高并发程序设计

<http://tutorials.jenkov.com/java-concurrency/concurrency-vs-parallelism.html>

<http://blog.csdn.net/net19880504/article/details/20807403>

# 一、 明确概念

## 1 并发与并行

1 并发（Concurrency）其反义词是sequential(顺序执行)

并行(Parallelism) ,其反义词是Serial（串行）

2 并发：一个cpu交替处理任务的能力,cpu对任务进行切换

并行：多个cpu（前提是多个），每个cpu各处理一个任务，同时执行

如果按照任务的角度：

并发：交替执行的是不同的任务，（顺序执行是一个执行完，再执行另外一个）

并行：任务分解，同时执行的是一个任务的子任务。（串行是任务不能分解）

参考：<http://tutorials.jenkov.com/java-concurrency/concurrency-vs-parallelism.html>

## 2 同步和异步

同步（Synchronous） 异步(Asynchronous)

二者形容的一次方法调用。

同步：调用一个方法，等待结果返回。函数的调用和执行是同一个线程。

异步：调用一个方法，不等待结果返回，自己接着其他的工作。在函数内部会**重启一个线程**来执行任务，当任务结束后，使用回调(callback)**通知**调用者或者Future （java中）或者A Message（Akka）

## 3 阻塞和非阻塞

通常形容**多线程**间访问共享资源的相互影响

阻塞：如访问临界区：锁、IO，一个线程在使用，另外一个线程**等待资源（阻塞）**，此时线程**被挂起**，放弃cpu。

非阻塞：一个线程在使用临界区，另外一个线程也来使用，当看到临界区被占用后，不是傻傻的阻塞等待，而是**立刻返回**，做其他事情。

## 4 同步IO和异步IO

我的理解：把IO当成了方法调用。

## 5阻塞IO和非阻塞IO

我的理解：多线程同时访问IO

# 二、java并行程序基础

# 三、JDK并发包

解决的问题是：

1 多线程对临界区资源的访问：

a同一时间只允许一个线程进行访问

b 对临界区的修改，其他线程可见。

2 多线程之前的通信：等待和通知

## 1 ReentrantLock重入锁

与Synchronized相比，有哪些改进思想？

1 具有Synchronized相同的功能，多线程访问临界区资源互斥

2 中断响应：

**阻塞的线程(等待获取资源的线程)**能被中断（Synchronized只能傻傻的等待）

api: lockInterruptibly

3 有限时间等待

Synchronized只会无限期等待

Api： tryLock() / tryLock(long time, TimeUnit unit)

4 公平锁

## 2 Condition

1 Condition代替的Object，实现的线程间的互相通信（等待->通知）

2 在await方法调用后，会释放已经获得的锁

## 3 Semaphore信号量

### 1 为什么会有Semaphore

1 锁只允许一个线程访问资源；信号量为多个线程

2 Lock是独占锁，而Sempahore是共享锁

### 2 都有哪些api

1 构造函数： 传入信号量的个数

2 获取资源

a acquire：阻塞api和非阻塞api

b acquire： 是否响应中断异常

c acquire: 获取1个或多个

3 释放资源

release： 释放1个或多个

### 3 实现原理

还是继承的AQS，将其中的状态位在构造函数中赋值为n。

## 4 ReadWriteLock

读写分离。

实现原理：

1 这个类其实封装了两个锁，ReadLock和WriteLock。

这两个类都继承自Lock接口，有共同的一个关键属性：sync(**abstract static class** Sync **extends** AbstractQueuedSynchronizer)

2 AQS中的状态位被分成两部分：

*把状态的高16位用作读锁，低16位用作写锁*

3 读锁，lock的时候，调用的是sync共享锁的api：**sync**.acquireShared(1);

在自己的tryAcquireShared(**int** unused)函数中，先检查写锁；如果写锁被占用，则构建一个节点放入链表尾部，并挂起线程，等待唤醒。如果写锁为0，则读锁申请成功，读锁个数加1。

写锁，lock的时候，调用的是sync独占锁的api：**sync**.acquire(1);会检查有没有锁存在，并考虑同一个线程重入的问题。

## 5 CountDownLatch

原理：使用AQS 共享锁的api。

在构造函数中，设置AQS的状态位为n。

await的线程：尝试获取共享锁（可以有多个线程进行await），如果状态位为0，则获取成功（即当状态为countdown为0时）。否则，进行链表尾部。

Countdown线程：每调用countdown，则释放资源，状态位减1.当状态为0时，则调用doReleaseShared，唤醒链表后续的线程。

## 6 CyclicBarrier

循环（可复用）栅栏（先集合，再一起去做各自的事）

实现原理：

## 7 LockSupport

随时随地挂起一个线程，内部是许可的机制。

LockSupport.park() 和LockSupport.unpark() 次序可以不分先后。

# 四、线程池

解决问题： 线程复用的问题

# 五、并行模式和算法

多线程并行执行，提高效率的模式和算法。

## 1 单例模式

适用场景：

1多个线程需要一个共享的资源对象。如：windows的任务管理器、数据库连接池

注意：1 保证单例：构造函数私有

2 对外获取：提供一个工厂函数

3 线程安全：

a dcl(double-check locking)

b synchorized上下各有一个检查。此时检查的变量使用volatile修饰，对上面的检查起作用。也可以不添加。

4 延迟加载

5 自定义序列化

1 重写readResolve方法，返回我们的单例对象。这个方法的作用是：返回自定义的对象替换序列化的对象。

2 为什么可以使用枚举？枚举和静态代码块类似，构造函数会被自动调用。

## 2 不变模式

1 类和其中的属性不变： final修饰

2 完整的构造方法： 一经构造，不可改变

3 示例：String， Long ……

## 3 生产者-消费者模式

## 4 Disruptor 框架(高性能生产者消费者模型)

参考： <https://my.oschina.net/OutOfMemory/blog/793275>

<http://www.360doc.com/content/15/0324/17/11962419_457721378.shtml>

为什么性能要比BlockingQueue高很多？

1 使用cas替换了lock

2 解决CPU cache伪共享的问题

3 RingBuffer 环形队列

数组是预分配的，避免了GC的开销。程序运行中，生产者不断对RingBuffer中的属性进行更新，而不是直接替换RingBuffer中的元素。

### Basic Tuning Options

#### Single vs. Multiple Producers

如果生产者只有1个，则指明使用一个线程的生产者，效率提高很多。

#### Alternative Wait Strategies

消费者如何监控缓存区，有了内容去消费

##### 1 BlockingWaitStrategy：

默认使用

实现原理：同BlockingQueue类似，使用锁和条件来实现,生产者通知消费者有数据了。

特点：最节省cpu，但是性能最差。

##### 2 SleepingWaitStrategy:

实现原理： 生产者不去通知消费者，而是专注于生产；消费者循环检查有没有 内容，并进行线程休眠1纳秒（linux线程大约会暂停60微秒）。

特点： 节省cpu； 有一定的延迟

##### 3 YieldingWaitStrategy:

实现原理：每个消费线程使用Thread.yield() 来放弃时间片。

特点：低延迟。消费线程的数量最好小于cpu逻辑线程数。

##### 4 BusySpinWaitStrategy:

死循环

### Clearing Objects From the Ring Buffer

当缓存区中的对象处理完了之后，怎么处理来加速内存回收？

1 如果只有1个handler，可同时清理。

2 如果是个handler chain， 可增加一个handler用于专门清理对象。

## 5 Future模式

异步，当调用时，返回一个凭证；另外一个线程去执行，获得执行结果后，赋值给凭证中的一个结果属性。

Future接口是凭证；

Runnable接口中的run()函数获得结果，赋值真实数据。

RunnableFuture是上述两个接口的综合。

FutureTask是具体实现类。

run(): 执行单元，并把执行结果赋值给outcome。

get():获取执行结果，如果没有执行完，则线程阻塞，等待通知。

## 6 并行流水线

一个完整的任务，有相互依赖的几个步骤。

## 7 并行搜索