# 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() {

}

//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");

} else {

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对象

\* @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更新成功 false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：java.util.concurrent.atomic.AtomicInteger

public class AtomicInteger {

//底层实现借助了Unsafe类

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

//指类中相应字段在该类的偏移量

//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);

}

/\*\*

\* 尝试比较和替换

\*@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：java.util.concurrent.atomic.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：java.util.concurrent.atomic.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：java.util.concurrent.atomic.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;

}

/\*\*

\* Atomically sets the value of both the reference and mark

\* to the given update values if the

\* current reference is {@code ==} to the expected reference

\* and the current mark is equal to the expected mark.

\*

\* <p><a href="package-summary.html#weakCompareAndSet">May fail

\* spuriously and does not provide ordering guarantees</a>, so is

\* only rarely an appropriate alternative to {@code compareAndSet}.

\*

\* @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);

}

/\*\*

\* Atomically sets the value of both the reference and mark

\* to the given update values if the

\* current reference is {@code ==} to the expected reference

\* and the current mark is equal to the expected mark.

\*

\* @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);

}

/\*\*

\* Atomically sets the value of the mark to the given update value

\* if the current reference is {@code ==} to the expected

\* reference. Any given invocation of this operation may fail

\* (return {@code false}) spuriously, but repeated invocation

\* when the current value holds the expected value and no other

\* thread is also attempting to set the value will eventually

\* succeed.

\*

\* @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：java.util.concurrent.atomic.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;

private static native boolean VMSupportsCS8();

//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); }

}

/\*\*

\* Eventually sets to the given value.

\*

\* @param newValue the new value

\* @since 1.6

\*/

public final void lazySet(long newValue) {

unsafe.putOrderedLong(this, valueOffset, newValue);

}

/\*\*

\* Atomically sets to the given value and returns the old value.

\*

\* @param newValue the new value

\* @return the previous value

\*/

public final long getAndSet(long newValue) {

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

}

/\*\*

\* Atomically sets the value to the given updated value

\* if the current value {@code ==} the expected value.

\*

\* @param expect the expected value

\* @param update the new value

\* @return {@code true} if successful. False return indicates that

\* the actual value was not equal to the expected value.

\*/

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

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

}

/\*\*

\* Atomically sets the value to the given updated value

\* if the current value {@code ==} the expected value.

\*

\* <p><a href="package-summary.html#weakCompareAndSet">May fail

\* spuriously and does not provide ordering guarantees</a>, so is

\* only rarely an appropriate alternative to {@code compareAndSet}.

\*

\* @param expect the expected value

\* @param update the new value

\* @return {@code true} if successful

\*/

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

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

}

/\*\*

\* Atomically increments by one the current value.

\*

\* @return the previous value

\*/

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;

}

/\*\*

\*

\* @param updateFunction a side-effect-free function

\* @return the previous value

\* @since 1.8

\*/

public final long getAndUpdate(LongUnaryOperator updateFunction) {

long prev, next;

do {

prev = get();

next = updateFunction.applyAsLong(prev);

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

return prev;

}

/\*\*

\*

\* @param x the update value

\* @param accumulatorFunction a side-effect-free function of two arguments

\* @return the previous value

\* @since 1.8

\*/

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;

}

/\*\*

\*

\* @param x the update value

\* @param accumulatorFunction a side-effect-free function of two arguments

\* @return the updated value

\* @since 1.8

\*/

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：java.util.concurrent.atomic.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变量的地址偏移

private static final long CELLSBUSY; //cellsBusy变量的地址偏移

private static final long PROBE; //threadLocalRandomProbe变量的地址偏移

//cpu可用核数，决定了槽数组的大小

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：也就是现在没有正在进行扩容或初始化cells

//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  
 } else if (!wasUncontended) // CAS already known to fail  
 wasUncontended = true; // Continue after rehash  
 else if (a.cas(v = a.value, ((fn == null) ? v + x : fn.applyAsLong(v, x))))  
 break;  
 else if (n >= NCPU || cells != as)  
 collide = false; // At max size or stale  
 else if (!collide)  
 collide = true;  
 else if (cellsBusy == 0 && casCellsBusy()) {  
 try {  
 if (cells == as) { // Expand table unless stale  
 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;  
 }  
 collide = false;  
 continue; // Retry with expanded table  
 }  
 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();

Class<?> sk = Striped64.class;

BASE = UNSAFE.objectFieldOffset(sk.getDeclaredField("base"));

CELLSBUSY = UNSAFE.objectFieldOffset(sk.getDeclaredField("cellsBusy"));

Class<?> tk = Thread.class;

PROBE = UNSAFE.objectFieldOffset(tk.getDeclaredField("threadLocalRandomProbe"));

} catch (Exception e) {

throw new Error(e);

}

}

}

1. 线程和进程的区别
2. 创建线程的几种方式

集成Thread类

实现Runnable接口

实现Callable接口

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: 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. JAVA线程相关的方法、关键字

Thread.sleep();//休眠当前正在运行的线程，进入阻塞状态，只有当睡眠的时间结束才会重新进入就绪状态

Thread.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

thread.join();//当前线程等待该线程执行完成才会执行

关于sleep()方法和yield()方的区别如下：

①、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。

②、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。

③、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。

<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和Static Synchronized区别

       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

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

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

常用关键字和常用方法  
 volatile  
 synchronized  
ThreadLocal类  
AQS  
ReentrantLock  
线程池的实现

AQS：AbstractQueuedSynchronizer

Lock

CountDownLatch类的使用

线程状态：

NEW（新建）：新创建了一个线程对象，但还没有调用start()方法

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。

“阻塞”与“等待”的区别：

（1）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。

（2）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

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

public class Object {

public final native void notify();

public final native void notifyAll();

public final native void wait(long timeout) throws InterruptedException;

//当前线程调用“锁对象”的wait()方法，当前线程释放对象锁，进入等待队列WAITING

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

public final void wait() throws InterruptedException {

wait(0);

}

}

thread.join(); //当前线程里调用其它线程t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，当前线程不会释放已经持有的对象锁。线程t执行完毕或者millis时间到，当前线程进入就绪状态

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

thread.yield();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

thread.sleep(timeout);不会释放对象锁,thread会进入TIMED\_WAITING，睡眠时间结束后进入就绪状态，当前线程会进入TIME\_WAITING

obj.wait();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(timeout);指定等待时间，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，当前线程会进入TIME\_WAITING

obj.notify();唤醒一个正在等待该对象锁的线程（即处于wait状态的线程），具体哪一个随机

obj.notifyAll();唤醒所有正在等待该对象锁的线程（即处于wait状态的线程)

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。

notify和notifyAll的区别：

1、notify只会随机选取一个处于等待池中的线程进入锁池去竞争获取锁的机会；

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

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. JAVA线程池

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);

}