# 第16天【JUC】

## 主要内容

1. JUC的简介
2. JUC组成部分
3. Atomic包
4. Locks包
5. Executor
6. Tools工具类
7. Collections

## 学习目标

|  |  |
| --- | --- |
| 知识点 | 要求 |
| JUC的简介 | 了解 |
| JUC组成部分 | 掌握 |
| Atomic包 | 掌握 |
| Locks包 | 掌握 |
| Executor执行器 | 理解 |
| Tools工具类 | 掌握 |
| Collections集合 | 掌握 |
| CountDownLatch | 了解 |
| CyclicBarrier | 了解 |
| Semaphore | 了解 |

## 一、JUC简介

### 1.1 JUC概念

JUC是什么

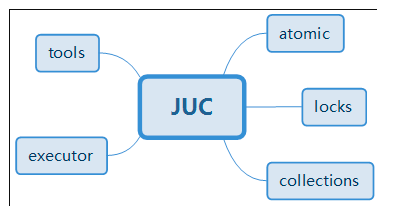
JUC是 在Java 5.0添加的 java.util.concurrent包的简称，目的就是为了更好的支持高并发任务，让开发者利用这个包进行的多线程编程时可以有效的减少竞争条件和死锁线程。

### **1.2JUC作者**

JUC的作者是Doug Lea



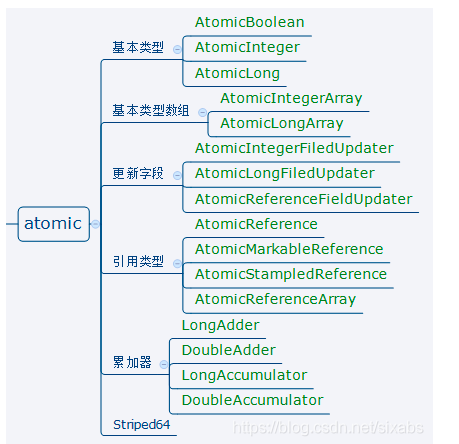
### 1.3JUC的结构组成



## 二、Atomic

### 2.1概念简介

核心是CAS，主要提供了一系列原子变量更新操作的类，提供非阻塞式算法基础包含有AtomicBoolean、AtomicInteger、AtomicIntegerArray等原子变量类，他们的实现原理大多是持有它们各自的对应的类型变量value，而且被volatile关键字修饰了。这样来保证每次一个线程要使用它都会拿到最新的值



### 2.2 volatile关键字

* 基本概念

**先补充一下概念：Java 内存模型中的可见性、原子性和有序性。**

volatile关键字，易变的; 不稳定的意思。使用volatile修饰的变量，可以保证在多个线程之间的可见性，并且避免指令重排。但是无法保证操作的原子性。

#### 【示例1】volatile的效果展示

|  |
| --- |
| **public class** Test {  **private static boolean** *flag* = **true**;  **public static void** main(String[] args) {  *//创建一个线程并启动* **new** Thread(**new** Runnable() {  @Override  **public void** run() {  **while**(*flag*){  *//System.out.println("=============");* }  }  }).start();  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  *flag* = **false**;  } } |

**可见性：**

　　可见性是一种复杂的属性，因为可见性中的错误总是会违背我们的直觉。通常，我们无法确保执行读操作的线程能适时地看到其他线程写入的值，有时甚至是根本不可能的事情。为了确保多个线程之间对内存写入操作的可见性，必须使用同步机制。

**可见性，是指线程之间的可见性，一个线程修改的状态对另一个线程是可见的。**也就是一个线程修改的结果。另一个线程马上就能看到。比如：用volatile修饰的变量，就会具有可见性。volatile修饰的变量不允许线程内部缓存和重排序，即直接修改内存。所以对其他线程是可见的。但是这里需要注意一个问题，volatile只能让被他修饰内容具有可见性，但不能保证它具有原子性。比如 volatile int a = 0；之后有一个操作 a++；这个变量a具有可见性，但是a++ 依然是一个非原子操作，也就是这个操作同样存在线程安全问题。

在 Java 中 volatile、synchronized 和 final 实现可见性。

**原子性：**

**原子是世界上的最小单位，具有不可分割性。**比如 a=0；（a非long和double类型） 这个操作是不可分割的，那么我们说这个操作是原子操作。再比如：a++； 这个操作实际是a = a + 1；是可分割的，所以不是一个原子操作。非原子操作都会存在线程安全问题，需我们使用同步技术（sychronized）来让它变成一个原子操作。一个操作是原子操作，那么我们称它具有原子性。Java的concurrent包下提供了一些原子类，我们可通过阅读API来了解这些原子类的用法。比如：AtomicInteger、AtomicLong、AtomicReference等。

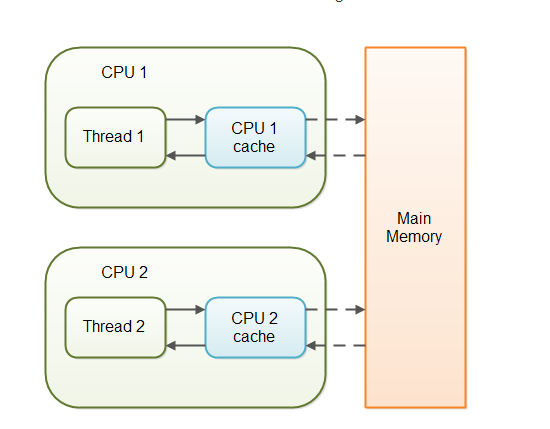
在 Java 中 synchronized 和在 lock、unlock 中操作保证原子性。

**有序性：**

Java 语言提供了 volatile 和 synchronized 两个关键字来保证线程之间操作的有序性，volatile 是因为其本身包含“禁止指令重排序”的语义，synchronized 是由“一个变量在同一个时刻只允许一条线程对其进行 lock 操作”这条规则获得的，此规则决定了持有同一个对象锁的两个同步块只能串行执行。

* volatile原理

　Java语言提供了一种稍弱的同步机制，即volatile变量，用来确保将变量的更新操作通知到其他线程。当把变量声明为volatile类型后，编译器与运行时都会注意到这个变量是共享的，因此不会将该变量上的操作与其他内存操作一起重排序。volatile变量不会被缓存在寄存器或者对其他处理器不可见的地方，因此在读取volatile类型的变量时总会返回最新写入的值。在访问volatile变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此volatile变量是一种比sychronized关键字更轻量级的同步机制。



　　当对非 volatile 变量进行读写的时候，每个线程先从内存拷贝变量到CPU缓存中。如果计算机有多个CPU，每个线程可能在不同的CPU上被处理，这意味着每个线程可以拷贝到不同的 CPU cache 中。而声明变量是 volatile 的，JVM 保证了每次读变量都从内存中读，跳过CPU cache 这一步。而写的内容在写入CPU cache的同时也同步到主存中。

**当一个变量定义为 volatile 之后，将具备两种特性：**

　　1.保证此变量对所有的线程的可见性，这里的“可见性”，如本文开头所述，当一个线程修改了这个变量的值，volatile 保证了新值能立即同步到主内存，以及每次使用前立即从主内存刷新。但普通变量做不到这点，普通变量的值在线程间传递均需要通过主内存来完成。

　　2.禁止指令重排序优化。volatile修饰的变量，赋值后多执行了一个“load addl $0x0, (%esp)”操作，该操作相当于一个**内存屏障**（指令重排序时不能把后面的指令重排序到内存屏障之前的位置），只有一个CPU访问内存时不需要内存屏障；（指令重排序：CPU会允许将多条不相关指令不按程序规定的顺序分开发送给各相应电路单元处理）。

**volatile 性能：**

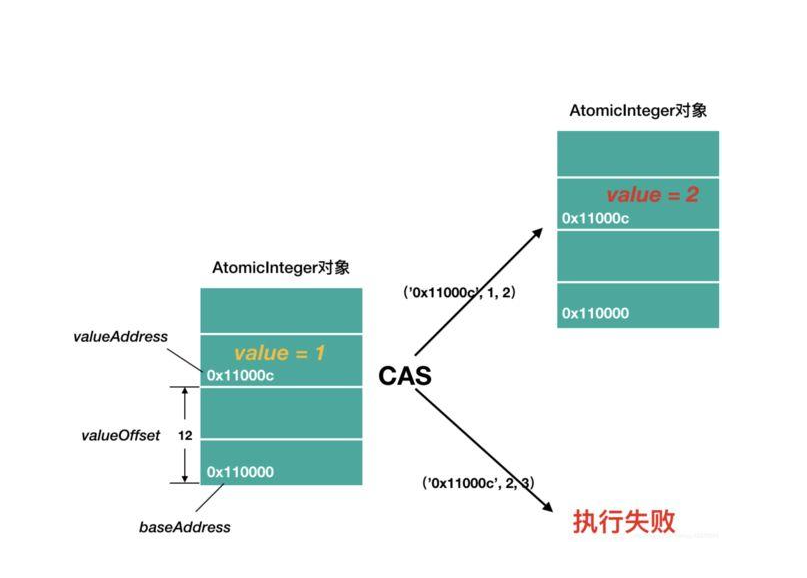
volatile 的读性能消耗与普通变量几乎相同，但是写操作稍慢，因为它需要在本地代码中插入许多内存屏障指令来保证处理器不发生乱序执行。

### 2.3 CAS和ABA问题

线程安全的三要素是原子性、可见性和有序性。synchronized可以保证这三点，可以作为线程安全解决方案。而volatile只能保证可见性和有序性，只要再提供一种保证多线程情况下原子性的技术，就能够实现线程安全。CAS就是这样一种保证原子性的技术。volatile+CAS相结合就可以作为一种线程安全方案。其实Lock底层就是采用volatile+CAS相结合的方案。

**CAS，Compare And Swap/Set，**比较并交换，比较并修改。它的作用是，对指定内存地址的数据，校验它的值是否为期望值，如果是，就修改为新值，返回值表示是否修改成功。CAS采用的直接操作系统底层的技术（通过native方法，调用C/C++开发的方法完成），与普通代码级别的比较交换相比，其特殊之处在于他的操作是原子性的，不会被其他指令所妨碍。

Java提供了一个非公开的类，**sun.misc.UnSafe，来专门做操作底层的操作**，它提供的方法都是native本地方法，它封装了一系列的原子化操作。



**关于CAS操作有个经典的ABA问题**：线程1使用CAS操作变量X，打算把值由A修改外B。需要首先获取其初始值为A，修改为B之前先先判断此时其值是否还是A（可能在此期间其他线程已经修改了），如果不是A说明被修改过了，要重新执行下一个CAS操作。如果是A，就说明中间没有被修改过，可以修改为B了。但是问题就在于完全有可能中间有一个线程B通过CAS操作将A修改为B，然后线程B或另外一个线程C将内容由B修改回A，此时的A已经其实不是线程A读取的那个A了。

如果ABA问题需要解决的话（也可能无所谓），可以通过时间戳的方式来解决。同时设计一个属性，记录每次修改的时间、或者记录每次修改的版本（版本递增），获取的时候同时获取两个属性的值，比较的时候也同时比较两个属性的值，就可以解决这个问题了。

#### 【示例2】volatile无法保证原子性

|  |
| --- |
| **public class** TestCAS {  **volatile static int** *n* = 0;  **public static void** main(String[] args) {  **for** (**int** i=0;i<10;i++){  **new** Thread(**new** Runnable() {  @Override  **public void** run() {  **for** (**int** j = 0; j <10000 ; j++) {  *n*++;  }  }  }).start();  }  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(*n*);  } } |

10个线程，每个线程执行n++一万次，最终结果应该是10万。但是运行的结果却相差很多，这就说明了volatile无法保证原子性。其中的n++一条语句变成机器指令后其实是多条语句，并没有进行线程同步。除了使用synchronized进行同步外，还可以使用volatile+CAS来实现。

AtomicInteger类，原子性Integer类，底层就使用了volatile+CAS来实现，保证了自增操作的原子性。我们直接使用AtomicInteger类来替代n++即可。

#### 【示例3】CAS保证原子性的效果展示

|  |
| --- |
| **public class** TestCAS2 {  **static** AtomicInteger *atomicInteger* = **new** AtomicInteger(0);  **public static void** main(String[] args) {  **for** (**int** i=0;i<10;i++){  **new** Thread(**new** Runnable() {  @Override  **public void** run() {  **for** (**int** j = 0; j <10000 ; j++) {  *atomicInteger*.incrementAndGet();  }  }  }).start();  }  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(*atomicInteger*);  } } |

## Locks

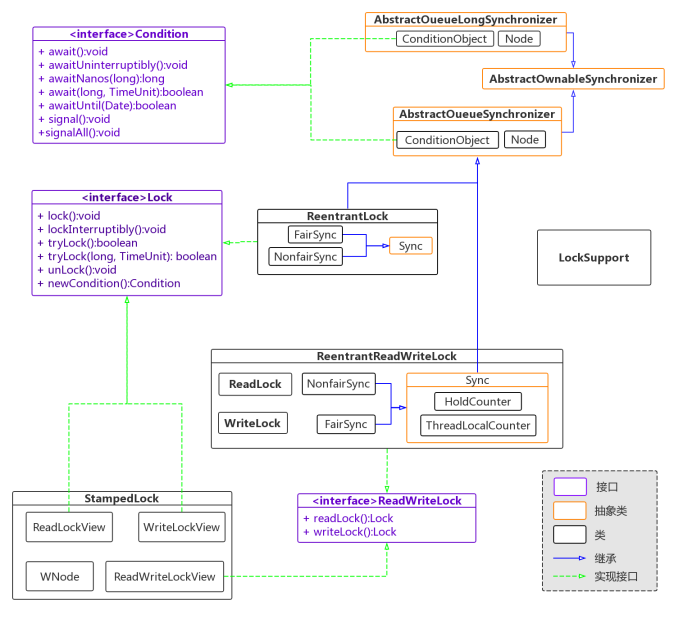
### 3.1概念简介

Locks是JDK提供的锁机制，相比synchronized关键字来进行同步锁，功能更加强大，它为锁提供了一个框架，该框架允许更灵活地使用锁包含的实现类有：

1）ReentrantLock   它是独占锁，是指只能被独自占领，即同一个时间点只能被一个线程锁获取到的锁。

2）ReentrantReadWriteLock   它包括子类ReadLock和WriteLock。ReadLock是共享锁，而WriteLock是独占锁。

3）LockSupport 它具备阻塞线程和解除阻塞线程的功能，并且不会引发死锁



### 3.2锁的分类

* 公平锁/非公平锁
* 可重入锁
* 独享锁/共享锁
* 互斥锁/读写锁
* 乐观锁/悲观锁
* 分段锁
* 偏向锁/轻量级锁/重量级锁
* 自旋锁

#### **公平锁/非公平锁**

公平锁是指多个线程按照申请锁的顺序来获取锁。  
非公平锁是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。

#### 可重入锁

可重入锁又名递归锁，是指在同一个线程在外层方法获取锁的时候，在进入内层方法会自动获取锁。说的有点抽象，下面会有一个代码的示例。  
对于Java ReentrantLock而言, 他的名字就可以看出是一个可重入锁，其名字是Re entrant Lock重新进入锁。  
对于Synchronized而言,也是一个可重入锁。可重入锁的一个好处是可一定程度避免死锁。

|  |
| --- |
| **synchronized void** setA() **throws** Exception{  Thread.*sleep*(1000);  setB(); } **synchronized void** setB() **throws** Exception{  Thread.*sleep*(1000); } |

上面的代码就是一个可重入锁的一个特点，如果不是可重入锁的话，setB可能不会被当前线程执行，可能造成死锁。

**独享锁/共享锁**

独享锁是指该锁一次只能被一个线程所持有。  
共享锁是指该锁可被多个线程所持有。

对于Java ReentrantLock而言，其是独享锁。但是对于Lock的另一个实现类ReadWriteLock，其读锁是共享锁，其写锁是独享锁。  
读锁的共享锁可保证并发读是非常高效的，读写，写读 ，写写的过程是互斥的。  
独享锁与共享锁也是通过AQS来实现的，通过实现不同的方法，来实现独享或者共享。  
对于Synchronized而言，当然是独享锁。

#### 互斥锁/读写锁

上面讲的独享锁/共享锁就是一种广义的说法，互斥锁/读写锁就是具体的实现。  
互斥锁在Java中的具体实现就是ReentrantLock  
读写锁在Java中的具体实现就是ReadWriteLock

#### 乐观锁/悲观锁

乐观锁与悲观锁不是指具体的什么类型的锁，而是指看待并发同步的角度。  
悲观锁认为对于同一个数据的并发操作，一定是会发生修改的，哪怕没有修改，也会认为修改。因此对于同一个数据的并发操作，悲观锁采取加锁的形式。悲观的认为，不加锁的并发操作一定会出问题。  
乐观锁则认为对于同一个数据的并发操作，是不会发生修改的。在更新数据的时候，会采用尝试更新，不断重新的方式更新数据。乐观的认为，不加锁的并发操作是没有事情的。

从上面的描述我们可以看出，悲观锁适合写操作非常多的场景，乐观锁适合读操作非常多的场景，不加锁会带来大量的性能提升。  
悲观锁在Java中的使用，就是利用各种锁。  
乐观锁在Java中的使用，是无锁编程，常常采用的是CAS算法，典型的例子就是原子类，通过CAS自旋实现原子操作的更新。

#### 分段锁

分段锁其实是一种锁的设计，并不是具体的一种锁，对于ConcurrentHashMap而言，其并发的实现就是通过分段锁的形式来实现高效的并发操作。  
我们以ConcurrentHashMap来说一下分段锁的含义以及设计思想，ConcurrentHashMap中的分段锁称为Segment，它即类似于HashMap（JDK7与JDK8中HashMap的实现）的结构，即内部拥有一个Entry数组，数组中的每个元素又是一个链表；同时又是一个ReentrantLock（Segment继承了ReentrantLock)。  
当需要put元素的时候，并不是对整个hashmap进行加锁，而是先通过hashcode来知道他要放在那一个分段中，然后对这个分段进行加锁，所以当多线程put的时候，只要不是放在一个分段中，就实现了真正的并行的插入。  
但是，在统计size的时候，可就是获取hashmap全局信息的时候，就需要获取所有的分段锁才能统计。  
分段锁的设计目的是细化锁的粒度，当操作不需要更新整个数组的时候，就仅仅针对数组中的一项进行加锁操作。

#### 偏向锁/轻量级锁/重量级锁

在Java（1.6+）中锁的状态一共有四种，级别由低到高分别是：无锁、偏向锁、轻量级锁、重量级锁，这几个状态会随着竞争情况逐渐升级，其中锁可以升级，但是不能降级。Java中加锁的最简单方式就是加synchronized关键字，那么为什么锁会有这么多状态的锁呢？

在Java早期，synchronized叫做重量级锁，加锁过程需要操作系统在内核态访问核心资源，因此操作系统会在用户态与内核态之间切换，效率很低下。于是JDK1.6之后，JVM为了提高锁的获取与释放效率，对synchronized进行了优化，引入了偏向锁和轻量级锁，根据线程竞争情况对锁进行升级，在线程竞争不激烈的情况避免使用重量级锁。

**无锁**：对象头中有31bit的空间来存储对象的hashcode，4bit用于存放对象分代年龄，1bit来表示是否是偏向锁，2bit存放锁标志位，偏向锁位与锁标志位合起来“001”就代表无锁。无锁就是没有对任何资源进行锁定，所有线程都能访问并修改资源。

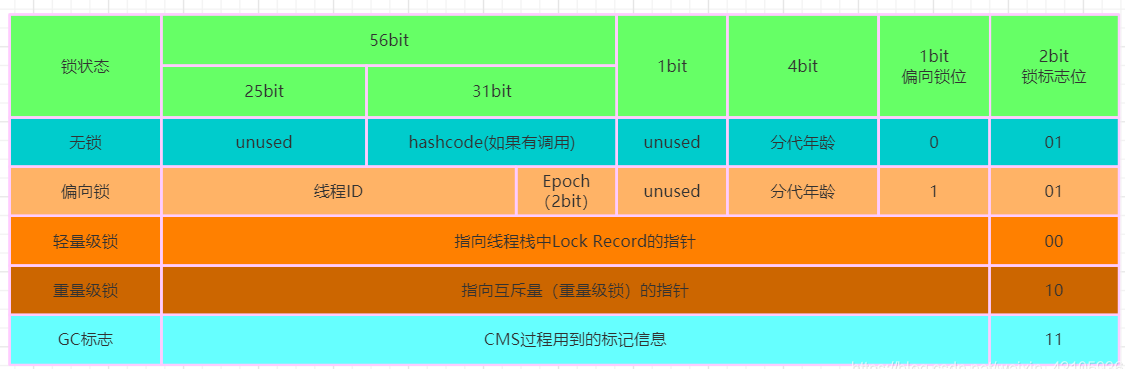
**偏向锁**：对象头中记录了获得偏向锁的线程ID，偏向锁与锁标志位合起来“101”就代表偏向锁。有研究发现，在大多数情况下，锁很少被多个线程同时竞争，而且总是由同一个线程多次获得，因此只需要将获得锁的线程ID写入到锁对象Mark Word中，相当于告诉其他线程，这块资源已经被我占了。当线程访问资源结束后，不会主动释放偏向锁，当线程再次需要访问资源时，JVM就会通过Mark Word中记录的线程ID判断是否是当前线程，如果是，则继续访问资源。所以，在没有其他线程参与竞争时，锁就一直偏向被当前线程持有，当前线程就可以一直占用资源或者执行代码。

**自旋锁（轻量级锁）**：一旦有另外一个线程参与锁竞争，偏向锁就会升级为自旋锁，此时撤销偏向锁，锁标志位变为“00”。竞争的两个线程都在各自的线程栈帧中生成一个Lock Record空间，用于存储锁对象目前Mark Word的拷贝，用CAS操作将Mark Word设置为指向自己这个线程的LR（Lock Record）指针，设置成功者获得锁，其他参与竞争的线程如果未获取到锁，则会一直处于自旋等待的状态，直到竞争到锁。

**重量级锁**：长时间的自旋操作是很消耗CPU资源的，为了避免这种盲目的消耗，JVM会在有线程超过10次自旋，或者自旋次数超过CPU核数的一半（JDK1.6以后加入了自适应自旋-Adaptive Self Spinning，由JVM自己控制自旋次数）时，会升级到重量级锁。重量级锁底层是依赖操作系统的mutex互斥锁，也就是有操作系统来负责线程间的调度。重量级锁减少了自旋锁带来的CPU消耗，但是由于操作系统调度线程带来的线程阻塞会使程序响应速度变慢。

在markword中标记锁的类型

普通对象在内存中的结构分为多部分，第一部分称为markword，共64位。在对应锁对象的markword字段的低位字段标记锁的类型。



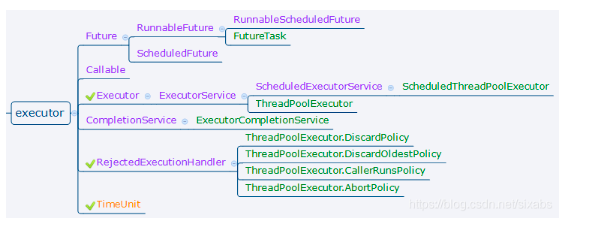
## Executor

### 4.1概念简介

executor(执行者)：是Java里面线程池的顶级接口，但它只是一个执行线程的工具，真正的线程池接口是ExecutorService，里面包含的类有：

1）ScheduledExecutorService  解决那些需要任务重复执行的问题

2）ScheduledThreadPoolExecutor   周期性任务调度的类实现



## 五、Tools工具类

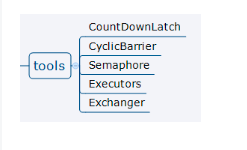
### 5.1概念简介

1，tools（工具类）：又叫信号量三组工具类，包含有

1）CountDownLatch（闭锁） 是一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待

2）CyclicBarrier（栅栏） 之所以叫barrier，是因为是一个同步辅助类，允许一组线程互相等待，直到到达某个公共屏障点 ，并且在释放等待线程后可以重用。

3）Semaphore（信号量） 是一个计数信号量，它的本质是一个“共享锁“。信号量维护了一个信号量许可集。线程可以通过调用 acquire()来获取信号量的许可；当信号量中有可用的许可时，线程能获取该许可；否则线程必须等待，直到有可用的许可为止。 线程可以通过release()来释放它所持有的信号量许可。



### 5.2 CountDownLatch门闩类

在开发中经常遇到在主线程中开启多个线程去并行执行任务，并且主线程需要等待所有子线程执行完毕后再进行汇总的场景。之前是使用join()来实现的，但是不够灵活，某些场合和还无法实现，所以开发了CountDownLatch这个类。底层基于AQS。

CountDown是计数递减的意思，Latch是门闩的意思。内部维持一个递减的计数器。可以理解为初始有n个Latch，等Latch数量递减到0的时候，就结束阻塞执行后续操作。

* countDown( )：减少Latch的计数，如果计数达到零，释放所有等待的线程。
* await()：导致当前线程等待，直到到Latch计数到零，或者被interrupt。

#### 【示例4】CountDownLatch使用示例

|  |
| --- |
| **public class** TestCountDownLatch {   **private static volatile** CountDownLatch *countDownLatch* = **new** CountDownLatch(2);  **public static void** main(String[] args) {  Thread thread1 = **new** Thread(()->{  **try** { Thread.*sleep*(10);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(**"Thread01执行完毕"**);  *countDownLatch*.countDown();//去掉一个Latch  });  Thread thread2 = **new** Thread(()->{  **try** { Thread.*sleep*(10);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(**"Thread02执行完毕"**);  *countDownLatch*.countDown();//去掉一个Latch  });  thread1.start(); thread2.start();  **try** {  *countDownLatch*.await();//等待Latch为0  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(**"两个线程执行完后，这里才会执行"**);  } } |

### 5.3 CyclicBarrier回环屏障

CountDownLatch优化了join()在解决多个线程同步时的能力，但CountDownLatch的计数器是一次性的。计数递减为0之后，再调用countDown()、await()将不起作用。为了满足计数器可以重置的目的，JDK推出了CyclicBarrier类。

Barrier：屏障，会等待线程数目满足指定数量后，冲破屏障，同时执行，Cyclic：回环，冲破屏障后数量重置，开始下一轮线程的等待和冲破屏障。底层基于AQS。

#### 【示例4】CyclicBarrier使用示例

|  |
| --- |
| **public class** TestCyclicBarrier {  **new** Runnable() {  @Override  **public void** run() {  System.***out***.println(Thread.*currentThread*().getName()+**":2个线程执行完后整合结果"**);  }  });  **public static void** main(String[] args){  ExecutorService pool = Executors.*newFixedThreadPool*(2);  pool.submit(**new** Runnable() {  @Override  **public void** run() {  System.***out***.println(Thread.*currentThread*().getName()+**"任务分解1"**);  **try** {  *cyclicBarrier*.await();  } **catch** (Exception e) {  e.printStackTrace();  }  }  });  pool.submit(**new** Runnable() {  @Override  **public void** run() {  System.***out***.println(Thread.*currentThread*().getName()+**"任务分解2"**);  **try** {  *cyclicBarrier*.await();  } **catch** (Exception e) {  e.printStackTrace();  }  }  });  pool.shutdown();  } } |

假设多个任务都有三个阶段组成，多个线程分别指向一个任务，必须保证每个任务的一个阶段结束后，才进入下一个阶段。此时使用CyclicBarrier正合适。

#### 【示例5】CyclicBarrier使用示例2

|  |
| --- |
| **public class** TestCyclicBarrier2 {  **private static** CyclicBarrier *cyclicBarrier* = **new** CyclicBarrier(2);  **public static void** main(String[] args) {  ExecutorService pool = Executors.*newFixedThreadPool*(2);  pool.submit(**new** Runnable() {  @Override  **public void** run() {  **try** {  System.***out***.println(**"task1 step1"**);  *cyclicBarrier*.await();  System.***out***.println(**"task1 step2"**);  *cyclicBarrier*.await();  System.***out***.println(**"task1 step3"**);  } **catch** (InterruptedException e) {  e.printStackTrace();  } **catch** (BrokenBarrierException e) {  e.printStackTrace();  }  }  });  pool.submit(**new** Runnable() {  @Override  **public void** run() {  **try** {  System.***out***.println(**"task2 step1"**);  *cyclicBarrier*.await();  System.***out***.println(**"task2 step2"**);  *cyclicBarrier*.await();  System.***out***.println(**"task2 step3"**);  } **catch** (InterruptedException e) {  e.printStackTrace();  } **catch** (BrokenBarrierException e) {  e.printStackTrace();  }  }  });  pool.shutdown();  } } |

### 5.4 Semaphore 计数信号量

CountDownLatch和CyclicBarrier的计数器递减的，而Semaphore的计数器是递增的，并可指定计数器的初始值，并且不需要事先确定同步线程的个数，等到需要同步的地方指定个数即可。且Semaphore也具有回环重置的功能，这一点和CyclicBarrier很像。底层也是基于AQS。

Semaphore：信号量的含义。常用方法如下：

* release()：释放许可证，将其返回到信号量，可用许可证的数量增加一个
* acquire(int n)：从该信号量获取给定数量的许可证，数量不足就阻塞等待

#### 【示例6】Semaphore 使用示例

|  |
| --- |
| **public class** TestSemaphore {  **private static** Semaphore *semaphore* = **new** Semaphore(0);  **public static void** main(String[] args) {  ExecutorService pool = Executors.*newFixedThreadPool*(2);  pool.submit(**new** Runnable() {  **public void** run() {  System.***out***.println(Thread.*currentThread*().getName()+**":task1 over"**);  *semaphore*.release();  }  });  pool.submit(**new** Runnable() {  **public void** run() {  System.***out***.println(Thread.*currentThread*().getName()+**":task2 over"**);  *semaphore*.release();  }  });  **try** {  *semaphore*.acquire(2);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName()+**":all children thread over"**);  pool.shutdown();  } } |

AQS底层原理

概念

AQS是一个同步队列，全名为AbstractQueuedSynchronizer，它是一个同步工具，是Lock用来实现线程同步的核心组件。

AQS的功能分为两种：独占和共享

独占锁：每次只能有一个线程获取锁，比如ReentrantLock是以独占的方式实现互斥

共享锁：允许多个线程同时获取锁，并发的访问共享资源，比如ReentrantReadWriteLock

内部实现

AQS队列内部维护的是一个FIFO的双向链表。每个Node由线程封装，当线程争抢锁失败后会封装成Node加入

AQS队列中去。当获取锁的线程释放锁以后，会从队列中唤醒一个阻塞的节点

## 六、Collections并发集合类

### 6.1概念介绍

collections(集合类)：主要是提供线程安全的集合， 比如：

1）ArrayList对应的高并发类是CopyOnWriteArrayList，

2）HashSet对应的高并发类是 CopyOnWriteArraySet，

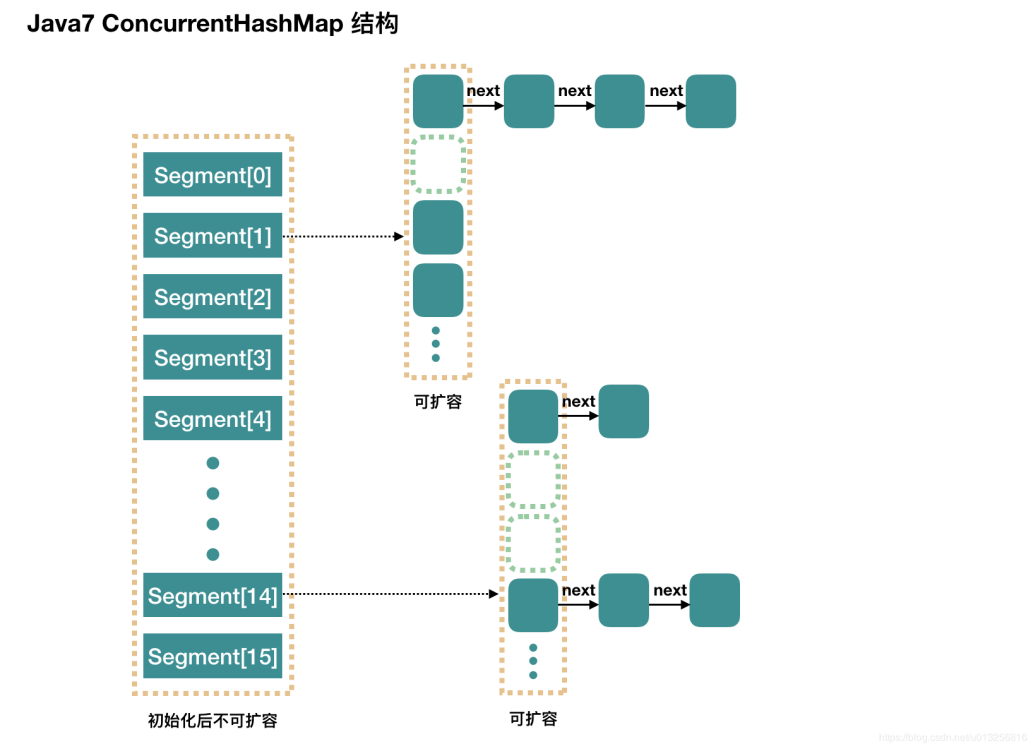
 3）HashMap对应的高并发类是ConcurrentHashMap等等

### 6.2 新一代并发集合类

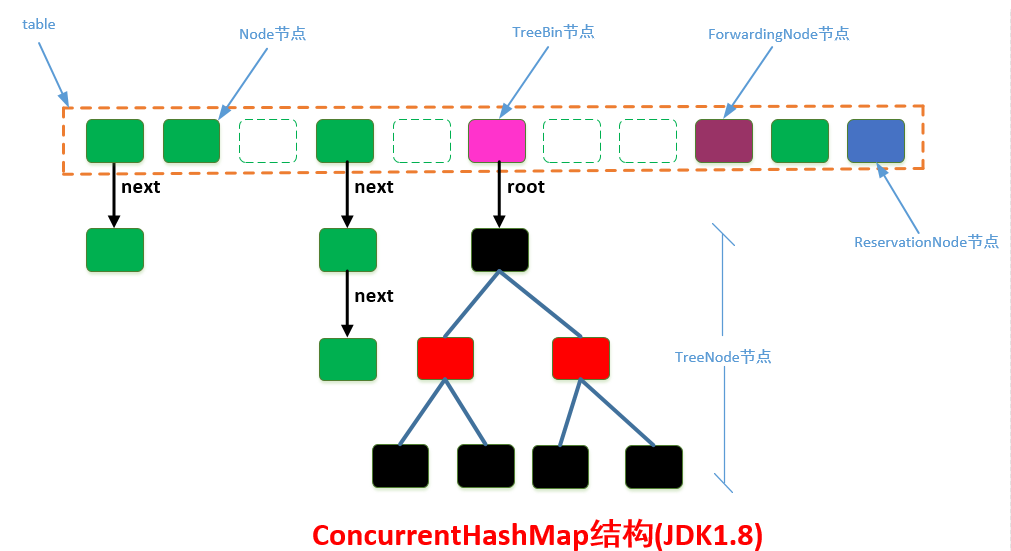
#### 6.2.1ConcurrentHashMap

* **ConcurrentHashMap: 分段（segment）锁定+Lock锁**

HashMap的线程安全班，性能比Hashtable、Collections.*synchronizedMap*(m);都有提高。使用的不是synchronized代码块锁，也不是synchronzied方法锁。并且使用了锁分离技术，**使用多个锁来控制对hash表的不同部分（段segment）进行的修改**，采用ReentrantLock锁来实现。如果多个修改操作发生在不同的段上，他们就可以并发进行，从而提高了效率。JDK1.7和JDK1.8的关于ConcurrentHashMap的实现差异较大，以上理论属于JDK1.7。



Concurr**entHashMap在JDK8中进行了巨大改动。**它摒弃了Segment（锁段）的概念，而是启用了一种全新的方式实现,利用synchronized + CAS 实现无锁化操作。 它底层由"数组"+链表+红黑树的方式思想(JDK8中HashMap的实现)， 为了做到并发，又增加了很多辅助的类，例如TreeBin，Traverser等对象内部类。



JDK1.8的实现降低锁的粒度，JDK1.7版本锁的粒度是基于Segment的，包含多个HashEntry，而JDK1.8锁的粒度就是HashEntry（首节点）

JDK1.8版本的数据结构变得更加简单，使得操作也更加清晰流畅，因为已经使用synchronized来进行同步，所以不需要分段锁的概念，也就不需要Segment这种数据结构了，由于粒度的降低，实现的复杂度也增加了

JDK1.8使用红黑树来优化链表，基于长度很长的链表的遍历是一个很漫长的过程，而红黑树的遍历效率是很快的，代替一定阈值的链表，这样形成一个最佳拍档

JDK1.8为什么使用内置锁synchronized来代替重入锁ReentrantLock，我觉得有以下几点

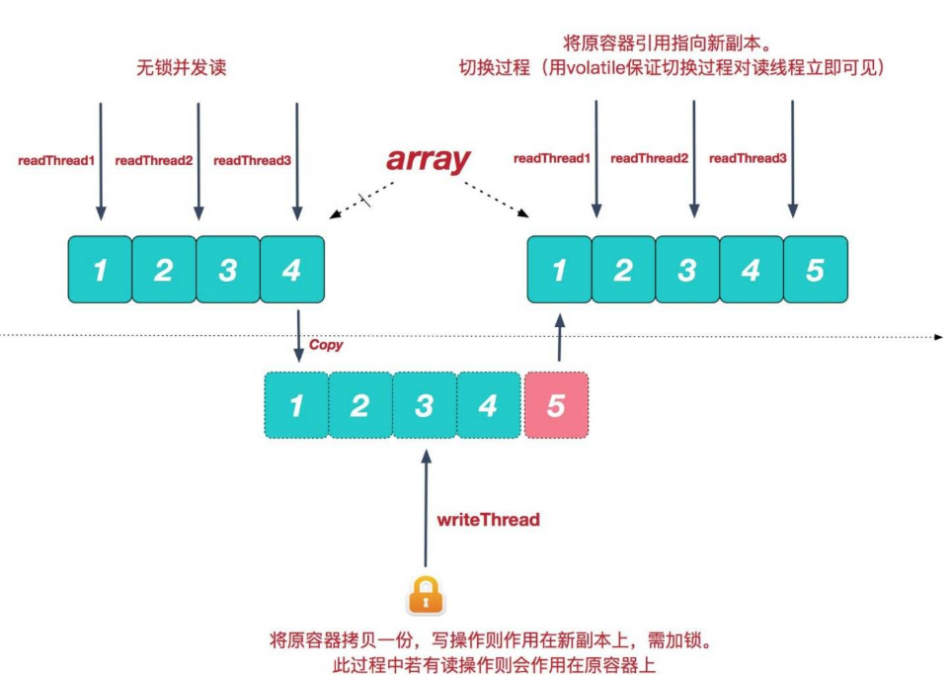
因为粒度降低了，在相对而言的低粒度加锁方式，synchronized并不比ReentrantLock差，在粗粒度加锁中ReentrantLock可能通过Condition来控制各个低粒度的边界，更加的灵活，而在低粒度中，Condition的优势就没有了

#### 6.2.2 CopyOnWriteArrayList

* **CopyOnWriteArrayList ：CopyOnWrite+Lock锁**

对于set()、add()、remove()等方法使用ReentrantLock的lock和unlock来加锁和解锁。读操作不需要加锁（之前集合安全类，即使读操作也要加锁，保证数据的实时一致）。

|  |
| --- |
| **CopyOnWrite原理：写时复制**。  通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，  添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器。  对于读操作远远多于写操作的应用非常适合，特别在并发情况下，可以提供高性能的并发读取。  CopyOnWrite容器只能保证数据的最终一致性，不能保证数据实时一致性。所以如果你希望写入的的数据，马上能读到，请不要使用CopyOnWrite容器。 |



|  |
| --- |
| **public boolean** add(E e) {  **final** ReentrantLock lock = **this**.**lock**;  lock.lock();  **try** {  Object[] elements = getArray();  **int** len = elements.**length**;  // 复制出新数组  Object[] newElements = Arrays.*copyOf*(elements, len + 1);  // 把新元素添加到新数组里  newElements[len] = e;  // 把原数组引用指向新数组  setArray(newElements);  **return true**;  } **finally** {  lock.unlock();  } }  **final void** setArray(Object[] a) {  **array** = a; }  **public** E get(**int** index) {  **return** get(getArray(), index); } |

注意：读的时候不需要加锁，如果读的时候有多个线程正在向ArrayList添加数据，读还是会读到旧的数据，因为写的时候不会锁住旧的ArrayList

6.2.3 CopyOnWriteArraySet

* **CopyOnWriteArraySet：CopyOnWrite+Lock锁**

它是线程安全的无序的集合，可以将它理解成线程安全的HashSet。 有意思的是，CopyOnWriteArraySet和HashSet虽然都继承于共同的父类AbstractSet； 但是，HashSet是通过"散列表(HashMap)"实现的，而CopyOnWriteArraySet则是通过"动态数组(CopyOnWriteArrayList)"实现的，并不是散列表。

CopyOnWriteArraySet在CopyOnWriteArrayList 的基础上使用了Java的装饰模式，所以底层是相同的。而CopyOnWriteArrayList本质是个动态数组队列，所以CopyOnWriteArraySet相当于通过通过动态数组实现的"集合"！

CopyOnWriteArrayList中允许有重复的元素；但CopyOnWriteArraySet是一个集合，所以它不能有重复集合。因此，CopyOnWriteArrayList额外提供了addIfAbsent()和addAllAbsent()这两个添加元素的API，通过这些API来添加元素时，只有当元素不存在时才执行添加操作！