目录

[**JAVA 修饰符** 2](#_Toc29807542)

[**java.util.concurrent.atomic：** 2](#_Toc29807543)

[AtomicBoolean 2](#_Toc29807544)

[AtomicInteger 2](#_Toc29807545)

[AtomicIntegerFieldUpdater 3](#_Toc29807546)

[AtomicMarkableReference 与AtomicStampedReference 3](#_Toc29807547)

[AtomicReference<T> 3](#_Toc29807548)

[AtomicReferenceArray 3](#_Toc29807549)

[AtomicReferenceFieldUpdater 3](#_Toc29807550)

[java.util.concurrent.locks 4](#_Toc29807551)

[AbstractQueuedSynchronizer 4](#_Toc29807552)

[JAVA之Finally的执行时机 4](#_Toc29807553)

[LockSupport原理？ 4](#_Toc29807554)

[Finally执行原理： 5](#_Toc29807555)

[java.util.concurrent： 5](#_Toc29807556)

[Queue与Deque的区别 5](#_Toc29807557)

[ArrayBlockingQueue 实现原理 5](#_Toc29807558)

[ConcurrentLinkedQueue实现原理 6](#_Toc29807559)

[CopyOnWriteArrayList/CopyOnWriteArraySet原理 7](#_Toc29807560)

[Exchanger实现原理 8](#_Toc29807561)

[DelayQueue队列 8](#_Toc29807562)

[PriorityBlockingQueue 8](#_Toc29807563)

[LinkedBlockingDeque，LinkedBlockingQueue 8](#_Toc29807564)

[SynchronousQueue队列 8](#_Toc29807565)

[LinkedTransferQueue 8](#_Toc29807566)

[ExecutorCompletionService 实现原理： 9](#_Toc29807567)

[JAVA设计模式（23种） 9](#_Toc29807568)

[Java设计模式分类 9](#_Toc29807569)

**JAVA 修饰符**

Transient （防止被序列化）

[如一个长方形类有三个属性：长度、宽度、面积（示例而已，一般不会这样设计），那么在序列化的时候，面积这个属性就没必要被序列化了]

Volatile

(让变量每次在使用的时候，都从主存中取。而不是从各个线程的“工作内存”, volatile具有synchronized关键字的“可见性”，但是没有synchronized关键字的“并发正确性”，也就是说不保证线程执行的有序性。也就是说，volatile变量对于每次使用，线程都能得到当前volatile变量的最新值。但是volatile变量并不保证并发的正确性,.volatile是变量修饰符，而synchronized则作用于一段代码或方法)

**java.util.concurrent.atomic：**

CAS[compareAndSet操作]

## AtomicBoolean

1.AtomicBoolean：原子性操作，使用UnSafe JNI底层调用，直接操作内存

内部使用volatile修饰一个value[int]变量，由UnSafe指向，每次修改该值，使用UnSafe比较并标记该值

compareAndSet()

weakCompareAndSet()

内部都是通过compareAndSwapInt进行交换保证满足expect后才能更新为update值

lazySet()

方法通过unsafe的putOrderedInt进行保存，网上说法是这个新修改的值会有一定的延迟可见性的效果

提供set与get方法

提供getAndSet方法解析（多线程保证数据一致性，操作成功）：

public final boolean getAndSet(boolean newValue) {  
 boolean prev;  
 do {  
 prev = get();  
 } while (!compareAndSet(prev, newValue));  
 return prev;  
}

首先取当前值，如果compareAndSet新值失败，比如prev在get之后被新线程修改了，导致修改失败，那么返回fasle，执行循环重新执行设置，直到比较成功，那么就会返回，这个过程是循环等待的（概率性）

## AtomicInteger

2. AtomicInteger原理同1

3.AtomicIntegerArray原理：

使用UnSafe JNI底层调用，里面维护一个数组，UnSafe每次存取修改offset数组下标偏移进行操作，因为数组是连续性的，执行赋值操作还是使用compareAndSet

## AtomicIntegerFieldUpdater

4.AtomicIntegerFieldUpdater：

使用AtomicIntegerFieldUpdater#newUpdater(Class<U> tclass,String fieldName) 创建变量操作类，但是变量必须要volatile修饰

进行内存操作某个类型的变量，初始化记录查找变量在类里面的下标offset，每次操作内存，找到改类型的下标地址，直接修改该内存

判断某个类型：

Field field = tclass.getDeclaredField(fieldName);

Modifier.isPrivate[Protect/static/Final] (filed.getModifier)

相同操作还有AtomicLong[Array / FieldUpdater]

## AtomicMarkableReference 与AtomicStampedReference

上面操作如果用户只关心多线程情况下的结果值，那么设置原子性操作没有问题，但是如果用户需要关心过程中有没有发生变化，比如

你倒了一杯水放桌子上，干了点别的事，然后同事把你水喝了又给你重新倒了一杯水，你回来看水还在，拿起来就喝，如果你不管水中间被人喝过，只关心水还在，这就是ABA问题

为了解决这个ABA问题，如果关心中间发生过改变，这里就引入AtomicMarkableReference 与AtomicStampedReference

其实现原理就是添加一个版本号（实现原理是键值对），AtomicMarkableReference使用boolean类型标志保证每次操作的时候是否被修改过，AtomicStampedReference使用int计数器来判断每次被修改的版本号，调用compareAndSet的时候判断期待值是否相同，如果相同修改更新值，这就解决了ABA问题

## AtomicReference<T>

5.AtomicReference<T>

上面鉴于只有Int与Long以及Boolean类型不能使用其他类型，所以这里需要对其他自定义类型进行扩展，进行原子性操作，那么可以使用AtomicReference包装，这里底层使用对象的内存地址比较，如果相同那么更新为新的内存地址，所以修改对象的时候，使用引用会指向新的内存地址，如果仅仅修改对象的变量，那么不会变更内存地址

对象比较使用o1.equals(o2)，不重写该方法，如果重写需要重写hashCode，而且重写hashCode有特定规则，比较内存地址，需要保证两者相同

## AtomicReferenceArray

6.AtomicReferenceArray<T>

对象数组，对于上面的进行数组操作，同AtomicIntegerArray使用offset进行下标操作

## AtomicReferenceFieldUpdater

7.AtomicReferenceFieldUpdater

使用类型中变量操作，更新类型里面的变量，使用AtomicReference相同操作，但是变量需要使用volatile修饰

# java.util.concurrent.locks

## AbstractQueuedSynchronizer

AbstractQueuedSynchronizer源码解析：（ReentrantLock可重入锁与Semaphore实现原理）

其具体实现方式是什么原理？

<https://www.cnblogs.com/snowwhite/p/9508169.html>

如何实现该流程的方法：

如果把井看成一个资源，现在有3个部分：

井，看井人（Manager），有序队列/无序集合

1. A过来了，使用lock获取了井资源

Lock方法实现原理，查询看井人，当前是否有占用，使用AtomicInteger#compareAndSet(0,1);//原子性操作

有续实现有序集合：

A家人过来，AtomicInteger自增，可以获取，每次解锁，AtomicInteger自减，直到AtomicInteger为0，才通知Manager获取队列下一个

B过来，当前是否有占用，使用AtomicInteger#compareAndSet(0,1);如果false，设置一个变量记录是否需要恢复，使用for死循环阻塞，内部维护这个变量，当A操作完成的时候，通知B，使用AtomicInteger#compareAndSet(0,1)，设置该变量为true，跳出for死循环，执行真正任务，B家人同A家人一样处理

无顺序的实现方式：

A家人过来，AtomicInteger自增，可以获取，每次解锁，AtomicInteger自减，直到AtomicInteger为0，才通知Manager获取队列下一个

B过来，当前是否有占用，使用AtomicInteger#compareAndSet(0,1);如果false，死循环查询AtomicInteger#compareAndSet(0,1)谁否为true，如果是，跳出循环，执行真正任务，此时其他CDE都是一样该实现，就不保证拿个线程，优先抢占设置成功的线程使用

或者使用Set来维护这个队列，因为无须，只需要使用concurrentSet来取线程就可以

# JAVA之Finally的执行时机

疑问点

线程排入队列，执行等待的方式是什么？

wait、sleep、join和yield方法的区别

等待与for死循环的区别（CPU与性能消耗）

# LockSupport原理？

LockSupport类可以阻塞当前线程以及唤醒指定被阻塞的线程。主要是通过park()和unpark(thread)方法来实现阻塞和唤醒线程的操作的

每个线程都有一个许可(permit)，permit只有两个值1和0,默认是0

1. 当调用unpark(thread)方法，就会将thread线程的许可permit设置成1(注意多次调用unpark方法，不会累加，permit值还是1)。
2. 当调用park()方法，如果当前线程的permit是1，那么将permit设置为0，并立即返回。如果当前线程的permit是0，那么当前线程就会阻塞，直到别的线程将当前线程的permit设置为1.park方法会将permit再次设置为0，并返回。

注意：因为permit默认是0，所以一开始调用park()方法，线程必定会被阻塞。调用unpark(thread)方法后，会自动唤醒thread线程，即park方法立即返回

ReentrantReadWriteLock 读写锁原理？

ReentrantReadWriteLock 读写锁内部使用了ReentrantReadWriteLock.ReadLock与ReentrantReadWriteLock.WriteLock

使用方式：

写写冲突，读写冲突，但是读读不冲突，使用读写锁提高只读情况下的效率，如果存在写操作，那么其他操作阻塞

# Finally执行原理：

解释下：

这里面，涉及到一个栈帧的问题。

在catch中执行的时候，return之前是需要去执行finally的。执行finally之前，把当前要返回的值的引用保存到一个slot【槽】中，也就是说return已经执行，但是还没返回（在MyEclipse的debug模式下可以看的很清楚），把当前要返回的值保存起来了。 然后执行finally，执行完finally，再从slot中取出要返回的值进行返回。如果返回的是一个对象的引用的话，那么将是finally执行后的值。如果是一个基本类型的话，那么返回的就只是那个基本类型的值了。

【在这里需要注意一点的是，当返回值是对象，而不是普通的java基本类型的时候，在finally中的修改会反映到返回值中（第一个例子返回对象），但是如果是普通的java基本类型，那么finally中的修改就不会反映到返回值中了，表现出来就是，finally虽然修改了，但是返回值仍然是2，见上面的代码（第二个例子）】。

修改会反映到返回值中，是因为存储在slot中的是对象的引用。在finally中修改的时候，修改的内容已经保存到堆中的对象了。此时return，会反映出修改后的结果。

# java.util.concurrent：

## Queue与Deque的区别

队列(queue)是一种常用的数据结构，可以将队列看做是一种特殊的线性表，该结构遵循的先进先出原则。Java中，LinkedList实现了Queue接口,因为LinkedList进行插入、删除操作效率较高

双向队列(Deque),是Queue的一个子接口，双向队列是指该队列两端的元素既能入队(offer)也能出队(poll),如果将Deque限制为只能从一端入队和出队，则可实现栈的数据结构。对于栈而言，有入栈(push)和出栈(pop)，遵循先进后出原则

Deque提供了Queue扩展方法，提供列表前元素操作

LinkedList是Deque实现类，使用Node节点进行链表操作，但是实现可以看出没有使用Lock机制，所以是非线程安全的，线程安全可以使用ConcurrentLinkedDeque/ConcurrentLinkedQueue (自旋锁+CAS实现)

## ArrayBlockingQueue 实现原理

1. 初始化固定大小的Array

2. 初始化notFull，notEmpty的reenterLock#Condition

3. 加入数组的时候调用reenterLock#lock，如果当前满了，那么notFull.wait当前线程进入等待（自旋）

public void put(E e) throws InterruptedException {  
 Objects.*requireNonNull*(e);  
 final ReentrantLock lock = this.lock;  
 lock.lockInterruptibly();  
 try {  
 while (count == items.length)  
 notFull.await();  
 enqueue(e);  
 } finally {  
 lock.unlock();  
 }  
}

4. 删除数组元素的时候reenterLock#lock，如果当前数组有空间，notFull.signal唤醒线程（使用codition特性，链式解锁，链式唤醒）

public E take() throws InterruptedException {  
 final ReentrantLock lock = this.lock;  
 lock.lockInterruptibly();  
 try {  
 while (count == 0)  
 notEmpty.await();

// notFull.signal  
 return dequeue();  
 } finally {  
 lock.unlock();  
 }  
}

与LinkedBlockingQueue区别，ArrayBlockingQueue采用连续数组来保存数据，而LinkedBlockingQueue使用单链表来保存数据，性能上面

## ConcurrentLinkedQueue实现原理

private static class Node<E> {  
 volatile E item;  
 volatile Node<E> next;  
}

使用链表形式，内部记录head与tail的Node变量。添加到队列原理如下：

public boolean offer(E e) {  
 final Node<E> newNode = *newNode*(Objects.*requireNonNull*(e));  
  
 for (Node<E> t = tail, p = t;;) {  
 Node<E> q = p.next;  
 if (q == null) {  
 // 将P的next使用原子性操作赋值为当前新加节点  
 if (*casNext*(p, null, newNode)) {  
 if (p != t) // 如果成功，再改变tail节点的指向，tail指向当前新节点  
 casTail(t, newNode);  
 return true;  
 }   
 }  
}

找到tail节点，使用自旋锁的原子性操作 添加新节点，然后将尾节点指向新节点

ConcurrentLinkedDeque 原理同ConcurrentLinkedQueue，双向节点多处prev上个指向

static final class Node<E> {  
 volatile Node<E> prev;  
 volatile E item;  
 volatile Node<E> next;

}

添加与删除都是使用CAS自旋操作

## CopyOnWriteArrayList/CopyOnWriteArraySet原理

这是一个线程安全的集合，在写操作的时候先锁住（ReentrantLock#Lock()）该写入方法，拷贝一份数组内存出来进行修改，然后再将修改的拷贝数据赋值到原先的数组上面，保证最后结果是一致的

提供get方法获取的是本身的内存数组，但是由于多线程的问题，get提供的数组可能是之前的数组，可能当前数组已经被拷贝了，但是没有被赋值，所以CopyOnWriteArrayList/CopyOnWriteArraySet不能保证数据的实时性，只能用于写少读多的场景

## Exchanger实现原理

只能用于2个线程交换数据，内部有2个空数据，当2个空数据都被A与B线程调用excharge方法的时候，此时返回交换的数据，否则单个线程等待

## DelayQueue队列

PriorityQueue + ReentrantLock 是一种线程安全的PriorityQueue，添加的时候PriorityQueue如果内存不足，那么会使用system.arrayCopy扩充50%的数组长度

## PriorityBlockingQueue

PriorityQueue + eentrantLock 是一种线程安全的PriorityQueue，添加的时候PriorityQueue如果内存不足，那么会等待，实现blocking效果

## LinkedBlockingDeque，LinkedBlockingQueue

使用链式+锁，表示线程安全的链表，前面的双向Node的节点链表，后面是单向节点的链表

## SynchronousQueue队列

每次添加只能对应一个获取，能使用的方法只能是take与put方法，同时是阻塞的，这就好像是一个快速信道，其中没有容量，相当于容量是1，每次添加只能在是空下面等待，添加的内容必须后面take拿走，这就相当于容量是1的ArrayBlockingQueue

## LinkedTransferQueue

LinkedTransferQueue是一个由链表结构组成的无界阻塞TransferQueue队列。相对于其他阻塞队列，LinkedTransferQueue多了tryTransfer和transfer方法。

LinkedTransferQueue采用一种预占模式。意思就是消费者线程取元素时，如果队列不为空，则直接取走数据，若队列为空，那就生成一个节点（节点元素为null）入队，然后消费者线程被等待在这个节点上，后面生产者线程入队时发现有一个元素为null的节点，生产者线程就不入队了，直接就将元素填充到该节点，并唤醒该节点等待的线程，被唤醒的消费者线程取走元素，从调用的方法返回。我们称这种节点操作为“匹配”方式。

ConcurrentHashMap ，ConcurrentSkipListSet，ConcurrentSkipListMap，LinkedTransferQueue，Phaser遗留

## ExecutorCompletionService 实现原理：

用于运行多个Runnable，并且将结果依次加入自定义传入的BlockQueue里面，然后可以调用BlockQueue的take取出每个runnable运行加入的结果，但是不保证每次运行runnable获取出来的结果的顺序，如果Exector使用多线程

# JAVA设计模式（23种）

## Java设计模式分类

|  |  |  |  |
| --- | --- | --- | --- |
| 范围\目的 | 创建型 | 结构型 | 行为型 |
| 对象模式 | 单例  原型  工厂模式  抽象工厂  建造者 | 代理  装饰  外观  (对象）适配器  桥接  享元  组合 | 策略  命令  职责链  状态  观察者  中介者  迭代器  访问者  备忘录  模板方法  解释器 |

创建型

单例：饿汉式，懒汉式，线程同步

原型模式：实现cloneable接口，返回对象的克隆

工厂模式：具体工厂，case相同接口不同实现类，返回统一接口

抽象工厂：在具体工厂上面再次抽象一层，产生不同工厂实现类

建造者模式：使用静态内部类，builder模式，有默认属性，属性同构造的类相同

结构型

代理：实现相同接口，传入需要代理的类（静态代理），使用Proxy（动态代理），控制/拦截代理对象的调用

装饰：实现相同接口，在调用相同方法的时候，添加一些方法调用，然后调用代理对象的相同方法，不控制/拦截代理对象的调用

外观：对一系列操作的封装，接口偏向高层调用，外部无感知

适配：类/接口/方法扩展调用，并实现新接口

桥接：内部成员/接口之间相互独立，定义抽象类设置不同实现类，并可以实现新接口

组合：树/文件夹结构，整体与部分类似

享元：内存缓存，使用创建并保存

行为型

策略：

命令：

职责链：

状态：

观察者：

中介者：

迭代器：

访问者：

备忘录：

模板方法：

解释器：