# JUC包简介

相关链接：

https://segmentfault.com/a/1190000015558984

https://segmentfault.com/blog/ressmix\_multithread //透彻理解Java并发编程(juc包)

JUC并发包，即java.util.concurrent包，是JDK的核心工具包，是JDK1.5之后，由Doug Lea实现并引入。

整个java.util.concurrent包，按功能划分为：

-1）juc-locks锁框架

-2）juc-atomic原子类框架

-3）juc-sync 同步器框架

-4）juc-collections 集合框架

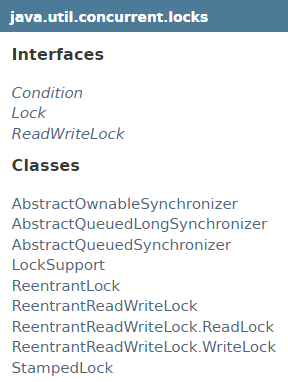
-5）juc-scheduling 调度框架

-6）juc-fj Fock/Join框架

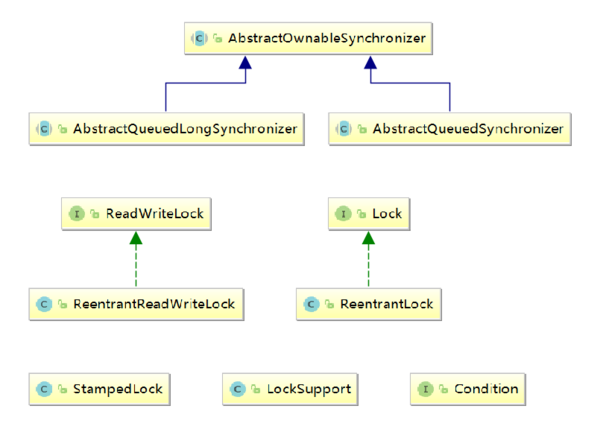
## juc-locks锁框架

早期JDK版本中，仅仅提供synchronized、wait、notify等实现多线程同步。JDK1.5之后，Doug Lea根据一系列常见的多线程设计模式，设计了JUC并发包，其中java.util.concurrent.locks包下提供了一系列基础的锁工具，用于对synchronized的补充。

java.util.concurrent.locks包的结构如下：



包内接口和类的简单UML图如下：



## juc-atomic原子类框架

JDK1.5之后，引入了java.util.concurrent.atomic工具包，该包提供了许多Java原始/引用类型的映射类，如Atomicinteger、AtomicLong、AtomicBoolean。这些类可以通过一种“无锁算法”，线程安全地操作Integer、Long、Boolean等原始类型。

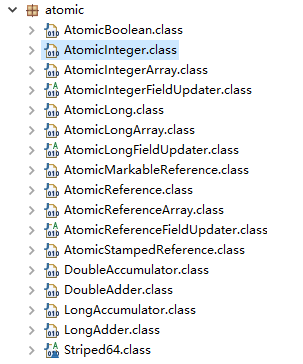
无锁算法：

其底层就是通过Unsafe类实现的一种比较并交换的算法，大致的结构如下：

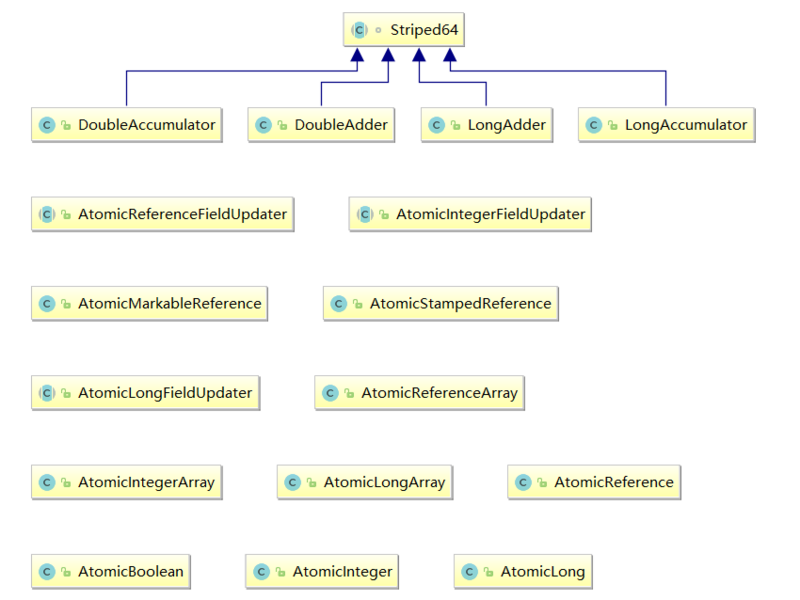
boolean compareAndSet(expectedValue, updateValue);

当希望修改的值与expectedValue相同时，则尝试将值更新为updateValue，更新成功返回true，否则返回false。

java.util.concurrent.atomic包结构如下(jdk1.8)：



简单UML图如下：



## juc-sync同步器框架

如下一些辅助同步器：

-1）倒数计数器（CountDownLatch）

作用：CountDownLatch非常实用，我们常常将一个比较大的任务进行拆分，然后开启多个线程来执行，等所有线程都执行完后，再往下执行其它操作；其内部实现了AQS框架。

-2）循环栅栏（CyclicBarrier）

作用：构造时设定等待线程数，当所有线程都到达栅栏后，栅栏放行；其内部通过ReentrantLock和Condition实现同步。

-3）信号量（Semaphore）

作用：类似于“令牌”，用于控制共享资源的访问数量；其内部实现了AQS框架。

-4）交换器（Exchanger）

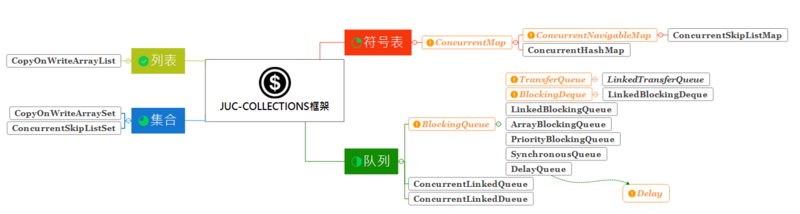
作用：类似于双向栅栏，用于线程之间的配对和数据交换；其内部根据并发情况有“单槽交换”和“多槽交换”之分。

-5）多阶段栅栏（Phaser）

作用：相当于CyclicBarrier的升级版，可用于分阶段任务的并发控制执行；其内部比较复杂，支持树形结构，以减少并发带来的竞争。

## juc-collections集合框架

按类型划分为4大类：符号表、队列、Set集合、列表。



## juc-scheduling调度框架

…

# juc-locks锁框架

juc-locks锁框架涉及三个接口：Lock、Condition、ReadWriteLock

## Lock接口

该接口提供了限时锁等待、锁中断、锁尝试等功能。

它提供了与synchronized关键字类似的功能，只是在使用时需要显式地获取和释放锁。拥有了锁获取与释放的可操作性，可中断的获取锁以及超时获取锁。

接口方法描述：

-1）void lock();

若锁不可用，出于线程调度目的，将禁用当前线程，并且在获得锁之前，该线程将一直处于休眠状态。

-2）void lockInterruptibly();

若锁不可用，则当前正在等待的线程是可以被中断的。

-3）Condition newCondition();

-4）boolean tryLock();

-5）boolean tryLock(long time, TimeUnit unit);

-6）void unlock();

实例：

Lock lock = ...;

if(lock.tryLock()) {

try {

...

} finally {

lock.unlock();

}

} else {

...//perform alternative actions

}

### 可重入内置锁

特点：

-1）每个对象都可以用做一个实现同步的锁，这些锁被称为内置锁或监视器锁。

-2）线程在进入同步代码块之前会自动获取锁，并且在退出同步代码块时会自动释放锁。

- 3）获得内置锁的唯一途径就是进入由这个锁保护的同步代码块或方法。

设计：

当某个线程请求一个由其他线程持有的锁时，发出请求的线程就会阻塞。然而，由于内置锁是可重入的，因此如果某个线程试图获得一个已经由它自己持有的锁，那么这个请求就会成功。“重入”意味着获取锁的操作的粒度是“线程”，而不是调用。重入的一种实现方法是，为每个锁关联一个获取计数值和一个所有者线程。当计数值为0时，这个锁就被认为是没有被任何线程所持有，当线程请求一个未被持有的锁时，JVM将记下锁的持有者，并且将获取计数值置为1，如果同一个线程再次获取这个锁，计数值将递增，而当线程退出同步代码块时，计数器会相应地递减。当计数值为0时，这个锁将被释放。

### ReentrantLock锁

重入锁，是Lock接口的实现类。

支持重新进入的锁，它表示该锁能够支持一个线程对资源的重复加锁。即在调用lock方法时已经获取到锁的线程，能够再次调用lock方法获取锁而不被阻塞，同时还支持获取锁的公平性和非公平性。

方法：

lock() :获取锁

tryLock() :尝试获取锁，返回true/false

unlock() :释放获取的锁

--1）同步执行代码：

Lock lock = new ReentrantLock(); //参数默认false，不公平锁

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

lock.lock(); //如果被其它资源锁定，会在此等待锁释放，达到暂停的效果

try {

...//操作省略

} finally {

lock.unlock(); //释放锁

}

--2）防止重复执行代码

Lock lock = new ReentrantLock();

if (lock.tryLock()) { //如果已经被lock，则立即返回false不会等待，达到忽略操作的效果

//或使用 lock.tryLock(5, TimeUnit.SECONDS) //如果已经被lock，尝试等待5s，看是否可以获得锁，如果5s后仍然无法获得锁则返回false继续执行

try {

... //操作

} finally {

lock.unlock();

}

}

## Condition接口

## ReadWriteLock接口

ReadWriteLock接口是一个单独的接口（未继承Lock接口），该接口提供了获取读锁和写锁的方法。

读写锁：

是一对相关的锁（读锁和写锁），读锁用于只读操作，写锁用于写入操作。读锁可以由多个线程同时保持，而写锁是独占的，只能由一个线程获取。

接口方法定义：

-1）Lock readLock()

-2）Lock writeLock()

读写锁的使用场景：

-1）高频的读操作，相对低频的写操作

-2）读操作所用的时间不会太短。（否则读写锁本身的复杂实现所带来的开销会成为主要消耗成本）

案例：在一张订单金额中，大多数情况下，线程只会高频繁地读取数据，而修改数据的频率较低。那么一般情况下，如果采用互斥锁，读/写和读/读都是互斥的，性能显然不如采用读写锁。

### ReentrantReadWriteLock锁

实现了ReadWriteLock接口，并非Lock接口。

方法：

readLock() :获取可读锁对象是Lock的实例，可读锁对象是共享锁

writeLock() :获取可写锁Lock对象是Lock的实例，可写锁对象是排他锁

# juc-sync同步器框架

## CountDownLatch（倒数计数器）

定义：

CountDownLatch非常实用，我们常常将一个比较大的任务进行拆分，然后开启多个线程来执行，等所有线程都执行完后，再往下执行其它操作。

简单实例：

--假设有N个任务，那么我们会用N来初始化CountDownLatch，然后将这个latch的引用传递到各个任务中，在每个线程完成任务后，调用latch.countDown()代表完成了一个任务，调用latch.await()方法的线程会阻塞，直到所有的任务完成。

public class Driver2 {

void main() throws InterruptedException {

CountDownLatch doneSignal = new CountDownLatch(N);

Executor e = Executors.newFixedThreadPool(8);

// 创建N个任务，提交给线程池来执行

for (int i = 0; i < N; ++i) {

e.execute(new WorkRunnable(doneSignal, i));

}

// 等待所有的任务完成，这个方法才会返回

doneSignal.await();

}

}

class WorkerRunnable implements Runnable {

private final CountDownLatch doneSignal;

private final int i;

WorkerRunnable(CountDownLatch doneSignal, int i) {

this.doneSignal = doneSignal;

this.i = i;

}

public void run() {

try {

doWork(i);

// 这个线程的任务完成了，调用 countDown 方法

doneSignal.countDown();

} catch (InterruptedException ex) {

} // return;

}

void doWork(int i) {...//省略}

}

## AQS介绍(AbstractQueuedSynchronizer)

### 概念

AQS为AbstractQueuedSychronizer类，翻译为抽象队列同步器。

如果说java.util.concurrent的基础是CAS的话，那么AQS就是整个Java并发包的核心了，ReentrantLock、CountDownLatch、Semaphore等等都用到了它。

AQS实际上以双向队列的形式连接所有的Entry，比方说ReentrantLock，所有等待的线程都被放在一个Entry中并连成双向队列，前面一个线程使用ReentrantLock好了，则双向队列实际上的第一个Entry开始运行。

AQS定义了对双向队列所有的操作，而只开放了tryLock和tryRelease方法给开发者使用，开发者可以根据自己的实现重写tryLock和tryRelease方法，以实现自己的并发功能。

实现思路：线程首先会尝试获取锁，若失败则将该线程及等待状态等信息包装成一个node节点加入到同步队列sync queue里。

AQS实现公平锁和非公平锁：

基于AQS的锁（如ReentrantLock）原理：

1）有一变量state，初始值为0，若当前线程A获取一次锁则state++，释放一次则state--，锁会记录当前持有的线程。

2）当线程A拥有锁时(state>0)。线程B尝试获取锁时对该state进行CAS(0, 1)操作，尝试几次失败后就挂起，进入一个等待队列。

3）若线程A恰好释放(--state == 0)，线程A会唤醒等待队列中第一个线程，即在等待队列的线程B，线程B被唤醒后再去对该state进行CAS(0, 1)操作

非公平锁实现：

线程A唤醒线程B时，若此时有线程C也在尝试对state进行CAS(0, 1)操作，会与等待队列中的线程B进行抢夺，谁抢到就算谁的。

公平锁实现：

线程A唤醒线程B时，若此时线程C发现有线程B在等待队列，直接将自己进入等待并挂起，线程B优先获取锁。

总结：

-1）锁状态

state为0时代表没有线程占用锁，可以去争抢这个锁，用cas将state设值。若CAS成功，说明抢到锁，则state为1，这样其它线程就抢不到了，若锁重入则将state进行+1操作，解锁则-1，直到state变成0，代表释放锁。所以lock()和unlock()要成对出现。

-2）线程的阻塞和解除阻塞

AQS中采用了LockSupport.park(thread)来挂起线程，用unpark来唤醒线程。

-3）阻塞队列

因为争抢锁的线程可能很多，但是只能有一个线程拿到锁，其他的线程都必须等待，这个时候就需要一个queue来管理这些线程。

AQS 用的是一个 FIFO 的队列，就是一个链表，每个 node 都持有后继节点的引用。AQS 采用了 CLH 锁的变体来实现。