# 3年JAVA技能总结-线程篇

1. 初识sun.msic.Unsafe类:CAS操作的核心类

java不能直接访问操作系统底层，而是通过native本地方法来访问。Unsafe类提供了硬件级别的原子操作

J.U.C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，按照功能可以大致划分如下：

juc-locks 锁框架

juc-atomic 原子类框架

juc-sync 同步器框架

juc-collections 集合框架

juc-executors 执行器框架

public final class Unsafe {

//私有化构造器，外界不能访问

private Unsafe() {}

static {theUnsafe = new Unsafe();}

//提供反射获取Unsafe实例的方法

public static Unsafe getUnsafe() {

Class var0 = Reflection.getCallerClass();

if (!VM.isSystemDomainLoader(var0.getClassLoader())) {

throw new SecurityException("Unsafe");

}

return theUnsafe;

}

//获取var1的实际地址偏移值，是一个native本地方法

public native long objectFieldOffset(Field var1);

/\*\*

\* cas更新var2对应的属性值

\* @param var1 操作的对象

\* @param var2 操作的对象属性地址相对偏移值，通过var1和var2获取当前对象属性的当前值

\* @param var4 期望值

\* @param var5 新值

public final native boolean compareAndSwapObject(Object var1, long var2, Object var4, Object var5);

//通过target对象和对象中元素的偏移地址在内存中找到指定元素的实际值

//是一个native方法，底层用c语言实现

public native int getIntVolatile(Object target, long offset);

//获取相对var1对象偏移var2地址的值

public native int getInt(Object var1, long var2);

/\*\*

\*@param target 指定内存对象

\*@param offset 该内存对象中需要操作的对象偏移量

\*@param value 期望值expectValue

\*@param newValue 新值newValue

\*@return true cas更新成功 false cas更新失败

\*/

public final native boolean compareAndSwapInt(Object target, long offset, int value, int newValue);

/\*\*

\* getAndAddInt方法解析：拿到内存位置的最新值v，使用CAS尝试修将内存位置的值修改为目标值v+delta，如果修改失败

\* 则获取该内存位置的新值v，然后继续尝试，直至修改成功

\*

\*@param target 内存对象

\*@param offset 该内存对象中需要操作的对象偏移量

\*@param var4 增量

\*/

public final int getAndAddInt(Object target, long offset, int var4) {

int var5;

//while循环尝试更新offset对应的对象元素值，知道成功才会退出，期间一直占据cpu

do {

//通过对象地址和相对地址获取内存数据var5即变量"value"

//不停的while循环开销很大

var5 = this.getIntVolatile(target, offset);

} while(!this.compareAndSwapInt(target, offset, var5, var5 + var4));

return var5;

}

public native long allocateMemory(long l); //分配内存

public native long reallocateMemory(long l, long l1); //扩充内存

public native void freeMemory(long l); //释放内存

//获取给定数组中第一个元素的偏移地址

public native int arrayBaseOffset(Class arrayClass);

// 获取用户给定数组寻址的换算因子，也就是数组中元素的增量地址

public native int arrayIndexScale(Class<?> var1); //将arrayBaseOffset与arrayIndexScale配合使用，可以定位数组中每个元素在内存中的位置。

//设置obj对象中offset偏移地址对应的整型field的值为指定值。这是一个有序或者有延迟的<code>putIntVolatile</cdoe>方法，并且不保证值的改变被其他线程立即看到。只有在field被<code>volatile</code>修饰并且期望被意外修改的时候使用才有用。

public native void putOrderedInt(Object obj, long offset, int value);

//设置obj对象中offset偏移地址对应的整型field的值为指定值。支持volatile store语义

public native void putIntVolatile(Object obj, long offset, int value);

//释放被park创建的在一个线程上的阻塞

public native void unpark(Thread thread);

//阻塞一个线程直到unpark出现、线程被中断或者timeout时间到期。如果一个unpark调用已经出现了，这里只计数。timeout为0表示永不过期

public native void park(boolean isAbsolute, long time);

}

1. CAS：Compare And Swap比较和替换

<https://www.jianshu.com/p/a142350e9b7a>

<https://blog.csdn.net/v123411739/article/details/79561458>

概念：是使用一个期望值和一个变量的当前值进行比较，如果当前变量的值与我们期望的值相等，就使用一个新值替换当前变量的值。CAS算法的过程是这样：它包含三个参数 CAS（V,E,N）。V表示要更新的变量，E表示预期的值，N表示新值。仅当V值等于E值时，才会将V的值设置成N，否则什么都不做。最后CAS返回当前V的值。CAS算法需要你额外给出一个期望值，也就是你认为现在变量应该是什么样子，如果变量不是你想象的那样，那说明已经被别人修改过。你就重新读取，再次尝试修改即可。

CAS操作被封装在Unsafe类下，由于需要直接操作内存值"value"，通过unsafe获取内存值"value"的内存地址（该地址是相对于AtomicaInteger对象的内存地址，Java会将AtomicaInteger对象的内存地址 + valueOffset映射成实际内存地址）。由于是相对内存地址，所以对于任意一个AtomicaInteger对象，该地址是固定的，只需要在static初始化时获取该地址即可。

1. CAS存在的问题和解决方案：

CAS虽然很高效的解决了原子操作问题，但是CAS仍然存在三大问题：

1.ABA问题：因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A-B-A 就会变成1A-2B-3A

2.循环时间开销大：自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销（时刻a的内存值和时刻b的内存值比较，如果相等则更新，否则一直循环）

3.只能保证一个共享变量的原子操作：

解决方案：

AtomicMarkableReference<V> 维护了boolean变量表示引用变量是否被更改过

AtomicStampedReference<V> 维护了int类型的变量表示版本号

1. 初识原子类一：AtomicInteger

public class AtomicInteger {

//底层实现借助了Unsafe类

private static final Unsafe unsafe = Unsafe.getUnsafe();

//在这里具体即是指value这个字段在AtomicInteger类的内存中相对于该类首地址的偏移量

private static final long valueOffset;

//AtomicInteger类的核心：维护了一个int类型的共享变量

//对于CAS操作的内存对象必须具有可见性，即需要使用volitile进行修饰

//保存每次对value的读写都会从主存中读写，即真值

//volatile保证在多线程中value的值时可见的，任何一个线程修改了value的值，都会被立即回写到主内存中

//保证了可见性，有序性（防止指令重排），不保证原子性

private volatile int value;

//获取value变量在该对象中的地址偏移值

static {

try {

valueOffset = unsafe.objectFieldOffset(AtomicInteger.class.getDeclaredField("value"));

} catch (Exception ex) { throw new Error(ex); }

}

//使用得最多的方法：value+1

public final int getAndIncrement() {return unsafe.getAndAddInt(this, valueOffset, 1);}

//使用得最多的方法：value-1

public final int getAndDecrement() {return unsafe.getAndAddInt(this, valueOffset, -1);}

/\*\*

\* @param expect 期望值

\* @param update 更新值

\* @return true更新成功 false更新失败

\*/

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

return unsafe.compareAndSwapInt(this, valueOffset, expect, update);

}

//通过该方法对共享变量值的改变，不一定能被其他线程立即看到。也就是说以普通变量的操作方式来写变量。即无法保证

//value的可见性

public final void lazySet(int newValue) {

unsafe.putOrderedInt(this, valueOffset, newValue);

}

}

1. 初识原子类二：AtomicIntegerArray

public class AtomicIntegerArray {

private static final Unsafe unsafe = Unsafe.getUnsafe();

//返回指定类型数组的第一个元素地址相对于数组起始地址的偏移

private static final int base = unsafe.arrayBaseOffset(int[].class);

private static final int shift;

private final int[] array;

static {

//返回指定类型数组的元素占用的字节数

int scale = unsafe.arrayIndexScale(int[].class);

if ((scale & (scale - 1)) != 0)

throw new Error("data type scale not a power of two");

//Integer.numberOfLeadingZeros(scale); //将scale转换为2进制，然后从左往右数遇到第一个非1的连续0的个数

shift = 31 - Integer.numberOfLeadingZeros(scale);

}

private long checkedByteOffset(int i) {

if (i < 0 || i >= array.length)

throw new IndexOutOfBoundsException("index " + i);

return byteOffset(i);

}

//返回数组中索引为i的元素相对于数组起始位置的偏移

private static long byteOffset(int i) {

//等价于i \* scale + base <=== 2 ^ shift = scale <=== 1向左移动shift位等于scale

return ((long) i << shift) + base;

}

}

1. 初识原子类三：AtomicStampedReference

public class AtomicStampedReference<V> {

//底层借助了Unsafe类的原子操作

private static final Unsafe UNSAFE = Unsafe.getUnsafe();

//获取pair属性对象的地址相对偏移值

private static final long pairOffset = objectFieldOffset(UNSAFE, "pair", AtomicStampedReference.class);

//核心：底层维护了一个par对象

private volatile Pair<V> pair;

private static class Pair<T> {

final T reference; //实际的业务对象

final int stamp; //戳值，类似版本号，解决了ABA问题

private Pair(T reference, int stamp) {

this.reference = reference;

this.stamp = stamp;

}

static <T> Pair<T> of(T reference, int stamp) {

return new Pair<T>(reference, stamp);

}

}

public AtomicStampedReference(V initialRef, int initialStamp) {

pair = Pair.of(initialRef, initialStamp);

}

//同时返回pair对象和stamp戳值（利用数组）

public V get(int[] stampHolder) {

Pair<V> pair = this.pair;

stampHolder[0] = pair.stamp;

return pair.reference;

}

/\*\*

\* 尝试更新pair

\*

\* @param expectedReference 期望值

\* @param newReference 新值

\* @param expectedStamp 期望戳值

\* @param newStamp 新戳值

\* @return true更新成功 false更新失败

\*/

public boolean weakCompareAndSet(V expectedReference, V newReference, int expectedStamp, int newStamp) {

return compareAndSet(expectedReference, newReference, expectedStamp, newStamp);

}

/\*\*

\* 尝试cas更新pair

\*

\* @param expectedReference 期望值

\* @param newReference 新值

\* @param expectedStamp 期望戳值

\* @param newStamp 新戳值

\* @return true更新成功 false更新失败

\*/

public boolean compareAndSet(V expectedReference, V newReference, int expectedStamp, int newStamp) {

//pair的内存可见性保证了此刻获取到的pair一定是真实值

Pair<V> current = pair;

//判断是否可以更新：只有期望值=旧值并且期望戳值=旧戳值的时候才可以更新（cas原则）

boolean canUpdate = expectedReference == current.reference && expectedStamp == current.stamp;

//判断是否没有必要更新：如果新值=旧值并且新戳值=旧戳值的时候，更新毫无意义

boolean noNeedUpdate= newReference == current.reference && newStamp == current.stamp;

return canUpdate && (noNeedUpdate || casPair(current, Pair.of(newReference, newStamp)));

}

/\*\*

\* 更新pair

\* @param newReference 对象的新值

\* @param newStamp 对象的新戳值

\*/

public void set(V newReference, int newStamp) {

Pair<V> current = pair;

if (newReference != current.reference || newStamp != current.stamp)

this.pair = Pair.of(newReference, newStamp);

}

/\*\*

\* 尝试只更新戳值

\*

\* @param expectedReference 期望值

\* @param newStamp 新戳值

\*/

public boolean attemptStamp(V expectedReference, int newStamp) {

Pair<V> current = pair;

return expectedReference == current.reference &&

(newStamp == current.stamp ||

casPair(current, Pair.of(expectedReference, newStamp)));

}

/\*\*

\* 尝试cas更新pair对象

\* @param cmp 对象期望值

\* @param val 对象新值

\*/

private boolean casPair(Pair<V> cmp, Pair<V> val) {

return UNSAFE.compareAndSwapObject(this, pairOffset, cmp, val);

}

static long objectFieldOffset(Unsafe UNSAFE, String field, Class<?> klazz) {

try {

return UNSAFE.objectFieldOffset(klazz.getDeclaredField(field));

} catch (NoSuchFieldException e) {

NoSuchFieldError error = new NoSuchFieldError(field);

error.initCause(e);

throw error;

}

}

}

1. 初识原子类四：AtomicMarkableReference

public class AtomicMarkableReference<V> {

private static final Unsafe UNSAFE = Unsafe.getUnsafe();

private static final long pairOffset = objectFieldOffset(UNSAFE, "pair", AtomicMarkableReference.class);

private volatile Pair<V> pair;

private static class Pair<T> {

final T reference;

final boolean mark;

private Pair(T reference, boolean mark) {

this.reference = reference;

this.mark = mark;

}

static <T> Pair<T> of(T reference, boolean mark) {

return new Pair<T>(reference, mark);

}

}

public V get(boolean[] markHolder) {

Pair<V> pair = this.pair;

markHolder[0] = pair.mark;

return pair.reference;

}

/\*\*

\*

\* @param expectedReference the expected value of the reference

\* @param newReference the new value for the reference

\* @param expectedMark the expected value of the mark

\* @param newMark the new value for the mark

\* @return {@code true} if successful

\*/

public boolean weakCompareAndSet(V expectedReference, V newReference,boolean expectedMark, boolean newMark) {

return compareAndSet(expectedReference, newReference, expectedMark, newMark);

}

/\*\*

\* @param expectedReference the expected value of the reference

\* @param newReference the new value for the reference

\* @param expectedMark the expected value of the mark

\* @param newMark the new value for the mark

\* @return {@code true} if successful

\*/

public boolean compareAndSet(V expectedReference, V newReference, boolean expectedMark, boolean newMark) {

Pair<V> current = pair;

return

expectedReference == current.reference &&

expectedMark == current.mark &&

((newReference == current.reference &&

newMark == current.mark) ||

casPair(current, Pair.of(newReference, newMark)));

}

/\*\*

\* Unconditionally sets the value of both the reference and mark.

\*

\* @param newReference the new value for the reference

\* @param newMark the new value for the mark

\*/

public void set(V newReference, boolean newMark) {

Pair<V> current = pair;

if (newReference != current.reference || newMark != current.mark)

this.pair = Pair.of(newReference, newMark);

}

/\*\*

\*

\* @param expectedReference the expected value of the reference

\* @param newMark the new value for the mark

\* @return {@code true} if successful

\*/

public boolean attemptMark(V expectedReference, boolean newMark) {

Pair<V> current = pair;

return

expectedReference == current.reference &&

(newMark == current.mark ||

casPair(current, Pair.of(expectedReference, newMark)));

}

private boolean casPair(Pair<V> cmp, Pair<V> val) {

return UNSAFE.compareAndSwapObject(this, pairOffset, cmp, val);

}

static long objectFieldOffset(sun.misc.Unsafe UNSAFE,String field, Class<?> klazz) {

try {

return UNSAFE.objectFieldOffset(klazz.getDeclaredField(field));

} catch (NoSuchFieldException e) {

NoSuchFieldError error = new NoSuchFieldError(field);

error.initCause(e);

throw error;

}

}

}

1. 初识原子类五：AtomicLong

public class AtomicLong {

private static final Unsafe unsafe = Unsafe.getUnsafe();

private static final long valueOffset;//value属性的相对地址偏移量

private volatile long value;

//获取value属性的相对地址偏移量

static {

try {

valueOffset = unsafe.objectFieldOffset(AtomicLong.class.getDeclaredField("value"));

} catch (Exception ex) { throw new Error(ex); }

}

public final void lazySet(long newValue) {

unsafe.putOrderedLong(this, valueOffset, newValue);

}

public final long getAndSet(long newValue) {

return unsafe.getAndSetLong(this, valueOffset, newValue);

}

public final boolean compareAndSet(long expect, long update) {

return unsafe.compareAndSwapLong(this, valueOffset, expect, update);

}

public final boolean weakCompareAndSet(long expect, long update) {

return unsafe.compareAndSwapLong(this, valueOffset, expect, update);

}

public final long getAndIncrement() {

return unsafe.getAndAddLong(this, valueOffset, 1L);

}

public final long getAndDecrement() {return unsafe.getAndAddLong(this, valueOffset, -1L);}

public final long incrementAndGet() {return unsafe.getAndAddLong(this, valueOffset, 1L) + 1L;}

public final long addAndGet(long delta) {return unsafe.getAndAddLong(this, valueOffset, delta) + delta;}

public final long getAndUpdate(LongUnaryOperator updateFunction) {

long prev, next;

do {

prev = get();

next = updateFunction.applyAsLong(prev);

} while (!compareAndSet(prev, next));

return prev;

}

public final long getAndAccumulate(long x, LongBinaryOperator accumulatorFunction) {

long prev, next;

do {

prev = get();

next = accumulatorFunction.applyAsLong(prev, x);

} while (!compareAndSet(prev, next));

return prev;

}

public final long accumulateAndGet(long x, LongBinaryOperator accumulatorFunction) {

long prev, next;

do {

prev = get();

next = accumulatorFunction.applyAsLong(prev, x);

} while (!compareAndSet(prev, next));

return next;

}

}

1. 初识原子类六：LongAdder

/\*\*

\* 在并发量较低的环境下，线程冲突的概率比较小，自旋的次数不会很多。

\* 但是，高并发环境下，N个线程同时进行自旋操作，会出现大量失败并不断自旋的情况，此时AtomicLong的自旋会成为瓶颈。

\* 这就是LongAdder引入的初衷——解决高并发环境下AtomicLong的自旋瓶颈问题。特别是写多的场景

\*

\* LongAdder的原理是，在最初无竞争时，只更新base的值，当有多线程竞争时通过分段的思想，让不同的线程更新不同的段，

\* 最后把这些段相加就得到了完整的LongAdder存储的值。

\*/

public class LongAdder extends Striped64{

private static final sun.misc.Unsafe UNSAFE;

private static final long BASE; //base变量的相对地址偏移

private static final long CELLSBUSY; //cellsBusy变量的相对地址偏移

private static final long PROBE; //threadLocalRandomProbe变量的相对地址偏移

//cpu可用核数，决定了槽数组cells的大小

static final int NCPU = Runtime.getRuntime().availableProcessors();

//底层使用数组存放，2倍扩容

transient volatile Cell[] cells; // cells数组，存储各个段的值

//基数，在两种情况下会使用：

//1.没有遇到并发竞争时，直接使用base累加数值

//2.初始化cells数组时，必须要保证cell数组只被初始化一次（即只有一个线程可以对cells初始化），其他竞争失败的线//程会将数值累加base上

transient volatile long base; // 最初无竞争时使用的，也算一个特殊的段

//锁标识：

//cells初始化或扩容时，通过cas操作将此标识设置为1-加锁状态，初始化或扩容完成后，将此标识设置为0-无锁状态

// 标记当前是否有线程在创建或扩容cells，或者在创建Cell；通过CAS更新该值，相当于是一个锁

transient volatile int cellsBusy;

public void increment() {add(1L);}

public void decrement() {add(-1L);}

public long sum() { //求和，最终值

Cell[] as = cells; Cell a;

long sum = base;

if (as != null) {

for (int i = 0; i < as.length; ++i) {

if ((a = as[i]) != null)

sum += a.value;

}

}

return sum;

}

public void add(long x) {  
 Cell[] as; long b, v; int m; Cell a;  
 if ((as = cells) != null //cells为空，需要进行cas；不为空表示已经出现了竞争，已经创建了cells

|| !casBase(b = base, b + x)) { //如果cas失败返回false，说明其他线程抢先一步修改了base，正在出现竞争  
 //true表示当前竞争还不激烈

//false表示竞争激烈，多个线程hash到同一个cell，可能要扩容

boolean uncontended = true;  
 if (as == null || (m = as.length - 1) < 0 ||

//一个线程对应一个probe，它是通过随机数生成的一个值，对于一个确定的线程这个值是固定的，除非刻意修改它

//通过prope和size-1做位与运算散列到cells对应的索引位置，如果为空说明该线程所在cells中的cell为空，应初始化一个Cell

//可能会有多个线程的cells索引位置相同，存在竞争  
 (a = as[getProbe() & m]) == null ||

//如果执行到这里，说明当前线程对应的cell不为空，尝试更新当前cell中的value。

//如果cell.cas成功， 也就是uncontended=true表示竞争不激烈；

//如果失败了， 也就是uncontended=false，说明这一个cell上一定有多个线程在同步修改cell，竞争激烈  
 !(uncontended = a.cas(v = a.value, v + x)))

longAccumulate(x, null, uncontended);  
 }  
 }

/\*\*

\*

\* @param x 增量

\* @param fn 回调函数

\* @param wasUncontended

\*/

final void longAccumulate(long x, LongBinaryOperator fn, boolean wasUncontended) {  
 int h;

//给当前线程分配hash值  
 if ((h = getProbe()) == 0) {//如果当前线程的prope为0的话  
 ThreadLocalRandom.current(); //强制初始化，防止prope还是为0

h = getProbe(); //重新获取prope  
 wasUncontended = true; //都未初始化，肯定还不存在竞争激烈  
 }  
 boolean collide = false; // True if last slot nonempty  
 for (;;) {  
 Cell[] as; Cell a; int n; long v;

//case1：cells已经被初始化过了  
 if ((as = cells) != null && (n = as.length) > 0) {  
 if ((a = as[(n - 1) & h]) == null) { //如果该线程对应的cell还没有被初始化

//如果还没有上锁：也就是当前无其它线程在创建或扩容cells，也没有线程在创建cell  
 if (cellsBusy == 0) {   
 Cell r = new Cell(x);

// 再次检测cellsBusy，并尝试更新它为1，相当于当前线程加锁  
 if (cellsBusy == 0 && casCellsBusy()) {   
 boolean created = false;  
 try {   
 Cell[] rs; int m, j;

// 重新获取cells，并找到当前线程hash到cells数组中的位置

// 这里一定要重新获取cells，因为rs并不在锁定范围内

// 有可能已经扩容了，这里要重新获取  
 if ((rs = cells) != null &&  
 (m = rs.length) > 0 &&  
 rs[j = (m - 1) & h] == null) {  
 rs[j] = r; // 把上面新建的Cell放在cells的j位置处  
 created = true;  
 }  
 } finally {  
 cellsBusy = 0;// 相当于释放锁  
 }  
 if (created) // 创建成功了就返回，值已经放在新建的Cell里面了  
 break;  
 continue; // Slot is now non-empty  
 }  
 }  
 collide = false; // 标记当前未出现冲突  
 }

// 当前线程所在的Cell不为空，且更新失败了

// 这里简单地设为true，相当于简单地自旋一次

// 通过下面的语句修改线程的probe再重新尝试

else if (!wasUncontended)   
 wasUncontended = true;

// 再次尝试CAS更新当前线程所在Cell的值，如果成功了就返回

else if (a.cas(v = a.value, ((fn == null) ? v + x : fn.applyAsLong(v, x))))  
 break;

// 如果cells数组的长度达到了CPU核心数，或者cells扩容了

// 设置collide为false并通过下面的语句修改线程的probe再重新尝试

else if (n >= NCPU || cells != as)  
 collide = false;

// 上上个elseif都更新失败了，且上个条件不成立，说明出现冲突了  
 else if (!collide)  
 collide = true;

// 明确出现冲突了，尝试占有锁，并扩容  
 else if (cellsBusy == 0 && casCellsBusy()) {  
 try {  
 // 检查是否有其它线程已经扩容过了

if (cells == as) {  
 Cell[] rs = new Cell[n << 1]; //2倍扩容  
 for (int i = 0; i < n; ++i)  
 rs[i] = as[i];  
 cells = rs;  
 }  
 } finally {  
 cellsBusy = 0;//释放锁  
 }  
 collide = false; //已解决冲突  
 continue;// 使用扩容的数组再次尝试  
 }

// 更新失败或者达到了CPU核心数，重新生成probe，并重试  
 h = advanceProbe(h);  
 }

//case2：cells没有加锁并且没有被初始化，则尝试对它加锁，并初始化cells数组  
 else if (cellsBusy == 0 && cells == as && casCellsBusy()) {

//成功获取锁后，准备初始化  
 boolean init = false;  
 try { // Initialize table  
 if (cells == as) {  
 Cell[] rs = new Cell[2];  
 rs[h & 1] = new Cell(x);  
 cells = rs;  
 init = true;  
 }  
 } finally {  
 cellsBusy = 0; //释放锁  
 }  
 if (init)  
 break;  
 }

//如果在初始化过程中，另一个线程ThreadB也进入了longAccumulate方法，就会进入分支

//case3：cells正在进行初始化，则尝试直接在base上进行累加操作  
 else if (casBase(v = base, ((fn == null) ? v + x : fn.applyAsLong(v, x))))  
 break;

}  
}

final boolean casCellsBusy() { //尝试cas更新cellsBusy的值从0到1，类似上锁的概念

return UNSAFE.compareAndSwapInt(this, CELLSBUSY, 0, 1);

}

final boolean casBase(long cmp, long val) { //尝试cas更新base的值由cmp改为val

return UNSAFE.compareAndSwapLong(this, BASE, cmp, val);

}

static final int getProbe() {//获取当前线程的threadLocalRandomProbe值

return UNSAFE.getInt(Thread.currentThread(), PROBE);

}

static {

try {

UNSAFE = sun.misc.Unsafe.getUnsafe();

BASE = UNSAFE.objectFieldOffset(Striped64.class.getDeclaredField("base"));

CELLSBUSY = UNSAFE.objectFieldOffset(Striped64.class.getDeclaredField("cellsBusy"));

PROBE = UNSAFE.objectFieldOffset(Thread.class.getDeclaredField("threadLocalRandomProbe"));

} catch (Exception e) {

throw new Error(e);

}

}

}

1. 线程和进程

进程：正在运行的程序，它是系统资源调度的独立单位，并且一个进程可以执行多个任务，而线程就是程序执行的任务，它是程序使用CPU的基本单位，因此也可以说线程是依赖于进程的

线程：进程中的一个执行单元，负责当前进程中程序的执行，一个进程中至少有一个线程。同一个进程中的多个线程之间可以并发的执行

1. 并发、并行和串行的区别

并发：同一时刻，只有一条命令是在处理器上执行的，但多个进程命令被快速轮换执行，使得在宏观上具有多个进程同时执行的效果

并行：同一时刻，多个处理器或多核处理器同时处理多个任务

串行：一个一个按顺序执行

对于一个CPU而言：只能在某一时间点执行一个程序

多进程的并发策略有：共用式的多任务操作策略（WIN3.1和Mac OS9），现在操作系统大多采用效率更高的抢占式多任务操作策略（Windows NT、Windows 2000以及UNIX/Linux）等操作系统。

多线程扩展了多进程了概念，使得同一个进程可以同时并发处理多个任务。

1. volatile关键字

volatile：保证共享变量的在内存的可见性，有序性（防止指令重排，但在一定程度上降低了代码执行效率），不保证原子性；作用：在了解java内存模型后，才能更加了解volatile在JMM中的作用，volatile在JMM中为了保证内存的可见性，即是线程之间操作共享变量的可见性

volatile写和读的内存语义

volatile 写的内存语义：

当写一个volatile修饰的共享变量时，JMM会把该线程的本地内存的共享变量副本值刷新到主内存中；

volatile 读的内存语义：

当读一个volatile修饰的共享变量时，JMM会将该线程的本地内存的共享变量副本置为无效，要求线程重新去主内存中获取最新的值。

java内存模型控制与volatile冲突吗？什么区别？

不冲突！java内存模型控制线程工作内存与主内存之间共享变量会同步，即线程从主内存中读一份副本到工作内存，又刷新到主内存，那怎么还需要 volatile来保证可见性，不是JMM自己能控制吗，一般情况下JMM可以控制 2份内存数据一致性，但是在多线程并发环境下，虽然最终线程工作内存中的共享变量会同步到主内存，但这需要时间和触发条件，线程之间同时操作共享变量协作时，就需要保证每次都能获取到主内存的最新数据，保证看到的工作变量是最后一次修改后的值，这个JMM没法控制保证，这就需要volatile或者后文要讲的 synchronized和锁的同步机制来实现了。

原子性：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。volatile是无法保证复合操作的原子性。要想在多线程环境下保证原子性，则可以通过锁、synchronized来确保。

可见性：指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。当一个变量被volatile修饰后，表示着线程本地内存无效，当一个线程修改共享变量后他会立即被更新到主内存中，当其他线程读取共享变量时，它会直接从主内存中读取。当然，synchronize和锁都可以保证可见性

有序性：程序执行的顺序按照代码的先后顺序执行

happens-before：指令重排序相关

1. synchronized关键字

在java中，每一个对象有且仅有一个同步锁。这也意味着，同步锁是依赖于对象而存在。

当我们调用某对象的synchronized方法时，就获取了该对象的同步锁。例如，synchronized(obj)就获取了“obj这个对象”的同步锁。

若实例被锁，则该实例的所有同步方法全部被锁

若类被锁，则该类的所有同步方法全部被锁

<https://blog.csdn.net/meetings/article/details/78530955>

<https://blog.csdn.net/cs408/article/details/48930803>

synchronized：不能被继承，实例锁，对类的当前实例进行加锁，防止其他线程同时访问该实例的所有synchronized块

static synchronized：类锁，控制类的所有实例的并发访问，限制多线程中该类的所有实例同时访问JVM中该类所对应的代码块。类的所有实例共一把锁，监视器上锁的对象为这个类对象，而不是一个具体的实例对象，就是所有该类的实例访问这个方法都会进行加锁

synchronized方法和static synchronized方法区别

synchronized方法锁的是实例对象，static synchronized方法锁的是类对象

   synchronized是对类的当前实例（当前对象）进行加锁，防止其他线程同时访问该类的该实例的所有synchronized块，注意这里是“类的当前实例”，类的两个不同实例就没有这种约束了。

   static synchronized恰好就是要控制类的所有实例的并发访问，static synchronized是限制多线程中该类的所有实例同时访问jvm中该类所对应的代码块

   实际上，在类中如果某方法或某代码块中有 synchronized，那么在生成一个该类实例后，该实例也就有一个监视块，防止线程并发访问该实例的synchronized保护块，而static synchronized则是所有该类的所有实例公用得一个监视块

1. synchronized的实现原理

<https://segmentfault.com/a/1190000020247956?utm_source=tag-newest>

1. JAVA创建线程的几种方式

继承Thread类重写run方法：java使用Thread类代表线程，所有的线程对象都必须是Thread或者其子类的实例，每个线程的作用是完成一定任务，实际上是就是执行一段程序流（一段顺序执行的代码）

继承Thread类创建线类

1：定义Thread类的子类 并重写该类的Run方法 该run方法的方法体就代表了线程需要完成的任务

2：创建Thread类的实例，即创建了线程对象

3：调用线程的start方法来启动线程

实现Runnable接口：使用Runnable接口创建线程类

1：定义Runnable接口的实现类，并重写它的Run方法，run方法同样是该线程的执行体！

2：创建Runnable实现类的实例，并将此实例作为Thread的target创建一个Thread对象，该Thread对象才是真正的线程对象！

3：调用start方法启动该线程

Runnable对象仅作为Thread对象的target，Runnable实现类里包含的run（）方法仅作为线程执行体。而实际的线程的对象依旧是Thread实例，只是线程实例负责执行其target的run（）方法

实现Callable接口：可以返回执行结果，是个泛型，和Future、FutureTask配合可以用来获取异步执行的结果

1. 对象锁

每个java对象都有一个锁对象.而且只有一把钥匙.

如何创建锁对象：

可以使用this关键字作为锁对象。

Java中的每个对象都有一个内置锁，只有当对象具有同步方法代码时，内置锁才会起作用，当进入一个同步的非静态方法时，就会自动获得与类的当前实例（this）相关的锁，该类的代码就是正在执行的代码。获得一个对象的锁也成为获取锁、锁定对象也可以称之为监视器来指我们正在获取的锁对象。

因为一个对象只有一个锁，所有如果一个线程获得了这个锁，其他线程就不能获得了，直到这个线程释放（或者返回）锁。也就是说在锁释放之前，任何其他线程都不能进入同步代码（不可以进入该对象的任何同步方法）。释放锁指的是持有该锁的线程退出同步方法，此时，其他线程可以进入该对象上的同步方法。

1：只能同步方法（代码块），不能同步变量或者类

2：每个对象只有一个锁

3：不必同步类中的所有方法，类可以同时具有同步方法和非同步方法

4：如果两个线程要执行一个类中的一个同步方法，并且他们使用的是了类的同一个实例（对象）来调用方法，那么一次只有一个线程能够执行该方法，另一个线程需要等待，直到第一个线程完成方法调用，总结就是：一个线程获得了对象的锁，其他线程不可以进入该对象的同步方法。

5：如果类同时具有同步方法和非同步方法，那么多个线程仍然可以访问该类的非同步方法。

同步会影响性能（甚至死锁），优先考虑同步代码块。

6：如果线程进入sleep（） 睡眠状态，该线程会继续持有锁，不会释放。

1. JAVA线程的分类

用户线程（比如main）和daemon

　　A.用户线程: 用户线程可以简单的理解为用户定义的线程,当然包括main线程(以前我错误的认为main线程也是一个daemon线程,但是慢慢的发现原来main线程不是,因为如果我再main线程中创建一个用户线程,并且打出日志,我们会发现这样一个问题,main线程运行结束了,但是我们的线程任然在运行).

B.daemon线程:运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。daemon线程是为我们创建的用户线程提供服务的线程,比如说jvm的GC等等,这样的线程有一个非常明显的特征: 当用户线程运行结束的时候,daemon线程将会自动退出.

1. Thread.sleep(long timeout);或者TimeUnit.MINUTES.sleep();响应中断

public static native void sleep(long millis) throws InterruptedException;

Thread类中的静态native方法，响应中断，使当前正在运行的线程休眠,进入TIMED\_WAITING状态，并释放CPU资源,但不会释放对象锁，当睡眠的时间结束或者中断才会重新进入就绪状态，因此sleep()方法不能保证该线程睡眠到期后就开始执行。

1. Thread.yield();不响应中断

public static native void yield();

使当前正在运行的线程让步，重新进入就绪的线程池中，让系统的线程调度器重新调度器重新调度一次，完全可能出现这样的情况：当某个线程调用yield()方法之后，线程调度器又将其调度出来重新进入到运行状态执行

当前线程放弃获取的CPU时间片，但不释放锁资源，由运行状态变为就绪状态

1. thread.join();响应中断

public final void join() throws InterruptedException {join(0);}

public final synchronized void join(long millis) throws InterruptedException {

long base = System.currentTimeMillis();

long now = 0;

if (millis < 0) {

throw new IllegalArgumentException("timeout value is negative");

}

if (millis == 0) {

while (isAlive()) { //只要子线程是活的，主线程就不停的等待

wait(0); //让当前在cpu上运行的线程进入无限期等待状态

}

} else {

while (isAlive()) {

long delay = millis - now;

if (delay <= 0) {

break;

}

wait(delay);

now = System.currentTimeMillis() - base;

}

}

}

主线程会等待子线程结束之后才能继续运行，通常用于在main()主线程内，等待其它线程完成再结束main()主线程

thread.join(); //当前线程里调用其它线程thread的join方法，当前线程等待thread线程执行完成才会执行，当前线程会进入WAITING无限期等待状态。

thread.join(timeout); //当前线程等待thread线程执行完成才会执行，最多等待time时间，当前线程会进入TIME\_WAITING

wait()的作用是让“当前线程”等待，而这里的“当前线程”是指当前在CPU上运行的线程。所以，虽然是调用子线程的wait()方法，但是它是通过“主线程”去调用的；所以，休眠的是主线程，而不是“子线程”！

1. obj.wait();响应中断

public final native void wait(long timeout) throws InterruptedException; //指定等待时间

public final void wait() throws InterruptedException {wait(0);} //无限期等待

Object类的方法，只能在同步方法或同步代码块中使用，它使得当前正持有该对象的锁的线程等待（即暂停），并释放对象锁，以便其它线程能够获取该对象的锁，当前线程会进入WAITING

obj.wait();线程进入WAITING等待状态并释放CPU资源和对象锁

obj.wait(timeout);指定等待时间，线程进入TIME\_WAITING超时等待状态并释放CPU资源和对象锁。如果指定时间内没有notify或者notifyAll唤醒它，则时间到了后自动唤醒，如果时间还没到但是有notify或者notifyAll唤醒它，则会提前唤醒

一个线程T被唤醒可能有一下四种情况，但是唤醒之后线程T需要重新去抢临界区的锁才能把代码真正执行下去：

其它的线程调用 obj.notify()，且当前线程 T，正好是被选中唤醒的。

其它的线程调用 obj.notifyAll()唤醒所有obj上的等待线程。

其它线程中断T。

指定的等待时间（timeout）超时，（时间精度会有些误差）。

1. thread.interrupt();

中断：Java中断机制是一种协作机制，通过中断并不能直接终止另一个线程，而需要被中断的线程自己处理中断

interrupt()中断：将线程的中断标志位置位

如果线程sleep()、wait()、join()等处于阻塞状态，那么线程会定时检查中断状态位如果发现中断状态位为true，则会在这些阻塞方法调用处抛出InterruptedException异常，并且在抛出异常后立即将线程的中断状态位清除，即重新设置为false。抛出异常是为了线程从阻塞状态醒过来，并在结束线程前让程序员有足够的时间来处理中断请求。

如果线程正在运行、争用synchronized、lock()等，那么是不可中断的，他们会忽略。

可以通过以下三种方式来判断中断：

1)isInterrupted()此方法只会读取线程的中断标志位，并不会重置。

2)interrupted()此方法读取线程的中断标志位，并会重置。

3)throw InterruptException抛出该异常的同时，会重置中断标志位。

thread.interrupt(); //实例方法 如果正在运行wait()，sleep()，join()这三个方法阻塞了线程，那么将会使得线程抛出InterruptedException异常，这是一个中断阻塞的过程。如果是其它的正在运行的状态，那么将不会有任何影响，也不会中断线程，或者抛出异常，只会会打上一个中断线程的标志，是否中断线程，将由程序控制。

boolean interrupted = Thread.interrupted(); //静态方法，返回当前线程是否有中断标志并清除当前线程的中断标志：就是如果当前线程已中断，第一次调用这个方法的返回值是true，第二次调用这个方法的返回值为false，因为调用方法时，会清除它的中断标志

boolean interrupted = thread.isInterrupted(); //实例方法 返回该线程是否有中断标志

中断的使用场景有以下几个：

点击某个桌面应用中的取消按钮时；

某个操作超过了一定的执行时间限制需要中止时；

多个线程做相同的事情，只要一个线程成功其它线程都可以取消时；

一组线程中的一个或多个出现错误导致整组都无法继续时；

当一个应用或服务需要停止时。

1. 锁池和等待池

锁池：假设线程A已经拥有对象锁，线程B、C想要获取锁就会被阻塞，进入一个地方去等待锁的等待，这个地方就是该对象的锁池；

等待池：假设线程A调用某个对象的wait方法，线程A就会释放该对象锁，同时线程A进入该对象的等待池中，进入等待池中的线程不会去竞争该对象的锁。

1. 单例模式下的双重校验锁

public class Singleton {

//volatile：内存可见性+防止指令重排

private static volatile Singleton instance = null;

private Singleton() {

}

public static Singleton getInstance() {

if (instance == null) { //如果还没有被实例化，再上锁

synchronized (Singleton.class) {

if (instance == null) {

instance = new Singleton();

}

}

}

return instance;

}

}

private static volatile Singleton singleton = null;//因为volatile关键字可以防止jvm指令重排优化，

意思就是 singleton = new Singleton() 可以理解为三个阶段：1为singleton分配空间；2初始化singleton；3将创建的singleton实例指向被分配的内存空间。

因为JVM具有指令重排的特性，执行顺序有可能变成 1-3-2。 指令重排在单线程下不会出现问题，但是在多线程下会导致一个线程获得一个未初始化的实例。例如：线程r1执行了1和3，此时r2调用 getInstance() 后发现 singleton 不为空，因此返回 singleton，但是此时的singleton还没有被初始化。

     使用volatile会禁止JVM指令重排，从而保证在多线程下也能正常执行。

1. wait和sleep的区别

类的不同：sleep()来自Thread，wait()来自Object

释放锁：sleep() 不释放锁；wait() 释放锁，wait通常用于线程时交互，sleep通常被用于暂停执行

用法不同：sleep() 时间到会自动恢复；wait() 可以使用 notify()/notifyAll()直接唤醒

1. sleep和yield的区别

sleep方法暂停当前线程后，会进入阻塞状态，只有当睡眠时间到了，才会转入就绪状态。而yield方法调用后 ，是直接进入就绪状态，所以有可能刚进入就绪状态，又被调度到运行状态。

sleep方法声明抛出了InterruptedException，所以调用sleep方法的时候要捕获该异常，或者显示声明抛出该异常。而yield方法则没有声明抛出任务异常。

sleep方法比yield方法有更好的可移植性，通常不要依靠yield方法来控制并发线程的执行。

1. notify()和notifyAll()有什么区别？

notifyAll()会唤醒所有的线程，notify()之后唤醒一个线程。notifyAll() 调用后，会将全部线程由等待池移到锁池，然后参与锁的竞争，竞争成功则继续执行，如果不成功则留在锁池等待锁被释放后再次参与竞争

1、notify只会随机选取一个处于该对象锁等待池中的线程进入锁池去竞争获取锁的机会，具体唤醒哪一个线程由虚拟机控制；

2、notifyAll会让所有处于该对象锁等待池的线程全部进入锁池去竞争获取锁的机会；

1. 线程优先级priority属性

java 中的线程优先级的范围是1～10，默认的优先级是5，10最高

有时间片轮循机制时：“高优先级线程”被分配CPU的概率高于“低优先级线程”。根据时间片轮循调度，所以能够并发执行。无论是是级别相同还是不同，线程调用都不会绝对按照优先级执行，每次执行结果都不一样，调度算法无规律可循，所以线程之间不能有先后依赖关系。

无时间片轮循机制时：高级别的线程优先执行，如果低级别的线程正在运行时，有高级别线程可运行状态，则会执行完低级别线程，再去执行高级别线程。如果低级别线程处于等待、睡眠、阻塞状态，或者调用yield()函数让当前运行线程回到可运行状态，以允许具有相同优先级或者高级别的其他线程获得运行机会。因此，使用yield()的目的是让相同优先级的线程之间能适当的轮转执行。但是，实际中无法保证yield()达到让步目的，因为让步的线程还有可能被线程调度程序再次选中。结论：yield()从未导致线程转到等待/睡眠/阻塞状态。在大多数情况下，yield()将导致线程从运行状态转到可运行状态，但有可能没有效果。

1. 线程的状态

NEW（新建）：新创建了一个线程对象，但还没有调用start()方法，例如Thread thread = new Thread();

RUNNABLE（可运行）：Java线程中将就绪（ready）和运行中（running）两种状态笼统的称为“运行”，包括了操作系统线程状态 中的Running和Ready，也就是处于此状态的线程可能正在运行，也可能正在等待系统资源，如等待CPU为它分配时间片，如等 待网络IO读取数据获取CPU的使用权，此时处于就绪状态（ready）。就绪状态的线程在获得CPU时间片后变为运行中状态 （running）

BLOCKED（阻塞）：表示线程阻塞于锁。当一个线程要进入synchronized语句块/方法时，如果没有获取到锁（waiting to enter monitor），会变成BLOCKED，直到另一个线程走完临界区或发生了相应锁对象的wait()操作后，它才有机会去争夺进入临界区 的权利。处于BLOCKED状态的线程，即使对其调用 thread.interrupt()也无法改变其阻塞状态，因为interrupt()方法只是 设置线程的中断状态，即做一个标记，不能唤醒处于阻塞状态的线程。ReentrantLock.lock()操作后进入的是WAITING状态， 其内部调用的是LockSupport.park()方法

WAITING（无限期等待）：处于这种状态的线程不会被CPU分配时间片，它们要等待被显式地唤醒，否则会处于无限期等待的状态

这种状态通常是指一个线程拥有对象锁后进入到相应的代码区域后，调用相应的“锁对象”的wait()方法操作后该线程会进入 WAITING状态

调用obj的wait(), notify()方法前，必须获得obj锁，也就是必须写在synchronized(obj) 代码段内

以下方法会让线程陷入无限期等待状态：

obj.wait();

thread.join();

LockSupport.park(); -- 驻留

注意：

LockSupport.park(Object blocker) 会挂起当前线程，参数blocker是用于设置当前线程的“volatile Object parkBlocker 成员变量”

parkBlocker 是用于记录线程是被谁阻塞的，可以通过LockSupport.getBlocker()获取到阻塞的对象，用于监控和分析线程用的。

TIMED\_WAITING（超时等待）：处于这种状态的线程也不会被分配CPU执行时间，不过无需等待被其它线程显示的唤醒，在一定时间 之后它们会由系统自动的唤醒

以下方法会让线程进入TIMED\_WAITING限期等待状态：

thread.sleep(timeout);

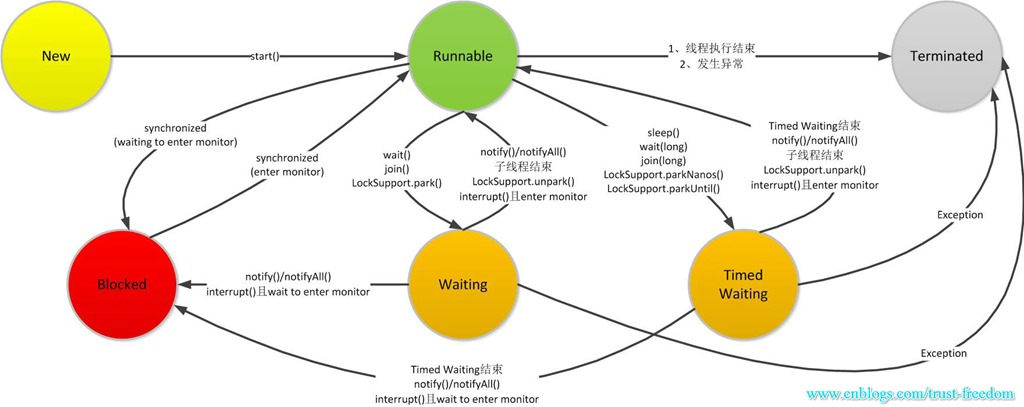
obj.wait(timeout);

thread.join(timeout);

LockSupport.parkUntil(deadline); -- 驻留

LockSupport.parkNanos(timeout);

TERMINATED（结束）：RUNNABLE线程run方法走完或者发生异常时



1. 阻塞与等待的区别

阻塞：当一个线程试图获取对象锁（非java.util.concurrent库中的锁，即synchronized），而该锁被其他线程持有，则该线程进入阻塞状态。它的特点是使用简单，由JVM调度器来决定唤醒自己，而不需要由另一个线程来显式唤醒自己，不响应中断。

等待：当一个线程等待另一个线程通知调度器一个条件时，该线程进入等待状态，是在等待一段时间或者唤醒动作的发生，进入“等待”状态是主动的。如主动调用Object.wait()，如无法获取到ReentraantLock，主动调用LockSupport.park()，如主线程主动调用 subThread.join()，让主线程等待子线程执行完毕再执行。离开“等待”状态是因为其它线程发生了唤醒动作或者到达了等待时间

1. LockSupport.park()和unpark()

public class LockSupport {

private LockSupport() {}

private static final Unsafe UNSAFE;

//可以响应中断，但是不会抛出InterruptedException

public static void park() { //获取许可，因为许可默认是被占用的，调用park()时获取不到许可，线程进入阻塞状态

UNSAFE.park(false, 0L);

}

public static void unpark(Thread thread) { //释放许可

if (thread != null) UNSAFE.unpark(thread);

}

}

1. ThreadLocal类

ThreadLocal是Java里一种特殊的变量。每个线程都有一个ThreadLocal就是每个线程都拥有了自己独立的一个变量，竞争条件被彻底消除了。如果为每个线程提供一个自己独有的变量拷贝，将大大提高效率。首先，通过复用减少了代价高昂的对象的创建个数。其次，你在没有使用高代价的同步或者不变性的情况下获得了线程安全。

1. AQS：AbstractQueuedSynchronizer抽象队列同步器的简称

AQS定义了一套多线程访问共享资源的同步器框架，许多java同步类实现都依赖于它

AQS维护了一个volatile int state（代表共享资源）和一个FIFO线程等待队列（多线程争用资源被阻塞时会进入此队列）

AQS定义两种资源共享方式：Exclusive（独占，只有一个线程能执行，如ReentrantLock）和Share（共享，多个线程可同时执行，如Semaphore/CountDownLatch）

Lock

CountDownLatch类的使用

ReentrantLock

Semaphore

1. CountDownLatch底层源码解析及使用

public class CountDownLatch {

private static final class Sync extends AbstractQueuedSynchronizer {

Sync(int count) {setState(count);}

//会被初始化为head = new Node();跟thread无关系，仅作为一个head标识

private transient volatile Node head;

//队尾节点，跟等待队列中最后一个thread挂钩

private transient volatile Node tail;

//如果state=0代表获取共享锁成功，否则失败，多个线程共用一把锁是为SHARED

private volatile int state;

//如果state!=0，就让线程加入等待队列；否则

public void await() throws InterruptedException {

sync.acquireSharedInterruptibly(1);

}

//state-1，如果state=0就唤醒所有等待的线程

public void countDown() {

sync.releaseShared(1);

}

//countDown实际调用，arg=1

//共享模式下线程释放共享资源的顶层入口，它会释放指定量的资源，如果成功释放且允许唤醒等待线程，它会唤醒等待队列里的其他线程来获取资源

public final boolean releaseShared(int arg) {  
 if (tryReleaseShared(arg)) { //尝试释放共享锁，如果state=0代表成功，准备唤醒线程了  
 doReleaseShared();  
 return true;  
 }  
 return false;  
 }

//尝试获取共享锁state，如果state=0代表获取共享锁成功返回1，否则代表获取共享锁失败，返回-1

protected int tryAcquireShared(int acquires) {return (getState() == 0) ? 1 : -1;}

//尝试释放共享资源

protected boolean tryReleaseShared(int releases) {

for (;;) {

int c = getState();  
 //已经等于0了，无法再释放了

//防止计数器值为0后，其他线程又调用了countDown方法，状态值就会变成负数

if (c == 0)

return false;

int nextc = c-1;

if (compareAndSetState(c, nextc)) //尝试cas更新--state

//cas成功

//这是第一次减为0的时候返回true，这时会唤醒后面排队的线程

return nextc == 0;

}

}

private void doReleaseShared() {

//遵循FIFO的规则唤醒线程

for (;;) {

Node h = head;

if (h != null && h != tail) { //不止有head节点

int ws = h.waitStatus;

if (ws == Node.SIGNAL) { //如果当前节点的ws状态为-1

if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))

//尝试修改当前节点的ws状态-1变为0，如果修改失败，continue

continue;

//尝试修改当前节点的ws状态-1变为0，成功

//唤醒当前节点

unparkSuccessor(h);

} else if (ws == 0 && !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))

continue;

}

if (h == head)

break;

}

}

public final void acquireSharedInterruptibly(int arg) throws InterruptedException {

//清除线程中断标识，并返回线程中断状态

if (Thread.interrupted()) throw new InterruptedException();

//如果state=0返回1，否则返回-1

if (tryAcquireShared(arg) < 0) //此时state还不是1，线程需要加入队列，等待被唤醒

doAcquireSharedInterruptibly(arg);

}

private void doAcquireSharedInterruptibly(int arg) throws InterruptedException {

//以SHARED方式加入到head链表尾部，并返回node

//SHARED表示标记线程是因为获取共享资源失败被阻塞添加到队列中的

final Node node = addWaiter(Node.SHARED);

boolean failed = true;

try {

for (;;) {

final Node p = node.predecessor();//p为node的前驱节点

if (p == head) {

//如果node为队首节点，按照队列FIFO的原则，第一个被唤醒

int r = tryAcquireShared(arg);

if (r >= 0) { //此时state=0，准备唤醒队列中的线程了

setHeadAndPropagate(node, r);

p.next = null; // help GC

failed = false;

return;

}

}

if (shouldParkAfterFailedAcquire(p, node) && parkAndCheckInterrupt())

throw new InterruptedException();

}

} finally {

if (failed)

cancelAcquire(node);

}

}

private void setHeadAndPropagate(Node node, int propagate) {

Node h = head;

setHead(node);

if (propagate > 0 || h == null || h.waitStatus < 0 || (h = head) == null || h.waitStatus < 0) {

Node s = node.next;

if (s == null || s.isShared())

doReleaseShared();

}

}

private void setHead(Node node) {

head = node;

node.thread = null;

node.prev = null;

}

//以SHARED模式插入节点到CLH队尾

private Node addWaiter(Node mode) {

//设置nextWaiter为SHARED

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

Node pred = tail;

if (pred != null) { //head还没有被初始化

node.prev = pred;

if (compareAndSetTail(pred, node)) {

pred.next = node;

return node;

}

}

enq(node);

return node;

}

private Node enq(final Node node) {

for (;;) {

Node t = tail;

if (t == null) {

if (compareAndSetHead(new Node()))

//尝试cas更新成功，head=new Node()，继续for循环插入node节点;

tail = head;

} else {

node.prev = t;

if (compareAndSetTail(t, node)) {

//尝试cas更新成功，tail=node;

t.next = node;

return t;

}

}

}

}

}

}

CountDownLatch的使用：

public class CountDownLatchDemo {

public static void main(String[] args) {

int peopleNum = 10;

CountDownLatch beginCountDownLatch = new CountDownLatch(1);

CountDownLatch endCountDownLatch = new CountDownLatch(peopleNum);

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

final int NO = i + 1;

new Thread(() -> {

try {

beginCountDownLatch.await();

System.out.println("No." + NO + " arrived");

} catch (InterruptedException e) {

e.printStackTrace();

} finally {

endCountDownLatch.countDown();

}

}).start();

}

System.out.println("2s后开始");

try {

TimeUnit.SECONDS.sleep(2);

} catch (InterruptedException e) {

e.printStackTrace();

}

beginCountDownLatch.countDown();

try {

endCountDownLatch.await();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Game Over.");

}

}

1. ReentrantLock底层源码解析和使用

ReentrantLock是一个可重入且独占式的锁。

可重入：任意线程获取到锁后可以再次获取该锁而不会被阻塞，该特性的首先需要解决以下两个问题：

线程再次获取锁：所需要去识别获取锁的线程是否为当前占据锁的线程，如果是，则再次获取成功；

锁的最终释放：线程重复n次获取了锁，随后在第n次释放该锁后，其它线程能够获取到该锁。锁的最终释放要求锁对于获取进行计数自增，计数表示当前线程被重复获取的次数，而被释放时，计数自减，当计数为0时表示锁已经成功释放。

公平锁和非公平锁：如果在绝对时间上，先对锁进行获取的请求你一定先被满足，那么这个锁是公平的，反之，是不公平的。公平锁的获取，也就是等待时间最长的线程最优先获取锁，也可以说锁获取是顺序的。ReentrantLock的公平与否，可以通过它的构造函数来决定。

public ReentrantLock() {sync = new NonfairSync();}//默认是非公平锁

public ReentrantLock(boolean fair) {sync = fair ? new FairSync() : new NonfairSync();}

以非公平锁NonfairSync为例分析ReentrantLock底层机制

此时waitStatus值可以是：

//表示当前结点已取消调度。当timeout或被中断（响应中断的情况下），会触发变更为此状态，进入该状态后的结点将不会再变化

//表示该线程因为被中断或者等待超时，需要从等待队列中取消等待

static final int CANCELLED = 1;

//被标识为该等待唤醒状态的后继结点，当其前继结点的线程释放了同步锁或被取消，将会通知该后继结点的线程执行。说白了，就是处于唤醒状态，只要前继结点释放锁，就会通知标识为SIGNAL状态的后继结点的线程执行

static final int SIGNAL = -1;

//表示当前Node结点的等待状态

//共有5种取值CANCELLED、SIGNAL、CONDITION、PROPAGATE、0初始化状态

//负值表示结点处于有效等待状态，而正值表示结点已被取消。所以源码中很多地方用>0、<0来判断结点的状态是否正常

volatile int waitStatus = 0;

//代表锁，如果获取锁成功cas更新state+1

private volatile int state;

lock

public void lock{sync.lock();}

final void lock() {

//尝试cas获取锁，如果true表示state=1成功获取锁，然后记录获取到独占锁的线程

if (compareAndSetState(0, 1))

setExclusiveOwnerThread(Thread.currentThread());//记录获取到独占锁的线程

else

//获取锁失败

acquire(1);

}

public final void acquire(int arg) {

if (!tryAcquire(arg) && //当前线程尝试获取锁，成功true，失败false

//如果当前线程获取锁失败了，当前线程以独占的方式加入到等待队列队尾，等待被唤醒

acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

selfInterrupt(); //具体实现： Thread.currentThread().interrupt();

}

//tryAcquire的具体实现

//首先判断同步状态是否为0，如果是0，则表示该锁还没有被线程持有，然后通过CAS尝试获取锁，如果修改成功，返回true。如果同步状态不为0，则表示该锁已经被线程持有，需要判断当前线程是否为获取锁的线程，如果是则获取锁，成功返回true。成功获取锁的线程再次获取该锁，只是增加了同步状态的值，这也就实现了可重入锁

final boolean nonfairTryAcquire(int acquires) {

final Thread current = Thread.currentThread();

int c = getState();

if (c == 0) { //state=0此时还没有线程获取到锁

if (compareAndSetState(0, acquires)) {//尝试cas获取锁 state++

setExclusiveOwnerThread(current);//获取锁成功，就设置当前独占模式下同步状态的持有者为当前线程

return true;

}

//已经有线程获取到锁，看是不是自己，这里就是可重入锁的逻辑

} else if (current == getExclusiveOwnerThread()) {

//当前线程已经获取到锁了

int nextc = c + acquires;

if (nextc < 0) throw new Error("Maximum lock count exceeded");

setState(nextc);//没有使用cas方式更新state的原因：此时一定是单线程操作 state++

return true;

}

//返回false表示获取到锁的不是当前线程

return false;

}

unlock

//成功获取锁的线程在完成业务逻辑之后，需要调用unlock()来释放锁

public void unlock() {release(1);}

public final boolean release(int arg) {

if (tryRelease(arg)) {//true表示当前线程已经完全释放锁了（重入n次就得释放n次）

//当前线程已经完全释放锁了，准备唤醒线程了

Node h = head;

if (h != null && h.waitStatus != 0)

unparkSuccessor(h); //唤醒线程

return true;

}

return false;

}

//返回true表明当前线程已经完全释放锁

//此时一定是单线程环境

protected final boolean tryRelease(int releases) {

int c = getState() - releases;

if (Thread.currentThread() != getExclusiveOwnerThread())

throw new IllegalMonitorStateException();

boolean free = false;

// 新的状态值是否为0，若为0，则表示该锁已经完全释放了，其他线程可以获取同步状态了

if (c == 0) {

free = true;

setExclusiveOwnerThread(null); //当前线程已经释放锁成功了

}

setState(c);

return free;

}

private void unparkSuccessor(Node node) {

int ws = node.waitStatus;

if (ws < 0)

//尝试cas更新node的waitStatus为0

compareAndSetWaitStatus(node, ws, 0); --尝试cas更新ws状态为SINGAL=0

Node s = node.next; //后继节点s

if (s == null || s.waitStatus > 0) {// 没有后继节点或者后继节点被取消

s = null;

//找到当前节点前面第一个ws<=0的节点，准备唤醒

for (Node t = tail; t != null && t != node; t = t.prev) //从后向前找

if (t.waitStatus <= 0) //SIGNAL-1或0

s = t;

}

if (s != null)

LockSupport.unpark(s.thread); //唤醒指定线程

}

lock()方法执行流程总结：

1. 尝试获取锁（cas更新state），如果成功，执行自己的业务逻辑，lock走完；如果失败，调用acquire(1)方法，执行2
2. 调用tryAcquire(1)方法尝试获取锁（可重入锁的逻辑），如果当前线程成功获取锁返回true，执行自己的业务逻辑，lock走完；如果失败调用addWaiter(Node.EXCLUSIVE)将线程以独占方式加入到CLH等待队列的队尾，执行3
3. 调用acquireQueued()方法，for自旋中执行4，如果在此过程中发生异常，执行7
4. 如果当前节点node的前驱节点是head，调用tryAcquire(1)方法尝试获取锁，如果失败，执行5；如果成功，更新head=node，返回等待过程中是否被中断过，如果被中断过，调用seltInterrupt()将中断补上，仅仅是打上中断标志，lock走完；如果没有被中断过，lock走完
5. 调用shouldParkAfterFailedAcquire(p, node)方法，如果前驱节点ws状态为SIGNAL，则说明当前线程会被前驱节点唤醒，当前线程现在可以被阻塞了，返回true后执行6；如果前驱节点ws状态为CANCELLED（被取消了），就一直往前找，找到第一个ws节点状态为0或SIGNAL(-1)的节点，排在他后面，并且把CANCELLED状态的节点从队列中移除，然后返回false后执行4；如果前驱节点ws状态为0，尝试cas更新前驱节点ws状态为SIGNAL(-1)，有可能会失败，返回false后执行4
6. 调用parkAndCheckInterrupt()方法，park当前线程，等待unpark唤醒或者被interrupt中断，返回线程中断状态并清除，再执行4
7. 调用cancelAcquire方法后，lock走完

unlock()方法执行流程总结：

1. 调用tryRelease()方法尝试释放锁，返回true表示当前线程完全释放锁成功，执行2，false表示当前线程仍然占据锁，unlock走完

//用于队列中的线程自旋地以独占且不可中断的方式获取同步状态（acquire），直到拿到锁之后再返回。该方法的实现分成两部分：如果当前节点已经成为头结点，尝试获取锁（tryAcquire）成功，然后返回；否则检查当前节点是否应该被park，然后将该线程park并且检查当前线程是否被可以被中断。

final boolean acquireQueued(final Node node, int arg) {

boolean failed = true; //标记是否拿到资源

try {

boolean interrupted = false; //标识等待过程中是否被中断过

//自旋，不断for循环尝试获取锁

for (;;) {

final Node p = node.predecessor(); //前驱节点

//如果p=head，则尝试获取锁，因为如果前一个是head说明head有可能已经释放锁，当前node可以尝试获取锁

if (p == head && tryAcquire(arg)) {

//就算当前节点是首节点，也要获取锁的原因：非公平

setHead(node); //如果获取成功则将当前node设为head

p.next = null; //setHead中node.prev已置为null，此处再将head.next置为null，就是为了方便GC回收以前的head结点。也就意味着之前拿完资源的结点出队了！

failed = false;

return interrupted; //返回等待过程中是否被中断过

}

if (shouldParkAfterFailedAcquire(p, node) && //检查前一个线程的ws状态，看当前获取锁失败的线程是否应该被挂起

parkAndCheckInterrupt()) //挂起当前线程，unpark或者interrupt后返回中断标识

interrupted = true; //该阻塞线程被是被中断唤醒的

}

} finally {

if (failed) //如果有异常，取消请求，对应到队列操作，就是将当前节点从队列中移除

cancelAcquire(node); --何时会执行cancelAcquire方法？

}

}

//acquireQueued()流程总结：

//1.结点进入队尾后，检查状态，找到安全休息点；

//2.调用park()进入waiting状态，等待unpark()或interrupt()唤醒自己；

//3.被唤醒后，看自己是不是有资格能拿到号。如果拿到，head指向当前结点，并返回从入队到拿到号的整个过程中是否被中断过；如果没拿到，继续流程1。

//acquire()流程总结：

//1.调用tryAcquire()尝试获取锁，如果成功直接返回

//2.如果获取锁失败，调用addWriter()将thread以独占方式加入到等待队列的尾部

//3.acquireQueued()使线程阻塞在等待队列中获取资源，一直获取到资源后才返回。如果在整个等待过程中被中断过，则返回true，否则返回false

//4.如果线程在等待过程中被中断过，它是不响应的。只是获取资源后才再进行自我中断selfInterrupt()，将中断补上

//判断当前线程获取锁失败后是否应该被挂起

private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {

int ws = pred.waitStatus;

if (ws == Node.SIGNAL) // -1

//如果前驱节点的状态为SIGNAL，说明当前节点届时会被前驱节点唤醒，所以现在可以阻塞自己等待被前驱节点唤醒 return true;

if (ws > 0) { //如果前驱节点的状态为CANCELLED

//把已经是cancelled的node从队列中移除，不再进行获取锁

// 如果前驱放弃了，那就一直往前找，直到找到最近一个正常等待的状态，并排在它的后边。

//注意：那些放弃的结点，由于被自己“加塞”到它们前边，它们相当于形成一个无引用链，被GC回收

do {

node.prev = pred = pred.prev;

} while (pred.waitStatus > 0);

pred.next = node;

} else {

//如果ws=0或者-1

compareAndSetWaitStatus(pred, ws, Node.SIGNAL); --尝试cas更新ws状态为SINGAL=0

}

return false;

}

//只有当该节点的前驱结点的状态为SIGNAL时，才可以对该结点所封装的线程进行park操作。否则，将不能进行park操作

// park之后有两种途径可以唤醒该线程：1）被unpark()；2）被interrupt()

private final boolean parkAndCheckInterrupt() {

LockSupport.park(this); //让当前线程进入waiting状态，可以响应中断，但是不抛出InterruptedException异常

//unpark或interrupt之后执行返回

return Thread.interrupted(); // 返回当前线程是否已中断，并清除中断标记位

}

private final boolean compareAndSetTail(Node expect, Node update) {

return unsafe.compareAndSwapObject(this, tailOffset, expect, update);

}

private static final boolean compareAndSetNext(Node node, Node expect, Node update) {

return unsafe.compareAndSwapObject(node, nextOffset, expect, update);

}

//<https://blog.csdn.net/dataiyangu/article/details/105024448>

private void cancelAcquire(Node node) {

if (node == null) return;

node.thread = null;

//while循环找到node前面的一个ws=0或SIGNAL-1的节点

Node pred = node.prev;

while (pred.waitStatus > 0) //CANCELLED=1

node.prev = pred = pred.prev;

Node predNext = pred.next;

node.waitStatus = Node.CANCELLED;

if (node == tail && compareAndSetTail(node, pred)) {

//如果node是tail节点，cas更新tail=pred

//cas更新pred.next = null

compareAndSetNext(pred, predNext, null);

} else {

int ws;

//5. 如果node既不是tail，又不是head的后继节点

//则将node的前继节点的waitStatus置为SIGNAL

//并使pre的前继节点指向node的后继节点

if (pred != head &&

((ws = pred.waitStatus) == Node.SIGNAL ||

(ws <= 0 && compareAndSetWaitStatus(pred, ws, Node.SIGNAL))) &&

pred.thread != null) {

Node next = node.next;

if (next != null && next.waitStatus <= 0)

compareAndSetNext(pred, predNext, next);

} else {

//6. 如果node是head的后继节点，则直接唤醒node的后继节点

unparkSuccessor(node);

}

node.next = node; // help GC

}

}

以公平锁FairSync为例分析ReentrantLock底层机制

protected final boolean tryAcquire(int acquires) {

final Thread current = Thread.currentThread();

int c = getState();

if (c == 0) {

if (!hasQueuedPredecessors() &&

compareAndSetState(0, acquires)) {

setExclusiveOwnerThread(current);

return true;

}

} else if (current == getExclusiveOwnerThread()) {

int nextc = c + acquires;

if (nextc < 0) throw new Error("Maximum lock count exceeded");

setState(nextc);

return true;

}

return false;

}

1. Semaphore底层源码解析和使用
2. Mutex底层源码解析和使用
3. JAVA线程池的创建方式

线程池创建有七种方式，最核心的是最后一种：

newSingleThreadExecutor()：它的特点在于工作线程数目被限制为 1，操作一个无界的工作队列，所以它保证了所有任务的都是被顺序执行，最多会有一个任务处于活动状态，并且不允许使用者改动线程池实例，因此可以避免其改变线程数目；

newCachedThreadPool()：它是一种用来处理大量短时间工作任务的线程池，具有几个鲜明特点：它会试图缓存线程并重用，当无缓存线程可用时，就会创建新的工作线程；如果线程闲置的时间超过 60 秒，则被终止并移出缓存；长时间闲置时，这种线程池，不会消耗什么资源。其内部使用 SynchronousQueue 作为工作队列；

newFixedThreadPool(int nThreads)：重用指定数目（nThreads）的线程，其背后使用的是无界的工作队列，任何时候最多有 nThreads 个工作线程是活动的。这意味着，如果任务数量超过了活动队列数目，将在工作队列中等待空闲线程出现；如果有工作线程退出，将会有新的工作线程被创建，以补足指定的数目 nThreads；

newSingleThreadScheduledExecutor()：创建单线程池，返回 ScheduledExecutorService，可以进行定时或周期性的工作调度；

newScheduledThreadPool(int corePoolSize)：和newSingleThreadScheduledExecutor()类似，创建的是个 ScheduledExecutorService，可以进行定时或周期性的工作调度，区别在于单一工作线程还是多个工作线程；

newWorkStealingPool(int parallelism)：这是一个经常被人忽略的线程池，Java 8 才加入这个创建方法，其内部会构建ForkJoinPool，利用Work-Stealing算法，并行地处理任务，不保证处理顺序；

ThreadPoolExecutor()：是最原始的线程池创建，上面1-3创建方式都是对ThreadPoolExecutor的封装。

public ThreadPoolExecutor(int corePoolSize, //核心线程数量

int maximumPoolSize, //最大线程数量

long keepAliveTime, //线程池维护线程所允许的空闲时间。当线程池中的线程数量大于corePoolSize的时候，如果这时没有新的任务提交，核心线程外的线程不会立即销毁，而是会等待，直到等待的时间超过了keepAliveTime

TimeUnit unit, //时间级别

BlockingQueue<Runnable> workQueue, //等待队列，当任务提交时，如果线程池中的线程数量大于等于corePoolSize的时候，把该任务封装成一个Worker对象放入等待队列

ThreadFactory threadFactory, //它是ThreadFactory类型的变量，用来创建新线程

RejectedExecutionHandler handler) { //RejectedExecutionHandler类型的变量，表示线程池的饱和策略。如果阻塞队列满了并且没有空闲的线程，这时如果继续提交任务，就需要采取一种策略处理该任务

}

private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));

private static final int COUNT\_BITS = 29; //常量

private static final int CAPACITY = (1 << COUNT\_BITS) - 1; // 0001 1111 1111 1111 1111 1111 1111 1111

// runState is stored in the high-order bits

//-1的补码为11111111111111111111111111111111，左移29位就是11100000000000000000000000000000

//2的补码为10，左移29位就是01000000000000000000000000000000

private static final int RUNNING = -1 << COUNT\_BITS; // 高3位为111， 低29位为0 该状态的线程池会接收新任务，也会处理在阻塞队列中等待处理的任务

private static final int SHUTDOWN = 0 << COUNT\_BITS; // 高3位为000， 低29位为0 该状态的线程池不会再接收新任务，但还会处理已经提交到阻塞队列中等待处理的任务

private static final int STOP = 1 << COUNT\_BITS; // 高3位为001， 低29位为0 该状态的线程池不会再接收新任务，不会处理在阻塞队列中等待的任务，而且还会中断正在运行的任务

private static final int TIDYING = 2 << COUNT\_BITS; // 高3位为010， 低29位为0 所有任务都被终止了，workerCount为0，为此状态时还将调用terminated()方法

private static final int TERMINATED = 3 << COUNT\_BITS; // 高3位为100， 低29位为0 terminated()方法调用完成后变成此状态

// Packing and unpacking ctl

private static int runStateOf(int c) { return c & ~CAPACITY; } // c & 高3位为1，低29位为0的~CAPACITY，用于获取高3位保存的线程池状态

private static int workerCountOf(int c) { return c & CAPACITY; } // c & 高3位为0，低29位为1的CAPACITY，用于获取低29位的线程数量

private static int ctlOf(int rs, int wc) { return rs | wc; } // 参数rs表示runState，参数wc表示workerCount，即根据runState和workerCount打包合并成ctl

public void execute(Runnable command) {

if (command == null)

throw new NullPointerException();

int c = ctl.get(); // 初始状态下，c=

if (workerCountOf(c) < corePoolSize) {RUNNING

if (addWorker(command, true))

return;

c = ctl.get();

}

if (isRunning(c) && workQueue.offer(command)) {

int recheck = ctl.get();

if (! isRunning(recheck) && remove(command))

reject(command);

else if (workerCountOf(recheck) == 0)

addWorker(null, false);

}

else if (!addWorker(command, false))

reject(command);

}

线程池都有哪些状态？

RUNNING：这是最正常的状态，接受新的任务，处理等待队列中的任务。

SHUTDOWN：不接受新的任务提交，但是会继续处理等待队列中的任务。

STOP：不接受新的任务提交，不再处理等待队列中的任务，中断正在执行任务的线程。

TIDYING：所有的任务都销毁了，workCount 为 0，线程池的状态在转换为 TIDYING 状态时，会执行钩子方法 terminated()。

TERMINATED：terminated()方法结束后，线程池的状态就会变成这个。

线程池中 submit() 和 execute() 方法有什么区别？

execute()：只能执行 Runnable 类型的任务。

submit()：可以执行 Runnable 和 Callable 类型的任务。

在 Java 程序中怎么保证多线程的运行安全？

方法一：使用安全类，比如 Java. util. concurrent 下的类。

方法二：使用自动锁 synchronized。

方法三：使用手动锁 Lock。

1. synchronized和Lock区别

synchronized 可以给类、方法、代码块加锁；而 lock 只能给代码块加锁。

synchronized 不需要手动获取锁和释放锁，使用简单，发生异常会自动释放锁，不会造成死锁；而 lock 需要自己加锁和释放锁，如果使用不当没有 unLock()去释放锁就会造成死锁。

通过 Lock 可以知道有没有成功获取锁，而 synchronized 却无法办到。

1. synchronized和ReentrantLock区别

synchronized 早期的实现比较低效，对比 ReentrantLock，大多数场景性能都相差较大，但是在 Java 6 中对 synchronized 进行了非常多的改进。

主要区别如下：

ReentrantLock 使用起来比较灵活，但是必须有释放锁的配合动作；

ReentrantLock 必须手动获取与释放锁，而 synchronized 不需要手动释放和开启锁；

ReentrantLock 只适用于代码块锁，而 synchronized 可用于修饰方法、代码块等。

volatile 标记的变量不会被编译器优化；synchronized 标记的变量可以被编译器优化。

1. 线程安全问题

线程安全问题遵循这三个规律：是否是多线程；是否有共享数据；是否有多个线程操作共享数据。

<https://mp.weixin.qq.com/s/rd13SOSxhLA6TT13N9ni8Q>

什么是 CPU 缓存行？

什么是内存屏障？

什么是伪共享？

如何避免伪共享？

1. 乐观锁和悲观锁

乐观锁：

总是假设最好的情况，每次拿数据都认为别人不会修改数据，所以不会加锁，但是更新的时候，会判断在此期间有没有人修改过；  
 乐观锁一般会使用版本号机制或CAS算法实现：  
 版本号机制：一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功  
 CAS算法：Compare And Swap无锁算法，即不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blocking Synchronization）

悲观锁：

总是假设最坏的情况，每次拿数据都认为别人会修改数据，所以要加锁，别人只能等待，直到我释放锁才能拿到锁；(共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程)

数据库的行锁、表锁、读锁、写锁都是这种方式。java中的synchronized和Lock的实现类也是悲观锁的思想

数据库级别悲观锁：采用的数据库内部的锁机制，一个典型的倚赖数据库的悲观锁调用：select \* from sys\_user where name=”张三” for update;

乐观锁和悲观锁的应用场景：

乐观锁适用于读多写少的情况，因为不加锁直接读可以让系统的性能大幅度的提高

悲观锁适用于写多读少的情况，因为等待到锁被释放后，可以立即获得锁进行操作

1. 公平锁和非公平锁

公平锁：

指多个线程按照申请锁的顺序来获取锁

非公平锁：

指多个线程获取锁的顺序并不是按照申请锁的顺序，有可能后申请的线程比先申请的线程优先获取锁。有可能，会造成优先级反转或者饥饿现象

对于Java ReentrantLock而言，通过构造函数指定该锁是否是公平锁，默认是非公平锁。非公平锁的优点在于吞吐量比公平锁大。 对于Synchronized而言，也是一种非公平锁。由于其并不像ReentrantLock是通过AQS的来实现线程调度，所以并没有任何办法使其变成公平锁

1. 可重入锁和不可重入锁
2. 自旋锁和阻塞锁

自旋锁：

就是让某线程进入已被其它线程占用的同步代码时等待一段时间，不会被立即挂起，看持有锁的线程是否会很快释放锁。这里等待的方式就是执行一段无意义的循环，CPU空转（不放弃CPU）

适应性自旋锁：

自旋的时间（次数）不再固定，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。

如果在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也是很有可能再次成功，进而它将允许自旋等待持续相对更长的时间。

如果对于某个锁，自旋很少成功获得过，那在以后尝试获取这个锁时将可能省略掉自旋过程，直接阻塞线程，避免浪费处理器资源

为什么会有自旋锁：

阻塞或唤醒一个线程需要操作系统切换cpu状态来完成，这种状态的转换需要耗费处理器时间，如果同步代码块的处理逻辑过于简单，那么有可能状态转换的时间要比用户代码执行的时间要长。所以为了让线程在短暂的等待之后就要执行下去，需要自旋，在自旋完成后，前面锁定了同步资源的线程释放锁，那么当前线程就不需要阻塞或唤醒就可以获取同步资源，从而避免了线程切换的开销，这就是自旋锁。

自旋锁的缺点：

自旋等待虽然避免了线程切换的开销，但是它要占用cpu时间片。如果锁被占用的时间很短，自旋等待的效果就会很好，反之如果锁被占用的时间很长，那么自旋的线程只会白白浪费处理器资源。所以，自旋等待的时间必须要有一定的限度，如果自旋超过了限定次数（默认是10次，可以使用-XX:PreBlockSpin来更改）没有成功获得锁，就应当挂起线程。

自旋锁在JDK1.4.2中引入，使用-XX:+UseSpinning来开启。JDK6中变为默认开启，并且引入了自适应的自旋锁（适应性自旋锁）

阻塞锁：

线程拿不到锁的时候，放弃CPU，进入阻塞状态

1. 死锁

是指两个或两个以上的进程在执行过程中,因争夺对方的资源而造成的一种互相等待的现象,若无外力作用,它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。

（1） 因为系统资源不足。

（2） 进程运行推进顺序不合适。

（3） 资源分配不当等。

当线程 A 持有独占锁a，并尝试去获取独占锁 b 的同时，线程 B 持有独占锁 b，并尝试获取独占锁 a 的情况下，就会发生 AB 两个线程由于互相持有对方需要的锁，而发生的阻塞现象，我们称为死锁。

发生死锁的四个必要条件：

（1） 互斥条件：一个资源每次只能被一个进程使用。

（2） 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

（3） 不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。

（4） 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

测试死锁代码：

public class diedlock {

private static Object lock1 = new Object(); //锁1

private static Object lock2 = new Object(); //锁2

public static void main(String[] args) throws InterruptedException{

Thread thread1 = new Thread(() -> {

synchronized (lock1) {

System.out.println("线程1成功获取锁1");

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("线程1等待获取锁2");

synchronized (lock2) {

System.out.println("线程1成功获取锁2");

}

}

});

Thread thread2 = new Thread(() -> {

synchronized (lock2) {

System.out.println("线程2成功获取锁2");

System.out.println("线程2等待获取锁1");

synchronized (lock1) {

System.out.println("线程2成功获取锁1");

}

}

});

thread1.start();

Thread.sleep(1000);

thread2.start();

thread1.join();

thread2.join();

System.out.println("线程1和线程2执行完成....");

}

}

1. 解决死锁的方法

尽量使用tryLock(long timeout, TimeUnit unit)的方法(ReentrantLock、ReentrantReadWriteLock)，设置超时时间，超时可以退出防止死锁。

尽量使用Java.util.concurrent并发类代替自己手写锁。

尽量降低锁的使用粒度，尽量不要几个功能用同一把锁。

尽量减少同步的代码块。

1. 活锁

指线程1可以使用资源，但它很礼貌，让其他线程先使用资源，线程2也可以使用资源，但它很绅士，也让其他线程先使用资源。这样你让我，我让你，最后两个线程都无法使用资源。活锁不会被阻塞，而是不停检测一个永远不可能为真的条件。除去进程本身持有的资源外，活锁状态的进程会持续耗费宝贵的CPU时间。