方法可抛出异常,而Runnable的run方法不能抛出异常;

**Future**表示异步计算的结果,它提供了检查计算是否完成的方法,以等待计算的完成,并检索计算的结果.Future的cancel方法取消任务的执行,有一个布尔参数,参数为true表示立即中断任务的执行,参数为false表示允许正在运行的任务运行完成.Future的get方法等待计算完成,获取计算结果

## 示例1：异步执行一个任务

|  |
| --- |
| ExecutorService threadPool = Executors.*newSingleThreadExecutor*();    // 异步执行一个任务  Future<String> future = threadPool.submit(**new** Callable<String>() {  @Override  **public** String call() **throws** Exception {  Thread.*sleep*(2000);  **return** "hello :" + ",Thread="+Thread.*currentThread*().getName();  }  });  System.*out*.println(future.get()); |

## 示例2：异步执行多个任务

|  |
| --- |
| ExecutorService threadPool = Executors.*newFixedThreadPool*(3);  CompletionService<String> completionService  = **new** ExecutorCompletionService<String>(threadPool);  // 异步执行一组任务  **for** (**int** i = 0; i < 10; i++) {  **final** **int** temp = i;  completionService.submit(**new** Callable<String>() {  @Override  **public** String call() **throws** Exception {  Thread.*sleep*(2000);  **return** "task\_" + temp + ",Thread="+Thread.*currentThread*().getName();  }  });  }    **for** (**int** i = 0; i < 10; i++) {  System.*out*.println(completionService.take().get());  }  threadPool.shutdown(); |

# Future

# CompletableFuture?

CompletableFuture.supplyAsync(…)

# CompletionStage

# Timer

tast = new TimerTask(){

public void run(){

System.out.pringln("bombing!");

}

};

waitTime = 10000; // 10s

new Timer().schedule(tast, waitTime);

new Timer().schedule(tast, waitTime, intever);

waitTime毫秒后开始执行，以后每intever毫秒后执行一次；

嵌套Timer

# ExecutorService?

ExecutorService service = Executors.newFixedThreadPool(**int** nThreads);

ExecutorService service = Executors.newCachedThreadPool();

ExecutorService service = Executors.*newSingleThreadExecutor*();

ScheduledExecutorService service = Executors.*newSingleThreadScheduledExecutor*();

ScheduledExecutorService service = Executors.*newScheduledThreadPool*(corePoolSize);

## 类结构

|  |
| --- |
| **public** **interface** Executor {  **void** execute(Runnable command);  } |

|  |
| --- |
| **public** **interface** ExecutorService **extends** Executor {  **void** shutdown();  List<Runnable> shutdownNow();  **boolean** isShutdown();  **boolean** awaitTermination(**long** timeout, TimeUnit unit)  **boolean** isTerminated();  <T> Future<T> submit(Runnable task, T result);  <T> Future<T> submit(Callable<T> task);  Future<?> submit(Runnable task);  <T> List<Future<T>> invokeAll(Collection<? **extends** Callable<T>> tasks)  <T> List<Future<T>> invokeAll(Collection<? **extends** Callable<T>> tasks,  **long** timeout, TimeUnit unit);  <T> T invokeAny(Collection<? **extends** Callable<T>> tasks) ;  <T> T invokeAny(Collection<? **extends** Callable<T>> tasks,  **long** timeout, TimeUnit unit);  } |

|  |
| --- |
| **public** **abstract** **class** AbstractExecutorService **implements** ExecutorService {  }  **public** **class** ThreadPoolExecutor **extends** AbstractExecutorService {  } |

## 构造函数？

## 队列？

## 拒绝策略

## 任务调度

scheduledExecutorService.schedule(Runnable cmd, long delay, TimeUnit unit);

scheduledExecutorService.scheduleAtFixedRate(Runnable cmd, long initialDelay, long period, TimeUnit unit);

从给定开始时间计划一个周期性的任务，initialDelay+n\* period；

如果上一个任务超过period，则任务结束后，立即运行下一个任务；

scheduledExecutorService.scheduleWithFixedDelay(Runnable cmd, long initialDelay, long delay, TimeUnit unit);

上一个任务结束后再经过delay时间进行任务调度

所有调度器，在任务异常时，如果不作处理，调度器会终止

# AKKA?

# JMM Java内存模型

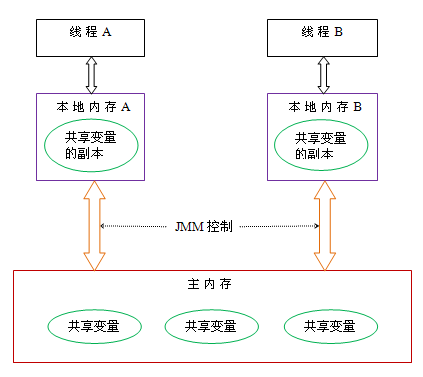
1 线程之间的通信：消息传递 和 共享内存

## JMM概念

Java线程之间的通信由Java内存模型（JMM）控制，JMM决定一个线程对共享变量的写入何时对另一个线程可见。

JMM定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本。

本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。JMM模型抽象如下



## 本地内存

如上，是线程私有的缓存集合；

## 重排序

在执行程序时为了提高性能，编译器和处理器常常会对指令做重排序。重排序分三种类型：

1. **编译器优化**的重排序。编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。

2. **指令级并行**的重排序。现代处理器采用了指令级并行技术（Instruction-Level Parallelism， ILP）来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。

3. **内存系统的重排序**。由于处理器使用缓存和读/写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

JMM的**编译器重排序规则**会禁止特定类型的编译器重排序

JMM的**处理器重排序规则**会要求编译器在生成指令序列时，插入特定类型的**内存屏障**(memory barriers，intel称之为memory fence）指令，通过内存屏障指令来禁止特定类型的处理器重排序（不是所有的处理器重排序都要禁止）。

JMM属于语言级的内存模型，它确保在不同的编译器和不同的处理器平台之上，通过禁止特定类型的编译器重排序和处理器重排序，为程序员提供一致的内存可见性保证。

## 内存屏障

编译器在生成内存屏障指令，JMM把内存屏障指令分为下列四类：

编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。

屏障类型 指令示例

1 LoadLoad Barriers Load1; LoadLoad; Load2

确保Load1数据的装载，之前于Load2及所有后续装载指令的装载。

2 StoreStore Barriers Store1; StoreStore; Store2

确保Store1数据对其他处理器可见（刷新到内存），之前于Store2及所有后续存储指令的存储。

3 LoadStore Barriers Load1; LoadStore; Store2

确保Load1数据装载，之前于Store2及所有后续的存储指令刷新到内存。

4 StoreLoad Barriers Store1; StoreLoad; Load2

确保Store1数据对其他处理器变得可见（指刷新到内存），之前于Load2及所有后续装载指令的装载。StoreLoad Barriers会使该屏障之前的所有内存访问指令（存储和装载指令）完成之后，才执行该屏障之后的内存访问指令。

执行该屏障开销会很昂贵，因为当前处理器通常要把写缓冲区中的数据全部刷新到内存中（buffer fully flush）

## happens-before

从JDK5开始，java使用新的JSR -133内存模型,在JMM中，如果一个操作执行的结果需要对另一个操作可见，那么这两个操作之间必须要存在happens-before关系。

这里提到的两个操作既可以是在一个线程之内，也可以是在不同线程之间。

与程序员密切相关的happens-before规则如下：

程序顺序规则：一个线程中的每个操作，happens- before 于该线程中的任意后续操作。

监视器锁规则：对一个监视器锁的解锁，happens- before 于随后对这个监视器锁的加锁。

volatile变量规则：对一个volatile域的写，happens- before 于任意后续对这个volatile域的读。

传递性：如果A happens- before B，且B happens- before C，那么A happens- before C。

一个happens-before规则通常对应于多个编译器和处理器重排序规则。

happens-before关系，并不意味着前一个操作必须要在后一个操作之前执行！

## 数据依赖性

如果两个操作访问同一个变量，且这两个操作中有一个为写操作，此时这两个操作之间就存在数据依赖性。数据依赖分下列三种类型：

名称 代码示例 说明

写后读 a = 1;b = a; 写一个变量之后，再读这个位置。

写后写 a = 1;a = 2; 写一个变量之后，再写这个变量。

读后写 a = b;b = 1; 读一个变量之后，再写这个变量。

上面三种情况，只要重排序两个操作的执行顺序，程序的执行结果将会被改变。

编译器和处理器在重排序时，会遵守数据依赖性，不会改变存在数据依赖关系的两个操作的执行顺序。

## as-if-serial

## 顺序一致性内存模型

## 总线串行

总线的这些工作机制可以把所有处理器对内存的访问以串行化的方式来执行；在任意时间点，最多只能有一个处理器能访问内存。这个特性确保了单个总线事务之中的内存读/写操作具有原子性。

比如在一个32位系统中，读写一个64位的数据，会分成两个操作来执行；

## volatile

参考：<http://ifeve.com/java-memory-model-4/>

对一个volatile变量的单个读/写操作，与对一个普通变量的读/写操作使用同一个锁来同步，它们之间的执行效果相同。

锁的happens-before规则保证释放锁和获取锁的两个线程之间的内存可见性，这意味着对一个volatile变量的读，总是能看到（任意线程）对这个volatile变量最后的写入。

volatile变量自身具有下列特性：

可见性：对一个volatile变量的读，总是能看到（任意线程）对这个volatile变量最后的写入；

原子性：对任意单个volatile变量的读/写具有原子性；

volatile变量的写-读可以实现线程之间的通信。

从内存语义的角度来说，volatile与锁有相同的效果：

volatile写和锁的释放有相同的内存语义；包括被修改的非volatile变量；

volatile读与锁的获取有相同的内存语义；包括被修改的非volatile变量；????

当读一个volatile变量时，JMM会把该线程对应的本地内存置为无效。线程接下来将从主内存中读取共享变量。 是所有本地内存吗？？？？

锁的语义决定了临界区代码的执行具有原子性。

下面是JMM针对编译器制定的volatile重排序规则表：

|  |  |  |  |
| --- | --- | --- | --- |
| 是否能重排序 | 第二个操作 | | |
| 第一个操作 | 普通读/写 | volatile读 | volatile写 |
| 普通读/写 |  |  | NO |
| volatile读 | NO | NO | NO |
| volatile写 |  | NO | NO |

从上表我们可以看出：

当第二个操作是volatile写时，不管第一个操作是什么，都不能重排序。这个规则确保volatile写之前的操作不会被编译器重排序到volatile写之后。

当第一个操作是volatile读时，不管第二个操作是什么，都不能重排序。这个规则确保volatile读之后的操作不会被编译器重排序到volatile读之前。

当第一个操作是volatile写，第二个操作是volatile读时，不能重排序。

hc补充：

volatile读前面的普通读写可以重排到后面？

volatile写后面的普通读写可以重排到前面？

# CPU缓存优化

填充padding

一个缓存行存两个主存数据，一个失效导致另一个一起失效；

# JUC

JUC是java.util.concurrent的简称，理解这一部分，对整个java体系是很重要的

JUC tools

Locks

Atomic

Executor

Collections

# AtomicInteger

两个属性

private volatile int value;

private static final long valueOffset;

提供一个简单，高效，安全的递增递减方案。

int get();

# AtomicLong

# AtomicBoolean

# AtomicReference

//eg：给余额小于20的用户送20元，只能一次；

AtomicReference<Integer> money = new AtomicReference<Integer>();

money.set(19)

Integer m =

# AtomicStampedReference

带时间戳的对象引用, 解决cas之前多次修改值的情况

public boolean compareAndSet(V expRef, V newRef, int expStamp, int newStamp);

public V getReference()

public int getStamp()

public void set(V newRef, int newStamp);

eg：

# AtomicIntegerArray

public final int get(int i);//

public final int length();

public final int getAndSet(int I, int newValue)

public final int getAndIncrement(int i)

public final int getAndDecrement(int i)

public final int getAndAdd(int I, int delta)

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

# AtomicLongArray

# AtomicRerefenceArray

# AtomicIntegerFieldUpdater工具

让普通变量也享受CAS安全性

scoreUpdater = AtomicIntegerFieldUpdater.newUpdater(Xxxx.class, “score”);

scoreUpdater.incrementAndGet(stu);

注意：

要保证变量被正确读取，必须是volatile类型；

# AtomicLongFieldUpdater

# AtomicReferenceFieldUpdater

# LongAdder

将64位整数进行分割，存储在不同的变量中，以防止多线程竞争；

分段，

避免了伪共享

添加注解@sun.misc.Contended，虚拟机自动解决伪共享，需要配置参数-XX:-RestrictContended

# LongAccumulator

LongAdder每次只能对给定的整数执行一次加法，LongAccuulator可以实现任意函数操作;

public LongAccumulator(LongBinaryOperator func, long identity);

# Unsafe

sun.misc.Unsafe //封装不安全的操作

Java中的指针

public final native boolean compareAndSwapInt(Object o, long offset, int expected, int x);

// 返回给定field的内存地址偏移量，这个值对于给定的filed是唯一的且是固定不变的。

public native long staticFieldOffset(Field field);//静态属性在类对象中的偏移量

public native long objectFieldOffset(Field field);//对象属性在对象中的偏移量

public native int getIntVolatile(Object obj, long l);

public native int arrayBaseOffset(Class class1);

public native int arrayIndexScale(Class class1);

|  |
| --- |
| public final class Unsafe {  public static final int ARRAY\_INT\_BASE\_OFFSET;  public static final int ARRAY\_INT\_INDEX\_SCALE;  public native int getInt (Object obj, long offset); //获取值  public native int putInt (Object obj, long offset, int x); //设置值  public native int putIntVolatile (Object obj, long offset, int x);//设置值，volatile语义  public native int getIntVolatile (Object obj, long offset,); //获取值  public native long getLong(Object obj, long l);  public native long staticFieldOffset(Field field);  public native int arrayBaseOffset(Class class1);/ 获取数组第一个元素的偏移地址  public native int arrayIndexScale(Class class1);//获取数组的转换因子，即数组中元素的增量地址  static  {  ARRAY\_INT\_BASE\_OFFSET = theUnsafe.arrayBaseOffset([I);  ARRAY\_INT\_INDEX\_SCALE = theUnsafe.arrayIndexScale([I);  }  } |

arrayBaseOffset方法是一个本地方法，可以获取数组第一个元素的偏移地址。

arrayIndexScale方法也是一个本地方法，可以获取数组的转换因子，也就是数组中元素的增量地址。

ARRAY\_INT\_BASE\_OFFSET

ARRAY\_BYTE\_BASE\_OFFSET

ARRAY\_INT\_INDEX\_SCALE

ARRAY\_BYTE\_INDEX\_SCALE

# CAS-ABA

CAS之前可能存在已被多次修改值的情况，ABA问题，比如修改成其他值后又改成当前值。（同时充值消费）

解决办法使用AtomicStamepedReference

# Semaphore信号量

控制访问自身的线程数量；同一时间最多允许多少线程访问

public Semaphore(int permits); // permits: 许可

public Semaphore(int permits,Boolean fair)

方法：

public void acquire() //获取一个准入许可，否则等待或被中断

public void acquireUninterruptibly()

public void tryAcquire() //

public void tryAcquire(long timeout,TimeUnit unit)

public void release()

|  |
| --- |
| **public** **static** **void** main(String[] args) {  ExecutorService pool = Executors.*newCachedThreadPool*();  **final** Semaphore semaph = **new** Semaphore(3);//定义3个信号量  **for** (**int** i = 0; i < 10; i++) {  pool.execute(**new** Runnable() {  @Override  **public** **void** run() {  **try** {  semaph.acquire(); // 获取信号量  // semaph.availablePermits()); 剩余信号量  // do...  Thread.*sleep*(2000);  } **catch** (Exception e) {} **finally** {  semaph.release(); // 归还信号量  }  }  });  }  } |

# Exchanger

实现两个线程之间的数据交换

|  |
| --- |
| **public** **static** **void** main(String[] args) {  ExecutorService service = Executors.*newCachedThreadPool*();  **final** Exchanger<String> exchanger = **new** Exchanger<String>();  // 线程1  service.execute(**new** Runnable(){  @Override  **public** **void** run() {  **try** {  String data = "data1";  System.*out*.println("线程1换出的数据是["+data+"]");  ThreadUtils.*sleepRandom*(2000);  data = exchanger.exchange(data);  System.*out*.println("线程1换回的数据是["+data+"]");  } **catch** (Exception e) {  }  }  });    // 线程2  service.execute(**new** Runnable(){  @Override  **public** **void** run() {  **try** {  String data = "data2";  System.*out*.println("线程2换出的数据是["+data+"]");  ThreadUtils.*sleepRandom*(2000);  data = exchanger.exchange(data);  System.*out*.println("线程2换回的数据是["+data+"]");  } **catch** (Exception e) {  }  }  });  } |

结果

|  |
| --- |
| 线程1换出的数据是[data1]  线程2换出的数据是[data2]  线程1换回的数据是[data2]  线程2换回的数据是[data1] |

# CountDownLatch

倒计时计数器

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** Exception {  **final** CountDownLatch downLatch = **new** CountDownLatch(1);  **final** CountDownLatch endLatch = **new** CountDownLatch(10);  **for** (**int** i = 0; i < 10; i++) {  **new** Thread() {  @Override  **public** **void** run() {  **try** {  ThreadUtils.*sleepRandom*(2000);  endLatch.countDown();  System.*out*.println(ThreadUtils.*getThreadName*()+ " wait");  // 线程在这停下，等待downLatch变为0  downLatch.await(); System.*out*.println(ThreadUtils.*getThreadName*() + " end");  } **catch** (Exception e) {  }  }  }.start();  }  endLatch.await();  downLatch.countDown();  System.*out*.println("main end!");  } |

# CyclicBarrier

循环路障

类似CountDownLatch，每次凑足一批线程后，计数器就会归零；

public CyclicBarrier(int parties, Runnable barrierAction);  
parties:计数

barrierAction：当计数器一次计数完成后，系统会执行的动作

异常：InterruptedException, BrokenBarrierException

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** Exception {  **final** CyclicBarrier cyclicBarrier = **new** CyclicBarrier(3);  **for** (**int** i = 0; i < 3; i++) {  **new** Thread(){  @Override  **public** **void** run() {  **try** {  ThreadUtils.*sleepRandom*(2000);  System.*out*.println("等待地点1 " + ThreadUtils.*getThreadName*());  cyclicBarrier.**await**(); //线程在这停下，当有3个等待时，所有线程被唤醒  //cyclicBarrier.getNumberWaiting();// 获取当前等待的线程数    ThreadUtils.*sleepRandom*(2000);  System.*out*.println("等待地点2 " + ThreadUtils.*getThreadName*());  cyclicBarrier.**await**();    ThreadUtils.*sleepRandom*(2000);  System.*out*.println("等待地点3 " + ThreadUtils.*getThreadName*());  cyclicBarrier.await();  } **catch** (Exception e) {  }    }    }.start();  }  } |

# Phaser 移相器

Java 7的Phaser是一个灵活的线程同步工具，他包含了CyclicBarrier和CountDownLatch的相关功能

但是提供了更灵活的用法，例如支持动态调整注册任务的数量等。

使用场景：在多线程开发中，经常会碰到将多个任务分配给多个线程，每个线程执行他的任务，但是，每个任务又分为好几个阶段，每个阶段期望各个线程同时达到，意思是，每一步每个线程都要同步，当有一个线程走完第一步的时候，他得等待其他的线程都完成第一步了才能继续下一步，步调一致能解决很多问题。

Phaser phaser = new Phaser(1); //此处可使用CountDownLatch(1)

phaser.awaitAdvance(phaser.getPhase()); //此处可使用latch.await()；Advance前进

phaser.arrive(); //此处可使用latch.countDown();arrive到达；到来

phaser.arriveAndAwaitAdvance();//等待其他线程到达阶段终点，再一起进入下一个阶段

phaser.arriveAndDeregister();//则在phaser中动态减少一个线程

phaser.register();//将当前线程增加到phaser

# 非阻塞Queue

**入**

offer(E e); 末尾添加 true|false

add(E e); 末尾添加 exception Collection的方法不推荐

**出**

poll 获取并移除前端 失败返回null

remove 获取并移除前端 exception Collection的方法不推荐

peek 获取前端不移除 失败返回null

element 获取前端不移除 失败异常

clear

**实现类**

LinkedList 非线程安全

PriorityQueue

ConcurrentLinkedQueue

# BlockingQueue

接口

offer/put

poll/tack

队列满put, tack会等待

实现

ArrayBlockingQueue

LinkedBlockingQueue

PriorityBlockingQueue

DelayQueue

DelayedWorkQueue

SynchronousQueue 它没有内部容量

# ArrayBlockingQueue

# LingkedBlockingQueue

# ConcurrentLinkedQueue

# Deque

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 异常 | 特殊值 |  |  |  |
| 插入 | addFirst(e)  addlast(e) | offerFirst(e)  offerLast(e) |  |  |  |
| 获取并移除 | removeFirst()  removeLast() | pollFirst()  pollLast() |  |  |  |
| 获取不移除 | getFirst()  getLast() | peekFirst()  peekLast() |  |  |  |

# SynchronousQueue底层逻辑？

SynchronousQueue是一个没有数据缓冲的BlockingQueue，容量为0；

生产者线程对其的插入操作put必须等待消费者的移除操作take，反过来也一样。

在双核情况下，队列中一般只有一个元素时，SynchronousQueue吞吐量是LinkedBlockingQueue 和 ArrayBlockingQueue 20倍。

Object Transferer.transfer(Object e, boolean timed, long nanos);

e位空是，表情请求一个数据，e不为空是表示当前操作传递给一个消费者；

返回值null表示失败(超时或中断)；非空表示成功；

SynchronousQueue内部有一个等待队列

# TransferQueue

<http://www.blogjava.net/yongboy/archive/2012/02/04/369575.html>

线程之间的元素交换

# CopyOnWriteArrayList

# SkipList

有序

# ConcurrentHashMap

size()方法需要全局锁/所有锁

# ConcurrentSkipListMap？

跳跃表

有序

static class Index<K, V>{

final Node<K, V> node;

final Index<K, V> down;

final Index<K, V> right;

}

static class HeadIndex<K, V> extends Index<K, V>{

final int level;

}

# CompletionService

# Disruptor框架

LMAX公司开发的高效无锁内存队列，他使用无锁的方式实现了一个环形队列，非常适合生产者消费者模式；如事件，消息的发布等

内部RingBuffer，大量缓存，无锁操作

性能比BlockingQueue高一个数量级

## 构造函数

Disruptor<T> disruptor = new Disruptor(

**final** EventFactory<T> eventFactory,

**final** **int** ringBufferSize,

**final** ThreadFactory threadFactory,//或**final** Executor executor,

**final** ProducerType producerType, // SINGLE|MULTI 生产者是一个还是多个

**final** WaitStrategy waitStrategy //提高消费者的响应事件策略

)

WaitStrategy：

BlockingWaitStrategy

最低效的策略，高并发下性能最差，但对CPU消耗最小，环境差别不大；

SleepingWaitStrategy

性能和BlockingWaitStrategy差不多，对生产者线程影响小，适合对延迟要求不高场合，如异步日志

YieldingWaitStrategy

适合低延迟场合，高性能，CPU资源有一定要求

死循环监控缓冲区，内部会Thread.yield()让出CPU

BusySpinWaitStragety

死循环监控缓冲区

只有在对延时非常苛刻场景才使用

## Maven

<dependency>

<groupId>com.lmax</groupId>

<artifactId>disruptor</artifactId>

<version>3.3.6</version>

</dependency>

## 参考资料

<http://ifeve.com/disruptor-getting-started/>

<http://ifeve.com/disruptor/>

# Lock

接口

# AQS:AbstractQueuedSynchronizer

AbstractQueuedSynchronizer,同步器

提供了一个基于FIFO队列，可以用于构建锁或者其他相关同步装置的基础框架。

用了一个int来表示状态

通过类似acquire和release的方式来操纵状态。

getState()

setState(int)

compareAndSetState(int, int)

利用同步器将锁的语义实现，然后在锁的实现中聚合同步器。

锁的API是面向使用者的，它定义了与锁交互的公共行为，实现是依托给同步器来完成;

Lock.lock()--> aqs.acquire() -> sync.tryAcquire()

Lock.unlock()--> aqs.release() -> sync.tryRelease()

|  |
| --- |
| class AbstractQueuedSynchronizer{}  private transient volatile Node head;  private transient volatile Node tail;  private volatile int state; //线程计数  getState()  setState()  compareAndSetState()  protected boolean tryAcquire(int arg) 排它的获取这个状态。  protected boolean tryRelease(int arg) 释放状态。  protected int tryAcquireShared(int arg) 共享的模式下获取状态。  protected boolean tryReleaseShared(int arg) 共享的模式下释放状态。  protected boolean isHeldExclusively() 在排它模式下，状态是否被占用。  } |

## Sync

class Sync extends AbstractQueuedSynchronizer

abstract void lock();

## Node

//mode：独占锁还是共享锁，默认为null，独占锁。

Node node = new Node(Thread.currentThread(), mode);

的包装的线程状态

SIGNAL(-1) ：线程的后继线程正/已被阻塞，当该线程release或cancel时要重新这个后继线程(unpark)

CANCELLED(1)：因为超时或中断，该线程已经被取消

CONDITION(-2)：表明该线程被处于条件队列，就是因为调用了Condition.await而被阻塞

PROPAGATE(-3)：传播共享锁

0：0代表无状态

|  |
| --- |
| **static** **final** **class** Node {  **static** **final** Node ***SHARED*** = **new** Node();  **static** **final** Node ***EXCLUSIVE*** = **null**;  // 节点的状态常量  // 节点因为超时或中断被取消  **static** **final** **int** ***CANCELLED*** = 1;  // 表示当前节点的后继节点包含的线程需要运行，也就是unpark；  // 后继节点将(或已)被阻塞，当前节点释放或取消时，需要unpark它的后继节点  **static** **final** **int** ***SIGNAL*** = -1;  // 表示当前节点在等待condition，也就是在condition队列中；  // 该状态仅供在条件队列中的节点使用。当该节点在同步队列中时，该状态为0。  **static** **final** **int** ***CONDITION*** = -2;  // 表示当前场景下后续的acquireShared能够得以执行；仅在共享模式下使用。在doReleaseShared()方法中，仅仅会设置头节点的状态为PROPAGATE。  **static** **final** **int** ***PROPAGATE*** = -3;  0：除以上场景外的其他情况，状态均设置为0。    **volatile** **int** waitStatus; //节点的状态  **volatile** Node prev;  **volatile** Node next;  **volatile** Thread thread;  Node nextWaiter; //共享锁new Node(); 排他锁null |

## 参考：

<http://suo.iteye.com/blog/132946>

java并发编程--AbstractQueuedSynchronizer加锁和解锁分析(二）- (六)

<http://www.infoq.com/cn/articles/jdk1.8-abstractqueuedsynchronizer>

深度解析Java 8：JDK1.8 AbstractQueuedSynchronizer的实现分析（上）

<http://www.infoq.com/cn/articles/java8-abstractqueuedsynchronizer>

深度解析Java 8：AbstractQueuedSynchronizer的实现分析（下）

<http://ifeve.com/introduce-abstractqueuedsynchronizer/>

# LockSupport

线程阻塞工具类

与Object.wait()相比，不需要获得某个对象的锁，不会InterrupedExcepton异常；

LockSupport.part() 把当前线程交给OS内核进行阻塞

LockSupport.part(Object) 在指定对象上阻塞当前线程，

LockSupport.unpart(thread)阻塞当前线程

parkNanos(), parkUntil()限时等待

LockSupport位每一个线程准备了一个许可，如果许可可用，park()立即返回，并消费这个许可(将许可变为不可用)，如果许可不可用，就会阻塞。unpark()使一个许可可用，许可不能累加，永远只有一个；

unpark()可以在park()之前发生

lockSupport与Thread.suspend()相比，resume就不能在suspend()之前调用；

park的线程状态是WAITING(parking)，resume状态是RUNNABLE

park()能够接受中断信息，但不会抛InterrptedException，只是默默返回，使用Thread.interrupted()判断;

lockSupport.park();

if(Thread.interrupted()){

}

# ReentrantLock

参考: <http://www.aichengxu.com/view/2506677>

多次获取锁，需要多次释放。过度释放会异常IllegalMonitorStateException；

## lock()-unlock()

## 公平/非公平锁

Lock lock = new ReentrantLock(true); //公平锁

// true就是公平锁，false非公平锁 。所谓公平锁就是让等待最长的线程最早获得该锁（获得锁的顺序和申请锁的顺序是一致的），性能比非公平锁性能差

单词：fair 公平的

## tryLock

tryLock(), tryLock(long time, TimeUnit unit)

lockInterruptibly()

可避免死锁

public void tryTimeLock() throws InterruptedException{

Lock lock = new ReentrantLock();

if (lock.tryLock(50, TimeUnit.SECONDS)) { // 50秒内未获取锁超时返回

try {

// manipulate protected state

} finally {

lock.unlock();

}

} else {

// perform alternative actions

}

}

## 可轮询

public void tryLock(){

Lock lock = new ReentrantLock();

if (lock.tryLock()) { //尝试获取锁，如果拿到用锁，拿不到转而其它操作

try {

// manipulate protected state

} finally {

lock.unlock();

}

} else {

// perform alternative actions

}

}

## lockInterruptibly()可中断锁

public void lockInterruptibly() throws InterruptedException{

Lock lock = new ReentrantLock();

try{

lock.lockInterruptibly(); //优先响应时间中断，而不是重新获取锁

// manipulate protected state

lock.unlock();

}catch (InterruptedException e){ }

}

# ReentrantLock源码分析

ReentrantLock的所有操作委托到内部内Sync上；

static abstract class Sync extends AbstractQueuedSynchronizer

具体内部了有两个

final static class NonfairSync extends Sync

final static class FairSync extends Sync

ReentrantLock.lock()

->FairSync.lock();

->AQS.acquire(1);

->FairSync.tryAcquire(1)

成功AQS.exclusiveOwnerThread = currentThread;

失败:加入队列尾部

addWaiter(Node.EXCLUSIVE), 1)

acquireQueued(Node)

shouldParkAfterFailedAcquire();

LockSupport.park()

sun.misc.Unsafe.park()

os.pthread\_mutex\_lock()

unlock()

release();

tryRelease(arg);

unparkSuccessor(h);

CLH队列

# Condition

wait()和notify()是和Synchronized配合使用的；

condition方法:

await() //使当前线程等待，并释放锁资源，直到其它线程调用signal或signalAll

awaitUninterruptibly //同await，但是在等待过程中不响应中断

awaitNanos()()

awaitUntil()

signal() //唤醒一个等待中的线程

signalAll()

1 锁只能互斥，不能通讯(告诉其他线程你可以工作了)，可以将lock与condition结合使用，达到synchronized + object.wait() + object.notify() 组合使用效果; 详情请参考JDK帮组文档搜索Condition，其中有BoundedBuffer的实现

2 一个lock可以生成多个condition

Condition condition1 = lock.newCondition();

Condition condition2 = lock.newCondition();

Condition

// 虚假唤醒

|  |
| --- |
| **static** **class** Business{  Lock lock = **new** ReentrantLock();  Condition condition = lock.newCondition();    **public** **void** sub() **throws** Exception{  // ...  condition.await();  // ...  condition.signal();  }  } |

阻塞队列

# ReentrantLock与synchronized区别

JDK5之前Lock性能比较高，JDK6开始性能差不多

Synchronized是JVM层实现的，系统而已监控锁的释放与否；

Synchronizedtongguo wait/notify进行线程间调度；Lock通过condition await/signal进行线程间调用

Synchronized只能在对象内方法

synchronized仅在方法内或者代码块内有效，而Lock可以跨越方法甚至通过在对象之间传递，跨越对象进行同步。

加锁后的状态不一样：synchronized加锁后是blocked状态；lock加锁后是waiting状态；

synchronized里面有自旋机制，先自选再阻塞；Lock没有，

Lock可以公平和非公平，Synchronized是非公平的，

Lock逻辑控制灵活，但需要程序员主动加锁及释放锁，

Lock是使用代码实现的，系统无法自动释放锁，需要显示unlock；

Lock提供了高级功能：可相应中断，限时等待（中断锁，定时等候锁）

Lock可以创建多个conditon用于不同条件的await()

Reentrant的中断或限时等待可以避免死锁

# Metux

Metux互斥体，

# ReadWriteLock/ReentrantReadWriteLock

详情参考:

<http://blog.csdn.net/coslay/article/details/45226147>

<http://blog.csdn.net/coslay/article/details/45226147>

ReadWriteLock是一个接口

ReentrantReadWriteLock是它的实现类，

ReentrantReadWriteLock包括子类ReadLock和WriteLock

# StampedLock

Java8

乐观锁

StampedLock sl = new StampedLock();

Long stamp = sl.writeLock();

//doOthers

sl.unlockWrite(stamp)

Long stamp = sl.readLock();

# 锁优化

减少锁持有时间，局部锁

减少锁粒度，比如ConcurrentHashMap

读写分离替换独占锁

锁分离，如BlockQueue的put/tack,使用了两把锁

锁粗化，顺序多次获取锁，或循环体内的获取锁

ThreadLocal

乐观锁

CAS

# JDK的锁优化

## 锁的四种状态

级别从低到高四种状态： **无锁**，偏向锁，轻量级锁，重量级锁。

随着竞争的情况逐渐升级，目的是以为了提高获取锁释放锁的效率；

## 锁偏向

开启偏向锁-XX:+UseBiasedLocking

## 轻量级锁

## 自旋锁

## 锁消除

JVM在JIT优化时，通过对上下文的扫描，去除不可能存在共享资源竞争的锁；

比如局部变量中的Vector，不可能被其它线程访问，JVM会去掉这些无用的锁

# 无锁

性能高

死锁免疫

# 关键词

Sync Metux

# 参考网址：

<http://ifeve.com/talk-concurrency/> OK

并发编程网 聊聊并发

<http://www.blogjava.net/xylz/archive/2010/07/08/325587.html>

《深入浅出 Java Concurrency》目录：

<http://lavasoft.blog.51cto.com/62575/27069>

Java多线程编程总结

<http://blog.csdn.net/column/details/concurrency.html>

Java并发编程系列

<http://blog.csdn.net/chunlongyu/article/category/6399716>

Java并发编程系列